Tiling of Iteration Spaces for Multicomputers

J. Ramanujam and **P. Sadayappan**
Department of Computer and Information Science
The Ohio State University, Columbus Ohio 43210

Abstract

We deal with compiler support for parallelizing perfectly nested loops for coarse-grain distributed memory machines. The relatively high communication start-up costs in these machines renders frequent communication very expensive. We study the effect of clustering communication and the ensuing loss of parallelism on performance and propose a method for aggregating a number of loop iterations into "tiles" where the tiles execute atomically where there are no synchronizations to be performed during the execution of a tile. As a result, it is important that dividing the loops into tiles does not lead to deadlock. Based on conditions for deadlock-free tiles, we present a method for deriving legal tiles for nested loops. We then develop an approach to optimize the shape and size of tiles along with the assignment of tiles to processors for load-balanced execution with reduced communication costs on distributed memory machines given communication setup and transfer rates and instruction execution rates.

1 Introduction

Distributed memory machines (or multicomputers) offer very high levels of flexibility, scalability and performance but leave the programmer or the algorithm designer with the difficult task of detailed planning of the computation, i.e., the **orchestration** of the entire parallel execution. For machines such as hypercubes, the programmer is forced to manually distribute code and data in addition to managing communication among tasks explicitly. This task, in addition to being error-prone and time-consuming, generally leads to non-portable code. Hence parallelizing compilers for multicomputers have received a great deal of attention recently [3, 6, 14, 15, 16, 17, 25, 26, 34].

In this paper, we deal with compiler support for parallelizing programs for coarse-grain distributed memory machines – we consider the class of programs expressible as perfectly nested loops with a regular dependence structure. The relatively high communication start-up costs in these machines renders frequent communication very expensive. We study the effect of clustering communication and the ensuing loss of parallelism on performance and propose a method for aggregating a number of loop iterations into "tiles", where the tiles execute atomically – a processor executing the iteration 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. As there is no communication during the execution of a tile, it is important that dividing the loops into tiles does not lead to deadlock. Given a computation expressed by nested loops to be executed on a multicomputer with given execution and communication costs, the tile shape and size have to be chosen to optimize the performance. In addition, the assignment of tiles to processors should be such that inter-processor communication is minimized and the computation is load-balanced through time. We develop an approach to determine the shape and size of tiles for execution of nested loops on distributed memory machines.

In section 2, we discuss data dependence analysis and iteration space graphs corresponding to nested loop programs and the hyperplane technique for identifying parallel threads of computation. Section 3 develops the approach to iteration space clustering (into tiles) and discusses conditions for the validity of tiles. Section 4 discusses the choice of the tile shape. Section 5 discusses the choice of tile size and the allocation of tiles to the processors in a distributed memory machine; we also discuss the effect of tile size and shape on performance and tiling for nested loop execution on shared memory machines, when the loop execution involves synchronization among different iterations – the tile size can be chosen to reduce the synchronization and task (a collection of iterations) initiation overhead. Section 6 compares this with work on loop blocking (tiling) for hierarchical memory multi-vector processors [12, 30, 31, 32] and the work on compiling programs for multicomputers. Section 7 summarizes the work presented here and concludes with a discussion of work in progress.

2 Data Dependence and Parallelism

The automatic detection of parallelism involves finding sets of computations that can be performed simultaneously. The potential parallelism in a computation is determined by data dependences [1, 18, 23, 24, 33]. The presence of a dependence between two computations (statements) implies that they cannot be executed in parallel. 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 Dependences

Consider a loop structure of the form:

for $I_1 = l_1$ to u_1 do
 ...
  for $I_n = l_n$ to u_n do
   $S_1(\vec{I})$
   ...
   $S_N(\vec{I})$

where l_j and u_j are integer-valued linear expressions involving $I_1, \ldots, I_{j-1}$ and $\vec{I} = (I_1, \ldots, I_n)$. $S_1, \ldots, S_N$ 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.

Nested loops define an *iteration space* that comprises a finite discrete Cartesian space whose dimensionality is the loop nesting level. To each iteration, we associate an integer vector index $\vec{L} = (I_1, \ldots, I_n)^T$. Thus, we have an index set (iteration space):

$$L^n(\vec{I}) = \{(I_1, \ldots, I_n) : l_1 \leq I_1 \leq u_1, \cdots, l_n \leq I_n \leq u_n\}$$

When the loop is executed sequentially, the elements of L^n are in lexicographical ordering. This induced (sequential) ordering is non-essential and exposing and exploiting parallelism is about modifying this ordering (to reduce parallel execution time) while while preserving the semantics of the loop.

A *flow dependence* occurs when the value assigned to a variable or array element in the execution of one instance of a statement is read by the subsequent execution of an instance of the same or another statement. An *anti-dependence* occurs when the value read from a variable or array element in the execution of one instance of a statement is reassigned by the subsequent execution of an instance of some statement. An *output dependence* occurs when the value assigned to a variable or array element in the execution of one instance of a statement is reassigned by the subsequent execution of an instance of some statement.

For nested loops, an instance of a particular statement S is the index of the iteration associated with the statement and is denoted $S(I_n)$. Data dependence relations among statements in nested loops are augmented with the indices of the related iterations. Many dependences have a constant distance along each dimension of the iteration space. Suppose there is a dependence from $S(I_1)$ to $S(I_2)$ then the vector $\vec{d} = \vec{I_2} - \vec{I_1}$ is called the **iteration space distance vector** or simply **distance vector** [4].

Dependence relations are often represented in *Iteration Space Graphs* (ISGs). 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 $\vec{I_2}$.

An algorithm may have a number of such dependence vectors. In general, the dependence vectors are functions of the index point $\vec{I}$. The dependence vectors of an algorithm can be written collectively as a **dependence matrix** $D = [\vec{d_1}, \vec{d_2}, \ldots, \vec{d_m}]$. In addition to the three types of dependence 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 whereas the other two forms of dependence arise due to the storage-based imperative programming model. In our analysis, we consider distributed memory machines where considering flow dependences alone is sufficient. These data dependences constrain the order of execution of the statements in a program. Methods to calculate data dependence vectors can be found in [1, 4, 31, 33].

In this paper, we consider program loops which can exhibit uniform or regular dependence structure – every index point has the same dependences – and to specify the dependences exhibited by the algorithm, one only needs to specify the dependence of one index point. The dependence distances are constant (not functions of indices). For such computations, a variety of systolic array synthesis techniques exist, most of which analyze the dependence structure through linear transformations (affine if the distances are linear function of the indices) [9].

2.2 Hyperplane of Parallelism

In this section, we reiterate a fundamental result due to Karp et. al., [13] (dealing with parallel execution of computations expressed as uniform recurrence equations) and Lamport [19] (dealing with parallel execution of nested loops in FORTRAN) – this result has been used by Moldovan [9], Miranker [21], Wolfe [29] and others. Lamport considered partitioning of the indices $\vec{I} = (I_1, I_2, \ldots, I_n)$ such that they lie on a family of parallel hyperplanes such that all indices lying on one hyperplane can be executed simultaneously. Let $\vec{H}$ be a vector $(t_1, t_2, \ldots, t_n)$. Then,

$$t_1 x_1 + t_2 x_2 + \cdots + t_n x_n = k$$

for various values of k defines a family of hyperplanes. We use the terms hyperplane, family of hyperplanes and the vector defining them interchangeably where the context is clear. A case of Lamport's result is the following:

Theorem 1 *Let $\vec{H} = (t_1, t_2, \ldots, t_n)$ be a hyperplane such that $H\vec{d_i} > 0$ for each dependence vector $\vec{d_i} = (d_{i1}, \ldots, d_{in})^T$. If two indices $\vec{I_1}$ and $\vec{I_2}$ satisfy $HI_1 = HI_2 = c$, then $\vec{I_1}$ and $\vec{I_2}$ are independent of each other and hence can be executed simultaneously.*

If no such H exists, then the computation defined by the nested loop involves deadlocks. In the event that communication costs are negligible as in systolic arrays, an optimal hyperplane $H^* = (t_1^*, t_2^*, \ldots, t_n^*)$ minimizes the number of hyperplanes required to partition the indices and gives the optimal time for a parallel implementation of the algorithm on an MIMD machines. Thus the execution time of the parallel algorithm is given by:

$$\frac{\max \quad H(\vec{I_1} - \vec{I_2}) + 1}{\min \quad H\vec{d_i}}$$

for any $\vec{I_1}, \vec{I_2} \in L^n$ and $d_i \in D$. Thus, the normal to the hyperplane defines the direction of sequentiality. In the case of 2-dimensional index spaces, the hyperplane is also referred to as the **wavefront.**

Communication is extremely regular and fast in systolic arrays, leaving only the exploitation of parallelism to the designer. However, in the case of multicomputers, communication (data access) is an important issue – In the case of shared memory machines with fast local memory, access to data stored in local memory can be an order of magnitude faster than access to global memory. In the case of distributed memory message-passing machines such as Intel iPSC/2, message start-up times are much larger; the message start-up time for Intel iPSC/2 in 250 μs for a packet whereas the per-byte transmission cost is about 10 μs, and access to local memory takes negligible time. Thus, exploiting spatial locality of reference is important and the trade-off between degree of parallelism and communication (data access) cost decides what a "good" schedule is. Thus, techniques developed for systolic arrays are not directly applicable in the context of multiprocessors. In the next section, we outline the scheme for tiling of iteration spaces for nested loops.

3 Iteration Space Tiling

The basic idea behind tiling is to aggregate a number of iterations of a loop so as to reduce communication overheads while exploiting the available parallelism. Each tile defines an atomic unit of computation comprised of several iterations. All necessary data must be available before the execution of the tile and all data that are needed for the execution of other tiles is available when the execution of the tile is complete. Thus, no synchronization or communication is necessary during the execution of the tile. This imposes the constraint that the division of the iteration space into tiles does not result in deadlock. Intuitively this means that all dependence vectors cross a given tile boundary between two tiles in the same direction. 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. We discuss a relaxation of the uniform size constraint later in the paper, taking into account the parallelism profile of the execution of tiles.

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

3.1 Legal Tiles

Tiles in K-dimensional (K-D) spaces are defined by K families of parallel hyperplanes (or planes) each of which is a K-1 dimensional hyperplane. Tiles so defined are parallelopipeds (except for those near the boundary of the iteration space) and each tile is a K-D subset of the iteration space. Thus, the shape of the tiles is defined by the families of planes and the size of the tiles is defined by the distance of separation between adjacent pairs of parallel planes in each of the K families.

Consider the following example:

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

Figure 1a shows the iteration space graph defined by this loop. The distance vectors are $(1, -1)$, $(1, 0)$ and $(0, 1)$. Tiles in this ISG are defined by 2 families of lines. (Note: A K-1 dimensional plane when K=2 is a line). Figure 1b shows tiles in this ISG which are 3×3 collections of iterations.

In the case of arbitrary dependence graphs, clustering the tasks into atomic collections might lead to deadlock [27]. Sarkar [27] has formulated the condition for the absence of deadlock in this case as the *convexity* constraint. Irigoin and Triolet [12] present these conditions for iteration space graphs.

For tiles in K-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 must be in the same tile and their sink in the other. In the ensuing discussion, we assume that the dependence matrix for a K-nested loop has rank K and that the number of dependence vectors is m. A vector $H_i = (h_{i,1}, h_{i,2}, \ldots, h_{i,K})^T$ defines a family of hyperplanes (in a K-dimensional space)

$$h_{i,1}x_1 + h_{i,2}x_2 + \cdots + h_{i,K}x_K = c$$

for various values of c.

In K dimensions, any K vectors $H_1, \ldots, H_K$ define legal tiles if

$$H_i \cdot \vec{d_j} \geq \vec{0}, \qquad i = 1, \ldots, K \qquad (1)$$

for all d_j in the set of dependence vectors. One can also define an equivalent condition:

$$H_i \cdot \vec{d_j} \leq \vec{0}, \qquad i = 1, \ldots, K$$

where $\vec{0}$ is the zero vector in K-D space. Each of the K-1 dimensional planes is boundary whose normal is one of the H_i. Condition 1 is from [12].

The above result is only a sufficient condition. We give the following example, where the condition is violated, but tiles of a certain fixed size does not lead to deadlock. Consider the following example:

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

Figure 2 shows the ISG and the tile boundaries b_1 and b_2 of which b_1 is spaced 3 apart. Even though the dependence vectors $(0,1)$ and $(3,-1)$ cross the boundary b_2 in opposite directions, there is no deadlock since the head of the vector $(3,-1)$ is not in the same tile as the tail of the vector $(0,1)$. Thus, one can make the tiles small enough to avoid deadlocks. Henceforth, we assume that the tiles are larger than the magnitude of the dependence vectors which we refer to as free-size tiles (FSTs).

3.2 Extreme vectors

Based on some results in the theory of integer numbers and integer programming [2, 28], given a set of m distinct dependence vectors ($m \geq K$), one can find a set of K vectors say $\vec{e_1}, \ldots, \vec{e_K}$ such that any dependence vector $\vec{d_i}$, $i = 1, \ldots, m$ can be expressed as a non-negative linear combination of vectors $\vec{e_1}, \ldots, \vec{e_K}$, *i.e*

$$c_i \vec{d_i} = \sum_{j=1}^{K} a_{i,j} \vec{e_j}$$

where $i = 1, \ldots, m$ and $a_{i,j} \geq 0$ and $c_i > 0$. The set of vectors $e_1 \ldots e_K$ (referred to as **extreme vectors**). Let $(e_{j,1}, e_{j,2}, \ldots, e_{j,K})^T$ be the vector e_j. The vectors e_j can be found by solving the system of linear inequalities given by

$$H_i \cdot \vec{d_j} \geq 0 \quad i = 1, \ldots, K \quad \text{and} \quad j = 1, \ldots, m \qquad (2)$$

where each H_i is a K-1 dimensional plane containing K-1 of the extreme vectors. From among K extreme vectors, exactly K such planes can be generated. Thus we have system of $m \times K$ inequalities in K^2 variables.

In the case of distributed memory machines, the dependence vectors that cross a tile boundary imply potential inter-processor communication which can be reduced if the set of extreme vectors e_j is a subset of the dependence vectors $d_i, i = 1, \ldots m$. To check this, for each set of K linearly independent vectors from among the d_i the inequalities $H_i.d_j \geq 0$ must hold where each H_i is the normal to a K-1 dimensional plane containing K-1 vectors from the set. There are $\binom{n}{k}$ such sets for which this is to be checked.

4 Choice of Tile Shape

The shape of tiles is defined by the collection of extreme vectors e_j. Given the extreme vectors, $e_1, \ldots, e_K$,

$$\sum_{i=1}^{K} \sum_{j=1}^{m} H_i \cdot \vec{d_j}$$

is a measure of the communication incurred due to dependence vectors crossing tile boundaries. Thus from among the set of valid collections of extreme vectors, we choose the collection that minimizes the communication cost. Thus, we focus on selecting the shape of tiles motivated by the legality condition and communication reduction; where as Irigoin and Triolet [12] discuss partitioning using hyperplanes for legal tiles that facilitate vectorized execution of iterations within a single tile.

We say that K-dimensional tiles are induced by a collection of K linearly independent vectors. From the point of view of communication reduction, including as many dependence vectors as possible in the extreme vector set is good. We show the following result:

Theorem 2 *Tiles (K-dimensional subsets) induced by a collection of K vectors at least one which is a non-extreme dependence vector are* **illegal** *tiles.*

Proof: Given a set of m distinct dependence vectors, $\vec{d_1}, \vec{d_2}, \ldots, \vec{d_m}$, let the corresponding extreme vectors be $\vec{e_1}, \vec{e_2}, \ldots, \vec{e_K}$. Therefore,

$$c_i \vec{d_i} = \sum_{j=1}^{K} a_{i,j} \vec{e_j}$$

where $i = 1, \ldots, m$ and $a_{i,j} \geq 0$ and $c_i > 0$. We assume that the dependence matrix has rank K. Without loss of generality, we consider tiles induced by the dependence vector $\vec{d_n}$ (for some n) and the extreme vectors $\vec{e_2}, \ldots, \vec{e_K}$ i.e., $\vec{e_1}$ is omitted. (note: $\vec{d_n}$ and $\vec{e_1}$ are not collinear; if they were, $\vec{d_n}$ is one of the extreme vectors). We need to consider the two cases:

Case 1: The coefficient $a_{n,1} = 0$: With $a_{n,1} = 0$, we have

$$c_n \vec{d_n} = \sum_{j=1}^{K} a_{n,j} \vec{e_j} = \sum_{j=2}^{K} a_{n,j} \vec{e_j}$$

In this case, the rank of the matrix consisting of the vectors $\vec{e_2}, \ldots, \vec{e_K}$ and $\vec{d_n}$ is K-1. Therefore, one can not derive K-dimensional tiles.

Case 2: The coefficient $a_{n,1} \neq 0$:
Consider the tile boundary formed by the vectors $\vec{e_2}, \ldots, \vec{e_{l-1}}, \vec{e_{l+1}}, \ldots, \vec{e_K}$ and $\vec{d_n}$ where $2 \leq l \leq K$ and $a_{n,l} \neq 0$. There is at least one such vector $\vec{e_l}$. Let h be the normal to the tile boundary under consideration. Since $\vec{d_n}$ lies on the boundary, $h \cdot \vec{d_n} = 0$. Equivalently,

$$h \cdot \left(\sum_{j=1}^{K} a_{n,j} \vec{e_j} \right) = 0$$

Since $\vec{e_2}, \ldots, \vec{e_{l-1}}, \vec{e_{l+1}}, \ldots, \vec{e_K}$ lie on the plane,

$$h \cdot (a_{n,1} \vec{e_1} + a_{n,l} \vec{e_l}) = 0.$$

Since, both $a_{n,1}$ and $a_{n,l}$ are positive, $h \cdot \vec{e_1}$ and $h \cdot \vec{e_l}$ must have opposite signs which results in **illegal** tiles. Hence the result.

□

The above condition is a sufficient condition for the generation of free-size tiles (FSTs) i.e., tiles that are legal independent of their size.

Consider the loop in example 1 again. In figure 3, we show tiles induced by the vectors, $(0,1)$ and $(1,0)$. The tiles thus induced are illegal – in this case, $(0,1)$ is an extreme vector which $(1,0)$ is a non-extreme vector. An example of legal tiles for this loop is shown in Figure 1a.

5 Scheduling of Tiles

The tiles defined in section 3.1 are parallelopipeds that tessellate the K-dimensional iteration space. Given the dependence vectors in the ISG, we define the *Tile Space Dependence Graph (TSG)* which indicates the dependences (precedence constraints) among the tiles. 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 vector. This assumption leads to the following:

- The source and sink of any dependence vector that crosses a tile boundary are neighboring tiles
- Legal tiles satisfy the condition in equation 1.

We are interested in choosing the size of the tile to optimize the execution time; hence we are interested in free size tiles (FSTs). In this case, all dependence vectors in the Tile Space Dependence Graph (TSG) are K-tuples where each component is 0 or 1, if the lexicographic sequential order of execution of the loop is valid. Thus, there are at most $2^K - 1$ other tiles that a given tile can depend on. For scheduling of tiles, the scheduling hyperplane must satisfy the conditions of Theorem 1 (Lamport's result). The scheduling hyperplane given by $\vec{1}$, which is a K-dimensional vector with all components equal to 1, satisfies the said conditions.

We will discuss scheduling, allocation and selection of tile sizes in the context of 2-dimensional iteration spaces. The results generalize to higher dimensional iteration spaces. In 2-D space with a regular ISG, any free size tile (FST) can depend on at most three neighbors which satisfy exactly one of the following conditions:

1. The tile depends on one, two or three neighbors.
2. All tiles are independent

We will focus on condition 1. This implies that the only dependences possible in the tile space are $(1,0)^T$, $(0,1)^T$ and $(1,1)^T$. Since tiles are executed atomically, the $(1,1)^T$ is automatically covered by the other two. Therefore, the tiles in the Tile Space Dependence Graph (TSG) have uniform dependence vectors $(0,1)^T$ and $(1,0)^T$. Thus the tiles can be scheduled using the wavefront technique. Applying the technique, we get the result: $(1,1)^T$ defines a valid schedule in the tile space and if both tile space vectors are present, defines the optimal scheduling of tiles. The computations within a tile are executed in the order defined by the schedule in the ISG.

5.1 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 dependences (1,0) and (0,1) choosing allocations of tiles along either direction "internalizes" all communication in that direction. In addition, since (1,1) is a valid scheduling hyperplane for tiles, no more than one tile per row or per column can be executed simultaneously. Thus wrapped row or column distribution internalizes communication along one dimension and provides good balance. The interplay between load balance and communication depends on tile size and discussed in greater detail in the next section.

5.2 Choice of Tile Size

We discuss the choice of tile size in the context of 2-D iteration spaces. Let the tiles be defined by parallelograms whose sides are l_1 and l_2. For simplicity, assume $l = l_1 = l_2$. Let the computation be defined by an ISG of length n and width n. Let the number of processors be p. Let the iteration space be divided into k strips where $k = \frac{n}{p \times l}$. (a good choice of k improves load balance). We also assume that all the computations in a strip are completed before proceeding to the next strip. This facilitates dealing with ISGs whose degree of parallelism varies throughout the iteration space. Let the set-up cost for a packet be s; let the packet size be b and let the cost of data transmission be w. (all these costs are normalized with respect to the cost of executing a single iteration).

Let $l_1 = l_2 = l$ for simplicity. Let us assume that c is the number of data items that may be needed from one iteration point to another. The cost of sending cl values to another processor is

$$\left\lceil \frac{c \times l}{b} \right\rceil \times s + c \times l \times w$$

Given that l^2 nodes have to be executed in a tile, the cost of execution of the tile is:

$$l^2 + \left\lceil \frac{c \times l}{b} \right\rceil \times s + c \times l \times w$$

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 ignore the ceiling function and write the cost as $l^2 + clx$ where $x = w + \frac{s}{b}$. Therefore, the cost of execution of the computation with a $(1,1)^T$ wavefront schedule in the tiles space is:

$$\frac{n}{pl}\left[l^2 + c \times l \times x\right]\left[\frac{n}{l} + p - 1\right]$$

To get the optimal tile size, we take the derivative of the above cost with respect to l and find its zeros. In this case, we get the value of optimal l as:

$$\sqrt{\frac{n \times c \times x}{p-1}}$$

Thus, the tiling planes induce a Tile Space Dependence Graph for which we have identified that the hyperplane given by $\vec{1}$, which is a K-dimensional vector with all components equal to 1 is a valid scheduling hyperplane. In the case of dependence matrices (for K-nested loops) whose rank is K, this is also the optimal schedule when unlimited processors are available for loop execution.

6 Related Work

6.1 Related work in compiling for complex memory machines

Balasundaram and others [3] 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. Chen et al. [8] describes how Crystal, a functional language, addresses the issue of programmability and performance of parallel supercomputers. Gallivan et al. [10] 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. [11] discuss program transformations for effective complex-memory management for a CEDAR-like architecture with a three-level memory. Koelbel et al. [15, 16] 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 [6] 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 [25] present a method which, given a sequential program and its data partition, performs task partitions to enhance locality of references. Zima et al. [17, 34] 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. Mace [20] 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.

6.2 Related work in tiling

In the context of hierarchical shared memory systems, Irigoin and Triolet [12] present a method, which divides the iteration space into nodes, with the goals of vector computation within a node, data re-use within a partition and parallelism between partitions. King and Ni [14] 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. Wolfe [32] 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 [22] and Callahan *et al.* [5].

7 Summary and Work in progress

In this paper, we dealt with compiler support for parallelizing programs for coarse-grain distributed memory machines – we considered the class of programs expressible as tightly nested loops with a regular dependence structure. The relatively high communication start-up costs in these machines renders frequent communication very expensive. We studied the effect of clustering communication and the ensuing loss of parallelism on performance and propose a method for aggregating a number of loop iterations into "tiles" where the tiles execute atomically. Since execution of tiles is atomic, it is important that dividing the iteration spaces of loops into tiles does not lead to deadlock. We presented an approach to determining legal tiles, and choosing the shape and size of tiles for efficient execution of nested loops on multicomputers.

Work is in progress to characterize the set of all legal tiles. Given the parallelism profile in the execution of tiles, it is conceivable that increasing the tile size along each dimension uniformly and then decreasing the same might lead to more efficient execution. Work is in progress in estimating the effect of varying the tile size and processor allocation on the performance of parallel execution of tiles. In addition, we are investigating the impact of tiling for nested loop execution on shared memory machines to decrease the synchronization and task (a collection of iterations) initiation overhead.

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] A. Bachem, "The Theorem of Minkowski for Polyhedral Monoids and Aggregated Linear Diophantine Systems," *Optimization and Operations Research,*

Lecture Notes in Economics and Mathematical Systems, Vol. 157, pp. 1-14, Springer Verlag, 1977.

[3] V. Balasundaram, *personal communication.*

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

[5] 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.

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

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

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

[9] J. Fortes and D. Moldovan, "Parallelism Detection and Transformation Techniques Useful for VLSI Algorithms," *Journal of Parallel and Distributed Computing,* Vol. 2, 1985, pp. 277-301.

[10] 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.

[11] 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.

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

[13] R.M. Karp, R.E. Miller and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," *Journal of the ACM,* Vol. 14, Jul. 1967, pp. 563-590.

[14] 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.

[15] 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.

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

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

[18] 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.

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

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

[21] W. Miranker and A. Winkler, "Spacetime Representation of Computational Structures," *Computing,* 32, 1984, pp. 93-114.

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

[23] D. Padua, D. Kuck and D. Lawrie, "High-speed Multiprocessors and Compilation Techniques," *IEEE Trans. Computers,* Vol. C-29, No. 9, Sep. 1980, pp. 763-776.

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

[25] 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.

[26] 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.

[27] V. Sarkar, and J. Hennessey, "Partitioning Programs for Macro-Dataflow," in *Proc. 1986 ACM Conf. on Lisp and Functional Programming,* August 1986, pp. 202-211.

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

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

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

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

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

[33] 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.

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