An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication

Amy W. Lim, Gerald I. Cheong*, and Monica S. Lam

Computer Systems Laboratory
Stanford University
Stanford, CA 94305
{aimee, gcheong, lam}@cs.stanford.edu

Abstract

An affine partitioning framework unifies many useful program transforms such as unimodular transformations (interchange, reversal, skewing), loop fusion, fission, scaling, reindexing, and statement reordering. This paper presents an algorithm, based on this unified framework, that maximizes parallelism while minimizing communication in programs with arbitrary loop nestings and affine data accesses.

Our algorithm can find the optimal affine partition that maximizes the degree of parallelism with the minimum degree of synchronizations. In addition, it uses a greedy algorithm to minimize communication between loops heuristically by aligning the computation partitions for different loops, trading off excess degrees of parallelism, and choosing pipelined parallelism over doall parallelism if it can significantly reduce the communication. The algorithm is optimal in maximizing the degrees of parallelism that require (1) no communication, (2) near-neighbor communication and a constant number of synchronizations, and (3) near-neighbor communication and $O(n)$ synchronizations where n is the number of iterations in a loop.

1 Introduction

A large number of compiler algorithms have been proposed that use loop transformations to maximize loop level parallelism in a program and minimize communication between processors. Unimodular transformations (equivalent to arbitrary combinations of loop interchange, reversal and skewing) and blocking/tiling (equivalent to unroll-and-jam or stripmine-and-interchange) are effective for parallelizing perfectly nested loops[4, 5, 23, 24, 25]. These transforms are also useful for improving the data locality on uniprocessors. These transformations treat the loop body as an atomic unit, whereas the techniques of loop fusion (jamming) and loop fission (distribution) can change the composition of the loops[16, 19, 22, 25]. Attempts have also been made to optimize parallelism and communication across multiple loop nests. For example, Anderson and Lam's algorithm can vary the parallelization scheme to minimize communication, selecting among a choice of parallelizable loops to minimize communication across loops, aligning the parallel loops to favor near-neighbor communication, and using pipelined parallelism if it eliminates the need for re-distributing entire data structures[3]. Their approach of first finding parallelism in individual perfectly nested loops before optimizing across them may introduce synchronizations that are unnecessary and can be eliminated[20].

While these algorithms together address many of the important aspects of automatic parallelization, the approach of incorporating all these different algorithms into an optimizing compiler has its limitations. First, the individual algorithms have limited applicability; for example, unimodular transforms and blocking can only be applied to perfectly nested loops. They are suboptimal; for example, the distance and direction vectors used to represent data dependences are imprecise. Moreover, the different transformations interact with each other and should be considered together so as to handle arbitrary sequences and nestings of loops properly. Finally, it is a non-trivial engineering task to get all the algorithms to work together perfectly, and the heuristic nature of some of these algorithms makes the compiler behavior unpredictable.

This paper proposes an algorithm that can achieve the equivalent of all of the above algorithms within a unified framework of affine partitions. In a previous paper, we introduced the concept of an affine partition, which maps instances of instructions into the time or processor space via an affine expression in terms of the loop index values of their surrounding loops[14, 15]. The affine partitioning abstraction can be used to represent arbitrary combinations of unimodular transforms (interchange, reversal, skewing), reindexing, distribution, fusion, scaling, and statement reordering[1, 5, 11, 13, 17, 25]. From the affine mappings, a straightforward mathematical algorithm based on Fourier-Motzkin elimination[18] can be used to generate the SPMD (single program multiple data) code. We also showed an algorithm that finds an optimal affine partition that maximizes the degree of parallelism in a program while minimizing the degree of synchronization. The algorithm minimizes the frequency of communication, but not the volume of communication itself.

This paper shows that this powerful theoretical framework can be used to handle important practical tradeoffs that must be addressed in a parallelizing compiler. We

*Currently with Microsoft Research in Seattle, Washington.

This research was supported in part by ARPA and Air Force Materiel Command Contract F30602-95-C-0098, Department of Energy Contract B341491, and ARPA grant no. DABT63-94-C-0054.

present an algorithm based on the affine partitioning framework that minimizes communication by favoring near-neighbor communication over data restructuring and by using pipelined parallelism if necessary. We have implemented the basic algorithm in the Stanford SUIF Parallelizing compiler, with the aid of the Omega library[21]. We will substantiate the claim that the algorithm is more powerful by proving that it can find transformations not derivable using traditional loop transformation techniques and dependence vectors. We will also show examples of real programs to demonstrate the generality and effectiveness of our technique compared to other approaches.

The organization of the paper is as follows. We first introduce the model and definition of the affine partitioning framework in Section 2. Section 3 shows how to formulate three important subproblems in this framework: finding communication-free parallelism, finding pipelined parallelism, and allowing only near-communication across loops. In Section 4, we present our integrated parallelization algorithm. We discuss related work in Section 5 and illustrate the generality and effectiveness of the algorithm with three examples in Section 6. A short summary and concluding remarks are included in Section 7.

2 Definitions

We first introduce some notations and definitions that we use in the rest of the paper. We define our model of a program, its dependences and finally affine partition mappings.

Throughout the paper, we use the notation v_i to represent the ith element of the vector v, and $v_{i:j}$ to represent the subvector from the ith to the jth element of vector v. ($v_{i:j}$ is the empty vector if $i > j$).

2.1 Programs

The domain of our algorithm is the set of sequential programs with arbitrary nestings and sequences of loops, whose array indices and loop bounds are affine expressions of outer loop indices or loop-invariant variables. We capture the information our algorithm needs in the following abstract representation of the program.

Definition 2.1 A *program* is $P =< \mathcal{S}, \delta, \mathcal{D}, \mathcal{F}, \omega, \eta >$, where

- $\mathcal{S}$ is the set of instructions. An *instruction* is an indivisible unit such as a simple arithmetic operation on program variables. Instruction s appears lexically before instruction s' iff $s <_p s'$.
- δ_s is the *depth*, or the number of surrounding loops, of instruction s .
- $\mathcal{D}_s(\vec{i}) = D_s(\vec{i}) + \vec{d}_s$ is an affine expression derived from the bounds of the loops that surround instruction s. $\vec{i}$ is a valid *iteration* for instruction s iff $\mathcal{D}_s(\vec{i}) \geq \vec{0}$. The iteration vector $\vec{i}$ is similar to the usual concept of an "iteration space" vector, designating values for loop index variables, but also includes entries for the symbolic constants in the program. Loop bounds containing unknown variables or non-affine expressions can safely be ignored.
- $\mathcal{F}_{zsr}(\vec{i}) = F_{zsr}(\vec{i}) + \vec{f}_{zsr}$ is an affine expression that maps an iteration $\vec{i}$ to an element of array z given by the rth access function in instruction s. Scalar variables are treated as an array with one element. A non-affine array index is handled conservatively by assuming that it may take on any value.
- ω_{zsr} is true iff the rth access function in instruction s to array z is a write operation.
- $\eta_{ss'}$ is the number of common loops shared by instructions s and s'.

2.2 Data Dependences

The access patterns in a program define the constraints of program transformations. A *data dependence set* of a program contains all pairs of data dependent access functions in the program and is formally defined below.

Definition 2.2 We define $\prec$ to be the "lexicographically less than" operator for program $P =< \mathcal{S}, \delta, \mathcal{D}, \mathcal{F}, \omega, \eta >$ such that $\vec{i} \prec_{ss'} \vec{i'}$ iff iteration $\vec{i}$ of instruction s is executed before iteration $\vec{i'}$ of s' in P. That is,

$$\begin{aligned} &\forall s, s' \in \mathcal{S}.\ \forall \vec{i} \in \mathbf{Z}^{\delta_s},\ \vec{i'} \in \mathbf{Z}^{\delta_{s'}} \ \text{ s.t.} \\ &\mathcal{D}_s(\vec{i}) \geq \vec{0} \wedge \mathcal{D}_{s'}(\vec{i'}) \geq \vec{0}, \\ &\quad \vec{i} \prec_{ss'} \vec{i'} \quad \equiv \quad (\exists m \leq \eta_{ss'} \text{ s.t.} \\ &\qquad\qquad \vec{i}_{1:m} = \vec{i'}_{1:m} \wedge \vec{i}_{m+1} < \vec{i'}_{m+1}) \\ &\qquad\qquad \vee\ (\vec{i}_{1:\eta_{ss'}} = \vec{i'}_{1:\eta_{ss'}} \wedge s <_p s') \end{aligned} \quad (1)$$

Definition 2.3 The *data dependence set* of a program $P =< \mathcal{S}, \delta, \mathcal{D}, \mathcal{F}, \omega, \eta >$ is

$$R = \left\{ < \mathcal{F}_{zsr}, \mathcal{F}_{zs'r'} > \left| \begin{array}{l} (\omega_{zsr} \vee \omega_{zs'r'}) \ \wedge \\ (\exists \vec{i} \in \mathbf{Z}^{\delta_s},\ \vec{i'} \in \mathbf{Z}^{\delta_{s'}} | (\vec{i} \prec_{ss'} \vec{i'}) \\ \ \wedge (\mathcal{F}_{zsr}(\vec{i}) - \mathcal{F}_{zs'r'}(\vec{i'}) = \vec{0}) \\ \ \wedge (\mathcal{D}_s(\vec{i}) \geq \vec{0} \wedge \mathcal{D}_{s'}(\vec{i'}) \geq \vec{0})) \end{array} \right. \right\}$$

Definition 2.4 Let R be the data dependence set for program $P =< \mathcal{S}, \delta, \mathcal{D}, \mathcal{F}, \omega, \eta >$. The *program dependence graph* for P is $G = (V, E)$, where node $v_s \in V$ represents instruction s, and $< v_s, v_{s'} > \in E$ iff $\exists z, r, r'$ s.t. $< \mathcal{F}_{zsr}, \mathcal{F'}_{zs'r'} > \in R$.

2.3 Affine Partition Mappings

An affine partition mapping is an affine expression that maps the dynamic instances of each instruction, identified by their iteration vector, to a partition number. A *space-partition mapping* maps the instances of instructions to processors, and a *time-partition mapping* maps the instances of instructions to iterations in a sequential loop. In the following, we define affine partition mappings formally and introduce a few properties used in the algorithms.

Definition 2.5 An m-dimensional *affine partition mapping* for instruction s in program P is an m-dimensional affine expression $\Phi_s(\vec{i}) = C_s\vec{i} + \vec{c}_s$ (with rational coefficients), which maps an instance of instruction s, indexed by $\vec{i}$, to an m-element vector. An m-dimensional affine partition mapping for a program P is $\Phi = [\Phi_1, \Phi_2, ...\Phi_k]$, where k is the number of instructions in P.

Definition 2.6 The *rank* of an affine partition mapping for an instruction s, $\Phi_s(\vec{i}) = C_s\vec{i} + \vec{c}_s$, is the rank of matrix C_s. The *rank* of an affine partition mapping Φ for a program is

the maximum of the ranks of the affine partition mappings for its instructions.

Definition 2.7 Two one-dimensional affine partition mappings for instruction s, $\Phi_s(\vec{y}) = C_s\vec{y} + c_s$ and $\Phi'_s(\vec{y}) = C'_s\vec{y} + c'_s$, are *linearly dependent* iff C_s and C'_s are linearly dependent. Two one-dimensional affine partition mappings Φ and Φ' for program P are *linearly dependent* iff Φ_s and Φ'_s are linearly dependent for all instructions s in P.

3 Affine Partitioning Constraints

Our algorithm to maximize parallelism and minimize communication needs to solve three subproblems:

1. Find affine partitions that maximize the synchronization-free parallelism in a whole program.
2. Find affine partitions that maximize pipelined parallelism and inner loop parallelism.
3. Find affine partitions that minimize major data restructuring and shuffling between strongly connected components in a program dependence graph.

We show below how we can express simply the necessary and sufficient constraints on the desired solution within the affine framework. These constraints can be transformed into a system of linear inequalities through the use of the Farkas lemma[18, 7]. The goal of maximizing the degree of parallelism can be achieved by finding the maximally independent solutions to the constraints, that is, finding the basis vectors of the null space of the linear inequality constraints. The first two subproblems were discussed in a previous paper, which also described the details of the algorithm to find solutions to these constraints[14, 15]. This section presents the background on the first two subproblems and discusses in detail the third constraint which provides the key to minimizing communication across different strongly connected components.

3.1 Synchronization-Free Parallelism

The most effective way to parallelize a computation is to divide it into totally independent units, if it is possible, and assign them to different processors. The processors can all start to execute in parallel from the beginning and need not synchronize with each other. The *Space-Partition Constraint*, as defined below, captures precisely the necessary and sufficient constraint for an affine mapping to divide the computation into synchronization-free partitions.

Definition 3.1 (*Space-Partition Constraint*) Let R be the data dependence set for program $P =< \mathcal{S}, \delta, \mathcal{D}, \mathcal{F}, \omega, \eta >$. An affine space-partition mapping Φ for P is *synchronization-free* iff

$$\begin{aligned} &\forall < \mathcal{F}_{zsr}, \mathcal{F}_{zs'r'} > \in R,\ \vec{i} \in \mathbf{Z}^{\delta_s},\ \vec{i'} \in \mathbf{Z}^{\delta_{s'}} \ \text{ s.t.} \\ &\mathcal{D}_s(\vec{i}) \geq \vec{0} \wedge \mathcal{D}_{s'}(\vec{i'}) \geq \vec{0} \wedge \mathcal{F}_{zsr}(\vec{i}) - \mathcal{F}_{zs'r'}(\vec{i'}) = \vec{0}, \\ &\Phi_{s'}(\vec{i'}) - \Phi_s(\vec{i}) = 0 \end{aligned} \tag{2}$$

This constraint simply states that data dependent operations must be placed in the same partition. That is, if iteration $\vec{i}$ of instruction s and iteration $\vec{i'}$ of instruction s' access the same data, then their mappings Φ_s and $\Phi_{s'}$ are constrained to assign these iterations to the same space partition, i.e. $\Phi_{s'}(\vec{i'}) - \Phi_s(\vec{i}) = 0$. Finding the set of highest-ranked affine mappings that satisfy this constraint maximizes the degree of synchronization-free parallelism[14, 15].

Operations that access the same read-only data are not constrained to map to the same partition. It is assumed that read-only data is replicated and initialized accordingly at the beginning of the code. Thus, we consider an affine mapping that is synchronization-free also communication-free.

3.2 Parallelism with O(n) Synchronization

Given a strongly connected component in a program dependence graph with no synchronization-free parallelism, our second subproblem is to extract the maximum degree of parallelism possible while permitting only $O(n)$ synchronizations, where n is the number of iterations in a loop. We formulate the problem as partitioning the computation into iterations of a legal outer sequential loop, with the objective that the degree of parallelism is maximized for each time partition. The *Time-Partition Constraint*, as defined below, is the necessary and sufficient constraint of a *legal* affine time-partition mapping, that is, a mapping that does not violate any data dependences.

Definition 3.2 (*Time-Partition Constraint*) Let R be the data dependence set for program $P =< \mathcal{S}, \delta, \mathcal{D}, \mathcal{F}, \omega, \eta >$. A one-dimensional affine time-partition mapping Φ for P is *legal* iff

$$\begin{aligned} &\forall < \mathcal{F}_{zsr}, \mathcal{F}_{zs'r'} > \in R,\ \vec{i} \in \mathbf{Z}^{\delta_s},\ \vec{i'} \in \mathbf{Z}^{\delta_{s'}} \ \text{ s.t.} \\ &\quad \vec{i} \prec_{ss'} \vec{i'} \ \wedge\ \mathcal{D}_s(\vec{i}) \geq \vec{0} \wedge \mathcal{D}_{s'}(\vec{i'}) \geq \vec{0} \\ &\quad \wedge\ \mathcal{F}_{zsr}(\vec{i}) - \mathcal{F}_{zs'r'}(\vec{i'}) = \vec{0}, \\ &\quad \Phi_{s'}(\vec{i'}) - \Phi_s(\vec{i}) \geq 0 \end{aligned} \tag{3}$$

This constraint simply states that if iteration $\vec{i'}$ of instruction s' depends on iteration $\vec{i}$ of instruction s, then $\vec{i'}$ must be assigned to a time partition that executes no earlier than the partition containing $\vec{i}$, i.e. $\Phi_{s'}(\vec{i'}) - \Phi_s(\vec{i}) \geq 0$.

The *maximally independent* solutions to this constraint can be found by an algorithm similar to the one for finding solutions to the Space-Partition Constraint[14, 15]. There is always at least one solution, which corresponds to the original sequential execution order. If there are m independent solutions, then the program has at least $m-1$ degrees of parallelism that requires $O(n)$ synchronization. Moreover, the $m-1$ degrees of parallelism available can be pipelined, which means that processors only need to synchronize and, in most cases, communicate with their neighbors. Blocking can be used to reduce the frequency and volume of the communication.

One way to implement pipelined parallelism is to use any $m-1$ of the legal time partitions as components of an $(m-1)$-dimensional space partition, and use the remaining legal time partition to serialize the computation assigned to each processor. With this scheme, data only flow from space partitions with lower partition numbers to higher ones. This is but one solution; reversing the space partition assignment, for example, will yield an equivalent pipelining solution. We can include such partitions in the solution space by simply modifying the $\Phi_{s'}(\vec{i'}) - \Phi_s(\vec{i}) \geq 0$ condition in the Time-Partition Constraint to allow also $\Phi_{s'}(\vec{i'}) - \Phi_s(\vec{i}) \leq 0$.

This extra degree of freedom is useful if we wish to minimize communication across several pipelined loops that ac-

cess data in opposite directions. Simple code inspection is usually sufficient to determine the preferable order in which to number the partitions.

3.3 Near-Neighbor Communications Across Components

When we minimize synchronization, we minimize the frequency, but not the volume, of communication. For example:

```
Example 1: Original Code
  FOR i = 1 to N
    FOR j = 1 to N
      B[i,j] = f(A[i,j])                          (S1)
  FOR i = 1 to N
    FOR j = 1 to N
      C[i,j] = g(B[i,j], B[i,j-1], B[i-1,j])      (S2)
  FOR i = 1 to N
    FOR j = 1 to N
      D[i,j] = h(B[j,i])                          (S3)
```

Each of the three loops in this example has two degrees of parallelism. Because of the accesses of `B[i,j]`, `B[i,j-1]`, `B[i-1,j]` in the second loop, the processors need to synchronize after executing the first loop in parallel. The amount of communication transferred between processors is highly sensitive to how the computation is distributed across the processors. For this example, it is desirable that the same processor be responsible for writing to the locations of `B[i,j]`, `C[i,j]` and `D[j,i]`. In addition, by assigning two-dimensional blocks of contiguous iterations to the same physical processor, each processor operates on a block of the `B` matrix and only the boundaries of the block need to be communicated between neighboring processors. In contrast, if the same processor were assigned the iterations that write to `B[i,j]` and `D[i,j]`, the entire matrix `B` must be transposed. Data restructuring of this sort is much more expensive than just transferring the boundaries of blocks of data between neighbors.

To develop an affine partitioning algorithm that favors near-neighbor communication over complete date restructuring, we need to define a constraint that distinguishes between the communication needs of different affine partitions:

Definition 3.3 (*Near-Neighbor-Communication Constraint*) Let R be the data dependence set for program $P =< \mathcal{S}, \delta, \mathcal{D}, \mathcal{F}, \omega, \eta >$. An affine space-partition mapping Φ for P requires near-neighbor communications across components iff

$$\forall s, s' \text{ in different strongly connected components}$$
$$\forall < \mathcal{F}_{zsr}, \mathcal{F}_{zs'r'} >\in R\ \exists \gamma \in \mathbf{Z}\ \forall \vec{i} \in \mathbf{Z}^{\delta_s}, \vec{i'} \in \mathbf{Z}^{\delta_{s'}} \quad \text{s.t.}$$
$$\mathcal{D}_s(\vec{i}) \geq \vec{0} \wedge \mathcal{D}_{s'}(\vec{i'}) \geq \vec{0} \wedge \mathcal{F}_{zsr}(\vec{i}) - \mathcal{F}_{zs'r'}(\vec{i'}) = \vec{0},$$
$$\Phi_{s'}(\vec{i'}) - \Phi_s(\vec{i}) = \gamma$$

This constraint states that dependent operations from different strongly connected components must be mapped to partitions separated by a fixed constant. In most programs, these constants are small, and if we assign blocks of contiguous partitions to the same processor, only a small of data need to be communicated between neighbors. This constraint can be solved using the same algorithm as that for solving the Space-Partition Constraint. Note that if the constraint can be solved with all the γ's set to 0, then the solution is a communication-free affine partition.

3.4 Doall Parallelism with Near-Neighbor Communication

By adding the Near-Neighbor-Communication Constraint to the Space-Partition Constraints for each strongly connected component, we reduce the solution space to those affine partitions that parallelize the program using only $O(1)$ synchronizations and near-neighbor communication. By formulating the parallelization and communication minimization constraints all in the affine framework, we can use the same solution technique to find among all the possible parallelization schemes the one that requires only near-neighbor communication.

For example, if we apply this technique to Example 1 in Section 3.3, we get the desired affine partitions as discussed above:

$$\Phi_1 = \begin{bmatrix} i \\ j \end{bmatrix},\ \Phi_2 = \begin{bmatrix} i \\ j \end{bmatrix},\ \Phi_3 = \begin{bmatrix} j \\ i \end{bmatrix}$$

Only a small portion of array `B` needs to be communicated if blocks of partitions are assigned to the same processor. Assuming that there are `NP1 x NP2` processors, and that the processor ID is `(p1,p2)`, a straightforward algorithm based on Fourier-Motzkin will directly translate the blocked partitions to the following SPMD code:

```
Example 1: SPMD Code
  b1 = ceiling(N / NP1), b2 = ceiling(N / NP2)
  FOR u = (b1 * p1 + 1) to min(b1 * (p1 + 1), N)
    FOR v = (b2 * p2 + 1) to min(b2 * (p2 + 1), N)
      B[u,v] = f(A[u,v])
  [synchronization]
  FOR u = (b1 * p1 + 1) to min(b1 * (p1 + 1), N)
    FOR v = (b2 * p2 + 1) to min(b2 * (p2 + 1), N)
      C[u,v] = g(B[u,v], B[u,v-1], B[u-1,v])
  FOR u = (b1 * p1 + 1) to min(b1 * (p1 + 1), N)
    FOR v = (b2 * p2 + 1) to min(b2 * (p2 + 1), N)
      D[v,u] = h(B[u,v])
```

3.5 Pipelined Parallelism with Near-Neighbor Communication

If we cannot find a parallelization scheme that uses only near-neighbor communication by synchronizing $O(1)$ times, it is often more preferable to introduce slightly more synchronizations than to introduce data restructuring. The use of pipelining offers more opportunities to find parallelism that uses only near-neighbor communication while incurring only $O(n)$ synchronizations. Recall that a strongly connected component has pipelined parallelism if there is more than one solution to its Time-Partition Constraint. If, in addition, we can find one solution that satisfies the Time-Partition Constraint as well as the Near-Neighbor-Communication Constraint with other strongly connected components, it corresponds to a space partition that requires only near-neighbor communication.

Let us use a simple example from an ADI (Alternating Direction Implicit) integration algorithm to illustrate the usefulness of pipelining.

```
Example 2: Original ADI Code
  FOR i = 0 to N
    FOR j = 1 to N
      A[i,j] = f(A[i,j], A[i,j-1])          (S1)
  FOR i = 1 to N
    FOR j = 0 to N
      A[i,j] = g(A[i,j], A[i-1,j])          (S2)
```

It is obvious that we can parallelize the i-dimension of the first loop and the j-dimension of the second loop. Such a parallelization scheme, however, requires transposing the array `A` across the processors at the end of the first loop. If we pipeline the i-dimension of the second loop instead, only

a small amount of near-neighbor communication is necessary. This scheme can be found simply by solving for an affine partition that satisfies the Time-Partition Constraint for each loop and the Near-Neighbor-Communication Constraint across the loops.

For this ADI example, our algorithm finds two solutions that satisfy the Time-Partition Constraints for both loops and the Near-Neighbor-Communication Constraint:

$$\Phi^1 = \left\{\Phi^1_1 = i,\ \Phi^1_2 = i\right\}$$
$$\Phi^2 = \left\{\Phi^2_1 = j,\ \Phi^2_2 = j\right\}$$

We can distribute either the i or the j dimensions of the two loops. We cannot distribute both dimensions because there is only one degree of parallelism in each loop. If we choose to distribute the i dimension in each loop, the second loop is pipelined. By assigning a block of contiguous partitions to the same processor, each processor only needs to transfer a fraction of a row of matrix `A` to its neighbor. Assuming that there are `NP` processors and that `p` is the processor id, the SPMD code generated for this example is:

```
Example 2: SPMD Code
  b = ceiling((N+1)/NP)
  FOR j = 1 to N
    FOR i = (b * p) to min(b * (p + 1) - 1, N)
      A[i,j] = f(A[i,j], A[i,j-1])
  FOR j = 0 to N
    [synchronization: wait for processor p-1 to finish iteration j]
    FOR i = max(b * p, 1) to min(b * (p + 1) - 1, N)
      A[i,j] = g(A[i,j], A[i-1,j])
    [synchronization: signal processor p+1 that iteration j is done]
```

4 The Algorithm

We can maximize parallelism and minimize communication by applying a succession of affine partitioning algorithms that allows more and more communication until sufficient parallelism is found. Such an approach, however, would be quite inefficient because as more communication is allowed, the constraints become more relaxed, and some of the same constraints are solved over and over again.

Our algorithm instead uses a recursive-descent approach to find the outermost parallelism available in each strongly connected component in the program, and incrementally adds more and more constraints to reduce the communication requirements starting from the inner levels of parallelism identified.

Algorithm 4.1 Maximize the degree of parallelism in a program dependence graph while minimizing communication.

Step 1: Break the dependence graph into strongly connected components.

Step 2: For each strongly connected component, solve its Space-Partition Constraint to find a non-trivial affine partition that yields communication-free parallelism.

Step 3: Determine if the graph can be parallelized with only $O(1)$ synchronization and near-neighbor communication. That is, find non-trivial communication-free affine partitions for all components that also satisfy the Near-Neighbor-Communication Constraint across the components.

Step 4: Skip to the next step if the answer to Step 3 is yes. This step looks for pipelined parallelism as well as inner loop parallelism if necessary, and avoids restructuring data between parallel loops.

1. For each strongly connected component, solve for affine partitions that satisfy its Time-Partition Constraint. If there are multiple independent solutions, the component can be pipelined. Note that a component may have both synchronization-free and pipelined parallelism. If a component has neither form of parallelism, it is a sequential loop. If the sequential loop contains other loops, recursively apply this algorithm to the dependence graph of its loop body to seek parallelism at an inner level.

2. If two or more strongly connected components in the dependence graph is found to contain parallelism, apply the following to minimize the need for restructuring data between components.

   We create a DAG (directed acyclic graph) whose nodes are the strongly connected components in the dependence graph and whose edges are the data dependences across the parallelizable regions of the components. The edges are weighted by the estimated volume of data shared. Since these weights are used only in prioritizing the order in which the dependences are considered, they need not be computed precisely.

   We observe that node n_2 can be executed immediately after n_1 if and only if n_1 does not depend on n_2 and there does not exist a distinct node m such that m depends on n_1 and n_2 depends on m. From among all the edges connecting pairs of nodes that can be run consecutively, we select the one with the greatest weight and remove it from the graph. We construct the Near-Neighbor-Communication Constraint for this data dependence edge and use it to constrain the affine partitions found for the two nodes. We succeed if non-trivial solutions exist for the nodes, and we require these two nodes be executed consecutively. We capture this requirement by creating a super node to contain these two nodes, unless such a node already exists. The super node inherits all the dependence edges that connect these two nodes with external nodes. However, if the internal nodes depend on the same external node due to the same data structure, only the edge from the earlier node is kept; similarly, if an external node depends on both of the internal nodes due to the same data structure, only the edge to the latter node is kept. We iterate this process of selecting a data dependence edge from the DAG to constrain the affine partitions for the nodes until all the dependences in the graph are considered.

   If the affine partitions found still contain multiple degrees of parallelism, we can reduce the need for data replication by considering the sharing of read data. A similar procedure adapted from the above can be used.

Step 5: This step tries to eliminate some of the near-neighbor communication found above, using a greedy algorithm similar to that in Step 4. Again, we use the amount of data shared by the strongly connected components to order our attempts to reduce near-neighbor communication. Affine partitions that satisfy the Near-Neighbor Communication Constraint may map iterations related by a data dependence to partitions that are some fixed constant apart.

Here we try to find affine partitions that can satisfy the additional constraint that the constant be zero. If we succeed, then no communication is necessary.

Step 6: We generate a sequential ordering for the strongly connected components in the program dependence graph by visiting the nodes in the DAG obtained above in topological order. Each super node is expanded (recursively) into a sequence that satisfies the dependences.

Although the algorithm above uses a relatively simple greedy approach in Steps 4 and 5 to minimize communication, it has the following important properties:

1. If a program can be parallelized with only near-neighbor communication and $O(1)$ synchronizations, Step 3 of the algorithm will find the maximum degrees of such parallelism.
2. If a program can be parallelized with only near-neighbor communication and $O(n)$ synchronizations, Step 4 of the algorithm will find the maximum degrees of such parallelism.
3. If a program can be parallelized without requiring any synchronizations at all, Step 5 of the algorithm will find the maximum degrees of such parallelism.

5 Comparison with Other Algorithms

The affine partitioning framework unifies a large number of previously proposed loop transformations, including unimodular transformations (interchange, reversal, skewing), reindexing, distribution, fusion, scaling, and statement reordering[1, 5, 11, 13, 17, 25]. We have a provably optimal algorithm that finds the affine transform that maximizes the degree of parallelism while minimizing the degree of synchronization[14, 15]. It is thus more powerful than previous approaches that use heuristics or pre-defined sequences of transforms to optimize code[16, 19, 22, 25]. It also subsumes previous work that uses unimodular transforms to maximize coarse-grain parallelism[23, 24]. Unimodular transforms are applicable only to perfectly nested loops and they models entire loop bodies as atomic units, whereas the affine partitioning framework is applicable to arbitrary sequences and nests of loops and it can optimize instructions in a loop individually. The affine partitioning framework captures the dependences of affine accesses precisely, in contrast to the imprecise abstractions of distance and direction vectors used in most loop transformations. Algorithms based on loop-transformation generally translate sequential loop nests into first semantically equivalent sequential loop nests. Our approach of generating the parallel SPMD code directly can produce more intricate parallelization schemes.

Anderson and Lam proposed a two-step data and computation decomposition algorithm to optimize across multiple loop nests[3]. Their algorithm first applies unimodular transformation to expose the parallelism in perfectly nested loop nests, then aligns and chooses the parallelism to minimize communication. This latter step also prefers near-neighbor communication and pipelined parallelism over data restructuring, and uses a greedy approach to minimize communication cost when necessary. In contrast, we have a more general and powerful algorithm that optimizes across sequences and nests of loops. It is guaranteed to find the best affine partition that maximizes the degree of parallelism requiring (1) no communication, (2) $O(1)$ synchronization and near-neighbor communication, and (3) $O(n)$ synchronization and near-neighbor communication. Unlike Anderson and Lam's algorithm, our algorithm will also re-order the execution of parallel loops to minimize data restructuring.

Kodukula et al. proposed the idea of data shackling whereby a compiler uses the owner-compute rule to distribute a program with arbitrary loop nests and sequences according to the user's data distribution directives[12]. In contrast, our algorithm does not require any user input. It finds automatically the optimal distribution of computation from partitioning constraints that minimizes both the frequency and volume of communication. The data distributions are not explicitly computed in our algorithm. However, once the computation partitioning is derived, it is easy to generate the data distributions expected for each section of the code. The data distributions we generate are more flexible as they may change dynamically.

There have also been other proposals to use affine transformations for parallelization. Feautrier has developed an algorithm that finds a piece-wise affine schedule to minimize the overall execution time of a program[7, 8]. Heuristics based on parametric integer programming are used to minimize the dimensionality of time. This problem formulation specifies when but not where each instruction is executed. Communication is not modeled and must be handled by postpass techniques[9]. Also, while translation from the original program to SPMD code is straightforward in our framework, it is nontrivial to translate the piece-wise affine schedules into parallel code.

Attempts have also been made to derive affine schedules that expose coarse-grain parallelism. Darte proposed an algorithm for detecting permutable loops for the restricted domain of perfectly nested loops[6]. Kelly and Pugh's algorithm finds one dimension of parallelism for programs with arbitrary nestings and sequences of loops[10]. Their repertoire of program transforms include loop permutations and reversals, but not loop skewing. The exclusion of loop skewing enables them to enumerate all the possible transformation choices and select the one with the lowest estimated communication cost.

6 Experiments

In this section, we use a set of three examples to illustrate the generality and effectiveness of our affine partitioning framework. First, we show that our new algorithm minimizes communication across multiple loops and creates a result equivalent to applying a series of state-of-the-art algorithms. Then, we demonstrate how our algorithm derives the best transform readily from our mathematical model for cases that other techniques would find difficult to optimize. Finally, we use the third example to substantiate the claim that the algorithm is more powerful by proving that it can find transformations not derivable using traditional loop transformation techniques and dependence vectors.

6.1 Communication Across Loops

We first illustrate how our algorithm minimizes communication, using the Tomcatv program from the SPEC95fp benchmark as an example. We include the entire kernel of the code here to illustrate all the practical issues that must be handled.

```
RXM(ITER)  = 0.D0
RYM(ITER)  = 0.D0
```

```
      M = N - 1

      DO     60    J = 2,M                                          (L1)
C
        DO     50    I = 2,M                                        (L2)
          XX = X(I+1,J)-X(I-1,J)
          YX = Y(I+1,J)-Y(I-1,J)
          XY = X(I,J+1)-X(I,J-1)
          YY = Y(I,J+1)-Y(I,J-1)
          A  = 0.25D0  * (XY*XY+YY*YY)
          B  = 0.25D0  * (XX*XX+YX*YX)
          C  = 0.125D0 * (XX*XY+YX*YY)
          AA(I,J) = -B
          DD(I,J) = B+B+A*REL
          PXX = X(I+1,J)-2.D0*X(I,J)+X(I-1,J)
          QXX = Y(I+1,J)-2.D0*Y(I,J)+Y(I-1,J)
          PYY = X(I,J+1)-2.D0*X(I,J)+X(I,J-1)
          QYY = Y(I,J+1)-2.D0*Y(I,J)+Y(I,J-1)
          PXY = X(I+1,J+1)-X(I+1,J-1)-X(I-1,J+1)+X(I-1,J-1)
          QXY = Y(I+1,J+1)-Y(I+1,J-1)-Y(I-1,J+1)+Y(I-1,J-1)
C
C     CALCULATE RESIDUALS (EQUAL TO RIGHT HAND SIDES.)
C
          RX(I,J)   = A*PXX+B*PYY-C*PXY
          RY(I,J)   = A*QXX+B*QYY-C*QXY
C
   50     CONTINUE
   60   CONTINUE
C
C     DETERMINE MAXIMUM VALUES RXM, RYM OF RESIDUALS
C
        DO     80    J = 2,M                                        (L3)
          DO     80    I = 2,M                                      (L4)
            RXM(ITER) = MAX(RXM(ITER), ABS(RX(I,J)))
            RYM(ITER) = MAX(RYM(ITER), ABS(RY(I,J)))
   80   CONTINUE
C
C     SOLVE TRIDIAGONAL SYSTEMS (AA,DD,AA)
C     IN PARALLEL, LU DECOMPOSITION
C
        DO     90    I = 2,M                                        (L5)
          D(I,2) = 1.D0/DD(I,2)
   90   CONTINUE
        DO    100    J = 3,M                                        (L6)
          DO    100    I = 2,M                                     (L7)
            R      = AA(I,J)*D(I,J-1)
            D (I,J) = 1.D0/(DD(I,J)-AA(I,J-1)*R)
            RX(I,J) = RX(I,J) - RX(I,J-1)*R
            RY(I,J) = RY(I,J) - RY(I,J-1)*R
  100   CONTINUE
        DO    110    I = 2,M                                        (L8)
          RX(I,M) = RX(I,M)*D(I,M)
          RY(I,M) = RY(I,M)*D(I,M)
  110   CONTINUE
        DO    120    J = N-2,2,-1                                   (L9)
          DO    120    I = 2,M                                     (L10)
            RX(I,J) = (RX(I,J)-AA(I,J)*RX(I,J+1))*D(I,J)
g           RY(I,J) = (RY(I,J)-AA(I,J)*RY(I,J+1))*D(I,J)
  120   CONTINUE
C
C     ADD CORRECTIONS OF ITER ITERATION
C
        DO    130    J = 2,M                                       (L11)
          DO    130    I = 2,M                                     (L12)
            X(I,J) = X(I,J)+RX(I,J)
            Y(I,J) = Y(I,J)+RY(I,J)
  130   CONTINUE
C
        ABX  = ABS(RXM(ITER))
        ABY  = ABS(RYM(ITER))
C
```

The compiler must first apply well-known parallelization techniques such as algorithms to recognize reductions and private variables to minimize the dependences in the program. There are, for example, many privatizable scalar variables in the first loop nest (`L1-L2`) and reduction operations in the second nest (`L3-L4`).

Our affine partitioning algorithm starts by constructing the dependence graph for this program and dividing it into strongly connected components. In the second step, the algorithm solves the Space-Partition Constraint for each component, and finds at least one degree of communication-free parallelism for every instruction in a loop. In step three, the algorithm finds that parallelizing the i-dimension of all the loops in the program requires only near-neighbor communication. The algorithm thus skips the fourth step. Step five places all but the last loop nest (`L11-12`) and operations not nested in any loops into one communication-free region. Thus, only one synchronization needs to be introduced before the last loop nest. Note that the loops `L3-L4` are parallelized by recognizing that the updates to the `RXM(ITER) and RYM(ITER)` are reduction operations. Each processor accumulates to its private version of `RXM(ITER) and RYM(ITER)`. The results are combined at the end of the communication-free region, after the one synchronization operation in the loop.

This parallelization scheme is similar to that obtained by applying a combination of unimodular transforms, Anderson and Lam's data and computation distribution algorithm, followed by Tseng's barrier synchronization elimination algorithm. Near perfect speedup was obtained with this scheme[2]. The difference between this algorithm and the previous approach is that here we use one unified framework to parallelize the code and minimize the synchronization and communication at the same time.

6.2 Imperfectly Nested Loops

To illustrate how our algorithm outperforms other techniques in handling imperfectly nested loops and sequences of loops, we have chosen as our second example the Cholesky subroutine from the SPEC92 nasa7 benchmark.

```
      DO 1 J = 0, N                                                 (L1)
        I0 = MAX ( -M, -J )

        DO 2 I = I0, -1                                             (L2)
          DO 3 JJ = I0 - I, -1                                      (L3)
            DO 3 L = 0, NMAT                                        (L4)
3             A(L,I,J) = A(L,I,J) - A(L,JJ,I+J) * A(L,I+JJ,J)
          DO 2 L = 0, NMAT                                          (L5)
2           A(L,I,J) = A(L,I,J) * A(L,0,I+J)

        DO 4 L = 0, NMAT                                            (L6)
4         EPSS(L) = EPS * A(L,0,J)
        DO 5 JJ = I0, -1                                            (L7)
          DO 5 L = 0, NMAT                                          (L8)
5           A(L,0,J) = A(L,0,J) - A(L,JJ,J) ** 2
        DO 1 L = 0, NMAT                                            (L9)
1         A(L,0,J) = 1. / SQRT ( ABS (EPSS(L) + A(L,0,J)) )

      DO 6 I = 0, NRHS                                             (L10)
        DO 7 K = 0, N                                              (L11)
          DO 8 L = 0, NMAT                                         (L12)
8           B(I,L,K) = B(I,L,K) * A(L,0,K)
          DO 7 JJ = 1, MIN (M, N-K)                                (L13)
            DO 7 L = 0, NMAT                                       (L14)
7             B(I,L,K+JJ) = B(I,L,K+JJ) -  A(L,-JJ,K+JJ) * B(I,L,K)

        DO 6 K = N, 0, -1                                          (L15)
          DO 9 L = 0, NMAT                                         (L16)
9           B(I,L,K) = B(I,L,K) * A(L,0,K)
          DO 6 JJ = 1, MIN (M, K)                                  (L17)
            DO 6 L = 0, NMAT                                       (L18)
6             B(I,L,K-JJ) = B(I,L,K-JJ) - A(L,-JJ,K) * B(I,L,K)
```

This code consists of 18 irregularly nested loops with some computation nested inside as many as 4 loops. The affine partitioning algorithm maps the entire computation onto `NMAT+1` totally independent partitions, by assigning to processor P all the computation involving data `A[P,*,*]`, `B[*,P,*]` and `EPSS[P]`. The SPMD code with `P` as the processor ID is shown below.

```
      DO 1 J = 0, N
        I0 = MAX ( -M, -J )

        DO 2 I = I0, -1
          DO 3 JJ = I0 - I, -1
3           A(P,I,J) = A(P,I,J) - A(P,JJ,I+J) * A(P,I+JJ,J)
2         A(P,I,J) = A(P,I,J) * A(P,0,I+J)

4       EPSS(P) = EPS * A(P,0,J)
        DO 5 JJ = I0, -1
5         A(P,0,J) = A(P,0,J) - A(P,JJ,J) ** 2
1       A(P,0,J) = 1. / SQRT ( ABS (EPSS(P) + A(P,0,J)) )

      DO 6 I = 0, NRHS
        DO 7 K = 0, N
8         B(I,P,K) = B(I,P,K) * A(P,0,K)
          Do 7 JJ = 1, MIN (M, N-K)
7           B(I,P,K+JJ) = B(I,P,K+JJ) -  A(P,-JJ,K+JJ) * B(I,P,K)

        DO 6 K = N, 0, -1
9         B(I,P,K) = B(I,P,K) * A(P,0,K)
          DO 6 JJ = 1, MIN (M, K)
6           B(I,P,K-JJ) = B(I,P,K-JJ) - A(P,-JJ,K) * B(I,P,K)
```

The resulting code does not require any synchronization operations. If we just analyze the last loop nest (loops `L10` to `L18`) in the program, we observe that there are in fact `NRHS x NMAT` independent partitions. That is, we can assign to processor `(P1,P2)` all computation that operates on data `B(P1,P2,*)`. However, when we consider the computation and usage of the array `A` in the rest of the program, we find that there is only one degree of communication-free parallelism in the entire program. This parallelization scheme is achieved by assigning all the computation involving `B[*,P,*]` to processor `P`.

Although the parallelization scheme found by our algorithm appears to be very simple, as far as we know, no other compiler can generate this scheme fully automatically. Algorithms based on unimodular transforms cannot handle the arbitrary nestings of loops directly. It is not possible to choose the right loops to parallelize by analyzing the individual perfect subnests separately. For this example, a unimodular transformer will parallelize the `L10` loop as it is an outermost loop. It can interchange the loops in the two perfect nests (`L13` and `L14`) and (`L17` and `L18`), but it would fail to recognize the possibility of parallelizing all the `L` loops within the loop nest `L10` to `L18`.

Because of the weakness of unimodular transformations, Anderson and Lam's algorithm cannot find the perfect parallelization scheme for this example. If the user specifies that arrays `A` and `B` are to be distributed along the first and second dimensions, respectively, then Kodukula et al.'s data shackling algorithm would generate a similar parallel code. Our algorithm, on the other hand, can find this mapping fully automatically.

Theoretically, we can achieve the desired effect via repeated applications of unimodular transforms and loop fusion. As this example shows, it is sometimes necessary to apply interchanges and fusions alternatively, and the number of transforms that need to be applied varies with the loop depth in the program. It is difficult to imagine any algorithm based on a pre-determined sequence of loop transforms can obtain such a transform. Our affine partitioning algorithm can find the best combination of unimodular transforms, fusion and fission, data and computation decompositions and barrier synchronization elimination. It is simpler and more effective than previous techniques.

To quantify the significance of our algorithm, we parallelized the Cholesky routines using three different techniques:

1. unimodular transform,
2. Anderson and Lam's data and computation decomposition followed by Tseng's barrier synchronization elimination algorithm, and
3. our affine partitioning algorithm.

The parallelized loops are block distributed among the processors in all the three cases, and all the loops are restructured so that the array `A` is accessed with unit strides to enhance uniprocessor locality.

The parallelized codes were run on a Digital Turbolaser with 8 300-Mhz 21164 processors. The performance results are presented in Figure 1. The code parallelized with unimodular transforms has about a 2-times speedup on 4 processors and the performance degrades as the number of processors increases to 8. The second scheme is slightly better than the first because it manages to eliminate some barrier synchronizations in the parallelized code. However, it fails to find the communication-free partition and its performance suffers from having to transpose the array `A`. Our affine partitioning algorithm achieves an impressive speedup of 6.2 times on 8 processors by eliminating all synchronization and communication between processors.

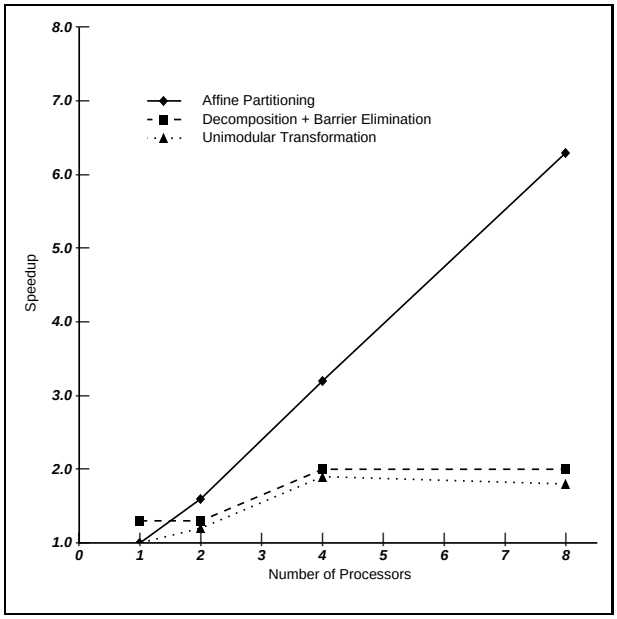

Figure 1: Performance for Cholesky in the SPEC92 NASA7 benchmark

6.3 Strength of the Affine Partitioning Approach

Finally, we show, by way of a 4-line example, how our new approach can produce solutions that cannot be derived using techniques based on dependence vectors and sequences of legal transformations.

```
FOR L1 = 1 to N
  FOR L2 = 1 to N
    A[L1,L2] = f(B[L1,L2])
    C[L1,L2] = g(A[L2,L1])
```

In this example, the first instruction simply applies the function `f` to each element in `B` and assigns the result to the corresponding element in `A`. The second instruction, however, is more complicated. It applies `g` to the old values in the upper left triangle of matrix `A` and the new values in the lower right triangle. The results are assigned to matrix `C`.

Since all the accesses are affine functions, the parallelization constraints are represented perfectly in our affine partitioning algorithm. The constraints for synchronization-free parallelization require that all operations using the same data be assigned to the same processor; that is, the (L1,L2) operation of instruction 1 must be paired with the (L2,L1) operation of instruction 2. The algorithm is able to find rank-2 affine mappings for both instructions that satisfy the Space-Partition Constraint, thus exposing 2 degrees of parallelism. Let $p = (p_1, p_2)$ be the processor id, then we have

$$\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} L1 \\ L2 \end{bmatrix} \quad \text{for instruction 1, and}$$

$$\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} L1 \\ L2 \end{bmatrix} \quad \text{for instruction 2.}$$

The code generation phase produces an SPMD program parameterized by processor id p1,p2. The routine first generates the loop bounds for each individual instruction, then merges the bounds such that the computation on each processor preserves the original sequential execution order. The algorithm automatically determines that the critical condition for instruction 2 to come before instruction 1 is p1>p2. The algorithm correctly inverts the execution order of instructions 1 and 2 for the portion of the computation that involves the upper triangle of the array, so that the appropriate old values are read before they are overwritten. The resulting SPMD program is:

```
IF p1 ≤ p2 THEN
  A[p1,p2] = f(B[p1,p2])
  C[p2,p1] = g(A[p1,p2])
ELSE
  C[p2,p1] = g(A[p1,p2])
  A[p1,p2] = f(B[p1,p2])
```

It is important for this example that the dependences of affine data accesses be captured exactly. Had the traditional dependence vectors been used, the dependences would have been represented as [<, *], which would have rendered the entire loop unparallelizable. The non-trivial SPMD code produced for this example also illustrates the generality and power of the algorithmic code generation technique.

More importantly, by directly solving the affine mappings for individual instructions, we can find parallelism that is not achievable via a sequence of legal loop transforms. For this example, the surrounding loops of the second instruction are interchanged while the first instruction is left unchanged. We cannot achieve this in the unimodular transform approach because it models each loop body as an atomic unit. Loop distribution is sometimes applied to create different loop bodies so that they can be manipulated separately, and fusion may be applied afterwards to put them back together. In this case, it is illegal to distribute the loop even if the dependences had been represented perfectly. Thus, not only is it difficult to find the right sequence of transformations to apply to a program, there may not exist a sequence of legal transformations that achieves the same effect.

7 Summary and Conclusions

This paper presents an algorithm based on the affine partitioning framework that maximizes parallelism while minimizing communication. Our previous paper showed how to maximize the degree of parallelism while minimizing the degree of synchronizations. This paper presents an algorithm that minimizes communication between parallel loops using the affine partitioning framework. The algorithm presented in this paper subsumes previous work that uses loop transforms (unimodular, loop fusion, fission, scaling, reindexing, and statement reordering) as well as previous data and computation distribution and barrier synchronization elimination algorithms.

We make a distinction between expensive communication that restructures data such as transposing an array across processors and cheap communication that shifts small amounts of data between neighboring processors. We introduce the notion of a Near-Neighbor-Communication Constraint to capture the conditions under which affine partitions require only near-neighbor communications across parallel loops. We can maximize parallelism and minimize communication simultaneously by simply solving for affine partitions that satisfy both the Near-Neighbor-Communication Constraint and the parallelization constraints.

Our algorithm first tries to expose the outermost levels of parallelism in the program, then uses a greedy approach that incorporates the communication constraints in decreasing order of importance. This simple procedure accomplishes several important tradeoffs. It may choose not to parallelize a loop if there are other alternatives that need less communication. It assigns the computation to processors in a way that minimizes communication across parallel loops. Finally, it will even use pipelining, which may increase the number of synchronizations, to avoid expensive data-restructuring communication. Although heuristics are used in our algorithm, it is optimal in maximizing the degrees of parallelism that require (1) no communication, (2) $O(1)$ synchronizations with only near-neighbor communication and (3) $O(n)$ synchronizations with only near-neighbor communication.

We illustrate the advantages of our algorithm using three carefully chosen examples. The Tomcatv example shows that our integrated algorithm can match the result generated by a series of state-of-the-art algorithms. The Cholesky example illustrates the algorithm's unique ability to find the best transform for programs with non-perfect loop nesting automatically. Finally, a small artificial example shows how our algorithm can find transforms that cannot be derived as a legal sequence of traditional loop transforms.

References

[1] J. R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 63–76, January 1987.

[2] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *Proceedings of the Fifth ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 166–178, July 1995.

[3] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 112–125, June 1993.

[4] U. Banerjee. Unimodular transformations of double loops. In *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, pages 192–219, August 1990.

[5] U. Banerjee. *Loop Transformations for Restructuring Compilers*. Kluwer Academic, 1993.

[6] A. Darte, G. Silber, and F. Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. Technical Report 96-34, Laboratoire de l'Informatique du Parallélisme, November 1996.

[7] P. Feautrier. Some efficient solutions to the affine scheduling problem, part I, one dimensional time. *International Journal of Parallel Processing*, 21(5):313–348, October 1992.

[8] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II, multidimensional time. *International Journal of Parallel Processing*, 21(6), December 1992.

[9] P. Feautrier. Towards automatic distribution. Technical Report 92.95, Institut Blaise Pascal/Laboratoire MASI, December 1992.

[10] W. Kelly and W. Pugh. Minimizing communication while preserving parallelism. In *Proceedings of the 1996 ACM International Conference on Supercomputing*, pages 52–60, May 1996.

[11] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, pages 323–334, July 1992.

[12] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 346–357, June 1997.

[13] D. J. Kuck, R. H. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[14] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1997.

[15] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3–4):445–475, May 1998.

[16] K. S. McKinley. Evaluating automatic parallelization for efficient execution on shared-memory multiprocessors. In *Proceedings of the 1994 ACM International Conference on Supercomputing*, July 1994.

[17] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[18] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, Chichester, 1986.

[19] K. Smith and B. Appelbe. Determining transformation sequences for loop parallelization. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, August 1992.

[20] C.-W. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the Fifth ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 144–155, July 1995.

[21] University of Maryland. *The Omega Library Version 1.1.0 Interface Guide*, November 1996.

[22] D. Whitfield and M. L. Soffa. Investigating properties of code transformations. In *Proceedings of the 1993 International Conference on Parallel Processing*. ACM, August 1993.

[23] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, August 1992. Published as CSL-TR-92-538.

[24] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *Transactions on Parallel and Distributed Systems*, 2(4):452–470, October 1991.

[25] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, MA, 1989.