Maximizing Parallelism and Minimizing Synchronization with Affine Transforms

Amy W. Lim and Monica S. Lam

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

Abstract

This paper presents the first algorithm to find the optimal affine transform that maximizes the degree of parallelism while minimizing the degree of synchronization in a program with arbitrary loop nestings and affine data accesses. The problem is formulated without the use of imprecise data dependence abstractions such as data dependence vectors. The algorithm presented subsumes previously proposed program transformation algorithms that are based on unimodular transformations, loop fusion, fission, scaling, reindexing and/or statement reordering.

1 Introduction

As multiprocessors become popular, it is important to develop compilers that can automatically translate sequential programs into efficient parallel code. Getting high performance on a multiprocessor requires not only finding parallelism in the program but also minimizing the synchronization overhead. Synchronization is expensive on a multiprocessor. The cost of synchronization goes far beyond just the operations that manipulate the locks or equivalent data structures. A processor may have to wait for the other processors to reach the same synchronization point. On a modern multiprocessor, even if all processors execute the same number of operations, their completion times can vary greatly due to the memory subsystem. Furthermore, high synchronization frequency is generally accompanied with high levels of data communication between processors, which again reduce the efficiency of the machine. In fact, experience with parallel applications on multiprocessors indicates that it is not uncommon for programs with fine-grain synchronization to run even slower than their serial counterpart. It is therefore important that a compiler find parallelism that requires minimal synchronization. In other words, we wish to identify the *coarsest granularity* of parallelism in a program by finding the largest units of independent computations that can be carried out by different processors without synchronization.

Research supported in part by the Air Force Materiel Command and ARPA, contract F30602-95-C-0098, and an NSF Young Investigator award.

Appears in the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, Jan 1997.

There has been a lot of research on program transformations to enhance a program's parallelism and data locality. The domain of such techniques is generally limited to loops, whose loop bounds and array accesses are affine functions of loop indices. Many forms of program transformations have been proposed, a catalog of which can be found in a survey by Bacon, Graham, and Sharp[5]. The key challenge that remains is how to combine these transformations optimally to achieve particular goals.

This paper presents an algorithm that finds the maximum degree of parallelism in the general program domain of arbitrarily nested loops and array accesses with affine index expressions. Moreover, the parallelism found is at the coarsest granularity. Informally, the degree of parallelism refers to the number of nested parallel loops; the granularity of parallelism is measured by the length of the computation in the parallelized loop.

Our algorithm is based on the concept of *affine partitioning*. Instances of each instruction are identified by the loop index values of their surrounding loops, and affine expressions are used to map these loop index values to a partition number. Partition numbers are used for two different purposes:

- *Space Partition.* Operations belonging to the same space partition are mapped to the same processor. One possible mapping is to assign operations in partition i to processor i.
- *Time Partition.* Operations belonging to time partition i execute before those in partition $i+1$. This execution order can be expressed as a loop, whose ith iteration executes all the operations belonging to partition i.

Our algorithm finds a combination of affine space and time partition mappings that maximizes the degree of parallelism with successively greater degrees of synchronization. The problem is formulated without using any imprecise abstractions, such as direction vectors, commonly used in existing parallelizing compilers. The algorithm can be used to find all the parallelism at the coarsest granularity, or just up to the degree needed to exploit a particular parallel hardware configuration. From these affine partition mappings, a straightforward algorithm based on Fourier-Motzkin elimination[20] can be used to generate the desired SPMD (Single Program Multiple Data) code.

The technique of affine partitioning can find all the loop level parallelism exposed through a combination of many previously defined transformations [1, 4, 6, 7, 16, 17, 19, 21, 22, 23], including

- loop fission (or loop distribution),
- loop fusion,
- unimodular transforms (interchange, reversal, skewing),
- loop scaling,
- loop reindexing (or index set shifting), and
- statement reordering.

The rest of the paper is organized as follows. We first explain the different forms of parallelism and present our problem statement in Section 2. In Section 3, we give an overview of our algorithm. We then formally define our model of a program and define affine partition mappings in Section 4. Next, we present algorithms for solving several subproblems that can be combined to find the maximum degree of coarse-grain parallelism. Finally, we discuss related work and conclude.

2 Forms of Parallelism

To motivate the problem this paper addresses, we explain the different forms of parallelism available in a program with a few simple examples.

2.1 Granularity of Parallelism

We say that a program has k *degrees of parallelism* if $O(n^k)$ units of computations can be executed in parallel, where n is the number of iterations in a loop. For example, a loop nest with k degrees of parallelism is one that can be transformed to contain k nested parallel loops. Loops with the same degree of parallelism may be differentiated by their granularity of parallelism, as illustrated by the following examples, all of which contain one degree of parallelism.

Example 1:
  for $l_1 = 1$ to n do
    for $l_2 = 1$ to n do
      $a[l_1, l_2] = a[l_1 - 1, l_2]$;

Example 2:
  for $l_1 = 1$ to n do
    for $l_2 = 1$ to n do
      $a[l_1, l_2] = a[l_1 - 1, l_2] + a[l_1 - 1, n - l_2 + 1]$;

Example 3:
  for $l_1 = 1$ to n do
    for $l_2 = 1$ to n do
      $a[l_1, l_2] = a[l_1 - 1, l_2] + a[l_1, l_2 - 1]$;

Figure 1 shows the iteration space for each of these programs. Each axis represents a loop, and the operation in each iteration $< l_1, l_2 >$ is represented by the point with coordinates (l_1, l_2). The arrows represent data dependences between the operations in the program. Each gray box groups together computation that is assigned to the same processor by a parallelization scheme. The thick lines in the figure represent barriers synchronizations.

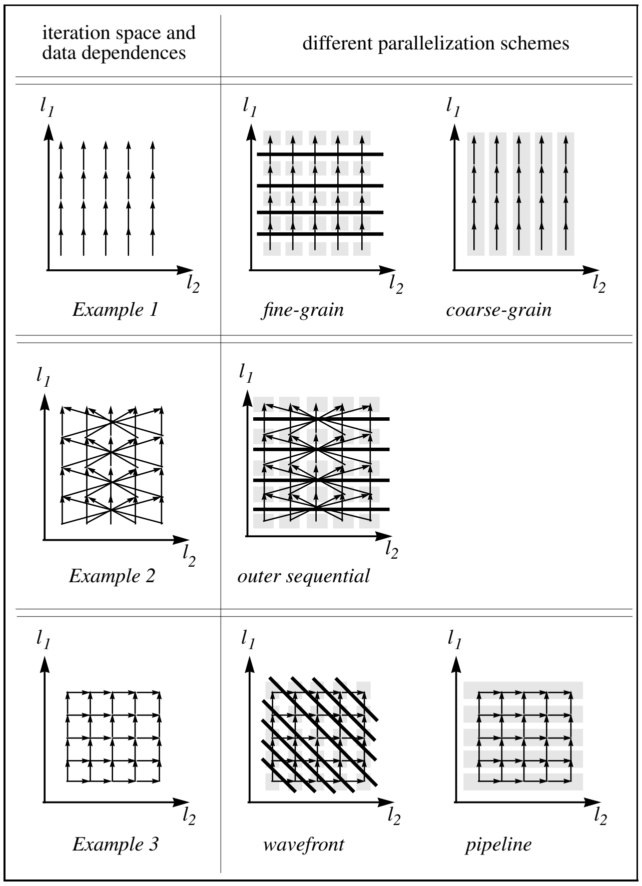

Figure 1: Iteration space, data dependences and parallelization schemes for Examples 1, 2, and 3.

Example 1 illustrates the difference between fine-grain and coarse-grain parallelism. There are many possible fine-grain parallelization schemes for this example, one of which is to execute each row of the iterations in parallel. A coarse-grain parallelization scheme, on the other hand, would specify that the different columns of iterations can execute in parallel. Both schemes expose the same degree of parallelism, but the latter allows the processors to run at their own pace without having to execute the same rows in lock step. Whereas $O(n)$ synchronizations are needed to enforce the fine-grain scheme, no synchronization is needed in the coarse-grain version.

Example 2 illustrates that it is not always possible to find *synchronization-free parallelism*, parallelism that requires no synchronization. Here, the only form of loop-level parallelism is to execute each row at a time. A barrier is needed to ensure that all processors have finished their assigned iteration before any can proceed to the next row.

Like Example 2, Example 3 does not have any synchronization-free parallelism. However, unlike Example 2, there is a choice in parallelization schemes. We can *wavefront* the computation; for example, we can have the processors execute iterations along a diagonal in parallel, as shown in the figure. A better choice, however, is to *pipeline* the computation. For example, by assigning a row to a processor, each processor can proceed as soon as its neighbor assigned the row below finishes executing from the same column.

Pipelining has many benefits over wavefronting: barriers are reduced to point-to-point synchronizations, processors need not work on the same wavefront at the same time, the SPMD code to implement pipelining is simpler, and finally the processors tend to have better data locality.

2.2 Degrees and Nesting of Parallelism

We say that different degrees of parallelism in a loop nest exist at the same nesting level if and only if they require the same amount of synchronization. For example, a 2-deep loop nest whose iterations are completely independent has two degrees of parallelism. If n is the number of iterations in each loop, n^2 processors can simultaneously execute one of the n^2 iterations and participate in one barrier synchronization at the end. The nesting structure of the original loop is immaterial, and we consider both of the degrees of parallelism in this case to be at the same level.

On the other hand, when one degree of parallelism in a loop nest requires more synchronization than another, we say that the former is *nested* within the latter. An example of nested levels of parallelism is as follows:

Example 4:

for $l_0 = 1$ to n do
  for $l_1 = 1$ to n do
    for $l_2 = 1$ to n do
      $a[l_0, l_1, l_2] = a[l_0, l_1 - 1, l_2]$
        $+ a[l_0, l_1 - 1, n - l_2 + 1];$

Each of the iterations of the outermost loop performs the same computation as Example 2 on a disjoint set of data. The outermost loop is trivially parallelizable, allowing n processors to execute in parallel without any synchronization at all. If, however, we wish to keep n^2 processors busy, we can assign each iteration of the outermost loop to a set of n processors. Each set of n processors must participate in n barrier synchronizations, one for each iteration in the middle loop, as discussed above. This example illustrates that degrees of parallelism may require different amounts of synchronization and we prefer to exploit outer levels of parallelism so as to minimize synchronization.

2.3 Problem Statement

The algorithm described in this paper locates all the degrees of parallelism in a program; it finds the maximum degree of parallelism at each level of granularity, starting from the coarsest to the finest; it also finds opportunities of pipelined parallelism. The SPMD code we show assumes that there is a processor for each independent thread of computation found in the program.

To generate code for a specific number of processors, we can simply combine multiple of these parallel threads and assign them to the same processor. The processor assignment algorithm uses fine-grain parallelism only if there is insufficient coarser-grain parallelism to keep all the processors occupied. Additional considerations such as data locality are used to guide the assignment among threads belonging to the same level of granularity. To maximize the flexibility in processor assignment, it is therefore important for the parallelizer to locate all the degrees of parallelism at each level of granularity. For example, *blocking*, a processor assignment scheme demonstrated to improve locality, cannot be applied to loop nests with only one degree of parallelism.

3 Overview of the Algorithm

We decompose the overall problem of finding the maximum degree of parallelism into several subproblems: how to maximize the degree of parallelism that requires 0, $O(1)$, $O(n)$ amounts of synchronization, respectively, where n is the number of iterations in a loop. By solving each of these problems in turn, the algorithm finds successively more degrees of parallelism at higher synchronization costs. These steps are then repeated to find parallelism requiring $O(n^2), O(n^3), \ldots$ synchronization until sufficient parallelism is found to occupy all the available hardware.

The subproblem of maximizing synchronization-free parallelism is formulated as dividing the dynamic operations into the largest number of independent partitions. More specifically, our algorithm finds an affine partition mapping for each instruction that maximizes the degree of parallelism in each instruction. The affine mappings are subject to a set of *space-partition constraints*, which ensures that processors executing operations in different partitions need not synchronize with each other.

The next subproblem is to find parallelism with $O(1)$ synchronizations; that is, the number of synchronizations performed must not depend on the number of iterations in a loop. Our parallelization scheme may introduce as many synchronizations as there are instructions in a program, which is constant for a given program. Our algorithm divides instructions into a sequence of stages. We use the algorithm above to locate synchronization-free parallelism in each stage and introduce barrier synchronizations at the end of each parallelized stage.

Finally, to find parallelism with $O(n)$ synchronizations, we wish to find an affine time-partition mapping for each instruction. The affine mappings are subject to *time-partition constraints*, which ensure that data dependences can be satisfied by executing the partitions sequentially. The objective is to find affine mappings that yield maximum parallelism among operations within each of the time partitions.

The space-partition and time-partition constraints are similar in many ways and are amenable to the same kind of techniques. We use the affine form of the Farkas lemma[9] to transform the constraints into systems of linear inequalities. The problem of finding a partition mapping that gives the maximum degree of loop-level and pipelined parallelism while satisfying the space-partition or time-partition constraints reduces to finding the null space of a system of equations. The desired affine partition mapping can be found easily with a set of simple algorithms.

4 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.

4.1 Programs

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 $i > j$).

The domain of our algorithm is the set of sequential programs consisting of arbitrary nestings and sequences of loops, whose array accesses are affine functions of outer loop indices or loop-invariant variables. Other constructs such as conditional statements and non-affine array accesses can be handled by treating them conservatively.

Definition 4.1 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 .
- $\mathcal{D}_s(\vec{i}) = D_s(\vec{i}) + \vec{d}_s$, derived from the loop bounds, is an affine expression such that $\vec{i}$ is a valid loop index for instruction s iff $\mathcal{D}_s(\vec{i}) \geq \vec{0}$. Constant loop bounds are represented as additional symbolic variables.
- $\mathcal{F}_{zsr}(\vec{i}) = F_{zsr}(\vec{i}) + \vec{f}_{zsr}$ is an affine expression that maps an iteration $\vec{i}$ to the array index computed by the rth access function in instruction s to array z.
- 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'.

4.2 Data Dependences

The access patterns in a program define the constraints of program transformations. The notion of data dependences is well understood, and the definitions are included here for completeness. Informally, there is a *data dependence* from an access function $\mathcal{F}$ to another access function $\mathcal{F}'$ iff some instance of $\mathcal{F}$ uses a location which is subsequently used by $\mathcal{F}'$, and one of the accesses is a write operation. A *data dependence set* of a program contains all pairs of data dependent access functions in the program.

Definition 4.2 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,

$$\begin{aligned} &\forall s, s' \in \mathcal{S}, \vec{i} \in \mathcal{Z}^{\delta_s}, \vec{i}' \in \mathcal{Z}^{\delta_{s'}} \\ &s.t.\ \mathcal{D}_s(\vec{i}) \geq \vec{0} \wedge \mathcal{D}_{s'}(\vec{i}') \geq \vec{0}, \\ &\vec{i} \prec_{ss'} \vec{i}' \equiv \bigvee_{m=0}^{\eta_{ss'}} L^m_{ss'}(\vec{i}, \vec{i}') \end{aligned} \tag{1}$$

where

$$L^m_{ss'}(\vec{i}, \vec{i}') = \begin{cases} \vec{i}_{1:\eta_{ss'}} = \vec{i}'_{1:\eta_{ss'}} \wedge s <_p s' & m = \eta_{ss'} \\ \vec{i}_{1:m} = \vec{i}'_{1:m} \wedge \vec{i}_{m+1} < \vec{i}'_{m+1} & 0 \leq m < \eta_{ss'} \end{cases}$$

In the literature, dependences between pairs of instances satisfying constraint $L^m_{ss'}$ are known as being *carried* by loop m.

Definition 4.3 The *data dependence set* of a program $P = <\mathcal{S}, \delta, \mathcal{D}, \mathcal{F}, \omega, \eta>$ is

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

For better accuracy, we can substitute the symbolic variables representing constant loop bounds by their actual values.

4.3 Affine Partition Mappings

All the operations in a program are identified by their loop index values. Our algorithm uses affine expressions to map a vector of the original loop index values to either a processor identification number or an iteration number in a sequential loop. In the following, we define affine partition mappings formally and introduce a few properties used in the algorithms.

Definition 4.4 An m-dimensional *affine partition mapping* for instruction s in program P is an m-dimensional affine expression $\Phi_s(\vec{i}) = C_s\vec{i} + \vec{c}_s$, which maps an instance of instruction s, indexed by $\vec{i}$, to an m-element vector.[1] 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.[2]

Definition 4.5 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 4.6 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.

5 Synchronization-Free Parallelism

We first consider the problem of finding parallelism that requires no synchronization. This is formulated as dividing the operations of a program into partitions, such that dependent operations are placed in the same partition. By assigning each partition to a different processor, no synchronization is needed between processors. In this paper, we seek a partitioning that can be described by an affine mapping for each instruction.

In the following, we first define the necessary and sufficient constraints for an affine partition mapping to be synchronization-free. We next describe an algorithm that solves the constraints and finds a partition mapping that has the highest rank among all the possible solutions. We will show that the rank of the partition mapping corresponds to the degree of synchronization-free parallelism. Therefore, our algorithm finds the maximum degree of synchronization-free parallelism.

5.1 Space-Partition Constraints

[1] Symbolic constants are incorporated into our algorithm by treating them like loop variables.

[2] The coefficients of the affine partition mapping are rational numbers. Each dimension or *row* of a partition mapping for a program may be scaled to have integral values in the code generation phase.

Definition 5.1 (*Space-Partition Constraints*) 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[3]

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

Let $\mathcal{D}_s(\vec{i}) = D_s\vec{i} + \vec{d}_s$, $\mathcal{F}_{zsr}(\vec{i}) = F_{zsr}\vec{i} + \vec{f}_{zsr}$, and $\Phi_s(\vec{i}) = C_s\vec{i} + c_s$. Each data dependence, $< \mathcal{F}_{zsr}, \mathcal{F}_{zs'r'} > \in R$, imposes the following constraints on each *row* of a synchronization-free mapping for instructions s and s', $X = \left[\begin{array}{ccc} -C_s & C_{s'} & (c_{s'} - c_s) \end{array} \right]$:

$$\forall \vec{y} \text{ s.t. } G \left[\begin{array}{c} \vec{y} \\ 1 \end{array} \right] \geq \vec{0} \ \wedge \ H \left[\begin{array}{c} \vec{y} \\ 1 \end{array} \right] = \vec{0}, \ X \left[\begin{array}{c} \vec{y} \\ 1 \end{array} \right] = 0 \tag{3}$$

where

$$\begin{aligned} \vec{y} &\equiv \left[\begin{array}{c} \vec{i} \\ \vec{i'} \end{array} \right] \\ G &= \left[\begin{array}{ccc} D_s & 0 & \vec{d}_s \\ 0 & D_{s'} & \vec{d}_{s'} \end{array} \right] \\ H &= \left[\begin{array}{ccc} F_{zsr} & -F_{zs'r'} & (\vec{f}_{zsr} - \vec{f}_{zs'r'}) \end{array} \right] \end{aligned}$$

5.2 Linearizing the Constraints

We first convert the space-partition constraints into a set of linear equations using the following algorithm:

Algorithm 5.1 Convert constraints in the form of (3) into a system of equations.

Step 1: Simplify the constraints by successively removing variables in y using the equations $H \left[\begin{array}{c} \vec{y} \\ 1 \end{array} \right] = \vec{0}$ in (3).[4] We use a variant of the Gaussian Elimination algorithm. We first select an equation and express one of its variable y_i, an element of $\vec{y}$, in terms of other variables in the equation. Then, we substitute all occurrences of y_i with the expression. The result of this step is of the form:

$$\forall \vec{y'} \ \text{ s.t. } G' \left[\begin{array}{c} \vec{y'} \\ 1 \end{array} \right] \geq \vec{0}, \quad XE \left[\begin{array}{c} \vec{y'} \\ 1 \end{array} \right] = 0 \tag{4}$$

where vector $\vec{y'}$ has less variables than $\vec{y}$.

Step 2: Reduce the system in (4) to a set of linear equations, using the Affine Form of the Farkas Lemma[9].

Lemma 5.1 *(Affine Form of the Farkas Lemma)*

Let $x(\vec{y})$ be an affine expression of $\vec{y}$ and $G \left[\begin{array}{c} \vec{y} \\ 1 \end{array} \right] \geq \vec{0}$ define a non-empty polyhedron.

$$\begin{aligned} &\forall \ \vec{y} \ \ s.t. \ G \left[\begin{array}{c} \vec{y} \\ 1 \end{array} \right] \geq \vec{0}, x(\vec{y}) \geq 0 \\ \textit{iff} \quad & x(\vec{y}) \equiv \left[\begin{array}{cc} \lambda_1 \cdots \lambda_m & \lambda_0 \end{array} \right] \left[\begin{array}{cc} & G \\ 0 \cdots 0 & 1 \end{array} \right] \left[\begin{array}{c} \vec{y} \\ 1 \end{array} \right] \\ \textit{with} \quad & \left[\begin{array}{cc} \lambda_1 \cdots \lambda_m & \lambda_0 \end{array} \right]^T \geq \vec{0} \end{aligned}$$

where m is the number of rows in G. The positive constants λ_k, whose existence is asserted by Farkas Lemma, are called Farkas multipliers.

We rewrite the constraint in (4) as

$$(\forall \ \vec{y'} \text{ s.t. } G' \left[\begin{array}{c} \vec{y'} \\ 1 \end{array} \right] \geq \vec{0}, \ \ XE \left[\begin{array}{c} \vec{y'} \\ 1 \end{array} \right] \geq 0) \tag{5}$$

$$\wedge \ (\forall \ \vec{y'} \text{ s.t. } G' \left[\begin{array}{c} \vec{y'} \\ 1 \end{array} \right] \geq \vec{0}, \ -XE \left[\begin{array}{c} \vec{y'} \\ 1 \end{array} \right] \geq 0) \tag{6}$$

Applying the Farkas Lemma to (5),

$$\begin{aligned} &XE \left[\begin{array}{c} \vec{y'} \\ 1 \end{array} \right] = \left[\begin{array}{cc} \lambda_1 \cdots \lambda_m & \lambda_0 \end{array} \right] \left[\begin{array}{cc} & G' \\ 0 \cdots 0 & 1 \end{array} \right] \left[\begin{array}{c} \vec{y'} \\ 1 \end{array} \right] \\ &\left[\begin{array}{cc} \lambda_1 \cdots \lambda_m & \lambda_0 \end{array} \right]^T \geq \vec{0} \end{aligned} \tag{7}$$

where m is the number of rows in G.

Since (7) holds for all $\vec{y'}$,

$$\begin{aligned} &XE = \left[\begin{array}{cc} \lambda_1 \cdots \lambda_m & \lambda_0 \end{array} \right] \left[\begin{array}{cc} & G' \\ 0 \cdots 0 & 1 \end{array} \right] \\ &\left[\begin{array}{cc} \lambda_1 \cdots \lambda_m & \lambda_0 \end{array} \right]^T \geq \vec{0} \end{aligned} \tag{8}$$

The unknowns in the new constraints (8) are X, $\lambda_0, \cdots, \lambda_m$. We are interested only in X, so we apply Fourier-Motzkin elimination[20] to eliminate all the Farkas multipliers to obtain a set of constraints in the form of

$$XA \geq \vec{0} \tag{9}$$

Renaming $-X$ in (6) as X' and applying the Farkas Lemma and Fourier Motzkin elimination as above, we get

$$X'A \geq \vec{0} \equiv XA \leq \vec{0} \tag{10}$$

Combining (10) with (9), we get a set of linear constraints on X:

$$XA = \vec{0} \text{ or } A^T X^T = \vec{0} \tag{11}$$

5.3 Solving Space-Partition Constraints

Now that the constraints are expressed as equations, we can use techniques from linear algebra to find the desirable partition mappings.

Algorithm 5.2 Find a highest-ranked synchronization-free affine partition mapping for program $P =< \mathcal{S}, \delta, \mathcal{D}, \mathcal{F}, \omega, \eta >$.

Step 1: Construct from each data dependence in R the

[3] For synchronization-free parallelism, it is not necessary to distinguish between the direction of the data dependence in R. The redundancy is included in the constraints to emphasize the similarity with the definition in Section 7. It is a matter of simple optimization to eliminate the redundant constraints.

[4] Pairs of inequalities in $G \left[\begin{array}{c} \vec{y} \\ 1 \end{array} \right] \geq \vec{0}$ can sometimes be reduced to an equation, which can then be eliminated.

space-partition constraints on a one-dimensional partition mapping in the form of (3).

Step 2: Apply Algorithm 5.1 to rewrite all the constraints as a system of linear equations in the form of $A\vec{x} = \vec{0}$, where $\vec{x}$ is a vector of variables representing all the unknown coefficients (C) and constant terms (c) of the affine partition mapping.

Step 3: Since the rank of an affine partition mapping is not dependent on the values of its constant terms c, we eliminate all the unknowns c from $A\vec{x} = \vec{0}$ using the same elimination technique in Step 1 of Algorithm 5.1. Let the simplified system be $A'\vec{x}' = \vec{0}$ where $\vec{x}'$ represents the unknown coefficients C.

Step 4: Find the solutions to $A'\vec{x}' = \vec{0}$, expressed as $\mathcal{B}$, a set of basis vectors spanning the null space of A'.

Step 5: Derive one row of the desired affine partition mapping from each basis vector in $\mathcal{B}$. The coefficients C are specified directly by the basis vector, and the constant terms c are derived from the coefficients C using $A\vec{x} = \vec{0}$.

Note that the basis vectors in $\mathcal{B}$, representing the coefficients of the affine mapping for all instructions, are linearly independent. The rows in the affine mappings for each instruction, however, are not necessarily linearly independent. A mapping may have more rows than its rank and the dimensionality of the processor space may be larger than the degree of parallelism. Thus, while this algorithm finds all the parallelism, a further step must be taken to map only the partitions with computations to physical processors.

5.4 Parallelism With No Synchronization

Algorithm 5.3 Find all the degrees of synchronization-free parallelism for program P.

Step 1: Apply Algorithm 5.2 to P to get a highest-ranked synchronization-free affine partition mapping Φ.

Step 2: Generate SPMD code such that each processor executes a partition of Φ, and all instructions within a partition are executed in their sequential order in P.

Theorem 5.2 *Algorithm 5.3 finds the maximum degree of synchronization-free parallelism.*

Proof: A synchronization-free partition mapping with rank r creates $O(n^r)$ independent partitions which, when mapped to different processors, produce r degrees of synchronization-free parallelism. Thus, a synchronization-free affine partition mapping with the highest rank, as found in Algorithm 5.3, has the highest degree of parallelism. The parallel program is correct because the computations on different processors are independent of each other, and operations executed on each processor follow the original sequential order. □

It is straightforward to generate the SPMD code with the maximum degree of synchronization-free parallelism found by Algorithm 5.3. Our code generation algorithm is based on Ancourt and Irigoin's polyhedron-scanning code generation technique[2]; details and examples can be found in [18].

To ensure that Algorithm 5.2 can be used as a step in finding the maximum degrees of parallelism with and without synchronizations, we have the following lemma:

Lemma 5.3 *The two-step approach of (1) finding synchronization-free partitions and (2) finding the maximum degree of parallelism in each of the partitions maximizes the degree of parallelism in a program.*

Proof: Mapping operations from different synchronization-free partitions to the same processor can only decrease the degree of parallelism in a program. □

5.5 Example

To illustrate our algorithm, we introduce a slightly more complicated example below.

Example 5:

```
for l1 = 1 to 100 do
  for l2 = 1 to 100 do
    a[l1,l2] = a[l1,l2] + b[l1 - 1,l2];    [1]
    b[l1,l2] = a[l1,l2 - 1] * b[l1,l2];    [2]
```

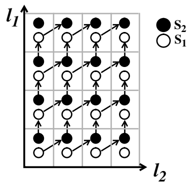

Figure 2: A subset of dynamic operations in Example 5.

In Figure 2, we show four iterations from each of the loops. Each iteration of the innermost loop is represented by a pair of nodes, with the white node representing an operation of instruction 1 and the black representing an operation of instruction 2. The arrows represent dependences between the operations. It is easy to see from the figure that the best way to parallelize this code is to assign each chain of alternating black and white nodes to a processor. We show step-by-step how Algorithm 5.2 systematically derives the affine partition mappings that describe this optimal parallelization scheme.

Step 1: Construct the space-partition constraints. The iteration space for either of the instructions in the loop body is:

$$\left\{ \begin{bmatrix} l_1 \\ l_2 \end{bmatrix} \;\middle|\; \mathcal{D}(\begin{bmatrix} l_1 \\ l_2 \end{bmatrix}) = D \begin{bmatrix} l_1 \\ l_2 \end{bmatrix} + \vec{d} > \vec{0} \right\}$$

where

$$D = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix}, \; \vec{d} = \begin{bmatrix} -1 \\ 100 \\ -1 \\ 100 \end{bmatrix}$$

The data dependence set R contains two pairs of access functions: $< \mathcal{F}_{b11}, \mathcal{F}_{b21} >$ and $< \mathcal{F}_{a11}, \mathcal{F}_{a21} >$ where

$$\mathcal{F}_{b11}(\begin{bmatrix} l_1 \\ l_2 \end{bmatrix}) = F_{b11} \begin{bmatrix} l_1 \\ l_2 \end{bmatrix} + \vec{f}_{b11} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} l_1 \\ l_2 \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

$$\mathcal{F}_{b21}(\begin{bmatrix} l_1' \\ l_2' \end{bmatrix}) = F_{b21}\begin{bmatrix} l_1' \\ l_2' \end{bmatrix} + \vec{f}_{b21} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} l_1' \\ l_2' \end{bmatrix}$$

$$\mathcal{F}_{a11}(\begin{bmatrix} l_1 \\ l_2 \end{bmatrix}) = F_{a11}\begin{bmatrix} l_1 \\ l_2 \end{bmatrix} + \vec{f}_{a11} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} l_1 \\ l_2 \end{bmatrix}$$

$$\mathcal{F}_{a21}(\begin{bmatrix} l_1' \\ l_2' \end{bmatrix}) = F_{a21}\begin{bmatrix} l_1' \\ l_2' \end{bmatrix} + \vec{f}_{a21} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} l_1' \\ l_2' \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

Let $\vec{y} \equiv \begin{bmatrix} l_1 & l_2 & l_1' & l_2' \end{bmatrix}^T$. We derive from the data dependent pair $< \mathcal{F}_{b11}, \mathcal{F}_{b21} >$ the constraint

$$\begin{aligned} \forall\ \vec{y}\ \text{ s.t. } & \begin{bmatrix} D & 0 & \vec{d} \\ 0 & D & \vec{d} \end{bmatrix}\begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} \geq \vec{0} \\ \wedge & \begin{bmatrix} F_{b11} & -F_{b21} & \vec{f}_{b11} - \vec{f}_{b21} \end{bmatrix}\begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} = \vec{0}, \\ & X_1 \begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} = 0 \end{aligned} \tag{12}$$

where $X_1 = \begin{bmatrix} -C_1 & C_2 & c_2 - c_1 \end{bmatrix}$.

Similarly, we derive from $< \mathcal{F}_{a11}, \mathcal{F}_{a21} >$ the constraint

$$\begin{aligned} \forall\ \vec{y}\ \text{ s.t. } & \begin{bmatrix} D & 0 & \vec{d} \\ 0 & D & \vec{d} \end{bmatrix}\begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} \geq \vec{0} \\ \wedge & \begin{bmatrix} F_{a11} & -F_{a21} & \vec{f}_{a11} - \vec{f}_{a21} \end{bmatrix}\begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} = \vec{0}, \\ & X_2 \begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} = 0 \end{aligned} \tag{13}$$

where $X_2 = \begin{bmatrix} -C_1 & C_2 & c_2 - c_1 \end{bmatrix}$.

Step 2: Apply Algorithm 5.1 to rewrite constraints (12) and (13) as a system of linear equations. For constraints (12), eliminate variables from X_1 using

$$\begin{bmatrix} F_{b11} & -F_{b21} & \vec{f}_{b11} - \vec{f}_{b21} \end{bmatrix}\begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} = \vec{0}$$

$$\equiv \begin{bmatrix} 1 & 0 & -1 & 0 & -1 \\ 0 & 1 & 0 & -1 & 0 \end{bmatrix}\begin{bmatrix} l_1 \\ l_2 \\ l_1' \\ l_2' \\ 1 \end{bmatrix} = \vec{0}$$

The algorithm substitutes l_1 with $l_1' + 1$, then l_2 with l_2'. The simplified constraint is

$$\forall\ \vec{y}'\ \text{ s.t. } G'\begin{bmatrix} \vec{y}' \\ 1 \end{bmatrix} \geq \vec{0},\ X_1 E \begin{bmatrix} \vec{y}' \\ 1 \end{bmatrix} = 0 \tag{14}$$

where

$$\vec{y}' \equiv \begin{bmatrix} l_1' \\ l_2' \end{bmatrix},\ G' = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 99 \\ 0 & 1 & -1 \\ 0 & -1 & 100 \\ 1 & 0 & -1 \\ -1 & 0 & 100 \\ 0 & 1 & -1 \\ 0 & -1 & 100 \end{bmatrix},\ E = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Apply Farkas Lemma to (14) and eliminate the Farkas multipliers λ's and μ's using Fourier-Motzkin elimination.[5] Farkas Lemma rewrites (14) as

$$\begin{aligned} X_1 E &= \begin{bmatrix} \lambda_1 \cdots \lambda_8 & \lambda_0 \end{bmatrix} G' \ \wedge\ \lambda_0, \ldots, \lambda_8 \geq 0 \\ -X_1 E &= \begin{bmatrix} \mu_1 \cdots \mu_8 & \mu_0 \end{bmatrix} G' \ \wedge\ \mu_0, \ldots, \mu_8 \geq 0 \end{aligned}$$

The final system of equations with the Farkas multipliers eliminated is

$$X_1 \begin{bmatrix} 100 & 100 & 2 & 2 & 2 & 2 \\ 1 & 100 & 1 & 2 & 99 & 100 \\ 99 & 99 & 1 & 1 & 1 & 1 \\ 1 & 100 & 1 & 2 & 99 & 100 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \vec{0} \tag{15}$$

For constraints (13), applying similar steps above yields

$$X_2 \begin{bmatrix} 1 & 100 & 1 & 2 & 99 & 100 \\ 1 & 1 & 99 & 99 & 99 & 99 \\ 1 & 100 & 1 & 2 & 99 & 100 \\ 2 & 2 & 100 & 100 & 100 & 100 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \vec{0} \tag{16}$$

Step 3: Eliminate the constant terms c from the space-partition constraints (15) and (16). Using the first column in (15), we substitute $c_2 - c_1$ with

$$\begin{bmatrix} -100 & -1 & -99 & -1 \end{bmatrix}\begin{bmatrix} -C_1 & C_2 \end{bmatrix}^T$$

The resulting constraints on the coefficients C are

$$\begin{bmatrix} 0 & 99 & 0 & 99 \\ -98 & 0 & -98 & 0 \\ -98 & 1 & -98 & 1 \\ -98 & 98 & -98 & 98 \\ -98 & 99 & -98 & 99 \\ -99 & 0 & -98 & 1 \\ 0 & 0 & 1 & 1 \\ -99 & 98 & -98 & 99 \\ -98 & 98 & -97 & 99 \\ -1 & 98 & 0 & 99 \\ 0 & 98 & 1 & 99 \end{bmatrix}\begin{bmatrix} -C_1 & C_2 \end{bmatrix}^T = \vec{0} \tag{17}$$

Step 4: Find the solution space to (17). The solution space is the null space of the matrix, which is spanned by the vector $\vec{v} = \begin{bmatrix} -1 & 1 & 1 & -1 \end{bmatrix}$.

Step 5: Construct a synchronization-free partition mapping from $\vec{v}$. The coefficients C are given by $\vec{v}$ and the corresponding $c_2 - c_1$ is 1. Setting c_2 to 0, we get $c_1 = -1$ and the space-partition mapping Φ with $\Phi_1 = l_1 - l_2 - 1$ and $\Phi_2 = l_1' - l_2'$.

The SPMD code generated from the space-partition mapping Φ is as follows: (p denotes the processor's id)

```
if (p = -99) then
   a[1, 100] = a[1, 100] + b[0, 100];
if (-98 ≤ p ≤ 99) then
   if (1 ≤ p ≤ 99) then
      b[p, 1] = a[p, 0] * b[p, 1];
```

[5] Each equation is normalized using the GCD of its coefficients.

$$
\begin{aligned}
&\text{for } l_1 = \max(1, 1+p) \text{ to } \min(100, 99+p)\\
&\quad a[l_1, l_1 - p] = a[l_1, l_1 - p] + b[l_1 - 1, l_1 - p];\\
&\quad b[l_1, l_1 - p + 1] = a[l_1, l_1 - p] * b[l_1, l_1 - p + 1];\\
&\text{if } (-98 \leq p \leq 0) \text{ then}\\
&\quad a[100 + p, 100] = a[100 + p, 100] + b[99 + p, 100];\\
&\text{if } (p = 100) \text{ then}\\
&\quad b[100, 1] = a[100, 0] * b[100, 1];
\end{aligned}
$$

6 Parallelism With $O(1)$ Synchronizations

To find parallelism that requires only a constant amount of synchronization, we use a *program dependence graph*, where all dynamic instances of the same instruction are represented by a single node.

Definition 6.1 Let R be the data dependence set for program $P =< \mathcal{S}, \delta, \mathcal{D}, \mathcal{F}, \omega, \eta >$. The 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. $< \mathcal{F}_{zsr}, \mathcal{F}'_{zs'r'} > \in R$.

Lemma 6.1 *All nodes in a strongly connected component (SCC) of a program dependence graph must share at least one common surrounding loop nest.*

Proof: If two instructions do not share a common surrounding loop nest, all instances of one instruction must be executed before any of the other instruction in the original program. Dependences between the two instructions must be unidirectional, and the instructions cannot belong to the same SCC of a program dependence graph. □

Algorithm 6.1 Find all the degrees of parallelism requiring $O(1)$ synchronizations for a program P.

Step 1: Construct a program dependence graph for P and partition the instructions into SCCs.

Step 2: Apply Algorithm 5.3 to each SCC to find all its synchronization-free parallelism.

Step 3: Generate code to execute the SCC in a topological order, and introduce a barrier at the end of each parallelized SCC.

Lemma 6.2 *The two-step approach of (1) partitioning a program into SCCs and (2) finding the maximum degree of parallelism in each SCC maximizes the degree of parallelism in a program.*

Proof: Executing SCCs in parallel can only increase parallelism by a constant factor and does not affect the degree of parallelism. As dependences within individual SCCs are only a subset of those in the entire program, the SCCs cannot have less degrees of parallelism than the entire program. □

Theorem 6.3 *Algorithm 6.1 finds the maximum degree of coarse-grain parallelism that uses at most $O(1)$ synchronizations.*

Proof: First, executing SCCs in topological order clearly honors all the dependences between operations in different SCCs. Second, the algorithm introduces, for a given program, only a constant number of synchronizations, which is bounded by the number of instructions in a program and not dependent on the number of iterations in a loop. Third, by Lemma 6.2, considering strongly connected components independently does not reduce the degree of parallelism in a program. Any parallelization scheme that assigns data dependent operations in an SCC to different processors requires at least $O(n)$ synchronizations, where n is the number of iterations in a loop. Thus, finding the maximum degree of synchronization-free parallelism in each SCC must maximize the parallelism that requires no more than $O(1)$ synchronizations. □

7 Parallelism Within Sequential Loops

We now consider those programs that have no synchronization-free parallelism and whose program dependence graph consists of only one strongly connected component. By Lemma 6.1, there must exist an outermost loop that surrounds all the instructions in such a program; by Theorem 6.3, any available parallelism must require at least $O(n)$ synchronizations, where n is the number of iterations in a loop. We can express the parallelized version of the computation as an outermost loop, where multiple processors cooperate to execute an iteration in parallel, and synchronize with each other at the end of each iteration to ensure that the iterations are executed in sequence. We refer to such loops as *outer sequential* loops.

Our goal is to partition the operations so that operations within each iteration of the outer sequential loop admit of the largest degree of parallelism. More specifically, we wish to find an affine mapping from the original loop index values of each operation to the iteration number of the new loop, so that it is legal to execute the loop sequentially and operations within each iteration have the largest degree of parallelism.

In the following, we first define a *legal* affine partition mapping such that executing the partitions sequentially does not violate any data dependence constraints. Next, we calculate the maximally independent solutions to the constraints. We show that a linear combination of these solutions yields an affine mapping that exposes all the degrees of parallelism with only one degree of synchronization. Furthermore, by making the maximally independent solutions rows of an affine mapping, we exploit all the possible degrees of pipeline parallelism without reducing the overall degree of parallelism.

7.1 Time-Partition Constraints

We wish to partition the operations into iterations such that executing the iterations sequentially does not violate any data dependences. In other words, if operation u depends on operation v, u must either execute in the same iteration as v, or u is executed in an iteration that comes after the iteration that executes v. Thus, the time-partition constraints are simply a more relaxed version of the space-partition constraints.

Definition 7.1 (*Time-Partition Constraints*) Let R be the data dependence set for program $P =< \mathcal{S}, \delta, \mathcal{D}, \mathcal{F}, \omega, \eta >$. A one-dimensional affine partition mapping Φ for P is *legal* iff

$$
\begin{gathered}
\forall < \mathcal{F}_{zsr}, \mathcal{F}_{zs'r'} > \in R, \vec{i} \in \mathcal{Z}^{\delta_s}, \vec{i}' \in \mathcal{Z}^{\delta_{s'}} \text{ s.t.}\\
\vec{i} \prec_{ss'} \vec{i}' \ \wedge \ \mathcal{D}_s(\vec{i}) \geq \vec{0} \wedge \mathcal{D}_{s'}(\vec{i}') \geq \vec{0}
\end{gathered}
$$

$$\wedge\ \mathcal{F}_{zsr}(\vec{i}) - \mathcal{F}_{zs'r'}(\vec{i}') = \vec{0},$$
$$\Phi_{s'}(\vec{i}') - \Phi_s(\vec{i}) \geq 0 \quad (18)$$

Using Definition 4.2, we can rewrite (18) as a set of linear time-partition constraints:

$$\begin{aligned}
&\forall < \mathcal{F}_{zsr}, \mathcal{F}_{zs'r'} >\in R, \vec{i} \in \mathcal{Z}^{\delta_s}, \vec{i}' \in \mathcal{Z}^{\delta_{s'}} \\
&\forall\ m \in \{0, \ldots, \eta_{ss'}\} \text{ s.t.} \\
&\quad L^m_{ss'}(\vec{i}, \vec{i}')\ \wedge\ \mathcal{D}_s(\vec{i}) \geq \vec{0} \wedge \mathcal{D}_{s'}(\vec{i}') \geq \vec{0} \\
&\quad \wedge\ \mathcal{F}_{zsr}(\vec{i}) - \mathcal{F}_{zs'r'}(\vec{i}') = \vec{0}, \\
&\qquad \Phi_{s'}(\vec{i}') - \Phi_s(\vec{i}) \geq 0 \qquad (19)
\end{aligned}$$

There exists at least one legal, affine partition mapping for an outer sequential loop; the time-partition constraints can be trivially satisfied by making the iteration number of the outermost loop the partition number. Furthermore, it is clearly possible to order the operations in each partition such that executing the partitions sequentially honors all the data dependences in the loop.

7.2 Solving Time-Partition Constraints

We use the concept of maximally independent partition mappings to describe the space of all the solutions to the time-partition constraints and show below how we find all the possible legal partitions.

Definition 7.2 A set of legal one-dimensional affine partition mappings $\mathcal{B} = \{\Phi^1, \Phi^2, \ldots, \Phi^n\}$ for program $P =< \mathcal{S}, \delta, \mathcal{D}, \mathcal{F}, \omega, \eta >$ is said to be *maximally independent* iff the set is minimal, and any legal one-dimensional affine partition mapping Φ can be expressed as an affine combination of the mappings in $\mathcal{B}$. That is, for all legal Φ

$$\begin{aligned}
&\exists k_1, \ldots, k_n \in Q \text{ s.t.} \\
&\forall s \in \mathcal{S}\ \exists a \in Q \text{ s.t.} \\
&\quad \Phi_s(\vec{y}) = k_1\Phi^1_s(\vec{y}) + k_2\Phi^2_s(\vec{y}) + \cdots + k_n\Phi^n_s(\vec{y}) + a
\end{aligned}$$

Algorithm 7.1 Find a set of legal, maximally independent, affine partition mappings for an outer sequential loop.

Step 1: Using the same steps in Algorithm 5.1, we rewrite each system of constraints from (19) as $A\vec{x} \geq \vec{0}$. Whether two affine partition mappings are linearly independent depends only on the coefficients C. Thus, we eliminate the constant terms c from $A\vec{x} \geq \vec{0}$ using Fourier-Motzkin elimination. Let the simplified system be $A_1\vec{x}_1 \geq \vec{0}$ where $\vec{x}_1$ represents the coefficients C.

Step 2: Find a maximal set of linearly independent solutions to $A_1\vec{x}_1 \geq \vec{0}$ using Algorithm A. In brief, the algorithm introduces a set of new variables $\vec{b}$, one for each row of A_1, such that

$$A_1\vec{x}_1 \geq \vec{0}\ \equiv\ \begin{bmatrix} A_1 & -I_{m\times m} \end{bmatrix} \begin{bmatrix} \vec{x}_1 \\ \vec{b} \end{bmatrix} = \vec{0},\ \vec{b} \geq \vec{0} \quad (20)$$

where the size of matrix A_1 is $m \times n$. It is obvious that any solution to (20) must also be a solution to

$$\begin{bmatrix} A_1 & -I_{m\times m} \end{bmatrix} \begin{bmatrix} \vec{x}_1 \\ \vec{b} \end{bmatrix} = \vec{0} \quad (21)$$

Conversely, solutions to (21) are solutions to (20) if the values for $\vec{b}$ are non-negative. One basis for the solutions to (21) consists of the rows of matrix $V = \begin{bmatrix} I_{n\times n} & A_1^T \end{bmatrix}$. Algorithm A applies elementary row operations[6] to V to maximize the number of rows that have non-negative values for $\vec{b}$. Furthermore, the set of rows with non-negative $\vec{b}$ is guaranteed to be linearly independent.

Step 3: From each solution of $\vec{x}_1$, found in Step 2, derive one affine partition mapping. The coefficients C are given directly by $\vec{x}_1$, and the constant terms c are derived using $A\vec{x} \geq \vec{0}$.

7.3 Parallelism With $O(n)$ Synchronizations

Our objective is to create affine partitions, each of which contains as much parallelism as possible. The parallelism available in a partition is determined by the data dependences between operations within a partition. By placing a pair of data dependent operations in different time partitions, a partition mapping *dismisses* the dependence between the operations from the consideration of parallelism within a partition. Thus, it is generally desirable that the affine time-partition mappings dismiss as many dependences as possible so as to maximize the parallelism within a partition. For the domain of affine mappings and the objective of maximizing the degree of parallelism, it is sufficient to seek affine mappings that dismiss the maximal set of linear time-partition constraints. In the following, we define the notion of dismissing a constraint more formally, propose an algorithm and show that the algorithm is optimal.

Definition 7.3 A legal affine partition mapping *dismisses* the linear time-partition constraint imposed by $< \mathcal{F}_{zsr}, \mathcal{F}_{zs'r'} >\in R$ carried at level m iff

$$\begin{aligned}
&\forall\ \vec{i} \in \mathcal{Z}^{\delta_s}, \vec{i}' \in \mathcal{Z}^{\delta_{s'}} \text{ s.t.} \\
&\quad L^m_{ss'}(\vec{i}, \vec{i}')\ \wedge\ \mathcal{D}_s(\vec{i}) \geq \vec{0} \wedge \mathcal{D}_{s'}(\vec{i}') \geq \vec{0} \\
&\quad \wedge\ \mathcal{F}_{zsr}(\vec{i}) - \mathcal{F}_{zs'r'}(\vec{i}') = \vec{0}, \\
&\qquad \Phi_{s'}(\vec{i}') - \Phi_s(\vec{i}) > 0 \qquad (22)
\end{aligned}$$

Algorithm 7.2 Find all the degrees of parallelism requiring $O(n)$ synchronizations for an outer sequential loop P.

Step 1: Apply Algorithm 7.1 to program P to get a set of maximally independent partition mappings $\mathcal{B}$. Partition the operations in time using $\Phi = \sum_{\Phi^i \in \mathcal{B}} \Phi^i$. Generate the code to execute the time partitions in sequential order, and introduce a barrier at the end of each partition.

Step 2: Apply Algorithm 6.1 to each time partition of Φ to find all the degrees of parallelism requiring at most $O(1)$ synchronizations.

To show that Algorithm 7.2 finds all the parallelism requiring one degree of synchronization, we introduce the following lemmas.

Lemma 7.1 *Any positive linear combination of legal affine partition mappings is also a legal affine partition mapping.*

[6] Elementary row operations include interchanging two rows, scaling a row, and adding a scalar multiple of one row to another.

Proof: Combinations of affine partition mappings that satisfy the time-partition constraints (18) must also satisfy the same constraints. □

Lemma 7.2 *There exists a legal one-dimensional affine time-partition mapping that dismisses all the linear time-partition constraints that can be dismissed by any legal one-dimensional mapping.*

Proof: Given any two legal one-dimensional affine time-partition mappings, Φ and Φ', which dismiss different linear time-partition constraints σ and σ', respectively. We can always construct a legal mapping $\hat{\Phi} = \Phi + \Phi'$ that dismisses $\sigma \cup \sigma'$. □

Lemma 7.3 *If two legal, one-dimensional affine partition mappings for an outer sequential loop are linearly dependent, they yield the same degree of parallelism at every level of granularity.*

Proof: Let Φ and Φ' be two legal, one-dimensional and linearly dependent affine partition mappings for an outer sequential loop P with m instructions. We show below that Φ yields at least the same degree of parallelism as Φ', and by showing the converse with a symmetric argument, we prove that Φ and Φ' must have the same degree of parallelism.

Each partition in Φ consists of instances of different instructions, $G = \{g_1, \ldots, g_m\}$, where g_i is a set of instances of instruction s_i. As Φ and Φ' are linearly dependent, for each set g_i, there exists a partition in Φ' that contains all the operations in the set. Thus, Φ' maps operations in G to at most m different partitions.

We subdivide operations in G into subpartitions according to their partition numbers in Φ'. Since Φ' is a legal mapping, it is legal to execute the subpartitions sequentially in order of their partition numbers. We can exploit all the parallelism in each subpartition by introducing at most $m-1$ barriers between each subpartition. Since an outer sequential loop needs at least $O(n)$ synchronizations to exploit any degree of parallelism, Φ yields at least the same degree of parallelism, at every level of granularity. □

Lemma 7.4 *Let $\mathcal{B} = \{\Phi^1, \Phi^2, \ldots, \Phi^n\}$ be a set of maximally independent legal affine partition mappings for an outer sequential loop P. The one-dimensional affine mapping $\Phi = \sum_{i=1}^{n} \Phi^i$ is a legal partition mapping for P, and the partitions Φ create have the largest degree of parallelism.*

Proof: Let Φ' be a legal one-dimensional affine partition mapping that dismisses the maximal set of linear time-partition constraints (Lemma 7.2). Φ' must therefore yield the maximum degree of parallelism among all one-dimensional legal mappings. Given a set of legal, maximally independent, partition mappings $\mathcal{B} = \{\Phi^1, \Phi^2, \ldots, \Phi^n\}$,

$$\begin{array}{l} \exists k_1, \ldots, k_n \geq 0 \text{ s.t.} \\ \quad \forall s \in \mathcal{S}\ \exists a \in Q \text{ s.t.} \\ \quad \Phi'_s(\vec{y}) = k_1 \Phi^1_s(\vec{y}) + k_2 \Phi^2_s(\vec{y}) + \cdots + k_n \Phi^n_s(\vec{y}) + a \end{array}$$

Let $m = \max_{i=1}^{n}(k_i)$.

$$\begin{aligned} \hat{\Phi}_s(\vec{y}) &= \Phi'_s(\vec{y}) + \sum_{i=1}^{n}((m-k_i)\Phi^i_s(\vec{y})) \\ &\equiv m \sum_{i=1}^{n} \Phi^i_s(\vec{y}) + a_s \ \equiv\ m\Phi_s(\vec{y}) + a_s \end{aligned}$$

Since $(m - k_i)$ is non-negative, $((m-k_i)\Phi^i)$ is a legal partition mapping (Lemma 7.1). As $\hat{\Phi}$ is a combination of Φ' and other legal mappings, $\hat{\Phi}$ must be legal and dismisses the same maximal set of linear constraints. By Lemma 7.3, since Φ and $\hat{\Phi}$ are linearly dependent, they have the same maximal degree of parallelism. □

Theorem 7.5 *Algorithm 7.2 finds the maximum degree of coarse-grain parallelism with at most one degree of synchronization in an outer sequential loop P.*

Proof: First, it is obvious from Definition 7.1 that executing the time partitions sequentially honor all the dependences between operations in different partitions. Second, both Steps 1 and 2 of the algorithm introduce at most $O(n)$ synchronizations. By Lemma 7.4, Step 1 of Algorithm 7.2 finds a legal mapping that creates partitions with the largest degree of parallelism in P. By Theorem 6.3, Step 2 finds all the parallelism that uses at most $O(1)$ synchronizations in each partition. Therefore, Algorithm 7.2 finds the maximum degree of parallelism that requires at most one degree of synchronization. □

7.4 Exploiting Pipeline Parallelism

When there are several linearly independent solutions to the time-partition constraints, the partition mapping chosen in Algorithm 7.2 is but one of the many equivalent ways of exposing the maximum degree of parallelism in a program. Our preferred approach is, in fact, not to use the mapping in Algorithm 7.2, but to pipeline the computation instead. We show below that pipelining can achieve the same degrees of parallelism, with the advantage of replacing barriers with point-to-point synchronizations, more regular code, better data locality, and being amenable to blocking, which can further increase the granularity of parallelism by a constant factor.

Definition 7.4 A multi-dimensional affine partition mapping Φ for a program P is *pipelinable* iff each row of the mapping is a legal mapping for P. That is, let R be the data dependence set for P

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

Algorithm 7.3 Find all the degrees of parallelism requiring $O(n)$ synchronizations for an outer sequential loop P, and exploit pipeline parallelism when possible.

Step 1: Apply Algorithm 7.1 to program P to get a set of maximally independent partition mappings $\mathcal{B} = \{\Phi^1, \ldots, \Phi^m\}$. Let Φ be an m-dimensional pipelinable partition mapping whose rows are the mappings in $\mathcal{B}$.

Step 2: Let Φ' be the first $m-1$ rows of Φ, and Φ^m be the last row. Map the operations in P to $O(n^{m-1})$ processors using Φ' where n is the number of iterations in a loop, and partition the computation assigned to each processor in time using Φ^m. Each processor executes its time partitions in sequence, and operations in each time partition are executed in their original sequential order. Each processor $(p_1, \ldots, p_{m-1})$ synchronizes with its $m-1$ neighbors $(p_1-1, \ldots, p_{m-1})$, ..., $(p_1, \ldots, p_{m-1}-1)$ before each time partition, and its $m-1$ neighbors $(p_1+1, \ldots, p_{m-1})$, ..., $(p_1, \ldots, p_{m-1}+1)$ after each time partition.

Step 3: Apply Algorithm 6.1 to each partition of Φ to find all the degrees of parallelism requiring at most $O(1)$ synchronizations.

Lemma 7.6 *Algorithm 7.3 yields $r-1$ degrees of pipeline parallelism where r is the rank of an affine partition whose rows are the legal, maximally independent affine mappings for an outer sequential loop.*

Proof: The synchronizations in the program ensure that partition $p = (p_1, \ldots, p_m)$ executes after $p' = (p'_1, \ldots, p'_m)$, $\forall p'_i \leq p_i$, whenever $p \neq p'$. By Definition 7.4 and the fact that all instructions in each partition execute in the original sequential order, the program is correct. Since the rank of Φ is r, the mapping divides the computation into $O(n^r)$ time partitions. From the synchronization pattern, we see that each partition only needs to wait at most $O(n)$ time step to start. Thus, there must be $r-1$ degrees of parallelism. □

Theorem 7.7 *Algorithm 7.3 finds the maximum degree of parallelism with at most one degree of synchronization in an outer sequential loop P.*

Proof: Let $\mathcal{B} = \{\Phi^1, \ldots, \Phi^m\}$ be the set of legal, maximally independent partition mappings found in Step 1 of Algorithm 7.3, and Φ of rank r be the pipelinable affine mapping used.

We prove the theorem by introducing another algorithm—we prove that the new algorithm has the property desired and establish its equivalence with Algorithm 7.3. Let $\Phi' = \sum_{\Phi^i \in \mathcal{B}} \Phi^i$, and $\hat{\Phi}$ be the first $m-1$ rows of Φ. The steps of the algorithm are: (1) partition the program P in time using Φ', (2) partition each time partition in space using $\hat{\Phi}$, and (3) find all the parallelism requiring at most $O(n)$ synchronizations in each of the space partitions. $\hat{\Phi}$ is a synchronization-free space-partition mapping for each partition in Φ', because Φ' dismisses all linear time-partition constraints between different partitions in Φ. If the rank of Φ is r, the rank of the m-dimensional mapping with rows Φ' and $\hat{\Phi}$ must also be r. Thus, the partitioning in time and space creates $O(n^r)$ partitions requiring $O(n)$ synchronizations, yielding $r-1$ degrees of parallelism. By Lemmas 7.4 and 5.3, the algorithm finds all the parallelism using $O(n)$ synchronizations.

By Lemma 7.6, pipelining P also yields $r-1$ degrees of parallelism with one degree of synchronization. Since Φ' is the sum of the rows in Φ, it is easy to show that for each partition in Φ, there exists a partition in $\hat{\Phi}$ with the same operations, and vice versa. Thus, Algorithm 7.3 must also find the maximum degree of parallelism with $O(n)$ synchronizations. □

7.5 Example

Consider the following example of an outer sequential loop:

Example 6:
```
for l1 = 1 to n-1 do
  for l2 = 0 to n-1 do
    for l3 = 0 to n-1-l2 do
      a[l1, l2+l3] += b[l2, l3] * a[l1-1, l2+l3];   [1]
    for l3' = l2+1 to n-1 do
      a[l1, l3'] -= b[n-1-l2, l3'] * a[l1, l2];      [2]
```

Instruction 1 modifies each l_1th row of matrix a using the lower triangular elements in matrix b, and instruction 2 updates $a[l_1]$ using the upper triangular portion of b.

Let us first consider how previous loop transformation algorithms would parallelize this code. Being able to handle only perfectly nested loops, algorithms based on unimodular transforms can only find parallelism in the innermost loops of this example. Loop fusion, a technique that combines a sequence of loops into one, has not been well integrated into the unimodular framework. It is not clear how one would fuse the innermost loops, given that the number of iterations in the loops are different. Even if fusion is applied, the result is still not a perfectly nested loop and is thus not amenable to unimodular transforms. In summary, previous approaches would find only one degree of parallelism in the program, with two degrees of synchronization. While our algorithm also finds one degree of parallelism, the parallelism we find is coarser grained, requiring only one degree of synchronization.

Let $\vec{l} = \begin{bmatrix} l_1 & l_2 & l_3 \end{bmatrix}^T$ and $\vec{l'} = \begin{bmatrix} l_1 & l_2 & l'_3 \end{bmatrix}^T$. When applied to this example, Step 1 of Algorithm 7.3 finds two legal, independent, affine time-partition mappings:

$$\Phi^1 = \left\{\Phi^1_1(\vec{l}) = l_1,\ \Phi^1_2(\vec{l'}) = l_1\right\}$$
$$\Phi^2 = \left\{\Phi^2_1(\vec{l}) = l_2 + l_3,\ \Phi^2_2(\vec{l'}) = l'_3\right\}$$

Step 2 of the algorithm pipelines the computation, by mapping the operations to processors using Φ^1 and partitioning the space partitions in time using Φ^2. The desired SPMD code can be derived from the affine partitions easily. The processor's ID is denoted by p; the ith wait(q) executed by processor p stalls its execution until processor q executes the ith signal(p).

```
if (1 <= p <= n-1) then
  if p > 1 then wait(p-1)
  a[p,0] += b[0,0] * a[p-1,0];                        [1]
  if p < n-1 then signal(p+1)
  for k1 = 1 to n-1 do
    if p > 1 then wait(p-1)
    for k2 = 0 to k1-1 do
      a[p,k1] += b[k2,k1-k2] * a[p-1,k1];             [1]
      a[p,k1] -= b[n-1-k2,k1] * a[p,k2];              [2]
    a[p,k1] += b[k1,0] * a[p-1,k1];                   [1]
    if p < n-1 then signal(p+1)
```

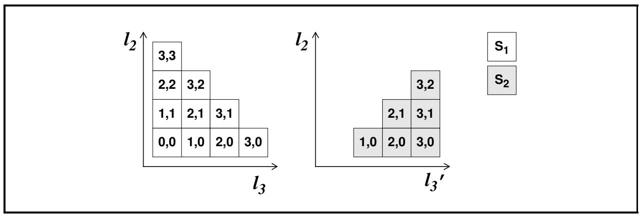

Figure 3: Execution order for each processor in Example 6.

The computation assigned to processor p is simply the pth iteration of the outermost loop. The execution order on each processor is illustrated in Figure 3. The figure shows the original iteration space of instructions 1 and 2 assigned to each processor, with each iteration labeled by their new indices (k_1, k_2) in the SPMD code. Note that the first column of the iteration space for instruction s_1 is executed outside the innermost loop. The labels on these operations indicate their relative excution order with respect to the other operations. The code generated by our algorithm requires each processor to synchronize with only two processors every $O(n)$ operations, whereas previous solutions of parallelizing only the innermost loops would require each processor to execute a barrier after every operation.

8 Maximum Degrees of Parallelism

We now present the final algorithm that uses all the components above to find all the parallelism, at their coarsest granularity, in a program. In practice, we only need to find enough coarse-grain parallelism to use the available parallel hardware. As the steps in the algorithm find successively finer granularities of parallelism, the algorithm can halt as soon as sufficient parallelism is found.

Algorithm 8.1 Find all the degrees of parallelism in a program, with all the parallelism being as coarse-grain as possible.

Step 1: Find the maximum degree of synchronization-free parallelism: Apply Algorithm 5.3 to the program.

Step 2: Find the maximum degree of parallelism that requires $O(1)$ synchronizations: Apply Algorithm 6.1 to each of the space partitions found in Step 1.

Step 3: Find the maximum degree of parallelism that requires $O(n)$ synchronizations: Apply Algorithm 7.2 or Algorithm 7.3 to each of the partitions found in Step 2.

Step 4: Find the maximum degree of parallelism with successively greater degrees of synchronization: Recursively apply Step 3 to computation belonging to each of the space partitions generated by the previous step until all parallelism is found.

Theorem 8.1 *Algorithm 8.1 finds all the degrees of parallelism in a program, with all the parallelism being as coarse-grain as possible.*

Proof: By Lemmas 5.3, 6.2, 7.4 and Theorem 7.7, the program maximizes the degree of parallelism in a program. By Theorems 5.2, 6.3, 7.5, and 7.7, Steps 1, 2 and 3 find all the degrees of parallelism with at most 0, $O(1)$ and $O(n)$ synchronization, respectively, and recursive application of Step 3 finds all the coarsest granularity of parallelism. □

9 Related Work

As there has been a lot of research on loop transformations we limit our comparison to work most closely related to ours. Loop interchanges, reversals, and skewing, and combinations thereof, have been modeled as unimodular transformations[6, 21, 22], and various algorithms based on this framework have been developed. Under this framework, the desired combination of loop transformations is achieved by finding the suitable unimodular matrix, which can then be used to generate the desired SPMD code in a straightforward manner. There are several major limitations to the unimodular loop transformation approach: imprecise dependence abstractions such as distance and direction vectors are used, operations within each iteration are treated as indivisible, and unimodular transformations do not apply to non-perfectly nested loops. Unimodular transformations cannot achieve the effects that can be obtained via loop fission, fusion, scaling, reindexing, or statement reordering.

Feautrier finds a piece-wise affine schedule that minimizes the overall execution time of a program[9, 10], and tries to minimize communication after parallelization[11]. For each instruction, he finds a piece-wise affine mapping that maps its loop indices to a multi-dimensional time index. He uses a heuristic based on parametric integer programming to minimize the dimensionality of time. Minimizing the dimensionality of a time schedule corresponds to maximizing the degree of parallelism.

In contrast, by solving for the coarsest granularity of parallelism, our algorithm minimizes synchronization as it finds parallelism. By focusing only on the degree of parallelism, rather than the shortest free schedule as in Feautrier's case, we can find optimal mappings via simple algorithms that find null spaces to a set of linear constraints. In addition, we can also generate pipeline parallelism that has better locality and less synchronization overhead. Code generation in our framework is straightforward because the schedules we find are more regular.

Kelly and Pugh presented a framework for unifying iteration reordering transformations[14]. They described a way to find an affine mapping for each instruction based on performance estimation[13]. Recently, they described an algorithm to minimize communication while preserving parallelism[15]. Their algorithm finds only one dimension of parallelism, rather than all the possible degrees of parallelism. It estimates the parallelism and the communication cost for all legal loop permutations and reversals for each instruction. It then builds a conflict graph and searches for the best mapping with the minimum cost. By restricting the search to only loop permutations and reversals for each instruction, the algorithm fails to find parallelism that requires skewing the loops.

Several algorithms have been proposed for solving the subproblem of finding communication-free parallelism. Huang and Sadayappan find one degree of communication-free parallelism from a sequence of perfectly nested loops[12]. Anderson and Lam apply unimodular transforms to loop nests to increase the granularity of parallelism, find the maximum degree of communication-free parallelism across loops, and heuristically introduce communication where necessary[3]. Bau et al. propose an improved algorithm to find communication-free parallelism[8]. While all the algorithms above treat operations within each loop iteration as indivisible, we previously presented a communication-free parallelization algorithm that transforms instructions individually[18]. The synchronization-free algorithm presented in this paper is simpler and more direct, made possible by the application of the Farkas lemma.

10 Conclusion

This paper presents an algorithm to find the optimal affine mapping that maximizes the degree of parallelism while minimizing the degree of synchronization. As the affine framework encompasses many previously proposed program transformations including unimodular transforms, loop fusion, loop fission, loop scaling, loop reindexing and statement reordering, our algorithm is more powerful than existing techniques.

We formulate the problem of maximizing parallelism and minimizing synchronization from first principles, without the use of imprecise abstractions such as data dependence vectors. We define two fundamental affine program transformation constraints: space-partition constraints and time-partition constraints, where the latter are relaxation of the former. We show that the problem of finding all the degrees of parallelism, at the coarsest granularity of parallelism, can be reduced to finding solutions to these constraints alternately.

The algorithm to this problem is made simple using two ideas. First, we use the Farkas lemma to reduce the constraints to a set of linear inequalities. Second, with the degrees of parallelism as the metric, optimal mappings can be obtained using simple algorithms that find solutions to linear inequalities.

Our algorithm to finding synchronization-free space partitions is not too different from the others in the literature. It is perhaps more surprising that a similarly simple technique can minimize synchronization and find time partitions with maximum degrees of parallelism. The key insight is that a set of maximally independent legal partition mappings can be combined linearly to form a time-partition mapping that yields maximum parallelism. Moreover, the legal mappings can be combined as rows of a multi-dimensional mapping to yield pipeline parallelism.

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] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the Third ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991.

[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] E. Ayguadé and J. Torres. Partitioning the statement per iteration space using non-singular matrices. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, pages 407–415, July 1993.

[5] D. Bacon, S. Graham, and O. Sharp. Compiler transformations for high-performance computing. *Computing Surveys*, 26(4):345–420, December 1994.

[6] 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.

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

[8] D. Bau, I. Kodukula, V. Kotlyar, K. Pingali, and P. Stodghill. Solving alignment using elementary linear algebra. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, pages 46–60. Springer-Verlag, August 1994.

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

[10] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II, multidimensional time. *Int. J. of Parallel Programming*, 21(6), December 1992.

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

[12] C. H. Huang and P. Sadayappan. Communication-free hyperplane partitioning of nested loops. *Journal of Parallel and Distributed Computing*, 19(2):90–102, October 1993.

[13] W. Kelly and W. Pugh. Determining schedules based on performance estimation. Technical Report CS-TR-3108, University of Maryland, 1993.

[14] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-2995.1, University of Maryland, April 1993.

[15] 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.

[16] 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.

[17] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, pages 301–320. Springer-Verlag, August 1993.

[18] A. W. Lim and M. S. Lam. Communication-free parallelization via affine transformations. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, pages 92–106. Springer-Verlag, August 1994.

[19] V. Sarkar and R. Thekkath. A general framework for iteration-reordering loop transformations. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 175–187, June 1992.

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

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

[22] 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.

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

APPENDIX A: Algorithm A

Input: A: a matrix
Output: B: a matrix

Description: The algorithm finds a maximal set of linearly independent solutions for $A\vec{x} \geq \vec{0}$, and expresses them as rows of matrix B.

Let m by n be the size of matrix A.
Let $a[b]$ denotes the bth component of a.

M: a matrix $:= A^T$;
$B := I_{n\times n}$; /* an n by n identity matrix */
$r_0 := 1; c_0 := 1$;

While true do

/* **1.** Make $M[r_0 : r' - 1][c_0 : c' - 1]$ into a diagonal matrix with positive diagonal entries and $M[r' : n][c_0 : m] = 0$. $M[r' : n]$ are solutions. */
$r' = r_0$; $c' = c_0'$;
While $\exists\ M[r][c] \neq 0$ s.t. $r - r', c - c' \geq 0$ do
  Move pivot $M[r][c]$ to $M[r'][c']$
    by row and column interchange.
  Interchange row r with row r' in B.
  If $M[r'][c'] < 0$ then
    $M[r'] := -1 * M[r']$;
    $B[r'] := -1 * B[r']$;
  end if;
  For $row = r_0$ to n do
    If $row \neq r' \ \wedge\ M[row][c'] \neq 0$ then
      $u := -(M[row][c']/M[r'][c'])$;
      $M[row] := M[row] + u * M[r']$;
      $B[row] := B[row] + u * B[r']$;
    end if;
  end for;
  $r' := r' + 1$; $c' := c' + 1$;
end while;

/* **2.** Find solution besides $M[r' : n]$. It must be a non-negative combination of $M[r_0 : r' - 1][c_0 : m]$. */
Find $k_{r_0}, \ldots, k_{r'-1} \geq 0$ s.t.
  $k_{r_0} M[r_0][c' : m] + \cdots + k_{r'-1} M[r' - 1][c' : m] \geq 0$;
If $\exists$ a non-trivial solution, say $k_r > 0$, then
  $M[r] := k_{r_0} M[r_0] + \cdots + k_{r'-1} M[r' - 1]$;
  $NoMoreSoln := False$;
else /* $M[r' : n]$ are the only solutions. */
  $NoMoreSoln := True$;
end if;

/* **3.** Make $M[r_0 : r_n - 1][c_0 : m] \geq 0$. */
If $NoMoreSoln$ then
  /* Move solutions $M[r' : n]$ to $M[r_0 : r_n - 1]$. */
  For $r = r'$ to n do
    Interchange rows r and $r_0 + r - r'$ in M and B;
  end for;
  $r_n := r_0 + n - r' + 1$;
else /* Use row addition to find more solutions. */
  $r_n := n + 1$;
  For $col = c'$ to m do
    If $\exists\ M[row][col] < 0$ s.t. $row \geq r_0$ then
      If $\exists\ M[r][col] > 0$ s.t. $r \geq r_0$ then
        For $row = r_0$ to $r_n - 1$ do
          If $M[row][col] < 0$ then
            $u := \lceil(-M[row][col]/M[r][col])\rceil$;
            $M[row] := M[row] + u * M[r]$;
            $B[row] := B[row] + u * B[r]$;
          end if;
        end for;
      else
        For $row = r_n - 1$ to r_0 step -1 do
          If $M[row][col] < 0$ then
            $r_n := r_n - 1$;
            Interchange $M[row]$ with $M[r_n]$;
            Interchange $B[row]$ with $B[r_n]$;
          end if;
        end for;
      end if;
    end if;
  end for;
end if;

/* **4.** Make $M[r_0 : r_n - 1][1 : c_0 - 1] \geq 0$ */
For $row = r_0$ to $r_n - 1$ do
  For $col = 1$ to $c_0 - 1$ do
    If $M[row][col] < 0$ then
      Pick an r s.t. $M[r][col] > 0 \wedge r < r_0$.
      $u := \lceil(-M[row][col]/M[r][col])\rceil$;
      $M[row] := M[row] + u * M[r]$;
      $B[row] := B[row] + u * B[r]$;
    end if;
  end for;
end for;

/* **5.** If necessary, repeat with rows $M[r_n : n]$ */
If $(NoMoreSoln \vee (r_n > n) \vee (r_n = r_0))$ then
  Remove row r_n to row n from B.
  Return B.
else
  $c_n := m + 1$;
  For $col = m$ to 1 step -1 do
    If $\nexists\ M[r][col] > 0$ s.t. $r < r_n$ then
      $c_n := c_n - 1$;
      Interchange column col with c_n in M;
    end if;
  end for;
  $r_0 := r_n$; $c_0 := c_n$;
end if;

end while;