ELSEVIER

Parallel Computing 24 (1998) 445–475

PARALLEL COMPUTING

Maximizing parallelism and minimizing synchronization with affine partitions [1]

Amy W. Lim [*], Monica S. Lam [2]

Computer Systems Laboratory, Stanford University, Stanford, CA 94305, USA

Received 10 April 1997; revised 2 September 1997

Abstract

This paper presents an algorithm to find the optimal affine partitions that maximize the degree of parallelism and minimize the degree of synchronization in programs 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 loop transformation algorithms that are based on unimodular transformations, loop distribution, fusion, scaling, reindexing, and statement reordering.

Keywords: Affine partitions; Affine transforms; Parallelizing compilers; Multiprocessors; Parallelism; Coarse granularity; Synchronization

1. Introduction

Experience with parallel applications on multiprocessors suggests that achieving high performance on such machines is nontrivial. Not only do we need to find sufficient parallelism in a program, but it is also important that we minimize the synchronization and communication overheads in the parallelized program. In fact, it is not uncommon to find parallel programs than run even slower than their serial counterpart due to the overhead of parallel execution. It is therefore important to increase the granularity of

[*] Corresponding author. E-mail: aimee@cs.stanford.edu

[1] A preliminary version of this paper appears in the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, Jan 1997 [26]. This research is supported in part by the Air Force Material Command and ARPA, contract F30602-95-C-0098, and an NSF Young Investigator award.

[2] E-mail: lam@cs.stanford.edu

0167-8191/98/$19.00
PII S0167-8191(98)00021-0

parallelism to reduce the frequency of synchronization and minimize the amount of data communicated between processors.

This paper presents a new program transformation model for the domain of sequential programs with arbitrary nestings and sequences of loops, whose array indices and loop bounds are affine expressions of outer loop indices. In this model, every instruction is given its own *affine partition* that divides the instances of the instruction across the processors or across time stages. More specifically, instances of each instruction are identified by the loop index values of their surrounding loops. The affine partitions map the loop indices to either virtual processor numbers or iteration numbers in a sequential loop. This model is general enough to represent all possible combinations of loop permutation (also known as interchange), reversal, skewing, reindexing (also known as alignment or index set shifting), distribution, fusion, scaling, and statement reordering.

We have developed an algorithm that finds the optimal affine partitions to maximize the degree of parallelism while minimizing the degree of synchronization. We say that a loop has k degrees of parallelism if $O(n^k)$ operations can execute in parallel, where n is the number of iterations. Similarly, a loop has k degrees of synchronization if $O(n^k)$ barrier synchronizations are needed in the parallel program. Our algorithm maximizes the degree of parallelism with successively greater degrees of synchronization. The algorithm can be used to find all the parallelism in a program at the coarsest granularity, or just find the parallelism up to the degree needed to exploit a particular hardware configuration. The algorithm uses the array index functions directly, and is therefore more powerful than previous algorithms based on imprecise abstractions such as distance and direction vectors. Once the affine partitions are found, simple algorithms based on Fourier–Motzkin elimination [1] can be used to generate the corresponding parallel program expressed in an SPMD (Single Program Multiple Data) style.

In Section 2, we provide background for this work by describing several major approaches to program transformations. We then explain the different forms of parallelism in Section 3, present our problem statement in Section 4, and give an overview of our algorithm in Section 5. In Section 6, we formally define our program representation and affine partition mappings. We present algorithms for solving several subproblems in Section 7 to 9 that can be combined to find the maximum degree of coarse-grain parallelism. Finally, we conclude in Section 11.

2. Background

Much research has been performed in the area of parallelization and communication minimization. In the following, we outline four major approaches and contrast our proposal with previous models.

2.1. Loop transformations

Many loop transformations have been shown to improve parallelism and data locality in programs [2], examples of which include loop permutation, reversal, skewing, reindexing, distribution and fusion [3–8].

Parallelization algorithms based on this approach translate sequential loop nests into semantically equivalent sequential loop nests, with the goal that the resulting code contains one or more parallelizable loops at the outermost possible nesting levels. Outer parallel loops are preferred over inner parallel loops as synchronization barriers are introduced before and after the execution of each parallel loop. Because it leaves the mapping of iterations to processors unspecified, this approach cannot be used to minimize the interprocessor communication across parallel loops.

The legality of the individual loop transformations is defined using the abstraction of *distance* and *direction vectors*. These vectors capture the common conditions, but not all the conditions, under which the transformations can be applied. It is easy to determine if a loop transformation on its own improves the parallelism of a program, but finding the optimal combination of transforms remains an open question. Current approaches tend to use either heuristics or choose a pre-defined sequence of transforms that have been found to be useful for some programs [8–11].

2.2. Systolic array synthesis techniques

Program transformations have also been influenced by the techniques developed for synthesizing systolic algorithms [12,13]. Intended to be mapped directly onto VLSI implementations, systolic designs are limited to the domain of computations specified by a set of recurrence equations. The regularity of a computation is directly translated into a hardware design, with the functional units typically arranged as an array with a regular interconnect. A systolic algorithm specifies precisely when and where each operation is performed, as well as when and how each data item is communicated between processors.

The synthesis of systolic algorithms has been formulated as a choice in mapping the index set representing the computation onto a space–time domain using geometric transformations. An $O(n^k)$ computation is represented by a k-dimensional index set; the dependences between the computations are represented as a set of constant distances between pairs of indices. Computations in this domain can be executed in parallel on an array with $O(n^{k-1})$ processors. This is achieved by mapping the k-dimensional space onto 1 dimension in time and $k-1$ dimensions in space. Projecting the dependence vectors onto the time axis must yield positive vectors in order to satisfy the dependences in the program; projecting the dependence vectors onto the processor subspace yields the interprocessor communication pattern.

2.3. Unimodular transformations

A geometric model similar to that used in systolic array synthesis has also been applied to transforms of perfectly nested loops. In the unimodular loop transformation framework [14,15], an n-deep loop nest is modeled as an n-dimensional space, where each iteration is treated as an atomic unit. The iterations are executed in lexicographical order in a sequential execution. The dependences in the loop are abstracted by a set of

distance and direction vectors. By applying a unimodular transform to the original iteration space to produce a new iteration space, a sequential loop is transformed into another loop. A transformation is legal as long as the dependence vectors in the original iteration space remains lexicographically positive in the transformed space.

Since any combination of loop permutation, skewing and reversal can be represented by a unimodular transformation, the problem of finding the best combination of these loop transforms reduces to finding the best unimodular transform. Algorithms have been developed to transform a sequential loop nest into a semantically equivalent one with better locality. The problem of finding coarse-grain parallelism in a loop nest is translated to finding a sequential loop nest such that the outermost possible loop is parallelizable.

2.4. Affine scheduling

Whereas unimodular transforms apply only to perfectly nested loops, affine scheduling [16–18] applies to arbitrary nestings and sequences of loops. Whereas unimodular transforms unify loop permutation, skewing and reversal, affine scheduling unifies unimodular transforms with loop distribution, fusion, reindexing and scaling.

In the affine scheduling model, each instruction is given its own affine mapping, which maps the instances of the instruction identified by their loop indices to their execution times. Unlike the source-to-source and unimodular transformation models that transform a sequential code into another equivalent sequential code, the parallelism is explicit in the affine scheduling model. Operations assigned the same time are executed in parallel. Execution time in this model may have multiple dimensions; the coordinates in the time domain are executed sequentially in lexicographical order.

Feautrier has developed an algorithm that finds a piecewise affine schedule to minimize the overall execution time of a program [19,20]. 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 at all and must be handled by postpass techniques [21]. Moreover, it is nontrivial to translate the piecewise affine schedules into parallel code and the resulting code can be quite complicated.

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 [22]. In contrast, Kelly and Pugh's algorithm finds one dimension of parallelism for programs with arbitrary nestings and sequences of loops [23]. 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.

2.5. Affine processor and time partitioning

We present a new model that uses affine mappings to explicitly partition computation across processors. This model makes it easy to develop algorithms that maximize

parallelism and minimize synchronization simultaneously. Moreover, this model lends itself to a straightforward translation from the original code to SPMD programs.

Our model also uses affine expressions to partition computation into sequential time stages. However, unlike affine scheduling, which totally orders all operations, we only need to specify the ordering between operations nested within the same sequential loops. This is significant because while the multi-dimensional time representation captures the timing relationships between operations within the same loop nesting well, it is not expressive enough to fully represent the general program structure containing arbitrarily nested sequences of loops.

In summary, the affine partitioning model enables the design of powerful transformation algorithms as it provides a unified notation for describing combinations of unimodular transforms, loop scaling, reindexing, distribution and fusion. Unlike the affine scheduling model, mappings from computation to processors are explicit in our model, thus allowing algorithms to optimize for parallelism and communication together. Finally, although systolic arrays also map computation to processors directly, our model is much more general. Systolic array synthesis techniques are applicable to simple computation domains characterized by uniform recurrence equations. In contrast, our algorithm can handle arbitrary nestings and sequences of loops, handle arbitrary affine array index expressions accurately without having to 'uniformize' them, optimize operations within an iteration individually, and finally produce schedules with multi-dimensional time.

3. Forms of parallelism

We introduce three examples in Fig. 1 to illustrate the different forms of parallelism. Using the iteration space representation, the point at coordinate (l_1,l_2) represents the operation in iteration $\langle l_1,l_2 \rangle$ and the arrows represent data dependences between operations. For the different parallelization schemes, each thick line represents a barrier synchronization, and each gray box groups together computations that are assigned to the same processor.

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 one row at a time. A barrier is needed to ensure that all processors have finished their assigned iterations, before any can proceed to the next row.

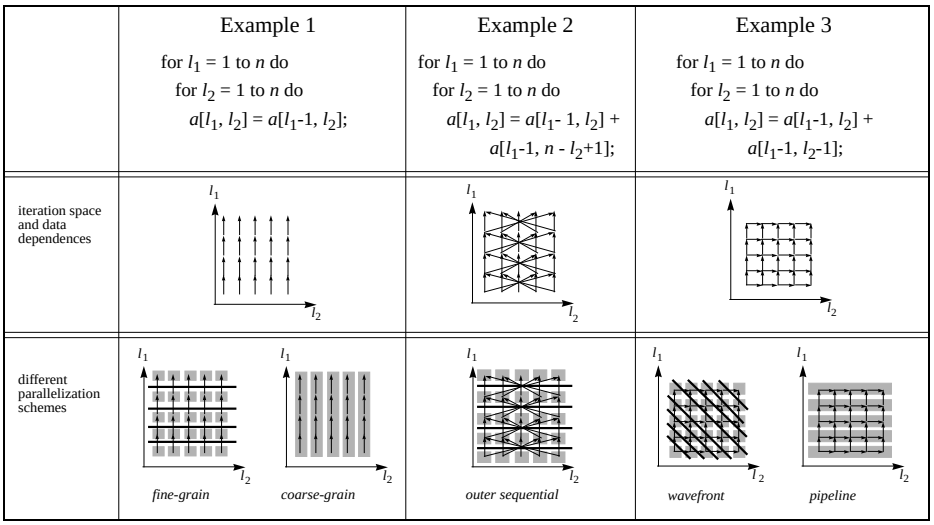

Fig. 1. Examples 1, 2, 3: iteration space, data dependences and parallelization schemes.

Like Example 2, Example 3 does not have any synchronization-free parallelism. However, unlike Example 2, there is a choice in parallelization schemes. One choice is to *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 is to *pipeline* the computation. For example, by assigning each row to a processor, each processor can proceed with the computation at a given column as soon as its neighboring processor that is assigned the row below finishes executing its assigned computation at 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 is simpler, and finally the processors tend to have better data locality.

When a program has multiple degrees of parallelism, we say that different degrees of parallelism are at the same nesting level if and only if they require the same degree of synchronization. For example, a 2-deep loop nest whose iterations are completely independent has two degrees of parallelism. Both degrees of parallelism are at the same level despite the original nesting structure, and they require no synchronization. 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:

```
for l0=1 to n do
    for l1=1 to n do
       for l2=1 to n do
           a[l0,l1,l2]=a[l0,l1 − 1,l2]+a[l0,l1 − 1,n − l2+1];
```

Each iteration of the outermost loop l_0 performs the same computation as Example 2 on a disjoint set of data. Loop l_0 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 l_0 to a set of n processors. Each set of n processors must participate in n barrier synchronizations, one for each iteration of the middle loop l_1. 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.

4. Problem statement

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. Constructs such as conditionals and non-affine accesses are handled by treating them conservatively. Our goal is to derive the optimal affine processor (or space) and time partitions for each instruction to maximize the degree of loop-level and pipelined parallelism with the minimum degree of synchronization.

From these affine partition mappings, a straightforward algorithm based on Fourier–Motzkin elimination [1] can be used to generate the desired SPMD (Single Program Multiple Data) code. In this paper, the SPMD code we show assumes each processor partition is mapped to a physical processor. To generate code for a specific number of processors, we can simply combine multiple 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.

5. 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, $O(1)$, and $O(n)$ 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 of the available hardware.

The subproblem of maximizing synchronization-free parallelism is formulated as dividing the 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 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 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 kinds of techniques. We use the affine form of the Farkas Lemma [19] 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.

6. Definitions

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

6.1. Program and data dependences

In the following, we describe our representation for a program and its data dependences.

Definition 6.1. A *sequential program* is represented as $P = \langle \mathscr{S}, \delta, \mathscr{D}, \mathscr{F}, \omega, \eta \rangle$, where

- $\mathscr{S}$ 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' if and only if $s <_p s'$.

- δ_s is the *depth*, or the number of surrounding loops, of instruction s.
- $\mathscr{D}_s(\vec{i}) = D_s\vec{i} + \vec{d}_s$ is an affine expression derived from the loop bounds such that $\vec{i}$ is a valid loop index for instruction s, if and only if $\mathscr{D}_s(\vec{i}) \geq \vec{0}$. [3]
- $\mathscr{F}_{zsr}(\vec{i}) = F_{zsr}\vec{i} + \vec{f}_{zsr}$ is the affine array index expression in the rth array reference to array z in instruction s.
- ω_{zsr} is true if and only if the rth array reference to array z in instruction s is a write operation.
- $\eta_{ss'}$ is the number of common loops shared by instructions s and s'.

The access patterns in a program define the constraints of program transformations. The notion of data dependences is well understood. Informally, there is a *data dependence* from an access function $\mathscr{F}$ to another access function $\mathscr{F}'$, if and only if some instance of $\mathscr{F}$ uses a location that is subsequently used by $\mathscr{F}'$, 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 6.2. We define $\prec$ to be the 'lexicographically less than' operator for program $P = \langle \mathscr{S}, \delta, \mathscr{D}, \mathscr{F}, \omega, \eta \rangle$ such that $\vec{i} \prec_{ss'} \vec{i'}$ if and only if iteration $\vec{i}$ of instruction s is executed before iteration $\vec{i'}$ of s' in P. That is,

$$\vec{i} \prec_{ss'} \vec{i'} \equiv \bigvee_{m=0}^{\eta_{ss'}} L_{ss'}^{m}\left(\vec{i}, \vec{i'}\right) = \begin{cases} \vec{i}_{1:\eta_{ss'}} = i_{1:\eta_{ss'}}{}' \wedge s <_p s' & m = \eta_{ss'} \\ \vec{i}_{1:m} = \vec{i'}_{1:m} \wedge i'_{m+1} & 0 \leq m < \eta_{ss'} \end{cases}$$

Definition 6.3. The *data dependence set* of a program $P = <\mathscr{S}, \delta, \mathscr{D}, \mathscr{F}, \omega, \eta>$ is

$$\begin{aligned} R = \Big\{ \langle \mathscr{F}_{zsr}, \mathscr{F}_{zs'r'} \rangle \Big| & \left(\omega_{zsr} \vee \omega_{zs'r'} \right) \\ & \wedge \left(\exists \vec{i} \in \mathscr{Z}^{\delta_s}, \vec{i'} \in \mathscr{Z}^{\delta_{s'}} \middle| \left(\vec{i} \prec_{ss'} \vec{i'} \right) \right. \\ & \quad \wedge \left(\mathscr{F}_{zsr}\left(\vec{i}\right) - \mathscr{F}_{zs'r'}\left(\vec{i'}\right) = \vec{0} \right) \\ & \quad \wedge \left(\mathscr{D}_s\left(\vec{i}\right) \geq \vec{0} \right. \\ & \qquad \left. \left. \wedge \mathscr{D}_{s'}\left(\vec{i'}\right) \geq \vec{0} \right) \right) \Big\} \end{aligned}$$

When known, the actual values of loop bounds are used for better accuracy.

[3] Constant loop bounds can be represented as symbolic variables to avoid the introduction of large coefficients in the linear systems. For example, $\{10000 - l_1 \geq 0\}$ can be represented as $\{n - l_1 \geq 0\}$ with the value for variable n set to 10,000 before code generation.

6.2. Affine partition mappings

In this section, we define affine partition mappings and a few properties used in our algorithm.

Definition 6.4. 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. [4] An *m*-dimensional affine partition mapping for a program *P* is $\Phi = [\Phi_1, \Phi_2, \ldots \Phi_k]$, where *k* is the number of instructions in *P*. [5]

Definition 6.5. 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 the coefficient 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 6.6. 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* if and only if the coefficient matrices C_s and C'_s are linearly dependent. Two one-dimensional affine partition mappings Φ and Φ' for program *P* are linearly dependent if and only if Φ_s and Φ'_s are *linearly dependent* for all instructions *s* in *P*.

7. 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.

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

[5] 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.

7.1. Space-partition constraints

Definition 7.1. (*Space-partition constraints*) Let R be the data dependence set for program $P = \langle \mathcal{S}, \delta, \mathcal{D}, \mathcal{F}, \omega, \eta \rangle$. An affine space-partition mapping Φ for P is *synchronization-free* if and only if [6]

$$\forall \langle \mathcal{F}_{zsr}, \mathcal{F}_{zs'r'} \rangle \in R, \vec{i} \in \mathcal{Z}^{\delta_s}, \vec{i'} \in \mathcal{Z}^{\delta_{s'}} \text{ s.t.} \mathcal{D}_s(\vec{i}) \geq \vec{0} \wedge \mathcal{D}_{s'}(\vec{i'}) \geq \vec{0}$$

$$\wedge \mathcal{F}_{zsr}(\vec{i}) - \mathcal{F}_{zs'r'}(\vec{i'}) = \vec{0},\ \Phi_s(\vec{i}) - \Phi_{s'}(\vec{i'}) = 0 \tag{1}$$

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, $\langle \mathcal{F}_{zsr}, \mathcal{F}_{zs'r'} \rangle \in R$, imposes the following constraints on each *row* of a synchronization-free mapping for instructions s and s', $X = [-C_s C_{s'}\ (c_{s'} - c_s)]$

$$\forall \vec{y} s.t. G \begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} \geq \vec{0} \wedge H \begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} = \vec{0},\ X \begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} = 0 \tag{2}$$

where

$$\vec{y} \equiv \begin{bmatrix} \vec{i} \\ \vec{iv'} \end{bmatrix},\ G = \begin{bmatrix} D_s & 0 & \vec{d}_s \\ 0 & D_{s'} & \vec{d}_{s'} \end{bmatrix},\ H = \begin{bmatrix} F_{zsr} & -F_{zs'r'} & \left(\vec{f}_{zsr} - \vec{f}_{zs'r'} \right) \end{bmatrix}$$

7.2. Linearizing the constraints

We first convert the space-partition constraints into a set of linear equations as follows:

Algorithm 7.1. Convert constraints in the form of Eq. (2) into a system of equations.

Step 1. Simplify the constraints by successively removing variables in y using the equations $H \begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} = \vec{0}$ in Eq. (2). [7] 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

[6] For synchronization-free parallelism, it is not necessary to distinguish between the directions of the data dependences in R. The redundancy is included in the constraints to emphasize the similarity with the definition in Section 9. It is a matter of simple optimization to eliminate the redundant constraints.

[7] Pairs of inequalities in $G \begin{bmatrix} \vec{y'} \\ 1 \end{bmatrix} \geq \vec{0}$ can sometimes be reduced to an equation, which can then be eliminated.

$\vec{y}$, in terms of the other variables in the equation. Then, we substitute all occurrences of y_i with the expression. The result is of the form:

$$\forall \vec{y}' \text{ s.t. } G' \begin{bmatrix} \vec{y}' \\ 1 \end{bmatrix} \geq \vec{0},\ XE \begin{bmatrix} \vec{y}' \\ 1 \end{bmatrix} = 0 \tag{3}$$

where vector $\vec{y}'$ has fewer variables than $\vec{y}$. E is constructed by applying the substitutions to an identity matrix and removing the substituted columns afterwards.

Step 2. Reduce Eq. (3) to a set of linear equations, using the Affine Form of the Farkas Lemma [19].

Lemma 7.1. (*Affine form of the Farkas Lemma*)

Let $x(\vec{y})$ *be an affine expression of* $\vec{y}$ *and* $G \begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} \geq \vec{0}$ *define a non-empty polyhedron.*

$$\forall \vec{y} \text{ s.t. } G \begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} \geq \vec{0},\ x(\vec{y}) \geq 0 \textbf{ if and only if}$$

$$x(\vec{y}) \equiv [\lambda_1 \cdots \lambda_m \lambda_0] \begin{bmatrix} G \\ 0 \cdots 0\ 1 \end{bmatrix} \begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} \text{ with } [\lambda_1 \cdots \lambda_m \lambda_0]^T \geq \vec{0}$$

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

We rewrite the constraint in Eq. (3) as Eqs. (4) and (5):

$$\forall \vec{y}' \text{ s.t. } G' \begin{bmatrix} \vec{y}' \\ 1 \end{bmatrix} \geq \vec{0},\ XE \begin{bmatrix} \vec{y}' \\ 1 \end{bmatrix} \geq 0 \tag{4}$$

$$\forall \vec{y}' \text{ s.t. } G' \begin{bmatrix} \vec{y}' \\ 1 \end{bmatrix} \geq \vec{0},\ -XE \begin{bmatrix} \vec{y}' \\ 1 \end{bmatrix} \geq 0 \tag{5}$$

Let m be the number of rows in G. Applying the Farkas Lemma to Eq. (4),

$$XE \begin{bmatrix} \vec{y}' \\ 1 \end{bmatrix} = [\lambda_1 \cdots \lambda_m \lambda_0] \begin{bmatrix} G' \\ 0 \cdots 0\ 1 \end{bmatrix} \begin{bmatrix} \vec{y}' \\ 1 \end{bmatrix} \wedge [\lambda_1 \cdots \lambda_m \lambda_0]^T \geq \vec{0} \tag{6}$$

Since Eq. (6) holds for all $\vec{y}'$,

$$XE = [\lambda_1 \cdots \lambda_m \lambda_0] \begin{bmatrix} G' \\ 0 \cdots 0\ 1 \end{bmatrix} \wedge [\lambda_1 \cdots \lambda_m \lambda_0]^T \geq \vec{0} \tag{7}$$

The unknowns in the new constraints (7) are $\lambda_0, \cdots, \lambda_m$, and the coefficients in X. We are interested only in the coefficients in X, so we apply Fourier-Motzkin elimination

[1] to eliminate all the Farkas multipliers to obtain a set of constraints in the form of $X A \geq \vec{0}$. Renaming $-X$ in Eq. (5) as X' and applying the Farkas Lemma and Fourier-Motzkin elimination as above, we get another set of constraints, $X'A \geq \vec{0} \equiv XA \leq \vec{0}$. Combining the two sets of constraints, we get a set of linear constraints on X:

$$XA = \vec{0} \text{ or } A^T X^T = \vec{0} \tag{8}$$

7.3. 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 7.2. Find a highest-ranked synchronization-free affine partition mapping for program $P = \langle \mathscr{S}, \delta, \mathscr{D}, \mathscr{F}, \omega, \eta \rangle$.

Step 1. From each data dependence in R, construct the space-partition constraints on a one-dimensional partition mapping in the form of Eq. (2).

Step 2. Apply Algorithm 7.1 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 in $C_1, C_2, \ldots$, and constant terms $c_1, c_2, \ldots$ of the affine partition mapping.

Step 3. Since the rank of an affine partition mapping is not dependent on the values any 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 7.1. Let the simplified system be $A'\vec{x}' = \vec{0}$, where $\vec{x}'$ represents only the unknown coefficients in C.

Step 4. Find the solutions to $A'\vec{x}' = \vec{0}$, expressed as $\mathscr{B}$, 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 $\mathscr{B}$. The coefficients in C are specified directly by the basis vector, and all the constant terms c are derived from the coefficients in C using $A\vec{x} = \vec{0}$.

Note that the basis vectors in $\mathscr{B}$, representing the coefficients of the affine mapping for all instructions, are linearly independent. However, the rows in the affine mappings for each instruction 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.

7.4. Parallelism with no synchronization

Algorithm 7.3. Find all the degrees of synchronization-free parallelism for program P.

Step 1. Apply Algorithm 7.2 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 7.2. *Algorithm 7.3 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 7.3, 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 synchronization-free parallelism found by Algorithm 7.3. Our code generation algorithm is based on Ancourt and Irigoin's polyhedron-scanning code generation technique [24]; details and examples can be found in Ref. [25].

To ensure that Algorithm 7.2 can be used as a step in finding the maximum degrees of parallelism with and without synchronizations, we have the following lemma:

Lemma 7.3. *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. □

7.5. Example

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

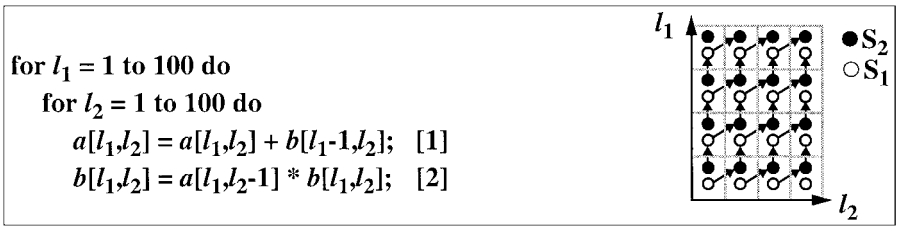

Fig. 2. An example with a subset of operations in iteration space representation.

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 7.2 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:

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

where

$$D = \begin{bmatrix} 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix}^T, \ \vec{d} = [-1\ 100\ -1\ 100]^T$$

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

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

$$\mathscr{F}_{b21}\left(\begin{bmatrix} l'_1 \\ l'_2 \end{bmatrix} \right) = F_{b21} \begin{bmatrix} l'_1 \\ l'_2 \end{bmatrix} + \vec{f}_{b21} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} l'_1 \\ l'_2 \end{bmatrix}$$

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

$$\mathscr{F}_{a21}\left(\begin{bmatrix} l'_1 \\ l'_2 \end{bmatrix} \right) = F_{a21} \begin{bmatrix} l'_1 \\ l'_2 \end{bmatrix} + \vec{f}_{a21} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} l'_1 \\ l'_2 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

We derive from the data dependent pair $\langle \mathscr{F}_{b11}, \mathscr{F}_{b21} \rangle$ the constraint

$$\forall \vec{y} \text{ s.t. } \begin{bmatrix} D & 0 & \vec{d} \\ 0 & D & \vec{d} \end{bmatrix} \begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} \geq \vec{0} \wedge \begin{bmatrix} F_{b11} & -F_{b21} & \vec{f}_{b11} - \vec{f}_{b21} \end{bmatrix} \begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} = \vec{0},\ X_1 \begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} = 0 \tag{9}$$

where $\vec{y} \equiv [\, l_1\ l_2\ l'_1\ l'_2\,]^T$ and $X_1 = [\, -C_1\ C_2\ c_2 - c_1\,]$.

Similarly, we derive from $\langle \mathscr{F}_{a11}, \mathscr{F}_{a21} \rangle$ the constraint

$$\forall \vec{y} \text{ s.t. } \begin{bmatrix} D & 0 & \vec{d} \\ 0 & D & \vec{d} \end{bmatrix} \begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} \geq \vec{0} \wedge \begin{bmatrix} F_{a11} & -F_{a21} & \vec{f}_{a11} - \vec{f}_{a21} \end{bmatrix} \begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} = \vec{0},\ X_2 \begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} = 0 \tag{10}$$

where $X_2 = [\, -C_1\ C_2\ c_2 - c_1\,]$.

Step 2. Apply Algorithm 7.1 to rewrite constraints (9) and (10) as a system of linear equations. For constraint (9), 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

$$\forall l'_1, l'_2 \text{ s.t. } G' \begin{bmatrix} l'_1 \\ l'_2 \\ 1 \end{bmatrix} \geq \vec{0},\ X_1 E \begin{bmatrix} l'_1 \\ l'_2 \\ 1 \end{bmatrix} = 0 \tag{11}$$

where

$$G' = \begin{bmatrix} 1 & -1 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 \\ 0 & 99 & -1 & 100 & -1 & 100 & -1 & 100 \end{bmatrix}^T,$$

$$E = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix}^T$$

Apply the Farkas Lemma to Eq. (11) and eliminate the Farkas multipliers λ's and μ's using Fourier-Motzkin elimination. [8] The Farkas Lemma rewrites Eq. (11) as

$$X_1 E = [\, \lambda_1\ \cdots\ \lambda_8\ \lambda_0\,] G' \wedge \lambda_0, \ldots, \lambda_8 \geq 0$$

$$-X_1 E = [\, \mu_1\ \cdots\ \mu_8\ \mu_0\,] G' \wedge \mu_0, \ldots, \mu_g \geq 0$$

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

The final system of equations with the Farkas multipliers eliminated is

$$X_1 \begin{bmatrix} 100 & 100 & 2 & 2 & 2 & 2 \\ 1 & 100 & 1 & 2 & 99 & 100 \\ 99 & 99 & 1 & 1 & 1 & 1 \\ 1 & 100 & 1 & 2 & 99 & 100 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \vec{0} \tag{12}$$

For constraint (10), applying similar steps above yields

$$X_2 \begin{bmatrix} 1 & 100 & 1 & 2 & 99 & 100 \\ 1 & 1 & 99 & 99 & 99 & 99 \\ 1 & 100 & 1 & 2 & 99 & 100 \\ 2 & 2 & 100 & 100 & 100 & 100 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \vec{0} \tag{13}$$

Step 3. Eliminate all the constant terms c from the space-partition constraints (12) and (13). Using the first column in Eq. (12), we substitute $c_2 - c_1$ with

$$[-100 \quad -1 \quad -99 \quad -1\,][-C_1 \quad C_2]^T$$

The resulting constraints on the coefficients in C are

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

Step 4. Find the solution space to the constraints derived in Step 3. The solution space is the null space of the matrix, which is spanned by the vector $\vec{v} = (-1\ 1\ 1\ -1)$.

Step 5. Construct a synchronization-free partition mapping from $\vec{v}$. The coefficients in 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];
  for l1=max(1, 1+p) to min(100, 99+p)
      a[l1, l1−p]=a[l1, l1−p]+b[l1−1, l1−p];
      b[l1, l1−p+1]=a[l1, l1−p]∗b[l1, l1−p+1];
  if (−98 ≤ p ≤ 0) then
      a[100+p, 100]=a[100+p, 100]+b[99+p, 100];
if (p=100) then
    b[100, 1]=a[100, 0]∗b[100, 1];
```

8. Parallelism with $O(1)$ synchronizations

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

Definition 8.1. Let R be the data dependence set for program $P = \langle \mathcal{S}, \delta, \mathcal{D}, \mathcal{F}, \omega, \eta \rangle$. The dependence graph for P is $G = (V, E)$, where node $v_s \in V$ represents instruction s, and $\langle v_s, v_{s'} \rangle \in E$ if and only if $\exists z,\ r,\ r'$ s.t. $\langle \mathcal{F}_{zsr},\ \mathcal{F}'_{zs'r'} \rangle \in R$.

Lemma 8.1. *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 instructions 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 8.1. Find all the degrees of parallelism requiring $O(1)$ synchronizations for a program.

Step 1. Construct the program dependence graph and partition the instructions into SCCs.

Step 2. Apply Algorithm 7.3 to each SCC to find all of 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 8.2. *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 8.3. *Algorithm 8.1 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 8.2, 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. □

9. 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 8.1, there must exist an outermost loop that surrounds all the instructions in such a program; by Theorem 8.3, 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 the processors 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 are 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 when there is more than one solution, there exists parallelism within one degree of synchronization. A linear combination of these solutions would yield 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.

9.1. 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 be executed either in the same iteration as v, or in a later iteration than v. Thus, the time-partition constraints are simply a more relaxed version of the space-partition constraints.

Definition 9.1. (*Time-partition constraints*) Let R be the data dependence set for program $P = \langle \mathscr{S}, \delta, \mathscr{D}, \mathscr{F}, \omega, \eta \rangle$. A one-dimensional affine partition mapping Φ for P is *legal* if and only if

$$\forall \langle \mathscr{F}_{zsr}, \mathscr{F}_{zs'r'} \rangle \in R, \vec{i} \in \mathscr{Z}^{\delta_s}, \vec{i'} \in \mathscr{Z}^{\delta_{s'}}$$

$$\text{s.t. } \vec{i} \prec_{ss'} \vec{i'} \wedge \mathscr{D}_s(\vec{i}) \geq \vec{0} \wedge \mathscr{D}_{s'}(\vec{i'}) \geq \vec{0}$$

$$\wedge \mathscr{F}_{zsr}(\vec{i}) - \mathscr{F}_{zs'r'}(\vec{i'}) = \vec{0},\ \Phi_{s'}(\vec{i'}) - \Phi_s(\vec{i}) \geq 0 \tag{14}$$

Using Definition 6.2, we can rewrite Eq. (14) as a set of linear time-partition constraints:

$$\forall \langle \mathscr{F}_{zsr}, \mathscr{F}_{zs'r'} \rangle \in R, \vec{i} \in \mathscr{Z}^{\delta_s}, \vec{i'} \in \mathscr{Z}^{\delta_{s'}} \forall m \in \{0, \ldots, \eta_{ss'}\} \text{ s.t. } L^m_{ss'}(\vec{i}, \vec{i'})$$

$$\wedge \mathscr{D}_s(\vec{i}) \geq \vec{0} \wedge \mathscr{D}_{s'}(\vec{i'}) \geq \vec{0} \wedge \mathscr{F}_{zsr}(\vec{i}) - \mathscr{F}_{zs'r'}(\vec{i'}) = \vec{0},\ \Phi_{s'}(\vec{i'}) - \Phi_s(\vec{i}) \geq 0 \tag{15}$$

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.

9.2. 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 how we find all the possible legal partitions.

Definition 9.2. A set of legal one-dimensional affine partition mappings $\mathscr{B} = \{\Phi^1, \Phi^2, \ldots, \Phi^n\}$ for program $P = \langle \mathscr{S}, \delta, \mathscr{D}, \mathscr{F}, \omega, \eta \rangle$ is said to be *maximally independent* if and only if the set is minimal, and any legal one-dimensional affine partition mapping Φ can be expressed as an affine combination of the mappings in $\mathscr{B}$. That is, for all legal Φ

$$\exists k_1, \ldots, k_n \in Q \forall s \in \mathscr{S} \exists a \in Q$$

$$\Phi_s(\vec{y}) = k_1 \Phi_s^1(\vec{y}) + k_2 \Phi_s^2(\vec{y}) + \cdots + k_n \Phi_s^n(\vec{y}) + a$$

Algorithm 9.1. Find a set of legal, maximally independent, affine partition mappings for an outer sequential loop.

Step 1. Using the same steps in Algorithm 7.1, we rewrite each system of constraints from Eq. (15) as $A\vec{x} \geq \vec{0}$. Whether two affine partition mappings are linearly independent depends only on their coefficient matrices C_s and C'_s. Thus, we eliminate all 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 in $C_1, C_2, \ldots$

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

$$A_1\vec{x}_1 \geq \vec{0} \equiv \begin{bmatrix} A_1 & -I_{m\times m} \end{bmatrix} \begin{bmatrix} \vec{x}_1 \\ \vec{b} \end{bmatrix} = \vec{0}, \vec{b} \geq \vec{0} \tag{16}$$

where the size of matrix A_1 is $m \times n$. It is obvious that any solution to Eq. (16) must also be a solution to

$$\begin{bmatrix} A_1 & -I_{m\times m} \end{bmatrix} \begin{bmatrix} \vec{x}_1 \\ \vec{b} \end{bmatrix} = \vec{0} \tag{17}$$

Conversely, solutions to Eq. (17) are solutions to Eq. (16) if the values for $\vec{b}$ are non-negative. One basis for the solutions to Eq. (17) consists of the rows of matrix $V = [I_{n\times n} \sim A_1^T]$. Algorithm A applies elementary row operations [9] 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 in C are given directly by $\vec{x}_1$, and all the constant terms c are derived using $A\vec{x} \geq \vec{0}$.

9.3. 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

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

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 9.3. A legal affine partition mapping *dismisses* the linear time–partition constraint imposed by $\langle \mathscr{F}_{zsr}, \mathscr{F}_{zs'r'} \rangle \in R$ at level m if and only if

$$\forall \vec{i} \in \mathscr{Z}^{\delta_s}, \vec{i'} \in \mathscr{Z}^{\delta_{s'}} \; s.t. \; L^m_{ss'}(\vec{i}, \vec{i'}) \wedge \mathscr{D}_s(\vec{i}) \geq \vec{0} \wedge \mathscr{D}_{s'}(\vec{i'}) \geq \vec{0} \wedge \mathscr{F}_{zsr}(\vec{i}) - \mathscr{F}_{zs'r'}(\vec{i'}) = \vec{0}, \; \Phi_{s'}(\vec{i'}) - \Phi_s(\vec{i}) > 0 \tag{18}$$

Algorithm 9.2. Find all the degrees of parallelism requiring $O(n)$ synchronizations for an outer sequential loop P.

Step 1. Apply Algorithm 9.1 to program P to get a set of maximally independent partition mappings $\mathscr{B}$. Partition the operations in time using $\Phi = \sum_{\Phi^i \in \mathscr{B}} \Phi^i$. 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 8.1 to each time partition of Φ to find all the degrees of parallelism requiring at most $O(1)$ synchronizations.

To show that Algorithm 9.2 finds all the parallelism requiring one degree of synchronization, we introduce the following lemmas.

Lemma 9.1. *Any positive linear combination of legal affine partition mappings is also a legal affine partition mapping.*

Proof. Combinations of affine partition mappings that satisfy the time-partition constraint (14) must also satisfy the same constraints. □

Lemma 9.2. *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 9.3. *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 9.4. *Let $\mathscr{B} = \{\Phi^1, \Phi^2, \ldots, \Phi^n\}$ 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 9.2). Φ' must therefore yield the maximum degree of parallelism among all one-dimensional legal mappings. Given a set of legal, maximally independent, partition mappings $\mathscr{B} = \{\Phi^1, \Phi^2, \ldots, \Phi^n\}$,

$$\exists k_1, \ldots, k_n \geq 0 \forall s \in \mathscr{S} \exists a \in Q$$

$$\Phi'_s(\vec{y}) = k_1 \Phi^1_s(\vec{y}) + k_2 \Phi^2_s(\vec{y}) + \cdots + k_n \Phi^n_s(\vec{y}) + a$$

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

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

Since $(m - k_i)$ is non-negative, $((m - k_i)\Phi^i)$ is a legal partition mapping (Lemma 9.1). 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 9.3, since Φ and $\hat{\Phi}$ are linearly dependent, they have the same maximal degree of parallelism. □

Theorem 9.5. *Algorithm 9.2 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 9.1 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 9.4, Step 1 of Algorithm 9.2 finds a legal mapping that creates partitions with the largest degree of parallelism in P. By Theorem 8.3, Step 2 finds all the parallelism that uses at most $O(1)$ synchronizations in each partition. Therefore, Algorithm 9.2 finds the maximum degree of parallelism that requires at most one degree of synchronization. □

9.4. Exploiting pipeline parallelism

When there are several linearly independent solutions to the time-partition constraints, the partition mapping chosen in Algorithm 9.2 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 9.2, 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 9.4. A multi-dimensional affine partition mapping Φ for a program P is *pipelinable* if and only if each row of the mapping is a legal mapping for P. That is, let R be the data dependence set for P

$$\begin{aligned}\forall\langle \mathscr{F}_{zsr}, \mathscr{F}_{zs'r'}\rangle \in R, \vec{i} \in \mathscr{Z}^{\delta_s}, \vec{i} \in \mathscr{Z}^{\delta_{s'}} \text{ s.t. } \vec{i} \prec_{ss'} \vec{i'} \wedge \mathscr{D}_s(\vec{i}) \\ \geq \vec{0} \wedge \mathscr{D}_{s'}(\vec{i'}) \geq \vec{0} \wedge \mathscr{F}_{zsr}(\vec{i}) - \mathscr{F}_{zs'r'}(\vec{i'}) \\ = \vec{0},\ \Phi_{s'}(\vec{i'}) - \Phi_s(\vec{i}) \geq \vec{0}\end{aligned}$$

Algorithm 9.3. 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 9.1 to program P to get a set of maximally independent partition mappings $\mathscr{B} = \{\Phi^1, \ldots, \Phi^m\}$. Let Φ be an m-dimensional pipelinable partition mapping whose rows are the mappings in $\mathscr{B}$.

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 $(p_1 - 1, \ldots, p_{m-1}), \ldots, (p_1, \ldots, p_{m-1} - 1)$ before each time partition, and its $m-1$ neighbors $(p_1 + 1, \ldots, p_{m-1}), \ldots, (p_1, \ldots, p_{m-1} + 1)$ after each time partition.

Step 3. Apply Algorithm 8.1 to each partition of Φ to find all the degrees of parallelism requiring at most $O(1)$ synchronizations.

Lemma 9.6. *Algorithm 9.3 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 $p'=(p'_1,\ldots,p'_m)$, $\forall p'_i \le p_i$, whenever $p \neq p'$. By Definition 9.4 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 9.7. *Algorithm 9.3 finds the maximum degree of parallelism with at most one degree of synchronization in an outer sequential loop P.*

Proof. Let $\mathscr{B}=\{\Phi^1,\ldots,\Phi^m\}$ be the set of legal, maximally independent partition mappings found in Step 1 of Algorithm 9.3, 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 9.3. Let $\Phi'=\sum_{\Phi^i\in\mathscr{B}}\Phi^i$, 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 must $\hat{\Phi}$ 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 9.4 and 7.3, the algorithm finds all the parallelism using $O(n)$ synchronizations.

By Lemma 9.6, 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 9.3 must also find the maximum degree of parallelism with $O(n)$ synchronizations. □

9.5. Example

Consider the following example of an outer sequential loop:

```
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 l'3=l2+1 to n − 1 do
    a[l1,l'3]-=b[n − 1 − l2,l'3]∗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 parallelization algorithms would parallelize this code. Being able to handle only perfectly nested loops, algorithms based on unimodular transforms can only parallelize the innermost loops, l_3 and l'_3. Given that the number of iterations in loops l_3 and l'_3 are different, algorithms based on general loop transformations would not attempt to fuse the two loops and would again parallelize only the innermost loops. Affine scheduling would map each of the instructions to a 2-dimensional time, giving one degree of innermost parallelism. Thus, previous approaches would find only one degree of parallelism within 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 $\vec{l} = [l_1 \quad l_2 \quad l_3]^T$ and $\vec{l'} = [l_1 \quad l_2 \quad l'_3]^T$. When applied to this example, Step 1 of Algorithm 9.3 finds two legal, independent, affine time–partition mappings:

$$\Phi^1 = \left\{ \Phi_1^1(\vec{l}) = l_1, \Phi_2^1(\vec{l'}) = l_1 \right\}$$

$$\Phi^2 = \left\{ \Phi_1^2(\vec{l}) = l_2 + l_3, \Phi_2^2(\vec{l'}) = l'_3 \right\}$$

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)
```

The computation assigned to processor p is simply the pth iteration of the outermost loop. The execution order on each processor is illustrated in Fig. 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.

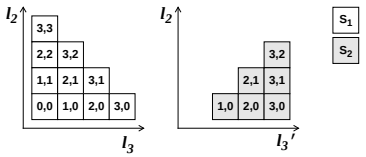

Fig. 3. Execution order for each processor in the example.

The labels on these operations indicate their relative execution 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.

10. 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 10.1. 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 7.3 to the program.

Step 2. Find the maximum degree of parallelism that requires $O(1)$ synchronizations: Apply Algorithm 8.1 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 9.2 or Algorithm 9.3 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 10.1. *Algorithm 10.1 finds all the degrees of parallelism in a program, with all the parallelism being as coarse-grain as possible.*

Proof. By Lemmas 7.3, 8.2, 9.4 and Theorem 9.7, the program maximizes the degree of parallelism in a program. By Theorems 7.2, 8.3, 9.5, and 9.7, 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. □

The worst-case complexity of the algorithm is exponential as the algorithm uses techniques such as Fourier-Motzkin Elimination. We plan to develop efficient algorithms based on this fundamental theory that take advantage of the typical simplicity found in programs in practice.

11. Conclusion

This paper presents an algorithm to find the optimal affine mapping that maximizes the degree of parallelism while minimizing the degree of synchronization. Our algorithm is powerful, as affine partitioning encompasses many previously proposed program transformations including loop distribution, fusion, scaling, reindexing, unimodular transforms, and statement reordering.

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 is a 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. Furthermore, we show that finding these solutions 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, which has better locality and less synchronization overhead.

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 $m \times n$ be the size of matrix A, and $a[b]$ denotes the bth component of a.

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

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. $k_{r_0} M[r_0][c'{:}m] + \ldots + k'_{r-1} M[r'-1][c'{:}m] \geq 0$;
If $\exists$ a non-trivial solution, say $k_r > 0$, then
$M[r] := k_{r_0} M[r_0] + \ldots + k_{r'-1} M[r'-1]$ *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] ≥ 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 ∧ r < r_0.
                u := ⌈(−M[row][col]/M[r][col])⌉;
                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 ∨ (r_n > n) ∨ (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 ∄ 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;
```

References

[1] A. Schrijver, Theory of Linear and Integer Programming, Wiley, Chichester, 1986.
[2] D. Bacon, S. Graham, O. Sharp, Compiler transformations for high-performance computing, Computing Surveys 26 (4) (1994) 345–420.

[3] J.R. Allen, D. Callahan, K. Kennedy, Automatic decomposition of scientific programs for parallel execution, in: Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, 1987, pp. 63–76.
[4] U. Banerjee, Loop Transformations for Restructuring Compilers, Kluwer Academic, 1993.
[5] K. Kennedy, K.S. McKinley, Optimizing for parallelism and data locality, in: Proceedings of the 1992 ACM International Conference on Supercomputing, 1992, pp. 323–334.
[6] D.J. Kuck, R.H. Kuhn, D. Padua, B. Leasure, M. Wolfe, Dependence graphs and compiler optimizations, in: Conference Record of the Eighth ACM Symposium on Principles of Programming Languages, 1981, pp. 207–218.
[7] D. Padua, M. Wolfe, Advanced compiler optimizations for supercomputers, Commun. ACM 29 (1986) 1184–1201.
[8] M.J. Wolfe, Optimizing Supercompilers for Supercomputers, MIT Press, Cambridge, MA, 1989.
[9] K.S. McKinley, Evaluating automatic parallelization for efficient execution on shared-memory multiprocessors, in: Proceedings of the 1994 ACM International Conference on Supercomputing, 1994.
[10] K. Smith, B. Appelbe, Determining transformation sequences for loop parallelization, in: Proceedings of the Fifth Workshop on Programming Languages and Compilers for Parallel Computing, 1992.
[11] D. Whitfield, M.L. Soffa, Investigating properties of code transformations, in: Proceedings of the 1993 International Conference on Parallel Processing, ACM, 1993.
[12] D.I. Moldovan, On the analysis and synthesis of vlsi algorithms, Trans. Comput. 31 (1982) 1121–1125.
[13] P. Quinton, Automatic synthesis of systolic arrays from uniform recurrent equations, in: Proceedings of the Eleventh Symposium on Computer Architecture, 1984.
[14] U. Banerjee, Unimodular transformations of double loops, in: Proceedings of the Third Workshop on Programming Languages and Compilers for Parallel Computing, 1990, pp. 192–219.
[15] M.E. Wolf, M.S. Lam, A loop transformation theory and an algorithm to maximize parallelism, Trans. Parallel Distributed Systems 2 (1991) 452–470.
[16] A. Darte, Y. Robert, Affine-by-statement scheduling of uniform loop nests over parametric domains. Technical Report 92-16, Laboratoire de l'Informatique du Parallélisme, 1992.
[17] P. Feautrier. Some efficient solutions to the affine scheduling problem, part i, one-dimensional time. Technical Report 92.28, Institut Blaise Pascal/Laboratoire MASI, 1992.
[18] P. Feautrier, Some efficient solutions to the affine scheduling problem: Part II. Multi-Dimensional Time, Technical Report 92.78, Institut Blaise Pascal/Laboratoire MASI, 1992.
[19] P. Feautrier, Some efficient solutions to the affine scheduling problem: Part I. One-dimensional time, Int. J. Parallel Programming 21 (5) (1992) 313–348.
[20] P. Feautrier, Some efficient solutions to the affine scheduling problem: Part II. Multidimensional time, Int. J. Parallel Programming 21 (6) (1992) .
[21] P. Feautrier, Towards automatic distribution, Technical Report 92.95, Institut Blaise Pascal/Laboratoire MASI, December 1992.
[22] A. Darte, G. Silber, F. Vivien, Combining retiming and scheduling techniques for loop parallelization and loop tiling. Technical Report 96-34, Laboratoire de l'Informatique du Parallélisme, 1996.
[23] W. Kelly, W. Pugh, Minimizing communication while preserving parallelism, in: Proceedings of the 1996 ACM International Conference on Supercomputing, 1996, pp. 52–60.
[24] C. Ancourt, F. Irigoin, Scanning polyhedra with DO loops, in: Proceedings of the Third ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1991, pp. 39–50.
[25] A.W. Lim, M.S. Lam, Communication-free parallelization via affine transformations, in: Proceedings of the Seventh Workshop on Programming Languages and Compilers for Parallel Computing, Springer, 1994, pp. 92–106.
[26] A.W. Lim, M.S. Lam, Maximizing parallelism and minimizing synchronization with affine transforms, in: Conference Record of the 24th Annual ACM Symposium on Principles of Programming Languages, 1997.