# Computation of Uniform Recurrence Equations over Finite Domains*

William J. Aldrich and P. R. Kumar†

**Abstract**

Uniform recurrence equations arise frequently in scientific problems. We examine the problem of evaluating a given set of recurrence functions on an arbitrary finite set of index points. We provide a necessary and sufficient condition to determine when the equations can be grouped by index points into batch sets, for all finite domains, such that the batch sets can be sequentially evaluated. We show how the batch sets can be ordered using the hyperplane method. Next we provide a necessary and sufficient condition to determine when one can "index shift" a set of recurrence equations so that it is batch computable for all finite domains. Finally, we provide an algorithm for performing an index shifting that enables all of the computations within a batch to be performed in parallel.

**Index terms:** Finite Domains, Hyperplane Method, Parallel Computation, Scientific Computation, Uniform Recurrence Equations

*The research reported here has been supported in part by the U.S. Army Research Office under Contract No. DAAL-03-91-G-0182, in part by the NSF under Contract No. ECS-92-16487, and in part by the JSEP under Contract No. N00014-90-J-1270.

†University of Illinois, Department of Electrical and Computer Engineering, and the Coordinated Science Laboratory, 1308 West Main Street, Urbana, IL 61801/USA.

# 1 Introduction

Scientific problems frequently involve the computations of variables, specified by uniform recurrence equations, over a domain of index points. Consider the system,

$$\begin{aligned} x_1(\boldsymbol{p}-\boldsymbol{o}_1) &= f_1(x_1(\boldsymbol{p}-\boldsymbol{d}_{11}),\cdots,x_n(\boldsymbol{p}-\boldsymbol{d}_{1n})),\\ &\vdots\\ x_i(\boldsymbol{p}-\boldsymbol{o}_i) &= f_i(x_1(\boldsymbol{p}-\boldsymbol{d}_{i1}),\cdots,x_n(\boldsymbol{p}-\boldsymbol{d}_{in})),\\ &\vdots\\ x_n(\boldsymbol{p}-\boldsymbol{o}_n) &= f_n(x_1(\boldsymbol{p}-\boldsymbol{d}_{n1}),\cdots,x_n(\boldsymbol{p}-\boldsymbol{d}_{nn})). \end{aligned} \tag{1}$$

Above, the $q$-dimensional vector $\boldsymbol{p}=(i,j,k,\cdots)$ is an *index point*. This system of recurrence equations is "uniform" since the vector differences between the variable indices of the LHS and RHS remain constant. (The equation $x(i,j)=f(x(2*i,j))$ is not uniform.) We are required to determine the values $\{x_1(\boldsymbol{p}-\boldsymbol{o}_1),\cdots,x_n(\boldsymbol{p}-\boldsymbol{o}_n)\}$ for all $\boldsymbol{p}$ in a given *domain* $\mathcal{D}$. We assume that the values of the variables $x_i(\boldsymbol{p}-\boldsymbol{o}_i)$ for all $\boldsymbol{p}$ outside the domain $\mathcal{D}$ are given as *boundary conditions*.

Consider the following example.

## Example 1:

Compute

$$x_1(i,j)=f_1(x_1(i,j+1),x_2(i-1,j))$$
$$x_2(i,j)=f_2(x_1(i+1,j-1),x_2(i-1,j-1))$$

on the domain

$$\mathcal{D}:=\{(i,j):0\le i\le 2,0\le j\le 2\},$$

given as boundary conditions, the values of $x_1(i,j)$ and $x_2(i,j)$ for all $(i,j)$ outside the domain $\mathcal{D}$. □

However, this problem is *not well-posed*. To see this, note that to determine $x_2(0,1)$ from the function $f_2$ we require the value of $x_1(1,0)$, which can be determined from $f_1$ if we know the value of $x_1(1,1)$. However, we cannot determine the value of $x_1(1,1)$ from $f_1$ unless we know $x_2(0,1)$.

Consider now the following well-posed problem.

## Example 2:

Compute

$$x_1(i,j) = f_1(x_2(i,j))$$
$$x_2(i+1,j) = f_2(x_2(i,j+1), x_1(i,j-1))$$

for all $(i,j)$ in the domain

$$\mathcal{D} := \{(i,j) : 1 \leq i \leq 3, 1 \leq j \leq 2\},$$

given the values $\{x_1(i,j), x_2(i+1,j)\}$ for all $(i,j)$ outside $\mathcal{D}$.

One can organize these computations so as to obtain the following *computable form.*

## Computable Form for Example 2:

for i = 1 to 3, Do:
    for j = 1 to 2 Do:
        $x_1(i,j) = f_1(x_2(i,j))$
        $x_2(i+1,j) = f_2(x_2(i,j+1), x_1(i,j-1))$
    end
end.

This computable form has certain special features. First we have ordered the *index points* in the domain $\mathcal{D} = \{(i,j) : 1 \leq i \leq 3, 1 \leq j \leq 2\}$, and for each index point evaluated both recurrence equations. This ability to group computations by index point will be referred to as *batch computability.*

A further advantageous feature is that this particular organization also allows both $x_1(i,j)$ and $x_2(i+1,j)$ to be computed simultaneously since neither variable depends on the other. We will refer to this parallelism of computations at an index point as *batch parallelism.*

Finally, yet another type of parallelism is *index parallelism*, which is the ability to simultaneously perform the computations for several index points. We will illustrate this in Section 4.

In this paper we determine a necessary and sufficient condition when the equations can be grouped by index points, for *all finite domains*, into batch sets, in such a way that the resulting batch sets can be sequentially computed. We also show how the batch sets can be ordered using the hyperplane method, which also allows some index parallelism. We provide a necessary and sufficient condition to determine when one can "index shift" a set of recurrence equations so that it is batch computable for all finite domains. Finally, we provide an algorithm for performing such index shifting in such a way that one can also perform all the computations within a batch in parallel.

# 2 History

Karp, Miller and Winograd [1] have considered systems of uniform recurrence equations, deriving tests for computability and parallelism. The most notable difference in the work presented here is the domain of computation, $\mathcal{D}$. In [1], the authors restrict their attention to infinite domains in which all of the indices were positive. By using infinite domains, certain index orderings are implicitly disallowed. Consider the one-dimensional domain $\mathcal{D} = \{i \in \mathcal{Z} : i \geq 0\}$. It is not possible to arrange the set in decreasing order. However, this would be possible for any finite set. For this reason, and the fact that all real computational problems are finite, we have considered only finite domains.

Parallelism for computation problems has been studied extensively and remains an active research area. The motivation for much of this work has been to develop vectorizing and parallelizing compilers for high-level languages such as FORTRAN. In this context, recurrence equations appear within nested loops, and the dependences are limited to lexicographically positive vectors, a considerable restriction. For example, both of the equations

$$x_1(i,j) = f_1(x_2(i+1,j))$$
$$x_2(i,j) = f_1(x_1(i,j+1))$$

would violate the lexicographic requirement since their dependence vectors, $(-1,0)$ and $(0,-1)$, are both lexicographically negative.

One of the simplest techniques developed for extracting parallelism from a loop has been *loop vectorization*, discussed by Muraoka in [2] and Padua and Wolfe in [3]. In this technique, all iterations of the loop are executed simultaneously provided that there are no dependence cycles within the body of the loop. The method has been shown to be extendable to nested loops by separately testing each loop dimension for dependence cycles starting with the innermost loop; see Wolfe [4]. The loop dimensions are successively vectorized until an index is found with a dependence cycle. Loop vectorization is particularly useful because it can be applied to nonuniform recurrences; however, more elaborate dependence analysis is required. Methods for nonuniform dependence analysis were developed by Banerjee in [5], [6].

Loop vectorization can be enhanced by several transformation techniques. *Loop interchange* is the process of interchanging the order of loops. This can enable vectorization if the innermost loop is not vectorizable, but another more outer loop is; see [4]. If all loops have dependence cycles, parallelism can sometimes be realized through *loop skewing*, the process of rotating coordinate vectors in the index space. Both of these transformations have been represented as unimodular transformations on the set of index points. Methods for finding optimal unimodular transformations are presented by Wolf and Lam in [7].

When loops cannot be completely vectorized, some partial parallelism can often be realized through *cycle shrinking*. This is a technique to perform a certain number of loop

iterations simultaneously, provided that there are no dependences between adjacent iterations. This technique is described by Polychronopoulos in [8], in which a number of different approaches are described for nested loops.

The more specific problem of extracting parallelism from nested loops with uniform recurrences was initially studied by Lamport [9]. His *hyperplane method*, referred to in Section 4, accomplishes the same result as skewing followed by vectorization [4].

More recently, authors have developed techniques to optimize the hyperplane method. Algorithms for finding the optimal scheduling vector, denoted by $\boldsymbol{q}$ in Section 4, were developed by Shang and Fortes [10]. These algorithms find the $\boldsymbol{q}$ vector that enables the hyperplane method to be used with the smallest number of steps. Methods for index shifting to enhance the hyperplane method with a given $\boldsymbol{q}$ vector were developed by Liu, Ho and Sheu in [11]. Methods to optimally combine these two techniques were developed by Robert and Song in [12].

In all these methods, viz., loop vectorization, cycle shrinking, and hyperplane scheduling, the body of the loop nest is executed sequentially. An alternative is *affine by statement* scheduling. This method chooses execution times for each equation separately, by specifying a $\boldsymbol{q}_i$ vector and a time step $t_i$ for the first computation. That is, we would organize the computations in the following framework:

for t = min to max, Do:
    Simultaneously compute all $x_i(\boldsymbol{p})$ such that $\boldsymbol{q}_i^T\boldsymbol{p} + t_i = t$:
        $x_1(\boldsymbol{p}) = f_1(\cdots)$
        $x_2(\boldsymbol{p}) = f_2(\cdots)$
        $\vdots$
end.

Darte, Risset and Robert [13], show how the optimal affine by statement parameters can be found from the solution of a linear program.

The shifting algorithm presented here extends the amount of parallelism using the hyperplane method by assuring that equations *within* a loop iteration can be performed concurrently.

For an excellent survey of parallelization techniques, including the more general case of nonuniform recurrences, we refer the reader to Banerjee et al [14].

# 3 Graph Representations

First we introduce some notation. For convenience we denote the RHS of (1) by $f_i(\boldsymbol{p})$. We will refer to a function, or the value of a variable, at an index point, as an *instance* of the

function or variable, e.g., $f_1(x_2(3,2))$, $x_1(1,2)$. It is convenient to define the set

$$\mathcal{D}_i \quad := \quad \{\boldsymbol{p} : \boldsymbol{p} + \boldsymbol{o}_i \in \mathcal{D}\},$$

noting that the goal is to determine $x_i(\boldsymbol{p})$ for all $\boldsymbol{p} \in \mathcal{D}_i$. At each index point $\boldsymbol{p}$ we group together all variable instances on the LHS of the recurrence equations into a *batch set* $\mathcal{B}(\boldsymbol{p})$. Thus, $\mathcal{B}(\boldsymbol{p}) := \{x_i(\boldsymbol{p}-\boldsymbol{o}_i),\ 1 \leq i \leq n\}$. We define $\mathcal{S}$ as the set of all pairs $(i,j)$ such that $x_j$ is an argument of the function $f_i$. For an equation of the form $x_i(\boldsymbol{p}-\boldsymbol{o}_i) = f_i(\cdot, x_j(\boldsymbol{p}-\boldsymbol{d}_j), \ldots)$ we say that $\boldsymbol{d}_j$ is a *dependence vector* between $x_j$ and $x_i$. We actually allow $x_j$ to either not occur as an argument of $f_i$, occur just once as such as an argument, or more than once. Hence there may be several dependence vectors between $x_j$ and $x_y$. We define a set $\mathcal{L}_{ij}$ for each pair $(i,j) \in \mathcal{S}$, containing all of the dependence vectors between $x_j$ and $x_i$. For example, the equation

$$x_1(\boldsymbol{p}-\boldsymbol{o}_1) \quad = \quad f_1(x_2(\boldsymbol{p}-\boldsymbol{d}_1), \quad x_2(\boldsymbol{p}-\boldsymbol{d}_2), \quad x_1(\boldsymbol{p}-\boldsymbol{d}_3))$$

would have the dependence sets

$$\mathcal{L}_{12} \quad = \quad \{\boldsymbol{d}_1, \boldsymbol{d}_2\}, \quad \mathcal{L}_{11} = \{\boldsymbol{d}_3\}.$$

Finally we introduce a single set of dependence vectors $\mathcal{L}$, containing the elements of all sets $\mathcal{L}_{ij}$, i.e., $\mathcal{L} = \bigcup_{(i,j)\in\mathcal{S}} \mathcal{L}_{ij}$.

We employ two types of graphs, a *dependence graph* and a *recurrence graph*, to represent the direction of data dependence.

We generate a *dependence graph* for a computation problem in the following way. A vertex is associated with each $x_i(\boldsymbol{p}-\boldsymbol{o}_i)$ for $i \in \{1, \cdots, n\}$ and $\boldsymbol{p} \in \mathcal{D}$. If $x_i(\boldsymbol{p})$ is required to determine $x_j(\boldsymbol{q})$, an arc is drawn from vertex $x_i(\boldsymbol{p})$ to vertex $x_j(\boldsymbol{q})$. The dependence graph for Example 2 is shown in Figure 1.

At some index points, the recurrence equations depend on values of variables from outside the index domain $\mathcal{D}$. These dependences are ignored in the graph, since we assign a vertex only to a variable instance that has to be computed. We assume that the values of all variables are available outside of the computation domain.

In the *recurrence graph*, a single vertex represents each variable, $x_i$, making this graph independent of the index domain. If $x_i(\mathbf{p}-\mathbf{o}_i) = f_i(\cdots, x_j(\boldsymbol{p}-\boldsymbol{o}_j-\boldsymbol{\delta}), \cdots)$, then a directed arc is drawn from vertex $x_j$ to $x_i$ and labeled with the vector $\boldsymbol{\delta}$. Here we extend the definition of a digraph in [15] to allow for multiple arcs from one vertex to another. We also allow multiple loops (i.e., arcs directed towards the same vertex from which they originated). The recurrence graph for Example 2 is shown in Figure 2.

The usefulness of graph theory for analyzing computations is based on the following lemma.

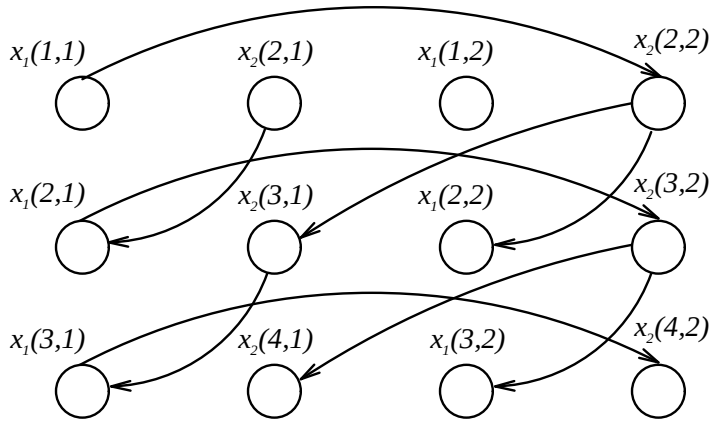


Figure 1: Dependence Graph for Example 2.

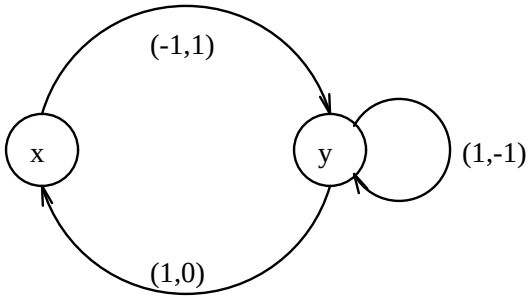


Figure 2: Recurrence Graph for Example 2

**Lemma 1.** *A digraph has an integer labeling of its vertices such that each arc is directed towards a vertex with a larger label than its source vertex if and only if the digraph has no cycles.*

**Proof**
Necessity is obvious since any two vertices labeled $a$ and $b$, contained in a cycle, would require that $a < b$ due to the path from $a$ to $b$, and $b < a$ due to the path from $b$ to $a$. Sufficiency is shown by labeling with 0 a vertex in the graph with no arcs directed towards it. Such a vertex exists for all acyclic graphs [15]. Removing this vertex we form the subgraph induced by the remaining vertices. The subgraph will also be acyclic. We label with 1 a vertex on the subgraph that has no arcs directed towards it. By proceeding inductively as described we generate an appropriate labeling. □

The following corollary characterizes when a given computational problem is well-posed.

**Corollary 1.** *A given computational problem is well-posed if and only if the dependence graph for the problem has no cycles.*

**Lemma 2.** *A system of uniform recurrence equations is computable on all finite domains if and only if the recurrence graph has no closed walks with the sum of labelings zero.*

**Proof**
We first prove sufficiency. If there were some domain in which the set of recurrence equations were not well-posed, the dependence graph for this domain would have a cycle. Each arc in this cycle corresponds to a dependence and, therefore, to some arc in the recurrence graph. Since each vertex of the dependence graph is an instance of some variable, there is also a corresponding vertex on the recurrence graph. We notice that this correspondence is such that every walk on the dependence graph has a corresponding walk on the recurrence graph. Now since a cycle on the dependence graph starts and ends in the same batch set, by the way the recurrence graph is labeled, the sum around its corresponding walk must be zero. Since the cycle on the dependence graph starts and ends with the same vertex, the walk on the recurrence graph must also be closed.

To show necessity, consider a computational problem where the recurrence graph has a closed walk with a sum of labelings equal to zero. We can choose a domain $\mathcal{D}$ such that starting at the point $\boldsymbol{p} = 0$ and repetitively moving a distance equal to the labelings of the recurrence graph in the closed walk, we always remain within the domain. On the dependence graph for such a domain, there is a corresponding walk. Since the sum of labelings is zero, the walk on the dependence graph must start and end in the same batch set. Since the walk on the recurrence graph is closed, the corresponding walk on the dependence graph must start and end with the same variable, so it is closed. A closed walk in turn implies a cycle [15]. □

# 4 Batch Computability

We now introduce a more restrictive property than well-posedness that will lead to desirable scheduling characteristics. We refer to this as *batch computability*, as defined below.

**Definition.** *Given a system of uniform recurrence equations and a domain $\mathcal{D}$, we say that the system is* **batch computable** *if there exists an ordering $\{\boldsymbol{p}_1, \boldsymbol{p}_2, \cdots\}$ of $\mathcal{D}$ such that one can compute $\mathcal{B}(\boldsymbol{p}_1) = \{x_i(\boldsymbol{p}_1 - \boldsymbol{o}_i) : 1 \leq i \leq n\}$ then $\mathcal{B}(\boldsymbol{p}_2) = \{x_i(\boldsymbol{p}_2 - \boldsymbol{o}_i) : 1 \leq i \leq n\}$, etc. That is, to compute $\mathcal{B}(\boldsymbol{p}_t)$ it suffices to know only the values $\{x_i(\boldsymbol{p}_k - \boldsymbol{o}_i) : 1 \leq i \leq n, \text{and } k < t\}$ and $\{x_i(\mathbf{p} - \mathbf{o}_i) : \boldsymbol{p} \notin \mathcal{D}\}$.*

We will describe the computation ordering $\{\boldsymbol{p}_1, \boldsymbol{p}_2, \cdots\}$ as *consecutive* by batch set. Note that batch computability implies that we can evaluate *all* of the recurrence functions at *each* index point before evaluating the functions at other index points.

Since we would like to organize equations consecutively by batch set in a batch computable problem, we must understand the dependences among batch sets. Conceptually, we could represent this with a function $\Phi(\cdot)$, which serves as a uniform recurrence equation for the entire batch set,

$$\mathcal{B}(\boldsymbol{p}) = \Phi(\mathcal{B}(\boldsymbol{p} - \boldsymbol{v}_1), \mathcal{B}(\boldsymbol{p} - \boldsymbol{v}_2), \cdots). \tag{2}$$

It follows that for $\mathcal{B}(\boldsymbol{p} - \boldsymbol{v}_k)$ to occur in the RHS of (2) it is necessary and sufficient that for some $i$, $j$,

$$x_i(\mathbf{p} - \mathbf{o}_i) = f_i(\cdots, x_j(\boldsymbol{p} - \boldsymbol{o}_j - \boldsymbol{v}_k), \cdots), \text{ where } \boldsymbol{v}_k \neq 0. \tag{3}$$

Hence, if $\mathcal{V} = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \cdots\}$, then

$$\mathcal{V} \triangleq \{\boldsymbol{d} - \boldsymbol{o}_j : \boldsymbol{d} \in \mathcal{D}_{ij}, \boldsymbol{d} \neq \boldsymbol{o}_j, (i,j) \in \mathcal{S}\}. \tag{4}$$

We refer to the vectors $\boldsymbol{v}_i \in \mathcal{V}$ as *batch dependence vectors.* These are simply the nonzero labels of the recurrence graph. As a uniform recurrence equation, $\Phi(\cdot)$ will also have its own recurrence graph, and for each domain, a dependence graph. We will distinguish these graphs as the *recurrence graph for* $\Phi(\cdot)$, and the *dependence graph for* $\Phi(\cdot)$. In Figure 3 below, we have drawn these graphs for Example 2. The recurrence graph for $\Phi(\cdot)$ has only one vertex; thus all of its arcs are loops.

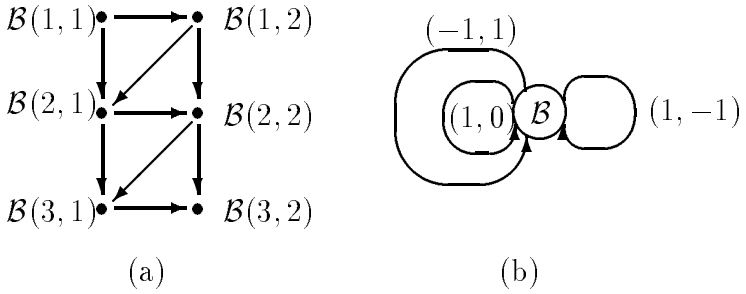


Figure 3: Graph Representations of $\Phi(\cdot)$ for Example 2; (a) Dependence Graph for $\Phi(\cdot)$, (b) Recurrence Graph for $\Phi(\cdot)$.

Note that an arc from $x_j(\boldsymbol{p} - \boldsymbol{o}_j)$ to $x_i(\boldsymbol{p} - \boldsymbol{o}_i)$ in the original dependence graph would correspond to an *inside batch dependency* for $\mathcal{B}(\boldsymbol{p})$. Such arcs have been excluded from $\mathcal{V}$. However, they still have to be considered when testing for batch computability. To capture this, we draw the dependence graph for a single batch $\mathcal{B}(\boldsymbol{p})$. That is, we consider the vertices $\{x_1(\boldsymbol{p} - \boldsymbol{o}_1), x_2(\boldsymbol{p} - \boldsymbol{o}_2), \cdots\}$ and the graph induced on these vertices from the original dependence graph. We call this the *inside batch dependence graph.* We note that the structure of the graph does not depend on $\boldsymbol{p}$. In Figure 4 below, the inside batch dependence graph is drawn for Example 2. This graph has no arcs because there are no inside batch dependencies.


Figure 4: Inside Batch Dependence Graph for Example 2.

## 4.1 Conditions for Batch Computability

We now characterize when a computation problem is batch computable over *all* finite domains.

**Theorem 1.** *A system of uniform recurrence equations is batch computable over all finite domains $\mathcal{D}$ if and only if both of the following two properties hold:*

*1.) The inside batch dependence graph for some $\mathcal{B}(\boldsymbol{p})$ has no cycles.*

*2.) There are no semipositive integer vectors $\boldsymbol{z}$ in the null space of*

$$\boldsymbol{A} \triangleq [\boldsymbol{v}_1, \boldsymbol{v}_2, \cdots], \boldsymbol{v}_i \in \mathcal{V}.$$

*A* **semipositive** *vector is a vector with every element greater than or equal to zero, and at least one element strictly greater than zero.*

**Proof of necessity**
Property *1.)* is clearly necessary since the inside batch dependence graph for $\mathcal{B}(\boldsymbol{p})$ is a subgraph of the dependence graph for the complete computational problem. To show that property *2.)* is necessary, assume that there is a semipositive integer vector $\boldsymbol{z}$ in the null space of $\boldsymbol{A}$. Since the vectors $\boldsymbol{v}_i$ that comprise $\boldsymbol{A}$ are all nonzero, the vector $\boldsymbol{z}$ will of necessity have at least two elements $z_i$ greater than zero. Now consider a hyperrectangular domain $\mathcal{D}$, such that

$$\boldsymbol{q} = \sum_{i=1}^{n} z_i |\boldsymbol{v}_i| \in \mathcal{D}, \tag{5}$$

and

$$\boldsymbol{r} = -\sum_{i=1}^{n} z_i |\boldsymbol{v}_i| \in \mathcal{D}. \tag{6}$$

In the dependence graph for $\Phi(\cdot)$ defined over $\mathcal{D}$, we can generate a walk by starting at 0 and consecutively taking the arc associated with $\boldsymbol{v}_i$ for $z_i$ times. By our choice of $\mathcal{D}$ we will always remain on the dependence graph. The sum of arc weights will be zero since $\boldsymbol{z}$ is a member of the null space of $\boldsymbol{A}$. This implies that the starting vertex and ending vertex are the same, and, therefore, a cycle exists. By Lemma 1 the existence of a cycle in the dependence graph for $\Phi(\cdot)$ is sufficient for the nonexistence of a batch set ordering. Therefore, the system of recurrence equations would not be batch computable on this domain.

**Proof of sufficiency**
Due to the uniformity of the recurrence equations, the inside batch dependence graphs for $\mathcal{B}(\boldsymbol{p})$ are isomorphic for differing $\boldsymbol{p}$. Hence, property *1.)* is equivalent to every batch set having an ordering that satisfies the definition of computability. Since the recurrence graph for $\Phi(\cdot)$ has only one vertex, all of its arcs are loops and any sequence of these arcs forms a closed walk. If we let $z_i$ equal the number of times the arc with label $\boldsymbol{v}_i$ is taken in a walk, the sum of the labels will be $\boldsymbol{Az}$. We can now see that property *2.)* is equivalent to no closed walks on the recurrence graph for $\Phi(\cdot)$ having a sum of labelings zero.

By Lemma 2, we know that an ordering of the batch sets exists such that each batch set is dependent only on earlier sets, for any arbitrary domain. Now if we take this ordering of batch sets and replace each set with an ordering of its elements satisfying the definition of computability, we obtain an ordering of all of the equations that satisfies the definition of batch computability. □

## 4.2 Scheduling

Theorem 1 characterizes when there exists a valid "schedule," i.e., when the domain $\mathcal{D}$ can be ordered by batch set as $\{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \}$ and when the equations within each batch set can be ordered in a computable fashion. We now show how a valid schedule can be found for any batch computable system of recurrence equations.

We start by extending the conditions of Theorem 1 to the more general case in which $\boldsymbol{z}$ is any real vector. Suppose that there is a semipositive real vector $\boldsymbol{z}'$ in the null space of $\boldsymbol{A}$. We can form a subvector of $\boldsymbol{z}'$ by removing all of the zero elements to form a vector $\boldsymbol{x}$, and similarly the associated columns from $\boldsymbol{A}$ to form $\boldsymbol{A}'$, making[1]

$$\boldsymbol{A}'\boldsymbol{x} = 0, \quad \boldsymbol{x} > 0. \tag{7}$$

Since $\boldsymbol{A}'$ has all integer elements, its null space is spanned by a set of rational vectors, $\{\mathbf{e}_i\}$. A solution to (7) can now be found from a vector $\mathbf{f}$, satisfying

$$[\mathbf{e_1}, \mathbf{e_2}, \cdots]\mathbf{f} > 0. \tag{8}$$

[1] We use the following definitions for relational operators on $R^n$:

$\boldsymbol{x} > \mathbf{y} \Rightarrow x_i > y_i \forall i$
$\boldsymbol{x} \stackrel{\geq}{=} \mathbf{y} \Rightarrow x_i \geq y_i \forall i$
$\boldsymbol{x} \geq \mathbf{y} \Rightarrow x_i \geq y_i \forall i$, and $x_i > y_i$ for at least one $i$.

Thus, if $\boldsymbol{x}$ is a semipositive vector, then $\boldsymbol{x} \geq 0$.

Since (8) defines an open set in $R^n$, there exists a rational $\mathbf{f}$. We can therefore find a rational $\boldsymbol{x} = [\mathbf{e_1}, \mathbf{e_2}, \cdots]\mathbf{f} > 0$ and a corresponding semipositive rational vector, $\boldsymbol{z}_R$, in the null space of $\boldsymbol{A}$. By multiplying $\boldsymbol{z}_R$ by the least common factor of its denominators, we obtain a semipositive integer vector $\boldsymbol{z}$ in the null space of $\boldsymbol{A}$. Therefore, the existence of a semipositive real vector $\boldsymbol{z}'$ in the null space of $\boldsymbol{A}$ is sufficient for the system not being batch computable on some domain.

By extending the field of $\boldsymbol{z}$ to real numbers, the semipositivity condition for $\boldsymbol{z}$ can be equivalently replaced by

$$[1, 1, \cdots]\boldsymbol{z} > 0, \quad \boldsymbol{z} \stackrel{\geq}{=} 0. \tag{9}$$

We can now see that property *2.)* is equivalent to the conditions

$$\begin{aligned} [1, 1, \cdots]\boldsymbol{z} &> 0, \\ \mathbf{I}\boldsymbol{z} &\stackrel{\geq}{=} 0, \\ \boldsymbol{A}\boldsymbol{z} &= 0, \end{aligned} \tag{10}$$

having no feasible solution. We now present Motzkin's Theorem of the Alternative, which shows the equivalence of infeasibility in one system of linear inequalities to feasibility in another. A proof can be found in [16].

**Lemma 3.** *Motzkin's Theorem of the Alternative. Let $A$, $C$, and $D$ be matrices of order $m_1 \times n$, $m_2 \times n$, and $m_3 \times n$, respectively. One of the following two systems has a feasible solution and the other is inconsistent:*

**System 1**

$$\begin{aligned} Ax &> 0 \\ Cx &\stackrel{\geq}{=} 0 \\ Dx &= 0 \end{aligned}$$

**System 2**

$$\pi A + \mu C + \gamma D = 0 \tag{11}$$

$$\pi \geq 0 \tag{12}$$

$$\mu \stackrel{\geq}{=} 0. \tag{13}$$

Applying this lemma to Theorem 1 we see that property *2.)* is equivalent to the linear program

$$\begin{aligned} \pi[1, 1, \cdots] + \mu\mathbf{I} + \gamma\boldsymbol{A} &= 0, \\ \pi &> 0, \\ \mu &\stackrel{\geq}{=} 0, \end{aligned} \tag{14}$$

having a feasible solution. One should note that $\pi$ is required to be a semipositive $1 \times 1$ vector, which is simply a strictly positive scalar. If we let $\beta = \pi[1, 1, \cdots] + \mu\mathbf{I}$, and $\boldsymbol{q}^T = -\gamma$ we derive

$$\boldsymbol{q}^T \boldsymbol{A} = \beta, \beta > 0. \tag{15}$$

Since $\boldsymbol{A} \triangleq [\boldsymbol{v}_1, \boldsymbol{v}_2, \cdots]$, this is equivalent to the existence of a vector $\boldsymbol{q}$, having a strictly positive inner product with all batch dependence vectors $\boldsymbol{v}_i$. Geometrically, we say that $\boldsymbol{q}$ forms an acute angle with all batch set dependence vectors.

If we consider the region of points in $R^n$ whose position vector forms an acute angle with some other set of vectors, the region will be a convex cone in $R^n$. Since this is also an open set, it must contain some rational vector, and after scaling, some integer vector. In the remainder of this thesis $\boldsymbol{q}$ will be restricted to integer vectors.

### Ordering the domain

The existence of the vector $\boldsymbol{q}$ permits the index points to be ordered using the hyperplane method of Lamport [9]. We simply order the domain $\mathcal{D}$ in the order of nondecreasing $\boldsymbol{q}^T\boldsymbol{p}$. Thus, if $\boldsymbol{q}^T\boldsymbol{p} > \boldsymbol{q}^T\boldsymbol{p}'$ for $\boldsymbol{p}$, $\boldsymbol{p}' \in \mathcal{D}$, we compute $\mathcal{B}(\boldsymbol{p})$ after $\mathcal{B}(\boldsymbol{p}')$. To see why this works, consider

$$\mathcal{B}(\boldsymbol{p}) = \Phi(\mathcal{B}(\boldsymbol{p} - \boldsymbol{v}_1), \mathcal{B}(\boldsymbol{p} - \boldsymbol{v}_2), \cdots). \tag{16}$$

Since $\boldsymbol{q}^T(\boldsymbol{p} - \boldsymbol{v}_i) = \boldsymbol{q}^T\boldsymbol{p} - \boldsymbol{q}^T\boldsymbol{v}_i < \boldsymbol{q}^T\boldsymbol{p}$, the computation of each $\mathcal{B}(\boldsymbol{p} - \boldsymbol{v}_i)$ is done before $\mathcal{B}(\boldsymbol{p})$, thus assuring that $\Phi(\cdot)$ can be evaluated at point $\boldsymbol{p}$.

With the hyperplane method, we can extract a further degree of parallelism by the simultaneous computation of batches $\mathcal{B}(\boldsymbol{p})$ and $\mathcal{B}(\mathbf{p}')$ if $\boldsymbol{q}^T\boldsymbol{p} = \boldsymbol{q}^T\boldsymbol{p}'$. We refer to this type of parallelism as *index parallelism*, to differentiate it from parallelism within a batch set.

We can now clarify these concepts by Example 3 below. By mere observation it is difficult to tell if the system is batch computable. We cannot nest an $i$ loop within a $j$ loop, or a $j$ loop within an $i$ loop.

## Example 3:

Compute

$$x_1(i,j) = f_1(x_1(i+1, j-1), x_2(i-2, j+1))$$
$$x_2(i,j) = f_2(x_1(i,j), x_2(i-1, j-1))$$

on the domain

$$\{(i,j) : 0 \le i \le 4, 0 \le j \le 4\}.$$

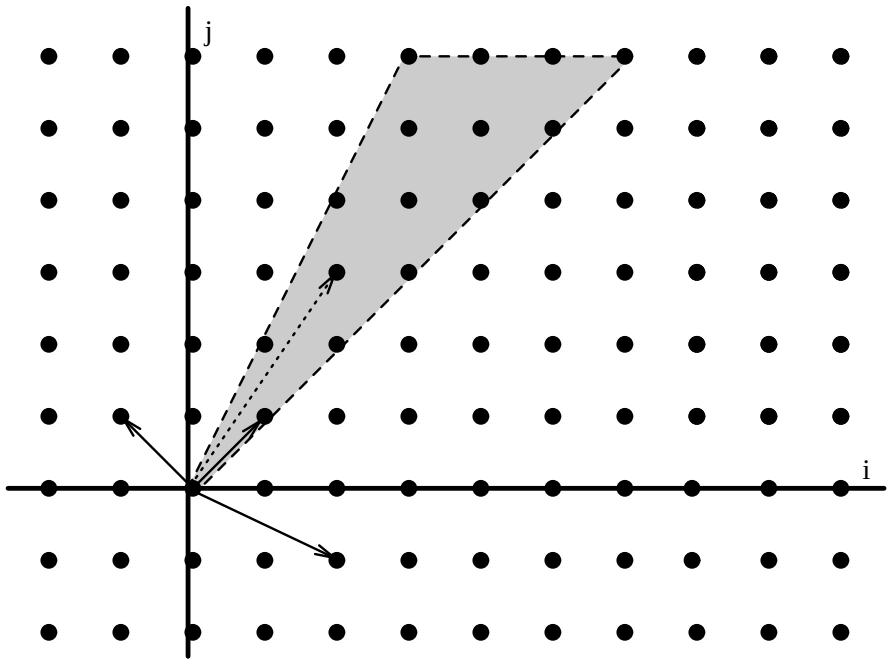


Figure 5: Nonzero Dependence Vectors in Example 3 with the Region of Vectors Forming Acute Angles.

Of the four dependences in this system, three are nonzero. Thus

$$\mathcal{V} = \{(-1, 1), (2, -1), (1, 1)\}.$$

These have been graphed in Figure 5. The shaded region is the set of points whose position vector $\boldsymbol{q}$ forms a strictly positive inner product with all of the vectors in $\mathcal{V}$. It contains the vector $\boldsymbol{q} = (2, 3)$, shown in a dotted line. Since such a region exists, the system of equations is batch computable on any arbitrary domain.

In Figure 6 we show the order in which the index points in the domain are computed when we use the hyperplane method for the vector $\boldsymbol{q} = (2, 3)$. We have labeled each point with the value of $\boldsymbol{q}^T\boldsymbol{p}$. To extract index parallelism, we would simultaneously perform computations at index points with the same value of $\boldsymbol{q}^T\boldsymbol{p}$.

## 4.3 Boundary Conditions

We have assumed so far that all values of $x_i(\boldsymbol{p})$ for $\boldsymbol{p} \notin \mathcal{D}_i$ are available as boundary conditions. Of course, we will require only a subset of these values.

Consider a dependence vector $\boldsymbol{d} \in \mathcal{L}_{ji}$. We require the value of $x_i(\boldsymbol{p})$ for $\{\boldsymbol{p} = \boldsymbol{q} + \boldsymbol{d} : \boldsymbol{q} \in \mathcal{D}\}$. The elements of this set that are not contained in $\mathcal{D}_i$ are part of the boundary

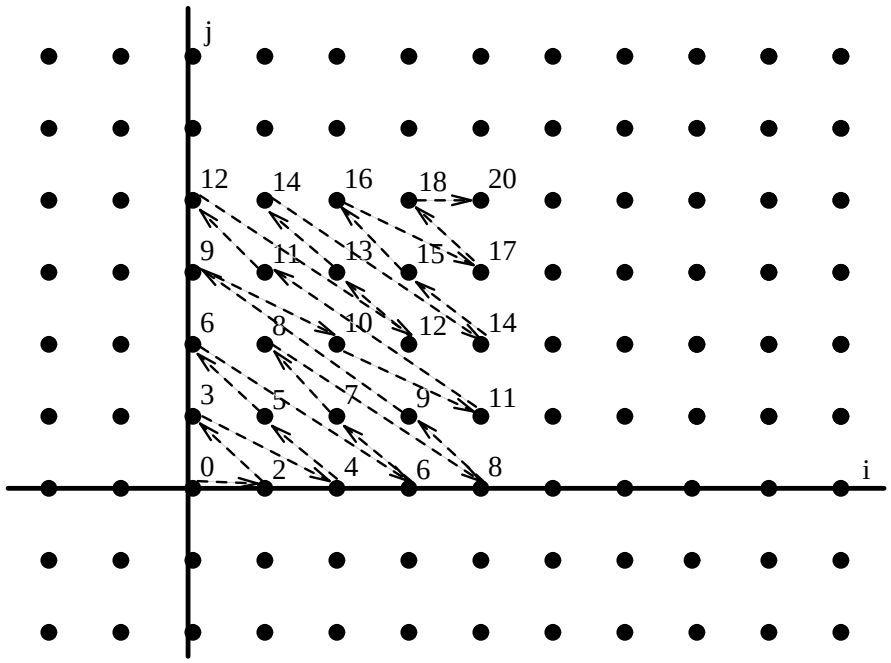


Figure 6: Order of Computation of the Index Points for Example 3, from the Hyperplane Method Using $\mathbf{q} = (2, 3)$.

conditions for $x_j$, that is, $\{\boldsymbol{p} = \boldsymbol{q} + \boldsymbol{d} : \boldsymbol{q} \in \mathcal{D}\} \setminus \mathcal{D}_i$. For each variable $x_i$ we can find a set $\mathcal{C}_i$ of index points in which the values of $x_i$ are needed as boundary conditions, as follows. Let

$$\mathcal{L}_i = \bigcup_{j=1}^{n} \mathcal{L}_{ji}, \tag{17}$$

$$\mathcal{C}_i = \Big( \bigcup_{\boldsymbol{d} \in \mathcal{L}_i} \{\boldsymbol{p} = \boldsymbol{q} + \boldsymbol{d} : \boldsymbol{q} \in \mathcal{D}\} \Big) \setminus \mathcal{D}_i. \tag{18}$$

Revisiting Example 3, we find

$$\begin{array}{rlllll} \mathcal{C}_1 = & \{(1,-1), & (2,-1), & (3,-1), & (4,-1), & (5,-1), \\ & (5,0), & (5,1), & (5,2), & (5,3)\}. & \end{array} \tag{19}$$

and

$$\begin{array}{rlllll} & \{(-2,1), & (-2,2), & (-2,3), & (-2,4), & (-2,5), \\ & (-1,-1), & (-1,1), & (-1,2), & (-1,3), & (-1,4), \\ \mathcal{C}_2 = & (-1,5), & (0,-1), & (0,1), & (0,2), & (0,3), \\ & (0,4), & (0,5), & (1,-1), & (1,5), & (2,-1) \\ & (2,5), & (3,-1), & (4,-1)\}. & & \end{array} \tag{20}$$

Thus the values of $x_i(\boldsymbol{p})$ for $\boldsymbol{p} \in \mathcal{C}_i$ are needed as boundary conditions.

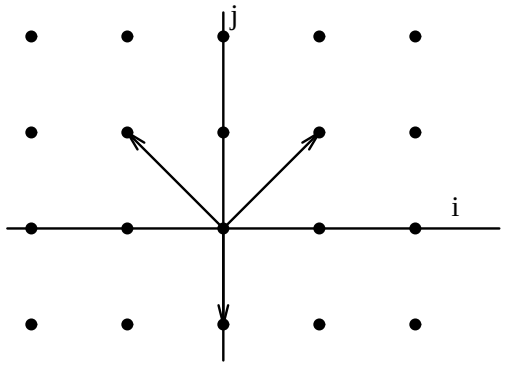


Figure 7: Batch Set Dependence Vectors for Example 4.

# 5 Index Shifting

We now analyze the technique of *index shifting* as a way of transforming a computational problem to another problem that encompasses the same computations.

Consider Example 4 below.

## Example 4:

Compute
$$x_1(i,j) = f_1(x_1(i-1,j-1), x_2(i,j+1))$$
$$x_2(i,j) = f_2(x_2(i+1,j-1))$$
on the domain
$$\{(i,j) : 1 \leq i \leq 5, 1 \leq j \leq 5\}.$$

In Figure 7, the batch set dependence vectors are graphed. Clearly, there is no vector forming an acute angle with all three of these dependences. We therefore know that for some domains this system of equations is not batch computable. The given domain is one such domain.

Now let us suppose that we shifted the $j$ index of the equation for $x_2$ forward by 2. If we expanded the domain of $j$ backwards by 2, we would still encompass all of the original computations. The new problem description would be

Compute
$$x_1(i,j) = f_1(x_1(i-1,j-1), x_2(i,j+1))$$
$$x_2(i,j+2) = f_2(x_2(i+1,j+1))$$

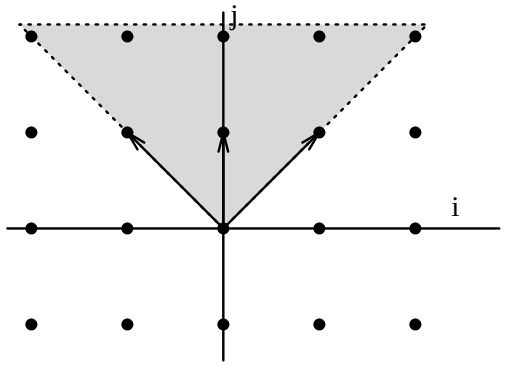


Figure 8: New Batch Set Dependence Vectors.

on the domain

$$\{(i,j) : 1 \leq i \leq 5, -1 \leq j \leq 5\}.$$

The new batch set dependence vectors are plotted in Figure 8. The shaded region again indicates the set of points whose vector position forms an acute angle with all batch set dependences. This system of recurrence equations is clearly batch computable on any arbitrary domain.

## 5.1 Dependence Relations

To understand how scheduling characteristics change due to index shifting we must first understand how the dependences change. For a given computational problem we can uniquely represent index shifting by a set of $n$ index vectors $\boldsymbol{i}_i$ representing the shift in each recurrence equation. Suppose we have a system of uniform recurrence equations which contained

$$x_i(\mathbf{p} - \mathbf{o}_i) = f_i(\cdots, x_j(\boldsymbol{p} - \boldsymbol{d}), \cdots). \tag{21}$$

After index shifting this equation by $\mathbf{i}_i$ we would obtain,

$$x_i(\boldsymbol{p} - \boldsymbol{o}_i - \mathbf{i}_i) = x_i(\boldsymbol{p} - \boldsymbol{o}_i') = f_i'(\cdots, x_j(\boldsymbol{p} - \boldsymbol{d}'), \cdots) \tag{22}$$

where $\boldsymbol{o}_i' = \boldsymbol{o}_i + \mathbf{i}_i$ and $\boldsymbol{d}' = \boldsymbol{d} + \mathbf{i}_i$. Note that $f_i'(\boldsymbol{p})$ differs from $f_i(\boldsymbol{p})$ only in its set of arguments.

Recall from the previous chapter that for a dependence vector $\boldsymbol{d}$, a member of the set $\mathcal{L}_{ij}$, there was an associated index set dependence vector, $\boldsymbol{v} = \boldsymbol{d} - \boldsymbol{o}_j$. Upon index shifting, the vector $\boldsymbol{v}$ will then be transformed to

$$\boldsymbol{v}' = \boldsymbol{d} - \boldsymbol{o}_j + \mathbf{i}_i - \mathbf{i}_j. \tag{23}$$

There is an added complication to our analysis because of the possibility that a $\boldsymbol{v} \neq 0$ can be transformed to a $\boldsymbol{v}' = 0$, or vice versa. Thus, the dependences associated with the set $\mathcal{V}$ can change after index shifting. For this reason we rely on graphical analysis to determine methods of index shifting.

We now give an important property of index shifting.

**Lemma 4.** *The sum of arc labels, $\boldsymbol{v}$, in a closed walk, is invariant under index shifting.*

This is clearly true since (22) shows that index shifting adds the shift to incoming edges and subtracts the shift from outgoing edges. Each vertex in a closed walk has the same number of incoming edges as outgoing edges so the changes cancel each other.

## 5.2 Conditions for Index Shifting to Allow Batch Computability

In this subsection we determine the necessary and sufficient conditions for a computation problem to be index shiftable to batch computability. The proof of sufficiency will be constructively shown through an algorithm based on the recurrence graph.

**Theorem 2.** *A system of uniform recurrence equations can be index shifted to a system that is batch computable on all finite domains if and only if there exists a vector $\boldsymbol{q}$ which has a strictly positive inner product with each cyclic sum of the recurrence graph. (Here, a cyclic sum is the sum of labelings around a cycle.)*

**Proof of necessity**
We show necessity by proving the contrapositive. Suppose that there is no vector having a strictly positive inner product with each cyclic sum. Then, by Lemma 3, there is some nontrivial linear combination of cyclic sums equal to zero,

$$\boldsymbol{\pi}_1 \mathbf{c}_1 + \boldsymbol{\pi}_2 \mathbf{c}_2 + \cdots = 0, \tag{24}$$

where $\boldsymbol{\pi}_i \geq 0$ and $\mathbf{c}_i$ is the sum of labels in cycle i. We can form a matrix $\overline{\boldsymbol{A}} = [\boldsymbol{v}_1, \boldsymbol{v}_2, \cdots, \boldsymbol{v}_m]$, where the $\boldsymbol{v}_i$ are the recurrence graph arc labels (some of which may be zero). For each cycle, with cyclic sum $\mathbf{c}_i$, we generate an m-dimensional vector $\mathbf{u}^i$, where an element $u^i_j$ equals 1 if the arc associated with $\boldsymbol{v}_j$ is in cycle i. Although the matrix $\overline{\boldsymbol{A}}$ will change with index shifting, the relationship $\overline{\boldsymbol{A}}\mathbf{u}^i = \mathbf{c}_i$ will always hold due to Lemma 4. Now from (24) we can write

$$\boldsymbol{\pi}_1 \overline{\boldsymbol{A}}\mathbf{u}^1 + \boldsymbol{\pi}_2 \overline{\boldsymbol{A}}\mathbf{u}^2 + \cdots = 0, \tag{25}$$

$$\overline{\boldsymbol{A}}(\boldsymbol{\pi}_1 \mathbf{u}^1 + \boldsymbol{\pi}_2 \mathbf{u}^2 + \cdots) = 0. \tag{26}$$

If we let $\overline{\mathbf{z}} = (\boldsymbol{\pi}_1 \mathbf{u}^1 + \boldsymbol{\pi}_2 \mathbf{u}^2 + \cdots)$, we arrive at the equation $\overline{\boldsymbol{A}}\overline{\mathbf{z}} = 0$. Since each $\mathbf{u}^i$ is a semipositive vector, and at least one $\boldsymbol{\pi}_i$ is greater than zero, we know that $\overline{\mathbf{z}}$ is semipositive.

Suppose that we remove all of the zero columns from $\overline{\boldsymbol{A}}$ and the corresponding rows of $\overline{\mathbf{z}}$; we form the matrix $\boldsymbol{A}$ from Theorem 1 and a vector $\mathbf{z}$. Although $\overline{\mathbf{z}}$ is semipositive, we may have $\mathbf{z} = 0$ if all of the nonzero components were removed. In this case, a cycle i corresponding to some $\boldsymbol{\pi}_i > 0$ will have all of its labels on the recurrence graph zero, and therefore we would contradict property *1.)* of Theorem 1. Otherwise, if $\mathbf{z} \geq 0$, we contradict property *2.)* of Theorem 1. □

**Proof of sufficiency**

Sufficiency is shown constructively by an algorithm for index dimensions of 2 or greater. Sufficiency for the special case of a one-dimensional system of recurrence equations is proven in Ellis, Ravikanth and Kumar [17], also constructively.

We use a single matrix $\mathbf{C}$ to represent all of the cyclic sums in the recurrence graph,

$$\boldsymbol{C} = [\mathbf{c}_1, \mathbf{c}_2, \cdots]. \tag{27}$$

We start with a vector $\boldsymbol{q}$ that has a strictly positive inner product with each cycle, $\boldsymbol{q}^T\mathbf{C} > 0$. We want to find index shifts resulting in a set $\mathcal{V}'$ in which every element $\boldsymbol{v}'$ has a strictly positive inner product with another vector $\boldsymbol{q}'$.

The first part of our algorithm involves choosing an appropriate relatively prime vector $\boldsymbol{q}'$. By "relatively prime vector" we mean a vector whose components are relatively prime integers. Consider a vector $\mathbf{n} = [n_1, n_2, \cdots]^T$, where $n_i$ is the number of vertices in cycle i. We want to find a $\boldsymbol{q}'$ that will satisfy the inequality

$$\boldsymbol{q}'^T\mathbf{C} \geqq \mathbf{n}^T. \tag{28}$$

We find $\boldsymbol{q}'$ following the method of Robert and Song [12] that involves scaling $\boldsymbol{q}$ and then adding a perturbation to regain a relatively prime vector. Suppose the initial vector $\boldsymbol{q}$ does not satisfy (28). We can easily find another relatively prime vector $\boldsymbol{r} \perp \boldsymbol{q}$ provided $\boldsymbol{q}$ has dimension 2 or greater. Since $\boldsymbol{r}$ is relatively prime, we can find a vector $\boldsymbol{s}$, such that $\boldsymbol{s}^T\boldsymbol{r} = 1$. The vector $\boldsymbol{q}'$ is then found from

$$\boldsymbol{q}' = \lambda\boldsymbol{q} + \boldsymbol{s}, \tag{29}$$

where $\lambda$ is the smallest positive integer such that

$$\lambda\boldsymbol{q}^T\mathbf{C} \geqq \mathbf{n}^T - \boldsymbol{s}^T\mathbf{C}. \tag{30}$$

Since $\boldsymbol{q}'^T\boldsymbol{r} = 1$, we know that $\boldsymbol{q}'$ is relatively prime.

From (22) we see that by index shifting, we change the value of $\boldsymbol{q}'^T\boldsymbol{v}$ by adding and subtracting $\boldsymbol{q}'^T\mathbf{i}$'s,

$$\boldsymbol{q}'^T\boldsymbol{v}' = \boldsymbol{q}'^T\boldsymbol{v} + \boldsymbol{q}'^T\mathbf{i}_i - \boldsymbol{q}'^T\mathbf{i}_j. \tag{31}$$

We will refer to the inner product $\boldsymbol{q}'^T\boldsymbol{v}$ as the *weight*, $w$, of an arc,

$$w = \boldsymbol{q}'^T\boldsymbol{v}. \tag{32}$$

Since $\boldsymbol{q}'^T\mathbf{r} = 1$, we can change the weights by integral amounts if we choose shifts $\mathbf{i}_i = \mu_i\boldsymbol{r}$, $\mu_i \in \mathcal{Z}$. We refer to the $\mu_i$ as *shifting weights*. Applying (31), we see that the weight, $w$, of an arc from $x_j$ to $x_i$ is transformed on index shifting to

$$w' = w - \mu_j + \mu_i. \tag{33}$$

## The Shifting algorithm

Index shifts are chosen in two steps. First we choose index shifts, $\mathbf{i}'_i = \mu_i\boldsymbol{r}$, for each equation that is contained in a cycle on the recurrence graph. These are chosen so that the arcs contained in a cycle will have $w' \geq 1$. Second, we find a complete set of shifts $\mathbf{i}_i$ so that *all* arcs will have $w' \geq 1$.

### Part 1.

We follow the method of Liu, Ho and Sheu [11] and start our algorithm by separating the variables into sets $\mathcal{X}_i$ corresponding to the strongly connected components of the recurrence graph. For every pair of variables $(x_i, x_j)$ we restrict our attention to the arc from $x_i$ to $x_j$ with the smallest weight. If all of these arcs have a strictly positive weight after index shifting, all other arcs will as well.

We define a function of the arcs, $g$, by

$$g(e) \triangleq w - 1, \text{where } w = \text{ weight of arc } e. \tag{34}$$

Using the function $g$, we find the shifting weights for the vertices of each strong component, $\mathcal{X}_i$. We start by arbitrarily assigning some vertex $x_i \in \mathcal{X}_i$ the value of $\mu_i = 0$. The other vertices $x_j \in \mathcal{X}_i$, have

$$\mu_j = \min\left\{\sum_{e\in P} g(e) : P \text{ is a path from } x_j \text{ to } x_i\right\}. \tag{35}$$

Thus $\mu_j$ can be regarded as the shortest path from vertex $x_j$ to $x_i$. Note that since $\boldsymbol{q}'^T\mathbf{C} > 0$, the shortest path cannot be made arbitrarily negative by going around in cycles, and is hence well-defined.

Consider an arc, $e$, from $x_p$ to $x_q$, where $x_p \neq x_i$. From the triangle inequality for shortest paths it is clear that

$$\mu_p \leq \mu_q + g(e). \tag{36}$$

We can substitute for $g(e)$ to find

$$\mu_p \leq \mu_q + w - 1, \tag{37}$$

$$w - \mu_p + \mu_q \geq 1. \tag{38}$$

Now from (33) we see that $w' \geq 1$.

Now consider an arc $e$, from $x_i$ to some $x_q$. Since $\mu_i = 0$,

$$w'(e) = w(e) + \mu_q. \tag{39}$$

Let $P$ be the shortest path from $x_q$ to $x_i$ with the function $g$. By (35) we know that $g(P) = \mu_q$. Since $e + P$ forms a cycle, we know that $g(e) + g(P) \geq 0$ from (28) and (34). By substituting $w = g + 1$, we have

$$\begin{aligned} w' &= g(e) + 1 + \mu_q, & (40)\\ &= g(e) + g(P) + 1, & (41)\\ &\geq 1. & (42) \end{aligned}$$

We have now assured that all arcs within a strong component have a strictly positive weight.

**Part 2.**

We now address those arcs that are incident from a vertex in one strong component to a vertex in another. In this second part of the shifting algorithm we assure that all of these arcs have a strictly positive weight, while preserving the weights of the arcs within the strong components.

We start by generating the condensation graph[2]of the recurrence graph as follows: For each strong component $\mathcal{X}_i$ we assign a vertex.[3] For each pair of strong components $(\mathcal{X}_i, \mathcal{X}_j)$, if there exists an arc on the recurrence graph from some $x_l \in \mathcal{X}_i$ to some $x_q \in \mathcal{X}_j$ we draw a corresponding arc from $\mathcal{X}_i$ to $\mathcal{X}_j$ on the condensation graph. We weight the arcs of the condensation graph with the smallest weight of their corresponding arcs on the recurrence graph after the first part of the shifting algorithm. It follows that the condensation graph will be acyclic; see [15].

Index shifting now proceeds by assigning a series of shifts $\boldsymbol{s}_i = \gamma_i \boldsymbol{r}$ applied to the strong components as a whole, i.e., $\mathbf{i} = \mathbf{s}_i, \forall x_j \in \mathcal{X}_i$. The weight of each arc within a strong

[2]The condensation of a graph is another graph with a vertex for each strong component, and arcs between these vertices if and only if there is some arc between the corresponding strong components; see [15].

[3]If a vertex $x_i$ on the recurrence graph is not part of any cycle we consider the singleton set $\{x_i\}$ as a strong component.

component will not change because the same shift is applied to the vertex it is incident from as well as the vertex it is incident to. If on the recurrence graph we can assure that the arc with the smallest weight between two strong components is positive, all of the other arcs between the same two strong components will also be positive.

Index shifting of the condensation graph follows a method similar to the index shifting of the strong components. We start by defining the function $g$ as

$$g = w - 1. \tag{43}$$

We assign shift weights $\gamma_i = 0$ to all vertices $\mathcal{X}_i$ that have no arcs incident from them, i.e., the root of the acyclic graph. For the remaining vertices we choose

$$\gamma_j = \min\{\sum_{e \in P} g(e) : P \text{ is a path from } \mathcal{X}_j \text{ to } \mathcal{X}_i \text{ such that } \gamma_i = 0\}. \tag{44}$$

By following the same analysis of (36) – (38), we see that all arcs of the condensation graph will now have a positive weight.

In terms of our proof, this algorithm has generated a computation problem in which each dependence vector $\boldsymbol{v}'$ has a strictly positive inner product with a vector $\boldsymbol{q}'$. Therefore, by Lemma 3 we have satisfied property *2.)* of Theorem 1. Since each $\boldsymbol{v}'$ has a strictly positive inner product, the dependence graph of every batch $\mathcal{B}(\boldsymbol{p})$ has no cycles of zero length. Thus property *1.)* of Theorem 1 is true, proving sufficiency. □

Our algorithm has the added benefit that it enables batch parallelism to be utilized.

**Lemma 5.** *Performing the index shifting algorithm, all variables within each batch $\mathcal{B}(\boldsymbol{p})$ can be generated simultaneously.*

Since every dependence has a weight $w \geq 1$, there can be no $\boldsymbol{v}' = 0$. Hence, there is no $x_i(\boldsymbol{p} - \boldsymbol{o}_i')$ needed for any $x_j(\boldsymbol{p} - \boldsymbol{o}_j')$.

For completeness, we now give a summary of our index shifting algorithm.

## Summary of the shifting algorithm

1. Form a matrix $\mathbf{C} = [\mathbf{c}_1, \mathbf{c}_2, \cdots]$, where $\mathbf{c}_i$ is the sum of labelings around cycle i.
2. Determine if there exists a vector $\boldsymbol{q}$ such that $\boldsymbol{q}^T\mathbf{C} > 0$. If no such vector exists, the problem cannot be index shifted to allow for batch computability on all finite domains. If such a vector exists, choose a relatively prime $\boldsymbol{q}$.
3. Form the vector $\mathbf{n} = [n_1, n_2, \cdots]^T$, where $n_i$ is the number of vertices in cycle i.

4. If $\boldsymbol{q}^T\mathbf{C} \geqq \mathbf{n}^T$, let $\boldsymbol{q}' = \boldsymbol{q}$ and choose $\boldsymbol{r}$ such that $\boldsymbol{q}^T\boldsymbol{r} = 1$. Otherwise, choose vectors $\boldsymbol{s}$ and $\boldsymbol{r}$ such that $\boldsymbol{r} \perp \boldsymbol{q}$ and $\boldsymbol{s}^T\boldsymbol{r} = 1$. Let $\boldsymbol{q}' = \lambda\boldsymbol{q} + \boldsymbol{s}$, where $\lambda$ is the smallest integer such that $\boldsymbol{q}'^T\mathbf{C} \geqq \mathbf{n}^T$.

5. Assign a weight $w = \boldsymbol{q}'^T\boldsymbol{v}$ to every arc, where $\boldsymbol{v}$ is the arc label.

6. Between any two vertices, remove all but the smallest weighted arc.

7. Assign the new weight function $g = w - 1$ to each of the remaining arcs.

8. Separate the vertices into strong components and form the induced subgraph on each such component.

9. Choose an arbitrary vertex $x_i$ in each strong component and let the index shift for it be $\mathbf{i}_i = 0$. Index shift the remaining vertices $x_j$ by $\mathbf{i}_j = \mu_j\boldsymbol{r}$, where $\mu_j$ is the shortest path to the vertex $x_i$ on the induced subgraph.

10. Form the condensation graph and label each arc with the smallest value of $g' = w' - 1$, where $w'$ is the weight of an associated arc on the recurrence graph *after* step 9.

11. Index shift the strong components forming the roots of the condensation graph by $\mathbf{s}_i = 0$. Index shift the remaining strong components by $\mathbf{s}_j = \gamma_j\boldsymbol{r}$, where $\gamma_j$ is the shortest path to a root of the condensation graph.

We now give a detailed example to illustrate this algorithm.

## Example 5:

Compute

$$\begin{aligned}
x_1(i,j) &= f_1(x_3(i,j+1)) \\
x_2(i,j) &= f_2(x_1(i,j), x_3(i+1,j-1)) \\
x_3(i,j) &= f_3(x_2(i,j-2), x_2(i,j-3)) \\
x_4(i,j) &= f_4(x_3(i+1,j), x_5(i-1,j-1)) \\
x_5(i,j) &= f_5(x_4(i-1,j+3))
\end{aligned}$$

on the domain

$$\{(i,j) : 1 \leq i \leq 100, 1 \leq j \leq 100\}.$$

The recurrence graph for this system is shown in Figure 9. All of the nonzero dependences have been plotted in Figure 10a. We can clearly see that no vector will have an acute angle with all of them. We therefore consider the set of cyclic sums. These have been plotted in Figure 10b, where the shaded region is the set of points having a vector with a strictly

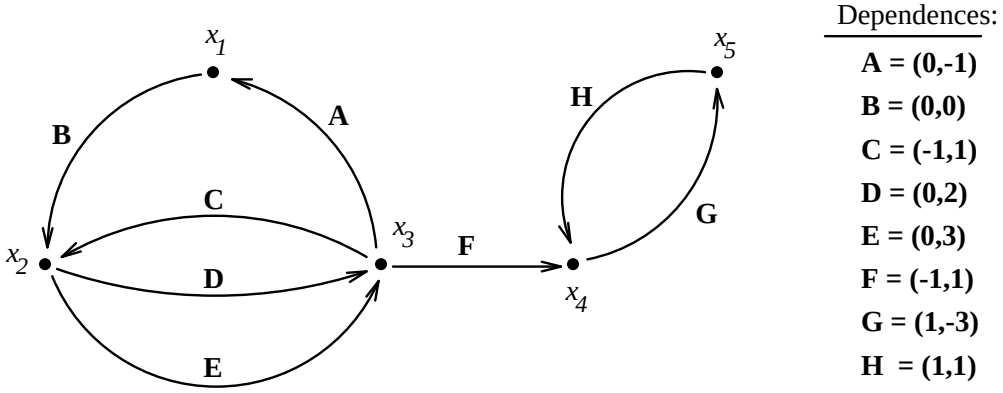


Figure 9: Recurrence Graph for Example 5.

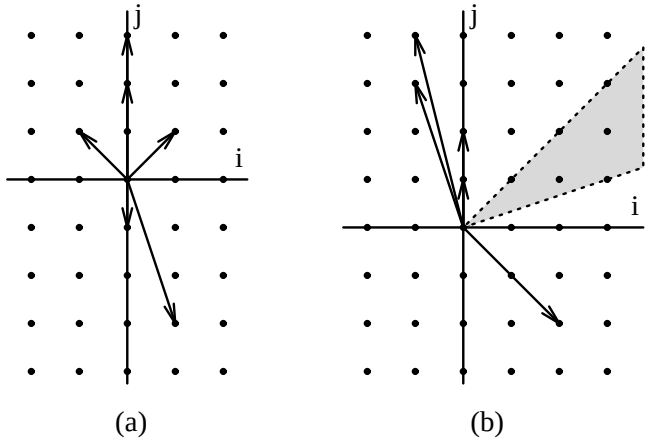


Figure 10: Graph of (a) Nonzero Batch Set Dependence Vectors, and (b) Cyclic Sums, for the System of Recurrence Equations in Example 5.

positive inner product with each cyclic sum. Since such a region exists, we know that the system of recurrence equations can be index shifted to a batch computable form. We choose the vector $(2, 1)$ as our initial $\boldsymbol{q}$. In Table 1 we have listed information about the cycles in the recurrence graph based on our choice of $\boldsymbol{q}$. We can see that (28) will not be satisfied due to cycles 1, 2, and 3, and therefore we have to find another vector $\boldsymbol{q}'$.

Following the method outlined earlier, we choose $\mathbf{r} = (-1, 2)$, which is clearly orthogonal to $\boldsymbol{q}$. We also choose $\mathbf{s} = (-1, 0)$ which satisfies $\mathbf{s}^T\mathbf{r} = 1$. We now solve (4.8) for $\lambda$. We know that

$$\mathbf{C} = \begin{bmatrix} 0 & 0 & -1 & -1 & 2 \\ 1 & 2 & 3 & 4 & -2 \end{bmatrix}, \tag{45}$$

$$\mathbf{n} = \begin{bmatrix} 3 & 3 & 2 & 2 & 2 \end{bmatrix}. \tag{46}$$

**Table 1:** Cycles Within the Recurrence Graph of Example 5.

| Cycle # | Arc Set | $\mathbf{c}_i$ | $\boldsymbol{q}^T\mathbf{c}_i$ | $n_i$ |
|---|---|---|---|---|
| 1 | {A,B,D} | (0,1) | 1 | 3 |
| 2 | {A,B,E} | (0,2) | 2 | 3 |
| 3 | {C,D} | (-1,3) | 1 | 2 |
| 4 | {C,E} | (-1,4) | 2 | 2 |
| 5 | {G,H} | (2,-2) | 2 | 2 |

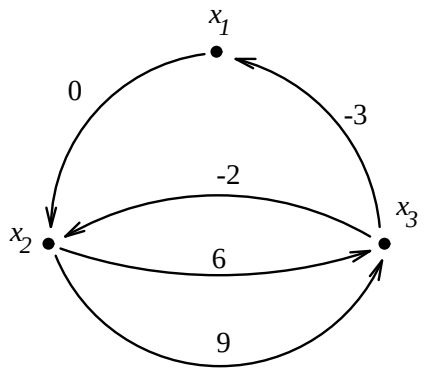


Figure 11: The Induced Subgraph for Strong Component $\mathcal{X}_1$.

Thus (29) becomes

$$\lambda \begin{bmatrix} 1 & 2 & 1 & 2 & 2 \end{bmatrix} \geq \begin{bmatrix} 3 & 3 & 1 & 1 & 4 \end{bmatrix}. \tag{47}$$

This is satisfied for $\lambda \geq 3$. For simplicity, we choose $\lambda = 3$, making

$$\boldsymbol{q}' = 3(2,1) + (-1,0) = (5,3). \tag{48}$$

We are now ready to start the shifting algorithm. Referring to the recurrence graph we see there are two strong components, $\mathcal{X}_1 = \{x_1, x_2, x_3\}$ and $\mathcal{X}_2 = \{x_4, x_5\}$. The induced subgraph for strong component $\mathcal{X}_1$ is drawn in Figure 11, where we have labeled the arcs with their weight from $\boldsymbol{q}' = (5,3)$.

We can neglect the lowermost arc since it is between the same vertices as the one above it. We now assign the function $g$ to the arcs, as illustrated in Figure 12.

If we arbitrarily choose $\mu_3 = 0$, the other shifting weights are determined from (34) to be $\mu_1 = 4$, and $\mu_2 = 5$. The index shifts for these three equations, as determined by $\mathbf{i}'_i = \mu_i \mathbf{r}$, are

$$\mathbf{i}'_1 = (-4,8), \mathbf{i}'_2 = (-5,10), \mathbf{i}'_3 = (0,0). \tag{49}$$

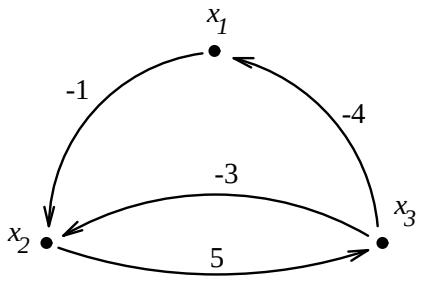


Figure 12: The Induced Subgraph for $\mathcal{X}_1$ with Labelings $g(e)$.

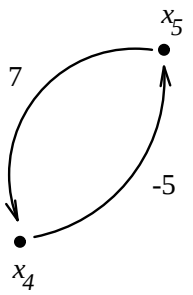


Figure 13: The Induced Subgraph for $\mathcal{X}_2$ with Labelings $g(e)$.

We follow the same procedure for strong component $\mathcal{X}_2$. In Figure 13 we have drawn the induced subgraph of $\mathcal{X}_2$ with the function $g$ assigned to the arcs.

By choosing $\mu_5 = 0$, we find that $\mu_4 = -5$, making the index shifts

$$\mathbf{i}'_4 = (5, -10), \mathbf{i}'_5 = (0, 0). \tag{50}$$

This completes the first part of our shifting algorithm. We have transformed the system of uniform recurrence equations to the new system shown below:

Compute

$$\begin{aligned}
&x_1(i+4, j-8) = f_1(x_3(i+4, j-7)) \\
&x_2(i+5, j-10) = f_2(x_1(i+5, j-10), x_3(i+6, j-11)) \\
&x_3(i, j) = f_3(x_2(i, j-2), x_2(i, j-3)) \\
&x_4(i-5, j+10) = f_4(x_3(i-4, j+10), x_5(i-6, j+9)) \\
&x_5(i, j) = f_5(x_4(i-1, j+3)).
\end{aligned}$$

We know that all arcs that are part of a cycle will now have strictly positive weights. Referring to Figure 9, we see that all arcs fall into this category with the exception of $\mathbf{F}$. After the first part of index shifting, this dependence has a batch set dependence vector

$$\boldsymbol{v} = \boldsymbol{d} - \boldsymbol{o}_3 = (4, -10) - (0, 0) = (4, -10). \tag{51}$$

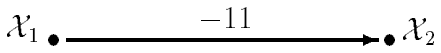


Figure 14: Condensation Graph for Example 5.

The weight of this vector is $(5,3)(4,-10)^T = -10$, making $g = -11$. From this information we can now draw the condensation graph, shown in Figure 14.

Following the procedure discussed earlier we let $\gamma_2 = 0$, and $\gamma_1 = -11$. We now have to apply the shift $\mathbf{s}_1 = \gamma_1 \mathbf{r} = (11, -22)$ to the entire set $\mathcal{X}_1$. This gives us the following system of uniform recurrence equations.

Compute

$$x_1(i-7, j+14) = f_1(x_3(i-7, j+15))$$
$$x_2(i-6, j+12) = f_2(x_1(i-6, j+12), x_3(i-5, j+11))$$
$$x_3(i-11, j+22) = f_3(x_2(i-11, j+20), x_2(i-11, j+19))$$
$$x_4(i-5, j+10) = f_4(x_3(i-4, j+10), x_5(i-6, j+9))$$
$$x_5(i, j) = f_5(x_2(i-1, j+3)).$$

The batch set dependence vectors for our new system of uniform recurrence equations are plotted in Figure 15. We have again shaded the region of points whose position vector forms an acute angle with all of the dependences. We can clearly see that the vector $\boldsymbol{q}' = (5, 3)$ is within this shaded region.

At this point we need to discuss the additional considerations of the domain of computation and the boundary conditions. In Section 5.4 we return to the example problem and show the complete organization of computations.

## 5.3 Domains of Computation

We now give a brief description of how the domain of computation needs to change after index shifting. In the original computation problem we were required to generate

$$x_i(\boldsymbol{p} - \boldsymbol{o}_i), \text{ for } \boldsymbol{p} \in \mathcal{D}. \tag{52}$$

After index shifting the equation for $x_i(\boldsymbol{p} - \boldsymbol{o}_i)$ by $\mathbf{i}_i$ our same computations for $x_i(\boldsymbol{p} - \boldsymbol{o}_i)$ occur at the points $\boldsymbol{p} + \mathbf{i}_i$. If we choose the domain

$$\mathcal{D}' = \{\boldsymbol{p} : \boldsymbol{p} + \mathbf{i}_i \in \mathcal{D}\}, \tag{53}$$

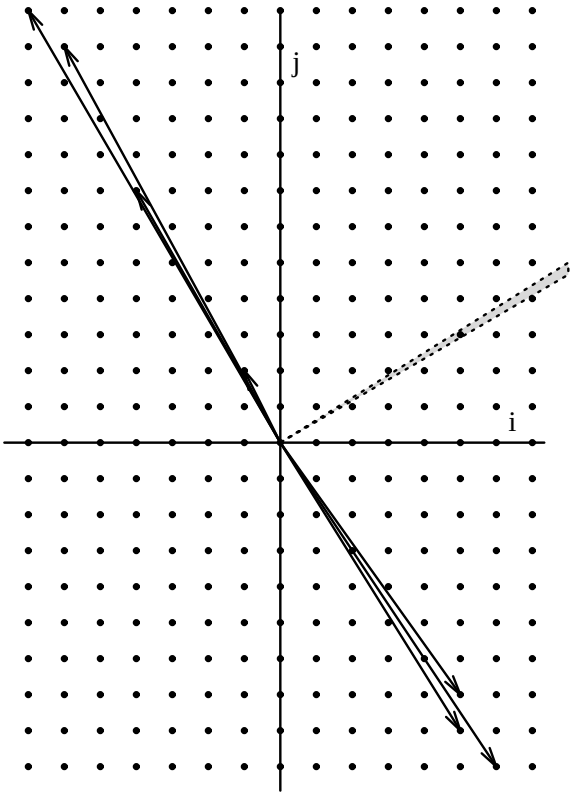


Figure 15: New Batch Set Dependences Vectors for Example 5 after Index Shifting.

we will generate all of the same instances of $x_i$. Since we have to generate all of the same instances of each variable, our new domain should be

$$\mathcal{D}' = \bigcup_{i=1}^{n} \{\boldsymbol{p} : \boldsymbol{p} + \mathbf{i}_i \in \mathcal{D}\}. \tag{54}$$

This will be the smallest possible domain over which all of the original variable instances are generated.

In some computational problems we may want to restrict index domains to hyper-rectangles, i.e., domains that can be expressed as

$$\mathcal{D} = \{\boldsymbol{p} : \boldsymbol{p}_{max} \geqq \boldsymbol{p} \geqq \boldsymbol{p}_{min}\}. \tag{55}$$

If we represent each index shift vector as

$$\mathbf{i}_i = [s_1^i, s_2^i, \cdots], \tag{56}$$

we can define two other vectors $\mathbf{s}_L$ and $\mathbf{s}_U$ as

$$\mathbf{s}_L \triangleq [\min_i\{s_1^i\}, \min_i\{s_2^i\}, \cdots], \tag{57}$$

$$\mathbf{s}_U \triangleq [\max_i\{s_1^i\}, \max_i\{s_2^i\}, \cdots]. \tag{58}$$

It should be clear that $\mathbf{s}_U \geqq \mathbf{i}_i \geqq \mathbf{s}_L$, for every index shift vector $\mathbf{i}_i$. We can therefore choose a new index domain

$$\mathcal{D}' = \{\boldsymbol{p} : \boldsymbol{p}_{max} + \mathbf{s}_U \geqq \boldsymbol{p} \geqq \boldsymbol{p}_{min} + \mathbf{s}_L\}. \tag{59}$$

Applying this method to Example 5, we see that $\mathbf{s}_L = [0, -22]$ and $\mathbf{s}_U = [11, 0]$. Therefore, the new domain should be

$$\mathcal{D}' = \{(i, j) : 1 \leq i \leq 111, -21 \leq j \leq 100\}.$$

## 5.4 Boundary Conditions

After index shifting, special attention has to be given to the *boundary conditions*. In the original description of Example 5, we required $x_2(1, -1)$ and $x_2(1, -2)$ to determine $x_3(1, 1)$. Both of these values are boundary conditions contained in the set $\mathcal{C}_2$. However, after index shifting, we have arranged to compute $x_2(1, -1)$ and $x_2(1, -2)$ by evaluating $f_2$, and we may generate different values than those given as boundary conditions.

Consider a computation problem with a variable $x_i$ and associated sets $\mathcal{D}_i$ and $\mathcal{C}_i$. If after index shifting we have $\mathcal{D}_i' \cup \mathcal{C}_i \neq \phi$, we will compute a variable instance $x_i(\boldsymbol{p})$ that was

previously a boundary condition. If the computed $x_i(\boldsymbol{p})$ is not the same as the boundary condition we will have incorrect results.

To compensate for this problem, we simply mask out the result of a computation that already exists as a boundary condition. This would make the final organization of computations for Example 5:

for t = -58 to 855, Do:
Simultaneously compute for all $(i,j) \in \mathcal{D}$ such that
5i + 3j = t:

$$x_1(i-7, j+14) = f_1(x_3(i-7, j+15))$$
$$x_2(i-6, j+12) = f_2(x_1(i-6, j+12), x_3(i-5, j+11))$$
$$x_3(i-11, j+22) = f_3(x_2(i-11, j+20), x_2(i-11, j+19))$$
$$x_4(i-5, j+10) = f_4(x_3(i-4, j+10), x_5(i-6, j+9))$$
$$x_5(i,j) = f_5(x_2(i-1, j+3))$$

Store results for all $x_p(i,j)$ such that $1 \leq i, j \leq 100$
end.

# 6 Concluding Remarks

We have shown how one may test a system of uniform recurrence equations for computability over arbitrary finite domains. Such an algorithm could be a useful addition to a high-level compiler for scientific applications. By automating dependence analysis, programs could be written directly from the recurrence description of their algorithm.

Our algorithm in Section 5 enables uniform recurrence equations to be entered in a program with arbitrary index shifts in an arbitrary order. The algorithm is also useful because it enables all of the recurrence equations to be executed simultaneously at a specific index point, an additional extension of parallelism.

Some useful extensions to this work would be algorithms for optimally finding the $\boldsymbol{q}'$ vector and the shifting vector $\boldsymbol{r}$ to extract the greatest parallelism. This would amount to an extension of the work in Robert and Song [12], in which index shifting was used only to enhance parallelism. There are many practical considerations, such as communication and synchronization, in mapping a computational problem to a machine architecture, that are beyond the scope of the problem dealt with here. It would be desirable to also develop computability tests for other scheduling methods such as the affine by statement method of Darte, Risset and Robert [13] .

**Acknowledgment** The authors are grateful to Carlos Humes, Jr. for instigating their interest in this problem.

# References


[1] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *Journal of the Association for Computing Machinery*, vol. 14, pp. 563–590, July 1967.

[2] Y. Muraoka, "Parallelism exposure and exploitation in programs," Ph.D. dissertation UIUCDCS-R-71-424, University of Illinois, Urbana, IL, February 1971.

[3] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Communications of the ACM*, vol. 29, pp. 1184–1200, December 1986.

[4] M. J. Wolfe, "Optimizing supercompilers for supercomputers," Ph.D. dissertation UIUCDCS-R-82-1105, University of Illinois, Urbana, IL, October 1982.

[5] U. Banerjee, "Data dependence in ordinary programs," M.S. thesis UIUCDCS-R-76-837, University of Illinois, Urbana, IL, November 1976.

[6] U. Banerjee, "Speedup of ordinary programs," Ph.D. dissertation UIUCDCS-R-79-989, University of Illinois, Urbana, IL, October 1979.

[7] M. E. Wolf and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 452–471, 1991.

[8] C. D. Polychronopoulos, "Compiler optimizations for enhancing parallelism and their impact on architecture design," *IEEE Transactions on Computers*, vol. 37, pp. 991–1004, 1988.

[9] L. Lamport, "The parallel execution of DO loops," *Communications of the ACM*, vol. 17, pp. 83–93, February 1974.

[10] W. Shang and J. A. B. Fortes, "Time optimal linear schedules for algorithms with uniform dependences," *IEEE Transactions on Computers*, vol. 40, pp. 723–742, 1991.

[11] L. Liu, C. Ho, and J. Sheu, "On the parallelism of nested for – loops using index shift method," in *Proceedings of the 1990 International Conference on Parallel Processing*, (University Park and London), pp. 119–123, The Pennsylvania State University, The Pennsylvania State University Press, August 13–17 1990.

[12] Y. Robert and S. W. Song, "Revisiting cycle shrinking," Technical Report 91-31, ENS-Lyon, Lyon, France, 1991.

[13] A. Darte, T. Risset, and Y. Robert, "Loop nest scheduling and transformation," in *Summer School on Scheduling Theory and Its Applications*, (Chateau de Bonas, Gers, France), 1992.

[14] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua, "Automatic program parallelization," *Proceedings of the IEEE*, vol. 81, no. 2, pp. 211–243, 1993.

[15] M. Behzad, G. Chartrand, and L. Lesniak-Foster, *Graphs and Digraphs*. Boston, MA: Prindle Weber and Schmidt, 1979.

[16] K. G. Murty, *Linear Programming*. New York, NY: John Wiley and Sons, 1983.

[17] R. D. Ellis, R. Ravikanth, and P. R. Kumar, "Automating the simulation of complex discrete-time control systems: A mathematical framework, algorithms and a software package," *IEEE Transactions on Automatic Control*, vol. 39, pp. 1795–1801, September 1994.