McGill University
School of Computer Science

Unimodular Transformations of Monolithic Array Constructors

ACAPS Technical Memo 43
May 1992

Nematollaah Shiri V.
Guang R. Gao

Advanced Compilers, Architectures and Parallel Systems Group

Abstract

Unimodular transformation has been proposed as a powerful techniques for loop parallelization in imperative language programs. In this paper, we propose to apply unimodular transformation to functional language programs. In particular, we propose a method of applying unimodular transformation to *monolithic* arrays such as Haskell array comprehensions. Using our method, a compiler can deduce a *safe* and parallel order of evaluation for a given monolithic array.

We formulate the problem of unimodular transformation under the context of functional programming. Instead of solving the result integer programming problem using expensive general solution techniques, we developed an efficient solution scheme by making use of the *dependence direction* and *distance* information in guiding the search of optimal solutions. We report an implementation of the algorithms in our experimental testbed.

Contents

1 Introduction **1**

2 Monolithic Arrays **1**

3 From MACs to Double Loops: An Example **4**

4 Problem Formulation and Solution Strategy **6**
4.1 Basic Definitions 6
4.2 Legal Array Constructors 7
4.3 Safe Order of Evaluation 15

5 Unimodular Transformations of MACs **17**
5.1 Outer Loop Parallelization 17
5.2 Inner Loop Parallelization 21
5.2.1 Problem Statement 21
5.2.2 Motivating Examples 22
5.2.3 Computing Ideal Pairs in Case 1 23
5.2.4 Inner Loop Parallelization Algorithm 33
5.2.5 Examples of Parallelized MACs 37

6 Implementation and Complexity Remarks **39**
6.1 AC-Transformer: A One-Step Parallelizing Program For MACs 40
6.2 Complexity Issue 41

7 Conclusion and Future Work **42**

8 Acknowledgment **43**

R References **43**

List of Figures

1 Index domain of the MAC with the index vector (i, j) 2
2 Transformation of the MAC by the unimodular matrix U 5
3 Index domains and their entry points 8
4 Dependence direction vectors in MACs 9
5 The hierarchies of the cases defined by D'' 10
6 Directions of edges of a distance cone 11
7 Direction vectors in Case 4 13
8 Unimodular transformations by U and U^{-1} 14
9 Unimodular transformations of distance cones 24
10 Solution spaces of the examples 25
11 Distance cones when directions $(-1, -1)$ and $(1, 1)$ exist 26
12 Distance cones when directions $(-1, 1)$ and $(1, -1)$ exist 29
13 The input/output diagram of the AC-Transformer 41

1 Introduction

A significant amount of the running time in most scientific programs is spent in the execution of loops. Such loops, in which arrays are frequently the dominant data structure, often contain ample parallelism, since different elements of an array may be processed concurrently. Therefore, exploiting *parallelism* considerably improves the speed of computation. Thus, efficient execution of loops is among the most important challenges for high-performance computer architectures. It has lead to the development of a variety of *dependence analysis*, *loop transformation*, and *parallelization* techniques [Ban76, Ban88, ABC86, ACK87, KKP+80, PW86, Wol89].

Banerjee in [Ban90] presents a technique for parallelizing *double loops* in imperative languages. The technique gives a uniform treatment of frequently used methods of transforming loops such as loop reversal, loop interchange, and loop skewing. His technique is based on unimodular transformations. Given a double loop $L = (L_1, L_2)$, the proposed algorithms determine a unimodular matrix U such that the transformation of L by U can be expressed as a double loop equivalent to L in which the outer loop or the inner loop can run in parallel. We will refer to a double loop as a loop or a nest.

In this work, we are interested in coarse-grain parallelism, where the whole body of the loop is treated as a single unit. In fine-grain parallelism, which is well suited for vector computers, the statements in the loop body are also parallelized (if possible). The underlying architecture is an *idealized machine*, where dependence in a program is the only constraint that may delay the execution of its tasks.

Outer loop parallelization is a more natural choice than the inner loop, when a nest is to be transformed. This is achievable when there is no dependence at level one [1]. Parallelizing the outer loop is to split the index domain of the array, used in the loop body, into distinct partitions, such that the *number* of the partitions is maximized and there is no dependence among the elements in different partitions. Parallelizing the inner loop is splitting the index domain of the nest into different partitions, such that the *size* of the partitions is maximized and there is no dependence among the elements within a partition. We say that a transformation L^U of a loop L by a unimodular matrix U is *optimal*, if L^U is executable in minimum number of steps. Based on this definition, the parallelizing algorithms in [Ban90] determine optimal transformations.

2 Monolithic Arrays

In this section, we define monolithic array constructors and describe our terms. We also specify one of the major problems that arises in parallel implementation of such array constructors. Given a MAC and a unimodular matrix U, in this section we show that we can express the

[1] All distance vectors in the loop are of the form $(0, D_2)$.

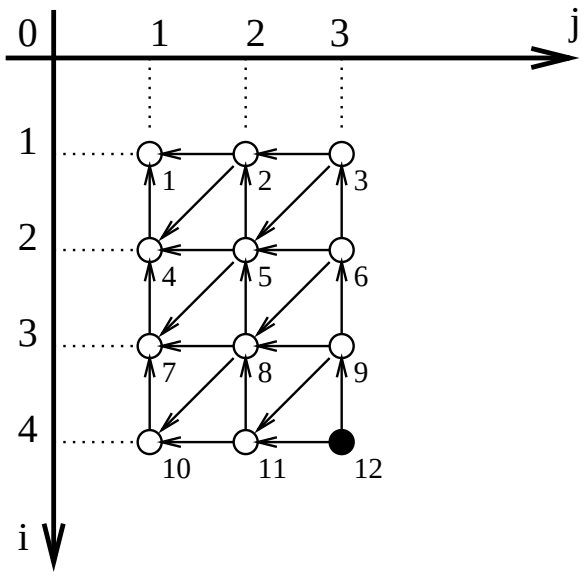


Figure 1: Index domain of the MAC with the index vector (i, j)

transformation of the MAC under U as a double loop (L_1^U, L_2^U). Consider the following example of a MAC that defines array A.

A = **For** $(i,\ j)$ **in** $[1:4,\ 1:3]$

$A[i+1,\ j] + A[i,\ j+1] + A[i-1,\ j+1]$

EndFor

The above construct recursively defines an array A. We assume that the boundary elements have already been defined. The array element $A[i, j]$ is defined by the body expression using the values of the elements $A[i+1, j]$, $A[i, j+1]$, and $A[i-1, j+1]$, where $1 \leq i \leq 4$ and $1 \leq j \leq 3$. The set of distance vectors in the MAC is $D = \{(-1, 0),\ (0, -1),\ (1, -1)\}$. Figure 1 shows the index domain of the MAC and the distance vectors. In the figure, an arc denotes the existence of a dependence from the source node to the sink node. The black node in the index domain is the array element $A[4, 3]$ which is an *entry point*. The index points in the figure are labeled to facilitate our explanation.

We explain the concept of monolithic array constructors in more details. Specifically, we use Haskell and Id in the examples.

The functional programming language Haskell [HWe90] has been proposed as a standard functional language for a broad range of applications. Haskell provides an expression form in which an array is specified by a list of pairs of the form $(index, value)$, where $index$ is a tuple

of integers, one for each dimension of the array being constructed, and *value* is an object of element type. The list is written as a concatenation of one or more *list comprehensions* [Pey87]. Such an expression defines an array monolithically and is known as an *array comprehension*. As an example, the wavefront computation may be written as follows [Hud89]:

```
A = array ((1,m),(1,n)) [((1,1),1)]
 ++ [((i,1),1) | i <- [2..m]]
 ++ [((1,j),1) | j <- [2..n]]
 ++ [((i,j),
       A!(i-1,j) + A!(i,j-1) + A!(i-1,j-1))
                | i <- [2..m], j <- [2..n]]
```

The Haskell program uses list comprehensions to specify four different regions which cover the index domain of A. In the wavefront example, the list comprehension for the interior region is created by two generators, one for i and one for j, producing index pairs for all points (i,j) not part of the north and west boundaries. To appreciate the semantics of an array comprehension, it is important to know that the order in which array elements are specified in the list *does not* specify the order in which element values are to be computed; the list is only a device for specifying the *values* of array elements. It is the responsibility of the implementation (the compiler and runtime system) to use an order of evaluation that honors the dependences expressed in the array comprehension. For a parallel computer it is also the job of the implementation to determine which elements may be safely evaluated concurrently. In the wavefront example, it is the sets of elements that make up the secondary diagonals of the array.

The intention in Id is that array definitions having arbitrary recursive dependences be written using the *I-structure* mechanism of Id. This mechanism allows for explicit allocation of storage for an array with all elements undefined, and supports write and read operations on the array elements. It is invalid to write the same element twice (an error that can only be detected during program execution, in general). Any reads of an array element are delayed until the element becomes defined. The general implementation of I-structures requires a special memory organization and a resource scheduling mechanism that may be invoked for each element of the constructed array. To avoid the penalties of using I-structures, the Id program could be analyzed by a compiler to recognize the important simple forms for which more efficient code can be constructed. The problems of doing this are similar to those for the monolithic array constructor in Haskell.

In many scientific computations an array may be recursively defined, but the pattern of self reference is regular, as in a linear recurrence or the wavefront computation. In these cases it is important that the inefficiencies of the general implementation of the Id and Haskell array constructors be avoided. Compile-time analysis can help if the regularities in the dependence relation can be identified and used to partition array elements into groups that can be safely evaluated in parallel. This is the subject of the rest of this report.

One feature of functional programming is implicit parallelism, i.e. no explicit evaluation order is specified. In a functional monolithic array constructor, the data dependences between the elements of the array may put constraints on the order of their evaluation. If a compiler does not identify the dependences and use this information to determine an order of execution, the implementation will have to rely on runtime schemes to ensure that an array element is defined before it can be used. Unfortunately, such runtime schemes are often quite expensive.

Our objective in this work is to present an efficient technique which avoids such expensive schemes. The technique introduces, at compile time, an explicit evaluation order between the elements of a recursively defined monolithic array constructor, while ensuring that the semantics of the program is not violated, i.e. the evaluation order is *safe*. Furthermore, we would find a safe order which also fully exploits parallelism. The technique is based on unimodular transformation designed for imperative loops. More specifically, we extended the technique presented in [Ban90], which we assume that the reader is familiar with.

3 From MACs to Double Loops: An Example

In this section we illustrate how the unimodular transformation can be applied to a MAC to obtain an efficient execution. Consider the example of a MAC we gave in the previous section. Let the unimodular matrix U be $\begin{bmatrix} -1 & 0 \\ -2 & 1 \end{bmatrix}$. The transformations of the elements in D by the matrix U are $(-1,0)\cdot U = (1,0) >_1 \mathbf{0}$, $(0,-1)\cdot U = (2,-1) >_1 \mathbf{0}$ and $(1,-1)\cdot U = (1,-1) >_1 \mathbf{0}$. Let $(K_1, K_2) = (-i-2j, j)$ be the transformation of the index vector (i,j) of the MAC. Since any distance vector in the MAC is transformed into a positive vector by U, the lexicographic order of the (K_1, K_2) is a safe order of evaluation of the array elements (Corollary 4.1). An index point $[i,j]$ in the MAC is transformed into $[-i-2j, j]$ by U. Figure 2 illustrates the transformed index points of the MAC. Now, we want to produce a double loop that its iteration points are the transformed index points and its index vector is (K_1, K_2).

To produce the desired double loop, we apply algorithm 4.1 in [Ban90]. Recall that the input to the algorithm is a double loop L and a unimodular matrix U, and its output is the double loop L^U that runs in the lexicographic order of (K_1, K_2), the transformation of the index vector in L. In an abstraction, we can regard L as a rectangular subset of $\mathcal{Z}^2$ and L^U as a transformation of that subset which may no longer be rectangular. By abstraction, we can apply the same algorithm to produce the transformation of the MAC by the unimodular matrix U. In our example, we have a rectangular set of index points defined by the index vector (i,j), where $n_1 \le i \le N_1$ and $n_2 \le j \le N_2$. The desired double loop (L_1^U, L_2^U) is:

```
DO K1 = -10, -3, 1
   DOALL K2 = Ceil (Max ((-4-K1)/2.0, 1)),
              Floor(Min ((-1-K1)/2.0, 3))
```

Figure 2: Transformation of the MAC by the unimodular matrix U

```
A(-K1-2*K2, K2) = A(-K1-2*K2+1, K2) + A(-K1-2*K2, K2+1) +
                  A(-K1-2*K2-1, K2+1)

   END
END
```

We note that evaluating the array elements in the lexicographic order of (K_1, K_2) is safe. Moreover, the inner loop L_2^U can be executed in parallel, since there is no dependence at level 2 in (L_1^U, L_2^U). We will use L_1^U and L_2^U, respectively, to refer to the outer loop and the inner loop in a resulting double loop. We will show that that there always exists a unimodular matrix U for a legal MAC, such that the transformation (L_1^U, L_2^U) of the MAC is equivalent to the MAC. Furthermore, with a properly chosen U, L_1^U or L_2^U is maximally parallelized. We will also present efficient algorithms to obtain, for a given MAC, such a desired unimodular matrix U.

4 Problem Formulation and Solution Strategy

4.1 Basic Definitions

In this section, we describe the structure of our monolithic array constructor and give some definitions. The program construct that we consider is a MAC that has the following structure:

A = **For** (i, j) **in** [n_1:N_1, n_2:N_2]

$H[i,\ j]$

EndFor

where the body expression H[i, j] defines A[i, j], an element of A at the index point [i, j]. We also refer to the above program as *the array constructor A* or simply the constructor. The index bounds n_1, N_1, n_2 and N_2 are integer constants. The *index vector* of the constructor is the ordered pair (i, j). We will refer to an instance of the index vector (i, j) as an *index point*. The *index domain* of the constructor is the set of all possible values of (i, j), i.e. $\{[i, j] \in \mathcal{Z}^2 \mid n_1 \leq i \leq N_1, n_2 \leq j \leq N_2\}$. Such an index domain is a rectangular subset of $\mathcal{Z}^2$, where $\mathcal{Z}$ denotes the integers. Thus, we regard an array as a function whose domain is a set of integer pairs. We denote a vector by (i, j) and an index point by $[i, j]$. For a detailed discussion of monolithic array constructors and a related work, the reader is referred to [GYDM90].

In a MAC, all the array elements are defined at once, at the time of definition of the array, and hence called *monolithic*. In evaluating the array elements in a MAC we may face two cases. In the first case, where there is no dependence among the array elements, all of them may be evaluated simultaneously. In the second case, evaluating an element references to some other elements of the array that are already defined. Our monolithic constructor permits an array to be defined recursively, but the pattern of recurrence is regular, as in a linear recurrence or the wavefront computation. In the example mentioned above, each element is defined based on its north and west neighbors. In the case where the pattern of recurrence is regular in a MAC, we use the regularities in the dependence relation to partition the elements of the array into groups that can be safely evaluated in parallel. In the following, we restate the notion of dependence relation.

Definition 4.1 *In an array constructor A, there is a dependence from $A[i', j']$ to $A[i, j]$ if the value of $A[i', j']$ defined by $H[i', j']$ is used in $H[i, j]$ to define $A[i, j]$. Accordingly, we say that there is a dependence from $H[i', j']$ to $H[i, j]$, or simply from the index point $[i', j']$ to $[i, j]$.*

Definition 4.2 *In an array constructor A, there is a dependence with the* dependence distance *vector $(d_1, d_2) = (i - i',\ j - j')$, if there are index points $[i', j']$ and $[i, j]$ such that there is a dependence from $[i', j']$ to $[i, j]$.*

Let (d_1, d_2) denote a distance vector in a MAC. If $d_1 \neq 0$, then we say that a dependence exists in the MAC at level 1. We say that a dependence exists at level 2 in the MAC if $d_1 = 0$ and $d_2 \neq 0$. Note that these definitions of dependences at a level are extensions to those given for imperstive loops. Distance (and accordingly, direction) vectors may be negative in functional programs. Any dependence in such programs is always a true data dependence, also called a *flow dependence*. This is because anti-dependence and output-dependence may not exist in functional programs.[2] Hence, direction of a dependence is always from the *definition* of an element to the *use* of that element. That is, in the index domain of a MAC, there is an arc from an element that defines a value to the element which uses that value.

We assume that distance vectors in a MAC are given. Thus, subscript forms of the array references in the body expression $H[i, j]$ are immaterial. In many scientific computations, an array may be recursively defined, but the pattern of recurrence is regular. That is, all index points in the domain of a MAC have the same fixed set of dependence vectors. Therefore, it is realistic to assume that the given distance vectors are regular and constant. We use this information to determine a transformation by which a safe evaluation order of the array elements in the MAC is introduced.

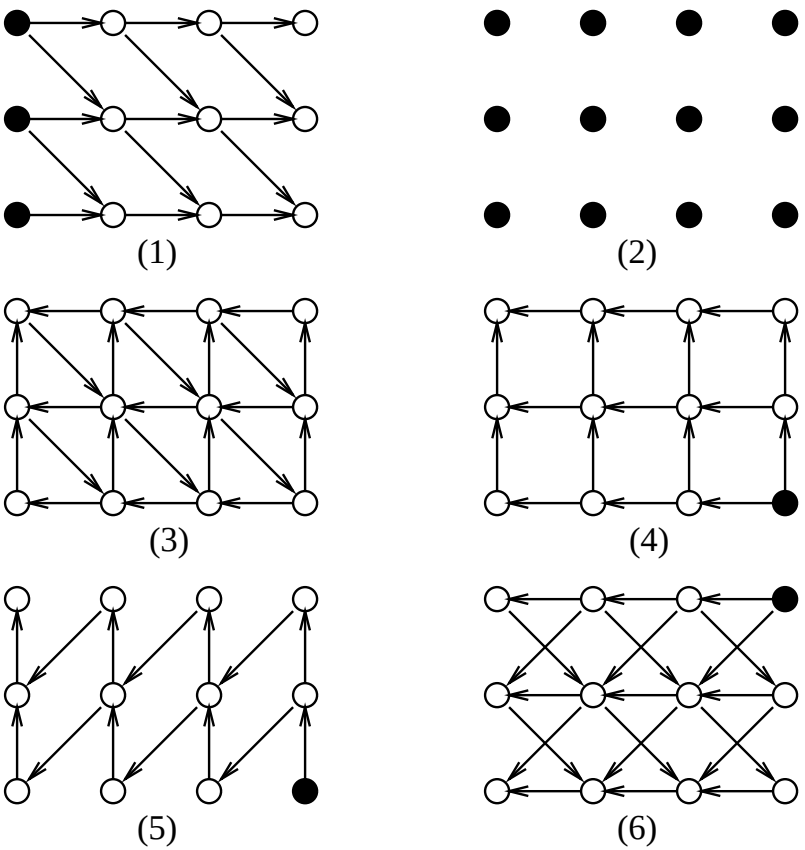
4.2 Legal Array Constructors

In this section we define *legal* array constructors in which no circular dependence may exist among their elements; otherwise, such a constructor is said to be *illegal*. Then, we study different combinations of direction vectors that may exist in legal MACs, in details. Given a legal MAC, we will show in Lemma 4.1 that any unimodular transformation of the constructor does not create cycles among the array elements. In Lemma 4.2 we will prove that cycles among the elements in an illegal MAC can not be broken by any unimodular transformation. We then give Theorem 4.1 that proves a MAC is legal if and only if there exists a unimodular matrix by which any distance vector in the MAC is transformed into a positive vector.

Definition 4.3 *Let $D = \{d_1, \ldots, d_k\}$ be the set of distance vectors in an array constructor. We say the constructor is illegal iff there exist non-negative integers $\alpha_1, \ldots, \alpha_k$, not all zeros, such that $\alpha_1 d_1 + \cdots + \alpha_k d_k = \mathbf{0}$. Otherwise we say that the constructor is legal.*

We will refer to the set of distance vectors in a legal (illegal) array constructor as a legal (illegal) set. Intuitively, some distance vectors in an illegal MAC form cycles among the array elements in the index domain. On the other hand, for any two index points p_1 and p_2 in a legal MAC, if there is a dependence from p_1 to p_2, then no (direct or indirect) dependence exists from p_2 to p_1. That is, there is no cycle among the elements in a legal MAC. Hence, there exists at least one entry point in the index domain of a legal array constructor. An *entry point*

[2] The compiler can issue an error message if such dependences are discovered.

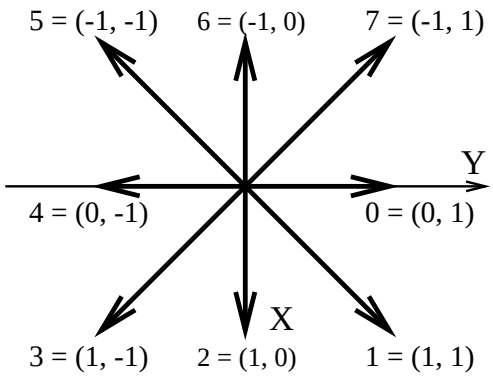


The index domain in (1) has three entry points. All the index points in (2) are entry points and can be evaluated in parallel. Example (3) is the index domain of an illegal MAC. In any of (4), (5), and (6) there exists one entry point.

Figure 3: Index domains and their entry points

is an index point to which no dependence exists from any other point in the index domain. Some examples of index domains of MACs are illustrated in Figure 3. A black node in an index domain represents an entry point. In the figure, the index domain in example 1 has three entry points. In example 2, all the index points are entry points. Example 3 in which no entry point exists represents the index domain of an illegal MAC. There exists one entry point in each of the examples 4, 5 and 6.

Let D and $D^{'}$ be, respectively, the sets of distance and direction vectors in a legal MAC. If we draw the elements of D in the X-Y plane, they form a cone with the apex at the origin O. We refer to this cone as a *distance cone*, or simply a cone. The *edges* of the cone are elements e_1 and e_2 in D such that any other element in D lies between them or coincides with e_1 and/or e_2. Note that edges of a distance cone may not be unique. The *size* of the cone is the smaller angle ($< \pi$) between e_1 and e_2. In a special case where e_1 and e_2 coincide, the cone size is $0°$. We are interested to obtain information about possible combinations of edges that a distance cone may



Vector number 2 is direction $(1, 0)$ and 6 is direction $(-1, 0)$.

Figure 4: Dependence direction vectors in MACs

have in a legal MAC. This helps to a better understanding of cones and also helps to analyze and classify them. To achieve this objective, we will use direction vectors as an alternative to the actual dependences. The followings are direction vectors in MACs [3], numbered from 0 to 7.

$$\begin{array}{llllllll} 0. & (0,1) & 1. & (1,1) & 2. & (1,0) & 3. & (1,-1) \\ 4. & (0,-1) & 5. & (-1,-1) & 6. & (-1,0) & 7. & (-1,1) \end{array}$$

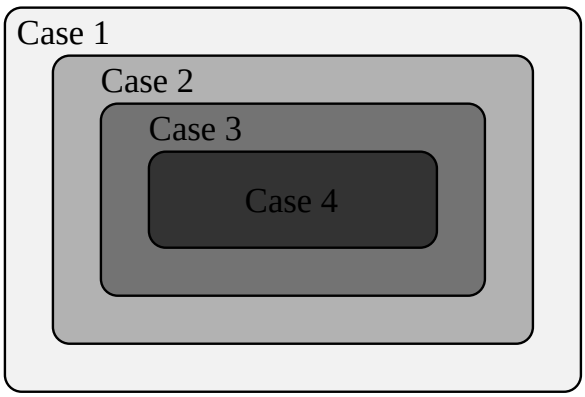
Figure 4 illustrates these directions in the X-Y plane. For simplicity, we refer to a direction vector by its number i, where $0 \leq i \leq 7$. Clearly, since the size of a distance cone is less than π, $D^{'}$ may contain at most five elements. Recall that we may have $(-1,-1)$ and $(1,1)$ as direction vectors in a legal MAC and so it is all right to have directions $(-1,1)$ and $(1,-1)$, but neither of these pairs are acceptable as distance vectors (Definition 4.3). In [GS91] different combinations of direction vectors in legal MACs are enumerated. There are $\sum_{i=1}^{8} \left(\begin{array}{c} 8 \\ i \end{array} \right) = 255$ sets of direction vectors from which 37 sets contain more than five elements and hence are illegal. By verification, from the other 218 sets that contain up to five directions, 124 sets are also illegal. Thus, there are 94 sets of direction vectors in legal MACs.

Since direction of any element in D that lies between the edges e_1 and e_2 of a distance cone does not matter, the number of cases to be considered can be further reduced to 34 sets. To give a compact representation of these 34 possibilities, let us define the set

$$D^{''} = \{(s_1 s_1^{'},\ s_2 s_2^{'}) \mid (s_1,\ s_2) \in D^{'} \ \text{ and } \ (s_1^{'},\ s_2^{'}) \in D^{'}\}.$$

Although $D^{''}$ does not exactly give the directions of e_1 and e_2, it classifies all the 34 direction sets into four cases. This classification is not only useful to analyze and understand distance cones in MACs, but also helps elegantly to conduct the proof steps of Theorem 5.2.

[3] We ignore direction vector $(0, 0)$.



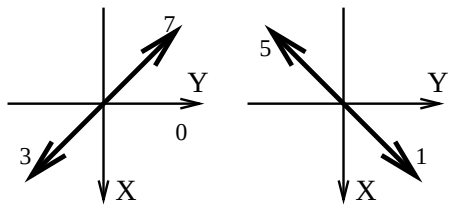
For all directions (s_1, s_2) and $(s_1^{'}, s_2^{'})$ in a MAC, we define $D^{''}$ as the set of ordered pairs $(s_1 s_1^{'},\ s_2 s_2^{'})$. If $(-1,\ -1) \in D^{''}$, then Case 1; otherwise, if $(-1, 0)$ or $(0, -1)$ is in $D^{''}$, then Case 2; otherwise, if $(-1, 1)$, $(1, -1)$ or $(0,\ 0)$ is in $D^{''}$, then we have Case 3. otherwise, it is Case 4 where one of $(1, 0)$, $(0, 1)$ or $(1, 1)$ exists in $D^{''}$.

Figure 5: The hierarchies of the cases defined by $D^{''}$

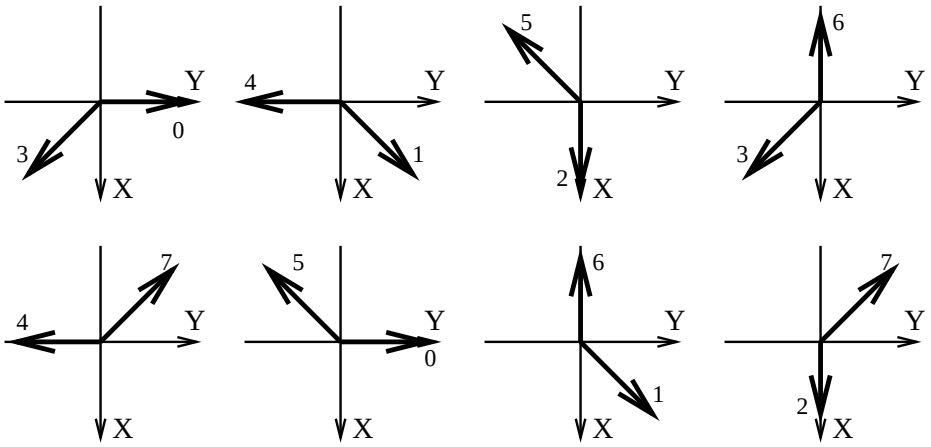
Depending on the elements in $D^{''}$ we distinguish four cases. The relation between these cases is shown in Figure 5. As we can see, the classification suggested by $D^{''}$ has a hierarchical structure. The most general case where $D^{'}$ *may* contain up to five elements is denoted as Case 1. The cases where $D^{'}$ may contain up to four, three, and two elements are denoted as Case 2, Case 3, and Case 4, respectively. Let us explain how $D^{''}$ can be useful. Given a set D of distance vectors in a legal MAC, we can determine $D^{'}$ and use it to define $D^{''}$ as specified. Then, we test if $(-1, -1)$ exists in $D^{''}$. If the answer is positive, then we have Case 1. Otherwise, we check whether we have $(-1, 0) \in D^{''}$ or $(0, -1) \in D^{''}$. If the answer is affirmative, then we have Case 2. It is important to note that at this point, $D^{''}$ *may not* contain both of the elements $(-1, 0)$ and $(0, -1)$. Otherwise, as in the following, we can show that $D^{''}$ will also contain $(-1, -1)$ and hence we have Case 1.

To be more clear, let us assume that $D^{'}$ contains directions 0 and 3. Thus, we have $(0, -1) \in D^{''}$. In this case, if $D^{''}$ also contains the element $(-1, 0)$, then from any pair of directions that may produce such an element[4], directions 2 and 7 can only coexist with 0 and 3 in $D^{'}$, since existence of any other such pairs with 0 and 3 in $D^{'}$ means to have an illegal MAC. Therefore, since $D^{'}$ contains directions 3 and 7, $D^{''}$ must contain $(-1, -1)$ and hence we have Case 1. This type of reasoning applies to make sure that we do not examine a case until we are sure that it does not belong to more general cases. If we do not have Case 2, then we test whether one of the elements $(0, 0)$, $(-1, 1)$, or $(1, -1)$ is in $D^{''}$. We have Case 3 if the condition holds. Otherwise, we have Case 4 where $D^{''}$ contains one of the elements $(0, 1)$, $(1, 0)$, or $(1, 1)$. In the following, we decode the meaning of the elements that may exist in $D^{''}$.

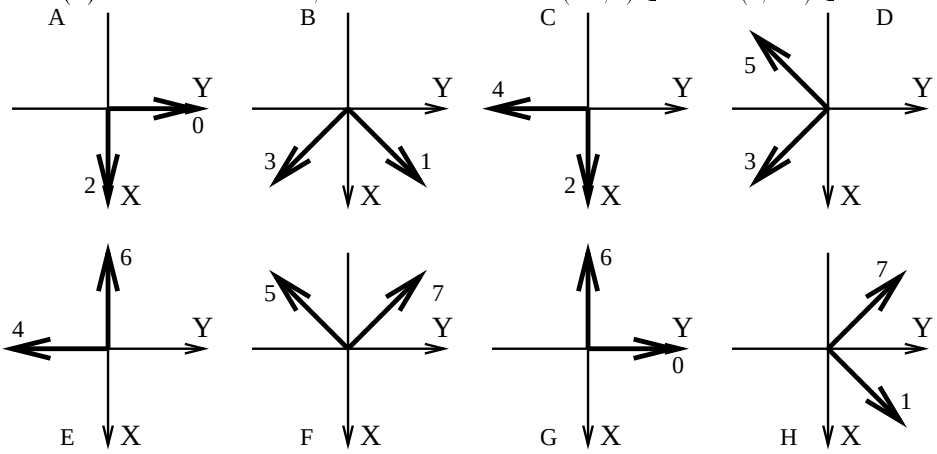
[4]Other pairs of direction vectors that produce $(-1, 0)$ in $D^{''}$ are 2 and 5, 1 and 6, and 3 and 6.



(a) Instances of Case 1, where $(-1,-1) \in D^{''}$.



(b) Instances of Case 2, where we have either $(-1,0) \in D^{''}$ or $(0,-1) \in D^{''}$.



(c) Instances of Case 3, where $D^{''}$ contains one of $(-1,1)$, $(1,-1)$, or $(0,0)$.

Figure 6: Directions of edges of a distance cone

Case 1. If $(-1,-1) \in D^{''}$, then we have $\{(-1,-1),(1,1)\} \subseteq D^{'}$ or $\{(-1,1),(1,-1)\} \subseteq D^{'}$, but not both (Figure 6(a)). It is clear that if we have both of these cases, then the MAC is illegal. In any of these two cases, where direction vectors (but not distance vectors) form a half plane, $D^{'}$ may also contain those directions that are on the same side of the half plane (Figure 4). For instance, in the first case where $D^{'}$ contains directions 1 and 5, it may also contain

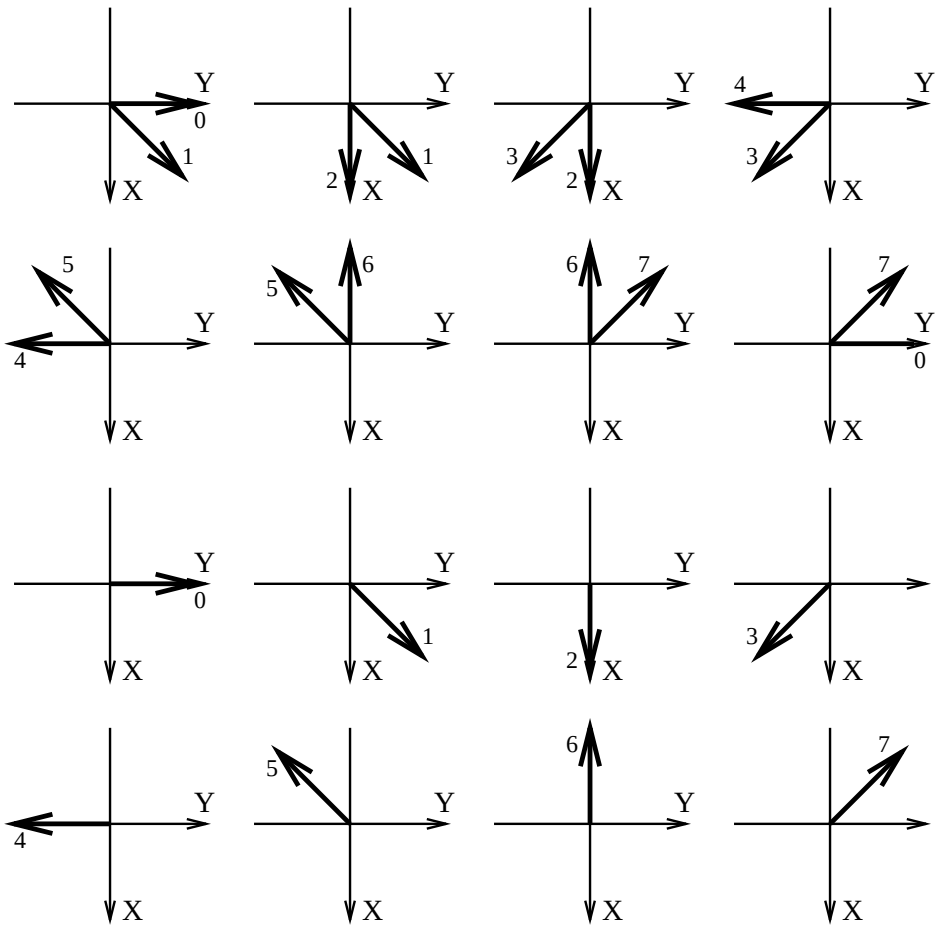
either directions 2, 3, and 4, or directions 6, 7, and 0. In the second case where $D^{'}$ contains $(-1,1)$ and $(1,-1)$, it may also contain directions 4, 5, and 6, or directions 0, 1, and 2. In other words, the set of direction vectors $D^{'}$ in Case 1 may contain at most five elements. It can be easily verified that from 94 possible sets of direction vectors in legal MACs, 30 sets are included in Case 1. This can be shown by considering the fact that direction of any distance vector that is not an edge of the cone may or may not exist in $D^{'}$. For example, if directions 1 and 5 are in $D^{'}$, then any combination of directions 2, 3, and 4 may also be in $D^{'}$. This gives us eight sets of direction vectors.[5] Similarly, there are eight other combinations of directions 0, 6, and 7. Thus, since the set that contains only directions 1 and 5 are counted twice, we have 15 direction sets. Likewise, we get 15 direction sets if $D^{'}$ contains 3 and 7. Therefore, Case 1 includes 30 sets of direction vectors. If $(-1,-1) \notin D^{''}$, then we can proceed to examine Case 2 in which $D^{'}$ may contain at most four elements.

Case 2. If we have $(0,-1) \in D^{''}$ or $(-1,0) \in D^{''}$, *but not both*, then directions of e_1 and e_2 are i and $i +_8 3$, where $0 \leq i \leq 7$ and $+_8$ denotes addition modulo 8. In each of these eight cases illustrated in Figure 6(b), directions $i +_8 1$ and $i +_8 2$ that lie between i and $i +_8 3$ may also be in $D^{'}$. For example, if $i = 0$, then directions 0 and 3 are in $D^{'}$ and so we have $(0,-1) \in D^{''}$. In this case, direction vectors 1 and 2 which lie between 0 and 3 may also exist in $D^{'}$ (see Figure 4). This is the only case that we may face in imperative language loops in which dependence vectors are positive. The number of direction sets that Case 2 represents is 32. This can be shown by considering that in any of the eight cases, any of $\sum_{i=0}^{2} \begin{pmatrix} 2 \\ i \end{pmatrix} = 4$ combinations of non-edge directions may also be in $D^{'}$. Recall that we reach this point only if we do not have Case 1. If none of the elements $(-1,-1)$, $(-1,0)$, and $(0,-1)$ are in $D^{''}$, then we examine the next case.

Case 3. If $D^{''}$ contains one of the elements $(0,0)$, $(1,-1)$, or $(-1,1)$, then we have eight possible cases in which $D^{'}$ contains directions i and $i +_8 2$ (and possibly $i +_8 1$), where $0 \leq i \leq 7$ and i, $i +_8 2$ are directions of the edges e_1 and e_2. That is, in any of these cases shown in Figure 6(c), $D^{'}$ contains two or three elements. For instance, if directions 0 and 2 are in $D^{'}$, then direction 1 which lies between 0 and 2 may also be in $D^{'}$. Since in each of these eight cases $D^{'}$ may also contain direction $i +_8 1$ which lies between i and $i +_8 2$, Case 3 includes 16 direction sets. If all the previous cases have failed to be considered, then we have Case 4 as follows.

Case 4. If $D^{''}$ contains $(0,1)$, $(1,0)$, or $(1,1)$, then $D^{'}$ contains either directions i and $i +_8 1$, or it just contains direction i, where $0 \leq i \leq 7$. The first two top rows in Figure 7 illustrates the former case where directions of edges e_1 and e_2 are i and $i +_8 1$. In the latter case that is shown in the bottom two rows in Figure 7, direction of any existing distance vector is i. We have grouped as Case 4, all these 16 direction sets in which $D^{'}$ contains one direction vector or two consecutive directions. Clearly, the size of the distance cone in any of these instances is less than 90°.

[5] $\sum_{i=0}^{3} \begin{pmatrix} 3 \\ i \end{pmatrix} = 8.$

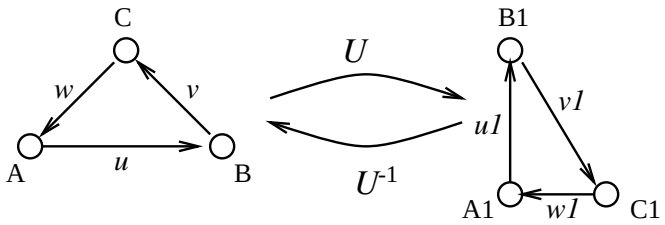


Direction sets defined in Case 4, where D'' contains $(0,1)$, $(1,0)$, or $(1,1)$. That is, D' contains directions i and $i +_8 1$ (the first two top rows), or it contains one direction vector. In any of these instances, the size of the distance cone is less than $90°$.

Figure 7: Direction vectors in Case 4

It is important to recall that in any of the 94 instances considered above, distance vectors form a cone of size less than π. Moreover, directions of the edges e_1 and e_2 are elements in D' that contribute to form a particular element in D''. As an example, if D' contains directions $(-1,-1)$, $(1,0)$, and $(1,1)$, then D'' contains $(-1,-1)$, $(1,0)$, $(1,1)$, and $(-1,0)$. Since $(-1,-1) \in D''$, we have Case 1 and directions of e_1 and e_2 are $(-1,-1)$ and $(1,1)$. As mentioned earlier, we will use this classification introduced by D'' in the proof steps of Theorem 5.2.

Next, we give Lemmas 4.1 and 4.2. In the former, we show that a unimodular transformation of a legal array constructor does not create cycles among the array elements. In the latter, we prove that cycles in an illegal array constructor can not be broken by any unimodular transformation.



There is a cycle among A, B and C on the left. The cycle remains among the transformations of these elements by any unimodular transformation.

Figure 8: Unimodular transformations by U and U^{-1}

Lemma 4.1 *Let $D = \{d_1, \ldots, d_k\}$ be the set of distance vectors in a legal array constructor. Then D is transformed into a legal set by any unimodular matrix.*

Proof. We prove by contradiction. Assume that a unimodular matrix U exists, by which D is transformed into an illegal set. Let $D' = \{d_1 \cdot U, \ldots, d_k \cdot U\}$ be the transformed set of distance vectors. Then, by Definition 4.3 there exist non-negative integers $\alpha_1, \ldots, \alpha_k$, not all zeros, such that $(\alpha_1 d_1 + \cdots + \alpha_k d_k) \cdot U = \mathbf{0}$. If both sides of the equation are multiplied by U^{-1} from the right, it concludes that D is illegal and hence a contradiction. □

Lemma 4.2 *Let $D = \{d_1, \ldots, d_k\}$ be the set of distance vectors in an illegal array constructor. Then D is transformed into an illegal set by any unimodular matrix.*

Proof. Since D is illegal, by Definition 4.3, there exist non-negative integers $\alpha_1, \ldots, \alpha_k$ not all zeros, such that $\sum_{i=1}^{k} \alpha_i d_i = \mathbf{0}$. Consider the transformation of D into $D' = \{d'_1, \ldots, d'_k\}$ by a unimodular matrix U. If we multiply both sides in $\sum_{i=1}^{k} \alpha_i d_i = \mathbf{0}$ from the right by U, we will have $\sum_{i=1}^{k} \alpha_i d'_i = \mathbf{0}$, where d'_i denotes $d_i \cdot U$. That is, D' is illegal with the same αs taken for D. □

Intuitively, the proof follows from the fact that any unimodular transformation is a bijection between a set and the transformed set. For instance, consider the transformation of $D = \{u, v, w\}$ into $D' = \{u1, v1, w1\}$ by a unimodular matrix U in Figure 8. Clearly, the elements in D' can be transformed back into the elements in D by U^{-1}.

Theorem 4.1 *Let D be the set of distance vectors in a monolithic array constructor. Then the constructor is legal, iff there exists a unimodular matrix U such that $\forall d_i \in D : d_i \cdot U > \mathbf{0}$.*

Proof. The *if* Part. First we prove that if the constructor is legal, then there exists a unimodular matrix U by which any element in D are transformed into a positive vector. If

$\forall d_i \in D$ we have $d_i > 0$, then we can take U to be the identity matrix. If $\forall d_i \in D$ we have $d_i < 0$, then we can take the negated identity matrix as U. Otherwise, D contains positive and negative vectors. If for any unimodular matrix U, there exist distance vectors d_i and d_j in D such that $d_i \cdot U > \mathbf{0}$ and $d_j \cdot U < \mathbf{0}$, then by Lemma 4.2, we can conclude that D is illegal and hence a contradiction.

The *only if* Part. Now, we prove that if a unimodular matrix U exists such that $\forall d_i \in D$ we have $d_i \cdot U = d_i^{'} > 0$, then the constructor is legal. If this is not true, then non-negative integers $\alpha_1, \ldots, \alpha_k$ not all zeros, exist such that $\alpha_1 d_1 + \cdots + \alpha_k d_k = \mathbf{0}$. Multiplying both sides of the equation by U from the right, we get $\alpha_1 d_1^{'} + \cdots + \alpha_k d_k^{'} = \mathbf{0}$, which is impossible since each $d_i^{'} > 0$. $\square$

The above theorem is important in the sense that given an n-dimensional MAC, it proves the existence of a unimodular matrix U by which all distance vectors in the MAC can be transformed into positive vectors. Therefore, having found such a matrix, we can apply the parallelizing algorithms designed for imperative loops to transform the MAC into a loop nest. The resulting parallelized program, while executed, preserves the semantics of the MAC. In [GS91], a conceptually two-step transformation is applied to find a safe parallel evaluation order of the elements in a two-dimensional MAC. In the first step, a unimodular matrix U is found by which distance vectors in the MAC are transformed into positive vectors. In the second step, parallelizing algorithms designed for imperative language loops are applied to the result of the first step. However, when both of the above steps are based on unimodular transformations, they can be merged into a one-step process. In Section 5 we will introduce parallelizing algorithms that directly determine a safe parallel order of evaluation of the array elements in the form of a parallelized double loop.

4.3 Safe Order of Evaluation

In a monolithic array constructor, no explicit order of evaluation of the array elements is specified. One of the objective is to find an evaluation order such that the semantics of the program be preserved. Let us formally define the notion of a *safe order*.

Definition 4.4 *An evaluation order $\prec$ of the elements in an array constructor is safe iff for all the index points defined by the constructor, if there is a dependence from $[i^{'}, j^{'}]$ to $[i, j]$, then $[i^{'}, j^{'}] \prec [i, j]$. That is, the array element at the index point $[i^{'}, j^{'}]$ must be defined before the array element at $[i, j]$.*

Let us examine the lexicographic order of the index vector (i, j) as an evaluation order of the elements in an array constructor. Recall that, unlike in imperative language loops, there is no explicitly defined order for a MAC. Thus, this lexicographic order may not be safe.

Lemma 4.3 *For an array constructor, the lexicographic order $\prec$ of its index vector (i, j) is a safe order, iff any distance vector (or equivalently, any direction vector) in the constructor is positive.*

Proof. Let (d_1, d_2) denote a distance vector in the MAC. That is, by Definition 4.2, there are index points $[i, j]$ and $[i', j']$ such that $(d_1, d_2) = (i - i', j - j')$. If $(d_1, d_2) > \mathbf{0}$, then we have $[i', j'] \prec [i, j]$. That is, the array element at the index point $[i', j']$ must be defined before the element at $[i, j]$. Thus, the lexicographic order of the index vector (i, j) is a safe order of evaluation of the array elements. Let us assume that $\prec$ is a safe order. By Definition 4.4, for all the index points defined by the MAC, if there is a dependence from $[i^{'}, j^{'}]$ to $[i, j]$, then we have $[i^{'}, j^{'}] \prec [i, j]$. That is, there is a dependence in the constructor with distance vector $(d_1, d_2) = (i - i', j - j') > \mathbf{0}$. □

In the previous chapter, we described the structure of a double loop construct. There, we specified that a double loop executes in the lexicographic order of its index vector (the conventional sequential order). We also said that it can be determined if the outer loop or the inner loop in a double loop could be executed in parallel. We will produce an equivalent double loop as the result of the transformation of a MAC under a unimodular matrix. Let us define this notion.

Definition 4.5 *The transformation (L_1^U, L_2^U) of a MAC by a unimodular matrix U is equivalent to the MAC, if the lexicographic order of the index vector of (L_1^U, L_2^U) is a safe evaluation order for the MAC.*

Note that what we benefit by expressing a MAC as a double loop is the explicit execution order of the double loop. That is, the MAC and the produced double loop are equivalent in the sense that in the definition of any element of the array, the double loop uses the same elements that the MAC would use. In other words, dependences in the MAC is preserved by evaluating the array elements in the order that the double loop executes. This fact is formally stated in the following.

Corollary 4.1 *Let D be the set of distance vectors in a legal MAC and U be a unimodular matrix. Then, the transformation (L_1^U, L_2^U) of the MAC by U is equivalent to the MAC, if any distance vector in D is transformed into a positive vector by U.*

Proof. There is a one to one correspondence between the index points of the MAC and the iteration points of (L_1^U, L_2^U), since unimodular transformation is a bijection. For all $(d_1, d_2) \in D$, since $(d_1, d_2) \cdot U > \mathbf{0}$ is a distance vector in (L_1^U, L_2^U), by Lemma 4.3, the lexicographic order of the index vector in (L_1^U, L_2^U) is a safe evaluation order of the array elements. That is, to define an array element, the MAC and the double loop use the same elements of the array, and hence they are equivalent. □

5 Unimodular Transformations of MACs

From Corollary 4.1 we can determine if the transformation (L_1^U, L_2^U) of a MAC by a unimodular matrix U is *equivalent* to the MAC. In other words, we can decide if dependences among the array elements in a MAC are not violated by evaluating the elements in the (sequential) order that (L_1^U, L_2^U) executes. The outer loop or the inner loop in (L_1^U, L_2^U) can run in parallel, iff there exists no dependence in the double loop at level 1 or level 2, respectively. Given a MAC, the question now is: Does there exist a unimodular matrix U such that transformation of the MAC by U can be expressed as a double loop (L_1^U, L_2^U) in which the outer loop or the inner loop can run in parallel?

In the next two sections we answer the question in forms of two theorems. In Section 5.1, Theorem 5.1 proves that for a given MAC, there exists a unimodular matrix U such that transformation of the MAC by U can be expressed as a double loop in which the outer loop can run in parallel, iff all distance vectors in the MAC are parallel to each other and have the same direction. Algorithm 5.1 is then presented that for such a MAC, efficiently finds a desired matrix U and produces the parallelized double loop. In Section 5.2, we discuss the problem of finding U such that in the transformation (L_1^U, L_2^U) of a MAC by U, the inner loop L_2^U can run in parallel. We formulate the problem in Section 5.2.1. Examples are given in Section 5.2.2 to illustrate the objectives. In Section 5.2.3, we first provide a basis for Theorem 5.2 in forms of two lemmas. Then, we give Theorem 5.2 that proves for a legal MAC, there always exists a unimodular matrix U such that we can express the result of the transformation of the MAC caused by U as a double loop in which the inner loop can be executed in parallel. In Section 5.2.4, we propose Algorithm 5.4 that a given MAC, efficiently finds U and produces a desired double loop. *AC-Transformer* [6], an implementation of our parallelizing algorithms will be presented in Chapter 6. Given the distance vectors and the index bounds in a MAC, the program determines a unimodular matrix and produces an equivalent parallelized double loop. This provides an efficient compile-time solution to the scheduling and synchronizing problem that arises in the parallel implementation of MACs.

5.1 Outer Loop Parallelization

The outer loop in a double loop L can run in parallel, iff there is no dependence at level 1. In this case, direction of any existing distance vector is either $(0,0)$ or $(0,1)$. In the former case, there is no dependence in L. In the latter case, all distance vectors in L are parallel to each other. In any case, let us assume that L is the result of transforming a MAC by a unimodular matrix U (see Section 3). Thus, transformation of distance vectors in L by U^{-1} gives the original distance vectors in the MAC, where U^{-1} denotes the inverse of U. Since unimodular transformation is linear, we conclude that distance vectors in the MAC must also be parallel to each other and have the same direction. In other words, if the cone size is $0°$, we can find

[6] For Array Constructors Transformer.

a unimodular matrix U by which transformation of the MAC can be expressed as a double loop in which the outer loop can run in parallel. In the following theorem , we formally state what we intuitively discussed above. The theorem is an extension of the outer loop parallelizing theorem in [Ban90]. This extended theorem allows also negative distance vectors that may exist in MACs.

Theorem 5.1 *There exists a 2×2 unimodular integer matrix U such that the transformation (L_1^U, L_2^U) of a MAC by U is equivalent to the MAC and the outer loop L_1^U can be executed in parallel, iff all distance vectors in the MAC are of the form $\alpha(a_1, a_2)$, where $\alpha > 0$ is any integer and (a_1, a_2) is an integer vector.*

Proof. Let D denote the set of distance vectors in the MAC.
The *if* part. Assume that all distance vectors $(d_1, d_2) \in D$ are of the form $\alpha(a_1, a_2)$, where α is a positive integer and (a_1, a_2) is an integer vector. We will show how to define U. Let $g = \gcd(a_1, a_2)$. Clearly, we have $g > 0$, since both of a_1 and a_2 are not zero. A desired unimodular matrix U is such that we have $(a_1, a_2) \cdot U >_2 \mathbf{0}$. That is, the following relations are to be satisfied.

$$\begin{cases} (1) & a_1 u_{11} + a_2 u_{21} = 0 \\ (2) & a_1 u_{12} + a_2 u_{22} > 0 \\ (3) & det(U) = u_{11} u_{22} - u_{12} u_{21} = 1. \end{cases}$$

The general solution to (1) is $(u_{11}, u_{21}) = t(\frac{a_2}{g}, -\frac{a_1}{g})$, where t is any integer. Substituting for u_{11} and u_{21} in (3), we get $t(\frac{a_2}{g} u_{22} + \frac{a_1}{g} u_{12}) = 1$. From this equation and (2) we get $t = 1$ and $\frac{a_2}{g} u_{22} + \frac{a_1}{g} u_{12} = 1$. Thus, we take $(u_{11}, u_{21})^T = (\frac{a_2}{g}, \frac{-a_1}{g})^T$. For the second column in U, we can obtain a pair of integers u_{12} and u_{22} such that $a_1 u_{12} + a_2 u_{22} = g$, to satisfy (3). We can apply the Extended Eucledean algorithm (EE) [Knu73] to find such a pair. The transformation of (d_1, d_2) by U is

$$(d_1, d_2) \cdot U = \alpha(a_1, a_2) \cdot U = \alpha(a_1 u_{11} + a_2 u_{21}, a_1 u_{12} + a_2 u_{22}) = (0, \alpha g) >_2 \mathbf{0}.$$

Since any distance vector in D is transformed into a positive vector by U, by Corollary 4.1, the transformation (L_1^U, L_2^U) of the MAC is equivalent to the MAC. Since there is no dependence at level 1 in (L_1^U, L_2^U), its outer loop L_1^U can run in parallel.

The *only if* part. Assume that a unimodular matrix U exists such that in the double loop (L_1^U, L_2^U), the outer loop L_1^U can run in parallel. We will show that any distance vector in D is of the form $\alpha(a_1, a_2)$, where α is a positive integer and (a_1, a_2) is an integer vector. Since there is no dependence at level 1 in (L_1^U, L_2^U), for all $(d_1, d_2) \in D$, we have $(d_1, d_2) \cdot U >_2 \mathbf{0}$. That is, the following relations are satisfied:

$$\begin{cases} (1) & d_1 u_{11} + d_2 u_{21} = 0 \\ (2) & d_1 u_{12} + d_2 u_{22} > 0. \end{cases}$$

From (1) we get $d_1 = -t u_{21}$ and $d_2 = t u_{11}$, where t is any integer. (2), we get $t\Delta > 0$, where $\Delta = det(U)$. Since $\Delta^2 = 1$, we have $(d_1, d_2) = t(-u_{21}, u_{11}) = t\Delta^2(-u_{21}, u_{11}) = (t\Delta) \cdot$

$\Delta(-u_{21}, u_{11}) = \alpha(a_1, a_2)$, where $\alpha > 0$. That is, any distance vector in D is of the form $\alpha(a_1, a_2)$. $\square$

Now, an algorithm is given that is an extension of the outer loop palarrelizing algorithm in [Ban90] to also allow negative distance vectors. For a given MAC, the extended algorithm finds a unimodular matrix U (if exists) such that transformation of the MAC by U can be expressed as a double loop (L_1^U, L_2^U) in which the outer loop can be executed in parallel. We have slightly modified step 4 which causes the inner loop in (L_1^U, L_2^U) to vary over a smaller range of values (see [Shi92], for details).

Algorithm 5.1 *Given a legal MAC, this algorithm finds a unimodular matrix U such that:*
(1) The transformation (L_1^U, L_2^U) is equivalent to the MAC.
(2) The outer loop L_1^U can run in parallel.
(3) The iteration count of the outer loop L_1^U is maximized.

1 Compute the set D of distance vectors in the MAC.

2 If $D = \emptyset$ or $D = \{(0,0)\}$, then take U to be

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \text{if } (N_1 - n_1) \geq (N_2 - n_2),$$

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad \text{otherwise.}$$

go to step 5.

3 If all the elements in D are not of the form $\alpha(a_1, a_2)$, where α is any positive integer and (a_1, a_2) is a fixed integer vector, then terminate the algorithm with the message: *No such a unimodular transformation exists.*

4 If $|a_1| = |a_2|$,
then begin:

take $(u_{11}, u_{21}) = (sig(a_2), sig(-a_1))$.

take (u_{12}, u_{22}) to be

$(0, sig(a_2))$ if $(N_1 - n_1) \geq (N_2 - n_2)$,

$(sig(a_1), 0)$ otherwise.

go to step 5.

end

otherwise, apply the EE algorithm to compute $g = \gcd(a_1, a_2)$ and a pair of integers u_{12} and u_{22}, such that $a_1 u_{12} + a_2 u_{22} = g$.

Let $u_{11} = \frac{a_2}{g}$, $u_{21} = \frac{-a_1}{g}$, and $U = \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix}$.

5 Apply Algorithm 4.1 in [Ban90]. to compute the transformation (L_1^U, L_2^U) of the MAC by U and mark the outer loop L_1^U as DOALL.

Example 5.1 Consider the following example of a MAC.

A = **For** $(i,\ j)$ **in** $[5:100,\ 5:110]$

$A[i+1,\ j-2] + A[i+2,\ j-4]$

EndFor

The set of distance vectors in the constructor is $D = \{(-1,2),(-2,4)\}$. We see that $(-1,2) = 1(-1,2)$ and $(-2,4) = 2(-1,2)$. That is, any element in D is a multiple of $(a_1, a_2) = (-1,2)$. Then by Theorem 5.1, there exist a unimodular matrix U such that the outer loop in the transformation (L_1^U, L_2^U) can run in parallel. Applying the EE algorithm, we get $\gcd(-1,2) = 1$ and the integer pairs $u_{12} = -1$ and $u_{22} = 0$. Taking $u_{11} = \frac{a_2}{\gcd(a_1,a_2)} = 2$ and $u_{21} = \frac{-a_1}{\gcd(a_1,a_2)} = 1$, we get

$$U = \begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix}.$$

Now, we apply algorithm 4.1 in [Ban90] to produce the following double loop (L_1^U, L_2^U), caused by transformation of the MAC under the matrix U.

```
DOALL K1 = 15, 310
   DO K2 = Ceil (Max (-100, (5-K1)/2.0)),
           Floor(Min (-5, (110-K1)/2.0)), 1

      A(-K2, K1+2*K2) = A(-K2+1, K1+2*K2-2) + A(-K2+2, K1+2*K2-4)

   End
End
```

Note that we have $(-1,2)\cdot U = (0,1)$ and $(-2,4)\cdot U = (0,2)$. Since distance vectors in the MAC are transformed into positive vectors by U, by Corollary 4.1, the nest (L_1^U, L_2^U) is equivalent to the given MAC. Since there is no dependence at level 1 in the resulting nest, the outer loop L_1^U can run in parallel and hence marked as DOALL.

The number of iterations of the outer loop in (L_1^U, L_2^U), the transformation of a double loop by a unimodular matrix U, is given to be

$$(N_1 - n_1)\cdot|u_{11}| + (N_2 - n_2)\cdot|u_{21}| + 1$$

where u_{11} and u_{21} are the elements in the first column of U [Ban90]. The integers n_1, N_1 are the bounds of the outer loop, and n_2, N_2 are the bounds of the inner loop in the original double loop. Due to the similarities, the above formula can also be used to obtain the number of iterations of the outer loop in the transformation of a MAC by a unimodular matrix U. We can use the same argument provided in [Shi92] to show that the above algorithm performs optimal transformations. That is, the number of iterations of the outer loop in (L_1^U, L_2^U) is maximized.

5.2 Inner Loop Parallelization

In this section we present Theorem 5.2 that for a legal MAC proves the existence of a unimodular matrix U such that transformation of the MAC by U can be expressed as double loop (L_1^U, L_2^U) in which the inner loop can run in parallel.

5.2.1 Problem Statement

The problem can be formulated as the following. Given the set D of distance vectors in a legal MAC, we are interested to find a unimodular matrix

$$U = \begin{bmatrix} x & x' \\ y & y' \end{bmatrix}$$

such that:

(1) $\forall (d_1, d_2) \in D$, we have $xd_1 + yd_2 > 0$ and

(2) the *objective function* $f(x, y) = c_1|x| + c_2|y|$ be minimized,

where the coefficients $c_1 = N_1 - n_1 > 0$ and $c_2 = N_2 - n_2 > 0$ are integer constants. Recall that n_1, N_1, n_2 and N_2 are the index bounds of the MAC. The inner loop in (L_1^U, L_2^U) can run in parallel, iff there is no dependence at level 2 in the double loop. In other words, the first component in any distance vector in (L_1^U, L_2^U) must be positive, which is formulated as constraint (1). As mentioned before, the number of iterations of the outer loop in (L_1^U, L_2^U) is a function of x and y, where $(x, y)^T$ is the first column of the transformation matrix U. We say that the transformation (L_1^U, L_2^U) in which L_2^U can run in parallel is optimal, if the number of iterations of L_1^U is minimized. This is formulated as constraint (2). Note that the objective function $f(x, y)$ is non-linear.

Let us define some terms that will be used in this chapter. We refer to a pair of integers x and y that satisfy constraint (1) as a *candidate* pair. A subset of $\mathcal{Z}^2$ that contains only such pairs of x and y is called a *solution space*, where $\mathcal{Z}$ denotes the integers. We refer to a candidate (x, y) as an *ideal* pair, if constraint (2) is also satisfied.

Note that the problem of integer linear programming is, in general, $\mathcal{NP}$-complete [Sch86]. In such problems the objective functions are linear. But, when the number of variables is fixed

to be n, as $n = 2$ in our problem, there is a polynomial (in the size of the numbers involved) algorithm for solving a system of linear inequalities in n integer variables ([Sch86]). However, we develop our solution method for this problem and show that the ideal pair (1) exists and (2) can be efficiently obtained.

To determine U, we first obtain integers x and y. Then, we use x and y to obtain integers x' and y' such that U be unimodular (see the EE algorithm). Let us assume that distance vectors $e_1 = (d_1, d_2)$ and $e_2 = (d_1', d_2')$ in D are identified as edges of the distance cone defined by D (see Algorithm 5.3). That is, any other element in D either lies inside the cone or coincides with an edge. In this case, the above problem can be reduced to the problem of finding integers x and y such that the relations

$$\begin{cases} d_1 x + d_2 y > 0 \\ d_1' x + d_2' y > 0, \end{cases}$$

are satisfied and the objective function $f(x, y)$ is minimized. This is because transformation of any element in D by U either coincides with $e_1' = e_1 \cdot U >_1 \mathbf{0}$ or $e_2' = e_2 \cdot U >_1 \mathbf{0}$, or lies inside the cone whose edges are e_1' and e_2'. That is, any element in D is transformed, by U, into a vector whose first component is positive (Corollary 2.2.1 in [Shi92])[7].

5.2.2 Motivating Examples

Before formulating a solution, let us consider two examples which help illustrate our objectives and the essential steps to achieve them. In either example, we want to find a unimodular matrix U such that constraints (1) and (2) specified in Section 5.2.1 be satisfied. That is, for any distance vector (d_1, d_2) in the MAC, we want to have $(d_1, d_2) \cdot U >_1 \mathbf{0}$ and the objective function $f(x, y)$ be minimized, where the ideal pair $(x, y)^T$ is the first column of U. As the first example, consider the distance cone defined by $D = \{(-1, 1), (-1, -2), (0, 1)\}$, where edges of the cone are $e_1 = (-1, -2)$ and $e_2 = (0, 1)$. Since $(0, -1) \in D''$, we have an instance of Case 2 (see Section 4.2). The relations that should be satisfied are

$$\begin{cases} -x - 2y > 0 \\ y > 0. \end{cases}$$

As we can see, any positive integer taken as y satisfies the second relation, from which we take $y = 1$, the possible minimum integer that also satisfies constraint (2). Substituting for y in the first relation, we derive that any integer less than -2 can be taken as x. Thus, direct computation delivered $(-3, 1)^T$ as the ideal pair for the first column of U. As its second column, we can take either $(1, 0)^T$ or $(-1, 0)^T$. Therefore, a desired unimodular matrix is $U = \begin{bmatrix} -3 & 1 \\ 1 & 0 \end{bmatrix}$. The transformations of the edges e_1 and e_2 by U are $e_1' = (1, -1) >_1 \mathbf{0}$ and

[7] Corollary 2.2.1. Let u, v and w be different vectors in R^2 such that they form a cone and v lies between u and w. Let also U be a unimodular matrix. Then, the transformations of u, v and w by U also form a cone and $v1 = v \cdot U$ lies between $u1 = u \cdot U$ and $w1 = w \cdot U$.

$e_2' = (1,0) >_1 \mathbf{0}$, respectively. The element $(-1,1)$ in D is transformed into $(4,-1) >_1 \mathbf{0}$ by U. As we note, no dependence exits at level 1 in the transformed vectors and the objective function $f(x,y) = c_1|x| + c_2|y| = 3c_1 + c_2$ is minimized. Recall that $c_1 > 0$ and $c_2 > 0$ are integer constants. In Figure 9, the distance cone and its transformation are shown at the top.

As the second example, let $D = \{(-2,-3),(1,1),(0,-1),(2,2)\}$ be the set of distance vectors in a MAC. In Figure 9, the figure at the left-bottom illustrates the distance cone in the X-Y plane, where $e_1 = (-2,-3)$ and $e_2 = (1,1)$ are the edges of the cone. In this case, we have $(-1,-1) \in D''$ (Case 1 in Section 4.2). Clearly, $(2,2) \in D$ could also be taken as e_2. The integers x and y should satisfy the relations

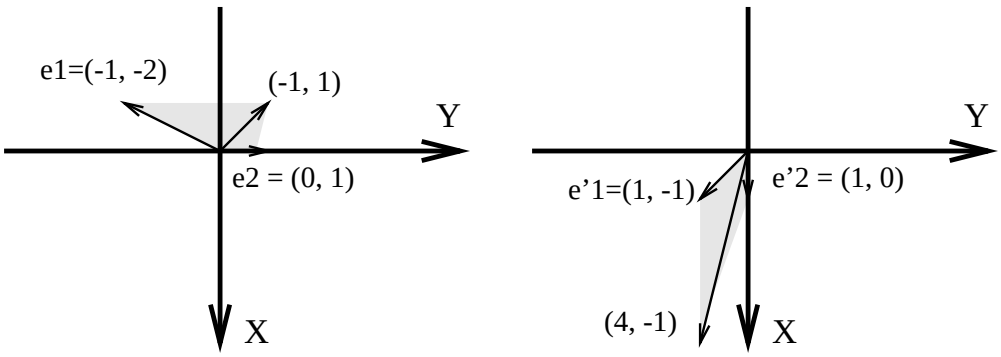
$$\begin{cases} -2x - 3y > 0 \\ x + y > 0. \end{cases}$$

Direct computation gives $x = 4$ and $y = -3$ as the ideal pair, by which the above relations are satisfied and the objective function $f(x,y)$ is minimized. In other words, if an integer pair (x_0, y_0) is also a candidate, we have $|x_0| \geq |x|$ and $|y_0| \geq |y|$, and accordingly $f(x_0,y_0) \geq f(x,y)$. Note that in this example, we obtained $|x| > 1$ and $|y| > 1$. In the inner loop parallelizing algorithm for imperative double loop in [Ban90] we would always get $|y| \leq 1$. As the second column of U, we can take $(-1,1)^T$. Now, let us examine the transformation of the elements in D caused by U. The elements $(-2,-3),(1,1),(0,-1)$, and $(2,2)$ are transformed into $(1,-1),(1,0),(3,-1)$, and $(2,0)$, respectively. As expected, the transformed elements also form a distance cone that is shown at the right-bottom in Figure 9, where the size of the cone is less that π and its edges are $e_1' = (1,-1)$ and $e_2' = (1,0)$, the transformations of e_1 and e_2, respectively. The element $(0,-1) \in D$ that lies inside the original distance cone is transformed into $(3,-1)$ that lies inside the transformed cone. The vector $(2,0)$ that is the transformation of $(2,2)$, coincides with $(1,0)$, the transformed $(1,1)$.

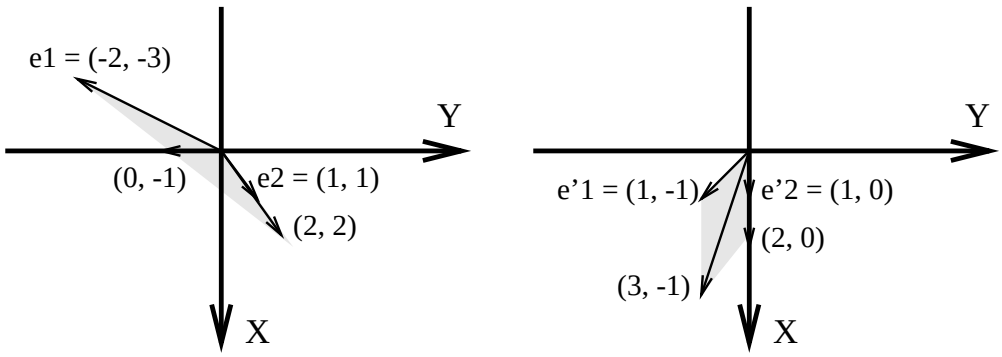
In Figure 10, the solution spaces of the examples are shown. The figure on the left shows the solution space for the first example. The lines that surround a solution space are determined by the components of the edges e_1 and e_2. In the first example, the line $y = 0$, the X axis, is a surrounding line. In fact, in any of the cases specified in Case 2 in Section 4.2, one of the axes X or Y is a surrounding line of the solution space. The figure on the right demonstrates the solution space of the the second example. As we can see, the solution space of this example lies completely inside a quadrant in the X-Y plane where the surrounding lines of the space do not coincide with neither of the axes. In fact, this is true if the corresponding components of edges of a distance cone have opposite signs, i.e., $(-1,-1) \in D''$.

5.2.3 Computing Ideal Pairs in Case 1

In the following, we give two lemmas by which we solve the problem of finding x and y, if edges $e_1 = (d_1, d_2)$ and $e_2 = (d_1', d_2')$ of the distance cone are identified and we have $d_1 d_1' < 0$ and $d_2 d_2' < 0$. In other words, we will show that such integers x and y exist when we have Case 1



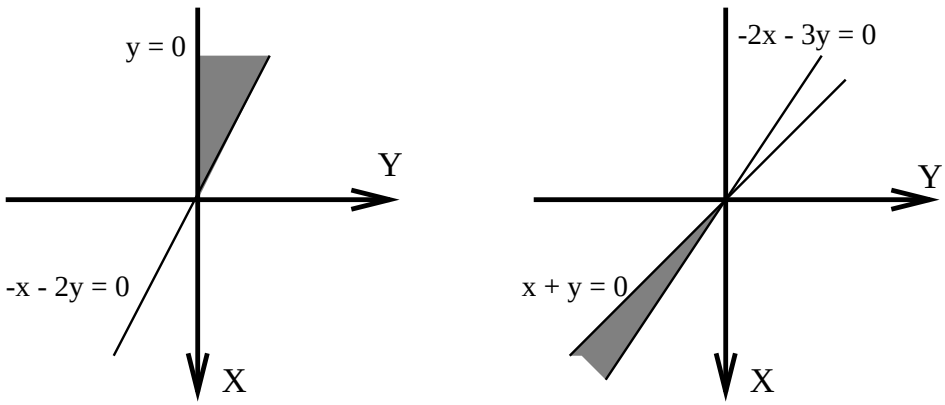
(a) The distance cone on the left figure is defined by the vectors $e_1 = (-1, -2)$, $(-1, 1)$, and $e_2 = (0, 1)$. The transformed cone, shown on the right, is caused by the unimodular matrix $\begin{bmatrix} -3 & 1 \\ 1 & 0 \end{bmatrix}$, where e_1, e_2 and $(-1, 1)$ are transformed into $e_1^{'}$, $e_2^{'}$ and $(4, -1)$, respectively.



(b) The cone on the left is defined by $D = \{(-2, -3), (0, -1), (2, 2), (1, 1)\}$. On the right, the transformed cone caused by the unimodular matrix $\begin{bmatrix} 4 & -1 \\ -3 & 1 \end{bmatrix}$ is shown. The edges $e_1 = (-2, -3)$ and $e_2 = (1, 1)$ are transformed into $e_1^{'} = (1, -1)$ and $e_2^{'} = (1, 0)$, respectively. The vector $(0, -1)$ is transformed into $(3, -1)$. The vector $(2, 2)$ that coincides with e_2 is transformed into $(2, 0)$ that coincides with $e_2^{'}$.

Figure 9: Unimodular transformations of distance cones

defined in Section 4.2. We study the role of the components of e_1 and e_2 in these lemmas to derive formulas for computing ideal pairs for any instance of Case 1. In Lemma 5.1, we solve the problem when directions of e_1 and e_2 are $(-1, -1)$ and $(1, 1)$, respectively. In Lemma 5.2, we find the integers x and y when directions of e_1 and e_2 are $(-1, 1)$ and $(1, -1)$, respectively. We will follow each lemma by showing how to obtain the ideal pair. Although there are similarities between the two lemmas, their separate treatments lead to more clear results which will be used in Theorem 5.2 and Algorithm 5.2.4. As illustrated in the second example, we will see that in the cases considered in these lemmas we may obtain an ideal pair x and y such that



The shaded areas illustrate the solution spaces of the given examples.
The figure on the left (right) is related to the first (second) example.

Figure 10: Solution spaces of the examples

$|x| > 1$ and $|y| > 1$. Recall that the parallelizing algorithm in [Ban90] one of the $|u_{11}|$ or $|u_{21}|$ was required to be greater than 1 in its general case.

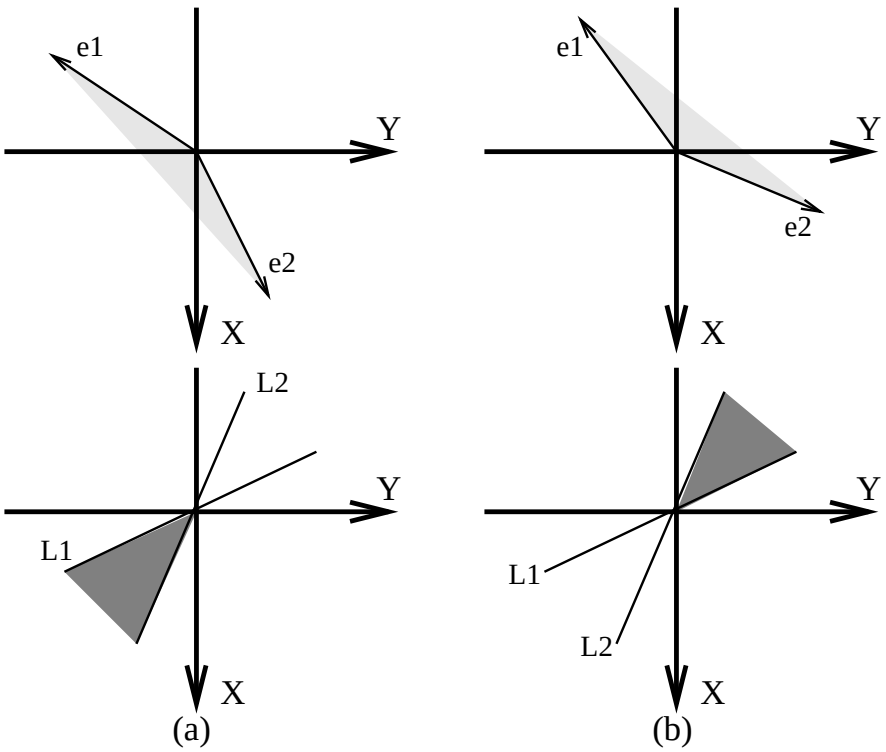
Lemma 5.1 *Let $e_1 = (-d_1, -d_2)$ and $e_2 = (d_1^{'}, d_2^{'})$ be two distance vectors, where d_1, d_2, $d_1^{'}$, and $d_2^{'}$ are positive integers. If $\frac{d_2}{d_1} \neq \frac{d_2^{'}}{d_1^{'}}$, then integers x and y exist that satisfy the relations*

$$\begin{cases} (1) & -d_1 x - d_2 y > 0 \\ (2) & d_1^{'} x + d_2^{'} y > 0. \end{cases}$$

Proof. Let us assume that relations (1) and (2) are satisfied. First, we show the relation between the components of e_1 and e_2 with the signs of x and y. From (1) we get $x < -\frac{d_2}{d_1} y$ or equivalently $y < -\frac{d_1}{d_2} x$. From (2) we have $x > -\frac{d_2^{'}}{d_1^{'}} y$ or equivalently $y > -\frac{d_1^{'}}{d_2^{'}} x$. From (2) and $y < -\frac{d_1}{d_2} x$ we get $-d_1^{'} x < d_2^{'} y < -\frac{d_1 d_2^{'}}{d_2} x$ from which we derive $(d_1 d_2^{'} - d_1^{'} d_2) x < 0$. Since $\frac{d_2}{d_1} \neq \frac{d_2^{'}}{d_1^{'}}$, we must have have either $d_1 d_2^{'} > d_1^{'} d_2$ or $d_1 d_2^{'} < d_1^{'} d_2$. If $d_1 d_2^{'} > d_1^{'} d_2$, then $x < 0$ and from $y > -\frac{d_1^{'}}{d_2^{'}} x$, we conclude that $y > 0$. If $d_1 d_2^{'} < d_1^{'} d_2$, then $x > 0$ and from $x < -\frac{d_2}{d_1} y$, we derive that $y < 0$. From (1) and (2) we get the relation

$$-\frac{d_2^{'}}{d_1^{'}} y < x < -\frac{d_2}{d_1} y. \qquad (1)$$

Next, we show how to pick integers x and y that satisfy relation 1. Let α and β be two real numbers defined as $\alpha = -\frac{d_2}{d_1} < 0$ and $\beta = -\frac{d_2^{'}}{d_1^{'}} < 0$. It can be easily shown that when $\alpha = \beta$,



At the left-top is an instance of a distance cone when $\alpha < \beta < 0$. In this case, shaded area in (a) shows the solution space. At the right-top is an instance when $\beta < \alpha < 0$. The shaded area in (b) shows the solution space, in this case. The equations of surrounding lines L1 and L2 are $-d_1 x - d_2 y = 0$ and $d_1^{'} x + d_2^{'} y = 0$.

Figure 11: Distance cones when directions $(-1, -1)$ and $(1, 1)$ exist

the vectors e_1 and e_2 form a half plane and hence are illegal (Definition 4.3). This case is excluded by assuming that $\frac{d_2}{d_1} \neq \frac{d_2^{'}}{d_1^{'}}$. Rewriting relation 1, we get $\beta y < x < \alpha y$. We can take as y, any integer such that $\alpha y - \beta y > 1$. If $\alpha > \beta$, we showed that $x < 0$ and $y > 0$. Thus, y can be any integer that is greater than or equal to $\left\lfloor \frac{1}{\alpha - \beta} + 1 \right\rfloor$. If $\alpha < \beta$, we showed that $x > 0$ and $y < 0$. In this case, we can take as y any integer that is less than or equal to $\left\lceil \frac{1}{\alpha - \beta} - 1 \right\rceil$. In either case, we can substitute a value taken for y in $\beta y < x < \alpha y$ to get x. That is, $\lfloor \beta y + 1 \rfloor \leq x \leq \lceil \alpha y - 1 \rceil$. $\square$

Clearly, there are infinite number of integer pairs (x, y) that satisfy relation 1. Such a pair is an integral point in the shaded areas (a) and (b) in Figure 11. When we have $\alpha < \beta < 0$, the shaded area in (a) shows the location of the pairs (x, y). An example of e_1 and e_2 is also illustrated in the top-left figure. When $\beta < \alpha < 0$, or equivalently $d_1 d_2^{'} > d_1^{'} d_2$, the shaded area in (b) shows the location of the pairs (x, y). In this case, the figure on the top-right illustrates an example of e_1 and e_2. As we can see, a solution space shown in the figure lies completely inside a quadrant in the X-Y plane. The lines by which a solution space is surrounded in the

figures are determined by the equations:

$$-d_1x - d_2y = 0 \quad \text{and} \quad d_1^{'}x + d_2^{'}y = 0.$$

Discussion. So far, we have shown that there always exist candidates (x, y) that satisfy relation 1. In the following, we show how to find the ideal pair (x^*, y^*), by which the objective function is minimized (constraint (2) in Section 5.2.1). First, we compare α and β. Assume that $\alpha > \beta$. If $\lfloor \beta + 1 \rfloor < \alpha$, then we take $y = 1$. Otherwise, $y = \left\lfloor \frac{1}{\alpha - \beta} + 1 \right\rfloor$. In either case, since $x < 0$, we take $x = \lceil \alpha y - 1 \rceil$. The pair (x, y) obtained this way is a candidate. Now, we apply the EE algorithm to compute $g = \gcd(x, y)$. The pair $(x_1, y_1) = (\frac{x}{g}, \frac{y}{g})$ may be a better candidate than (x, y), however, it may not be the ideal pair and hence we perform the following exhaustive search to obtain it. If $y_1 < x_1$, then for y ranging from 2 up to y_1, we get, as y^*, the integer y that satisfies relation 1. Substituting y^* for y in relation 1, we can obtain x^*, the minimum absolute x. If $y_1 > x_1$, then for x ranging from -2 down to x_1, we get x^*, the minimum absolute x that satisfies $\frac{x}{\alpha} < y < \frac{x}{\beta}$, from which we obtain y^*. Note that this relation is the *dual* of relation 1 that defines y in terms of x, α, and β. Obviously, relation 1 and its dual have the same solution space. Shortly we will show that the order in which the components of the ideal pair is obtained does not matter and, in fact, such a pair is unique.

Now, assume that we have $\alpha < \beta$. Recall that $x < 0$ and $y > 0$. If $\lfloor \alpha + 1 \rfloor < \beta$, then we take $y = -1$. Otherwise, $y = \left\lceil \frac{1}{\alpha - \beta} - 1 \right\rceil$. Having determined y, the value of x is $\lfloor \beta y + 1 \rfloor$. As in the above, we apply the EE algorithm to compute $g = \gcd(x, y)$ and use g to get a better candidate $(x_1, y_1) = (\frac{x}{g}, \frac{y}{g})$. The same exhaustive search yields the ideal pair (x^*, y^*).

Thus, given e_1 and e_2 with directions $(-1, -1)$ and $(1, 1)$, respectively, we showed how the ideal pair can be obtained. We use (x, y) to denote the ideal pair. In the above computation of the ideal pair (x, y), we may first get y and then used it to get x or vice versa. Now, we want to show that the order of obtaining the components x and y does not matter. Intuitively, this is true because the ideal pair is unique as it should be clear from the geometry of the solution space. First of all, we must have $g^* = \gcd(x, y) = 1$. Otherwise, $(\frac{x}{g^*}, \frac{y}{g^*})$ is also a candidate, where $|\frac{x}{g^*}| < |x|$ and $|\frac{y}{g^*}| < |y|$. However, this contradicts the fact that y is taken as the minimum absolute. Secondly, one can not find an ideal pair $(x^{'}, y^{'})$ such that $|x^{'}| < |x|$ and $|y^{'}| \geq |y|$. Assume that $(x^{'}, y^{'})$ is obtained by first taking the minimum absolute $x^{'}$ and then the minimum absolute $y^{'}$. In this case, we show the contradiction by proving that $(x^{'}, y)$ is the ideal pair. Suppose that $\beta < \alpha < 0$. Therefore, we have $x < 0$ and $y > 0$. By the assumption, we have that $x < x^{'} < 0$ and $0 < y \leq y^{'}$. Since (x, y) is a candidate, we have $\beta y < x < \alpha y$, from which we get $\beta y < x < x^{'}$, since $x < x^{'} < 0$. Since $(x^{'}, y^{'})$ is also a candidate, we have $\beta y^{'} < x^{'} < \alpha y^{'}$, from which we get $x^{'} < \alpha y^{'} < \alpha y$, since $\alpha < 0$ and $y \leq y^{'}$. Thus, we have $\beta y < x^{'} < \alpha y$. That is, $(x^{'}, y)$ is a candidate, where $|x^{'}| < |x|$. However, this contradicts the fact that (x, y) is ideal, since when we first took y, the integer x was taken as to be the minimum absolute. A similar proof applies if we have $\alpha < \beta < 0$. As the result, if $(x^{'}, y^{'})$ is any candidate that satisfy relation 1, we have $|x| \leq |x^{'}|$ and $|y| \leq |y^{'}|$.

Let D be the set of distance vectors in a legal MAC. Let also e_1 and e_2 be two elements in D with directions $(-1,-1)$ and $(1,1)$, respectively, such that any other vector in D either coincides with e_1 or e_2, or lies between them. We showed that for such e_1 and e_2, how to obtain the ideal pair (x,y) by which relation 1 is satisfied and the objective function $f(x,y)$ is minimized. Now, we use x and y to define a desired unimodular matrix U. To determine U, we take as its first column the pair $(x,y)^T$. Recall that $\gcd(x,y)=1$. As its second column, we can find two integers such that U be unimodular (see the EE algorithm). Having defined U as above, we will have $e_1 \cdot U >_1 \mathbf{0}$ and $e_2 \cdot U >_1 \mathbf{0}$. By Corollary 2.2.1 in [Shi92], if e_i is any other vector in D, we will also have $e_i \cdot U >_1 \mathbf{0}$. That is, the transformation of any element in D by U is a vector whose first component is positive, as we wanted. Thus, by Corollary 4.1, we can conclude that the double loop (L_1^U, L_2^U) caused by transformation of the MAC by U is equivalent to the MAC. Furthermore, since there is no dependence at level 2 in the resulting double loop, its inner loop L_2^U can be executed in parallel. Since (x,y) is an ideal pair, the matrix U is optimal. That is, the number of iterations of the outer loop L_1^U is minimized.

Let us reconsider the second example in Section 5.2.2, where $e_1=(-2,-3)$ and $e_2=(1,1)$. Since $\alpha=-1.5<\beta=-1$, then $x>0$ and $y<0$. The relation between x and y is given by $-y<x<-1.5y$. Since $\lfloor -1.5+1 \rfloor \not< -1$, then $y=\left\lceil \frac{1}{-0.5}-1 \right\rceil = -3$. Substituting for y in the relation, we get $3<x<4.5$ and hence $x=4$. The values of x and y are what we obtained earlier by direct computation.

Lemma 5.2 *Let $e_1=(-d_1,d_2)$ and $e_2=(d_1',-d_2')$ be two distance vectors, where d_1, d_2, d_1', and d_2' are positive integers. If $\frac{d_2}{d_1} \neq \frac{d_2'}{d_1'}$, then integers x and y exist that satisfy the relations*

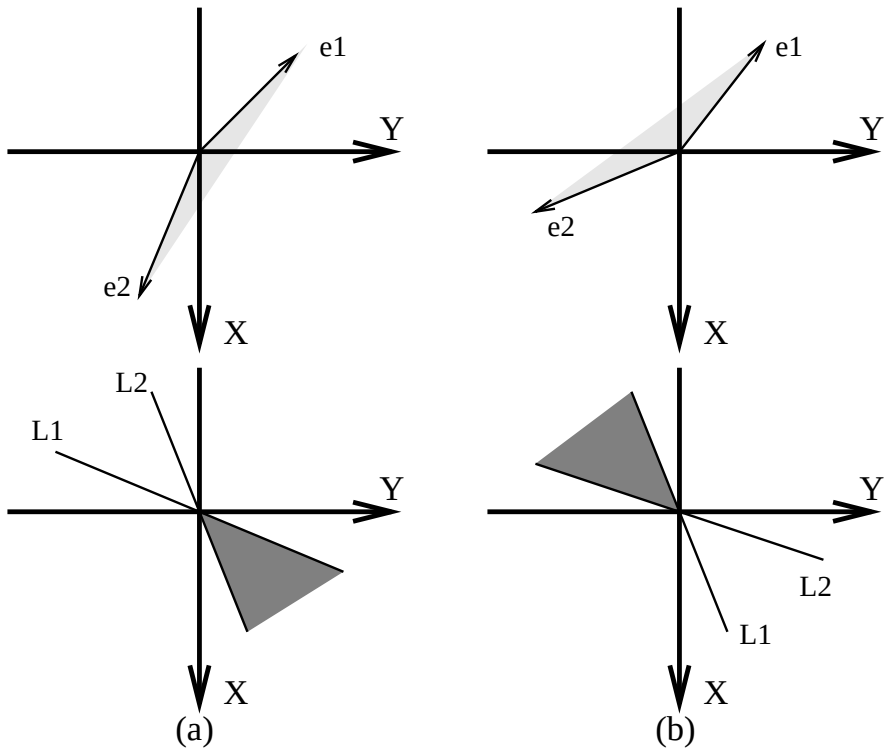
$$\begin{cases} (1) & -d_1x+d_2y>0 \\ (2) & d_1'x-d_2'y>0. \end{cases}$$

Proof. The proof is similar to the one given for Lemma 5.1. In this case, we have $\alpha=\frac{d_2}{d_1}>0$ and $\beta=\frac{d_2'}{d_1'}>0$. The illegal case of having $\alpha=\beta$ is ruled out by assuming that $\frac{d_2}{d_1}\neq\frac{d_2'}{d_1'}$. The relation between the integers x and y is given by

$$\beta y < x < \alpha y. \tag{2}$$

It is clear that if $\alpha>\beta$, then we will have $x>0$ and $y>0$. Thus, we can take any $y \geq \left\lfloor \frac{1}{\alpha-\beta}+1 \right\rfloor$. If $\alpha<\beta$, we will get $x<0$ and $y<0$. In this case, we can take as y any integer that is less than or equal to $\left\lceil \frac{1}{\alpha-\beta}-1 \right\rceil$. In either case, we substitute a value taken as y in relation 2 to obtain x. That is, $\lfloor \beta y+1 \rfloor \leq x \leq \lceil \alpha y-1 \rceil$. The pair (x,y) is a candidate. □

Figure 12 shows examples of distance cones defined by e_1 and e_2 as well as their solution spaces. The figures on the left illustrate a case where $\alpha>\beta>0$, and the ones on the right show



At the left-top is an instance of a distance cone when $\alpha > \beta > 0$. In this case, shaded area in (a) shows the solution space. At the right-top is an instance when $\beta > \alpha > 0$. The shaded area in (b) shows the solution space, in this case. The equations of surrounding lines L1 and L2 are $-d_1 x + d_2 y = 0$ and $d_1^{'} x - d_2^{'} y = 0$.

Figure 12: Distance cones when directions $(-1, 1)$ and $(1, -1)$ exist

otherwise. The shaded areas at the bottom show solution spaces. Any integral point (x, y) in a shaded area is a candidate that satisfies relation 2. As in Lemma 5.1, a solution space here in either case lies completely inside a quadrant in the X-Y plane.

Discussion. To find the ideal pair by which the objective function $f(x, y)$ is also minimized, we first compare α and β. Let us assume that $\alpha < \beta$. A similar treatment applies, if $\alpha > \beta$. We know that $x < 0$ and $y < 0$, when $\alpha < \beta$. If $\lfloor \alpha + 1 \rfloor < \beta$, then we take $y = -1$. Otherwise, take $y = \left\lceil \frac{1}{\alpha - \beta} - 1 \right\rceil$. In either case, we take $x = \lceil \alpha y - 1 \rceil$. Applying the EE algorithm, we get $g = \gcd(x, y)$ and use it to get the better candidate $(x_1, y_1) = (\frac{x}{g}, \frac{y}{g})$. The next step will give the ideal pair (x^*, y^*) through a search. If $y_1 < x_1$, then for y ranging from -2 down to y_1, we get y^* that satisfies relation 2. Then, we substitute y^* for y in this relation to get x^*. If $y_1 > x_1$, we first find x^* that satisfies $\frac{x}{\alpha} < y < \frac{x}{y}$, the dual of relation 2, and then get y^*. Thus, given e_1 and e_2 with directions $(-1, 1)$ and $(1, -1)$, respectively, the above steps finds the ideal pair (x, y). In the previous discussion, we showed that the pair (x, y) found in such a way is ideal. Let us summarize the results of both Lemmas 5.1 and 5.2 in the form of the following algorithm.

Algorithm 5.2 *Given edges* $e_1 = (d_1, d_2) < \mathbf{0}$ *and* $e_2 = (d_1^{'}, d_2^{'}) > \mathbf{0}$ *of the distance cone of a legal MAC, this algorithm returns the ideal pair.*

Remark. This algorithm is applied in step 4 of Algorithm 5.4. In step 3, here, an integer pair $(x^{'}, y^{'})$ may be obtained. If this pair is computed and if the ideal pair found in step 4 is the same as (x_1, y_1), the case is remembered and these integers will also be returned to avoid another call to the EE algorithm in step 6 of Algorithm 5.4.

1 Let α and β be two real numbers, where $\alpha = \frac{d_2}{d_1}$ and $\beta = \frac{d_2^{'}}{d_1^{'}}$.

2 If $\alpha > \beta$,

then {

if $\lfloor \beta + 1 \rfloor < \alpha$, then $y = 1$; else $y = \left\lfloor \frac{1}{\alpha - \beta} + 1 \right\rfloor$.

if $\alpha < 0$, then $x = \lceil \alpha y - 1 \rceil$; else $x = \lfloor \beta y + 1 \rfloor$.}

else {

If $\lfloor \alpha + 1 \rfloor < \beta$, then $y = -1$; else $y = \left\lceil \frac{1}{\alpha - \beta} - 1 \right\rceil$.

If $\alpha < 0$, then $x = \lfloor \beta y + 1 \rfloor$; else $x = \lceil \alpha y - 1 \rceil$. }

3 If $|x|$ or $|y|$ is 1, then return (x, y) as the ideal pair. Otherwise, apply the EE algorithm to obtain $g = \gcd(x, y)$ and a pair of integers $(x^{'}, y^{'})$ such that $xy^{'} + x^{'}y = g$. Let $(x_1, y_1) = (\frac{x}{g}, \frac{y}{g})$.

4 If $|x_1|$ or $|y_1|$ is 1, then return (x_1, y_1) as the ideal pair. Otherwise, search for the ideal pair using the relation $\beta y < x < \alpha y$, if $y_1 < x_1$; or using its dual, if $y_1 > x_1$.

Let D be the set of distance vectors in a legal MAC in Case 1, where we have either $\{(-1,-1),(1,1)\} \subseteq D^{'}$ or $\{(-1,1),(1,-1)\} \subseteq D^{'}$. Now, we want to identify edges $e_1 \in D$ and $e_2 \in D$ of the distance cone and use them as vectors considered in Lemmas 5.1 or 5.2. In other words, having determined e_1 and e_2, we can use the results obtained so far in this section to find an optimal U. We give the following algorithm that identifies these edges.

Algorithm 5.3 *Given the set* D *of distance vectors in a legal MAC in* Case 1*, this algorithm identifies edges* $e_1 < \mathbf{0}$ *and* $e_2 > \mathbf{0}$ *of the distance cone.*

Remark. This algorithm is applied in step 4 of Algorithm 5.4. There are two subcases in Case 1. If directions $(-1,-1)$ and $(1,1)$ exist, we apply the algorithm below to find e_1 and e_2. If there exist directions $(-1,1)$ and $(1,-1)$, we apply the same algorithm provided that in its steps, we replace any occurrence of $(-1,-1)$ by $(-1,1)$, and any occurrence of $(1,1)$ by $(1,-1)$.

1 For all elements in D with direction $(-1,-1)$, find an element (d_1, d_2) that has the maximum $\frac{d_1}{d_2}$. Take (d_1, d_2) as e_1.

2 For all elements in D with direction $(1,1)$, find an element $(d_1^{'}, d_2^{'})$ that has the maximum $\frac{d_1^{'}}{d_2^{'}}$. Take $(d_1^{'}, d_2^{'})$ as e_2.

3 If $\frac{d_1}{d_2} < \frac{d_1^{'}}{d_2^{'}}$, then take as e_2 an element in D that has the minimum $\frac{d_1^{'}}{d_2^{'}}$, where the minimum is taken over all the elements in D with direction $(1,1)$. Otherwise [8], take as e_1 an element in D that has the minimum $\frac{d_1}{d_2}$, where the minimum is taken over all the elements in D with direction $(-1,-1)$.

Let us briefly describe the steps of this algorithm, when directions $(-1,1)$ and $(1,-1)$ exist. In step 1, among all the elements in D with direction $(-1,1)$, we initially take as e_1 an element $(d_1, d_2) \in D$ that has the minimum angle with the positive direction of the Y axis. Likewise, among all the elements in D with direction $(1,-1)$, we initially take as e_2 an element $(d_1^{'}, d_2^{'}) \in D$ that has the minimum angle with the negative direction of the Y axis, in step 2. In step 3, the values of $\frac{d_1}{d_2}$ and $\frac{d_1^{'}}{d_2^{'}}$ are compared. Obviously, both of these values are negative in this case. If the condition holds, it means that we have $0 < \frac{-d_2}{d_1} = \alpha < \beta = \frac{-d_2^{'}}{d_1^{'}}$ (the right-top drawing in Figure 12). Hence, in this case, we take as e_2 an element in D with direction $(1,-1)$ that has the maximum angle with the negative direction of the Y axis. If $\frac{d_1}{d_2} > \frac{d_1^{'}}{d_2^{'}}$, as illustrated in the left-top drawing in Figure 12, e_1 is changed to be an element with direction $(-1,1)$ that has the maximum angle with positive direction of the Y axis.

Assume that directions $(-1,-1)$ and $(1,1)$ exist. If the test in step 3 fails, i.e., we have $0 > \frac{-d_2}{d_1} = \alpha > \beta = \frac{-d_2^{'}}{d_1^{'}}$, the case is illustrated in Figure 12 (right-top); otherwise Figure 12 (left-top) shows the case. The complexity of the algorithm is $\mathbf{O(d)}$, where $\mathbf{d}$ is the number of elements in D.

Next, we present a theorem which proves that for a legal MAC, there always exists a unimodular matrix U such that the transformation of the MAC by U can be expressed as a double loop, (L_1^U, L_2^U) in which the inner loop L_2^U can run in parallel. Note that we will prove the *existence* of U in the theorem (constraint (1) in Section 5.2.1). In the next section, we present an algorithm that determines an *optimal* U (constraint (2) in Section 5.2.1). In the proof, we will use the set $D^{''}$ introduced in Section 4.2. This set provides general yet enough information that helps classify the large number of direction sets which may exist in legal MACs. In fact, existence of a particular element in $D^{''}$ enables us to consider a group of direction sets for which we can determine U in a somehow similar way. This particular element gives information about directions of both edges e_1 and e_2 in D of the distance cone in a MAC. This is what helps to

[8] $\frac{d_1}{d_2} > \frac{d_1^{'}}{d_2^{'}}$.

classify all possible cases into four. Recall that the cone contains distance vectors in a MAC, where e_1 and e_2 are its edges and any other element in D either lies in the cone or coincides with e_1 and/or e_2.

Theorem 5.2 *There always exists a 2×2 unimodular integer matrix U such that the transformation (L_1^U, L_2^U) of a MAC is equivalent to the MAC and the inner loop L_2^U can be executed in parallel.*

Proof. Let D denote the set of distance vectors in the MAC.
The *if* part. Assume that all distance vectors $(d_1, d_2) \in D$ are of the form $\alpha(a_1, a_2)$, where α is a positive integer and (a_1, a_2) is an integer vector. We will show how to define U. Let $g = \gcd(a_1, a_2)$. Clearly, we have $g > 0$, since both of a_1 and a_2 are not zero. A desired unimodular matrix U is such that we have $(a_1, a_2) \cdot U >_2 \mathbf{0}$. That is, the following relations are to be satisfied.

$$\begin{cases} (1) & a_1 u_{11} + a_2 u_{21} = 0 \\ (2) & a_1 u_{12} + a_2 u_{22} > 0 \\ (3) & det(U) = u_{11}u_{22} - u_{12}u_{21} = 1. \end{cases}$$

The general solution to (1) is $(u_{11}, u_{21}) = t(\frac{a_2}{g}, -\frac{a_1}{g})$, where t is any integer. Substituting for u_{11} and u_{21} in (3), we get $t(\frac{a_2}{g}u_{22} + \frac{a_1}{g}u_{12}) = 1$. From this equation and (2) we get $t = 1$ and $\frac{a_2}{g}u_{22} + \frac{a_1}{g}u_{12} = 1$. Thus, we take $(u_{11}, u_{21})^T = (\frac{a_2}{g}, \frac{-a_1}{g})^T$. For the second column in U, we can obtain a pair of integers u_{12} and u_{22} such that $a_1 u_{12} + a_2 u_{22} = g$, to satisfy (3). We can apply the EE algorithm to find such a pair. The transformation of (d_1, d_2) by U is

$$(d_1, d_2) \cdot U = \alpha(a_1, a_2) \cdot U = \alpha(a_1 u_{11} + a_2 u_{21}, a_1 u_{12} + a_2 u_{22}) = (0, \alpha g) >_2 \mathbf{0}.$$

Since any distance vector in D is transformed into a positive vector by U, by Corollary 4.1, the transformation (L_1^U, L_2^U) of the MAC is equivalent to the MAC. Since there is no dependence at level 1 in (L_1^U, L_2^U), its outer loop L_1^U can run in parallel.

The *only if* part. Assume that a unimodular matrix U exists such that in the double loop (L_1^U, L_2^U), the outer loop L_1^U can run in parallel. We will show that any distance vector in D is of the form $\alpha(a_1, a_2)$, where α is a positive integer and (a_1, a_2) is an integer vector. Since there is no dependence at level 1 in (L_1^U, L_2^U), for all $(d_1, d_2) \in D$, we have $(d_1, d_2) \cdot U >_2 \mathbf{0}$. That is, the following relations are satisfied:

$$\begin{cases} (1) & d_1 u_{11} + d_2 u_{21} = 0 \\ (2) & d_1 u_{12} + d_2 u_{22} > 0. \end{cases}$$

From (1) we get $d_1 = -t u_{21}$ and $d_2 = t u_{11}$, where t is any integer. If we substitute for d_1 and d_2 in (2), we get $t\Delta > 0$, where $\Delta = det(U)$. Since $\Delta^2 = 1$, we have $(d_1, d_2) = t(-u_{21}, u_{11}) = t\Delta^2(-u_{21}, u_{11}) = (t\Delta) \cdot \Delta(-u_{21}, u_{11}) = \alpha(a_1, a_2)$, where $\alpha > 0$. That is, any distance vector in D is of the form $\alpha(a_1, a_2)$. □

Now, we present an efficient algorithm such that, given a legal MAC, it determines an optimal unimodular matrix U. That is, the transformation of the MAC by U can be expressed as a double loop in which the inner loop can run in parallel and the number of iterations of the outer loop is minimized.

5.2.4 Inner Loop Parallelization Algorithm

In Theorem 5.2, we proved that there always exists a 2×2 unimodular matrix U for a legal MAC such that (L_1^U, L_2^U), the transformation of the MAC by U, is equivalent to the MAC and its inner loop L_2^U can be executed in parallel. In this section, we first present an efficient algorithm to find a desired matrix U. Then, a description of the steps of the algorithm is given. We conclude the section with three examples. We will use (#n) to label a line in the algorithm as "n" which will be referenced in the description of the algorithm and also in the examples. Note that in the algorithm, we do not compute and use D'' as we did in the proof of the theorem.

Given a MAC, our objective is to find a unimodular matrix U such that the transformation of the MAC by U can be expressed as a double loop (L_1^U, L_2^U) in which the inner loop can run in parallel. That is, for all distance vectors (d_1, d_2) in the MAC, we want to have $(d_1, d_2)\cdot U >_1 \mathbf{0}$. Moreover, we want the number of iterations of L_1^U be minimized. We called the integers u_{11} and u_{21} in the first column of such a matrix U as an ideal pair. To find U, a list List of candidates for $(u_{11}, u_{21})^T$ is proposed that initially contains eight elements (step 2). The order of the elements in List does not matter.

An element $(l_1, l_2)^T$ of List is non-acceptable if there is a direction vector (s_1, s_2) in the MAC such that $s_1l_1 + s_2l_2 \leq 0$. In step 3, a non-acceptable candidate is deleted from List only if it has not been already deleted. In some occasions, in step 4, an element $(x, y)^T$ is computed and added to List. Such an element will be picked as the ideal pair in step 5. If we have Case 1, where directions of edges e_1 and e_2 of the cone are $(-1,-1)$ and $(1,1)$, or $(-1,1)$ and $(1,-1)$, distance vectors with any other directions are ignored. Thus, Algorithm 5.3 is applied to determine edges e_1 and e_2 of the distance cone. In any case, the ideal pair for $(u_{11}, u_{21})^T$ is picked from a final list in step 5. Finally, in step 6 the second column $(u_{12}, u_{22})^T$ is determined to make U unimodular.

Algorithm 5.4 *Given a legal MAC, this algorithm finds a unimodular matrix U such that:*
(1) The transformation (L_1^U, L_2^U) is equivalent to the MAC.
(2) The inner loop L_2^U can run in parallel.
(3) The iteration count of the outer loop L_1^U is minimized.

1 Compute D and D', the sets of distance and direction vectors in the MAC. Let (d_1, d_2) and (d_1', d_2') denote, respectively, elements in D and D'.

2 Define List as a list that is initialized with the elements:
$(0,1)^T$, $(1,1)^T$, $(1,0)^T$, $(1,-1)^T$, $(0,-1)^T$, $(-1,-1)^T$, $(-1,0)^T$, $(-1,1)^T$.

3 /* Deleting non-acceptable candidates from List */
For all $(s_1,s_2) \in D^{'}$, if there exists an element $(l_1,l_2)^T$ in List such that $s_1l_1 + s_2l_2 \leq 0$, then delete $(l_1,l_2)^T$ from List. If List is not empty, goto 5.

4 /* Adding the (ideal) pair $(x,y)^T$ to List */
if there are directions $(-1,-1)$ and $(1,1)$, or directions $(-1,1)$ and $(1,-1)$,
then /* Case 1 */ (#10)

apply Algorithm 5.3 to find $e_1 = (d_1,d_2) < \mathbf{0}$ and $e_2 = (d_1^{'},d_2^{'}) > \mathbf{0}$,

edges of the distance cone. Then, apply Algorithm 5.2 to obtain the

ideal pair $(x,y)^T$; **else**
if directions $(1,1)$ and $(-1,0)$ exist, /* Case 2 */ (#21)
then $(x,y)^T = (-1, \lfloor max\{\frac{d_1}{d_2}\} + 1 \rfloor)^T$, where the maximum is taken over

all distances (d_1,d_2) with direction $(1,1)$; **else**
if directions $(1,1)$ and $(0,-1)$ exist, /* Case 2 */ (#22)
then $(x,y)^T = (\lfloor max\{\frac{d_2}{d_1}\} + 1 \rfloor, -1)^T$, where the maximum is taken over

all distances (d_1,d_2) with direction $(1,1)$; **else**
if directions $(1,-1)$ and $(0,1)$ exist, /* Case 2 */ (#23)
then $(x,y)^T = (\lfloor max\{\frac{-d_2}{d_1}\} + 1 \rfloor, 1)^T$, where the maximum is taken over

all distances (d_1,d_2) with direction $(1,-1)$; **else**
if directions $(1,-1)$ and $(-1,0)$ exist, /* Case 2 */ (#24)
then $(x,y)^T = (-1, \lceil min\{\frac{d_1}{d_2}\} - 1 \rceil)^T$, where the minimum is taken over

all distances (d_1,d_2) with direction $(1,-1)$; **else**
if directions $(-1,-1)$ and $(1,0)$ exist, /* Case 2 * / (#25)
then $(x,y)^T = (1, \lceil min\{\frac{-d_1}{d_2}\} - 1 \rceil)^T$, where the minimum is taken over

all distances (d_1,d_2) with direction $(-1,-1)$; **else**
if directions $(-1,-1)$ and $(0,1)$ exist, /* Case 2 */ (#26)
then $(x,y)^T = (\lceil min\{\frac{-d_2}{d_1}\} - 1 \rceil, 1)^T$, where the minimum is taken over

all distances (d_1,d_2) with direction $(-1,-1)$; **else**
if directions $(-1,1)$ and $(1,0)$ exist, /* Case 2 */ (#27)
then $(x,y)^T = (1, \lfloor max\{\frac{-d_1}{d_2}\} + 1 \rfloor)^T$, where the maximum is taken over

all distances (d_1,d_2) with direction $(-1,1)$; **else**
if directions $(-1,1)$ and $(0,-1)$ exist, /* Case 2 */ (#28)
then $(x,y)^T = (\lceil min\{\frac{d_2}{d_1}\} - 1 \rceil, -1)^T$, where the minimum is taken over

all distances (d_1,d_2) with direction $(-1,1)$.

add to List the pair $(x,y)^T$ determined in any of the cases above.

5 /* Determining the ideal pair for $(u_{11}, u_{21})^T$ */
if $(N_1 - n_1) \leq (N_2 - n_2)$,
then take the first pair that exists in List from the sequence $(1,0)^T$, $(-1,0)^T$, $(0,1)^T$, $(0,-1)^T$, $(1,1)^T$, $(-1,1)^T$, $(1,-1)^T$, $(-1,-1)^T$, $(x,y)^T$;

else take the first pair that exists in List from the sequence $(0,1)^T$, $(0,-1)^T$, $(1,0)^T$, $(-1,0)^T$, $(1,1)^T$, $(-1,1)^T$, $(1,-1)^T$, $(-1,-1)^T$, $(x,y)^T$.

6 /* Finding $(u_{12}, u_{22})^T$ such that U be unimodular */
if $|u_{11}| = 1$, **then** $(u_{12}, u_{22})^T = (0,1)^T$; **else**
if $|u_{21}| = 1$, **then** $(u_{12}, u_{22})^T = (1,0)^T$; **else**
apply [9] the EE algorithm to compute $g = \gcd(x,y)$ and a pair of integers $x^{'}$ and $y^{'}$, such that $xy^{'} + yx^{'} = g$. Let $u_{11} = \frac{x}{g}$, $u_{21} = \frac{y}{g}$, $u_{12} = -x^{'}$ and $u_{22} = y^{'}$, and $U = \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix}$.
Apply algorithm 4.1 in [Ban90] to compute the transformation (L_1^U, L_2^U) of the MAC by U and mark the inner loop L_2^U as DOALL.

Given a legal MAC, the above algorithm finds an optimal unimodular matrix U, where the integers u_{11} and u_{21} in the first column of U are an ideal pair. In the following, we explain the steps of the algorithm in details. In the first step, the sets D and $D^{'}$ are computed that contain distance and direction vectors in the MAC, respectively. Let (d_1, d_2) and (s_1, s_2) denote, respectively, an element in D and $D^{'}$. In the second step, List is defined as a list and initialized with eight good candidates for the pair $(u_{11}, u_{21})^T$. The order of elements in List is not important. The proposed elements in List are good in the sense that they are the first eight vectors whose components are -1, 0, or 1.

In step 3, direction vectors are used to determine non-acceptable candidates in List. An element $(l_1, l_2)^T$ in List is a non-acceptable candidate if for any direction (s_1, s_2) in $D^{'}$, we have that $s_1 l_1 + s_2 l_2 \leq 0$. All non-acceptable elements are deleted from List. For instance, if there is a direction vector $(0,1)$, the integer l_2 can not be non-positive. Therefore, the candidates $(1,0)^T$, $(1,-1)^T$, $(0,-1)^T$, $(-1,-1)^T$ and $(-1,0)^T$ are deleted from List. By inspection, existence of any direction vector causes the deletion of five initial candidates from List.

In step 4, a new candidate may be added to List, only if a direction (s_1, s_2) exists, where $s_1 \neq 0$ and $s_2 \neq 0$. If the algorithm decides to add a new element to List, a maximum or minimum is taken over all distances with direction (s_1, s_2) to obtain the ideal pair. This operation implicitly determines an edge of the distance cone. In step 4, Case 1 is considered in line #10, and Case 2 is considered in lines #21, #22, #23, #24, #25, #26, #27, and #28. These line numbers are where a new integer pair $(x,y)^T$ may be determined and added to List. The algorithm will not add a new candidate unless it will be picked as the ideal pair in step 5. This occurs only if we have Case 1 or Case 2 in which all the initial elements are deleted

[9] See the remark in Algorithm 5.2.

from List. In Case 3 and Case 4, the ideal pair is already among the initial elements in List and hence no new candidate will be obtained.

Assume that we have Case 1 and directions $(1,1)$ and $(-1,-1)$ exist. It is straightforward to verify that all the initial candidates in List are non-acceptable and hence are deleted. In this case, since directions of edges e_1 and e_2 of the cone are, respectively, $(-1,-1)$ and $(1,1)$, distance vectors with any other directions are ignored. Then, the control reaches to the line #10, where Algorithm 5.3 is applied to find edges e_1 and e_2 as follows. First, the algorithm takes $e_1 = (d_1, d_2) \in D$ with direction $(-1,-1)$ that has the maximum $\frac{d_1}{d_2}$ and also it takes $e_2 = (d_1^{'}, d_2^{'}) \in D$ with direction $(1,1)$ that has the maximum $\frac{d_1^{'}}{d_2^{'}}$. Then, the values of $\frac{d_1}{d_2}$ and $\frac{d_1^{'}}{d_2^{'}}$ are compared. Depending on the test, the algorithm decides if e_1 or e_2 should be taken to have the minimum values of $\frac{d_1}{d_2}$ or $\frac{d_1^{'}}{d_2^{'}}$, respectively (see Figures 11 and 12). Having determined (the components of) e_1 and e_2, Algorithm 5.2 is then called to compute the ideal pair for $(u_{11}, u_{21})^T$. This computation is based on Lemma 5.1 and the discussion after the lemma. Similarly, if directions $(-1,1)$ and $(1,-1)$ coexist, the case is treated as above except that direction of e_1 is $(-1,1)$ and of e_2 is $(1,-1)$. The ideal pair in this case is computed based on Lemma 5.2 and its related discussion.

Let us consider a case where direction vector $(1,1)$ exists in $D^{'}$, but not $(-1,-1)$. The non-acceptable candidates $(1,-1)^T$, $(0,-1)^T$, $(-1,-1)^T$, $(-1,0)^T$ and $(-1,1)^T$ are deleted from List in step 3. In this case, new candidate may be obtained for List. That is, $(1,1)$ may be direction of an edge of the distance cone. This is true if (a) direction $(-1,0)$ exists (#21), or (b) direction $(0,-1)$ exists (#22). If condition (a) or (b) holds, a new candidate is obtained. Otherwise, no new element for List is computed here. Note that both of (a) and (b) can not be true when direction $(1,1)$ exists, since the MAC is legal. If (a) holds, the algorithm tries to find a distance vector with direction $(1,1)$ that forms the minimum angle with the positive direction of the X axis. That is, it finds a distance vector (d_1, d_2) in D with direction $(1,1)$ that has the maximum $\frac{d_1}{d_2}$. If (b) holds, a distance vector with direction $(1,1)$ is taken that has the maximum $\frac{d_2}{d_1}$. In any case of (a) or (b), the fraction found is used to obtain the ideal pair $(x,y)^T$ to be added to List.

There are cases in which the index bounds of a given MAC must also be considered in order to determine an optimal pair as $(u_{11}, u_{21})^T$. For instance, if $D^{'}$ contains only direction $(1,1)$, the ideal pair can be either $(1,0)^T$ or $(0,1)^T$. Consider another case where $D^{'} = \{(-1,-1)\}$ in which we can take $(-1,0)^T$ or $(0,-1)^T$ as the ideal pair. Thus, in step 5 of the algorithm, two sequences of pairs are proposed from which the ideal pair $(u_{11}, u_{21})^T$ is chosen, depending on the number of instances of the indices i and j. If we have $N_1 - n_1 \leq N_2 - n_2$, the ideal pair is the first element in the given sequence which exists in List. If $N_1 - n_1 > N_2 - n_2$, the ideal pair is the first element in the other given sequence which also exists in List. Recall that n_1, N_1, n_2, and N_2 are the index bounds.

Finally in step 6, $(u_{12}, u_{22})^T$ is determined to make U unimodular. To find such a pair, we

may require to apply the EE algorithm. Once U is determined, we can use algorithm 4.1 [Ban90] to generate the transformation (L_1^U, L_2^U) and mark its inner loop as DOALL. Next, we illustrate how, in each of the following examples, the algorithm determines an optimal unimodular matrix U.

5.2.5 Examples of Parallelized MACs

In this section, we present three examples to illustrate how Algorithm 5.4 is applied to achieve the objectives.

Example 5.2 Consider the following MAC:

A = **For** $(i,\ j)$ **in** $[1:4,\ 1:3]$

$A[i+1,\ j] + A[i,\ j+1] + A[i-1,\ j+1]$

EndFor

This example was given in Section 3, where we also gave a unimodular matrix U to demonstrate how the transformation of the MAC under U can be expressed as a double loop (L_1^U, L_2^U). Now, we show how U is determined by following the steps of the algorithm. The resulting double loop can be found in Section 3. The distance vectors in the array constructor are $(-1,0)$, $(0,-1)$ and $(1,-1)$. Clearly, in this case, direction vectors are the same as the distance vectors. The list List is initialized with the proposed candidates in step 2. Then, the non-acceptable candidates are deleted from List in step 3 by verifying the existing direction vectors. Since direction $(0,-1)$ exists, the elements $(0,1)^T$, $(1,1)^T$, $(1,0)^T$, $(-1,0)^T$ and $(-1,1)^T$ are deleted from List. This leaves List with the three elements $(1,-1)^T$, $(0,-1)^T$ and $(-1,-1)$. The existence of direction $(-1,0)$ causes the deletion of two more elements from List. Thus, the list contains the pair $(-1,-1)$. Since direction $(1,-1)$ exists, the only element in List is also deleted. The algorithm goes to step 4, since the List is empty. That is, it has to compute the ideal pair. So, the control reaches the line #24, where the test succeeds. Since both directions $(1,-1)$ and $(-1,0)$ exist. Therefore, the pair $(x,y)^T = (-1,-2)^T$ is computed and added to List. As the result of step 4, List contains only the pair $(-1,-2)^T$.

In step 5, the element $(-1,-2)^T$ in List is picked as the ideal pair from the second sequence, since $4-1 > 3-1$. In step 6, the pair $(0,1)^T$ is taken to make U unimodular. Therefore, we get $U = \begin{bmatrix} -1 & 0 \\ -2 & 1 \end{bmatrix}$. Then, algorithm 4.1 in [Ban90] is applied to produce the transformation (L_1^U, L_2^U) of the MAC by U. The produced double loop can be found in Section 3. Consider the transformations of distance vectors in the MAC by U. Since we have $(-1,0)\cdot U = (1,0) >_1 \mathbf{0}$, $(0,-1)\cdot U = (2,-1) >_1 \mathbf{0}$ and $(1,-1)\cdot U = (1,-1) >_1 \mathbf{0}$, by Corollary

4.1, the double loop (L_1^U, L_2^U) is equivalent to the MAC. The inner loop L_2^U can be executed in parallel, since the first component of any transformed distance vector is positive.

Example 5.3 As another example consider the following MAC

A = **For** $(i,\ j)$ **in** $[5:100,\ 5:100]$

$A[i+2,\ j-3] + A[i-1,\ j+2]$

EndFor

Distance vectors in this MAC are $(-2,3)$ and $(1,-2)$, and direction vectors are $(-1,1)$ and $(1,-1)$, respectively. In step 3, all the elements in List are deleted, since directions $(1,-1)$ and $(-1,1)$ exist. That is, we have Case 1 (#10). Then, Algorithm 5.3 is applied, in step 4, that identifies edges $e_1 = (d_1, d_2) = (-2,3)$ and $e_2 = (d_1^{'}, d_2^{'}) = (1,-2)$ of the distance cone. Algorithm 5.2 is then applied to compute the ideal pair. At this point, $\alpha = \frac{3}{2} = 1.5$ and $\beta = \frac{2}{1} = 2$ are defined and we have $\alpha < \beta$. Since $\lfloor 1.5+1 \rfloor \not< 2$ we get $y = \left\lceil \frac{1}{1.5-2} - 1 \right\rceil = -3$. Since $\alpha > 0$, we get $x = \lfloor (-2 \times -3) + 1 \rfloor = -5$. In step 3 of Algorithm 5.2, a call to the EE algorithm gives $\gcd(-5,-3) = 1$ and $(x^{'}, y^{'}) = (-2,1)$. In step 4 of Algorithm 5.2, two more iterations are performed to ensure that these values are in fact ideal $(\min(|-5|,|-3|)-1)$. Thus, the pair $(x,y)^T = (-5,-3)^T$ is returned to Algorithm 5.4 and added to List as a candidate. The pair $(-2,1)$ is also returned to the main algorithm (5.4). In step 5 of the main algorithm, the pair $(-5,-3)$ is taken as the ideal pair for $(u_{11}, u_{21})^T$. Note that the ideal pair is taken from the first sequence, since the number of instances of the i index is equal to the number of instances of the j index. In step 6, the pair $(2,1)^T$ is taken, as the second column in U. This pair is obtained from the pair $(-2,1)$ that is returned from Algorithm 5.2. That is, it was not required to apply, again, the EE algorithm. Therefore, the algorithm gives $U = \begin{bmatrix} -5 & 2 \\ -3 & 1 \end{bmatrix}$ by which, distance vectors $(-2,3)$ and $(1,-2)$ are transformed into $(1,-1) >_1 \mathbf{0}$ and $(1,0) >_1 \mathbf{0}$, respectively. Then, algorithm 4.1 in [Ban90] is applied to produce the following transformation of the MAC by U.

```
DO K1 = -800, -40, 1
   DOALL K2 = Ceil (Max ((5-K1)/3.0, (-100-2*K1)/5.0)),
              Floor(Min ((100-K1)/3.0, (-5-2*K1)/5.0))

     A(K1+3*K2, -2*K1-5*K2) = A(K1+3*K2+2, -2*K1-5*K2-3) +
                              A(K1+3*K2-1, -2*K1-5*K2+2)

   End
End
```

Since distance vectors in the MAC are transformed into positive vectors by U, the resulting double loop (L_1^U, L_2^U) is equivalent to the given MAC (Corollary 4.1). That is, the lexicographic order of the index vector (K_1, K_2) in (L_1^U, L_2^U) is a safe evaluation order of the array elements. Since there is no dependence at level 2, the inner loop L_2^U can run in parallel. On the idealized machine defined in Section 1, the resulting nest will execute in 761 steps which corresponds to the number of iterations of L_1^U, the serial outer loop. Note that the number of array elements defined by the MAC is 9216. For an instance of K_1, the number of instances of K_2 does not exceed $(\min(\frac{95}{5}, \frac{95}{3}) + 1)$. That is, (L_1^U, L_2^U) requires at most 20 processors at each step of its execution.

Example 5.4 How an optimal unimodular matrix U is determined for a MAC with distance vectors $(-3,-2)$, $(-2,-1)$, $(1,3)$, $(0,2)$, $(2,4)$, and $(-1,2)$?

The existing direction vectors are $(-1,-1)$, $(1,1)$, $(-1,1)$ and $(0,1)$. We want to show how the algorithm obtains an optimal unimodular matrix U. The elements $(1,0)^T$, $(1,-1)^T$, $(0,-1)^T$, $(-1,-1)^T$ and $(-1,0)^T$ are deleted from List, since a direction $(0,1)$ exists. Since directions $(1,1)$ and $(-1,-1)$ exist, we have Case 1 (#10). Thus, any initial candidate in List is non-acceptable and hence deleted from List. In step 4, Algorithm 5.3 is applied in which the maximum of $\frac{d_1}{d_2}$ over all distance vectors with direction $(-1,-1)$ is found to be 2 and hence $e_1 = (-2,-1)$. Similarly, the vector $(2,4)$ is taken as e_2, since it has the maximum over all distances with direction $(1,1)$. Since $\frac{-2}{-1} > \frac{2}{4}$, e_1 is reconsidered to be $(-3,-2)$. Having returned edges e_1 and e_2 of the distance cone to the main algorithm, the control is passed to Algorithm 5.2 to compute the ideal pair $(x,y)^T$. There, $\alpha = -\frac{2}{3} = -0.66$ and $\beta = -\frac{4}{2} = -2$ are defined and we have $\alpha > \beta$. Since $\lfloor -2 + 1 \rfloor < -0.66$, we get $y = 1$. Since $\alpha < 0$, we get $x = \lceil (-0.66 \times 1) - 1 \rceil = -1$. Thus, the ideal pair $(x,y)^T = (-1,1)^T$ is returned to the main algorithm, where the pair is added to List. In step 5, this pair will be taken as $(u_{11}, u_{21})^T$. In step 6, the unimodular matrix defined by the algorithm is $U = \begin{bmatrix} -1 & 0 \\ 1 & 1 \end{bmatrix}$. The transformations of the elements in D under U are:
$(-3,-2) \cdot U = (1,-2) >_1 \mathbf{0}$,
$(-2,-1) \cdot U = (1,-1) >_1 \mathbf{0}$,
$(1,3) \cdot U = (2,3) >_1 \mathbf{0}$,
$(0,2) \cdot U = (2,2) >_1 \mathbf{0}$,
$(2,4) \cdot U = (2,4) >_1 \mathbf{0}$ and
$(-1,2) \cdot U = (3,2) >_1 \mathbf{0}$.
As expected, the first component in any transformed distance vector is positive.

6 Implementation and Complexity Remarks

In this section, we describe an implementation of the algorithms and address their complexity.

6.1 AC-Transformer: A One-Step Parallelizing Program For MACs

We have implemented Algorithms 5.1 and 5.4 in a program called *AC-Transformer*[10]. If a given MAC is illegal, the program reports this through an error message. Otherwise, AC-Transformer computes a unimodular matrix and uses it to apply a one-step transformation to produce an equivalent double loop in which the outer loop or the inner loop can be executed in parallel. The input to the program is a file containing distance vectors in the MAC. Each distance vector is a pair of integers that form a record in the file. The index bounds n_1, N_1, n_2, and N_2 are given as inputs to the program at run time. In Figure 13, the input/output diagram of the AC-Transformer is illustrated.

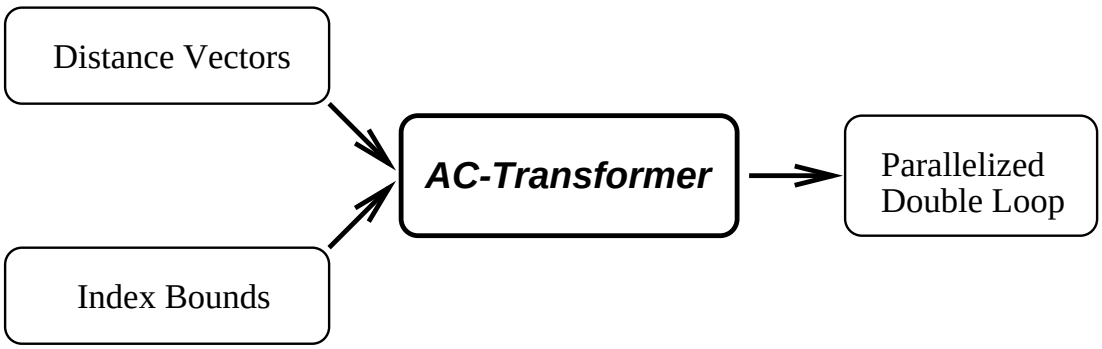
AC-Transformer is implemented in about 1450 lines of C code. About 70% of the program is the implementation of the algorithm 4.1 in [Ban90] which would generate the resulting double loop. This is because many cases are considered to produce a double loop that has simpler terms as the loop bounds and the substitute expressions for the indices in the body. For instance, the term "$1 * K1$" is simplified to "$K1$", or "$(-2 - 3 * K1)/(-1)$" is reduced to "$2 + 3 * K1$". These are useful reductions, if the program is integrated in a real compiler. See [Shi91] for more deatils about our implementation of this algorithm.

Let D be the set of dependence distance vectors in a given MAC. Depending on the elements in D, we have one of the following cases.

1. D is an empty set or contains only $(0, 0)$.
2. $\forall (d_1, d_2) \in D$, we have $d_1 \neq 0$ and
   (a) $\frac{d_2}{d_1}$ is a fixed constant.
   (b) There exist at least two elements (d_1, d_2) and (d_1', d_2') in D, such that $\frac{d_2}{d_1} \neq \frac{d_2'}{d_1'}$.
3. All the elements in D are of the form $(0, d_2)$, where they are either all positive or all are negative.
4. There exist at least two elements (d_1, d_2) and (d_1', d_2') in D, such that $d_1 = 0$ and $d_1' \neq 0$.

Recalling the definitions of dependence at level 1 and level 2 from Section 4.1, case 1 means no dependence exists in the MAC. Case 2 implies the existence of dependence at level 1 but not at level 2 and case 3 is the existence of dependence at level 2 but not at level 1. There exist dependences at both levels in case 4. In the cases 1, 2(a), and 3, where the distance vectors in D are parallel to each other and have the same direction vector, it is possible to express the array constructor as a double loop in which the outer loop or the inner loop can run in parallel. Therefore, the user is given an option to choose the loop to be parallelized. In the other cases 2(b) and 4, the MAC can only be expressed as a double loop in which the inner loop is parallelized.

[10] For Array Constructors Transformer.



The inputs to AC-Transformer are the index bounds and the distance vectors in a MAC which are used to determine a unimodular matrix U such that the transformation of the MAC caused by U can be expressed as a double loop in which the outer loop or the inner loop can safely run in parallel.

Figure 13: The input/output diagram of the AC-Transformer

6.2 Complexity Issue

The complexity of the outer loop parallelizing Algorithm 5.1 is $\mathbf{O}(\mathbf{d} + \log \mathbf{a})$, where $\mathbf{d}$ is the number of distance vectors in the given MAC and $\mathbf{a} = \min(|a_1|, |a_2|)$. Recall that (a_1, a_2) is a fixed integer vector that any distance vector in L is a multiple of it. The term $\log \mathbf{a}$ is the complexity of the extended Euclidean algorithm used to compute $g = \gcd(a_1, a_2)$ and two integers u_{12}, u_{22} such that $a_1 u_{12} + a_2 u_{22} = g$ [CLR90]. The pair $(u_{12}, u_{22})^T$ is the second column of the transformation matrix U. Algorithm 4.1 in [Ban90] is used to compute the resulting double loop; The complexity of this algorithm is $O(1)$.

The complexity of Algorithm 5.4 is $\mathbf{O}(\mathbf{d} + \mathbf{b})$, where $\mathbf{d}$ is the number of distance vectors in the given MAC and $\mathbf{b} = \min(|u_{11}|, |u_{21}|) - 1$. Integers u_{11} and u_{21} are elements of the first column of the transformation matrix U. This main algorithm uses Algorithms 5.3, 5.2, EE, and 4.1 in [Ban90]. The complexity of Algorithm 5.3 is $\mathbf{O}(\mathbf{d})$. This algorithm may be applied in step 4 of the main algorithm (line #10) to identify edges e_1 and e_2 of the distance cone. In the worst case analysis of Algorithm 5.2, which is applied in step 4 of the main algorithm if we have Case 1, $b = \min(|x|, |y|) - 1$ iterations are at most required to ensure that the computed pair (x, y) is ideal. Note that there is also a call to the EE algorithm in step 3 of Algorithm 5.2. Thus, the overall complexity of Algorithm 5.2 is $O(b)$ which is the dominate term in $O(b + \log b)$. Note also that in some cases we may avoid calling the EE algorithm in step 6 of the main algorithm (see the remark in Algorithm 5.2). If we don't have Case 1, the main algorithm runs in $O(d)$. Recall that in Case 1, we may obtain $|u_{11}| > 1$ and $|u_{21}| > 1$, and hence the EE algorithm is applied to determine integers u_{12} and u_{22} such that U be unimodular. In the inner loop parallelizing algorithm in [Ban90], the pair $(u_{12}, u_{22})^T$ can be easily obtained, since u_{21} is always 0 or 1.

In terms of the number of the iterations of the outer loop in a transformation of a MAC, the proposed Algorithms 5.1 and 5.4 perform optimal transformations. That is, the number of

iterations of the outer loop in the transformation (L_1^U, L_2^U) is maximized, if L_1^U is parallelized. If the inner loop L_2^U is parallelized, the iteration count of the serial outer loop is minimized.

7 Conclusion and Future Work

In this research we studied unimodular transformations, originally proposed to parallelize loops in imperative languages. Specifically, we reviewed a parallelizing technique for imperative double loops and presented *U-Transformer*, an implementation of the technique. We used Parafrase-2 compiler as an interface to the above program to produce distance vectors in an input double loop.

We also studied recursively defined monolithic array constructors (MACs) that are well suited for use in scientific computations. In a MAC, no explicit order of evaluation of the array elements is specified. Data dependences may only impose an order. The major problem that arises in the implementation of a MAC is the overhead of scheduling and synchronizing of the array elements. If dependences are not identified and used to determine a safe evaluation order, an implementation has to rely on inefficient or expensive schemes to ensure that an array element is defined before it is used.

For a two-dimensional MAC, we formulated the problem and proposed a compile-time solution based on unimodular transformations. This solution is developed by extending the parallelizing algorithms mentioned above. We proved the safety and the optimality of the transformations. We also introduced *AC-Transformer*, an implementation of the proposed algorithms. Give a MAC, the program determines a unimodular matrix and applies a one-step transformation to produce an equivalent double loop in which the outer loop or the inner one can be executed in parallel. That is, a safe parallel evaluation order of the array elements is determined as the result of the transformation. As in [Ban90], we assumed that distance vectors in a given MAC are regular and constant. The index bounds are also assumed to be integer constants.

In the following, we propose some future work. As an extension to this thesis, algorithms can be developed to parallelize higher dimensional MACs based on unimodular transformations. Another work would be measuring the quality and the quantity of the resulting programs by executing the programs U-Transformer and AC-Transformer on non-sequential machines, such as vector and parallel computers. As another extension, we may relax on the assumption that the body expression in a MAC contains a single statement which shows what is defined as an array element. In this thesis, we showed that safe parallel evaluation order of the elements in a MAC can be determined while the programmer is not required to explicitly define an order. Basically, we applied a technique developed for parallelizing imperative programs. Therefore, compilers in functional languages can be enhanced to incorporate appropriate techniques of subscript analysis and program transformations developed for imperative languages.

8 Acknowledgment

The authors are grateful to the National Sciences and Engineering Research Council of Canada and the Founds Pour Formation De Chercheurs Et L'Aides 'A La Recherch of Quebec for their support of this research.

Shiri is also grateful to the Ministry of Culture and Higher Education of The Islamic Republic of Iran for the scholarship he was awarded.

References

[ABC86] Todd Allen, Michael Burke, and Ron Cytron. A practical and powerful algorithm for subscript dependence testing. Technical report, IBM, 1986. Internal Report.

[ABC+89] F. Allen, M. Burke, P. Charles, R. Cytron, J. Ferrante, V. Sarkar, and D Shields. PTRAN methodologies for the automatic detection of parallelism in FORTRAN programs: Some experimental results (extended abstract). In *Proceedings of the Fourth International Conference on Supercomputing*, May 1989.

[ABKP86] R. Allen, D. Baumgartner, K. Kennedy, and A. Porterfield. PTOOL: A semi-automatic parallel programming assistant. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 164–170, August 1986.

[AC72] F. E. Allen and J. Cocke. A cataloge of optimizing transformations in design and optimization of compilers. *R. Rustin, Ed. Prentice-Hall, Englewood Cliffs, New Jersey*, pages 1–30, 1972.

[ACK87] Randy Allen, David Callahan, and Ken Kennedy. Automatic decomposition of scientific programs for parallel execution. *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 63–76, January 1987.

[AD79] W. B. Ackerman and J. B. Dennis. VAL—a value-oriented algorithmic language. Technical Report 218, Laboratory for Computer Science, MIT, 1979.

[AH90] S. Anderson and P. Hudak. Compilation of Haskell array comprehensions for scientific computing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (SIGPLAN Notices, vol. 25, no. 6)*, pages 137–149, June 1990.

[AK83] J. R. Allen and K. Kennedy. PFC: A program to convert FORTRAN to parallel form. Technical Report MASC-TR82-6, Rice University, Houston, TX, April 1983.

[AK84] John R. Allen and Ken Kennedy. Automatic loop interchange. *SIGPLAN Notices*, 19(6):233–246, June 1984. Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.

[AK87] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9:491–542, 1987.

[ANP89] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM TOPLAS*, 11(4):598–632, October 1989.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers—Principles, Techniques, and Tools*. Addison-Wesley Publishing Co., 1986.

[Ban76] Utpal Banerjee. Data dependence in ordinary programs. Master's thesis, University of Illinois at Urbana-Champaign, 1976.

[Ban79] U. Banerjee. *Speedup of Ordinary Programs*. PhD thesis, University of Illinois at Urbana-Champaign, October 1979. Dept. of Computer Science Rpt. 79-989.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, MA, 1988.

[Ban90] U. Banerjee. Unimodular transformations of double loops. In *Proceedings of the Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990. Also published in *Monographs in Parallel and Distributed Computing*, Pages 192–219, Pitman, 1991.

[BC86] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, 21(7):162–175, July 1986.

[Can89] David C. Cann. *Compilation Techniques for High Performance Applicative Computation*. PhD thesis, Colorado State University, Boulder, CO, May 1989. Technical Report CS-89-108.

[CFS90] Ron Cytron, Jeanne Ferrante, and Vivek Sarkar. Compact representations for control dependence. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, White Plains, NY*, pages 337–351, June 1990.

[CLR90] Thomas H. Cormen, Charles E. Leisorson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[Cor79] Burroughs Corp. Numerical aerodynamic simulation facility feasibility study. *Burroughs Copr., Paoli, Pa., March*, 1979.

[Cra76] Cray Research, Inc. *Cray-1 Computer System Reference Manual.* Publication 2240004, Cray Research, Inc., Bloomington, Minn., 1976.

[Dow90] Michael L. Dowling. Optimal code parallelization using unimodular transformations. *Parallel Computing*, 16:157–171, June 1990. North-Holand.

[Flo78] Floating Point Systems, Inc. *AP-120 Programmers Reference Manual.* Publication 860-7319-000, Floating Point Systems, Inc., Beaverton, Ore., 1978.

[Gri54] H. Griffin. *Elementary Theory of Numbers.* McGraw-Hill, 1954.

[GS91] G. R. Gao and N. Shiri. Unimodular transformations of 2_d monolithic array constructors. ACAPS Technical Memo 24, School of Computer Science, McGill University, Montreal, July 1991.

[Guz87] Mark D. Guzzi. *Cedar Fortran Prpgrammer's Manual.* Document No. 601, draft, University of Illinois at Urbana Champaign, March 1987.

[GYDM90] G. R. Gao, R. K. Yates, J. B. Dennis, and L. M. R. Mullin. An efficient monolithic array constructor. ACAPS Technical Memo 19-revised, School of Computer Science, McGill University, Montreal, Sepember 1990.

[Hal85] R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, October 1985.

[Hud84] P. Hudak. ALFL Reference Manual and Programmer's Guide. Technical Report YALEU/DCS/TR-322, Yale University, July 1984.

[Hud89] P. Hudak. Conception, evolution, and application of functional programming languages. *Computing Surveys*, 21(3):359–411, September 1989.

[Hug85] J. Hughes. Lazy memo-functions. In *Functional Programming Languages and Computer Architecture*, pages 129–146. Springer-Verlag, LNCS-201, 1985.

[HWe90] P. Hudak and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (version 1.0). Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science, April 1990.

[Ins89] American National Standards Institute. FORTRAN 8X draft. *FORTRAN Forum*, 8(4), December 1989.

[Iri88] Francois Irigoin. Loop reordering with dependence direction vectors. *Journees Firtech Systemes et Telematique Architectures Futures: Programmation Parallel et integration VLSI, Document EMP-CAI-I A/184, Paris, France*, November 9-10 1988.

[Kel80] R.M. Keller. Divide and concer. *In ACM Symposium on Lips, Palo Alto, California*, pages 196–202, August 1980.

[Kel82] R.M. Keller. Fel programmer's guide. *Technical Report AMRS TR 7, University of Utah*, March 1982.

[KKBP91] D. Kulkarni, K. G. Kumar, A. Basu, and A. Paulraji. Loop partitioning unimodular transformations for distributed memory multiprocessors. *Proceedings of the Fifth International Parallel Processing Symposium*, April 1991. To appear.

[KKP+80] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Analysis and transformation of programs for parallel computation. In *Proceedings of the Fourth International Computer Software and Application Conference*, October 1980.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, Reading Mass., second edition, 1973.

[KRP+81] D. J. Kuck, Kuhn R., D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the Eighth ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[Kuc78] David J. Kuck. *The Structure of Computers and Computations*. John Wiley and Sons, 1978.

[Lam74] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, pages 83–93, February 1974.

[LB80] S. F. Lundstrom and G. H. Barnes. A controlable mimd architecture. *In Proceedings of the 1980 International Conference on Parallel Processing (Bellaire, Mich. Aug. 26-29). IEEE Press, NewYork*, pages 19–27, 1980.

[Lea85] Bruce Leasure. The Parafrase project's FORTRAN analyzer major module documentation. Technical report, University of Illinois at Urbana-Champaign Center for Supercomputing Research and Development, July 1985. CSRD Rpt. No. 504 PR-85-5, UILU-ENG-85-8005.

[LYZ90] Z. Li, P. C. Yew, and C. Q. Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):26–34, January 1990.

[NA90] R. S. Nikhil and Arvind. Id: A language with implicit parallelism. Computation Structures Group Memo 305, Laboratory for Computer Science, MIT, 1990.

[Nic84] A. Nicolau. *Parallelism, Memory Anti-Aliasing, and Correctness for Trace Scheduling Compilers*. PhD thesis, Yale University, 1984.

[Nik88] R. S. Nikhil. Id (Version 88.0) reference manual. Computation Structures Group Memo 284, Laboratory for Computer Science, MIT, March 1988.

[Pey87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International Series in Computer Science, 1987.

[PGH+88] Constantine Polychronopoulos, Miland Girkar, Mohammad Reza Haghighat, Chia Ling Lee, Bruce Leung, and Dale Schouten. Parafrase-2: A new generation parallelizing compiler. Technical report, University of Illinois, CSRD, December 1988. CSRD Rpt. No. 837.

[PK87] Constantine D. Polychronopoulos and David J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12), December 1987.

[PKL80] D. A. Padua, D. J. Kuck, and D. L. Lawrie. High speed multiprocessor and compilation techniques. *IEEE Transactions on Computers*, C–29(9):763–776, September 1980.

[PW86] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[Rep90] John H. Reppy. *Concurrent Programming with Events—The Concurrent ML Manual*, November 1990. Updated for Version 0.9.5, July 1991.

[Rus78] R. M. Russell. The cray-1 computer system. *Commun. ACM 21, 1*, pages 63–72, January 1978.

[Sch85] D. A. Schmidt. Detecting global variables in denotational specifications. *ACM TOPLAS*, 7(2):299–310, April 1985.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, 1986.

[Shi91] Shiri V., Nematollaah U_Transformer: An implementation of unimodular transformations of double loops. ACAPS Technical Note 26, School of Computer Science, McGill University, Montreal, February 1991.

[Shi92] Shiri V., Nematollaah. A design and implementation of unimodular transformations of loops. Master's thesis, McGill University, School of Computer Science, Montreal, March 1992.

[Sto85] W. Stoye. The implementation of functional languages using custom hardware. *Phd thesis, Cambridge University. Computer Laboratory Technical Report No. 81.*, December 1985.

[Tow76] Ross Albert Towle. *Control and Data Dependence for Program Transformations.* PhD thesis, University of Illinois at Urbana-Champaign, 1976. Report UIUCDCS-R-76-788.

[Tur86] D. A. Turner. An overview of miranda. *SIGPLAN Notices*, 21(12):158–166, December 1986.

[Wad86] Philip Wadler. A new array operation. *Programming Research Group, Oxford University, UK and Programming Methodology Group, Chalmers University, Goteborg, Sweden*, 1986.

[WL90] Michael E. Wolf and Monica S. Lam. Maximizing parallelism via loop parallelism. *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, August 1990. Also published in *Monograph in Parallel and Distributed Computing*, Pages 243–259, Pitman, 1991.

[Wol82] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, October 1982. Rpt. No. UIUCDCS-R-82-1105.

[Wol86] Michael Wolfe. Advanced loop interchanging. *Proceedings of the 1986 International Conference on Parallel Processing*, pages 536–543, Aug. 19–22 1986.

[Wol89] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman, London and MIT Press, Cambridge, MA, 1989. In the series, Research Monographs in Parallel and Distributed Computing. Revised version of the author's Ph.D. dissertation, Published as Technical Report UIUCDCS-R-82-1105, University of Illinois at Urbana-Champaign, 1982.