Transformations of Nested Loops with Non-Convex Iteration Spaces

Jingling Xue

Department of Mathematics, Statistics and Computing Science
University of New England
Armidale 2351, Australia

Abstract

When compiling for parallel machines, it is often necessary to generate a loop nest to scan a region of index points in lexicographic order. One well-known application example is the use of loop transformations to restructure loop nests. Previous work shows how to generate code to scan a convex polyhedron (possibly intersected with a lattice), a region derived from an application of a non-singular transformation to a convex iteration space. The driving force behind this work is to investigate how to generate code for non-singular transformations of nested loops with non-convex iteration spaces. The final result is a method that generates code for a union of convex polyhedra, possibly intersected with a lattice. Two other applications that are discussed include generation of data transfer code for machines with a hierarchical memory and construction of SPMD programs for distributed memory parallel machines.

Key words: Parallel processing; Loop transformation; Nested loops; Non-convex iteration space; Quasi-convex polyhedron; Maximal normal form; Minimal normal form; SPMD

1 Introduction

Parallelising compilers often use loop transformations to reorder the iterations of a loop nest in order to expose the available parallelism and improve the memory hierarchy performance. A loop transformation maps the iteration space of a loop nest to a new iteration space, and is modeled by a non-singular integer matrix. The new iteration space is known as the *transformed iteration space*. In the transformed iteration space, the iterations are executed lexicographically with respect to that space. To implement the loop transformation, one must generate a loop nest to scan the iterations in the transformed iteration space in lexicographic order. This *code generation problem* has been previously studied in the two special cases:

1. **Unimodular transformations of a loop nest with a convex iteration space.** The importance of unimodular transformations, such as loop interchange, skewing and reversal, in generating good code for vector and parallel machines is well-known [1, 15, 17]. If the iteration space is a convex polyhedron, the transformed iteration space under every unimodular transformation is also a convex polyhedron. Several methods can be used to construct a perfect loop nest to scan a convex polyhedron. These include the Fourier-Motzkin elimination [3], Feautrier's parametric integer programming (PIP) [7] and the algorithm described in [15]. All these methods generate

loop nests with two interesting properties. First, each lower (upper) bound is a maximum (minimum) of several affine expressions of the surrounding loop variables and problem size parameters. In [5], an attempt was made to reduce each loop bound to a single affine expression. Second, all loops have step sizes 1. This is because a convex polyhedron is dense; consequently, every point in the polyhedron must be scanned.

2. **Non-singular transformations of a loop nest with a convex iteration space.** The motivation for this investigation is to generalise the work on unimodular loop transformations. The final results are methods that can generate code for any non-singular (unimodular or non-unimodular) transformations of nested loops with convex iteration spaces [13, 18]. Non-unimodular code generation is useful in systolic array design [14, 19], loop tiling [16] and distributed memory code generation [2], to name just a few.

We summarise below our code generation algorithm for non-singular transformations of convex iteration spaces [18]. Let $\{x \mid Ax \le b\}$ be the iteration space of a loop nest, which describes a convex polyhedron. Let T be a non-singular loop transformation matrix: $Tx = y$. The transformed iteration space under T can be precisely specified as:

$$\{y \mid AT^{-1}y \le b\} \cap \mathcal{L}(T)$$

where $\mathcal{L}(T)$ is the *lattice* defined by the columns of T, i.e.,

$$\mathcal{L}(T) \quad = \quad \{u \mid u{=}Tv \wedge v{\in}\mathbb{Z}^n\}$$

To scan such a non-dense iteration space, our algorithm produces a perfect loop nest of the following form in two steps:

$$\begin{array}{l} \textsf{do } y_1 = L_1 + \delta_1,\, U_1,\ \ Step_1 \\ \quad \cdots \\ \qquad \textsf{do } y_n = L_n + \delta_n,\, U_n,\ \ Step_n \end{array}$$

The first step calculates L_k and U_k and produces

$$\begin{array}{l} \textsf{do } y_1 = L_1,\, U_1 \\ \quad \cdots \\ \qquad \textsf{do } y_n = L_n,\, U_n \end{array}$$

to scan $\{y \mid AT^{-1}y \le b\}$. This step can be accomplished using any of existing code generation methods for unimodular transformations, such as the Fourier-Motzkin elimination. In the second step, the Hermite normal form, H, of T is used to calculate δ_k and $Step_k$ to skip the points in $\{y \mid AT^{-1}y \le b\}$ but not in $\mathcal{L}(T)$. Let $h_{i,j}$ denote the (i,j)-th element in H. The step sizes are simply the diagonal elements of H:

$$Step_k \quad = \quad h_{k,k} \tag{1}$$

The lower bound offsets δ_k are also derived from H:

$$\delta_k \quad = \quad (-\det \begin{bmatrix} h_{1,1} & \cdots & h_{1,k-1} & y_1 \\ & \cdots & & \\ h_{k-1,1} & \cdots & h_{k-1,k-1} & y_{k-1} \\ h_{k,1} & \cdots & h_{k,k-1} & L_k \end{bmatrix} / \ (h_{1,1} \times \ldots \times h_{k-1,k-1}))\ \%\ h_{k,k} \tag{2}$$

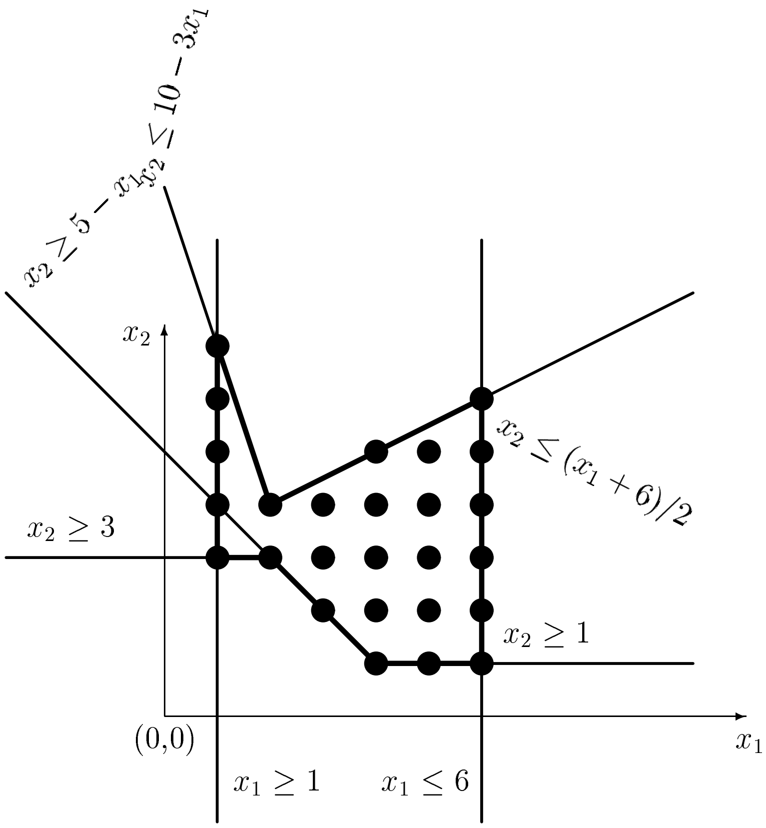


Figure 1: A loop nest with a non-convex iteration space.

where % denotes the following modulo operation: for $a, b \in \mathbb{Z}$, $a\%b = r$ iff $a = qb + r$, where $q, r \in \mathbb{Z}$ and $0 \leq r < |b|$.

In this paper, we study the problem of transforming nested loops with non-convex iteration spaces under non-singular loop transformations. An example with a non-convex iteration space is given below (Figure 1):

$$
\begin{array}{l}
\textsf{do } x_1 = 1,\ 6 \\
\quad \textsf{do } x_2 = \min(3, \max(5 - x_1, 1)),\ \max(10 - 3x_1, \lfloor (x_1+6)/2 \rfloor) \\
\quad\quad a(x_1, x_2) = a(x_1 - 1, x_2)
\end{array}
\tag{3}
$$

We describe a method for constructing DO loops to scan such non-convex iteration spaces. We also discuss how this method can be used in other code generation problems that arise in programming shared and distributed memory machines.

The rest of this paper is organised as follows. Section 2 discusses the formulation of the problem. Some basic terminology and definitions are introduced. Examples that illustrate the development of our method are also defined. Section 3 presents our method that implements non-singular transformations of nested loops with non-convex iteration spaces. Section 4 discusses applications of our method in other code generation problems in compiling for parallel machines. Section 5 concludes the paper.

2 Problem Statement

We consider a class of perfect loop nests of the form:

$$\begin{array}{l} \texttt{do}\ x_1 = L_1,\ U_1 \\ \quad \cdots \\ \qquad \texttt{do}\ x_n = L_n,\ U_n \end{array} \tag{4}$$

where

- All loop step sizes are 1.
- Loop bounds L_k and U_k are expressions built using
  - affine expressions of surrounding loop variables $x_1, \cdots, x_{k-1}$ and problem size parameters (e.g., array sizes), and
  - max, min, ceiling and floor operators.

  Sometimes, ceiling and floor operators are needed to ensure that all loop bounds evaluate to integer values.

One loop nest that fits this model is given in (3), and its iteration space is a non-convex polyhedron as depicted in Figure 1. For convenience, we refer to the iteration spaces under this algorithm model as *quasi-convex iteration spaces* or *quasi-convex polyhedra*.

In the rest of this section, we first show that the constraints that specify a quasi-convex polyhedron can be put into two kinds of normal forms: maximal normal form and minimal normal form. The maximal normal form allows us to identify the non-convex surfaces of a quasi-convex polyhedron, and is the basis for our code generation algorithm. The minimal normal form exposes the descriptive power of quasi-convex polyhedra: a quasi-convex polyhedron is nothing but a union of convex polyhedra. We then formulate the problem we address in this paper.

We recall several useful properties about expressions involving max and min.

Lemma 1 *Let a, $b_1, b_2, \cdots$ be real-valued expressions.*

$$\begin{array}{lllll} (1) & a \geq \max(b_1, b_2, \cdots) & \equiv & a \geq b_1 \wedge a \geq b_2 \wedge \cdots \\ (2) & a \leq \max(b_1, b_2, \cdots) & \equiv & a \leq b_1 \vee a \leq b_2 \vee \cdots \\ (3) & a \geq \min(b_1, b_2, \cdots) & \equiv & a \geq b_1 \vee a \geq b_2 \vee \cdots \\ (4) & a \leq \min(b_1, b_2, \cdots) & \equiv & a \leq b_1 \wedge a \leq b_2 \wedge \cdots \\ (5) & -\max(b_1, b_2, \cdots) & \equiv & \min(-b_1, -b_2, \cdots) \\ (6) & -\min(b_1, b_2, \cdots) & \equiv & \max(-b_1, -b_2, \cdots) \end{array}$$

If $a, b_1, b_2, \cdots$ are affine expressions over integer variables, both $a \geq \max(b_1, b_2, \cdots)$ and $a \leq \min(b_1, b_2, \cdots)$ each represent a conjunction of affine constraints. Unfortunately, the same is not true for $a \leq \max(b_1, b_2, \cdots)$ and $a \geq \min(b_1, b_2, \cdots)$. This is illustrated in Figure 2. Therefore, if both max and min appear in the same loop bound, the iteration space may be a non-convex region. One such a example is shown in Figure 1.

In practice, a loop bound may not contain a large number of max and min operators. But such a loop bound is nevertheless permitted to exist by definition. It is therefore useful if all loop bounds can be put into some standard formats.

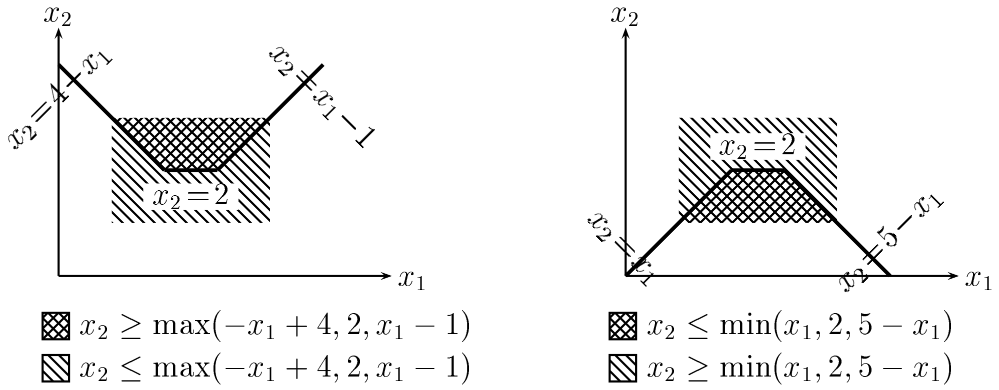


Figure 2: Constraints with max and min operators.

Lemma 2 *A loop bound can be put into the following* max-min *normal form*

$$\max(\min(a_{1,1}, a_{1,2}, \cdots),\ \min(a_{2,1}, a_{2,2}, \cdots),\ \cdots)$$

and the following min-max *normal form*

$$\min(\max(a_{1,1}, a_{1,2}, \cdots),\ \max(a_{2,1}, a_{2,2}, \cdots),\ \cdots)$$

where $a_{i,j}$ are affine expressions free of max *and* min *operators.*

Proof. To put a loop bound into a max-min normal form, we repeatedly apply the rule:

$$\min_{k=1}^{m} \max(b_{k,1}, b_{k,2}, \cdots, b_{k,n_k}) \ \equiv \ \max_{\forall\ 1\leq k\leq m\ :\ 1\leq p_k\leq n_k} \min(b_{1,p_1}, b_{2,p_2}, \cdots, b_{m,p_m}) \tag{5}$$

inside-out by replacing the left hand side with the right hand side and eliminating redundant max and min operators along the way. Similarly, to put a loop bound into a min-max normal form, we apply the same rule inside-out except that this time we will replace the right hand side with the left hand side. ■

As an example, to put $\max(a, \min(b, \max(c, d)))$ into the max-min normal form where a, b, c and d are affine expressions, we perform the following sequence of algebraic simplifications:

$$\begin{array}{ll} & \max(a, \min(b, \max(c, d))) \\ \equiv & \{\text{apply (5) to the second argument of the outer max }\} \\ & \max(a, \max(\min(b, c), \min(b, d))) \\ \equiv & \{\text{delete the redundant inner max }\} \\ & \max(a, \min(b, c), \min(b, d)) \end{array}$$

Applying (5) once more yields the min-max normal form:

$$\min(\max(a, b), \max(a, b, d), \max(a, b, c), \max(a, c, d))$$

which can be simplified to:

$$\min(\max(a, b), \max(a, c, d))$$

because

$$\min(\max(a,b),\max(a,b,d),\max(a,b,c)) \equiv \min(\max(a,b))$$

Next, we present an important theorem showing that a quasi-convex polyhedron can be put into two kinds of normal forms: maximal normal form and minimal normal form.

Theorem 1 *A quasi-convex polyhedron can be put into the following maximal normal form:*

$$\max(a_{1,1}, a_{1,2}, \cdots) \geq 0 \wedge \max(a_{2,1}, a_{2,2}, \cdots) \geq 0 \wedge \cdots$$

and the following minimal normal form:

$$\min(a_{1,1}, a_{1,2}, \cdots) \geq 0 \vee \min(a_{2,1}, a_{2,2}, \cdots) \geq 0 \vee \cdots$$

where $a_{i,j}$ are affine expressions free of max *and* min *operators.*

Proof. For reasons of symmetry, it suffices to show that a quasi-convex polyhedron can always be put into the maximal normal form. The iteration space of a generic loop nest given in (4) is defined by:

$$L_1 \leq x_1,\ \cdots,\ L_n \leq x_n,\ x_1 \leq U_1,\ \cdots,\ x_n \leq U_n$$

For every constraint $L_k \leq x_k$, we apply Lemma 2 to put L_k into the max-min normal form and denote the result as follows:

$$\max(\min(b^k_{1,1}, b^k_{1,2}, \cdots),\ \min(b^k_{2,1}, b^k_{2,2}, \cdots),\ \cdots)$$

Then $L_k \leq x_k$ becomes:

$$\max(\min(b^k_{1,1}, b^k_{1,2}, \cdots),\ \min(b^k_{2,1}, b^k_{2,2}, \cdots),\ \cdots) \leq x_k$$

which is equivalent to by Lemma 1:

$$\min(b^k_{1,1}, b^k_{1,2}, \cdots) \leq x_k \wedge \min(b^k_{2,1}, b^k_{2,2}, \cdots) \leq x_k \wedge \cdots$$

Multiplying the two sides of each $\min(b^k_{i,1}, b^k_{i,2}, \cdots) \leq x_k$ by -1 and simplifying according to Lemma 1 yields:

$$\max(-b^k_{1,1} + x_k, -b^k_{1,2} + x_k, \cdots) \geq 0 \wedge \max(-b^k_{2,1} + x_k, -b^k_{2,2} + x_k, \cdots) \geq 0 \wedge \cdots$$

The upper bound constraints $x_k \leq U_k$ can be dealt with similarly as $-U_k \leq -x_k$. ■

In this theorem, $\max(a_{k,1}, a_{k,2}, \cdots) \geq 0$ can be equivalently replaced with $a_{k,1} \geq 0 \vee a_{k,2} \geq 0 \vee \cdots$ and $\min(a_{k,1}, a_{k,2}, \cdots) \geq 0$ with $a_{k,1} \geq 0 \wedge a_{k,2} \geq 0 \wedge \cdots$. It then becomes clear that the maximal normal form corresponds to the usual conjunctive normal from and the minimal normal form corresponds to the usual disjunctive normal form. This way, it is easy to see that a quasi-convex polyhedron, once put into the minimal normal form, is simply a union of convex polyhedra. This important fact is stated as a corollary of Theorem 1.

Corollary 1 *A quasi-convex polyhedron is a union of a finite number of convex polyhedra, and vice versa.*

When generating code to scan the transformed iteration space, we require the transformed iteration space be initially put into the maximal normal form. A constraint $\max(a) \geq 0$ with a single argument represents an affine constraint $a \geq 0$. A constraint $\max(a_1, a_2, \cdots) \geq 0$ with more than one argument is referred to as a max-*type constraint*.

The code generation problem concerning us in this paper is defined precisely as follows. Let the iteration space of a loop nest be given in the maximal normal form:

$$\Phi \;=\; \{x \mid M_1(x) \geq 0 \wedge M_2(x) \geq 0 \wedge \cdots\}$$

where $M_k(x) \geq 0$ are max-type constraints. Let $T \in \mathbb{Z}^{n \times n}$ be a non-singular loop transformation matrix: $Tx = y$. The transformed iteration space under T is given by $\Phi' \cap \mathcal{L}(T)$, where

$$\Phi' \;=\; \{y \mid M_1(T^{-1}y) \geq 0 \wedge M_2(T^{-1}y) \geq 0 \wedge \cdots\}$$

Clearly, Φ' is also a quasi-convex polyhedron. Our objective is to construct DO loops to scan the points in the transformed iteration space $\Phi' \cap \mathcal{L}(T)$ in lexicographic order.

Note that the transformed iteration space $\Phi' \cap \mathcal{L}(T)$ is not a quasi-convex polyhedron unless T is a unimodular matrix, in which case $\mathcal{L}(T) = \mathbb{Z}^n$. In general, the transformed iteration space is the intersection of a lattice and a quasi-convex polyhedron, which will be referred to as a *quasi-convex polyhedral lattice*.

Let us put the iteration space of (3) into a maximal normal form. This iteration space is defined by the loop bounds in (3) as follows:

$$\left.\begin{array}{r} 1 \leq x_1 \\ \min(3, \max(5 - x_1, 1)) \leq x_2 \\ x_1 \leq 6 \\ x_2 \leq \max(10 - 3x_1, (x_1 + 6)/2) \end{array}\right\}$$

The first and third are affine constraints. It is simple to change the last constraint into a max-type constraint: $\max(10 - 3x_1 - x_2, x_1 - 2x_2 + 6) \geq 0$. We now apply Theorem 1 to transform the second constraint into two max-type constraints.

$$\begin{array}{cl} & \min(3, \max(5 - x_1, 1)) \leq x_2 \\ \equiv & \{\ \min(3, \max(5 - x_1, 1)) \equiv \max(\min(3, 5 - x_1), \min(3, 1)) \text{ by Lemma 2 }\} \\ & \max(\min(3, 5 - x_1), 1) \leq x_2 \\ \equiv & \{\text{ Property (1) in Lemma 1 }\} \\ & \min(3, 5 - x_1) \leq x_2 \ \wedge\ 1 \leq x_2 \\ \equiv & \{\text{ algebraic manipulation, according to Lemma 1 }\} \\ & \max(x_2 - 3, x_1 + x_2 - 5) \geq 0 \ \wedge\ x_2 - 1 \geq 0 \end{array}$$

So the iteration space is put into the following maximal normal form:

$$\left.\begin{array}{r} x_1 - 1 \geq 0 \\ \max(x_2 - 3, x_1 + x_2 - 5) \geq 0 \\ x_2 - 1 \geq 0 \\ -x_1 + 6 \geq 0 \\ \max(-3x_1 - x_2 + 10, x_1 - 2x_2 + 6) \geq 0 \end{array}\right\} \tag{6}$$

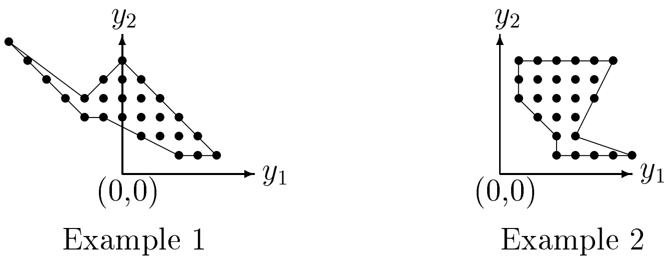


Figure 3: The transformed iteration spaces.

where the first max-type constraint specifies the non-convex surface on the bottom in Figure 1, and the second specifies the one on the top.

We shall use the following two examples to illustrate the development of our code generation method. Both are involved with unimodular transformations only (Figure 3).

Example 1 Generate code for a loop skewing transformation of the double loop (3):

$$T \;=\; \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}, \quad T\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

Since T is unimodular, the new iteration space given below

$$\left.\begin{array}{r} y_1 + y_2 - 1 \geq 0 \\ \max(y_2 - 3, y_1 + 2y_2 - 5) \geq 0 \\ y_2 - 1 \geq 0 \\ -y_1 - y_2 + 6 \geq 0 \\ \max(-3y_1 - 4y_2 + 10, y_1 - y_2 + 6) \geq 0 \end{array}\right\}$$

is obtained directly from (6) by loop variable substitutions.

Example 2 Generate code for a loop interchange of the double loop (3):

$$T \;=\; \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad T\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

Since T is unimodular, the new iteration space given below

$$\left.\begin{array}{rcl} y_2 - 1 & \geq & 0 \\ \max(y_1 - 3, y_1 + y_2 - 5) & \geq & 0 \\ y_1 - 1 & \geq & 0 \\ -y_2 + 6 & \geq & 0 \\ \max(-y_1 - 3y_2 + 10, -2y_1 + y_2 + 6) & \geq & 0 \end{array}\right\}$$

is obtained directly from (6) by loop variable substitutions.

3 Code Generation

Given the transformed iteration space $\Phi' \cap \mathcal{L}(T)$, we want to construct a set of possibly perfectly nested loops to iterate over the iterations in that space in lexicographic order. We will make use of existing code generation techniques for convex iteration spaces. Conceptually, we can divide our code generation process into two phases:

1. Generate loops to scan the quasi-convex polyhedron Φ'. All loops will have unity step sizes because Φ' is dense. Due to the non-convexity of Φ', these loops are not perfectly nested. In the case that a loop variable is not explicitly bounded by some existing constraints, we generate the missing loop bounds using, for example, the Fourier-Motzkin elimination or Feautrier's PIP.

2. For each loop generated in the first phase:

   $$\text{do } y_k = L_k,\ U_k$$

   we replace it with:

   $$\text{do } y_k = L_k + \delta_k,\ U_k,\ Step_k$$

   so that the points in Φ' but not in $\mathcal{L}(T)$ are skipped. Here, δ_k and $Step_k$ are calculated according to (1) and (2).

Therefore, we focus in this section on generating code to scan a quasi-convex polyhedron. It is assumed that a quasi-convex polyhedron is specified by a conjunction of max-type constraints. We first characterise max-type constraints by distinguishing those that are loop bounds from those that are not. Based on this characterisation, we identify a class of quasi-convex polyhedra called triangular polyhedra, each of which can be scanned by a perfect loop nest. We present our code generation algorithm in four separate subsections. We first give a quick review on code generation for a convex polyhedron. We then discuss how to construct a perfect loop nest to scan a triangular polyhedron. To scan an arbitrary quasi-convex polyhedron, we describe a recursive algorithm that generates code by conceptually partitioning the quasi-convex polyhedron into a number of triangular polyhedra. Finally, we discuss code generation for a quasi-convex polyhedral lattice.

3.1 Loop Bound Constraints

Let $y_1, \cdots, y_n$ be loop variables ordered from the outermost loop to the innermost loop. Consider a generic affine constraint $b_0 + b_1 y_1 + \cdots + b_k y_k \geq 0$. If $b_k \neq 0$, this constraint is a loop bound for loop variable y_k. It is a *lower bound* if $b_k > 0$ or an *upper bound* if $b_k < 0$.

Next, consider a generic max-type constraint $\max(a_1, a_2, \cdots) \geq 0$, where each a_i is an affine expression of the form $a_{i,0} + a_{i,1} y_1 + \cdots + a_{i,m_i} y_{m_i}$. This max-type constraint is a *lower bound* (*upper bound*) of y_k if every affine constraint $a_i \geq 0$ is a lower bound (upper bound) of y_k. If $\max(a_1, a_2, \cdots) \geq 0$ is a loop bound of y_k, then each argument a_i has the form $a_{i,0} + a_{i,1} y_1 + \cdots + a_{i,k} y_k$. Let $a'_i = (a_{i,0} + a_{i,1} y_1 + \cdots + a_{i,k-1} y_{k-1})/a_{i,k}$. According to the properties stated in Lemma 1, we can rewrite $\max(a_1, a_2, \cdots) \geq 0$ to $y_k \geq \min(-a'_1, -a'_2, \cdots)$

if it is a lower bound (i.e., if $\forall\ i, a_{i,k} > 0$) or $y_k \leq \max(-a'_1, -a'_2, \cdots)$ if it is an upper bound (i.e., if $\forall\ i, a_{i,k} < 0$).

For instance, both max-type constraints in Example 1 are loop bounds of y_2. Specifically, $\max(y_2 - 3, y_1 + 2y_2 - 5) \geq 0$ is a lower bound of y_2 because $y_2 - 3 \geq 0$ and $y_1 + 2y_2 - 5 \geq 0$ are lower bounds of y_2. According to Lemma 1, this constraint can be rewritten to $y_2 \geq \min(3, (5 - y_1)/2)$. The other constraint $\max(-3y_1 - 4y_2 + 10, y_1 - y_2 + 6) \geq 0$ is an upper bound of y_2. This is obvious once it is put into the form $y_2 \leq \max((10 - 3y_1)/4, y_1 + 6)$.

However, there are max-type constraints that are not loop bounds for any loop variables. In fact, both max-type constraints in Example 2 are not loop bounds: $\max(y_1 - 3, y_1 + y_2 - 5) \geq 0$ contributes a lower bound to y_1 and a lower bound to y_2, and $\max(-y_1 - 3y_2 + 10, -2y_1 + y_2 + 6) \geq 0$ contributes both a lower and upper bound to y_2.

In general, every max-type constraint, if not redundant, represents a non-convex surface for a quasi-convex polyhedron. However, for correctness, a loop bound constraint (redundant or not) can always be used as a loop bound for some loop variable. It is exactly those non-loop bound constraints that complicate the code generation process. For each non-loop bound constraint, we will need to extract the containing loop bounds by some means. One obvious way is to split it into as many loop bound constraints as possible. As a consequence, the loops generated are non-longer perfectly nested. However, in the special case when all max-type constraints that specify a quasi-convex polyhedron are loop bounds, we are still able to construct a perfect loop nest to scan the quasi-convex polyhedron. This class of quasi-convex polyhedra is defined below.

Definition 1 (**Triangular Polyhedra**) A quasi-convex polyhedron is *triangular* if all the specifying constraints are loop bounds.

This name is chosen to generalise the notion of a triangular system of inequalities [3].

The transformed iteration space in Example 2 is not triangular because the two max-type constraints are not loop bounds. However, the one in Example 1 is a triangular polyhedron but not a convex polyhedron. Thus, all convex polyhedra form a strict subclass of the class of triangular polyhedra.

3.2 Convex Polyhedra

It is possible to use a perfect loop nest to scan a convex polyhedron. All loops in the loop nest have unity step sizes. Thus, all one needs to do is to calculate the loop bounds.

As an example, the following constraints define the convex polyhedron shown in Figure 4:

$$\left.\begin{array}{rcl} -y_1 + 3 & \geq & 0 \\ y_1 - y_2 + 1 & \geq & 0 \\ y_2 - 1 & \geq & 0 \\ y_1 + y_2 - 3 & \geq & 0 \end{array}\right\}$$

A convex polyhedra can be specified by affine constraints only. All affine constraints are loop bounds for some variables. But some loop variables may not be bounded explicitly by any existing constraints. In this example, neither of the four constraints is a lower bound of y_1. Previous work indicates that we can use the Fourier-Motzkin elimination or Feautrier's PIP to construct these missing loop bounds. Using the Fourier-Motzkin elimination, we find

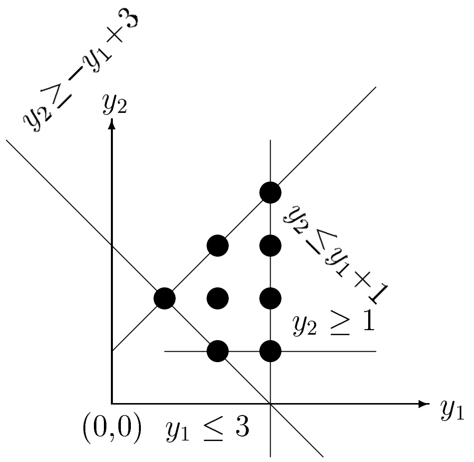


Figure 4: Code generation for a convex polyhedron.

a lower bound for y_1: $y_1 \geq 1$. This enables us to generate the following perfect loop nest to scan the given convex polyhedron:

```
do y1 = 1, 3
   do y2 = max(-y1 + 3, 1), y1 + 1
      Enumerate (y1, y2)
```

In general, if a loop variable has several lower bound constraints, we take a maximum of these constraints as the lower bound for that variable. On the other hand, if a loop variable has several upper bound constraints, we take a minimum of these constraints as the upper bound for the variable.

Instead of $y_1 \geq 1$, we could have used any lower bound constraint for y_1 as long as it is redundant with respect to the original set of constraints. For example, choosing $y_1 \geq -10$ would yield:

```
do y1 = -10, 3
   do y2 = max(-y1 + 3, 1), y1 + 1
      Enumerate (y1, y2)
```

Both programs will iterate over exactly the same iteration space. But the first is preferable since it is more efficient. For the second double loop, the majority of the iterations for the outer loop do not correspond to any iterations in the inner loop.

3.3 Triangular Polyhedra

Like a convex polyhedron, we can use a perfect loop nest to scan a triangular polyhedron. Therefore, all that remains to be done is to generate the lower and upper bounds for every loop variable. We know that all the constraints that specify a triangular polyhedron are loop bounds. If every loop variable has at least one lower bound and at least one upper bound, we can generate code straightforwardly. Otherwise, we can construct the missing loop bounds in a similar manner as we did for a convex polyhedron.

Let Q be a triangular polyhedron being scanned. Let us say that loop variable y_k does not have a lower bound and/or an upper bound. Let C be the convex hull of Q. We can use, for example, the Fourier-Motzkin elimination to generate from C a lower bound $L_k \le y_k$ and/or an upper bound $y_k \le U_k$. The loop nest constructed this way will iterate exactly over the points in Q because all missing loop bounds introduced are redundant with respect to Q.

Let us show how to generate code for the transformed iteration space in Example 1. The first three constraints are lower bounds of y_2 and the other two are upper bounds of y_2. However, the lower and upper bounds of y_1 are not directly available. Using the Fourier-Motzkin elimination on the convex-hull of the transformed iteration space, we obtain $-6 \le y_1 \le 5$ as the loop bounds for y_1. In this two-dimensional case, a visual inspection of Figure 3 can verify that this is indeed the loop bounds for y_1. We will then generate the following code:

$$
\begin{array}{l}
\textsf{do } y_1 = -6,\ 5 \\
\quad \textsf{do } y_2 = \max(1, 1-y_1, \min(3, \lceil (5-y_1)/2 \rceil)),\ \min(6-y_1, \max(\lfloor (10-3y_1)/4 \rfloor, 6+y_1)) \\
\qquad a(y_1+y_2, y_2) = a(y_1+y_2-1, y_2)
\end{array}
$$

Recall that each max-type constraint that is a lower bound has the form $y_k \ge \min(a_1, a_2, \cdots)$, and each upper bound constraint has the form $y_k \le \max(a_1, a_2, \cdots)$. Therefore, in the transformed code, each lower bound is always in the max-min normal form and each upper bound is always in the min-max normal form.

3.4 Quasi-Convex Polyhedra

Given a quasi-convex polyhedron, we will construct a set of imperfectly nested loops to enumerate the polyhedron in lexicographic order. As an example, we will generate the following code to scan the transformed iteration space in Example 2:

$$
\begin{array}{ll}
 & \textsf{do } y_1 = 1,\ 2 \\
 & \quad \textsf{do } y_2 = 5 - y_1,\ 6 \\
S_1: & \qquad a(y_2, y_1) = a(y_2 - 1, y_1) \\
 & \textsf{do } y_1 = 3,\ 7 \\
 & \quad \textsf{do } y_2 = 1,\ \lfloor (10 - y_1)/3 \rfloor \\
S_2: & \qquad a(y_2, y_1) = a(y_2 - 1, y_1) \\
 & \quad \textsf{do } y_2 = \max(\lceil (11 - y_1)/3 \rceil, 2y_1 - 6),\ 6 \\
S_3: & \qquad a(y_2, y_1) = a(y_2 - 1, y_1)
\end{array}
$$

In the transformed code, each statement is surrounded by n loops, one loop for each dimension of the transformed iteration space. The iteration space for a statement is defined by the loop bounds of all n surrounding loops. Such an iteration space is trivially a triangular polyhedron. The iteration spaces for all statements must partition the quasi-convex polyhedron being scanned. Otherwise, there must be an iteration that is either not enumerated at all or is enumerated more than once. In this example, the three statements S_1, S_2 and S_3 are associated with the following iteration spaces (which all happen to be convex polyhedra):

$$
\left.\begin{array}{r}
y_1 + y_2 - 5 \ge 0 \\
y_1 - 1 \ge 0 \\
-y_2 + 6 \ge 0 \\
y_1 - 3 < 0
\end{array}\right\} : S_1
\qquad
\left.\begin{array}{r}
y_1 - 3 \ge 0 \\
-y_1 + 7 \ge 0 \\
-y_1 - 3y_2 + 10 \ge 0 \\
y_2 - 1 \ge 0
\end{array}\right\} : S_2
\qquad
\left.\begin{array}{r}
y_1 - 3 \ge 0 \\
-y_2 + 6 \ge 0 \\
-2y_1 + y_2 + 6 \ge 0 \\
-y_1 - 3y_2 + 10 < 0
\end{array}\right\} : S_3
\qquad (7)
$$

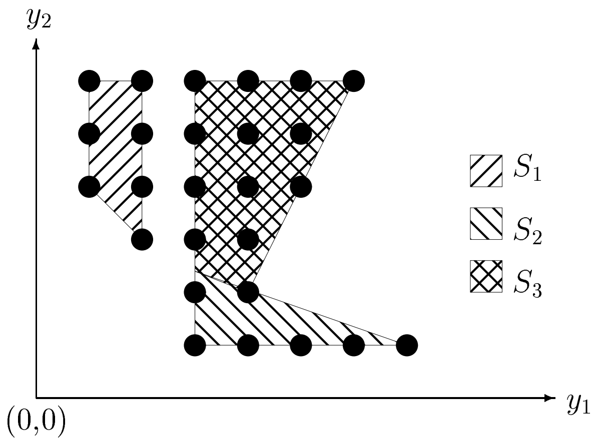


Figure 5: Partitioning of the transformed iteration space in Example 2.

These three convex polyhedra partition the transformed iteration space in Example 2 as shown in Figure 5. Therefore, code generation for a quasi-convex polyhedron can be understood as a process of partitioning that polyhedron into a number of triangular polyhedra.

We must address two issues when constructing a set of imperfectly nested loops:

- Generation of the structure of the loop nest.
- Generation of the loop bounds for each individual loop.

Since a quasi-convex polyhedron is dense, all loop step sizes are 1. Once we have generated the loops, it is simple to generate the loop bodies. In the transformed code, a copy of the original loop body appears inside each innermost loop, and all original loop variables are replaced with appropriate functions of the new loop variables according to the loop transformation.

A quasi-convex polyhedron is specified by a conjunction of max-type constraints. When generating code, we pay attention to only those max-type constraints that are not loop bounds. A non-loop bound constraint contributes loop bounds to more than one loop variable and/or both a lower and upper bound to the same loop variable. Consider again the two max-type constraints in Example 2. Both are not loop bounds: $\max(y_1 - 3, y_1 + y_2 - 5) \geq 0$ contributes one lower bound each to y_1 and y_2, respectively, and $\max(-y_1 - 3y_2 + 10, -2y_1 + y_2 + 6) \geq 0$ contains both a lower and upper bound for y_2.

When generating the transformed code, we must extract the loop bounds from a non-loop bound constraint. To this end, we split the non-loop bound constraint into several disjoint constraints. Each splitting operation effectively partitions the quasi-convex polyhedron under consideration into as many quasi-convex polyhedra as there are the total number of disjoint constraints. All splitting operations together partition the quasi-convex polyhedron being scanned into a number of triangular polyhedra. Essentially, we generate the transformed code based entirely on these splitting operations:

- Each splitting operation partitions a quasi-convex polyhedron Q under consideration into several disjoint quasi-convex polyhedra. To scan Q, we sequentially combine the code segments generated for scanning these disjoint polyhedra. Therefore, all splitting operations determine the restructure of the loop nest.

- Each splitting operation produces the loop bounds for some loop variables. Once all constraints are loop bound constraints, we can generate the loop bounds for all loop variables. In the case that some loop variables are not explicitly bounded by any existing constraints, we use, for example, the Fourier-Motzkin elimination to generate the missing loop bounds for these loop variables.

3.4.1 Partitioning by Splitting Non-Loop Bound Constraints

We introduce the concept of y_k-constraints so that we can talk about conveniently about extracting loop bounds from non-loop bound constraints.

Definition 2 (y_k**-Constraints**) Let $\max(a_1, a_2, \cdots) \geq 0$ be a max-type constraint that is not a loop bound. This constraint is a y_k*-constraint* if every $a_i \geq 0$ is a loop bound of y_j, where $j \geq k$ and if there is at least one a_i such that $a_i \geq 0$ is a loop bound of y_k.

That is, a max-type constraint is y_k-constraint if it contributes loop bounds to y_k but not to any surrounding loop variables $y_1, \cdots, y_{k-1}$.

Let $\max(a_1, a_2, \cdots) \geq 0$ be a y_k-constraint. We can always divide the list of its arguments into three sublists L, U and I such that

- $\max(L) \geq 0$ is a lower bound of y_k,
- $\max(U) \geq 0$ is an upper bound of y_k, and
- $\max(I) \geq 0$ contributes loop bounds to some of the inner loop variables $y_{k+1}, \cdots, y_n$.

Using the notations of L, U and I, we can express $\max(a_1, a_2, \cdots) \geq 0$ as $\max(L, U, I) \geq 0$. By the definition of y_k-constraints, L and U cannot be both empty but I can be empty.

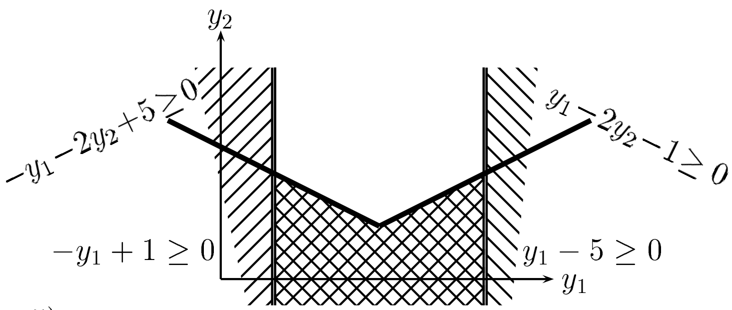
Consider the two max-type constraints in Example 2. Constraint $\max(y_1-3, y_1+y_2-5) \geq 0$ is a y_1-constraint. With $L = (y_1 - 3)$ and $I = (y_1 + y_2 - 5)$, this constraint can be rewritten as $\max(L, I) \geq 0$ such that $\max(L) \geq 0$ is a lower bound of y_1 and $\max(I) \geq 0$ is a lower bound of y_2. Constraint $\max(-y_1 - 3y_2 + 10, -2y_1 + y_2 + 6) \geq 0$ is a y_2-constraint and can be put into the form $\max(L, U) \geq 0$, where $L = (-2y_1 + y_2 + 6)$ and $U = (-y_1 - 3y_2 + 10)$. Here, $\max(L) \geq 0$ is a lower bound of y_2 and $\max(U) \geq 0$ is an upper bound of y_2.

To construct the loop bounds for loop variable y_k, we need to extract the loop bounds of y_k from all y_k-constraints. We accomplish this by splitting a y_k-constraint into two or three disjoint constraints. Let

$$C \;=\; \max(L, U, I) \geq 0$$

be a y_k-constraint. To avoid making case distinctions, L, U or I, if empty, will be set to $(-\infty)$. We extract the lower bound $\max(L) \geq 0$ and the upper bound $\max(U) \geq 0$ by splitting C into three disjoint constraints such that $C \equiv C_1 \vee C_2 \vee C_3$, where

$$\begin{aligned} C_1 &= \max(U) \geq 0 \\ C_2 &= \max(U) < 0 \wedge \max(L) < 0 \wedge \max(I) \geq 0 \\ C_3 &= \max(U) < 0 \wedge \max(L) \geq 0 \end{aligned} \tag{8}$$



$L=(y_1-5)$
$U=(-y_1+1)$
$I\ =(-y_1-2y_2+5, y_1-2y_2-1)$

$\max(-y_1+1)\geq 0$
$\max(-y_1+1)<0 \wedge \max(y_1-5)<0 \wedge \max(-y_1-2y_2+5, y_1-2y_2-1)\geq 0$
$\max(-y_1+1)<0 \wedge \max(y_1-5)\geq 0:$ $\quad \max(-y_1+1) < 0$ is redundant

Figure 6: Extraction of loop bounds of y_1 from a y_1-constraint $\max(y_1-5, -y_1+1, -y_1-2y_2+5, y_1-2y_2-1)\geq 0$, which defines the shaded non-convex region.

It is understood that $\max(-\infty) < 0 \equiv$ **true** and $\max(-\infty) \geq 0 \equiv$ **false**. Therefore, if $L=(-\infty)$, $C_3 \equiv$ **false** can be ignored, if $U=(-\infty)$, $C_1 \equiv$ **false** can be ignored, and if $I=(-\infty)$, $C_2 \equiv$ **false** can be ignored.

Figure 6 shows an example of a splitting operation. By construction, it is easy to see that C_1, C_2 and C_3 are mutually disjoint. We prove $C \equiv C_1 \vee C_2 \vee C_3$ as follows:

$$
\begin{array}{ll}
 & \max(L,U,I)\geq 0 \\
\equiv & \{\text{ Property (2) in Lemma 1 }\} \\
 & (\max(U)\geq 0 \vee \max(L,I)\geq 0 \\
\equiv & \{\text{ by the fact that } x\vee y = x \vee(\neg x \wedge y)\text{: } x=\max(U)\geq 0;\ y=\max(L,I)\geq 0\ \} \\
 & (\max(U)\geq 0)\vee(\max(U)<0\wedge \max(L,I)\geq 0) \\
\equiv & \{\text{ Property (2) in Lemma 1 }\} \\
 & (\max(U)\geq 0)\vee(\max(U)<0\wedge(\max(L)\geq 0\vee \max(I)\geq 0)) \\
\equiv & \{\text{ again by the fact } x\vee y=(\neg x\wedge y)\vee x\text{: } x=\max(L)\geq 0;\ y=\max(I)\geq 0\} \\
 & (\max(U)\geq 0)\vee(\max(U)<0\wedge(\max(L)<0\wedge \max(I)\geq 0\vee \max(L)\geq 0)) \\
\equiv & \{\text{ algebraic simplification }\} \\
 & (\max(U)\geq 0)\vee(\max(U)<0\wedge \max(L)<0\wedge \max(I)\geq 0)\vee(\max(U)<0\wedge \max(L)\geq 0)
\end{array}
$$

Let Q be a conjunction of max-type constraints. Let $C=\max(L,U,I)\geq 0$ be a y_k-constraint in Q to be eliminated. By splitting C into the three disjoint constraints as in (8), we effectively partition Q into three disjoint quasi-convex polyhedra:

$$
\begin{array}{lcl}
Q_1 & = & (Q\setminus C)\wedge C_1 \\
Q_2 & = & (Q\setminus C)\wedge C_2 \\
Q_3 & = & (Q\setminus C)\wedge C_3
\end{array} \tag{9}
$$

where the notation $Q\setminus C$ means the Q with C removed. Of course, if $C_k = \mathsf{false}$, Q_k represents an empty set and will be ignored. As Figure 6 (or rather its version in higher dimension) illustrates, Q is partitioned along two constraints $\max(U) \geq 0$ and $\max(L) \geq 0$. When L and U are nonempty lists, $\max(U) = 0$ and $\max(L) = 0$ represent two piece-wise separating hyperplanes that separate strictly Q_1, Q_2 and Q_3 from each other.

3.4.2 Code Generation Algorithm

Given a quasi-convex polyhedron, we need to generate code to enumerate the points in the polyhedron in lexicographic order. In the transformed code, the outermost loops iterate over the values of the first loop variable y_1, those immediate inside the outermost loops iterate over the values of the second loop variable y_2, and so on. We generate the code recursively, level by level, starting at the outermost level for y_1. Our code generation algorithm is given in Figure 7. Below we explain the algorithm and illustrate it using an example.

Function `GenMain`(Q) is the main program; it calls `MaximalNormalForm`(Q) to put Q in the maximal normal form and then `GenLoopNest` to generate code, starting at the first level.

Function `GenLoopNest` splits recursively non-loop bound constraints into loop bound constraints. During this process, it calls `GenLoop` to generate the loop bounds for a loop variable, `GenStatements` to generate the statements in an innermost loop, and `ElimRedundancies` to remove redundant constraints in the quasi-convex polyhedron under consideration. It also defines the structure of the transformed loop nest.

When `GenLoopNest`($Q, level$) is called, the $level - 1$ outermost loops $y_1, \cdots, y_{level-1}$ have already been generated for scanning the first $level-1$ dimensions of Q. When returning to the caller, `GenLoopNest`($Q, level$) will have generated the inner loops controlled by $y_{level}, \cdots, y_n$ for scanning the remaining $n-level+1$ dimensions of Q. In general, these $n-level+1$ inner loops are not perfectly nested. Thus, several loops at the same depth k may be controlled by the same loop variable y_k, $level \leq k \leq n$.

When working at a single level $level$, `GenLoopNest`($Q, level$) performs two tasks:

Extracting the loop bounds of y_{level} from y_{level}-constraints. We choose from Q an arbitrary y_{level}-constraint $C = \max(L, U, I) \geq 0$. We extract from C the loop bounds $\max(L) \geq 0$ and $\max(U) \geq 0$ by splitting C into three disjoint constraints C_1, C_2 and C_3 as in (8). This partitions Q into three disjoint quasi-convex polyhedra Q_1, Q_2 and Q_3 as in (9). Because of this splitting operation, some constraints in these three regions may become redundant. Such redundant constraints are detected and eliminated in procedure `ElimRedundancies`. We then repeat this splitting process on each of the three regions Q_1, Q_2 and Q_3 at the same level if it is nonempty. If we are left with no y_{level}-constraints, we move to the next level $level + 1$ unless $level = n$.

Positioning the loops of y_{level} relative to each other. We use L_k to denote the code segments generated for Q_k, $1 \leq k \leq 3$; the outermost loop in L_k is a loop for y_{level}. The code for Q is a sequential composition of the three code segments L_1, L_2 and L_3, as denoted by the notation **return**(L_1 L_2 L_3) in `GenLoopNest`. This produces the desired result so that all points in Q are enumerated in lexicographic order.

`GenLoop`($Q, level$) is responsible for generating a loop of y_{level} to scan Q. If Q contains no existing constraints as a lower or upper bound for y_{level}, we use, for example, the Fourier-Motzkin elimination to generate the missing loop bound(s) from the convex hull of Q. Each lower bound constraint of y_{level} has the form $y_k \geq \min(a_1, a_2, \cdots)$, and each upper bound

```
/* GenMain calls GenLoopNest to scan Q, starting at level=1 */
GenMain(Q)
    Q := MaximalNormalForm(Q)
    return GenLoopNest(Q, 1)

/* GenLoopNest generates code recursively, level by level */
GenLoopNest(Q, level)
    if (Q contains some y_level-constraints) then
        Select one arbitrarily: C := max(L, U, I) ≥ 0
        Let C_1, C_2 and C_3 be as defined in (8)
        Let Q_1, Q_2 and Q_3 be as defined in (9)
        do k = 1, 3
            ElimRedundancies(Q_k)
            if Empty(Q_k) then
                L_k := (null code)
            else
                L_k := GenLoopNest(Q_k, level)
        return (L_1 L_2 L_3) /* sequential combination of the codes */
    else
        Loop := GenLoop(Q, level)
        if (level = n) then
            LoopBody := GenStatements(Q)
        else
            LoopBody := GenLoopNest(Q, level + 1)
        return (Loop LoopBody)

/* GenLoop generates the loop bounds for a loop of y_level */
GenLoop(Q, level)
    if (Q does not contain a lower or upper bound constraint for y_level) then
        Construct the missing loop bound(s) from the convex hull of Q using,
        for example, the Fourier-Motzkin elimination or or Feautrier's PIP
    Let all lower bounds of y_level be: y_level ≥ min(ℓ^level_{k,1}, ℓ^level_{k,2}, ···), k = 1, 2, ···
    L_level := max(min(ℓ^level_{1,1}, ℓ^level_{1,2}, ···), min(ℓ^level_{2,1}, ℓ^level_{2,2}, ···), ···)
    Let all upper bounds of y_level be: y_level ≤ max(u^level_{k,1}, u^level_{k,2}, ···), k = 1, 2, ···
    U_level := min(max(u^level_{1,1}, u^level_{1,2}, ···), max(u^level_{2,1}, u^level_{2,2}, ···), ···)
    return (do y_level = L_level, U_level)

/* Eliminate redundant constraints in Q */
ElimRedundancies(Q)
    Q := MinimalNormalForm(Q)
    for (each conjunct C in Q) do
        if Empty(C) then Delete C from Q
    for (each conjunct C in Q) do
        for (each affine constraint c in C) do
            Delete c from C if c is redundant in C
```

Figure 7: Code generation algorithm.

constraint has the form $y_k \le \max(a_1, a_2, \cdots)$. Therefore, in the transformed code, the lower bound of y_{level} is always in the max-min normal form, and the upper bound of y_{level} is always in the min-max normal form.

`ElimRedundancies` is an important component of the algorithm. In `GenLoopNest`, every time before we call the same function `GenLoopNest`$(Q, level)$ recursively, we first call `ElimRedundancies`(Q) to eliminate redundant constraints in Q. `ElimRedundancies` works by using Q in the minimal normal form. In our implementation, we represent the constraints for a quasi-convex polyhedron Q in a linked data structure (a multiply linked list). We simultaneously represent Q in both the maximal and minimal normal forms. Therefore, the function `MinimalNormalForm` that converts Q from the maximal to the minimal normal form does not incur any extra cost. Once in the minimal normal form, Q is a union of convex polyhedra. We first eliminate all empty convex polyhedra, if any, and then eliminate all redundant constraints in each convex polyhedron in Q. A constraint in a constraint system is called *redundant* if it is implied by the other constraints in the system. Some techniques for eliminating redundant constraints are described in [6]. By eliminating redundant max-type constraints as soon as possible at earlier levels, we will eliminate unnecessary splitting operations, and consequently avoid generating more loops than necessary. By eliminating redundant affine constraints, loop bounds will be simplified.

To understand our code generation algorithm, we trace the actions of the recursive function `GenLoopNest` when it generates code for the transformed iteration space in Example 2. We display the recursive calls that `GenLoopNest` makes to itself in Figure 8, often known as the *call tree*. In each recursive call, we show the values of the two parameters Q and *level* (with the second being displayed below the first). If Q has a y_{level}-constraint, then the `then` branch of `GenLoopNest` will be executed. In this case, several recursive calls will be created. When Q contains no y_{level}-constraints, the `else` branch of `GenLoopNest` will be reached. This creates a call to `GenLoop` to generate a loop for y_{level} and a call to `GenLoopNest` to generate loops for $y_{level+1}$ recursively at the next level.

To illustrate `ElimRedundancies`, consider the first call `GenLoopNest`$(Q, 1)$. As soon as Q is split into Q_2 and Q_3, `ElimRedundancies` is called to eliminate redundant constraints in Q_2 and Q_3. In the case of `ElimRedundancies`(Q_2), Q_2 is a union of the following two convex polyhedra:

$$\left.\begin{array}{r} y_2-1\ge 0\\ y_1-3<0\\ y_1+y_2-5\ge 0\\ y_1-1\ge 0\\ -y_2+6\ge 0\\ -y_1-3y_2+10\ge 0 \end{array}\right\} \qquad \left.\begin{array}{r} y_2-1\ge 0\\ y_1-3<0\\ y_1+y_2-5\ge 0\\ y_1-1\ge 0\\ -y_2+6\ge 0\\ -2y_1+y_2+6\ge 0 \end{array}\right\}$$

For each polyhedron, the first and last constraints are redundant; thus, Q_2 is simplified to a single convex polyhedron defined by the remaining four constraints. In the call `ElimRedundancies`(Q_3), the redundant constraint $y_1 - 1 \ge 0$ is detected and removed.

To illustrate `GenLoop`, consider the recursive call `GenLoopNest`$(Q_3, 1)$. Since there are not any y_1-constraints left, `GenLoop`$(Q_3,1)$ is called to generate a loop for y_1. The upper bound of y_1 is not explicitly defined. By constructing the convex hull for Q_3, we use the Fourier-Motzkin elimination to find an upper bound constraint $y_1 \le 7$. In this two-dimensional case,

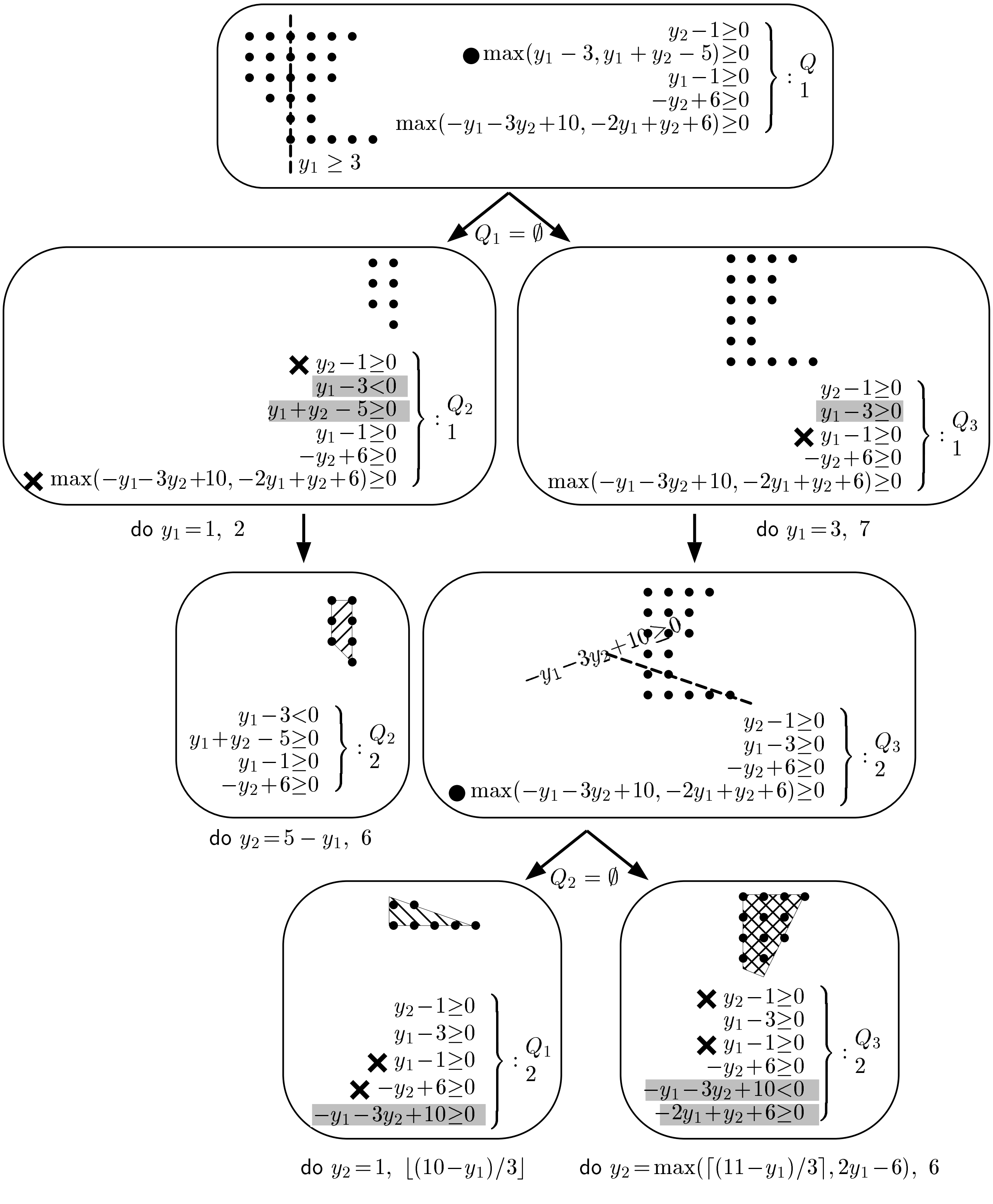


Figure 8: Trace of `GenLoopNest` on the transformed iteration space in Example 2. In each recursive call represented by a box, the constraint selected for splitting is highlighted with a fat dot, the shaded constraints are the ones generated by the splitting operation in the parent call, the constraints indicated with crosses are redundant constraints detected and deleted by `ElimRedundancies` in the parent call, and the loop displayed below the box is generated by `GenLoop` in the current call.

a visual inspection of Figure 5 can verify that this constraint is indeed an upper bound for y_1.

We can observe an interesting property from the call tree shown in Figure 8. We can find all loops needed for scanning Q in a recursive call `GenLoopNest`$(Q, level)$ as follows. Let R be the node representing the recursive call in the call tree in Figure 8. The $level - 1$ outermost loops $y_1, \cdots, y_{level-1}$ are the ones associated with R's ancestor nodes and the innermost loops $y_{level}, \cdots, y_n$ are the ones obtained by a preorder traversal of the subtree whose root is R. During the traversal, it is assumed that the children of a node are visited from left to right. In the special case when `GenLoopNest`$(Q, level)$ is a leaf node in the call tree, Q is scanned by a set of n perfectly nested loops, which are associated with R's ancestor nodes and the R itself. It is clear that all the Q's at the leaf nodes are triangular and they partition the Q at the root of the call tree, i.e., the quasi-convex polyhedron being scanned. There are three leaf nodes in the call tree in Figure 8. These leaves are associated with Q_2, Q_1, Q_3 listed from left to right; they represent the iteration spaces for S_1, S_2 and S_3 in the loop nest (3) shown in Figure 5.

Our code generation algorithm has an exponential time complexity. We generate code by splitting non-loop bound constraints. The total number of recursive calls produced can be exponential on the number of non-loop bound constraints. When working at level *level*, let there be M y_{level}-constraints. Assume the worst case that each y_{level}-constraint has the form $\max(L, U, I) \geq 0$ such L, U and I are all nonempty. On eliminating one such a constraint, three recursive calls are made. To eliminate all M y_{level}-constraints, a total of 3^M recursive calls will be created. In the average case, there will not be these many recursive calls because redundant y_{level}-constraints will be eliminated in `ElimRedundancies`. We will split a y_{level}-constraint if and only if it is non-redundant. For example, by eliminating the redundant y_2-constraint in the recursive call `GenLoopNest`$(Q_2, 2)$, no further recursive calls are necessary from that call. Finally, `GenLoop` is exponential in the worst case when the Fourier-Motzkin elimination is used. In practice, a quasi-convex polyhedron does not have many non-loop bound constraints. Therefore, our code generation algorithm is expected to run efficiently.

3.5 Quasi-Convex Polyhedral Lattices

The basic idea about scanning a quasi-convex polyhedral lattice was summarised at the beginning of Section 3. Let $Q \subset \mathbb{Z}^n$ be a quasi-convex polyhedron and $L \in \mathbb{Z}^{n\times n}$ be a non-singular matrix. To generate code for Q, we first use the algorithm in Figure 7 to construct a loop nest to scan Q. Let a loop of y_k be of the following form:

$$\textsf{do } y_k = L_k,\ U_k$$

we then refine it to:

$$\textsf{do } y_k = L_k + \delta_k,\ U_k,\ Step_k$$

in order to skip the points in Q but not in $\mathcal{L}(T)$, where δ_k and $Step_k$ are computed according to (1) and (2).

4 Other Applications

This section discusses two additional applications in which our code generation algorithm can be used. One application is concerned with the generation of data transfer code for machines with a hierarchical memory, and the other is to construct SPMD (Single Program, Multiple Data) programs with explicit communication code for distributed memory machines. In both applications, we need to construct a set of loops to scan a region of index points. We can apply our code generation algorithm as long as the region to be scanned can be specified as a union of convex polyhedra, possibly intersected by a lattice. Such a region is what we have called a quasi-convex polyhedral lattice.

4.1 Generation of Data Transfer Code

For machines with complex memory hierarchies, a substantial performance gain can be achieved if copies of frequently referenced data structures can be stored in local memories. The original study reported in [8, 9] was restricted to a *single* array reference with a simple indexing function. In addition, all loop nests under consideration were restricted to have rectangular iteration spaces. Later in [2], array indexing functions were generalised to be arbitrary affine expressions and the iteration spaces to convex polyhedra. But still a single array reference was considered. In many cases, however, a variable may be referenced at several points in the loop body, and in addition, these accessed regions may overlap. Using these existing methods, one must copy each reference individually. As a result, data elements that are accessed multiple times are transferred multiple times unnecessarily. We propose to represent multiple references as a union of their accessed regions. As long as the data elements to be transferred can be specified as a quasi-convex polyhedral lattice, we can generate data transfer code so that the same datum is transferred exactly once.

Consider the case of a parallel program for execution in a linear array of P processors consisting of a local memory private to each processor and a global memory shared by all processors:

$$
\begin{array}{l}
\textsf{doall } p = 0,\, P-1 \\
\quad \textsf{do } x_1 = 1, N \\
\quad\quad \textsf{do } x_2 = 1, N \\
\quad\quad\quad \cdots = a(x_1+p, 2x_2) + a(x_1+x_2+p, 2x_2+2)
\end{array}
$$

where p is the "processor index". That is, every processor executes the two inner loops whose iteration space is

$$\{(x_1, x_2) \mid 1 \le x_1, x_2 \le N\} \tag{10}$$

Note that, if so desired, it is possible to choose the indexing function of the LHS (indicated with the three dots) so that different processors write disjoint subsets of the LHS [8].

We assume that array a is initially stored in global memory. We would like to generate the extra code needed to make part of array a accessed by a processor into its local memory.

Using matrix notation, we write the indexing functions of references $a(x_1+p, 2x_2)$ and $a(x_1+x_2+p, 2x_2+2)$ as follows:

$$A_1 x + b_1 = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} p \\ 0 \end{bmatrix} \quad A_2 x + b_2 = \begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} p \\ 2 \end{bmatrix}$$

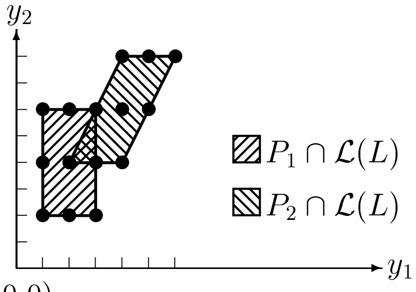


Figure 9: The array elements accessed at processor $p = 0$ ($N = 3$).

It can be verified that $\mathcal{L}(A_1) = \mathcal{L}(A_2) = \mathcal{L}(L)$, where

$$L = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$$

and $b_1, b_2 \in \mathcal{L}(L)$. Therefore, the two indexing functions define the same lattice $\mathcal{L}(L)$.

The set of the elements accessed by each array reference at processor p is the image of the iteration space (10) under that reference's indexing function. We can calculate that processor p accesses the following elements via reference $a(x_1 + p, 2x_2)$:

$$P_1 \cap \mathcal{L}(L) \quad = \quad \{(y_1, y_2) \mid p + 1 \leq y_1 \leq p + N \wedge 2 \leq y_2 \leq 2N\} \cap \mathcal{L}(L)$$

and the following elements via reference $a(x_1 + x_2 + p, 2x_2 + 2)$:

$$P_2 \cap \mathcal{L}(L) \quad = \quad \{(y_1, y_2) \mid 2p \leq 2y_1 - y_2 \leq 2p + 2N - 2 \wedge 4 \leq y_2 \leq 2N + 2\} \cap \mathcal{L}(L)$$

Therefore, the local copies of a that should be stored at processor p is a quasi-convex polyhedral lattice $(P_1 \cup P_2) \cap \mathcal{L}(L)$, as depicted in Figure 9. If we copy each reference separately, the elements in the overlapped region are transferred twice. To transfer each datum exactly once, we apply our code generation algorithm to scan the union of the two accessed regions $(P_1 \cup P_2) \cap \mathcal{L}(L)$. We will generate the following data transfer code:

```
doall p = 0, P − 1
   do y1 = p + 1, p + N
      do y2 = 2, 2N, 2
         atmp(y1, y2) = fetch(a(y1, y2))
   do y1 = p + N + 1, p + 2N
      do y2 = max(4, 2y1 − 2p − 2N + 2), min(2y1 − 2p, 2N + 2), 2
         atmp(y1, y2) = fetch(a(y1, y2))
```

Since the accessed region are not dense, the step sizes of the two inner loops of y_2 are 2. Here, *atmp* denotes a copy of a in the local processor memory and fetch is the instruction that copies a datum from the global memory to the local memory.

In general, we can use our code generation algorithm to generate data transfer code for all uniform dependence algorithms [15]. These algorithms have the characteristic that all their data dependences can be represented by constant integer vectors. In such a loop nest of the form (4), every array variable subscript has the form $x_k + b_k$, where b_k is a constant. Therefore, the indexing functions of all array references define the lattice $\mathbb{Z}^n$. This implies that the union of these references is always a quasi-convex polyhedron. In the case of algorithms with *uniformly generated references* [9], our code generation algorithm may also be used. For example, $a(2x_1+3x_2, x_2)$ and $a(2x_1+3x_2+1, x_2+1)$ are two uniformly generated references when they appear in the body of a two-dimensional loop nest. The indexing functions of all uniformly generated references have the identical linear part. They will define the same lattice if the constant parts are contained in the lattice generated by the linear parts. In this case, the union of these references can be specified as a quasi-convex polyhedral lattice. This has been illustrated by the current example.

4.2 Construction of SPMD Programs

There are two phases in compiling to machines with a distributed address space. The first phase determines how to decompose the data and computation across the processors. One primary objective is to minimise the communication overhead. The second phase is to generate the SPMD program for each processor such that each processor executes its allocated computations and communicates with other processors whenever needed.

There are two major steps in generating a SPMD program. The first step constructs a set of loops to execute the computations allocated to a processor. The second step inserts the send and receive code as necessary to communicate messages across the processors. If we choose to generate communication code as a set of loops, then the problem is similar to that of generating data transfer code in shared memory machines. The only difference is that the data elements are now transferred between processors instead of between the memory levels. Below we will discuss how to to construct the computation code only.

Suppose we want to compile a simple HPF program for a 4-processor distributed memory machine.

```
         REAL a(40), b(40)
!HPF$    PROCESSOR procs(4)
!HPF$    DISTRIBUTE (BLOCK) ONTO procs :: a, b
         DO x1 = 1, 40
            IF (3 ≤ x1 ≤ 38) THEN
S1:            a(x1 + 2) = b(x1 − 2) * 2
            ENDIF
            IF (x1 ≠ 2) THEN
S2:            b(x1) = a(x1) + 1
            ENDIF
         END DO
```

The DISTRIBUTE directive recommends to the compiler that the arrays a and b should be distributed among the 4 processors with blocks of 10 contiguous elements per processor. So the following elements are mapped to processor p, $0 \le p \le 3$:

$$\begin{aligned}
\mathsf{local_data}(p,a) &= \{x_1 \mid 10p+1 \le x_1 \le 10(p+1)\} \\
\mathsf{local_data}(p,b) &= \{x_1 \mid 10p+1 \le x_1 \le 10(p+1)\}
\end{aligned}$$

When applying the owner computes rule, we can derive the computation decomposition of the program from the data decomposition. The local iteration space for a statement at a

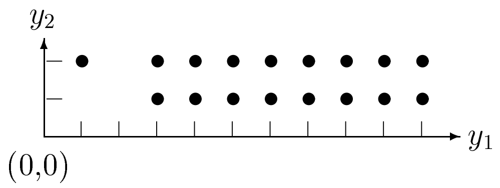


Figure 10: The iterations allocated to processor $p = 0$.

processor is the set of iterations that cause the data elements owned by the LHS of the statement to be accessed. It can be calculated by applying the inverse of the indexing function of the LHS to the set of the data elements owned by the LHS, and then intersecting with the iteration space of the surrounding loops. By also taking into account the guards of the two statements, we obtain the computations of each statement allocated to processor p:

$$\begin{aligned}
\mathsf{local_iter}(p, S_1) &= \{y_1 \mid 10p - 1 \le y_1 \le 10p + 8 \wedge 3 \le y_1 \le 38\} \\
\mathsf{local_iter}(p, S_2) &= \{y_1 \mid 10p + 1 \le y_1 \le 10(p+1) \wedge (y_1 \le 1 \vee y_1 \ge 3)\}
\end{aligned}$$

Since the two local iteration spaces are associated with different statements, we must generate code to scan the *discriminating* rather than the logical union of the two. We can do so by using our code generation algorithm as follows. Firstly, we add one extra dimension, y_2, to each local iteration space to memorise the statement with which the iteration space is associated:

$$\begin{aligned}
\mathsf{new_local_iter}(p, S_1) &= \{(y_1, y_2) \mid 10p - 1 \le y_1 \le 10p + 8 \wedge 3 \le y_1 \le 38 \wedge y_2 = 1\} \\
\mathsf{new_local_iter}(p, S_2) &= \{(y_1, y_2) \mid 10p + 1 \le y_1 \le 10(p+1) \wedge (y_1 \le 1 \vee y_1 \ge 3) \wedge y_2 = 2\}
\end{aligned}$$

Secondly, we generate code to scan the union of the two new sets, which can be a non-convex region as illustrated in Figure 10. Using our code generation algorithm, we generate the code:

$$\begin{array}{l}
\mathsf{do}\ y_1 = \max(3, 10p-1),\ 10p \\
\quad \mathsf{do}\ y_2 = 1, 1 \\
\mathsf{do}\ y_1 = 10p+1,\ \min(1, 10p+8) \\
\quad \mathsf{do}\ y_2 = 2, 2 \\
\mathsf{do}\ y_1 = \max(3, 10p+1),\ 10p+8 \\
\quad \mathsf{do}\ y_2 = 1, 2 \\
\mathsf{do}\ y_1 = 10p+9,\ 10p+10 \\
\quad \mathsf{do}\ y_2 = 2, 2
\end{array}$$

Finally, we obtain the computation code by replacing the inner loops of y_2 with the statements they represent:

$$\begin{array}{ll}
 & \mathsf{do}\ y_1 = \max(3, 10p-1),\ 10p \\
S_1: & \quad a(y_1+2) = b(y_1-2) * c(y_1-2) + c(41-y_1) \\
 & \mathsf{do}\ y_1 = (10p+1),\ \min(1, 10p+8) \\
S_2: & \quad b(y_1) = a(y_1) + 1 \\
 & \mathsf{do}\ y_1 = \min(3, 10p+1),\ 10p+8 \\
S_1: & \quad a(y_1+2) = b(y_1-2) * c(y_1-1) + c(41-y_1) \\
S_2: & \quad b(y_1) = a(y_1) + 1 \\
 & \mathsf{do}\ y_1 = 10p+9,\ 10p+10 \\
S_2: & \quad b(y_1) = a(y_1) + 1
\end{array}$$

Processor 0 does not execute the first loop but is the only one running the second loop.

In general, suppose $S_1, S_2, \cdots$ are the statements in the body of an n-dimensional loop nest. Let the local iteration space of S_i be specified as a union of convex polyhedra $P_{i,1}, P_{i,2}, \cdots$ in $\mathbb{Z}^n$, possibly intersected with a lattice $\mathcal{L}(L)$, where

$$P_{i,j} \;=\; \{y \mid A_{i,j}y \leq b_{i,j}\}$$

We generate computation code as follows:

1. Add a dimension y_{n+1} to represent the statements by modifying $P_{i,j}$ to:

$$P'_{i,j} \;=\; \{(y, y_{n+1}) \mid A_{i,j}y \leq b_{i,j} \wedge y_{n+1} = i\}$$

2. Apply our code generation algorithm to generate code to scan $(\bigcup_{i,j} P'_{i,j}) \cap \mathcal{L}(L)$.

3. Replace the innermost loop y_{n+1} with the statements they represent.

Most of the existing compilers for distributed memory machines have a more or less similar approach in how they generate code from user-specified data decompositions in [10, 11, 12]. The Fortran D compiler appears to have been substantially experimented. In this compiler, a local iteration space is specified using a *regular section descriptor* (RSD), a rectangular section with constant steps. For a loop nest that has multiple statements, it appears that separate loop nests are constructed to scan the local iteration spaces of different statements and then these loop nests are combined to form the computation code using a technique called the *statement groups*. In this approach, expensive execution guards might have to be introduced for the statements that are not executed for the entire iteration space. In a more recent work [4], RSDs are generalised to convex polyhedra but how to generate the computation code in the presence of multiple statements was not discussed in detail. An example was used to illustrate a technique called the *loop splitting* to merge the computation loops and the communication loops into a single SPMD program. In this paper, we go one step further by allowing a local iteration space to be a quasi-convex polyhedral lattice. In the general case where several statements are contained inside a loop nest, we provide a systematic approach to generating the computation code that is free of conditionals.

5 Concluding Remarks

In the last decade, a loop transformation theory, based on matrix transformations, has been developed to restructure perfect loop nests to expose the available parallelism or otherwise optimise memory hierarchy performance. In this framework, loop transformations such as loop interchange, skewing and reversal are all represented by matrix transformations. Loop transformations are limited to loop nests with convex iteration spaces. To satisfy this restriction, a lower (upper) bound has been restricted to be a maximum (minimum) of affine expressions. A unimodular transformation always maps a convex polyhedron to a convex polyhedron. Therefore, the transformed iteration space is always convex. This has motivated the developments of earlier code generation methods for convex polyhedra [3, 7]. If a loop transformation is non-unimodular, the transformed iteration space is a convex polyhedron intersected with a lattice. Methods for scanning such a region also exist [13, 18].

In this paper, we presented a method for constructing a loop nest to scan lexicographically a quasi-convex polyhedral lattice, which is shown to be equivalent to the intersection of a lattice and a quasi-convex polyhedron (i.e., a union of convex polyhedra). This generalises the previous work on scanning a convex polyhedron, possibly intersected with a lattice. We developed our method in the context of parallelising loop nests with non-convex iteration spaces. In general, we can handle perfectly nested loops whose loop bounds are expressions built using affine expressions over the surrounding loop variables and max, min, ceiling and floor operators. The iteration space of such a loop nest is modeled as a quasi-convex polyhedron. A non-singular transformation maps a quasi-convex polyhedron to a quasi-convex polyhedral lattice. If the transformation is unimodular, the transformed iteration space is a (dense) quasi-convex polyhedron.

To scan a quasi-convex polyhedron, we proposed a useful concept of maximal normal form, showing that a quasi-convex polyhedron can always be specified by a conjunction of max-type constraints. This special normal form allowed us to identify the non-convex surfaces of a quasi-convex polyhedron. Based on this normal form, we developed a code generation algorithm for constructing a loop nest to scan a quasi-convex polyhedron. To scan a quasi-convex polyhedral lattice – the intersection of a quasi-convex polyhedron and a lattice, we used our previous work [18] to skip the points that are in the quasi-convex polyhedron but not in the lattice.

By putting a quasi-convex polyhedron in the minimal normal form, we showed that a quasi-convex polyhedron is nothing but a union of convex polyhedra. To demonstrate that quasi-convex polyhedral lattices are powerful enough to model many non-convex iteration spaces in practical applications, we considered two code generation problems in compiling for parallel machines. One is the generation of data transfer code between the memory levels in shared memory machines, and the other is the generation of the SPMD program for distributed memory machines. To generate data transfer code, we considered the case where there are multiple references to the same data array. We showed how to combine these multiple references into a quasi-convex polyhedral lattice and then produced the data transfer code so that the same datum is transferred exactly once. To construct a SPMD program, we explained how to generate the computation code to iterate over the computations allocated to a processor. For a loop nest with multiple statements, we gave a procedure for constructing the computation code that is free of conditional tests. The basic requirement is that the set of computations executed by a processor must form a quasi-convex polyhedral lattice.

Acknowledgements

I wish to thank the anonymous referees for their constructive and helpful suggestions and comments. This work is supported by an Australian Research Council Grant A49600987.

References

[1] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Trans. on Programming Languages and Systems*, 9(4):491–542, Oct. 1987.

[2] C. Ancourt. Code generation for data movements in hierarchical memory machines. In *Int. Workshop on Compilers for Parallel Computers*, pages 91–102, Dec. 1990.

[3] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *PROc. of the 3rd ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 39–50. ACM Press, Apr. 1991.

[4] S. P. Armarasinghe and M. S. Lam. Communication optimisation and code generation for distributed memory machines. In *Proc. of the ACM SIGPLAN'93 Conf. on Programming Language Design and Implementation*, Jun. 1993.

[5] Z. Chamski. Nested loop sequences: Towards efficient loop structures in automatic parallelization. In *Proc. of the 27th Int. Conf. on System Sciences*, volume Part II: Software Technology, pages 14–22, 1994.

[6] M .C. Cheng. General criteria for redundant and nonredundant linear inequalities. *J. of Optimization Theory and Applications*, 53(1):37–42, Apr. 1987.

[7] P. Feautrier. Parametric integer programming. *Operations Research*, 22:243–268, 1988.

[8] K. Gallivan, W. Jalby, and D. Gannon. On the problem of optimizing data transfers for complex memory systems. In *Supercomputing '88*, pages 238–253. ACM Press, 1988.

[9] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *J. Parallel and Distributed Computing*, 5:587–616, 1988.

[10] S. Hiranandani, K. Kennedy, and C. W.-Tseng. Compiling Fortran D for MIMD distributed memory machines. *Communications of the ACM*, 35(8):66–80, Aug. 1992.

[11] S. Hiranandani, K. Kennedy, and C. W.-Tseng. Preliminary experiences with the Fortran D compiler. In *Supercomputing '93*, pages 338–350. IEEE Computer Society Press, Nov. 1993.

[12] C. Koelbel. Compile-time generation of regular communication patterns. In *Supercomputing '91*, pages 101–110. ACM Press, 1991.

[13] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matric es. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science 757, pages 391–405. Springer-Verlag, 1992.

[14] P. Quinton and V. van Dongen. The mapping of linear recurrence equations on regular arrays. *J. VLSI Signal Processing*, 1(2):95–113, Oct. 1989.

[15] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, Oct. 1991.

[16] M. J. Wolfe. More iteration space tiling. In *Supercomputing '88*, pages 655–664, Nov. 1989.

[17] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[18] J. Xue. Automating non-unimodular loop transformations for massive parallelism. *Parallel Computing*, 20(5):711–728, 1994.

[19] J. Xue. Closed-form mapping conditions for the synthesis of linear processor arrays. *J. VLSI Signal Processing*, 10(2):181–199, 1995.