To be presented at the 9th International Parallel Processing Symposium, IEEE Computer Society Press, Santa Barbara, April 25 -- 28, 1995

Constructing DO Loops for Non-Convex Iteration Spaces in Compiling for Parallel Machines

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

Abstract

A need that frequently arises in code generation for shared and distributed memory machines is to scan a region of index points intersected by an integer lattice in their lexicographical order. Methods have been proposed in the special case when the given region is a convex polyhedron. This paper contributes a method in the more general case when the given region is a union of convex polyhedra - a non-convex polyhedron in general. As a secondary contribution, we show the usefulness of this method by presenting its applications in several areas, such as non-singular transformations of nested loops with non-convex iteration spaces, generation of data transfer code for shared memory machines, and generation of SPMD DO loops with communication code for distributed memory machines, where existing methods do not apply. Thus, these various code generation problems can all be unified as one of constructing DO loops to scan a union of convex polyhedra, possibly intersected by a lattice.

1 Introduction

This work is motivated primarily by our concern of providing a unified solution to several closely related code generation problems that arise in programming shared and distributed memory machines.

Problem I. Rewriting a set of nested loops with a non-convex iteration space given a non-singular loop transformation. This issue arises in optimising and parallelising compilers that use loop transformations to reorder the iterations of nested loops in order to expose parallelism and improve use of the memory hierarchy. If the iteration space is a convex polyhedron, several existing methods can be used [2, 9, 10, 11, 12]. However, neither of these methods works if the iteration space is non-convex, which may occur when some loop bounds contain both max and min operators.

Problem II. Generating the code needed for moving data between the memory levels for shared memory machines. 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 [5, 6] was restricted to a single array reference in a loop nest with a rectangular iteration space. Later in [1], 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 accessed at several points in the loop body, and in addition, these accessed regions may overlap. The problem concerning us here is that, given multiple references to an array variable with an affine indexing function, generate the code needed to move exactly once each element in the union of the regions accessed by these multiple references - a non-convex polyhedron in general - between the memory levels.

Problem III. Generating SPMD programs for distributed memory machines. The two major steps in compiling for distributed memory machines are decomposing the data and computation across processors, then deriving SPMD (Single-Program Multiple Data) programs as a set of DO loops with explicit data movement. There are two subproblems:

A. Generating a set of DO loops to execute the iterations allocated to a processor.
B. Generating a set of DO loops (i.e., communication code) to communicate messages across the processors.

Existing methods [3, 7, 8] use rectangular sections or convex polyhedra to represent local iterations and communication messages and generate code using the known methods for convex polyhedra [2, 4, 12] or their variants.

In practice, the set of iterations allocated to a processor may be non-convex in the presence of multiple statements. The use of rectangular sections or convex polyhedra may introduce expensive conditional tests in the target code [8]. Similarly, when there are multiple references to an array variable, the data elements of that variable to be communicated may not form a convex polyhedron. In [3], the convex hull of these multiple references is communicated, causing sometimes a large amount of redundant data movement.

The problem concerning us is the following. Given an arbitrary set of iterations or messages, which needs not necessarily be a convex polyhedron, construct DO loops to scan the given set in such a way that each element in the set is enumerated exactly once.

In this paper, we show that Problem I and many instances of Problems II and III can be reduced to the following problem.

Scanning Problem. Given a union of integral convex polyhedra, \( P_1, P_2, \ldots \in \mathbb{Z}^m \) and a non-singular integer matrix \( T \in \mathbb{Z}^{m \times n} \), construct DO loops to scan the points in \((P_1 \cup P_2 \cup \ldots) \cap \mathbb{Z}(T)\) in their lexi-
graphical order, where \( \mathcal{L}(T) \) is the lattice defined by the columns of \( T \), i.e., the set \( \{ y | y = Tx \land x \in \mathbb{Z}^n \} \).

We first describe a method for solving the scanning problem. We then show how it can be used to solve several code generation problems. We finally conclude the paper with some future research work.

2 Solving the Scanning Problem

A union of convex polyhedra is called a *quasi-convex polyhedron*. The intersection of a quasi-convex polyhedron and a lattice is called a *quasi-convex polyhedral lattice*.

This section gives an algorithm for constructing DO loops to scan a quasi-convex polyhedral lattice. For details about the development of the algorithm, we refer to our technical report [12].

The central to our algorithm is the notion of *maximal normal form* that identifies the non-convex surfaces of a quasi-convex polyhedron.

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

\[
\begin{align*}
\max(a_{11}, a_{12}, \ldots) & \geq 0 \\
\max(a_{21}, a_{22}, \ldots) & \geq 0 \\
& \vdots \\
\end{align*}
\]

where \( a_{ij} \) are affine expressions of the constraint variables in the system.

As an example, the quasi-convex polyhedron depicted in Fig. 1 is specified by the following system:

\[
\begin{align*}
\max(y_1 - y_2, y_1 + y_2 - 6) & \geq 0 \\
\max(-y_1 - 3y_2 + 13, y_1 - y_2 - 1) & \geq 0 \\
-y_1 + 5 & \geq 0 \\
y_2 - 1 & \geq 0
\end{align*}
\]

The first and second max-type constraints create the non-convex surfaces on the left and top boundary of this quasi-convex polyhedron, respectively.

Unlike convex polyhedra, it is non-longer possible to scan every quasi-convex polyhedron using a perfect loop nest. (Note that we disallow conditional tests inside the loop nest.) But it is possible to do so for a subclass of quasi-convex polyhedra, which are referred to here as *triangular polyhedra*. The term "triangular" is chosen to generalise the notion of the triangular set of inequalities [2].

A constraint \( \max(a_1, a_2, \ldots) \geq 0 \) is a lower (upper) bound of \( y_k \) if it can be rewritten as \( y_k \geq \min(a_1, a_2, \ldots) \) (\( y_k \leq \max(a_1, a_2, \ldots) \)) such that \( a_1, a_2, \ldots \) are affine expressions of \( y_1, \ldots, y_{k-1} \). For example, \( \max(y_1 + y_2 - 1, y_2 - 2) \geq 0 \) is a lower bound of \( y_2 \) since it is equivalent to \( y_2 \geq \min(-y_1 - 1, 2) \). But \( \max(-y_2 + 1, y_1 + 2y_2) \geq 0 \) is not a loop bound.

A quasi-convex polyhedron is *triangular* if all its defining constraints are loop bounds.

Lemma 2 Let \( P \subset \mathbb{R}^n \) be a triangular polyhedron and \( T \in \mathbb{Z}^{\times n} \) be a non-singular matrix. A set of perfect loop nest that scans \( P \cap \mathcal{L}(T) \) is constructed as follows, where the \( k \)-th loop has the form:

\[
do \ y_k = L_k + \delta_k, \ U_k, \ step_k
\]

- The calculation of \( \delta_k \) and \( step_k \) follows [12].
- If there are constraint(s) in \( P \) as the lower (upper) bound(s) of loop variable \( y_k \), we let \( L_k (U_k) \) be a maximum (minimum) of all such lower (upper) bound(s). Otherwise, there are many ways to construct \( L_k (U_k) \). We can let \( L_k (U_k) \) be the lower (upper) bound constructed from the convex hull of \( P \) using either the Fourier-Motzkin elimination algorithm or Faugère's F4 as described in \([2, 4, 12]\).

If a quasi-convex polyhedron is not triangular, we partition it into a number of triangular polyhedra, and then apply Lemma 2 to construct DO loops for every such triangular polyhedron thus obtained. Finally, we combine these DO loops to form the DO loops to scan the given quasi-convex polyhedron.

A partitioning tree is used to record all information relevant to the partitioning. A partitioning tree for a quasi-convex polyhedron \( P \subset \mathbb{R}^n \) is defined below.

1. The tree is completely balanced in the sense that all leaf nodes are at level \( n \) (the root at level 0).
2. The root of the tree is \( P \).
3. For every node at level \( 0 \leq k < n \), its children \( P_1, P_2, \ldots \) from left to right form a partition of the father node such that \( P_k \subseteq \bigcup_{k+1} \cdots \). The notation \( \subseteq \) is defined as follows. Let \( P \) and \( Q \) be two quasi-convex polyhedra in \( \mathbb{R}^n \). We write \( P \subseteq Q \) if, for every pair of points \((y_1, \ldots, y_{k-1}, y_k, \ldots) \in P \) and \((y_1, \ldots, y_{k-1}, y_k, \ldots) \in Q \), we have \( y_k \leq y_k \).
4. All leaf nodes are triangular polyhedra (which form a partition of the root \( P \) by (3)).

Our algorithm is sketched below.

Algorithm 1 (A Solution to the Scanning Problem)

**Input**: Convex polyhedron \( P_1, P_2, \ldots \) in \( \mathbb{R}^n \) and a non-singular matrix \( T \in \mathbb{Z}^{\times n} \).

**Output**: DO loops to scan \( (P_1 \cup P_2 \cup \cdots) \cap \mathcal{L}(T) \).

1. Apply Lemma 1 to put \( P_1 \cup P_2 \cup \cdots \) into a maximal normal form.
2. Build a partitioning tree for \( P_1 \cup P_2 \cup \cdots \) [13].
3. Generate DO loops to scan \( (P_1 \cup P_2 \cup \cdots) \cap \mathcal{L}(T) \) lexicographically.
   (a) Apply Lemma 2 to construct DO loops for each of the triangular polyhedra associated with the leaves of the tree.
   (b) The root has no associated code.
   (c) The code associated with every leaf node is:
   \[
do \ y_k = L_k + \delta_k, \ U_k, \ step_k
\]
   Enumerate \((y_1, \ldots, y_k)\)
Figure 2: A partitioning tree for (1).

where $L_n$ and $U_n$ are derived in Step 3(a) and $\text{step}_k$ and $\delta_k$ are calculated as per [12].

(d) The code associated with every non-leaf node at level $1 \leq k < n$ is:

$$
do y_k = L_k + \delta_k, \quad U_k, \text{step}_k
$$

where $L_k$ ($U_k$) is a minimum (maximum) of the lower (upper) bounds of all its descendants, and the calculation of step_k and $\delta_k$ follows [12].

(e) The target DO loops is obtained by a classic preorder traversal of the partitioning tree.

**Theorem 1** Let $P \subset \mathbb{Z}^n$ be a quasi-convex polyhedron and $T \in \mathbb{Z}^{n \times n}$ be a non-singular matrix. The DO loops generated by Algorithm 1 scan the points $P \cap \mathcal{L}(T)$ in their lexicographical order.

For the above example, Fig. 2 shows one partitioning tree, where $P_1$ and $P_2$ are two quasi-convex polyhedra separated along the dashed line in the figure:

$$
\begin{align*}
&y_1 - y_2 \geq 0 \\
&\max(-y_1 - 3y_2 + 13, y_1 - y_2 - 1) \geq 0 \\
&y_1 - 5 \geq 0 \\
&y_2 - 1 \geq 0 \\
&y_1 - y_2 < 0 \\
&2y_1 + y_2 - 6 \geq 0 \\
&y_1 - 3y_2 + 13 \geq 0
\end{align*}
$$

The target DO loops returned by Algorithm 1 are:

$$
\begin{align*}
do y_1 &= 1, 5 \\
do y_2 &= 1, \min(y_1, \max([(13 - y_1)/3], y_1 - 1)) \\
\text{Enumerate} (y_1, y_2)
\end{align*}
$$

and is shown in Fig. 3(a). The new iteration space is specified by

$$
\begin{align*}
&\max(y_1 - 3, y_1 + y_2 - 5) \geq 0 \\
&\max(-y_1 - 3y_2 + 10, -2y_1 + y_2 + 6) \geq 0
\end{align*}
$$

and is depicted in Fig. 3(b).

An application of Algorithm 1 yields the following DO loops to scan this new iteration space:

$$
\begin{align*}
do y_1 &= 1, 2 \\
do y_2 &= 5 - y_1, 6 \\
&\max(y_2 - y_1 - 1) \geq 0 \\
&\max(y_1 + y_2 - 9, 3, y_2 - 6) \geq 0 \\
&\max(y_1 - y_1 - 1) \geq 0
\end{align*}
$$

In general, we can handle perfect nested loops whose loop bounds are affine expressions of the surrounding loop indices and some structural parameters and contain max, min, floor and ceiling operators.

### 3.2 Problem II: Data Transfer Code

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 the processors:

$$
doall p = 0, P - 1 \\
do x_1 = 1, N \\
do x_2 = 1, N \\
\cdots = a(x_1 + p, 2x_2) + a(x_1 + x_2 + p, 2x_2 + 2)
$$

where $p$ is the “processor index”. That is, every processor executes the two inner loops whose iteration space is $\{(x_1, x_2) | 1 \leq x_1, x_2 \leq N\}$.

Assuming that array $a$ is initially stored in global memory, we want to generate the extra code needed to make part of array $a$ accessed by a processor into its local memory.

We write the indexing functions of references $a(x_1 + p, 2x_2)$ and $a(x_1 + x_2 + p, 2x_2 + 2)$ as:

$$
\begin{align*}
A_1x + b_1 &= \begin{bmatrix} 1 & 0 & x_1 \\ 0 & 2 & x_2 \end{bmatrix} + \begin{bmatrix} p \\ 0 \end{bmatrix} \\
A_2x + b_2 &= \begin{bmatrix} 1 & 1 & x_1 \\ 0 & 2 & x_2 \end{bmatrix} + \begin{bmatrix} p \\ 2 \end{bmatrix}
\end{align*}
$$

(a) The original iteration space

(b) The new one

Figure 3: A loop interchange.
It can be verified that \( \mathcal{L}(A_1) = \mathcal{L}(A_2) = \mathcal{L}(L) \), where

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

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

The set of elements accessed by an array reference at processor \( p \), \( 0 \leq p < P \), is the image of the iteration space \( \{ (x_1, x_2) \mid 1 \leq x_1, x_2 \leq N \} \) under the reference’s indexing function. The set of elements accessed by reference \( a(x_1 + p, 2x_2 + 2) \) at processor \( p \) is:

\[
\{ y \mid y = A_2 x + b_2, 1 \leq x \leq (N, N) \} = \{ (y_1, y_2) \mid 2p \leq y_1 - y_2 \leq 2p + 2N - 2, 4 \leq y_2 \leq 2N \} \cap \mathcal{L}(L) = P_2 \cap \mathcal{L}(L)
\]

Since \( A_1, A_2 \), and \( 1 \) are non-singular square matrices, we have eliminated \( y = A_2 x + b_2 \) by solving for \( x \) and substituting \( x = A_1^{-1} (y - b_1) \) into \( (1, 1) \leq x \leq (N, N) \).

Using Algorithm 1, we obtain the following data transfer code that ensures that every datum in \( (P_1 \cup P_2) \cap \mathcal{L}(L) \) is transferred exactly once (Fig. 4).

```
doall p = 0, P - 1
  do x1 = p + 1, p + N
    do x2 = 2, 2N, 2
      atmp(x1, x2) = fetch(a(x1, x2))
      do x2 = 1 + p + N + 2, 2N
        atmp(x1, x2) = fetch(a(x1, x2))
      do x2 = max(2, 2x1 - 2p - 2N + 2), min(2x1 - 2p, 2N + 2), 2
        atmp(x1, x2) = fetch(a(x1, x2))
    do x2 = max(2, 2x1 - 2p - 2N + 2), min(2x1 - 2p, 2N + 2), 2
      atmp(x1, x2) = fetch(a(x1, x2))
  do x1 = p + 1, p + N
```

Note that the step sizes of the two inner loops 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 a local memory.

If one uses the original loop structure to copy the two regions \( P_1 \cap \mathcal{L}(L) \) and \( P_2 \cap \mathcal{L}(L) \) separately, the elements in \( P_1 \cap P_2 \cap \mathcal{L}(L) \) will be transferred twice.

### 3.3 Problem III: SPMD Programs

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

```
real a(40), b(40)
 decomposition T(40)
 align a, b with T
 distribute T(block)
 do i = 1, 40
   S1 if (3 \leq i \leq 38) then a(i + 2) = b(i - 2) + 2
   S2 if (i != 2) then b(i) = a(i) + 1
```

We can calculate that the following set of array elements of \( a \) are stored locally in processor \( p, 0 \leq p < 3 \):

\[
\text{local_data}(p, a) = \{ i \mid 10p + 1 \leq i \leq 10(p + 1) \}
\]

When applying the owner computes rule, the local iteration set for a statement is the inverse of the indexing function of the right-hand side:

\[
\text{local_iter}(p, s_1) = \{ i \mid 10p - 1 \leq i \leq 10p + 8 \}
\]

Notice that loop distribution is illegal due to a dependence cycle between the two statements.

Since the two local iteration sets are associated with distinct statements, we must generate DO loops to scan the discriminating rather than the logical union of the two sets. Algorithm 1 cannot be used directly.

To apply our method, we firstly add one more dimension, say, \( s \), to local iteration sets to represent the subscripts of their corresponding statements:

\[
\text{local_iter}(p, s_1) = \{ (i, s) \mid i \in \text{local_iter}(p, s_1), s = 1 \}
\]

We then apply Algorithm 1 to generate DO loops to scan the union of the two new local iteration sets – a non-convex polyhedron shown in Fig. 5:

```
  do i = max(3, 10p - 1), 10p
    do s = 1, 1
      do i = 10p + 1, min(1, 10p + 8)
        do s = 2, 2
          do i = max(3, 10p + 1), 10p + 8
            do s = 1, 2
              do i = 10p + 9, 10p + 10
                do s = 2, 2
```

Finally, we obtain the SPMD DO loops for processor \( p, 0 \leq p < 3 \), by replacing the iterations of the inner loops with their corresponding statements:

```
  do i = max(3, 10p - 1), 10p
    a(i + 2) = b(i - 2) + 2
  do i = (10p + 1), min(1, 10p + 8)
    b(i) = a(i) + 1
  do i = min(3, 10p + 1), 10p + 8
    a(i + 2) = b(i - 2) + 2
  b(i) = a(i) + 1
```
Processor $0$ does not execute the first loop but is the only one running the second loop.

Note that the approach outlined here can be generalised to any loop nest with multiple statements. The SPMD DO loops as reported in [8] may contain expensive execution guards for statements that are not executed for the entire iteration space. Our method dispenses with execution guards by means of using imperfectly nested loops to specify the target code. As a result, the operation known as “statement groups” in [8] is not needed because the same loop bounds are derived for those statements with the identical local iteration set, i.e., in the same statement group.

The problem of generating communication code is similar to that of generating data transfer code in shared memory machines. The only difference is that the data is now transferred between processors instead of between the memory levels.

In the presence of multiple references to an array variable, we are able to generate code to communicate every element in the union of these references exactly once, as long as the union can be specified as a quasi-convex polyhedral lattice.

4 Conclusion

The primary contribution of this paper is a method that takes as input a quasi-convex polyhedral lattice and produces as output a set of DO loops that scans the points in the given region in their lexicographical order. As a secondary contribution, we demonstrated the usefulness of this method by showing its applications in code generation for programming shared and distributed memory machines. Example applications we considered include non-singular loop transformations of nested loops, generation of data transfer code between the memory levels in shared memory machines, and generation of SPMD programs with communication code in distributed memory machines. For most practical cases, these various code generation problems were identified as one of constructing DO loops to scan a quasi-convex polyhedral lattice.

Existing methods on loop transformations [2, 9, 10, 11, 12] cannot handle nested loops with non-convex iteration spaces. As a result, they are not applicable to generate SPMD programs when the given region is non-convex. It is possible, however, to apply these methods to generate data transfer code in shared memory machines or communication code in distributed memory machines. The problem arises in the case of multiple references to the same array. Then, code for each distinct reference must be generated separately, and consequently, the same data may be transferred multiple times unnecessarily. The code thus produced is very inefficient if there is a great deal of overlap between the regions accessed by these multiple references. On the contrary, the code produced by our method transfers every data element exactly once, with no redundant transfers introduced.

One future investigation is to consider code generation for structures more general than quasi-convex polyhedral lattices, which can occur, for example, in the case where several indexing functions of the same array variable define distinct lattices.

References


