On the Parallelization of Loop Nests Containing while Loops

Christian Lengauer and Martin Griebl

Fakultät für Mathematik und Informatik
Universität Passau
D–94030 Passau, Germany
email: {lengauer,griebl}@fmi.uni-passau.de

Abstract

Recently, efforts have been made to devise automatic methods, based on a mathematical model, for the parallelization of loop nests with while *loops. These methods are extensions of methods for the parallelization of nested* for *loops.*

1 Introduction

The parallelization of nested loops has been a topic of research for at least twenty years [13]. Until recently, efforts have focussed on for loops.

A for loop is distinguished by the fact that its bounds are determined before the loop is executed. Modeling a set of nested for loops by a convex polytope in which each dimension enumerates the iterations of one loop has proved useful in characterizing and analyzing the space of possible parallelizations. With methods based on the polytope model, one can automatically make an optimal choice from the set of all parallelizations and achieve orders of magnitude of speed-up [2, 15].

The parallelization of while loops has been investigated for a number of years. The general approach has been to pipeline the successive iterations where possible (e.g., [17, 19]). This does not require methods based on the polytope model, and it yields at most constant speed-up.

Because of the dependence of the loop's termination on the state produced by the current iteration, a single while loop cannot be parallelized in the polytope model. However, the polytope model can be used to parallelize a while loop with respect to other loops in a loop nest. Recently, efforts have been made at the Ecole Normale Supérieure de Lyon and at the Universität Passau to generalize the polytope model to loop nests that contain also while loops. The purpose of this paper is to illustrate the feasibility of such an extension and to discuss the problems that occur in it.

A paper published elsewhere demonstrates the polytope methods for for loop parallelization on the example of the polynomial product [15]. The present paper can be viewed as a "sequel"; it also has the polynomial product as example.

A while loop is commonly written:

while *condition* do *body*.

We view the while loop as a for loop with dynamic upper bound, that is, we add an artificial loop index that counts the number of executed iterations. Our while loop notation is:

for *index* := 0 while *condition*(*index*) do *body*

where *index* is a new variable that does not occur in the original loop. As in normal for loops, the increment of the loop index is implicit and follows the execution of the loop body. By supplying *index* as a parameter to *condition*, we make the dependence of *condition* on the while iteration explicit. Every while loop can be represented this way.

2 for loop nests in the polytope model

The parallelization of nested for loops in the polytope model proceeds as follows.

First, one transforms d perfectly nested source loops into a d-dimensional polytope. We call this polytope the *index space*. Each loop defines the extent of the index space in one dimension. Each point of the index space represents one step of the loop nest. The coordinates of the point are given by the values of the loop indices at that step.

Then, one applies an affine coordinate transformation, the *space-time mapping*, to the polytope and obtains another polytope in which one dimension enumerates time and the others enumerate space. We call the transformed polytope the *target space*.

Finally, one translates this polytope back to a nest of target loops; each space dimension becomes a parallel loop and the time dimension becomes a sequential loop. The target loop nest specifies synchronous parallelism if the outer loop is the sequential one, and asynchronous parallelism if the inner loop is the sequential one.

We would like to draw attention to one possible complication in for loop parallelization. When the space-time mapping is not unimodular (i.e., when its inverse is not an integer matrix) the target space has "holes", i.e., it is not convex even though the index space is [3]. In this case, one has to take care that the target loops do not enumerate the holes. We call the process of enumerating the points of the target space *scanning*. Luckily, non-unimodular mappings distribute holes evenly throughout the target space. Therefore, it is always possible to construct a target loop nest that scans the target space precisely—whether the space-time mapping is unimodular [4] or not [20].

3 Polynomial product with a for loop nest

The material in this section is taken from [15]. We are cursory here and refer the reader for more explanations to that paper. We use Dijkstra's quantifier format and guarded command notation [8], but we give the alternative construct the meaning of the empty statement when no guard is true.

3.1 Specification

Let $\mathcal{A}$ and $\mathcal{B}$ be two polynomials in variable z and of degree n, i.e.:

$$\begin{aligned} \mathcal{A} &= (\textstyle\sum\ k : 0 \leq k \leq n : a[k]\, z^k) \\ \mathcal{B} &= (\textstyle\sum\ k : 0 \leq k \leq n : b[k]\, z^k) \end{aligned}$$

Desired is the polynomial $\mathcal{C}$ of degree $2\,n$ defined by:

$$\mathcal{C} \;=\; \mathcal{A}\mathcal{B} \;=\; (\textstyle\sum\ k : 0 \leq k \leq 2\,n : c[k]\, z^k)$$

where:

$$\begin{aligned} &(\forall\, k : 0 \leq k \leq 2\,n : c[k] \;= \\ &\quad (\textstyle\sum\ i,j : 0 \leq i \leq n \wedge 0 \leq j \leq n \wedge i+j = k : a[i]\, b[j])) \end{aligned}$$

3.2 Source program

The following imperative program computes the coefficients $c[k]$ from $a[i]$ and $b[j]$; we define the body of the loop nest separately:

```
for i := 0 to n do
   for j := n downto 0 do
      (i,j)

(i,j) : if i=0 ∨ j=n → c[i+j] := a[i] b[j]
        [] i≠0 ∧ j≠n → c[i+j] := c[i+j] + a[i] b[j]
        fi
```

3.3 Index space

Figure 1 shows the index space that represents our polynomial product program. The sequential execution enumerates the points in this space bottom to top and then left to right.

3.4 Target space

There are many ways of parallelizing a loop nest; each is represented by a space-time mapping—a coordinate transformation of the index space. Informally, the idea is to slice the index space into time by partitioning it into parallel hyperplanes—lines in our two-dimensional case. Each line represents iterations that are executed simultaneously. In order not to violate any data dependence, the slicing of the index space must follow the direction of all dependence arrows. When one has chosen a time slicing, one partitions the index space a second time, this time into parallel lines (for any dimensionality). Each line represents the sequential computation of one processor, so the lines must cut through the hyperplanes. The mathematical formulation of this process can be found in [15], with additional detail in [1, 2]. Let us call a space-time mapping that does not violate the dependences of the index space *valid*.

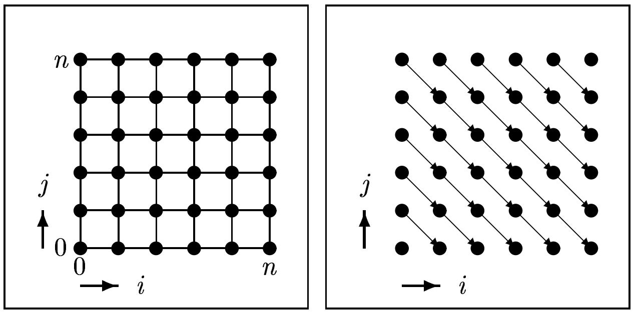

Figure 1: Index space on a coordinate grid (left) and dependence graph (right)

We repeat here the choice of space-time mapping that we make in [15]; it is depicted in Figure 2. This choice is not optimal, but it is more interesting for our purposes.

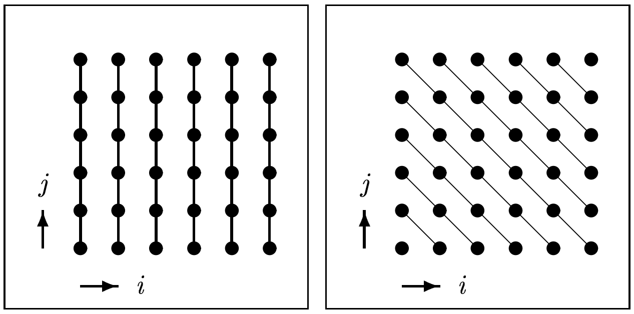

Figure 2: Temporal partitioning (left) and spatial partitioning (right)

In our example, after the mapping, one coordinate enumerates time (we name it t) and one enumerates space (we name it p). The number of time lines tells us that we need n execution steps, the number of space lines that we need $2\,n{+}1$ processors. Again, the details can be found in [15]. Figure 3 shows the target space.

3.5 Target programs

The last step in the parallelization is to transform the target space back into a loop nest.

Before we can proceed with the transformation, we must choose the order in which the target coordinates shall be nested as target loops. If we make the t-dimension the outer loop, we obtain synchronous parallelism. If we make it the inner loop, we obtain asynchronous parallelism. How either target loop nest can

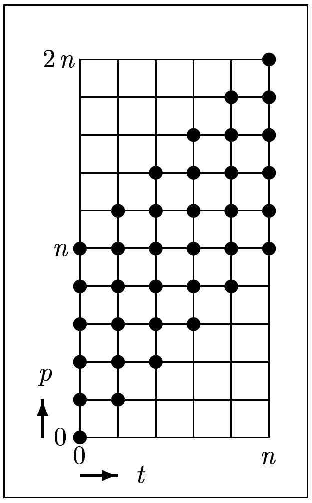

Figure 3: Target space on a coordinate grid

be computed from the target space is explained in [15]. We only state both programs here. We use the keywords seqfor and parfor for the temporal and spatial loop, respectively.

3.5.1 Synchronous target program

The synchronous program can be directly read off from the bounds of the target space:

```
seqfor t := 0 to n do
    parfor p := t to t+n do
        (t, p−t)
```

Note that the processor layout must be determined at every time step—and so must the termination of all processors (this is called *barrier synchronization* [18]). Typically, synchronous parallelism is combined with the shared memory model; we make this assumption here.

3.5.2 Asynchronous target program

Enumerating coordinate p with the outer loop causes a problem, because the bounds on p depend on coordinate t (the respective borders of the index space are not orthogonal to the p-axis). This dependence can be eliminated via a polyhedron projection algorithm: one adds redundant borders that are independent of t (in this case, the horizontal borders represented by the conditions $p \geq 0$ and $p \leq 2\,n$). The details are again in [15]. This leads to the program:

```
parfor p := 0 to 2n do
    seqfor t := max(0, p−n) to min(n, p) do
        (t, p−t)
```

Here, the processor layout is determined only once. Typically, asynchronous parallelism is combined with the distributed memory model. In this model, in general, additional message passing directives are necessary to enforce the data dependences.

Let us assume, for brevity, that variables a and b are simultaneously available at the processors that require them. (These variables are not recomputed.) The dependences on variable c, depicted in Figure 1 (right), turn out to be local to the processors defined by the space-time mapping shown in Figure 2, i.e., horizontal in the target space depicted in Figure 3. Thus, in this particular case, no message passing is necessary.

4 while loops in the polyhedron model

A loop nest with while loops is modeled by a polyhedron, not by a polytope. Since, at compile time, the upper bound of a while loop is unknown, the polyhedron is infinite in the corresponding dimension.

Also, a point of the polyhedron has a slightly different interpretation than in the for loop model: it represents not only one iteration step but also the boundary test preceding that step. One consequence is that an empty loop is modeled by one point, not by the empty set. Thus, any point of the index space that represents the first action of, say, loop w at level d stands for boundary tests for all outer loops up to and including w (i.e., d tests) and for the computation at the first iteration of loop w, if any.

A nest with while loops executes only a subset of the infinite index space. We call this subset the *execution space*. We call its transformation under the space-time mapping the *target execution space*. The dependences of points in the execution space make it look like a (possibly, multi-dimensional) comb, as demonstrated in Figure 4 (due to the changed interpretation of a point, we introduce circles to represent successful termination tests). Therefore, we call it also the *execution comb*. The sequence of points along a line of arrows in the execution comb, we call it a *tooth*, corresponds to the execution of one entire while loop. The arrows along a tooth represent the dependences of the while loop iterations on the loop's index; we call them while *dependences*. In contrast to a while loop, a for loop does not have these dependences, although it has an index.

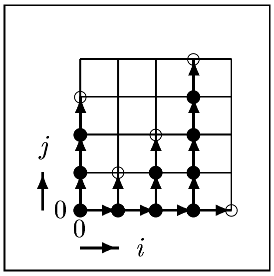

Figure 4: Execution space for two nested while loops, on a coordinate grid, with while dependences

5 Problems with while loops

In this section, we discuss some problems that occur when introducing while loops into loop nests.

5.1 Irregularities of the execution space

Signal processing operations like filtering are formulated as a for loop nest whose outer loop is infinite, i.e., has no upper bound. The model of such a loop nest is a polyhedron that is infinite in the dimension of the outer loop. This type of problem can be handled with the normal for loop parallelization methods. For an example, see the treatment of the linear phase filter in ALPHA [14].

Now, let us assume a for loop nest with an enclosing while loop. Then, the compile time model of the loop nest coincides with that of a for loop nest with an outer infinite loop. At run time, the infinite polyhedron turns into a polytope since it is truncated in the while dimension at the point where the while loop terminates. This polytope can be treated with the methods for for loop parallelization.

If we allow at least one while loop to be properly inside the nest, not all space-time mappings yield a target execution space that can be scanned precisely. Figure 5 shows two transformations of our execution space in Figure 4. The left target execution space can be scanned if the outer loop is on p and the inner loop on t, but a scan of the right target space always encounters the hole at coordinates $(2, 3)$, no matter how we order the two target loops.

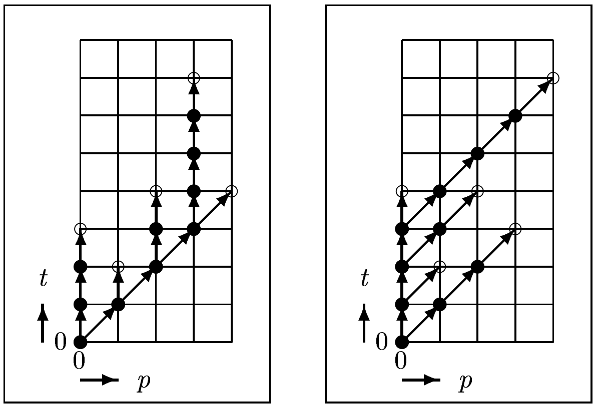

Figure 5: Scannable (left) and unscannable (right) transformation of the execution space in Figure 4

We discuss two approaches to dealing with the irregularities of the target execution space: speculative and conservative execution.

5.1.1 Speculative execution

One may choose to permit computations at holes of the target space. This amounts to the execution of what Collard calls *speculative* iterations [5, 6]. Then, one must take care that their effect either is not harmful or can be undone. Speculative iterations that can be undone are also called *safe*.

The idea of the speculative approach is that one is allowed to disregard certain while dependences. The gain is more potential parallelism than is possible with traditional polytope methods—at the expense of an invalid space-time mapping. This can be useful even in the case of a scannable transformation [6].

Algorithms for which speculative iterations are not harmful are, e.g., iterative algorithms that approximate a solution by monotonic convergence. For such algorithms, additional iterations of the enclosing while loop can only lead to an improvement of the result (modulo possible problems with floating-point arithmetic). No special parallelization techniques are required.

Collard proposes a method for the parallelization of a larger class of algorithms: loop nests whose iterations are safe [5]. He avoids the possibly bad effects of speculative iterations by moving to the single-assignment model, in which all intermediate values of variables are remembered. Then, he provides a way of "backtracking", at the point of termination of the target loop nest, to the values of the iteration that would have been the last one in the sequential execution. Note that the single-assignment model offers the possibility of more parallelism than the imperative model because it imposes only flow dependences but not output or anti-dependences [1, 5].

5.1.2 Conservative execution

The conservative approach is not to permit speculative iterations but to scan the target execution space precisely.

We have proposed a condition, called *scannability* [12], that guarantees that the target loop nest does not perform any speculative iterations. It requires that the iterations of a while loop with loop index, say, w at some level, say, l of the source loop nest are not part of some target loop (then also a while loop) at a level less than or equal to l. In other words, a while loop in the source can only be distributed across higher levels of the target loop nest. Provided the dependences do not forbid it (i.e., the space-time mapping is valid), moving an iteration farther inwards to a higher level in a loop nest is never a problem: this only postpones its execution. However, moving some iteration outwards to a lower loop level raises the possibility that, at the point of its execution, not all data necessary for its computation are available. (See also the theory of loop permutations [2].)

Note that scannability is not a property of the space-time mapping alone; it is influenced by the nesting of the axes of the target execution space. One might call a target execution space, or also the space-time mapping that generates it, *scannable* if it permits a target loop nest that satisfies the scannability condition.

Note also that scannability does not imply the validity of the space-time mapping. Take, e.g., the execution space in Figure 4 and the identity as the space-time mapping. That is, leave the loops as they are, only map one of them—it does not matter which—entirely to space. This satisfies scannability, since no loops are permuted, but it violates the while dependences of the while loop that is mapped to space.

One might wonder whether it is necessary to consider unscannable transformations at all. Unfortunately, the answer is yes—if one is interested in synchronous target loop nests, i.e., nests whose outer loop is sequential. Consider some while loop in the source loop nest but not at the outer level. Because of the dependences on its index, every while loop must be partially laid out in time. But time is the outer target loop, so portions of the while loop must move to an outer level—a violation of the scannability condition! Thus, only the trivial case of a for loop nest with an enclosing while loop can have a synchronous target loop nest that satisfies scannability.

How do we deal with unscannable transformations in the conservative approach? The target loop nest must enumerate a superset of the target execution space. To prevent execution of the body at the holes, we provide a masking predicate *active*. Activity is a property specific to a tooth, so the value of *active* depends on the particular tooth. Each tooth is rooted at some loop level, so the value of *active* also depends on the loop level. Only holes are not active at any level. The precise definition of *active* can be found in [11].

Whether a point is active is determined by whether the point previous to it in the execution comb is active. Thus, to determine whether a point is active, we propagate predicate *active* along the teeth of the execution comb, from the roots to the tips. If the tooth stretches across different processors, a signal must be sent to the appropriate neighboring processor and the evaluation of the while condition and the execution of the next point must not start before the signal has been received.

5.2 Detecting termination

Regardless of whether one chooses speculative or conservative execution, it is unclear when the entire loop nest has terminated. This problem occurs even with scannable transformations. Let us demonstrate it and its solution on Figure 5.

Even though, on the left, each processor can detect the termination of the tooth local to it, no processor has enough information to detect global termination. There are processor networks and operating systems that provide this information, but we do not want to rely on such a feature, especially, since we envision loop nests with whiles also to be realized in hardware. We have proposed a signaling scheme for termination detection whose precise formulation [11] and verification [10] is rather involved. The problem is one of exchanging termination information between different teeth. Let us just give a brief impression here and then demonstrate a simplified version with the example.

We want the scheme to work in the distributed model. Therefore, we avoid broadcasting and permit only neighbor communication. Our scheme is formulated by induction on the dimensions of the execution space. Consider first any tooth representing the innermost loop. Its tip is reached at the first point of it that is not active in the sense of the previous subsection. Thus, we may use predicate *active* for two purposes: hole detection in unscannable transformations and identifying the tip of a tooth for termination detection.

Consider Figure 6. To accumulate the termination information of the teeth of the innermost loop, the processor responsible for the tip of such a tooth propagates a termination signal to its neighboring tooth along the direction of the enclosing loop, e.g., diagonally up and right in Figure 6 (left) and up in Figure 6 (right). This signal is not sent before an according signal has been received from the previous tooth (if any). A termination signal is forwarded to the host when all signals have reached the point with maximal coordinates in each dimension (i.e., in the upper right-hand corner).

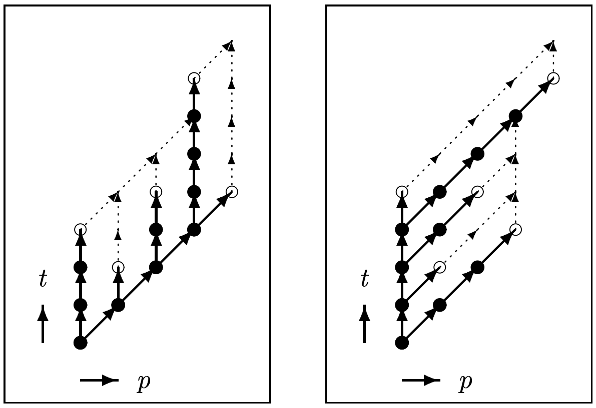

Figure 6: Signal propagation for the target execution spaces in Figure 5

This signaling scheme, formulated for arbitrarily deep loop nests, enables us not only to detect global termination but also to deal with the holes encountered in the enumeration of unscannable transformations in the conservative execution model—e.g., the transformation in Figure 6 (right) is unscannable.

5.3 Determining the data dependences

Another problem is that one cannot be sure before run time what the actual data dependences of the loop nest are. This problem is independent of the problem of parallelization; it occurs already in the sequential setting of the source loop nest. Collard observed this phenomenon and proposes a treatment of it [5].

Figure 4 shows only the while dependences of the two loops. Let us now postulate some additional data dependences. We choose a slight modification of a program by Collard [5]; note that the while conditions depend on the indices but not on array a:

for $i := 0$ while $condition(i)$ do
    for $j := 0$ while $condition(i,j)$ do
        $a[i+j] := a[i+j-1] + j$

Figure 7 depicts the data dependences (only the flow dependences [1]) and the while dependences for the execution space in Figure 4 side by side.

Figure 8 depicts the data and while dependences for a different execution space in which the second iteration of the outer loop in i leads to immediate termination of the inner loop on j. At the circle at coordinates $(1, 0)$, the termination test of the inner loop succeeds but that of the outer loop fails.

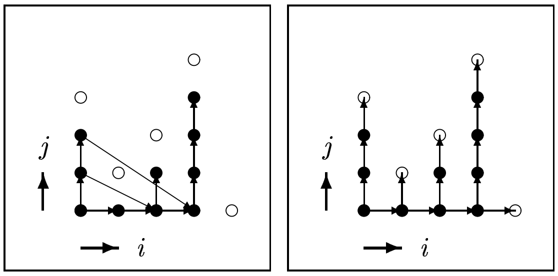

Figure 7: Execution space in Figure 4; data dependences (left); while dependences (right)

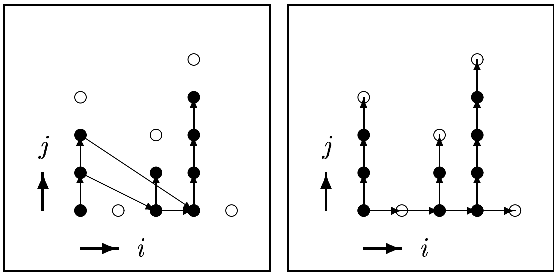

Figure 8: Another execution space; data dependences (left); while dependences (right)

Consider the sequential executions of the two execution spaces. The points are scanned bottom to top and then left to right. Say, $a[-1]$ is initialized to 0. Then, the first iterations are as follows:

at point $(0,0)$: $a[0] := a[-1]{+}0 = 0$,

at point $(0,1)$: $a[1] := a[0]{+}1 = 1$,

at point $(0,2)$: $a[2] := a[1]{+}2 = 3$, etc.

Now, in the case of Figure 7,

at point $(1,0)$: $a[1] := a[0]{+}0 = 0$,

at point $(2,0)$: $a[2] := a[1]{+}0 = 0$,

but, in the case of Figure 8,

at point $(2,0)$: $a[2] := a[1]{+}0 = 1$,

since point $(1,0)$ does not perform a computation, so the value of $a[1]$ is provided by the previous dependence in the sequential execution order, which is on point $(0,1)$.

The correct dependence is determined during the execution of the loop nest. However, the set of possible dependences can be determined at compile time. Feautrier and Collard propose an according dataflow analysis method [9]. They offer the generation of a choice of approximate data dependence graphs, depending on properties of the loop nest: whether it is single-assignment (i.e., safe) or not, and whether it is executed speculatively or not. The space-time mapping is then chosen with respect to the approximate dependence graph, i.e., it is also approximate.

5.4 Bounding the number of processors

Collard [6] proposes a limitation condition on the space-time mapping that guarantees that the execution of the target loop nest will, at no time, require infinitely many processors: he forbids all space-time mappings that map some while loop entirely to space [6]. Let us call a space-time mapping that satisfies this condition *limited*. Note that a limited mapping does not guarantee that the number of processors does not grow beyond any bound. Let us call a mapping guaranteeing the latter *bounded*. Of course, every bounded mapping is limited.

A valid space-time mapping must not lay out a while loop entirely in space (because of its while dependences), but it can lay it out partially in space and partially in time. Thus, a valid mapping must be limited, and the target execution space produced by it must have a bound on the number of processors. But this bound will not be known before the while loop has terminated. In the speculative approach, one has to check a space-time mapping for limitedness, because violations of the while dependences are permitted in order to facilitate speculative execution.

Boundedness and scannability are not comparable conditions. Consider Figure 5. The execution space on the left is scannable but unbounded, the one on the right is both unscannable and unbounded. For the other two cases, consider Figure 4, but take the outer loop on i to be a for with five iterations, not a while. Map the for loop to space and the while loop to time. The layout of a for loop in space is always bounded. Leaving the loops nested as they are also gives us scannability, but exchanging them moves the while loop outward, violating scannability.

A target space that is the result of an unbounded mapping must be folded in space in order to provide a bound on the number of processors (e.g., [7, 16]).

6 Polynomial product with a for-while loop nest

In this section, we modify the polynomial product of Section 3, by implementing the multiplication of two integers in a binary representation as a while loop of additions and shift operations. To keep the loop body simple, we assume that the two polynomials to be multiplied have no negative coefficients. We do not claim that ours is the best way to multiply; our example is for illustrational purposes.

We choose the conservative execution model.

6.1 Source program

Our modified program consists of a first loop nest for the calculation of all products that figure into the polynomial product, and a second loop nest for the calculation of the polynomial product itself. We assume that array $a'[i,j]$ is initialized to $a[i]$, array $b'[i,j]$ to $b[j]$ and array $pr[i,j]$ to 0:

for $i := 0$ to n do
  for $j := 0$ to n do
    for $\omega := 0$ while $a'[i,j] > 0$ do
      (i,j,ω)

```
for k := 0 to n do
    for l := n downto 0 do
        (k, l)
```

(i,j,ω) : if $odd(a'[i,j]) \;\rightarrow$
$\quad pr[i,j] := pr[i,j] + b'[i,j]$
fi ;
$a'[i,j] := shift\_right\; a'[i,j]$;
$b'[i,j] := shift\_left\; b'[i,j]$

(k,l) : if $k=0 \vee l=n \rightarrow c[k+l] := pr[k,l]$
$\;[]\; k\neq 0 \wedge l\neq n \rightarrow c[k+l] := c[k+l] + pr[k,l]$
fi

6.2 Index and execution space

The index space of the first loop nest is a three-dimensional parallelepiped with a square base and infinite height. A possible execution space with dependences is given in Figure 9. The while dependences (on ω) and the data dependences (on pr, a' and b') coincide.

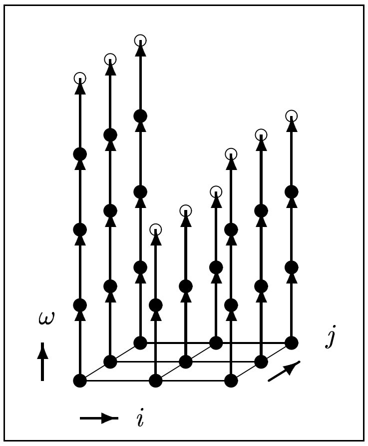

Figure 9: An execution space of the first loop nest on a base grid

The index space of the second loop nest is that in Figure 1.

Again, for simplicity of the exposition, we consider the two loop nests separately. This has the consequence that no step of the second loop nest is in parallel with any step of the first loop nest. Then, the treatment of the second loop nest is as in Section 3, and it remains only to discuss the first loop nest.

6.3 Target space

The best temporal partitioning of the first loop nest is immediately obvious from Figure 9: choose as the time slices the horizontal planes, as depicted in Figure 10. The best spatial partitioning is the projection onto these planes, i.e., the space lines are incident with the dependence lines.

There is no coordinate transformation; the target space is identical to the source space. We just map one loop to time and the others to space. Since the

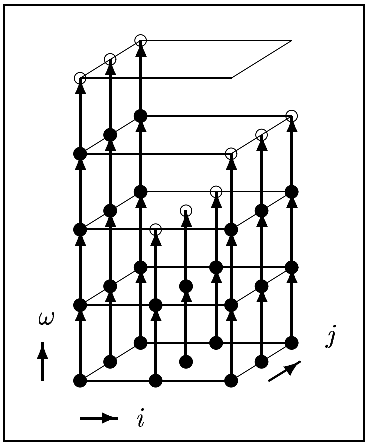

Figure 10: A time slicing of the execution space in Figure 9

while loop must not be mapped entirely to space, it is the one mapped to time.

This is similar to the trivial choice that we avoid in Section 3. Here it helps us to understand a few points. Less trivial choices, like a partial mapping of the while loop to space, are possible, of course.

6.4 Asynchronous target program

We can immediately write down the asynchronous target loop nest:

```
parfor i := 0 to n do
    parfor j := 0 to n do
        seqfor ω := 0 while a'[i,j] > 0 do
            (i, j, ω)
```

Let us comment on the problems arising with while loops, as listed in Section 5.

6.4.1 Conservative execution

Scanning the execution space is no problem: the while loop, innermost in the source, remains innermost in the target.

6.4.2 Detecting termination

Figure 11 illustrates our signaling scheme for the asynchronous target program.

First, we propagate, for every (ω,j)-plane, the termination information of its teeth along dimension j. This is necessary even though we know that all teeth in a fixed (ω,j)-plane are of the same height: with asynchronous parallelism, it is not guaranteed that processors at different j-coordinates run at the same speed. These signals project the maximal execution time of every (ω,j)-plane to the back plane, i.e., to the (i,ω)-plane at coordinate $j=n$.

Then, we propagate the termination information of the different ω-j-planes in the back plane towards $i=n$: every vertical tooth sends a signal to its right

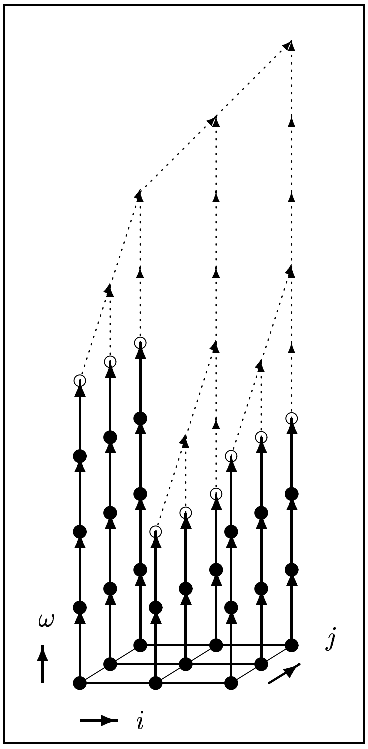

Figure 11: Signaling scheme for the asynchronous target program

neighbor when the tooth itself is ready to terminate and the termination signal of its left neighbor (if any) has been received.

Thus, when the tooth at (i,j)-coordinates (n,n) is ready to terminate and has received two signals—one along the j-dimension and the other along the i-dimension—the termination of all other teeth has been indicated at this tooth, and the program may terminate.

Here is the adorned program; the signaling commands are self-explanatory:

```
parfor i := 0 to n do
   parfor j := 0 to n do
      seqfor ω := 0 while a′[i,j] > 0 do
         (i,j,ω) ;
      if j≠0  →  wait_for_signal_from(i,j−1) fi ;
      if j≠n  →  send_signal_to(i,j+1) fi ;
      if j=n ∧ i≠0  →
         wait_for_signal_from(i−1,n)
      fi ;
      if j=n ∧ i≠n  →  send_signal_to(i+1,n)
      [] j=n ∧ i=n  →  terminate_program
      fi
```

6.4.3 Determining the data dependences

The determination of the data dependences raises no problems, since there is no dependence between any two distinct teeth of the execution space.

6.4.4 Bounding the number of processors

Our space-time mapping is bounded, since it maps only for loops to space.

6.5 Synchronous target program

In our synchronous target program, the where clause denotes the common processor masking command in the SIMD model:

```
seqfor ω := 0 while ¬terminated do
   parfor i := 0 to n do
      parfor j := 0 to n do
         where active(i,j) do
            (i,j,ω) ;
            if a′[i,j]=0  →  active(i,j) := false fi
```

Predicate *active* is our model of the SIMD processor mask. Condition *terminated*, initially false, indicates that there is no active processor. Its evaluation could be performed by the SIMD machine, e.g., by a test of the processor mask, or our signaling scheme might be used to evaluate predicate *active* (see Section 6.5.2).

Again, let us check through the list of problems.

6.5.1 Conservative execution

This space-time transformation is unscannable, since the while loop has moved from the innermost to the outermost level.

The propagation of activity information is cheap, in this case, because it is local to a processor.

6.5.2 Detecting termination

Since every tooth is local to a processor, every processor can determine its own termination autonomously. Let us employ our signaling scheme to define the predicate *terminated* via predicate *active*. When using this scheme, we must not mask inactive processors by a where clause, because inactive processors might have to pass on a termination signal. Instead, we guard the loop body of the synchronous program with the condition $active(i,j)$; signaling is only necessary if $\neg active(i,j)$:

```
seqfor ω := 0 while ¬terminated do
   parfor i := 0 to n do
      parfor j := 0 to n do
         if active(i,j)  →
            (i,j,ω) ;
            if a′[i,j]=0  →  active(i,j) := false fi
         [] ¬active(i,j)  →
            if got_signal_from(i−1) ∨ i=0  →
               if i≠n → send_signal_to(i+1)
               [] i=n → terminated := true
               fi
            fi
         fi
```

The synchronous program does not require the first phase of signals: propagating the termination of every (ω,j)-plane. On the other hand, the asynchronous program requires no inactivity test before sending signals, since inactivity is automatically given after the termination of the sequential loop.

6.5.3 Determining the data dependences

See Section 6.4.3. This problem is tied to the source program, which has not changed.

6.5.4 Bounding the number of processors

See Section 6.4.4. This problem is tied to the space-time mapping, which has not changed.

7 Conclusions

As we present the theory here, the parallelization of a nest of d loops is guaranteed to lay out one loop in time and $d-1$ loops in space (one can trade off additional dimensions from space to time). The space-time complexity of the target loop nest is equal to the time complexity of the source loop nest (whose space complexity is constant). Because of the restrictions imposed on the loop nest, the number of steps of each for loop must be affine in the problem size. This is not true for the while loops in the nest; the number of steps of a while loop may vary wildly. However, we are still guaranteed to have only one sequential loop in the target loop nest and, in the conservative approach, the space-time complexity of the target loop nest still equals the time complexity of the source loop nest.

Let m be the value of the largest coefficient of $\mathcal{A}$ and $\mathcal{B}$. Then, the time complexity of the target programs in Section 3 is $O(n \log m)$ if we charge log time for multiplications. The time complexity of the target programs in Section 6 is $O(\log m)$ for the first loop nest plus $O(n)$ for the second loop nest. We pay for the speed-up with $O(n)$ additional processors.

Acknowledgements

Thanks to Jean-François Collard and the members of the Lehrstuhl für Programmierung for comments. Financial support was received from the Deutsche Forschungsgemeinschaft (DFG).

References

[1] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer, 1993.

[2] U. Banerjee. *Loop Transformations for Restructuring Compilers: Loop Parallelization.* Kluwer, 1994.

[3] M. Barnett and C. Lengauer. Unimodularity and the parallelization of loops. *Parallel Processing Letters*, 2(2–3):273–281, 1992.

[4] J.-F. Collard. Code generation in automatic parallelizers. In C. Girault, editor, *Proc. Int. Conf. on Applications in Parallel and Distributed Computing, IFIP W.G. 10.3*, pages 185–194. North-Holland, April 1994.

[5] J.-F. Collard. A method for static scheduling of dynamic control programs. Technical Report 94-34, LIP, Ecole Normale Supérieure de Lyon, December 1994.

[6] J.-F. Collard. Automatic parallelization of `while`-loops using speculative execution. *Int. J. Parallel Programming*, 1995. To appear. Earlier version: *Proc. 1994 Scalable High Performance Computing Conf.*, pages 429–436. IEEE, May 1994.

[7] A. Darte. Regular partitioning for synthesizing fixed-size systolic arrays. *INTEGRATION*, 12(3):293–304, December 1991.

[8] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[9] P. Feautrier and J.-F. Collard. Fuzzy array dataflow analysis. Technical Report 94-24, LIP, Ecole Normale Supérieure de Lyon, July 1994.

[10] M. Griebl and C. Lengauer. A communication scheme for the distributed execution of **while** loops. Technical Report MIP-9406, Fakultät für Mathematik und Informatik, Universität Passau, June 1994.

[11] M. Griebl and C. Lengauer. On scanning space-time mapped **while** loops. In B. Buchberger and J. Volkert, editors, *Parallel Processing: CONPAR 94 – VAPP VI*, Lecture Notes in Computer Science 854, pages 677–688. Springer-Verlag, 1994.

[12] M. Griebl and C. Lengauer. On the space-time mapping of WHILE-loops. *Parallel Processing Letters*, 4(3):221–232, September 1994.

[13] L. Lamport. The parallel execution of DO loops. *Comm. ACM*, 17(2):83–93, February 1974.

[14] H. Le Verge, C. Mauras, and P. Quinton. The ALPHA language and its use for the design of systolic arrays. *J. VLSI Signal Processing*, 3:173–182, 1991.

[15] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[16] J. Teich and L. Thiele. Partitioning of processor arrays: A piecewise regular approach. *INTEGRATION*, 14(3):297–332, 1993.

[17] P. P. Tirumalai, M. Lee, and M. S. Schlansker. Parallelization of while loops on pipelined architectures. *J. of Supercomputing*, 5:119–136, 1991.

[18] M. Wolfe. Multiprocessor synchronization for concurrent loops. *IEEE Software*, pages 34–42, January 1988.

[19] Y. Wu and T. G. Lewis. Parallelizing while loops. In H. D. Schwetman, editor, *Int. Conf. on Parallel Processing*, volume II, pages 1–8. CRC Press, 1990.

[20] J. Xue. Automating non-unimodular transformations of loop nests. *Parallel Computing*, 20(5):711–728, May 1994.