A Communication Scheme for the Distributed Execution of Loop Nests with while Loops*

Martin Griebl and Christian Lengauer

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

Abstract. The mathematical model for the parallelization, or "space-time mapping", of loop nests is the polyhedron model. The presence of while loops in the nest complicates matters for two reasons: (1) the parallelized loop nest does not correspond to a polyhedron but instead to a subset that resembles a (multi-dimensional) comb and (2) it is not clear when the entire loop nest has terminated. We describe a communication scheme which can deal with both problems and which can be added to the parallel target loop nest by a compiler.

Keywords: loop parallelization, parallelizing compilation, space-time mapping, while loop

1 Introduction

With the advent of increasingly large and affordable parallel computing systems, the topic of loop parallelization has recently received increased attention. Modeling loops with convex polyhedra has proved useful in characterizing and analyzing the space of possible parallelizations of sets of nested loops. Nests of for loops are modeled with polytopes, i.e., finite polyhedra [15]. We present aspects of an extension of the model to loops whose upper bounds are not known before compile time, i.e., while loops.

---
* This work has been presented in part at the conference CONPAR 94 – VAPP VI [10].

The root of the complications that arise in this extension is the irregularity of the execution model in the presence of while loops: in the general case, one does not deal anymore with a regular computation space like a polytope or polyhedron but with a space that looks much like a multi-dimensional comb, the length of whose teeth is determined at run time by the unpredictable termination points of the while loops in the nest. There are two consequences:

1. To preserve the semantics, we must make sure that no other iterations than those executed by the sequential loop nest contribute to the result of its parallel counterpart. For for loops, this is done easily by an inspection of the (flat) surface of the polytope—but the comb of a while loop has, in general, a highly irregular surface.
2. We must detect when all loops in the parallel loop nest have terminated. For for loops, we know the last point of computation before run time, since all loop bounds are known. In the presence of a while loop, the computation space is infinite before run time; it becomes finite only when the points of termination of the while loops become known.

We address both of these problems by the same means: a propagation scheme of activity signals between the different iterations of the loop nest. This scheme is static (i.e., it can be imposed by a compiler), it requires only neighboring communication (i.e., it can also be used in a distributed processor network), and we have proved it formally correct. The proof is not part of this paper [9], but its existence is very important to us, since the code we derive is not trivial.

2 Notation

Our quantifier notation follows Dijkstra and Scholten [7]. Quantification over a dummy variable x is written $(Q\ x : R.x : P.x)$. Q is the quantifier, R is a predicate in x representing the range, and P is a term that depends on x. Formal logical deductions are given in the form:

$$\begin{array}{ll} & \textit{formula}_1 \\ op & \{ \text{ comment explaining the validity of relation } op \ \} \\ & \textit{formula}_2 \end{array}$$

where *op* is an operator from the set $\{\Leftarrow, \Leftrightarrow, \Rightarrow\}$.

A while loop is commonly written: while *condition* do *body* end. We view the while loop as a for loop with a dynamic upper bound, that is, we add an artificial for loop index that counts the number of executed iterations. Our while loop notation is: for $index := 0$ while $condition(index)$ do *body* end, 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.

Note that every while loop can be represented this way.

3 for loops and the polytope model

Parallelization in the polytope model proceeds as follows [15].

First, one transforms d perfectly nested source loops into a d-dimensional polytope. We call this polytope the *index space* and denote it with $\mathcal{I}$. Each loop defines the extent of $\mathcal{I}$ in one dimension. Each point of $\mathcal{I}$ represents one iteration step of the loop nest. The coordinates of the point are given by the values of the loop indices at that step, i.e., the $\mathcal{I} \subset \mathbb{Z}^d$.

Then, one applies an affine coordinate transformation $\mathcal{T}$, the "space-time mapping", to the polytope and obtains another polytope in which some dimensions enumerate space and the others enumerate time. We call the transformed polytope the *target space* and denote it with $\mathcal{T}\mathcal{I}$.

Finally, one translates this polytope back to a nest of target loops, where each space dimension becomes a parallel loop and each time dimension becomes a sequential loop. The target loop nest specifies asynchronous parallelism if the outer loops are the parallel ones, and synchronous parallelism if the outer loops are the sequential ones. When $\mathcal{T}$ is not unimodular, i.e., its inverse is not an integer matrix, $\mathcal{TI}$ contains "holes", i.e., it is not convex even though $\mathcal{I}$ is [2]. 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, there is always a target loop nest that scans $\mathcal{TI}$ precisely—whether $\mathcal{T}$ is unimodular [4] or not [27].

4 while loops and the polyhedron model

4.1 Execution combs

A perfect 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 the 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 test for all outer loops up to and including w (i.e., d tests) and for the first iteration of loop w, if any. Due to the changed interpretation of a point, we introduce circles to represent successful termination tests.

A nest with while loops executes only a subset of the infinite index space $\mathcal{I}$. We call this subset the *execution space* and name it $\mathcal{X}$. The dependences of points in $\mathcal{X}$ make it look like a (possibly, multi-dimensional) comb [11], as demonstrated in Fig. 1. We call it also the *execution comb*. We give its transformation under the space-time mapping the name $\mathcal{TX}$. 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.

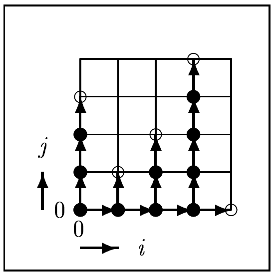

Fig. 1. A non-convex execution space with while dependences.

We disregard data dependences completely in our exposition. (This is not to suggest that there is nothing to be said about them [6].) Our understanding is that whatever data dependences exist in the loop nest must be preserved by the space-time mapping. This is not the only choice one can take (see Sect. 8).

4.2 Scannability

If only the outermost loop of the nest is a while loop, the execution space of the nest is still convex and, thus, can be scanned precisely. But if we allow at least one while loop properly inside the nest, only space-time mappings that fulfill certain conditions yield a precisely scannable execution space [11]. For example, consider Fig. 2. You may interpret p as the space (processor) and t as the time coordinate. 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.

We have proposed a condition, called *scannability* [11], which guarantees that the target loop nest can scan the execution comb precisely. Formally, it is a condition on the matrix that represents the space-time mapping and the order of the target loops. Intuitively, 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

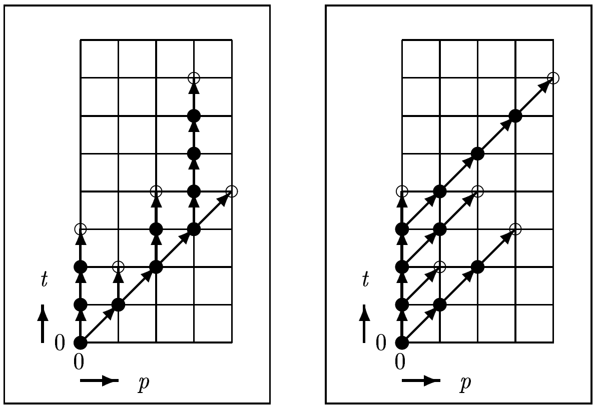

Fig. 2. Scannable (left) and unscannable (right) transformation of the execution space in Figure 1

(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 (in space-time). 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 [1].)

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.

4.3 Unscannability

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.

We do find it worth-while to consider unscannability: the synchronous programming model is the basis of many SIMD machines.

5 Controlling the scan of an unscannable execution space

This section presents our approach to dealing with unscannable target loop nests. We proceed quite formally, so, for a better understanding, here is a short preview.

5.1 Idea

As a small motivation, we consider a simple example:

Example 1. The following program sums up n rows of numbers; the number of entries that shall be summed up varies for different rows; the end is indicated by $\perp$:

for $i := 0$ to n do
    for $j := 0$ while $A[i,j] \neq \perp$ do
        $sum[i] := sum[i] + A[i,j]$
    end
end

Obviously, schedule $\Theta(i,j) = j$ and allocation $\Pi(i,j) = i$ are valid. Thus, we obtain an asynchronous target program by simply replacing the (outer) for loop by a parfor loop. To get a synchronous target program, we must choose an unscannable transformation since we have to permute the loops on space (index p) and time (index t). Additionally, the permutation puts the while loop whose condition depends on *both* loop indices, outermost. How to determine a suitable termination condition for the while loop is subject of Section 6; let us use here some unspecified upper bound up:

for $t := 0$ while up do
    parfor $p := 0$ to n do

```
        sum[p] := sum[p] + A[p, t]
    end
end
```

Note that, for *any* value of *up*, the scanned space is a rectangle, and, thus, different from the source space which is a comb. This problem is typical of unscannable transformations.

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.

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.

For our example, we are aiming at the following target program, where the calculation of *terminated* and *active* are the subject of the next two sections:

```
for t := 0 while ¬terminated do
    parfor p := 0 to n do
        if active and A[i, j] ≠ ⊥ then sum[p] := sum[p] + A[p, t]  fi
    end
end
```

The complete target program, including all signals, is not presented before Section 7.2 since some needed auxiliary functions are not yet defined.

To avoid additional case distinctions in the formalization, we shall assume that all loops in the nest are while loops.

5.2 Formalization

We call an execution space that cannot be scanned by any target loop nest without encountering holes *unscannable*. Scanning an unscannable execution space $\mathcal{TX}$ with some fixed target loop nest yields a proper superset of $\mathcal{TX}$; we name this set $\mathcal{TS}$ and its inverse image under the space-time mapping $\mathcal{S}$. In this section, we define a logical predicate that is an accurate recognizer of the points in $\mathcal{TX}$, i.e., that distinguishes the points in $\mathcal{TX}$ from those outside. In the following section, we use this predicate to prevent the execution of holes in the target polyhedron at run time.

In order to avoid case distinctions in definitions and proofs, we assume in our formalization a unified loop syntax, in which a for loop is written as a while loop with an index that starts at some lower bound (say, lb_r for the loop at level r) and that traverses the fixed range of values defined by the loop.

Definition 1 (Activity recognizer $active_r$). Let r be some level of the source loop nest and w the while loop at that level. $active_r$ holds for any point p in $\mathcal{I}$ iff the source program enumerates p, that is, iff at least the while condition $cond_r$ of loop w is evaluated at point p. Formally:

$$
\begin{array}{lll}
\multicolumn{3}{l}{(\forall\ (p_1,\cdots,p_d)\ :\ (p_1,\cdots,p_d)\in\mathcal{I}\ :\ (\forall\ r:1\leq r\leq d\ :\ active_r(p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d) =} \\
\quad \textsf{if}\ p_r > lb_r & \rightarrow active_r(p_1,\cdots,p_r-1,lb_{r+1},\cdots,lb_d)\wedge cond_r(p_1,\cdots,p_r-1) & (1)\\
\quad [\,]\ p_r = lb_r \wedge r>1 & \rightarrow active_{r-1}(p_1,\cdots,p_{r-1},lb_r,\cdots,lb_d)\wedge cond_{r-1}(p_1,\cdots,p_{r-1}) & (2)\\
\quad [\,]\ p_r = lb_r \wedge r=1 & \rightarrow \text{true} & (3)\\
\quad [\,]\ p_r < lb_r & \rightarrow \text{false} & (4)\\
\quad \textsf{fi}\ )) & &
\end{array}
$$

The alternatives of the defining equation can be explained as follows. Point $(p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)$ is active with respect to level r iff

(1) the point represents some non-leading step of a loop, the while condition holds and the previous step is active with respect to level r, or

(2) the point represents the first step of an inner loop, the while condition holds for the immediately enclosing loop and the point is active with respect to the level of that loop, or

(3) the point represents the first step of the entire loop nest.

In all other cases, $(p_1, \cdots, p_r, lb_{r+1}, \cdots, lb_d)$ is inactive with respect to level r. These include the case where the while condition of w is violated (covered by alternatives (1) and (2)), and the case that the point is not even in the index space (alternative (4)). Note, that points for which the while condition holds at level r but not at level $r+1$ are active with respect to level r but not with respect to levels $r+1$ and deeper.

The recursive definition of predicate $active_r$ follows the dependences which are introduced by the while indices. Since our space-time mapping must respect these dependences, we can be sure that, during scanning, the activity of any point p in $\mathcal{TI}$ need not be checked before the activity of its predecessor has been checked. Therefore, we can compute predicate $active_r$, for every point on every tooth of the execution comb, in sequence from the root to the tip and store the result until it is needed. Note that $active_r$ at point $(p_1, \cdots, p_r, lb_{r+1}, \cdots, lb_d)$ is calculated from $cond_r(p_1, \cdots, p_r, lb_{r+1}, \cdots, lb_d)$ or $cond_{r-1}(p_1, \cdots, p_{r-1}, lb_r, \cdots, lb_d)$, which is usually data-dependent on the loop body at $(p_1, \cdots, p_r, lb_{r+1}, \cdots, lb_d)$ or $(p_1, \cdots, p_{r-1}, lb_r, \cdots, lb_d)$, resp. In this case, the computation of the values of $active_r$ is executed alternately with the computations of the loop body. (If the condition does not depend on the loop body one can replace the while loop by a for loop [26].)

Note that $active_r$ at point $(p_1, \cdots, p_r, lb_{r+1}, \cdots, lb_d)$ depends on $active_r$ at exactly one other point (see the definition); $active_r$ at point $(p_1, \cdots, p_r, lb_{r+1}, \cdots, lb_d)$ is used for the computation of $active_r$ at point $(p_1, \cdots, p_r{+}1, lb_{r+1}, \cdots, lb_d)$ and, if r is not the innermost loop level, also for the computation of $active_{r+1}$ at itself.

Since the polyhedron of a while loop nest contains points that do not model a loop step but only a terminating test, we also require a recognizer, *executed*, for points of $\mathcal{TI}$, or $\mathcal{I}$, that do represent the execution of the loop body.

Definition 2 (Recognizer $\mathit{executed}_r$ and $\mathit{executed}$).

$$(\forall\, r \,:\, 1 \leq r \leq d \,:\, (\forall\, p \,:\, p \in \mathcal{I} \,:\, \mathit{executed}_r(p_1, \cdots, p_r, \mathit{lb}_{r+1}, \cdots, \mathit{lb}_d) \Leftrightarrow$$
$$(\mathit{active}_r(p_1, \cdots, p_r, \mathit{lb}_{r+1}, \cdots, \mathit{lb}_d) \wedge \mathit{cond}_r(p_1, \cdots, p_r))))$$

$$(\forall\, p \,:\, p \in \mathcal{I} \,:\, \mathit{executed}(p) \Leftrightarrow \mathit{executed}_d(p_1, \cdots, p_d))$$

At this point, we have the machinery for a formal definition of the execution space:

Definition 3 (Execution space). $\mathcal{X} = \{p \in \mathcal{I} \,:\, \mathit{executed}(p)\}$

Later on, we need its extension to all points that are active in some dimension:

Definition 4 (Activity space). $\widehat{\mathcal{X}} = \{p \in \mathcal{I} \,:\, (\exists\, r \,:\, 1 \leq r \leq d \,:\, \mathit{active}_r(p))\}$

6 Termination Detection

The most difficult problem in the derivation of target loop code is the determination of the upper bound for the outermost loop (if the source contains at least one inner while loop). This is even a problem for scannable transformations.

Example 2. Take again the loop nest of Example 1 and the space-time mapping $(i,j) \mapsto (p, t-p)$. (This fits the left part of Fig. 2, except that the upper bound of the loop on p is known. We assume that the transformation respects the dependences of the body. The while dependence is satisfied.) Every processor executes the while loop for a fixed value of i. Clearly, each processor can determine the condition for its termination locally, but no processor can detect the termination of any other processor, unless additional measures are taken. Say some statement follows the parallel loop nest:

parfor $p := 0$ to n do

    seqfor $t := p$ while $A[p, t-p] \neq \bot$ do

        $\mathit{sum}[p] := \mathit{sum}[p] + A[p, t-p]$

```
    end
end
statement
```

How do we know when to start the execution of *statement*?

Here, we choose the asynchronous case (where the time loop is innermost), but the same problem occurs in the synchronous case (where the time loop is outermost): there is no elegant way of expressing the termination condition of the outermost loop. In this example, the termination condition at a given time step t is given by

$$(\forall\ p\ :\ 0 \leq p \leq n\ :\ (\exists\ t'\ :\ p \leq t' \leq t\ :\ A[p, t'-p] = \perp)).$$

These quantifications are potentially costly because, in general, there is no bound on their ranges.

We present a solution of the termination problem that requires only communication between neighboring processors. Our solution is based on the propagation of predicate *active*. Again, let us precede a rigorous formal treatment by a short informal preview.

6.1 Idea

The basic idea of our solution is as follows: if carefully selected teeth along dimension r of the execution space inform their (still executing) neighbors in dimensions $1, \cdots, r-1$ of their termination, the maximal coordinates of every dimension of the execution space are propagated.

If we ensure that no tooth terminates before it has been informed of the termination of its neighbors, points that are involved in these communications are partially maximal until the point $(p_1, \cdots, p_d)$ in $\mathcal{S}$ is reached whose coordinates are all maximal, i.e., have the property $(\forall\ p'\ :\ p' \in \mathcal{X}\ :\ (\forall\ r\ :\ 1 \leq r \leq d\ :\ p'_r \leq p_r))$. When scanning this point, we can terminate all target loops.

The propagation of the maxima, up to depth r, proceeds by valueless signals. Signal $sig_k^r(p_1, \cdots, p_r, lb_{r+1}, \cdots, lb_d)$ starts at point $(p_1, \cdots, p_r, lb_{r+1}, \cdots, lb_d)$ at depth r and is sent to the neighboring tooth in direction k, where k is some outer level with respect to r, i.e., $k < r$.

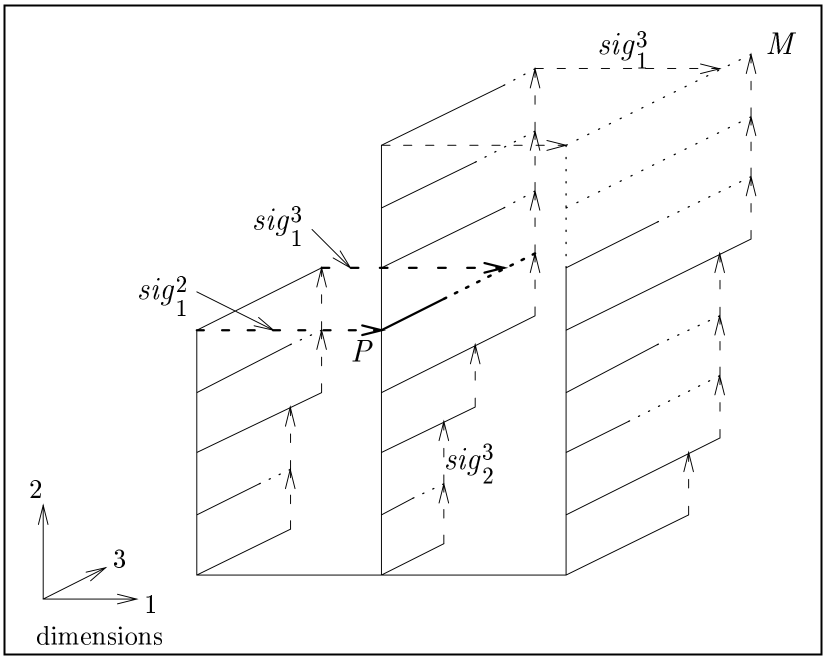

Fig. 3. A three-dimensional comb.

The main problem is how to establish whether a tooth can terminate immediately when the corresponding while condition is violated or whether it has to wait for some signal first.

Example 3. Consider the three-dimensional comb of Fig. 3.

Our scheme is more easily understood in the synchronous model.

In the figure, the teeth of $\mathcal{X}$ are represented by solid lines. Some teeth are extended by dotted lines, indicating that they are waiting for at least one signal. Points (of $\mathcal{S}$) on dotted lines do not execute the loop body, they only wait for signals. Signals are represented by dashed arrows.

Our aim is to identify the point M whose coordinates are maximal in every dimension. The first coordinate of M is quite easy to determine: it is the value at which the outermost loop terminates.

The second coordinate of M is the maximum of the lengths of all teeth pointing up (in the figure). To determine it, every vertical tooth tells its right neighbor the maximum of its own height and the maximal height left of it. This is the meaning of sig_1^2. If a vertical tooth is ready to terminate but did not yet receive sig_1^2 from its left neighbor, it must wait (except for the leftmost tooth) until this signal is received. (In the figure, the tooth at the right must wait.) Then, it itself sends sig_1^2 on to its right neighbor and terminates. The following formal property holds for all teeth in dimension 2:

$$sig_1^2(p_1, p_2, 0) \Rightarrow (\forall\ p_1', p_2'\ :\ (p_1', p_2', 0) \in \mathcal{X} \wedge p_1' \leq p_1\ :\ p_2' \leq p_2)$$

The determination of the maximal depth of teeth in each vertical plane (p_1 constant) proceeds analogously. Signals sig_2^3 are sent from every (perhaps extended, since waiting) tooth τ along dimension 3 to its upper neighbor of that plane, indicating that the current depth (the length of τ) is maximal for all teeth in dimension 3 to the left and including τ, for fixed p_1. Formally:

$$sig_2^3(p_1, p_2, p_3) \Rightarrow (\forall\ p_2', p_3'\ :\ (p_1, p_2', p_3') \in \mathcal{X} \wedge p_2' \leq p_2\ :\ p_3' \leq p_3)$$

To combine the maxima of all vertical planes, the maximal point of each plane sends a signal sig_1^3 to its right neighbor. Again, it is important that this right neighbor must not terminate before the signal is received. Which teeth must wait? The maximal depth in every vertical plane is reached at the end of the (perhaps extended) vertical tooth that forms the base of this plane. This height was propagated to the right neighbor by sig_1^2. At that height, the maximal depth will also be propagated. Therefore, the tooth, rooted at that point (e.g., P in the figure) which received sig_1^2 and which points into dimension 3 (the thick tooth in the figure), must wait until sig_1^3 is received. Again, formally:

$$sig_1^3(p_1, p_2, p_3) \Rightarrow (\forall\ p_1', p_2', p_3'\ :\ (p_1', p_2', p_3') \in \mathcal{X} \wedge p_1' \leq p_1\ :\ p_2' \leq p_2 \wedge p_3' \leq p_3)$$

The formal properties implied by the signals form a pattern that we call the partial maximality (of a point). M is partially maximal with respect to all dimensions and is, therefore, the maximal point.

6.2 Formalization

In the following, we define partial maximality recursively for an arbitrary number of dimensions. Then, we construct a mechanism that sends signals from partially maximal points to the appropriate destinations.

We state two lemmata without proof. The proofs can be found in complete rigour in [9]. To include the host of the processor array, we introduce a little hack. We imagine one more dimension, 0, which has extent 2. At position 0 is the polyhedron, and at position 1 is the host. Then, we introduce signals that travel from position 0 to position 1. They are meant to communicate the termination of the target loops to the host.

Definition 5 (Partial maximality m_k^r). $m_k^r(p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)$ iff for fixed indices at levels 1 to $k-1$ and for points $(p'_1,\cdots,p'_r,lb_{r+1},\cdots,lb_d)$ below $(p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)$ at level k $(p'_k \leq p_k)$, point $(p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)$ is maximal in all dimensions $k+1,\cdots,r$. Formally:

$$
\begin{aligned}
&(\forall\ r\ : 0<r\leq d\ :\ (\forall\ (p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d) : (p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)\in\widehat{\mathcal{X}}\ :\\
&\qquad(\forall\ k\ :\ 0\leq k<r\ :\ m_k^r(p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)\ =\\
&\qquad\quad(\forall\ p'_k,\cdots,p'_r\ :\ (p_1,\cdots,p_{k-1},p'_k,\cdots,p'_r,lb_{r+1},\cdots,lb_d)\in\widehat{\mathcal{X}}\wedge p'_k\leq p_k\ :\\
&\qquad\qquad\qquad\qquad p'_{k+1}\leq p_{k+1},\cdots,p'_r\leq p_r))))
\end{aligned}
$$

If $m_k^r(p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)$, we call point $(p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)$ *partially maximal* with respect to dimensions $k+1$ to r.

Note that, for $k=0$, the right-hand side of Def. 5 simplifies to:

$$(\forall\ p'_1,\cdots,p'_r\ :\ (p'_1,\cdots,p'_r,lb_{r+1},\cdots,lb_d)\in\widehat{\mathcal{X}}\ :\ p'_1\leq p_1\wedge\cdots\wedge p'_r\leq p_r)$$

For our communication scheme of m by signals, we need an additional predicate, $w_k^r(p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)$, which indicates that the tooth that is rooted at $(p_1,\cdots,p_{r-1},lb_r,\cdots,lb_d)$ and extends along dimension r must wait until signal sig_k^r arrives at some point $(p_1,\cdots,p_{r-1},p'_r,lb_{r+1},\cdots,lb_d)$ with $p'_r\geq p_r$. These additional points at which a tooth waits but executes nothing make the difference between $\mathcal{S}$ and $\mathcal{X}$. The lemmata that follow are valid for all points in $\mathcal{S}$.

Definition 6 (sig and w).

$(\forall\ (p_1,\cdots,p_d)\ :\ (p_1,\cdots,p_d)\in\mathcal{I}\ :\ (\forall\ k,r\ :\ 0\le k<r\le d\ :\ sig_k^r(p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)\ =$

$\neg executed_r(p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)\wedge$

$(\forall\ s\ :\ 1\le s<r\ :\ \neg w_s^r(p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)\vee sig_s^r(p_1,\cdots,p_s-1,\cdots,p_r,lb_{r+1},\cdots,lb_d))\wedge$

$(r>k+1\Rightarrow sig_k^{r-1}(p_1,\cdots,p_{r-1},lb_r,\cdots,lb_d))\ ))$

For all other points sig is initialized with false:

$(\forall\ (p_1,\cdots,p_d)\ :\ (p_1,\cdots,p_d)\in\mathbb{Z}^d-\mathcal{I}\ :\ (\forall\ k,r\ :\ 0\le k<r\le d\ :$

$sig_k^r(p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)\ =\text{false}))$

$(\forall\ (p_1,\cdots,p_d)\ :\ (p_1,\cdots,p_d)\in\mathcal{I}\ :\ (\forall\ k,r\ :\ 0\le k<r\le d\ :\ w_k^r(p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)\ =$

$\mathsf{if}\ k=0 \rightarrow \text{false}$ (1)

$[]\ r=k+1\wedge k>0\wedge p_{k+1}=lb_{k+1}\rightarrow p_k\neq lb_k$ (2)

$[]\ r>k+1\wedge k>0\wedge p_{k+1}=lb_{k+1}\rightarrow sig_k^{r-1}(p_1,\cdots,p_k-1,\cdots,p_{r-1},lb_r,\cdots,lb_d)$ (3)

$[]\ r\ge k+1\wedge k>0\wedge p_{k+1}>lb_{k+1}\rightarrow$ (4)

$w_k^r(p_1,\cdots,p_{r-1},p_r-1)\ \wedge\ \neg sig_k^r(p_1,\cdots,p_k-1,\cdots,p_{r-1},p_r-1,lb_{r+1},\cdots,lb_d)$

fi))

These equations can be explained as follows.

sig states that any point of a tooth that need not be executed and that does not have to wait for any signal sends signal sig_k^r if either the tooth and the signal lie in a two-dimensional plane (recursion base) or the root $(p_1,\cdots,p_{r-1},lb_r,\cdots,lb_d)$ of the tooth has already sent the signal into the same direction (recursion).

$w_k^r(p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)$ states whether the point $(p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)$ at level r has to wait for some signal from direction k:

(1) No point has to wait for signals from the host.

(2) In every two-dimensional subspace (dimensions k and $r=k+1$), every tooth, i.e., at least the first point of it (with $p_{k+1}=lb_{k+1}$), has to wait for a signal from the tooth immediately preceding it—if any, i.e., if $p_k\neq lb_k$.

(3) In every at least three-dimensional subspace (dimensions k to $r>k+1$),

every tooth parallel to dimension r, i.e., at least the first point of it, has to wait for some signal from direction k iff its root at level $r-1$ has received a signal from the same direction k.

(4) Any point of a tooth that is not the first point has to wait for a signal iff its predecessor on the tooth had to wait and did not receive the signal it was waiting for.

6.3 Signals and their significance for local maximality

The main result of this section is that signals *sig* correctly propagate property m of local maximality. We state this in two separate lemmata.

Lemma 7 (Local maximum). *A point of some tooth along dimension r that need not be executed with respect to r and need not wait for a signal is maximal with respect to dimension r. Formally:*

$$(\forall\ (p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)\ :\ (p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)\in\mathcal{S}\ :\ \neg executed_r(p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)\wedge$$
$$(\forall\ s\ :\ 1\leq s<r\ :\ (\neg w_s^r(p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)\vee sig_s^r(p_1,\cdots,p_s-1,\cdots,p_r,lb_{r+1},\cdots,lb_d)))\Rightarrow$$
$$(\forall\ p'_r\ :\ (p_1,\cdots,p_{r-1},p'_r,lb_{r+1},\cdots,lb_d)\in\mathcal{S}\ :\ p'_r\leq p_r))$$

Lemma 8 (*sig* implements m).

$$(\forall\ k,r\ :\ 0\leq k<r\leq d\ :\ (\forall\ (p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)\ :\ (p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)\in\mathcal{S}\ :$$
$$sig_k^r(p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)\Rightarrow m_k^r(p_1,\cdots,p_r,lb_{r+1},\cdots,lb_d)))$$

Our aim has been to identify the point of $\mathcal{S}$ with maximal coordinates in all dimensions. The scanning of this point indicates the termination of the entire target loop nest. We have constructed a signaling scheme in which this point sends signal sig_0^d to the host.

Remark 9.

- A tooth in direction r must send at least one signal, into direction $r-1$.
- One simple optimization can be made immediately: signals need not be sent to points p along a tooth that has terminated, i.e., points that neither are

active nor wait for a signal. Thus, $sig_k^r(p_1, \cdots, p_r, lb_{r+1}, \cdots, lb_d)$ implies that, for any k' with $k' < k$, $sig_{k'}^r(p_1, \cdots, p_r, lb_{r+1}, \cdots, lb_d)$ need not be sent. In Example 3, we have already omitted these signals, but they are, of course, part of Def. 6. Thus, every terminating tooth has to send at most one signal.

– In summary, every tooth must send exactly one signal when it terminates.

One could reduce the size of $\mathcal{S}$ by extracting tighter individual termination conditions for each target loop instead of one common termination condition for all loops.

7 The target code

There are standard techniques with which a target polyhedron can be transformed into a parallel loop nest [15]. In this section, we present an augmentation of the target loop body such that the irregular shape of the transformed execution space is dealt with properly. Our augmentation implements the signaling scheme presented in the previous section. We prove that, while the target loop bounds coincide with the boundaries of $\mathcal{TS}$, the augmented body reduces the execution of the loop nest to precisely the points of $\mathcal{TX}$.

We present the target code in several stages. We start with a target program that is still abstract in the following sense: (1) it is single-assignment, and (2) it does not specify whether the parallelism is synchronous or asynchronous. Next, we explain how to introduce re-assignments into the target program by a simple transformation. Finally, we discuss the differences between synchrony and asynchrony in the parallel execution.

We have worked out a formal proof of the correctness of our implementation [9]. It certifies that the target program executes the transformed source loop body for all and only those points whose inverse image is in the execution space.

7.1 The single-assignment code

The target code must state communications (send and receive primitives) explicitly. We use three primitives that are executable on the PARIX operating system

[18], one for transmitting and two for receiving. The reception mode depends on the type of message transmitted.

Our signaling scheme contains two types of message.

– One type is responsible for the propagation of information along one tooth. An example is the value of the predicate *active*. The receipt of a message of this type is necessary for the execution of the loop body at the respective point.
– The other type of messages is for signals *sig*. These signals must be “probed” [17], since execution of a loop iteration may proceed without their receipt.

Let us briefly describe the three primitives:

– $\mathsf{asend}(dir, list\_of\_vals)$ is a PARIX command that transmits a list of values into a specified direction in $(d-1)$-dimensional space. Communication is asynchronous; this makes our code generation scheme independent of the space-time mapping, as discussed further in the conclusions.
– $\mathsf{receive}(dir, list\_of\_vals)$ is a PARIX command that performs a blocking receive of a list of values from a specified direction.
– $\mathsf{creceive}(dir, list\_of\_vals)$ is a command defined by us that performs a non-blocking receive of a list of values from a specified direction. Martin [17] defines the value of a probe $\overline{Y}$ on some communication action Y as a boolean value that indicates whether the corresponding communication action is pending. In our context, Y is a receive command from some direction and, thus, $\overline{Y}$ holds iff the corresponding asend command did already take place. With this construct, we define $\mathsf{creceive}(dir, list\_of\_vals)$ as follows:

$$\mathsf{if}\ \overline{\mathsf{receive}(dir, list\_of\_vals)}\ \mathsf{then}\ \mathsf{receive}(dir, list\_of\_vals)\ \mathsf{fi}.$$

Note: $\overline{\mathsf{receive}(dir, list\_of\_vals)}$ corresponds roughly to the PARIX command **Select**(**ReceiveOption**(*dir*)) [18].

We are now able to present the single assignment target program. For simplicity, we implement termination by the PARIX command **AbortServer** that is executed only by an iteration that is maximal for all dimensions (i.e., partially maximal with respect to dimensions 1 to d, as defined in Def. 5). Thus, the target while loops need no upper bound.

The target loop nest is displayed in Fig. 4 and refined in the subsequent Figs. 5–6. This code is for asynchronous parallel execution on distributed-memory machines. The necessary changes for synchronous execution and/or shared-memory are discussed in Sect. 7.2.

```
k_1(lb_1) := 0
targ_active_1(lb_1) := true
for q_1 := lb_1 while true do
⋮
    for q_d := lb_d while true do
        b_1^pre(p_1)
        ⋮
        b_d^pre(p_1, ···, p_d)
        if targ_active_d(p_1, ···, p_d) and cond_d(p_1, ···, p_d) then
            b(p_1, ···, p_d)
        fi
        b_1^post(p_1)
        ⋮
        b_d^post(p_1, ···, p_d)
    end
⋮
end
```

Fig. 4. The target loop nest

The indicated augmentation of the loop body for every level r, i.e., $b_r(p_1, \cdots, p_r)$, consists of two parts:

– The prefix $b_r^{\text{pre}}(p_1, \cdots, p_r)$, executed before the transformed source loop body, is displayed in Fig. 5. It receives all necessary data and calculates the output values of all variables that are introduced by our signaling scheme.

- The postfix $b_r^{\mathrm{post}}(p_1, \cdots, p_r)$, executed after the source loop (at least for asynchronous execution, see Remark 10), is displayed in Fig. 6. It is responsible for sending all necessary information.

$(\forall\ r\ :\ 1 \leq r \leq d\ :\ b_r^{\mathrm{pre}}(p_1, \cdots, p_r)$:

if $p_{r+1} = lb_{r+1}$ and $\cdots$ and $p_d = lb_d$ then

A :
/* *receive signals* */
creceive$(chan_{\mathrm{S}}^1, [rcved_1^r(p_1, \cdots, p_r)])$
$\vdots$
creceive$(chan_{\mathrm{S}}^{r-1}, [rcved_{r-1}^r(p_1, \cdots, p_r)])$

B :
/* *receive data* */
receive$(chan_{\mathrm{D}}^r, [targ_active_r(p_1, \cdots, p_r),\ k_r(p_1, \cdots, p_r),$
$w_1^r(p_1, \cdots, p_r), \cdots, w_{r-1}^r(p_1, \cdots, p_r)])$

/* *calculate output values for all signals* */

C :
$w_1^r(p_1, \cdots, p_r{+}1) := w_1^r(p_1, \cdots, p_r)$ and not $rcved_1^r$
$\vdots$
$w_{r-1}^r(p_1, \cdots, p_r{+}1) := w_{r-1}^r(p_1, \cdots, p_r)$ and not $rcved_{r-1}^r$

D : ← only for $r < d$
$w_1^{r+1}(p_1, \cdots, p_r, lb_{r+1}) := rcved_1^r(p_1, \cdots, p_r)$
$\vdots$
$w_{r-1}^{r+1}(p_1, \cdots, p_r, lb_{r+1}) := rcved_{r-1}^r(p_1, \cdots, p_r)$
$w_r^{r+1}(p_1, \cdots, p_r, lb_{r+1}) := (p_r \neq lb_r)$

E :
$k_r(p_1, \cdots, p_r + 1) := k_r(p_1, \cdots, p_r)$

if $targ\_active_r(p_1, \cdots, p_r)$ and $cond_r(p_1, \cdots, p_r)$ then
$sigval_r(p_1, \cdots, p_r) :=$ false
$targ\_active_r(p_1, \cdots, p_r{+}1) :=$ true

F : ← only for $r < d$
$targ\_active_{r+1}(p_1, \cdots, p_r, lb_{r+1}) :=$ true
$k_{r+1}(p_1, \cdots, p_r, lb_{r+1}) := r$

else

G :
$w^r(p_1, \cdots, p_r) := w_1^r(p_1, \cdots, p_r{+}1)$ or $\cdots$ or $w_{r-1}^r(p_1, \cdots, p_r{+}1)$
$sigval_r(p_1, \cdots, p_r) :=$ not $w^r(p_1, \cdots, p_r)$
$targ\_active_r(p_1, \cdots, p_r{+}1) :=$ false

H : ← only for $r < d$
$targ\_active_{r+1}(p_1, \cdots, p_r, lb_{r+1}) :=$ false
if $sigval_r(p_1, \cdots, p_r)$ then
$k_{r+1}(p_1, \cdots, p_r, lb_{r+1}) := k_r(p_1, \cdots, p_r)$
else
$k_{r+1}(p_1, \cdots, p_r, lb_{r+1}) := r$
fi

fi

fi

)

Fig. 5. The prefix of the transformed loop body

$(\forall\ r\ :\ 1 \le r \le d\ :\ b_r^{\text{post}}(p_1, \cdots, p_r)\ :$

if $p_{r+1} = 0$ and $\cdots$ and $p_d = 0$ then

I : if $k_r(p_1, \cdots, p_r) = 0$ and $sigval_r(p_1, \cdots, p_r)$ then **AbortServer**() fi $\leftarrow$ only for $r = d$

J : /* *send data* */
asend($chan_{\mathrm{D}}^r$, $[targ\_active_r(p_1, \cdots, p_r{+}1), k_r(p_1, \cdots, p_r{+}1), w_1^r(p_1, \cdots, p_r{+}1), \cdots, w_{r-1}^r(p_1, \cdots, p_r{+}1)]$)

K : /* *send signals* */
if $k_r(p_1, \cdots, p_r) \neq 0$ and $sigval_r(p_1, \cdots, p_r)$ then
asend($chan_{\mathrm{S}}^{k_r}$, [true])
fi

fi

$)$

Fig. 6. The postfix of the transformed loop body

Let us now discuss the code in detail:

Variables:

- The readers who want to interpret the list of parameters $(p_1, \cdots, p_r)$ as a point in d-space should add the constant trailing coordinates $lb_{r+1}, \cdots, lb_d$.
- $chan_{\mathrm{S}}^k$ represents the channel for the signals that travel along dimension k. $chan_{\mathrm{D}}^r$ represents the channel for the data, i.e. $targ\_active_r$, k_r, and w_k^r, $1 \le k < r$, that are propagated along every tooth.
- k_r is the direction in which the tooth must send a signal before it terminates. It corresponds to the lower index k of $sig_k^r(p_1, \cdots, p_r, lb_{r+1}, \cdots, lb_d)$. The optimization outlined in Remark 9 ensures that there is a unique k for each tooth.
- $sigval_r$ is equal to sig_k^r of Def. 6.
- w_k^r is as in Def. 6.

- $targ\_active_r(\mathcal{T}p)$ is the counterpart of the predicate *active* in the source loop nest, i.e., for every point p of $\mathcal{I}$, the value of $targ\_active_r(\mathcal{T}p)$ at the end of the program is equal to the value of $active_r(p)$. We say *targ_active implements* predicate *active*. The value of $targ\_active(\mathcal{T}p)$ is undefined if $\mathcal{T}p$ is not scanned by the target loop nest.

Execution:

- The outermost if clause prevents the body from receiving, calculating and sending signals and messages that are not specified by our signaling scheme of Def. 6, i.e., signals at depth $r' > r$.
- Part A probes signals that are expected and receives those that are actually being sent. Thus, $rcved^r_k(p_1, \cdots, p_r)$ is equal to $sig^r_k(p_1, \cdots, p_k - 1, \cdots, p_r, lb_{r+1}, \cdots, lb_d)$.
- Part B propagates all necessary information along the tooth in direction r. (Remark: this is not necessary in the single-assignment version.)
- Part C implements alternative (4) of Def. 6.
- All but the last line of part D implements alternative (3); the last line implements alternative (2).
- Line E copies the value of k for the next iteration.
- The if clause after line E tests whether the current iteration must be executed with respect to level r (Def. 2) The value of $targ\_active_r$, $sigval_r$ and, if they exist, $targ\_active_{r+1}$ and k_{r+1} depend on the outcome of the test.
- sig^r_k has three conjuncts (Def. 6). The first corresponds to the then or the else branch of said if clause, the second to the calculation at line G. The third conjunct is satisfied by an appropriate setting of k_r in parts F and H.
- *targ_active* implements *active* (Def. 1), as proved in [9].
- Part I applies only for $r = d$. It tests for sig^d_0 (i.e., $sigval_d$ and $k_d = 0$) and, if so, terminates the entire program (compare Lemma 8 and Def. 5).
- Part J sends the data that are received by Part B on to the next point

of the tooth.

- Part K sends sig_k^r, if it has to be sent, i.e., if *sigval* holds. (At present, we ignore signals in direction 0. In the future, we plan to use this information in the development of smarter loop bounds.)

Remark 10. In the synchronous model, we get the same semantics if the postfixes are made prefixes instead. This is an optimization if each processor has a co-processor for the transmission of messages so that computation and message handling can proceed in parallel, as is the case for the transputer [8]. For asynchronous machines, there is no similar optimization since only the receipt of messages can guarantee that the sender has updated all transformed source variables.

7.2 The target program for synchronous and/or shared-memory machines

For a real implementation, we first introduce re-assignments to processor-local variables by a simple transformation:

- all variables that are indexed with $(p_1, \cdots, p_r)$ become scalars, e.g., $\mathit{targ\_active}_r(p_1, \cdots, p_r)$ becomes $\mathit{targ\_active}_r$;
- all variables that are indexed with $(p_1, \cdots, p_r+1)$ become renamed scalars, e.g., $\mathit{targ\_active}_r(p_1, \cdots, p_r+1)$ becomes $\mathit{targ\_active\_out}_r$;
- all variables that are indexed with $(p_1, \cdots, p_r, lb_{r+1})$ become scalars, e.g., $\mathit{targ\_active}(p_1, \cdots, p_r, lb_{r+1})$ becomes $\mathit{targ\_active}_{r+1}$ (note the different index).

Note, that the upper bound d on r is known at compile time.

The more interesting implementation problem is the adaptation to the synchronous or the asynchronous model. As we mentioned already, the signaling scheme is most easily described for synchronous machines. In that case, the given target code is complete and correct.

In the asynchronous case, there is a greater chance for scannability than in the synchronous case. Scannable transformations have a distinct advantage over unscannable transformations: their test for the activity of a point creates no run-time overhead. Thus, the most efficient parallel target code for an asynchronous machine is generated by a scannable transformation. Still, for whatever reasons, one has the option of an unscannable transformation also for the asynchronous model, but with a slightly modified version of the target code just presented.

The modifications result from the fact that, in the asynchronous case, there is no global clock, i.e., the time component of every space-time mapped iteration cannot be interpreted globally. Thus, sending any message from an iteration on processor P at time t to another iteration on processor P' with execution time t' makes no sense—t' might be in the past with respect to the clock of processor P.

We can avoid this problem as follows: instead of the conditional sending of valueless signals, send unconditionally messages carrying the value of the condition, and use the blocking receive for the receipt of these messages. The modified part K is

```
if k_r(p_1, ..., p_r) ≠ 0  then
        asend(chan_S^{k_r}, [sigval_r(p_1, ..., p_r)])
fi
```

For shared-memory systems, parts B and J are superfluous. The exchange of the signals in part A and K, however, must be transformed to shared-memory access.

Example 4. Finally, we present the hand-optimized synchronous target program of Example 1 from Section 5.1. Many parts of the automatically generated target program disappear, mainly because the example is only two-dimensional but also because one of the two loops is a for loop, As stated in the last subsection, $active_2$, w_1^2 and $rcved_1^2$ are scalar variables which are local to the processors:

```
for t := 0 while true do
```

```
parfor p := 0 to n do
    if t = 0 then
        w_1^2 := (p ≠ 0)
        active_2 := true
    fi
    creceive(chan_S^1, [rcved_1^2])
    if active_2 and A[p, t] ≠ ⊥ then
        sum[p] := sum[p] + A[p, t]
    else
        active_2 := false
    fi
    if not w_1^2 then
        if p = n then
            AbortServer()
        else
            asend(chan_S^1, [true])
        fi
    fi
    w_1^2 := w_1^2 and not rcved_1^2
end
end
```

8 Conclusions

Since the appearance of the first paper on the parallelization of recurrence equations [12], the theory of loop parallelization has developed considerably; one big step was the introduction of the theory of polyhedra to loop parallelization by [13] which was later developed and promoted by Quinton [19]. The basic theory has been extended in various directions [15]. The existing systems can be divided into two groups: (1) systolic design systems like Alpha [14] and Presage [23] and (2) parallelizing compilers like Suif [24], Tiny [25], PAF [20] and Bouclettes [3].

None of these systems can treat the general case of loop nests with while loops.

Until recently, very little work was devoted to the parallelization while loops. One approach was pursued by Wu and Lewis [26]: they separate the "loop control" part from the while loop body and try to determine, at compile time, an upper bound for the number of iterations of the while loop. If this is possible, the while loop is actually a disguised for loop and the usual methods of for loop parallelization apply. For while loops which do not correspond to for loops, they and others [22, 21] suggest pipelining, but with methods that cannot go beyond a constant speed-up.

Our approach is not to isolate special cases where while loops can be treated well, but we offer an extension to the theory of parallelizing for loop nests which can treat *any* kind of while loops. The results presented here are currently being implemented in our automatic loop parallelizer LooPo.

We are considering a loop nest that contains while loops and a space-time mapping $\mathcal{T}$ that parallelizes this nest. If $\mathcal{T}$ is unscannable in the technical sense of [11], we cannot scan the image $\mathcal{TX}$ of the execution space $\mathcal{X}$ under $\mathcal{T}$ precisely.

For this case, we have here proposed a signaling scheme with which a target loop nest can scan a finite superset $\mathcal{TS}$ of $\mathcal{TX}$ and distinguish the points of $\mathcal{TX}$ from the points of $\mathcal{TS} \setminus \mathcal{TX}$. The determination of the upper bounds for the target loops that scan $\mathcal{S}$ proceeds by neighboring communications only; after the mapping of $\mathcal{S}$ to $\mathcal{TS}$ communication distances may exceed 1 but are still constant, i.e., independent of the problem size (Explanation: two source index points with distance 1 are distinct in one coordinate, say s, and, thus, their images have as distance vector the space components of column s of $\mathcal{T}$).

The price we have to pay for unscannable space-time mappings is that the necessary scanning of a superset of $\mathcal{TX}$ increases the execution time and/or the amount of processors needed. A more detailed analysis is left to future work. Are unscannable transformations at all useful at such a price? It turns out that they are definitely interesting for synchronous parallelism because, in this case, there are no scannable transformations for a loop nest with an inner while loop.

Our approach is conservative in the sense that the parallel target program does not execute while loop iterations that do not lie on the execution comb. Collard [5] permits the "speculative" execution of iterations that might afterwards be recognized as not belonging to the execution comb. One consequence is that one need not worry about unscannability—but note that some form of distributed termination detection is still required. The speculative approach may lead to superfluous iteration steps but also to increased parallelism, since it amounts to temporarily disregarding some dependences. Loop nests that are not parallelizable in the conservative approach can become parallelizable this way. The conservative approach works well in cases where different while loops do not influence each other's termination—and it is also the appropriate approach for for loop nests which have non-affine upper bounds and which, therefore, cannot be treated with the regular polytope methods. Both the conservative and the speculative approach achieve orders of magnitude of speed-up, since both are based on the polytope model. A more thorough comparison and an example of a conservative parallelization of a loop nest with a while loop can be found in [16].

Acknowledgements

Thanks to Jean-François Collard and Max Geigl for a careful reading and to Max Geigl, in addition, for numerous discussions with the first author on multi-dimensional spaces that led to the definition of *sig*.

Financial support was received from the DFG under Project *RecuR* and through a PROCOPE grant from the DAAD.

References

1. U. Banerjee. *Loop Transformations for Restructuring Compilers: Loop Parallelization.* Kluwer, 1994.
2. M. Barnett and C. Lengauer. Unimodularity and the parallelization of loops. *Parallel Processing Letters*, 2(2–3):273–281, 1992.

3. P. Boulet, M. Dijon, E. Lequiniou, and T. Risset. Reference manual of the bouclettes parallelizer. Technical Report 94-04, LIP, Ecole Normale Supérieure de Lyon, October 1994.
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. 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.
6. J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *Proc. 5th ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP)*. ACM Press, July 1995. To appear. Earlier version: Technical Report TR94-24, LIP, Ecole Normale Supérieure de Lyon, July 1994.
7. E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.
8. I. Graham and T. King. *The Transputer Handbook*. Prentice-Hall, 1990.
9. 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.
10. 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.
11. M. Griebl and C. Lengauer. On the space-time mapping of WHILE-loops. *Parallel Processing Letters*, 4(3):221–232, September 1994.
12. R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, July 1967.
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. C. Lengauer and M. Griebl. On the parallelization of loop nests containing while loops. In N. N. Mirenkov, Q.-P. Gu, S. Peng, and S. Sedukhin, editors, *Proc. 1st Aizu Int. Symp. on Parallel Algorithm/Architecture Synthesis (pAs'95)*, pages 10–18. IEEE Computer Society Press, 1995.
17. A. Martin. The probe: An addition to communication primitives. *Information*

Processing Letters, 20(3):125–130, 1985.

18. Parsytec. *PARIX 1.2 Reference Manual*, March 1993.
19. P. Quinton. The systematic design of systolic arrays. In F. F. Soulié, Y. Robert, and M. Tchuente, editors, *Automata Networks in Computer Science*, chapter 9, pages 229–260. Manchester University Press, 1987. Also: Technical Reports 193 and 216, IRISA (INRIA-Rennes), 1983.
20. M. Raji-Werth and P. Feautrier. On parallel program generation for massively parallel architectures. In M. Durand and F. El Dabaghi, editors, *High Performance Computing II*. North-Holland, 1991.
21. L. Rauchwerger and D. Padua. Parallelizing while loops for multiprocessor systems. In *Proc. 9th Int. Parallel Programming Symposium (IPPS'95)*, pages 347–355. IEEE Computer Society Press, 1995.
22. P. P. Tirumalai, M. Lee, and M. S. Schlansker. Parallelization of while loops on pipelined architectures. *J. Supercomputing*, 5:119–136, 1991.
23. V. van Dongen and M. Petit. PRESAGE: A tool for the parallelization of nested loop programs. In L. J. M. Claesen, editor, *Formal VLSI Specification and Synthesis (VLSI Design Methods-I)*, pages 341–359. North-Holland, 1990.
24. R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. Suif: An infrastructure for research on parallelizing and optimizing compilers. In *Proc. Forth ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 31–37. ACM Press, 1994.
25. M. Wolfe. Experiences with data dependence and loop restructuring in the tiny research tool. Technical Report 90-016, Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, 1990.
26. 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.
27. J. Xue. Automating non-unimodular transformations of loop nests. *Parallel Computing*, 20(5):711–728, May 1994.

List of Figures

This article was processed using the LaTeX macro package with LLNCS style