Loop Parallelization in the Polytope Model*

Christian Lengauer

Fakultät für Mathematik und Informatik, Universität Passau,
D–94030 Passau, Germany. E-mail: lengauer@fmi.uni-passau.de

Abstract. During the course of the last decade, a mathematical model for the parallelization of FOR-loops has become increasingly popular. In this model, a (perfect) nest of r FOR-loops is represented by a convex polytope in $\mathbb{Z}^r$. The boundaries of each loop specify the extent of the polytope in a distinct dimension. Various ways of slicing and segmenting the polytope yield a multitude of guaranteed correct mappings of the loops' operations in space-time. These transformations have a very intuitive interpretation and can be easily quantified and automated due to their mathematical foundation in linear programming and linear algebra.
With the recent availability of massively parallel computers, the idea of loop parallelization is gaining significance, since it promises execution speed-ups of orders of magnitude. The polytope model for loop parallelization has its origin in systolic design, but it applies in more general settings and methods based on it will become a part of future parallelizing compilers. This paper provides an overview and future perspective of the polytope model and methods based on it.

1 Introduction

Fifteen years ago, a first, restrictive form of massive parallelism received a name: *systolic array* [23]. The restrictions, motivated by the desire to simplify and parametrize the process of designing logic circuitry and borne out by the limitations of hardware technology at the time, included a regular processor and interconnection topology, synchronous parallelism, only local point-to-point communication, and input/output only through border processors. In the years since, some of these limitations have been lifted. For example, certain modern programmable processor networks are asynchronously parallel, others permit limited broadcasting.

In the Eighties, a model of computation emerged that makes it possible to synthesize guaranteed correct systolic arrays automatically: the polytope model. It corresponds to a class of particularly uniform programs: nested FOR-loops with regularity restrictions on the loop indices. Its rich mathematical theory of mappings from loops to processor networks permits a detailed classification of systolic arrays with a quantification of their properties (execution time, throughput, amount of processors and communication, etc.) and, therefore, a calculational mechanizable selection of maximum-quality arrays. This and its intuitive geometric interpretation has made this model increasingly popular—and also promising for the synthesis and analysis of more general forms of massive parallelism.

* Corrected version.

The polytope model is a computational model for sequential or parallel programs: a program is modelled by a *polytope*, a finite convex set of some dimensionality with flat surfaces. The data dependences between the points (computations) in the polytope must be regular and local. Drawing the dependences as arrows yields a regular, acyclic, directed graph: the *data dependence graph*. These properties required of the model (convexity, flat surfaces, regularity of the dependences) restrict the set of programs that can be modelled. A set of, say, r perfectly nested loops with a constant-time loop statement, with bounds that are linear expressions in the indices of the enclosing loops and in the problem size, and with certain additional restrictions on the use of the loop indices can be represented by a polytope embedded in $\mathbb{Z}^r$: each loop defines the extent of the polytope in one dimension. When modelling sequential execution, the dimensions are "scanned", one at a time, to enumerate the points in the polytope. For a parallel execution, the polytope that represents the source program—we call it the *source polytope*—is segmented into *time slices*, sets of points that can be executed concurrently. The parallelization methods based on the polytope model address the problem of this segmentation. It can be formulated as an affine mapping that transforms the source polytope into a *target polytope* that contains the same points, but in a new coordinate system in which some dimensions are strictly temporal—i.e., scanning along them enumerates time—and the others are strictly spatial—i.e., scanning along them enumerates space. (Usually, there is only one temporal dimension. If there are several, they can be reduced to one.) This transformed polytope can then be used to synthesize chip layouts—the spatial dimensions define a processor layout—or parallel loop programs.

In principle, the synthesis process takes as input an algorithmic description that does not specify concurrency or communication—usually a single-assignment or imperative program—and maps the program's operations to space-time. The mapping preserves the data dependences in the program and is usually chosen to fulfill some efficiency criteria—a typical one is execution time minimality with respect to the choices possible in the model. The basis for an automatic synthesis with the polytope model was laid in the Sixties by the seminal paper of Karp/Miller/Winograd on uniform recurrences [21]. In the Seventies, Lamport was the first to apply this approach to the question of parallelizing compilation [24]. However, only in the early Eighties, after the birth of the systolic array, was the idea picked up and developed further. Significant contributions were made by Kuhn [22], Moldovan [32], Cappello/Steiglitz [8], Miranker/Winkler [31] and Quinton [36]. The dissertation of Rao in 1985 unified these results in a theory for the automatic synthesis of all systolic arrays [42, 43].

The polytope model is increasingly being recognized as useful for parallelizing loop programs for massively parallel architectures. Note that the parallelization methods based on this model are static, i.e., derive all parallelism before run time (at compile time). This has the advantage that no overhead of the discovery of parallelism is introduced at run time. However, one must still take care to avoid overhead for the administration of parallelism; for more, see Section 3.2.

The benefit of the polytope model is that, at the price of certain regularity conditions, a very high standard of parallelization is made possible: an automatic or, in less regular cases, interactive mapping of programs or specifications to massively parallel processor networks. This mapping is guaranteed to preserve the desired behaviour of the source program (or specification) and to yield massively parallel solutions that are quantifiable, i.e., whose properties like execution time, throughput, number of processors and communication channels, memory requirements, etc. can be stated precisely and compared. The synthesis can also be geared towards maximizing efficiency with respect to a given criterion like any of the requirements just mentioned. With the possibilities promised by new hardware techniques and computer architectures (increasing communication bandwidths and distances, decreasing communication cost, asynchronous parallelism in programmable processor arrays), the polytope method is undergoing extensions to encompass less restricted regular processor arrays, and more and more general source programs can be parallelized this way. However, the polytope model and the methods based on it are, by nature, restricted to nested loop programs. If a parallelization method is to be restricted, nested loop programs are a fruitful domain to aim at, because their parallelization offers the possibility of an order-of-magnitude speed-up.

The concepts of the polytope model are more explicit in single-assignment programs than in imperative programs; imperative nested loop programs can be transformed to single-assignment programs [5, 17]. In a single-assignment format, algorithms that are now well understood in the polytope model can be described by a set of *recurrence equations*, each of the form:

$$(\forall\ x \in \mathcal{IS} : v[f(x)] = \mathcal{F}_v(w[g(x)], \ldots)) \qquad (*)$$

The three dots in the argument list of the strict function $\mathcal{F}_v$ stand for an arbitrary but fixed number of similar arguments; v and w are indexed variables. The restrictions imposed on the form of the *index functions* f and g and on the shape and properties of the *index space* $\mathcal{IS}$ were initially severe and became successively weaker:

1. *Uniform recurrence equations.* In this initial version, the index domain $\mathcal{IS}$ is the intersection of a polytope with $\mathbb{Z}^r$, where r is the number of recurrences (or nested loops), and the indexing functions $f(x)$ and $g(x)$ are of the form $x+d$, where d is a constant vector [21]. This enforces that data dependences between points in the polytope are point-to-point (x) and local (d).
2. *Affine recurrence equations.* Here, the index functions are of the more general form $A\,x+d$, where A is a constant matrix and d is a constant vector [38, 39]. This permits data sharing (if A is singular, i.e., defines a non-injective mapping).
3. *Piecewise linear/regular algorithms.* Here, one permits the index domain to be not convex but partitioned into convex polytopes [12, 46]. This permits a sequence of perfect loop nests instead of just one perfect loop nest.
4. *Linearly bounded lattices.* Here, one does not embed into $\mathbb{Z}^r$ but instead into an integral affine transformation of it that need not be convex [50]. The intersection of a polytope with an integer lattice is one special case. This enables the imposition of resource limitations, i.e., a *partitioning* of the parallel solution.

2 An Illustration

We present the polytope model and method with a simple example: polynomial product. We specify the problem (Section 2.1), present a source program (Section 2.2), present the source polytope (Section 2.3), transform it into a target polytope (Section 2.4), derive one synchronous and one asynchronous parallel program from it (Section 2.5), and touch on a couple of variations (Section 2.6).

2.1 Specification

We use Dijkstra's quantifier format [14]. Let $\mathcal{A}$ and $\mathcal{B}$ be two polynomials of degree n in variable z, i.e.:

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

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

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

where:

$$(\forall\, k : 0 \le k \le 2\,n : c[k] \;=\; (\sum i,j : 0 \le i \le n \wedge 0 \le j \le n \wedge i+j=k : a[i]\, b[j]))$$

2.2 A Source Program

We provide two programs that compute the coefficients $c[k]$ from $a[i]$ and $b[j]$: one imperative, the other single-assignment.

In the imperative program, we denote the loop statement with its vector (i,j) of loop indices:

$$\left.\begin{array}{l} \textbf{for } i \;=\; 0 \leftarrow 1 \rightarrow n \\ \quad \textbf{for } j \;=\; 0 \leftarrow -1 \rightarrow n \\ \qquad (i,j) \end{array}\right\} \text{with} \left\{\begin{array}{l} (i,j): \\ \quad \textbf{if } i=0 \vee j=n \rightarrow c[i+j] := a[i]\, b[j] \\ \quad [\,]\;\; i \neq 0 \wedge j \neq n \rightarrow c[i+j] := c[i+j] + a[i]\, b[j] \\ \quad \textbf{fi} \end{array}\right.$$

Here, the outer loop increments its index from 0 to n, while the inner loop decrements its index from n to 0 [4].

In the single-assignment program—in this case, it is a set of affine recurrence equations—we must use an auxiliary variable, say, C with an additional index to keep track of successive updates of c:

$$\begin{array}{lclcl} (\forall\, j & : & 0 \le j \le n & : & C[0,j] \;=\; a[0]\, b[j]) \\ (\forall\, i & : & 0 \le i \le n & : & C[i,n] \;=\; a[i]\, b[n]) \\ (\forall\, i,j & : & 0 < i \le n \wedge 0 \le j < n & : & C[i,j] \;=\; C[i-1,j+1] + a[i]\, b[j]) \\ (\forall\, k & : & 0 \le k \le n & : & c[k] \;=\; C[k,0]) \\ (\forall\, k & : & n \le k \le 2\,n & : & c[k] \;=\; C[n,k-n]) \end{array}$$

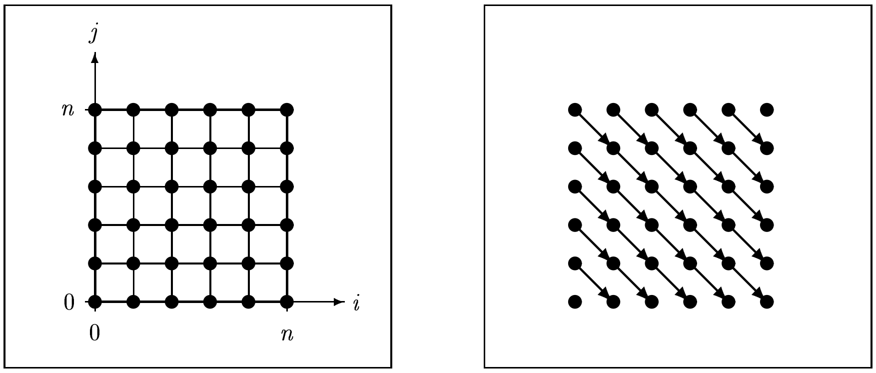

Fig. 1. The index space (left) and the dependence graph (right).

2.3 The Source Polytope

Figure 1 depicts the square polytope and the dependence graph. The only dependences between different points of the polytope are introduced by the "pipelining" of computations of C. Copies of the values of a and b can be made available at every point that needs them.

In linear programming, a polytope is described by a set of inequations. Each inequation defines a *halfspace* of $\mathbb{Z}^r$; the intersection of all halfspaces is the polytope. The points of our index space $\mathcal{IS}$ satisfy four linear inequations:

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \\ n \\ n \end{bmatrix} \qquad \begin{array}{rr} \text{left border} & \text{(lower bound of } i) \\ \text{lower border} & \text{(lower bound of } j) \\ \text{right border} & \text{(upper bound of } i) \\ \text{upper border} & \text{(upper bound of } j) \end{array} \qquad \begin{array}{r} i \geq 0 \\ j \geq 0 \\ i \leq n \\ j \leq n \end{array}$$

We denote the polytope defined by the set of inequations $A\,x \leq b$ by (A, b). The upper part of A is calculated from the lower bounds, the lower part from the upper bounds of the loops [4]. Both parts can be made lower-triangular, since the bounds of any loop must depend on the indices of enclosing loops only (they may also depend on the problem size).

2.4 A Target Polytope

The success of the polytope model is rooted in the existence of a mathematical theory with which correctness-preserving space-time transformations of index spaces can be quantified and selected according to given efficiency criteria.

Let $\mathcal{IS}$ be an r-dimensional polytope and consider the dependence graph $(\mathcal{IS}, E)$, where E is the edge set defining the dependences.

– Function $t : \mathcal{IS} \to \mathbb{Z}$ is called a *schedule* if it preserves the data dependences:

$$(\forall\ x, x' : x, x' \in \mathcal{IS} \wedge (x, x') \in E : t(x) < t(x'))$$

The schedule that maps every $x \in \mathcal{IS}$ to the first possible time step allowed by the dependences is called the *free schedule*.

– Function $a : \mathcal{IS} \rightarrow \mathbb{Z}^{r-1}$ is called an *allocation* with respect to schedule t if each process it defines is internally sequential:

$$(\forall\ x, x' : x, x' \in \mathcal{IS} : t(x) = t(x') \Rightarrow a(x) \neq a(x'))$$

This is the *full-dimensional* case, in which space takes up r–1 dimensions and time the remaining one dimension of the target polytope. One can also trade dimensions from space to time [27] and reduce multi-dimensional time to one dimension [42], ending up with fewer dimensions on the target than on the source side. A full-dimensional solution offers a maximum speed-up.

Most parallelization methods based in the polytope model require the schedule and allocation to be affine functions:

$$\begin{array}{llll}(\exists\ \lambda, \alpha : \lambda \in \mathbb{Z}^r & \wedge\ \alpha \in \mathbb{Z} & : (\forall\ x : x \in \mathcal{IS} : t(x) & = \lambda\, x + \alpha)) \\ (\exists\ \sigma, \beta : \sigma \in \mathbb{Z}^{r-1} \times \mathbb{Z}^r & \wedge\ \beta \in \mathbb{Z}^{r-1} & : (\forall\ x : x \in \mathcal{IS} : a(x) & = \sigma\, x + \beta))\end{array}$$

The matrix T formed by λ and σ is called a *space-time mapping*:

$$T \;=\; \begin{bmatrix} \lambda \\ \sigma \end{bmatrix}$$

In the full-dimensional case, T is a square matrix and the requirement on the allocation is: $|\,T\,| \neq 0$. We call $T(\mathcal{IS})$ the *target space*.

The challenge is to find not only valid but also good-quality schedules and allocations—if possible, automatically. Herein lies the strength of the polytope method. Integer linear programming methods [34] can always determine a time-minimal schedule. A matching allocation is easily selected by checking the determinant with λ for different σ. One can impose optimization criteria like a minimal number of processors or channels, maximal throughput, etc. Note that the claim of optimality is relative to:

1. the data dependences of the source program (which might restrict parallelism needlessly) and
2. the requirements on the space-time mapping (here affinity).

Let us consider our example. Integer linear programming reveals that one time-minimal schedule is $\begin{bmatrix} 1\ 0 \end{bmatrix}$; a consistent processor-minimal allocation is $\begin{bmatrix} 0\ 1 \end{bmatrix}$. This makes the space-time mapping the identity—an example of the trivial case in which the space-time mapping is simply a permutation of the source coordinates and the source loop nest can be parallelized directly [56, 60]. To make our illustration more interesting, we choose a different allocation: $\begin{bmatrix} 1\ 1 \end{bmatrix}$. It is not processor-minimal. Figure 2 depicts the according segmentations of the index space.

The schedule slices the index space into parallel *hyperplanes*, i.e., subspaces whose dimensionality is one less than that of the index space. Each hyperplane represents one *time slice*. (This terminology suggests synchronous parallelism, but there is no need for insisting on synchrony in the implementation.) The definition of the schedule prescribes that the hyperplanes be not parallel to any edge of the dependence graph.

The allocation segments the index space into parallel *lines*, i.e., subspaces of dimensionality 1. Each line contains the points executed by a fixed processor. The

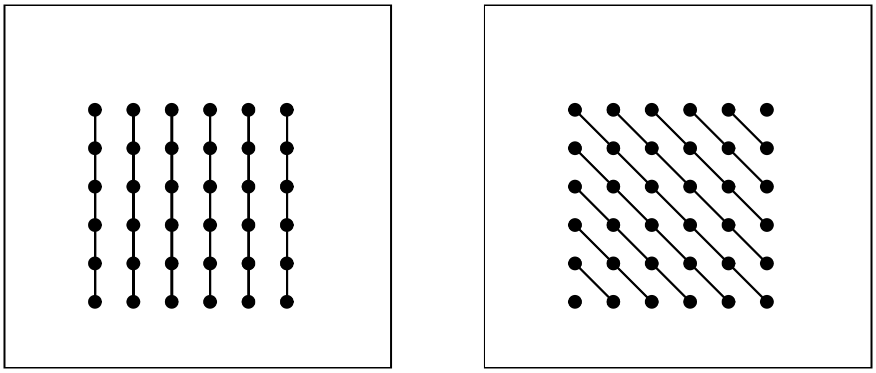

Fig. 2. The index space partitioned by a minimal affine schedule (left) and a matching allocation (right).

consistency requirement for schedule and allocation prescribes that the lines generated by the allocation be not parallel to the hyperplanes generated by the schedule.

The most concise description of the parallel solution is the *iteration vector* [42, 43]. This is the vector u that is parallel to the allocation lines (which are orthogonal to the allocation: $\sigma\, u = 0$), that points into the direction of the computation ($\lambda\, u > 0$), and whose length is the distance between adjacent points on an allocation line.

In the case of polynomial product, the hyperplanes are also lines because $r = 2$. There are $n+1$ times slices. We need a one-dimensional processor layout; our particular allocation specifies a row of $2\,n{+}1$ processors ($n{+}1$ is minimal). The lines induced by our allocation happen to coincide with the dependences of C. That is, each C-element is only needed at one processor. Elements of A and B are needed at more than one processor. The iteration vector is $\begin{bmatrix} 1 & -1 \end{bmatrix}$.

Since the space-time mapping is invertible, the target polytope defined by it can be described very easily. Say, the space-time mapping is T and the source polytope is (A, b) in the coordinate system of $\mathcal{IS}$. The target polytope must contain the same points as the source polytope but in a different coordinate system, namely, in space-time coordinates:

$$\begin{array}{rl} & \{\; A\,x \le b \;\} \\ = & \{\; T\,x = y \;\} \\ & (A\,T^{-1})\,y \le b \end{array}$$

If T is unimodular, i.e., the matrix of T^{-1} has only integer elements, the target polytope is precisely the target space: $(A\,T^{-1}, b) = T(\mathcal{IS})$. If T is not unimodular, i.e., the matrix of T^{-1} has rational elements, the target polytope is only the convex hull of the target space: $(A\,T^{-1}, b) \supset T(\mathcal{IS})$. Some parallelization methods based on the polytope model exclude non-unimodular transformations [26, 44], but this is overly restrictive. There are ways of treating non-unimodularity (see Section 3.2).

Consider our space-time mapping and its inverse:

$$T = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad\qquad T^{-1} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$$

In the coordinate system on the target side, we choose to denote the time coordinate with t and the space coordinate with p (for "processor"). The set of inequations for the target polytope under T is:

$$\begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} t \\ p \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \\ n \\ n \end{bmatrix} \qquad \begin{array}{rl} \text{left border} & t \geq 0 \\ \text{lower border} & p \geq t \\ \text{right border} & t \leq n \\ \text{upper border} & p \leq t+n \end{array}$$

Figure 3 depicts the target polytope.

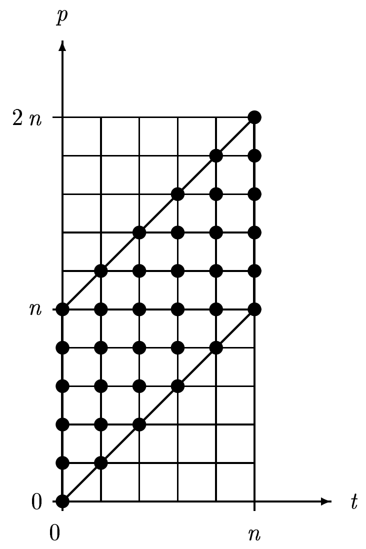

Fig. 3. The target polytope for the affine recurrence equations.

2.5 Two Target Programs

The last step in the parallelization is to transform the target polytope back into a loop program. In the full-dimensional case, the schedule dimension becomes a sequential FOR-loop, the allocation dimensions become parallel FOR-loops.

Before we can proceed with the transformation, we must choose the order of the loop nest of the target program. If we make the schedule dimension the outer-most loop, we obtain synchronous parallelism. If we make it the inner-most loop, we obtain asynchronous parallelism.

The synchronous target program. The synchronous program can be directly read off from the inequations for the target polytope. We use the occam keywords **seq** and **par** for the temporal and spatial loop, respectively. Note that the processor layout must be determined at every time step—and so must the termination of all processors (*barrier synchronization* [55]):

$$
\begin{array}{l}
\textbf{seq } t \;=\; 0 \leftarrow 1 \rightarrow n \\
\quad \textbf{par } p \;=\; t \leftarrow 1 \rightarrow t+n \\
\qquad (t, p-t)
\end{array}
$$

The loop statement specifies the points of the source polytope but in the coordinates of the target polytope. This correspondence is defined by the inverse of T:

$$
\begin{bmatrix} i \\ j \end{bmatrix} \;=\; T^{-1} \begin{bmatrix} t \\ p \end{bmatrix} \;=\; \begin{bmatrix} t \\ p-t \end{bmatrix}
$$

The asynchronous target program. The inequations for the target polytope may not be suitable for the creation of loops: all inequalities may depend on all new loop indices—in our case, t and p—whereas loop bounds must depend on indices of enclosing loops only. Here, this problem crops up when we want to synthesize an asynchronous program: the bounds of the outer loop index p are given with respect to the inner loop index t.

We must transform the polytope $(A\,T^{-1}, b)$ to an equivalent one, (A', b') whose defining inequations refer only to the indices of enclosing loops. Several algorithms for calculating A' and b' have been proposed [1, 16, 44, 54]; most of them are based on Fourier-Motzkin elimination [47], a technique by which one eliminates variables from the inequalities. Geometrically, this projects the polytope on the different axes of the target coordinate system to obtain the bounds in each dimension.

To bound the target polytope only in the allocation dimension, we add the horizontal halfspaces $p \geq 0$ and $p \leq 2\,n$. Then we sort our six inequations such that the first three define lower loop bounds and the last three upper loop bounds:

$$
\begin{bmatrix} 0 & -1 \\ -1 & 0 \\ -1 & 1 \\ 0 & 1 \\ 1 & 0 \\ 1 & -1 \end{bmatrix}
\begin{bmatrix} t \\ p \end{bmatrix} \;\leq\;
\begin{bmatrix} 0 \\ 0 \\ n \\ 2\,n \\ n \\ 0 \end{bmatrix}
\qquad
\begin{array}{ll}
\text{lower bound on } p & p \geq 0 \\
\text{lower bound on } t & t \geq 0 \\
\text{lower bound on } t & t \geq p-n \\
\text{upper bound on } p & p \leq 2\,n \\
\text{upper bound on } t & t \leq n \\
\text{upper bound on } t & t \leq p
\end{array}
$$

In the program, multiple lower bounds are combined with the max function, multiple upper bounds with the min function:

$$
\begin{array}{l}
\textbf{par } p \;=\; 0 \leftarrow 1 \rightarrow 2\,n \\
\quad \textbf{seq } t \;=\; \max(0, p-n) \leftarrow 1 \rightarrow \min(n, p) \\
\qquad (t, p-t)
\end{array}
$$

Here, the processor layout is determined only once. However, in the absence of synchrony, the data dependences must be enforced some other way. In a distributed-memory implementation, this is imposed by channel communications; in a shared-memory implementation, one must take other measures.

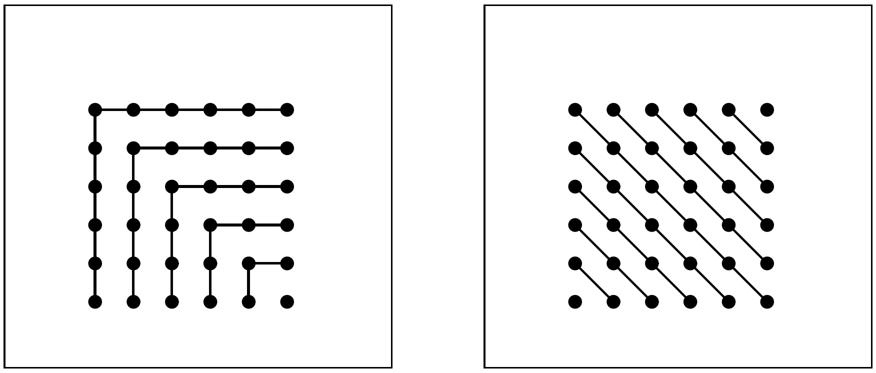

Fig. 4. The index space segmented by the free schedule of the affine recurrence equations (left) and by the previous allocation, which is now processor-minimal (right).

2.6 Other Schedules

The free schedule. If we schedule every point as early as possible, given the dependence graph of Figure 1, we obtain the time slices depicted in Figure 4.

The hyperplanes are creased, i.e., the free schedule is only piecewise affine:

$$t(\begin{bmatrix} i \\ j \end{bmatrix}) \;=\; \begin{cases} \textbf{if} \;\; i{+}j \leq n \rightarrow i \\ [\,] \;\; i{+}j \geq n \rightarrow n{-}j \\ \textbf{fi} \end{cases}$$

This schedule has the same length as the minimal affine schedule, but it requires double as many processors and provides less load balance: the time slices are not of equal length. So it is not of practical interest. But, in general, it pays to consider piecewise affinity. The free schedule is often piecewise affine and shorter than the minimal affine schedule. Piecewise affinity is also used in allocations to fold processor layouts for better processor usage [7, 12].

Localization. In a distributed-memory implementation, one might prefer to avoid multiple copies of a variable, e.g., to reduce storage requirements or, if the variable is updated, prevent inconsistencies. To achieve this, one can pipeline shared variable accesses, which results in additional, artificial dependences. This process is called *localization* [25]—or, if all shared variables of the program are being pipelined, also *uniformization*, since then the resulting set of recurrence equations is uniform [38].

We uniformize the recurrence equations for polynomial product by adding auxiliary variables A and B to pipeline the accesses of a and b:

$$\begin{array}{llllll}
(\forall\; i & : & 0 \leq i \leq n & : & A[i,n] & = \; a[i]) \\
(\forall\; i,j & : & 0 \leq i \leq n \wedge 0 \leq j < n & : & A[i,j] & = \; A[i,j{+}1]) \\
(\forall\; j & : & 0 \leq j \leq n & : & B[0,j] & = \; b[j]) \\
(\forall\; i,j & : & 0 < i \leq n \wedge 0 \leq j \leq n & : & B[i,j] & = \; B[i{-}1,j]) \\
(\forall\; j & : & 0 \leq j \leq n & : & C[0,j] & = \; A[0,j]\,B[0,j]) \\
(\forall\; i & : & 0 \leq i \leq n & : & C[i,n] & = \; A[i,n]\,B[i,n]) \\
(\forall\; i,j & : & 0 < i \leq n \wedge 0 \leq j < n & : & C[i,j] & = \; C[i{-}1,j{+}1] + A[i,j{+}1]\,B[i{-}1,j]) \\
(\forall\; k & : & 0 \leq k \leq n & : & c[k] & = \; C[k,0]) \\
(\forall\; k & : & n \leq k \leq 2\,n & : & c[k] & = \; C[n,k{-}n])
\end{array}$$

The new dependence graph, on the old index space, is depicted in the left part of Figure 5. The dependences of A point down, those of B point to the right.

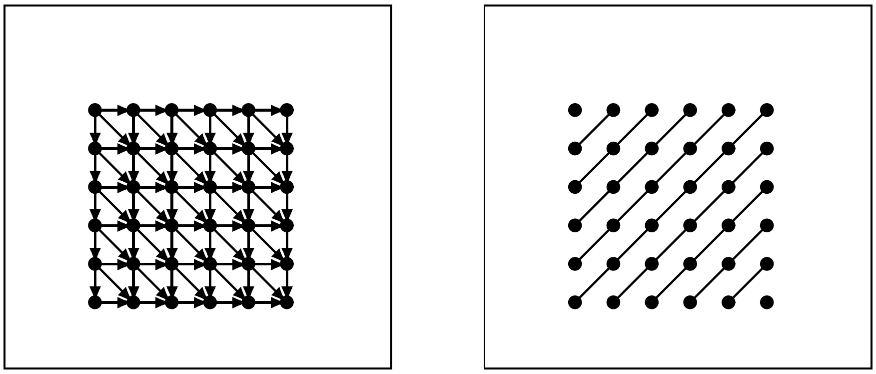

Fig. 5. The dependence graph of the uniform recurrence equations (left) and the time slices of a minimal affine schedule (right).

The additional dependences reduce the potential for parallelism. The new minimal affine schedule is $\begin{bmatrix} 1 & -1 \end{bmatrix}$; it specifies $2\,n+1$ time slices (see the right part of Figure 5). The allocation and the iteration vector can remain unchanged. This makes the space-time mapping and its inverse:

$$T \;=\; \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} \qquad\qquad T^{-1} \;=\; \begin{bmatrix} 1/2 & 1/2 \\ -1/2 & 1/2 \end{bmatrix}$$

The set of inequations for the target polytope under T is:

$$\begin{bmatrix} -1/2 & -1/2 \\ 1/2 & -1/2 \\ 1/2 & 1/2 \\ -1/2 & 1/2 \end{bmatrix} \begin{bmatrix} t \\ p \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \\ n \\ n \end{bmatrix} \quad \begin{array}{rr} \text{left lower border} & 0 \leq t/2+p/2 \\ \text{right lower border} & 0 \leq p/2-t/2 \\ \text{right upper border} & t/2+p/2 \leq n \\ \text{left upper border} & p/2-t/2 \leq n \end{array}$$

Figure 6 displays the resulting target polytope. Note that not all integer-valued points correspond to a computation.

Let us obtain the asynchronous program for this target polytope. To bound the polytope only in the allocation dimension, we add the horizontal halfspaces $p \geq 0$ and $p \leq 2\,n$. After sorting the inequations, we normalize each inequation for the coordinate to be bounded:

$$\begin{bmatrix} 0 & -1 \\ -1 & -1 \\ -1 & 1 \\ 0 & 1 \\ 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} t \\ p \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \\ 2\,n \\ 2\,n \\ 0 \\ 2\,n \end{bmatrix} \quad \begin{array}{ll} \text{lower bound on } p & p \geq 0 \\ \text{lower bound on } t & t \geq -p \\ \text{lower bound on } t & t \geq p-2\,n \\ \text{upper bound on } p & p \leq 2\,n \\ \text{upper bound on } t & t \leq p \\ \text{upper bound on } t & t \leq 2\,n-p \end{array}$$

Here, the correspondence between source and target coordinates is as follows:

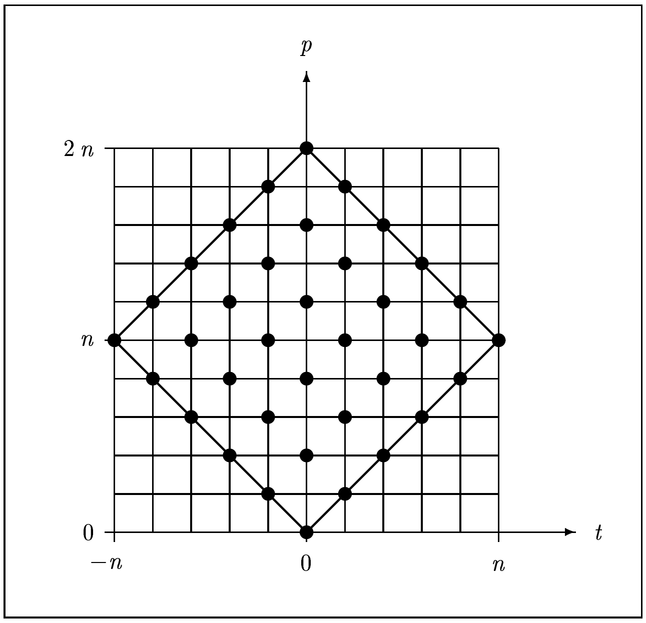

Fig. 6. The target polytope for the uniform recurrence equations.

$$\begin{bmatrix} i \\ j \end{bmatrix} = T^{-1} \begin{bmatrix} t \\ p \end{bmatrix} = \begin{bmatrix} (t+p)/2 \\ (p-t)/2 \end{bmatrix}$$

In the target program, the inner loop step of 2 accounts for the "holes" in the target polytope:

$$\begin{array}{l} \textbf{par}\ p\ =\ 0 \leftarrow 1 \rightarrow 2\,n \\ \quad \textbf{seq}\ t\ =\ \max(-p, p-2\,n) \leftarrow 2 \rightarrow \min(p, 2\,n-p) \\ \qquad ((t+p)/2, (p-t)/2) \end{array}$$

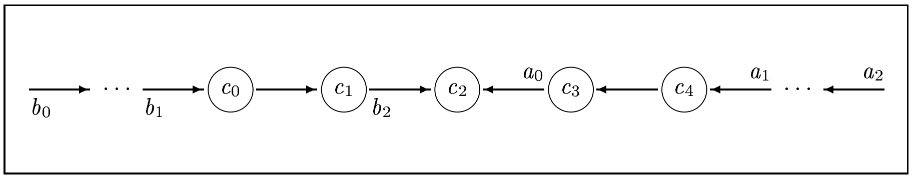

Fig. 7. The systolic array based on the uniform recurrence equations.

The distributed-memory version of this program requires no multiple copies of a- or b-elements. Moreover, the propagation of the corresponding variables A and B requires only connections between neighbouring processors. This is the systolic principle; the processor array for size $n=2$ is depicted in Figure 7. Circles represent the five processors, arrows represent unbuffered communication channels. The variables are distributed as they are needed at the first time slice (in which only the processor in the middle is active). All elements in the figure, i.e., the location of the processors and data and the propagation direction and speed of the data are derivable from the space-time mapping [37].

Note that we have to be judicious in the choice of the dependences that we introduce in a localization; it influences the amount of potential parallelism that is retained. E.g., when A is pipelined up rather than down in the dependence graph of Figure 5, the minimal affine schedule has $3\,n+1$ time slices.

3 Recent Results and Current Research

This section provides some pointers to on-going research. No claim of completeness is made.

3.1 Parallelizers

Over a dozen systems, that are in some form or other related to the methods that we have described, have reportedly been implemented at different sites during the last decade. We shall only mention some modern representatives.

ALPHA. One of the most comprehensive implementations of the polytope model and method is the language ALPHA developed at IRISA, Rennes [26]. In ALPHA, one can specify affine and uniform recurrences directly and perform automatic transformations of the kind we have described. ALPHA has a denotational semantics and can be viewed as a higher-order functional language for the special purpose of recurrence parallelization. For this reason, reduction operations have received special consideration in the design of ALPHA [25].

Crystal. The Crystal project at Yale University started out in the mid-Eighties with a strong orientation towards the polytope model [10], but has since evolved away from it [11]. Now, Crystal is based on a more general equational theory [9]. This permits the treatment of more general source programs at the expense of automation.

HiFi. HiFi is being developed at Delft University of Technology. It is an environment for real-world applications in algorithm development and architecture design [51]. The application domain for which the system lends most support at present covers models and methods for the design of regular architectures. In this domain, the polytope model is used extensively [19].

PAF. This is the automatic FORTRAN parallelizer of Paul (A.) Feautrier [16]. The system converts nested DO-loops to single-assignment form [17] and uses parametric integer programming [15] to find a time-minimal, not necessarily affine, shared-memory parallel schedule. The source loops need not be perfectly nested.

Presage. The novelty of Presage [53] was that it dealt with violations of affinity in the polytope model—in particular with quasi-affinity [52]. A quasi-affine mapping is obtained by taking an affine mapping and coercing rational to integers. Quasi-affine space-time mappings enable a full trading of time versus space and can lead to shorter schedules (with more processors) or fewer processors (with longer schedules) than affine mappings [36].

The Barnett compiler. This parallelizer generates distributed-memory parallel code from nested FOR-loops. The source program is first fed through a systolizer to obtain the target polytope. Both source program and target polytope are then passed on to a code generator that produces a machine-independent distributed program [4]. This program can then be translated to a target language, e.g., occam 2 [35].

3.2 Increasing Parallel Code Efficiency

We said already in the introduction that parallelism modelled by polytopes is static. That is, the discovery of parallelism incurs no run-time overhead. We mentioned that two choices can have a large impact on the efficiency of the parallel implementation: (1) the choice of recurrence equations (or loops), and (2) the choice of space-time mapping. In the latter, one has to choose one's optimization criterion appropriately. E.g., if communications are very expensive, communication optimization takes precedence over time or processor minimization.

In this subsection, we are not concerned with these choices—although they are a central issue—but with the run-time inefficiencies that can occur in the parallel loop program *after* the recurrence equations and the space-time mapping have been chosen. These inefficiencies stem from the fact that one must determine "where" in the index space or target space one is when executing the target loops.

Efficient target loops. One problem is to establish the boundaries of the target polytope. We used min and max functions in the inner loop of most of our programs. In general, the loop boundaries tend to contain bigger and bigger case analyses for loops that are increasingly deep inside the loop nest. Also, we observed that, when the space-time mapping is not unimodular, holes appear in the target polytope. Holes that are "inside" the polytope (see Figure 6) can be bridged by non-unit loop steps but holes at the boundaries require further case analyses in the boundary computation [2, 3, 30].

The aim of parallelizing compilation is to avoid the run-time case analyses by making the respective decisions before run time. One idea that researchers are working on at present is to segment the target space and generate one nest of loops for each segment, rather than generating just one nest of loops and testing at run time for the segment that is being scanned. This is, in general, a very complex problem.

Control signals. A similar problem arises when several distinct operations must be applied at one point at different times of the execution. Then, one must be aware, at each point in time, which is the current operation to be performed. This amounts to a segmentation of the index space. Again, one would like to pay as little as possible at run time for the determination of the segment that one is scanning. Here, one approach has been to make the respective case analyses as cheap as possible by supplying, in parallel with the data streams, additional streams of control signals that specify the operation. This way, the run-time tests reduce to tests for equality with a constant value. There are automatic methods for the synthesis of such control signals [48, 58, 59].

3.3 Model Extensions

It seems natural to include parallelization methods based on the polytope model into compilers for massively parallel computers like the Connection Machine, the Maspar or transputer networks. Massive parallelism is only feasible if it contains some amount of regularity. The requirements of regularity imposed by the polytope model as presented here are more severe than need be. We have already discussed some ideas for extensions of the model that allow for local violations of regularity and thereby increase the amount programming problems for which competitively efficient solutions can be obtained. Much research under way is on extensions, in order to make the model feasible as an ingredient for parallelizing compilers.

More general recurrence patterns. The trend is towards extending the polytope model to more general classes of recurrences: more general loop patterns (nesting violations, less regular variable indexing, etc.), index domains whose extent is not completely determined at compile time, or data dependences that are not completely specified at compile time.

Resource constraints. There is also work on the imposition of realistic resource constraints. The first resource constraints studied were limitations of the dimensionality of the processor layout (*projection*, e.g., [27, 43, 57]) or the number of processors in a dimension (*partitioning*, e.g., [6, 13, 33, 49]). A different reason for reducing the number of processors is to reduce the number of communications (*data alignment*, e.g., [20, 29, 41, 45, 60]). Other constraints recently considered are on the cache size and bus or communication bandwidth.

More general communication patterns. Until recently, distributed computers offered local point-to-point communication as specified with send and receive primitives. New massively parallel computers provide special facilities for, say, broadcasting information to some subset of processors or for remote point-to-point communication. Taking advantage of these facilities in the send/receive paradigm may force a serious reorganization of the source program in order to achieve an efficient mapping onto the architecture. New target language features are needed that can express a variety of communication patterns directly (e.g., [40]). The polytope model needs to be extended to accommodate methods for a compilation of these patterns.

4 Conclusions

There are many different models for parallel computation: Petri nets, process algebras, modal logics, partial order models, etc. The polytope model is distinguished by the fact that it serves specifically to support parallelization methods.

This illustration was intended to demonstrate that parallelization in the polytope model is not only possible but also very convenient. A territory of choices that clearly reaches beyond the systolic principle can be characterized and quantified precisely. Given an optimization criterion—in the illustration it was exclusively the minimization of execution time—an optimal choice within this territory can be made automatically or with minimal human interaction. Choices that are typically made by the human include the additional dependences in a localization or in the synthesis of control signals.

The presence of a computational model is in contrast to traditional loop parallelization, which works directly on the source program causing a bias of certain mappings. The polytope model represents the source program in a formal setting in which no valid mappings are discouraged or excluded. (To be fair, one should add that the principle it is based on—integer programming—is NP-complete in the worst case [18].) The polytope model provides a sound basis for the synthesis of largely regular parallelism and should be one ingredient in a parallelizing compiler—beside other components that address the parallelization of non-repetitive or too irregular code. One approach of the practitioner might be to transform a too irregular repetitive program first into a sufficiently regular one or break it into regular pieces that fit the model (for an example, see [28]). As the model is being extended, such adaptations should become less elaborate.

Acknowledgements

Thanks to Mike Barnett, Hervé Le Verge, Martin Griebl and Jan van de Snepscheut for discussions of the manuscript. Thanks to Lothar Thiele, Patrice Quinton, Ed Deprettere and Vincent van Dongen for discussions of loop parallelization.

References

1. C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proc. 3rd ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 39–50. ACM Press, 1991.
2. M. Barnett and C. Lengauer. Unimodularity and the parallelization of loops. *Parallel Processing Letters*, 2(2–3):273–281, 1992.
3. M. Barnett and C. Lengauer. Unimodularity considered non-essential (extended abstract). In L. Bougé, M. Cosnard, Y. Robert, and D. Trystram, editors, *Parallel Processing: CONPAR 92 – VAPP V*, Lecture Notes in Computer Science 634, pages 659–664. Springer-Verlag, 1992.
4. M. Barnett and C. Lengauer. A systolizing compilation scheme for nested loops with linear bounds. In P. E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, Lecture Notes in Computer Science 693, pages 374–398. Springer-Verlag, 1993.
5. J. Bu. *Systematic Design of Regular VLSI Processor Arrays*. PhD thesis, Department of Electrical Engineering, Delft University of Technology, May 1990.
6. J. Bu and E. F. Deprettere. Processor clustering for the design of optimal fixed-size systolic arrays. In E. F. Deprettere and A.-J. van der Veen, editors, *Algorithms and Parallel VLSI-Architectures*, volume A, pages 341–362. Elsevier (North-Holland), 1991.
7. P. R. Cappello. A processor-time-minimal systolic array for cubical mesh algorithms. *IEEE Trans. on Parallel and Distributed Systems*, 3(1):4–13, January 1992.
8. P. R. Cappello and K. Steiglitz. Unifying VLSI array design with linear transformations of space-time. In F. P. Preparata, editor, *Advances in Computing Research, Vol. 2: VLSI Theory*, pages 23–65. JAI Press, 1984.

9. M. Chen, Y. Choo, and J. Li. Crystal: Theory and pragmatics of generating efficient parallel code. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 7. ACM Press, 1991.
10. M. C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *J. Parallel and Distributed Computing*, 3(4):461–491, 1986.
11. M. C. Chen, Y. Choo, and J. Li. Compiling parallel programs by optimizing performance. *J. Supercomputing*, 2:171–207, 1988.
12. P. Clauss, C. Mongenet, and G. R. Perrin. Calculus of space-optimal mappings of systolic algorithms on processor arrays. *J. VLSI Signal Processing*, 4(1):27–36, February 1992.
13. A. Darte. Regular partitioning for synthesizing fixed-size systolic arrays. *INTEGRATION*, 12(3):293–304, December 1991.
14. E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science. Springer-Verlag, 1990.
15. P. Feautrier. Parametric integer programming. *Operations Research*, 22(3):243–268, 1988.
16. P. Feautrier. Semantical analysis and mathematical programming. In M. Cosnard, Y. Robert, P. Quinton, and M. Raynal, editors, *Parallel & Distributed Algorithms*, pages 309–320. North-Holland, 1989.
17. P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–53, February 1991.
18. M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman & Co., 1979.
19. P. Held and E. F. Deprettere. HiFi: From parallel algorithm to fixed-size VLSI processor array. In F. Catthoor and L. Svensson, editors, *Application-Driven Architecture Synthesis*, pages 71–92. Kluwer, 1993.
20. C.-H. Huang and P. Sadayappan. Communication-free hyperplane partitioning of nested loops. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing.* The MIT Press, 1990.
21. 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.
22. R. H. Kuhn. *Optimization and Interconnection Complexity for Parallel Processors, Single-Stage Networks and Decision Trees.* PhD thesis, University of Illinois at Urbana-Champaign, 1980.
23. H. T. Kung and C. E. Leiserson. Algorithms for VLSI processor arrays. In C. Mead and L. Conway, editors, *Introduction to VLSI Systems*, chapter 8.3. Addison-Wesley, 1980. Previously published as: Systolic arrays for VLSI, in *SIAM Sparse Matrix Proceedings*, 1978, 245–282.
24. L. Lamport. The parallel execution of DO loops. *Comm. ACM*, 17(2):83–93, February 1974.
25. H. Le Verge. Reduction operators in ALPHA. In D. Etiemble and J.-C. Syre, editors, *Parallel Architectures and Languages Europe (PARLE '92)*, Lecture Notes in Computer Science 605, pages 397–410. Springer-Verlag, 1992.
26. 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.
27. P. Lee and Z. Kedem. Synthesizing linear-array algorithms from nested for loop algorithms. *IEEE Trans. on Computers*, C-37(12):1578–1598, December 1988.
28. C. Lengauer and J. Xue. A systolic array for pyramidal algorithms. *J. VLSI Signal Processing*, 3(3):239–259, 1991.
29. J. Li and M. Chen. The data alignment phase in compiling programs for distributed memory machines. *J. Parallel and Distributed Computing*, 13(2):213–221, October

1991.
30. W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. Technical Report TR 92-1294, Department of Computer Science, Cornell University, July 1992.
31. W. L. Miranker and A. Winkler. Space-time representation of computational structures. *Computing*, pages 93–114, 1984.
32. D. I. Moldovan. On the design of algorithms for VLSI systolic arrays. *Proc. IEEE*, 71(1):113–120, January 1983.
33. D. I. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms into fixed-size systolic arrays. *IEEE Trans. on Computers*, C-35(1):1–12, January 1986.
34. G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, 1988.
35. D. D. Prest. Translation of abstract distributed programs to occam 2. 4th-Year Report, Department of Computer Science, University of Edinburgh, June 1992.
36. P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proc. 11th Ann. Int. Symp. on Computer Architecture*, pages 208–214. IEEE Computer Society Press, 1984.
37. P. Quinton and Y. Robert. *Systolic Algorithms and Architectures.* Prentice-Hall, 1990.
38. P. Quinton and V. van Dongen. The mapping of linear recurrence equations on regular arrays. *J. VLSI Signal Processing*, 1(2):95–113, October 1989.
39. S. V. Rajopadhye and R. M. Fujimoto. Synthesizing systolic arrays from recurrence equations. *Parallel Computing*, 14(2):163–189, June 1990.
40. S. V. Rajopadhye and M. Muddarangegowda. Parallel assignment, reduction and communication. In *Proc. SIAM Conference on Parallel Processing for Scientific Computing*, pages 849–853. SIAM, 1993.
41. J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):472–482, 1991.
42. S. K. Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays.* PhD thesis, Department of Electrical Engineering, Stanford University, October 1985.
43. S. K. Rao and T. Kailath. Regular iterative algorithms and their implementations on processor arrays. *Proc. IEEE*, 76(3):259–282, March 1988.
44. H. B. Ribas. *Automatic Generation of Systolic Programs from Nested Loops.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, June 1990. Technical Report CMU-CS-90-143.
45. Y. Robert and S. W. Song. Revisiting cycle shrinking. *Parallel Computing*, 18:481–496, 1992.
46. V. Roychowdhury, L. Thiele, S. K. Rao, and T. Kailath. On the localization of algorithms for VLSI processor arrays. In R. Brodersen and H. Moscovitz, editors, *VLSI Signal Processing III*, pages 459–470. IEEE Press, 1988.
47. A. Schrijver. *Theory of Linear and Integer Programming.* Series in Discrete Mathematics. John Wiley & Sons, 1986.
48. J. Teich and L. Thiele. Control generation in the design of processor arrays. *J. VLSI Signal Processing*, 3(1–2):77–92, 1991.
49. J. Teich and L. Thiele. Partitioning of processor arrays: A piecewise regular approach. *INTEGRATION*, 14(3):297–332, 1993.
50. L. Thiele. CAD for signal processing architectures. In P. Dewilde, editor, *The State of the Art in Computer Systems and Software Engineering*, pages 101–151. Kluwer, 1992.

51. A. van der Hoeven. *Concepts and Implementation of a Design System for Digital Signal Processing*. PhD thesis, Department of Electrical Engineering, Delft University of Technology, October 1992.
52. V. van Dongen. Quasi-regular arrays: Definition and design methodology. In J. V. McCanny, J. McWhirter, and E. E. Swartzlander, editors, *Systolic Array Processors*, pages 126–135. Prentice Hall, 1989.
53. 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.
54. M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, October 1991.
55. M. Wolfe. Multiprocessor synchronization for concurrent loops. *IEEE Software*, pages 34–42, January 1988.
56. M. Wolfe. *Optimizing Supercompilers for Supercomputers*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.
57. Y. Wong and J. M. Delosme. Optimal systolic implementations of n-dimensional recurrences. In *Proc. IEEE Int. Conf. on Computer Design (ICCD 85)*, pages 618–621. IEEE Press, 1985. Also: Technical Report 8810, Department of Computer Science, Yale University, 1988.
58. J. Xue. Specifying control signals for systolic arrays by uniform recurrence equations. *Parallel Processing Letters*, 1(2):83–93, 1992.
59. J. Xue and C. Lengauer. The synthesis of control signals for one-dimensional systolic arrays. *INTEGRATION*, 14:1–32, 1992.
60. H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. Frontier Series. Addison-Wesley (ACM Press), 1990.

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