LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - The University of Edinburgh

Loop Parallelization and Unimodularity

by

Michael Barnett
Christian Lengauer

Loop Parallelization and Unimodularity

LFCS Report Series ECS-LFCS-92-197

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

January 1992

Copyright © 1992, LFCS

Copyright © 1992, Laboratory for Foundations of Computer Science, University of Edinburgh. All rights reserved.

Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.

Loop Parallelization and Unimodularity

Michael Barnett
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712–1188, U.S.A.
mbarnett@cs.utexas.edu

Christian Lengauer
Department of Computer Science
University of Edinburgh
Edinburgh EH9 3JZ, Scotland
lengauer@dcs.ed.ac.uk

January 30, 1992

To be presented at the *20ème Ecole de Printemps du LITP "Algorithmique Parallèle"*, Côte Normande, 25–29 May 1992.

Abstract

The parallelization of loops can be made formal by basing it on an algebraic theory of loop transformations. In this theory, the concept of unimodularity arises. We discuss the pros and cons of insisting on unimodularity and review the choices made in two different areas of parallelization: systolic arrays and parallelizing compilers.

1 Introduction

Recently, the research areas of systolic arrays and parallelizing compilers have begun to address common concerns with common techniques and approaches. Initially, these two areas had developed separately.

Systolic design was concerned with the derivation of custom VLSI chips that embody some special-purpose highly parallel algorithm [18]. The central problem is the discovery of a mapping (or transformation) of the algorithm's operations from some specification to space-time. Thus, systolic design research concentrated on the algebraic theory of such mappings and was initially successful under the constraint that they be linear [24]. Algorithms were required to have data dependences simple enough to permit this linearity. Such dependences are called *regular* or *constant-distance* dependences; algorithms that fulfill this requirement can be phrased as uniform recurrence equations [16].

Parallelizing compilation was concerned with vectorizing compilers, principally, for "dusty deck" FORTRAN programs [17]. Researchers in this area focused on the dependence analysis of more general data access patterns (specified by *direction vectors*). They accumulated a large collection of transformations that were applied heuristically to take advantage of independent operations.

Evidence of the proximity between the two areas emerged before either area was well developed [19], but only recent advances on the theoretical and architectural

front have brought it into focus. Much recent work in parallelizing compilation has been aimed at finding a theory that underlies the transformations discovered earlier. The architectures considered for parallelizing compilation have evolved from SIMD vector processors to MIMD shared-memory and, subsequently, to distributed-memory machines. From the systolic design perspective, the newest generation of distributed-memory parallel processor networks with their rich, regular and efficient connection patterns [1, 14, 26, 27] resemble general-purpose, programmable versions of the traditional systolic chip. This poses a combined challenge to both areas.

The rest of this chapter is organized as follows. First, in Section 2, we introduce some basic concepts and terminology. Then we brief review the research results in both areas: parallelizing compilation (Section 3) and systolic design (Section 4). Unimodular transformations and their importance are discussed in Section 5. Section 6 describes different methods for coping with non-unimodular transformations. Finally, Section 7 presents our conclusions.

We wish to stress that our description of the two areas and of the role of unimodularity is entirely personal and is meant as a point of discussion. By necessity, we have ignored much research in both areas that does not pertain to loop transformations.

2 Basic Concepts and Terminology

In both areas, one starts with a *source program*, $\mathcal{SP}$, without parallelism and produces a *target program*, $\mathcal{TP}$, with parallelism. But, for a long time, both areas understood the derivation of the target from the source in different terms. In systolic design, the two programs were related by a space-time transformation [24], in parallelizing compilation by a set of heuristic loop transformations schemes like loop jamming, loop reversal, strip mining, skewing, tiling, and unrolling [2, 10, 23, 29]. Recent work in parallelizing compilation expresses each transformation scheme as a matrix [6, 28]. A compound transformation is the functional composition of its component transformations; thus, we can now specify the derivation of the target program in either area by a (linear) mapping, T.

For our purposes, the source program is a set of, say, r perfectly nested loops; it is represented by a set of integer points that is a convex polyhedral subset of the cartesian space $\mathbf{R}^r$. This space, $\mathcal{IS}$, is called the *index space* or *iteration space*. Each loop must have a unit stride. (A *unit* value is either $+1$ or -1.) Thus, each integer-valued point in the iteration space corresponds to one iteration. Target programs are represented by another subset of $\mathbf{R}^r$, the *target space*, $\mathcal{TS}$. We refer to the points in the spaces and the operations in the respective programs interchangably.

The target program is derived by considering the (in)dependence of pairs of operations. Each operation corresponds to a point in $\mathbf{R}^r$, pairs of operations correspond to directed vectors in $\mathbf{R}^r$. The target program must execute operations that are dependent in the order specified by the source program. Dependences between operations are represented by *direction vectors.* Operations that are not dependent may be executed in parallel. In particular, when all iterations of a loop are mutually independent, they may be executed in parallel.

Thus, both areas are concerned with the discovery of two functions:

$T :: \mathcal{IS} \longrightarrow \mathcal{TS}$ This *space-time transformation* must, under preservation of the dependences of $\mathcal{SP}$, provide a distribution of its operations in time and space – preferably such that at least some dimensions of $\mathcal{TS}$ can be represented by parallel rather than sequential loops.

$P :: (\mathcal{SP}, T) \longmapsto \mathcal{TP}$ This *code generator* takes a source program and a space-time transformation of it and produces a distributed program.

Most work in both areas has focused on T: how to derive a space-time transformation that has certain properties such as minimizing the extent of the temporal dimension(s) of $\mathcal{TS}$, or the extent of the spatial dimension(s) of $\mathcal{TS}$, or maximizing throughput, or a combination of these. Instead, we concentrate on P and on the way in which the unimodularity or non-unimodularity of T affects it.

An explicit representation of $\mathcal{TP}$ must provide (1) a spatial enumeration of the processes and a temporal enumeration of the operations of each process and (2) in order to obtain the correct data references as specified in the source program, for each point in $\mathcal{TS}$, the inverse image under T. We require that $\mathcal{TP}$ be defined by a set of loops. For spatial composition we use the operator **par**, temporal composition is denoted by **seq**. Both may be used in an indexed form that allows enumeration. Communications are indicated by **send** and **recv**.

3 Parallelizing Compilation

Parallelizing compilers create two types of loops: DOALL and DOACROSS. DOALL loops have the property that each iteration is completely independent of the others. When each iteration of a DOALL loop is executed on a separate processor, no communication is needed between the processors. The iterations of a DOACROSS loop may depend on each other. They are constrained to execute in sequence, either on the same processor or on different processors. In the latter case, communication must enforce the sequencing. In our notation, both DOALL and DOACROSS loops are **par** loops: a DOALL loop is a **par** loop without communications in its body.

The first step in determining which iterations are independent is to capture the dependences that could constrain the order of their execution. Conditions on the set of direction vectors determine which loops may be executed in parallel. When these conditions are not met, certain transformations, when legal (that is, preserving the desired aspects of the source program's behaviour), can establish them by changing the loops and/or the way variables are indexed. Recent research has provided a theory for these transformations [6, 22, 28], but only for DOALL loops.

4 Systolic Design

Systolic arrays are a restricted form of parallel processor network. A systolic array comprises a set of simple processors laid out in a regular topology. Only local data connections exist and the parallelism is usually synchronous. (Synchrony is not required [18]. It simplifies VLSI implementations of communication but is inappropriate for asynchronous programmable processor networks [20].) These restrictions require algorithms that exhibit extremely simple and regular data access patterns:

traditionally, ones with constant dependence vectors [24] (but recently also more general forms). The source programs are expressed either as recurrence equations, defined on a subset of $\mathbf{Z}^r$, or, equivalently [9], as a set of nested loops.

The space-time transformation can be divided into two linear functions: **place** determines the spatial and **step** the temporal dimension(s) of $\mathcal{TS}$. In the conventional approach, $\mathcal{TS}$ has $r-1$ spatial dimensions and one temporal dimension (where r is the number of loops in the source program). The sequential (**seq**) loop is either the outermost loop or the innermost loop; the former results in a synchronous, the latter in an asynchronous target program.

5 Unimodular Transformations

In both areas, one requires that the transformation T be invertible. Otherwise, two operations may be mapped to the same place and the same time, which contradicts the premiss that the multiprocessor array consists of sequential processors (from the programmer's point of view).

Definition 1 *A transformation is* unimodular *if and only if*

1. *it is invertible,*
2. *it maps integer points to integer points, and*
3. *its inverse maps integer points to integer points.*

An integer matrix is unimodular if and only if it has a unit determinant.

Thus, a unimodular transformation T and its inverse T^{-1} are both matrices in $\mathbf{Z}^{r \times r}$.

Unimodularity has extremely pleasant consequences for the derivation of the target program. It means that the target program need only enumerate every integer valued point within the convex hull of the target space and, for each point, execute the operation defined by applying T^{-1} to it. Non-unimodular transformations can be thought of as creating "holes" in the target space: integer points in the convex hull of $\mathcal{TS}$ that are not in the range of T.

Is it easy to obtain unimodular transformations? The basic transformations in parallelizing compilation correspond to elementary matrices that are unimodular [5]. Moreover, the composition of the transformations corresponds to the matrix product – which preserves unimodularity. When using unimodular transformations, it suffices to coerce non-integer points derived in the target space to the closest interior integer points (using *floor* or *ceiling* functions), since all integer-valued points inside the space boundaries are "good" points. Dowling [12] and Irigoin [15] present algorithms for deriving a unimodular transformation from an initial vector representing the wavefront direction (for programs that are amenable to wavefronting).

Ribas [25] uses only unimodular transformations, taking the position that unimodular transformations are sufficient for the actual programs one encounters.

Leverge, Mauras and Quinton [21] require unimodularity. For a non-unimodular transformation, they define a unimodular extension on a target space with one added dimension.

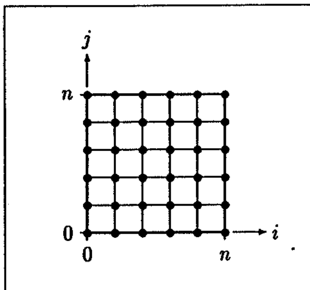

Figure 1: The index space for the example

An Example

The following program multiplies the coefficients of two polynomials a and b, both of degree n, and produces a polynomial c of degree $2n$.

$$\begin{aligned}&\textbf{for } i \;=\; 0 \leftarrow 1 \rightarrow n\\ &\quad\textbf{for } j \;=\; 0 \leftarrow 1 \rightarrow n\\ &\qquad c[i+j] \;:=\; c[i+j] + a[i]\; b[j]\end{aligned}$$

Figure 1 depicts the index space $\mathcal{IS}$ of the program. One valid transformation and its inverse are:

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

T skews the inner loop and then wavefronts both loops. It is unimodular and has the determinant $+1$. This transformation produces the target space depicted in Figure 2. The horizontal axis, with coordinate t (for *time*), can be implemented by a sequential loop. The vertical axis, with coordinate p (for *process*), can be implemented by a parallel loop – a DOALL loop if it is the inner loop. Which one is the outer loop, and which the inner, depends on whether the program is implemented with synchronous or asynchronous parallelism. The target program is executed by enumerating the points in $\mathcal{TS}$ and, for each point, applying a modified operation. The point (t,p) in the target space corresponds to the point $(i,j) \;=\; T^{-1}\,(t,p) \;=\; (t-p, 2p-t)$ in the index space. In the target program, variables must be indexed in terms of target space coordinates; for example, $c[i+j]$ becomes $c[(t-p)+(2p-t)] \;=\; c[p]$. The target program is:

$$\begin{aligned}&\textbf{seq } t \;=\; 0 \leftarrow 1 \rightarrow 3n\\ &\quad\textbf{par } p \;=\; \max(\lceil t/2\rceil, t-n) \leftarrow 1 \rightarrow \min(t, \lfloor (t+n)/2 \rfloor)\\ &\qquad c[p] \;:=\; c[p] + a[t-p]\; b[2p-t]\end{aligned}$$

To preserve the dependences of the source program, a *barrier synchronization* between iterations of the outer loop is required: all processes p of one **seq** iteration must

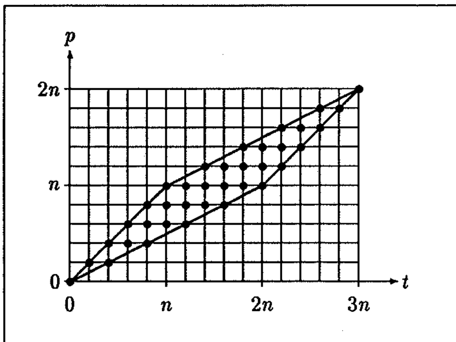

Figure 2: The target space produced by a unimodular transformation. The thick lines represent the loop bounds of the target program.

terminate before the next **seq** iteration can begin. That is, the **seq** loop imposes synchrony on the **par** loop.

As written above, the target program implicitly assumes the use of a shared-memory multiprocessor. Execution on a distributed memory machine requires the insertion of communication directives for the exchange of non-local data between processors. (On an asynchronous distributed architecture, one can save the overhead of the barrier synchronization by making the **seq** loop the inner loop; then the data dependences are imposed by the data communications per se.)

There are several methods for the derivation of the loop bounds in the target program: Wolf and Lam [28], Irigoin [15], Feautrier [13], and Ancourt [4] all give general algorithms. Ribas [25] uses an algorithm that covers simple cases (linear loop bounds) and decomposes the target space when necessary.

6 Non-Unimodular Transformations

Any approach that permits non-unimodular transformations must ensure that only the "good" points – those in $\mathcal{TS}$ – are actually executed; integer points that do not correspond to iterations of the source program must not be executed. Perhaps the simplest solution is to enumerate every integer point in a superset of $\mathcal{TS}$ – the convex hull is particularly simple – and then to check each point at run time to see whether T^{-1} applied to the point corresponds to a point in $\mathcal{IS}$. With this strategy, one can use the same methods as for the unimodular case. The question is whether it is possible to move (at least some of) the computation involved in determining "goodness" from run time to compile time. A related question is what amount of run-time overhead such compilation methods induce.

This is the approach of Lu and Chen [22]. They do not make clear how inefficient this method is or what the complexity of the tests must be. (In their examples they hand-optimize them to simple inequalities.) Yang and Choo [30], who belong to

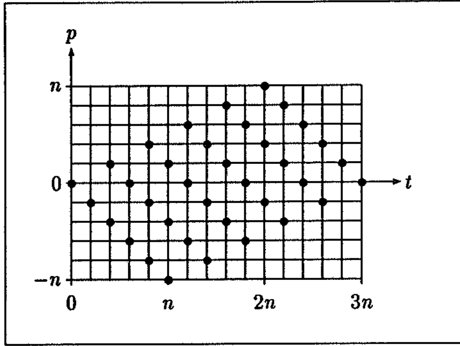

Figure 3: The target space produced by a non-unimodular transformation.

the same research group as Lu and Chen, also disregard unimodularity but express the boundaries of the target space as linear equations instead of loop bounds. A transformation of the linear equations to loop bounds is possible but non-trivial [13, 15].

A more complicated, but also more accurate approach is to enumerate the points in $\mathcal{TS}$ precisely. In our own work [7, 8], we define the target program piecewise, using *repeaters* – a notation of tuple-valued loop indices. By using piecewise loop bounds and non-unit strides in the target programs, we avoid the imprecise bounds of Wolf and Lam [28] and the holes of Leverge, Mauras and Quinton [21].

The Example Revisited

For the same example, one valid non-unimodular transformation and its inverse are:

$$T = \begin{bmatrix} 2 & 1 \\ 1 & -1 \end{bmatrix} \qquad T^{-1} = \begin{bmatrix} 1/3 & 1/3 \\ 1/3 & -2/3 \end{bmatrix}$$

The determinant of T is -3, the target space is depicted in Figure 3. Note that it is not dense in the integers: points in $\mathcal{TS}$ are separated by integer points not in $\mathcal{TS}$.

We illustrate first the simple approach. Figure 4 depicts the convex hull of $\mathcal{TS}$. The synchronous program with a guard that establishes "goodness" is:

seq $t = 0 \leftarrow 1 \rightarrow 3n$
  par $p = \max(-t, \lceil (t-3n)/2 \rceil) \leftarrow 1 \rightarrow \min(\lfloor t/2 \rfloor, 3n-t)$
    if $T^{-1}(t,p) \in \mathcal{IS} \rightarrow c[(2t-p)/3] := c[(2t-p)/3] + a[(t+p)/3]\, b[(t-2p)/3]$
    [] $T^{-1}(t,p) \notin \mathcal{IS} \rightarrow$ **skip**
    fi

The guard $T^{-1}(t,p) \in \mathcal{IS}$ may be simplified, but it is not clear how much of the simplification can be mechanized.

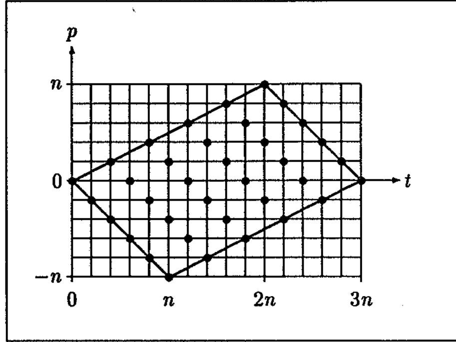

Figure 4: The convex hull of the target space produced by a non-unimodular transformation.

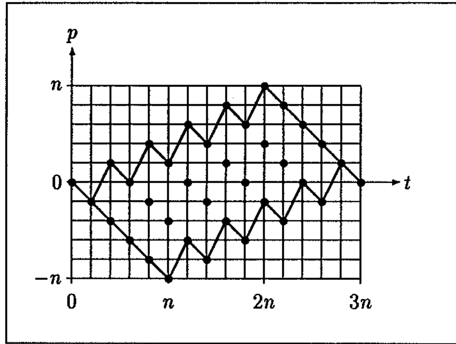

Figure 5: Non-convex boundaries for the synchronous program

Following the second approach, Figure 5 shows the precise bounds of the target space. A synchronous program that specifies these bounds and uses non-unit strides to omit the points that do not correspond to iterations in the source program is:

seq $t \;=\; 0 \leftarrow 1 \rightarrow 3n$
$\quad lb, rb \;:=\; f.t,\, g.t$
$\quad$ **par** $p \;=\; lb \leftarrow 3 \rightarrow rb$
$\qquad c[(2t-p)/3] \;:=\; c[(2t-p)/3] + a[(t+p)/3]\; b[(t-2p)/3]$

where

$$
\begin{array}{lcll}
f.t & = & \textbf{if} & 0 \le t \le n & \rightarrow & -t \\
 & & [] & n \le t \le 3n \ \wedge \ 2 \mid (t-3n) & \rightarrow & (t-3n)/2 \\
 & & [] & n \le t \le 3n \ \wedge \ 2 \not\mid (t-3n) & \rightarrow & (t-3n+3)/2 \\
 & & \textbf{fi} & & &
\end{array}
$$

and g is a similar piecewise linear function. Recalculating the number of active processes on each sequential iteration may be inefficient, in general. Consider, however, an asynchronous program – one defined by indexing the inner instead of the outer loop by t. In the absence of a synchronous loop enforcing the data dependences, the data communication constrains the order of execution, as explained before. Each process begins by calculating the loop bounds for its sequential loop. Given asynchronous processes that are static and do not contain nested loops (which is the case for full-dimensional systolic arrays), the overhead incurred by this calculation should be negligible. This is our approach. Our asynchronous target program is:

$$
\begin{array}{l}
\textbf{par}\ p\ =\ -n \leftarrow 1 \rightarrow n \\
\quad lb, rb\ :=\ f.p,\ g.p \\
\quad \textbf{seq}\ t\ =\ lb \leftarrow 3 \rightarrow rb \\
\qquad \textbf{recv}\ a[(t+p)/3]\ \textbf{from}\ p+1 \\
\qquad \textbf{recv}\ b[(t-2p)/3]\ \textbf{from}\ p-1 \\
\qquad \textbf{recv}\ c[(2t-p)/3]\ \textbf{from}\ p-2 \\
\qquad c[(2t-p)/3]\ :=\ c[(2t-p)/3] + a[(t+p)/3]\ b[(t-2p)/3] \\
\qquad \textbf{send}\ a[(t+p)/3]\ \textbf{to}\ p-1 \\
\qquad \textbf{send}\ b[(t-2p)/3]\ \textbf{to}\ p+1 \\
\qquad \textbf{send}\ c[(2t-p)/3]\ \textbf{to}\ p+2
\end{array}
$$

where f and g are similar as before. Our method generates these functions automatically, but may create more processes than necessary: for simplicity, we use the rectangular closure of the dimensions of $\mathcal{TS}$ that are represented by parallel loops. A better algorithm that calculates more accurate **par** loop bounds could be used. When there is only one parallel loop, as in our example, our method creates no extra processes.

7 Conclusions

We conclude that, although unimodularity clearly simplifies the derivation of the target program, it is not essential. The most obvious concept introduced by non-unimodular transformations, non-unit loop strides in the target program, is the easiest part of the problem. A much more difficult matter is the precise description of the non-convex boundaries that delineate the "good" points. Methods for coping with non-unimodular transformations require not only the ability to generate piecewise linear loop bounds but also to analyze the transformation T along with $\mathcal{TS}$ in order to specify the non-convex boundaries. There certainly is enough information available at compile-time to make the alternative – testing at run time – unnecessary. Research intended for slightly different purposes (for example, [3]) may be adaptable to such derivations. Possibly, the "folding" proposed by Clauss, Mongenet, and Perrin [11], could also be used to make non-convex domains convex.

Our viewpoint is that the drawbacks of non-unimodularity depend on the structure of the target program. At least for some asynchronous distributed-memory programs, non-unimodularity incurs less overhead than for synchronous programs intended for shared-memory machines. On the other hand, one should not simply dismiss the concept of unimodularity: dealing with a non-unimodular target space is non-trivial.

Acknowledgements

In addition to sending their papers, Corinne Ancourt, Paul Feautrier, Lee-Chung Lu, Hudson Ribas, Michael Wolf and Allan Yang explained their work in numerous e-mail discussions. This helped us greatly in forming our opinion. Thanks also to Jingling Xue for a careful reading. Financial support from SERC, grant no. GR/G55457, and from LITP is gratefully acknowledged.

References

[1] H. Aida, J. A. Goguen, and J. Meseguer. Compiling cuncurrent rewriting onto the rewrite rule machine. In S. Kaplan and M. Okada, editors, *Conditional and Typed Rewriting Systems*, Lecture Notes in Computer Science 516. Springer-Verlag, 1991. Also: Technical Report SRI-CSL-90-03R, SRI International, Dec. 1990.

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

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

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

[5] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[6] U. Banerjee. Unimodular transformations of double loops. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, chapter 10, pages 192–219. MIT Press, 1991.

[7] M. Barnett and C. Lengauer. The synthesis of systolic programs. In *Proc. Seminar on Research Directions in High-Level Parallel Programming Languages.* Springer-Verlag, 1991. To appear.

[8] M. Barnett and C. Lengauer. A systolizing compilation scheme. Technical Report TR-91-03, Department of Computer Sciences, The University of Texas at Austin, Jan. 1991. Also: Technical Report ECS-LFCS-91-134, Department of Computer Science, University of Edinburgh. Abstract: *Proc. 1991 Int. Conf. on Parallel Processing, Vol. II*, Pennsylvania State University Press, 1991, 305–306.

[9] J. Bu and E. F. Deprettere. Converting sequential iterative algorithms to recurrent equations for automatic design of systolic arrays. In *Proc. IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP 88), Vol. IV: VLSI; Spectral Estimation*, pages 2025–2028. IEEE Press, 1988.

[10] D. Callahan and K. Kennedy. Compiling programs for distributed-memory processors. *J. Supercomputing*, 2(2):151–169, Oct. 1989.

[11] P. Clauss, C. Mongenet, and G. R. Perrin. Calculus of space-optimal mappings of systolic algorithms on processor arrays. In S. Y. Kung and E. E. Swartzlander, editors, *Application Specific Array Processors*, pages 5–18. IEEE Computer Society Press, 1990.

[12] M. L. Dowling. Optimal code parallelization using unimodular transformations. *Parallel Computing*, 16(2–3):157–171, Dec. 1990.

[13] 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.

[14] INMOS Ltd. *The T9000 transputer • Products Overview • Manual.* SGS-Thompson Microelectronics Group, first edition, 1991.

[15] F. Irigoin. Code generation for the hyperplane method and for loop interchange. Technical Report ENSMP-CAI-88-E102/CAI/I, Ecole Nationale Superieure des Mines de Paris, Oct. 1988.

[16] 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.

[17] D. J. Kuck. *The Structure of Computers and Computations.* John Wiley & Sons, 1978.

[18] S.-Y. Kung. *VLSI Processor Arrays.* Prentice-Hall Int., 1988.

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

[20] C. Lengauer, M. Barnett, and D. G. Hudson. Towards systolizing compilation. *Distributed Computing*, 5(1):7–24, 1991.

[21] H. Leverge, C. Mauras, and P. Quinton. A language-oriented approach to the design of systolic chips. In E. F. Deprettere and A.-J. van der Veen, editors, *International Workshop on Algorithms and Parallel VLSI Architectures, Vol. A: Tutorials*, pages 309–327. Elsevier (North-Holland), 1990. To appear in *J. VLSI Signal Processing.*

[22] L.-C. Lu and M. Chen. New loop transformation techniques for massive parallelism. Technical Report YALEU/DCS/TR-833, Department of Computer Science, Yale University, Oct. 1990.

[23] K. Pingali and A. Rogers. Compiler parallelization of SIMPLE for a distributed memory machine. Technical Report TR 90-1084, Department of Computer Science, Cornell University, Jan. 1990.

[24] S. K. Rao and T. Kailath. Regular iterative algorithms and their implementations on processor arrays. *Proc. IEEE*, 76(3):259–282, Mar. 1988.

[25] 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.

[26] C. E. Seitz. Multicomputers. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, chapter 5, pages 131–200. Addison-Wesley, 1990.

[27] Thinking Machines Corporation. *The Connection Machine CM-5, Technical Summary*, Oct. 1991.

[28] 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, Oct. 1991.

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

[30] J. A. Yang and Y.-I. Choo. Parallel-program transformation using a metalanguage. In *Third ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 11–20. ACM Press, Apr. 1991.