Parallel Processing Letters

UNIMODULARITY AND THE PARALLELIZATION OF LOOPS*

MICHAEL BARNETT
Laboratory for Applied Logic
Department of Computer Science
University of Idaho
Moscow, Idaho 83843, U.S.A.

and

CHRISTIAN LENGAUER
Fakultät für Mathematik und Informatik
Universität Passau
D–W8390 Passau, Germany

Received
Revised
Accepted by

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.

Keywords: parallelizing compiler, space-time transformation, systolic array, unimodularity.

1. Introduction

In the parallelization of loops, one starts with a *source program*, $\mathcal{SP}$, that does not specify parallelism and produces a *target program*, $\mathcal{TP}$, that does specify parallelism.

For our purposes, the source program is a set of, say, r perfectly nested loops; it is represented by a set, $\mathcal{IS}$, of integer points that is a convex polyhedral subset of the cartesian space $\mathbb{R}^r$. $\mathcal{IS}$ 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 $\mathbb{R}^r$, the *target space*, $\mathcal{TS}$. We refer to the points in the spaces and the operations in the respective programs interchangably.

*Earlier and expanded version presented at the *20ème Ecole de Printemps du LITP "Algorithmique Parallèle"* [6].

The derivation of $\mathcal{TP}$ rests on 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. T should 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).

$\mathcal{CG} :: (\mathcal{SP}, T) \longmapsto \mathcal{TP}$ 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 $\mathcal{CG}$ 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**.

2. Unimodular Transformations

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 $\mathbb{Z}^{r \times r}$.

The benefit of a unimodular transformation is that it preserves convexity. Thus, if $\mathcal{IS}$ is convex so is $\mathcal{TS}$. Then the target program can simply 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 applied by parallelizing compilers correspond to elementary matrices that are unimodular [3]. Moreover, the composition of the transformations corresponds to the

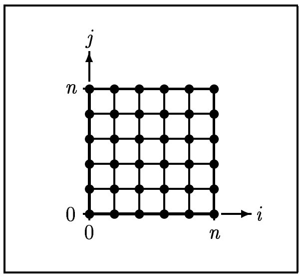

Fig. 1. The index space for the example.

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 the functions *floor* or *ceiling*), since all integer-valued points inside the space boundaries are "good" points. Dowling [9] and Irigoin [11] present algorithms for deriving a unimodular transformation from a vector representing the wavefront direction (for programs that are amenable to wavefronting).

Ribas [15] uses only unimodular transformations, taking the position that unimodular transformations are sufficient for the actual programs one encounters.

Le Verge, Mauras and Quinton [12] require unimodularity. For a non-unimodular transformation, they define a unimodular extension on a target space with one added dimension.

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 $2n$, provided c is initially everywhere zero.

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

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

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

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

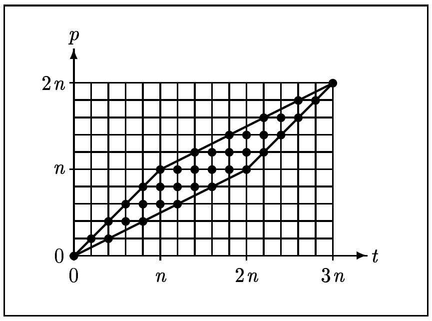

Fig. 2. The target space produced by a unimodular transformation. The thick lines represent the loop bounds of the target program.

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 [17]. 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:

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

To preserve the dependences of the source program, a *barrier synchronization* [17] between iterations of the outer loop is required: all processes p of one **seq** iteration must 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 [16], Irigoin [11], Feautrier [10], and Ancourt [2] all give general algorithms. Ribas [15] uses an algorithm that covers simple cases (linear loop bounds) and decomposes the target space when necessary.

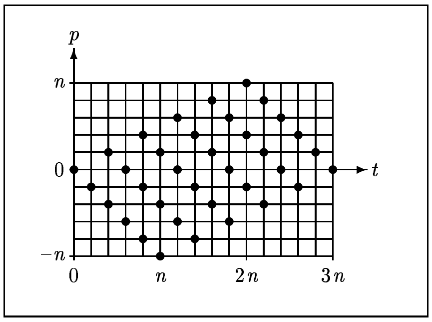

Fig. 3 The target space produced by a non-unimodular transformation.

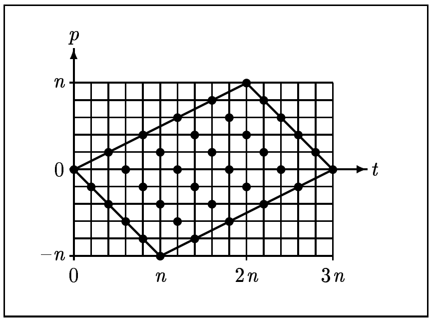

Fig. 4 The convex hull of the target space.

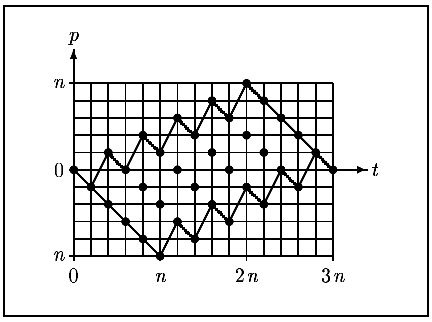

Fig. 5 The precise boundaries of the target space.

3. Non-Unimodular Transformations

Any approach that permits non-unimodular transformations must ensure that only the "good" points—those in $\mathcal{TS}$—are 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 test at run time which points T^{-1} maps to $\mathcal{IS}$. With this strategy, one can use the same methods as for the unimodular case.

This is the approach of Lu and Chen [14]. 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 tests to simple inequalities.) Yang and Choo [18] 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 [10, 11].

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.

In our own work, we enumerate the points in $\mathcal{TS}$ precisely [4, 5]. We consider time and space separately—which avoids the issue of unimodularity altogether—and define the target program piecewise using a notation of tuple-valued loop indices (called "repeaters"). By using piecewise loop bounds and non-unit strides in the target programs, we avoid the imprecise bounds of Wolf and Lam [16] and the holes of Le Verge, Mauras and Quinton [12].

The Example Revisited

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

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

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:

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

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.

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:

$$\begin{array}{l} \textbf{seq}\ t \ = \ 0 \leftarrow 1 \rightarrow 3n \\ \quad lb, rb \ := \ f.t,\ g.t \\ \quad \textbf{par}\ p \ = \ lb \leftarrow 3 \rightarrow rb \\ \qquad c[(2t-p)/3] \ := \ c[(2t-p)/3] + a[(t+p)/3]\ b[(t-2p)/3] \end{array}$$

where

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

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. An algorithm for 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.

4. Conclusions

We conclude that, although unimodularity clearly simplifies the derivation of the target program, it is not essential. There certainly is enough information available at compile-time to make the alternative—testing for membership in $\mathcal{TS}$ at run time—unnecessary.

An accurate description of a non-convex target space is also possible in the traditional setting where transformation T is not split into a time and a space component. Non-unit steps can be obtained by a change of basis and jagged boundaries can be accurately described by modifying the loop bounds appropriately [7]. (For a different treatment, see [13].) Research intended for slightly different purposes (for example, [1]) may also be adaptable. Possibly, the "folding" proposed by Clauss, Mongenet, and Perrin [8], could be used to make non-convex domains convex.

The run-time testing involved in conditional bounds can be reduced if the target space is partitioned into the regions that are defined by the alternatives of the conditional, and each region is considered separately, yielding loops with linear bounds for each partition. Work on a theory of partitioned polyhedral domains of computation is under way in several places.

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. Thanks to all of them. Thanks also to Jingling Xue for a careful reading. Financial support from the Science and Engineering Research Council of Britain (SERC), grant no. GR/G55457, is gratefully acknowledged.

References

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

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

3. U. Banerjee, *Dependence Analysis for Supercomputing*, (Kluwer Academic Publishers, 1988).

4. M. Barnett and C. Lengauer, The synthesis of systolic programs, *Research Directions in High-Level Parallel Programming Languages*, eds. J.-P. Banâtre and D. Le Métayer, LNCS 574 (Springer-Verlag, 1991) 309–325.

5. M. Barnett, A systolizing compiler, Ph. D. Thesis, Department of Computer Sciences, The University of Texas at Austin, May 1992. Technical Report TR-92-13.

6. M. Barnett and C. Lengauer, Loop parallelization and unimodularity, *Algorithmique Parallèle*, eds. M. Cosnard, M. Nivat and Y. Robert, (Masson, 1992) 369–379.

7. M. Barnett and C. Lengauer, Unimodularity considered non-essential (extended abstract), *CONPAR 92 – VAPP V*, eds. L. Bougé, M. Cosnard, Y. Robert and D. Trystram, LNCS 634 (Springer-Verlag, 1992) 659–664.

8. P. Clauss, C. Mongenet, and G. R. Perrin, Calculus of space-optimal mappings of systolic algorithms on processor arrays, *J. VLSI and Signal Processing*, **4** (1992) 27–36.

9. M. L. Dowling, Optimal code parallelization using unimodular transformations, *Parallel Computing*, **16** (1990) 157–171.

10. P. Feautrier, Semantical analysis and mathematical programming *Parallel & Distributed Algorithms*, eds. M. Cosnard, Y. Robert, P. Quinton, and M. Raynal (North-Holland, 1989) 309–320.

11. 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.

12. H. Le Verge, C. Mauras, and P. Quinton, A language for the design of regular parallel algorithms, *J. VLSI Signal Processing*, **3** (1991) 173–182.

13. W. Li and K. Pingali, A singular loop transformation framework based on non-singular matrices, Technical Report TR 92-1924, Department of Computer Science, Cornell University, July 1992.

14. 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.

15. H. B. Ribas, Automatic generation of systolic programs from nested loops, Ph. D. Thesis, Department of Computer Science, Carnegie-Mellon University, June 1990. Technical Report CMU-CS-90-143.

16. M. Wolf and M. Lam, A loop transformation theory and an algorithm to maximize parallelism, *IEEE Trans. on Parallel and Distributed Systems*, **2** (1991) 452–471.

17. M. Wolfe, Multiprocessor synchronization for concurrent loops, *IEEE Software*, (1/1988) 34–42.

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