The Static Parallelization of Loops and Recursions

Christian Lengauer, Sergei Gorlatch and Christoph A. Herrmann

Fakultät für Mathematik und Informatik
Universität Passau, Germany

lengauer@fmi.uni-passau.de
http://www.uni-passau.de/~lengauer/

Abstract

We demonstrate approaches to the static parallelization of loops and recursions on the example of the polynomial product. Phrased as a loop nest, the polynomial product can be parallelized automatically by applying a space-time mapping technique based on linear algebra and linear programming. One can choose a parallel program that is optimal with respect to some objective function like the number of execution steps, processors, channels, etc. However, at best, linear execution time complexity can be attained. Through phrasing the polynomial product as a divide-and-conquer recursion, one can obtain a parallel program with sublinear execution time. In this case, the target program is not derived by an automatic search but given as a program skeleton, which can be deduced by a sequence of equational program transformations. We discuss the use of such skeletons, compare and assess the models in which loops and divide-and-conquer recursions are parallelized and comment on the performance properties of the resulting parallel implementations.

Keywords: divide-and-conquer, higher-order function, homomorphism, loop nest, parallelization, polytope model, skeletons, SPMD.

1 Introduction

We give an overview of several approaches to the static parallelization of loops and recursions[1], which we pursue at the University of Passau. Our emphasis in this paper is on divide-and-conquer recursions.

Static parallelization has the following benefits:

1. *Efficiency of the target code.* One avoids the overhead caused by discovering parallelism at run time and minimizes the overhead caused by administering parallelism at run time.

2. *Precise performance analysis.* Because the question of parallelism is settled at compile time, one can predict the performance of the program more accurately.

3. *Optimizing compilation.* One can compile for specific parallel architectures.

One limitation of static parallelization is that methods which identify large amounts of parallelism usually must exploit some regular structure in the source program. Mainly, this

[1] We can equate loops with tail recursions, as is done in systolic design [28, 36].

structure concerns the dependences between program steps, because the dependences impose an execution order. Still, after a source program has been "adapted" to satisfy the requirements of the parallelization method, the programmer need think no more about parallelism but may simply state his/her priorities in resource consumption and let the method make all the choices.

We illustrate static parallelization methods for recursive programs on the example of the polynomial product. We proceed in four steps:

Sect. 2. We provide a specification of the polynomial product. This specification can be executed with dynamic parallelism. The drawback is that we have no explicit control over the use of resources.

Sect. 3. We refine the specification to a double loop nest with additional dependences. We parallelize this loop nest with the space-time mapping method based on the polytope model [29]. This method searches automatically a large number of possible parallel implementations, optimizing with respect to some objective function like the number of execution steps, processors, communication channels, etc.

Sect. 4. We refine the specification to a divide-and-conquer (D&C) algorithm which has fewer dependences than the loop nest. This is the central section of the paper. For D&C, parallelization methods are not as well understood as for nested loops. Thus, one derives parallel implementations by hand, albeit formally, with equational reasoning. However, most of the parallelization process is problem-independent. The starting point is a program schema called a skeleton [9]. We discuss two D&C skeletons, instantiated to the polynomial product, and their parallelizations:

- *Subsect. 4.1.* The first is a skeleton for call-balanced fixed-degree D&C, which we parallelize with an adapted space-time mapping method based on the method for nested loops [25]. The target is again a parallel loop nest, which can also be represented as an SPMD program.
- *Subsect. 4.2.* The second skeleton is a bit less general. It is parallelized based on the algebraic properties of its constituents [20]. It is used to generate coarser-grained parallelism in the form of an SPMD program.

In this paper, we are mainly comparing and evaluating. The references cited in the individual sections contain the full details of the respective technical development. Our comparison is concerned with the models and methods used in the parallelization and with the asymptotic performance of the respective parallel implementations.

2 The polynomial product

Our illustrating example is the product of two polynomials A and B of degree n, specified in the quantifier notation of Dijkstra [14]:

$$\begin{aligned} A &= \langle \Sigma\ k\ :\ 0 \le k \le n\ :\ a[k]\cdot z^k \rangle \\ B &= \langle \Sigma\ k\ :\ 0 \le k \le n\ :\ b[k]\cdot z^k \rangle \end{aligned}$$

Let us name the product polynomial C:

$$C\ =\ A \odot B\ =\ \langle \Sigma\ k\ :\ 0 \le k \le 2\cdot n\ :\ c[k]\cdot z^k \rangle \tag{1}$$

$$\langle \forall\, k \;:\; 0 \le k \le 2 \cdot n \;:\; c[k] := \langle \Sigma\, i, j \;:\; 0 \le i \le n \wedge 0 \le j \le n \wedge i + j = k \;:\; a[i] \cdot b[j] \rangle \rangle \tag{2}$$

Note that this specification does not prescribe a particular order of computation for the cumulative sums which define the coefficients of the product polynomial.

We can make this specification executable without having to think any further. A simple switch of syntax to the programming language Haskell [40] yields:

$$
\begin{array}{l}
c\ \ a\ \ b\ \ n = \\
\qquad \mathsf{array}(0, 2*n)[\, k := \mathsf{sum}[\, a!i * b!j \mid i \leftarrow [0..n], j \leftarrow [0..n], i + j \equiv k \,] \mid k \leftarrow [0..2*n] \,]
\end{array}
$$

Haskell will reduce the sums in some total order, given by its sequential semantics, or in some partial order, given by its parallel semantics. The programmer pays for the benefit of not having to choose the order with a lack of control over the use of resources in the computation. The main resources are time (the length of execution) and space (the number of processors), others are the number of communication channels, the memory requirements, etc.

3 A nested loop program

3.1 From the source program to the target program

To apply loop parallelization, one must first impose a total order on the reductions in the specification. This means adding dependences to the dependence graph of the specification. The choice of order may influence the potential for parallelism, so one has to be careful. We choose to count the subscript of A up and that of B down; automatic methods can help in exploring the search space for this choice [3]. Another change we make is that we convert the updates of c to single-assignment form, which gives rise a doubly indexed variable $\bar{c}$; there are also automatic techniques for this kind of conversion [15]. This leads to the following program, in which the elements $\bar{c}[*, 0]$ and $\bar{c}[n, *]$ contain the final values of the coefficients of the product polynomial:

$$
\begin{array}{l}
\mathsf{for}\ i := 0\ \mathsf{to}\ n\ \mathsf{do} \\
\quad \mathsf{for}\ j := n\ \mathsf{downto}\ 0\ \mathsf{do} \\
\qquad \mathsf{if}\ \ i = 0 \vee j = n \rightarrow \bar{c}[i, j] := a[i] * b[j] \\
\qquad [\,]\ \ i \neq 0 \wedge j \neq n \rightarrow \bar{c}[i, j] := \bar{c}[i-1, j+1] + a[i] * b[j] \\
\qquad \mathsf{fi}
\end{array}
$$

The dependence graph of a loop nest with affine bounds and index expressions forms a *polytope*, in which each loop represents one dimension. The vertices of graphs of this form can be partitioned into temporal and spatial components. This is done by linear algebra and integer linear programming. In our example, the choices are fairly obvious.

Consider Fig. 1 for the polynomial product. The upper left shows the polytope on the integer lattice; each dot represents one loop step. The upper right is the dependence graph; only the dependences on $\bar{c}$ are shown, since only $\bar{c}$ is updated. The lines on the lower left represent "time slices": all points on a line can be executed in parallel. This choice is minimal with respect to execution time. Note that dependence arrows must not be covered by these lines! The lines in the lower right represent processors: a line contains the sequence of loop steps executed by a fixed processor. These lines may not be parallel to the temporal

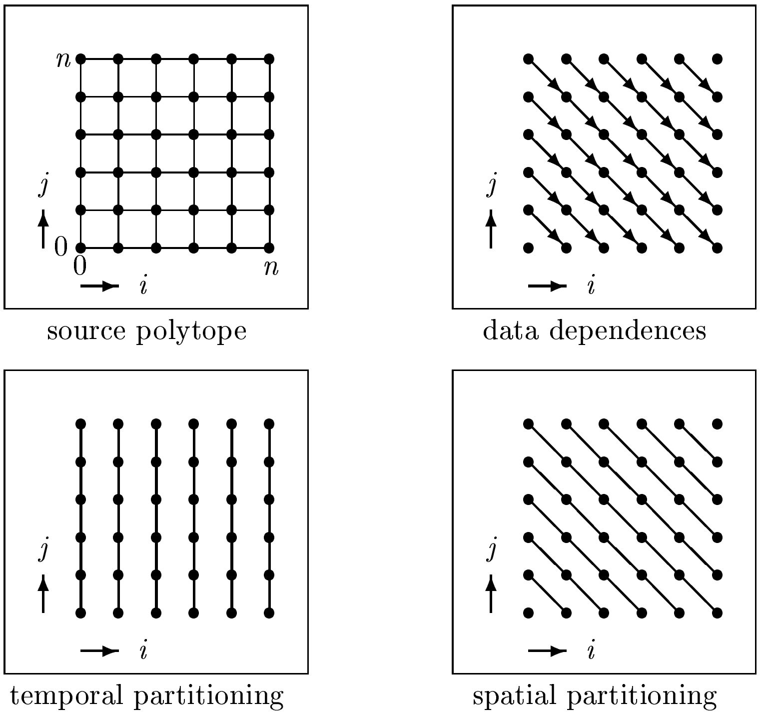
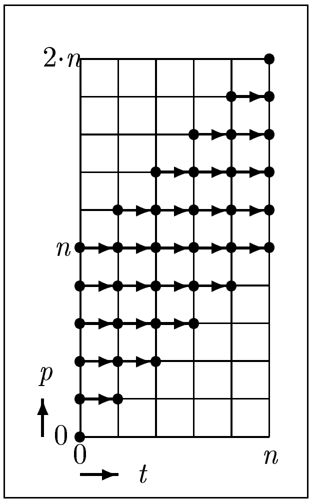


Figure 1: The polytope and its partitionings



synchronous program:

seqfor $t := 0$ to n do
  parfor $p := t$ to $t+n$ do
    $(t, p-t)$

asynchronous program:

parfor $p := 0$ to $2 * n$ do
  seqfor $t := \max(0, p-n)$
        to $\min(n, p)$ do
    $(t, p-t)$

Figure 2: The target polytope and the target programs

lines. We have chosen them to cover the dependences, i.e., we have minimized the number of communication channels between processor (to zero).

The partitionings in time and space can be combined to a *space-time mapping*, an affine transformation to a coordinate system in which each axis is exclusively devoted to time (i.e., represents a sequential loop) or space, i.e. represents a parallel loop. This is like adjusting a "polarizing filter" on the source polytope to make time and space explicit. Fig. 2 depicts the target polytope and two target programs derived from it. Again, we have a choice of order, namely the order of the loops in the nest. If the outer loop is in time and the inner loop in space, we have a *synchronous* program with a global clock, typical in the SIMD style. To enforce the clock tick, we need a global synchronization after each step of the time loop. If we order the loops vice versa, we have an *asynchronous* program, with a private clock for each processor, typical in the SPMD style. Here, we need communication statements to enforce the data dependences, but we have chosen the spatial partitioning such that no communications are required—at the expense of twice the number of processors necessary. In both programs, the code of the loop body is the same as in the source program, only the indices change because the inverse of the space-time mapping must be applied: (i, j) turns into $(t, p-t)$.

3.2 Complexity considerations

The most interesting performance criteria—at least the ones we want to consider here—are execution time, number of processors and overall cost. The cost is defined as the product of execution time and number of processors. One is interested in *cost maintenance*, i.e., in the property that the parallelization does not increase the cost complexity of the program.

With the polytope model, the best time complexity one can achieve is linearity in the problem size:[2] at least one loop must enumerate time, i.e., be sequential. In the pure version of the model, one can usually get away with just one sequential loop [5]. The remaining loops enumerate space, i.e., are parallel. This requires a polynomial amount of processors since the loops bounds are affine expressions.

The cost is not affected by the parallelization: one simply trades between time and space, the product of time and space stays the same.

3.3 Evaluation

Let us sum up the essential properties of the polytope model for the purpose of a comparison:

1. The dependence graph can be embedded into a higher-dimensional space. The dimensionality is fixed: it equals the number of loops in the loop nest.

2. The extent of the dependence graph in each dimension is usually variable: it depends on the problem size. However, a very important property of the polytope model is that the complexity of the optimizing search is independent of the problem size.

3. Each vertex in the dependence graph represents roughly the same amount of work. More precisely, we can put a constant bound on the amount of work performed by any one vertex.

[2] The only exception is the trivial case of no dependences at all, in which all iterations can be executed in one common step.

4. There is a large choice of problem-dependent affine dependences. Thus, given a loop nest with its individual dependences, an automatic optimizing search is performed to maximize parallelism.

5. One can save processors by "trading" one spatial dimension to time, i.e., emulating a set of processors by a single processor.

6. The end result is a nest of sequential and parallel loops with affine bounds and dependences.

7. If one loop is sequential and the rest are parallel (which can always be achieved) [5], one obtains an execution time linear in the problem size. But, to save processors, one can trade space to time at the price of an increased execution time complexity. In particular, one can make the number of processors independent of the problem size by partitioning the resulting processor array into fixed-size "tiles" [13, 39].

4 A divide-and-conquer algorithm

Rather than enforcing a total order on the cumulative summation in specification (2) of the coefficients of the product polynomial, we can accumulate the summands with a D&C algorithm by exploiting the associativity of polynomial addition $\oplus$ on the left side of the following equality:

$$A \odot B \;=\; (A_1 \oplus A_2) \odot (B_1 \oplus B_2) \;=\; (A_1 \odot B_1) \oplus (A_1 \odot B_2) \oplus (A_2 \odot B_1) \oplus (A_2 \odot B_2) \tag{3}$$

We can write this more explicitly, showing the degrees and the variable of the polynomials. Let $m = n \operatorname{div} 2$ and assume for the rest of this paper, for simplicity, that n is a power of 2:

$$\begin{aligned} a \odot b \;&=\; ((ah \cdot z^m) \oplus al) \odot ((bh \cdot z^m) \oplus bl) \\ &=\; ((ah \odot bh) \cdot z^n) \oplus ((ah \odot bl) \cdot z^m) \oplus ((al \odot bh) \cdot z^m) \oplus (al \odot bl) \end{aligned} \tag{4}$$

The suffix l stands for "lower part", h for "higher part" of the polynomial; a and b are the input polynomials.

The dependence graph of this algorithm is depicted in Fig. 3; c, d, e, and f, the resulting polynomials of the four subproblems. This is our starting point for a parallelization which will give us sublinear execution time.

The fundamental difference in the parallelization of D&C as opposed to nested loops is that there is no choice of dependences: the dependence graph is always the same as the call graph. Since we have no problem-dependent dependences, we have no need for an automatic parallelization based on them. Instead, we can take the skeleton approach [9]: we can provide a program schema, a so-called *algorithmic skeleton*, for D&C which is to be filled in with further program pieces—we call them *customizing functions*—in order to obtain a D&C application. Then, our task is to offer for the D&C skeleton one or several high-quality parallel implementations, we call these *architectural skeletons*. This may, again, involve a search, but the search space is problem-independent and, thus, need not be redone for every application. For the user at the application end, the only challenge that remains is to cast the problem in the form of the algorithmic skeleton. Alternatively, the user might develop an architectural skeleton with even better performance by exploiting problem-specific properties of his/her application.

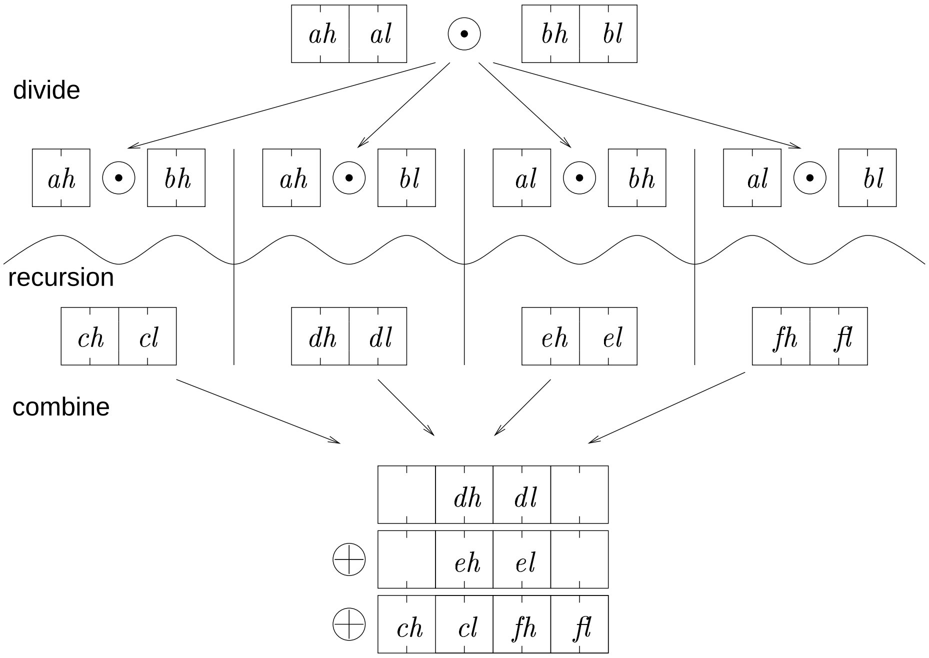


Figure 3: Call graph of the D&C polynomial product

Research on the parallelization of D&C is at an earlier stage than that of nested loops. Many different algorithmic skeletons—and even more architectural skeletons—can be envisioned. No common yardstick by which to evaluate them has been found as of yet.

We discuss two algorithmic skeletons and the respective approaches to their parallelization.

4.1 Space-time mapping D&C

4.1.1 From the Source Program to the Target Program

The call graph in Fig. 3 matches an algorithmic skeleton for call-balanced fixed-degree D&C which we have developed. The skeleton is phrased as a higher-order function in Haskell [25]:

$$
\begin{array}{l}
\mathit{divcon} :: \mathsf{Int} \to (\alpha \to \beta) \to (\alpha \to \alpha \to [\alpha]) \to ([\beta] \to (\beta,\beta)) \to [\alpha] \to [\beta] \\
\mathit{divcon}\ k\ \mathit{basic}\ \mathit{divide}\ \mathit{combine} \ = \ \mathit{solve} \\
\quad \mathsf{where}\ \mathit{solve}\ \mathit{indata} \ = \ \mathsf{if}\ \mathit{length}\ \mathit{indata} \equiv 1 \\
\qquad\qquad\qquad\qquad\qquad \mathsf{then}\ \ \mathit{map}\ \mathit{basic}\ \mathit{indata} \\
\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \ \mathsf{let}\ x = \mathit{zipWith}\ \mathit{divide}\ (\mathit{left}\ \mathit{indata})\ (\mathit{right}\ \mathit{indata}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad y = \mathit{transpose}\ (\mathit{kmap}\ k\ \mathit{solve}\ (\mathit{transpose}\ x)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ \ (\lambda l.\mathit{map}\ \mathit{fst}\ l +\!\!+ \mathit{map}\ \mathit{snd}\ l)\ (\mathit{map}\ \mathit{combine}\ y) \\
\qquad\quad \mathit{kmap}\ k\ f\ l \ = \ \mathsf{if}\ \mathit{length}\ l \equiv k\ \ \mathsf{then}\ \ \mathit{map}\ f\ l\ \ \mathsf{else}\ \ \mathsf{error}\ \text{“list length”} \\
\qquad\quad \mathit{left}\ l \ = \ \mathit{take}\ (\mathit{div}\ (\mathit{length}\ l)\ 2)\ l \\
\qquad\quad \mathit{right}\ l \ = \ \mathit{drop}\ (\mathit{div}\ (\mathit{length}\ l)\ 2)\ l \\
\qquad\quad \mathit{transpose} \ = \ \mathit{foldr}\ (\lambda xs.\lambda xss.\mathit{zipWith}\ (:)\ xs\ (xss +\!\!+ \mathit{repeat}\ [\,]))\ [\,]
\end{array}
$$

Let us comment on a few functions which you will see again in this paper:

map op xs applies a unary operator *op* to a list *xs* of values and returns the list of the results; *map* is one main source of parallelism in higher-order programs.

$xs \mathbin{+\!\!+} ys$ returns the concatenation of list *xs* with list *ys*.

zip xs ys "zips" the lists *xs* and *ys* to a list of corresponding pairs of elements.

The further details of the body of *divcon* are irrelevant to the points we want to make in this paper—and indeed irrelevant to the user of the skeleton. All that matters is that it is a higher-order specification, which specifies a generalized version of the schema depicted in Fig. 3, and whose parameters k, *basic*, *divide* and *combine* the caller fills with the division degree and with appropriate functions for computing in the basic case and dividing and combining in the recursive case.

The example below shows, using the names in Fig. 3, how to express the polynomial product in terms of the *divcon* skeleton. The polynomials to be multiplied have to be represented as lists of their coefficients in order:

$$
\begin{array}{lll}
(\odot) :: \mathsf{Num}\ \alpha\ \Rightarrow\ [\alpha] \to [\alpha] \to [\alpha] & & \\
x \odot y\ =\ \mathit{postadapt}\ (\mathit{divcon}\ 4\ \mathit{basic}\ \mathit{divide}\ \mathit{combine}\ (\mathit{preadapt}\ x\ y)) & & \\
\quad \mathsf{where}\ \mathit{preadapt}\ x\ y & = & \mathit{zip}\ x\ y \\
\qquad\quad \mathit{postadapt}\ z & = & \mathit{map}\ \mathit{fst}\ z \mathbin{+\!\!+} \mathit{map}\ \mathit{snd}\ z \\
\qquad\quad \mathit{basic}\ (a, b) & = & (0, a * b) \\
\qquad\quad \mathit{divide}\ (\mathit{ah}, \mathit{bh})\ (\mathit{al}, \mathit{bl}) & = & [(\mathit{ah}, \mathit{bh}), (\mathit{ah}, \mathit{bl}), (\mathit{al}, \mathit{bh}), (\mathit{al}, \mathit{bl})] \\
\qquad\quad \mathit{combine}\ [(\mathit{ch}, \mathit{cl}), (\mathit{dh}, \mathit{dl}), (\mathit{eh}, \mathit{el}), (\mathit{fh}, \mathit{fl})] & & \\
 & = & ((\mathit{ch}, \mathit{dl}+\mathit{el}+\mathit{fh}), (\mathit{dh}+\mathit{eh}+\mathit{cl}, \mathit{fl}))
\end{array}
$$

The skeleton takes as input and delivers as output lists of size n. The operands and result of the polynomial product have to be formatted accordingly: function *preadapt* zips both input polynomials to a single list, and function *postadapt* unpacks the zipped higher and lower parts for the result again.

Given unlimited resources, it is clear without a search what the temporal and the spatial partitioning should be: each horizontal layer of the call graph should be one time slice. This seems to suggest a two-dimensional geometrical model with one temporal axis (pointing down) and one spatial axis (pointing sideways). However, it pays to convert the call graph to a higher-dimensional structure. The reason is that the vertices in the graph represent grossly unequal amounts of work. In other words, the amount of work of any one vertex cannot be capped by a constant: because of the binary division of data, a node in some fixed layer of the divide phase represents double the amount of work as a node in the layer below, and the reverse applies for the combine phase. This behaviour holds for all algorithms, which fit into this skeleton.

We obtain a graph in which the work a node represents is bounded by a constant if we split a node which works on aggregate data into a set of nodes each of which works on atomic data only. This fragmentation of nodes is spread across additional dimensions, yielding the higher-dimensional graph of Fig. 4. Time now points into depth and, for the given size of the call graph, each time slice is two-dimensional, not just one-dimensional. With increasing problem size, further spatial dimensions are added.

A parallel loop program, which scans this graph in a similar manner as it would scan a polytope, can be derived [25]; here, r is the number of recursive calls ($r = \log n$), the elements

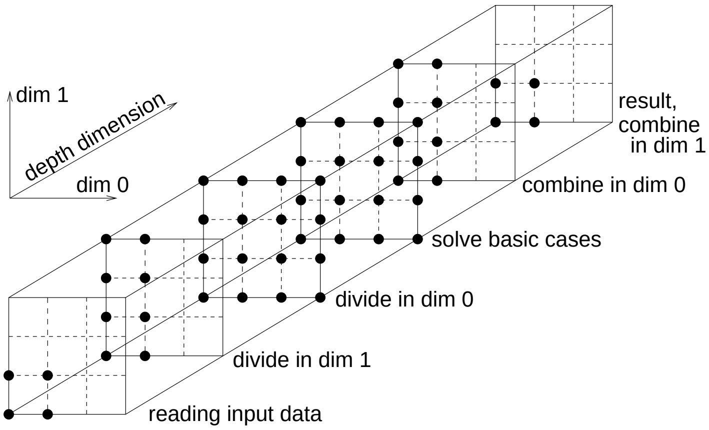


Figure 4: Higher-dimensional call graph of the D&C polynomial product

of AB are pairs of input coefficients, and the elements of C are pairs of output coefficients of the higher and lower result polynomial:

seqfor $d = 1$ to r do
    parfor $q = 0$ to $k^r - 1$ do
        $AB[d, q] = \mathit{divide}[q_{(k)}[r - d]]\ (AB[d-1, (q_{(k)}; r-d:0)],$
        $AB[d-1, (q_{(k)}; r-d:1)])$;
parfor $q = 0$ to $k^r - 1$ do
    $C[r+1, q] = \mathit{basic}\ AB[r, q]$;

seqfor $d = r+2$ to $2*r+1$ do
    parfor $q = 0$ to $k^r - 1$ do
        $C[d, q] = \mathit{combine}[q_{(k)}[d-(r+2)]]\ (C[d-1, (q_{(k)}; d-(r+2):0)], ...,$
        $C[d-1, (q_{(k)}; d-(r+2):k-1)])$

The program consists of a sequence of three loop nests, for the divide, sequential, and combine phase.

The loops on d enumerate the levels of the graph and are therefore sequential, whereas the loops on q are parallel because they enumerate the spatial dimensions. The spatial dimensions are indexed by the digits of q in radix k representation. This allows us to describe iterations across an arbitrary number of dimensions by a single loop, which makes the text of the program independent of the problem size. $q_{(k)}$ denotes the vector of the digits of q in radix k. $q_{(k)}[d]$ selects the dth digit. In accesses of the values of the points whose index differs only in dimension d, we use the notation $(q_{(k)}; d : i)$ for the number, which one obtains from q by replacing the dth digit by i. This representation differs from target programs in the polytope model, in which each dimension corresponds to a separate loop.

The data is indexed with a time component d and a space component q. The divide and combine functions are given the actual coordinate as a parameter in order to select the appropriate functionality for the particular subproblem for *divide* resp. data partition for *combine*.

Such loop programs can also be derived formally by equational reasoning [26].

This program is data-parallel. Therefore, it can be implemented directly on SIMD machines. Additionally, the program can be transformed easily to an SPMD program for parallel machines with distributed memory using message passing. The two-dimensional arrays now become one-dimensional, because the space-component has been projected out by selecting a particular processor. The all-to-all communications are restricted to groups of k processors. Processor q executes the following:

seqfor $d = 1$ to r do
    all-to-all $AB[d-1]$ along virtual dimension $r - d$;
    $AB[d] = \mathit{divide}[q_{(k)}[r-d]]$ (list of values received by all-to-all) ;
$C[r+1] = \mathit{basic}\ AB[r]$;
seqfor $d = r+2$ to $2*r+1$ do
    all-to-all $C[d-1]$ along virtual dimension $d-(r+2)$;
    $C[d] = \mathit{combine}[q_{(k)}[d-(r+2)]]$ (list of values received by all-to-all)

One is interested in transforming the computation domain in Fig. 4 further in two ways:

1. As in loop parallelization, this approach has the potential of very fine-grained parallelism. As in loop parallelization, spatial dimensions can be moved to time to save processors. This is more urgent here, since the demand for processors grows faster with increasing problem size.

2. If the spatial part of the computation domain remains of higher dimensionality after this, its dimensionality can be reduced as depicted in Fig. 5. This is done, e.g., if the target processor topology is a mesh. It works because the extent of each dimension is fixed.

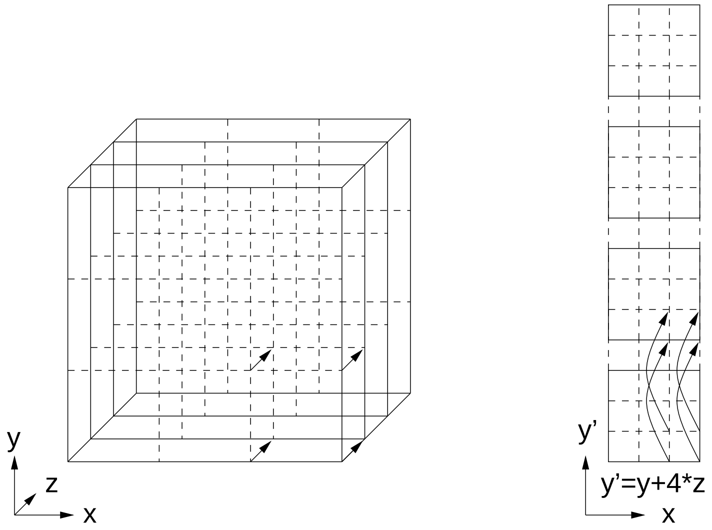


Figure 5: Dimensionality reduction of the computation domain

4.1.2 Complexity Considerations

For the implementation, it is not very efficient to assign each basic problem to a separate physical processor. Instead, spatial dimensions are mapped to time. The result is s slightly modified SPMD program. In brief, the operations on single elements are replaced by operations on segments of data.

Let n be the size of the polynomials, and P the number of processors. In the basic phase on segments, the work is equally distributed among the processors, i.e., each processor is responsible for $n/2^{\log P/\log 4} = n/\sqrt{P}$ elements. The time for computing the basic phase on segments is therefore $O(n^2/P)$. The computation in the divide and combine phase takes $O(\log n)$ steps. In each step, a segment of size $n/\sqrt{P}$ has to be divided or combined in parallel. The entire computation time for both phases is therefore in $O(\log n{\cdot}n/\sqrt{P})$. Then, if the dimensionality of the target mesh equals the number of dimensions mapped to space, the total time is in:

$$O\left(\frac{n^2}{P} + \log n{\cdot}\frac{n}{\sqrt{P}}\right)$$

The execution time is sublinear if the number of processors is asymptotically greater than the problem size, and it maintains the cost of $O(n^2)$ if the number of processors is asymptotically not greater than $O(n^2/\log^2 n)$. The best execution time, which can be achieved under cost maintenance is $O(\log^2 n)$.

If the dimensionality of the target topology is taken into account, our calculations have revealed the following:

1. Sublinear execution time can only be achieved if the dimensionality is at least 3.

2. The computation is sublinear and cost-maintaining for a cubic three-dimensional mesh if the number of processors is asymptotically between n and $n^{1.2}$.

There is an algorithm for polynomial product with a sequential time complexity of $O(n^{\log 3})$, the so-called Karatsuba algorithm [1, Sect. 2.6]. It has a division degree of 3 and can be expressed with our skeleton, with whose parallel implementation [25] the algorithm is cost-maintaining for reasonable problem sizes. Our experiments have shown that the sequential version of the Karatsuba algorithm beats the sequential version of our conventional algorithm if both polynomials have a size of at least 16. Since its subproblems are slightly more computation-intensive, the parallel version of the Karatsuba algorithm (with our skeleton) is a bit slower than the parallel version of the conventional algorithm, but one saves processors (precisely, $4^r - 3^r$ many).

4.1.3 Evaluation

What are the properties of this space-time mapping model, compared with the polytope model of the previous section?

1. The dimensionality of the call graph is variable: it equals the number of layers of the divide phase, which depends on the problem size.

2. The extent of the dependence graph in each spatial dimension is fixed: it is the degree of the problem division.

3. Each vertex in the call graph represents the same amount of work.

4. There is no choice of dependences and no search for parallelism is necessary.

5. The only variety in parallelism is given by the option to trade off spatial dimensions to time.

6. The end result is, again, a nest of sequential and parallel loops.

7. The upper bound of the temporal loop is logarithmic in the problem size, and the upper bound of the spatial loop is exponential in the upper bound of the temporal loop, i.e., polynomial in the problem size. When looking at the computation domain (Fig. 4), the extent of each spatial dimension is constant, but the number of spatial dimensions grows with the problem size.

   Sublinear execution time (in a root of the problem size) are possible on mesh topologies, but the conditions for maintaining cost-optimality in this case are very restrictive.

4.2 Homomorphisms

A very simple D&C skeleton is the homomorphism. It does not capture all D&C situations, and it is defined most often for lists [7, 37], although it can also be defined for other data structures, e.g., trees [17] and arrays [33].

4.2.1 From the source program to the target program

Unary function h is a *list homomorphism* [7] iff its value on a concatenation of two lists can be computed by combining the values of h on the two parts with some operation $\circledast$:

$$h\,(x +\!\!\!+\, y) \;=\; h\,x \;\circledast\; h\,y \tag{5}$$

The significance of homomorphisms for parallelization is given by the promotion property, a version of which is as follows:

$$h \;=\; \mathit{red}\,(\circledast) \;\circ\; \mathit{map}\,h \;\circ\; \mathit{dist} \tag{6}$$

This equality is also proved by equational reasoning. In the literature, one has used the Bird-Meertens formalism (BMF) [7], an equational theory for functional programs in which *red* and *map* are the basic functions on lists: *red* reduces a list of values with a binary operator (which, in our case, inherits associativity from list concatenation) and returns the result value, and *map* we have seen in the previous subsection. Both *red* and *map* have a high potential for parallelism: *red* can be performed in time logarithmic in the length of the list and *map* can be performed in constant time, given as many processors as there are list elements.

The third function appearing in the promotion property, *dist* (for *dist*ribute), partitions an argument list into a list of sublists; it is the right inverse of reduction with concatenation: $\mathit{red}\,(+\!\!\!+) \;\circ\; \mathit{dist} \;=\; \mathit{id}$.

Equality (6) reveals that every homomorphism h can be computed in three stages: (1) an input list is distributed, (2) function h is computed on all sublists independently, (3) the results are combined with operator $\circledast$. The efficiency of this parallel implementation depends largely on the form of operation $\circledast$. E.g., there is a specialization of the homomorphism skeleton, called DH (for *d*istributable *h*omomorphism), for which a family of practical, efficient implementations exists [19, 21].

The similarity of (3) and (5) is obvious: h should be the polynomial product $\odot$, and operation $\circledast$ should be polynomial addition $\oplus$. However, there are two mismatches:

1. Operations $\odot$ and $\oplus$ are defined on polynomials of possibly different degree. Thus, the list representation of a polynomial needs to be refined to a pair $(int, list)$, where *int* is the power of the polynomial and *list* is the list of the coefficients. For simplicity, we ignore this subtlety in the remainder of this paper.

2. A more serious departure from the given homomorphism skeleton is that polynomial product $\odot$, our equivalent of h, requires two arguments, not one. To match this with the skeleton, we might give h a list of coefficient pairs, but this destroys the homomorphism property: in the format provided by h, the product of two polynomials cannot be constructed from the products of the polynomials' halves.

It is just as well that we cannot fit the (binary) polynomial product into the (unary) homomorphism skeleton. The second argument gives us an additional dimension of parallelism which the unary homomorphism cannot offer: because we have two arguments each of whom is to be divided, we obtain four partial results to be combined rather than two, as prescribed by the skeleton. To exploit the quadratic parallelism, we use a different, binary homomorphism skeleton:

$$h2(x{+\!\!+}y, u{+\!\!+}v) = h2(x,u) \circledast h2(y,u) \circledast h2(x,v) \circledast h2(y,v) \tag{7}$$

Now, the polynomial product fits perfectly. The resulting promoted, two-dimensional skeleton does everything twice—once for each dimension: dividing (with *dist*), computing in parallel (with *map*) and combining (with *red*); we write $map^2 f$ for $map\,(map\,f)$ and zip^2 for $map\,(zip) \circ zip$:

$$\begin{aligned}
h2 &= combine \circ compute \circ distribute \\
&\quad \text{where} \\
distribute &= zip^2 \circ (copy \circ dist \times map\,(copy) \circ dist) \\
compute &= map^2\, h2 \\
combine &= red^2\,(\circledast)
\end{aligned} \tag{8}$$

The complication of *distribute* is due to the fact that list portions must be spread out across two dimensions now. Note also that we have not provided a definition of the two-dimensional reduction: $red^2\,(\circledast)$ can be computed in two orders: row-major or column-major. Actually, each of the two choices leads to an equal amount of parallelism.

However, by a problem-specific optimization of the *combine* stage, we can do even better. Fig. 6 depicts this optimized solution on a virtual square of processors (we make no assumptions about the physical topology). Exploiting the commutativity of the customizing operator $\oplus$, we can reduce first along the diagonals—the corresponding polynomials have equal power—and then reduce the partial results located at the northern and eastern borders. The latter step can be improved further to just pairwise exchange between neighbouring processors if we allow for the result polynomial product to be distributed blockwise along the border processors.

The three-stage format of the promoted homomorphism skeleton suggests an SPMD program which also has three stages. For the binary homomorphism, optimized for the polynomial product, every processor q executes the following MPI-like program:

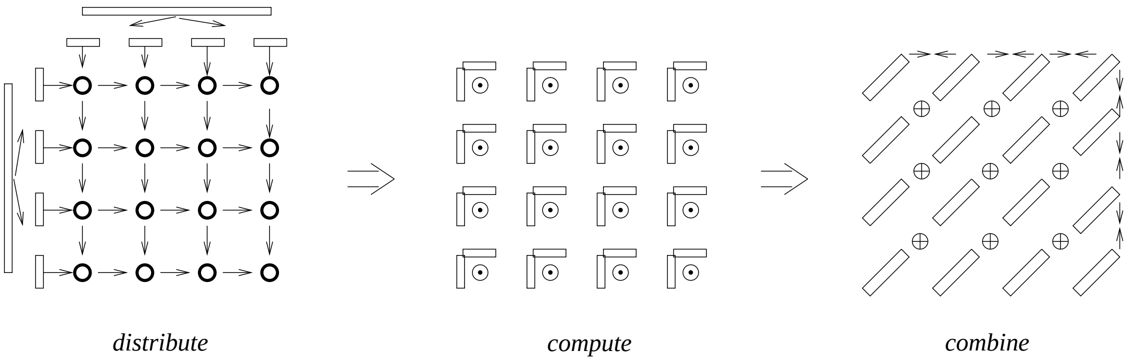


Figure 6: Three stages of the two-dimensional promoted homomorphism skeleton

/* Input: A, B (Block) */
  broadcast (A) in row ;
  broadcast (B) in column ;
  $C := PolyProd(A, B)$;
  reduce $(\oplus)$ in diagonal(q) ;
  exchange-neighbours
/* Output: C (Block) */

This program does not show loops explicitly—but, of course, they are there. The outer, spatial loops are implicit in the SPMD model and the inner, sequential loop is hidden in the call of function *PolyProd* which is a *sequential* implementation of $\odot$. With MPI-like communications, this SPMD program could be the point where the programmer stops the refinement and the machine vendor takes over. Alternatively, the user can him/herself program broadcasts, reductions and exchanges with neighbours and define a suitable physical processor topology, e.g., a mesh of trees [18].

4.2.2 Complexity considerations

We consider multiplying two polynomials of degree n on a virtual square of p^2 processors. The time complexity with pipelined broadcasting and reduction is [18]:

$$t = O\left(\frac{n \cdot \log p}{p \cdot m(n,p)} + (n/p)^2\right)$$

where $m(n,p) = \min(n/p, (\log p + 1))$. The value of p can be chosen between 1 and n.

If $p = n/\sqrt{\log n}$ then $t = O(\log n)$, which yields the optimal, logarithmic time complexity on $p^2 = (n^2/\log n)$ processors. The cost is $O(n^2)$, and is thus maintained.

Other interesting cases are:

- $p = 1$: we get $t = O(n^2)$, so the parallelization has not worsened the sequential cost;
- $p = n$: we achieve $t = O(\log n)$ on n^2 processors; this solution yields the optimal time but is clearly not cost-maintaining;
- $p = n/\log n$: a cost-maintaining solution with $t = O(\log^2 n)$ on $(n/\log n)^2$ processors;

- $p = \sqrt{n}$: $t = O(n)$ on n processors: equal to the systolic solution of Sect. 3 and cost-maintaining.

In practice, the processor number is an arbitrary fixed value and the problem size n is relatively large. Then the term $(n/p)^2$ dominates in the expression of the time complexity which guarantees a so-called *scaled linear speed-up* [35]. This term can be improved to $O((n/p)^{\log 3})$ or to $O((n/p) \cdot \log(n/p))$ by applying the Karatsuba or FFT-based algorithm, respectively, in the processors at the *compute* stage.

Whether the Karatsuba algorithm can be phrased as a homomorphism is an open question.

4.2.3 Evaluation

Let us review the parallelization process in the homomorphism approach. Actually, it is not very different from the skeleton approach of the previous section. One just uses a different skeleton and is led in the parallelization by algebra rather than by geometry.

1. The homomorphism skeleton is more restrictive than the Haskell skeleton in the previous section, and also more restrictive than the earliest D&C skeleton [34], in which it corresponds to a postmorphism (see [21] for a classification of D&C skeletons). The strength of the homomorphism is its direct correspondence with a straight-forward three-stage SPMD program. For the polynomial product, it yields a parallel implementation which is both time-optimal and cost-maintaining.

   Time optimality cannot be achieved on a mesh topology with constant dimensionality. In the homomorphism approach, we obtain a topology-independent program with MPI-like communication primitives. The best implementations of these primitives on topologies such as the hypercube and the mesh of trees are time-optimal [27].

2. As many other skeletons, the homomorphism skeleton can come in many different varieties: for unary, binary, ternary operations, for lists and other data structures. At the present state of our understanding, all these versions are developed separately.

3. Even if all these skeletons are available to the user, an adaptation problem remains. This is true for the previous approaches as well. E.g., in loop parallelization, a dependence which does not satisfy the restrictions of the model is replaced by a set of dependences which do [29]. In the previous subsection, we format the input and the output of polynomial product with adaptation functions to make it match with our Haskell skeleton.

   The homomorphic form of a problem may exist but be not immediately clear. An example is *scan* [20]. Other algorithms can be turned into a D&C and, further, into a homomorphic form with the aid of auxiliary functions [10, 38].

4. The application of the promotion property gives us a parametrized granularity of parallelism which is controlled by the size of the chunks in which distribution function *dist* splits the list. Depending on the available number of processors, input data can be distributed more coarsely or finely, down to a single list element per processor. The only requirement on the number of processors in the case of the two-dimensional homomorphism is that it be a square, which is not as restrictive as in the skeleton of the previous subsection, where a power of the division degree k is required. Moreover, homomorphic solution is not restricted to polynomials whose length is a power of 2.

5. Note that the promotion property is only applied once—as opposed to the previous subsection, in which we parallelize at each level of the call graph. This decreases the amount of necessary communication. The number of parallelized levels depends on the choice of granularity; all remaining levels are captured by a call of the sequential implementation *PolyProd* of $h2$. This enables an additional optimization: processors can call an optimal sequential algorithm for the given problem, rather than the algorithm which was chosen for the parallelization. In our case, *PolyProd* can be the more efficient Karatsuba or FFT-based algorithm for the polynomial product.

5 Summary

Let us summarize the present state of the art of the static parallelization of loops and recursions, as illustrated with the polynomial product.

Static parallelization works best for programs which exhibit some regular structure. The structural requirements can be captured by restrictions on the form of the program, but many applications will not satisfy these restrictions immediately. Thus, static parallelization is definitely not for "dusty decks".

However, many algorithms can be put into the required form and parallelized. In particular, certain computation-intensive application domains like image, signal, text and speech processing, numerical and graph algorithms are candidates for a static parallelization. Dense data structures hold more promise of regular dependences, but sparse data structures might also be amenable to processing with `while` loop nests or with less regular forms of parallel D&C.

5.1 Loop parallelization

Loop parallelization is much better understood than the parallelization of divide-and-conquer. The polytope model has been extended significantly recently:

1. Dependences and space-time mappings may be piecewise affine (the number of affine pieces must be constant, i.e., independent of the problem size) [16].
2. Loop nests may be imperfect (i.e., not all computations must be in the innermost loop) [16].
3. Upper loop bounds may be arbitary and, indeed, unknown at compile time [23].

A consequence of (3) is that `while` loops can be handled [12, 22, 30]. This entails a serious departure from the polytope model.

The space-time mapping of loops is becoming a viable component of parallelizing compilation [31]. Loop parallelizers that are based on the polytope model include Bouclettes [8], LooPo [24], OPERA [32], Feautrier's PAF, and PIPS [2]. However, recent sophisticated techniques of space-time mapping have not yet filtered through to commercial compilers. In particular, automatic methods for partitioning and projecting (i.e., trading space for time) need to be carried through to the code generation stage. Most large academic parallelizing compilation projects involve also loop parallelization techniques that are not phrased in (or even based on) the polytope model. Links to some of them are provided in the Web pages cited here.

A good, unhurried introduction to loop parallelization with an emphasis on the polytope model is the book series of Banerjee [4, 5, 6].

5.2 Divide-and-conquer parallelization

For the parallelization of D&C, there is not yet a unified model, in which different choices of parallelization can be evaluated with a common yardstick and compared with each other. The empirical approach taken presently uses skeletons: algorithm patterns with a high potential for parallelism are linked with semantically equivalent architectural patterns which provide efficient implementations of these algorithms on available parallel machines. This approach makes fewer demands on compiler technology. However, it expects the support of a "systems programming" community which provides architectural skeletons for existing parallel machines. The application programmer can then simply look for the schema in a given skeleton library, and adapt his/her application to this schema.

In the last couple of years, the development and study of skeletons has received an increasing amount of attention and a research community has been forming [11]. The skeleton approach can become a viable paradigm for parallel programming if

1. the parallel programming community manages to agree on a set of algorithmic skeletons which capture a large number of applications and are relatively easy to fill in, and
2. the parallel machine vendors community, or some application sector supporting it, succeeds in providing efficient implementations of these skeletons on their products.

One can attempt to classify the best parallel implementations of some skeleton, which represents a popular programming paradigm, by tabulating special cases. We have done this for the paradigm of linear recursion [41]. The interesting special cases are *copy*, *red* and *scan*. Compositions of these cases can be optimized further.

5.3 Conclusions

Ultimately, one will have to wait and see whether the static or some dynamic approach to parallelism will win the upper hand. Since parallelism stands for performance, the lack of overhead and the precision of the performance analysis are two things in favor of static as opposed to dynamic parallelism—for problems which lend themselves to a static parallelization.

Acknowledgements

This work received financial support from the DFG (projects *RecuR* and *RecuR2*) and from the DAAD (ARC and PROCOPE exchange programs). Thanks to J.-F. Collard for a reading and comments. The Parsytec GCel 1024 of the Paderborn Center for Parallel Computing (PC^2) was used for performance measurements.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Series in Computer Science and Information Processing. Addison-Wesley, 1974.

[2] C. Ancourt, F. Coelho, B. Creusillet, F. Irigoin, P. Jouvelot, and R. Keryell. PIPS: A framework for building interprocedural compilers, parallelizers and optimizers. Technical Report A/289, Centre de Recherche en Informatique, Ecole des Mines de Paris, April 1996. WWW: http://www.cri.ensmp.fr/~pips/.

[3] D. G. Baltus and J. Allen. Efficient exploration of nonuniform space-time transformations for optimal systolic array synthesis. In L. Dadda and B. Wah, editors, *Proc. Int. Conf. Application Specific Array Processors (ASAP'93)*, pages 428–441. IEEE Computer Society Press, 1993.

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

[5] U. Banerjee. *Loop Parallelization*. Series on Loop Transformations for Restructuring Compilers. Kluwer, 1994.

[6] U. Banerjee. *Dependence Analysis*. Series on Loop Transformations for Restructuring Compilers. Kluwer, 1997.

[7] R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, NATO ASI Series F: Computer and Systems Sciences, Vol. 55, pages 151–216. Springer-Verlag, 1988.

[8] P. Boulet, M. Dijon, E. Lequiniou, and T. Risset. Reference manual of the Bouclettes parallelizer. Technical Report 94-04, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, October 1994. WWW: http://www.prism.uvsq.fr/public/bop/base/bclt/bouclettes.html.

[9] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[10] M. I. Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2):191–204, June 1995.

[11] M. I. Cole, S. Gorlatch, C. Lengauer, and D. B. Skillicorn, editors. Theory and practice of higher-order parallel programming. Technical Report 169, Schloß Dagstuhl, February 1997.

[12] J.-F. Collard. Automatic parallelization of `while`-loops using speculative execution. *Int. J. Parallel Programming*, 23(2):191–219, 1995.

[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. Array expansion. In *Proc. Int. Conf. on Supercomputing*, pages 429–441. ACM Press, 1988.

[16] P. Feautrier. Automatic parallelization in the polytope model. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, Lecture Notes in Computer Science 1132, pages 79–103. Springer-Verlag, 1996.

[17] J. Gibbons. Upwards and downwards accumulations on trees. In R. Bird, C. Morgan, and J. Woodcock, editors, *Mathematics of Program Construction*, Lecture Notes in Computer Science 669, pages 122–138. Springer-Verlag, 1992.

[18] S. Gorlatch. From transformations to methodology in parallel program development: a case study. *Microprocessing and Microprogramming*, 41:571–588, 1996.

[19] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.

[20] S. Gorlatch. Systematic extraction and implementation of divide-and-conquer parallelism. In H. Kuchen and D. Swierstra, editors, *Programming Languages: Implementation, Logics and Programs*, Lecture Notes in Computer Science 1140, pages 274–288. Springer-Verlag, 1996.

[21] S. Gorlatch and H. Bischof. Formal derivation of divide-and-conquer programs: A case study in the multidimensional FFT's. In D. Mery, editor, *Formal Methods for Parallel Programming: Theory and Applications*, pages 80–94. IEEE Computer Society Press, 1997.

[22] M. Griebl. *The Mechanical Parallelization of Loop Nests Containing* while *Loops*. PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1996. Technical report MIP-9701.

[23] M. Griebl and C. Lengauer. Classifying loops for space-time mapping. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96, Vol. I*, Lecture Notes in Computer Science 1123, pages 467–474. Springer-Verlag, 1996.

[24] M. Griebl and C. Lengauer. The loop parallelizer LooPo. In M. Gerndt, editor, *Proc. Sixth Workshop on Compilers for Parallel Computers*, volume 21 of *Konferenzen des Forschungszentrums Jülich*, pages 311–320. Forschungszentrum Jülich, 1996. WWW: http://www.uni-passau.de/~loopo/.

[25] C. A. Herrmann and C. Lengauer. On the space-time mapping of a class of divide-and-conquer recursions. *Parallel Processing Letters*, 6(4):525–537, 1996.

[26] C. A. Herrmann and C. Lengauer. Parallelization of divide-and-conquer by translation to nested loops. Technical Report MIP-9705, Fakultät für Mathematik und Informatik, Universität Passau, March 1997.

[27] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings, 1994.

[28] C. Lengauer. A view of systolic design. In N. N. Mirenkov, editor, *Parallel Computing Technologies (PaCT-91)*, pages 32–46. World Scientific, 1991.

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

[30] C. Lengauer and M. Griebl. On the parallelization of loop nests containing while loops. In N. N. Mirenkov, Q.-P. Gu, S. Peng, and S. Sedukhin, editors, *Proc. 1st Aizu Int. Symp. on Parallel Algorithm/Architecture Synthesis (pAs'95)*, pages 10–18. IEEE Computer Society Press, 1995.

[31] C. Lengauer, L. Thiele, M. Wolfe, and H. Zima, editors. Loop parallelization. Technical Report 142, Schloß Dagstuhl, April 1996.

[32] V. Loechner and C. Mongenet. OPERA: A toolbox for loop parallelization. In I. Jelly, I. Gorton, and P. Croll, editors, *Proc. 1st Int. Workshop on Software Engineering for Parallel and Distributed Systems*, pages 134–145. Chapman & Hall, 1996. WWW: http://icps.u-strasbg.fr/opera/.

[33] R. Miller. *A Constructive Theory of Multidimensional Arrays.* PhD thesis, Programming Research Group, Oxford University, February 1993.

[34] Z. G. Mou and P. Hudak. An algebraic model for divide-and-conquer algorithms and its parallelism. *Journal of Supercomputing*, 2(3):257–278, 1988.

[35] M. J. Quinn. *Parallel Computing.* McGraw-Hill, 1994.

[36] P. Quinton and Y. Robert. *Systolic Algorithms and Architectures.* Prentice-Hall, 1990.

[37] D. B. Skillicorn. *Foundations of Parallel Programming.* Cambridge International Series on Parallel Computation. Cambridge University Press, 1994.

[38] D. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, 8(3):213–229, 1987.

[39] J. Teich and L. Thiele. Control generation in the design of processor arrays. *J. VLSI Signal Processing*, 3(1–2):77–92, June 1991.

[40] Simon Thompson. *Haskell – The Craft of Functional Programming.* International Computer Science Series. Addison-Wesley, 1996.

[41] C. Wedler and C. Lengauer. Parallel implementations of combinations of broadcast, reduction and scan. In G. Agha and S. Russo, editors, *Proc. 2nd Int. Workshop on Software Engineering for Parallel and Distributed Systems (PDSE'97).* IEEE Computer Society Press, 1997.