Parallel Processing Letters

ON THE SPACE-TIME MAPPING OF A CLASS OF DIVIDE-AND-CONQUER RECURSIONS

CHRISTOPH HERRMANN and CHRISTIAN LENGAUER
Fakultät für Mathematik und Informatik
Universität Passau
D-94030 Passau, Germany

Received
Revised
Accepted by

ABSTRACT

We propose a functional program skeleton for balanced fixed-degree divide-and-conquer and a method for its parallel implementation on message-passing multiprocessors. In the method, the operations of the skeleton are first mapped to a geometric computational model which is then mapped to space-time in order to expose the inherent parallelism. This approach is inspired by the method of parallelizing nested loops in the polytope model.

Keywords: divide-and-conquer, functional program, parallelization, skeleton, space-time mapping.

1. Introduction

The divide-and-conquer ($\mathcal{DC}$) paradigm is a special case of cascading recursion which enables efficient solutions to many practical problems like the multiplication of matrices or large integers, Fast Fourier Transform, sorting, etc. We are interested in the parallelization of $\mathcal{DC}$ recursions with the goal of sublinear execution times on a mesh. Sublinearity can only be achieved if the input is read in parallel. We choose a mesh because it is a widely used general-purpose processor topology with a large bisection width. In practice, meshes have only a few dimensions (typically two) and have problems with non-local communication.

For tail recursions (i.e., loops), there is a powerful compile-time parallelization method based on the idea of the space-time mapping of polytopes (finite polyhedra) [8]. The computational model of a loop nest is a *polytope*, which is embedded into a multi-dimensional integer lattice. Each dimension of the polytope corresponds to one loop. The points of the polytope correspond to individual iterations of the loop nest. The idea of a *space-time mapping* refers to coordinate transformations with which time (the schedule) and space (the processor allocation) can be made explicit in this model. However, the polytope model does not allow for an acceleration to (non-constant) sublinear execution time.

Our goal is to approach the parallelization of $\mathcal{DC}$ recursions analogously: we are looking for an appropriate *geometric* computational model in which time and space can be *made explicit* by *formal mappings* whose choice can be guided by certain *optimization criteria*.

Specifically considered is the case of a non-binary division: our case study is on the polynomial product using ternary $\mathcal{DC}$.

2. The Model

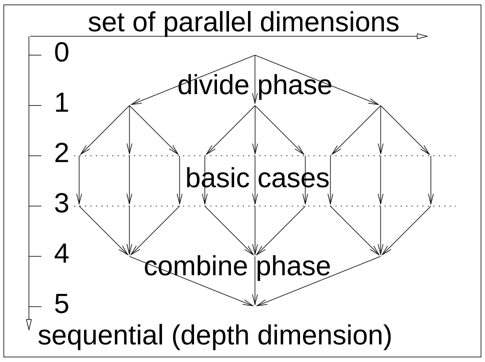

Figure 1: Dependence graph ($n=2, k=3$)

Let the term *fixed-degree divide-and-conquer* [3] describe the class of algorithms which solve a problem by dividing it into a fixed number, say k, of smaller subproblems of the same type, then recursively applying itself to the subproblems until a basic case is reached and, finally, combining the solutions of the subproblems to obtain the solution of the original problem. The basic cases can often be solved by a trivial algorithm, but can also be of the $\mathcal{DC}$ type themselves. Because we want to map statically, i.e., independently of the values of the input data, we assume that the recursion depth is equal for all basic cases. So, for a recursion depth of n, the number of basic cases in our fixed-degree $\mathcal{DC}$ recursions balanced in this way is k^n.

The computation of a balanced fixed-degree $\mathcal{DC}$ algorithm with k-ary division is most easily depicted by its *call tree.* This tree is traversed, from the root down to the leaves, while the problem is being divided. At the leaves, the basic cases are computed, and then the tree is traversed back up to the root while the partial results are being combined. Fig. 1 shows these two traversals of the tree unfolded into a graph, the dependence graph of the $\mathcal{DC}$ algorithm. This graph has a simple (synchronous) parallel execution. The calls at each fixed level are executed in parallel, and the levels are executed in sequence, top-down in Fig. 1.

We choose a computational model which differs from the call tree for the following reason: the nodes in the call tree are not the appropriate candidates for the points in the computational model. In many $\mathcal{DC}$ algorithms, the computational effort grows as partial results are combined on the way from the leaves back to the root. An example is the bitonic sort [7]. Thus, in order to increase the potential parallelism, it is useful to view nodes which are higher up in the tree not as an atomic computation but as a collection of computations whose processing can also be parallelized.

We focus on a restricted format of $\mathcal{DC}$ where only left/right vector partitioning and elementwise vector operations are permitted as divide and combine operations. The underlying data structure is a list whose length is a power of 2, the so-called *powerlist* [9]. Our geometrical model for $\mathcal{DC}$ recursions, which we call the *computation domain*, is embedded into a multi-dimensional integer lattice, like in the

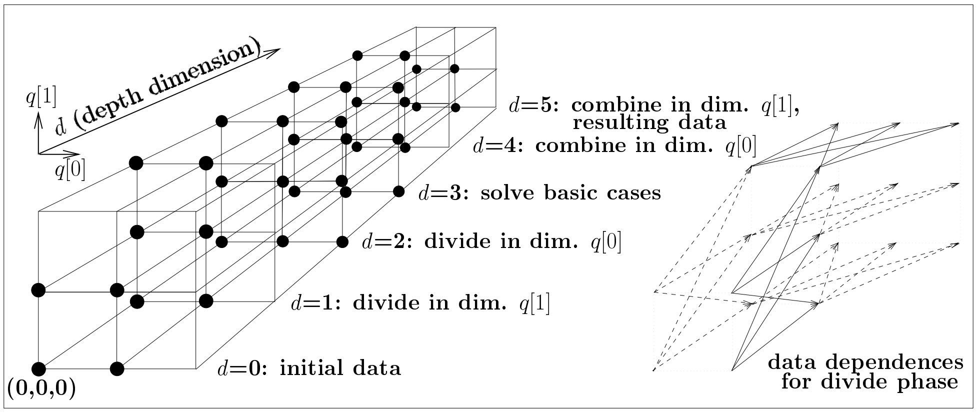

Figure 2: Computation domain ($n = 2$, $k = 3$)

polytope model, but it is not a polytope. As in the polytope model, points in the computation domain represent (roughly equally-sized) pieces of work.

The computation domain has one distinguished dimension, the *depth dimension*, and n dimensions of extent k. The depth dimension must be associated with time. The other dimensions span a k-ary n-cube [4] and can be associated with time or with space, depending on the number of available processors. The coordinates of the depth dimension reflect the *free schedule*, the parallel execution in which each computation is performed as soon as possible. All data dependences can be described by vectors in the computation domain which have at most two non-zero coordinates: the depth coordinate, which is always 1, and one other coordinate which depends only on the depth coordinate of the target point of the dependence. I.e., communication in the k-ary n-cube is restricted to one dimension at a time.

How does this model relate to the call tree? The call tree has $n+1$ levels. Each level of its unfolding (the dependence graph) corresponds to an intersection with a *hyperplane*, i.e., to an n-dimensional "slice" of the computation domain at some depth coordinate (Fig. 2). We distinguish the division of work and the division of data:

- The work is split into k parts, at each level. In many vector-valued D&C functions, the amount of work is nearly equal at each level of the call tree, and the operations within each level can be applied in parallel. In the model, we split the work of each of the k^d nodes at level $d \in \{0, ..., n\}$ into 2^{n-d} computation points.
- The data are always split into two halves. I.e., the input and output is located at a hypercube (binary n-cube) embedded in the k-ary n-cube.

For a better understanding, let us go through the first slices of Fig. 2:

1. In the first slice ($d = 0$), we distribute the input data along both the horizontal and the vertical dimension. We have two times two ($= 2^n$) pieces of data which, together, form the root of the call tree. We could have chosen any four points of the slice; we made our choice based on the regularity of the data dependences the distribution incurs, to facilitate later space-time mapping.

2. In the second slice ($d = 1$), we must choose one dimension to distribute the work. We choose the vertical dimension. The effect is that the data are not distributed in this dimension anymore; see the right of Fig. 2 for the data dependences. We obtain two times three pieces of computation (corresponding to the second level of the call tree).
3. In the third slice ($d = 2$), we must distribute the work again, this time across the only dimension left, the horizontal dimension. Again, see the right of Fig. 2 for the data dependences. Because of our requirement of balance, this fills all grid points with work (corresponding to the third level of the call tree).

3. The Program Skeleton

The following functional program, in the syntax and semantics of Haskell [11], is the specification of our skeleton for balanced fixed-degree $\mathcal{DC}$; to define a specific $\mathcal{DC}$ algorithm, instantiate the constituting functions *basic*, *divide* and *combine* and the degree k appropriately:

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

This skeleton serves as a specification. Its aim is to reflect the structural properties at a high level of abstraction and not to have an efficient execution.

The idea of componentwise vector operations is taken from [10], where it is called “corresponding communication”. Only left/right partitioning of vectors is permitted: the powerlist is split in the middle. If $k > 2$, the collective size of the subproblems at one level of the call tree grows towards the leaves.

Our skeleton supports the idea of distributing the input and output over a subset of the processors. Supplied with the degree k and the constituting functions *basic*, *divide* and *combine*, it takes a single input vector of size 2^n and delivers a single output vector of the same size but, maybe, of a different type. If the function requires more than one argument, as does polynomial product, the different arguments are first zipped, i.e., a powerlist of tuples is used in place of a tuple of powerlists.

Let us now comment on the code of the skeleton. If the input is a list of a single element, the basic function is applied to the element. Otherwise (in the recursive case), the divide function is applied elementwise to the zip of the left and right subvector. The result of this application is a list of half the original length, whose elements consist of k-lists, where k is the degree and there is one list position for each subproblem-specific input vector. To make the recursion work out, x must be transposed to get a k-list of lists. The recursive application to each of the subproblems is done by function ($\mathsf{kmap}\ k$) which applies a function to all elements of a k-list. Next, the k-list of the result vectors is transposed again to undo the preceding transposition, yielding a list of k-lists, where the ith entry of the jth k-list is the jth element of the ith subproblem solution. The combine function is applied elementwise to this list, i.e., its input is a k-list and the left part of the result vector is computed by the first part of the combine function and the right part by the second part.

4. Example: Polynomial Product

In the following, ld stands for $\log_2$ and log for $\log_k$.

In 1962, Karatcuba published a $\mathcal{DC}$ algorithm for the multiplication of large integers of bitsize N with cost $\mathcal{O}(N^{\mathrm{ld}\,3})$ based on ternary $\mathcal{DC}$ [2, Sect. 2.6]. As an example of ternary $\mathcal{DC}$, we choose the polynomial product, which is the part of Karatcuba's algorithm that is responsible for its complexity.

Here, we concentrate on the product of two polynomials which are represented by powerlists of their coefficients in order. The length of both powerlists is the smallest power of two which is greater than the maximum of both degrees. To keep the specification simple, we assume that the built-in data type integer is unbounded. We consider operations $+$, $-$ and $*$ on polynomials; when applying them to integers, we pretend to deal with the respective constant polynomial. If a, b, c and d are polynomials in the variable X of a degree less than $N = 2^{n-1}$, then $(a * X^N + b) * (c * X^N + d) = h * X^{2*N} + m * X^N + l$, where $h = a * c$ (h is for "high"), $l = b * d$ (l is for "low") and $m = (a * d + b * c)$ (m is for "middle"). The ordinary polynomial product uses two subproducts for computing m, leading to quadratic cost, whereas the Karatcuba algorithm uses the equality $m = (a+b)*(c+d) - h - l$ to compute only a single additional polynomial subproduct. Polynomial addition and subtraction does not influence the asymptotic cost because it can be done in parallel in constant time and in sequence in linear time.

Due to the data type and data dependence restrictions imposed by our skeleton, the input vector of the skeleton is the zip of two coefficient vectors and the result is the zip of the higher and lower part of the resulting coefficient vector, as can be seen in the definition of $\mathsf{karatcuba}$, which multiplies two polynomials represented by equal-size powerlists:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{karatcuba}\ a\ b\ =\ (\mathsf{map\ fst}\ y) +\!\!\!+ (\mathsf{map\ snd}\ y)\ \ \mathsf{where}} \\
\multicolumn{3}{l}{\ y\ = \mathsf{divcon\ 3\ basic\ divide\ combine}\ (\mathsf{zip}\ a\ b)} \\
\ \mathsf{basic} & (x,y) & = (0, x*y) \\
\ \mathsf{divide} & (xh,yh)\ (xl,yl) & = [(xh,yh),(xl,yl),(xh+xl,yh+yl)] \\
\ \mathsf{combine} & [(hh,hl),(lh,ll),(mh,ml)] & = ((hh, lh+ml-hl-ll),(hl+mh-hh-lh, ll))
\end{array}
$$

Of the constituting functions, basic multiplies two constant polynomials (of degree 0). Function divide divides a problem into three subproblems: the first is working on the high parts, the second on the low parts and the third on the sum of the high and the low parts, corresponding to $(a+b)$ and $(c+d)$ of the formula for m. The function combine combines the results (hh,hl) (the high parts), (lh,ll) (the low parts) and (mh,ml) (the middle parts). The high positions mh of the middle parts overlap with the low positions hl of the high parts, and the low positions ml of the middle parts with the high positions lh of the low parts. Results of overlapping positions have to be summed. Further, the results of the high and low part have to be subtracted from the result of the middle part.

5. A Nested Loop Program

The user perceives our skeleton through its functional specification. Its implementation is an imperative program consisting of sequential and parallel loops. Important is that the functional skeleton restricts the expressional power of Haskell in a way, that the transformation into a loop program is possible, e.g., it guarantees bounds on the length of the lists, dependent on the input, and under the assumption, that elements of the types α and β can be represented with a constant amount of memory. The loop program enumerates the points of the computation domain, where each of these points is represented by an array element of type α or β. Due to the lack of space, the transformational process is not presented here. The idea is first to transform non-linear recursion into linear recursion, whose correctness can be proved by induction on the recursion depth, and then to use correspondences between language constructs on the functional and imperative side for transforming the linear recursion into loops. The transformations are not purely syntactic but rely heavily on index computations which end up in array indexing schemes.

The loop program we present in this section is not very well suited for a realistic mesh: it may require a processor network of high dimensionality and it permits non-neighboring communication. How to rectify these drawbacks is the subject of the next two sections.

We choose a double loop nest, where the outer loop (on d) enumerates the depth dimension and the inner loop (on q) enumerates *all* other dimensions of the computation domain. This is a departure from traditional loop parallelization, where the depth of the loop nest corresponds to the dimensionality of the computation domain. One reason to insist on this correspondence is the central role which the bounds of the computation domain play in the polytope model. At this point, we are not interested in a similar correspondence for $\mathcal{DC}$ recursions. However, we would like to be able to retrieve the position, i.e., the coordinates of an iteration in the computation domain. Therefore, we represent q as a vector of digits in radix k, written $q_{(k)} = (q_{(k)}[0], ..., q_{(k)}[n-1])$, with the dimensionality of the computation domain (minus the depth dimension).

Let n be defined as above, i.e., let the size of the problem be 2^n. The number of parallel iterations equals the number of base cases, which is k^n. Thus, $q = \sum_{i=0}^{n-1}(q_{(k)}[i] * k^i)$ ranges from 0 to $k^n - 1$ but, to distribute the iterations appropriately across our computation domain, we operate on the vector representation of $q_{(k)}$. We use the notation of [5] for substituting into a vector v: $(v; i : a)$ is equal to v except that it has an a in position i.

If q is a natural number, the notation $q_{(k)}[i]$ in the program is like an abbreviation for $((q \text{ div } k^i) \bmod k)$ and $(q_{(k)}; i : a)$ for $(q+(a - q_{(k)}[i])*k^i)$.

Our loop program has three loop nests; the first corresponds to the divide phase, the second to the basic cases and the third to the combine phase:

seqfor $d = 1, ..., n$
  parfor $q = 0, ..., k^n - 1$
    $A[d, q] = \mathit{divide}[q_{(k)}[n-d]] \quad (A[d-1, (q_{(k)}; n-d : \mathit{left})],$
    $A[d-1, (q_{(k)}; n-d : \mathit{right})])$;
parfor $q = 0, ..., k^n - 1$
  $B[n+1, q] = \mathit{basic}\ A[n, q]$;

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

A and B are arrays, which are indexed by points of the computation domain. The elements of A are of the input data type α and the elements of B are of the output data type β. $\mathit{left}, \mathit{right} \in \{0, ..., k-1\}$, $\mathit{left} \neq \mathit{right}$. Note that we select a component of the vector-valued function *divide* (or *combine*) by normal array indexing. Our index is $q_{(k)}[n-d]$ because, for depth coordinate d of the divide phase, communication takes place only along dimension $n-d$, and similarly in the combine phase. In the divide phase, the dimensions are counted down, in the combine phase they are counted up.

With the loop bounds given in the program and a stride of 1 for each loop, the program enumerates all points of the k-ary n-cube in each slice, also those which are not in the computation domain. This is a standard problem which arises in geometric space-time mapping: how should one treat “irregularities” in the computation domain? Applying the loop body at undefined points does not interfere with the rest of the program but may require additional time (if some dimensions of the computation domain are mapped to time).

6. Mappings to Space-Time

The goal of our space-time mapping is to adapt an algorithm to a parallel architecture of low dimensionality by optimizing with respect to some objective function, and taking into account constraints imposed on the class of (linear and non-linear) mappings (see Sect. 6.1) and constraints defining properties of the implementation.

6.1. Constraints on the space-time mapping

Injectivity (invertibility): The injectivity of the space-time mapping guarantees that the processor array can be made of sequential processors. In the polytope model, it is checked by computing the determinant of the space-time matrix. Here, the check is more difficult since, if the space-time mapping can be put at all into matrix form (for linear mappings), it will not be square because of dimensionality reduction.

Consistency: The time mapping must respect the data dependences.

Feasibility: $time(x) - time(y) \geq distance(space(x),\ space(y))$ if x depends (or is treated as if it would) on y. This condition guarantees that there will always be enough time to get a datum from the source to the destination, where one time unit is defined as the maximum of the execution times of all points in the domain. In a mesh of fixed dimensionality as well as in the hypercube the distance function will be the sum of the weighted absolute component-wise differences, where the weights model the communication costs along the dimensions.

We suggest that the optimization determines a mapping which satisfies the feasibility constraint and the necessary refinement of the communication is left to routing after the optimization, regarding the determined allocation. Additionally, long distances should be assigned high costs to balance communication costs and reduce the critical path.

6.2. Testing the invertibility of bounded linear mappings

Linear mappings are particularly important because of the existence of efficient linear optimization methods. As stated before, the injectivity of a space-time mapping cannot be established by checking a determinant because the mapping might not have a square matrix. Still, even a mapping with a non-square matrix can be injective because of the constant (finite) extent of the domain. The simplest way of checking for injectivity is to examine the mapping at every point of the domain. A more systematic way is to use integer linear programming.

Definition 1 $K_+ = \{0, ..., k-1\}^r$ *and* $K = \{-k+1, ..., k-1\}^r$.

Lemma 1 *(Injectivity test)*
Let $A : K \to \mathbb{Z}^s$ *be linear and* $B : K \to \mathbb{Z}$ *linear and injective. Then* A *is injective on* K_+ *if and only if* $\min\{B\,z \mid z \in K, A\,z = \vec{0}\} = 0$ *and* $\max\{B\,z \mid z \in K, A\,z = \vec{0}\} = 0$.

The proof, omitted due to space limitations, can be obtained from the authors.

Corollary 1 *(Unique representation of a natural number* z *in radix* b*)*
$B : \{0, ..., b-1\}^r \to \mathbb{Z}$;
$B(z_{(b)}[0], ..., z_{(b)}[r-1]) = \sum_{j=0}^{r-1}(z_{(b)}[j] * b^j) \implies (B \text{ is injective})$.

Therefore, we propose the following integer linear program:

- the decision variables are $d[j]$ for $0 \leq j < r$
- the constraints are:
  - $(\forall j : 0 \leq j < r : -k < d[j] < k)$
  - $(\forall i : 0 \leq i < s : \sum_{j=0}^{r-1}(A[i,j] * d[j]) = 0)$, i.e., $A\,d = \vec{0}$
- the objective function is: $\sum_{j=0}^{r-1}((k-1+d[j]) * (2*k-1)^j)$

According to Corollary 1, the objective function is injective (let $b = 2*k-1$ and $z_{(b)}[j] = k-1+d[j]$). We minimize and maximize this objective function via integer linear programming. Lemma 1 states that the value of the objective function is zero in both cases if and only if the mapping is injective.

6.3. Division of space and time

In the polytope model, the dimensionality of the computation domain is fixed and maintained by the space-time mapping; the extent of each dimension depends on the problem size [8]. Here, the dimensionality of the computation domain depends on the problem size and the extent of each (but the depth) dimension is fixed.

For division of space and time, choose a natural m with $0 < m < k$ and r with the property that $m * k^r$ is not greater than the number of processors. The *high* dimensions, which are the first to be involved in communication in the *divide* phase are mapped to space, motivated by the independence property of the subproblems. For simplicity, let $M = \lceil k/m \rceil$ and $R = n-r-1$. Then, the source dimensions can be mapped as follows:

depth	$\rightarrow$	time
$q[n-1], ..., q[n-r]$	$\rightarrow$	space
$q[R]$	$\rightarrow$	space ($q[R]$ div M) and time ($q[R]$ mod M)
$q[R-1], ..., q[0]$	$\rightarrow$	time

The spatial part of the image of dimension $q[R]$ is selected by the div function, in order to map some neighboring points to the same processor to minimize inter-processor communication. The disadvantage is that, in some cases, the first application of the left and right part of the combine function is not parallelized which would become possible if we interchanged div and mod. We made our choice because transmitting a vector element is more expensive than applying a (small) operation to it.

For arbitrary integers x, y and $c > 0$, we know that $(x \text{ div } c = y \text{ div } c) \wedge (x \text{ mod } c = y \text{ mod } c) \implies x = y$. This guarantees the injectivity of the mapping of dimension $q[R]$. If we want to get a linear mapping, then we must choose $m = 1$, so $M = k$.

7. Specific Mapping Techniques

7.1. Refinement of the depth dimension

In this subsection, we describe the transformation of non-neighboring connections to sequences of neighboring connections. Making this our first step simplifies the later mapping phases. Our transformation scheme does not introduce link contentions. We provide a higher resolution of the depth axis: single coordinates are refined to $\lceil k/2 \rceil + 2$ coordinates. A refined coordinate is denoted by a pair: the left part is the coordinate before refinement, and the right part corresponds to a substep along the depth axis. Fig. 3 shows one step of the divide phase with its substeps. The refinement of the combine phase is likewise, only with opposite directions of the communications. In the first substep, the input data, which is distributed over the points with coordinate values *right* $= \lfloor k/2 \rfloor$ and *left* $=$ *right* -1 in a certain dimension is collected in pairs at the points whose values are *right* using only communication along this dimension. In the next $\lceil k/2 \rceil$ substeps, this pair is distributed to all coordinates and the last substep performs, in addition, the application of the respective part of the *divide* function. We interpret the pair notation (d, i) of the refined depth coordinate on a linear scale to be $d * (\lceil k/2 \rceil + 2) + i$. Note that the extent of the depth dimension grows only with a constant factor of $\lceil k/2 \rceil + 2$.

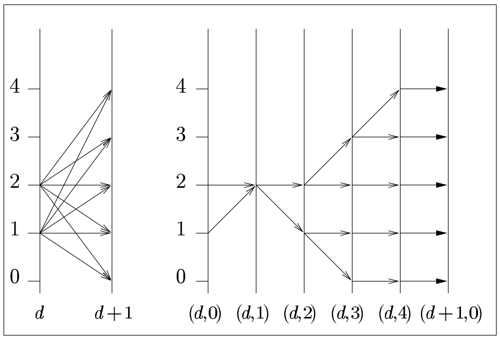

Figure 3: Depth refinement ($k = 5$)

7.2. Reduction of dimensionality

The dimensionality s of the processor array will usually be substantially smaller than r, but with larger extents of each dimension than in the dimensions of the computation domain. Therefore, our space mappings must reduce the dimensionality. In Fig. 4 is shown a reduction from a three-dimensional to a two dimensional space. As can be seen, the length of some dependence vectors increases.

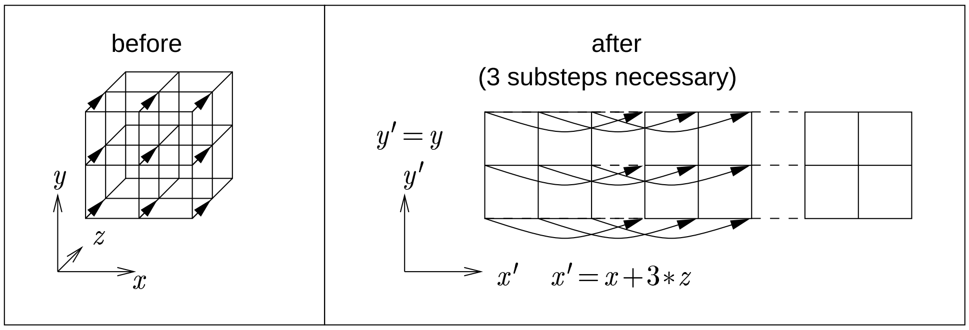

Figure 4: Dimensionality reduction and dependences

8. Complexity Considerations

Let k be the division degree, N the size of the problem (in our setting a power of 2), p the number of processors and, for simplicity, assume $m = 1$. Then $n = \mathrm{ld}\, N$ is the recursion depth, $r = \lfloor \log p \rfloor$ is the number of parallel division steps and $Q = 2^{n-r}$ is the size of the data located on each processor. This size does not change during parallel division or combination, so it is also the size of the problems to be computed sequentially on every processor. The time $T1$ for computing the sequential cases is:

- for $k=2$: $\mathcal{O}(Q * \mathrm{ld}\, Q) = \mathcal{O}(N/p * \mathrm{ld}\,(N/p))$
- for $k>2$: $\mathcal{O}(Q^{\mathrm{ld}\, k}) = \mathcal{O}(k^{n-r}) = \mathcal{O}(N^{\mathrm{ld}\, k}/p)$

The time $T2$ for the parallel divide and combine is in $\mathcal{O}(r * Q) = \mathcal{O}(\log p * N/2^{\log p})$ because we have r parallel division steps. Further, we have to consider the communication overhead introduced by mapping to a lower-dimensional mesh (of dimensionality s). If the mesh has the same extent in each dimension, we can achieve a communication time complexity $T3$ of $\mathcal{O}(Q * s * p^{1/s}) = \mathcal{O}(N/2^{\log p} * s * p^{1/s})$. The overall time complexity is $T1 + T2 + T3$.

$T1$ is sublinear if p is asymptotically greater than $\mathrm{ld}\, N$ in the case of $k=2$ or greater than $N^{(\mathrm{ld}\, k)-1}$ in the case of $k>2$ whereas, for the sublinearity of $T2$, it is sufficient that p increases very slowly with N. Sublinearity of $T3$ requires $s > \mathrm{ld}\, k$ and very slow increase of p with N.

9. Related Work

Contrary to Mou [10], Misra [9], and Achatz and Schulte [1], our approach is not restricted to binary $\mathcal{DC}$ and, contrary to Cole [3], we can handle distributed input and output.

We take the view that it is methodologically important that the algorithmic skeleton be independent of the size of the topology, and we prefer to leave the task of distinguishing between the sequential cases of the implementation and the basic cases of the algorithm (depending on the problem size and the topology) to the compiler. Therefore, we do not propose an architectural skeleton but a mapping for an algorithmic skeleton. The mapping can be selected semi-automatically by optimizing an objective function.

10. Conclusions

We have shown that the mapping of a special class of divide-and-conquer algorithms to meshes with distributed input and output and parallelization of the divide and combine function can be described in a very clear and precise way, even for the case of non-binary division. The model supports automatic compile-time optimization for a given target topology. Examples of specific linear and piecewise linear space-time mappings and performance experiments are described elsewhere [6]. The main result of our experiments is that polynomial product can be parallelized efficiently because the communication time takes only a small share of the overall execution time (if $k>2$).

In an s-dimensional mesh, the execution time of our $\mathcal{DC}$ scheme is, in general, not logarithmic, as it would be in a high-dimensional mesh (of dimensionality n), but it can be still sublinear (if $s > \mathrm{ld}\, k$). This is an improvement over parallel loops derived with the polytope model. Note that sublinear execution times cannot be achieved by a tree implementation of this skeleton, because the input and output must then be centralized in the root processor.

While the skeleton distinguishes work and data, and the dimensions in *each slice* of the computation domain are explicitly assigned to work or data, this distinction is lost in the loop program, which is no surprise because losing structural information is typical in code generation.

Acknowledgements

Thanks to L. Bouge, B. Caillaud and J. O'Donnell for helpful discussions, which were made possible by exchanges through the programs ARC and PROCOPE by the DAAD. Thanks also to S. Gorlatch and C. Wedler for numerous discussions. Thanks to E. Zehendner for comments.

References

1. K. Achatz and W. Schulte, Architecture independent massive parallelization of divide-and-conquer algorithms, in *Mathematics of Program Construction*, Lecture Notes in Computer Science 947, ed. B. Möller (Springer-Verlag, 1995) 97–127.

2. 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).

3. M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, Research Monographs in Parallel and Distributed Computing (Pitman, 1989).

4. J. Gosh, S. Das, and A. John, Concurrent processing of linearly ordered data structures on hypercube multicomputers, *IEEE Trans. on Parallel and Distributed Systems* **5**, 9 (1994) 898–911.

5. D. Gries, *The Science of Programming*, Texts and Monographs in Computer Science (Springer-Verlag, 1981).

6. C. Herrmann and C. Lengauer, Notes on the space-time mapping of divide-and-conquer recursions, in *Proc. GI/ITG FG PARS '95*, PARS Mitteilungen Nr. 14 (Gesellschaft für Informatik e.V., 1995).

7. C.-H. Huang and C. Lengauer, The automated proof of a trace transformation for a bitonic sort, *Theoretical Computer Science* **46**, 2–3 (1986) 261–284.

8. C. Lengauer, Loop parallelization in the polytope model, in *CONCUR'93*, Lecture Notes in Computer Science 715, ed. E. Best (Springer-Verlag, 1993) 398–416.

9. J. Misra, Powerlist: A structure for parallel recursion, *ACM Trans. on Programming Languages and Systems* **16**, 6 (Nov. 1994) 1737–1767.

10. Z. G. Mou, Divacon: A parallel language for scientific computing based on divide-and-conquer, In *Proc. 3rd Symp. Frontiers of Massively Parallel Computation*, (IEEE Computer Society Press, 1990) 451–461.

11. J. Peterson and K. Hammond editors, Report on the programming language Haskell (Version 1.3), http://haskell.cs.yale.edu/haskell-report/haskell-report.html, (May, 1996).