Notes on the Space-Time Mapping of Divide-and-Conquer Recursions

Christoph Herrmann and Christian Lengauer
Fakultät für Mathematik und Informatik
Universität Passau, Germany

{herrmann,lengauer}@fmi.uni-passau.de
http://brahms.fmi.uni-passau.de/cl/

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 programming, parallelization, polytope model, skeleton, space-time mapping.

1 Introduction

The divide-and-conquer (D&C) 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 D&C recursions with the goal of sublinear execution times on a mesh. Sublinearity can only be achieved if input data are read in parallel and processed 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) [7]. 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 (parallelism) and space (processor allocation) can be made explicit in this model. A space-time mapping puts the polytope into a coordinate system in which some axes enumerate space (i.e., represent parallel loops) and the others enumerate time (i.e., represent sequential loops). The space-time mapping can be chosen to *optimize* any one of a number of objective functions like the number of execution steps, processors, channels, communications, etc. In order for the mathematical theory to apply, the loop bounds, the data dependences and the space-time mappings must be *linear*. Any loop nest which qualifies for the polytope model can be accelerated to linear execution time, i.e., requires only one time dimension. The sequential execution time of a nest of n loops (with ranges which depend on the problem size) is a polynomial of degree n.

Our goal is to approach the parallelization of D&C 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.

We must leave the polytope model because D&C recursions do not fulfill several of its requirements. Still, as in loop parallelization, the underlying data structure is the vector variable whose elements can be accessed concurrently. Specifically considered is the case of a non-binary division: our case study is on the polynomial product using ternary D&C.

2 The Model

Let the term *fixed-degree divide-and-conquer* [2] 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 D&C type themselves. Because we want to map statically, i.e., independently of the values of the

input data, we assume that the recursion depth of all basic cases is the same. So, for a recursion depth of n, the number of basic cases in our fixed-degree D&C recursions balanced in this way is k^n.

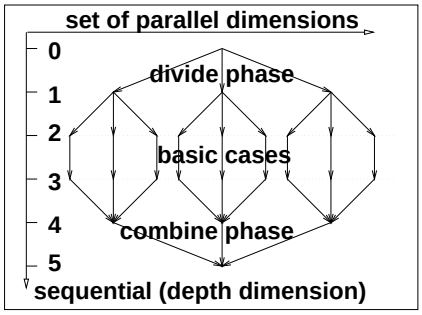

Figure 1: D&C call graph ($n=2, k=3$)

The computation of a balanced fixed-degree D&C 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. 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 D&C 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 [5]. 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 smaller computations which can also be parallelized.

We focus on a restricted format of D&C 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* [8]. Our computational model for D&C recursions, which we call the *computation domain*, is again embedded into a multi-dimensional integer lattice, but it is not a polytope. It 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 [3] and can be associated with time or with space, depending on the number of available processors. The coordinates of the depth dimension

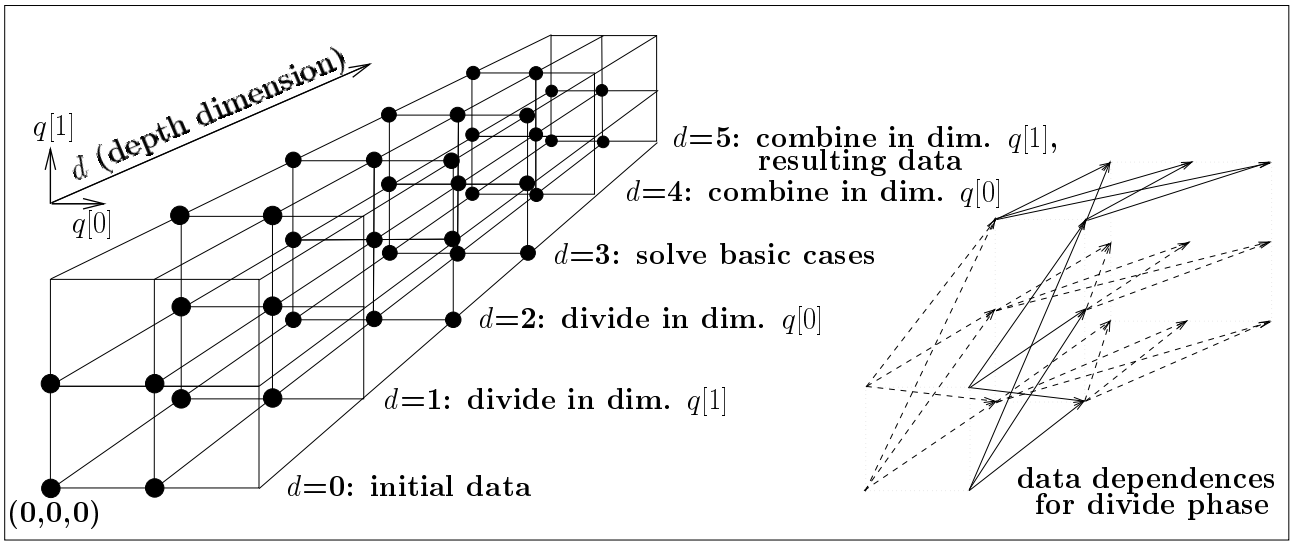

Figure 2: Computation domain ($n=2$, $k=3$)

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. That is, 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 call graph) corresponds to 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. That is, the input and output is located at a hypercube (binary n-cube) embedded in the k-ary n-cube.

Because it is unrealistic to assume a mesh of arbitrarily high dimensionality in hardware, we are interested in methods of reducing the dimensionality of the subspace of the computation domain which is mapped to space while maintaining its cardinality. I.e., we want to distribute the computation points that are mapped to space over an integer lattice of fewer dimensions.

3 The Program Skeleton

The following functional program schema, in the syntax of Haskell [6], defines our skeleton for balanced fixed-degree divide-and-conquer; to formulate a specific D&C problem, fill in the constituting functions *basic*, *divide* and *combine* and the degree k appropriately:

$$divcon :: \mathrm{Int}\to(a\to b)\to(a\to a\to[a])\to([b]\to(b,b))\to[a]\to[b]$$

```
divcon k basic divide combine = solve
    where solve indata = if length indata == 1
                         then map basic indata
                         else let x = zipWith divide (left indata) (right indata)
                                  y = transpose (kmap k solve (transpose x))
                              in  map (fst . combine) y) ++ (map (snd . combine) y )
```

Of the following functions used by this schema all but *kmap*, *left* and *right* are standard in Haskell:

length		$[a_0,\ldots,a_{u-1}]$	$= u$
map	f	$[a_0,\ldots,a_{u-1}]$	$= [f\ a_0,\ldots,f\ a_{u-1}]$
kmap	$k\ f$	$[a_0,\ldots,a_{k-1}]$	$= [f\ a_0,\ldots,f\ a_{k-1}]$
left		$[a_0,\ldots,a_{2*u-1}]$	$= [a_0,\ldots,a_{u-1}]$
right		$[a_0,\ldots,a_{2*u-1}]$	$= [a_u,\ldots,a_{2*u-1}]$
zipWith	f	$[a_0,\ldots,a_{u-1}]\ [b_0,\ldots,b_{u-1}]$	$= [f\ a_0\ b_0,\ldots,f\ a_{u-1}\ b_{u-1}]$
transpose		$[[a_{(0,0)},\ldots,a_{(0,u-1)}],\ldots,[a_{(v-1,0)},\ldots,a_{(v-1,u-1)}]]$	$= [[a_{(0,0)},\ldots,a_{(v-1,0)}],\ldots,[a_{(0,u-1)},\ldots,a_{(v-1,u-1)}]]$
fst	(a,b)		$= a$
snd	(a,b)		$= b$
"*append*"	$[a_0,\ldots,a_{u-1}] ++ [a_u,\ldots,a_{2*u-1}]$		$= [a_0,\ldots,a_{2*u-1}]$

The idea of componentwise vector operations is taken from [9], 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 input vector to the subproblem. 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 *kmap* 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. The *transpose* operations provide the interface between the data division and the problem division.

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

4 Example: Polynomial Product

In 1962, Karatcuba published a D&C algorithm for the multiplication of large integers of size N with cost $\mathcal{O}(N^{\mathrm{ld}\,3})$ based on ternary D&C [1, Sect. 2.6]. As an example of ternary D&C, 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 datatype integer is unbounded.

We consider $+$, $-$ and $*$ operations 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 degree $N < 2^{n-1}$, 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 polynomial 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 datatype and data dependence restrictions imposed by our skeleton, the input vector of the skeleton is the zip of two coefficient vectors ($zip\ [a_0,...,a_{u-1}]\ [b_0,...,b_{u-1}] = [(a_0, b_0),...,(a_{u-1}, b_{u-1})]$) 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 *karat*, which multiplies two polynomials represented by equal-size powerlists:

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

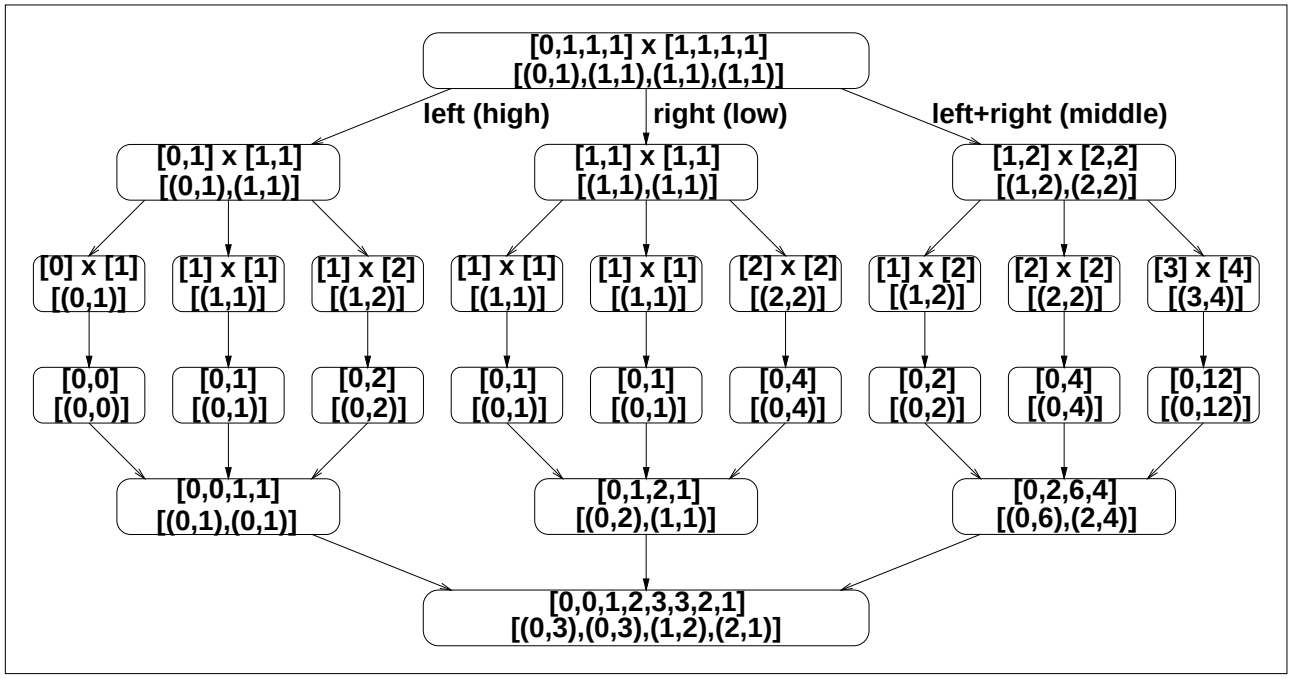

Figure 3: Call graph for a call of *karat*

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. As example Fig. 3 depicts the call graph for multiplication of the polynomials $(X^2 + X + 1)$ and $(X^3 + X^2 + X + 1)$ whose result is the polynomial $(X^5 + 2 * X^4 + 3 * X^3 + 3 * X^2 + 2 * X + 1)$. In each node, we give on top the polynomials as lists of their coefficients and below the representation as required by the skeleton.

5 A Nested Loop Program

Our implementation of the skeleton is not a functional but an imperative program with sequential and parallel loops which enumerate the computation domain defined before. 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 section.

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 D&C 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 $(q[0], ..., q[n-1])$ 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 (Sect. 2). Thus, $q = \sum_{i=0}^{n-1} q[i]*k^i$ 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. We use the notation of [4] for substituting into q: $(q; i : a)$ is equal to q except that it has an a in position i. Our loop program has three double loop nests; the first corresponds to the divide phase, the second to the basic cases and the third to the combine phase:

for $d = 1, ..., n$
  for $q = 0, ..., k^n - 1$
    $A[d, q] = \mathit{divide}[q[n-d]]\ A[d-1, (q; n-d : \mathit{leftpos})]\ A[d-1, (q; n-d : \mathit{rightpos})]$
for $d = n+1$
  for $q = 0, ..., k^n - 1$
    $B[d, q] = \mathit{basic}\ A[d-1, q]$
for $d = n+2, ..., 2*n+1$
  for $q = 0, ..., k^n - 1$
    $C[d, q] = (B[d-1, (q; d-(n+2) : 0)], \ldots, B[d-1, (q; d-(n+2) : k-1)])$
    $B[d, (q; d-(n+2) : \mathit{leftpos})] = (\mathit{fst}\ .\ \mathit{combine})\ C[d, q]$
    $B[d, (q; d-(n+2) : \mathit{rightpos})] = (\mathit{snd}\ .\ \mathit{combine})\ C[d, q]$

A and B are arrays, which are indexed by points of the computation domain. The elements of A are of the input data type and the elements of B are of the output data type. The elements of C are k-tuples of elements of B for constructing the input to function *combine*. Note that we select a component of the vector-valued function *divide* by normal array indexing. Our index is $q[n-d]$ 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.

The input and output is distributed across the points which have only coordinate values *leftpos* and *rightpos* in q, where $\mathit{rightpos} = \lfloor k/2 \rfloor$ and $\mathit{leftpos} = \mathit{rightpos} - 1$. The values supplied at the other points are immaterial. They are also used as input—but for unnecessary computations which are performed in parallel to the necessary ones, require no additional time and cause no interference. Masking the unnecessary computations leads to a more complex program but may permit more efficient mappings.

6 The Mappings

6.1 Refinement of the Depth Dimension

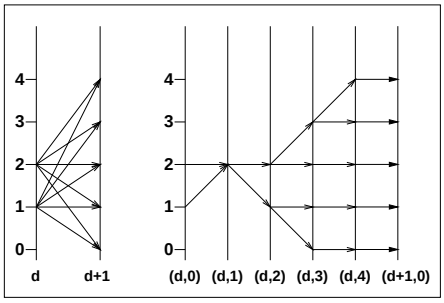

Figure 4: Depth refinement ($k=5$)

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 new 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. 4 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 $\mathit{rightpos} = \lfloor k/2 \rfloor$ and $\mathit{leftpos} = \mathit{rightpos} - 1$ in a certain dimension is collected in pairs at the points whose values are *rightpos* 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 $\lceil k/2 \rceil + 2$.

6.2 Constraints on the Space-Time Mapping

A space-time mapping must satisfy the following conditions:

Injectivity: 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 at all be put into matrix form, it will not be square.

Consistency: This requirement says that the time mapping must respect the data dependences.

Feasibility: If x depends on y then $time(x) - time(y) \geq distance(space(x), space(y))$. This condition guarantees that there will always be enough time to get a datum from the source to the destination via neighboring connections. We intend that the satisfaction of this constraint is left to routing after optimization. Then, long distances should be assigned high costs.

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 and the extent of each dimension depends on the problem size [7]. Here, the dimensionality of the computation domain depends on the problem size and the extent of each (but the depth) dimension is fixed.

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 are 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 $(x \text{ div } c = y \text{ div } c) \wedge (x \bmod c = y \bmod c) \Rightarrow x = y$. 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 $M = k$. 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.

6.3.1 Example mappings with $m = 1$

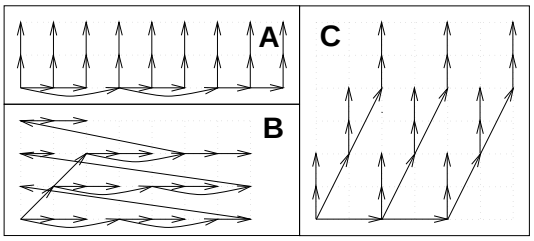

Figure 5: Data dependences in mappings

Let $k = 3$ (ternary D&C), $r = 3$ (at least 27 processors) and $s = 2$. Tab. 1 presents a small selection of mappings; mappings A and C are linear. The path column contains the length of the longest path in the Manhattan metric (the sum of absolute coordinate values). Total communication times obtained by experiments on our 8×8 mesh with a problem size of 32768 are shown in the last column. An experiment with mapping A was not possible because a 9×3 problem does not fit the format of our mesh. Fig. 5 shows the dependences in the mappings which contribute to the propagation of data from processor $(0, 0)^{\mathrm{T}}$.

	space mapping	topology	path	exp.
A	$(3 * q[n-1] + q[n-2],\ q[n-3])^{\mathrm{T}}$	9×3	10	—
B	$(\bmod 8, \text{ div } 8)^{\mathrm{T}} \circ (9 * q[n-1] + 3 * q[n-2] + q[n-3])$	8×4	17	1.9s
C	$(2 * q[n-1] + q[n-2],\ 2 * q[n-2] + q[n-3])^{\mathrm{T}}$	7×7	12	1.5s

Table 1: Characteristics of example mappings

The disadvantage of mapping C is that processors are wasted because there are holes in the image. Mapping C yields the minimal total communication time, for all *linear* mappings to an 8×8 mesh and with respect to the objective function given by the sum of all absolute coefficients which is 6, so the three division steps (depth coordinates) take together $6 * (\lceil k/2 \rceil + 2) = 24$ substeps, regarding the number of refined substeps of the depth dimension.

Consider $a_{(i,j)}$ ($i \in \{0,1\}$, $j \in \{0,1,2\}$) to be the coefficients of a mapping matrix. The rows, indexed by i, correspond to the dimensions of the mesh and the columns, indexed by j, to the dimensions $q[n-1-j]$ of the computation domain which are mapped to space. Because of the bounds of the mesh, all $a_{(i,j)}$ must be non-negative and must satisfy the additional constraint $(\forall\ i : i \in \{0,1\} : \sum_{j=0}^{2} 2 * a_{(i,j)} < 8)$, and mapping $\mathrm{C} = \begin{pmatrix} 2 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix}$ does so.

With all three mappings, every processor has to compute 3^{n-3} basic cases, i.e., we use only 27 processors although our mesh provides 64. The following subsection demonstrates that mapping a dimension to both space and time may increase processor utilization.

6.3.2 Example mapping with $m > 1$

Take the same values for k, r and s as in the previous subsection. Remember that we require 27 processors. Mapping one more dimension linearly to space would require 81 processors which our mesh does not provide. But, if we map one dimension partly to time and partly to space, we can use $2 * 27 = 54$ processors of our 64 by choosing $m = 2$, so $M = 2$.

Our non-linear mapping is composed of three stages. In the first stage, dimension $q[R]$ of the computation domain is partitioned; the part mapped to time is $time = q[R] \bmod 2$, the part mapped to space is $space = q[R] \operatorname{div} 2$. Dimensions $q[n-1]$ to $q[n-r]$ are mapped exclusively to space. In the second stage, we map all dimensions $q[n-1], ..., q[n-r]$ and the space image of $q[R]$ to a single dimension to prevent holes as caused by mapping C. In the third stage, we expand the single dimension to as many dimensions as our mesh provides, obtaining something similar to mapping B (Fig. 5).

Take our example. Because our mesh is two-dimensional, we split the single dimension into two by using (mod 8) and (div 8): $space = (\bmod 8, \operatorname{div} 8)^{\mathrm{T}} \circ (9 * q[n-1] + 3 * q[n-2] + q[n-3] + 32 * (q[n-4] \operatorname{div} 2))$ and $time = ((d,i), q[n-4] \bmod 2, q[n-5], ..., q[0])$, where (d,i) is the refined depth coordinate.

The maximum number of basic cases each processor has to compute is $2 * 3^{n-4}$, a third less than before. This leads to an improvement of about a factor of 1.5 in speed, using twice as many processors.

Choosing the factor 32 rather than 27 in the mapping limits irregularity because it is a better match with the application of (mod 8, div 8). The local irregularity introduced by the use of 9 in the mapping is not so serious.

One of our goals is to classify different kinds of mappings, like $(a_0 * q[n-1] + a_1 * q[n-2] + a_2 * q[n-3] + a_3 * (q[n-4] \operatorname{div} 2))$, and to determine the optimal coefficients automatically. The optimality of $(a_0, a_1, a_2, a_3) = (9, 3, 1, 32)$ with respect to realistic objective functions based on the Manhattan metric has been verified by enumeration.

Our non-linear mappings consist of a linear part, which is determined by integer optimization, and additional parts like integer division by a constant (div c) and remainder (mod c) which have to be determined manually and have to be taken into account in the specification of the constraints and the objective function. We apply these additional mappings before the linear mapping to partition a dimension of the computation domain into time and space and after the linear mapping for adaptations to the target topology, e.g., its extent.

7 Evaluation

7.1 Complexity Considerations

Let k be the division degree, N the size of the problem, p the number of processors and, for simplicity, assume $m = 1$. Then $n = \lceil \mathrm{ld}\, N \rceil$ 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 $T1 + T2 + T3$.

7.2 Experiments

Tab. 2 lists, in seconds, the total runtime results on our mesh using mapping B for 1, 3, 9 and 27 processors. Note the slow increase when using the next power of 3 as the number of processors for the double problem size, which is due to communication. For $m=1$ and $n-r>15$, an estimate of the total execution time is $3^{n-r} * 60\mu$s plus a few percent of communication effort increasing with r and memory management effort increasing with $n-r$. The efficiency results we obtained are depicted in Fig. 6. Here, efficiency is defined to be the speed-up divided by the number of *used* processors.

problem size	number of processors			
	1	3	9	27
512	1.13	0.40	0.16	0.08
1024	3.45	1.20	0.46	0.20
2048	10.4	3.58	1.30	0.54
4096	31.5	10.7	3.80	1.48
8192	94.9	32.1	11.2	4.14
16384	286	96.1	33.0	11.9
32768	859	288	97.9	34.4

Table 2: Total execution times in seconds

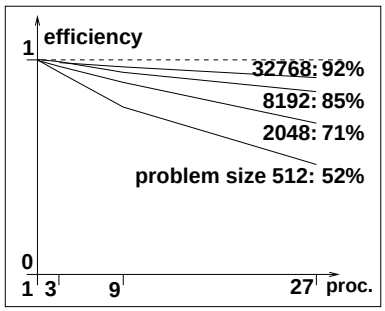

Figure 6: Efficiency

8 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 with respect to a given target topology. Furthermore, our experimental results have demonstrated that polynomial product can be parallelized efficiently and that 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 D&C 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 $s>1$). 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.

Acknowledgements. Thanks to J. O'Donnell, S. Gorlatch and C. Wedler for helpful discussions.

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. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.
3. 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):898–911, 1994.
4. D. Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1981.
5. C.-H. Huang and C. Lengauer. The automated proof of a trace transformation for a bitonic sort. *Theoretical Computer Science*, 46(2–3):261–284, 1986.
6. P. Hudak, S. Peyton Jones, and P. Wadler, editors. Report on the programming language Haskell (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.
7. 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.
8. J. Misra. Powerlist: A structure for parallel recursion. *ACM Trans. on Programming Languages and Systems*, 16(6):1737–1767, November 1994.
9. Z. G. Mou. Divacon: A parallel language for scientific computing based on divide-and-conquer. In *Proc. 3rd Symp. Frontiers of Massively Parallel Computation*, pages 451–461. IEEE Computer Society Press, October 1990.