Parallelization of Divide-and-Conquer by Translation to Nested Loops

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

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

Abstract

We propose a sequence of equational transformations and specializations which turns a divide-and-conquer skeleton in Haskell into a parallel loop nest in C. Our initial skeleton is often viewed as general divide-and-conquer. The specializations impose a balanced call tree, a fixed degree of the problem division, and elementwise operations. Our goal is to select parallel implementations of divide-and-conquer via a space-time mapping, which can be determined at compile time. The correctness of our transformations is proved by equational reasoning in Haskell; recursion and iteration are handled by induction. Finally, we demonstrate the practicality of the skeleton by expressing Strassen's matrix multiplication in it.

Keywords: divide-and-conquer, equational reasoning, Haskell, parallelization, skeleton, space-time mapping.

1 Introduction

Divide-and-conquer ($\mathcal{DC}$) is an important programming paradigm. It prescribes the solution of a problem by dividing it into a number of subproblems, which are solved recursively until a basic case is reached, and then combining the solutions of the subproblems to get the solution of the original problem. Because of the wide applicability of the $\mathcal{DC}$ paradigm, it has often been formulated as a so-called *algorithmic skeleton* [6, 7], which can be used as a basic building block for programming. One purpose of the skeleton concept is to provide the user with efficient implementations of popular paradigms. In this approach, the algorithmic skeleton for a paradigm corresponds to an executable, but unintuitive *architectural skeleton* [17]. To make the correspondence between the algorithmic and the architectural skeleton formally precise, we work in the domain of functional programming, in which skeletons are predefined higher-order polymorphic functions.

The fact that the subproblems are independent makes $\mathcal{DC}$ particularly attractive for a parallelization. That is, one major purpose of an architectural skeleton for $\mathcal{DC}$ is to provide an efficient implementation for a given parallel computer. However, in order for a corresponding efficient architectural skeleton to exist, the algorithmic skeleton has to satisfy certain conditions.

The aim of this treatise is to specialize an algorithmic skeleton for $\mathcal{DC}$ to a form for which there is an efficient parallel implementation. We impose specializations step by step, e.g., a fixed division degree of data or of work, etc. We present only a single path in the tree of possible specializations of $\mathcal{DC}$; other specializations can be envisioned as well. Some specialized skeletons (we call them *sources*) can be transformed to *functional target* skeletons, which have an obvious correspondence with nested parallel loop programs. Our so-called "call-balanced fixed-degree $\mathcal{DC}$" skeleton and all of its specializations can be compiled into an intermediate code consisting of sequential and parallel loops, which can be translated easily to different parallel machines. The loop nest can be viewed as our architectural skeleton. In the absence of resource constraints, it will provide the fastest possible execution given the data dependences imposed by divide-and-conquer.

Many specializations are not really restrictions. We will show in some cases how to express the more general skeleton in terms of the more specialized one.

The abstract computational model in which we justify and describe our specializations is the *call tree*. The root of the call tree represents the entire problem instance, the ith son of a node N represents the ith subproblem instance of the problem instance which N represents.

One property we require in our specializations is the balance of the call tree. We distinguish two kinds of balance. In *call balance*, all leaves of the call tree have the same distance from the root. In *data balance*, all sons of a node of the tree carry approximately the same amount of data. Our mapping techniques are static, i.e., we cannot enforce either form of balance on the call tree at run time.

In our understanding, specialization of a skeleton does not necessarily imply the preservation of its type. For a transformation to an efficient implementation, we take the liberty, e.g., to omit tuple components that become irrelevant, or list wrappers if the lists are always singletons, etc. Our criterion for a skeleton `B` to be a specialization of a skeleton `A` is that `B` can be defined by an expression `E` which contains a single application of `A`, and the parameters of this application as well as the other functions in `E` cannot express `B`.

Early work on transforming recursion with dependent calls into *sequential* loops was based on a depth-first traversal [21]. This method is not very useful in a parallelization, where a breadth-first traversal is called for. The parallelization technique of Harrison and Khoshnevisan [12] can be extended [8] to handle $\mathcal{DC}$ if certain conditions are fulfilled. We start with similar considerations and develop a method for translating $\mathcal{DC}$ into a nested linear recursive skeleton, which can easily be interpreted as a loop nest. Our approach uses a tree structure for an intermediate representation, because it is a data structure that allows the correct typing of recursively defined objects and is useful for exploiting structural properties in the parallelization. We prove the semantic equivalence of the specialized algorithmic skeleton and the loop nest.

The Haskell definitions and equalities have been type checked automatically. To understand the development process the reader is not required to understand all of the code we present. Some equalities, which are quite easily understood intuitively, require some amount of formalism to make the equational reasoning in Haskell work.

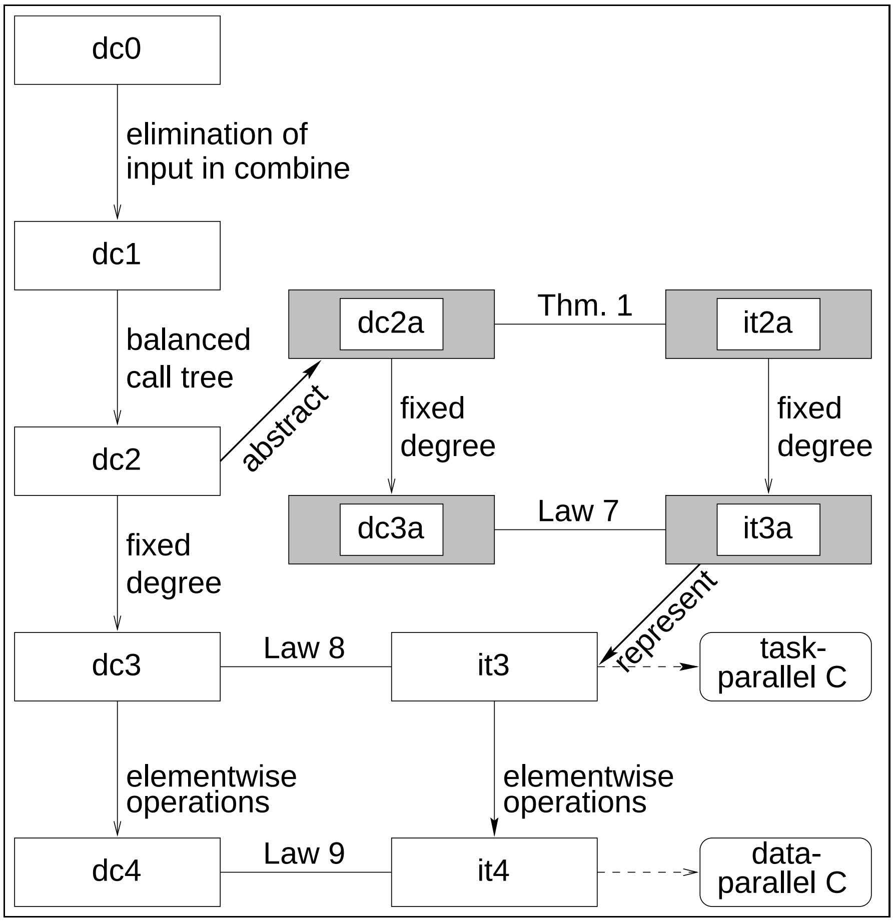

Figure 1: specializations of $\mathcal{DC}$

Fig. 1 depicts the skeletons we present in this paper, and their dependences. Downward arrows denote specializations, undirected edges equivalences, dashed arrows a switch in language (from Haskell to C).

The paper is structured as follows. In Sect. 2, the skeletons `dc0` to `dc3` are defined. In Sect. 3, we transform the call-balanced fixed-degree $\mathcal{DC}$ skeleton `dc3`, which is recursive, to skeleton `it3`, which is iterative (in the sense that it uses list comprehensions). Then we provide rules for translating `it3`, which is as close to loops as we can get in Haskell, to a parallel loop nest in C. In Sect. 4, we specialize skeleton `dc3` further, with elementwise operations on balanced data partitions, and present a corresponding C loop program. In Sect. 5, we specify Strassen's matrix multiplication with the instantiated skeleton `dc4` to demonstrate an application. The last section summarizes our results and discusses related work.

In our specializations, we enforce first call balance and then data balance. The only exception is Subsect. 2.2.2, where we present a skeleton which provides a facility for preserving data balance, but which can be used also in the case of an unbalanced call tree.

2 Specializing $\mathcal{DC}$

In this section, we propose a sequence of specializations of a skeleton for general divide-and-conquer. We denote our skeletons in the functional language Haskell [16]. First, we present a general form of $\mathcal{DC}$ which is then specialized to enforce a balanced call tree, and subsequently further to enforce a fixed degree of the problem division.

2.1 General $\mathcal{DC}$ (dc0)

Under the premise that the subproblems can be solved independently, the general $\mathcal{DC}$ skeleton can be specified as follows:

```
dc0 :: (a->Bool)->(a->b)->(a->[a])->((a,[b])->b)->a->b
dc0 p b d c x = r x
    where r x = if p x then b x
                       else c (x, map r (d x))
```

As aggregating data structure, we have chosen the list arbitrarily; other choices can be envisioned. The skeleton is parametrized by four so-called *customizing functions*: the predicate `p`, which recognizes the basic case, the basic function `b`, which is applied in this case, and the functions `d` for dividing a problem into a list of independent subproblems, and `c` for combining the input data and the list of subproblem solutions to the solution of the original problem. The customizing functions are parameters which must be fixed at *compile time*, i.e., *before* we parallelize. Only the last parameter `x`, the data, is a run-time parameter.

To proceed to loop form, it is necessary to eliminate irregularities in the structure of the skeleton. The most apparent irregularity is that the combine function is defined on data not only of different levels of recursion, but even of different phases, namely the divide and combine phase. This complicates the proof of correctness and later the generation of the data-parallel program. In the following subsection, we eliminate the use of the input data in the combine function, without loss of generality.

2.2 $\mathcal{DC}$ without combining the input data (dc1)

In the following skeleton the input data is not used explicitly any more in the combine function:

```
dc1 :: (a->Bool)->(a->b)->(a->[a])->([b]->b)->a->b
dc1 p b d c x = r x
    where r x = if p x then b x
                       else (c . map r . d) x
```

The expressive power of `dc0` and `dc1` is equivalent. That `dc1` can be expressed in terms of `dc0` is trivial, and that `dc0` can be expressed in terms of `dc1` is shown below. There are at least two reasonable ways. The first one is appropriate if the size of the data does not matter, especially in the case where the input and output data is not distributed. The second one is appropriate if the size of the data is critical and the user is able to describe how the data can be distributed.

2.2.1 Expressing dc0 in terms of dc1

Every `dc0` skeleton can be expressed as a `dc1` skeleton by modification of the customizing functions. The application of $\mathcal{DC}$ unfolds into a tree of calls. The sons of each node in the tree represent the subproblems of the problem the node represents. By applying `dc0`,

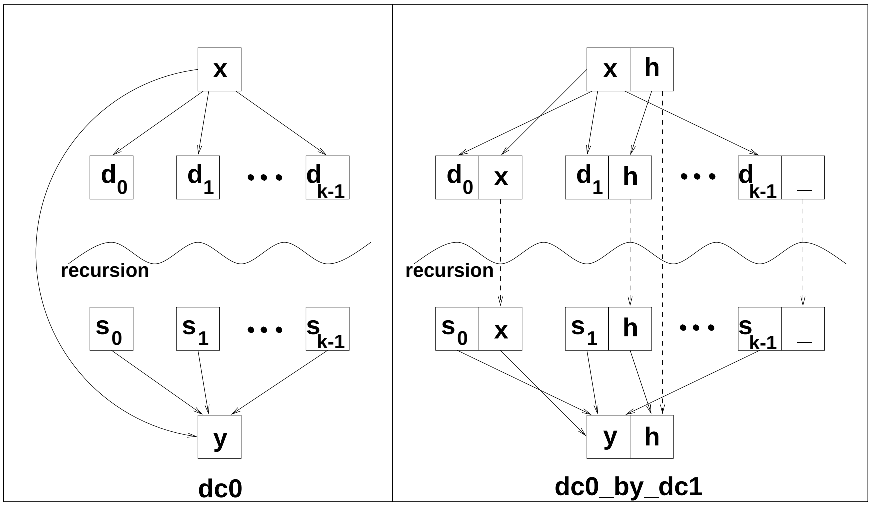

Figure 2: Expressing `dc0` by `dc1`

each call stores the input data into its node until its sons contain the subproblem solutions, which are then combined with the stored input data.

In `dc1` the nodes have no capability of storing data. To express `dc0` in terms of `dc1`, we pass the input data `x` of some subproblem downwards and upwards the subtree rooted at the leftmost son, see Fig. 2.

The data `h` a node gets from his father in order just to give it back to him later is given to the subtree rooted at the second son from the left to avoid an accumulation of data growing with the recursion depth on a single processor.

Technically speaking, we add an auxiliary element `h` to the data, i.e., a datum of `dc1` is now a pair of data of `dc0`. The names `x` and `h` in the figure correspond to the ones in the argument pattern of the functions `p`, `b` and `d` in the skeleton below. The left side of the pair carries the value we had before, i.e., an input value when sweeping down the tree and an output value when sweeping up. The right side carries input values also when sweeping up the tree.

If a node is a first son (from the left) then its right side carries the left side of its father's data. If it is a second son then its right side carries the right side of its father's data. If a node is neither the first nor the second son (this happens in $\mathcal{DC}$ with a division degree greater than 2) then its right side is undefined.

```
dc0_by_dc1 :: (a->Bool)->(a->b)->(a->[a])->((a,[b])->b)->a->b
dc0_by_dc1 pp bb dd cc x  = fst (dc1 p b d c (x,dummy))
   where dummy            = x
         p (x,h)          = pp x
         b (x,h)          = (bb x,h)
         d (x,h)          = let (d0:d1:ds) = dd x
                                dummy      = x
                            in [(d0,x),(d1,h)] ++ map (\di->(di,dummy)) ds
         c (s0x:s1x:sxs)  = (cc (snd s0x,map fst (s0x:s1x:sxs)),snd s1x)
```

2.2.2 Expressing `dc0` in terms of `dc1`, using additional data-balancing functions

The principle of balanced data division is that the input and output data is distributed equally among the processors, or a subset of them, and that in each division or combination step the balance is maintained. The reason for this constraint is that the memory of a single processor is too small and/or serial operations with the data is very time-consuming.

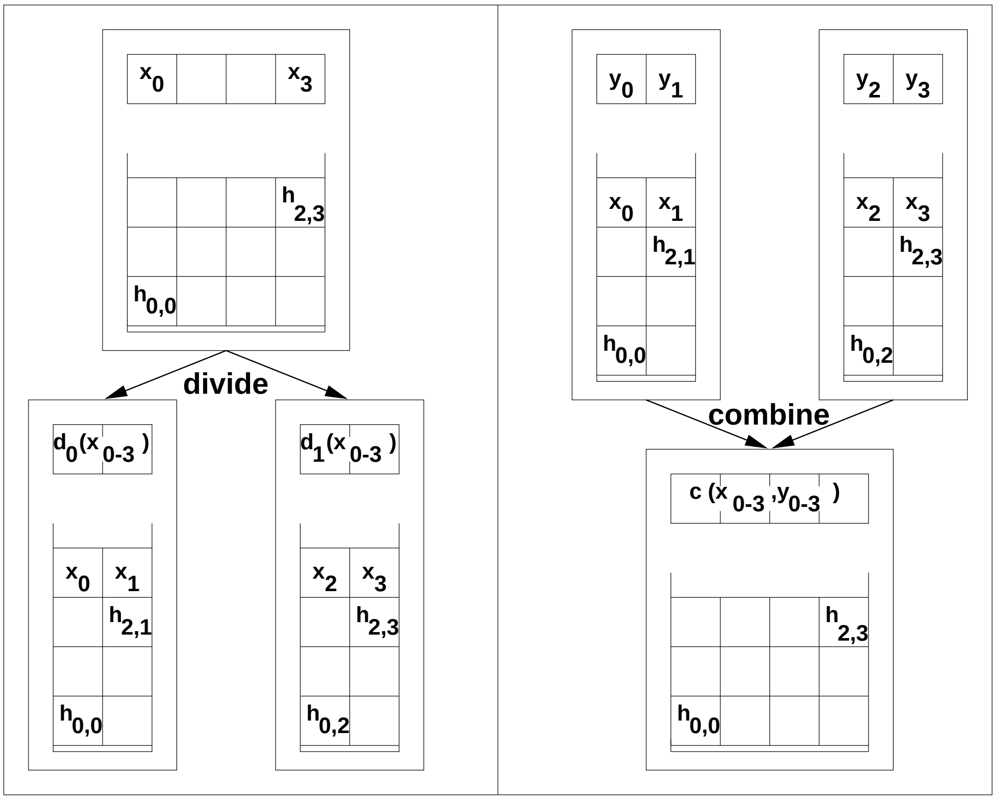

Figure 3: Expressing `dc0` by `dc1`, balanced

Skeleton `dc0_by_dc1` does not maintain the balance of data because a copy of the total input data has to be placed onto a single processor. Therefore we propose a different solution, which avoids this problem but whose corresponding skeleton expects two additional customizing functions: one defining a balanced division of the input data (this can but need not be the divide function) and another defining the gathering of these parts. These functions must be supplied by the user because they act as interfaces between the (encapsulated) abstract data types of input and output and the skeleton implementation, with knowledge of nothing but the size of memory to be allocated for these data aggregates.

In order to avoid unnecessary copying, the balancing function should let every processor resp. memory partition keep the amount of input data that is located on it for later use in the combine function. However, the data may also be needed on another processor. Then the involved communication is performed during the combine phase. Because at every division step new input data (for the subproblems) is generated, the input data is stored in a stack. In the world of sequential processing, a lot of effort has been put into the theory of recursion elimination [21, 3]. One of the reasons is to avoid the overhead of copying data onto a stack.

Here we do not have this problem: the data to be stored in each processor is very small because the "stack" is *distributed*! Furthermore we are not restricted to things like associative combine functions and do not have to consider the invertibility of functions as, e.g., [21]. However we are just handling divide-and-conquer: this means that the recursive calls are independent of each other.

Imagine that the rows of the stack contain the data of one recursive call and all elements of a row are pushed or popped in parallel, i.e., the stack is partitioned into columns for different processors. The distribution of the columns is organized as follows. Each node of the call tree contains, additionally to the input data, the columns of the stack which hold the input data of the previous division steps; see the left side of Fig. 3 for the case of a binary problem and data division and balancing functions.

In order to save communication overhead and memory space, we prefer balancing functions that just change the type (format) of the data but do not affect their location. We call such balancing functions *adequate*. In general, whether a balancing function is adequate depends on the mapping of data to the memory.

In Subsect. 4.3, where we discuss the skeleton with balanced data distribution, we construct adequate balancing functions by dividing the data into the same number of parts as the problem. The balancing function is equal to a divide function which computes the identity, where the `i`th subproblem consists exactly of the `i`th data partition.

One frequent case is that a problem is divided into two subproblems, which are assigned to subtopologies of equal size. If the input data is represented by a distributed list, one can store the left part of the list on the subtopology assigned to the first and the right part on the one assigned to the second subproblem. This is the case shown in Fig. 3. The input data is called `x`, with a subscript denoting the processor the part of `x` is placed on. The stack is called `h` (for history). The subscripts of `h` state the level of recursion the input data is taken from and the subscript of the corresponding part of `x`.

In a divide step, the new input data for branch `i` is computed by selecting the `i`th part of the result of applying the divide function `d` to the old input data, and the old input data is pushed onto the stack of input data histories.

The figure indicates that the left part of the stack is located at the subtopology that is assigned to the first subproblem, likewise for the right part and the second subproblem. The right side of Fig. 3 shows what happens in the combine phase. The new output data is generated by applying the combine function `c` to the input data `x`, which is popped from the stack, and the output data `y` of the subproblems.

The distribution of the stack slices will not change if one uses adequate balancing functions, but the number of columns of the stack that contain useful data can grow towards the top of stack; that is the case if the degree of problem division is greater than the degree of data division.

In general, the method is defined by the following skeleton `bal_dc0_by_dc1`, which has two arguments in addition to those of `dc0_by_dc1`: the balancing functions `ddd` ("`dd` for data") and `ccd` ("`cc` for data"). `ddd` must divide the data into the same number of parts `dd` divides the problem into, and `ccd` must be the left inverse of `ddd`. As in the previous skeleton `dc0_by_dc1`, the data is a tuple, whose first component contains the input resp. output data but whose second component is a vertical slice of the stack of input data histories.

```
bal_dc0_by_dc1 ::
   (a->Bool)->(a->b)->(a ->[a])->((a,[b])->b)->(a->[a])->([a]->a)->a->b
bal_dc0_by_dc1 pp bb dd cc ddd ccd x = fst (dc1 p b d c (x,[]))
    where p (x,h) = pp x
          b (x,h) = (bb x,h)
          d (x,h) = let divided = dd x
                        history = map ddd (x:h)
                    in zip divided (transpose history)
          c s     = let solutions = map fst s
                        histories = transpose (map snd s)
                        history   = map ccd histories
                    in (cc (head history,solutions),tail history)
          transpose  = foldr (\xs xss -> zipWith (:) xs (xss ++ repeat [])) []
```

In the case of binary $\mathcal{DC}$ on lists, i.e., where type `a` is `[ea]` and type `b` is `[eb]`, say, adequate balancing functions can be:

```
ddd x       = [left x,right x]
ccd [y0,y1] = y0 ++ y1
```

where `left` and `right` select the left and right part of a list, respectively.

The effect of an adequate balancing function on the target program is that some data movements have a processor distance of zero and need not be implemented. Note that the following skeletons could be adapted to use the input data in the combine function as well. The adaptations for the skeletons upto `dc3` can be made by substitution of `dc1` and modification of the interface. Consider, e.g., `dc3_bd`, which is the skeleton `dc3` of Sect. 2.4 with a data distribution determined by `ddd` and `ccd`, and omit the definition of `p`, which is not used any longer.

```
dc3_bd :: Nat->(a->b)->(a->[a])->((a,[b])->b)->(a->[a])->([a]->a)->Nat->a->b
dc3_bd k bb dd cc ddd ccd n x = fst (dc3 k b d c n (x,[]))
   where ...
```

The other skeletons require a change of the code in the where clause.

2.3 Call-balanced $\mathcal{DC}$ (dc2)

Two more specializations have to be applied to `dc1` in order to transform it to a parallel loop program: (1) we must fix the degree of the problem division, and (2) we must balance the call tree, i.e., all paths from the root to any leaf must have the same length. The notion of balance, as we use it here, does not imply a balanced processor load. One question is in which order these specializations should be applied. Later we will see that the loop program is doubly nested: the outer loop enumerates the levels of the tree and the inner loop the nodes at a single level. There are other ways of scanning the nodes of a tree, but this is the only way of processing each point of the scan as soon as possible while respecting the data dependences. If the division degree is fixed but the tree is not balanced, we cannot construct the outer loop (and therefore also not the inner one, because it depends on the outer loop). On the other hand, balance without a fixed degree means that we are able to construct the outer loop, but not the inner one. Therefore, we impose balance first and fix the degree later.

Because balance implies that each path of the call tree contains the same number of recursive calls, we can replace predicate `p` by a counter `n` for the remaining recursion levels; `n` appears late in the list of curried parameters because it is not constant during a computation like `p` but changes frequently. The following skeleton `dc2` describes the class of $\mathcal{DC}$ with a balanced call tree:

```
type Nat = Int

dc2 :: (a->b)->(a->[a])->([b]->b)->Nat->a->b
dc2 b d c n x = r n x
   where r n x = if n==0 then b x
                         else (c . map (r (n-1)) . d) x
```

This is an optimized version. If one combines `n` and `x` to a pair, we obtain the following skeleton `dc2_by_dc1`, in which `dc2` is expressed in terms of `dc1`.

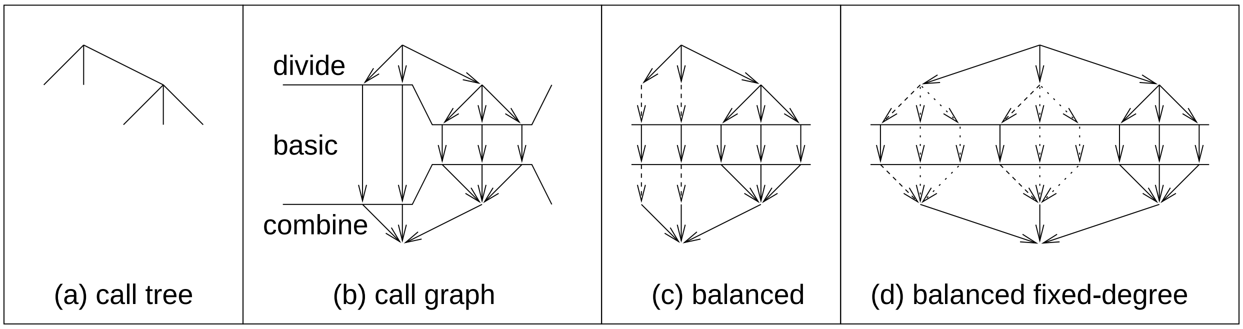

Figure 4: Transformation steps of $\mathcal{DC}$ towards regularity

```
dc2_by_dc1 :: (a->b)->(a->[a])->([b]->b)->Nat->a->b
dc2_by_dc1 b d c n x = dc1 p bb dd c (n,x)
   where p  (m,y) = m==0
         bb (m,y) = b y
         dd (m,y) = map (\z->(m-1,z)) (d y)
```

If one allows `n` to be instantiated dependent on `x`, terminating instances of skeleton `dc1` can be expressed in terms of `dc2` with a loss of efficiency in the parallel implementation; this requires a modification of the customizing functions. We have to consider two cases. In case 1, the base case is not necessarily reached within the given number of recursion levels. In case 2, the base case is reached after fewer levels of recursion than given. We suggest a solution for both cases:

1. We need a function `depth` which tells us the maximal recursion depth dependent on `p`, `d` and `x`, to be given as parameter `n` to `dc2`. In [12] this is computed by a `while` loop, which we have to avoid since we want to apply a static space-time mapping. Telling the depth is the responsibility of the environment; often the depth can be computed easily from the size of the input data. From a practical point of view, there is a simpler and more efficient possibility: the depth remains a structural parameter and is chosen later with regard to the size of the processor topology, and the function `b` is replaced by the sequential $\mathcal{DC}$ algorithm. Additionally, this avoids that a lot of threads are created on a single processor.

2. We have to extend the domain of the customizing functions `d` and `c` in order to establish the so called overrun-tolerance property [8]. In our setting, i.e., although the predicate `p` determines that the function `b` has to be applied, further recursion is not harmful. This can be achieved by extending the definitions of `d` and `c`, in the base case, to `d x = [x]` and `c [y] = y`. If, in the recursive case, the problem is not divisible any further, it is simply passed on to the next level of recursion in a singleton list. On the way back, if there is only a single subproblem solution, this becomes the solution of the original problem.

2.4 Call-balanced fixed-degree $\mathcal{DC}$ (`dc3`)

In skeleton `dc2`, the existence of a potential son of a node in the call tree depends on run-time data, which makes it impossible to compute a static allocation of the nodes at a level of the call tree.

In future research, we plan to investigate a semi-dynamic allocation, i.e., by computing the allocation of the nodes at a level in the call tree with a parallel scan of the number of sons of the nodes at the level above. For now, we restrict ourselves to a fixed-degree $\mathcal{DC}$.

For an efficient static parallelization, it is convenient to bound the number of subproblem instances by a constant. In this case, subtrees of the call tree can be assigned to partitions of the topology at compile time and administration overhead at run time is avoided. In most cases, the degree of the problem division is 2. Examples of higher degrees are, e.g., Karatsuba's polynomial product [2, Sect. 2.6] with a degree of 3 and Strassen's matrix multiplication [25, 14] with a degree of 7. For some algorithms, the degree is not fixed. One example is the multiplication of large integers by Schönhage and Strassen using Fermat's numbers [22], where the division degree is approximately the square root of the input vector size.

Whereas, for a particular division degree, a $\mathcal{DC}$ skeleton can be defined in Haskell (using tuples) and checked at compile time, this cannot be done for the entire class of fixed-degree $\mathcal{DC}$, due to the limitations of the type system of Haskell. Therefore, skeleton `dc3` is defined to be `dc2` with an additional constraint on function `d`. This constraint is written as a list comprehension:

```
dc3 :: Nat->(a->b)->(a->[a])->([b]->b)->Nat->a->b
dc3 k b d c n x = dc2 b dd c n x
   where dd x = [(d x)!!i | i<-[0..k-1]]
```

The definition of `dc3` imposes a run-time restriction on the class of problems: if the customizing function `d` does not always produce a list of length at least `k`, a run-time error occurs. List elements, whose index exceeds `k`, are ignored. We state this condition in Haskell. This way, we can reason about it equationally in correctness proofs.

To simplify equational reasoning and the implementation of the skeleton, we suggest that the definition of `d` always make use of an explicitly given list of length `k`, stated as a list comprehension or as an explicit pattern.

2.4.1 Expressing call-unbalanced fixed-degree $\mathcal{DC}$ in terms of `dc3`

Unfortunately, some algorithms do not guarantee that the call tree is balanced. Furthermore, a lot of algorithms only lead to a balanced tree in the case that the input data is of a particular size. We want to be able to handle these algorithms for all possible sizes. An observation made in Subsect. 2.3 is that we can achieve balance if we transform the divide and combine function to an equivalent, overrun-tolerant form, destroying a possible fixed degree (see Fig. 4(b–c)). The branches that are missing in comparison to a tree of fixed degree have to be simulated in the implementation.

Let us consider what happens in the case when a problem instance is not divisible any further, but the parameter value `n>0` forces further unfolding. Then the divide function sends its input data just down the leftmost path until `n=0` (see Fig. 4(c–d)), the other paths carry useless information.

Similarly, the corresponding combine functions just deliver the value from the leftmost son. How the combine function should behave is therefore determined by an additional tag of a problem resp. solution instance, which counts the number of levels from the actual level to the level of the last division. For an example see Fig. 5, which describes the process of an artificial $\mathcal{DC}$ algorithm on numbers, which sorts in its divide phase, squares in the basic phase and constructs the prefix sum in its combine phase. The following skeleton `dc3tobal` performs the adaptation under the assumption that the user's algorithm matches the list pattern enforcing a fixed degree `k`, as pointed out in the previous subsection:

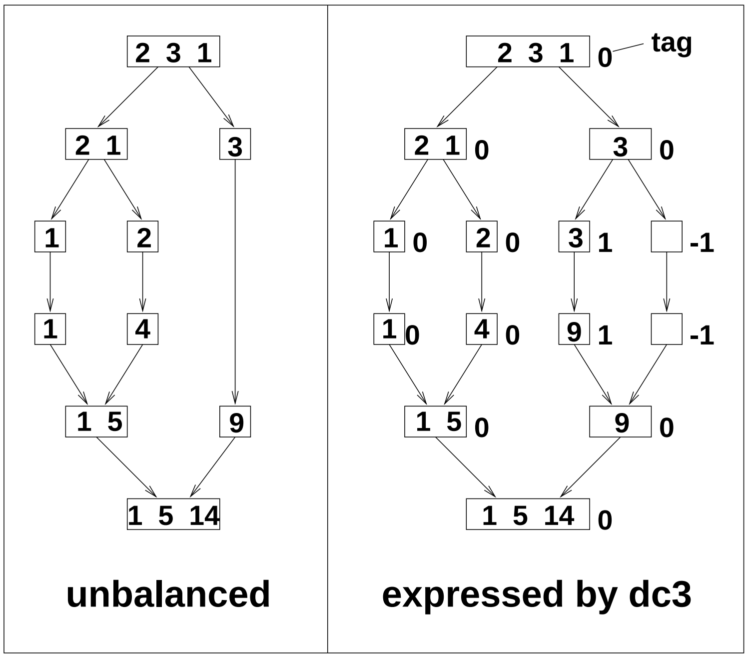

Figure 5: Adaptation to `dc3`

```
dc3tobal :: Int->(a->Bool)->(a->b)->(a->[a])->([b]->b)->Int->a->b
dc3tobal k divisible bb dd cc n x = fst (dc3 k b d c n (x,0))
   where idle    = x
         gen n x = map (const x) [1..n]
         b (x,l) = (bb x,l)
         d (x,l) = if divisible x
                      then kmap k (\d->(d,0)) (dd x)
                      else (x,l+1):gen (k-1) (idle,-1)
         c s     = let ys = map fst s
                       l  = snd (head s)
                   in if l==0 then (cc ys,0)
                              else (head ys,l-1)
         kmap k f xs = [f (xs!!i) | i<-[0..k-1]]
```

3 Transforming call-balanced fixed-degree $\mathcal{DC}$ to loops

In this section, we show how the recursive call-balanced fixed-degree $\mathcal{DC}$ skeleton (`dc3`) can be transformed to an intermediate iterative program which can later be implemented easily on many parallel systems. The important thing concerning the transformations is, that all but the last step are in Haskell, i.e., amenable to equational reasoning.

In Subsect. 3.2, we transform `dc2` to linear recursion. We state in Theorem 1 that the abstract version `dc2a` (see Fig. 1) of the call-balanced skeleton `dc2` is equivalent to the linearly recursive skeleton `it2a` which enumerates the levels of the call graph. In Subsect. 3.3, we use this equivalence to state the equivalence of `dc3a` and `it3a` in Law 7. We present some laws, which introduce concrete versions of the abstract expressions occurring in `it3a`. In Subsect. 3.4, we replace in `it3a` the abstract by the concrete expressions, simplify, replace iterators by list comprehensions, and introduce names for the intermediate values computed by the phases. We obtain the functional target skeleton `it3`, whose equivalence with `dc3` is given by Law 13. it3 iterates across the nodes of a fixed level of the call graph by a further, nested linear recursion. In Subsect. 3.5 we transform it3 to a C program with annotations for parallelism.

But first we define a few Haskell functions, which are used in the remainder of this paper.

3.1 Definition of auxiliary Haskell functions

We use a data type PS (for "powerstructure") to represent a list structure by a tree with empty inner nodes. A single element is defined with the constructor Sgt, a list of powerstructures is made a powerstructure using the constructor Com.

```
data PS a = Sgt a | Com [PS a] deriving (Eq,Show)
```

We use the following functions to work with data type PS:

```
sgt   :: a->PS a
sgt a = Sgt a
unsgt :: PS a->a
unsgt (Sgt a)  = a
com   :: [PS a]->PS a
com as = Com as
uncom :: PS a->[PS a]
uncom (Com as) = as

dmap :: (PS a->PS b)->Nat->(PS a->PS b)
dmap f 0       = f
dmap f n | n>0 = com . map (dmap f (n-1)) . uncom

comp :: [a->a]->a->a
comp [] = id
comp (f:fs) = f . comp fs

down :: (Nat->a->a)->Nat->a->a
down f n = comp [f i | i<-[0..n-1]]

up :: (Nat->a->a)->Nat->a->a
up f n   = comp [f i | i<-[n-1,n-2..0]]

partition :: Nat->Nat->[a]->[[a]]
partition k n xs = [ [xs !! (i*k+j) | j<-[0..k-1]]
                   | i <- [0..k^n-1]]

unpartition :: Nat->Nat->[[a]]->[a]
unpartition k n xs = [ xs !! i !! j | i<-[0..k^n-1], j<-[0..k-1]]

singleton :: a->[a]
singleton x = [x]
```

sgt, unsgt, com and uncom are wrappings resp. unwrappings of data type PS. dmap f n applies a function to the nth level of a powerstructure. comp takes a list of functions and composes them. The functions `down`/`up` take a function and a number `n` and compose this function `n` times with itself, while counting the number down resp. up. The function `partition` takes parameters `k` and `n`, and maps a list of length k^{n+1} bijectively to k^n lists of length `k` taking successive elements. `unpartition k n` is the left inverse of `partition k n`. `singleton` creates a singleton list.

3.2 Transforming `dc2` to linear recursion

Our goal is a linearly recursive program, which iterates through the levels of the call tree. Consider the collection of input data at different levels. At level 0, the input data is a single object (the input data of the problem). At level 1, it is a list (of input data of the subproblems). At level 2, it is a list of lists (of input data of the subproblems of the subproblems), etc. In Haskell, a list and a list of lists are of different type, i.e., a function which can deal with all levels, taking the level as a parameter, is not well-typed. Therefore, we use instead the algebraic data type `PS`, which defines a superset of what we intend to define.

Here is the new Haskell definition for `dc2`, which works on powerstructures instead of lists; we name it `dc2a` (`a` is for *abstract*):

```
dc2a :: (PS a->PS b)->(PS a->PS a)->(PS b->PS b)->Nat->PS a->PS b
dc2a bb dd cc n =
   if n==0 then bb
           else cc . dmap (dc2a bb dd cc (n-1)) 1 . dd
```

Let us show how to express `dc2` in terms of `dc2a`:

```
dc2_by_dc2a :: (a->b)->(a->[a])->([b]->b)->Nat->a->b
dc2_by_dc2a b d c n = unsgt . dc2a bb dd cc n . sgt
   where bb = sgt . b . unsgt
         dd = com . map sgt . d . unsgt
         cc = sgt . c . map unsgt . uncom
```

We observe that the linearly recursive function `it2a`:

```
it2a :: (PS a->PS b)->(PS a->PS a)->(PS b->PS b)->Nat->PS a->PS b
it2a bb dd cc n = down (dmap cc) n . dmap bb n . up (dmap dd) n
```

equals `dc2a`, and state this in Theorem 1.

THEOREM 1 `dc2a = it2a`

Proof: by induction on ***n***

$\boxed{\text{n=0}}$

```
  dc2a bb dd cc 0 xx
=     { Selection of the then branch }
  bb xx
=     { Identity twice }
  (id . bb . id) xx
=     { Definition of down, up, and dmap }
  (down (dmap cc) 0 . dmap bb 0 . up (dmap dd) 0) xx
=     { Definition it2a }
  it2a bb dd cc 0 xx
```

$\boxed{\texttt{n} \rightarrow \texttt{n+1}}$

```
  dc2a bb dd cc (n+1) xx
=    { Selection of the else branch }
  (cc . dmap (dc2a bb dd cc n) 1 . dd) xx
=    { Induction hypothesis }
  (cc . dmap (it2a bb dd cc n) 1 . dd) xx
=    { Definition of it2a }
  (cc . dmap (down (dmap cc) n . dmap bb n . up (dmap dd) n) 1 . dd) xx
=    { dmap distribution }
   (cc . dmap (down (dmap cc) n) 1 . dmap (dmap bb n) 1 .
    dmap (up (dmap dd) n) 1 . dd) xx
=    { Laws 2, 3, and 6 }
   (cc . down (dmap cc . (+1)) n . dmap bb (n+1) .
    up (dmap dd . (+1)) n . dd) xx
=    { Definition of dmap }
   (dmap cc 0 . down (dmap cc . (+1)) n . dmap bb (n+1) .
    up (dmap dd . (+1)) n . dmap dd 0) xx
=    { Laws 4 and 5 }
  (down (dmap cc) (n+1) . dmap bb (n+1) . up (dmap dd) (n+1)) xx
=    { Definition it2a }
  it2a bb dd cc (n+1) xx
```

□

Law 2 `dmap (down (dmap f) n) m  =  down (dmap f . (+m)) n`

Law 3 `dmap (up (dmap f) n) m  =  up (dmap f . (+m)) n`

Law 4 `f 0 . down (f . (+1)) n  =  down f (n+1)`

Law 5 `up (f . (+1)) n . f 0  =  up f (n+1)`

Law 6 `dmap (dmap f n) m  =  dmap f (n+m)`

3.3 The abstract skeletons for `dc3`

Skeleton `dc3a` is the abstract version of `dc3` (see Fig. 1), and skeleton `it3a` is its iterative counterpart:

```
dc3a :: Nat->(PS a->PS b)->(PS a->PS a)->(PS b->PS b)->Nat->PS a->PS b
dc3a k b d c n = dc2a b dd c n
   where dd x = com [uncom (d x) !! i | i<-[0..k-1]]

it3a :: Nat->(PS a->PS b)->(PS a->PS a)->(PS b->PS b)->Nat->PS a->PS b
it3a k b d c n = it2a b dd c n
   where dd x = com [uncom (d x) !! i | i<-[0..k-1]]
```

Law 7 `dc3a = it3a`

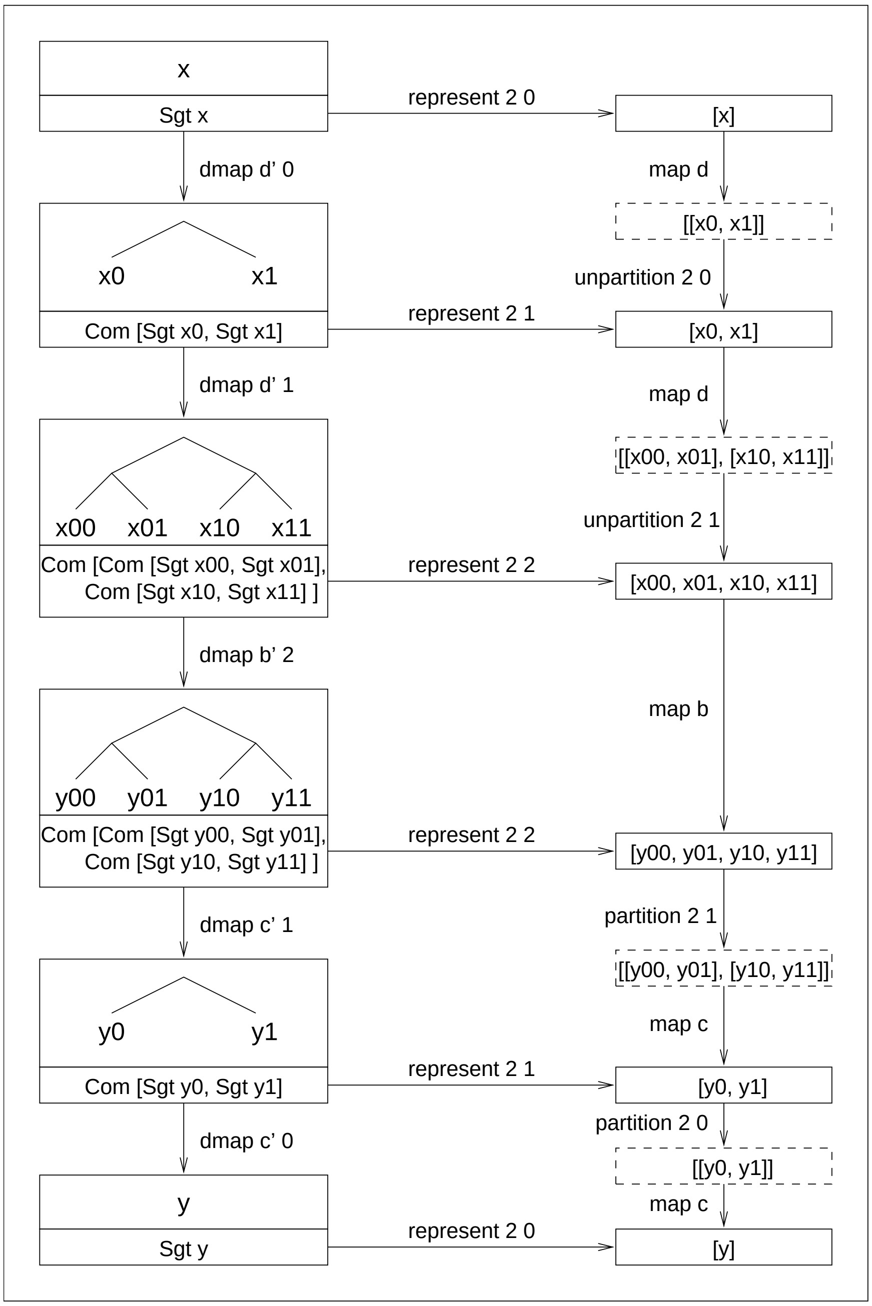

Figure 6: Effect of the linearization

Proof: By Thm. 1, `dc2a` and `it2a` are equivalent. We impose the fixed division degree on both skeletons, obtaining `dc3a` and `it3a`.

□

In the previous subsection, the function `dmap` is used for distributing a function call to all nodes at a fixed level of the call tree. For the divide function and the basic function, this is the leaf level. For the combine function, this is the level above the leaf level because the information stored in the leaf nodes is combined in their father nodes and the leaf nodes are deleted. In order to express this application easily by a single linear recursion, the nodes at the mentioned level have to be represented by a one-dimensional data structure; we use a list. The representation mapping is given in Def. 8. The structural information that is contained in the tree is used to derive functions which manipulate the linear structure.

To picture how this is done, see Fig. 6, where:

```
d' = com . map sgt . d . unsgt
b' = sgt . b . unsgt
c' = sgt . c . map unsgt . uncom
```

The abstract level (the level which makes use of the tree or powerstructure) is depicted on the left side, the concrete level (performing the corresponding computations on a linear structure) on the right side. The abstract level is used to show that the transformations are correct because it contains the structural information needed. The computation proceeds from top to bottom. If one projects all tree drawings in this figure onto each other, one obtains the complete call tree.

In the rest of this subsection, we work out how the abstract functions are expressed in terms of the concrete functions.

3.3.1 Expressing the abstract functions in terms of the concrete ones

Our aim is to get a linear representation of the nodes at level `n` of the balanced `k`-degree call tree. This representation is defined below by function `represent k n`, and depicted in Fig. 6:

Definition 8 Linearization

We call the mapping of a level of a `k`-degree tree to a list a *linearization*. We define the function `lintrans k n`, which performs this mapping of level `n`, and its inverse `invlintrans k n`. Based on these, we define a representation function `represent k n`, which expects a tree of depth `n`, and an abstraction function `abstract k n`:

```
lintrans :: Nat->Nat->PS a->[PS a]
lintrans k 0 (Sgt x) = [Sgt x]
lintrans k n x | n>0 = [(lintrans k (n-1) (uncom x !! i)) !! j
                       | i<-[0..k-1], j<-[0..k^(n-1)-1]]


invlintrans :: Nat->Nat->[PS a]->PS a
invlintrans k 0 [Sgt x] = Sgt x
invlintrans k n x | n>0 = com [invlintrans k (n-1)
                                [x!!(i*k^(n-1)+j) | j<-[0..k^(n-1)-1]]
                              | i<-[0..k-1]]

represent :: Nat->Nat->PS a->[a]
represent k n = map unsgt . lintrans k n

abstract :: Nat->Nat->[a]->PS a
abstract k n  = invlintrans k n . map sgt
```

LAW 9 Concretization of `dmap`

```
dmap f n = invlintrans k n . map f . lintrans k n
```

Let us derive the concrete from the abstract implementation. Note that Fig. 6 consists of commuting diagrams. Starting from a position at the abstract side, one can first perform the abstract function, moving downwards, and then go to the representation side, or one can first apply the representation function and then the concrete function. The existence of the concrete function is guaranteed by the invertibility of the representation function, but, a priori, the concrete function is unknown. Aside from the basic function, the concrete function consists of a composition of a calculation on the data and a type adaptation. In order to find the concrete function, we first write down the equation for the commuting diagram in general, i.e., using variables where different expressions occur:

```
represent k n . dmap f m = g . represent k p,
```

where (`dmap f m`) is the abstract implementation, `g` the concrete one, and everything but `g` is known. Because the inverse of (`represent k p`) is (`abstract k p`), we can compute `g` by using the equation (`g = represent k n . dmap f m . abstract k p`).

We can replace each occurrence of the abstract function (`dmap f m`) by first applying the representation, then the concrete implementation `g` and then the abstraction. Of course, this only makes sense if we get rid of representation and abstraction functions.

We exploit properties of the customizing functions that are made explicit by functions `sgt`, `unsgt`, `com` and `uncom` in combination together and with `map`: A divide step increments the height of the tree, because the divide function takes a leaf (a problem) and delivers a tree of height 1 (containing the subproblems). The basic step maintains the height of the tree. A combine step decrements the height of the tree because the combine function is applied to all nodes above the leaf level, takes a subtree of height 1 (containing the subproblem solutions) and delivers a leaf (a solution).

We present the following three concretization laws, one for each phase. The transformations are straight-forward.

Law 10 Concretization of `dmap d' n`

```
   dmap (com . map sgt . d . unsgt) n
=  abstract k (n+1) . unpartition k n . map d . represent k n
```

Proof

```
  dmap (com . map sgt . d . unsgt) n
=    { Law 9 }
  invlintrans k n . map (com . map sgt . d . unsgt) .  lintrans k n
=    { map distribution }
  invlintrans k n . map com . map (map sgt) . map d . map unsgt . lintrans k n
=    { invlintrans k n . map com = invlintrans k (n+1) . unpartition k n }
  invlintrans k (n+1) . unpartition k n . map (map sgt) . map d . represent k n
=    { property unpartition }
  invlintrans k (n+1) . map sgt . unpartition k n . map d . represent k n
=    { definition abstract }
  abstract k (n+1) . unpartition k n . map d . represent k n
```

□

Law 11 Concretization of `dmap b' n`

```
   dmap (sgt . b . unsgt) n
=  abstract k n . map b . represent k n
```

Law 12 Concretization of `dmap c' n`

```
   dmap (sgt . c . map unsgt . uncom) n
=  abstract k n . map c . partition k n . represent k (n+1)
```

Proof: Analogously to Law 10

3.4 Transformations towards the functional target skeleton `it3`

Let us give an overview of the transformation process divided into the following subsections:

- The definition of the balanced fixed-degree $\mathcal{DC}$ skeleton in its recursive form `dc3` is the topic of Subsect. 3.4.1. It can be expressed in terms of `dc2a` (like its predecessor `dc2`, but with the additional fixed-degree constraint on `d`), the recursive form at the abstract side.
- In Subsect. 3.4.2, we use our results of Theorem 1, which states the equivalence of recursion and iteration on the abstract side. That is, `dc3` can be expressed in terms of `it2a`.
- In Subsect. 3.4.3, we replace every function on the abstract side by its linear representation developed in Subsect. 3.3.1.
- In Subsect. 3.4.4, we eliminate all remaining (useless) conversions between the abstract and the concrete side.
- Subsect. 3.4.5 presents the functional target skeleton `it3`.

3.4.1 Expressing `dc3` by the abstract `dc2a`

Remember the skeleton for expressing `dc2` in terms of `dc2a`. We can reuse `dc2a` in the skeleton for `dc3`, because `dc3` is a specialization of `dc2`.

```
dc3_by_dc2a :: Nat->(a->b)->(a->[a])->([b]->b)->Nat->a->b
dc3_by_dc2a k b d c n = unsgt . dc2a bb dd cc n . sgt
   where bb = sgt . b . unsgt
         dd = com . map sgt . (\xs->[(d xs)!!i | i<-[0..k-1]]) . unsgt
         cc = sgt . c . map unsgt . uncom
```

3.4.2 Equivalence of recursion and iteration on the abstract side

We know that `dc2a` and `it2a` are equivalent, so we can substitute `dc2a` with the (non-recursive) `it2a`:

```
dc3_by_it2a :: Nat->(a->b)->(a->[a])->([b]->b)->Nat->a->b
dc3_by_it2a k b d c n x = (unsgt . down (dmap (sgt . c . map unsgt . uncom)) n
                          . dmap (sgt . b . unsgt) n
                          . up   (dmap (com . map sgt . dd . unsgt)) n
                          . sgt) x
                    where dd x = [(d x)!!i |i<-[0..k-1]]
```

3.4.3 Replacing all abstract operations by their representation

Now, we replace the operations on the trees by their representation developed in Subsect. 3.3.1, obtaining it3_1.

```
it3_1 :: Nat->(a->b)->(a->[a])->([b]->b)->Nat->a->b
it3_1 k b d c n =
   unsgt
 . down (\m ->   abstract k m
               . map c . partition k m
               . represent k (m+1) ) n
 . abstract k n . map b . represent k n
 . up   (\m ->   abstract k (m+1)
               . unpartition k m . map (\xs->[(d xs)!!i | i<-[0..k-1]])
               . represent k m     ) n
 . sgt
```

3.4.4 Elimination of useless conversions

At this point, we can eliminate all occurrences of a representation function composed with an abstraction function and vice versa, which results in it3_2. The equivalence between it3_1 and it3_2 can be proved by induction over the structure of up and down. The functions represent k n and abstract k n tell us about the size of the lists, so we can replace map by list comprehensions with parametrized sizes. This is necessary for the later code generation.

```
it3_2 :: Nat->(a->b)->(a->[a])->([b]->b)->Nat->a->b
it3_2 k b d c n =
  head
  . down (\m xs -> [ c [xs !! (k*i+j) | j <- [0..k-1]] | i <- [0..k^m-1]]) n
  . (\xs->[b (xs!!i) | i<-[0..k^n-1]])
  . up   (\m xs -> [d (xs !! (l `div` k)) !! (l `mod` k)
                  | l <- [0..k^(m+1)-1]] ) n
  . singleton
```

3.4.5 The functional target skeleton

The *free schedule* of a parallel computation assigns each operation to the first possible time step permitted by the data dependences (not by the number of available resources). Our aim is to derive a functional target whose free schedule can be detected easily. In our target skeleton, what will be a loop after the translation to C is a list comprehension. The loop will be sequential if list elements depend on their predecessor, and parallel, if they are independent of each other.

In the skeleton it3_2 there are still linear recursions left which perform iterations through the levels of the call tree. The intermediate results of this recursions are now stored in lists g and h in skeleton it3, in order to make the correspondence to a C program with single assignment obvious.

```
it3 :: Nat->(a->b)->(a->[a])->([b]->b)->Nat->a->b
it3 k b d c n x =
  let a0 = singleton x
      a1 = (let h = a0:[ [d (h !! (m-1) !! (l`div`k)) !! (l`mod`k)
                          | l <- [0..k^m-1]]
                        | m<-[1..n]] in h) !! n
      a2 = [b (a1!!i) | i<-[0..k^n-1]]
      a3 = (let g = a2:[ [ c (let args = [g !! (m-1) !! (k*i+j)
                                          | j <- [0..k-1]]
                               in args)
                          | i <- [0..k^(n-m)-1]]
                        | m<-[1..n]] in g) !! n
      a4 = head a3
  in a4
```

LAW 13   `dc3 = it3`

Proof: The representation of `dc3a` is `dc3`, because the representation of `dc2a` is `dc2` and we imposed a fixed degree on both `dc2` and `dc2a`, obtaining `dc3` and `dc3a`. Now, because Law 7 states the equivalence of `dc3a` and `it3a`, the representation of `dc3a`, i.e., `dc3` is equivalent to the representation of `it3a`, which is `it3` due to application of the representation function. □

The (Haskell) semantics of the given source skeleton `dc3` and the (functional) target skeleton `it3` are equal; the difference is in the efficiency, e.g., with respect to a cost calculus like [23].

3.5 Transformation to C

In this subsection, we transform the functional target skeleton `it3` into an imperative skeleton in C. We use correspondences of data structures resp. control structures between Haskell and C which should be obvious. We do not provide a formal proof of their semantic correctness; this would require a formal semantics for both Haskell and C.

3.5.1 Correspondences between Haskell and C

1. `(Int,+,-,*,`div`,`mod`,^ )` in Haskell and `(int,+,-,*,/,%,pow(.,.))` in C correspond.

2. The run-time argument of the Haskell function is referred to as `input` in the C code, the result of the function is assigned to the variable `output`.

3. The body of a non-recursive `let` expression with equations sorted in the direction of the data dependences is transformed into a sequence of C assignments.

4. Lists in Haskell are represented in C as arrays. This correspondence is sound with respect to the structure, because in C different elements of an array can represent arrays of different sizes (like in Haskell lists can contain lists of different lengths).

5. Due to the correspondence (4), the application of the transformed `singleton` function to `a[0]` has to be `a` and the one of `(!!i)` to `a` has to be `a[i]` (especially the application of `head` to `a` has to correspond to `a[0]`).

6. List comprehensions in Haskell have a correspondence to loops in C, which iterate through an array. Whether a loop can be implemented in parallel, depends on lack of data dependences between array elements.

3.5.2 Implementational issues

`seqfor` defines a loop whose iterations are executed in sequence. `parfor` defines a loop whose iterations can be processed in parallel. If programmed correctly, both `seqfor` and `parfor` must respect the semantics of the ordinary (sequential) `for` loop. This is expressed by the C definitions:

```
#define seqfor for
#define parfor for
```

3.5.3 The C code

Applying the correspondences from the Subsect. 3.5.1 to `it3`, we obtain the following C program. The functions `divide`, `basic`, and `combine` denote the result of a compilation of `d`, `b`, and `c` resp. from Haskell to C. They take as first argument the location of the result and as second argument the run-time argument of the corresponding Haskell function. The function `divide` takes an additional argument which determines the position of the needed element in the list which the corresponding Haskell function `d` delivers, i.e., the selection of an element of a list is pushed into the function `divide`.

```
/* input */
input(&(a0[0]));

/* divide phase */
h[0] = a0;
seqfor(m=1;m<=n;m++)
   parfor(l=0;l<pow(k,m);l++)
      divide(&(h[m][l]),h[m-1][l/k],l%k);
a1 = h[n];

/* basic phase */
parfor(i=0;i<pow(k,n);i++)
   basic(&(a2[i]),a1[i]);

/* combine phase */
g[0] = a2;
seqfor(m=1;m<=n;m++)
   parfor(i=0;i<pow(k,n-m);i++)
      {
      parfor(j=0;j<k;j++)
         args[m][i][j] = g[m-1][k*i+j];
      combine(&(g[m][i]),args[m][i]);
      }
a3 = g[n];

/* output */
output(a3[0]);
```

In the two-dimensional arrays we use in the divide and combine phase, the first index corresponds to time and the second to (processor/memory) space.

Because the time component of all data dependence vectors is 1 and nests of an outer sequential and an inner parallel loop require global synchronization after each step of the outer loop [18], it is sufficient to keep memory space just for two successive steps of the outer loop.

4 Instantiation with balanced data division and elementwise operations

In this section, we instantiate the call-balanced fixed-degree $\mathcal{DC}$ skeleton in two ways.

First we impose a balance on the data division. This means that the data is split into a fixed number of partitions of the same size. Each partition is assigned to the part of the topology which handles the according problem instance. In Subsect. 2.2.2, the partitioning was delegated to a customizing function. Here, it is imposed by a structural constraint on the skeleton. We choose the same degree for the data division as is given for the problem division. This has the advantage that communications are avoided in cases where each each partition makes up the input of a separate subproblem.

The specialization with a balanced data division does not incur a loss of expressive power. The only difference is that the access to a data element is now indirect: first one selects the partition which contains the element and then the position of the element in the partition.

The second specialization restricts the expressive power of the skeleton to elementwise operations on the zip of the partitions, i.e., only elements which have the same index within their partition can be combined. The advantage is that communications become much more regular and the customizing functions can be viewed as vectorized operations.

In the remainder of this section, we proceed as follows: Subsect. 4.1 revisits the generation of the inner, parallel loop. In Subsect. 4.2 we make the actual level of recursion available as a parameter for modified customizing functions. This is helpful for the further proof process. In Subsect. 4.3 we present a skeleton which partitions the data using doubly nested lists. Subsect. 4.4 then presents one possible way of making efficient use of this partitioning: by elementwise operations between the partitions. The skeleton with elementwise operations is then transformed in Subsect. 4.5 to a skeleton based on list comprehensions. Subsect. 4.6 shows the conversion of this skeleton into C with annotations for parallelism. We show the use of the skeleton with the `scan` function as an example in Subsect. 4.7.

4.1 Generating the inner, parallel loop, revisited

In Subsect. 3.4.5, the work of step `m` of the divide phase is given by the Haskell expression

```
[d (h !! (m-1) !! (l ‘div‘ k)) !! (l ‘mod‘ k) | l <- [0..k^m-1]]
```

which becomes via the transformation in Subsect. 3.5.3

```
parfor(l=0;l<pow(k,m);l++)
  divide(&(h[m][l]),h[m-1][l/k],l%k);
```

h is a two-dimensional array, indexed by the level m of the call tree and by the position l at a level. That is, h[m][l] identifies a subproblem instance. All subproblem instances at a level can be computed in parallel.

Now, with the skeleton instantiated with a balanced data division and elementwise operations, we can extract information about d and the data d is applied to: h[m][l] is itself an array, say h[m][l][j], and d contains a map of a function g. So g is applied in parallel to all h[m][l][j]. This is exploited by transformation to an additional inner, parallel loop. Both parallel loops, the one on l and the one on j can be merged. Then, the multiplication of the number of calls (enumerated by p) is compensated by a diminishing number of data elements involved in each call (enumerated by j).

In a previous paper [13], we have presented a geometrical model which illustrated this compensation. E.g., in the case of a binary problem division, the parallel execution of our target skeleton can be interpreted geometrically as performing a descend/ascend scheme on a hypercube, where in each divide and combine step communications are restricted to one dimension of a hypercube and the number of this dimension is decremented in the divide phase and incremented in the combine phase.

Although the hypercube is a model often used in parallel programming, it is not a consequence of the skeleton; other models can be used as well. The derivation of a parallel target program by means of equational reasoning does not require a geometrical model at all.

4.2 dc3 with level information

The semantics and also the type of the following skeleton dc3li differ from dc3 in that the customizing functions d and c take as a first argument s the number of the recursion level and as second argument y the one they take in dc3. The customizing functions can compute the size of the data from the level number. This argument facilitates the partitioning of the data; the level number disappears again in a further specialization.

```
dc3li :: Nat->(a->b)->(Nat->a->[a])->(Nat->[b]->b)->Nat->a->b
dc3li k b d c n x = fst (dc3 k bb dd cc n (x,n))
   where bb (y,s) = (b y,s)
         dd (y,s) = map (\z->(z,s-1)) (d s y)
         cc xs    = let s    = snd (head xs) + 1
                        sols = map fst xs
                    in (c s sols,s)
```

The argument x is paired with the number n of levels of recursion, which can be computed from the size of the input data. Usually it will be the ceiling of a logarithm to a particular radix not greater than k. During the divide phase, n is decremented in each step; in the combine phase, it is incremented in each step.

4.3 Balancing the distribution of data

In skeleton dc3baldata, the input and output data is in a list, i.e., the list structure is used for data aggregation. Any other structure that is indexable, e.g., the array structure, could be used instead. As a random access structure, the array is an easier target for a parallelization than the list. We choose the list anyway because it is the most common data structure in functional programming. With list comprehensions, we can treat lists like arrays. We need three auxiliary functions: (`kzip k n`) partitions a list of length $\mathtt{k}^{\mathtt{n}}$ into a list of length $\mathtt{k}^{\mathtt{n-1}}$, each of whose elements is a list of `k` elements with the same index relative to the partition. (`unkzip k n`) is the inverse of (`kzip k n`). (`sitranspose n m`) transposes a matrix of size ($\mathtt{n}\times\mathtt{m}$) represented by a list of lists. Its inverse is (`sitranspose m n`).

```
kzip :: Nat->Nat->[a]->[[a]]
kzip k n xs = let s = k^(n-1)
              in [[xs!!(i*s+j)| i<-[0..k-1]] | j<-[0..s-1]]

unkzip :: Nat->Nat->[[a]]->[a]
unkzip k n xs = let s = k^(n-1)
                in [xs!!(i `mod` s)!!(i `div` s) | i<-[0..k^n-1]]

sitranspose :: Nat->Nat->[[a]]->[[a]]
sitranspose n m xss = [ [ xss!!i!!j | i<-[0..n-1] ]
                      | j<-[0..m-1]]
```

`dc3baldata` calls `dc3li` with modified customizing functions and imposes additional constraints on the structure of the data. The modified customizing functions are given the number of the recursion level as parameter `m`. The customizing functions `d` and `c`, which are passed as parameters to `dc3baldata`, have to map between lists of lists, i.e., they have to respect the balanced division of data by first addressing one of the partitions of equal size and then the location within the selected partition. The modified divide function `dd` first partitions the data in a balanced way (applying (`kzip k m`)), then applies `d`, and finally transposes the list of data elements of the subproblems into a list of subproblems containing lists of data elements.

```
dc3baldata :: Nat->(a->b)->([[a]]->[[a]])->([[b]]->[[b]])->Nat->[a]->[b]
dc3baldata k b d c n = dc3li k bb dd cc n
   where bb   = map b
         dd m = sitranspose (k^(m-1)) k . d . kzip k m
         cc m = unkzip k m . c . sitranspose k (k^(m-1))
```

Although function `d` need not necessarily be applied elementwise, corresponding elements are close together in its input. This does not restrict the power of the divide step, but it allows for an easy instantiation with functions which are either purely elementwise or contain at least a large share of elementwise operations. This is often the case and a good strategy for obtaining efficient parallel algorithms. The modified basic function `b` is applied to the only element of a singleton list by `map`. The combine function `cc` does the same as `dd` in the opposite direction and with `c` instead of `d`. First, we undo the transposition applied by `dd`. If the lists are distributed on a hypercube, in the case of binary problem division, the effect of a transposition is a change of the dimension which enumerates the partitions. After that, `c` is applied, delivering a list of corresponding elements of the partitions, which is then unzipped to separate the data elements which belong to different partitions.

If we specialize the skeleton `bal_dc0_by_dc1` by eliminating the input data as argument of the combine function, adequate balancing functions, which make use of parameter `m`, can be obtained from `dd m` and `cc m` by replacing `d` and `c` with the identity.

4.4 Elementwise operations on balanced data partitions

Skeleton `dc4` restricts the divide and combine function in `dc3baldata` to elementwise operations, using the function `map`. The type differs slightly from `dc3`. Here, divide function `d` and combine function `c` are supposed to take a list of length `k` as input and output. The input elements of divide and the output elements of combine correspond to the data elements with the same index in different partitions, the output elements of divide and the input elements of combine correspond to the data elements with the same index in different subproblems resp. subproblem solutions. Not all list elements have to carry useful data. The dummy places originate from empty partitions in the input and output data distribution.

```
dc4 :: Nat->(a->b)->([a]->[a])->([b]->[b])->Nat->[a]->[b]
dc4 k b d c n  = dc3baldata k b (map d) (map c) n
```

4.5 The functional target for `dc4`

The aim of this subsection is to make the elementwise operations of `dc4`. We achieve this by a sequence of transformations which consists of 17 macro steps, where each step itself involves, e.g., an induction, flattening of lists, index shifts, projection, arithmetic simplification at multiple positions, etc. The interested reader can obtain the transformation from the authors. We present just the result, the skeleton `it4`, which is semantically equivalent to `dc4` but accesses list elements by index and can therefore be translated easily into a C program with annotations for parallelism.

Before we present `it4`, we have to introduce two auxiliary functions. `digpos k d v` computes of the representation of number `v` in radix `k`, the digit at position `d`. `digchange k d v i` replaces this digit by `i`.

```
digpos :: Nat->Nat->Nat->Nat
digpos k d v = v `div` k^d `mod` k

digchange :: Nat->Nat->Nat->Nat->Nat
digchange k d v i = v + (i - digpos k d v) * k^d

it4 :: Nat->(a->b)->([a]->[a])->([b]->[b])->Nat->[a]->[b]
it4 k b d c n x =
  let a0 = x
      a1 = (let h = a0:[ [ let arg_d = [ h !! (m-1) !! digchange k (n-m) q i
                                       | i<-[0..k-1]]
                           in d arg_d !! digpos k (n-m) q
                         | q<-[0..k^n-1]]
                       | m<-[1..n]] in h) !! n
      a2 = [b (a1!!q) | q<-[0..k^n-1]]
      a3 = (let g = a2:[ [ let arg_c = [ g !! (m-1) !! digchange k (m-1) q i
                                       | i<-[0..k-1]]
                           in c arg_c !! digpos k (m-1) q
                         | q<-[0..k^n-1]]
                       | m<-[1..n]] in g) !! n
  in a3
```

LAW 14 `dc4 = it4`

Proof: dc4 and it4 have both been obtained from dc3 resp. it3 by specialization with elementwise operations, and dc3 and it3 are equivalent by Law 13.
□

4.6 Transformation to C

Skeleton it4 can be transformed to C like it3 in the previous section.

Without flattening the nested list comprehensions during the equational reasoning process, we would obtain two nested parallel loops. However, we can get away with just one parallel loop, which even has a constant extent. This form (SEQ of PAR) makes this program data-parallel, so it can be implemented easily on SIMD or, after conversion into an SPMD program, on MIMD machines. We show only the most interesting parts of the program, i.e., the loops that implement the skeleton. Because of the lazy semantics of Haskell, we felt free not to evaluate the unused elements that are produced by functions d and c. Instead of d, b, and c, we use divide, basic, and combine, respectively. The functions take as first argument the location of the result and as second argument the run-time argument of the corresponding Haskell functions. divide and combine expect an additional argument that indicates which element of the result list is desired. The function pow(b,m) determines b to the power of m. The sequential loops enforce a global synchronization, so the first indices of the arrays do not range over all values of m, but just over their values in the modulus of 2, i.e., the set {0,1}. We implemented this using functions new(m) and old(m). digpos and digchange are equivalent to the Haskell functions defined in the previous subsection.

```
/* input */
parfor(q=0;q<pow(k,n);q++)
   input(&(a0[q]),q);

/* divide phase */
h[0] = a0;
seqfor(m=1;m<=n;m++)
   parfor(q=0;q<pow(k,n);q++)
      {
      for (i=0;i<k;i++)
         arg_d[q][i] = h[old(m)][digchange(k,n-m,q,i)];
      divide(&(h[new(m)][q]),arg_d[q],digpos(k,n-m,q));
      }
a1 = h[old(m)];

/* basic phase */
parfor(q=0;q<pow(k,n);q++)
   basic(&(a2[q]),a1[q]);

/* combine phase */
g[0] = a2;
seqfor(m=1;m<=n;m++)
   parfor(q=0;q<pow(k,n);q++)
      {
      for(i=0;i<k;i++)
         arg_c[q][i] = g[old(m)][digchange(k,m-1,q,i)];
      combine(&(g[new(m)][q]),arg_c[q],digpos(k,m-1,q));
```

```
    }
a3 = g[old(m)];

/* output */
parfor(q=0;q<pow(k,n);q++)
   output(q,a3[q]);
```

4.7 Example scan

We take the scan function as a short example to demonstrate the use of skeleton `dc4`. The scan function takes an associative operator (`op :: a->a->a`) and a list of type `[a]` with elements, say a_0 to a_{m-1} and computes a list of the same type and length with elements, say, b_0 to b_{m-1}, where $b_0 = a_0$ and $(\forall\, i \,:\, 0 < i < m \,:\, b_i = b_{i-1}\,\text{`op`}\,a_i)$. The scan function is a useful auxiliary function in many parallel algorithms, especially sorting algorithms. In [5], a parallel algorithm for scan is presented which fits into skeleton `dc4` after applying a method called "broadcast elimination". We present the program for scan below with the additional parameter `n` for determining the recursion depth.

```
scan :: (a->a->a)->Nat->[a]->[a]
scan op n xs = map fst (dc4 2 b d c n xs)
   where d [x,y] = [x,y]
         b x  = (x,x)
         c [(x,sx),(y,sy)] = let s = sx `op` sy
                             in [(x,s),(sx `op` y,s)]
```

The divide function `d` behaves like the identity. The basic function copies the input into both positions of a pair. The role of the pair during the combine phase is that the first position contains the result value for that particular position with respect to the subproblem instance, and the second contains the result value of the last element of the partition. So, in the combine phase, every operation for an element simulates the operation for the last element of that partition too. This implements the technique called "broadcast elimination". A combine function without broadcast elimination would pass the left solution as left part of the result and combine the last element of the left solution with `op` elementwise with all elements of the right solution delivering the right part of the result. This *with-all-elements* operation requires a broadcast, which is eliminated by simulation: a copy of the last element of the left solution is stored as the second element of the pair, in all positions, and can be accessed by the right part via elementwise operations in the zip of the partitions. Finally, the solution can be obtained by extraction of the first element.

If `scan` is implemented in C with annotations, pairs can be represented by the C data structure `struct`.

5 Strassen's matrix multiplication

Strassen's matrix multiplication [25] computes the product of two matrices of size $m \times m$ sequentially in time of $O(m^{\log_2 7})$ ($\log_2 7 \approx 2.81$) instead of $O(m^3)$ of the trivial algorithm. For a parallel computation, the gain is in the savings of processors. Where, e.g., for the trivial algorithm, 512 ($= 8^3$) processors are necessary to reduce a problem of size $2^{n+3} \times 2^{n+3}$ to problems of size $2^n \times 2^n$, which can be solved in parallel, our modification of Strassen's algorithm requires only 343 ($= 7^3$) processors. Minor disadvantages are the overhead in parallel dividing and combining, and a more complicated data dependence pattern which may lead to more communications on some machines.

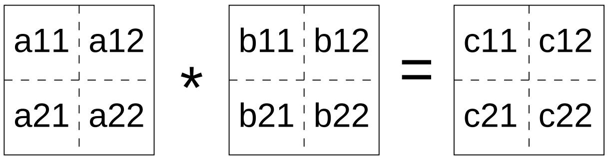

Figure 7: Matrix partitions

In the following program `strassen`, matrices are represented by lists of rows, where each row is represented by a list of column elements. Program `strassen` takes a parameter `n` and two matrices `xss` and `yss` of size $2^n \times 2^n$, and returns the product of `xss` and `yss`. Fig. 7 shows one step of the recursive decomposition of the matrices. How the `cs` are computed from the `as` and `bs` can be taken from the `where` clause of program `strassen`.

```
strassen :: Num a => Nat->[[a]]->[[a]]->[[a]]
strassen n xss yss =
   (   d1d2 n
      .  from_quadtree n
         .  project 4 7 n
            .  dc4 7 b d c n
         .  embed 4 7 n (0,0)
      .  to_quadtree n
    . d2d1 n )
   (zipWith zip xss yss)
   where b (a,b) = a*b
         d [(a11,b11),(a12,b12),(a21,b21),(a22,b22),_,_,_] =
           [(a11+a22,b11+b22),
            (a21+a22,b11),
            (a11,b12-b22),
            (a22,-b11+b21),
            (a11+a12,b22),
            (-a11+a21,b11+b12),
            (a12-a22,b21+b22)]
         c [m1,m2,m3,m4,m5,m6,m7]
           = let c11 = m1+m4-m5+m7
                 c12 = m3+m5
                 c21 = m2+m4
                 c22 = m1+m3-m2+m6
             in [c11,c12,c21,c22,0,0,0]
```

`strassen` is based on the skeleton `dc4`, but the data are rearranged as follows:

1. The two input matrices are zipped together with `zipWith zip xs ys`
2. The zipped matrices, represented as list of lists, are transformed with `d2d1` to a single list, whose elements are in row major order.

3. `to_quadtree` performs a bit-unshuffle permutation, which changes the order to the leaf sequence of a complete quadtree. In the quadtree, each non-leaf node represents a matrix by dividing it into four submatrices. The principle of Strassen's algorithm is to perform elementwise operations on these submatrices.

4. The function `embed` inserts empty data partitions, because the problem division, whose degree is 7, exceeds the data division, whose degree is 4.

The definitions of these functions can be found in the appendix. After `dc4` has been applied, index transformations 2–4 have to be reversed. (For an efficient implementation, the allocation of input and output data for `dc4` should be computed from these index transformations, i.e., no adaptations should be made at run time.)

To express Strassen's algorithm with the parallel C skeleton presented in Subsect. 4.6, one has to compile the customizing functions from Haskell to C. We are working on a compiler which translates a tiny subset of Haskell to C for this purpose.

6 Results and related work

Starting with a general specification of $\mathcal{DC}$ we have obtained, through a series of stepwise refinements of the skeleton, a data-parallel nested loop program for a class of $\mathcal{DC}$ algorithms. From `dc2` on, each specialized skeleton can be implemented by a parallel loop program, representing a different class of $\mathcal{DC}$ problems.

Huang et al. [15] have presented a derivation of a parallel implementation of Strassen's matrix multiplication algorithm using tensor product formulas. The result is a loop program similar to ours, but with a nesting depth of 6; the additional loops ensure that dummy points are not scanned.

We took Strassen's matrix multiplication as a motivating example, and obtained a loop program with, contrary to [15], a nesting depth of just 3 (the outer in time, the next in space, and one additional inner loop to collect the arguments). Our program includes unnecessary operations (which do not add to the execution time or the number of processors), and we assume an appropriate distribution of the input and output data. It is interesting that the two outer loops in [15] (the one which enumerates the levels and the one which enumerates the calls at one level) are the same as ours, i.e., the skeleton `dc3` is the most refined one we have in common. Therefore, future work should include a generalization of `dc3baldata` which allows different division degrees of problem and data, and its specialization to elementwise operations, to obtain the loop program presented in [15].

The strength of our method is in that an algorithm, which is well structured (i.e., fits into a skeleton after adaptations) but hard to implement by hand without recursion (like Strassen's), can be compiled from a functional specification to a low-level target program, whose structure is so simple that every operation can be given a point in time and space at compile time.

Our skeletons are given in the functional language Haskell, i.e., they have a syntax, a type, and a semantics which is referentially transparent. This enables reasoning about the correctness of an implementation. Furthermore, because Haskell is executable and has a C interface, one might use our fast, parallel C program for the skeleton and still keep its parameters, the customizing functions, in Haskell.

Aside from [15], there is other work related to ours:

Misra [19], and Achatz and Schulte [1] restrict themselves to a binary division of data and problems.

Mou's [20] approach allows an arbitrary division of problems and a division of multidimensional data into two parts per dimension, but does not say anything about a higher division degree.

Cole [6] restricts himself to centralized I/O.

None of these papers presents explicitly a nested loop program, and Mou's approach is the only one that is powerful enough to handle Strassen's matrix multiplication with distributed I/O data, aside from ours.

There has been related work in our own group.

First, there is work on the parallelization of the *homomorphism* [4], a basic $\mathcal{DC}$ skeleton somewhat more restrictive than ours. There exists a theory for the transformational parallelization of homomorphisms [24, 10]. The class of *distributable homomorphisms* (*DH*) [9] corresponds to the combine phase of our skeleton `dc4` with a binary divide function (this class is called C-algorithms in [11]). For all functions of the DH class, a common hypercube implementation can be derived by transformation in the Bird-Meertens formalism [9].

The class of "static $\mathcal{DC}$" [11] is an analog of our `dc3` skeleton, however, with the capability of applying different divide (combine) functions at different descendants (ascendants) in the call tree. The analog of our Theorem 1 is their Theorem 2. The result of [11] is an asynchronous, SPMD program as opposed to our synchronous nested loop program.

In our own previous work [13], we obtained loop programs similar to the one presented here by parallelization in a space-time mapping model related to the hypercube. In this paper, we have presented a more precise, top-down development in the framework of equational reasoning.

7 Acknowledgement

Thanks to John O'Donnell for many fruitful discussions in which he convinced us to base our approach on equational reasoning in Haskell. Thanks also to Sergei Gorlatch for discussions. Financial support was provided by the DFG through project RecuR2 and by the DAAD through an ARC exchange grant.

A Auxiliary functions

```
numrep :: Nat->Nat->Nat->[Nat]
numrep radix n v = [v `div` radix^(n-i-1) `mod` radix | i<-[0..n-1]]

numabs :: Nat->Nat->[Nat]->Nat
numabs radix n xs = sum [(xs!!i)*radix^(n-i-1) | i<-[0..n-1]]

embed :: Nat->Nat->Nat->a->[a]->[a]
embed lowval highval n dummy xs = [ let r=numrep highval n i
                                    in if or (map (>=lowval) r)
                                          then dummy
                                          else xs !! numabs lowval n r
                                  | i<-[0..highval^n-1] ]

project :: Nat->Nat->Nat->[a]->[a]
project lowval highval n xs = [ xs !! numabs highval n (numrep lowval n i)
                              | i<-[0..lowval^n-1] ]
```

```
shuffle :: Nat->[a]->[a]
shuffle n v | n `mod` 2 == 0
               = let m = n `div` 2
                 in [v!!((i `div` 2)+m*(i `mod` 2)) | i<-[0..n-1]]

unshuffle :: Nat->[a]->[a]
unshuffle n v | n `mod` 2 == 0
                 = let r = [0..n `div` 2-1]
                   in [v!!(2*i)|i<-r]++[v!!(2*i+1)|i<-r]

bitshuffle :: Nat->Nat->Nat
bitshuffle   n = numabs 2 n . shuffle n   . numrep 2 n

bitunshuffle :: Nat->Nat->Nat
bitunshuffle n = numabs 2 n . unshuffle n . numrep 2 n

to_quadtree :: Nat->[a]->[a]
to_quadtree n xs = [xs !! bitunshuffle (2*n) i | i<-[0..4^n-1]]

from_quadtree :: Nat->[a]->[a]
from_quadtree n xs = [xs !! bitshuffle (2*n) i | i<-[0..4^n-1]]

d1d2 :: Nat->[a]->[[a]]
d1d2 n xs = [ [xs !! (i*2^n+j) | j<-[0..2^n-1]] | i<-[0..2^n-1]]

d2d1 :: Nat->[[a]]->[a]
d2d1 n xss = [ xss!!i!!j | i<-[0..2^n-1], j<-[0..2^n-1]]
```

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, pages 97–127. Springer-Verlag, 1995.

[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] F. L. Bauer and H. Wössner. *Algorithmic Language and Program Development*. Springer-Verlag, 1982.

[4] 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.

[5] B. Carpentieri and G. Mou. Compile-time transformations and optimization of parallel divide-and-conquer algorithms. *ACM SIGPLAN Notices*, 26(10):19–28, 1991.

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

[7] J. Darlington, A. Field, P. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. While. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *Parallel Architectures and Languages Europe (PARLE '93)*, Lecture Notes in Computer Science 694, pages 146–160. Springer-Verlag, 1993.

[8] I. P. de Guzmán, P. G. Harrison, and E. Medina. Pipelines for divide-and-conquer functions. *Computer J.*, 36(3):254–268, 1993.

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

[10] 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.

[11] 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. Workshop at IPPS'97*, pages 80–94, 1997.

[12] P. G. Harrison and H. Khoshevisan. A new approach to recursion removal. *Theoretical Computer Science*, 93:91–113, 1992.

[13] 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.

[14] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Computer Software Engineering Series. Computer Science Press, 1984.

[15] C.-H. Huang, J.R. Johnson, and R.W. Johnson. Generating parallel programs from tensor product formulas: A case study of strassens's matrix multiplication algorithm. In *International Conferenceon Parallel Processing*, pages III 104–108, 1992.

[16] 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.

[17] S. Kindermann. Flexible program and architecture specification for massively parallel systems. In B. Buchberger and J. Volkert, editors, *Parallel Processing: CONPAR 94 – VAPP VI*, Lecture Notes in Computer Science 854, pages 160–171. Springer-Verlag, 1994.

[18] 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.

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

[20] 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.

[21] H. Partsch and P. Pepper. A family of rules for recursion removal. *Information Processing Letters*, 5(6):174–177, December 1976.

[22] A. Schönhage and V. Strassen. Schnelle Multiplikation grosser Zahlen. *Computing*, 7:281–292, 1971.

[23] D. B. Skillicorn. Deriving parallel programs from specifications using cost information. *Science of Computer Programming*, 26:205–221, 1993.

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

[25] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.