J. Functional Programming **9** (3): 279–310, May 1999. Printed in the United Kingdom

Parallelization of divide-and-conquer by translation to nested loops

CHRISTOPH A. HERRMANN and CHRISTIAN LENGAUER

Fakultät für Mathematik und Informatik,
Universität Passau, Germany
(*e-mail:* {herrmann,lengauer}@fmi.uni-passau.de)

Abstract

We present a hierarchical classification of specializations of the divide-and-conquer paradigm. The aim is to identify a subclass of divide-and-conquer algorithms with an efficient parallel implementation which can be viewed as a static space-time mapping. The specializations impose a balanced call tree, a fixed degree of the problem division, and elementwise operations. The correctness of our compile-time transformations is proved by equational reasoning in Haskell; recursion and iteration are handled by induction. We demonstrate the practicality of the skeleton by some examples, one of which is Strassen's matrix multiplication.

Capsule Review

The divide-and-conquer paradigm is simple and intuitive, but has subtle technical dimensions when studied closely. This paper describes a variety of divide-and-conquer strategies (expressed as 'skeletons' in Haskell), proves various properties/equivalences amongst them, and shows how to translate them into iterative loops in C for execution on either sequential or parallel architectures. Although the number of different versions of the paradigm may seem daunting, their definition and use is well motivated, and their interrelations are carefully defined.

1 Introduction

In the divide-and-conquer paradigm ($\mathscr{DC}$) (Aho *et al.*, 1974; Horowitz & Sahni, 1984), the solution of a problem is specified by dividing the problem into a number of subproblems, which are solved recursively until a basic case is reached, and then combining the solutions of the subproblems to the solution of the original problem. This programming paradigm is used frequently for computations on large data sets.

The intensive study of the parallelization of $\mathscr{DC}$ (Mou & Hudak, 1988; Mou, 1989; Cole, 1989) is a tiny but key portion of all work on $\mathscr{DC}$. Because of the wide applicability of the $\mathscr{DC}$ paradigm, it has often been formulated as a so-called *algorithmic skeleton* (Cole, 1989; Darlington *et al.*, 1993), 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* (Kindermann, 1994). 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 $\mathscr{DC}$ particularly attractive for a parallelization. That is, one major purpose of an architectural skeleton for $\mathscr{DC}$ 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 $\mathscr{DC}$ 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 $\mathscr{DC}$; 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. For example, our so-called 'call-balanced fixed-degree $\mathscr{DC}$' skeleton and all of its specializations can be compiled into such a skeleton.

Of course, the functional target skeleton is not meant to be executed in Haskell – this would be less efficient than executing the source skeleton. But it is our stepping stone to a loop program in a language which is closer to the machine architecture, like C. The loop program can be viewed as an abstract architectural skeleton and translated easily to different parallel machines. In the absence of resource constraints, it will provide the fastest possible execution given the data dependences imposed by $\mathscr{DC}$.

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 child of a node N represents the ith subproblem instance of the problem instance which N represents.

In our view, a 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 to add structural parameters. We consider skeleton `B` to be a specialization of skeleton `A` if `B` can be defined by an optional projection composed with an application of `A` with possibly modified arguments. Every function is a specialization of itself and of the identity function (`f = id f`).

Early work on transforming recursion with dependent calls into *sequential* loops was based on a depth-first traversal (Partsch & Pepper, 1976). This method is not very useful in a parallelization, where a breadth-first traversal is called for. The parallelization technique of Harrison and Khoshnevisan (1992) can be extended to handle $\mathscr{DC}$ if certain conditions are fulfilled (de Guzmán *et al.*, 1993). We start with similar considerations and develop a method for translating $\mathscr{DC}$ into a nested linear recursive skeleton, which can easily be interpreted as a loop nest. Our approach uses a tree structure as intermediate representation, because the tree 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.

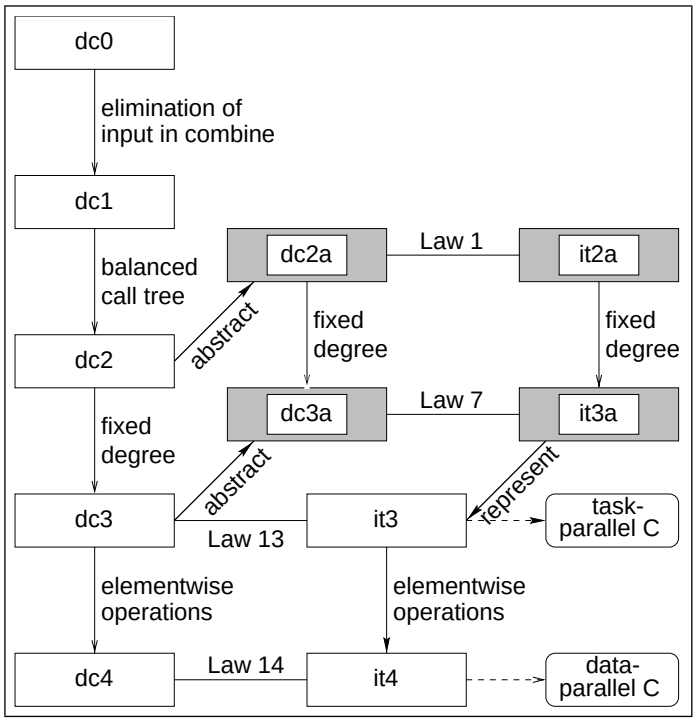

Fig. 1. Specializations of $\mathscr{DC}$.

Figure 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 section 2, the skeletons `dc0` to `dc3` are defined. In section 3, we transform the call-balanced fixed-degree $\mathscr{DC}$ 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 section 4, we specialize skeleton `dc3` further, with elementwise operations on balanced data partitions, and present a corresponding C loop program. In section 5, we demonstrate the use of the most specific skeleton `dc4` by discussing three examples: scan, Strassen's matrix multiplication, and Karatsuba's polynomial product. The last section summarizes our results and discusses related work.

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

2 Specializing $\mathscr{DC}$

In this section, we propose a sequence of specializations of a skeleton for general $\mathscr{DC}$. We denote our skeletons in the functional language Haskell (Hudak *et al.*, 1992) (see also `http://haskell.org/`). First, we present a general form of $\mathscr{DC}$, which is then specialized to enforce a balanced call tree, and subsequently further to enforce a fixed degree of the problem division.

At the outset, let us make some remarks about programming methodology. We see two kinds of people interacting with skeletons:

- the *application programmer* must choose for his application one from a library of algorithmic skeletons,
- the *systems programmer* must provide one or more architectural skeletons for each algorithmic skeleton in the library.

In providing specialized $\mathscr{DC}$ skeletons, we play part of the role of the systems programmer. Let us briefly discuss some issues relevant to the application programmer.

Consider the skeleton library consisting of `dc0` to `dc4`. To maximize efficiency, the application programmer should choose the most specialized skeleton which accommodates his/her algorithm. Frequently, the application will have to be 'adapted' to the skeleton. Such adaptations include the reformating of the input or output (like enlarging the length of a vector to the next power of 2) and the introduction of additional information to make the problem solution fit into the given skeletal structure. One must watch that the cost of these adaptations does not outweigh the savings gained by the chosen implementation.

Another important issue is granularity. In general, a compiler cannot determine the appropriate data size, at which to stop the parallelization and prescribe a sequential execution. The application programmer can help, e.g. in the case of $\mathscr{DC}$, by making this choice explicitly and supplying an optimized sequential algorithm to be used in the base case of the $\mathscr{DC}$ skeleton.

Before we continue with the description of our increasingly specialized skeletons `dc0` to `dc4`, figure 2 gives an impression of what kind of problems are covered by each of them. (Skeleton `dc4` is deferred to section 4.)

Auxiliary functions cited in the programs that follow can be found in the appendix. We use the following type synonyms, in which `Nat` denotes the natural numbers, `TDiv` the type of the divide function, and `TICom`/`TCom` the types of the combine function with/without using input data.

```
type Nat       = Int

type TDiv  a   = a -> [a]
type TCom  b   = [b] -> b
type TICom a b = (a,[b]) -> b
```

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

`dc0`	Quicksort (Blelloch, 1990; Horowitz & Sahni, 1984), Evaluation of expressions, Downsweep/Upsweep in a tree, Step in a simulation of a discrete system
`dc1`	Pseudomorphisms (Mou & Hudak, 1988), Reduce ∘ Map (Axford & Joy, 1993) on trees, Shannon Expansion (Norwood & McCluskey, 1996)
`dc2`	Radix Sort (Blelloch, 1990)
`dc3`	Convex Hull (Blelloch, 1990), Mergesort (Horowitz & Sahni, 1984), Reduce ∘ Map (Axford & Joy, 1993) on vectors, Maximum Segment Sum (Smith, 1987), Simple numerical integration
`dc4`	Vector scan, Bitonic Sort (Knuth, 1973), Fast Fourier Transform (Leighton, 1992), (Karatsuba's) polynomial product (Aho *et al.*, 1974), (Strassen's) matrix multiplication (Strassen, 1969)

Fig. 2. Applications of skeletons `dc0` to `dc4`.

2.1 Most general 𝒟𝒞 (`dc0`)

We assume that the subproblems can be solved independently. This puts, e.g. branch-and-bound algorithms outside our hierarchy, because of the existence of a global variable, the currently best value.

Under this assumption, the most general *𝒟𝒞* skeleton can be specified as follows:

```
dc0 :: (a->Bool) -> (a->b) -> TDiv a -> TICom a b -> a -> b
dc0 p b d c = r
    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. Take the example of a variant of the quicksort algorithm (Blelloch, 1990; Horowitz & Sahni, 1984):

```
quicksort :: Integral a => [a] -> [a]
quicksort = dc0 ((<2).length) id d c
```

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

```
 where pivot xs     = let l = length xs
                      in ((xs!!0)+(xs!!(l`div`2))+(xs!!(l-1)))`div`3
       fil f xs     = filter (`f` (pivot xs)) xs
       d xs         = [fil (<) xs, fil (>) xs]
       c (xs,[l,g]) = l ++ fil (==) xs ++ g
```

Note that the pivot is computed in the divide as well as in the combine function, but this does not justify a structural refinement of the skeleton. The reusage of values of common subexpressions is the responsibility of the compiler. The crucial point is that the information taken directly from the input data can be used in the combine function of `dc0`.

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 $\mathscr{DC}$ without combining the input data (`dc1`)

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

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

Skeleton `dc1` is well suited for Reduce ∘ Map algorithms (Axford & Joy, 1993) on trees:

```
data Tree a = Leaf a | Branch [Tree a]

tree_map_reduce :: (a->b) -> ([b]->b) -> Tree a -> b
tree_map_reduce mapfun redfun = dc1 p b d redfun
 where p (Leaf _)    = True
       p (Branch _)  = False
       b (Leaf x)    = mapfun x
       d (Branch xs) = xs
```

The following application computes the sum of all squares in a particular tree:

```
tree_map_reduce (\x->x*x) sum
                (Branch [Leaf 2,Branch [Leaf 3,Leaf 4,Leaf 5]])
```

which gives the result `54`.

`dc0` and `dc1` have the same expressive power (Herrmann & Lengauer, 1997); `dc0` is semantically equivalent to a specialization of `dc1`.

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

2.3 Call-balanced 𝒟𝒞 (`dc2`)

Two more specializations have to be applied in order to obtain 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 variant of the outer loop gives us the free schedule, in which each point of the scan is processed 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 𝒟𝒞 with a balanced call tree.

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

A good application for `dc2` is the radix sort (Blelloch, 1990). A set of lexicographically comparable lists of the same length is partitioned into blocks in which elements have the same most significant component. These blocks are ordered with respect to this component and the algorithm is applied recursively using the next component for comparison. The auxiliary function `which_digits` delivers a list of all digits which are relevant in the division for the actual subproblem instance.

```
radix_sort :: Ord a => [[a]] -> [[a]]
radix_sort xs = dc2 fst d flatten (length (xs!!0)) (xs,0)
 where d (x,l) = let digits = which_digits l [] x
                     parts  = [ filter ((==i).(!!l)) x | i<-digits ]
                 in zip parts (repeat (l+1))
```

`dc2` is an optimized version of a specialization of `dc1`. 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`.

```
dc2_by_dc1 :: (a->b) -> TDiv a -> TCom b -> Nat -> a -> b
dc2_by_dc1 b d c n x = dc1 p bb dd c (n,x)
   where p  (m,_) = m==0
         bb (_,y) = b y
         dd (m,y) = zip (repeat (m-1)) (d y)
```

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

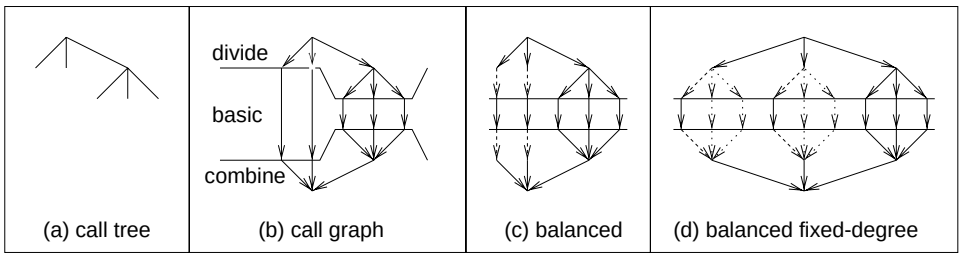

Fig. 3. Transformation steps of $\mathscr{DC}$ towards regularity.

If one allows `n` to be instantiated depending on `x`, terminating instances of skeleton `dc1` can be expressed in terms of `dc2` with a potential 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:

Case 1. We need a function `depth` which tells us the maximal recursion depth depending on `p`, `d` and `x`, to be given as parameter `n` to `dc2`. In (Harrison & Khoshnevisan, 1992), this is computed by a `while` loop, which we have to avoid since we want to apply a static space-time mapping. Determining the depth is the responsibility of the environment; often, it can be computed easily from the size of the input data. From a practical point of view, there is a simpler and more efficient option: the depth remains a structural parameter and is chosen later in dependence of the size of the processor topology, and function `b` is replaced by the sequential $\mathscr{DC}$ algorithm. This also avoids the creation of a lot of threads on a single processor.

Case 2. We have to extend the domain of the customizing functions `d` and `c` in order to establish the so-called overrun tolerance property (de Guzmán *et al.*, 1993), i.e. although the truth of predicate `p` indicates that 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 $\mathscr{DC}$ (`dc3`)

In skeleton `dc2`, the existence of a potential child 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.

Another non-static variant is 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 children of the nodes at the level above. This is similar to Blelloch (1989).

For an efficient static parallelization, it is convenient to have a fixed division

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

degree. 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 (Aho *et al.*, 1974) with a degree of 3 and Strassen's matrix multiplication (Strassen, 1969; Horowitz & Sahni, 1984) with a degree of 7.

Whereas, for a particular division degree, a *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 *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`.

```
dc3 :: Nat -> (a->b) -> TDiv a -> TCom b -> Nat -> a -> b
dc3 k b d = dc2 b (takeE k . d)
```

The definition of `dc3` uses `takeE` to impose a restriction on the class of problems: the customizing function `d` is treated as if it always produced a list of length `k`. If `d` is given by the user, the compiler must check that the restriction is satisfied. For purposes of equational reasoning, `takeE` is defined to access elements by index. For robustness and to make things easier for the user, the slightly more restrictive function `takeE'` could be used instead.

```
takeE :: Nat -> [a] -> [a]
takeE k xs = [ xs!!i | i<-[0..k-1] ]

takeE' :: Nat -> [a] -> [a]
takeE' k xs = if length xs == k
                then xs
                else error "length constraint violated"
```

`dc3` can be used for integration, e.g. consider the following algorithm based on Simpson's rule, which divides always into two parts:

```
integrate :: (Floating a, RealFrac a) => (a->a) -> a -> (a,a) -> a
integrate f eps (lower,upper) =
 (dc3 2 b d c n (lower,upper) + correct_border) * scale
 where correct_border  = (f upper - f lower) / 2
       scale           = (upper-lower) / (2^n * 3)
       b (lower,upper) = f lower + 2 * f ((lower+upper) / 2)
       d (lower,upper) = let med = (lower+upper)/2
                         in [(lower,med),(med,upper)]
       c [lval,rval]   = lval+rval
       n               = ceiling (logBase 2 ((upper-lower)/eps))
```

The integration works as follows. The range is split into 2^n intervals, for the minimal `n` which makes each interval not longer than `eps`. Then, each interval is integrated using Simpson's rule and the results are combined by addition.

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

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

Unfortunately, some algorithms do not guarantee that the call tree is balanced. Furthermore, a lot of algorithms lead to a balanced tree only 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 section 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 figure 3(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 only down the leftmost path of the extended call tree until `n=0` (see figure 3(c–d)), the other paths carry useless information.

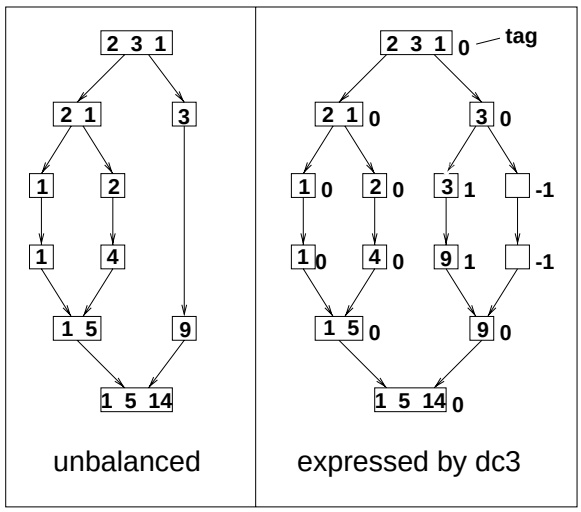

Fig. 4. Adaptation to `dc3`.

Similarly, the corresponding combine functions just deliver the value from the leftmost child. 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 figure 4, which describes the process of an artificial $\mathscr{DC}$ 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 before.

```
dc3tobal :: Int->(a->Bool)->(a->b)->TDiv a->TCom b->Int->a->b
dc3tobal k divisible bb dd cc n x = fst (dc3 k b d c n (x,0))
   where b (x,l) = (bb x,l)
```

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

```
d (x,l) = if divisible x
             then takeE k (zip (dd x) (repeat 0))
             else (x,l+1) : take (k-1) (repeat (x,-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)
```

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

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

In section 3.2, we transform `dc2` to linear recursion. We state in Law 1 that the abstract version `dc2a` (see figure 1) of the call-balanced skeleton `dc2` is equivalent to the linearly recursive skeleton `it2a` which enumerates the levels of the call graph. In section 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 section 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 section 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 'power structure') to represent a list structure by a tree with empty inner nodes. A single element is defined with the constructor `Sgt`, a list of power structures is made a power structure 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`.

```
unSgt :: PS a -> a
unSgt (Sgt a)  = a

unCom :: PS a -> [PS a]
unCom (Com as) = as
```

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

```
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

dkmap :: Nat -> (PS a->PS b) -> Nat -> PS a -> PS b
dkmap k f 0       = f
dkmap k f n | n>0 = Com . takeE k . map (dkmap k f (n-1)) . unCom

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

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]]
```

`Sgt`, `unSgt`, `Com` and `unCom` are wrappings resp. unwrappings of data type `PS`. `dmap f n` applies a function to the `n`-th level of a power structure. `dkmap k f n` does the same as `dmap f n`, but its domain is restricted to those power structures in which all nodes at the first `n` levels have a branching degree of `k`. `comp` takes a list of functions and composes them. The functions `down` and `up` take a function and a number `n` and compose this function `n` times with itself, while counting the first argument 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 inverse of `partition k n`.

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.

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

Here is the new Haskell definition for dc2, which works on power structures 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) -> TDiv a -> TCom 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
```

Now, we define 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
```

We observe that it2a equals dc2a, and state this in Law 1. The proof uses Laws 2–6, whose proofs we omit here.

Law 1

```
dc2a = it2a
```

Proof

By induction on n

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
```

n → 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 }
```

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

```
  (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 5, 4, 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 2 and 3 }
  (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

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

Law 3

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

Law 4

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

Law 5

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

Law 6

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

Laws 2 and 3 address the addition of a further iteration to a sequence of iterations. Law 4 states how nested maps can be replaced by a single map. Laws 5 and 6 extend Law 4 to the case, in which an iterator `up` or `down` iterates on the inner map. Analogously, in loop parallelization, this describes the process of bringing an inner sequential loop to the top level.

3.3 The abstract skeletons for `dc3`

Skeleton `dc3a` is the abstract version of `dc3` (see figure 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 = dc2a b (Com . takeE k . unCom . d)
```

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

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

Law 7

```
   dc3a = it3a
```

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

In the previous subsection, 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 parent nodes and the leaf nodes are deleted. In order to express this application easily by a single linear recursion, the nodes at said level have to be represented by a one-dimensional data structure; we use a list. The representation mapping is given in Definition 8. The structural information that is contained in the tree is used to derive functions which manipulate the linear structure.

Figure 5 illustrates how this is done. We have used the local definitions `dd`, `bb`, and `cc` from `dc2_by_dc2a` on the abstract side. The abstract level (the level which makes use of the tree or power structure) is depicted on the left side, the concrete level (performing the corresponding computations on the linear structure) on the right side. At the abstract level, we can exploit Laws 1 and 7 because they contain 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 obtain a linear representation of the nodes at level `n` of the balanced `k`-degree call tree. To be able to exploit the property of fixed degree, we use function `dkmap k` instead of `dmap`. The representation is defined below by function `represent k n`, and is depicted in figure 5.

Definition 8
(Linearization)
We call the mapping of a level of a `k`-degree tree to a list a *linearization.*

We define 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` which is the inverse of `represent k n` on lists of length k^n.

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

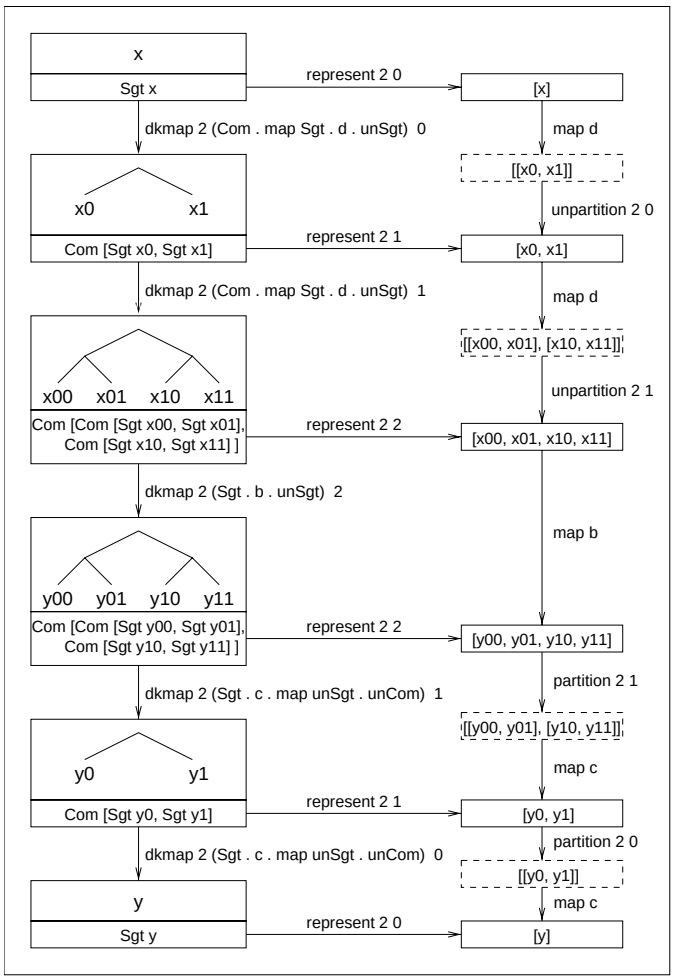

Fig. 5. Effect of linearization.

```
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
```

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

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

Law 9
(Concretization of `dkmap`)

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

Let us derive the concrete from the abstract implementation. Note that figure 5 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. This equation contains free variables. Its solution consists of bindings of the free variables to expressions.

```
represent k n . dkmap k f m = g . represent k p
```

(`dkmap k 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 . dkmap k f m . abstract k p
```

We can replace each occurrence of the abstract function (`dkmap k 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. A proof of Law 10 can be found in (Herrmann & Lengauer, 1997).

Law 10
(Concretization of `dkmap k dd n`)

```
   dkmap k (Com . map Sgt . d . unSgt) n
=  abstract k (n+1) . unpartition k n . map d . represent k n
```

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

Law 11
(Concretization of `dkmap k bb n`)

```
   dkmap k (Sgt . b . unSgt) n
=  abstract k n . map b . represent k n
```

Law 12
(Concretization of `dkmap k cc n`)

```
   dkmap k (Sgt . c . map unSgt . unCom) n
=  abstract k n . map c . partition k n . represent k (n+1)
```

3.4 Transformations towards the functional target skeleton *it3*

Let us give an overview of the transformation process. The balanced fixed-degree $\mathscr{DC}$ skeleton in its recursive form `dc3` can be expressed in terms of `dc2a`, but with the additional fixed-degree constraint on `d`. We use Law 1, which states the equivalence of recursion and iteration on the abstract side. That is, `dc3` can be expressed in terms of `it2a`. Then, we replace every function on the abstract side by its linear representation, obtaining function `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
```

The conversions `represent k n` and `abstract k n` in `it3_1` are inverses of each other and can be eliminated; see figure 5. The transformation continues by replacing functions `map`, `partition`, and `unpartition` with list comprehensions, which will lead to parallel loops in C. There are still linear recursions left (in `down` and `up`) which perform iterations through the levels of the call tree. These are also transformed into list comprehensions, which will lead to sequential loops in C. Intermediate results of these recursions are stored in lists `g` and `h` in skeleton `it3`, in order to make the correspondence to a single-assignment C program obvious.

```
it3 :: Nat -> (a->b) -> (a->[a]) -> ([b]->b) -> Nat -> a -> b
it3 k b d c n x =
  let a0 = (:[]) x
      h  = a0:[ [d (h !! (m-1) !! (l`div`k)) !! (l`mod`k)
```

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

```
                | l <- [0..k^m-1]]
              | m<-[1..n]]
    a1 = h !! n
    a2 = [b (a1!!i) | i<-[0..k^n-1]]
    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]]
    a3 = g !! n
    a4 = head a3
in a4
```

Law 13

```
dc3 = it3
```

Proof

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. □

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 equivalence.

3.5.1 Correspondences between Haskell and C

Let us present the main rules we used in our transformations. The rules do not form a complete translation system from Haskell to C, e.g. the laziness of Haskell is not preserved. Giving up laziness is necessary for a static space-time mapping.

Before applying these rules, we must use a sophisticated algorithm (e.g. (Bell *et al.*, 1997)) to eliminate higher-order functions for which no skeleton implementation exists, and polymorphism.

1. `(Int,+,-,*,‘div‘,‘mod‘,^)` in Haskell corresponds with `(int,+,-,*,/,%, pow(.,.))` in C. ‘`mod`‘ and `%` correspond only on the natural numbers. Function `pow(b,m)` in C returns `b` to the power of `m`.
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 `let` expression without recursive equations, and with the equations sorted in the direction of the data dependences, is transformed to a sequence of C statements.

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

4. Lists in Haskell are represented in C as arrays. This correspondence is sound with respect to the static 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 `(:[])` function to `a[0]` has to be `a` and the one of `(!!i)` to `a` has to be `a[i]`.
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 The C code

Applying the correspondences from the previous subsection to `it3`, we obtain the C program below. `seqfor` defines a loop whose iterations are executed in sequence. `parfor` defines a loop whose iterations can be processed in parallel. If programmed correctly, the semantics of both `seqfor` and `parfor` is defined by the ordinary (sequential) `for` loop. This is expressed by the C definitions:

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

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. 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 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++) {
```

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

```
      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 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 (Lengauer, 1993), it is sufficient to keep memory space for at most 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 $\mathscr{DC}$ skeleton in two steps:

1. First, we impose a balance on the data division. This means that the data is split into a fixed number of partitions of equal size. Partition i is assigned to the part of the topology which handles the i-th subproblem instance. This is important for problems in which the input or output data do not fit into the memory of a single processor, because it enables the data distribution.
2. Then we impose elementwise operations on the zip of the partitions. As a consequence, 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, which accelerates the computation of the divide and combine functions by additional massive parallelism.

Details of the derivation can be found in Herrmann and Lengauer (1997). The rest of this section covers only the most interesting aspects.

4.1 Elementwise operations on balanced data partitions

Skeleton `dc4` restricts the divide and combine function to elementwise operations. We omit the definition of `dc4` (Herrmann & Lengauer, 1997), because it uses a lot of auxiliary functions which describe properties like balanced data division and elementwise operations. The type of `dc4` differs slightly from that of `dc3`.

```
dc4 :: Nat              -- degree of problem division
    -> (a->b)           -- basic function
    -> ([a]->[a])       -- divide function
    -> ([b]->[b])       -- combine function
    -> Nat              -- recursion depth
    -> [a]              -- input data
    -> [b]              -- output data
```

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

Here, the input and output data are lists. Divide function `d` and combine function `c` are supposed to take a list of length `k` as input and output. The input elements of `d` and the output elements of `c` correspond to the list elements with the same index in different partitions, the output elements of `d` and the input elements of `c` correspond to the list elements with the same index in different subproblems resp. subproblem solutions. Not all list elements have to carry useful data. Dummy places originate from empty partitions in the input and output data distribution.

4.2 *it4: The functional target for dc4*

Before we present `it4`, we have to introduce two more 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`.

```
it4 :: Nat->(a->b)->([a]->[a])->([b]->[b])->Nat->[a]->[b]
it4 k b d c n x =
  let a0 = x
      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]]
      a1 = h !! n
      a2 = [b (a1!!q) | q<-[0..k^n-1]]
      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]]
      a3 = 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. □

4.3 *Transformation to C*

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

Due to the outer sequential loops and the inner parallel loops, this program is data-parallel. Thus, 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. Instead of `d`, `b`,

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

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 sequential loops enforce a global synchronization, so the first (sequential) indices of the arrays need 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]);
```

5 Examples

5.1 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

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

with elements, say, b_0 to b_{m-1}, where $b_0 = a_0$ and $\forall\, i : 0 < i < m : b_i = b_{i-1}$ `` `op` `` a_i. The scan function is useful in many parallel algorithms, especially sorting algorithms (Blelloch, 1989). In (Carpentieri & Mou, 1991; Gorlatch, 1996a), a parallel algorithm for scan is presented which fits into skeleton `dc4` after applying a method called 'broadcast dissolving' (Mou, 1989), later renamed to 'broadcast elimination'. `ilog2` computes the integer logarithm in base 2.

```
scan :: (a->a->a) -> [a] -> [a]
scan op xs = map fst (dc4 2 b id c n xs)
   where n       = ilog2 (length xs)
         b x     = (x,x)
         c [(x,sx),(y,sy)] = let s = sx`op`sy
                             in [(x,s),(sx`op`y,s)]
```

The divide function behaves like the identity, i.e. the i-th partition becomes part of the i-th subproblem. 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 on an element of some partition also replicates the operation on the last element of that partition. This implements 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.

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

5.2 Strassen's matrix multiplication

Strassen's matrix multiplication (Strassen, 1969), a $\mathscr{DC}$ algorithm with a division degree of 7, 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 a communication overhead on some machines.

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 two matrices `xss` and `yss` of size $2^{\mathtt{n}} \times 2^{\mathtt{n}}$, and returns the product of `xss` and `yss`. Figure 6 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`.

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

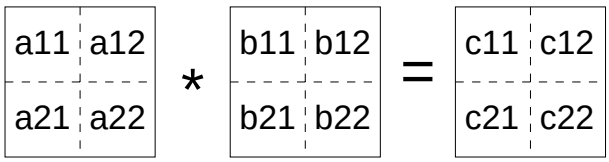

Fig. 6. Matrix partitions.

```
strassen :: Num a => [[a]] -> [[a]] -> [[a]]
strassen xss yss =
 let n = ilog2 (length xss)
 in (  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,c12,c21,c22) = (m1+m4-m5+m7,m3+m5,m2+m4,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 xss yss`.
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. rearrangements at run time should be avoided.

To express Strassen's algorithm with the parallel C skeleton presented in section 4.3, one has to compile the customizing functions from Haskell to C which can be done easily because they contain only list, tuple, and arithmetic operations.

5.3 Karatsuba's polynomial product

This subsection contains material that we have published before with respect to a slighly modified *𝒟𝒞* skeleton (Herrmann & Lengauer, 1996).

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

In 1962, Karatsuba published a $\mathscr{DC}$ algorithm for the multiplication of large integers of bitsize N with cost $O(N^{\log_2 3})$ ($\log_2 3 \approx 1.58$) based on ternary $\mathscr{DC}$ (Aho *et al.*, 1974). A trivial algorithm has complexity $O(N^2)$. As an example of ternary $\mathscr{DC}$ we choose the polynomial product, which is the part of Karatsuba's algorithm that is responsible for its complexity.

Here, we concentrate on the product of two polynomials which are represented by powerlists (Misra, 1994) of their coefficients in order. The length of both lists is the smallest power of two which is greater than the maximum of both degrees. 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 at most $N < 2^{n-1}$, 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 Karatsuba 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` $[a_0,...,a_{2*N-1}]$ $[b_0,...,b_{2*N-1}] = [(a_0,b_0),...,(a_{2*N-1},b_{2*N-1})]$) 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 `karatsuba`, which multiplies two polynomials represented by equal-size powerlists:

```
karatsuba :: Num a => [a] -> [a] -> [a]
karatsuba x y =
 let n = ilog2 (length x)
 in ((\x-> fst x ++ snd x) . unzip . project 2 3 n .
    dc4 3 b d c n .
    embed 2 3 n (0,0)) (zip x y)
 where b  (x,y)
        = (0,x*y)
       d  [(xh,yh),(xl,yl),_]
        = [(xh,yh),(xl,yl),(xh+xl,yh+yl)]
       c  [(hh,hl),(lh,ll),(mh,ml)]
        = [(hh,lh+ml-hl-ll),(hl+mh-hh-lh,ll),(0,0)]
```

Of the constituting functions, `b` multiplies two constant polynomials. Function `d` 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 `c` 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

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

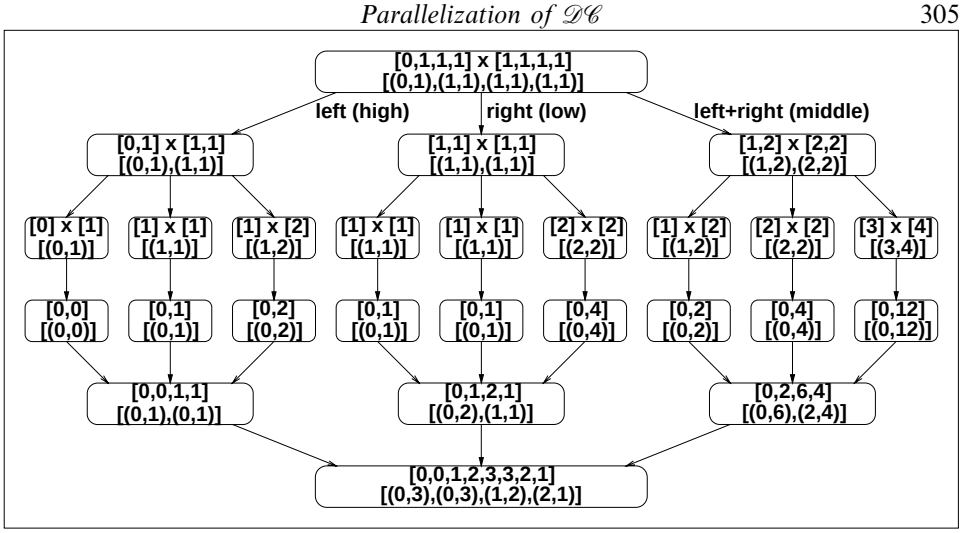

Fig. 7. Call graph for a call of `karatsuba`.

the result of the middle part. As an example, figure 7 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.

6 Results, related work, and further research

Starting with a general specification of $\mathscr{DC}$, we have obtained, through a series of systematic stepwise refinements of the skeleton, a data-parallel nested loop program for a class of $\mathscr{DC}$ algorithms. From `dc2` on, each specialized skeleton can be implemented by a parallel loop program, representing a different class of $\mathscr{DC}$ problems. We presented implementations for `dc3` and `dc4` that can be mapped to space and time at compile-time.

Intentionally, most of our derivation has been conducted on the formal territory of Haskell. For central steps we have developed equational proofs.

Huang *et al.* (1992) have presented a derivation of a parallel implementation of Strassen's matrix multiplication algorithm using tensor product formulas. The result is a loop program with a nesting depth of 4 (if loops of constant extent are not considered), but they also perform the initial and final permutations and ensure that dummy points are not scanned.

We took Strassen's matrix multiplication as a motivating example and, contrary to Huang *et al.* (1992), obtained a loop program with a nesting depth of 2 (the outer in time and the inner in space). Both loop programs are similar but ours does not exploit that the data division is only 4 but not 7 like the problem division. Therefore, our program includes unnecessary operations, but they do not add to the execution time (using the number of processors needed for implementing the free schedule).

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

Additionally, we assume a special distribution of the input and output data instead of the row- or column-major order which is often presumed in matrix algorithms.

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. Adding a static type to all functional equalities enforces that only those implementations are derived which are well defined on the specified domain. Some algebraic equations hold, e.g. for the type `Integer` but not for the type `Float`, due to numerical approximation.

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 Huang *et al.* (1992), there is other work related to ours. Misra (1994) and Achatz and Schulte (1996) restrict themselves to a binary division of data and problems. The approach of Mou and Hudak (1988) and Mou (1989, 1990) allows an arbitrary division of problems and a division of multi-dimensional data into two parts per dimension. Cole (1989) 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* (Bird, 1988), a basic $\mathscr{DC}$ skeleton somewhat more restrictive than ours. There exists a theory for the transformational parallelization of homomorphisms (Skillicorn, 1994; Gorlatch, 1996b). The class of *distributable homomorphisms* (*DH*) (Gorlatch, 1996a) corresponds to the combine phase of our skeleton `dc4` with a binary divide function. For all functions of the DH class, a common hypercube implementation can be derived by transformation in the Bird-Meertens formalism (Gorlatch, 1996a).

The class of 'static $\mathscr{DC}$' (Gorlatch & Bischof, 1997) 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 Law 1 is their Theorem 2. The result of Gorlatch and Bischof (1997) is an asynchronous, SPMD program as opposed to our synchronous nested loop program.

In our own previous work (Herrmann & Lengauer, 1996), 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.

In our future work we want to make the results available for practical use. It is necessary to have a language for the user to define customizing functions and to compose skeletons. As language, we chose a subset of Haskell, that is processed in an eager fashion. At the moment, we are working on a compiler that translates this

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

subset into C with MPI. The parallel construct is the list and list comprehensions are compiled into potentially parallel loops.

Acknowledgements

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

```
flatten :: [[a]] -> [a]
flatten = foldl (++) []

which_digits :: (Ord b, Integral a) => a -> [b] -> [[b]] -> [b]
which_digits l ds [] = ds
which_digits l ds (x:xs) = which_digits l (insert (x!!l) ds) xs

insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) =
 if x>y then y : insert x ys
        else if x==y then y:ys
                     else x:y:ys

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

ilog2 :: Int -> Int
ilog2 x = let powers = [ (i,2^i) | i<-[0..] ]
              v = fst (head (filter ((>= x) . snd) powers))
          in if 2^v ==x then v
                        else error "ilog applied to non-power of two"

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]
```

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

```
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

Achatz, K. and Schulte, W. (1996) Massive parallelization of divide-and-conquer algorithms over powerlists. *Science of Computer Programming*, **26**(1–3), 59–78.

https://doi.org/10.1017/S0956796899003287 Published online by Cambridge University Press

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

Axford, T. and Joy, M. (1993) List processing primitives for parallel computation. *Computer Languages*, **19**(1), 1–17.

Bell, J. M., Bellegarde, F. and Hook, J. (1997) Type-driven defunctionalization. *ACM SIGPLAN Notices*, **32**(8), 25–37. *Proc. ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'97)*.

Bird, R. S. (1988) Lectures on constructive functional programming. *Pages 151–216 of:* Broy, M. (ed), *Constructive Methods in Computing Science*. NATO ASI Series F: Computer and Systems Sciences, Vol. 55. Springer-Verlag.

Blelloch, G. E. (1989) Scans as primitive parallel operations. *IEEE Trans. on Computers*, **38**(11), 1526–1538.

Blelloch, G. E. (1990) *Vector Models for Data-Parallel Computing*. MIT Press.

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

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

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

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

Gorlatch, S. (1996a) Systematic efficient parallelization of scan and other list homomorphisms. *Pages 401–408 of:* Bougé, L., Fraigniaud, P., Mignotte, A., and Robert, Y. (eds), *Parallel Processing. Euro-Par'96, Vol. II*. Lecture Notes in Computer Science 1124. Springer-Verlag.

Gorlatch, S. (1996b) Systematic extraction and implementation of divide-and-conquer parallelism. *Pages 274–288 of:* Kuchen, H., and Swierstra, D. (eds), *Programming Languages: Implementation, Logics and Programs (PLILP'96)*. Lecture Notes in Computer Science 1140. Springer-Verlag.

Gorlatch, S. and Bischof, H. (1997) Formal derivation of divide-and-conquer programs: A case study in the multidimensional FFT's. *Pages 80–94 of:* Mery, D. (ed), *Proc. 2nd Int. Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'97)*. IEEE Computer Society Press.

Harrison, P. G. and Khoshnevisan, H. (1992) A new approach to recursion removal. *Theoretical Computer Science*, **93**, 91–113.

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

Herrmann, C. A. and Lengauer, C. (1997) *Parallelization of Divide-and-Conquer by Translation to Nested Loops. Technical report* MIP-9705. Fakultät für Mathematik und Informatik, Universität Passau.

Horowitz, E. and Sahni, S. (1984) *Fundamentals of Computer Algorithms*. Computer Software Engineering Series. Computer Science Press.

Huang, C.-H., Johnson, J. R. and Johnson, R. W. (1992) Generating parallel programs from tensor product formulas: A case study of Strassens's matrix multiplication algorithm. *Pages 104–108 of:* Stout, Q. F. (ed), *Proc. Int. Conf. on Parallel Processing, Vol. III: Algorithms & Applications*. CRC Press.

Hudak, P., Peyton Jones, S. and Wadler, P. (eds), (1992) Report on the programming language Haskell (Version 1.2). *ACM SIGPLAN Notices*, **27**(5).

Kindermann, S. (1994) Flexible program and architecture specification for massively parallel systems. *Pages 160–171 of:* Buchberger, B., and Volkert, J. (eds), *Parallel Processing: CONPAR 94 – VAPP VI.* Lecture Notes in Computer Science 854. Springer-Verlag.

Knuth, D. E. (1973) *The Art of Computer Programming, Vol. 3: Searching and Sorting.* Addison-Wesley.

Leighton, F. T. (1992) *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann.

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

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

Mou, Z. G. (1989) *A Formal Model for Divide-and-Conquer and Its Parallel Realization.* Ph.D. thesis, Department of Computer Science, Yale University. Technical report YALEU/DCS/RR-795.

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

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

Norwood, R. B. and McCluskey, E. J. (1996) Synthesis-for-scan and scan chain ordering. *Pages 87–92 of: Proc. 14th IEEE VLSI Test Symp.* IEEE Press.

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

Skillicorn, D. B. (1994) *Foundations of Parallel Programming.* Cambridge International Series on Parallel Computation. Cambridge University Press.

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

Strassen, V. (1969) Gaussian elimination is not optimal. *Numerische Mathematik*, **13**, 354–356.