Transformation of Divide & Conquer to Nested Parallel 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/~lengauer

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 describes 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 $\mathcal{DC}$ it has often been formulated as an *algorithmic skeleton* [4,5], 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* [13]. 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 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 paper is to specialize an algorithmic skeleton for $\mathcal{DC}$ to a form for which there is an efficient parallel implementation. Each specialization step imposes a new restriction to that end, 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 are also possible. Some specialized skeletons (we call them *sources*) can be transformed into *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.

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.

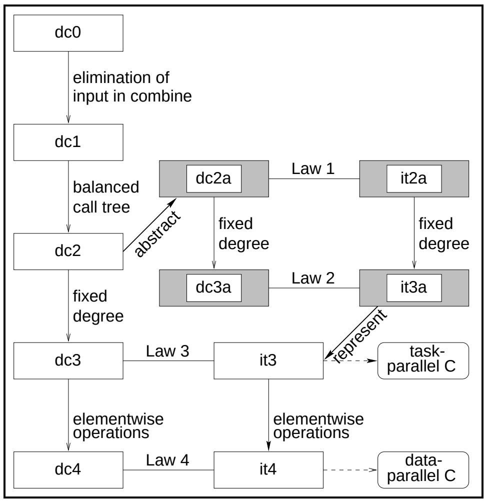

Fig. 1. Specializations of $\mathcal{DC}$

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, into 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. Implementational and proof details omitted due to lack of space can be found in [10].

2 Specializing $\mathcal{DC}$

In this section, we propose a sequence of specializations of a skeleton for general divide-and-conquer. We write our skeletons in the language Haskell [19].

First, we present a form of $\mathcal{DC}$, which we think of as most general. Essentially, the only restriction we make at this point is that subproblems be solvable independently. This property seems to us at the heart of the notion of $\mathcal{DC}$.

This most general form 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 composite data structure, we have chosen the list arbitrarily; other choices can be envisioned. The skeleton is parametrized by four *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.

Our justification of presenting `dc0` is that it is the only one in our sequence of skeletons in which important $\mathcal{DC}$ algorithms like the quicksort are expressed naturally. In the quicksort, the input data is used by the combine function.

```
quicksort :: Ord a => [a]->[a]
quicksort xs = dc0 p b d c xs
   where p xs             = length xs < 2
         b xs             = xs
         d (x:xs)         = [filter (<x) xs, filter (>x) xs]
         c ((x:xs),[l,g]) = l ++ x:(filter (==x) xs) ++ g
```

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

Our first specialization is to eliminate the use of the input data `x` in the combine function and achieve a more regular form `dc1`. This skeleton has the same expressive power as `dc0` [10].

```
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
```

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

Whether skeleton `dc1` will terminate cannot be decided before run time. As a consequence, a schedule cannot be determined at compile time. Thus, we propose a skeleton `dc2`, which is `dc1` with the additional property of *call balance*, which is that all leaves are located at the same level of the call tree, say level `n`. The effect of balance is discussed in Sect. 3.5. We can choose parameter `n` in dependence of the number of available processors. We have to adapt the customizing functions a bit [10]. The basic function has to be replaced by the application of `dc1`. The divide and combine function have to be extended to handle the case in which the basic case is reached after fewer than `n` recursive calls.

```
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
```

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.

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 [18,11] 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 [16], 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]]
```

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

In this section, we show how the recursive call-balanced fixed-degree $\mathcal{DC}$ skeleton (`dc3`) can be transformed into an intermediate iterative program, which can be implemented easily on many parallel systems.

In Subsect. 3.2, we state as Law 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 prove the equivalence of `dc3a` and `it3a`, stated as Law 2. Subsect. 3.4 contains the functional target skeleton `it3` whose equivalence with `dc3` is stated in Law 3. In Subsect. 3.6, we transform `it3` into 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

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` (for "powerstructure"), which defines a superset of what we intend to define. 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 `n`th level of a powerstructure. `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 number down resp. up. The function `partition` takes parameters `k` and `n`, and maps a list of length $\mathtt{k}^{\mathtt{n+1}}$ bijectively to $\mathtt{k}^{\mathtt{n}}$ lists of length `k` taking successive elements. `unpartition k n` is the inverse of `partition k n`. `singleton` creates a singleton list.

3.2 Transforming `dc2` into linear recursion

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` equals `dc2a`, and state this as Law 1.

```
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
```

Imagine function `up` to build *up* the call tree, and function `down` to tear it *down*.

LAW 1 `dc2a = it2a`

3.3 The abstract version of `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 2 `dc3a = it3a`

3.4 The functional target skeleton `it3`

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.

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 structural information that is contained in the tree is used to derive functions which manipulate the linear structure. Fig. 2 shows on the left the application on the abstract side (using trees) and on the right the concrete side (using lists) obtained from the left by the representation mapping. We obtain the functional target skeleton, `it3`, after the following sequence of transformations: we express `dc3` in terms of `dc2a`, exploit the equivalence of recursion and iteration on the abstract side, i.e., express `dc3` in terms of `it2a`, and replace all abstract operations by their representation.

```
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 3 `dc3 = it3`

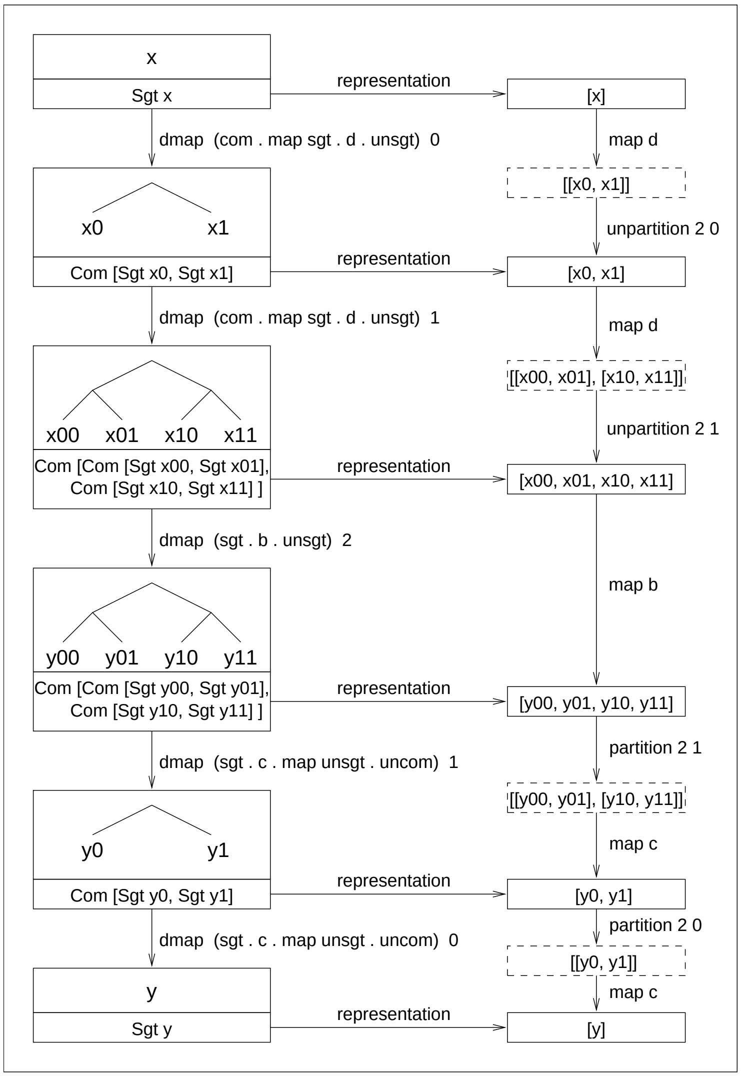

Fig. 2. Effect of the linearization

3.5 Implementational issues introduced by `dc2`

With `dc2`, we have obtained a schedule for the parallel implementation:

`for` `t` = 0 to `n-1`
    perform divide operations at level `t` in parallel ;
solve the subproblems at level `n` in parallel ;
`for` `t` = `n+1` to `2*n`
    perform combine operations at level `2*n-t` in parallel

Algorithms on sparse data structures (e.g., the product of sparse matrices), which are represented by trees of logarithmic depth, can be given a schedule at compile time by expressing them in `dc2`, whereas the allocation remains an issue for run time.

3.6 Transformation into C

Let us present the main rules we used in our transformation from Haskell into C.

We do not provide a formal proof of their semantic correctness; this would require a formal semantics for both Haskell and C. The correspondences of data structures resp. control structures between Haskell and C should be obvious.

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.

These rules do not form a complete translation system from Haskell to C:

1. The laziness of Haskell is not preserved.
2. Partial application is not possible in the C programs. This does not hurt here because we have only one run-time argument.
3. At present, nested skeleton definitions have to be expanded in the transformation by hand.
4. Higher-order functions not eliminated in the transformation are not handled, since there is no interface for passing functions between Haskell and C.
5. Polymorphic types have to be instantiated at compile time.

Our transformation scheme is sufficient for our purposes, since our foremost goal is to derive efficient parallel schedules at compile time.

4 Further specializations and their implementation

The specializations made so far consider only parallelism introduced by solving independent subproblems. To exploit parallelism in the application of the divide and combine function, the skeleton has to be specialized further for data-parallel operations.

4.1 Specialization with balanced data division and elementwise operations

First, we impose a balance on the data division, in order to achieve a balanced load of distributed data and parallel operations.

Balancing the data division means splitting the data into a fixed number of partitions of the same size. This is always possible by filling partitions with empty elements. Each partition is assigned to the part of the topology which handles the according problem instance. 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 partition makes up the input of a separate subproblem.

The specialization with a balanced data division does not lead to 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. Because the partitions all have the same size, due to the preceding specialization, zipping is always possible without additional restrictions. As a result, each partition contains the zip of its own data with the data of the other partitions. Then, all partitions contain the same data, but the function they apply to it is different. The advantage is that communications become much more regular and the customizing functions can be viewed as vectorized operations.

Other, more liberal specializations could be considered as well but are not discussed here. For a particular implementation of the scan function, it could be useful to broadcast an element of one data partition to all elements of another data partition. Even more flexibility could be achieved, if the communication operations were defined in a small language, which is not too powerful in order to assure a space-time mapping at compile time.

The definition of the skeleton with elementwise operations, which we call `dc4`, is given in terms of the skeleton with a balanced data division, which is itself expressed in terms of `dc3`:

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

We omit the body due to lack of space; see [10]. 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.

4.2 The intermediate C code for `dc4`

Skeleton `it4` [10], the functional target skeleton of the transformation of `dc4`, can be translated to the C code below using the rules we proposed in Subsect. 3.6.

LAW 4 `dc4 = it4`

In the process of equational reasoning, nested list comprehensions appear. These correspond to nested parallel loops at the imperative side. The number of iterations of the two inner-most loops differ dependent on the context, but their product does not. Therefore, we flatten the two nested comprehensions to obtain a single parallel loop. The program we obtain is data-parallel: the outer iterations are sequential, the inner ones parallel. Therefore, the program 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`, and `c`, we use `divide`, `basic`, and `combine`, respectively. These functions expect an additional parameter that indicates which element of the result list is desired. The function `pow(b,m)` determines `b` to the power of `m`. The input data after algorithm-specific adaptations is located in `h[0]`, the output data before adaptations is delivered in `g[old(m)]`. `digpos(k,d,v)` delivers the `d`th least significant digit of the natural number `v` denoted in radix `k`, and `digchange(k,d,v,i)` replaces this digit by `i`. The innermost `for` loops collect only a constant number of pointers and can be resolved at compile time.

```
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));  }

parfor(q=0;q<pow(k,n);q++)
   basic(g[0][q],h[old(m)][q]);

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)); }
```

5 Strassen's matrix multiplication

Strassen's matrix multiplication [18] 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. In principle, one can achieve a faster execution time with the same number of processors using Strassen's algorithm in place of the conventional one. Multiprocessors often come in sizes of powers of 2, but Strassen's algorithm is expressed for virtual machines with sizes of powers of 7. This requires a mapping from virtual space to a cartesian product of real space and real time, which motivates the need for advanced space-time mapping techniques.

$$\begin{array}{|c:c|}\hline \mathrm{a11} & \mathrm{a12} \\ \hdashline \mathrm{a21} & \mathrm{a22} \\ \hline \end{array} \; * \; \begin{array}{|c:c|}\hline \mathrm{b11} & \mathrm{b12} \\ \hdashline \mathrm{b21} & \mathrm{b22} \\ \hline \end{array} \; = \; \begin{array}{|c:c|}\hline \mathrm{c11} & \mathrm{c12} \\ \hdashline \mathrm{c21} & \mathrm{c22} \\ \hline \end{array}$$

Fig. 3. 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. 3 shows one step of the recursive decomposition of the matrices. How the `c`s are computed from the `a`s and `b`s 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 skeleton `dc4` and implemented using the C code presented in Subsect. 4.2. If one assumes matrices stored in row major order, as we do, some permutations have to be applied before and after. To express Strassen's algorithm with the parallel C skeleton presented in Subsect. 4.2, 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 a sequence of skeletons, corresponding to different classes of $\mathcal{DC}$.

Quicksort fits well into skeleton `dc0`, because it uses the input data in combining. Problems like adaptive integration, evaluation of expressions, and satisfiability of boolean expressions can be solved by `dc1`, because only the results of the subproblem solutions are needed. `dc2` is ideal for $\mathcal{DC}$ operations on sparse structures represented by trees. The class of `dc3` covers algorithms by which information can be computed from a dense structure in a balanced way, e.g., reductions, list homomorphisms, the maximum segment sum problem, the convex hull problem, etc. The benefit of `dc4` is that it enforces a balanced division of output data as well, as is important for the scan algorithm, the fast Fourier transform, bitonic sort, Karatsuba's polynomial product, and Strassen's matrix multiplication.

For the less specialized skeletons, efficient implementations will need to exploit load information at run time whereas, for the more specialized skeletons, a space-time mapping determined at compile time is better.

Huang et al. [12] 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.

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 [12], there is other work related to ours. Misra [14], and Achatz and Schulte [1] restrict themselves to a binary division of data and problems. Mou's [15] approach allows an arbitrary division of problems and a division of multi-dimensional data into two parts per dimension, but does not say anything about a higher division degree. Cole [4] does not consider input and output data which are distributed among the processors. 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 input and output data, aside from ours.

There has also been related work in our own group. First, there is work on the parallelization of the *homomorphism* [3], a basic $\mathcal{DC}$ skeleton somewhat more restrictive than ours. There exists a theory for the transformational parallelization of homomorphisms [17,7]. The class of *distributable homomorphisms* (*DH*) [6] corresponds to the combine phase of our skeleton `dc4` with a binary divide function (this class is called C-algorithms in [8]). For all functions of the DH class, a common hypercube implementation can be derived by transformation in the Bird-Meertens formalism [6].

The class of "static $\mathcal{DC}$" [8] 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 in [8] is Theorem 2.

In our own earlier work [9], we have obtained loop programs similar to the one presented here by parallelization in a space-time mapping model related to the hypercube. There, we have been informal; here we have focussed on a more precise, top-down development in the framework of equational reasoning.

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. Two PLILP reviews contained very valuable feedback on the presentation. Financial support was provided by the DFG through project RecuR2 and by the DAAD through an ARC exchange grant.

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

5. 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.
6. S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.
7. 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.
8. S. Gorlatch and H. Bischof. Formal derivation of divide-and-conquer programs: A case study in the multidimensional FFT's. In D. Mery, editor, *Formal Methods for Parallel Programming: Theory and Applications*, pages 80–94. IEEE Computer Society Press, 1997.
9. 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.
10. C. A. Herrmann and C. Lengauer. Parallelization of divide-and-conquer by translation to nested loops. Technical Report MIP-9705, Fakultät für Mathematik und Informatik, Universität Passau, March 1997.
11. E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*. Computer Software Engineering Series. Computer Science Press, 1984.
12. 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 *Proc. Int. Conf. on Parallel Processing*, volume III, pages 104–108, 1992.
13. 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.
14. J. Misra. Powerlist: A structure for parallel recursion. *ACM Trans. on Programming Languages and Systems*, 16(6):1737–1767, November 1994.
15. 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.
16. A. Schönhage and V. Strassen. Schnelle Multiplikation grosser Zahlen. *Computing*, 7:281–292, 1971.
17. D. B. Skillicorn. *Foundations of Parallel Programming*. Cambridge International Series on Parallel Computation. Cambridge University Press, 1994.
18. V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.
19. S. Thompson. *Haskell – The Craft of Functional Programming*. International Computer Science Series. Addison-Wesley, 1996.