The $\mathcal{HDC}$ Compiler Project

Christoph A. Herrmann and Christian Lengauer
Fakultät für Mathematik und Informatik
Universität Passau
D–94030 Passau, Germany
{herrmann,lengauer}@fmi.uni-passau.de

Abstract

We present a compiler for the functional language $\mathcal{HDC}$, which is syntactically a subset of the widely used language Haskell. $\mathcal{HDC}$ facilitates the clean integration of skeletons with a predefined efficient parallel implementation into a functional program. Skeletons are higher-order functions which represent program schemata that can be specialized by providing customizing functions as parameters. With $\mathcal{HDC}$, we focus on the divide-and-conquer paradigm, which has a high potential for an efficient parallelization.

1. Introduction

This paper is a condensation of a recent technical report on the $\mathcal{HDC}$ compiler [22]. The aim of the $\mathcal{HDC}$ project is to explore transformational techniques for the static parallelization of divide-and-conquer ($\mathcal{DC}$) programs. Our approach to the parallelization of recursive computations is based on the concept of *space-time mapping*, which we have borrowed from loop parallelization [28]. Since there is no general understanding of the entire space of parallel implementations for $\mathcal{DC}$ programs, as there is for loop programs, we are taking a more heuristic approach and have developed a hierarchy of skeletons which express different forms of $\mathcal{DC}$ [21].

A *skeleton* [10] is a program template which can be instantiated for the solution of a specific problem by supplying *customizing functions*. To be able to reason about our transformations, which turn a skeleton into an efficient implementation, we stay within the functional programming style, in which a skeleton is a higher-order function and the customizing functions are (some of) its parameters. Our language of choice is Haskell, but we consider only a subset of it, which we call $\mathcal{HDC}$, and we depart from Haskell in one aspect: we give $\mathcal{HDC}$ a strict semantics, while Haskell is lazy. Laziness stands in the way of a static parallelization.

Some of the technical points we make and examples we provide will be understood better with a familiarity with Haskell. We must refrain from introducing Haskell here in detail [23], for lack of space. Our $\mathcal{HDC}$ report [22] can be understood without other prior reading.

How do we envision the use and environment of $\mathcal{HDC}$? Certain central higher-order functions in the language are skeletons which come with a custom implementation for a parallel environment. Among them are our divide-and-conquer skeletons but also other basic functions which have a high potential for parallelism, like `map`, `red` (reduction) and `filter`. The programmer writes an $\mathcal{HDC}$ program with calls to these skeletons and supplies customizing functions as their arguments. Our $\mathcal{HDC}$ compiler, whose internals are described in this paper, compiles the $\mathcal{HDC}$ program into C+MPI code, using the custom implementations of the skeletons. Note that, strictly speaking, we are not doing parallelizing compilation: for each skeleton, the compiler is simply choosing a parallel implementation, parametrized in structure variables, from a library of choices. The parallel behavior of the entire program, e.g., the granularity can be controlled by assigning values to the structure variables.

How do we derive custom implementations of parallelizable skeletons? We apply a sequence of equational transformations in Haskell, which bring the body of the skeleton into a form whose structure resembles C+MPI code. In this form, a higher-order function has turned into a set of first-order functions (except for currying) and some list recursions have turned into list comprehensions, roughly Haskell's equivalent to a loop. Some comprehensions represent parallel loops. Then we translate to C+MPI.

Why did we not choose the obvious route of extending the Haskell syntax with annotations for skeleton calls and making use of one of the existing, powerful Haskell compilers? Then, the skeleton implementations (also, e.g., in C+MPI) would take Haskell closures as arguments and pass them to a new Haskell run-time environment. We decided against this for the following reasons:

- List operations in $\mathcal{HDC}$ are subject to parallelization, especially those based on list comprehensions. Today's Haskell compilers transform list comprehensions and maps into recursive expressions based on the list constructor (`:`), which introduces additional dependences. Providing a new abstract data type, which represents parallel lists, would clutter up the syntax. Also, our skeletons would have to be expressed in terms of the new data type.

- As far as we know, there exists, at present, no interface for passing Haskell closures between heaps on different processors of a distributed-memory machine.
- Our target language, C+MPI, is implemented on a wider range of systems than the special libraries which the Haskell compilers require.

Together with the compiler, we also provide an interpreter which analyzes the intermediate code produced by certain phases of the compilation and reports certain properties of the program to the user, like the *free schedule* (the number of steps if each operation is performed as soon as the data dependences permit), the average *degree of parallelism* (the number of parallel processors required by the free schedule), etc. Compiler and interpreter are either controlled interactively or by running a Haskell script which must be compiled together with the rest of the system.

2. The language $\mathcal{HDC}$

The $\mathcal{HDC}$ report [22] describes the language in detail. Here, we provide only a brief list of its features:

- An $\mathcal{HDC}$ program is a list of functions, one of which is named `parmain`. It constitutes the main function.
- $\mathcal{HDC}$ offers the primitive data types `Unit` (the unit type), `Bool` (the truth values), `Int` (the restricted integers) and `Double` (the floating point numbers). The type combinators in $\mathcal{HDC}$ are `_->_` (the function type), `[_]` (the list type), `(_,`...`,_)` (the tuple type) and `IO _` (the monadic input/output type). In addition, the user can define algebraic data types.
- Expressions in $\mathcal{HDC}$ are variables, constants, function application (also binary infix operators with sectioning), lambda abstractions, conditionals, tuples, lists, patterns, `let` expressions, `case` expressions, arithmetic sequences and list comprehensions – all akin to Haskell. The usual list constructor in functional programming, cons (`:`), stands in the way of parallelism by inducing dependences between adjacent list elements. Instead, we construct lists by special combinators and list comprehension and use the list indexing operator (`!!`), also offered by Haskell, for selection, which treats the list more like an array. The parallel implementation of list accessing operations is invisible.

3. The skeletons of $\mathcal{HDC}$

Skeletons have been used widely for parallel programming. $\mathcal{HDC}$ lends special support to programming with skeletons. Our central goal is to provide skeletons for the $\mathcal{DC}$ strategy and supply efficient parallel implementations for them.

We can divide the skeletons offered by $\mathcal{HDC}$ into three distinct categories: (1) commonly used functions, (2) skeletons for improved efficiency and (3) $\mathcal{DC}$ skeletons. (There are two more categories not mentioned here [22].)

The following subsections list the skeletons which are implemented at present. For each skeleton we provide the signature, an algorithmic definition in $\mathcal{HDC}$ and an example application. We do not present their efficient implementation here.

3.1. Skeletons for commonly used functions

These skeletons are commonly used functions which have efficient parallel implementations.

3.1.1. `map`

Applies a function to all elements of a list.

```
map :: (a->b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

```
map (+1) [0,1,2] = [1,2,3]
```

3.1.2. `red`

Uses an associative function `f` to reduce a list of values to a single value with a balanced tree computation.

```
red :: (a->a->a) -> a -> [a] -> a
red f n []     = n
red f n (x:xs) = f x (red f n xs)
```

```
red (+) 0 [1,2,3] = 6
```

3.1.3. `scan`

Applies `red` to all prefixes of the given list.

```
scan :: (a->a->a) -> a -> [a] -> [a]
scan f n xs =
  map (\i -> red f n (take i xs))
      [0..length xs]
```

```
scan (+) 0 [1,2,3] = [0,1,3,6]
```

3.1.4. `filter`

Filters all elements that satisfy some predicate.

```
filter :: (a->Bool) -> [a] -> [a]
filter p []     = []
filter p (x:xs) =
  let rest = filter p xs
  in if p x then x : rest
             else    rest
```

`filter (>2) [0,5,3,1,5]` = `[5,3,5]`

3.2. Skeletons for improved efficiency

These functions are too uncomfortable to be used by $\mathcal{HDC}$ programmers. They are generated by program transformations in view of a subsequent parallelization.

3.2.1. `while`

Takes a predicate `p`, a function `f` and a value `x` and iterates `f`, starting from `x`, as long as the predicate on the input for `f` is *True*. The `while` skeleton is intended to be the result of an elimination of tail recursion, in order to avoid the recursion stack.

```
while :: (a->Bool) -> (a->a) -> a -> a
while p f x = if p x
                 then while p f (f x)
                 else x
```

```
while (\(i,s) -> i<3)
      (\(i,s) -> (i+1,s+i*i)) (0,0)
  = (3,5)
```

3.2.2. `sinGen`

Takes a function `f` and a value `n` and generates a list of length `n` whose value at position `i` is computed by applying `f` to `i`. The aim of `sinGen` is to have a short representation for large, regular index sets, e.g., the odd numbers from 1 to 1000001. To make this work, `sinGen` has to be fused in program optimization (see Sect. 4.12.2).

```
sinGen :: (Int->a) -> Int -> [a]
sinGen f n = map f [0..n-1]
```

`sinGen (\i -> i*i) 4` = `[0,1,4,9]`

3.3. $\mathcal{DC}$ skeletons

Though a single $\mathcal{DC}$ skeleton would be sufficient for a definition of the paradigm itself, it would not adequately reflect the variety in the structure of different $\mathcal{DC}$ algorithms. In order to exploit the specific structure of a $\mathcal{DC}$ algorithm, the $\mathcal{DC}$ paradigm can be refined into different specialized forms (different skeletons) with varying patterns of data dependence and data distribution. A hierarchy of five such skeletons, which we call `dc0` to `dc4`, is described in [21].

At present, $\mathcal{HDC}$ provides the general $\mathcal{DC}$ schema, in the form of skeleton `dc0`, and a modified form `dc4io` of the most specialized skeleton `dc4`, which allows for massive data parallelism.

3.3.1. `dc0`

$\mathcal{DC}$ in its general form.

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

If the predicate function `p` determines that the problem `x` can be solved trivially, the basic function `b` is applied. Otherwise the problem is divided by `d`, producing a list of subproblems. The algorithm is mapped recursively onto the subproblems. At last, the combine function `c` uses the input data `x` and the solutions of the subproblems to compute the solution of the original problem.

A functional version of the *quicksort* algorithm can be expressed in terms of `dc0`:

```
quicksort :: Ord a => [a] -> [a]
quicksort xs =
  let d (p:ps) =
         [filter (<p) ps,
          filter (>p) ps]
      c (p:ps) [le,gr] =
         le ++ p :
            (filter (==p) ps ++ gr)
  in dc0 ((<2).length) id d c xs
```

`p` is the name of the *pivot*. `d` generates two subproblems of the elements that are less resp. greater than the pivot. `le` resp. `gr` are the solutions of these subproblems. `c` combines them and inserts the elements which equal the pivot in the middle.

3.3.2. `dc4io`

A special kind of $\mathcal{DC}$ whose call tree is balanced and which requires elementwise divide and combine operations on subblocks of data.

The definition of `dc4io` is more involved than the others, and we can only sketch it here:

```
dc4io prob in out basic divide
      combine levels xs = ...
```

The parameters of `dc4io` have the following meaning:

- `prob::Int`: the degree of problem division, i.e., the number of subproblems which are generated for each problem not trivially solved; this degree is fixed in `dc4io`, in contrast to `dc0`.
- `in::Int`: the degree of division of input data; it tells in how many blocks the input data is to be divided.
- `out::Int`: the degree of composition of output data; it tells of how many blocks the output data is to be composed.
- `basic::(a->b)`: the function to be applied in the trivial case.
- `divide::([a]->[a])`: the function `divide` takes a list of length `indegree` as input and delivers a list of length `probdegree` as output; it describes how the elementwise operation computes the i-th element of each particular subproblem, using the i-th element from each input block.
- `combine::([b]->[b])`: the function `combine` takes a list of length `probdegree` as input and delivers a list of length `outdegree` as output; it describes how the elementwise operation computes the i-th element of each particular output block, using the i-th element from each subproblem solution.
- `levels::Int`: the number of recursive levels into which the $\mathcal{DC}$ tree unfolds. This replaces the predicate of `dc0`. In the model, `levels` reflects the number of levels upto the trivial cases. In practice, the user can make additionally two other choices (which are not mutually exclusive): control granularity of parallelism or solve small problem instances (which are not the basic case) by an algorithm tailored for small sizes.
- `xs::[a]`: the input data; it is a list on which the division into blocks apply; likewise the output data is also of type list.

`dc4io` works well for vector and matrix algorithms like FFT, bitonic merge, polynomial multiplication and matrix multiplication [19].

`indegree` and `outdegree` depend much on the data representation, e.g., for vectors they have the value 2 (left and right part), for matrices they have the value 4 (upper left part, upper right part, lower left part and lower right part), see Tab. 1.

The parenthesized values are for the optimized version of the respective algorithm, e.g., for Karatsuba's polynomial multiplication and Strassen's matrix multiplication [1].

problem	`prob`	`in`	`out`
FFT, bitonic merge	2	2	2
polynomial multiplication	4 (3)	2	2
matrix multiplication	8 (7)	4	4

Table 1. example $\mathcal{DC}$ division degrees

4. The compiler for $\mathcal{HDC}$

The $\mathcal{HDC}$ compiler translates a subset of Haskell into an imperative language – at present, C with MPI calls. The main difference to Haskell is that $\mathcal{HDC}$ is strict, in order to facilitate a compile-time parallelization. Two implementational differences to a typical Haskell compiler are that (1) higher-order functions without a skeleton implementation are eliminated and (2) list comprehensions are simplified to a combination of (parallel) skeletons. The reason is that higher-order functions complicate but list comprehensions simplify a static space-time mapping.

The compiler is based on the principle of compilation by transformation, which has already been used successfully for the Glasgow Haskell compiler **GHC**, and which consists of a number of phases described in the following subsections.

4.1. Scanner and parser

The source text is translated into a set of syntax trees, one for each function in the program. Each syntax tree is represented as an algebraic data type in Haskell.

The layout style of Haskell is supported, i.e., indentation can be used instead of braces and semicolons to group together items at the same level of particular syntactic structures. The user can declare new operators and state their precedence and associativity. This information is exploited by the parser.

4.2. Desugaring

In this phase, complex syntactic structures are translated to compositions of simpler structures. Nested patterns are eliminated, in order to simplify the code structure for the following phases. After the transformation, the pattern is either a simple variable or a constructor, possibly, with arguments. An equational transformation of a into b is denoted by $a \overset{\rightarrow}{=} b$.

4.3. List comprehension simplification

GHC resolves comprehensions completely, up to the construction by the empty list (`[]`) and cons (`:`), by traversing the list of qualifiers from left to right. Our goal is to base list

lcEmpty	$\overset{\rightarrow}{=}$	`[` e `| ]`
		 `[` e `]`

lcSinGuard
 $\{g$ is a guard$\}$
	$\overset{\rightarrow}{=}$	`[` e `|` g `]`
		 `if` g `then [` e `] else []`

lcOptGuard
 $\{g_i$ are guards,
 $x \notin \mathit{freevars}(g_k)\}$
	$\overset{\rightarrow}{=}$	`[` e `|` q_0`,`$\ldots$`,`q_n`,`x `<-` xs`,`g_0`,`$\ldots$`,`g_k`,`$\ldots$`,`g_m `]`
		 `[` e `|` q_0`,`$\ldots$`,`q_n`,`g_k`,`x `<-` xs`,`g_0`,`$\ldots$`,`g_{k-1}`,`g_{k+1}`,`$\ldots$`,`g_m `]`

lcXGen	$\overset{\rightarrow}{=}$	`[` e `|` q_0`,`$\ldots$`,`q_n`,`x `<-` xs `]`
		 `concat [ map (\`x `->` e`)` xs `|` q_0`,`$\ldots$`,`q_n `]`

lcGenGuard
 $\{g$ is a guard$\}$
	$\overset{\rightarrow}{=}$	`[` e `|` q_0`,`$\ldots$`,`q_n`,`x `<-` xs`,`g `]`
		 `concat [ map (\`x `->` e`) (filter (\`x `->` g`)` xs`) |` q_0`,`$\ldots$`,`q_n `]`

lcTwoGuards
 $\{g_0, g_1$ are guards$\}$
	$\overset{\rightarrow}{=}$	`[` e `|` q_0`,`$\ldots$`,`q_n`,`g_0`,`g_1 `]`
		 `[` e `|` q_0`,`$\ldots$`,`q_n`, if` g_0 `then` g_1 `else False ]`

Figure 1. Simplification of list comprehensions

comprehensions on (parallel) skeletons. As presented here, our rewrite rules in Fig. 1 specify the traversal of the list of qualifiers in the opposite order: from right to left. This has two advantages: (1) nested `map`s are not intertwined with nested `concat`s, which preserves structural information; (2) an efficient `filter` skeleton is used instead of generating lots of empty lists in cases in which guards fail. The disadvantage is that the rules will become far more complicated if extended to the full capability of Haskell.

The rewrite rules shown in Fig. 1 cover all possible list comprehension formats in our restricted language. They replace a list comprehension by applications of the skeletons `concat`, `map` and `filter` which are supposed to have efficient implementations. The rules are applied until no further application is possible. If there is a choice between several rules, the one highest up in Fig. 1 is most efficient.

Rule **lcEmpty** deals with the case that the sequence of qualifiers has become empty by the other transformations.

Rule **lcSinGuard** simplifies a qualifier list consisting of a single guard. Depending on the value of the guard, the result is a list of either length 1 or length 0.

Rule **lcOptGuard** shifts a guard as far as possible to the left, in order to avoid multiple evaluations.

Rule **lcXGen** deals with the case that the last qualifier is a generator. The other qualifiers define a list of environments. In the comprehension before simplification, the last qualifier refines each element of the environment list by a set of new bindings for the last generator variable x. After the transformation, this refinement is shifted to the expression on the left side of the comprehension, which has been replaced by a list, one element for each instance of the last generator in the current environment, as defined by the other qualifiers. We reuse the name x of the generator variable for the lambda expression to preserve the bindings in the transformation. Note that the left side is in the scope of the environment defined by the qualifiers on the right side. Therefore, all free variables of xs are bound to the same values as before.

If a guard appears behind a generator, rule **lcGenGuard** helps to fuse the two. It is similar to rule **lcXGen**, except that the new bindings for the last generator variable, which lead to a failure of the last guard, are eliminated from the list via the `filter` skeleton before. The previous application of rule **lcOptGuard** assures that the guard really refers to the variable bound by the generator.

If two guards appear next to each other, they can be simplified to a single guard according to rule **lcTwoGuards**.

4.4. Lambda lifting, `let` elimination

Lambda abstractions and `let` expressions are eliminated by introducing auxiliary functions [24].

4.5. Type checking

The type checker is based on unification using the Martelli-Montanari rules [30]. A simple type class system is implemented by assigning a type variable a set of possible types. The unification of two type variables then involves computing the intersection of both sets.

4.6. Monomorphization

In this phase, all type variables are eliminated and replaced by the types actually needed. This requires the duplication of each function for all concrete types which occur in the context.

Monomorphization is needed because our aim is not to translate to a high-level target language but to stay close to the machine representation of data and instructions.

4.7. Elimination of functional arguments

This phase, also called *higher-order elimination* (HOE), takes a program which must be well-typed according to the Hindley-Milner rules. Also, the program must be closed, i.e., all functions cited in the program must be available to the HOE procedure for a global analysis and transformation. The result of the HOE is an equivalent first-order functional program, which is also well-typed.

4.7.1. Principles

The HOE algorithm we found [5] uses a set of seven rewrite rules for the transformation. The idea is to replace the partial applications of a function by a kind of closure. A closure contains a function identifier and the values of the free variables in the partial application.

The replacement of functional arguments by closures proceeds as follows:

- A variable of a function type is left unchanged because it represents already a closure.
- A partial application of a function is replaced by an instance of an algebraic data type in which the function identifier is represented by a constructor. The arguments of the constructor carry the values of the free variables in the partial application. These values are taken from the context of the call.
- All locations at which a functional variable is applied are replaced by a call of an apply function constructed for the respective function type. The first argument of the apply function is the closure, the following arguments are the arguments of the encoded function. The apply function applies the original function, with the respective partial application derived by the constructor, to the argument expression in the context of the call which can use values of the closure, also encoded functions.

4.7.2. Rules

Some of the seven rules, which the original HOE algorithm [5] is based on, deal with restricting polymorphism and become obsolete in our monomorphic setting. We use four rules, which are described in brief [17] and in full [22], with a detailed example, elsewhere.

After the HOE, all function applications are saturated with arguments, such that the result is not a function. Also, no argument to a function is a function. In principle, one could now replace all curried definitions and applications by tuple representations. This is not done in the $\mathcal{HDC}$ compiler for two reasons:

1. The tuples, which are objects of the $\mathcal{HDC}$ language, are, in turn, expressed in terms of pattern matching `case` expressions, which require curried functions on the right hand side again.
2. The interpreter can remain simpler if it only has to deal with curried functions.

We adopt the following convention: after the HOE, any application of an $\mathcal{HDC}$ function has to supply all curried arguments. This schema can be regarded as first-order and is equivalent to a schema of tupled arguments.

4.8. Elimination of mutual recursion

The $\mathcal{HDC}$ compiler uses two methods for the removal of mutual recursion in programs: *elimination by inlining* and *elimination by emulation*. Mutual recursion is identified by calculating the strongly connected components (SCCs) in the graph of functional dependences. Since there is no mutual recursion between SCCs, the methods can be applied to each SCC independently.

4.8.1. Elimination by inlining

This method can only be used for an SCC which contains a node f whose removal from the SCC would make the residual graph s acyclic. The set of functions which s represents is, therefore, free of mutual recursion. Thus, it is possible to inline all calls of functions in s in the body of f, until the only recursive calls left are directly recursive [26].

4.8.2. Elimination by emulation

If all mutual recursion in a program is to be removed, an alternative approach has to be taken for SCCs in which mutual recursion cannot be eliminated by inlining. It is always possible to transform an SCC to a supernode. The function associated with a supernode emulates the work of all functions of the SCC by encoding the actual parameters and decoding the formal parameters. Let f_i, $1 \leq i \leq n$, be the functions of an SCC and $m(i)$ the number of arguments of function f_i. Function f', which emulates the f_i, is given in Fig. 2.

$f_i :: t_{(i,1)} \ \rightarrow\ t_{(i,2)} \ \rightarrow\ \texttt{...}\!\rightarrow\ t_{(i,m(i))} \ \rightarrow\ t_{(i,0)}$
$f_i \ arg_{(i,1)} \ arg_{(i,2)} \ldots arg_{(i,m(i))} = body_i$

can be emulated by a new function

$f' :: \texttt{Data} \rightarrow \texttt{Data}$
$f' \ arg$
$\quad = \texttt{case}\ arg\ \texttt{of}$
$\qquad C_1 \ arg_{(1,1)} \ldots arg_{(1,m(1))} \ \rightarrow CR_{t(1,0)} \ body_1$
$\qquad \vdots$
$\qquad C_n \ arg_{(n,1)} \ldots arg_{(n,m(n))} \rightarrow CR_{t(n,0)} \ body_n$

Figure 2. Elimination of mutual recursion

To avoid type conflicts, it is necessary to create, for each function f_i of the SCC, a constructor C_i to encode the arguments and a constructor $CR_{t(i,0)}$ for the result. The constructor name is used to select the body of function f_i. Finally, calls to the functions f_i have to be adapted to fit f'.

Whenever possible, elimination by inlining should take precedence over elimination by emulation. Inlining does not spoil the structure of the program and the resulting intermediate code can usually be optimized more effectively.

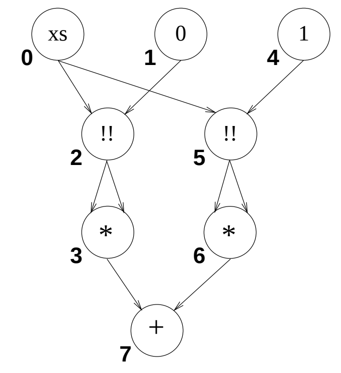
Both methods are expensive if the program contains cycles of mutual recursion with more than three to four functions. Unfortunately, cycles may be introduced by the transformations of earlier compilation phases. If programs are getting too big, due to the removal of mutual recursion, the elimination process can be turned off by setting a compiler switch. The default is to apply elimination by inlining, where possible, and then use the alternative method for the remaining mutual recursions. The $\mathcal{HDC}$ programmer should prefer the use of skeletons to user-defined recursive functions in order to keep the amount of recursion low.

4.9. `case` elimination

Pattern matching is not available in lower-level programming languages, such as C or assembler, which are suitable for the target code of $\mathcal{HDC}$. A run-time system with pattern matching would incur too much overhead. Therefore, we eliminate `case` expressions. We replace a `case` expression by nested `if` expressions. The branches of the `if` expressions contain the former right-hand sides of the `case` branches.

`case` expressions which have only one branch receive a special treatment: no `if` expression is needed, assuming that the branch will always match.

```
sqrplus ::  [Int] → Int
sqrplus xs  = (xs!!0)*(xs!!0)
            + (xs!!1)*(xs!!1)
```



0:	`xs`		:: `[Int]`
1:	`0`		:: `Int`
2:	`(!!)`	↑**0** ↑**1**	:: `Int`
3:	`(*)`	↑**2** ↑**2**	:: `Int`
4:	`1`		:: `Int`
5:	`(!!)`	↑**0** ↑**4**	:: `Int`
6:	`(*)`	↑**5** ↑**5**	:: `Int`
7:	`(+)`	↑**3** ↑**6**	:: `Int`

Figure 3. Example: program, DAG, and table

4.10. Generation of intermediate DAG code

The syntax tree of each function is transformed to a directed acyclic graph (DAG) to enable sharing of common subexpressions. A DAG corresponds to a set of expressions with associated numbers, ordered by their dependences. Subexpressions are referenced by the corresponding numbers. Fig. 3 shows an example program with its DAG and the corresponding table representation. The direction of the references is inverse to that of the data flow, which is depicted by the arrows in the figure.

The transformation of a syntax tree into a DAG is by a standard technique called the *value number method* [2]. The nodes are enumerated such that the source of each data dependence has a smaller number than the target. The transformation proceeds by a depth-first traversal of the syntax tree. The subject of the transformation of each node is an expression, in which subexpressions have already been converted to numbers by recursive application of the transformation. It returns a number for the node as follows: if there is already an expression in the DAG that matches the input, then the number of the existing expression is returned; otherwise, a new node for the expression is created and its number is returned. Optimizing transformations (as described below) are performed at this intermediate code level.

4.11. Tuple elimination

Tuples are replaced by algebraic data types, one for each occurring tuple type. Each tuple is tagged with the appropriate constructor for its particular type. This simplifies the runtime system and, at the same time, provides fast access to information about the types and sizes of the components of the tuple by looking them up in a table. As a consequence, the memory management functions need not be specialized for each particular type.

4.12. Optimization cycle

Code optimization proceeds in a cycle. Each iteration performs three steps in sequence: inlining of functions calls, rule-based DAG optimizations, and size inference.

The process of replacing the call of a function by its body, after substituting the actual for the formal parameters, is called *inlining*. We use inlining to enable further optimizations on DAGs, e.g., deletion of dead code and sharing of common subexpressions. Inlining is performed on DAGs in which common subexpressions already appear only once. Due to the sharing of common subexpressions, there is no risk of duplicating work.

Inlining triggers common subexpression elimination for two reasons. First, it aggregates common subexpressions which have been spread across the program – maybe due to transformations made before, maybe due to the program itself. Second, it specializes variables by replacing the formal parameters of the function inlined by the actual parameters. This permits partial evaluation and checking for value equality rather than name equality when identifying common subexpressions. Value equality is a coarser equivalence relation, i.e., it induces more commonality.

The information gathered in the size inference is useful for the inlining heuristics of the following iteration of the optimization cycle and for the space-time mapping. Size inference has to be reapplied in each iteration because of the changes in the program due to inlining.

During each iteration, every DAG needed is processed as follows.

4.12.1. Inline expansion

The inline expansion transforms a source DAG into a corresponding target DAG with possibly inlined calls. First, the nodes of the source DAG are copied successively. If a node representing a function call is reached, a heuristic decision, based on the expected amount of code increase, is made as to whether to inline this call or just copy the call node. In the case of inlining, the copying process switches its source temporarily from the caller to the callee. All nodes of the DAG of the callee will be copied. There is no recursive inlining of calls. Copied calls may be inlined in the next pass. Every time the inline expansion of a DAG for some function is completed, the body in the function definition is changed to the target DAG.

After inlining in the current pass is finished, the DAGs are simplified. Unused function arguments, except from apply functions which are called from skeletons, are deleted and dead code is removed. If it was possible to inline at least one call in the current pass and a specified maximum number of passes is not yet reached, inlining is repeated in the next pass. We chose two major strategies for inlining: *current version inlining* and *original version inlining*.

- **Current version inlining**
  The most recent DAG for the called function is inlined. This method requires fewer inlining operations, since DAGs with already inlined calls are used for inlining again. One drawback is that the functions are growing very fast and, therefore, the inlining process may be suppressed after only a few steps.

- **Original version inlining**
  The original definition of the function is inlined. This incurs a linear code growth when inlining recursive functions. Original version inlining offers more possibilities for optimization and, therefore, may lead to better results [25].

Kaser et al. [25] also compare static and profile-based approaches. At present, we do not accumulate or exploit profiling information.

Common subexpressions are eliminated during the conversion of the syntax tree into a DAG, which has already been described in Sect. 4.10. However, new common subexpressions may appear during the inlining of a call. Since new nodes introduced by the inlining process are always created with the same function as used for DAG construction, no unshared common subexpressions will be created.

4.12.2. Rule-based DAG optimizations

In this step, various algebraic optimizations can be applied. In the interest of brevity, let us focus here on optimizations in the context of space-time mapping (Sect. 4.14).

In numerical algorithms, in a call `map` f xs, the list xs is often of type `[Int]`, which defines a set of indices. In the simplest case, it is an arithmetic sequence (`[`a`..`b`]`) or obtained from index set transforming functions. Since explicit enumerations of index sets are, in general, too inefficient, we represent them by functions (this requires a certain regularity). We can generate an index set from its describing function by a skeleton named `sinGen`, see Sect. 3.2.2. It takes a function f, which describes an index set, and an integer n and delivers a list of length n, in which the i-th element is defined by applying f to i.

intr-sinGen

`map` f xs $\{xs ::$ `[Int]`$, xs$ regular, <u>fresh</u> $i\}$

$\stackrel{\rightarrow}{=}$ `sinGen (\`i `->` f `(`xs`!!`i`)) (length` xs`)`

elim-sinGen

`(sinGen` f n`) !!` i

$\stackrel{\rightarrow}{=}$ f i

Figure 4. Optimization rules for `sinGen`

Fig. 4 contains two optimization rules: one which introduces and one which eliminates `sinGen`. Note that this enables some potential for optimization, e.g.:

`map` f xs `!!` j

$\{$intr-sinGen, <u>fresh</u> $i\}$ $\stackrel{\rightarrow}{=}$

`sinGen (\`i `->` f `(`xs`!!`i`)) (length` xs`) !!` j

$\{$elim-sinGen$\}$ $\stackrel{\rightarrow}{=}$

`(\`i `->` f `(`xs`!!`i`))` j

$\{$application$\}$ $\stackrel{\rightarrow}{=}$

f `(`xs`!!`j`)`

These optimizations have the problem that higher-order arguments are reintroduced (e.g., for the lambda expressions introduced above). Of course, one could apply such optimizations before HOE but, at this point, they would miss the applications that are enabled by the inlining specializations coming later.

4.12.3. Size inference

The size inference algorithm derives symbolic information about the result returned by a function from structural variables which represent the symbolic information of its arguments. The goal is to improve the decisions made during each iteration of the optimization cycle and to determine automatically a space-time mapping at compile time, if possible.

We are interested in the following symbolic information about an $\mathcal{HDC}$ function:

1. the size of the result – in the case of nested lists a comprehensive description of all levels [20],

2. the number of operations,

3. the length of the longest path in the DAG, if all calls are expanded,

4. the number of steps for a given number of processors, if the communication cost is disregarded – this can be estimated from the number of operations and the path length, using Brent's theorem [32].

The size inference computes an abstract version of the $\mathcal{HDC}$ function, which takes the same number of arguments and has the same structure as the original function, but the operations it performs are abstract counterparts of the original operations. E.g., an abstract size operation for the append operator of plain lists is, simply put, "addition", because the size of the result is the sum of the sizes of both operands. The abstract version of a function application is the application of an abstract function to an abstract size.

The sizes are represented in a symbolic form, as objects of an algebraic data type `Size`, containing, e.g., the following constructors:

- `Con ::  Int` $\rightarrow$ `Size`
- `Var ::  String` $\rightarrow$ `Size`
- `Add ::  Size` $\rightarrow$ `Size` $\rightarrow$ `Size`

If, e.g., `Con 2` is the size of the list `[3,4]` and `Var "x"` is the size of a list `x`, then the size of the list `[3,4] ++ x` is `Add (Con 2) (Var "x")`.

Abstract functions take variables representing symbolic expressions. E.g., the constructor `Add` above is an abstract function. Abstract functions can be composed of other abstract functions. E.g., the number of operations needed for a list append depends on the length of both argument lists. (That is the $\mathcal{HDC}$ append; for the usual sequential append, the length of the second list is immaterial.) From this point of view, underlying memory optimization techniques like sharing in DAGs, which are not visible at the level of intermediate code, are not considered. The structures are treated as if they were flattened and the abstract values obtained are upper bounds and not exact.

The abstract functions are expressed in terms of the abstract values of their arguments, in order to make size inference a local computation, independent of its context, and allow for a largely polymorphic implementation of the skeletons. If the amount of space in terms of memory cells or the amount of time in terms of clock cycles is of interest, the abstract function must be supplied with according context information.

Because of the complexity of the symbolic expressions involved, we believe that the size inference of recursive functions is beyond the capability of present-day mathematical tools.

We expect all recursion to be captured in skeletons, which are supplied with the four types of size information stated above. The size information of other, non-recursive functions can be obtained by composition of symbolic functions, preferably, using simplification.

Size inference is applicable only if the structure analyzed does not depend on run-time data, e.g., if the length of lists does not depend on input values.

We see a use for a complete size inference mainly in functional programs which represent a static system, e.g., a hardware description.

4.13. Abstract code generation

The defining expression of an intermediate function is mapped to a DAG in order to facilitate the sharing of common subexpressions. As described in Sect. 4.10, each DAG is represented by a table: each node of the DAG corresponds to a table entry, and each directed edge of the DAG is represented at the entry of the target of the data dependence by the table index of the source of the data dependence.

The phase of abstract code generation switches the interpretation of a DAG: before, it is interpreted with a denotational semantics, afterwards with an operational semantics. A small fraction of the nodes of the DAG also change: one type of node is eliminated and three other types are introduced.

Let us reflect on the denotational interpretation. Here, a DAG is interpreted by starting evaluation of a distinguished node, the *root*. The result of the root node is considered the result of the function represented by the DAG. If the evaluation of a node requires the result of another node, this node is visited and evaluated. There is a special kind of node for accessing formal arguments. `if` nodes require a special treatment: the value of the condition has to be tested, and then only one of both branches is evaluated.

In the operational interpretation, the evaluation proceeds by traversing the table entries in sequence. If the result of another node is required, it has already been computed and can be looked up in a previous table entry. The root node is the last entry in the table and contains the result of the function. The problem with the `if` nodes is that when they are reached (if ever!) both branches already have been evaluated, also the wrong branch. Therefore, a mechanism is implemented to skip nodes belonging to the wrong branch. If a DAG does not contain `if` expressions, it is used as abstract code without modification.

The $\mathcal{HDC}$ report [22] contains the technical details of the generation of abstract code generation, and an example.

4.14. Space-time mapping

A space-time mapping is a one-to-one mapping from a domain of computation points to the Cartesian product of discrete space and time. The space part is known as *allocation*, the time part as *schedule*. The technique of space-time mapping has a long tradition in loop parallelization [27]. Some of the ideas can be adapted to `while` loops [11, 16] and even to non-linear recursion [18]. However, the structure of the dependences we are encountering makes integer linear optimization, which is the central search method for a space-time mapping in loop parallelization and which has the nice property of yielding the best solution in the considered search space, unsuitable for general $\mathcal{HDC}$ programs.

Space-time mapping is most effective when applied in the individual derivation of parallel skeleton implementations. This approach is described in detail elsewhere [18]. If the dependence structure of the skeleton is sufficiently regular –as, e.g., for some kinds of $\mathcal{DC}$– the points of computation can be laid out in time and space at compile time. The size of the computation space will depend on the problem size and the number of processors, but its shape will not [18].

The user is well advised to construct his/her program by composition of appropriate skeletons, which have efficient implementations.. Note that the generation of each skeleton is done by a Haskell function which is to be delivered by the skeleton implementer. It is up to this Haskell function, to use the results of the size inference provided or even to call external tools. The task of the $\mathcal{HDC}$ compiler is, at a minimum, to transmit symbolic space-time mapping information via the call structure of the program to the points where it is needed, by making use of the abstract functions delivered by the skeleton implementer. The nodes of each DAG are scheduled sequentially by the compiler, no parallelization is done in this phase.

4.15. Target code generation

The code generation phase first produces C code, which is then compiled with a standard C compiler and linked together with the functions of the $\mathcal{HDC}$ run-time library, which are also written in C. The C code is generated in two phases. First, the abstract code of the user program is translated; see Sect. 4.15.1. Second, an appropriate implementation is generated for each skeleton instance used in the program; see Sect. 4.15.2.

4.15.1. DAG compilation

For each DAG of the abstract code, a C function is generated and appended to a file. Each node in the DAG is treated seperately. Each constructor used is inserted into a table, in order to provide the run-time environment with the necessary information about types and sizes of the components of the object it constructs. For each call of a skeleton, the name of the skeleton is stored with the actual types of the arguments.

4.15.2. Skeleton generation

After all DAGs have been processed, the instantiations of the skeletons are generated and stored.

$\mathcal{HDC}$ offers a special, very flexible mechanism for the integration of custom-implemented skeletons. For each skeleton, the implementer delivers a Haskell function, say, Φ, which is called by the code generator of the $\mathcal{HDC}$ compiler and which produces the actual instance of the skeleton.

In the simplest case, the body of function Φ will be just a Haskell string of C target code, but Φ can also prescribe decisions based on type and size information provided by the compiler.

Remember that the C code generated must be monomorphic; this applies also to the implementation of a skeleton. Thus, the programmer of Φ has to consider at least the root symbol of the type tree of each argument, i.e., the implementation must differ, e.g., between lists and integers, but not necessarily between lists of `Int` and lists of lists of `Int` since, in the latter case, the root of the type tree is in both cases the list type constructor.

To illustrate what a parametrized skeleton implementation may look like, let us discuss a bit further an abstract version of the implementation of the `map` skeleton in a parallel model, in which all data is passed along with the control. The details of the run-time system supporting this implementation are presented in Sect. 5.4.

The `map` skeleton takes a function (really a closure, i.e, the code of a function together with an environment) and a list and applies the function to every list element.

For simplicity, we assume a space-time mapping which allots roughly the same number of list elements to each processor. This mapping is efficient if the amount of work is nearly equal for each element of the list.

One might consider the use of collective MPI operations [31] like *broadcast* (to distribute the function closure), *scatter* (to distribute the list among the processors) and *gather* (to collect the results from the processors). This would work for lists of `Int`, `Bool` or `Double` but would require special skeleton implementations for these types. In general, *gather* and *scatter* cannot be used, since they assume that lists are plain and do not contain references to a heap. E.g., if the elements are functions, the list contains just pointers to a shared part of the heap. As a consequence, we have to custom-implement collective operations for $\mathcal{HDC}$, using the MPI primitives *send* and *receive*. Here, again, $\mathcal{DC}$ proves to be a useful technique.

4.15.3. Run-time library

The run-time library contains the implementation of all functions which do not depend on the user program – especially, predefined functions which cannot be coded with a few C statements (those are inserted directly in the code), which perform memory management, and which marshal/unmarshal data structures, i.e., encode/decode a heap-allocated DAG into/from a linear buffer.

5. The parallel run-time environment

5.1. The model of parallel execution in $\mathcal{HDC}$

Our aim has been to provide a platform which does not limit the design choices concerning $\mathcal{DC}$ parallelism. Still, we are staying away from unstructured fork-join parallelism [3] and, where possible, make use of the $\mathcal{DC}$ paradigm. In the interest of generality and scalability, our execution model is SPMD.

The unfolding of recursion can be viewed as a call tree in which the root represents the entire problem and the successor relation corresponds to the subproblem relation. Time proceeds down the tree: each level corresponds to one parallel computation step.

At the start, all processors are assigned to the root of the call tree. Every node partitions its processors among its successors. We call the set of processors belonging to a node a *block*, and the set of processors of each of its successors a *subblock*. Each block has a distinguished processor we call the *master*, which is responsible for the operations at the node the block is assigned to. In problem division, the master of a block activates the master of the subblocks and, if the computations of the subproblems have been finished, the masters of the subblocks become idle.

No control is passed and no argument or result values are exchanged across a block's border. We call this the principle of *communication-closed blocks*, in analogy to the principle of *communication-closed layers* [14]. Its goal is to avoid the introduction of deadlocks or races in the composition of skeletons. It does not rule out access to distributed data, because we use the model of a virtual shared memory whose underlying implementation is transparent to the skeleton implementer.

Another important issue is the data layout, i.e., the way in which the data is distributed among the processors. We distinguish three styles of data layout. The first applies to atomic data and to tuples. Only lists and algebraic data types, which can become large, are subject to the other distributions.

- **Centralized data layout:** All input data of a task is passed along with the signal of initiation of the task and all output data is passed back with the report on the task's completion. Obviously, this is a good choice if the amount of data is small, although it might incur some unnecessary communication. For large data, a centralized data layout will lead to unacceptable overhead, due to data transmission, or even to memory overflow.

In the remaining two layout styles, instead of passing the data with the control, only information about the location of the data is passed.

- **Hierarchical data layout:** The input and output data of each block is distributed among the processors assigned to the block, as prescribed by the space-time mapping. The default space-time mapping is that the data is distributed in balance across all available processors. This layout is especially convenient for $\mathcal{DC}$ algorithms on large data which does not fit onto a single processor, but only if the data size decreases with the division of the problem, as in the `dc4io` skeleton.

- **Globally distributed data layout:** The input data is distributed according to a space-time mapping. Each intermediate and result value is located on the processor that produces it. This is known as the *owner-computes rule* [35].

  The only difference to the centralized data layout is the use of a call-by-reference mechanism: instead of a large volume of data, only a reference to the data is passed. Remote data retrieval is realized by direct access to the remote memory. Remote access can only be reading, not writing. Using reference counting, a data object is preserved as long as there exists a direct reference to it.

5.2. Organization

The available implementations of a skeleton determine the set of possible space-time mappings which can be chosen in a parallel execution. Thus, it is important to realize that skeleton implementations are generated in dependence of the context in which they are called, exploiting type and possibly also symbolic size information (see Sect. 4.15).

The first argument of each skeleton implementation, like the first argument of the other functions generated, contains a pointer to system information, comprising a description of the master and the current part of the topology the processor belongs to.

The user has the option of providing all functions with an additional explicit functional argument, which contains a mapping strategy coded in $\mathcal{HDC}$. Skeleton implementations can use this strategy in the space-time mapping.

5.3. Memory management

The memory for the local variables of functions is stored on the stack. This results from a direct translation to C. Lists and algebraic data types, which are not supported by C, receive special support from our run-time system. They are stored as linked structures in the heap. Common subexpressions are shared, which means that the linked structures form a directed acyclic graph.

By default, each processor manages its own local heap using reference counting for release [13].

Distributed data structures have to be maintained explicitly by skeleton implementations using functions of the run-time system for remote memory access. Arguments passed to another processor are marshalled in their entirety. It remains the task of program optimization not to pass large structures to a function if only a small part of it is actually required.

5.4. Interaction with skeleton implementations

If the computation is divided into subcomputations according to the $\mathcal{DC}$ paradigm, the block of processors is divided into subblocks. Each processor belongs to exactly one subblock. Each subproblem is solved on its own subblock. At the beginning, the block has one master processor, the other processors are *slave*s. The division of a block involves the creation of new masters by the old master, one for each subblock.

Let us now revisit the implementation of the `map` skeleton from Sect. 4.15.2. If the list does not contain at least two elements or the block has only one processor, `map` must be computed sequentially. (`map` *may* be computed sequentially if parallelization does not pay off according to a strategy chosen by the skeleton implementer.) Otherwise the block is divided into two subblocks; let us call them the *left* subblock and the *right* subblock. The left subblock computes `map` on the left part, and the right subblock on the right part of the list. The processor responsible for the whole block before, say, L retains the responsibility for the left subblock and sends the packed function closure and the right part of the list to another distinguished processor, say, R responsible for the right subblock. Computation proceeds recursively until the left and right subblock are united again and R gives back control for its part to L.

Now, let us have a closer look at the call mechanism. Processor L is the one on which the `map` skeleton is called. Thus, it receives all formal arguments via a function call. Processor R is activated by L with an index of the actual skeleton instance. R uses this index to call a *slave skeleton*. This skeleton does not receive the application data via a function call, because this data is not yet available on R. Instead, the following interaction takes place: L sends the data to R, both L and R call the master skeleton with their particular subproblem, R returns into the slave skeleton and sends its result back to L. Note that Φ of Sect. 4.15.2 has to generate two skeleton instances here: the one for the master and the one for the slave!

6. Experimental Results

We conducted our experiments on the NEC Cenju-4 parallel computer. All entries are times measured in seconds on

# procs.	# queens		
	10	11	12
1	4.87	25.05	139.44
2	9.86	47.78	247.94
3	6.83	32.18	162.91
4	3.10	17.25	92.17
5	1.75	9.12	53.13
8	0.81	3.84	25.31
10	0.52		
11		2.40	
12			12.14
16		2.41	12.18
32		4.86	23.08
64			5.61

Table 2. N queens

# procs.	size		
	2048	4096	8192
1	13.89	↑	↑
2	34.27	↑	↑
3	4.70	13.94	↑
4	4.68	14.00	↑
5	4.69	13.96	↑
8	11.71	34.99	↑
9	1.70	4.91	15.89
16	1.74		
27	0.72	1.94	6.92
64		5.32	16.94

Table 3. Polynomial multiplication

the I/O performing processor, without the time for I/O itself. Both example programs are based on the $\mathcal{DC}$ skeleton `dc0`. Note that the significant speedups are achieved where the $\mathcal{DC}$ skeleton meets the balance of the problem. Excessively high execution times are due to imbalance and parallelization in places where it does not pay, triggered by the availability of processors. This suggests the use of a skeleton which is better tailored for the particular problem than `dc0`, in combination with an analysis of the amount of work needed for the evaluation of the customizing functions.

6.1. N queens

The N queens problem is to compute the number of all possible solutions for placing N queens on an $N \times N$ board, such that no two queens are on the same row, the same column, or the same diagonal, by using a decision tree. Although there may be sophisticated combinatorial ways to simplify this problem, we use it as a representative for exhaustive search problems here. Our only heuristic is to try to recognize conflicts as early as possible.

The problem is most balanced if the number of processors is a multiple of the number of queens. It turned out that the implementation of the free schedule across the $\mathcal{DC}$ tree is not ideal here. The reason is the imbalance of work for different branches of the tree. Experiments we made with a parallelization of a plain map-recursive version of the N queens problem showed a smoother growth of the speedup.

6.2. Optimized polynomial multiplication

We use the same method Karatsuba used for the multiplication of large integers [1]. The principle is to exploit the fact that two of the four naive subproblems in a left-right division are not needed seperately, but only in their sum. This leads to only three subproblems with a reduction in complexity from $\theta(n^2)$ to $\theta(n^{1.58})$, which reduces the execution time significantly for large problem sizes.

In Tab. 3 an uparrow indicates a memory overflow.

Here, the balance of the problem is met best if the number of processors chosen is a power of 3, which is the number of subproblems.

6.3. Summary

The speedups we achieved (compared with the running time of the parallel program on a single processor) are quite encouraging.

The execution times on one processor are obviously far greater than a custom-coded C implementation. Compared to the code produced by the Haskell compiler **GHC** (V.4.04), we are slower by a factor of about 5 for the N queens problem and 1.5 for polynomial multiplication. There is still a high potential of optimization in our compiler, especially concerning memory management and the representation of data structures.

7. Related Work

There have been many approaches to skeletal and functional programming. We concentrate here on those which have been most successful and/or have had significant influence on our work.

Two functional languages have been designed explicitly with parallelism in mind; both make use of parallel vector operations. The focus of the language *Sisal* [33] is on numerical computations, using loops on arrays. For some programs, its performance is superior to FORTRAN. Sisal is compiled to a data flow graph language. In contrast to Sisal, the focus of the language *Nesl* [6] is on recursive programs using nested sequences. Nesl is compiled to an intermediate language, which uses parallel vector operations.

Both Sisal and Nesl do not use skeletons and do not permit higher-order functions.

The language GpH [34] is an extension of Haskell with a new primitive `par`, to be used together with the Haskell primitive `seq` to prescribe where values are supposed to be computed in sequence or in parallel. However, in contrast to $\mathcal{HDC}$, aside from a restriction of the evaluation order via `seq`, no schedule and allocation can be defined in GpH. Instead, parallel processes are distributed dynamically. GpH puts no restriction on the use of higher-order functions in Haskell. The user can define new skeletons, using evaluation strategies specified with `seq` and `par`.

There is another difference to $\mathcal{HDC}$: in order to preserve laziness, the input data for a process is only sent partially – if evaluation proceeds, further data must be requested. However, due to its treatment of higher-order functions, GpH is the language which is most similar to $\mathcal{HDC}$.

The idea to use a skeleton for $\mathcal{DC}$ was introduced by Cole [10]. The group of Darlington at Imperial College has published a collection of functional skeletons for parallel programming [12].

P3L [4] is an imperative language, which uses skeletons at the top level but does not support functions as run-time parameters of the skeleton. David Busvine and Tore Bratvold presented in their Ph.D. theses [7, 9] extensions of ML with skeletons, but their use of higher-order functions is very restricted.

The language Eden [8, 15] facilitates the definition of skeletons on top of Concurrent Haskell. Eden imposes no restriction on higher-order functions. Eden differs from $\mathcal{HDC}$ in that skeletons have more restricted signatures and, therefore, cannot be used as generally; skeleton instances have to be wired together using channels.

8. Conclusions

$\mathcal{HDC}$ is not meant to be a general-purpose language but a platform for experiments with parallel $\mathcal{DC}$. As such, it is much leaner than its superset language Haskell. Still, the development of a compiler has been a substantial undertaking.

We have been following a different route than other Haskell implementers: while their compilers are based in the concept of graph reduction, ours generates loop code. In the process, we eliminate higher-orderness and polymorphy. In the pursuit of parallelism, we deemphasize the list constructor cons and emphasize list comprehension and indexing.

The functional programming style has been a choice of convenience for us. We hope that the lessons of the $\mathcal{HDC}$ project will be interesting not only for functional parallel programmers but for the entire parallel computing community.

We are optimistic that the use of skeletons enables a scaling of the problem size with increasing number of processors and only a small increase in execution time.

9. Acknowledgements

We are especially indepted to Peter Faber for helping us with the experiments on the Cenju-4 of the GMD. We also thank the former $\mathcal{HDC}$ team members: Robert Günz, Jan Laitenberger, and Christian Schaller. Thanks to Sergei Gorlatch and John O'Donnell for many fruitful discussions. The idea of skeleton `sinGen` stems from the work by Lisper [29].

This work has been supported by the DFG under project RecuR2 and by the DAAD under an ARC exchange project with Great Britain.

References

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

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Series in Computer Science and Engineering. Benjamin/Cummings, 1989.

[4] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^{3}L: A structured high level programming language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.

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

[6] G. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-93-129, Department of Computer Science, Carnegie-Mellon University, 1992.

[7] T. A. Bratvold. *Skeleton-Based Parallelisation of Functional Programs*. PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, 1994.

[8] S. Breitinger, U. Klusik, and R. Loogen. An Implementation of Eden on top of Concurrent Haskell. In W. Kluge, editor, *Implementation of Functional Languages (IFL'96)*, LNCS 1268, pages 142–161. Springer-Verlag, 1997.

[9] D. J. Busvine. *Detecting Parallel Structures in Functional Programs*. PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, 1993.

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

[11] J.-F. Collard. Automatic parallelization of `while`-loops using speculative execution. *Int. J. Parallel Programming*, 23(2):191–219, 1995.

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

[13] L. P. Deutsch and D. G. Bobrow. An efficient, incremental, automatic garbage collector. *Comm. ACM*, 19(9):522–526, Sept. 1976.

[14] T. Elrad and N. Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, 2(2):155–173, 1982.

[15] L. A. Galán, C. Pareja, and R. Peña. Functional skeletons generate process topologies in Eden. In H. Kuchen and S. D. Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs (PLILP'96)*, LNCS 1140, pages 289–303. Springer-Verlag, 1996.

[16] M. Griebl and C. Lengauer. On the space-time mapping of WHILE-loops. *Parallel Processing Letters*, 4(3):221–232, Sept. 1994.

[17] C. A. Herrmann, J. Laitenberger, C. Lengauer, and C. Schaller. Static parallelization of functional programs: Elimination of higher-order functions & optimized inlining. In P. Amestoy, P. Berger, M. Daydé, I. Duff, V. Frayssé, L. Giraud, and D. Ruiz, editors, *Euro-Par'99: Parallel Processing*, LNCS 1685, pages 930–934. Springer-Verlag, 1999.

[18] 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.

[19] 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, Mar. 1997.

[20] C. A. Herrmann and C. Lengauer. Size inference of nested lists in functional programs. In K. Hammond, T. Davie, and C. Clack, editors, *Proc. 10th Int. Workshop on the Implementation of Functional Languages (IFL'98)*, pages 346–364. Department of Computer Science, University College London, 1998.

[21] C. A. Herrmann and C. Lengauer. Parallelization of divide-and-conquer by translation to nested loops. *J. Functional Programming*, 9(3):279–310, May 1999.

[22] C. A. Herrmann, C. Lengauer, R. Günz, J. Laitenberger, and C. Schaller. A compiler for $\mathcal{HDC}$. Technical Report MIP-9907, Fakultät für Mathematik und Informatik, Universität Passau, May 1999.

[23] P. Hudak and J. H. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5):T1–T53, May 1992.

[24] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In J.-P. Jouannaud, editor, *Proc. Conf. on Functional Programming Languages and Computer Architecture (FPCA'85)*, LNCS 201. Springer-Verlag, 1985.

[25] O. Kaser, C. R. Ramakrishnan, and S. Pawagi. A new approach to inlining. Technical Report 92/06, Computer Science Department, SUNY at Stony Brook, 1992.

[26] O. Kaser, C. R. Ramakrishnan, and S. Pawagi. On the conversion of indirect to direct recursion. *ACM Letters on Programming Languages and Systems*, 2(1–4):151–164, 1993.

[27] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, LNCS 715, pages 398–416. Springer-Verlag, 1993.

[28] C. Lengauer, S. Gorlatch, and C. A. Herrmann. The static parallelization of loops and recursions. *J. Supercomputing*, 11(4):333–353, Dec. 1997.

[29] B. Lisper. Data parallelism and functional programming. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, LNCS 1132, pages 220–251. Springer-Verlag, 1996.

[30] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. on Programming Languages and Systems*, 4(2):258–282, Apr. 1982.

[31] P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.

[32] M. J. Quinn. *Parallel Computing*. McGraw-Hill, 1994.

[33] S. K. Skedzielewski. Sisal. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 4. ACM Press, 1991.

[34] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *J. Functional Programming*, 8(1):23–60, Jan. 1998.

[35] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1995.