Parallel Processing Letters,

$\mathcal{HDC}$: A Higher-Order Language for Divide-and-Conquer

CHRISTOPH A. HERRMANN and CHRISTIAN LENGAUER
Fakultät für Mathematik und Informatik, Universität Passau
D-94030 Passau, Germany
http://www.fmi.uni-passau.de/cl/
{herrmann,lengauer}@fmi.uni-passau.de

Received (received date)
Revised (revised date)
Communicated by Yves Robert

ABSTRACT

We propose the higher-order functional style for the parallel programming of algorithms. The functional language $\mathcal{HDC}$, a subset of the language Haskell, facilitates the clean integration of skeletons into a functional program. Skeletons are predefined programming schemata with an efficient parallel implementation. We report on our compiler, which translates $\mathcal{HDC}$ programs into C+MPI, especially on the design decisions we made. Two small examples, the n queens problem and Karatsuba's polynomial multiplication, are presented to demonstrate the programming comfort and the speedup one can obtain.

Keywords: divide-and-conquer, functional program, Haskell, parallelization, skeleton

1. Introduction

Several decades after the benefits of structure were discovered for sequential programming, parallel programming still suffers from the lack of high-level language support. Currently, the most popular style of parallel programming is with an imperative language like Fortran or C and a communication library like MPI.

Numerous efforts have been made to raise the level of abstraction in parallel programming with Fortran or C in order to increase programming comfort, safety and productivity. MPI offers so-called *collective operations*, which are more abstract than the primitives *send* and *receive* for point-to-point communication – an example is a restricted form of reduction – but memory allocation remains the responsibility of the programmer. The lack of polymorphism makes a general implementation of powerful forms of reduction and scan [11] difficult in Fortran or C.

Our prime concerns are the structure, generality and portability of parallel programs. The *structure* should be imposed by the problem, not by the limitations of the mainstream technology – be it in software or hardware. One dominant source of *generality* is polymorphism: many parallel solutions are independent of the element type of the distributed data structure. *Portability* can only be achieved if the parallel solution is not geared towards a specific parallel architecture or processor topology.

We believe that the appropriate abstraction mechanism for achieving these goals is the higher-order functional programming style. *Higher-order functions* permit functions as arguments or result. For some higher-order functions, so-called *skeletons*, predefined efficient parallel implementations are supplied.

We have chosen the functional language Haskell [19] as a basis. The source programs are currently restricted to a subset of Haskell, which we call $\mathcal{HDC}$. The only intended difference to Haskell is that we must impose eager evaluation –at least for skeleton applications– in order to adhere to the static execution schema specified in some skeleton implementations. $\mathcal{HDC}$ stands for *higher-order divide-and-conquer* and was originally developed for the parallelization of divide-and-conquer ($\mathcal{DC}$) recursions, but is appropriate for programming with skeletons of any kind. Our focus here is still the parallelization of $\mathcal{DC}$.

We distinguish between the definition and the implementation of a skeleton. The definition is purely functional, denoted in Haskell, and does not specify operational aspects. The implementation is a parametrized function in the target language (C), contains side-effecting communications (MPI) and memory management, and is linked together with compiled $\mathcal{HDC}$ functions written by the application programmer. The use of the same language (Haskell) for both the definition and the application of skeletons distinguishes $\mathcal{HDC}$ from coordination languages like SCL [8] or P3L [3] and gives us a substantial amount of programming comfort. Any user-defined function can be turned into a skeleton, transparently to the programmer, by providing a dedicated implementation for it.

Our main goals in designing a compiler for the language $\mathcal{HDC}$ have been to investigate the terrain of paradigms, especially $\mathcal{DC}$, classify it by different skeletons which span different ranges of parallel implementations, and provide an environment for the exploration of different algorithms for parallelization and for prototyping parallel programs.

The central property of a skeleton we make use of is that it specifies a class of algorithms with common properties, specified by its body, while allowing for variation in properties via its arguments which, in a call, are instantiated with algorithm-specific, *customizing* functions. Our skeletons for $\mathcal{DC}$ are arranged in a specialization hierarchy, which is described in detail elsewhere [12]. Starting with a skeleton which defines our notion of general $\mathcal{DC}$ (dcA), successive skeletons are derived by specialization: by restricting the application domain, we enable a more efficient implementation. We impose the restrictions of fixed recursion depth (dcB), constant division degree (dcC), multiple block recursion (dcD), elementwise operations (dcE) and correspondent communication (dcF).

Tab. 1 illustrates that $\mathcal{DC}$ has a wide spectrum of applications and can have various characteristics, which have repercussions for an efficient parallel implementation. For an input size of n, we list the sequential execution time, the depth of the recursion, the degree of the problem division (i.e., the number of subproblem instances generated per problem instance) and the fact whether the determination

Table 1. Some divide-and-conquer problems.

algorithm	seq. exec. time	rec. depth	degree	sched.	alloc.
mergesort	$\Theta(n \cdot \log n)$	$\log_2 n$	2	stat.	stat.
2D convex hull (of presorted points)	$\Theta(n \cdot \log n)$	$\log_2 n$	2	stat.	stat.
$\sqrt{n} \times \sqrt{n}$ matrix multiplication	$\Theta(n^{1.40..})$	$\log_4 n$	7	stat.	stat.
triangular matrix inversion	$\Theta(n^{1.40..})$	$\log_4 n$	2	stat.	stat.
2D component labeling	$\Theta(n^{1.5})$	$\log_4 n$	4	stat.	stat.
Karatsuba multiplication	$\Theta(n^{1.58..})$	$\log_2 n$	3	stat.	stat.
quicksort	$O(n^2)$	$\leq n$	2	dyn.	dyn.
maximum independent set	$O(2^n)$	$\leq n$	2	dyn.	dyn.
tautology check	$O(2^n)$	$\leq n$	2	dyn.	dyn.
frequent set	$O(2^n)$	$\leq n$	2	dyn.	dyn.
n **queens**	$O(n!)$	n	$\leq n$	stat.	dyn.

of the schedule (the distribution of the operations across time steps) and the allocation (the distribution of the operations across processors) can be done at compile time (statically) or must be left to run time (dynamically). We have used the two examples stated in bold font in experiments which we comment on later.

2. Examples

We present our examples in the language Haskell with a beautified syntax.

2.1. Application of the most general skeleton: the n queens problem

Skeleton dcA (dc0 in [13,14]) specifies our notion of general $\mathcal{DC}$:

```
dcA ∈ (α→𝔹)→(α→β)→(α→[α])→(α→[β]→β)→α→β
dcA istrivial basic divide combine = r
  where r x = if istrivial x then basic x
                             else combine x (map r (divide x))
```

The skeleton takes the customizing functions *istrivial*, *basic*, *divide* and *combine*. A call of dcA with these four arguments returns the recursive function r which is applied to the input data x. If the problem is trivial (i.e., solvable without dividing it into subproblems), it is solved by function *basic*. Otherwise it is divided into a list of subproblems. map applies r to every subproblem, yielding a list of subproblem solutions which is used to compute the solution of the problem. We took this skeleton from [16] and [20].

Note that its definition, based on map, strictly enforces the independence of the subproblems. Consequently, e.g., the branch-and-bound paradigm does not match dcA. See the dissertation of the first author [12] for a discussion of this issue.

Let us demonstrate the use of dcA on the n queens problem [1]. We compute all possible solutions for placing n queens on an $n \times n$ board using a decision tree, such that no two queens are on the same row, the same column or the same diagonal. There may be sophisticated combinatorial ways of simplifying this problem, but we use it as a representative for exhaustive search problems. Our only heuristic is to try to recognize conflicts as early as possible. Here is our $\mathcal{HDC}$ program, which calls dcA:

$$
\begin{array}{l}
\mathsf{queens} \in \mathbb{N} \to [[\mathbb{N}]] \\
\mathsf{queens}\ n = \mathsf{dcA}\ \mathit{istrivial}\ \mathit{basic}\ \mathit{divide}\ \mathit{combine}\ ([\,], [0\,..\,n-1])\ \underline{\texttt{where}} \\
\quad \mathit{istrivial}\ (\_, \mathit{remain}) = \mathsf{null}\ \mathit{remain} \\
\quad \mathit{basic}\ (\mathit{placed}, \_) = [\mathit{placed}] \\
\quad \mathit{divide}\ (\mathit{placed}, \mathit{remain}) = \\
\qquad \underline{\texttt{let}}\ \mathit{diagonal\_attack}\ i = \mathsf{or}\ [\ (\mathsf{length}\ \mathit{placed} - j) = \mathsf{abs}\ (i - \mathit{placed}\ !!\ j) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid j \leftarrow [0\,..\,\mathsf{length}\ \mathit{placed} - 1]\ ] \\
\qquad \underline{\texttt{in}}\ [\ (\mathit{placed} +\!\!+ [i], \mathsf{filter}\ (\neq i)\ \mathit{remain}) \\
\qquad\qquad \mid i \leftarrow \mathit{remain},\ \mathsf{not}\ (\mathit{diagonal\_attack}\ i)\ ] \\
\quad \mathit{combine}\ \_ = \mathsf{concat}
\end{array}
$$

The algorithm starts with an empty board and adds recursively, row by row, a column position at which a new queen can be placed. The number of subproblems for a placement already made is determined by the number of possible placements for the next row.

The input data for dcA is a pair of the list of placements made and the column positions which have not yet been allocated. The divide function only has to check for an attack of two queens on the same diagonal, since the rows are distinguished by the position in the list of placed queens and the columns are distinguished by the distinct elements in the list of remaining column positions. The number of subproblems can also be 0 or 1. The combine function collects the placements of all subproblems.

The performance of program queens, with powers of 2 as the numbers of processors, is charted in Fig. 1. To investigate the potential for a massive parallelization, we used a transputer network with 1024 processors, the Parsytec GCel-1024. Since most work performed for communication is done by the processor itself (marshaling), we expect that similar speedup results could be achieved with modern machines, provided that a similarly large number of processors is available.

The parallelization is based on the independence inside the map in dcA, applied recursively. This is more flexible than our current dedicated dcA implementation concerning the number of processors used. Applications of map produced by desugared list comprehensions in the divide function were excluded from parallelization because the incurred overhead is larger than the gain.

Our straightforward implementation of the skeletons map and dcA is to employ a hierarchical management of the processors. Each skeleton instance carries information about a subset of the processors which it can use exclusively. This subset is then divided equally among the subinstances. The control structure is organized as follows. At the beginning, all processors form a single block. In a parallel computation, each block assigned to a set of processors –let us call it the *superblock*– is divided into a number of *subblocks* and the *master* processor of the superblock sends a task to the masters of the subblocks. When the task a subblock

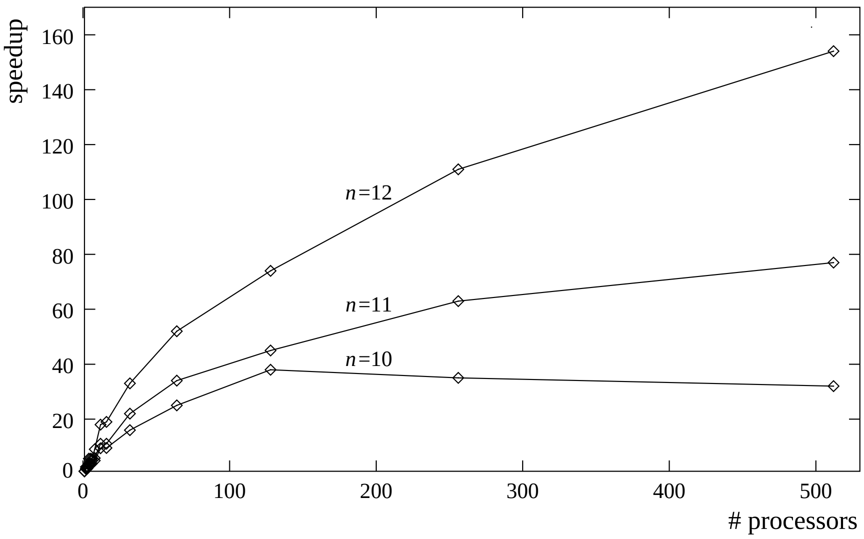

Fig. 1. Speedup for program queens.

is assigned to is terminating, the subblock's master sends an according signal to the master of the superblock. During the time in which each master processes the problem instance assigned to it, no control message is sent from any processor of the master's block across the block's border. Other messages, e.g., for accesses to globally distributed data aggregates, are not restricted. We call this the principle of *control-closed blocks*, in analogy to the principle of *communication-closed layers* [9]. In our current implementation, all input data of a task is passed along with the signal of task initiation and all output data is passed back with the report on the task's completion.

The subproblems of a problem are distributed evenly among the available processors, e.g., if 32 processors are assigned to a problem instance that has two subproblems, each of the subproblem instances can use 16 processors exclusively. This can lead to load imbalance if one subproblem can be solved trivially and the other requires further division. Thus, we have developed a purely iterative, i.e., not recursive, execution schema of dcA, called itA [12], which supports load rebalancing at every level of the $\mathcal{DC}$ call tree. In our example, this would mean that the subproblem which requires further division can use 32 processors again. However, since the MPI implementation of this schema is complex and still under development, no experimental results are available yet.

Our aim is to have control over the time and space consumption to make parallelization accessible for automatic optimization, similarly to the tradition of loop parallelization [17]. Note that, although the actual number of processors assigned to a particular task depends on run-time parameters, the structure of the parallel execution is determined by the skeletons at compile time. If this is not required, one might consider using an environment with a powerful run-time load-balancing mechanism, like Glasgow parallel Haskell [21] or Cilk [10].

An alternative treatment of the n queens problem was reported at CPC 2000 by Cohen [6]. His recursive procedure Queens contains a doubly nested loop which computes imperatively on an array. The static parallelization of Queens requires an automatic dependence analysis which determines the most recent update of an array cell. In contrast, our $\mathcal{HDC}$ program requires no such dependence analysis.

2.2. Application of our most specific skeleton: the Karatsuba multiplication

The Karatsuba multiplication of polynomials fits into the most specialized skeleton in our $\mathcal{DC}$ hierarchy, dcF [12] (dc4io in [13,14]). Its call tree is balanced and it requires elementwise divide and combine operations on subblocks of data. The definition of dcF is quite involved; we present only the signature:

$$\mathsf{dcF}\ \textit{probdegree indegree outdegree basic divide combine levels xs} = ...$$

The parameters of dcF have the following meaning:

- *probdegree* $\in \mathbb{N}$: the degree of problem division, i.e., the number of subproblems which are generated for each problem not trivially solved; this degree remains constant in a recursive call of dcF, in contrast to dcA.

- *indegree* $\in \mathbb{N}$: the degree of division of input data; specifies in how many blocks the input data is to be divided.

- *outdegree* $\in \mathbb{N}$: the degree of composition of output data; specifies of how many blocks the output data is to be composed.

- *basic* $\in (\alpha \rightarrow \beta)$: the function to be applied in the trivial case.

- *divide* $\in (\mathbb{N} \rightarrow [\alpha] \rightarrow \alpha)$: computes any element i of subproblem sp; it takes sp $(0 \leq sp < \textit{probdegree})$ and a list of length *indegree* which carries at position b element i of input block b.

- *combine* $\in (\mathbb{N} \rightarrow [\beta] \rightarrow \beta)$: computes any element i of output block ob; it takes ob $(0 \leq ob < \textit{outdegree})$ and a list of length *probdegree* which carries at position sp element i of the solution of subproblem sp.

- *levels* $\in \mathbb{N}$: the number of recursive levels into which the $\mathcal{DC}$ tree unfolds. This is used instead of the predicate of dcA. In theory, *levels* reflects the number of levels upto the trivial cases. In practice, the user can make two other choices: control granularity of parallelism or solve small problem instances, which are not the basic case, by an algorithm tailored for small sizes.

- *xs* $\in [\alpha]$: the input data; a list to which the division into blocks is applied; likewise, the output data is of type $[\beta]$.

dcF works well for vector and matrix algorithms like the fast Fourier transform, bitonic merge, polynomial multiplication and matrix multiplication. Here, it is applied to the Karatsuba multiplication of polynomials [2].

We represent each polynomial of *size* n, say $\sum_{i=0}^{n-1} c_i X^i$, by the list $[c_{n-1}, ..., c_0]$. Our function karatsuba is restricted to the multiplication of two polynomials of size $n = 2^m$. If $m > 0$, the problem is reduced to three multiplications of pairs of polynomials, each of size 2^{m-1}. Thus, the depth of recursion is m and the number of basic cases created is 3^m. This number determines the complexity, which is $\theta(3^{\log_2 n}) = \theta(n^{1.58..})$.

If the size of both polynomials is not equal, the multiplication can be partitioned into blocks. If the size is not a power of 2, the polynomials can be extended with coefficients carrying the value 0. This causes a small amount of overhead, which is justified by the gain in complexity compared to the trivial algorithm.

Polynomial multiplication can be extended to the multiplication of large integers, if X is viewed as the radix of the number system and the coefficients of the result are normalized to the range from 0 to $X-1$ by carry propagation.

```
karatsuba ∈ ([ℤ]×[ℤ]) → [ℤ]
karatsuba (a, b) =
  let  basic (x, y) = (0, x∗y)
       divide sp [(xh, yh), (xl, yl)] = case sp of
                                          0 → (xh, yh)
                                          1 → (xl, yl)
                                          2 → (xh+xl, yh+yl)
       combine ob [(hh, hl), (lh, ll), (mh, ml)] = if ob=0 then (hh, lh+ml−hl−ll)
                                                         else (hl+mh−hh−lh, ll)
       n = ilog2 (length a)
       z =  dcF 3 2 2 basic divide combine n (zip a b)
  in map fst z ++ map snd z
```

The experimental results were obtained after manually replacing the standard pair notation by a new datatype of unboxed pairs and rewriting the pattern matching. The respective program can be found elsewhere [12, App. D.1.3]. We plan to implement both transformations in the compiler. They achieve a gain in run time by a factor of about 3, since they reduce the extent of dynamic memory management substantially.

Function ilog2 computes the value of n, the depth of recursion. Since dcF requires a single input for recursive division, the arguments a and b are zipped together to a list of pairs. The output of dcF is the zip of the higher and lower coefficients of the result which are extracted by map fst and map snd. The product of two polynomials of size 2^m is of size 2^{m+1}.

Since the customizing functions are restricted to elementwise operations on corresponding elements of balanced blocks, the user only has to specify the operations on single elements. Function *divide* takes two elements (corresponding elements of two blocks) and produces one of three elements (corresponding elements of three subproblems).

Fig. 2 demonstrates that a satisfactory speedup can be achieved if the operand size (line label in the figure) increases with the number of processors. We have also depicted the relative execution times if dcA was used instead of dcF.

Our implementation of dcF was developed in two steps. In the first, we derived an abstract data-parallel loop program called itF [12] by imposing the same restrictions on itA that we imposed on dcA to obtain dcF. This was done by equational transformations in the language Haskell. In the second, we implemented this data-parallel program, which is much simpler than itA, in C+MPI, replacing non-local accesses by communications. The schema of communications is remotely related to schemata known from butterfly networks. Since our current run-time system does not yet support skeleton calls with distributed input or output data, the data has to be scattered at the time of call of dcF and gathered into a single processor at the time of return.

Due to the structure of the computation, the number of processors should be a power of 3, like in Fig. 2, to avoid load imbalance. Although additional processors could be used to parallelize also the customizing functions, this does not pay off here due to the incurred overhead, as we obtained by more extensive experiments.

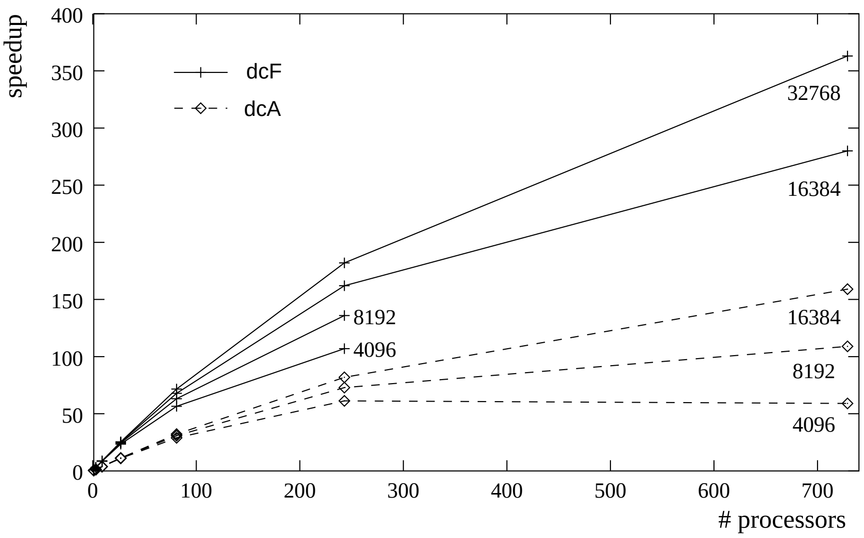

Fig. 2. Speedup for program karatsuba.

3. Elimination of functional arguments

A major challenge in the compilation of higher-order functions is to organize the access to the values of the free variables in the functional arguments. For this purpose, we use an algorithm for transforming a closed, higher-order and well-typed functional program into an equivalent first-order program [4]. The idea of the algorithm is to replace each functional argument by an encoding of its closure. A closure contains a function identifier and the values of the free variables in the functional argument.

The elimination algorithm proceeds as follows:

- Each variable of a function type changes its type because, after the transformation, it carries the appropriate encoding.
- Each functional argument at the caller's side is replaced by an element 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 functional argument. These values are taken from the context of the caller.
- Each application of a variable representing a function is replaced by a call of an *apply* function constructed for the respective function type. The first argument of the apply function is the closure encoding, the following arguments are the arguments of the encoded function. The apply function extracts the original function and applies it in an environment which provides the current values of the free variables in the closure. These values can again contain closures.

A detailed example of an elimination can be found in the $\mathcal{HDC}$ report [15].

4. Integration of skeleton implementations

The $\mathcal{HDC}$ compiler offers a special, very flexible mechanism for the integration of custom-implemented skeletons. For each skeleton, the implementer delivers a Haskell function 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 the implementation function will be just a Haskell string of C target code, but it can also prescribe decisions based on type and size information provided by the compiler. E.g., the implementation of dcF might determine the granularity of parallelism, based on the recursion depth parameter and the availability of processors. Note that the C target code is monomorphic and first-order. This applies also to the implementation of a skeleton: polymorphic functions must be instantiated for every unboxed argument type currently used, and higher-order functions must be represented by an encoding of their closures [15].

5. The $\mathcal{HDC}$ compiler

The compiler takes an $\mathcal{HDC}$ source program and delivers a target program in C+MPI, which is linked together with a customized run-time library. The compiler is written in Haskell. It traverses a number of phases, as follows:

- Scanning and parsing. We use the parser generator *happy* which takes an annotated grammar in the style of *yacc*. The semantic actions produce a syntax tree as an element of an algebraic data type in Haskell.
- Simplification of complicated language constructs like list comprehensions, exploiting opportunities for parallelization [12,14,15].
- Type checking in the Hindley-Milner system [7], using the Martelli-Montanari rules for unification [18].
- Elimination of functional arguments (higher-order elimination) according to Sect. 3.
- Generation of intermediate code. For each function, a directed acyclic graph (DAG) is generated to exploit common subexpressions [14,15].
- DAG code optimization. Here, inline expansion, rule-based DAG optimizations and size inference of data structures are applied iteratively [14,15].
- Abstract code generation. This phase analyzes alternation in functions and introduces conditional and unconditional jumps [15].
- Target code generation.
- Generation of the skeleton implementations in C, according to the types used in the program. The code for the skeletons contains the calls to the MPI library.

6. The $\mathcal{HDC}$ run-time system

The run-time system deals mainly with memory management and marshaling. Every processor manages its own local heap. Simple data types have a plain, unboxed representation. Lists, tuples and algebraic data types are represented by a pointer to a data structure which contains a descriptor and a record or array with the data elements. Each such structure has a reference counter which is incremented when the data structure is used as part of another structure or a copy of its pointer is made in the program. It is decremented when a data structure, which used it, is deleted or at the earliest program point for which the flow analysis reveals that a copy of its pointer is no longer required. If the counter of the data structure reaches the value 0, its memory is released. The implementer of a skeleton has to follow these conventions, e.g., to decrement the reference count after the last use of an argument of the skeleton.

The run-time system also performs marshaling and unmarshaling of data structures, because MPI cannot communicate dynamically linked data structures. In marshaling, such a data structure is encoded as a byte array which can be communicated to another processor, where it is transformed again into a linked structure.

7. Conclusions

The $\mathcal{HDC}$ compiler is still under development. Its purpose is to provide an open platform for prototyping, program analysis and experimental implementation – not a production environment of competitive code for any particular purpose.

We have managed to raise the level of abstraction to the purely functional style of programming, while preserving as much control as required at the implementation side. This has been achieved by the use of skeletons. The skeleton, as a higher-order function, fits perfectly in the functional source program. Also, the elimination of higher-order functions makes it possible to write the skeleton implementations in an imperative language and thus, to control computations and communications precisely.

We have noted two main reasons for a loss of performance: (1) load imbalance and (2) parallelization where the overhead of communication is larger than the amount of computation time gained. We observed both factors during our experiments. Future research will have to deal with a computer-aided analysis of which parts should be parallelized.

$\mathcal{HDC}$ supports the parallelization of efficient algorithms, e.g., the Karatsuba polynomial multiplication, in contrast to the naive multiplication which is easy to parallelize. This is the basis for good absolute results.

Most of the problems we plan to parallelize are strict. Thus, the eager evaluation we have implemented in contrast to the laziness of Haskell does not lead to a loss of performance.

Probably the most well known project, which fostered the use of a functional language for high performance programming, is Sisal [5]. It competed in performance directly with Fortran. In order to win, it left functional pearls like polymorphism and higher-order functions aside. With $\mathcal{HDC}$, we do not seek the competition on speedup with current techniques of high-performance programming quite as strongly. Rather we seek to win on the issues of flexibility and portability. Thus, we do not make the compromises Sisal made. We are hoping, though, that, by offering flexibility on the implementation side, $\mathcal{HDC}$ will enable programmers to develop applications with a competitive ratio of execution performance to development cost.

Acknowledgements

This work has been supported by a four-year grant from the DFG under project name RecuR2. We thank the Paderborn Center for Parallel Computing for access to the GCel-1024. The anonymous referees deserve sincere thanks for their valuable comments.

References

[1] B. Abramson and M. Yung. Divide and conquer under global constraints: A solution to the n-queens problem. *J. Parallel and Distributed Computing*, 6:649–662, 1989.

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

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

[5] D. Cann. Retire Fortran? A debate rekindled. *Comm. ACM*, 35(8):81–89, Aug. 1992.

[6] A. Cohen. *Program Analysis and Transformation: From the Polytope Model to Formal Languages.* PhD thesis, Laboratoire PRiSM, Université de Versailles, Dec. 1999.

[7] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symp. on Principles of Programming Languages (POPL'82)*, pages 207–212. ACM Press, 1982.

[8] J. Darlington, Y. Guo, H.W. To and J. Yang. Parallel skeletons for structured composition. In *Proc. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'95)*, pages 19–28. ACM Press, 1995.

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

[10] M. Frigo, C.E. Leiserson and K.H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices*, 33(5):212–223, May 1998. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'98).*

[11] S. Gorlatch, C. Wedler and C. Lengauer. Optimization rules for programming with collective operations. In *Proc. 13th Int. Parallel Processing Symp. & 10th Symp. on Parallel and Distributed Processing (IPPS/SPDP'99)*, pages 492–499. IEEE Computer Society Press, 1999.

[12] C.A. Herrmann. *The Skeleton-Based Parallelization of Divide-and-Conquer Recursions.* PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, 2000. In press.

[13] 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.

[14] C.A. Herrmann and C. Lengauer. The $\mathcal{HDC}$ compiler project. In A. Darte, G.-A. Silber and Y. Robert, editors, *Proc. Eighth Int. Workshop on Compilers for Parallel Computers (CPC 2000)*, pages 239–253. LIP, ENS Lyon, 2000.

[15] 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.

[16] P. Kelly. *Functional Programming for Loosely-Coupled Multiprocessors.* Pitman, 1989.

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

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

[19] S.L. Peyton Jones and J. Hughes, editors. Haskell 98: A non-strict, purely functional language. Technical report, http://haskell.org, 1999.

[20] F.A. Rabhi. Exploiting parallelism in functional languages: A "paradigm-oriented" approach. In J.R. Davy and P. Dew, editors, *Abstract Machine Models for Highly Parallel Computers*, pages 118–139. Oxford University Press, 1995.

[21] 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.