Parallel Processing Letters, Vol. 15, No. 3 (2005) 305–320

World Scientific
www.worldscientific.com

GENERATING MESSAGE-PASSING PROGRAMS FROM ABSTRACT SPECIFICATIONS BY PARTIAL EVALUATION

CHRISTOPH A. HERRMANN
FMI, University of Passau, Germany
http://www.fmi.uni-passau.de/~herrmann

Received 16 September 2004
Revised 31 October 2004
Communicated by Sergei Gorlatch

ABSTRACT

This paper demonstrates how parallel programs with message-passing can be generated from abstract specifications embedded in the functional language MetaOCaml. The functional style permits to design parallel programs with a high degree of parameterization, so-called skeletons. Programmers who are unexperienced in parallelism can take such skeletons for a simple and safe generation of parallel applications. Since MetaOCaml also has efficient imperative features and an MPI interface, the entire program can be written in one language, without the need to use a language interface restricting the set of data objects which could be exchanged.

The semantics of abstract specifications is expressed by an interpreter written in MetaOCaml. A cost model is defined by abstract interpretation of the specification. Partial evaluation of the interpreter with a specification, a feature which MetaOCaml provides, yields a parallel program. The partial evaluation process takes time on each MPI process directly before the execution of the application program, exploiting knowledge of the number of processes, the current process identifier and the communication structure. Our example is the specification of a divide-and-conquer skeleton which is used to compute the multiplication of multi-digit numbers using Karatsuba's algorithm.

Keywords: cost model, meta-programming, parallelism, partial evaluation, skeletons

1. Introduction

Message-passing clusters are quite commonly used today, but the style of programming them explicitly for parallel processing is still not as effective as it could be: application programmers have to design the task structure and the communications by hand, except if they use skeletons.

For us, a skeleton is an abstract algorithmic programming pattern at the side of the source program which has an efficient parallel implementation at the side of the target program. Collective operations in the message-passing library MPI [14], like scans or reductions, are modest examples of skeletons; divide-and-conquer, branch-and-bound and the like are more powerful examples.

Implementing a skeleton requires much effort if flexibility, type-safety and efficiency are to be established together. Therefore, we propose an automatic generation of skeletons from abstract specifications by means of meta-programming.

1.1. Program Generation from Abstract Specifications

The objective to design safe and predictable parallel programs motivates a high level of program construction. An abstract specification is most suitable to express the parallelism in the program without restrictions imposed by the particular programming language or target architecture. Even more, a specification can be declarative in some aspects, i.e., prescribe constraints to be established. From the specification, an expert could derive a parallel implementation by a sequence of semantics-preserving, performance-directed program transformation steps. The result can be, e.g., a skeleton in C with MPI calls.

A standard compiler will very likely not be able to generate efficient target programs from such abstract specifications directly. Inspired by the approach of domain-specific program generation [11], we are looking for a semi-automatic compilation concept to replace the tedious manual derivation process. The contributions of this paper are a step towards this goal, in particular:

1. A language which separates the parallel from the general-purpose aspects.
2. Automatic generation of placement and communications by partial evaluation.
3. Practical example: implementation of a divide-and-conquer skeleton in terms of the specification language, and application to long number multiplication.

This paper does not deal with program transformations; they are coming into play as soon as a program is constructed by combining multiple skeletons and could be implemented by a rewrite of the specification, e.g., introducing performance-directed case distinctions. The appropriate case is then selected by the partial evaluation, when the values of the structural parameters are known.

1.2. Interpretation + Partial Evaluation = Compilation

Interpretation is a well-known technique for embedding a domain-specific language in a general-purpose programming language, and writing an interpreter is easier than writing a compiler. From a commercial point of view, the aim is to reuse existing language implementation technology, the one the interpreter is implemented in, to save development time and cost.

The run-time overhead of interpretation, including program analysis, can be eliminated by a partial application of the interpreter with the source program: the residual program which still has to take the input data is in a kind of compiled form. The impact of partial application on compiler generator technology is a seminal contribution by Futamura [5].

Assume that an interpreter for a language EL (embedded language) is written in a language H (host language); if a partial evaluator specializes the source code of the interpreter with the program in EL to be interpreted, then the result is a stand-alone program in H, which is semantically equivalent to the interpreted program. Then, a compiler for programs in H can translate the specialized interpreter into a

machine program. This way, we can implement a domain-specific compiler just by writing an interpreter!

1.3. The Impact on Parallel Programming

The domain of high-performance computing contains a special challenge. Since automatic parallelization often does not satisfy requirements on performance, programmers are spending much effort to improve performance-critical program parts. On the one hand, a stand-alone domain-specific language (DSL) for high-performance computing could provide a concise formalism to investigate safety properties and apply program transformations. On the other hand, many programmers in high-performance computing prefer languages like Fortran or C to more high-level languages, because of the predictable high speed of sequential code parts and the availability of program libraries. In order to add programming comfort without impairing performance, the DSL presented here is an extension of the general-purpose programming language Objective Caml (OCaml) [19], which can call external functions written in Fortran and C.

A high degree of programming comfort in the domain of parallelism is due to a bird's-eye view on the parallel computation, abstracting, e.g., from considerations of how to implement a set of communications without deadlocks and race conditions. The Fortran extension High-Performance Fortran (HPF) supports this view but has not been very successful yet, probably due to unpredictable performance of the resulting target programs. MPI provides collective operations on groups of processes but, aside from that, the programmer has to spell out case distinctions to implement the spatial and temporal distribution of the parallel computation and take care that they are correct for all processes.

The functional programming style provides mechanisms for abstraction. Unfortunately, the use of the functional style for numerical, array-based computations, which are typical in the high-performance domain, incurs a considerable execution overhead, e.g., due to indirect accesses and a lack of locality of the sequential code parts. For high-performance computing, programming comfort does not justify this overhead. The approach proposed here benefits from the hybrid character of OCaml: the sequential program parts can be imperative, but the specification and the coordination of parallel program parts are functional.

Our DSL provides a bird's-eye view with a hierarchical refinement of the program into parallel parts. Even the parallel programming expert, when implementing a skeleton, need not know implementation details like the number of processes assigned to a particular task. This context information is computed by the partial evaluator on each process and, e.g., used to generate the appropriate communications. Nevertheless, the expert can take control over aspects like the degree of parallelism, but by strategy functions instead of dealing with concrete mapping information directly. The functional programming style provides a simple way to use such functions in a specification.

Interpretation of the program in each process applies the administrative opera-

tions to run a program written in the bird's-eye view on a message-passing parallel machine, and partial evaluation removes the overhead of the interpretation.

Parallel programming still remains a task for a domain expert but, in combination with features provided by the host language OCaml, e.g., recursion and higher-order functions, the expert can design program skeletons in which all parallelism is hidden from the application programmer.

In OCaml [19] we can exchange large arrays with external Fortran or C functions without conversion overhead, call MPI functions for communication and marshal arbitrary program values to send them over a network. The meta-programming extension MetaOCaml [17] can do the necessary partial evaluation. Careful use of side-effects would also permit distributed data structures and parallel I/O for large data sets.

1.4. Overview

The next section discusses related work. Section 3 gives a brief introduction into MetaOCaml. Section 4 presents the specification language for parallelism. In Section 5, we discuss how a parallel programming expert can use this language for the definition of skeletons; our example is a kind of divide-and-conquer skeleton where the branching degree and the recursion depth are available as structural parameters. Section 6 describes the work of the application programmer, in particular, how to define a parallel multiplication of long numbers in terms of the skeleton provided by the parallelism expert. Section 7 presents experimental results for this example and Section 8 concludes.

2. Related Work

Gorlatch [7] stated that parallel programming is still a challenging task, since it relies on the imperative software technology of the past. He proposed the predominant use of collective operations rather than the use of single, point-to-point transmissions. Collective operations contain communication patterns involving many processors, intertwined with computations, and their performance is predictable [2]. The collective operations correspond to the skeletons here, and they contain communication patterns which are specified for the entire process group affected by the skeleton activation.

It is useful to consider the possibilities that domain-specific program generation provides for automatic assistance in the construction of parallel programs. There are several approaches for the specialization of parallel programming support for a particular application area. Two are the parallel linear algebra library ScaLAPACK [4] and the program generator FFTW [6] for the Fast Fourier Transform.

Michaelson and Scaife [13] developed a compiler for PMLS (Parallel ML with Skeletons). Their use of a combination of OCaml, C and MPI as a target language for PMLS encouraged us to investigate the potential of OCaml as a target language.

An important technique to avoid the so-called abstraction penalty of meta-

programming is partial evaluation. Berlin and Weise [1] used this technique to specialize application programs in Scheme. Program specialization in the parallel setting is not restricted to the functional style: Liniker, Beckmann and Kelly [12] developed a run-time specialization system in C++ to increase the performance of parallel numerical programs.

The power of the approach presented here is mainly due to functional meta-programming. Sheard [15] gives a survey of meta-programming, especially addressing the functional style and introducing MetaML, a meta-programming extension of ML. MetaML provides annotations for turning functional values into code, for insertion of code into larger code parts and for execution of constructed code. The language MetaOCaml [17] is a similar extension of the language OCaml [19]. In a tutorial paper [16], its developer Taha discusses the principle of *staging* an interpreter, i.e., preparing it for an evaluation in two phases: the first phase specializes the interpreter for the application program and the second phase runs the specialized program on the application data. The formal precision and simplicity of the staging and the suitability as a programming language motivate the use of MetaOCaml as a formalism for describing the partial evaluation algorithm and for rapid prototyping.

The role of partial evaluation in generative programming has been elaborated by Jones and Glenstrup [9], giving useful hints to avoid pitfalls like nontermination and code explosion.

This paper differs in two independent ways from its predecessor [8]: (1) the communications denoted by side-effects in the object program before have been made a part of the parallel specification language, which changes the specification of the divide-and-conquer skeleton and (2) the example of polynomial multiplication has been extended to long number multiplication to increase the practical relevance.

3. MetaOCaml

MetaOCaml [17], briefly named MOC, is a meta-programming extension of the programming language OCaml [19]. OCaml provides functional, imperative and object-oriented features. Higher-order functions support the reuse of algorithmic schemata by flexible problem-specific customization. Loops and mutable arrays are useful imperative features for high-performance operations, e.g., for linear algebra and graph algorithms.

Meta-Programming is a technique which uses programs (so-called *meta-programs*) to analyze, transform or generate other programs (so-called *object programs*). During the execution of the meta-program, MOC constructs object programs and calls them. This principle is similar to the use of functional arguments but, in contrast to them, object programs are going to be compiled and the programmer has control over their representation.

MOC has three value constructors in addition to OCaml, named **Brackets**, **Escape** and **Run**, and a type constructor named `code`.

- **Brackets** (`.< >.`) enclose an object program part:

```
# let a = .< 2*4 >. ;;
val a : ('a, int) code = .<(2 * 4)>.
```

  The meta-language variable `a` carries the code for computing `2*4`.

- The **Escape** operator (`.~`) inserts an object program part constructed by the enclosing meta-program into a larger part:

```
# let b = .< 9 + .~ a >. ;;
val b : ('a, int) code = .<(9 + (2 * 4))>.
```

- The **Run** operator (`.!`) evaluates a generated object program expression.

```
# let c = .! b ;;
val c : int = 17
```

  In the next section, we will apply `.!` to a function expression, where `.!` compiles the corresponding function, i.e., delivers its bytecode representation.

According to the taxonomy of Sheard [15], MOC has the following properties:

- **Homogeneous meta-programming:** Both meta- and object programs are true MOC programs and, thus, can use the entire set of MOC features.
- **Multiple staging:** An object program can act as a meta-program itself when being executed.
- **Type safety:** The type correctness of the meta-program implies the type correctness of the generated object program. MOC establishes this by a syntactic integration of object program expressions in meta-program expressions and a connection between expressions of both parts by the type system, including typing rules for the three meta-programming constructs.
- **Static scoping:** Each variable name refers to its definition in the smallest enclosing block. The MOC system automatically renames local variables to avoid name clashes in the generation of object program parts.

4. Specification Language for Parallelism

This section deals with the core of the presented method: a domain-specific language for parallelism (DSLP) which is embedded into MOC.

4.1. Syntax

The embedding is expressed by the following algebraic data type `exp` in MOC whose abstract syntax is first given in a simplified form:

```
type exp = Atom of a code-extending function
         | Seq  of a sequence of exp
         | Par  of a parallel composition of exp
         | Comm of a set of communications
```

DSLP is defined by induction on the structure of `exp`. Programs in DSLP can be implemented by specifying a set of four mutually recursive interpretation rules, one for each of the cases `Atom`, `Seq`, `Par` and `Comm`.

Since DSLP is intended to deal with parallelism aspects only, each sequential part of the application program is written as a MOC function and embedded into DSLP with the constructor `Atom`, making use of MOC as a higher-order language. An interpreter would evaluate an embedded part simply by application of the MOC function. Let us have a look at the following equation:

$$\texttt{implementation}^{\texttt{MOC}}\ \texttt{input} = \texttt{interpreter}^{\texttt{MOC}}\ \texttt{specification}^{\texttt{DSLP(MOC)}}\ \texttt{input},$$

where the superscripts indicate the host language and `DSLP(MOC)` is the parameterization of DSLP with MOC functions.

The aim here is to obtain a compiled solution from such an interpreter, i.e., cancel the input data on both sides. Therefore, the embedded MOC parts have to be protected from being evaluated during interpretation, since this evaluation would require the input data. Additionally, this has to be done in a way that the interpretation overhead is really removed and not just hidden in large functional closures. Staging the program is the solution, i.e., the embedded program parts are composed in a purely syntactic form. Then, the interpreter produces a single MOC program by composition from smaller parts, dependent on the semantics of the DSLP program. From the view of the control flow, the interpreter is called when the MOC run operator is applied:

$$\texttt{compiled\_code}^{\texttt{MOC}} = \texttt{.!(interpreter}^{\texttt{MOC}}\ \texttt{specification}^{\texttt{DSLP.<MOC>.}}\texttt{)},$$

where `DSLP.<MOC>.` denotes the parameterization of the specification language with MOC code expressions.

4.2. Intended Meaning

DSLP provides only a small set of constructors which are, in addition to the features of the underlying host language, necessary to define a static parallel task structure. These are the atomic computations, communications and sequential and parallel composition. Alternation, loops and recursion can be implemented by means of the host language. There exists no dynamic task creation or termination; after partial evaluation, each MPI process determines its communication actions statically, according to the specification. Although the parallelism expert could specify communications in the host language directly, it is desirable to restrict them to the parallel specification, for optimization, safety checks etc.

- `Atom` f:
  `Atom` embeds a MOC code-transforming function in DSLP. In the skeleton generation, we will pass a function f to `Atom`, which adds new code to the code already produced for the current process. Function f will be applied when the partial evaluation process enters this occurrence of `Atom`.

- `Seq` (n,f):
  A sequence of parallel program parts can form a new parallel program part. The number of steps in the sequence is specified by n and the steps are specified by f, which maps the index of the step to the specification of the step. `Seq` causes a textual sequence of code parts. In contrast, loops with a dynamic control can be expressed in the host language.
  A constant sequence of a few steps can also be specified by a list, using the combinator `cseq`, defined as `cseq xs = Seq (length xs, nth xs)`.
- `Par` (n,f):
  `Par` specifies a parallel composition. The number of tasks is n. Function f maps the index of the task to the specification of the task. `Par` causes that code parts are generated individually in each process.
- `Comm` (n,c,b):
  A collective communication of n messages is defined among all tasks spawned by the smallest enclosing `Par`. Message i is described by ci, which defines a communication record of type `commrec`, see below. The third argument, b, specifies an array of buffers in the MOC code which has to be used for communication.

```
type commrec = { source: int; sindex: int;
                 dest: int;   dindex: int;
                 ctag: int;               }
```

  Each communication record specifies one particular point-to-point communication. The source (`source`) and destination (`dest`) process of a message refer to the index of the task in the smallest enclosing `Par (n,.)`, ranging from `0` to `n-1`. The communication buffer in the source and destination is specified by an index to the communication buffer array (`sindex`/`dindex`). Additionally, a tag (`ctag`) is specified to distinguish two messages with the same source and destination, since MPI does not establish that messages are received in the order they have been sent.

4.3. Types

The precise definition of the typed syntax of DSLP is:

```
type ('a,'b,'c) exp =
    Atom of 'a
  | Seq  of (int * (int -> ('a,'b,'c) exp))
  | Par  of (int * (int -> ('a,'b,'c) exp))
  | Comm of (int * (int -> commrec) * (('b, 'c array) code))
```

The type parameter `'a` stands for a piece of MOC code, `'b` is required by the `code` constructor, and `'c` is the type of buffers used for communication.

4.4. The Cost Model and its Use

Our parallel language comes with a simple cost model which is a variation of the work/depth model [3]. We deal with three kinds of cost information: the work (number of atomic operations), the depth (length of the longest path in the dependence graph) and the number of used processes. The last number, computed for the subtasks, is required for the allocation of the subtasks to processes. Our current model does not take communication volume into account and is therefore restricted to applications where computation costs are much higher than communication costs.

Function `wdu` stated below (lines 10–14) computes the triple (work,depth,used processes) by means of an abstract interpretation of the specification. The auxiliary function `wduSeqPar` (lines 1–9) exploits the duality of sequential and parallel composition, using the boolean parameter `isSeq` for distinction.

```
 1 let rec wduSeqPar isSeq n f =
 2  let w = ref 0 and d = ref 0 and u = ref 0 in
 3  for i=0 to n-1 do
 4    let (w1,d1,u1) = wdu (f i) in
 5    w := !w + w1;
 6    d := if isSeq then !d + d1 else max !d d1;
 7    u := if isSeq then max !u u1 else !u + u1
 8  done;
 9  (!w,!d,!u)
10 and wdu case = match case with
11    Atom _       -> (1,1,1)
12  | Comm (n,_,_) -> (1,1,1)
13  | Seq  (n,f)   -> wduSeqPar true  n f
14  | Par  (n,f)   -> wduSeqPar false n f
```

After the meta-program has been loaded by each MPI process, it generates the process-specific object program, in which computations and communications depend on the result of the cost function, the number of MPI processes and the own process identifier. This object program is then executed and controls the further process actions.

4.5. Interpretation

The interpretation function (`interpret` below) takes the specification (`spec`) and two further arguments for internal bookkeeping. These are the relative position of the task in the nearest enclosing `Par` (`relpart`) and the master processes of all tasks of this `Par` (`submasters`). A master process is one of potentially many processes assigned to a task and responsible for the entire task. Knowledge of the master processes is necessary to compute the communication partners for data exchange. The result of the `interpret` application is a function which takes an object program expression and extends it. This is due to our aim to generate code in a mostly imperative flavor, turning sequential compositions into code sequences and parallel composition into a set of parallel subtasks which are communicating with each other in a deterministic order.

```
 1 let rec interpret spec relpart submasters = match spec with
 2   Atom addCode
 3    -> if myrank=submasters.(relpart) then addCode else id
 4 | Seq (0,f)
 5    -> id
 6 | Seq (n,f) when n>0
 7    -> let interp s = interpret s relpart submasters
 8       in fun x -> interp (f (n-1)) (interp (Seq (n-1,f)) x)
 9 | Par (0,f)
10    -> id
11 | Par (n,f) when n>0
12    -> let partadrs = Array.make n 0
13       and base = ref submasters.(relpart)
14       and mypart = ref 0 in
15       for i=0 to n-1 do
16          let (_,_,u) = wdu (f i) in
17          partadrs.(i) <- !base;
18          if !base<=myrank && !base+u>myrank then mypart := i;
19          base := !base + u
20       done;
21       interpret (f !mypart) !mypart partadrs
22 | Comm (n,ci,buffers)
23    -> fun preCode
24          ->
25          if myrank!=submasters.(relpart)
26          then preCode
27          else
28           let step acc i =
29              let c = ci i in
30              if c.source = relpart
31              then send buffers (c.sindex)
32                        (submasters.(c.dest)) (c.ctag) acc
33              else if c.dest = relpart
34                   then receive buffers (c.dindex)
35                                (submasters.(c.source)) (c.ctag) acc
36                   else acc
37           in
38           let commCode = fold_left step .<()>. (fromto 0 (n-1))
39           in .< begin let y = .~preCode in .~commCode ; y end >.
```

Aside from `relpart` and `submasters`, the process identifier (`myrank`) decides what code is to be generated. In the case of an `Atom`, additional code is generated at this stage only if the process is the master of the current subtask (line 3). The other processes can come into play when the task is divided further. The static hierarchical division leads to a serial/parallel task graph, but not necessarily to a divide-and-conquer schema.

The sequential composition (lines 4–8) is quite simple: the interpretation proceeds along the steps and the functions of the steps are composed. In the parallel

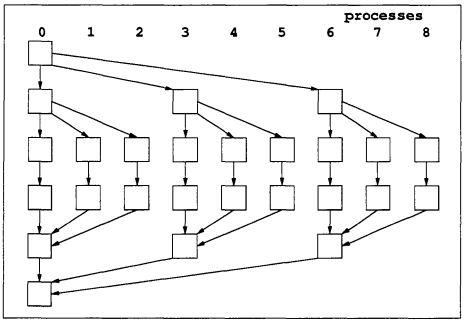

Figure 1: Divide-and-conquer with `degree=3` and `depth=2`

composition (lines 9–21), the partial evaluation has to compute the task the process has to be assigned to (`mypart`) and the master processes of all tasks (`partadrs`). This information is then used for the interpretation of the particular task, on all and only the processes assigned to this task.

A collective communication (lines 22–39) is treated by iterating (line 38) through all the single data exchanges. The local function `step acc i` (lines 28–36) processes exchange `i`. If the current process is a sender (line 30), then a send operation is added to the code `acc`. Otherwise, if the current process is a receiver (line 33), a receive operation is added. If the current process is not involved in exchange `i`, no code is added. In line 39, the state value `y` that has been computed before the communication code has been inserted is returned.

5. Specification of a Divide-and-Conquer Skeleton

This section demonstrates that a parallel skeleton can be specified relatively easy compared to a plain OCaml/MPI implementation, since the administrative calculations have been delegated to the functions `interpret` and `wduSeqPar`.

The kind of divide-and-conquer discussed here has a fixed degree of problem division (`degree`) and a fixed depth of recursion (`depth`), as depicted in Figure 1. A parallel skeleton for this kind of divide-and-conquer can be defined inductively on the depth of the recursion tree, by the MetaOCaml function `dc` shown below.

The structural parameter `degree` is used as an argument of the `Par` constructor (line 4), the structural parameter `depth` controls the unfolding of the specification function `dc` (lines 2 and 19). Three additional functions: `basic`, `divide` and `combine`, introduce the problem-specific operations into the skeleton implementation.

We can see the principle of code generation easily on line 3, when the recursion depth is 0. The argument of `Atom` is a function which takes the code already produced (`x`) and delivers a new piece of code, inserted in brackets. The old code is kept by applying the escape operator to `x`, while its return value is assigned to

a new pair of values (orig,y). The let expression is added as a new piece of code which returns a pair: a copy of the input data history (orig) required by the combine function later and the result of the application of the basic function.

Since the code produced for a Par on a particular process consists only of the path in the tree the process is involved into, the amount of code of each process depends only linearly on the depth of recursion.

```
 1 let rec dc degree basic divide combine depth  =
 2   if depth=0
 3   then Atom (fun x -> (.< let (orig,y)=.~x in (orig,basic y) >.))
 4   else Par (degree,
 5             fun mypart ->
 6  cseq
 7  [ Atom (fun x -> .< begin  let (orig,y) = .~x in
 8                             buffers.(depth) <- y;
 9                             (orig,y)               end >.);
10    Comm (degree-1,
11          (fun i -> {source=0; sindex=depth;
12                     dest=i+1; dindex=0; ctag=depth }),
13          .<buffers>.);
14    Atom (fun x -> .< let q = .~x in
15                      let (orig,y) = if mypart=0
16                                     then q
17                                     else ([],buffers.(0)) in
18                      (y::orig, divide mypart y) >.);
19    dc degree basic divide combine (depth-1);
20    Atom (fun x -> .< let q = .~x in buffers.(0) <- snd q; q >.);
21    Comm (degree-1,
22          (fun i -> {source=i+1; sindex=0;
23                     dest=0; dindex=i+1; ctag=depth }),
24          .<buffers>.);
25    Atom (fun x -> .< let q = .~x in
26                      if mypart>0
27                         then q
28                         else let (inp::orig,y) = q in
29                              let res = combine (inp,buffers) in
30                              (orig,res)
31                  >.)
32  ])
```

The depth of recursion has to be computed such that the number of processes produced does not exceed the number of reserved processors significantly. Thus, the basic case of the skeleton is not identical with the basic case of the algorithm, but must provide an implementation for a nontrivial problem size as well.

6. Karatsuba's Multiplication of Long Numbers

We cannot assume that the average application programmer has advanced knowledge in parallel programming, partial evaluation or program generation. Instead,

we must provide an interface to the application programmer which offers the performance of parallelism and hides all implementation detail. The skeleton is such an interface. From the point of view of the application programmer, there is no difference between a skeleton and a function of a sequential library. Our example is the implementation of Karatsuba's multiplication of long numbers [10], using the divide-and-conquer skeleton presented in the previous section.

The usual representation of a long number N is by a sequence of m digits c_i in a radix X system, such that $N = \sum_{i=0}^{m-1} c_i X^i$. Multiplication of numbers can conceptually be split into a polynomial multiplication (abstracting from the value of X) and a carry-correction of partial products (regarding the value of X).

The coefficients of the polynomial are stored in an array such that c_i is located at position i. To achieve a divide-and-conquer computation, the coefficients of both polynomials to be multiplied can be split as to whether they belong to a power of X less than some n or not. If $m > 1$, the polynomial multiplication can be reduced to the multiplication of smaller polynomials using the following equation:

$$(aX^n + b) * (cX^n + d) = (a * c)X^{2n} + (a * d + b * c)X^n + (b * d)$$

where n is preferably roughly $m/2$. Only the multiplications shown as asterisks are computationally expensive; multiplication with X^n is implemented simply by an index shift.

The straightforward divide-and-conquer multiplication reduces the problem to the four smaller polynomial multiplications $(a * c)$, $(a * d)$, $(b * c)$ and $(b * d)$. Karatsuba's algorithm exploits the following algebraic identity:

$$a * d + b * c = (a + b) * (c + d) - (b * d) - (a * c)$$

This way, we need only three smaller multiplications: $(a * c)$, $(b * d)$ and $(a + b) * (c + d)$. As a consequence, the asymptotic complexity reduces from $\Theta(n^2)$ to $\Theta(n^{\log_2 3})$, where $\log_2 3 \approx 1.58$. For polynomials of sizes upto 32, a loop program is more efficient (without carry correction, the value is 16). Thus, the sequential implementation of Karatsuba's multiplication (`karat_seq` below) performs a case distinction in line 3.

We see the three recursive calls of the Karatsuba algorithm in lines 19-21: `low` stands for $(b*d)$, `high` for $(a*c)$ and `mixed` for $(a+b)*(c+d)$. The implementation of the combine function (`seq_combine`) is omitted here; it is lengthy due to the carry correction and because we permit polynomials whose sizes are not powers of two. Carry correction in each step instead of once at the end enables the choice of a much larger radix (10^9 instead of 10), with the consequence that the polynomials to be multiplied are shorter by a factor of 9. This outweights the overhead of repeated carry propagation and of the 64-bit arithmetic (`addl` instead of +, `mull` instead of * etc.).

In the parallel implementation, the skeleton `dc` from the previous section is instantiated with a basic function which almost equals `karat_seq`, and a divide and

combine function. The divide function performs a similar task like in one of the lines 19-21 the functions inside the arguments of karat_seq. The combine function is almost the same in the sequential and parallel version. Details of the customizing functions for the polynomial multiplication are given in the predecessor of this paper [8]. The entire program can be obtained from the author's project webpage [18].

```
 1 let rec karat_seq xs ys =
 2 let n = Array.length xs and m = Array.length ys in
 3  if min m n <= 32
 4  then                        (* solution for small problem sizes *)
 5    let zs = Array.make (n+m) 0 in
 6    let cy = ref zerol in
 7    for i=0 to n-1 do
 8      cy := zerol;
 9      for j=0 to m-1 do
10        let sum = addl (addl !cy (of_int zs.(i+j)))
11                       (mull (of_int xs.(i)) (of_int ys.(j))) in
12        cy := divl sum radixl;
13        zs.(i+j) <- to_int (subl sum (mull !cy radixl))
14      done;
15      zs.(i+m) <- to_int !cy
16    done;
17    zs
18  else                        (* solution for large problem sizes *)
19    let low   = karat_seq (lowpartCY    xs) (lowpartCY    ys)
20    and high  = karat_seq (highpartCY   xs) (highpartCY   ys)
21    and mixed = karat_seq (mixedpartsCY xs) (mixedpartsCY ys)
22    in seq_combine (.,low,high,mixed)
```

7. Experimental Results

The experiments have been carried out on a Siemens hpcLine cluster with dual Pentium-1GHz boards with 512MB of main memory, connected by an SCI network. The execution times of the Karatsuba number and polynomial multiplication are presented in Table 1. Each single value was computed as the mean time of three runs. Table 2 shows the speedups relative to the bytecode of the sequential program (karat_seq).

Since the run-time evaluation in MOC produces bytecode, let us first compare the performance of the bytecode with the one of the highly-optimized native code. Depending on the kind of the program, the native code is by a factor of about 2-12 faster than the bytecode, as far as our experiments have shown.

Partial evaluation takes a few milliseconds; the time increases with the number of processors and, thus, case distinctions. Anyway, the time is too short to show an impact on the results presented. There is no significant difference between the sequential program in OCaml bytecode (without being affected by MOC or MPI) and the parallel MPI/MOC program on one processor.

multiplication of	long numbers			polynomials		
$\log_2$(operand size)	14	15	16	14	15	16
seq. native code	22.6	68.8	216.0	0.7	2.4	7.2
bytecode	49.9	151.9	464.2	8.4	25.6	77.1
par., #procs.=1	50.4	152.1	462.0	8.6	26.0	78.0
3	17.1	51.5	155.9	3.0	8.8	26.4
9	5.9	17.6	52.9	1.1	3.1	9.1
27	2.2	6.3	18.5	0.5	1.3	3.4

Table 1: Execution times (in seconds)

	long numbers						polynomials					
#	14		15		16		14		15		16	
ps.	SU	%	SU	%	SU	%	SU	%	SU	%	SU	%
1	0.99	99	1.00	100	1.00	100	0.98	98	0.98	98	0.99	99
3	2.92	97	2.95	98	2.98	99	2.80	93	2.91	97	2.92	97
9	8.46	94	8.63	96	8.78	98	7.64	85	8.26	92	8.47	94
27	22.68	84	24.11	89	25.09	93	16.80	62	19.69	73	22.68	84

Table 2: Speedup (SU) and efficiency (% processor utilization)

8. Conclusions

We have demonstrated that the use of partial evaluation can establish a higher degree of abstraction in parallel programming without a significant loss of performance. The most critical point in using run-time partial evaluation is the time it consumes; but for computationally expensive problems this is not an issue.

To enable this method for the practice, the circumstances under which partially evaluated code can be executed have to be improved: the current compilation of OCaml with a run-time specialization can only produce bytecode, whose interpretation is significantly slower than the optimized native code. We are going to examine current efforts in the OCaml community concerning just-in-time compilation, native code generation and offshoring, i.e., automatic translation of simple OCaml expressions directly into C or Fortran.

Acknowledgements

The author would like to thank Albert Cohen, Paul Kelly, Tobias Langhammer, Christian Lengauer and the anonymous reviewers for valuable comments. Many thanks go to the organization *Deutscher Akademischer Austauschdienst (DAAD)* for generous travel support in ARC and Procope exchanges.

References

[1] A. Berlin and D. Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, 1990.

[2] H. Bischof, S. Gorlatch, and E. Kitzelmann. Cost optimality and predictability of parallel programming with skeletons. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *Euro-Par 2003: Parallel Processing*, Lecture Notes in Computer Science 2790, pages 682–693. Springer-Verlag, 2003.

[3] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–96, 1996.

[4] J. J. Dongarra and D. W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2):151–180, 1995.

[5] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Originally published in *Systems.Computers.Controls*, 2(5):45–50, 1971.

[6] K. S. Gatlin and L. Carter. Faster FFTs via architecture-cognizance. In *IEEE PACT*, pages 249–260, 2000.

[7] S. Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Trans. on Programming Languages and Systems*, 26(1):47–56, 2004.

[8] C. A. Herrmann. Functional meta-programming in the construction of parallel programs. In S. Gorlatch, editor, *4th International Workshop on "Constructive Methods for Parallel Programming" (CMPP 2004)*, Stirling (Scotland, UK), pages 3–17. Technical Report of Westfälische Wilhelms-Universität Münster, 2004.

[9] N. D. Jones and A. J. Glenstrup. Program generation, termination, and binding-time analysis. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering*, Lecture Notes in Computer Science 2487, pages 1–31. Springer-Verlag, 2002.

[10] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 145(2):293–294, 1962. In Russian.

[11] C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors. *Domain-Specific Program Generation*. Lecture Notes in Computer Science 3016. Springer-Verlag, 2004.

[12] P. Liniker, O. Beckmann, and P. H. J. Kelly. Delayed evaluation, self-optimising software components as a programming model. In B. Monien and R. Feldmann, editors, *Euro-Par 2002: Parallel Processing*, Lecture Notes in Computer Science 2400, pages 666–673. Springer-Verlag, 2002.

[13] G. Michaelson and N. Scaife. Skeleton realisations from functional prototypes. In F. A. Rabhi and S. Gorlatch, editors, *Patterns and Skeletons for Parallel and Distributed Computing*, chapter 5, pages 129–153. Springer-Verlag, 2003.

[14] MPI Forum. *MPI-2: Extensions to the Message-Passing Interface*. Univ. of Tennessee at Knoxville, 1997. http://www.mpi-forum.org/.

[15] T. Sheard. Accomplishments and research challenges in meta-programming. In W. Taha, editor, *Semantics, Applications, and Implementation of Program Generation (SAIG'01)*, Lecture Notes in Computer Science 2196, pages 2–44. Springer-Verlag, 2001.

[16] W. Taha. A gentle introduction to multi-stage programming. In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, Lecture Notes in Computer Science 3016, pages 30–50. Springer-Verlag, 2004.

[17] http://www.cs.rice.edu/~taha/MetaOCaml/.

[18] http://www.infosun.fmi.uni-passau.de/cl/metaprog/.

[19] http://caml.inria.fr/.