Constraint Functional Multicore Programming

Petra Hofstedt and Florian Lorenzen
Department of Software Engineering and Theoretical Computer Sciene
Technische Universität Berlin, Germany
ph@cs.tu-berlin.de, florian.lorenzen@tu-berlin.de

Abstract: In this paper we present the concurrent constraint functional programming language CCFL and an abstract machine for the evaluation of CCFL programs in a multicore environment.

The source language CCFL is a simple lazy functional language with a polymorphic type system augmented by ask-/tell-constraints and conjunctions to express concurrent coordination patterns.

As execution model for CCFL we propose the abstract machine ATAF. ATAF implements a G-machine to evaluate functional expressions and provides facilities to run multiple cooperating processes on a fixed set of CPUs. Processes communicate via a shared constraint store realizing residuation semantics and committed choice.

We show a few scaling results for parallel programs obtained with a prototypical implementation of ATAF on a quadcore machine.

1 Introduction

Multicore architectures have become more and more important in recent years. Unfortunately, only truly parallel programs are able to benefit from their increase in computational power. There is, however, not yet an established method of programming these architectures, which is competitive to maintainability, stability, and performance of serial program development. Especially, many parallel programs do not automatically turn an increase in the number of cores into shorter run times like serial programs used to profit from higher clock rates. Regarding stability and maintainability, a declarative programming approach is desirable since side-effects, and explicit communication/synchronization of the imperative style are the root of many bugs hard to find or reproduce. Performance has traditionally not been the strength of declarative languages but they can be efficiently parallelized and the ubiquity of multicore computers gives a new impulse to parallel declarative programming.

In this paper, we present a small programming language CCFL of the constraint-functional family for concurrent and parallel program development. As execution model for CCFL, we propose an abstract machine ATAF that can be efficiently implemented on multicore architectures. CCFL programs compiled to ATAF are able to utilize several cores to gain performance in this way. Besides this, since CCFL is a declarative language, programs are written on a high level of abstraction by the virtue of a polymorphic type system, higher order functions, and recursive datatypes, as well as robust and understandable because of

```
fun length :: List a -> Int
def length l = case l of []     -> 0;
                         x : xs -> 1 + length xs
```

Prog. 1: List length.

the absence of side-effects.

Outline In Sect. 2 we introduce the concurrent constraint-functional language CCFL by example, where we present in particular language constructs to express concurrent and parallel computations. Section 3 is dedicated to the abstract machine ATAF as execution model for CCFL. We discuss its general structure and semantics, go into detail wrt. the realization of concurrent processes and their coordination and show performance results. We summarize the paper in Sect. 4.

2 The Source Language CCFL

The *C*oncurrent *C*onstraint *F*unctional *L*anguage CCFL is a multiparadigm programming language combining the constraint-based and the functional paradigms and it allows for parallel multicore programming.

CCFL enables a pure functional programming style, but also the usage of constraints for the description and solution of problems with incomplete knowledge on the one hand and for the communication and synchronisation of concurrent processes on the other hand.

2.1 Lazy Constraint-Functional Programming

CCFL is a lazy functional language with a polymorphic type system. A CCFL program is a sequence of data type definitions, functional and constraint abstractions. Functions are used to express deterministic computations while constraint abstractions enable the description of concurrent cooperative processes and non-deterministic behaviour.

The functional sublanguage of CCFL. The functional part of CCFL encompasses typical constructs such as case- and let-expressions, function application, and some predefined infix operations, constants, variables, and constructor terms. Program 1 shows the declaration and definition of a function `length` on lists as defined in the CCFL prelude.

Free variables. One of the main characteristics of constraints are free variables. However, in CCFL also functional expressions are allowed to contain free variables. Thus,

```
1  fun map :: (a -> b) -> List a -> List b
2  def map f l =
3    case l of []      -> [];
4              x : xs -> (f x) : (map f xs)
5
6  fun farm :: (a -> b) -> List a -> List b -> C
7  def farm f l r =
8    case l of []      -> r =:= [];
9              x : xs -> with rs :: List b
10                       in r =:= (f x) : rs & farm f xs rs
```

Prog. 2: Functional map and constraint-based farm patterns.

function applications are evaluated using the residuation principle [Smo93], i. e. the evaluation of functional expressions is suspended until variables are bound to values such that a deterministic reduction becomes possible. For example, in CCFL, the computation `(length [2,x,y,1])` of the length of a list containing the unbound variables `x` and `y` is possible, while the evaluation of an arithmetic expression `(z+2)` suspends as long as `z` is not bound to any ground value.

Constraints in CCFL. While the functional part of CCFL can be considered as *computational core* of the language, the *coordination core* responsible for the coordination of concurrent processes is based on constraints.

Consider Prog. 2 defining the function `map` and a constraint-based farm skeleton. They have a similar functionality: Both apply a function `f` on each element of a list `l` and compose the results into a list.

In contrast to `map` the abstraction `farm` is of result type `C` and, thus, a constraint abstraction (or user-defined constraint). It introduces a free variable `rs` using the **`with`**-construct in line 9 and computes the resulting list `r` by two concurrently working processes which are generated from the two constraints in line 10 which are separated by the concurrency operator `&`. That is, one of the main differences between functions and constraints in CCFL – or `map` and `farm`, resp. – is that the former are processed purely sequentially while the latter allow a concurrent computation.

While we stress this property of constraints in the present paper, constraints in CCFL have a wider functionality. Actually, we distinguish ask-constraints and tell-constraints. Both types of constraints are used in constraint abstractions. *Ask-constraints* appear in guards of rules (not discussed in the rest of this paper in detail, see [Lor06, Hof08]) and enable a synchronization of processes on the one hand and a non-deterministic rule choice (and, thus, the description of non-deterministic behaviour) on the other hand. (Conjunctions of) *Tell-constraints* allow to express concurrent coordination patterns. The constraints in line 10 in Prog. 2 are tell-constraints. The first constraint, i. e. `(r =:= (f x) : rs)`, states the equality between two functional expressions. A tell-equality constraint `t =:= s` is interpreted as strict [HAB+06]. During evaluation (as discussed in Sect. 3) a constraint

```
1  fun nfarm :: Int -> (a -> b) -> List a -> List b -> C
2  def nfarm n f l r = with rs :: List (List b)
3                      in let parts = partition n l;
4                             pf    = map f
5                         in farm pf parts rs & r =:= concat rs
6
7  fun pfarm :: (a -> b) -> List a -> List b -> C
8  def pfarm f l r = nfarm noPE f l r
```

Prog. 3: Farm parallelization patterns.

`t =:= K` e_1 `...` e_n (with non-variable expressions e_i and constructor `K`) reduces into an equality `t =:= K` v_1 `...` v_n with fresh variables v_i and constraints v_1 `=:=` $e_1, \ldots,$ v_n `=:=` e_n which may be evaluated concurrently. The second constraint `farm f xs rs` recursively generates according processes for the remaining list elements. The `&` operator combines the concurrently working processes. Thus, the `farm` constraint abstraction generates one process for each application of the function `f` on a list element. We discuss further examples of programming of concurrent systems of processes based on tell-constraints in the subsequent section.

Note that currently CCFL only supports ask- and tell-constraints over terms while an extension to external constraint domains (and solvers) such as finite domain constraints or linear arithmetic constraints is discussed in [Hof08].

2.2 Data and Task Parallelism

The description of concurrent processes can be used to specify typical parallelization schemes.

Consider Prog. 3 which defines a second version `pfarm` of the farm parallelization pattern. In contrast to `farm` as discussed above, the constraint abstraction `pfarm` includes a granularity control. It limits the number of processes to the number of processing elements `noPE`. The `nfarm` abstraction is called from `pfarm` and it partitions a list into `noPE` sublists and generates an according number of processes for list processing. These processes are distributed across different parallel computing nodes by the runtime system ATAF of CCFL as discussed in Sect. 3.

The farm patterns as described above allow a selective control of the parallelization of computations. While `farm` illustrates the generation of a possibly huge number of concurrenly working processes, `pfarm` realizes a data parallel skeleton in CCFL.

Program 4 shows another typical parallelization pattern, a data parallel fold skeleton. Again a list is partitioned into sublists according to the number of available processing nodes `noPE`. The ATAF runtime system manages their parallel processing. Eventually the list `rs` of intermediate results is (sequentially) folded using `foldl` in a final step into the

```
fun pfold :: (a -> a -> a) -> a -> List a -> a -> C
def pfold f e l r = with rs :: List (List a)
                    in let parts = partition noPE l
                       in farm (foldl f e) parts rs &
                          r =:= foldl f e rs
```

Prog. 4: Parallel fold.

```
fun pmergesort :: List Int -> List Int -> C
def pmergesort l r = pmsort noPE l r

fun pmsort :: Int -> List Int -> List Int -> C
def pmsort n l r =
  case l of
    []     -> r =:= [];
    x : xs -> case n > 1 of
                True -> let (first, second) = split l;
                            n1 = n/2
                        in with sl :: List Int, sr :: List Int
                           in pmsort n1 first sl      &
                              pmsort (n-n1) second sr &
                              r =:= merge sl sr;
                False -> r =:= msort l

fun msort :: List Int -> List Int
def msort l = ...

fun merge :: List Int -> List Int -> List Int
def merge l r = ...
```

Prog. 5: Parallel mergesort.

overall result `r`.

As one can see from these examples, CCFL does not feature specialized data structures to support data parallelism in contrast to other approaches [Nit05, CLJ+07]. Instead, the user provides a regular splitting of the data structure controlling the granularity of parallelism in this way[1], while the runtime system is responsible for an equitable distribution of the data (and tasks) onto the processing nodes. Thus, the step from data to task parallel skeletons is smooth in CCFL.

Consider the formulation of a parallel mergesort `pmergesort` (resp. `pmsort`) in Prog. 5. The list to sort is again partitioned into subparts according to the number of processing elements, where the same task (i. e. sorting) is applied to the different sublists. However, we observe two things: 1) The list partitioning is performed in a tree-like fashion, i. e. each

[1] Thus, in our approach the number of processing elements `noPE` plays a role not only in the machine space but also on the level of the problem description.

`pmsort` process bears two `pmsort` child processes. 2) Besides the two `pmsort` processes, there is a third concurrent sibling process, namely `(r =:= merge sl sr)`, realizing a merge of the sorting results.[2] Thus, Prog. 5 is already an example of the simultaneous use of data and task parallelism within one constraint abstraction.

3 The Abstract Machine ATAF

In this section we describe the design of the execution model for CCFL: the abstract machine ATAF.

We start by giving an overview of ATAF's general structure and operational semantics. We then focus on the concurrency part, i. e. ATAF's scheduling, process management, and communication instructions, sketch the compilation of CCFL programs to ATAF machine code, and conclude with some performance measurements.

3.1 Overview and Memory Layout

ATAF evaluates functional expressionss by a G-machine performing graph reduction [Joh87, Aug87]. Multiple expressions are evaluated concurrently exchanging values using the store $\mathcal{S}$, a shared memory segment. Each of these independent evaluations forms a *process*. A process executes a sequence of machine instructions of a global code segment $\mathcal{I}$.

Since graph reduction of an expression requires a heap and a stack, we assign two local memory areas to each process. Together with the store and code segment we obtain a memory layout of two shared segments and several heap and stack segments as shown in Fig. 1 (left). Mapping this layout onto a CPU with a fixed set of P cores we end up with the structure shown in the right-hand side of Fig. 1 and define the machine state Σ as a triple

$$\Sigma := \mathcal{M}^P \times \mathcal{S} \times \mathcal{I} \ , \tag{1}$$

where an instance $\mathcal{M}$ bascially contains the process local memory areas $\mathcal{P}$ and two additional components $\mathcal{R}$, $\mathcal{W}$ for scheduling (cf. Sect. 3.3):

$$\mathcal{M} := \mathcal{R} \times \mathcal{W} \times \mathcal{P} \tag{2}$$

The state of Eq. (1) fits perfectly on modern multicore architectures: each core concurrently evaluates a set of processes using local memory $\mathcal{P}$ in parallel to all other cores and processes communicate via $\mathcal{S}$ which resides in the machine's shared memory. Usually, no core in a multicore architecture has local memory, it is shared among all cores instead. The memory area $\mathcal{P}$ therefore also resides in a physically shared memory but is logically assigned to s particular core. This separation, in principle, enables the possibility to respect non-uniform memory architectures (NUMA).

[2] Note, that `msort` and `merge` are just functions and, thus, evaluated sequentially.

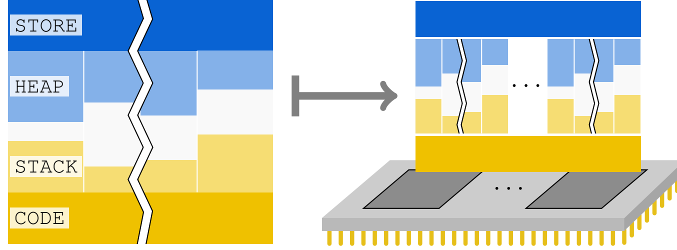


Figure 1: The memory architecture of ATAF (left) mapped onto a multicore CPU (right).

3.1.1 Machine Instructions and Semantics

The operational semantics of ATAF instructions are defined by a transition relation

$$\Mapsto \; \subseteq \Sigma \times \Sigma \tag{3}$$

connecting two subsequent states. In an execution step

$$\langle M_0, \ldots, M_{P-1}, S, I \rangle \Mapsto \langle M_0', \ldots, M_{P-1}', S', I \rangle \tag{4}$$

all instances proceed simultaneously from M_i to M_i' (with $M_i = M_i'$ permitted) and the meaning of each instruction is specified by rules defining valid transitions $\sigma \Mapsto \sigma'$.

We skip the details of $\Mapsto$ (they can be found in [Lor06]), and give an informal description of the concurrency part of ATAF instead. A good description of the graph reduction part to evaluate purely functional expression with lazy semantics is [LP92].

3.2 Communication and Synchronization

Processes communicate by reading and writing values to and from the store. The store $\mathcal{S}$ is a triple

$$\mathcal{S} := V \times \mathcal{B} \times \mathcal{B}[v] \;\; , \tag{5}$$

with the words of the store segment V, a *blocked* queue $\mathcal{B}$, and *suspended* queues $\mathcal{B}[v], v \in V$.

To prevent inconsistencies in the shared structures, access to the store has to be enclosed by the instruction pair LOCK, UNLOCK. We restrict the valid transitions such that only one M_i may execute LOCK or UNLOCK, i. e. reading and writing the store is subject to mutual exclusion. All processes waiting for access to the store are enqueued in $\mathcal{B}$ by LOCK if necessary and dequeued by UNLOCK.

Because of the residuation semantics of CCFL, processes may have to wait on variables, either because no guard is fulfilled or because a function is applied to an unbound variable. When a process has to wait on a word v in the store, it executes the instruction SUSPEND v which adds the process to $\mathcal{B}[v]$, a queue containing processes expecting a modification

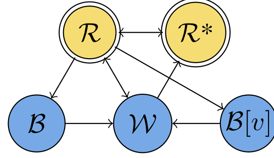


Figure 2: Process state transition diagram of ATAF: running $\mathcal{R}$, running uninterruptible $\mathcal{R}*$, suspended $\mathcal{B}[v]$, ready $\mathcal{W}$, and blocked $\mathcal{B}$. Nodes with double border indicate states occupied by at most one process, nodes with single border represent queues.

of v. As soon as the word's content is changed $\mathcal{B}[v]$ is flushed and the processes are re-enqueued in $\mathcal{W}$ of their instance to re-examine its value.

Changes of variables in the store happen on the evaluation of a tell-constraint `v =:= e` which is implemented by the TELL instruction. TELL also transfers all processes of $\mathcal{B}[\texttt{v}]$ into the ready state.

3.3 Scheduling and Process Management

In each processing instance, only one process is executed, i. e. is in the *running* state and occupies the $\mathcal{R}$ register. Other processes waiting for execution are in state *ready* and enqueued in $\mathcal{W}$. All processes $\{\mathcal{R}\} \cup \mathcal{W}$ on an instance are scheduled for execution by a preemptive time-slice scheduler in round-robin fashion. Preemption may be circumvented but this privilege is restricted to the process currently accessing the store. It is therefore not exposed to the programmer but encapsulated in the LOCK and UNLOCK instructions instead. This strategy helps to keep locking intervals relatively short because a process cannot be preempted while manipulating the store. Figure 2 summarizes the possible process states and transitions.

Conjunctions like `e & f` create new processes to evaluate the expressions `e` and `f`. The ATAF instruction SPAWN sets up a new process for `e` and transfers all required data to its local stack and heap to evaluate `e`. Let `e` be the expression `r =:= g v` in the context

```
let v = x in e & f .
```

`e` needs the expression `x` to evaluate `g v`. Since `x` may contain further references to the heap of the context's process, which are not available in the new empty process environment for `e`, these referenced structures have to be copied.

Unlike process creation there is no instruction for termination. A process terminates as soon as its stack is empty. The G-machine instruction UNWIND, responsible to find the next reducible expression, detects this situation and deallocates the stack and heap of the terminating process.

A newly created process will run on that instance that currently has fewest processes to execute, thus implementing a simple load balancing. This scheme provides the possibility that each of P processes spawned on an empty machine is executed by one core, thus

achieving true parallel execution (cf. Sect. 2.1 and 2.2).

Initial and Final State The initial state of a program loaded into ATAF is a single process evaluating the `main` function which starts additional processes. A program terminates as soon as no process is left on any instance.

3.4 Compiling CCFL to ATAF

To illustrate the generation of ATAF instructions for CCFL programs we present compilation schemes for two language constructs (for details, we again refer to [Lor06]):

- Conjunctions: e_1 & ... & e_n.
- Tell-equalities: v =:= e

These two types of expressions characterize most of CCFL's concurrency features as demonstrated in Sect. 2.1 and use the five instructions LOCK, UNLOCK, TELL, SPAWN, SUSPEND, which have been introduced in the previous section.

Our compiler is a function

$$\mathcal{C} : \mathit{Expr} \rightarrow [\mathit{Instr}] \tag{6}$$

mapping a CCFL expression to a sequence of ATAF instructions.

3.4.1 Compiling Conjunctions

We compile a conjunction with the function

$$\mathcal{C}[\![e_1 \;\texttt{\&}\; \dots \;\texttt{\&}\; e_n]\!] = \varphi +\!\!+ \psi_1 +\!\!+ \cdots +\!\!+ \psi_{n-1} +\!\!+ \tilde{\psi} \tag{7}$$

where

$$\varphi = [\mathsf{SPAWN}\ f(\psi_1), \dots, \mathsf{SPAWN}\ f(\psi_{n-1}), \mathsf{JUMP}\ f(\tilde{\psi})] \tag{8}$$

$$\psi_i = \mathcal{C}[\![e_i]\!] +\!\!+ [\mathsf{JUMP}\ b(\tilde{\psi})] \tag{9}$$

$$\tilde{\psi} = \mathcal{C}[\![e_n]\!] \ . \tag{10}$$

For each but the last expression e_n, the sequence φ calls SPAWN to create a new process. Each of these $i = 1, \dots, n-1$ processes executes the code beginning at relative address $f(\psi_i)$, the address of the first instruction of sequence ψ_i, which is the result of recursively compiling expression e_i. For expression e_n, no new process is created but its instructions sequence $\tilde{\psi}$ is executed by the parent process, hence the jump to $f(\tilde{\psi})$ at the end of φ. To prevent any of the e_i, $i = 1, \dots, n-1$ to execute instructions not belonging to ψ_i, we insert a jump to the address $b(\tilde{\psi})$, the first instruction following $\tilde{\psi}$.[3]

[3] In ATAF, all instruction addresses as in JUMP or SPAWN are instruction pointer relative.

3.4.2 Compiling Tell-Equalities

A tell-equality is compiled into the following subsequences:

$$\mathcal{C}[\![v \ \texttt{=:=} \ e]\!] = \mathcal{C}[\![e]\!] +\!\!+ \varphi(v) +\!\!+ \psi \tag{11}$$

The process evaluating $v \ \texttt{=:=} \ e$ first computes e by the instruction sequence $\mathcal{C}[\![e]\!]$. Then it proceeds by locking the store to augment the new value by sequence $\varphi(v)$:

$$\varphi(v) = [\mathsf{LOCK}, \mathsf{ISBOUND}\ r\ 4, \mathsf{TELL}\ v, \mathsf{UNLOCK}, \mathsf{JUMP}\ 3] \tag{12}$$
$$+\!\!+ [\mathsf{SUSPEND}\ r, \mathsf{JUMP}\ -6] \tag{13}$$

Since the result r of e may be an unbound variable, the process may have to suspend. This check is done by the conditional jump ISBOUND:

- If r is bound, ISBOUND transfers control to the next instruction TELL, which records the value r for v in the store. Finally, the store is unlocked and the remaining two instructions are skipped by JUMP 3.
- If r is an unbound variable, the process jumps to the beginning of (13) and suspends on r, which simultaneously unlocks the store. As soon as r is modified, the process jumps back to the head of sequence (12) to re-examine the value of r.

If the result r is a constructor, the operational semantics of CCFL require that its subterms are evaluated (see Sect. 2.1) by new processes. In order to spawn these processes, TELL pushes a variable-expression pair (v_i, e_i) for each subterm e_i to be evaluated by $v_i \ \texttt{=:=} \ e_i$ onto the stack. The compiler creates processes to evaluate the expressions `lift` $v_i \ e_i$ with

```
fun lift :: a -> a -> C
def lift r e = r =:= e
```

in the sequence ψ:[4]

$$\psi = [\mathsf{ISBOUND}\ v_i\ 3, \mathsf{POP}\ 1, \mathsf{JUMP}\ 8] \tag{14}$$
$$+\!\!+ [\mathsf{SPAWN}\ 3, \mathsf{POP}\ 2, \mathsf{JUMP}\ -5] \tag{15}$$
$$+\!\!+ [\mathsf{PUSHFUN}\ \texttt{lift}, \mathsf{MKAP}, \mathsf{MKAP}, \mathsf{UNWIND}] \tag{16}$$

The ISBOUND instruction in (14) checks if a (v_i, e_i) is returned and continues with sequence (15) in that case, or jumps ahead of (16) by JUMP 8 otherwise.

Sequence (15) spawns a new process which executes (16), a sequence of G-machine instructions to evaluate the application `lift` $v_i \ e_i$, removes the current (v_i, e_i) pair (POP 2), and jumps back to the head of ψ to check for a next pair.

[4] `lift` is added to all CCFL programs by the compiler.

3.5 Prototypical Implementation and Scaling Behaviour

We have implemented a protoype interpreter of ATAF in HASKELL using the OPENMPI implementation of MPI [SOHL+96] as communication library to evaluate the feasibility and scaling characteristics of our approach. Measurements have been performed on an Intel quadcore machine with 2.4 GHz and 4 GB of memory running GNU/LINUX. We plot the average speed-up of five samples obtained by running our examples on 1, 2, 3, and 4 cores of the test machine.

The speed-up of a parallel program is defined as

$$S_P = \frac{T_1}{T_P} \ , \tag{17}$$

where T_1, T_P are the times the program runs on 1, P cores resp. and indicates how well a program is parallelized.

3.5.1 Speed-up of `pfarm`

To demonstrate the scaling of `pfarm` (Prog. 3) we calculate the square root of the numbers $1, 2, \ldots, m$ with $m = 1000, 2000$.

Figure 3(a) shows an acceptable speed-up of the `pfarm` coordination, especially for an unoptimized interpreter prototype. Due to Amdahl's law [Amd67], it stays in the sublinear regime, of course. The program shows good scaled speed-up performance approaching optimal speed-up for larger inputs.

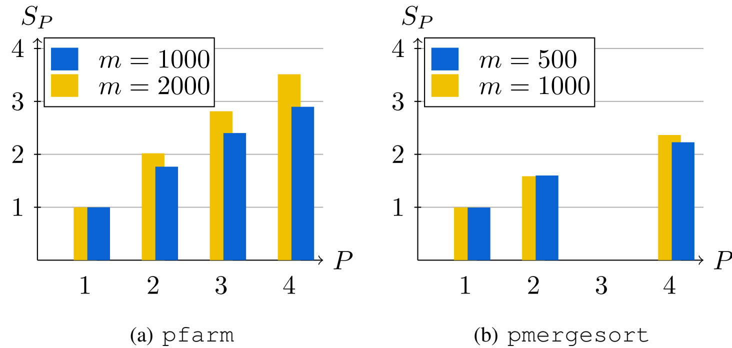
Similar to `pfarm` is `pfold` (see Prog. 4) but the folding function `f` has to be sufficiently time-consuming. Otherwise, the run-time is dominated by the list partitioning. For example, calling `pfold + 0 l r` on a list of integers `l` gives of course no performance benefits for $P = 2, 3, 4$.

3.5.2 Speed-up of `pmergesort`

We sort two lists with length $m = 500, 1000$ of random integers with the `pmergesort` program of Prog. 5. Due to the divide-and-conquer nature of mergesort running on a number of processors P in the range $2^k < P < 2^{k+1}$ is not faster than running on $P = 2^k$ – we therefore skip $P = 3$. The scaling of `pmergesort` as shown in Fig. 3(b) is not as good as of `pfarm` because the fraction of inherently serial computation (`merge`) is larger.

4 Conclusion

We presented the design, implementation, and performance measurements of an abstract machine for the concurrent constraint-functional language CCFL for parallel multicore programming. CCFL is a declarative language using constraints for the description of systems



(a) pfarm

(b) pmergesort

Figure 3: Speed-up of two CCFL examples for different problem sizes.

of concurrent processes. We have shown how to elegantly and abstractly express typical data and task parallel execution patterns. The actual data and task distribution as well as the process coordination is controlled by the abstract machine ATAF. Despite its simplicity, ATAF already shows acceptable scaling behaviour for a small number of cores, but there are certainly possibilities for improvement, especially regarding the following two aspects:

- The scheduler does not migrate processes and the load-balancing will show poor results when the workload per process is very irregular. More sophisticated scheduling algorithms like work stealing as used in CIAO [CCH08] or the Penny system [MH97] would definitely improve the performance and should be investigated.
- The global lock realizing mutually exclusive access to the store is a potential performance bottleneck. Techniques for efficient implementations of message passing via a shared heap as e. g. studied in [JSW03] for the Erlang/OTP system could be adapted for ATAF such that send- and receive-operations in a one-to-many communication play the role of TELL- and SUSPEND-instructions.

Related Work From the programming languages' structure point of view, EDEN and GOFFIN can be seen as close relatives of CCFL. EDEN [LOP05] is an extension to the lazy functional language HASKELL [Pe03] that adds explicit process constructs to allow the description of parallel applications. An EDEN program defines a system of processes which, evaluating functional expressions, exchange data via unidirectional communication channels between a writer and a reader modelled by head-strict lazy lists. Non-determinism is realized by means of the predefined process abstraction `merge`.

The target architecture for EDEN are distributed memory systems like networks of workstations or clusters. Nevertheless it has recently been investigated in a multicore environment [BDLL09]. The execution model of EDEN is DREAM [BKL$^+$97], an extension of the STG-machine, supporting channel and communication primitives. DREAM, unlike ATAF, has no shared memory segments since processes cooperate exclusively via message-passing.

The language GOFFIN [CGKL98] combines HASKELL with a constraint-based coordi-

nation language to express parallelism and non-deterministic computation. It provides a similar structure like CCFL, while CCFL's constraint abstractions are more oriented to predicates than to functions and the ask-constraint's functionality is a bit more extended. Moreover, in [Hof08] we discuss the extension of CCFL constraints to typical constraint systems.

Further related work from the functional realm are e. g. DATA PARALLEL HASKELL, MANTICORE, and ERLANG. DATA PARALLEL HASKELL [CLJ+07] is an extension of HASKELL targeting multicore architechtures. It allows nested data-parallel programming [Les05] based on a built-in type of parallel arrays by distributing data among processors and applying transformations to flatten nested parallel operations in order to reduce synchronization points and intermediate structures. The strict functional language ERLANG [Arm07] supports concurrency by explicit constructs for process creation and message passing.

MANTICORE [FFR+07] is a strict functional language offering several levels of parallelism: fine-grained data-parallelism for arrays and tuples, parallel bindings, which are all implicitly threaded, and explicit threading as in CML [Rep99] for coarse-grained parallelism and concurrent programming. One important aspect of the MANTICORE is to support different scheduling policies in the same program, e. g. gang scheduling and work stealing [FRR08]. Therefore, the core of MANTICORE only provides primitive operations for scheduling and thread management like stopping and preemping that are composed to complex scheduling algorithms.

Concurrent Constraint Programming (CCP) [Sar93] is an important area of development, which is, though not especially designed for parallel architectures, the origin of many concepts regarding the coordination of processes like ask-/tell-constraints, and guarded expressions as in CCFL.

Ackknowledgements We thank our colleague Martin Grabmüller for many valuable suggestions and useful discussions regarding the design and implementation of the prototype. The work of Petra Hofstedt has been partially supported by a postdoctoral fellowship No. PE 07542 from the Japan Society for the Promotion of Science (JSPS).

References

[Amd67] G. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485. AFIPS Press, 1967.

[Arm07] J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Programmers, 2007.

[Aug87] L. Augustsson. *Compiling Lazy Functional Languages, part II.* PhD thesis, Chalmers Tekniska Högskola, Göteborg, 1987.

[BDLL09] J. Berthold, M. Dieterle, O. Lobachev, and R. Loogen. Distributed Memory Programming on Many-Cores – A Case Study Using Eden Divide-&-Conquer Skeletons. In K.-E. Großpitsch, A. Henkersdorf, A. Uhrig, T. Ungerer, and J. Hähner, editors, *ARCS*

'09 – 22th International Conference on Architecture of Computing Systems 2009 – Workshop proceedings, pages 47–55, 2009.

[BKL+97] S. Breitinger, U. Klusik, R. Loogen, Y. Ortega-Mallén, and R. Peña. DREAM: The DistRibuted Eden Abstract Machine. In *Implementation of Functional Languages*, pages 250–269. Springer Verlag, 1997.

[CCH08] A. Casas, M. Carro, and M. V. Hermenegildo. Towards a High-Level Implementation of Execution Primitives for Unrestricted, Independent And-Parallelism. In *10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, volume 4902 of *LNCS*, pages 230–247. Springer, 2008.

[CGKL98] M. M. T. Chakravarty, Y. Guo, M. Köhler, and H. C. R. Lock. GOFFIN: Higher-Order Functions Meet Concurrent Constraints. *Science of Computer Programming*, 30(1–2):157–199, 1998.

[CLJ+07] M. M. T. Chakravarty, R. Leshchinskiy, S. L. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: a status report. In N. Glew and G. E. Blelloch, editors, *DAMP*, pages 10–18. ACM, 2007.

[FFR+07] M. Fluet, N. Ford, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Status Report: The Manticore Project. In *ML '07: Proceedings of the 2007 workshop on Workshop on ML*, pages 15–24, New York, NY, USA, 2007. ACM.

[FRR08] M. Fluet, M. Rainey, and J. Reppy. A Scheduling Framework for General-purpose Parallel Languages. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 241–252, New York, NY, USA, 2008. ACM.

[HAB+06] M. Hanus, S. Antoy, B. Braßel, H. Kuchen, F. J. López-Fraguas, W. Lux, J. José Moreno-Navarro, and F. Steiner. Curry. An Integrated Functional Logic Language. Version 0.8.2, March 2006.

[Hof08] P. Hofstedt. CCFL – A Concurrent Constraint Functional Language. Technical Report 2008-8, Technische Universität Berlin, 2008. (Available from World Wide Web: `http://iv.tu-berlin.de/TechnBerichte/2008/2008-08.pdf` [cited: April 24, 2009]).

[Joh87] T. Johnsson. *Compiling Lazy Functional Languages*. PhD thesis, Chalmers Tekniska Högskola, Göteborg, 1987.

[JSW03] E. Johansson, K. Sagonas, and J. Wilhelmsson. Heap Architectures for Concurrent Languages using Message Passing. *SIGPLAN Notices*, 38(2 supplement):88–99, 2003.

[Les05] R. Leshchinskiy. *Higher-Order Nested Data Parallelism: Semantics and Implementation*. PhD thesis, TU Berlin, 2005.

[LOP05] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[Lor06] F. Lorenzen. An Abstract Machine for a Concurrent (and Parallel) Constraint Functional Language. Master's thesis, Technische Universität Berlin, 2006. (Available from World Wide Web: `http://user.cs.tu-berlin.de/~florenz/dt/thesis.pdf` [cited: April 24, 2009]).

[LP92] D. R. Lester and S. L. Peyton Jones. *Implementing Functional Languages: A Tutorial*. Prentice Hall, 1992.

[MH97] J. Montelius and S. Haridi. An evaluation of Penny: a system for fine grain implicit parallelism. In *PASCO '97: Proceedings of the second international symposium on Parallel symbolic computation*, pages 46–57, 1997.

[Nit05] T. Nitsche. *Data Distribution and Communication Management for Parallel Systems*. PhD thesis, Technische Universität Berlin, 2005.

[Pe03] S. L. Peyton Jones and et. al. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, April 2003.

[Rep99] J. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.

[Sar93] V. A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[Smo93] G. Smolka. Residuation and Guarded Rules for Constraint Logic Programming. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming, Selected Research*, pages 405–419. MIT Press, 1993.

[SOHL+96] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1996.