D'Caml: Native Support for Distributed ML Programming in Heterogeneous Environment

Ken Wakita, Takashi Asano, and Masataka Sassa

Department of Mathematical and Computing Sciences,
Tokyo Institute of Technology, Tokyo 152-8552, Japan
{wakita,asano,sassa}@is.titech.ac.jp

Abstract. Distributed Caml (D'Caml) is a distributed implementation of Caml, a dialect of ML. The compiler produces native code for diverse execution platforms. The distributed shared memory allows transmission and sharing of arbitrary ML objects including higher-order functions, exceptions, and mutable objects without affecting the sequential semantics of ML. The distributed garbage collector reclaims unused distributed data-structures. Examples demonstrate expressivity of higher-order distributed programming using Distributed Caml. The paper presents the design, implementation, and preliminary performance results of the system.

1 Introduction

Due to significant advances in network and platform technologies, exchange of digital information on the Internet is replacing the traditional methods of information exchange and has increased the demand for high-quality distributed services and applications for various use. However development of distributed applications remains hardest in software development and thus it is getting more difficult to meet the rapidly growing demand for high-quality Internet software. Obstacles in development of distributed software, among others, are (i) hardware and software heterogeneity of computers that participate in distributed computing, (ii) data conversion required for inter-node communication, and (iii) branches of knowledge required for system software development such as OS system calls, message passing libraries, theories of parallel/distributed/real-time computing, fault tolerance, security, and variety of libraries for system programming.

Distributed Caml (D'Caml) is a distributed programming language and system being developed by the authors. Our research goal is to offer a programming system which alleviates above mentioned obstacles, hereby invite novice programmer to practical distributed programming.

A D'Caml application is a collection of cooperative distributed processes that execute on a heterogeneous workstation cluster. Here, "heterogeneity" refers to difference in hardware resources (instruction set, CPU architecture, memory hierarchy, accessible hardware devices, etc.) and software configuration (operating system, versions of libraries, etc.). For instance, our development platform comprises Sun SPARC Stations (Ultra SPARC/Solaris), Digital workstations (Alpha/Digital UNIX and Linux), and PCs (Pentium/Linux and Solaris) connected via Ethernet.

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 914–924, 1999.

The distributed shared memory (DSM) substrate of the D'Caml runtime system entirely hides the underlying heterogeneous, distributed environment and provides the programmer for virtually shared, single address space abstraction. This functionality enables transmission and sharing of arbitrary ML objects including functional closures (functions with binding of free variables).

Remote closure invocation is a programming mechanism in D'Caml that allows a thread to invoke an arbitrary function as a new thread in a remote process. Integration of distributed shared memory and remote closure invocation offers the programmer a programming style similar to single-node multi-thread programming like Java, Concurrent ML [13], and Objective Caml [9].

Remote closure invocation differs from standard RPC mechanisms, as found in Ada and Java + RMI, in several important respects. It supports dynamically created functions (closures). When a closure is passed, not only the code pointer but also its free lexical references are transmitted to the remote site. Secondly, these variables are virtually shared between the sender and the receiver so that sharing semantics is maintained consistent with the sequential semantics. On the other hand, in most RPC system, variables are simply copied and thus violating sharing semantics for them (i.e., side-effect made on the variable is invisible by remote processes). Thirdly, unhandled exception is forwarded to and can be caught by the thread that issued remote invocation. Finally, there is no constraint on the kinds of objects that can be transmitted during remote closure invocation.

The rest of the paper is organized as follows. Section 2 describes the language and the system of D'Caml. Section 3 shows the implementation scheme of the system and Section 4 presents its preliminary performance results. Section 5 compares our work with others and concludes this paper.

2 The Distributed Caml System

An application produced by the D'Caml system executes as a collection of cooperative distributed processes running on a (possibly heterogeneous) workstation cluster.

D'Caml supports two kinds of execution platforms . The D'Caml/MPI configuration utilizes the standard message passing interface [7] for low-level communication. In this configuration, an application starts as a process which invokes all other cooperative processes in the cluster. We call the first process that invokes others the *host process*. A distributed application created with D'Caml/MPI configuration is static in the sense that it does not support dynamic creation of processes after the application startup time.[1] The D'Caml/TCP configuration supports dynamic process creation. In this configuration, an application starts as a single-process application which can spawn a new process at remote computer arbitrarily.

The D'Caml language is a superset of Objective Caml (O'Caml) [9] developed at INRIA. In addition to functionality supported by O'Caml, D'Caml offers distributed shared memory, types and functions for distributed computing as summarized in Figure 1), and distributed garbage collector.

[1] The lack of dynamic process creation is due to the specification of MPI 1. MPI 2 proposal suggests inclusion of this facility.

```
type node
val current_node: unit -> node
val is_host: unit -> bool
val get_nodes: unit -> node array    (* only for MPI *)
val start_node: string -> string -> string -> node (* only for TCP *)
val spawn: node -> (unit -> 'a) -> unit
val rcall: node -> (unit -> 'a) -> 'a
```

Fig. 1. Some of the primitive functions defined in the *Dcaml* module

The `node` type is an abstraction of the distributed processes which encapsulates heterogeneity of the underlying hardware and software resources. An instance of the `node` type identifies one of the distributed processes that participate in the distributed computing. The function *current_node* gives the node that corresponds to the process where this function is executed. The function *is_host* tells if the current process is the host process.

In the D'Caml/MPI configuration the *get_nodes* function gives an array of the participant nodes In the D'Caml/TCP configuration, the *start_node* function is used to spawn a new D'Caml process at a remote site. Three parameters specify the network address, the working directory, and path to the D'Caml executable code.

All the D'Caml processes execute the same Caml program in a SPMD manner. However typical D'Caml application initiates execution only at the host process which serves as the master process using the *is_host* primitive:

```
if Dcaml.is_host () then print_string (Unix.gethostname ())
```

Two functions, *spawn* and *rcall*, achieves remote closure invocation. They take a node and a closure, send the closure to the remote node, and invoke the closure as a new thread in the remote address space of the node. The *spawn* primitive is asynchronous: its execution immediately finishes and the remote thread runs independently with the current one. On the other hand, execution of *rcall* synchronizes with the termination of the remote thread and takes its return value as its own return. In this way, *rcall* behaves like a remote procedure call. The following program collects host names for all the nodes in an array *hostnames* using the *rcall* primitive.

```
if is_host () then
  let nodes = Dcaml.get_nodes () in
  let hostnames = Array.make (Array.length nodes) "" in
  Array.iteri
    (fun i node ->
      Dcaml.rcall node
        (fun () -> hostnames.(i) <- Unix.gethostname ()))
    nodes
```

This program is interesting in two aspects. The lexical scope of the closure being *rcall*ed contains free references to local names, *hostnames* and *i*. As mentioned earlier, these free references are sent as part of the closure and become accessible by the

*rcall*ed closure to a remote node. Secondly, values referenced by these free references are shared between distributed nodes. Therefore side-effect made on them by one node is observable by the others. In the above example, the host node can retrieve hostnames assigned by remote nodes.

Inter-node communication using remote closure invocation is advantageous in comparison with standard message passing because the programmer is released from implementing with communication protocols. D'Caml offers basic communication protocols as functions (e.g., *spawn* and *rcall*). Higher order functional programming allows us to build higher-level communication protocols from lower-level primitives. For example, let us consider the *future* primitive as found in Multilisp [8]. The future primitive takes a node and a function and executes the function at the designated node. Execution of the future primitive immediately finishes and returns a handle for the return value which can be used to retrieve the return value in an arbitrary future time.

```
let future node f =  (* node -> (unit -> 'a) -> (unit -> 'a) *)
  let rbox = ref None in
  let sem = Mutex.create () in
  Mutex.lock sem;
  Dcaml.spawn node
    (fun () -> rbox := Some (f ()); Mutex.unlock sem);
  (fun () -> Mutex.lock sem; match !rbox with Some v -> v);;

let touch = future some_node a_closure
in ...
   let result = touch () in ...
```

In this implementation, the function *f* is *spawn*ed to the designated node and its result is stored in a reply box. The handle to the reply box is represented by a function which synchronizes with the execution of *f* using a semaphore.

The last example in this section demonstrates how standard parallelizing techniques are expressed in D'Caml. The program presented in Figure 2 paints a graphical image of Mandelbrot's fractal set defined over the complex number space. Because each pixel color can be computed independently from others, this computation is an inherently parallel computation. The program achieves parallelism in a *master-workers style*: single master process divides the entire computation into smaller sub-computations and feed them to multiple worker processes. In the program, computation for a two-dimensional matrix is divided into many sub-computations that calculates pixel colors in a row. The host node serves as the master and all other become workers. For each worker, the host node starts a master thread which continuously feeds the corresponding worker node with a *new_job* until entire computation finishes.

3 Implementation

In this section, we describe the implementation scheme of the D'Caml system. We briefly overview its software organization in the next subsection. Subsequent subsections describe the implementation scheme in detail.

```
exception Done;;
let size = 500;;
let image = new_image size size and next = ref 0;;
let mandelbrot i j = ...;;
let compute_row i =
  let row = Array.create size 0 in
  Array.iteri (fun j point -> point <- mandelbrot i j) row;;
let new_job () =
  let i = !next in
  if i < size then (incr next; fun () -> compute_row i)
  else raise Done;;
let master node =
  try while true do paint image (Dcaml.rcall node (new_job ()))
    done
  with Done -> ();;
if Dcaml.is_host () then
  Array.iter (fun node ->
    Dcaml.spawn (Dcaml.current_node ()) (fun () -> master node))
  (Dcaml.get_nodes ())
```

Fig. 2. Master-workers style parallel computation of the Mandelbrot set

3.1 Software Architecture

Figure 3 illustrates the software architecture of the Distributed Caml system. A D'Caml application runs on a heterogeneous workstation cluster. Given a Caml program, the D'Caml compiler produces native code for all architectures involved in the cluster.

D'Caml DSM is a software-only implementation of single address space that spans over heterogeneous, physically distributed address spaces. Major difficulties implementation of a DSM for a functional language, namely transmission of functional closures (Subsection 3.2) and dealing with shared mutable objects (Subsection 3.3). Low-level message transmission is established by per-node thread that we call *message handler* (Subsection 3.6).

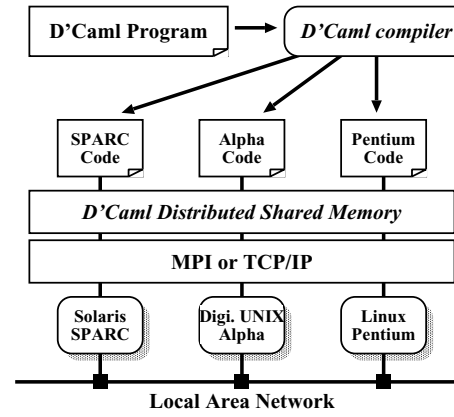


Fig. 3. Software architecture of the Distributed Caml system

3.2 Marshaling

To transmit Caml objects over the network we need a way to convert their internal data representation to and from their architecture-independent forms, namely *marshaling* and *unmarshaling*.

With most implementations of functional languages, each runtime data is attached with GC tag which gives its structural information, size and sort of object (e.g., constant, arrays, or pointer aggregates, etc). This gives sufficient information for marshaling most heap allocated Caml objects. In fact, the built-in marshaling mechanism of O'Caml traces the data structure using this information.

The marshaling mechanism of O'Caml, however, does not deal with distributed and heterogeneous environment because it is assumed that the marshaled data will be loaded back by the same process at later time. D'Caml extends the marshaling mechanism of O'Caml by support for distributed data structures and ability to safely transmit closures. Distributed data structures are expressed using remote pointers as described in Subsection 3.3. A difficulty in passing a closure over the network is the representation of code that implements the closure. This issue is discussed in Subsection 3.4.

3.3 Remote Pointers

In order to express distributed data structures, D'Caml DSM incorporates the notion of *remote pointers* by which a data can remotely reference an object allocated in a remote address space. Figure 4 illustrates the representation of a remote pointer. A remote pointer points to an entry in a statically allocated *import table* whose entry consists of the node identifier n_1 and entry index j of the referenced object. The pair of n_1 and j identifies an entry in the export table of the remote node, which contains a regular pointer pointing the remotely referenced object.

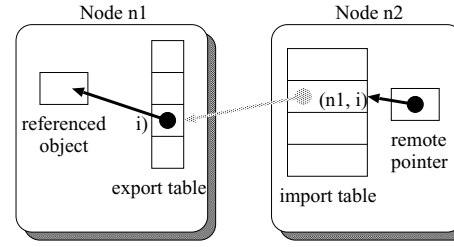


Fig. 4. Implementation of remote pointers.

The export table is contained in the GC root set of the per-node local garbage collector so that remotely referenced object is not collected even if it is not referenced locally.

Import table achieves fast dereference of regular pointers. Dereference of a possibly remote pointer requires to determine if it is a remote pointer (*pointer testing*). In D'Caml, pointer testing is issued for each access to *possibly remote* pointer. Because import table is statically allocated each pointer testing is as cheaply done as single address comparison.

Now, we discuss the data transmission scheme and how remote pointers are introduced there. Types of data transmission in D'Caml falls in three cases: (1) Remote closure invocation transmits a closure which may contain various Caml objects in its

defining environment. (2) Dereference of a remote pointer requires the remotely referenced object to be transfered. (3) Update of remotely referenced object requires a value to be sent and stored in the remote object.

When a node transmits a D'Caml object to other node, it replicates the object in the remote node substituting all regular pointers referencing mutable objects with remote pointers. This scheme guarantees that mutable objects are not replicated but referenced through remote pointers.

D'Caml compiler takes advantage of the fact that immutable objects are always pointed to by regular pointers and removes pointer testing for them. Therefore D'Caml program coded in mostly functional style does not suffer from pointer testing overhead. The type system of ML, which covers object mutability, offers the compiler precise information and enables this optimization.

3.4 Closure Passing

A closure is a function defined in a local scope. The value assignment environment for the variables in the lexical scope is called the defining environment for the closure. The defining environment is a set of value assignments to variables that occur free in the function definition.

In modern implementation of functional programming languages, runtime closure representation comprises the code fragment that implements the function and an array of values assigned to the free variables. In native implementation, the code is actually represented by the address of the entry point of the code fragment of the function implementation which we call *entry address*.

In order to transmit closures across the heterogeneous network, we need to interpret an entry address used in one address space into another entry address which correctly points to the corresponding code fragment in the remote address space.

The D'Caml compiler assumes that all the D'Caml processes are running native code that is produced from the same Caml program. Given this, the D'Caml compiler and linker assigns for each code fragment a unique identifier (*entry point identifier*) and the runtime system translates entry addresses to and from entry point identifiers.

Figure 5 sketches our remote closure passing scheme. The D'Caml compiler processes module source programs. The code generator of D'Caml assigns an architecture-independent, module-wide unique number for each entry point in the produced code. The module identifier assigned by the statup-time linker and the entry point number produced by the D'Caml compiler together serve as network-wide, unique identifier of the code fragment. Figure 5 depicts passing of the following closure defined in a module identified by m from node n_1 to another node n_2:

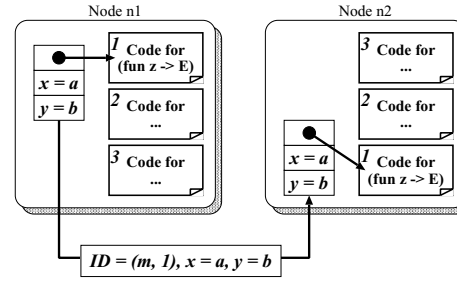


Fig. 5. Remote closure passing scheme

let x = a **and** y = b **in** (**fun** z -> E)

A variant of this address translation technique is used to implement remote exception handling and transmission of atomic objects.

3.5 Garbage Collection

The D'Caml system supports automatic reclamation of unreferenced objects by coordination of per-node local garbage collectors and a distributed garbage collector. The local garbage collector is a slightly modified version of O'Caml garbage collector. It deallocates unreferenced heap-allocated objects. As mentioned before, the GC root set includes the export table, to inform the local garbage collector of remote references.

The distributed garbage collector deallocates unused entries in the export tables which point to heap-allocated objects. If these objects are not locally referenced, they will be deallocated at the next attempt of local garbage collection. The distributed garbage collection mechanism is based on a variant of reference counting algorithm called *weighted reference counting (WRC)* [3].

3.6 Message Handler

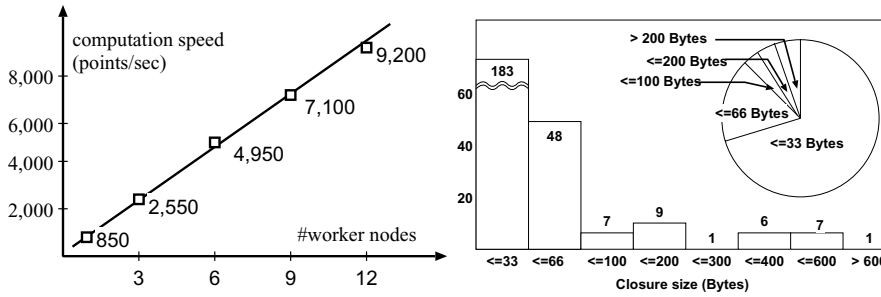
Each D'Caml process executes a thread called the *message handler* which sends outgoing messages and dispatches incoming requests. Types of messages include requests for remote closure invocation, requests for dereference and update of remote pointer, and reply for remote dereference, and messages used by the runtime system such as the distributed garbage collector.

In the D'Caml/MPI configuration, the message handler waits for the message by polling every few milli-seconds. This active polling consumes noticeable CPU time and reduces system's response time for a communication-intensive applications. This design is due to thread unawareness of the underlying MPI implementation. This overhead is eliminated in the D'Caml/TCP configuration.

4 Evaluation

This section presents some of preliminary evaluation results of the D'Caml/MPI.

Speed-up: The first result shown in Figure 6-(a) is obtained from running the program shown in Figure 2 for 500×500 matrix on varying number of nodes in the cluster. The cluster consists of 15 nodes of SPARC Station 20 running Solaris 2.5.1 connected with 100 Base-T Ethernet. The result is the average of 10 runs of the program ranging the number of clients for 1, 3, 6, 9, and 12 nodes. The result does not say much but it suggests D'Caml effectively utilizes inherent parallelism in a small scale workstation cluster.



(a) Speed-up (b) Closure size

Fig. 6. Evaluation results

Size of marshaled closures: A closure involves all its free references to other closures, which in turn involves other closures, and so on. The reader may fear that the external representation of closures can be huge making sending over the network impractical. It turned out, however, the extern-ed closures are practically small enough. Figure 6-(b) is the result we obtained from measuring the size for each marshaled string of 262 functions defined in the standard library. About 90% of them occupy less than 100 bytes and all of them fit in an Ether packet whose size is about 1,000 bytes. The reason for their smaller size against our intuition is that highly optimizing closure conversion [1] mechanism of the O'Caml compiler excludes top-level defined functions from closure representation. We have found aggressive use of this technique can further be applied to reduce extern-ed closure size. This technique reduces the largest closure (683 bytes) to 150 bytes and others less than 100 bytes. Currently we have not adopted this optimization because the changes required for closure representation introduces small invocation overhead.

5 Related Work and Conclusions

There are several projects working on distributed functional programming languages. ParaML [2] is a distributed ML based on SPMD computation model and Distributed ML [6] allows channel based inter-node communication. Both of them do not consider heterogeneous environment.

Facile [14] is a mobile programming language that integrates a concurrent calculus with ML. It allows passing ML modules across the heterogeneous network and execute them native by using dynamic compiler and linker. It does not support distributed address space and is incapable to send closures across the network.

The dML [11] incorporates dynamic types in the ML type system and proposes a theoretical foundation of inter-node communication between independent ML programs.

Kali Scheme [5] and NeXeme [10] are distributed dialects of Scheme and Obliq [4] is a mobile programming language. They offer single address space abstractions (or in Obliq's terminology *distributed scope*) and support heterogeneous environment. Kali Scheme and Obliq provides automatic coherence protocol for distributed shared mutable objects but the NeXeme leave this programmers' responsibility allowing development of relaxed coherence protocols. All of them are executed by bytecode interpreters.

An alternative to native implementation like D'Caml is the combination of bytecode compiler and just-in-time technology. Attempts have been made to target functional languages for JVM (e.g, Kawa scheme) but currently JIT fails to accelerate such implementations as efficient as standard implementation. As for distributed implementation, inability to modify the VM leads to significant degradation of execution efficiency due to pointer-testing overhead.

Distributed dialects of Scheme [12, 5] typically prohibit assignment to variables but incorporate mutable data structures called boxes, similar to `ref` type in ML. The purpose of boxes is for optimization of DSM and its coherence protocol [12]. In ML, the type system precisely tells mutability information and thus we can apply the same optimization without affecting the syntax and semantics of the language.

Distributed Caml offers a single address space for native code distributed application that executes on a heterogeneous network cluster. Integration of higher-order programming and communication based on remote closure invocation achieves a flexible and extensible environment for distributed programming. Two important limitation of our system are (1) communication between independently developed Caml programs and (2) the support of mobility. These issues require future work.

Acknowledgment The authors thank anonymous Euro-Par '99 reviewers for giving us precious comments. Tetsu Ushijima implemented earlier version of D'Caml/MPI. The work is funded by Research Institute of Software Engineering, Japan.

References

[1] A. Appel. *Compiling with Continuations*, chapter 10. Cambridge UP, 1992.
[2] P. Bailey et al. Supporting coarse and fine grain parallelism in an extention of ML. In *CONPAR '94, LNCS 854*, pages 693–704, 1994.
[3] D. Bevan. Distributed garbage collection using reference counting. In *Parallel Architectures and Languages, LNCS 258*, pages 176–187, 1987.
[4] L. Cardelli. A language with distributed scope. In *POPL '95*, pages 286–298, 1995.
[5] H. Cejtin, S. Jagannathan, and R. Kelsey. Higher-order distributed objects. *TOPLAS*, 17(5):704–739, 1995.
[6] R. Cooper and C. Krumvieda. Distributed programming with asynchronous ordered channels in distributed ML. In *Workshop on ML and Applications*, 1992.
[7] Message Passing Interface Forum. A Message Passing Interface Standard. Technical Report CS-94-230, Dept. CS, Univ. Tennessee, 1994.
[8] R. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans. on Programming Languages and Systems*, pages 501–538, October 1985.
[9] X. Leroy. *Objective Caml*, 1997. Available at http://pauillac.inria.fr/ocaml/.
[10] L. Moreau et al. NeXeme: A distributed Scheme based on Nexus. In *EuroPar '97, LNCS 1300*, pages 581–590, 1997.

[11] A. Ohori and K. Kato. Semantics for communication primitives in a polymorphic language. In *POPL'93*, pages 99–112, 1993.
[12] C. Queinnec. DMEROON: a distributed class-based causally-coherent data model. In *Parallel Symbolic Computing: Languages, Systems, and Applications, LNCS 1068*, 1995.
[13] J. Reppy. CML: A higher-order concurrent language. In *PLDI '91*, pages 293–306, 1991.
[14] B. Thomsen et al. Facile antigua release – programming guide. Technical Report ECRC-93-20, ECRC, 1993.