Functional Programs on Clusters*

Viktória Zsók, Zoltán Horváth, Zoltán Varga

Department of General Computer Science
University of Eötvös Loránd, Budapest
e-mail: hz@inf.elte.hu, Zoltan.2.Varga@nokia.com, zsv@inf.elte.hu

Abstract. The implemented Clean-CORBA and Haskell-CORBA interfaces open a way for developing parallel and distributed applications on clusters consisting of components written in functional programming languages, like Clean and Haskell.
We focus on a specific application of this tool in this paper. We design and implement an abstract communication layer based on CORBA server objects. Using this layer we can build up computations in form of distributed process-networks consisting of components written in several programming languages, some components written in functional style in Clean, while other components written in an object-oriented language like Java or C++.
The speed-up of computations is investigated using a simple example.

1 Introduction

One of the easiest way to provide powerful infrastructure for parallel and distributed computing is to build a cluster and interconnect clusters via the internet into a Grid.

Functional programming is very suitable for expressing parallelism. Composition of functions is an associative operation, so evaluation of functional programs can be done in parallel or distributed way. So functional programs are inherently parallel but the evaluation in parallel of an expression is not always worthwhile.

There are several elements in the functional programming language Clean which support to control parallel and distributed evaluation and communication [11,1,13]. Also the Haskell language has several dialects with parallel features: GpH [9], pH [10], Eden [4], Distributed Haskell with Ports [8]. These solutions are different in efficiency and in power of expressiveness and require different hardware and software infrastructure.

A higher degree of abstraction level expressing parallelism can be achieved by parameterizing computational skeletons with evaluation strategies. Evaluation strategies [12,6] may be applied in parallel computations separating dynamic evaluation issues from static requirements. Evaluation strategies are appropriate tools in order to control the evaluation order and degree, the dynamic behaviour and the parallelism [12]. A skeleton is a parameterized algorithmic scheme. Skeletons in functional languages are higher order functions parameterized by functions, types and evaluation strategies.

* Position paper. Technical paper to appear in Proceedings of SPLST'03 [7]. Supported by the Hungarian National Science Research Grant (OTKA), Grant No. T037742 and by IKTA 89/2002 (JiniGrid).

There were several studies regarding skeletons [3,12] from the apparently very simple but very useful skeleton `parmap`, to the more complex skeletons like the parallel elementwise processing [6].

Functional programs can also be developed and tested on cluster systems. The first study was the comparison of the GpH and the Eden languages regarding their performances [9]. The GpH and Eden comparison was done on a Beowulf cluster. A Haskell version of parallel elementwise processing implemented on a cluster was presented in [5].

Our intention is to test and to verify how the Clean functional programming language fits into the parallel programming framework offered by clusters. We use an architecture, which allows to build up applications consisting components written in several programming languages, some components written in pure functional style for example in Clean, while other components written in an object-oriented language.

A Clean-CORBA interface [13] is used as an infrastructure for parallel communication. The interface implements a language mapping from Clean to IDL. Our Clean-CORBA interface uses the MICO CORBA implementation and allows to write CORBA clients and servers in the lazy functional programming language Clean.

We designed and implemented an abstract communication layer based on this software architecture. The distributed computation is built up from components implemented in form of CORBA clients. These components communicate via channels which are CORBA server objects. The channel object is written in two variants, in Clean and in C++. The clients may be written in any language with CORBA interface.

We have chosen an implementation of a pipeline computation as an example in this paper to present the main features of our approach. We implemented the clients of this example in functional style, in Clean. We measured the performance of the application on a cluster consisting of 16 processors.

Section 2 describes the Clean-CORBA interface. The mapping from the CORBA IDL to the Clean functional language is described according to the language elements.

The third section presents an implementation of asynchronous communication channel, which can be used for connecting Clean programs and other programs in a cluster environment.

The pipeline skeleton is very suitable for the computation of functions which can be built by the composition of small components, for the detailed description of the problem see the fourth section.

The last section (section 5) concludes.

2 Clean-CORBA interface

To access CORBA from a programming language a language mapping for the particular language is needed. This mapping should contain the following elements: an IDL module mapping to the specific language, the simple and composed types of IDL association with the types of the language, the projections of the definitions and operations of the IDL interface, the implementation of services offered by the CORBA server and of the pseudo-objects of the CORBA into the language.

In Clean-CORBA interface the operations are associated with functions, CORBA objects with Clean records. For communication through TCP ports and for IP identification the services of MICO Binder are used. Interfaces are generated differently for clients and for servers.

The identifiers of the IDL are the same in Clean, the names of the modules are included in the identifiers. IDL constants are mapped to Clean constants.

The different integer types are associated with the `Int` type of Clean, in the same way the real types are projected into the `Real` type of the Clean language.

Enumeration types are mapped to Clean algebraic types.

IDL Structures are mapped to Clean records. The field names remain the same. If the structure contains an 'anonymous' field (like `sequence <long> m3`), then the IDL compiler will create a new Clean type (in this case `Foo__m3`), and this will be the type of the corresponding field in the Clean record. Recursive structures and unions are supported too. IDL unions map to Clean algebraic data types, with one data constructor for each legal discriminator value. IDL sequences map to Clean lists.

The most interesting is the mapping of `TypeCode`, which gives us information about the IDL types during runtime.

IDL Interfaces map to abstract Clean types, which contain the object reference in their hidden parts. Each interface type has a corresponding `<T>__nil` function which returns a NIL object reference of the given type. Conversions between interface types are supported through `<T>__narrow` and `<T>__widen` functions generated by the IDL compiler.

Each IDL operation maps to a Clean function which performs the CORBA call (for examples see [13,7]). The first argument of each function is the receiver CORBA object. Since these functions have side effects, they both take and return a unique `World` argument which represents the environment of a Clean program. The operation my fail, so the result belongs to the algebraic type `ResultOrException`, which is an union type. If the IDL operation has `out` or `inout` arguments, the functions return them, too. For example:

```
Account_balance2 :: Account *World
->(((ResultOrException (CORBA_Void,CORBA_Long) CORBAException),
     *World))
```

For each IDL attribute, the IDL compiler will generate both a getter and a setter function.

The Dynamic Invocation Interface (DII) is supported through the following function:

```
CORBA_invoke :: CORBA_Object String [CORBAArg] TypeCode [TypeCode]
*World -> (Any, CORBAException, [CORBAArg], *World)
```

The meaning of the arguments: target CORBA object, the name of the operation, a list of the arguments, return type of the operation, typecodes of IDL exceptions, the unique environment: the world.

The result is a tuple with the following parts: the return value of the operation, the exception raised by the operation, if any, the value of the `out` and `inout` arguments, the new World.

The server side mapping uses a simplified version of the Object IO framework [1]. The IDL compiler generates servant types for each IDL interface. A servant is a record type with one field for each IDL operation in the interface. The programmer must create an instance of this servant type, and register it with the system before it can answer CORBA requests.

The implementation consists of a CORBA-CLEAN interface library, and an IDL-TO-CLEAN compiler. The interface library consists of three layers:

1. The lowest layer is a collection of C functions giving access to CORBA functionality.
2. The middle layer simply consists of Clean wrapper functions around the C functions in the previous layer.
3. The third layer contains the high level interface described above.

The implementation uses CORBA DII and DSI for communication, similarly to the MICO-TCL interface software TclMico.

The IDL compiler works by first uploading the contents of the IDL file into a CORBA Interface Repository daemon, then reading this data using normal CORBA calls into an intermediate representation, and finally generating Clean code.

For detailed description end examples of the mapping see [13,7].

3 The implementation of a channel object

Many problems can be viewed as networks of message-communicating processes, therefore it is very useful to implement an abstract channel object for asynchronous message passing.

To interconnect processes or distributed programs we can implement communication primitives for asynchronous message passing using CORBA server objects. We store the messages in the local state of the server.

The program has to import the `channel` interface, which defines the channel operations. The program also has to import the Clean standard environment and the `Corba` package. These are the basic modules for our Clean-CORBA interface.

The initialization of the CORBA system uses the `CORBA_ORB_init` function, which returns a `CORBA_ORB` object. `CORBA_Server_run` initializes the CORBA server.

In our model the channel initializes the ORB and starts a CORBA event handler. By the `ServerInit` we create a servant, which will be registered by the ORB. `ServerInit` transforms the general object reference into the desired type. The event handler system will assure that the requests of the clients are passed to the servant objects.

```
Start w
   # (orb,_,w) = CORBA_ORB_init args w
   = CORBA_Server_run orb Void ServerInit w
where
```

```
ServerInit ps w
 # (obj, ps, w) = Channel__servant_open ps servant w
 # w
   = WriteIORToFile (CORBA_Server_get_orb ps) obj "channel.ior" w
 = (ps, w)
servant = { Channel__servant |
                      ls              = messages,
                      impl_send       = my_send,
                      impl_receive    = my_receive,
                      }
       my_send (ls, ps) what w
               = ((ls ++ [what], ps), Result Void , w)
       my_receive ([x:xs], ps) w
               = ((xs, ps), Result x, w)
```

`Channel__servant_open` registers the servant at the IO system. The servant defines the operations of the channel. These operations are state transition functions, which modifies the local state of the channel (`ls`). The `messages` is the sequence containing the elements of the channel. The function `my_send` is the implementation of the channel operation `send` and adds to the sequence an element sent by the client. The `my_receive` function implements the channel function `receive` and sends to the client one data from the sequence.

4 The pipeline skeleton

The pipeline skeleton is a special type of process network usually applied for calculating a composite function. The processes are organized linearly. A processes running on a pipeline element calculates a component function and sends intermediate results to its immediate successor. The data input is processed at the beginning of the pipeline.

We consider a simple description of the pipeline problem [2].

Let $D =\ll d_0, d_1, \ldots, d_M \gg$ be a sequence of data, where $M \gg N$, and let $F =\ll f_0, f_1, \ldots, f_N \gg$ be a sequence of functions.

Let $f^i(x)$ denote $f_i(f_{i-1}(\ldots f_0(x)\ldots))$; we assume that $f^i(x)$ is defined for all i, $0 \leq i \leq N$ and all x in D.

We compute the sequence $f^N(D)$, where $f^N(D) =\ll f^N(d_0), \ldots, f^N(d_M) \gg$.

The pipeline problem is implemented in the following form: the Clean programs are Corba-clients and calculate the components of F.

The computation can be parameterized by the component function f_i and by the type of its argument (skeleton). The send and receive functions are implemented by the abstract channel CORBA server, the object presented in the previous section.

For sending data on the channel we have the following function:

```
sendf x obj w
    # (Result l, w) = Channel_full obj w
    | l             = sendf x obj w
    = Channel_send obj (f x) w
```

The function checks if the sequence of data is full. In case is full will try again, in case it is not full will send the data to the server object. For receiving data we have the following function:

```
receivef obj w
    # (Result l, w) = Channel_empty obj w
    | l             = receivef obj w
    = Channel_receive obj w
```

The function verifies if the sequence is empty. If it is empty then will try again, otherwise receives a data from the server.

As an example we compute $sin(x) \approx \sum_{i=0}^{n} (-1)^i * \frac{x^{2i+1}}{(2i+1)!}$. For this we use the following data structure: $d = (xx : Real, s : Real, e : \{1.0, -1.0\}, h : Real)$. The function $sin(x) \approx sin_n \circ \ldots \circ sin_0(x)$, where

$sin_0(x) = (x^2, x, -1.0, x)$

$sin_i(d) = (d.xx, d.s + d.e * d.h * \frac{d.xx}{(2i)*(2i+1)}, d.e * (-1), d.h * \frac{d.xx}{(2i)*(2i+1)})$

$sin_n(d) = d.s + d.e * d.h * \frac{d.xx}{(2n)*(2n+1)}$

The following lemma can be proved:

$f^i(x) = f_i \circ \ldots \circ f_0(x) = (x^2, \sum_{j=0}^{i} (-1)^j * \frac{x^{2j+1}}{(2j+1)!}, (-1)^{i+1}, \frac{x^{2i+1}}{(2i+1)!})$

for all $i = 0, \ldots, n-1$.

According to the lemma the pipeline skeleton will produce a correct result.

The evaluation order of Clean programs is lazy, so the evaluation of some expressions may be postponed by the run-time system. In case of distributed applications the order of evaluation may be important in several cases, so for some expressions a strict evaluation should be enforced. Functions returning the value of the system clock has to be evaluated strictly for example.

5 Performance measurement

The cost of the communication via the CORBA server objects is relatively high compared to the cost of this simple computation. If we slow down the computation sin_i functions simulating a most complex computation (we apply a weighted function), then we can observe even a small speedup (see figure 1).

Fig. 1. Speedup with different number of input data

6 Conclusions

The presented Clean-CORBA interface and the abstract communication layer on top of it is applicable for implementing computations in form of distributed process-networks. The application may consist of components written in several programming languages. We presented a simple pipeline computation written in the pure functional language Clean. We observed a small speed-up of this computation on a 16 processor cluster.

References

1. Achten, P., Wierich, M.:*A Tutorial to the Clean Object I/O Library*, University of Nijmegen, 2000. http://www.cs.kun.nl/~ clean
2. Chandy, K. M., Misra, J.: *Parallel Program Design*, Addison-Wesley, 1989.
3. Cole, M.: Algorithmic Skeletons, In: Hammond, K., Michaelson, G., eds., *Research Directions in Parallel Functional Programming*, pp. 289-303, Springer-Verlag, 1999.
4. Galán, L.A., Pareja, C., Peña, R.: Functional Skeletons Generate Process Topologies in Eden, In: *Int. Symp. on Programming Languages, Implementations Logics and Programs PLILP'96*, Aachen, Germany, LNCS, Vol. 1140, pp. 289-303, Springer-Verlag, 1996.
5. Horváth Z., Hernyák Z., Kozsik T., Tejfel M., Ulbert A.: A Data Intensive Application on a Cluster - Parallel Elementwise Processing, In: Kacsuk P., Kranzlmüller D., Neméth Zs., Volkert J. (eds.): *Distributed and Parallel System - Cluster and Grid Computing, Proc. of 4th Austrian-Hungarian Workshop on Distributed and Parallel Systems*, Kluwer Academic Publishers, The Kluwer International Series in Engineering and Computer Science, Vol. 706, pp. 46-53, Linz, Austria, September 29-October 2, 2002.
6. Horváth Z., Zsók V., Serrarens, P., Plasmeijer, R.: Parallel Elementwise Processable Functions in Concurrent Clean, to appear in *Computers & Mathematics with Applications*, Elsevier.
7. Horváth Z., Varga Z., Zsók V.: Clean-CORBA Interface for Parallel Functional Programming on Clusters. To appear in: Proceedings of SPLST'03.
8. Huch, F., Norbisrath, U.: Distributed Programming in Haskell with Ports, *Implementation of Functional Programming Languages, 12th International Workshop, IFL2000*, Aachen, Germany, September 4-7, 2000, LNCS, Vol. 2011, pp. 107-121, Springer 2001, http://www-i2.informatik.rwth-aachen.de/hutch/distributedHaskell.
9. Loidl, H.W., Klusik, U., Hammond, K., Loogen, R., Trinder, P.W.: GpH and Eden: Comparing Two Parallel Functional Languages on a Beowulf Cluster in Gilmore, S. (ed.): *Trends in Functional Programming*, Vol. 2, pp. 39-52, Intellect, 2001.
10. Rishiyur S. Nikhil, Arvind: *Implicit Parallel Programming in pH*, Morgan Kaufmann, 2001.
11. Serrarens, P.R.: *Communication Issues in Distributed Functional Computing*, PhD Thesis, Catholic University of Nijmegen, 2001.
12. Trinder, P.W., Hammond, K., Loidl, H.W., Peyton Jones, S.J.: Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, Vol. 8, No. 1, pp. 23-60, 1998.
13. Varga Z.: Clean-CORBA Interface, Master thesis, University of Eötvös Loránd, Budapest, 2000. (Supervisor: Horváth Z.)