Functory: Distributed Computing for the Common Man*

Jean-Christophe Filliâtre K. Kalyanasundaram
CNRS, LRI, Univ Paris-Sud 11, Orsay F-91405
INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893
`filliatr@lri.fr,kalyan.krishnamani@inria.fr`

December 2, 2010

Abstract

We present Functory, a distributed computing library for functional programming languages. The main features of this library include (1) a polymorphic API, (2) several implementations to adapt to different computing environments such as sequential, multi-core or network, and (3) a reliable fault-tolerance mechanism. This paper describes the motivation behind this work, as well as the design and implementation of the library. It also demonstrates the potential of the library using realistic experiments.

1 Introduction

This paper introduces a generic library for distributed computing, targeted at functional programming. This work was initially motivated by the computing needs that exist in our own research team. Our applications include large-scale deductive program verification, which amounts to checking the validity of a large number of logical formulas using a variety of automated theorem provers [7]. Our computing infrastructure consists of a few powerful multi-core machines (typically 8 to 16 cores) and several desktop PCs (typically dual-core). However, for our application needs, there is no library that helps in exploiting such a computing infrastructure in our favorite functional programming language. Hence we designed and implemented such a library, which is the subject of this paper. This library is implemented in OCaml, but the implementation should be straightforward in any functional programming language.

The distributed computing library presented in this paper is not a library that helps in parallelizing computations. Rather, it provides facilities for reliable, distributed execution of parallelizable computations. In particular, it provides a set of APIs that allows the execution of large-scale parallelizable computations, very relevant to our application needs

*This research was partly supported by the French national project U3CAT (*Unification of Critical C Code Analysis Techniques*, ANR-08-SEGI-021).

(and actually relevant to a variety of real-world applications), over multiple cores in the same machine or over a network of machines. The most important features of our library are the following:

- *Genericity*: it allows various patterns of polymorphic computations;
- *Simplicity*: switching between multiple cores on the same machine and a network of machines is as simple as changing a couple of lines of code;
- *Task distribution and fault-tolerance*: it provides automatic task distribution and a robust fault-tolerance mechanism, thereby relieving the user from implementing such routines.

The application domain of such a distributed computing library is manyfold. It is worth noting that the library is not targeted at applications running on server farms, crunching enormous amounts of data, mostly due to the lack of data locality. It is rather more relevant for research labs, to make use of available computing resources efficiently. It serves a variety of users and a wide spectrum of needs, from desktop PCs to networks of machines, and hence the title. Typical applications would involve a large number of computation expensive tasks. This is the case in our research endeavours, that is validating thousands of verification conditions using automated theorem provers.

The remainder of this section introduces our approach to distributed computing in a functional programming setting.

Distributed Computing. A typical distributed computing library, as Functory, provides the following (we borrow some terminology from Google's MapReduce [6]):

- A notion of *tasks* which denote atomic computations to be performed in a distributed manner;
- A set of processes (possibly executing on remote machines) called *workers* that perform the tasks, producing results;
- A single process called a *master* which is in charge of distributing the tasks among the workers and managing results produced by the workers.

In addition to the above, distributed computing environments also implement mechanisms for fault tolerance, efficient storage, and distribution of tasks. This is required to handle network failures that may occur, as well as to optimize the usage of machines in the network. Another concern of importance is the transmission of messages over the network. This requires efficient *marshalling* of data, that is encoding and decoding of data for transmission over different computing environments. It is desirable to maintain architecture independence while transmitting marshalled data, as machines in a distributed computing environment often run on different hardware architectures and make use of different software platforms. For example, machine word size or endianness may be different across machines on the network.

A Functional Programming Approach. Our work was initially inspired by Google's MapReduce[1]. However, our functional programming environment allows us to be more generic. The main idea behind our approach is that workers may implement any polymorphic function:

worker: $\alpha \rightarrow \beta$

where α denotes the type of tasks and β the type of results. Then the master is a function to handle the results together with a list of initial tasks:

master: $(\alpha \rightarrow \beta \rightarrow \alpha$ list$) \rightarrow \alpha$ list $\rightarrow$ unit

The function passed to the master is applied whenever a result is available and may in turn generate new tasks (hence the return type α list). The master is executed as long as there are pending tasks.

Our library makes use of OCaml's marshalling capabilities as much as possible. Whenever master and worker executables are exactly the same, we can marshal polymorphic values and closures. However, it is not always possible to have master and workers running the same executable. In this case, we cannot marshal closures anymore but we can still marshal polymorphic values as long as the same version of OCaml is used to compile master and workers. When different versions of OCaml are used, we can no longer marshal values but we can still transmit strings between master and workers. Our library adapts to all these situations, by providing several APIs.

The paper is organized as follows. In Section 2, we describe the main idea behind the generic API, and introduce master and worker programs. In Section 3, we present some additional functions provided by Functory, which are derived from the generic API. Section 4 illustrates the API usage with several case studies. Section 5 delves into the implementation details of our library, while Section 6 illustrates the potential of the presented library through experimental evaluation. We compare our approach with relevant related work in Section 7 and outline our future work.

2 API

In this section, we describe the generic API which we shall extend with several specialized functions in the next section. The main function in our API follows the idea sketched in the introduction. It has the following signature:

```
val compute :
  worker:(α → β) →
  master:(α × γ → β → (α × γ) list) →
  (α × γ) list → unit
```

Tasks are pairs, of type $\alpha \times \gamma$, where the first component is passed to the worker and the second component is local to the master. The worker function should be pure[2] and is executed in parallel in all worker processes. The function master, on the contrary, can be impure and is only executed in the master process. The master function typically stores

[1]Ironically, Google's approach itself was inspired by functional programming primitives.

[2]We mean *observationally pure* here but we allow exceptions to be raised to signal failures.

results in some internal data structure. Additionally, it may produce new tasks, as a list of type $(\alpha \times \gamma)$ list, which are then appended to the current set of pending tasks. The next section will describe the usage of this generic interface to perform the traditional map and fold operations.

Actually, our library provides not just a single compute function as above, but instead five different versions depending on the execution context. The first two contexts are the following:

1. **Purely sequential execution:** this is mostly intended to be a reference implementation for performance comparisons, as well as for debugging;

2. **Several cores on the same machine:** this implementation is intended to distribute the computation over a single machine and it makes use of UNIX processes;

The next three implementations are intended for distributing the computation over a network of machines.

3. **Same executable run on master and worker machines:** this implementation makes use of the ability to marshal OCaml closures and polymorphic values. Depending on whether the program is run as a master or as a worker, the relevant arguments of compute are used.

4. **Master and workers are different programs, compiled with the same version of OCaml:** we can no longer marshal closures but we can still marshal polymorphic values. As a consequence, the compute function is split into two polymorphic functions, to implement the master and workers respectively:

   ```
   val Worker.compute : (α → β) → unit → unit
   val Master.compute :
     (α × γ → β → (α × γ) list) →
     (α × γ) list → unit
   ```

5. **Master and workers are different programs, not even compiled with the same version of OCaml:** we can no longer use marshalling, so the compute function is split into two monomorphic functions over strings:

   ```
   val Worker.compute : (string → string) → unit → unit
   val Master.compute :
     (string × γ → string → (string × γ) list) →
     (string × γ) list → unit
   ```

Our library is organized into three modules: Sequential for the pure sequential implementation, Cores for multiple cores on the same machine and Network for a network of machines, respectively. The Network module itself is organized into three sub-modules, called Same, Poly and Mono, corresponding to contexts 3, 4 and 5 above. The next section presents more functions provided by Functory, which are all derived from the generic API just presented.

3 Derived API

In most cases, the easiest way to parallelize an execution it to make use of operations over lists, where processing of the list elements are done in parallel. To facilitate such a processing, we now derive the most commonly used list operations from our generic API.

The most obvious operation is the traditional map operation over lists, that is:

```
val map : (α → β) → α list → β list
```

The next natural operation is a combination of map and fold operations, that is a function like

```
val map_fold : f:(α → β) → fold:(γ → β → γ) → γ → α list → γ
```

which, given two functions, an accumulator a and a list l, computes

$$\mathsf{fold}...(\mathsf{fold}(\mathsf{fold}\ a\ (\mathsf{f}\ x_1))(\mathsf{f}\ x_2))...(\mathsf{f}\ x_n) \tag{1}$$

for some permutation $[x_1, x_2, ..., x_n]$ of the list l. We assume that the f operations are always performed in parallel. Regarding fold operations, we distinguish two cases:

- either fold operations are computationally less expensive than f and we perform them locally;
- or fold operations are computationally expensive and we perform them in parallel.

We thus provide two functions map_local_fold and map_remote_fold.

In the case of map_remote_fold, only one fold operation can be performed at a time (possibly in parallel with f operations), as obvious from (1). However, there are cases where several fold operations can be performed in parallel, as early as intermediate results of fold operations are available. This is the case when fold is an associative operation (which implies that types β and γ are the same). Whenever fold is also commutative, we can perform even more fold operations in parallel. Thus our API provides two functions map_fold_a and map_fold_ac for these two particular cases. The five operations of the derived API are summarized in Figure 1.

These five functions are actually derived from the generic API. The generality is achieved by implementing these derivations as a functor. This functor, Derive, is parameterized by the compute function, as follows:

```
module Derive
  (X : sig
     val compute :
       worker:(α → β) → master:(α × γ → β → (α × γ) list) →
       (α × γ) list → unit
   end) = struct ... end
```

We now explain how each function from Figure 1 is implemented using compute inside the functor body.

```
val map :
  f:(α → β) → α list → β list
val map_local_fold :
  f:(α → β) → fold:(γ → β → γ) →
  γ → α list → γ
val map_remote_fold :
  f:(α → β) → fold:(γ → β → γ) →
  γ → α list → γ
val map_fold_ac :
  f:(α → β) → fold:(β → β → β) →
  β → α list → β
val map_fold_a :
  f:(α → β) → fold:(β → β → β) →
  β → α list → β
```

Figure 1: Derived API.

map. Function map is easily implemented by associating an integer index with each element (of type α) of the input list. Thus the list of tasks passed to compute is a list of pairs of type $\alpha \times$ int. As soon as an intermediate result is available, it is passed to master, which stores it in a local table using the index as a key and generates no new task. Whenever compute returns, we easily recover the output list from the values stored in the table.

map_local_fold. Function map_local_fold is implemented using a local reference storing the current accumulator (of type γ). As soon as an intermediate result is available, the master combines it with the accumulator using the function fold. Whenever compute returns, we simply return the current accumulator.

map_remote_fold. In this case, combining the accumulator with an intermediate result is itself a task to be performed in parallel with f operations. Thus tasks are of two different kinds and we use a sum type to make the distinction. When an intermediate f result is available, the accumulator may be already in use in a fold operation. Hence we need to store the intermediate f result in a table local to the master, to be used as soon as the accumulator becomes available. The master function distinguishes between f and fold results and, accordingly, updates the table and the accumulator. For a f result, either we combine it with the accumulator in a new task or we simply store it in the local table. For a fold result, either we simply store it into the accumulator or we immediately combine it with a pending f result from the table. Whenever compute returns, the accumulator must be available and we simply return it.

map_fold_ac. The function map_fold_ac is easier than the previous one, since we do not need to store more than one intermediate result. Indeed, as soon as two intermediate results are available, we immediately combine them using a fold operation. Thus, whenever the master receives some result, it either stores it or combines it with the accumulator, depending on the availability/non availability of the accumulator. Whenever compute returns, the accumulator must be available and we simply return it.

map_fold_a. Function map_fold_a is definitely the trickiest. Indeed, the operation being only associative, we can only combine *adjacent* intermediate results. In order to perform this, we store intermediate results in a local table according to indices of the input list. More precisely, if we have already computed

$$x_i \oplus x_{i+1} \oplus \cdots \oplus x_j$$

where $\oplus$ denotes the fold operation, then we associate this value with indices i, j. As soon as a result of indices $k, i-1$ or $j+1, k$ is available, it can be combined with the result of i, j. Whenever compute returns, the table should only contain a result corresponding to the full input list and we simply return it.

These five functions are the most natural derivations of the compute functions. There are obviously other possible derivations (e.g. a variant of map_fold where only fold is meaningful); in most cases, they could be derived in a straightforward way from our generic API.

4 Illustrative Case Studies

In this section, we show how to use Functory on several case studies. We focus here on the use of the API. Experimental results of these case studies are given in Section 6. The source code for all these case studies is contained in the distribution, in sub-directory `tests`.

4.1 Matrix Multiplication

As a first example, let us consider matrix multiplication. We assume two matrices a and b, respectively of size n × p and p × m, given as input, as well as a matrix c of size n × m to store the result. Assuming a being a row-matrix and b a column-matrix, the standard matrix multiplication would be as follows:

```
for i = 0 to n-1 do
  for j = 0 to m-1 do
    for k = 0 to p-1 do
      c.(i).(j) ← c.(i).(j) + a.(i).(k) × b.(j).(k)
    done
  done
done
```

and runs in $O(\mathsf{n} \times \mathsf{p} \times \mathsf{m})$, assuming addition and multiplication over coefficients to be denoted + and × respectively. An easy way to distribute this computation is obviously to turn each computation of the inner loop over k into a task. To do so, we first prepare the list of tasks:

```
let tasks =
  let l = ref [] in
  for i = 0 to n-1 do for j = 0 to m-1 do
    tasks := ((a.(i), b.(j)), (i,j)) :: !tasks
  done done;
  !l
```

Each task consists of row a.(i) and column b.(j) as first tuple, together with position (i,j) as a second tuple. The worker is receiving the first tuple as argument and computes the dot product:

```
let worker (ai, bj) =
  let c = ref 0 in
  for k = 0 to p-1 do c := !c + ai.(k) × bj.(k) done;
  !c
```

The master is a one line function which receives the result r from the worker and simply stores it into c, according to the position contained in the input task. It produces no new task.

```
let master (_, (i,j)) r = c.(i).(j) ← r; []
```

Finally, the overall computation is started by invoking compute as follows:

```
let () = compute ~worker ~master !tasks
```

The total number of tasks is n × m.

Using the sequential implementation provided by Functory is as simple as including the following line in the code:

```
open Sequential
```

Then the computation is roughly similar to the standard multiplication described above.

Now, let us assume we want to make use of a 4-core machine to perform the same computation. This is achieved by just replacing the line above by

```
open Cores
let () = set_number_of_cores 4
```

while the rest of the code remains unchanged.

Later, we may want to use a network of machines instead, for example two machines called `orcus` and `belzebuth` with 4 and 8 cores respectively. Similar to the above, we turn the above lines into

```
open Network
let () = declare_workers ~n:4 "orcus"
let () = declare_workers ~n:8 "belzebuth"
open Same
```

Here, Same is a module which is to be used when master and worker are running the same executable. It provides a compute function which has the same signature as in modules Sequential and Cores, so that the rest of the code remains unchanged. Master and worker processes are distinguished at run-time using an environment variable `WORKER` which is set/unset.

If we need to write two different programs for master and worker, for reason of binary incompatibility or any other reason, the library API is providing functions to do that. If master and worker are still compiled with the same version of OCaml, we use the Poly module which provides a polymorphic API. Let us start with the worker program. It now looks like:

```
open Poly
let worker (ai, bj) = ...
let () = Worker.compute worker ()
```

The Worker.compute function enters a loop which waits for tasks sent by the master and returns results computed using worker. The master program is almost the same as before. First, we replace module Same with module Poly:

```
open Network
let () = declare_workers ~n:4 "orcus"
let () = declare_workers ~n:8 "belzebuth"
open Poly
```

Tasks and master function are unchanged:

```
let tasks = ...
let master (_, (i,j)) r = ...
```

Finally, we start the computation with Master.compute, which does not have a worker parameter anymore:

```
let () = Master.compute ~master tasks
```

When master and worker programs are compiled with different versions of the OCaml compiler, our library still provides a monomorphic API over strings. As a consequence, we need to convert tasks and results to and from strings in both master and worker. The modified worker program then looks as follows:

```
open Mono
let worker (ai, bj) = ...
let worker_string s = string_of_coeff (worker (task_of_string s))
let () = Worker.compute worker_string ()
```

The master program is modified in a similar way. We simply replace Poly with Mono and encode/decode coefficients as strings, as follows:

```
let tasks = ... string_of_task ...
let master (_, (i,j)) r = c.(i).(j) ← coeff_of_string r; []
```

where string_of_task and task_of_string are user-defined functions to convert tasks to and from strings.

4.2 N-queens

The next example is the classical N-queens problem, where we compute the total number of ways to place N queens on a $N \times N$ chessboard in such a way no two queens attack each other. We use a standard backtracking algorithm for this problem, which places the queens one by one starting from the first row. Distributing the computation is thus quite easy: we consider all possible ways to place queens on the first D rows and then perform the subsequent search in parallel. Choosing $D = 1$ will result in exactly N tasks; choosing $D = 2$ will result in $N^2 - 3N + 2$ tasks; and so on.

Each task only consists of three integers and its result is one integer, which is the total number of solutions for this task. We make use of function map_local_fold from the derived API, where f is performing the search and fold simply adds the intermediate results. In the network configuration, we make use of the Network.Same module, workers and master being the same executable.

4.3 Mandelbrot Set

Drawing the Mandelbrot set is another classical example that could be distributed easily, since the color of each point can be computed independently of the others. Let us assume we are given the coordinates of the region to be drawn, along with the width w and height h of the resulting image (in pixels). Let us assume the total number of tasks $\mathsf{t} \geq 1$ is given as a parameter. It is immediate to split the image into t sub-images, each of which is computed in parallel with and independently of the others. For instance, the image could be split into horizontal slices or, more generally, into rectangular blocks.

Each task is thus four floating-point numbers denoting the region coordinates, together with two integers denoting the dimensions of the sub-image to be drawn. The result of the task is a matrix of pixels, of size $(\mathsf{w} \times \mathsf{h})/\mathsf{t}$. For instance, drawing a 800×600 image using 20 tasks will result in 20 sub-images of size $176,000$ bytes each, assuming each pixel is encoded in four bytes.

In the network configuration, we deliberately choose to have two different programs for master and workers, using the Network.Poly module (actually, a single source code with a command line option).

4.4 SMT Solvers

Here we demonstrate the potential of our library for our application needs as mentioned in the introduction. In this case study, we consider 80 challenging verification conditions (VC) obtained from the Why platform [7]. Each VC is stored in a file, which is accessible over NFS. The purpose of the experiment is to check the validity of each VC using several automated provers (namely Alt-Ergo, Simplify, Z3 and CVC3), which are all installed on each of the machines in use.

The master program proceeds by reading the file names, turning them into tasks by multiplying them by the number of provers (more than 300 tasks in total). Each worker in turn invokes the given prover on the given file, within a timeout limit. Each task completes with one of the four possible outcomes: *valid*, *unknown* (depending on whether the VC is valid or undecided by the prover), *timeout* and *failure*.

The result of each computation is a pair denoting the status and the time spent in the prover call. The master collects these results and sums up the timings for each prover and each possible status.

5 Implementation Details

We now describe the implementations of the various modules introduced in Section 2. The implementation of the Sequential module is straightforward and does not require any explanation.

5.1 Multiple Cores

The Cores module implements the distributed computing library for multiple cores on the same machine. It provides a function set_number_of_cores: int $\rightarrow$ unit to indicate the number of cores to be used. The number passed to this function may be different from the actual number of cores in the machine; it is rather the number of tasks to be performed simultaneously.

The Cores module is implemented with UNIX processes, using the fork and wait system calls provided by the Unix library of OCaml. The idea is pretty simple. The compute function maintains a global table of pending tasks and keeps track of the number of idle cores. Whenever there is a pending task and an idle core, a new sub-process is created using Unix.fork; once completed, the sub-process marshals the result into a local file. The main process maintains a table mapping each sub-process ID to the input task and the local file name. It waits for any completed sub-process using Unix.wait and recovers the result from the local file. Then function master is applied, which may generate new tasks. The main loop can be depicted in the following way:

```
while pending tasks ∨ pending sub-processes
  while pending tasks ∧ idle cores
    create new sub-process for some task
  wait for any completed sub-process
    push new tasks generated by master
```

We also have an alternative implementation using UNIX pipes instead of local files.

The scheduling of tasks to the different cores is left to the operating system, through Unix.fork. Thus it may be the case that two tasks are executed on the same core, even if the declared number of cores is less or equal than the actual number of cores on the machine.

5.2 Network of Machines

The Network module implements the distributed computing library for a network of machines. It provides a function declare_workers: n:int $\rightarrow$ string $\rightarrow$ unit to fill a table of worker machines.

The Network module is based on a traditional TCP-based client/server architecture, where each worker is a server and the master is the client of each worker. The main execution loop is similar to the one in the Cores module, where distant processes on remote machines correspond to sub-processes and idle cores are the idle cores of remote workers. The master is purely sequential. In particular, when running the user master function, it is not capable of performing any task-related computation. This is not an issue, as we assume the master function not to be time-consuming. The worker, on the other hand, forks a new process to execute the task and hence can communicate with the master during its computation. We subsequently describe issues of message transfer and fault tolerance.

5.2.1 Protocol

Messages sent from master to workers could be any of the following kinds:

Assign(id:int, f:string, x:string) This message assigns a new task to the worker, the task being identified by the unique integer id. The task to be performed is given by strings f and x, which are interpreted depending on the context.

Kill(id:int) This message tells the worker to kill the task identified by id.

Stop This message informs the worker about completion of the computation, so that it may choose to exit.

Ping This message is used to check if the worker is still alive, expecting a Pong message from the worker in return.

Messages sent by workers could be any of the following kinds:

Pong This message is an acknowledgment for a Ping message from the master.

Completed(id:int, s:string) This message indicates the completion of a task identified by id, with result s.

Aborted(id:int) This message informs the master that the task identified by id is aborted, either as a response to a Kill message or because of a worker malfunction.

Our implementation of the protocol works across different architectures, so that master and workers could be run on completely different platforms w.r.t. endianness, version of OCaml and operating system.

5.2.2 Network Sub-modules

As mentioned in Section 2, the Network module actually provides three different implementations, according to three different execution scenarios. There are provided in three sub-modules, as described below.

Same. This module is used when master and workers are running the same executable. The master and workers have to be differentiated in some manner. We use an environment variable `WORKER` for this purpose. When set, it indicates that the executable acts as a worker. At run-time, a worker immediately enters a loop waiting for tasks from the master, without even getting into the user code. As explained in Section 2, the master function has the following signature.

```
val compute :
      worker:(α → β) →
      master:(α × γ → β → (α × γ) list) → (α × γ) list → unit
```

The master uses marshalling to send both a closure of type $\alpha \rightarrow \beta$ and a task of type α to the worker. The resulting strings are passed as argument f and x in message Assign. Similarly, the worker uses marshalling to send back the result of the computation of type β, which is the argument s in message Completed.

Though the ability to run the same executable helps a lot in deploying the program in different machines, it comes at a small price. Since the worker is not getting into the user code, closures which are transmitted from the master cannot refer to global variables in the user code. Indeed, the initialization code for these global variables is never reached on the worker side. For instance, the Mandelbrot set example could be written as follows:

```
let max_iterations = 200
let worker (xmi, xma, ymi, yma, w, h) =
  ... draw sub-image using max_iterations ...
```

That is, the global function worker makes use of the global variable max_iterations. The worker gets the function to compute from the master, namely the closure corresponding to function worker in that case, but on the worker side the initialization of max_iterations is never executed.

One obvious solution is not to use global variables in the worker code. This is not always possible, though. To overcome this, the Same sub-module also provides a Worker.compute function to start the worker loop manually from the user code. This way, it can be started at any point, in particular after the initialization of the required global variables. Master and worker are still running the same executable, but are distinguished using a user-defined way (command-line argument, environment variable, etc.).

There are situations where it is not possible to run the same executable for master and workers. For instance, architectures or operating systems could be different across the network. For that reason, the Network module provides two other implementations.

Poly. When master and workers are compiled with the same version of OCaml, we can no longer marshal closures but we can still marshal polymorphic values. Indeed, an interesting property of marshalling in OCaml is to be fully architecture-independent, as long as a single version of OCaml is used. It is worth pointing out that absence of marshaled closures now enables the use of two different programs for master and workers. This is not mandatory, though, since master and workers could still be distinguished at run-time as in the previous case.

On the worker side, the main loop is started manually using Worker.compute. The computation to be performed on each task is given as an argument to this function. It thus looks as follows:

Worker.compute : $(\alpha \rightarrow \beta) \rightarrow$ unit $\rightarrow$ unit

On the master side, the compute function is simpler than in the previous case, as it has one argument less, and thus has the following signature.

Master.compute : master:$(\alpha \times \gamma \rightarrow \beta \rightarrow (\alpha \times \gamma)$ list$) \rightarrow (\alpha \times \gamma)$ list $\rightarrow$ unit

For realistic applications, where master and workers are completely different programs, possibly written by different teams, this is the module of choice in our library, since it can still pass polymorphic values over the network. The issues of marshalling are automatically taken care of by OCaml run-time.

The derived API presented in Section 3 is adapted to deal with the absence of closures. Exactly as the compute function, each API now takes two forms, one for the master and another for the workers. For example, map_fold_ac takes the following forms.

$$\begin{array}{ll} \mathsf{Worker.map\_fold\_ac} & : \mathsf{f}{:}(\alpha \rightarrow \beta) \rightarrow \mathsf{fold}{:}(\beta \rightarrow \beta \rightarrow \beta) \rightarrow \mathsf{unit} \rightarrow \mathsf{unit} \\ \mathsf{Master.map\_fold\_ac} & : \beta \rightarrow \alpha\ \mathsf{list} \rightarrow \beta \end{array}$$

It is the responsibility of the user to ensure type consistency between master and workers.

Mono. When master and workers are compiled using different versions of OCaml, we can no longer use marshalling. As in the previous case, we split compute into two functions, one for master and one for workers. In addition, values transmitted over the network can only be strings. The signature thus takes the following form.

$$\begin{array}{ll} \mathsf{Worker.compute} & : (\mathsf{string} \rightarrow \mathsf{string}) \rightarrow \mathsf{unit} \rightarrow \mathsf{unit} \\ \mathsf{Master.compute} & : \mathsf{master}{:}(\mathsf{string} \times \gamma \rightarrow \mathsf{string} \rightarrow (\mathsf{string} \times \gamma)\ \mathsf{list}) \rightarrow \\ & \quad (\mathsf{string} \times \gamma)\ \mathsf{list} \rightarrow \mathsf{unit} \end{array}$$

Any other datatype for tasks should be encoded to/from strings. This conversion is left to the user. Note that the second component of each task is still polymorphic (of type γ here), since it is local to the master.

5.2.3 Fault Tolerance

The main issue in any distributed computing environment is the ability to handle faults, which is also a distinguishing feature of our library. The fault tolerance mechanism of Functory is limited to workers; handling master failures is the responsibility of the user, for instance by periodically logging the master's state. Worker faults are mainly of two kinds: either a worker is stopped, and possibly later restarted; or a worker is temporarily or permanently unreachable on the network. To provide fault tolerance, our master implementation is keeping track of the status of each worker. This status is controlled by two timeout parameters T_1 and T_2 and Ping and Pong messages sent by master and workers, respectively. There are four possible statuses for a worker:

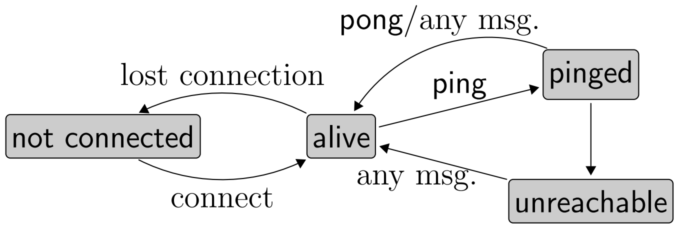
not connected: there is no ongoing TCP connection between the master and the worker;

alive: the worker has sent some message within T_1 seconds;

pinged: the worker has not sent any message within T_1 seconds and the master has sent the worker a Ping message within T_2 seconds;

unreachable: the worker has not yet responded to the Ping message (for more than T_2 seconds).

Whenever we receive a message from a worker, its status changes to alive and its timeout value is reset.



Fault tolerance is achieved by exploiting the status of workers as follows. First, tasks are only assigned to workers with either alive or pinged status. Second, whenever a worker executing a task t moves to status not connected or unreachable, the task t is rescheduled, which means it is put back in the set of pending tasks. Whenever a task is completed, any rescheduled copy of this task is either removed from the set of pending tasks or killed if it was already assigned to another worker.

It is worth noticing that our library is also robust w.r.t. exceptions raised by the user-provided worker function. In that case, an Aborted message is sent to the master and the task is rescheduled. It is the responsibility of the user to handle such exceptions if necessary.

6 Experiments

Our library, Functory, is implemented in OCaml and is available from `http://www.lri.fr/~filliatr/functory/`. In this section, we show the experimental results pertaining to the case studies of Section 4.

6.1 N-queens

The following table shows execution times for various values of N and our three different implementations: Sequential, Cores, and Network. The purpose of this experiment is to measure the speedup w.r.t. the sequential implementation. Therefore, all computations are performed on the same machine, a 8 cores Intel Xeon 3.2 GHz running Debian Linux. The sequential implementation uses a single core. The multi-core implementation uses the 8 cores of the machine. The network implementation uses 8 workers running locally and a master running on a remote machine (which incurs communication cost).

The first column shows the value of N. The number of tasks is shown in second column. Then the last three columns show execution times in seconds for the three implementations. The figures within brackets show the speedup w.r.t. sequential implementation.

N	D	#tasks	Sequential	Cores	Network
16	1	16	15.2	2.04 (7.45×)	2.35 (6.47×)
	2	210	15.2	2.01 (7.56×)	21.80 (0.69×)
17	1	17	107.0	17.20 (6.22×)	16.20 (6.60×)
	2	240	107.0	14.00 (7.64×)	24.90 (4.30×)
18	1	18	787.0	123.00 (6.40×)	125.00 (6.30×)
	2	272	787.0	103.00 (7.64×)	124.00 (6.34×)
19	1	19	6120.0	937.00 (6.53×)	940.00 (6.51×)
	2	306	6130.0	796.00 (7.70×)	819.00 (7.48×)

From the table above, it is clear that the Cores and Network implementations provide a significant speedup. As evident from the last row, the speedup is almost 8, which is also the number of cores we use. It is also evident from the last column that the Network implementation performs significantly better when the computation time dominates in the total execution time. The two extreme cases correspond to the second and the last row: in the second row, the communication time dominates and is in fact more than 91% of the total execution time; on the other hand, for the last row communication time amounts to just 4.6% of the total execution time. As expected, the network implementation is only beneficial when the computation time for each individual task is significant, which is the case in realistic examples.

6.2 Mandelbrot Set

This benchmark consists in drawing the fragment of the Mandelbrot set with lower left corner $(-1.1, 0.2)$ and upper right corner $(-0.8, 0.4)$, as a $9,000 \times 6,000$ image. The sequential computation of this image consumes 29.4 seconds. For Cores and Network implementations, the computation times in seconds are tabulated below.

#cores	#tasks	Cores	Network
2	10	15.8 (1.86×)	20.3 (1.45×)
	30	15.7 (1.87×)	18.7 (1.57×)
	100	16.1 (1.83×)	19.8 (1.48×)
	1000	19.6 (1.50×)	38.6 (0.76×)
4	10	9.50 (3.09×)	14.4 (2.04×)
	30	8.26 (3.56×)	11.4 (2.58×)
	100	8.37 (3.51×)	11.4 (2.58×)
	1000	10.6 (2.77×)	20.5 (1.43×)
8	10	9.40 (3.13×)	12.6 (2.33×)
	30	4.24 (6.93×)	7.6 (3.87×)
	100	4.38 (6.71×)	7.5 (3.92×)
	1000	6.86 (4.29×)	11.3 (2.60×)

The best timings are achieved for the Cores configuration, where communications happen within the same machine and are thus cheaper. There are two significant differences with respect to the n-queens benchmark. On one hand, the number of tasks can be controlled more easily than in the case of n-queens. We experimentally figured out the optimal number of tasks to be 30. One the other hand, each computation result is an image, rather than just an integer as in the case of n-queens. Consequently, communication costs are much greater. In this particular experiment, the total size of the results transmitted is more than 200 Mb.

6.3 Matrix Multiplication

This benchmark was inspired by the PASCO'10 programming contest [4]. It consists of multiplication of two square matrices of dimension 100, that is $n = p = m = 100$, with integer coefficients. Coefficients have several thousands of digits, hence we use GMP [2] to handle operations over coefficients.

We compare the performances of two different implementations. In the first one, called mm1, each task consists of the computation of a single coefficient of the resultant matrix, as described in Section 4.1. In the second one, called mm2, each task consists of the computation of a whole row of the resultant matrix. As a consequence, the total number of tasks in $\mathsf{n} \times \mathsf{m} = 10,000$ for mm1 and only $\mathsf{n} = 100$ for mm2. The experimental results (in seconds) are tabulated below.

	mm1 (10,000 tasks)	mm2 (100 tasks)
Sequential	20.3	20.2
Cores (2 cores)	22.7 (0.89×)	11.3 (1.79×)
(4 cores)	12.3 (1.65×)	6.1 (3.31×)
(6 cores)	8.6 (2.36×)	4.3 (4.70×)
(8 cores)	8.0 (2.54×)	3.5 (5.77×)

We do not include results for the network configuration, as they do not achieve any benefit with respect to the sequential implementation. The reason is that the communication cost dominates the computation cost in such a way that the total execution time is always greater than 30 seconds. Indeed, irrespective of the implementation (mm1 or mm2), the total size of the transmitted data is $O(\mathsf{n} \times \mathsf{m} \times \mathsf{p})$, which in our case amounts to billions of bytes.

A less naive implementation would have the worker read the input matrices only once, *e.g.* from a file, and then have the master send only row and column indices. This would reduce the communication cost to $O(\mathsf{n} \times \mathsf{m})$ only.

6.4 SMT Solvers

As explained in Section 4.4, this benchmark consists of 80 verification conditions, each being checked by 4 different SMT solvers. Each task is executed with a timeout limit of 1 minute. Our computing infrastructure for this experiment consists of 3 machines with 4, 8 and 8 cores respectively. The figure below shows the total time in minutes spent by each prover for each possible outcome.

prover	valid	unknown	timeout	failure
Alt-ergo	406.0	3.0	11400.0	0.0
Simplify	0.5	0.4	1200.0	222.0
Z3	80.7	0.0	1800.0	1695.0
CVC3	303.0	82.7	4200.0	659.0

These figures sum up to more than 6 hours if provers were executed sequentially. However, using our library and our 3-machine infrastructure, it completes in 22 minutes and 37 seconds, giving us a speedup of more than 16×. We are still far away from the ideal ratio of 20× (we are using 20 cores), since some provers are allocating a lot of memory and time spent in system calls is not accounted for in the total observed time. However, a ratio of 16× is already a significant improvement for our day-to-day experiments.

7 Conclusion and Future Work

We presented a distributed programming environment for functional programming. The main features are the genericity of the interface, which makes use of polymorphic higher-order functions, and the ability to easily switch between sequential, multi-core, and network implementations. In particular, Functory allows to use the same executable for master and workers, which makes the deployment of small programs immediate — master and workers being only distinguished by an environment variable. Functory also allows master and workers to be completely different programs, which is ideal for large scale deployment. Another distinguishing feature of our library is a robust fault-tolerance mechanism which relieves the user of cumbersome implementation details. Finally, Functory also allows to cascade several distributed computations inside the same program.

Related Work. Closest to the approach in this paper is Yohann Padioleau's MapReduce implementation in OCaml [12]. It is built on top of OCamlMPI [9], while our approach uses a homemade protocol for message passing. Currently, we have less flexibility w.r.t. deployment of the user program than OCamlMPI; on the other hand, we provide a more generic API together with fault tolerance. There are other distributed computing libraries on top of which one could implement the library discussed in this paper. Jo&Caml [11] is one of them. However, Jo&Caml does not provide fault tolerance, which is indispensable in a distributed setting. The user has to include code for fault tolerance, as already demonstrated in some Jo&Caml experiments [10].

There are other implementations of distributed computing in the context of functional programming. One is the Disco project [3], which implements exactly Google's MapReduce in Erlang [1]. Our library, on the contrary, is not an OCaml implementation of Google's MapReduce. There are other ways to exploit multi-core architectures. One of these is data parallelism, which is also relevant in the functional programming setting [8]. Our work does not target data parallelism at all.

Future Work. There are still some interesting features that could be added to our library.

- One is the ability to efficiently assign tasks to workers depending on resource parameters, such as data locality, CPU power, memory, etc. This could be achieved by providing the user with the means to control task scheduling. This would enable Functory to scale up to MapReduce-like applications.

  Currently, without any information about the tasks, the scheduling is completely arbitrary. In both Cores and Network modules, we use traditional queues for the pending tasks; in particular, new tasks produced by the master are appended at the end of the queue.

- Another interesting feature could be the ability to add or remove machines dynamically. Currently, our library assumes that the list of machines to be used is given *a priori*, as part of the code. An alternative would be to read machine names from a file, watched periodically by the master.

- Our library provides limited support for displaying real-time information about computations and communications. Processing and storing information about workers and tasks locally in the master is straightforward; monitoring it in real-time could be done using Ocamlviz [5].
- One very nice feature of Google's MapReduce is the possibility to use redundantly several idle workers on the same tasks for speedup when reaching the end of computation. Since we already have the fault tolerance implemented, this optimization should be straightforward to add to our library.

We intend to enrich our library with all above features.

Acknowledgements. The authors are grateful to the ProVal team for support and comments on early versions of the library and of this paper.

References

[1] The Erlang Programming Language. `http://www.erlang.org/`.

[2] The GNU Multiple Precision Arithmetic Library. `http://gmplib.org/`.

[3] The Disco Project, 2009. `http://discoproject.org/`.

[4] Parallel Symbolic Computation 2010 (PASCO), 2010. `http://pasco2010.imag.fr/`.

[5] Sylvain Conchon, Jean-Christophe Filliâtre, Fabrice Le Fessant, Julien Robert, and Guillaume Von Tokarski. Observation temps-rel de programmes Caml. In *Vingt-et-unièmes Journées Francophones des Langages Applicatifs*, Vieux-Port La Ciotat, France, January 2010. INRIA.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[7] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.

[8] Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *FSTTCS*, volume 2 of *LIPIcs*, pages 383–414. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.

[9] Xavier Leroy. OCamlMPI: Interface with the MPI Message-passing Interface. `http://pauillac.inria.fr/~xleroy/software.html`.

[10] Louis Mandel and Luc Maranget. Programming in JoCaml (tool demonstration). In *17th European Symposium on Programming (ESOP 2008)*, pages 108–111, Budapest, Hungary, April 2008.

[11] Louis Mandel and Luc Maranget. The Jo&Caml Language, 2008. `http://jocaml.inria.fr/`.

[12] Yoann Padioleau. A poor's man MapReduce for OCaml, 2009. `http://www.padator.org/ocaml/mapreduce.pdf`.