Functory: A Distributed Computing Library for Objective Caml [⋆]

Jean-Christophe Filliâtre and 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

Abstract. We present Functory, a distributed computing library for Objective Caml. The main features of this library include (1) a polymorphic API, (2) several implementations to adapt to different deployment scenarios 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 Functory, a generic library for distributed computing for a widely used functional programming language, Objective Caml (OCaml for short). 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, no existing library provides a polymorphic API with usual map/fold higher-order operations, built-in fault-tolerance, and the ability to easily switch between multi-core and network infrastructures. Hence we designed and implemented such a library, which is the subject of this paper. The library is available at `http://functory.lri.fr/`.

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 user-friendly APIs that allows distributed execution of large-scale parallelizable computations, very relevant to our application needs (and also relevant to a variety of real-world applications). Further, the distributed execution could be over multiple cores in the same machine or over a network of machines. The most important features of our library are the following:

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

- *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 serves a variety of users and a wide spectrum of needs, from desktop PCs to networks of machines. Typical applications would involve executing a large number of computationally expensive tasks in a resource-optimal and time-efficient manner. This is also the case in our research endeavours, that is validating thousands of verification conditions using automated theorem provers, utilizing the computing infrastructure to the maximum. It is worth noting that Functory is not targeted at applications running on server farms, crunching enormous amounts of data, such as Google's MapReduce [6].

In the following, we introduce our approach to distributed computing in a functional programming setting and distinguish it from related work.

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

- 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 *marshaling* 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

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

more generic. The main idea behind our approach is that workers may implement any polymorphic function:

```
worker: 'a -> 'b
```

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

```
master: ('a -> 'b -> 'a list) -> 'a list -> unit
```

The function passed to the master is applied whenever a result is available. The first argument is the task (of type 'a) and the second one its result (of type 'b). It may in turn generate new tasks, hence the return type 'a list. The master is executed as long as there are pending tasks.

Our library makes use of OCaml's marshaling 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.

Related Work. In order to compare and better distinguish Functory from others work with related goals and motivations, we can broadly classify the related work in this domain into:

1. *Distributed Functional Languages (DFLs)* — functional languages that provide built-in primitives for distribution. Examples include ML5, JoCaml, Glasgow Distributed Haskell, Erlang, etc.
2. *Libraries for existing functional languages* — that could be readily used in order to avoid implementing details like task distribution, fault-tolerance, socket programming, etc.

Functory belongs to the second category. For reasons of completeness, though, we first describe some existing DFLs related to functional programming.

JoCaml is one of the DFLs which provides communication primitives (like channels) for facilitating transmission of computations. However, it does not provide ready-made language features for fault-tolerance, which is indispensable in a distributed setting. The user has to include code for fault-tolerance, as already demonstrated in some JoCaml library [10]. ML5 [11], a variant of ML, is a programming language for distributed computing, specialized for web programming. It provides primitives for transferring control between the client and the server, as well as low-level primitives for marshaling the data. As in the case before, ML5 is a programming language that offers primitives for code mobility, and the code for distribution of computation and fault-tolerance has to be included by the user. ML5 implements type-safe marshaling and Functory does not, though an existing type-safe marshaling library could be used with Functory. Glasgow

Distributed Haskell (GdH) [13] is a pure distributed functional language that is built on top of Glasgow Haskell and provides features for distributed computing. It is an extension of both Glasgow Parallel Haskell, that supports only one process and multiple threads and Concurrent Haskell that supports multiple processes. It also offers features for fault-tolerance - error detection and error recovery primitives in the language.

CamlP3l [1] mixes the features of functional programming with predefined patterns for parallel computation to offer a parallel programming environment. Again, it is a programming language offering primitives for distributing computation to parallel processes and also to merge the results from parallel executions. Erlang [3] is a programming language which has features for distribution and fault-tolerance. In particular, it has features for task distribution and is more well-known for its rich error detection primitives and the ability to support hot-swapping. The error detection primitives of Erlang allow nodes to monitor processes in other nodes and also facilitate automatic migration of tasks in failed nodes to recovered or active nodes.

Any DFL above could have been used to implement our library. Our motivation, though, was neither to implement our system using any existing DFL nor to come up with a new DFL. The goal of Functory is rather to provide the users of an *existing* general-purpose functional programming language, namely OCaml, high-level user-friendly APIs that hide the messy details of task distribution and fault-tolerance. We now turn to distributed computing libraries for general purpose functional languages and weed out the distinguishing features of Functory.

There are several implementations of Google's MapReduce in functional programming languages. But Functory was just inspired by Google's MapReduce and is not exactly a MapReduce implementation. The simplest difference comes from the very fact that Functory does not operate on key/value pairs. PlasmaMR [2] is an OCaml implementation of Google's MapReduce on a distributed file system PlasmaFS. It is able to use PlasmaFS to its advantage — the ability of the file system to handle large files and query functions that implement data locality to optimize network traffic. However, PlasmaMR does not support fault-tolerance which is indispensable in any distributed computing application. Another MapReduce implementation in OCaml is Yohann Padioleau's [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. We feel that an indispensible need for any distributed computing library is fault-tolerance, and using a homemade protocol enables us to tune our implementation to our needs of fault-tolerance.

The iTask system [8] is a library for the functional language 'Clean' targeted at distributed workflow management. The library provides a set of combinators (some of which perform map/fold operations) that facilitate applications running in different nodes of a distributed system to communicate, exchange information and coordinate their computations in a type-safe manner.

2 API

This section describes our API. We start from a simple API which is reduced to a single higher-order polymorphic function. Then we explain how this function is actually implemented in terms of low-level primitives, which are also provided in our API. Conversely, we also explain how the same function can be used to implement high-level distribution functions for map and fold operations. Finally, we explain how our API is implemented in five different ways, according to five different deployment scenarios.

2.1 A Generic Distribution Function

The generic distribution function in our API follows the idea sketched in the introduction. It has the following signature:

```
val compute:
  worker:('a -> 'b) ->
  master:('a * 'c -> 'b -> ('a * 'c) list) -> ('a * 'c) list -> unit
```

Tasks are pairs, of type `'a * 'c`, 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 sequentially in the master process. The `master` function typically stores results in some internal data structure. Additionally, it may produce new tasks, as a list of type `('a * 'c) list`, which are then appended to the current set of pending tasks.

2.2 Low-level Primitives

The function `compute` above can actually be implemented in terms of low-level primitives, such as adding a task, adding a worker, performing some communication between master and workers, etc. These primitives are provided in our API, such that the user can interact with the execution of the distributed computation. For instance, a monitoring-like application can use these primitives to allow observation and modification of resources (tasks, workers) during the course of a computation. A type for distributed computations is introduced:

```
type ('a, 'c) computation
```

A computation is created with a function `create`, which accepts the same `worker` and `master` as `compute`:

```
val create: worker:('a -> 'b) ->
  master:('a * 'c -> 'b -> ('a * 'c) list) -> ('a, 'c) computation
```

Contrary to `compute`, it takes no list of tasks and returns immediately. Tasks can be added later using the following function:

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

```
  val add_task: ('a, 'c) computation -> 'a * 'c -> unit
```

A function is provided to perform *one step* of a given computation:

```
  val one_step: ('a, 'c) computation -> unit
```

Calling this function results in one exchange of messages between master and workers: task assignments to workers, results returned to the master, etc. A few other functions are provided, such as status to query the status of a computation, clear to remove all tasks, etc.

Using these low-level primitives, it is straightforward to implement the compute function. Basically, it is as simple as the following:

```
let compute ~worker ~master tasks =
  let c = create worker master in
  List.iter (add_task c) tasks;
  while status c = Running do one_step c done
```

2.3 High-level API

In most cases, the easiest way to parallelize an execution is to make use of operations over lists, where processing of the list elements are done in parallel. To facilitate such a processing, our library provides most commonly used list operations, all implemented using our generic compute function.

The most obvious operation is the traditional map operation over lists, that is val map: f:('a -> 'b) -> 'a list -> 'b list. Each task consists of the application of function f to a list element. More interesting is a combination of map and fold operations. For instance, we provide different flavors of function

```
  val map_fold: f:('a -> 'b) -> fold:('c -> 'b -> 'c) -> 'c -> 'a list -> 'c
```

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. Thus we 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 'b and 'c 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, with types

```
val map_fold_ac, map_fold_a:
  f:('a -> 'b) -> fold:('b -> 'b -> 'b) -> 'b -> 'a list -> 'b
```

It is rather straightforward to derive these five functions from the generic compute function; we invite readers interested in details to refer to the source code.

2.4 Deployment Scenarios

Actually, our library provides not just one implementation for the API above, but instead five different implementations depending on the deployment scenario. The first two scenarios 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 scenarios 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.
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. API functions are split into two sets, used to implement master and workers respectively.
5. **Master and workers are different programs, not even compiled with the same version of OCaml:** we can no longer use marshaling, so API functions are restricted to work on strings instead of polymorphic values.

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.

2.5 Several Libraries in One

From the description above, it is clear that our library provides several APIs of different granularities, as well as several implementations for various deployment scenarios. Most combinations are meaningful, resulting in thirteen possible different ways of using our library. For instance, one may use the low-level API on a single multi-core machine, or use the high-level API on a network of machines all running the same executable, etc. From the implementation point of view, there is almost no code duplication. We are using OCaml functors to derive specific implementations from generic ones.

3 Implementation Details

The implementation of the Sequential module is straightforward and does not require any explanation. The Cores module is implemented with UNIX processes, using the fork and wait system calls provided by the Unix library of OCaml. We do not describe this implementation but rather focus on the more interesting module Network.

3.1 Marshaling

As mentioned in Section 2, the Network module actually provides three different implementations as sub-modules, according to three different execution scenarios, the details of which are presented 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 runtime, 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:('a -> 'b) ->
  master:('a * 'c -> 'b -> ('a * 'c) list) -> ('a * 'c) list -> unit
```

The master uses marshaling to send both a closure of type 'a -> 'b and a task of type 'a to the worker. The resulting strings are passed as argument f and x in message Assign. Similarly, the worker uses marshaling to send back the result of the computation of type 'b, which is the argument s in message Completed. These messages are described in detail in Section 3.2.

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, some code for drawing Mandelbrot's set could be written as follows:

```
let max_iterations = 200
let worker si = ... draw sub-image si 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 marshaling 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 runtime 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: ('a -> 'b) -> unit -> 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:('a * 'c -> 'b -> ('a * 'c) list) -> ('a * 'c) list -> 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 marshaling are automatically taken care of by the OCaml runtime.

The derived API presented in Section 2.3 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.

```
Worker.map_fold_ac: f:('a -> 'b) -> fold:('b -> 'b -> 'b) -> unit
Master.map_fold_ac: 'b -> 'a list -> 'b
```

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

Mono. When master and workers are compiled using different versions of OCaml, we can no longer use marshaling. 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.

```
Worker.compute: (string -> string) -> unit
Master.compute: master:(string * 'c -> string -> (string * 'c) list) ->
  (string * 'c) list -> unit
```

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 'c here), since it is local to the master.

3.2 Protocol

The Network module implements the distributed computing library for a network of machines. It provides a function declare_workers: n:int -> string -> 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.

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.

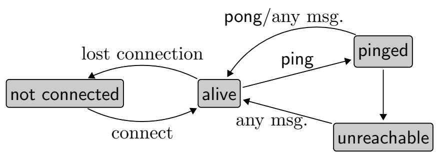
Our implementation of the protocol works across different architectures, so that master and workers could be run on completely different platforms w.r.t. endianness, version of OCaml and operating system.

3.3 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.

4 Experiments

In this section, we demonstrate the potential of using Functory on several case studies. The source code for all these case studies is contained in the distribution, in sub-directory `tests/`.

The purpose of the following experiments is to compare the various deployments, namely sequential, cores and network. For this comparison to be fair, all computations are performed on the same machine, an 8 core Intel Xeon 3.2 GHz running Debian Linux. The sequential implementation uses a single core. The multi-core implementation uses up to 8 cores of the machine. The network implementation uses 8 workers running locally and a master running on a remote machine over a LAN (which incurs communication cost).

4.1 N-queens

The first 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; greater values for D would result in too many tasks.

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.

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. 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. Speedup ratios are also displayed in Fig. 1 (note the logarithmic scale).

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 and Fig. 1, 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

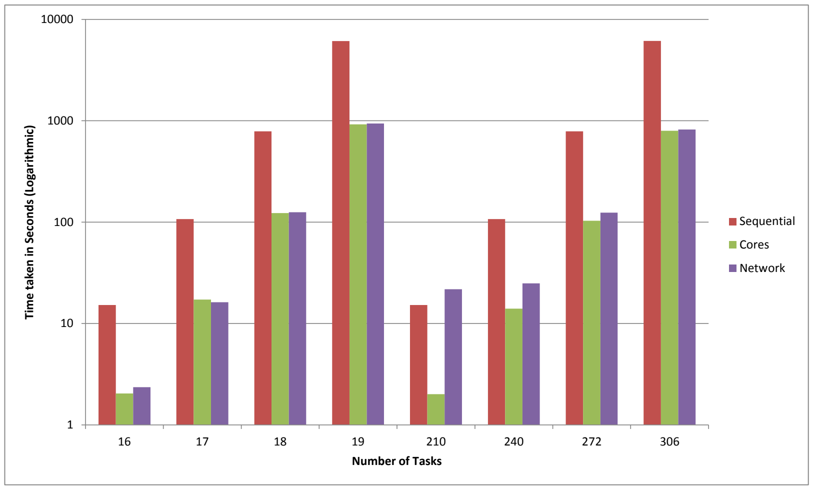


Fig. 1. Speedup ratios for the N-queens experiment.

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.

4.2 Matrix Multiplication

This benchmark was inspired by the PASCO'10 programming contest [5]. It consists of multiplication of two square matrices of dimension 100 with integer coefficients. Coefficients have several thousands of digits, hence we use GMP [4] 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. 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 is $10,000$ for mm1 and only 100 for mm2. On the contrary, each task result for mm1 is a single integer, while for mm2 it is a row of 100 integers. 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×)

The difference in the number of tasks explains the differences in the speedup ratios above. 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 10^6 integers, 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 amount of transmitted data to $10,000$ integers only.

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. 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. If the total number of tasks $\mathsf{t} \geq 1$ is given as a parameter, it is straight forward to split the image into t sub-images, each of which is computed in parallel with and independently of the others. In our case, the image is split into horizontal slices. 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 $54,000,000/\mathsf{t}$. For instance, using $\mathsf{t} = 20$ tasks will result in 20 sub-images of size 10.3 Mb each, assuming each pixel is encoded in four bytes.

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 ($\underline{1.87}$×)	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 ($\underline{3.56}$×)	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 ($\underline{6.93}$×)	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. On 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.

4.4 SMT Solvers

Here we demonstrate the potential of our library for our application needs as mentioned in the introduction. 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).

The master program proceeds by reading the file names, turning them into tasks by multiplying them by the number of provers, resulting in 320 tasks in total. Each worker in turn invokes the given prover on the given file, within a timeout limit of 1 minute. 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.

Our computing infrastructure for this experiment consists of 3 machines with 4, 8 and 8 cores respectively, the master being run on a fourth machine. 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. Further a large parallelizable computation could be distributed by just adding 3-4 lines of code (to just specify the module to be used and the tasks) which is an important user-friendly feature of the library. Further we assume files available over NFS. Intelligent distribution of data over a network is in itself an area of research which is beyond the scope of our work.

5 Conclusions and Future Work

In this paper, we presented a distributed programming library for OCaml. 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. Yet another interesting feature of the library is the ability to add workers dynamically. Functory also allows to cascade several distributed computations inside the same program. Finally, the low-level API of Functory can be used to write interactive programs where one can adjust certain parameters in a GUI, like increasing or decreasing the number of workers, to observe the progress in computation, resource consumption, etc.

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 to the end of the queue.
- Our library provides limited support for retrieving real-time information about computations and communications. Processing and storing information about workers and tasks locally in the master is straightforward.
- 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.

Acknowledgments. We are grateful to the ProVal team for support and comments on early versions of the library and of this paper. We thank the anonymous reviewers for their helpful comments and suggestions.

References

1. CamlP3l. `http://camlp3l.inria.fr/`.

2. Plasma. `http://plasma.camlcity.org/plasma`.
3. The Erlang Programming Language. `http://www.erlang.org/`.
4. The GNU Multiple Precision Arithmetic Library. `http://gmplib.org/`.
5. Parallel Symbolic Computation 2010 (PASCO), 2010. `http://pasco2010.imag.fr/`.
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. Jan Martin Jansen, Rinus Plasmeijer, Pieter Koopman, and Peter Achten. Embedding a Web-based Workflow Management System in a Functional Language. In *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, LDTA '10, pages 7:1–7:8, New York, NY, USA, 2010. ACM.
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. Tom Murphy, VII., Karl Crary, and Robert Harper. Type-safe Distributed Programming with ML5. In *Proceedings of the 3rd conference on Trustworthy global computing*, TGC'07, pages 108–123, Berlin, Heidelberg, 2008. Springer-Verlag.
12. Yoann Padioleau. A Poor Man's MapReduce for OCaml, 2009. `http://www.padator.org/ocaml/mapreduce.pdf`.
13. Robert F. Pointon, Philip W. Trinder, and Hans-Wolfgang Loidl. The Design and Implementation of Glasgow Distributed Haskell. In *Selected Papers from the 12th International Workshop on Implementation of Functional Languages*, IFL '00, pages 53–70, London, UK, 2001.