A poor's man MapReduce for OCaml

Yoann Padioleau
`yoann.padioleau@gmail.com`

June 10, 2009

Contents

1 Introduction **1**

2 Requirements **2**

3 Example of use **3**
- 3.1 A simple map and reduce . . . 3
- 3.2 Using the `distribution.mli` API . . . 4
- 3.3 Compiling . . . 5
- 3.4 Running on a single machine . . . 5
- 3.5 Configuring multiple machines, or what is the MPI model . . . 6
  - 3.5.1 Multiple processors . . . 7
  - 3.5.2 Multiple machines . . . 7
  - 3.5.3 Running . . . 8
- 3.6 Benchmark . . . 9

4 Interface **10**
- 4.1 Main interface, `Distribution.map_reduce` . . . 10
- 4.2 Auxiliary interfaces . . . 11

5 Implementation **12**
- 5.1 The OCamlMPI API . . . 13
- 5.2 Main entry point . . . 14
- 5.3 The master/workers protocol . . . 15
- 5.4 The master . . . 16
- 5.5 The workers . . . 19
- 5.6 Extra code . . . 19

6 Limitations **20**

7 Conclusion **21**

References **22**

1 Introduction

The goal of the `distribution.ml` module documented here is to provide a simple API *a la* MapReduce [1] for OCaml programs.

MapReduce takes his name and core ideas from the functional programming community where the higher order functions `map` and `reduce` (called `fold` in OCaml) are commonly used. Those 2 functions serve as the basis for a very simple parallel programming model where many computationally intensive functions can be programmed by combining a `map` function and its argument (usually in OCaml an anonymous function) with a `reduce` function and its argument, and by applying this combination in a clever way on a huge dataset. In the case of Google's MapReduce, the arguments of the `map` and `reduce` are written in C++. The trick is that internally the 2 functions `map` and `reduce` themselves can be implemented in such a way to distribute *automatically* under the hood the computation to different machines. The programmer does not have to handle all the tedious work that usually comes with writing distributed applications.

Even if OCaml gives access to a good library for distributing computation, MPI [2], via the OCamlMPI OCaml binding [3], and even if MPI is already very good at hiding some of the complexity that usually comes with writing distributed applications[1], MPI (and OCamlMPI) as it is still requires a little plumbing to make it work like the MapReduce programming model. The modest goal of `distribution.ml` is thus to provide this little boilerplate code via some higher-order polymorphic functions that further hide some complexity. Moreover, even if you don't use `distribution.ml`, this document can still serve as an MPI tutorial for OCaml programmers.

In the following section we explain how to install the dependencies required to compile `distribution.ml`. We then present in Section 3 a simple example using the API provided by `distribution.ml`. Section 4 gives the complete reference of the API. Then, for those interested, Section 5 gives more details about the implementation. Section 6 discusses limitations and alternatives. Finally Section 7 concludes and describes how to get `distribution.ml`.

2 Requirements

To compile `distribution.ml` and the examples in this document you first need to install MPI on your machine as well as the OCamlMPI [3] binding from Xavier Leroy's homepage and the utility library Commons [7] used internally by `distribution.ml`. Here is a short script to automate all of this:

2a ⟨*get_dependencies.sh* 2a⟩≡

```
#!/bin/sh
```

[1] MPI (and OCamlMPI) can also be used to leverage the multiple processors on a single machine, which in the case of OCaml programs is useful in its own right as OCaml threads can not leverage such processors.

```
⟨get and install MPI 2b⟩

wget http://caml.inria.fr/distrib/bazar-ocaml/ocamlmpi.tar.gz
wget http://aryx.kicks-ass.org/~pad/ocaml/commons-latest.tgz

tar xvfz ocamlmpi.tar.gz
mv ocamlmpi-1.01 ocamlmpi
cd ocamlmpi/
⟨adjust MPI path in OCamlMPI Makefile 3a⟩
make depend
make

cd ..
tar xvfz commons-latest.tgz
mv commons-latest commons
cd commons/
make depend
make
cd ..
```

MPI is a standard and there are multiple implementations of this standard such as MPICH [4] or OpenMPI [5] (formerly known as LAM/MPI). In the following we focus on MPICH, a popular and free implementation of MPI for network of Unix workstations, but any implementation should work. You can install MPICH from its source but it is usually easier to install it from your Linux distribution, for instance on Debian or Ubuntu with:

2b ⟨*get and install MPI* 2b⟩≡ (2a)

```
sudo apt-get install libmpich1.0-dev # for mpi.h
sudo apt-get install mpich-bin       # for mpirun
```

Note that to *execute* programs using `distribution.ml` you will also need to install MPI on all the machines that will participate in the computation and have the `mpirun` program accessible in your PATH. You will also need `rsh` (or `ssh`) correctly configured in order for the `mpirun` MPI utility to be able to execute the necessary remote commands on all those machines. Section 3.5.2 gives more information about those issues.

Adjusting the Makefile of OCamlMPI is quite simple:

3a ⟨*adjust MPI path in OCamlMPI Makefile* 3a⟩≡ (2a)

```
sed -i -e 's+MPIINCDIR=.*+MPIINCDIR=/usr/include/mpi+'   Makefile
sed -i -e 's+MPILIBDIR=.*+MPILIBDIR=/usr/lib/mpich/lib+' Makefile
```

3 Example of use

The following subsections show how to use `distribution.ml` to distribute a toplevel computation executed usually just after the `main` entry of your pro-

gram[2].

3.1 A simple map and reduce

The following excerpt describes a naive (well, dumb is more accurate) mapping function `map_ex` and reduce function `reduce_ex`. The goal is to compute the sum of a list where each element in this list is the result of applying the Fibonacci function to the original element of another list. In practice Fibonacci would have been replaced by a really heavy and useful computation which would have forced the programmer to distribute it.

3b ⟨*distribution_test.ml* 3b⟩≡ 4a ▷

```
open Common

let rec fib n =
  if n = 0 then 0
  else
   if n = 1 then 1
    else fib (n-1) + fib (n-2)


let map_ex arg =
  pr (spf "map: %d" arg);
  fib arg

let reduce_ex acc e =
  pr (spf "reduce: acc=%d, e=%d" acc e);
  acc + e
```

The code excerpt uses the module `Common` [7], briefly mentioned in Section 2, which provides convenient utility functions or short aliases of utility functions not provided by default in the OCaml standard library. `pr` is for printing on stdout, `spf` is an alias for `Printf.sprintf`, etc.

Just for explanation purposes, here is how a naive `map_reduce` could be written:

4a ⟨*distribution_test.ml* 3b⟩+≡ ◁ 3b 4b ▷

```
let naive_map_reduce ~fmap ~freduce acc xs =
  List.fold_left freduce acc (List.map fmap xs)
```

and here is some code using this `naive_map_reduce`, which of course does not leverage MPI and so does not distribute any computation:

4b ⟨*distribution_test.ml* 3b⟩+≡ ◁ 4a 4c ▷

```
let test_no_mpi () =
```

[2] To distribute deeply nested code of your application requires more code. The main reason for the added complexity is the SPMD (Single Program Multiple Data) model assumed by MPI.

```
    let res = naive_map_reduce ~fmap:map_ex ~freduce:reduce_ex
      0 [1;2;3;10] in
    pr (spf "result = %d" res);
    ()
```

One can also simplify the code by inlining some definitions and by leveraging OCaml anonymous functions with:

4c ⟨*distribution_test.ml* 3b⟩+≡ ◁4b 4d▷

```
  let test_no_mpi_bis () =
    let res = naive_map_reduce ~fmap:fib ~freduce:(fun acc e -> acc + e)
      0 [1;2;3;10] in
    pr (spf "result = %d" res);
    ()
```

3.2 Using the `distribution.mli` API

Now we are ready to use the `Distribution` module to distribute the computation. Here is the code:

4d ⟨*distribution_test.ml* 3b⟩+≡ ◁4c ??▷

```
  let test_mpi () =
    let res = Distribution.map_reduce ~fmap:map_ex ~freduce:reduce_ex
      0 [35;35;35;35] in
    pr (spf "result = %d" res);
    ()

  let main =
    ⟨set debug mpi flag if necessary 8⟩
    test_mpi ()
```

Et voila! You got a distributed application by just changing one line, i.e. by replacing `naive_map_reduce` by `Distribution.map_reduce`. `map_reduce` is a higher-order function taking the mapping and reduce functions as arguments, the initial accumulator (as for a classic fold), and a list, and conceptually has the following specification:

$$\texttt{map_reduce}\ \mathit{fmap}\ \mathit{freduce}\ \mathit{acc}\ \mathit{xs} \equiv \texttt{List.fold}\ \mathit{freduce}\ \mathit{acc}\ (\texttt{List.map}\ \mathit{fmap}\ \mathit{xs}) \quad (1)$$

but internally distributes computations instead of using the `List.fold` and `List.map` functions from the OCaml standard library.

3.3 Compiling

Here is a simple `Makefile` to test our example[3]. We assume the `commons/` and `ocamlmpi/` directories mentioned in Section 2 are present in the current directory.

5 ⟨*Makefile.test* 5⟩≡ 7c ▷

```
INCLUDES=-I commons -I ocamlmpi
OCAMLC=ocamlc
COMMONSYSLIBS=str.cma unix.cma bigarray.cma

distribution_test: distribution.cmo distribution_test.cmo
        $(OCAMLC) $(INCLUDES) -custom $(COMMONSYSLIBS) commons/commons.cma \
            ocamlmpi/mpi.cma $^ -o $@

# the test program
distribution_test.cmo: distribution_test.ml
        $(OCAMLC) $(INCLUDES) -c distribution_test.ml

# the library
distribution.cmo: distribution.ml distribution.mli
        $(OCAMLC) $(INCLUDES) -c distribution.mli
        $(OCAMLC) $(INCLUDES) -c distribution.ml
```

To compile our test example then simply do:

```
$ make -f Makefile.test
```

3.4 Running on a single machine

For debugging purpose, one can run programs using `distribution.ml` on a single machine in which case `Distribution.map_reduce` behaves internally as the `naive_map_reduce` mentioned in Section 3.1. To test on a single machine simply do:

```
$ ./distribution_test
```

The output should be:

```
map: 35
map: 35
map: 35
map: 35
reduce: acc=0, e=9227465
reduce: acc=9227465, e=9227465
reduce: acc=18454930, e=9227465
reduce: acc=27682395, e=9227465
result = 36909860
```

[3] In practice you should write rules to factorize things, to automatically compute the dependencies, etc, but the goal here is not to teach you how to use `make`.

3.5 Configuring multiple machines, or what is the MPI model

Up until now, we have not leveraged multiple machines nor the multiple processors on a single machine to distribute the computation. Before doing so, some explanations about MPI, its programming model, tools, configuration files and runtime support are required.

To use different machines or different processors, one must first describe somewhere those machines and processors so that the MPI runtime can find them and instantiate some processes on them. There are different ways to configure a MPI *cluster* depending on the MPI implementation you use. You can sometimes define a *machine file*, containing a simple list of machines, you can also configure some MPI *daemons*, you can give information directly on the command line, etc. In this document we will focus on the *ch_p4* method provided by MPICH, arguably the simplest one, and its *P4 ProcGroup* configuration file[6]. This file usually ends with a `.pg` suffix (for process group). The syntax of this configuration file is a list of lines having the following format:

```
<hostname> <#procs> <progname>
```

where `<hostname>` can also be an IP address, `<#procs>` represents the number of processes to launch on the specified host, and `<progname>` the path *on the host* of the program to be launched which will participate in the computation. One can also also put `1` in `<#procs>` and duplicate the line multiple times with the same hostname as in:

```
aryx.cs.uiuc.edu 1 /path/to/prog
aryx.cs.uiuc.edu 1 /path/to/prog
```

which will be the method we use in the following [4]. Lines starting with a `#` are comments. This file can thus be used to define a cluster of machines. Each line can be seen as defining a *node* of this cluster, be it another processor on the same machine, or a different machine.

3.5.1 Multiple processors

Here is the MPI P4 ProcGroup configuration file to leverage a multi-processors machine:

7a ⟨*config.pg* 7a⟩≡ 7b ▷

```
aryx.cs.uiuc.edu 0 /home/pad/c__syncweb/demos/mapreduce/distribution_test

#aryx: 2 processors, local machine

aryx.cs.uiuc.edu 1 /home/pad/c__syncweb/demos/mapreduce/distribution_test
```

[4] The semantic of this P4 configuration file is not completely clear. For instance if one writes `aryx.cs.uiuc.edu 2 /path/to/prog`, then this seems to imply that the 2 processes will be run in a shared memory mode which requires more configurations. So, to simplify, I always use `1` for `<#procs>`.

```
aryx.cs.uiuc.edu 1 /home/pad/c__syncweb/demos/mapreduce/distribution_test
```

The first line with the number `0` can seem strange, but is for some reasons required by MPICH. You can see the documentation of P4 for more information [6].

3.5.2 Multiple machines

To leverage other machines, possibly with multi processors too, just add as many entries as required, for instance with:

7b ⟨*config.pg* 7a⟩+≡ ◁7a

```
#phantom: a 4 processors machine
phantom.cs.uiuc.edu 1  /tmp/distribution_test
phantom.cs.uiuc.edu 1  /tmp/distribution_test
phantom.cs.uiuc.edu 1  /tmp/distribution_test
phantom.cs.uiuc.edu 1  /tmp/distribution_test

#put other machines here:
#...
```

Note that as the configuration file mentions the path to the MPI program on the corresponding machine (here `/tmp/distribution_test`), this program must be present on those different machines. It is the programmer's responsibility to copy the program to the appropriate places on the different machines; MPI does not handle that (probably to keep with the KISS philosophy and UNIX philosophy to manage just one thing, and not more).

Here is a simple target to add in your Makefile to automate the copy:

7c ⟨*Makefile.test* 5⟩+≡ ◁5

```
install:
  scp distribution_test phantom.cs.uiuc.edu:/tmp
# scp to other machines here
```

Note that if all the machines in your cluster use NFS, then the previous step can be avoided as one can specify the path to the same binary on every machines [5].

Moreover, to instantiate the different programs on the different machines, the `mpirun` tool, where the use is describe below, will need to execute remote commands on those machines. MPI can use `rsh` or `ssh`. One can use the `RSHCOMMAND` environment variable to specify which one to use. In the following we will use `ssh` and so do:

```
$ export RSHCOMMAND=ssh
```

Before testing your program, you must also check that your `ssh` configuration is correctly set. `ssh` must be configured in a certain way so that you do not need

[5] Under the condition that all the machines have the same architecture.

to give a password at each connection by using the automatic authentification scheme with a private key file (usually in /.ssh/id_rsa), as explained in the ssh manual or one of the numerous tutorials on ssh (e.g., [8]).

Here is a simple test of ssh:

```
$ ssh phantom.cs.uiuc.edu
...
$ ssh aryx.cs.uiuc.edu
...
```

If at one point ssh asks you for a password, then your configuration is not correct and the commands explained in the following section will not work.

3.5.3 Running

Once the MPI configuration file is ready, and ssh correctly configured, one can finally use the mpirun MPI tool that will then internally instantiate the different programs mentioned in the configuration file on the different machines and initiate the appropriate connections. Here is an example of such a command for our test program:

```
$ mpirun -p4pg config.pg ./distribution_test
```

Before showing the output of the previous mpirun command, one can first turn on the debugging feature of distribution.ml, to better see what happens, by adding the following line in our test program:

8 ⟨*set debug mpi flag if necessary* 8⟩≡ (4d)

```
Distribution.debug_mpi := true;
```

The output should now be:

```
MS:DEBUG: mpi master, number of clients=6
W1:DEBUG: mpi worker axyr, rank=1
W3:DEBUG: mpi worker phantom.cs.uiuc.edu, rank=3
W4:DEBUG: mpi worker phantom.cs.uiuc.edu, rank=4
W2:map: 35
W2:DEBUG: mpi worker axyr, rank=2
W5:DEBUG: mpi worker phantom.cs.uiuc.edu, rank=5
W6:DEBUG: mpi worker phantom.cs.uiuc.edu, rank=6
W1:map: 35
W4:map: 35
W3:map: 35
W6:DEBUG: worker exiting
W5:DEBUG: worker exiting
MS:reduce: acc=0, e=9227465
W1:DEBUG: worker exiting
MS:reduce: acc=9227465, e=9227465
MS:reduce: acc=18454930, e=9227465
```

```
MS:reduce: acc=27682395, e=9227465
MS:result = 36909860
W2:DEBUG: worker exiting
W4:DEBUG: worker exiting
W3:DEBUG: worker exiting
P4 procgroup file is config.pg.
```

You can compare with the output without MPI at Section 3.4. Note that the order of the lines may be different but both output should contain a line with a `result = xxx` where `xxx` has the same value.

3.6 Benchmark

Finally, we can now compare the speed of our "MPIfied" program with the original one. Here are the results on our test data:

```
---- without MPI, naive_map_reduce ---
TIME: 9.18s
---- with MPI  ---
TIME: 3.50s
```

Note that even if we specified 6 nodes (6 processors among 2 machines), in our configuration file, we don't get a 6x speedup because of communication overheads and mainly because our test data exercises only 4 processors. Indeed, because our input list contains only 4 numbers (cf Section 3.2), 2 nodes are not used at all. The total running time is thus bounded by the speed of the slowest processor in the cluster. Had we use a bigger list, this would have been less of a concern because the implementation of `Distribution.map_reduce` internally uses a *pool of workers* strategy that continuously feeds idle nodes (cf Section 5.4).

4 Interface

4.1 Main interface, `Distribution.map_reduce`

The main interface of the `distribution.ml` consists really in just a single function[6], `map_reduce`, which takes as an argument the mapping function, the reduce function, the accumulator, and a list and returns the result of the reduction applied to the mapped elements.

10 ⟨*distribution.mli* 10⟩≡ 11a ▷

```
  val map_reduce:
    fmap:('a -> 'b) -> freduce:('c -> 'b -> 'c) ->
    'c -> 'a list -> 'c
```

[6] whereas using MPI requires usually at least 6 functions and multiple types

Even if the type of this function looks pretty simple, there are actually a few assumptions on the context of the code using it; it can not be used anywhere in the same way as a `naive_map_reduce`:

- The user must have a working MPI cluster environment set before executing the program. This means machines with MPI and especially the `mpirun` program installed on, a correct MPI configuration file (e.g., a p4 file) with all programs mentioned in this file correctly copied at the right place, and a working `ssh` or `rsh` configuration.
- the `fmap` must be a pure function as any global mentioned in it would not be shared by the other machines. Moreover, the argument of `fmap`, the elements in the list, must make sense also to the other machines. For instance, if the elements are filenames that `fmap` will then open and process, then those filenames must be accessible on the other machines (again if you use NFS this is not an issue, except that the shared disk may become a bottleneck).
- the `reduce` function must be commutative. Indeed the implementation strategy used in `map_reduce` makes the order of the reductions independent of the order of the elements in the original list.
- Internally the `mpirun` program communicates information to the different nodes through the command line, for instance by adding some `-p4rank xxx` arguments (cf Section 5.2 for more details). So, your program must either not play (nor assume certain things) on `Sys.argv`, or use the `mpi_adjust_argv` helper function.
- `distribution.ml`, and MPI for a big part, assumes a *SPMD* (Single Program Multiple Data) model. That means the same code will be executed on all nodes. While the code executed inside `map_reduce` will eventually diverge for the different nodes, as it internally uses MPI primitives, for instance to either send or receive data, the code before the call to `map_reduce` will not, so such code will be executed on all machines; this may be an issue. This is not an issue of course if the call to `map_reduce` is placed directly after the `main` entry of your program.

4.2 Auxiliary interfaces

A variation of `map_reduce` is `map_reduce_lazy` which takes the list encapsulated in a closure:

11a ⟨*distribution.mli* 10⟩+≡ ◁10 11b▷

```
val map_reduce_lazy:
  fmap:('a -> 'b) -> freduce:('c -> 'b -> 'c) ->
  'c -> (unit -> 'a list) -> 'c
```

The need, sometimes, to encapsulate the list in a closure comes from the SPMD model of `distribution.ml` mentioned before. Indeed, computing the list could itself takes some time, for instance because the list is the result of computing a list of names of files on your disk, and with an SPMD model, it means all nodes would execute the same program before getting the chance to diverge in `map_reduce`. `map_reduce_lazy` solves this problem by delaying the evaluation of the list that in the end will be done internally in `distribution.ml` only by one node (later called the master, cf Section 5.2).

Finally, The `debug_mpi` global, mentioned before, can be used to turn on debugging information related to MPI:

11b ⟨*distribution.mli* 10⟩+≡ ◁11a 11c▷

```
val debug_mpi: bool ref
```

They are a few remaining functions in `distribution.mli` but they are here mostly for documentation purpose, to list and document the types of the important internal functions.

11c ⟨*distribution.mli* 10⟩+≡ ◁11b

```
(*****************************************************************************)
(* Private *)
(*****************************************************************************)
⟨distribution.mli private 14b⟩
```

5 Implementation

The overall structure of `distribution.ml` is as follows:

12 ⟨*distribution.ml* 12⟩≡

```
⟨copyright header 21⟩
open Common

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)

(* cf the distribution.ml.nw literate document for the documentation *)

(*****************************************************************************)
(* Globals *)
(*****************************************************************************)
⟨debug global 19d⟩

(*****************************************************************************)
(* Protocol *)
(*****************************************************************************)
⟨protocol for master/workers 14d⟩

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
⟨worker 19a⟩

⟨master 16c⟩

⟨under_mpirun 15c⟩

(*****************************************************************************)
(* Main entry point *)
(*****************************************************************************)
⟨map_reduce 14a⟩

⟨map_reduce_lazy 15b⟩

(*****************************************************************************)
(* Extra *)
(*****************************************************************************)
⟨protocol for argv ??⟩
⟨mpi_adjust_argv ??⟩
```

In the following we will explain those functions in a top-down approach and so we will start by explaining the functions nearly at the bottom and work out up in this list. But first, some explanations about OCamlMPI are needed.

5.1 The OCamlMPI API

As `distribution.ml` is essentially a thin layer above OCamlMPI, it uses internally OCamlMPI functions and global variables. The important elements of this API are:

- The global `Mpi.comm_world` with type `communicator` that maintains global bookkeeping MPI information about the node. It is initialized by `mpirun` and is represented as an opaque type in OCaml.
- The function `Mpi.comm_size` with type `communicator -> int`, returning the number of nodes participating in the computation.
- The function `Mpi.comm_rank` with type `communicator -> rank` to know the *rank* of the current node, where `rank` is an integer type alias which can take a value in the following range: `[0..comm_size-1]`
- The function `Mpi.send` with type `'a -> rank -> tag -> communicator -> unit`, where `tag` is an integer type alias, and is not important, and where `rank` corresponds to the rank of the current node. Note that this function is polymorphic and internally uses the `Marshall` OCaml module to serialize data.
- The function `Mpi.receive` with type `rank -> tag -> communicator -> 'a`, the counterpart of `send`.
- The constant `Mpi.any_source` with `rank` type which is used when one wants to receive something from any node.
- The function `Mpi.receive_status` with type `rank -> tag -> communicator -> 'a * rank * tag`, which when combined with the previous constant `any_source` can be used as a kind of demultiplexer *a la* `Unix.select`.
- function `Mpi.barrier` with type `communicator -> unit` to implement barriers, but is actually not needed by `distribution.ml`.

Those functions corresponds to the original MPI C functions but an important difference is that OCamlMPI provides polymorphic versions of `Mpi.send` and `Mpi.receive` thanks to the marshalling capabilities of the OCaml runtime. For more information about OCamlMPI, have a look at `mpi.mli` which is well documented. This file is only about 400 lines of code, and describes among other things the 6 functions used internally in `distribution.ml`. Six functions and 400 LOC are quite small numbers, but the goal of `distribution.ml` is to go one step further and to reduce the API to only one function: `Distribution.map_reduce`.

5.2 Main entry point

The skeleton code for `map_reduce` is as follows:

14a ⟨*map_reduce* 14a⟩≡ (12)
```
let map_reduce ~fmap:map_ex ~freduce:reduce_ex acc xs =
  if under_mpirun ()
  then begin
    ⟨map_reduce mpi case 14c⟩
  end
  else
    List.fold_left reduce_ex acc (List.map map_ex xs)
```

which allows to use `map_reduce` even without MPI, in which case it behaves like `naive_map_reduce`. It also uses the following helper:

14b ⟨*distribution.mli private* 14b⟩≡ (11c) 14e ▷
```
val under_mpirun : unit -> bool
```

which can detect if the program was run through `mpirun` (by using the global `Sys.argv`, hence the `unit` argument).

We can now go back to the real part of `map_reduce` which implements a strategy with a *master* and a set of *workers* where the master will continuously feed with jobs the workers:

14c ⟨*map_reduce mpi case* 14c⟩≡ (14a)
```
let rank = Mpi.comm_rank Mpi.comm_world in
if rank = rank_master
then
  master ~freduce:reduce_ex acc xs
else begin
  worker ~fmap:map_ex;
  raise TaskFinished (* for the type system *)
end
```

The important parts of this code and sub functions are:

(1): The rank of the node is used to know who plays what, with as a convention `0` for the master:

14d ⟨*protocol for master/workers* 14d⟩≡ (12) 15a ▷
```
let rank_master = 0
```

(2): The master is responsible for the reduction, which means that right now we distribute only the mapping computation. We could also distribute the reduction computation, but it would make the protocol and the code slightly more complex, so it is simpler for now to just distribute the `map`.

14e ⟨*distribution.mli private* 14b⟩+≡ (11c) ◁ 14b 14f ▷
```
val master : freduce:('c -> 'b -> 'c) -> 'c -> 'b list -> 'c
```

(3): The workers are responsible for the mapping:

14f ⟨*distribution.mli private* 14b⟩+≡ (11c) ◁ 14e ?? ▷
```
val worker : fmap:('a -> 'b) -> unit
```

(4): For typing reason, an exception must be raised after the worker call as they both have different return types. It also means that the workers will never return anything to their callers, only the master will, so only the master will actually execute the code after the call to `map_reduce` in the main program.

15a ⟨*protocol for master/workers* 14d⟩+≡ (12) ◁14d 16a▷
```
exception TaskFinished
```

The code of `map_reduce_lazy` is very similar to `map_reduce`:

15b ⟨*map_reduce_lazy* 15b⟩≡ (12)
```
(* same but with xs lazy, so workers don't need to compute it *)
let map_reduce_lazy ~fmap:map_ex ~freduce:reduce_ex acc fxs =
  if under_mpirun ()
  then begin
    let rank = Mpi.comm_rank Mpi.comm_world in
    if rank = rank_master
    then
      master ~freduce:reduce_ex acc (fxs()) (* changed code *)
    else
      begin
        worker ~fmap:map_ex; (* normally raise already a UnixExit *)
        raise TaskFinished
      end
  end
  else
    let xs = fxs() in (* changed code *)
    List.fold_left reduce_ex acc (List.map map_ex xs)
```

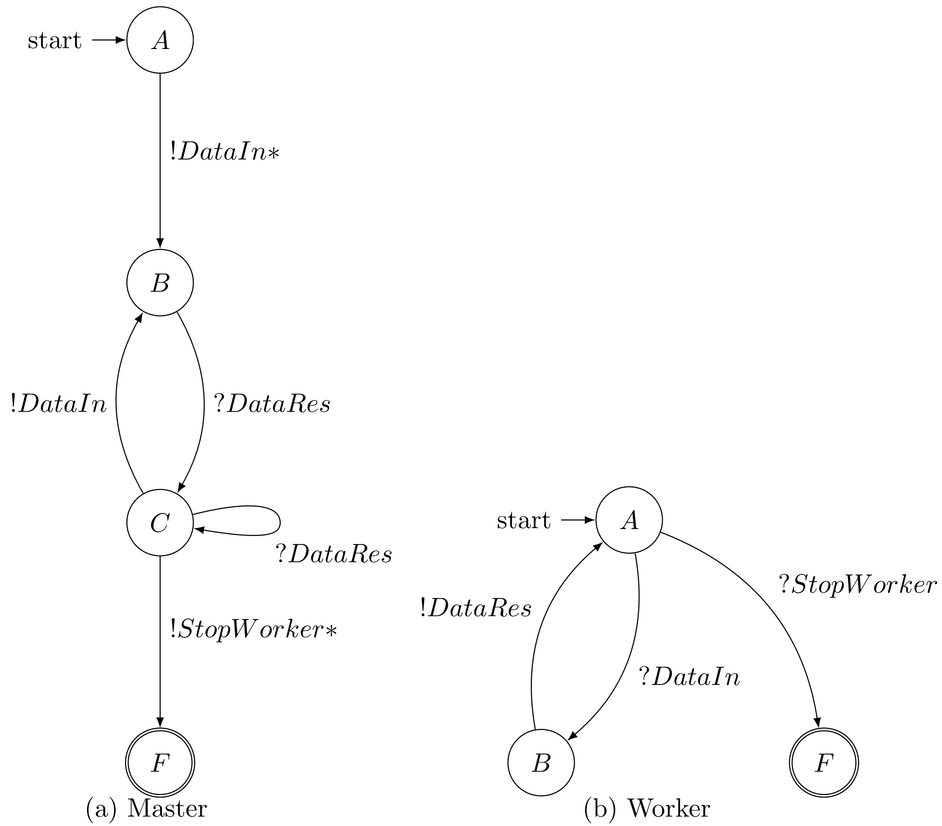
As explained in the previous section, MPI relies a lot on the command line arguments to pass information such as the rank of the current executing program and node. The following function is used to check that the program is run under MPI, that is that the program was launched via `mpirun`[7].

15c ⟨*under_mpirun* 15c⟩≡ (12)
```
let under_mpirun () =
  Sys.argv +> Array.to_list +> List.exists (fun x ->
    x ="-p4pg"  || x = "-p4rmrank"
  )
```

5.3 The master/workers protocol

Before showing the code of `master` and `worker`, we describe the format for the messages that will be exchanged between the master and workers, that is the

[7] TODO: It is currently quite specific to the P4 MPICH method, so one may have to extend it.



(a) Master (b) Worker

Figure 1: Protocol

protocol:

16a ⟨*protocol for master/workers* 14d⟩+≡ (12) ◁15a 16b▷

```
type ('a, 'b) protocol = DataIn of 'a | DataRes of 'b | StopWorker
```

Figure 1 presents an overview of the behavior of the master and workers regarding the exchange of messages.

We also define an exception for errors related to the protocol:

16b ⟨*protocol for master/workers* 14d⟩+≡ (12) ◁16a 17b▷

```
exception ProtocolError
```

5.4 The master

Here is the outline of the code for `master`:

16c ⟨*master* 16c⟩≡ (12)

```
let master ~freduce:reduce_ex acc xs =
  let available_workers = Mpi.comm_size Mpi.comm_world - 1 in
```

```
let actual_workers = min (List.length xs) available_workers in

⟨debug master 20a⟩
⟨killing_workers helper 18e⟩
in


let in_list  = ref xs in
let out_list = ref [] in
let working = ref 0 in
Common.unwind_protect (fun () ->

  assert(List.length !in_list >= actual_workers);

  ⟨send initial work to valid workers 17a⟩
  ⟨kill idle workers 18d⟩
  ⟨enter server loop, in_list shrinks and out_list grows 18a⟩
  ⟨no more remaining, kill workers 18b⟩

  (* big work *)
  List.fold_left reduce_ex acc !out_list

) (fun e ->
  ⟨kill workers because problem 18c⟩
)
```

The first 2 lines are used to cover the case where the input list is smaller than the number of nodes participating in the commutation, in which case many nodes will be idle. The organization of the nodes are then as follow:

0	1..actual_workers	actual_workers+1..comm_size-1
master	workers	idle workers

The first part of the protocol consists in feeding for the first time the workers with `DataIn` messages; it corresponds to the transition from `A` to `B` in Figure 1:

17a ⟨*send initial work to valid workers* 17a⟩≡ (16c)

```
for i = 1 to actual_workers do
  let arg = Common.pop2 in_list in
  Mpi.send (DataIn arg) i notag Mpi.comm_world;
  incr working;
done;
```

Note the use of `i` to iterate over the ranks of the workers. It starts from 1 because 0 is used by the master.

The previous excerpt mentions the `notag` variable which is one of the argument needed by `Mpi.send` but is not relevant in our context:

17b ⟨*protocol for master/workers* 14d⟩+≡ (12) ◁16b

```
let notag = 0
```

The main part of the the master consists in a loop where the master acts as a kind of *server* waiting for the results from the heavy computation (the `fmap`) done by his workers, and giving them back more work if some elements are remaining in `in_list`:

18a ⟨*enter server loop,* `in_list` *shrinks and* `out_list` *grows* 18a⟩≡ (16c)

```
while !working > 0 do
  let (res, src, _) = Mpi.receive_status Mpi.any_source notag Mpi.comm_world in
  (match res with
  | DataRes x ->
      Common.push2 x out_list;
  | DataIn _ | StopWorker -> raise ProtocolError
  );

  if not (null !in_list) then begin
    let arg = Common.pop2 in_list in
    Mpi.send (DataIn arg) src notag Mpi.comm_world;
  end
  else decr working;
done;
```

Note that because `receive_status` is non deterministic (some nodes may be faster than other nodes, some congestion on certain paths in the network, etc), the order of the `push2` in `out_list` are also non deterministic and so the order of the elements in this list may not correspond to the order of the mapped elements in the original list. This is why `reduce` must be commutative. Note also the use of the same `src` variable in receiving and sending commands, which is the rank of the node currently involved in the communication.

Here is the code related to stopping the appropriate workers (the actual workers, or the idle ones) which use the `StopWorker` message:

18b ⟨*no more remaining, kill workers* 18b⟩≡ (16c)

```
killing_workers (Common.enum 1 actual_workers);
flush stderr;flush stdout;
```

18c ⟨*kill workers because problem* 18c⟩≡ (16c)

```
pr2 (spf "PB: mpi master dying: %s" (Common.exn_to_s e));
killing_workers (Common.enum 1 available_workers);
```

18d ⟨*kill idle workers* 18d⟩≡ (16c)

```
killing_workers (Common.enum_safe (actual_workers+1) available_workers);
```

18e ⟨*killing_workers helper* 18e⟩≡ (16c)

```
let killing_workers xs =
  xs +> List.iter (fun i -> Mpi.send StopWorker i notag Mpi.comm_world)
```

5.5 The workers

The `worker` code is simpler than the one for the master:

19a ⟨*worker* 19a⟩≡ (12)

```
let worker ~fmap:map_ex =
  let rank = Mpi.comm_rank Mpi.comm_world in
  let hostname = Unix.gethostname () in

  ⟨debug worker 20b⟩

  Common.unwind_protect (fun () ->
   ⟨enter worker loop 19b⟩
  )
  (fun e ->
    ⟨exit worker 19c⟩
  )
```

The worker also uses a loop, waiting for more work (and then executing an `fmap`) or an end-of-work message (`StopWorker`):

19b ⟨*enter worker loop* 19b⟩≡ (19a)

```
while true do
  let req = Mpi.receive rank_master notag Mpi.comm_world in
  match req with
  | DataIn req ->
      (* big work *)
      let res = map_ex req in
      Mpi.send (DataRes res) rank_master notag Mpi.comm_world
  | StopWorker ->
      ⟨debug worker exit 20c⟩
      raise (UnixExit (0))
  | DataRes _ -> raise ProtocolError
done
```

19c ⟨*exit worker* 19c⟩≡ (19a)

```
match e with
| UnixExit(0) -> exit 0
| _ ->
    pr2 (spf "PB: mpi worker dying: %s" (Common.exn_to_s e));
```

This last excerpt means that the worker never returns to the caller, and so the code after the call to `map_reduce` in the main program will be executed only by the master node.

5.6 Extra code

19d ⟨*debug global* 19d⟩≡ (12)

```
let debug_mpi = ref false
```

In the following, the global `Common._prefix_pr` is modified. This global is internally used by all the printing functions in the `Common` library (e.g., `pr`, or `pr2` which prints on `stderr`) which in turns allows to better trace from which machine an output comes from as shown in Section 3.5.3.

20a ⟨*debug master* 20a⟩≡ (16c)
```
if !debug_mpi
then Common._prefix_pr := ("MS:");
if !debug_mpi
then pr2 (spf "DEBUG: mpi master, number of clients=%d" available_workers);
```

20b ⟨*debug worker* 20b⟩≡ (19a)
```
if !debug_mpi
then Common._prefix_pr := (spf "W%d:" rank);
if !debug_mpi
then pr2 (spf "DEBUG: mpi worker %s, rank=%d" hostname rank);
```

20c ⟨*debug worker exit* 20c⟩≡ (19b)
```
if !debug_mpi
then pr2 ("DEBUG: worker exiting");
flush stderr; flush stdout;
```

20d ⟨*MISC1* 20d⟩≡
```
val mpi_debug_argv : 'a -> unit
```

20e ⟨*MISC2* 20e⟩≡
```
let mpi_debug_argv _argv =
  let rank = Mpi.comm_rank Mpi.comm_world in
  Sys.argv +> Array.to_list +> List.iter (fun s ->
    pr2 (spf "%d: %s" rank s);
  );
  ()
```

6 Limitations

The original MapReduce [1] has not only support for automatically distributing computations but also provides automatic load balancing and fault-tolerance. None of this is currently provided by `distribution.ml`.

The code is very polymorphic but not type-safe as OCamlMPI uses internally the `Marshall` module. So, take care to run every instance of your program with the latest compiled source code. To avoid possible inconsistencies, you should ensure this automatically by coding appropriate targets in your Makefile such as `make install`.

For your information, here are some alternatives to MPI and `distribution.ml` for writing distribute applications in OCaml:

- charm

- netplex
- chameleon
- jocaml
- ocaml3pl
- ensemble

Here is a list of TODOs:

- Handle dead worker ? and redirect his job to another worker ?
- collect some statistics on workers ? maybe can first broadcast an identification request where worker just do a 'hostname' ?
- typecheck ? add always as first element a string containing the type so worker can assert `((fst req) = "string")`
- distribute also the fold

7 Conclusion

This document has presented the `distribution.ml` OCaml module to help distributing OCaml code. "MPIfying" an OCaml program, or "MapReduceing" it takes a little effort, but in addition to expected important performance benefits, it also has the good virtue to force you, the programmer, to use less global variables (as they would not be shared by the program running on the other machines) and to show more clearly the dependencies between functions. Your code may be paradoxically clearer after making it distributed :)

Availability

The code described in this document is available at:

http://aryx.kicks-ass.org/~pad/ocaml/mapreduce-latest.tgz

Copyright

`distribution.ml` is governed by the following copyright:

21 ⟨*copyright header* 21⟩≡ (12)

```
(* Yoann Padioleau
 *
 * Copyright (C) 2009 University of Urbana Champaign
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License (GPL)
```

```
* version 2 as published by the Free Software Foundation.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
* file license.txt for more details.
*)
```

Changelog

- Version 0.1 (June 2009). Initial release

References

[1] Dean Jeffrey and Ghemawat Sanjay, *MapReduce: Simplified Data Processing on Large Clusters*, OSDI 2004.

[2] The Message Passing Interface (MPI) Standard, http://www.mcs.anl.gov/research/projects/mpi/

[3] Xavier Leroy, OCamlMPI: Interface with the MPI Message-Passing Interface, http://pauillac.inria.fr/~xleroy/software.html#ocamlmpi

[4] MPICH, http://www.mcs.anl.gov/research/projects/mpich2/

[5] OpenMPI, http://www.open-mpi.org/

[6] The P4 Procgroup File, http://www.mcs.anl.gov/research/projects/mpi/mpich1/docs/mpichman-chp4/node14.htm#Node21

[7] Yoann Padioleau, Commons_{pad} OCaml Library, http://aryx.kicks-ass.org/~pad/ocaml/Commons.pdf

[8] SSH Automatic login, http://wp.uberdose.com/2006/10/16/ssh-automatic-login/