FUNCTORY
A Distributed Computing Library for Objective Caml

Jean-Christophe Filliâtre & K. Kalyanasundaram

CNRS / INRIA Saclay – Île-de-France

TFP 2011, Madrid, Spain

Motivation

- In our team, we do deductive program verification
- Generates numerous verification conditions
- Discharged by various automated provers
- Typically takes hours to complete

- Some multi-core machines at our disposal
- How to make the best possible use of them?

A Distributed Computing Library

Requirements

- Fault-tolerance
- User-friendly API
- In our favorite programming language (OCaml)

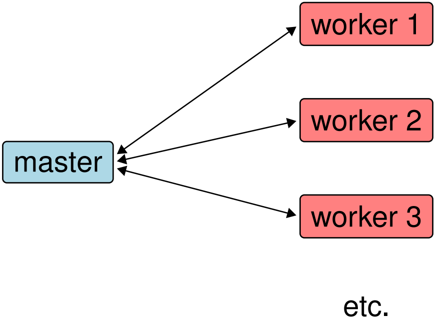
Also

- General purpose library
- Portability

Basic Design

Inspired by Google's Map/Reduce (OSDI 2004)

- Workers in parallel
- Master



Outline

- API
  - general-purpose `compute` function
  - high-level: map/fold operations
  - low-level: micro-step computations
- Deployment Scenarios
  - Sequential
  - Cores
  - Network (3 flavours)
- Many libraries in one

A General-Purpose `compute` Function

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

- A task is of type $\alpha \times \gamma$, its result of type β
- A completed task may in turn generate new tasks
- `compute` returns when there is no more task

High-Level API

- most common map/fold operations over lists

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

- `f` operations always in parallel
- Two flavours: `map_local_fold` and `map_remote_fold`
- More parallelism when fold is associative and commutative

```
val map_fold_ac, map_fold_a:
  f: (α → β) →
  fold: (β → β → β) → β → α list → β
```

Low-Level API

- User can interact with the execution of the distributed computation
- Examples:
  - Monitoring applications: observation of consumption of resources, etc
  - Interactive programs

Low-Level API

- type (α, γ) computation
- creation

```
val create: worker: (α → β) →
              master:(α × γ → β → (α × γ)list) →
              (α × γ) computation
```

- adding new tasks

```
val add_task: (α × γ) computation → α × γ → unit
```

- performing one step of the computation

```
val one_step: (α × γ) computation → unit
```

- etc.

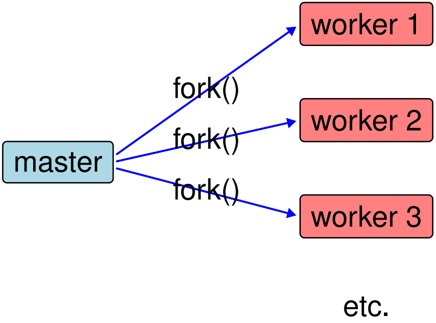
Outline

- API
  - general-purpose `compute` function
  - high-level: map/fold operations
  - low-level: micro-step computations
- Deployment Scenarios
  - Sequential
  - Cores
  - Network

Cores Implementation

Uses `Unix.fork` (no control over scheduling)

```
open Cores
let () = set_number_of_cores 3
```



master maintains a queue of pending tasks

Network Implementation

based on

- TCP/IP client/server architecture
- Ocaml's marshaling capabilities

marshaling considerations

1. same binary: we can marshal closures
2. same version of Ocaml: we can only marshal values
3. otherwise: we can only marshal strings

Three Implementations of Network

Same binary

```
val compute : (* same as before *) ...
```

Same version of Caml

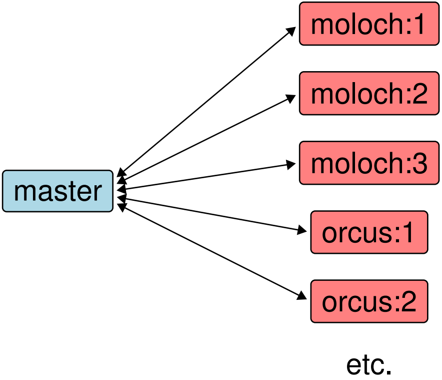
```
val Worker.compute : (α → β) → unit
val Master.compute :
  (α × γ → β → (α × γ) list) →
  (α × γ) list → unit
```

Otherwise

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

Network Implementation Details

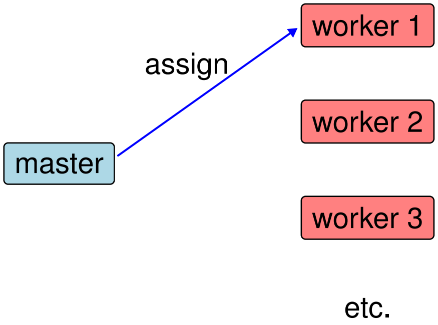
```
open Network
let () = declare_workers ~n:3 "moloch"
let () = declare_workers ~n:2 "orcus"
```



Each worker behaves as a server, the master being the client

Protocol

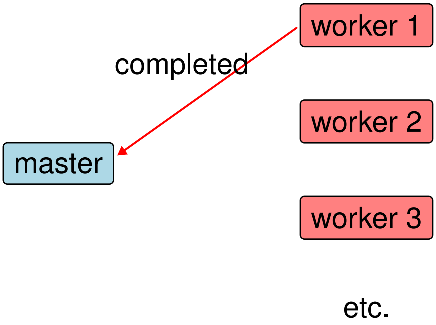
Master sends a task to a worker



Assign(42, f, a)

Protocol

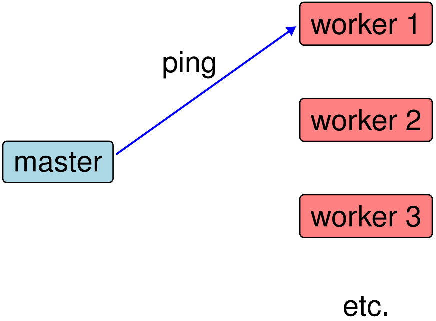
Worker computes and sends back a result



Completed (42, b)

Protocol

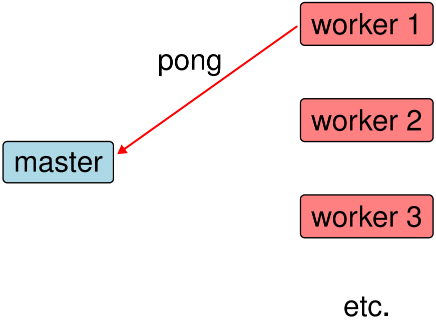
Master and workers exchange ping/pong messages



Ping

Protocol

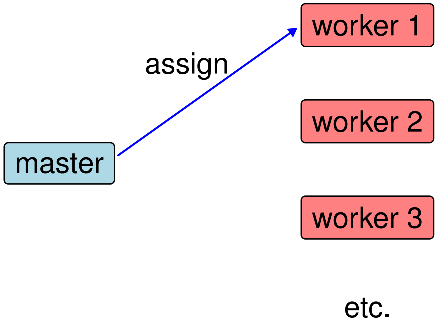
Master and workers exchange ping/pong messages



Pong

Protocol

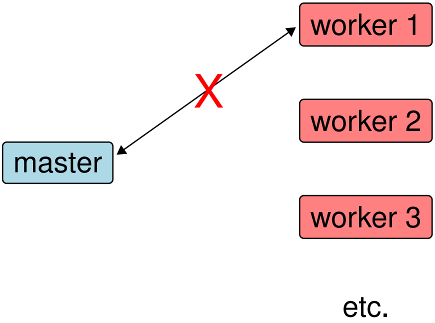
Master sends another task to a worker



Assign(43, f, a)

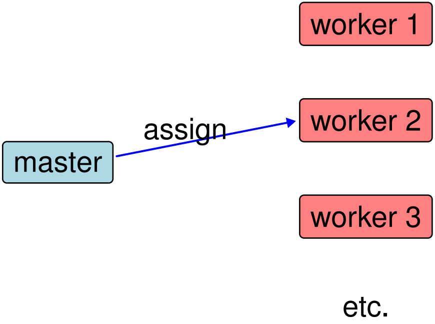
Protocol

In case of a disconnection...



Protocol

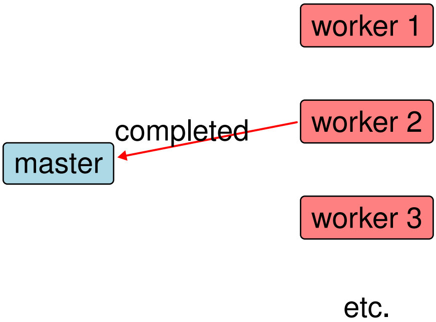
The task is rescheduled to another worker



Assign(43, f, a)

Protocol

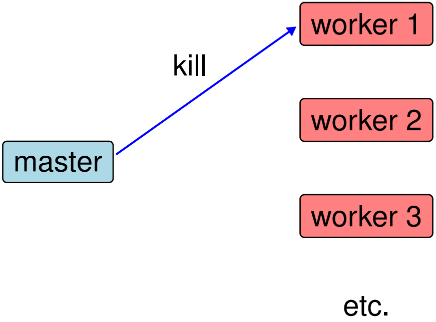
Whenever one completes...



Completed (43, b)

Protocol

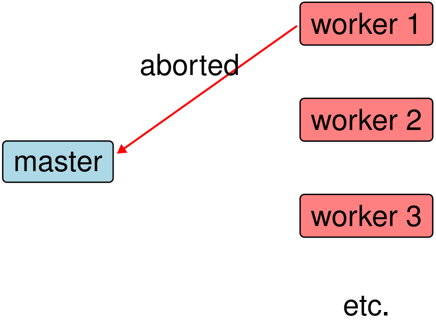
The other one is stopped



Kill 42

Protocol

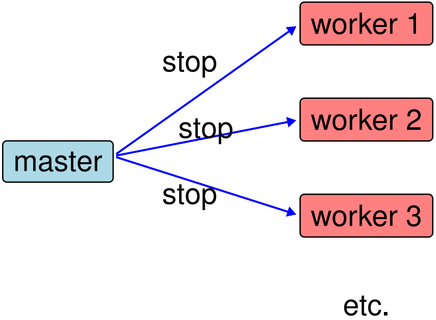
The master is notified when a computation fails



Aborted 42

Protocol

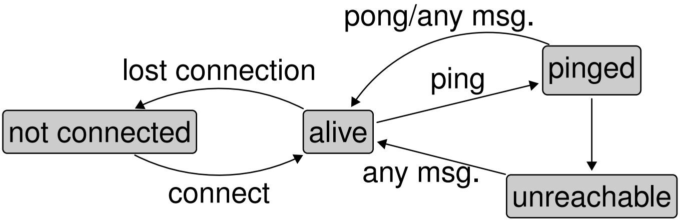
At the very end, the master may ask the workers to stop



Stop

Fault Tolerance

Master knows the state of each worker through ping/pong messages



- Time-out values for deciding the status of the worker

Experimental Results

Motivating example

- 80 verification conditions / 4 provers = 320 tasks
- network of 3 machines (4, 8 and 8 cores)

- sequential computation: $>$ 6 hours
- with Functory: 22.5 minutes
- speedup ratio = 16 (optimal is 20)

More experimental results in the paper:
N-queens, Mandelbrot set, matrix multiplication

Related Work

Distributed Functional Languages (DFL)

- Jo&Caml - rich communication primitives, no primitive features for fault-tolerance
- ML5 - code mobility, type-safe marshalling, etc, but no primitive features for fault-tolerance
- Glasgow Distributed Haskell - features for fault-tolerance, error detection/recovery

Libraries for existing functional languages — like Functory

- Plasma MR - OCaml implementation of Map/Reduce
- iTask - Clean library for distributed workflow management

Future Work

- More user control
  - Scheduling of tasks, etc
- Real-time visualization
  - Resource consumption, task distribution patterns, etc.
- Speeding up using idle workers

Thanks

Check out:

http://functory.lri.fr/

Feedback, comments welcome!