Adapting the OCaml ecosystem for Multicore OCaml

Anonymous Author(s)

Abstract

OCaml 5.0 with support for shared-memory parallelism being around the corner, there's increasing interest in the community to port existing libraries to Multicore. This talk will take the attendees through what the arrival of Multicore means to the OCaml ecosystem, and existing tools and methods for a smooth transition to benefit from Multicore parallelism. We aim to share some insights from our experience of porting existing libraries to Multicore OCaml. We will cover:

- Building your package with the Multicore compiler
- Breaking changes in the runtime
- Global state & thread-safety
- Multiprocess vs multicore
- Example of parallelizing a library: Lwt

1 Introduction

Multicore OCaml[1] is an extension of OCaml with support for shared memory parallelism through Domains [Sivaramakrishnan et al. 2020] and concurrency through Algebraic effects [Sivaramakrishnan et al. 2021]. Efforts are underway for upstreaming Multicore OCaml's domains-only parallelism to mainstream OCaml and is slated for release in its OCaml 5.0 version. OCaml 5.0 will be accompanied by a robust set of libraries such as domainslib to aid writing efficient parallel programs. Algebraic effects is expected to land in later versions of OCaml.

Parallelizing your OCaml code with Multicore OCaml [Jaffer et al. 2020] gives an overview of the parallel programming primitives and libraries that Multicore OCaml offers and performance tuning methods for writing efficient parallel programs in Multicore OCaml. This presentation will focus on adapting the ecosystem to Multicore, potential roadblocks and ideas to resolve them. This will be of particular interest to users who wish to port their libraries to Multicore.

[1] https://github.com/ocaml-multicore/ocaml-multicore

2020. 2475-1421/2020/1-ART1 $15.00
https://doi.org/

2 Building your package with multicore compiler

The multicore compiler can be obtained from multicore-opam[2]. Currently the multicore tree is at compiler version 4.12, with two variants:

- **4.12+domains** : With support for domains-only parallelism. This branch is close to what's intended to be shipped with OCaml 5.0.
- **4.12+domains+effects** : With support for parallelism and algebraic effects.

2.1 OPAM Health Chek

opam-health-check is deployed to keep a tab on any build failures occurring on opam packages. One of the instances of opam-health-check tracks the multicore branch `4.12+domains`. The server builds packages and runs their testsuites once every week. Any build or run failures can be viewed in a web interface. So far, this has been useful in identifying package failures, debugging and fixing bugs in the package or compiler.

2.2 OCaml Multicore CI

A select set of widely used OCaml packages are to be monitored for every commit made to the `4.12+domains` branch, by building and running the testsuites of those packages. This will ensure compatibility of the Multicore compiler with external packages and keep check on any digression in the testsuite results from that of stock OCaml. OCaml Multicore CI can be added as a GitHub app to the library's repository.

3 Changes in the runtime

Multicore OCaml comes with a concurrent Garbage Collector (GC). Good news is, the parallel minor collector is compatible with OCaml C API, so C FFI will be unaffected by the Multicore GC. Care has been taken to ensure that we maintain backwards compatibility to a large extent. Development of the multicore GC led to quite some changes in the runtime primitives. Hence, libraries using runtime primitives that may no longer be compatible with the Multicore runtime will need to be updated.

Some examples of changes in the runtime are:

- A large number of global variables are now moved to a domain-specific environment called `Caml_state`.

[2] https://github.com/ocaml-multicore/multicore-opam

- Garbage collector colors are changed with gray colour eliminated. Multicore compiler uses the categories `MARKED`, `UNMARKED` and `GARBAGE` as opposed to stock OCaml's `BLACK`, `WHITE`, `GRAY` and `BLUE`.
- GC module no longer has attributes that are not necessary for the concurrent GC, and some assumptions made about GC hooks will vary from stock OCaml.
- Naked pointers are strongly discouraged from use and tools are developed to detect them.
- Lazy values can be forced from only one domain at a time. Concurrently forcing the same lazy value would result in an exception.

More such changes may be seen in the runtime in coming releases of OCaml. To ease the process of adjusting with the runtime changes, the compiler ships with `compatibility.h` providing necessary primitives. The presentation will go through the recommendations for tackling such changes.

4 Multiprocess vs Multicore

Multiprocess primitives such as `Unix.fork` provide a way to execute processes on multiple cores and such techniques have been used to speedup OCaml programs. A notable distinction between multiprocess programs written with `Unix.fork` and Multicore programs using domains is; Multicore programs share data amongst different domains while multiprocess programs have a separate copy of data per process. Upon comparison on the same benchmarks, Multicore programs tend to exhibit better performance especially when the number of cores is high, as opposed to multiprocess programs. We will cover details one needs to be aware of while porting multiprocess programs to Multicore OCaml.

5 Global state & Thread-safety

OCaml allows mutable data to be stored in variables, unlike some purely functional languages. This has proved to be an advantage for OCaml programmers and has been useful to implement some algorithms easily. It is also easier to achieve better performance with imperative programs as opposed to their purely functional counterparts. However, global data without synchronizations will not sit well in a multicore environment. An example to illustrate this:

5.1 Mutable counter

```
let counter = ref 0

let () =
    incr counter;
    print_int !counter
    incr counter;
    print_int !counter
```

`counter` is well-behaved as long as it is accessed from only one domain like above.

```
let counter = ref 0

let () =
  let domains = Array.init 4 (fun _ ->
      Domain.spawn(fun () -> incr counter)) in
  Array.iter Domain.join domains;
  print_int !counter
```

The output of this program is non-deterministic, because the order of execution is not constant between different runs of the same program. We will cover various synchronization methods to make mutable code thread-safe, their advantages and downsides. They are mainly:

- Eliminating global data
- Making mutable data atomic
- Using thread-safe data structures available in stdlib and external libraries
- Synchronization primitives in the compiler. (Mutex and Condition variables)

6 Example of parallelizing a library: Lwt

One of the libraries to have been successfully ported to benefit from parallelism is the monadic concurrency library `Lwt`. In our prototype, a new module `Lwt_domain` was added which has the feature to offload CPU intensive tasks to multiple cores. Ideas to integrate this module with Lwt are being worked out with `Lwt` maintainers. This presentation will walk through the process we adopted for parallelising Lwt and how this feature could be incorporated in programs written with `Lwt`.

References

Sadiq Jaffer, Tom Kelly, Sudha Parimala, KC Sivaramakrishnan, and Anil Madhavapeddy. 2020. Parallelising your OCaml code with Multicore OCaml. *OCaml Workshop* (2020).

KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting Parallelism onto OCaml. In *Proceedings of the 25th ACM SIGPLAN International Conference on Functional Programming (ICFP 2020).*

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting Effect Handlers onto OCaml. *arXiv preprint arXiv:2104.00250* (2021).