Parallelising your OCaml Code with Multicore OCaml

Sadiq Jaffer
Opsian
OCaml Labs
sadiq@toao.com

Tom Kelly
OCaml Labs
tom.kelly@cantab.net

Sudha Parimala
IIT Madras
sudharg247@gmail.com

KC Sivaramakrishnan
IIT Madras
OCaml Labs
kcsrk@iitm.ac.in

Anil Madhavapeddy
University of Cambridge Computer Laboratory
OCaml Labs
avsm2@cl.cam.ac.uk

Abstract

With the availability of multicore variants of the recent OCaml versions (4.10 and 4.11) that maintain backwards compatibility with the existing OCaml C-API [Sivaramakrishnan et al. 2020] there has been increasing interest in the wider OCaml community for parallelising existing OCaml code. From our experience with a range of programs, linear speedups on up to 24 cores are possible. This presentation will take the attendees through the following steps aimed at developing parallel programs with Multicore OCaml:

- Installing the latest Multicore OCaml compiler
- Brief overview of the low-level API for parallel programming
- A tour of domainslib – a high-level parallel programming library for Multicore OCaml
- Common pitfalls when parallelising
- Tools for diagnosing Multicore OCaml performance

1 Multicore OCaml

Multicore OCaml[1] is an extension of OCaml with support for concurrency and shared-memory parallelism. Concurrency is expressed through effect handlers [Dolan et al. 2018a] and parallelism through *domains* [Sivaramakrishnan et al. 2020]. The current focus is to upstream domains-only support to OCaml, followed by effect handlers in a subsequent release. Recently, Multicore OCaml has decided to adopt a stop-the-world parallel minor collector as opposed to a concurrent minor collector. The parallel minor collector generally exhibits better performance than the concurrent minor collector, but crucially, does not necessitate changes in the C API. This simplifies the migration story to adapt the ecosystem to Multicore OCaml. Existing single-threaded OCaml code can be built and run as is on the multicore runtime. The multicore OCaml compiler is available as an opam switch for easy installation [2].

2 Domains

Domains are the unit of parallelism in Multicore OCaml. Each domain maps to a system thread, and can be created and destroyed dynamically. Multicore OCaml provides minimal support for domains in the standard library. There is support for creating and joining domains, waiting for notifications and notifying waiting domains. The standard library distributed with the compiler does not provide advanced concurrent programming primitives like mutexes, channels, fork-join parallelism and transactional memory. Instead the compiler provides atomic memory operations in its standard library to support the development of concurrent libraries. Multicore OCaml comes equipped with a memory model that ensures that data race free parts of the program have sequentially consistent semantics, and importantly, ensures type safety in the presence of data races [Dolan et al. 2018b].

Any advanced concurrent programming libraries can thus live outside the compiler as libraries which can evolve independently from the compiler development cycle. Moreover, this removes the burden of maintenance of the libraries from the compiler developers. That said, Multicore OCaml will be accompanied by a blessed set of libraries for advanced concurrent programming which will be distributed through opam.

3 Domainslib

One such library is *Domainslib*[3] which provides high-level parallel programming primitives built on top of domains. Domainslib provides support for first-class, multi-producer, multi-consumer channels, task pools

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

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

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

with async/await parallelism, and parallel for loops. The snippet below shows the code for computing the 50th Fibonacci number using 4 domains with the `Domainslib.Task` module:

```
module T = Domainslib.Task

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

let rec fib_par pool n =
  if n <= 30 then fib n
  else
    let a = T.async pool (fun _ ->
      fib_par pool (n-1)) in
    let b = T.async pool (fun _ ->
      fib_par pool (n-2)) in
    T.await pool a + T.await pool b

let main =
  let pool = T.setup_pool ~num_domains:4 in
  let n = 50 in
  let res = fib_par pool n in
  T.teardown_pool pool;
  Printf.printf "fib(%d) = %d\n" n res
```

The following is parallel matrix multiplication using the `parallel_for` construct. The `chunk_size` determines the granularity of tasks:

```
module T = Domainslib.Task
let n_domains = 4

let mat_mul pool m1 m2 =
  let i_n = Array.length m1 in
  let j_n = Array.length m2.(0) in
  let k_n = Array.length m2 in
  let m3 = Array.make_matrix i_n j_n 0 in

  T.parallel_for pool ~chunk_size:(i_n/n_domains)
  ~start:0 ~finish:(i_n - 1)
  ~body:(fun i -> for j = 0 to pred j_n do
      for k = 0 to pred k_n do
        m3.(i).(j) <-
        (m3.(i).(j) + (m1.(i).(k) * m2.(k).(j)));
      done;
    done);
  m3

let _ =
    let m1 = Array.make_matrix 1024 1024 128 in
    let m2 = Array.make_matrix 1024 1024 256 in

    let pool = T.setup_pool
          ~num_domains:(n_domains - 1) in
    let _ = mat_mul pool m1 m2 in
    T.teardown_pool pool
```

The following shows how to send and receive on a domasinslib channel:

```
module C = Domainslib.Chan

let worker i (queue : string C.t) () =
  let msg = C.recv queue in
  Printf.printf "worker [%d] received [%s]\n"
    i msg

let main =
  let n_workers = 2 in
  let queues =
    Array.init n_workers (fun i ->
        C.make_bounded 8)
  in
  let domains =
    Array.init n_workers (fun i ->
        Domain.spawn (worker i queues.(i)))
  in
  Array.iteri
    (fun i q ->
      C.send q (Printf.sprintf "Hello %d" i))
    queues;
  Array.iter (fun d -> Domain.join d) domains
```

Thus, domainslib makes it easy to parallelise existing sequential code.

4 Common pitfalls when parallelising

In our experience of parallelising existing OCaml programs, we have encountered several different types of problems that limited performance scalability. The majority fall in to two categories.

4.1 Poor granularity of parallel tasks

Whilst domains are the unit of parallelism, their creation and termination can be relatively heavyweight and can soon dominate execution time if the tasks run are small. Domainslib provides a much more lightweight task pooling mechanism but care still needs to be taken to ensure tasks are of sufficient size that their distribution does not exceed the time spent doing useful work.

4.2 Shared state write contention

Parallel writes to shared state leads to poor scalability on modern processors due to cache lines being ping-ponged between cores. The effect can be even more dramatic on processors with significantly non-uniform memory access. This write contention can often be hidden within standard library or runtime code and can be hard to find without specialised tooling to do so.

5 Tools for diagnosing Multicore OCaml performance

We will cover the two main tools used to diagnose the performance of Multicore OCaml programs. These can cover Garbage Collection, CPU time and shared state related issues.

5.1 OCaml Eventlog

Multicore OCaml includes an instrumented runtime by default that can output GC events in a format compatible with Chrome's tracing viewer. An extended version of this instrumentation will be available as an option in OCaml 4.11.0. We will cover how to use this GC event information to understand where the GC is under pressure and possible solutions.

Figure 1. Example eventlog trace from a Multicore OCaml program

5.2 Linux perf

The perf performance subsystem and tools are powerful but complex and difficult to use for newcomers. We will discuss performance events and profiling, and how these can be analysed with perf to identify performance bottlenecks.

References

Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K. C. Sivaramakrishnan, and Leo White. 2018a. Concurrent System Programming with Effect Handlers. In *Trends in Functional Programming*, Meng Wang and Scott Owens (Eds.). Springer International Publishing, Cham, 98–117.

Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. 2018b. Bounding Data Races in Space and Time. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 242–255. https://doi.org/10.1145/3192366.3192421

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)*.