Parallelising your OCaml code with Multicore OCaml

Sadiq Jaffer, Tom Kelly, Sudha Parimala, KC Sivaramakrishnan, Anil Madhavapeddy

Overview

- Multicore OCaml
- Domains
- Domainslib
- Further optimisation

Multicore OCaml

- Concurrency is overlapping computations
- Parallelism is simultaneous computations

Multicore OCaml ⇨ concurrency *and* shared-memory parallelism

Multicore OCaml

- Compatible with existing OCaml code (inc ppx)
- OCaml 5 will have parallelism via Domains
- Concurrency via effects and fibers to follow

Domains

- Unit of parallelism
- Heavyweight
- Functionality
  - Spawn/join
  - Wait/notify
  - Atomic memory operations
  - Local storage

N-Body

- Derived from benchmarks game
- Models orbit of a number of bodies

N-Body serial

```
let advance bodies n_bodies dt =
  for i = 0 to n_bodies - 1 do
    let b = bodies.(i) in
    for j = i+1 to n_bodies - 1 do
      let b_o = bodies.(j) in
      let dx = b.x -. b_o.x  and dy = b.y -. b_o.y
        and dz = b.z -. b_o.z in
      let dist2 = dx *. dx +. dy *. dy +. dz *. dz in
      let mag = dt /. (dist2 *. sqrt(dist2)) in

      b.vx <- b.vx -. dx *. b_o.mass *. mag;
      b.vy <- b.vy -. dy *. b_o.mass *. mag;
      b.vz <- b.vz -. dz *. b_o.mass *. mag;

      b_o.vx <- b_o.vx +. dx *. b.mass *. mag;
      b_o.vy <- b_o.vy +. dy *. b.mass *. mag;
```

N-Body serial

```
let advance bodies n_bodies dt =
  for i = 0 to n_bodies - 1 do
    let b = bodies.(i) in
    for j = i+1 to n_bodies - 1 do
      let b_o = bodies.(j) in
      let dx = b.x -. b_o.x  and dy = b.y -. b_o.y
        and dz = b.z -. b_o.z in
      let dist2 = dx *. dx +. dy *. dy +. dz *. dz in
      let mag = dt /. (dist2 *. sqrt(dist2)) in

      b.vx <- b.vx -. dx *. b_o.mass *. mag;
      b.vy <- b.vy -. dy *. b_o.mass *. mag;
      b.vz <- b.vz -. dz *. b_o.mass *. mag;

      b_o.vx <- b_o.vx +. dx *. b.mass *. mag;
      b_o.vy <- b_o.vy +. dy *. b.mass *. mag;
```

N-Body serial

```
    for j = i+1 to n_bodies - 1 do
      let b_o = bodies.(j) in
      let dx = b.x -. b_o.x  and dy = b.y -. b_o.y
        and dz = b.z -. b_o.z in
      let dist2 = dx *. dx +. dy *. dy +. dz *. dz in
      let mag = dt /. (dist2 *. sqrt(dist2)) in

      b.vx <- b.vx -. dx *. b_o.mass *. mag;
      b.vy <- b.vy -. dy *. b_o.mass *. mag;
      b.vz <- b.vz -. dz *. b_o.mass *. mag;

      b_o.vx <- b_o.vx +. dx *. b.mass *. mag;
      b_o.vy <- b_o.vy +. dy *. b.mass *. mag;
      b_o.vz <- b_o.vz +. dz *. b.mass *. mag;
    done
  done
```

N-Body serial

All experiments on an 2x Xeon E5-2695 v4

```
real    1m23.423s
user    1m23.422s
sys      0m0.000s
```

(256 iterations, 8192 bodies)

How fast can we go?

Amdahl's law for parallel programs:

$$\frac{1}{(1-p) + (\frac{p}{s})}$$

p = proportion of parallelisable code

s = degree of parallelism

Linux Perf

Sampling profile of `nbody_serial.exe` run:

	Children	Self	Command	Shared Object	Symbol
+	99.95%	0.00%	nbody_serial.ex	nbody_serial.exe	[.] caml_start_program
+	99.95%	0.00%	nbody_serial.ex	nbody_serial.exe	[.] caml_program
+	99.95%	0.00%	nbody_serial.ex	nbody_serial.exe	[.] camlDune__exe__Nbody_serial__entry
+	99.55%	99.55%	nbody_serial.ex	nbody_serial.exe	[.] camlDune__exe__Nbody_serial__advance_90
	0.34%	0.34%	nbody_serial.ex	nbody_serial.exe	[.] camlDune__exe__Nbody_serial__energy_159
	0.05%	0.05%	nbody_serial.ex	nbody_serial.exe	[.] camlDune__exe__Nbody_serial__update_152
	0.03%	0.00%	nbody_serial.ex	nbody_serial.exe	[.] _start

Perfect scalability ⇨ ~220x speedup

If $p < 0.89$, max speedup single digits!

Iteration 1: Domain per body

Don't do this

```
let advance bodies n_bodies dt =
  let ds =
    Array.init n_bodies (fun i -> Domain.spawn (fun _ ->
      let b = bodies.(i) in
      for j = 0 to n_bodies - 1 do
        let b_o = bodies.(j) in
        if (i!=j) then begin
          let dx = b.x -. b_o.x  and dy = b.y -. b_o.y
            and dz = b.z -. b_o.z in
          let dist2 = dx *. dx +. dy *. dy +. dz *. dz in
          let mag = dt /. (dist2 *. sqrt(dist2)) in
          b.vx <- b.vx -. dx *. b_o.mass *. mag;
          b.vy <- b.vy -. dy *. b_o.mass *. mag;
          b.vz <- b.vz -. dz *. b_o.mass *. mag;
        end
      done
```

Iteration 1: Domain per body

Don't do this

```
let ds =
 Array.init n_bodies (fun i -> Domain.spawn (fun _ ->
  let b = bodies.(i) in
  for j = 0 to n_bodies - 1 do
   let b_o = bodies.(j) in
   if (i!=j) then begin
    let dx = b.x -. b_o.x  and dy = b.y -. b_o.y
     and dz = b.z -. b_o.z in
    let dist2 = dx *. dx +. dy *. dy +. dz *. dz in
    let mag = dt /. (dist2 *. sqrt(dist2)) in
    b.vx <- b.vx -. dx *. b_o.mass *. mag;
    b.vy <- b.vy -. dy *. b_o.mass *. mag;
    b.vz <- b.vz -. dz *. b_o.mass *. mag;
   end
  done
 )) in
```

Iteration 1: Domain per body

Don't do this

```
   Array.init n_bodies (fun i -> Domain.spawn (fun _ ->
    let b = bodies.(i) in
    for j = 0 to n_bodies - 1 do
     let b_o = bodies.(j) in
     if (i!=j) then begin
      let dx = b.x -. b_o.x  and dy = b.y -. b_o.y
       and dz = b.z -. b_o.z in
      let dist2 = dx *. dx +. dy *. dy +. dz *. dz in
      let mag = dt /. (dist2 *. sqrt(dist2)) in
      b.vx <- b.vx -. dx *. b_o.mass *. mag;
      b.vy <- b.vy -. dy *. b_o.mass *. mag;
      b.vz <- b.vz -. dz *. b_o.mass *. mag;
     end
    done
   )) in
  Array.iter (Domain.join) ds
```

Iteration 1: Domain per body

Oops.

```
real     8m10.965s
user     25m24.372s
sys       11m30.816s
```

Domains are heavyweight

Aim for same number as cores

Spawn/join infrequently

Domainslib

- Task pool
  - Parallel
    - for / reduce / scan
  - Async/await
- Channels

https://github.com/ocaml-multicore/domainslib/

Iteration 2: Domainslib

```
let advance pool n_bodies n_domains bodies dt =
 T.parallel_for pool
  ~chunk_size:(n_bodies/n_domains)
  ~start:0
  ~finish:(n_bodies - 1)
  ~body:(fun i ->
   let b = bodies.(i) in
   for j = 0 to n_bodies - 1 do
    let b_o = bodies.(j) in
    if (i!=j) then begin
     let dx = b.x -. b_o.x  and dy = b.y -. b_o.y
      and dz = b.z -. b_o.z in
     let dist2 = dx *. dx +. dy *. dy +. dz *. dz in
     let mag = dt /. (dist2 *. sqrt(dist2)) in
     b.vx <- b.vx -. dx *. b_o.mass *. mag;
     b.vy <- b.vy -. dy *. b_o.mass *. mag;
```

Iteration 2: Domainslib

```
    ~finish:(n_bodies - 1)
    ~body:(fun i ->
      let b = bodies.(i) in
      for j = 0 to n_bodies - 1 do
        let b_o = bodies.(j) in
        if (i!=j) then begin
          let dx = b.x -. b_o.x  and dy = b.y -. b_o.y
            and dz = b.z -. b_o.z in
          let dist2 = dx *. dx +. dy *. dy +. dz *. dz in
          let mag = dt /. (dist2 *. sqrt(dist2)) in
          b.vx <- b.vx -. dx *. b_o.mass *. mag;
          b.vy <- b.vy -. dy *. b_o.mass *. mag;
          b.vz <- b.vz -. dz *. b_o.mass *. mag;
        end
      done
```

Iteration 2: Domainslib

Cores	Time	Vs Serial	Vs Self
1	128.076s	0.65x	1x
2	54.987s	1.51x	2.32x
3	42.577s	1.94x	3.00x
4	32.753s	2.53x	3.90x
8	18.868s	4.39x	6.78x
16	11.438s	7.28x	11.19x
24	8.465s	9.80x	15.12x

Shared state pitfalls

Minimise writes to frequently read shared state
Avoid contended writes to shared state

Use `perf stat`, `perf record` and `perf c2c`

```
perf stat -e cycles,cycle_activity.cycles_no_execute,cycle_activi

   458,432,211,938      cycles
    74,023,598,176      cycle_activity.cycles_no_execute
    67,741,246,568      cycle_activity.stalls_mem_any
       481,904,795      cache-misses
        14,350,052      mem_load_uops_l3_miss_retired.remote_hitm

       8.726805280 seconds time elapsed
```

Iteration 3: Avoiding contention

```
type planet_pos = { mutable x : float;  mutable y : float;  mu
type planet_vec = { mutable vx: float;  mutable vy: float;  mu

let advance pool n_domains n_bodies bodies_pos bodies_vec dt =
  T.parallel_for pool
    ~chunk_size:(n_bodies/n_domains)
    ~start:0
    ~finish:(n_bodies - 1)
    ~body:(fun i ->
      let bp = bodies_pos.(i) in
      let bv = bodies_vec.(i) in
      let vx, vy, vz = ref bv.vx, ref bv.vy, ref bv.vz in
      for j = 0 to n_bodies - 1 do
        let bp' = bodies_pos.(j) in
        if (i!=j) then begin
          let dx = bp.x -. bp'.x  and dy = bp.y -. bp'.y  and
```

Iteration 3: Avoiding contention

```
let advance pool n_domains n_bodies bodies_pos bodies_vec dt
  T.parallel_for pool
    ~chunk_size:(n_bodies/n_domains)
    ~start:0
    ~finish:(n_bodies - 1)
    ~body:(fun i ->
      let bp = bodies_pos.(i) in
      let bv = bodies_vec.(i) in
      let vx, vy, vz = ref bv.vx, ref bv.vy, ref bv.vz in
      for j = 0 to n_bodies - 1 do
        let bp' = bodies_pos.(j) in
        if (i!=j) then begin
          let dx = bp.x -. bp'.x  and dy = bp.y -. bp'.y  and
          let dist2 = dx *. dx +. dy *. dy +. dz *. dz in

          let mag = dt /. (dist2 *. sqrt(dist2)) in
          let mass = bp'.mass in
```

Iteration 3: Avoiding contention

```
    let vx, vy, vz = ref bv.vx, ref bv.vy, ref bv.vz in
    for j = 0 to n_bodies - 1 do
      let bp' = bodies_pos.(j) in
      if (i!=j) then begin
        let dx = bp.x -. bp'.x  and dy = bp.y -. bp'.y  and
        let dist2 = dx *. dx +. dy *. dy +. dz *. dz in

        let mag = dt /. (dist2 *. sqrt(dist2)) in
        let mass = bp'.mass in
        vx := !vx -. dx *. mass *. mag;
        vy := !vy -. dy *. mass *. mag;
        vz := !vz -. dz *. mass *. mag;
      end
    done;
    bv.vx <- !vx;
    bv.vy <- !vy;
    bv.vz <- !vz);
```

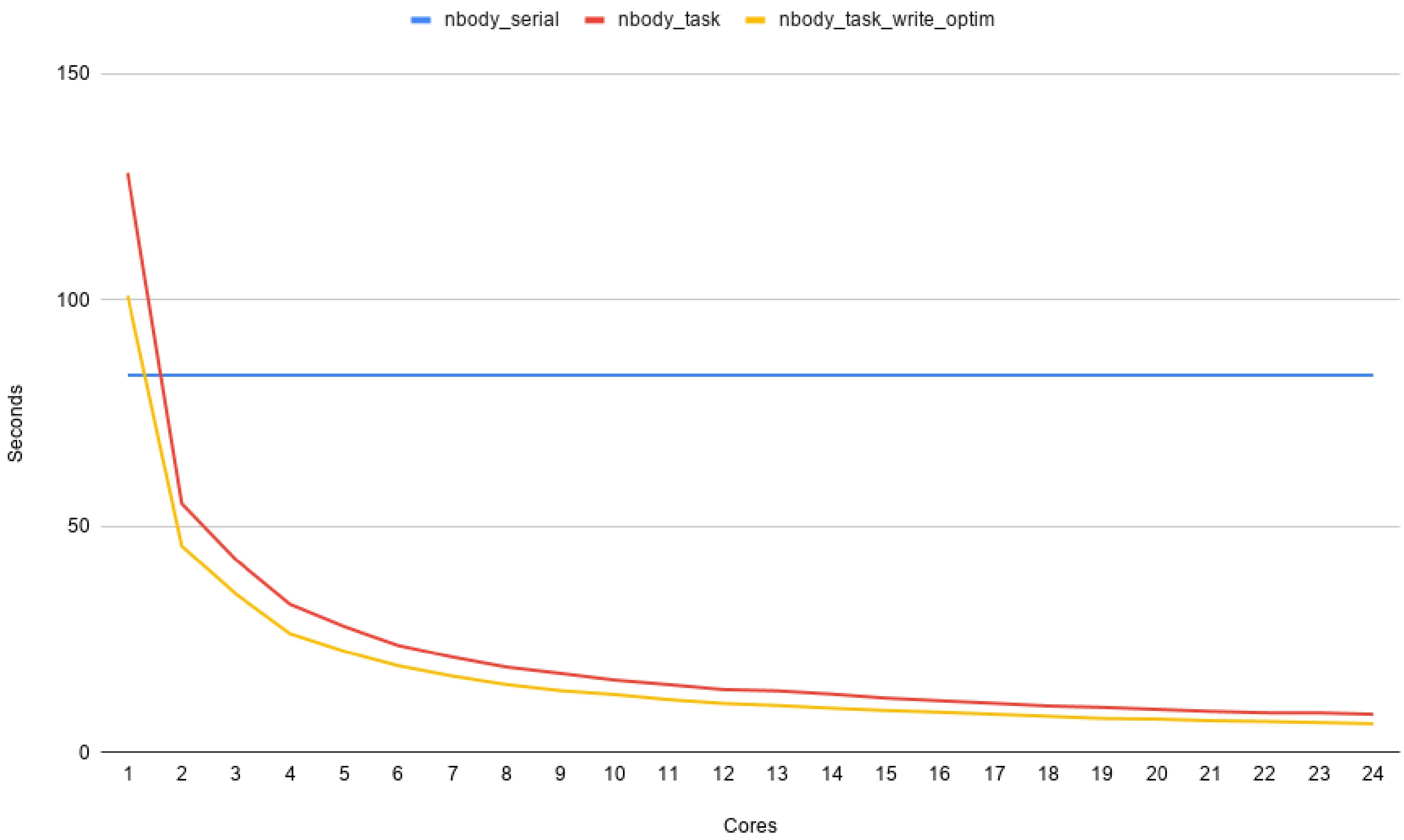
Iteration 3: Avoiding contention

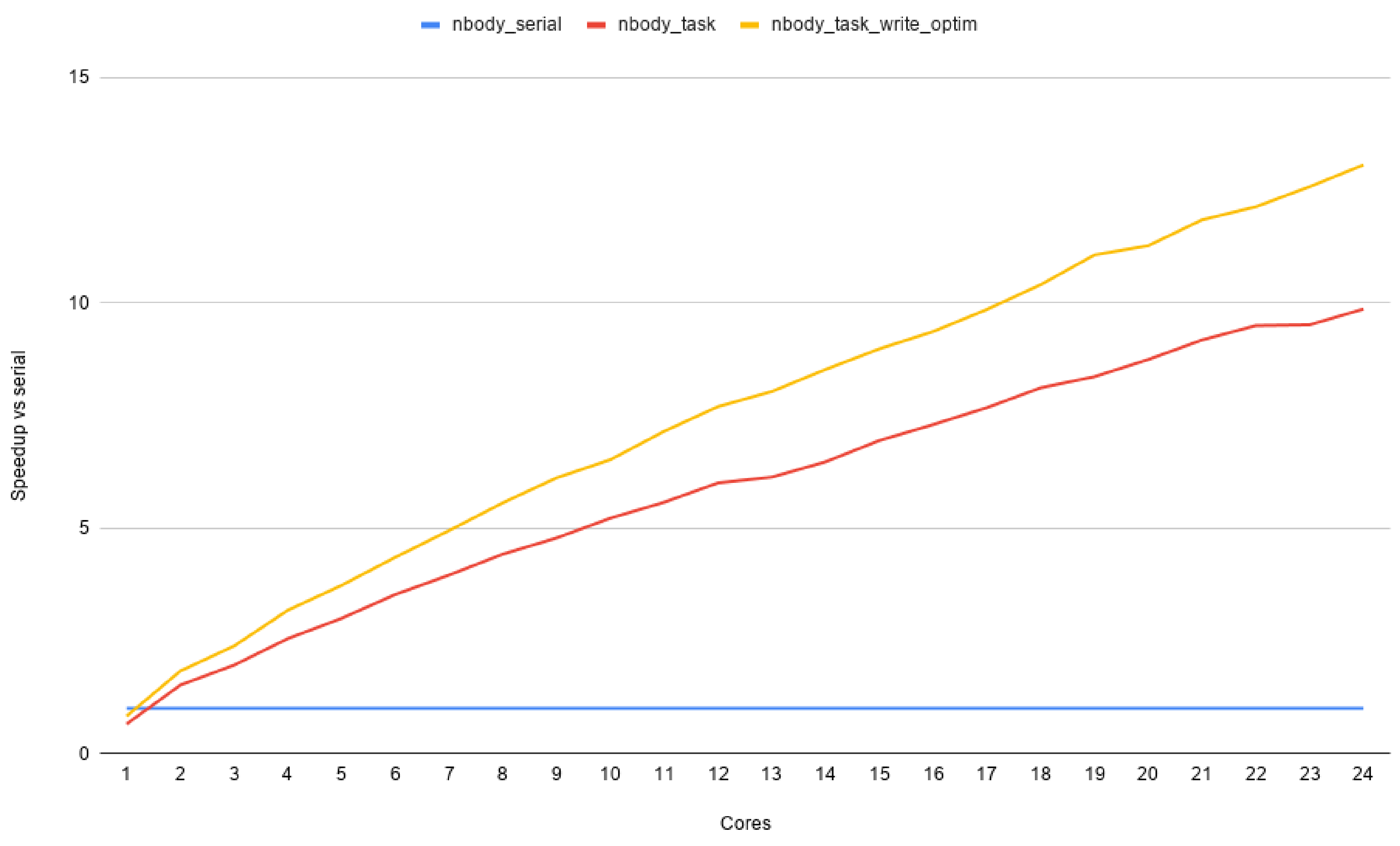
```
perf stat -e cycles,cycle_activity.cycles_no_execute,cycle_activi

   341,067,775,833      cycles
    31,523,253,083      cycle_activity.cycles_no_execute
    29,569,589,458      cycle_activity.stalls_mem_any
         7,009,375      cache-misses
           956,792      mem_load_uops_l3_miss_retired.remote_hitm

       6.639116502 seconds time elapsed
```

~35% speedup at 24 cores


nbody_serial
nbody_task
nbody_task_write_optim
150
100
50
0
Seconds
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
Cores



nbody_serial
nbody_task
nbody_task_write_optim
15
10
5
0
Speedup vs serial
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
Cores

Take aways

- Multicore is ready to use, upstreaming in progress
- Profile to understand serial performance
- Use Domainslib abstractions to add parallelism
- Share work as coarsely as you can
- Avoid writing to shared state as much as possible

Use the multicore github issue tracker if you find unexpected behaviour