Parallelising your OCaml code with Multicore OCaml

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

Speaker notes

Extended notes will go here.

You can get the slides, speaker notes and runnable examples for this talk at https://github.com/ocaml-multicore/ocaml2020-workshop-parallel

Overview

- Multicore OCaml
- Domains
- Domainslib
- Further optimisation

Multicore OCaml

- Concurrency is overlapping computations
- Parallelism is simultaneous computations

Multicore OCaml ⇨ concurrency *and* shared-memory parallelism

Speaker notes

Multicore OCaml is a project to bring concurrent *and* shared-memory parallelism to OCaml. It features a parallel stop-the-world minor collector and a concurrent, parallel major collector.

To learn more about how Multicore OCaml's internals work you can read our ICFP2020 paper Retrofitting Parallelism onto OCaml. To get started with installation there are instructions on the multicore opam repo.

From a user's perspective, Multicore OCaml brings fibers and effects for concurrency and domains for parallelism.

What do we mean by concurrency vs parallelism? Concurrency is how we partition multiple computations such that they can run in overlapping time periods rather than strictly sequentially. Parallelism is the act of running multiple computations simultaneously, primarily by using multiple cores on a multicore machine.

There are degrees of support for both concurrency and parallelism. The multicore wiki gives a good overview and notes on the current situation. The short summary though is that there is not strong language support for direct-style concurrency *and* parallelism in OCaml code.

Multicore OCaml

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

Speaker notes

Multicore maintains compatibility with existing OCaml code, supporting the existing C API along with tricky parts of the language like ephemerons and finalisers.

There is a 'no effect' branch that removes some syntax extensions and maintains ppx compatibility. If your existing code breaks, let us know on the Multicore issue tracker.

Upstreaming is already in progress and you can see PRs that have already landed. The plan is that OCaml 5 will feature domains-only parallelism. Effects and fibers will follow in a later release.

With that in mind, this talk will focus only on parallelism via Domains.

Domains

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

Speaker notes

So what exactly are Domains? A Domain is a unit of parallelism. It is essentially an operating system thread with additional state for managing OCaml's Garbage Collector. Domains are heavyweight and you should aim to only have as many as you have cores. It's also a good idea to create and destroy them as infrequently as possible.

Domains come with an interface that provides the following functionality:

- Spawning and joining domains
- Waiting on notifications from other domains and notifying other domains
- Atomic memory operations (technically this in in `Atomic`)
- Domain-local storage (for values that are cached per-domain)

N-Body

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

Speaker notes

In this talk we're going to iterate on a benchmark and show how we can use some abstractions offered in Domainslib along with performance profiling to parallelise it.

The benchmark we're going to use is n-body, it models the orbit of a number of bodies around a solar body. It's a useful example to use because the code responsible for the bulk of processing time can just about fit on a slide and we can get some substantial parallel speedups without changing the actual algorithm (too much).

N-Body serial

```ocaml
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
```

Speaker notes

This is the core of the n-body benchmark.

Line 2 to 5 of the advance function set up two nested loops between the bodies Line 6 to 9 we calculate the distance between them Line 11 to 17 we update both bodies velocities based on their mutual interaction

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)

Speaker notes

We're running all benchmarks in this talk on a 2x Xeon E5-2695 v4 machine.

You can see the initial serial implementation takes just over 83 seconds to run and that it's all user time. This is expected, there's no IO or anything that should involve the system.

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

Speaker notes

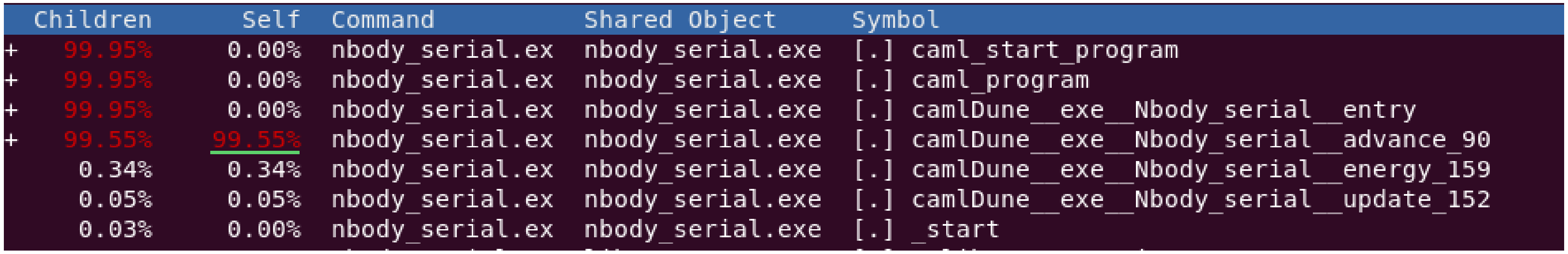
Before parallelising some code, we should some expectations in mind for how fast we can actually go. It turns out there's a reasonable simple way of doing this. We something called Amdahl's law. For parallel programs this essentially states that you are limited by the un-parallelisable proportion of your program.

It makes intuitive sense if we think it through. If you have a program where you can only parallelise 50% of it, no matter how many cores we throw at it we're only going to get a 2x speedup. We're stuck with that 50% of serial code.

If we assume we have unlimited cores available then (p/s) becomes 0 and the maximum speedup we could achieve is (1 / (1-p)).

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!

Speaker notes

But where do we get the numbers to plug in to Amdahl's law? That's where performance profiling comes in. For this we're going to use Linux's perf profiler. If you've not familiar with perf I would suggest Brendan Gregg's website as a good resource.

Running Linux's perf on our benchmark as so:

```
perf record --call-graph dwarf ./nbody_serial.exe 256 8192
```

We can get a profile that tells us how much time we spent in the `advance` function we wish to parallelise. At 99.55% of CPU time, the maximum speedup we could achieve is ~220x and so this is a reasonably good candidate to parallelise.

Something to take away from this is that if the only hotspot you can speed up takes up less than 90% of your runtime in the serial program, the maximum speedup you can achieve will be single digits.

To get beyond that you will need to make algorithmic changes or consider parallelism at a higher level.

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
```

Speaker notes

This is an example of what not to do. We've encountered it a few times when helping people add parallelism to their code. We'll see on the next slide why this is a bad idea.

Line 3 we spawn a Domain per body Line 5 to 10 for each body we iterate over every other body and update state as before Line 18 we join all the Domains, waiting for each one to complete

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

Speaker notes

As you can see, this did not end well. Domains are heavyweight and not intended to be spawned/joined frequently (this is where Multicore's fibers will come in handy). You should aim to only have as many as you have cores.

Accompanying Multicore (though not part of the compiler or standard library) is a set of parallelism utilities in Domainslib. These provide abstractions that are normally easy to fit to existing patterns in your code and provide good scalability.

Domainslib

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

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

Speaker notes

Domainslib primarily provides a Task Pool which maintains a pool of Domains that will pull work from a queue of tasks. There are various ways of adding tasks to the pool, via parallel for/reduce/scan functions or with an async/await mechanism. This tutorial we'll only use the parallel_for method but there may be others that are a better fit for your problem.

Domainslib also provides Channels - we won't be discussing these today.

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
```

Speaker notes

This code is almost identical to the Domain example we saw in iteration 1 but instead of spawning a Domain for each body, we're using a parallel for loop which adds tasks to a Domainslib task pool.

Line 2 to 6 we set up the parallel for loop. Very similar to a normal OCaml for loop, chunk_size lets us hand large chunks of work to the domains in the pool at a time - reducing overhead. chunk_size makes a big difference to scalability - you want as large a chunk size as you that's still divisible by the number of domains in the pool.

Line 7 to 16 is the same as iteration 1

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

Speaker notes

For a very simple change the pay off has been pretty good. At 24 cores we're seeing a nearly 10x speedup compared to the original serial implementation. Note that we're slower than serial on only one core.

Can we do better though?

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
```

Speaker notes

There's a lot to cover here and there might be a bit of handwaving because we've got little time left but it's important because this is the second biggest issue we see people hit when adding parallelism to their code.

In short when it comes to shared-memory parallelism, reading shared state on multiple cores scales well but writing to shared state scales poorly. This is down to how cache coherency works. Writes causes cache invalidation which cause expensive cache misses on readers. Writes that contend with other writes are even worse.

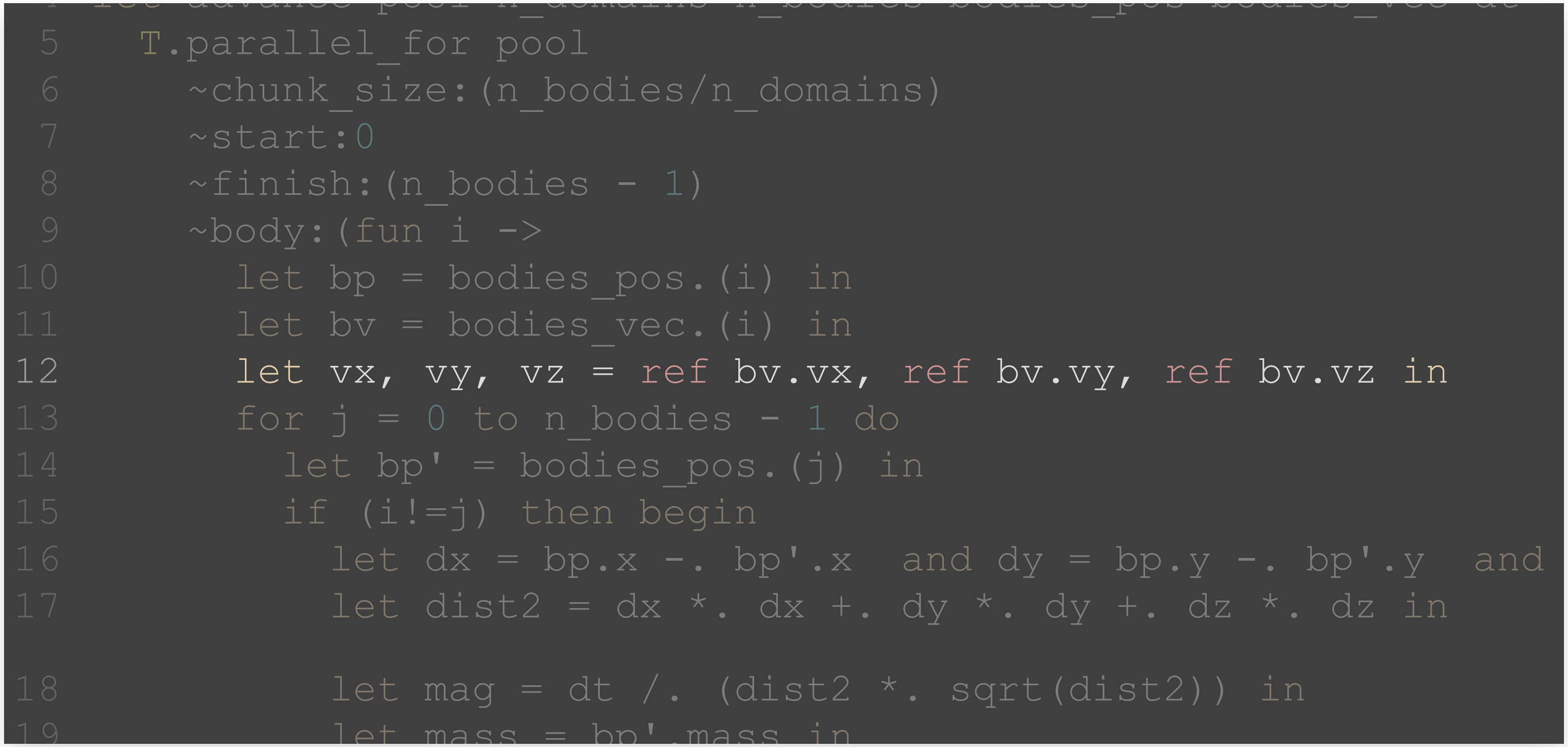
When possible cache changes to shared global state and apply it infrequently if you can. For example, instead of incrementing a set of global counters in a loop, you could maintain counters local to a Domain and then drain them to the global counters periodically.

Here we use `perf stat` to check how often we're stalling when the benchmark is running and how many of those stalls are becaues we're waiting on some kind of memory (cache or main memory). There's also a specific event for when we're hitting a modified cache line that's in another core's cache. We can see we're spending about 15% of our time stalled.

Iteration 3: Avoiding contention

```ocaml
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



```
  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);
```

Speaker notes

We can restructure this code to avoid shared state write contention by seperating the position and velocity data for the planets since the positions are read very frequently by all domains but written infrequently only from one, and the velocities are written frequently by all domains but read infrequently by only one.

Also instead of writing to the velocities each time, our domains cache the velocity updates and apply them only at the end of the iteration. The strategy we mentioned earlier.

Line 1 to 2 we split the positions and velocities and store them independently to avoid them sharing a cache line, which we would contend when updating velocities. Line 12 and 25-27 instead of writing to the velocities every iteration, we keep a local counters and then only update global state when done.

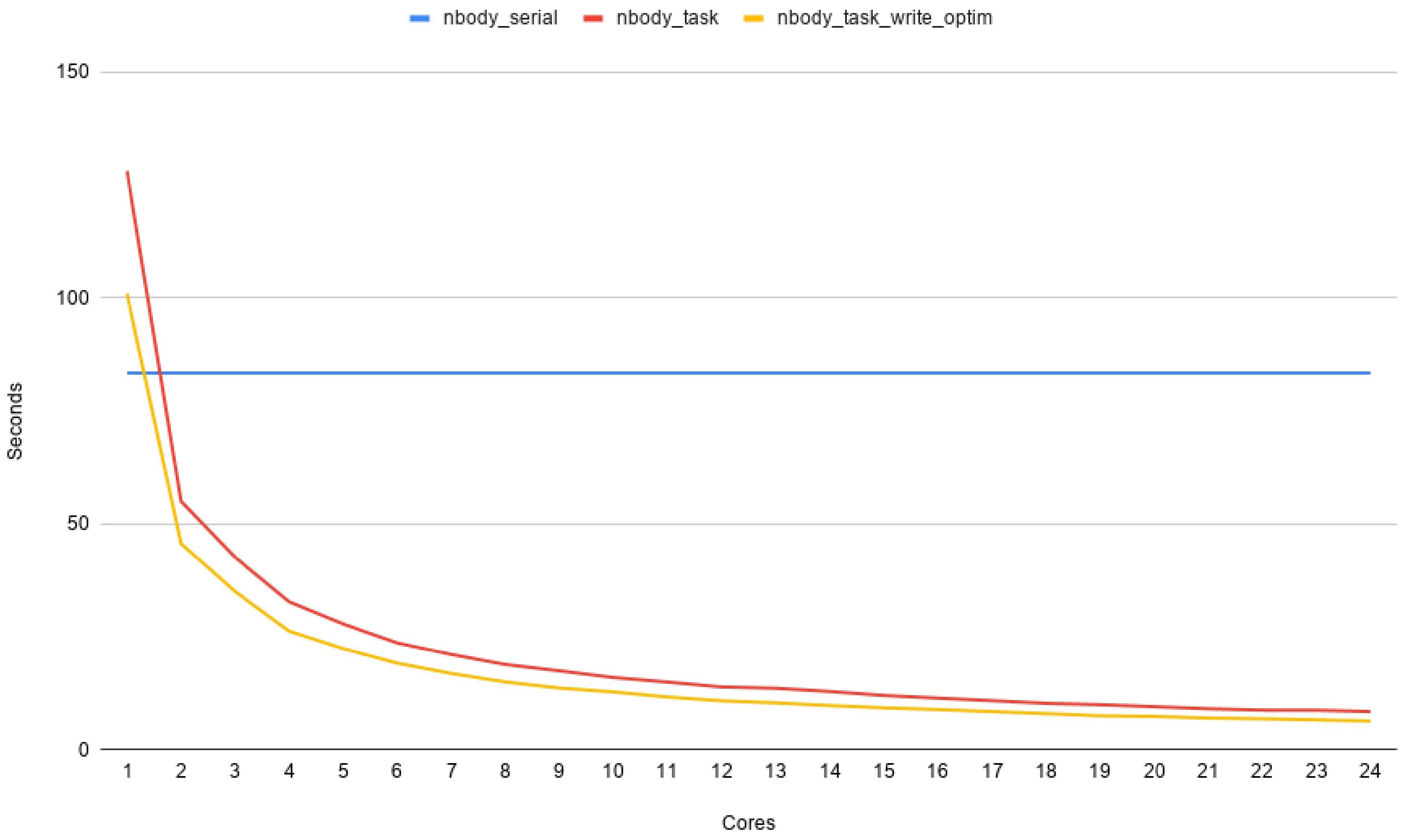
Iteration 3: Avoiding contention

```
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

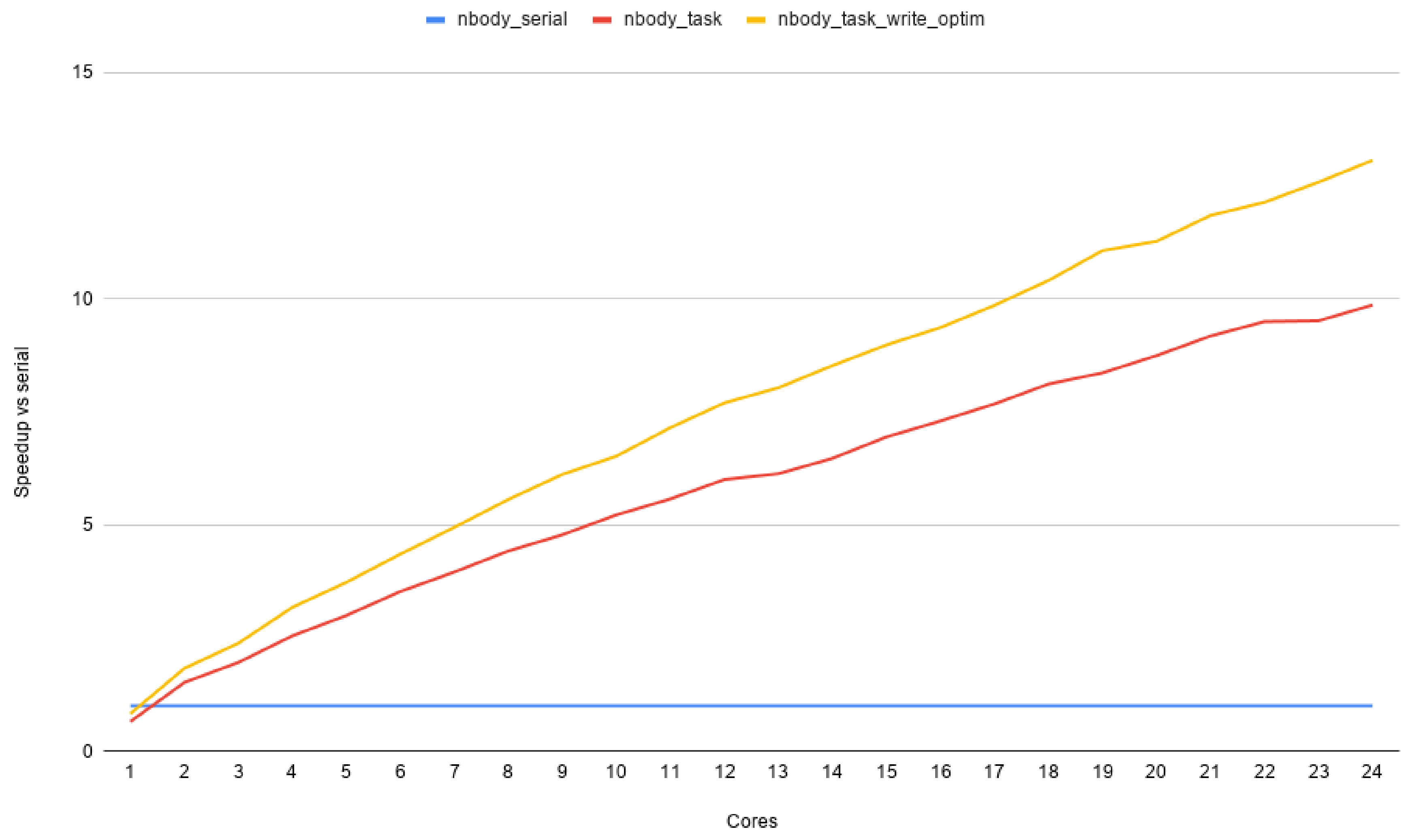
Speaker notes

So we can see that applying the principles we talked about in the earlier slide has made a big difference as we scaled up the number of 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

Speaker notes

This shows the time to run for the benchmark iterations at different core counts. More interesting is the scalability graph on the next slide.


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

Speaker notes

We can see a consistent speedup with more cores for our contention optimised verison of the benchmark. Interestingly at 24 cores profiling indicates that 97.6% of CPU time is being spent in the `advance` function and more specifically the parallel for loop over bodies. That we're not seeing a greater speedup indicates we might actually be hitting some other limit.

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

Speaker notes

The main things you should take away from this talk is that:

1. While there is still active development happening, Multicore is ready for use and the upstreaming process is on-going as we speak.
2. Before parallelising your code you should profile the serial implementation to understand where you might want to introduce parallelism.
3. Domainslib probably has an abstract that will work for you. It's generally performant and maintained.
4. Experiment with different granularities of jobs until you find something that trades off work imbalance and communication overhead. It will depend on the job.

We should be around now if you have any questions and if you run in to any unexpected behaviour then please create an issue on the Github multicore repo.

Lastly there was vastly more than we could reasonably cover in twenty minutes, you should read Sudha's Parallel Programming chapter that covers many things we had to leave off here.