Parallel Sequences in Multicore OCaml

Andrew Tao
Advised by Professor Andrew Appel

July 31, 2023

1. Introduction

I present my implementation of a parallel sequences abstraction that utilizes the support for shared-memory parallelism in the new OCaml 5.0.0 multicore runtime.[1] This abstraction allows clients to create highly parallelizable programs without needing to write, or even understand, the low-level implementation details necessary to parallelize large tasks. The full list of supported operations can be seen in Table 1.

Operation	Description
`iter f s`	Apply `f` to every element of `s` sequentially
`iteri f s`	Apply `f` to every index and value of `s` sequentially
`length s`	Return the length of sequence `s`
`empty ()`	Create an empty sequence
`singleton x`	Create a sequence containing only the value `x`
`nth s i`	Get the element at index `i` of `s`
`cons x s`	Add `x` to the beginning of `s`
`tabulate f n`	Create a sequence with $\forall i \in [0,n)$, `nth s i = f i`
`repeat x n`	Create a sequence by repeating `x` `n` times
`append s1 s2`	Create a sequence by appending `s2` to `s1`
`seq_of_array a`	Create a sequence containing every element of `a`
`array_of_seq s`	Create an array containing every element of `s`
`zip s1 s2`	Zip two sequences of type `'a seq` and `'b seq` into one sequence of type `('a * 'b) seq`
`split s i`	Split sequence of length n into two sequences of $[0,i)$ and $[i,n)$
`map f s`	Create a sequence by applying `f` to every element of `s`
`reduce g b s`	Combine elts of `s` using function `g` and base value `b`
`map_reduce f g b s`	Map values of `s` with `f`, then reduce with `g` and `b`
`scan g b s`	Return inclusive scan of function `g` with base value `b`
`flatten ss`	Flatten nested sequence `ss`
`filter p s`	Remove elts of `s` which do not satisfy predicate `p`

Table 1: Supported operations in the Parallel Sequence abstraction.

As an example, consider the (somewhat contrived) function which calculates the sum of the first n terms of the harmonic series in parallel:

[1] See https://github.com/aytao/parallel_seq

```
let harmonic_sum (n : int) : float =
  let denominators = tabulate (fun x -> x + 1) n in
  let terms = map (fun x -> 1. /. (float_of_int x)) denominators in
  reduce ( +. ) 0. terms
```

We see that by leveraging the abstraction provided, we can ignore all of the underlying low-level details of how parallelism is actually being implemented under the hood, and instead focus on the basic computations and steps we want to perform.

While similar parallel abstractions are already a well-established field of study, since OCaml lacked a parallel runtime until the release of 5.0.0, the language currently has very little support for high-level parallel constructs. As a result, my work attempts to provide a good starting point for the development of high-level shared memory parallelism constructs in Multicore OCaml.

2. Related Work

Much of my work was heavily influenced by Blelloch's work on the NESL programming language [1]. In NESL, the language provides sequences as a primitive parallel data type, and parallelism in the language is achieved exclusively by performing operations on those sequences. These sequence functions have been a crucial source of inspiration for the operations supported by my implementation, and a vast majority of the operations in my work are similar or identical to sequence functions supplied by NESL.

The other major source of inspiration for my work has been the parallel sequences abstraction used as a pedagogical tool in the Princeton University course COS 326, "Functional Programming" [8]. This abstraction has been taught since long before the addition of multicore support to OCaml in version 5.0.0, so none of the previously existing implementations of this abstraction are able to leverage the performance benefits of shared-memory parallelism. Because of this, a major goal for my work was to create an implementation that can be immediately used in the course.

3. Approach

In OCaml, the units of parallelism are domains, which are low-level, heavy-weight entities [3]. It is recommended that rather than work with domains directly, users should utilize higher-level parallel programming libraries such as domainslib [4]. The domainslib library is implemented using a work-stealing queue [4] and provides several useful parallel programming operations, including `async` and `await` on promises and a `parallel_for`. Using these operations in my implementation allowed me to divert my attention from the details of task scheduling and instead focus on the high-level decisions regarding algorithmic design.

The module I implemented uses an array as the underlying type. Arrays allow for a significant number of simple operations to be done in constant work and span, most notably giving a simple way to access any element in constant work. Additionally, the use of arrays allows for mutability, which is critical for performance during sequence creation. The module interface prevents clients from modifying a sequence in any way after creation, so mutability is only used under the hood for performance and the module can still be used as a functional interface.

3.1. Issues with Array Initialization

Unfortunately, the use of arrays introduces its own set of problems that result from the difficulty in handling array initialization, especially when trying to do this in parallel. First, we note that the only

operations in OCaml's array module that create new arrays, `make` and `init`, either take in a default value for every element of the array or take in a function to initialize each element [3].[2] Using `init` would require us to do all of our work sequentially, which would be an unacceptable performance hit for all of our sequence creation operations. On the other hand, the `make` operation is also difficult to work with due to the strict type system in OCaml. Consider the `tabulate` function from our interface:

```
val tabulate : (int -> 'a) -> int -> 'a t
```

We see that the `tabulate` function does not (and should not ever need to) know anything about the type `'a`. However, with the way that `make` works, if we wanted to create an empty `'a array` to store the eventual results of each function application, we'd have to know some default value for `'a`, which we do not yet know at the beginning of `tabulate`, before we've performed any function applications.

In an ideal world, we'd want to have something like

```
val get_uninitialized_array : int -> 'a array
```

that just gives us some polymorphic, uninitialized array. This seems like this could be relatively easily done by just getting a large uninitialized block by using a C stub with a call to one of the `caml_alloc` functions. However, we need to be able to eventually hold pointers in our array, and since OCaml 5.0.0 removed the `Obj.set_tag` operation [6], the only time that blocks can be marked as garbage collected is at creation; as a result, we need our blocks to be safe for the garbage collector at all times. Since the garbage collector in OCaml only distinguishes integers from pointers using the lowest bit, and it's certainly not safe for the garbage collector to process a large block of completely garbage pointers, we would have to ensure that any uninitialized array we created has every field set to some safe, non-pointer value.

To address this, my implementation relies on an `Array_handler` module, which uses a C stub to implement the function `get_uninitialized` which returns a polymorphic uninitialized array. The C stub is a modified `Array.make` [5] that sets every field of a newly allocated block to a "safe" nonpointer value (a value with a set lowest bit) before returning the block to be used as an array. This function saves time over `Array.make` since we know that the value being used is not a pointer, and so we can remove a few redundant checks and operations.

I concede that relying on a fast but ultimately sequential subroutine for array creation is certainly not ideal, but I believe this concession is acceptable for the following reasons:

1. The array initialization is relatively fast. In most realistic applications of the abstraction (see Section 4), the time spent on getting an array takes up a negligible fraction of the runtime.
2. It is entirely feasible that some day, a better way of creating an uninitialized array in parallel is discovered (or some future update to OCaml adds this to one of the standard library modules). We could then replace the current subroutine with this new way without having to modify anything else.

[2] Technically, the module also provides a method for creating an uninitialized, unboxed-float array. However, this resulting array is not garbage collected and so is not useful for us, since we want to store garbage collected pointers in our array.

3.2. Function Implementations

Once array initialization has been properly handled, the implementation is much more straightforward. We begin with the implementation of `tabulate`:

```
let tabulate (f : int -> 'a) (n : int) : 'a t =
  if n = 0 then empty ()
  else if n < 0 then
    raise (Invalid_argument "Parallel_seq.tabulate")
  else
    let arr : 'a array = Array_handler.get_uninitialized n in
    parallel_for n (fun i -> arr.(i) <- f i);
    arr
```

Once tabulate is implemented, many other functions can be implemented using relatively simply. For example, consider the implementations of `map` and `append`:

```
let map (f : 'a -> 'b) (s : 'a t) : 'b t =
  let body idx = f s.(idx) in
  tabulate body (length s)

let append (s1 : 'a t) (s2 : 'a t) : 'a t =
  let len1, len2 = (length s1, length s2) in
  let body (idx : int) : 'a =
    if idx < len1 then s1.(idx) else s2.(idx - len1)
  in
  tabulate body (len1 + len2)
```

We see that these functions are derived directly from tabulate, with just the addition of defining a very simple body function to be done in parallel. Many functions in the interface can be implemented with this approach, including `cons`, `repeat`, `seq_of_array`, `array_of_seq`, `zip`, and `split`.

For the "combine" operations, `reduce` and `scan`, my implementations both use the classic Ladner and Fischer parallel scan algorithm [2]. To implement `reduce`, I simply stop halfway through the algorithm, and sum together the necessary partial sums. As a slight modification from the original algorithm, at each layer, my algorithm combines a sequential cutoff constant k number of elements at a time, instead of always 2 at a time. This helps facilitate better memory locality and reduce the number of layers needed. It is worth noting that the `scan` function implements an inclusive scan; that is, when calculating `y = scan(x)`, the value `y[i]` will consider the value `x[i]`. However, an exclusive scan can very easily be recovered by simply shifting the result by 1 index and replacing `y[0]` with the base value.

Finally, let us examine the implementations for `flatten`, an operation which flattens a sequence of nested sequences into a single sequence, and `filter`. In `flatten`, we perform an exclusive integer addition scan over the lengths of each nested sequence to get the offset that each nested sequence should have in the final array, and then copy over each nested sequence in parallel. In `filter`, we begin by mapping each element to 1 if it should be kept, or 0 if it should be filtered out. Then, we perform an exclusive integer addition scan over this bit map to get both the length of the final array as well as the indices where each included element should be in the final array. Finally, we copy over the kept elements in parallel.

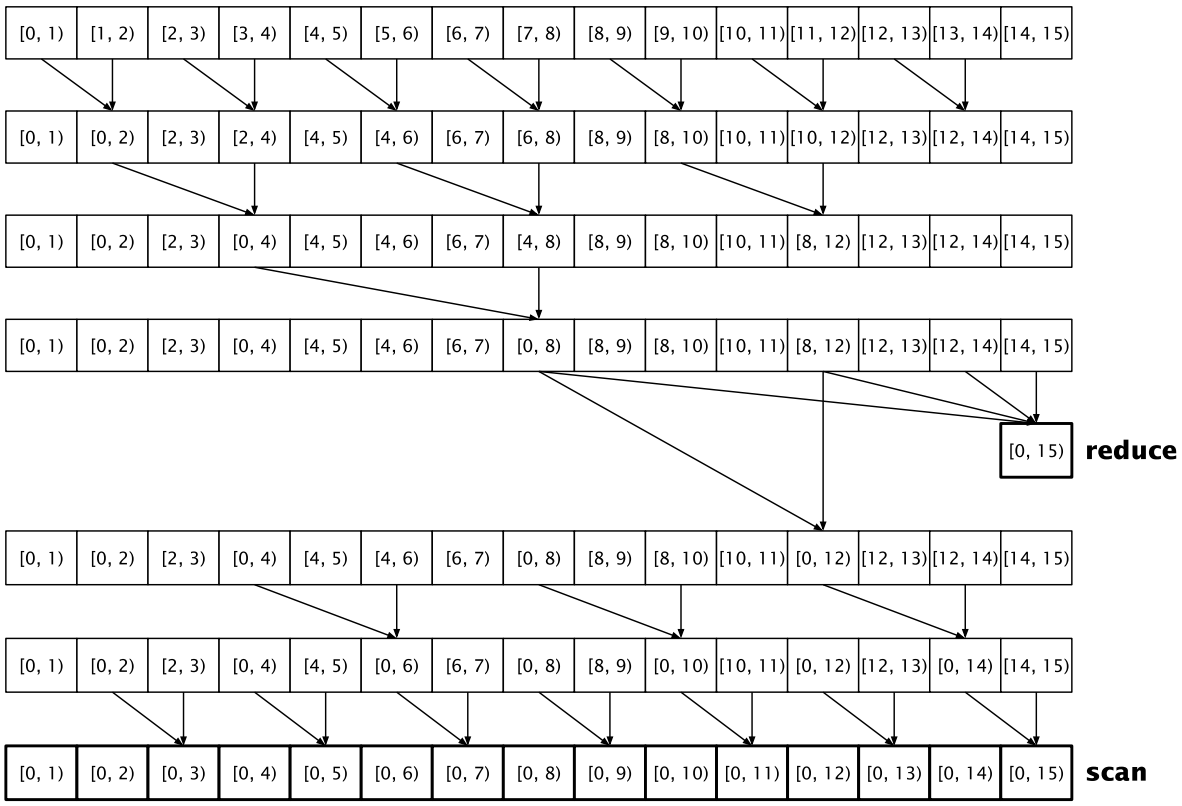

Figure 1: Visual representations of `scan` and `reduce` for $k = 2$

4. Evaluation

We analyze the speedup we receive compared to a purely sequential implementation of the module. This purely sequential implementation is heavily based on the array implementation of the COS 326 parallel sequences abstraction used in one of the course assignments [7]. Figure 2 shows the speedup for a block matrix multiplication performed on two 2000×2000 float matrices, where the size of each block used was 100×100. Figure 3 shows the speedup for indexing a 822MB file of 1 million generated pages into an inverted index. The pages used were generated from a file of pages used in the "Parallel Sequences" assignment of Princeton's COS 326 course [7]. All benchmarks were run on a machine with 64 processors, each a Intel(R) Xeon(R) Gold 5218 CPU at 2.30GHz, with 500GB of RAM.

5. Limitations and Future Work

5.1. Memory Locality

One notable limitation of my work is its lack of consideration for memory locality during task scheduling, resulting from its dependence on the work-stealing queue approach offered by domainslib. Since the domainslib library does not provide the ability to influence which domain actually ends up performing a task, we can lose quite a bit of time simply due to cache misses. To

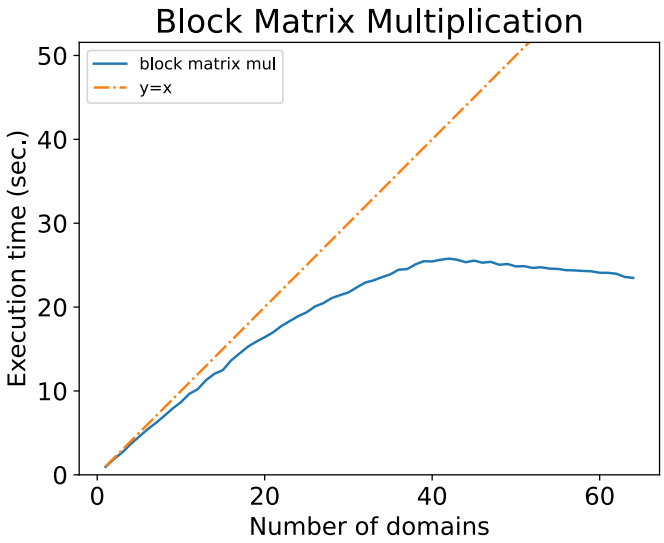

Figure 2: Speedup for block matrix multiplication

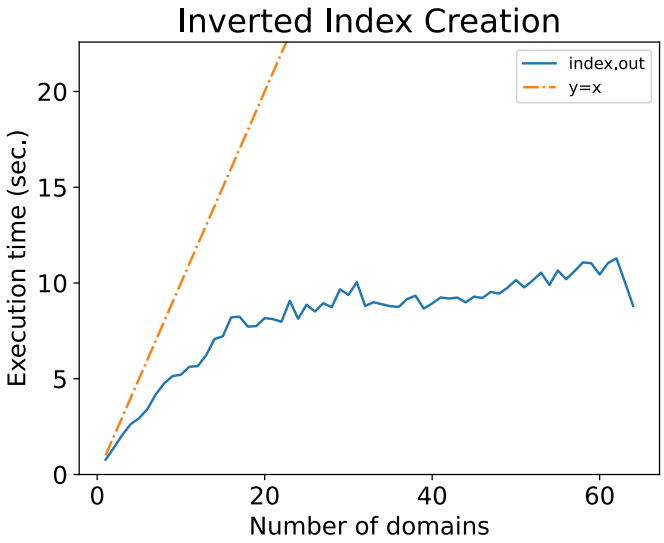

Figure 3: Speedup for inverted index creation

combat this, future work done in this area could choose to abandon the dependence on domainslib altogether, and instead work directly with the domains primitive provided by OCaml.

5.2. Array Creation

Another limitation of my work comes from its ultimately sequential approach to uninitialized-array creation (see Section 3.1). While many reasonable applications of the module will likely have array creation take a negligible fraction of the runtime, it is entirely plausible that applications could be working with large enough sequences and large enough amounts of parallelism such that the predominant influence on runtime is the linear hit we take when creating an empty block and setting every value to a safe value. As it currently stands, it seems that the challenges with parallel array initialization are rooted in a lack of support by the OCaml runtime. In fact, even domainslib depends on a call to `Array.copy` in its `parallel_scan` function [4].

5.3. Using Mutation

As one of the primary motivations for my work was to create materials for an academic functional programming course, the module I've written is intentionally a fully functional interface. However, there are some notable areas where allowing for mutation can lead to significant performance improvements. Most prominently, supporting mutating versions of certain functions could allow for intermediate computations to be performed in place, which would help reduce the performance costs of array creation discussed in Section 5.2 by simply reducing the number of arrays created.

The potential of mutation for performance improvement is extremely apparent when examining the domainslib `parallel_scan` function. In `parallel_scan`, there is a call to `Array.copy` which serves no purpose other than to allow the scanning to be performed in a separate array, keeping everything functional. However, when the function being scanned is simple (for example integer addition), this call actually takes up an extremely large proportion of the execution time. To illustrate this, I've defined an alternate scan function `in_place_scan` which is essentially just `parallel_scan` with the `Array.copy` removed (so that the scan happens in-place on the input array). Figure 4 shows the execution time required for both functions to perform an integer addition scan on an array of 1 billion integers, as well as the execution time required for a single call to `Array.copy` on an array of 1 billion integers.

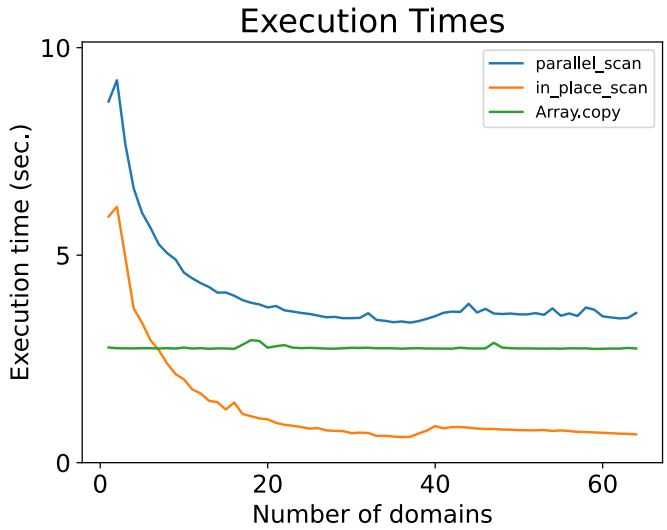

Figure 4: Execution times of `parallel_scan`, `in_place_scan`, and `Array.copy` on an array of 1 billion integers.

As is shown by Figure 4, simply allowing `parallel_scan` to mutate the array would be a significant performance improvement over what domainslib currently offers. While I completely understand the merit of having a functional `parallel_scan`, it's extremely unfortunate that the only way for a client of domainslib to avoid waiting on an expensive sequential function is to implement an in-place scan by themselves, like what I have done with `in_place_scan`. To address this, I propose that domainslib could have something similar to

```
(* Performs an in-place scan which mutates the input array *)
val parallel_scan_in_place : pool -> ('a -> 'a -> 'a) -> 'a array -> unit
```

This would give users the choice as to whether or not keeping things functional is worth the cost of the expensive `Array.copy` call. Moreover, replicating the functional behavior of the current `parallel_scan` would be extremely simple:

```
let parallel_scan pool op elements =
  let copy = Array.copy elements in
  parallel_scan_in_place pool op copy;
  copy
```

6. Conclusion

The parallel sequences abstraction allows for an elegant high-level approach to parallel programming, and allows us to gain notable performance increases by parallelizing highly complex tasks without having to notably increase the difficulty required for implementation. Despite the limitations the module faces, it still has quite favorable performance on the workloads we examined in Section 4, and provides a good starting point for the development and discussion of high-level shared-memory parallelism constructs in Multicore OCaml. Additionally, since my module provides an implementation of every method described in the COS 326 parallel sequence abstraction, my work is immediately applicable for use as a pedagogical tool in COS 326.

Acknowledgements

I would like to thank Professor Andrew Appel for the invaluable support he has provided me. Many of the techniques, insights, and approaches I used in my work would not have been possible without the guidance and advice he has given me over the past several months.

References

[1] G. E. Blelloch, "NESL: A nested data-parallel language (version 3.1)."

[2] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," vol. 27, no. 4, pp. 831–838, place: New York, NY, USA Publisher: Association for Computing Machinery. https://doi.org/10.1145/322217.322232

[3] X. Leroy *et al.*, "The OCaml system release 5.0: Documentation and user's manual," pp. 1–989. https://hal.inria.fr/hal-00930213

[4] Multicore OCaml. Domainslib 0.5.0. https://github.com/ocaml-multicore/domainslib/releases/tag/v0.5.0

[5] OCaml. OCaml source. https://github.com/ocaml/ocaml/tree/61f10168da60e94a5f9c2d1ce4cc4e4d512d0007

[6] K. C. Sivaramakrishnan *et al.*, "Retrofitting parallelism onto OCaml," vol. 4, pp. 1–30, publisher: Association for Computing Machinery (ACM). https://doi.org/10.1145%2F3408995

[7] D. Walker. Assignment 7: Parallel sequences. https://www.cs.princeton.edu/courses/archive/fall19/cos326/ass/a7.php

[8] D. Walker. Parallel sequences. https://www.cs.princeton.edu/courses/archive/fall22/cos326/notes/parallel-sequences.php