Parallel Sequences in Multicore OCaml

Andrew Tao
Adviser: Andrew Appel

May 1, 2023

Abstract

~~In this paper,~~ 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. This abstraction allows ~~for~~ clients to create highly parallelizable programs without needing to write, or even understand, the complex, low-level implementation details necessary to parallelize large tasks. Benchmarks performed against a fully sequential implementation of the parallel sequences abstraction indicate respectable speedup on reasonable applications.

1. Introduction

As the complexity of problems we want to solve using computers continues to skyrocket with time, leveraging the increased computational power offered by parallel processing becomes more and more crucial to performance on any reasonably sized task. However, parallelizing large tasks is often quite challenging, and there are many cases where the tasks themselves are conceptually simple, but are only complex due to the sheer size and scalability we wish to achieve using parallelism. With this in mind, it is often in our best interest to implement our tasks at a high level of abstraction, allowing us to separate the high-level description of complex tasks from the low-level details of how to actually parallelize individual operations.

While such parallel abstractions are already a well established field of study, OCaml currently has very little support for high-level parallel constructs, in general. This is because up until very recently, OCaml was one of the exceptionally few established high-level programming languages that lacked support for shared memory parallelism. However, after years of effort from the Multicore

OCaml team to retrofit parallelism into OCaml [11], the release of OCaml 5.0.0 finally introduced an entirely rewritten runtime system that offered support for shared memory parallelism [7]. With this newly supported capability in mind, ~~the goal of this work has been to implement~~ an abstraction for data parallelism using parallel sequences in Multicore OCaml.

The abstraction I describe in my work was heavily inspired by existing parallel collections abstraction found in various forms across many programming languages [2, 10, 14]. The full set of supported operations in my implementation of this abstraction, as well as each operation's associated work and span (depth), can be seen in Table 1. This set of operations offers a very simple interface

Operation	Description	Work	Span
`iter f s`	Apply `f` to every element of `s` in order	n	n
`iteri f s`	Apply `f` to every index and value of `s` in order	n	n
`length s`	Return the length of sequence `s`	1	1
`empty ()`	Create an empty sequence	1	1
`singleton x`	Create a sequence containing only the value `x`	1	1
`nth s i`	Get the element at index `i` of `s`	1	1
`cons x s`	Add `x` to the beginning of `s`	n	1
`tabulate f n`	Create a sequence with $\forall i \in [0,n)$, `nth s i = f i`	n	1
`repeat x n`	Create a sequence by repeating `x` `n` times	n	1
`append s1 s2`	Create a sequence that appends `s2` to `s1`	n	1
`seq_of_array a`	Create a sequence containing every element of `a`	n	1
`array_of_seq s`	Create an array containing every element of `s`	n	1
`zip s1 s2`	Zip two sequences of type `'a seq` and `'b seq` into one sequence of type `('a * 'b) seq`	n	1
`split s`	Split a sequence of type `('a * 'b) seq` into two sequences of type `'a seq` and `'b seq`	n	1
`map f s`	Apply `f` to every element of `s`	n	1
`reduce g b s`	Combine elts of `s` using function `g` and base value `b`	n	$\log n$
`map_reduce f g b s`	Map values of `s` with `f`, then reduce with `g` and `b`	n	$\log n$
`scan g b s`	Return scan of function `f` with base value `b`	n	$\log n^*$
`flatten ss`	Flatten nested sequence `ss`	n	$\log n$
`filter p s`	Remove elts of `s` which do not satisfy predicate `p`	n	$\log n$

Table 1: Supported operations in the Parallel Sequence abstraction. n is the size of the sequence given as input or produced as output. In `flatten`, n^* is the length of the nested sequence.

that allows for complex tasks to be represented by a high-level description of each intermediate step performed. For example, consider the (somewhat contrived) example which calculates the sum of the first n terms of the harmonic series in parallel:

```
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.

2. Problem Background and Related Work

Much of my work has been heavily influenced by Blelloch's work on the NESL programming language [2]. 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. In fact, even the choice to analyze work and span for each operation in my implementation has been inspired by the Blelloch's work in analyzing the work and depth of each of NESL's parallel functions.

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 [14]. This abstraction has been taught since long before the addition of multicore support to OCaml in version 5.0.0, so none of the currently existing implementations of this abstraction are able to the 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. As a result, I've made sure that every function described in the COS 326 parallel sequences abstraction has been included in my interface.

3. Approach

In OCaml, the standard library only offers low-level primitives for parallel programming [7], and as a result, it is recommended that users utilise higher-level parallel programming libraries such as domainslib [12]. The domainslib library is implemented using a work-stealing queue [7], and provides several useful parallel programming operations, including `async` and `await` on promises and a `parallel_for`. These operations are especially useful in my implementation, as they allow me to divert my attention from the details of task scheduling, and instead focus on the high-level decisions regarding algorithmic design.

4. Implementation

Since basic parallelism constructs have been handled, we can now move on to underlying representation and implementation details. 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 (as we then do not need to know the eventual values of every element before creating the sequence). ~~This being said,~~ 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.

4.1. Basic Array Operations

As a result of the underlying representation of the sequence being an array, many basic operations can be implemented as calls to an existing array method in the standard library. For example, see the following implementation of length:

```
let length : 'a t -> int = Array.length
```

Other operations implemented this way include `iter`, `iteri`, `empty`, `singleton`, and `nth`. The `iter` and `iteri` operations are primarily provided for ease of debugging (e.g. to be used as a way to

print out the contents of a sequence), and so are not intended to be efficient: these two operations take linear work and span with respect to the size of the sequence. However, the remaining operations all take constant work and span.

4.2. 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 [7].[1] 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. For less obvious reasons, the `make` operation is also difficult to work with due to the strict type system in OCaml. Consider the `tabulate` function:

```
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 the array `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. There are a couple ways within OCaml's type system that may make it possible to circumvent this challenge, but all of these options are unideal for both performance and elegance reasons. We will actually see that things become much easier when we decide to subvert the strict type system that OCaml imposes on us.

4.2.1. Typing

There are two obvious ways for us to gain more control over the type system in OCaml:

- Use the provided module `Obj`, which allows us to work directly with the internal representations of values. Notably, the function `magic` allows us to change the type of a value.

[1] Technically, the module also provides a method for creating an uninitialized, unboxed-float array. However, this resulting array is both not polymorphic and also not garbage collected. For reasons we will soon discuss, this method will give us the exact same problems as the standard set.

- Write a function in C and call it as a foreign stub in our OCaml code. We get to make the return type of the C stub whatever we want.

We will eventually end up using the latter for performance reasons, but it will be much easier to understand how everything works by first reasoning how we would do things with just the OCaml `Obj` module.

Intuitively, we know that all of our functions that need to create a new array will eventually set every element of the array. With this in mind, if we could have some type of function that quickly gives us a large block of uninitialized data, it seems reasonable that we would then be able to "cast" it to the correct type. This seems relatively straightforward, and in fact, OCaml's standard library `Array` module provides a method which gives us an uninitialized, unboxed float array. Unfortunately, this naive solution does not immediately work due to an interesting quirk of how the OCaml garbage collector handles values during runtime.

4.2.2. Garbage Collection

~~While this may be clear, I establish that~~ any sequence we ~~end up~~ creat~~ing~~ will need to be registered with the garbage collector, and so must be allocated through the OCaml heap allocation system as a garbage collected block. ~~After all, our sequences will be the only reference to the values we compute during the intermediate steps of sequence creation, and it would be extremely unfortunate if any of the values were thrown out by the garbage collector.~~ Additionally, ~~at the time that this paper is being written,~~ it is not possible to "register" a previously initialized value with the OCaml garbage collector (See Section 6 for more details).

Next, I will discuss the constraints that using garbage collected blocks brings. Because the OCaml compilation process for each source file involves a type checking pass over the generated abstract syntax tree, OCaml is free to discard much of the information about type during runtime [8]. Then, in order to distinguish boxed types (e.g. pointers) ~~with~~ unboxed types (e.g. integers) at runtime, OCaml does a trick where all words corresponding to pointers have a 0 as the least significant digit, while all words corresponding to integers have a 1 as the least significant digit [8].

For us, this means that if were to ever have some uninitialized value that happened to end with a

0 as its least significant bit, OCaml's garbage collector would treat it as a valid pointer, and attempt to dereference it. For example, consider the following code which causes a segmentation fault:

```
let arr = Array.make 1 1
let null_ptr = Obj.field (Obj.repr 0.0) 0
let _ = Obj.set_field (Obj.repr arr) 0 null_ptr
let _ = Gc.full_major ()
```

Because of this, even if we were able to create some sort of large uninitialized block, we would need every word in the block to have a 1 as its least significant bit, so that the garbage collector always knows that these uninitialized values can be ignored.

Then, suppose that we have some way of generating some uninitialized block that can guarantee every word ends as a 1 in its least significant bit. We could implement getting an uninitialized array of any type as follows:

```
let get_uninitialized_array (n : int) : 'a array =
  let chunk = get_large_uninitialized_chunk n in
  let arr : 'a array = Obj.magic chunk in
  arr
```

The astute reader may notice that it seems plausible that eventually storing a pointer into a field that used to hold an integer value could cause problems if the garbage collector was keeping track of which fields were previously integers and then assuming these fields would be always safe to ignore. Fortunately, this is not the case, and although reading the array and garbage collection source code for the OCaml runtime makes this abundantly clear [9], Appendix A provides a simple test that also demonstrates this fact.

4.2.3. Workaround

To address the issues we've discussed, my implementation relies on an `Array_handler` module, which uses an external C stub to implement the function `get_uninitialized` which returns a polymorphic uninitialized array. The C stub is a modified `Array.make` [9] that sets every field of a

newly allocated block to a "safe" nonpointer value 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. Additionally, if we had chosen to use `Array.make` to create our array, we would also have to waste a bit of time calling `Obj.magic` to appease the OCaml type checker. For the actual C stub code, see Appendix B.

We will then trust that our array initialization is an efficient operation, and make the approximation that for all intents and purposes, one call to our C stub will have linear work and constant span. I concede that this approach of treating a fast but ultimately sequential subroutine as having constant span is certainly not ideal, but I believe this concession is acceptable for the following reasons:

1. The array initialization is extremely fast. In most realistic applications of the abstraction (see Section 5), 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.
3. ~~This paper would be quite boring if I had to say that every operation involving sequence creation had linear span.~~

4.3. Tabulate and Friends

Once array initialization has been properly handled, the rest of 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 "ParallelSequence.tabulate")
  else
    let arr : 'a array = Array_handler.get_uninitialized n in
```

```
    parallel_for n (fun i -> arr.(i) <- f i);
    arr
```

We see that this is essentially just creating an empty array and then populating the empty array with the results of our function applications in parallel. The `parallel_for` is the only interesting computation performed, and as a result the `tabulate` has linear work and constant span with respect to the size of the resulting sequence (the parameter n).

Once tabulate is implemented, many functions can be implemented very easily. For example, see 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 `seq_of_array`, `array_of_seq`, `cons`, `repeat`, `zip`, and `split`.

Since the body method in each of these take constant time, each of the listed operations all take linear work and constant span with respect to the size of the sequence(s) provided as input or created as output.

4.4. Combines

In this section, we examine the implementations of `reduce`, which sums all of the elements of a sequence, and `scan`, which computes an inclusive prefix sum for each index of a sequence. Since sequences can be of any type, both of these operations take in a client-provided function which it uses to combine two elements together, which we will refer to as a "combining" function. The parallel sum and parallel prefix sum have both already been studied in depth in the past, and there are already well established, highly parallelizable algorithms that achieve logarithmic span on both problems [5, 6]. However, these established algorithms are often tailored to contexts that pose desirable, exploitable qualities which simply do not exist in many reasonable applications of our module.

4.4.1. Unique Concerns

First, ~~note that~~ many of the existing parallel sum and parallel prefix sum algorithms are intended for use in contexts where parallelism can be treated as essentially limitless [5, 6]. Then, for these algorithms, it can sometimes be worth it to increase the asymptotic complexity of work being done if it decreases the span, even if this decrease is by a constant factor. However, we do not have this privilege; in many typical uses of our module, the number of domains will be many orders of magnitude smaller than the number of elements in our sequence, meaning that work will play such a critical role in performance that it simply cannot be sacrificed in the name of decreasing span.

Next, ~~note that~~ existing parallel sum and parallel prefix sums are virtually always implemented for contexts where the combining operation being performed is some primitive operation where each call requires constant work regardless of the input (e.g. sum, multiply). This means that when two partial sums are being combined during an intermediate step of these algorithms, the sizes of the intervals that the partial sums correspond to can be ignored. However, since the combining functions for our module are client-provided, it is entirely plausible that the amount of work required to combine two partial sums is dependent on the size of the interval each partial sum represents. Then, the order of application actually has a significant influence on the running time of our `reduce`

or `scan`. For example, consider the following contrived example:

```
let get_array (n : int) : int array =
  let singletons = tabulate (fun x -> [|x|]) n in
  reduce (Array.append) [||] singletons
```

In this case, since the `Array.append` operation takes time proportional to the size of both arrays, the order in which we combine partial sums is crucial. While a divide-and-conquer (binary tree) approach would only take linearithmic work, a purely sequential sequential approach would actually take quadratic time. With this in mind, we would desire that our implementations of `reduce` and `span` attempt to order combines so that we achieve good performance even in cases where the client-provided combining functions have nonconstant asymptotic runtime complexity.

4.4.2. K Indexed Trees

In my implementations of both `reduce` and `span`, I used a method inspired by binary indexed trees, which are also referred to as Fenwick trees [4]. A binary indexed tree is a data structure that supports querying inclusive prefix sums on a dynamically updating array. Both processing updates to the corresponding array as well as querying the prefix sum up to any point in the array takes at most logarithmic work. For the purposes of my implementation, we will only be interested in the prefix sum query, and will not be updating the tree at any point once it has been built.

In my implementation, the binary indexed tree has been generalized to be k indexed instead of binary indexed, where k is an integer greater than or equal to 2 that can be thought of as a sequential cutoff. For the purposes of work and span analysis, we will treat this as a constant, but in practice, I dynamically determine the value of k for each `reduce` or `span` call based on the ratio between the size of the sequence and the number of processors. Additionally, for ease of indexing arithmetic, we will treat all arrays as one-indexed. To accommodate for the fact that arrays are actually zero-indexed in OCaml, we can simply subtract one from every index before getting it when actually writing code. Lastly, while binary indexed trees were initially proposed for use with integer addition, my implementation will be able to work for any associative combining function

f and base value b. The base value b must satisfy that for all x, $f(x,\ b) = x$ and $f(b,\ x) = x$. For example, when f is integer addition, b is just 0.

The k indexed tree derives it power primarily from an important indexing trick, which we'll refer to as the strip_k operation. The strip_k operation considers a positive integer in its base k representation, and strips off its least significant non-zero digit. For example, the strip_2 operation on the index $13 = 1101_2$ removes the digit at index 0 and returns $12 = 1100_2$.

Next, I will define a few things for ease of notation. When notating sums, I will stick to writing cases where integer arithmetic is used, but note that any sum of the form $\sum_{i=p}^{q} s[i]$ can be generalized to the expression

$$f(\ldots f(f(b,\ s[p]),\ s[p+1])\ldots,\ s[q])$$

and any sum of the form $x+y$ can be generalized to the expression $f(x,\ y)$. Additionally, I define ~~that~~

$$s[p\ldots q] := \sum_{i=p}^{q} s[i]$$

Now that definitions are out of the way, we can begin reasoning about how information is stored within our k indexed tree. For any index i, we will have that $t[i] = s[\text{strip}_k(i)+1\ldots i]$. If this

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
t	(1, 1)	(1, 2)	(3, 3)	(1, 4)	(5, 5)	(5, 6)	(7, 7)	(1, 8)	(9, 9)	(9, 10)	(11, 11)	(9, 12)	(13, 13)	(13, 14)	(15, 15)

Figure 1: A k indexed tree for $k = 2$ and a sequence of size 15, chosen to demonstrate the cases where the length is not a power of k. Value (p,q) at index i represents that $t[i] = s[p\ldots q]$.

invariant holds, then we can get the inclusive prefix sum up to some index i using the following procedure, shown in pseudocode below, and displayed visually in Figure 2:

```
def get(i, t, k, f, b):
  total = b, idx = i
  while idx > 0:
    total = f(t[idx], total)
```

```
    idx = strip(idx, k)
  return total
```

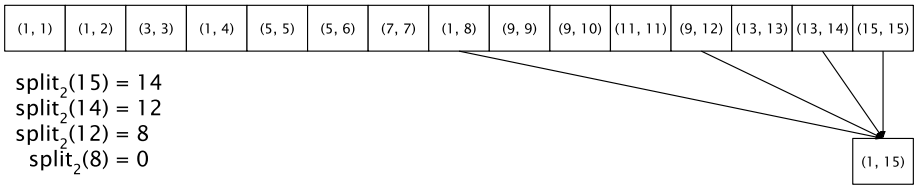

Figure 2: Getting index 15 from a k indexed tree with $k = 2$.

Here, the parameter i is the index we're interested in getting the prefix sum up to, the parameters t and k represent the k indexed tree t, f is the combining function used to sum elements of the tree, and b is the base value for the function f. The outline for the proof of correctness is as follows:

- If we apply strip_k repeatedly to i until the index returned is 0, we will get some sequence of indices $\{i, \ldots, j, 0\}$, where $j = k^x$ for some integer x
- The get method calculates the value $t[j] + \cdots + t[\text{strip}_k(i)] + t[i]$
- By construction of t, we know that
  - $t[j] = s[1 \ldots j]$

    $\ldots$
  - $t[\text{strip}_k(i)] = s[\text{strip}_k(\text{strip}_k(i)) + 1 \ldots \text{strip}_k(i)]$
  - $t[i] = s[\text{strip}_k(i) + 1 \ldots i]$
- Then, $t[j] + \cdots + t[\text{strip}_k(i)] + t[i]$

  $= s[1 \ldots j] + \cdots + s[\text{strip}_k(\text{strip}_k(i)) + 1 \ldots \text{strip}_k(i)] + s[\text{strip}_k(i) + 1 \ldots i]$

  $= s[1 \ldots i]$

■

~~Additionally, note that~~ the number of times the strip_k operation can be applied for some starting index i is equal to the number of non-zero digits in the base k representation of i. Since this number is upper-bounded by $\lceil \log_k i \rceil + 1$, we see that getting a prefix sum from a k indexed tree takes logarithmic work and span.

4.4.3. Parallel K Indexed Tree Build and Collapse

Now that we have defined the k indexed tree data structure we will use, we now need to understand how to create such a tree from an existing sequence in parallel. The way this is accomplished is shown in pseudocode below:

```
def ceil_div(num, den):
  return (num + den - 1) / den


def build(f, b, k, s):
  t = s.copy()
  subtree_size = 1
  while subtree_size < s.length:
    num_groups = ceil_div(s.length, subtree_size)
    parallel_for group_num in [0, num_groups):
      offset = group_num * subtree_size * k
      acc = b
      last_idx = min(k, (len(s) - offset) / subtree_size)
      for j in [1, last_idx + 1):
        idx = offset + (j * subtree_size)
        acc = f(acc, t[idx])
        t[idx] = acc
    subtree_size = subtree_size * k
  return t
```

The general intuition for the code above is that we build up the k-indexed tree layer by layer, and all of the groupings done within one layer are independent and so can be done in parallel. There are three loops of interest. The inner `for` simply merges a group of subtrees together into one larger subtree. While there are typically k subtrees per group, extra care is needed in calculating bounds

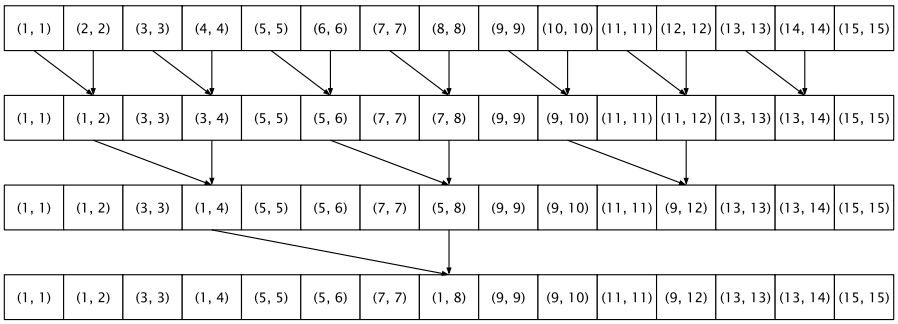

Figure 3: Building a k indexed tree with $k = 2$.

for the loop, since there will often be fewer than k subtrees in right-most group of the tree. Next, the `parallel_for` simply does all of the subtree groupings in parallel, since they can all be computed independently. Finally, the outer `while` loop simply repeats the grouping with increasing subtree sizes, until the subtree size is large enough such that no more grouping is possible.

The complexity of this construction is quite straightforward to analyze. Since we treat k as a constant for the sake of analysis, we have that the inner `for` is a constant cost operation, and the `parallel_for` will then have constant span, so that building each layer of the resulting k indexed tree takes constant span. Finally, we see that the number of layers we need to build up will simply be the number of times we can multiply the size of the subtrees by k before the subtree size is larger the length of the sequence, which is clearly logarithmic with respect to the size of the sequence. Since we know each layer is a constant span operation, we see that building the entire tree can be done in logarithmic span. Let us next analyze the work. We see that for each layer, each "node" of the layer is involved in exactly one application of f, which occurs within the inner-most `for` loop. Next, the size of each layer is simply the length of the sequence divided by the size of the subtree. Then, the work done is represented sum below:

$$\sum_{i=0}^{\lfloor \log_k n \rfloor} \left\lfloor \frac{n}{k^i} \right\rfloor \leq \sum_{i=0}^{\infty} \frac{n}{k^i} = n \cdot \frac{k}{k-1}$$

~~Then, we know that~~ the work for this `build` operation is linear with respect to the size of the sequence.

~~At this point,~~ now that we have built a k indexed tree, we have enough to do both reduce and scan

in logarithmic span. After all, we have defined a `get` method to get prefix sums from a k indexed tree which we we know takes both logarithmic work and span. We can implement reduce as a `get` call on the last index of the sequence:

```
def reduce(f, b, s):
  k = max(2, s.length / num_processors)
  t = build(f, b, k, s)
  return get(s.length, t, k, f, b)
```

Since the `get` operation takes at most logarithmic work and span, the asymptotic complexity is dominated by the tree building step, so the `reduce` operation takes linear work and logarithmic span.

We could also (naively) implement scan by simply calling `get` in parallel:

```
def naive_scan(f, b, s):
  k = max(2, s.length / num_processors)
  t = build(f, b, k, s)
  return tabulate(fun i -> get(i + 1, t, k, f, b), s.length)
```

While this naive implementation would give us logarithmic span, the work required is $\Theta(n \log_k n)$, where n is the length of the sequence (See Appendix C for proof).

As it turns out, we can actually get our work for `scan` down to linear by employing a clever way of collapsing down our already built k indexed tree. The key insight is that when we do a series of `get` calls in parallel, we often recompute the same values. For example, consider a case where $k = 2$, and we want to find the prefix sums for the indices 14 and 15. When we do repeatedly call `strip` on $15 = 1111_2$, we get the sequence of indices $\{15, 14, 12, 8\}$, and compute

$$t[8]+t[12]+t[14]+t[15] = s[1\ldots8]+s[9\ldots12]+s[13\ldots14]+s[15] = s[1\ldots14]+s[15] = s[1\ldots15]$$

If we weren't doing everything in parallel at the same time, we could've saved time by only calculating $s[1\ldots14]$ once.

By collapsing the tree top-down after it has been built, we can prevent the redundant calculation shown in the previous example. The implementation is shown in pseudocode below, and displayed visually in Figure 4:

```
def collapse(f, k, t):
  group_size = 1
  while (group_size * k < t.length):
    group_size *= k
  while (group_size > 1):
    num_groups = ceil_div(t.length, group_size)
    subtree_size = group_size / k
    parallel_for group_num in range(1, num_groups):
      prev_group_idx = group_num * group_size
      last = min(k - 1, (t.length - prev_group_idx) / subtree_size)
      for i in range (1, last + 1):
        idx = prev_group_idx + (i * subtree_size)
        t[idx] = f(t[prev_group_idx], t[idx])
    group_size = group_size / k
  return t
```

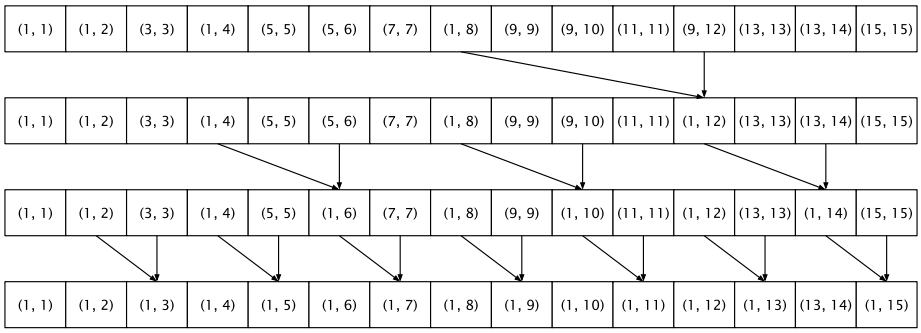

Figure 4: Collapsing a k indexed tree with $k = 2$.

The general intuition for the procedure above is that we collapse the k indexed tree top-down one

layer at a time. When I refer to a layer l, I am referring to all of the indices which are divisible by k^l. We begin the collapse at layer $\lceil \log_k n \rceil$, and when we collapse some layer l, we will make layer $l-1$ have the property that for each index i in this layer, $t[i] = s[1 \dots i]$ (note that layer $\lceil \log_k n \rceil$ begins with this property). For example, for the $k = 2$ example shown in Figure 4, we start with a fully built tree, and at this point, and indices i divisible by 8, the value is the full prefix sum $s[1 \dots i]$.[2] Then, we collapse this layer, so that now, every index divisible by 4 contains its full prefix sum. We repeat this process again twice until every index (divisible by 1) contains its full prefix sum, and we are finished.

The first `while` loop of the procedure simply calculates the size of the largest subtree contained in the tree to figure out what layer to start at.[3] The second outer `while` loop repeatedly collapses each layer of the tree, the inner-most `for` loop handles each subtree in one group, and the `parallel_for` is used to collapse each group in parallel.

The work and span analysis for the `collapse` operation is essentially identical to that of the `build` operation, and so the `collapse` operation is ~~linear and the span is logarithmic~~. Then, by tying everything together, we can finally implement the `scan` operation as

```
def scan(f, b, s):
  k = max(2, s.length / num_processors)
  t = build(f, b, k, s)
  return collapse(f, k, t)
```

Since we know that the `build` and `collapse` steps both take linear work and logarithmic span, our `scan` operation takes linear work and logarithmic span with respect to the length of the sequence.

Finally, let us note that our approach to `reduce` and `scan` outlined in this section addresses the two concerns listed in Section 4.4.1:

[2] We see that in this case, we actually can start with layer 3, not layer $\lceil \log_k 15 = 4$. This occasionally occurs because the first subtree of every layer is a power of k, so if there is only one subtree in layer $l-1$, we can start there instead. In practice, the code just does a single no-op for layer l before continuing.

[3] This value is actually already known to the `build` method when it finishes build the k indexed tree. In the actual implementation, the `build` method returns both the k indexed tree and the size of the largest subtree to skip recomputing this value.

1. Each `reduce` and `scan` operation calls the provided combining function an amount of times that only grows linearly with respect to the size of the sequence
2. The tree organization means that when two partial sums are merged, the two corresponding intervals are typically of similar size.

4.5. Miscellaneous Operations

In this section, we will cover the remaining operations, which includes `map_reduce`, `flatten`, and `filter`. We will not discuss `map_reduce` in detail, as my implementation just does a `map` followed by a `reduce`.

Let us examine the implementation of `flatten`:

```
let flatten (ss : 'a t t) : 'a t =
  let lens = map (fun s -> length s) ss in
  let total_len = reduce ( + ) 0 lens in
  let starts = scan ( + ) 0 lens in
  let arr : 'a array = Array_handler.get_uninitialized total_len in
  parallel_for (length ss) (fun i ->
      let start = starts.(i) in
      let s = ss.(i) in
      parallel_for (length s) (fun j -> arr.(start + j) <- s.(j)));
  arr
```

The general outline is that we get the starting index for each inner sequence by doing a `scan` over the lengths of the inner sequences, and then we copy over the elements of each sequence in parallel. The work in this operation is dominated by the nested `parallel_for` calls, which take linear work with respect to the number of elements between all inner sequences. However, this nested call actually only takes constant span, and the bottle-neck for span is actually the `scan` performed on the lengths of the inner sequences. Then, the span of this operation is logarithmic with respect to the number of inner sequences in the outer sequence.

Finally, us examine the implementation of `filter`:

```
let filter (pred : 'a -> bool) (s : 'a t) : 'a t =
  let len = length s in
  if len = 0 then empty ()
  else
    let bit_vec = map (fun v -> if pred v then 1 else 0) s in
    let prefixes = scan ( + ) 0 bit_vec in
    let filtered_length = nth prefixes (len - 1) in
    let filtered : 'a array =
      Array_handler.get_uninitialized filtered_length
    in
    let body idx =
      let l = if idx = 0 then 0 else nth prefixes (idx - 1) in
      if nth prefixes idx > l then filtered.(l) <- nth s idx else ()
    in
    parallel_for len body;
    filtered
```

After the check for empty corner case, we begin by mapping each element to a 1 or 0 based on if the element satisfies the predicate, and then perform a prefix sum scan over the bit vector. The sequence resulting from the scan gives us two useful pieces of information for each index. First, we can recover the original value of the predicate bit-vector for each index by simply comparing the value with the value to its immediate left; if the values differ, than the element of the original sequence at this index must have satisfied the predicate. Additionally, we now also know the number of predicate-satisfying elements to the left of any index, so we can simply use this as the index our value belongs to in the resulting filtered array. In our implementation, the calls `map`, `scan`, and `parallel_for` all take linear work with respect to the size of the sequence, so the `filter` operation also takes linear work with respect to the size of the sequence. The bottle-neck for span

comes from the prefix sum scan, and so we know that the span for the operation is logarithmic with respect to the length of the sequence.

5. Evaluation

In this section, we will examine two basic applications of the parallel sequences module, both of which will implemented in a context where parallelism is achieved exclusively by performing operations on parallel sequences. In both applications, we will analyze the speedup we receive compared to a run that uses a purely sequential implementation of the module (see Appendix D). This purely sequential implementation is heavily based ~~off~~ of the array implementation of the COS 326 parallel sequences abstraction used in one of the course assignments [13]. 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.

5.1. Block Matrix Multiplication

The first benchmark we will examine is a block matrix multiplication on two float matrices. Each float element was generated uniformly randomly from $[0, 1]$, both float matrices had size 2000×2000, and the size of each block used was 100×100. The speedup for this benchmark is shown in Figure 5.

5.2. Web Indexing

The other benchmark I used was indexing webpages into an inverted index. This is a particularly interesting application of the parallel sequences abstraction, as this form of web indexing was among some of the earliest computational tasks that demonstrated the utility of data parallelism. Moreover, web indexing is very relevant to this work, as the COS 326 course uses web indexing as one of the only concrete use cases for its abstraction [1].

I performed the web indexing in two phases. In the first phase, I parse every webpage in parallel using `tabulate`, creating many balanced binary search tree dictionaries. Each of these dictionaries map words to a set of locations within documents. Then, in the second phase, I reduce every binary

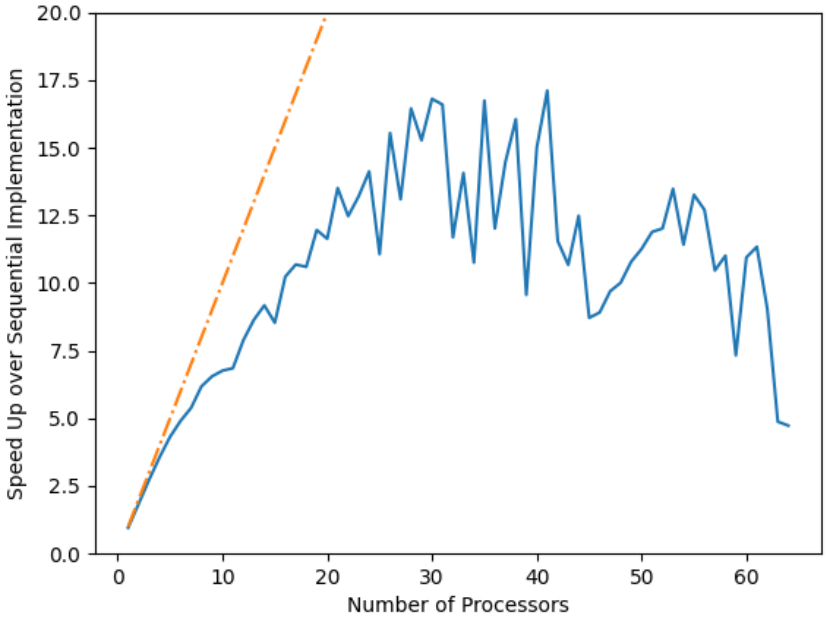

Figure 5: Speedup for block matrix multiplication

search tree together, to form one unified inverted index. This two-phase approach is directly inspired by the MapReduce programming model [3], and is also outlined in the COS 326 course [1].

In this work, I performed the benchmark on collection of one million pages, which was about 800MB. These pages were generated from a file containing 924 pages used in the data parallelism assignment of COS 326 [13]. The speedup for this benchmark is shown in Figure 6.

6. Limitations and Future Work

The biggest limitation of my work is its lack of consideration for memory locality, resulting from its dependence on the work-stealing queue approach offered by domainslib. Consider the function `map_reduce`, which is a `map` followed by a `reduce`. In order to maximize locality, we would hope that the processor in charge of mapping certain elements of the sequence during the `map` phase would be the processor assigned to work with those mapped values in the `reduce` phase. However, since the domainslib library only supplies the ability to asynchronously schedule tasks, and not the ability to influence which domain actually ends up performing the task, we have no way of expressing or accounting for this desire. Because of this, we can lose quite a bit of time simply

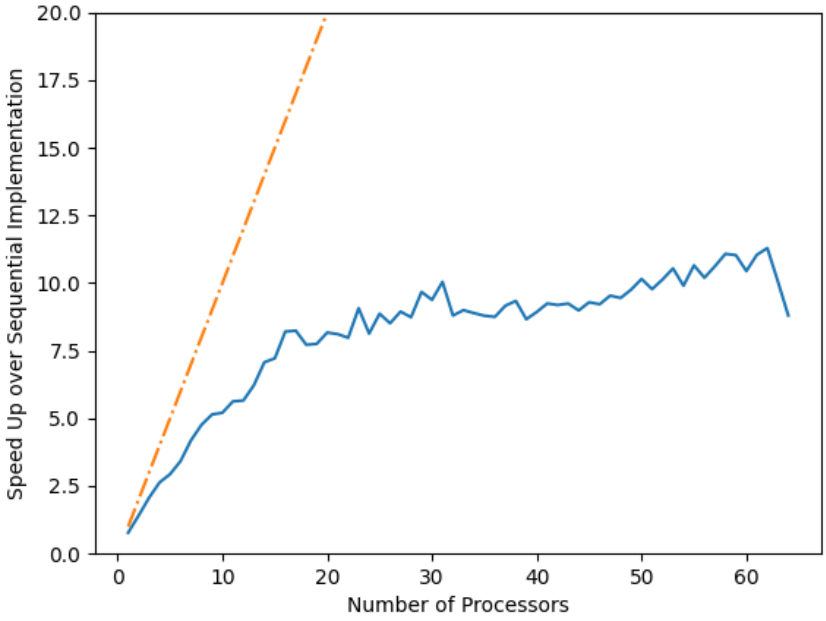

Figure 6: Speedup for inverted index creation

due to cache misses, and this unfortunate performance hit is likely one of biggest factors for the limited performance increases seen in Section 5. To combat this, future work done in this area could likely abandon the dependence on domainslib altogether, and and instead opt to work directly with the domains primitive provided by OCaml. Although this would make implementation much more difficult, manually enforcing task scheduling would give the module a much greater capability to maximize locality.

Another limitation of my work comes from the inherently sequential approach to safe uninitialized-array creation used. While I did suggest that many reasonable applications of the module will likely have array creation take a negligible fraction of the runtime, it is entirely plausible that certain applications could be working with large enough sequences and large enough amounts of parallelism such that the predominant influence on runtime is the linear span hit we take by setting every value to a nonpointer value. As a result, we would still desire to have an array initialization step that takes truly constant span. As it currently stands, it seems that the challenges with parallelizible array initialization are rooted in a lack of support by the OCaml runtime. In fact, even the recommended parallel library, domainslib, depends on a call to `Array.copy` in its `parallel_scan` function.

Future work that deals with this limitation will likely have to deal with the main source of the constraint: the garbage collector. In the current runtime, allocated blocks each have a tag byte in the header that contains meta information, and certain tag bytes tell the garbage collector not to inspect the contents of the block [8]. Because of this, solutions for parallel initialization of integer arrays do exist, as they can ignore our safety initialization step by simply tagging an uninitialized block as something the should be ignored by the garbage collector. In Section 4.2.2, I suggested that it is currently not possible to "register" a previously initialized block with the OCaml garbage collector - that is, we can only decide whether or not a block is garbage collected at allocation. ~~The reason I said~~ this was because OCaml 5.0.0 removed the operation `Obj.set_tag` due to conflicts with concurrent garbage collector threads non-atomically marking the object [11]. `Obj.set_tag` was an operation that allowed us to change the header tag of a block, and so in previous versions of OCaml, we could retroactively mark a block as scanned by the garbage collector. This may seem minor, but if we still had the ability register blocks retroactively, we would be able to initialize arrays with truly constant span. A parallel initialization of an array could proceed as follows:

1. Allocate a block with a tag that prevents it from being scanned by the garbage collector
2. Set every field of the block to a safe (integer) value in parallel
3. Change the tag of block to the array tag (so that it is scanned by the garbage collector) and change its type to an array of desired type
4. Set each array element in parallel

At first, it may seem like having two sets to each field of the block is redundant, and that we could replace step **2** with step **4**. However, if we put pointers into the array before it's marked as garbage collected, the garbage collector could try and reclaim some of the values before the array has been registered as referencing them.

7. Conclusion

The parallel sequences abstraction allows for an elegant description of high-level parallel programming, and allows us to gain notable performance increases for highly complex tasks without having

to notably increase the difficulty required for implementation. Despite the unfortunate limitations that the module does face, it still has quite favorable performance in the realistic workloads we aimed to emulate in Section 5, and provides a good starting point for the development of high-level shared memory parallelism constructs in Multicore OCaml. Moreover, 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, and the course will finally have access to a truly parallel OCaml implementation of its parallel sequences abstraction.

8. Acknowledgements

Still working on this.

References

[1] A. Appel and D. Walker, "Parallel prefix scan and assignment 7." Available: https://www.cs.princeton.edu/courses/archive/fall22/cos326/lec/23-parallel-scan.pdf

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

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters."

[4] P. M. Fenwick, "A new data structure for cumulative frequency tables," vol. 24, no. 3, pp. 327–336, eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380240306. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380240306

[5] W. D. Hillis and G. L. Steele, "Data parallel algorithms," vol. 29, no. 12, pp. 1170–1183, place: New York, NY, USA Publisher: Association for Computing Machinery. Available: https://doi.org/10.1145/7902.7903

[6] 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. Available: https://doi.org/10.1145/322217.322232

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

[8] A. Madhavapeddy and Y. Minsky, *Real World OCaml: Functional Programming for the Masses*. Cambridge University Press.

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

[10] S. Peyton Jones *et al.*, "Harnessing the multicores: Nested data parallelism in haskell," in *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

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

[12] M. O. Team. Domainslib 0.5.0. Available: https://github.com/ocaml-multicore/domainslib/releases/tag/v0.5.0

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

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

A. Replacing Integers with Pointers

In OCaml, type constructors that carry associated values are boxed. For example, with the type `int option`, the value `Some 1` is boxed. We use this in the test shown below:

```
(* Make a boxed int option and place it in the weak array and int array *)
let f weak_a int_array i =
  let int_opt_array : int option array = Obj.magic int_array in
  let boxed_val = Some i in
  Weak.set weak_a 0 (Some boxed_val);
  int_opt_array.(0) <- boxed_val

let weak_arr : int option Weak.t = Weak.create 1
let int_arr : int array = Array.make 1 (-1)
let _ = f weak_arr int_arr 326;;

(* The boxed int optional will not be garbage collected, since the int
 * array contains a reference to it. *)
Gc.full_major ();;
assert (Weak.check weak_arr 0);;

(* Overwrite the int array's reference to the int optional. Now, no
 * references to it exist, so it should be garbage collected. *)
int_arr.(0) <- 0;;
Gc.full_major ();;
assert (not (Weak.check weak_arr 0))
```

B. Uninitialized Array Creation

```
#include <string.h>
#include <caml/mlvalues.h>
#include <caml/alloc.h>
#include <caml/memory.h>
#include <caml/fail.h>

#define SAFE_BYTE -1

CAMLprim value caml_make_uninitialized_vect(value len)
{
  CAMLparam1(len);
  CAMLlocal1(res);
  mlsize_t size, i, num_bytes;

  size = Long_val(len);
  num_bytes = size * sizeof(value);
  if (size == 0)
  {
    res = Atom(0);
  }
  else
  {
    if (size <= Max_young_wosize)
    {
      res = caml_alloc_small(size, 0);
```

```
    }
    else if (size > Max_wosize)
      caml_invalid_argument("Array_handler.get_uninitialized");
    else
    {
      res = caml_alloc_shr(size, 0);
    }

    // Replaces:
    // for (i = 0; i < size; i++)
    //   Field(res, i) = init;
    memset((void *)res, SAFE_BYTE, num_bytes);
  }
  /* Give the GC a chance to run, and run memprof callbacks */
  caml_process_pending_actions();
  CAMLreturn(res);
}
```

C. Naive Scan Work

We wish to show that the work of the naive scan, shown below, is linearithmic.

```
def naive_scan(f, b, s):
  k = max(2, s.length / num_processors)
  t = build(f, b, k, s)
  return tabulate(fun i -> get(i + 1, t, k, f, b), s.length)
```

Let us denote the length of the array as n. We already know that the call to `build` will take linear work (and so will not dominate our work in the end). In our `tabulate` call, we will end up calling `get` for every index from 1 to n (our tree is treated as one-indexed). Let us now consider the work

required for one call to the `get` operation on some arbitrary index i. Since the `get` operation iterates by repeatedly calling strip_k on i until the value is zero, we see that the work required for the `get` call is simply equal to the number of nonzero digits in the base k representation of the integer i. With this in mind, in order to find the total amount of work done by the n calls to `get`, we simply need to know how many total nonzero digits there are in the base k representations of all integers in $\{1,2,\ldots,n\}$. First, we see that each of these numbers will be representable using only $\lfloor \log_k n \rfloor + 1$ digits. Next, each of these digits (except perhaps the most significant one) will be nonzero in approximately a $\frac{k-1}{k}$ of the integers. Then, the number of nonzero elements can be approximated as $\frac{k-1}{k} n \cdot \log_k n$, and since we treat k as a constant, the work of the program must be linearithmic with respect to the size of the sequence.

D. A Fully Sequential Implementation

```
module ArraySeq : S = struct
  type 'a t = 'a array


  let tabulate (f : int -> 'a) (n : int) : 'a t = Array.init n f
  let seq_of_array (a : 'a array) : 'a t = Array.copy a
  let array_of_seq (s : 'a t) : 'a array = Array.copy s
  let iter = Array.iter
  let iteri = Array.iteri
  let length = Array.length
  let empty () : 'a t = [||]
  let cons (x : 'a) (s : 'a t) : 'a t = Array.append [| x |] s
  let singleton (x : 'a) : 'a t = [| x |]
  let append = Array.append
  let nth = Array.get
  let map = Array.map
```

```
let map_reduce (f : 'a -> 'b) (g : 'b -> 'b -> 'b) (b : 'b) (s : 'a t) : 'b =
  let n = Array.length s in
  let rec iter acc i =
    if i = n then acc else iter (g acc (f s.(i))) (i + 1)
  in
  iter b 0

let reduce = Array.fold_left

let flatten (a : 'a array array) : 'a array =
  let n = Array.length a in
  let m = map_reduce Array.length ( + ) 0 a in
  if m = 0 then empty ()
  else
    let b = Array_handler.get_uninitialized m in
    let rec copy i k =
      if i = n then ()
      else
        let len = Array.length a.(i) in
        Array.blit a.(i) 0 b k len;
        copy (i + 1) (k + len)
    in
    copy 0 0;
    b

let repeat (x : 'a) (n : int) : 'a t = Array.make n x
```

```
let zip ((s1, s2) : 'a t * 'b t) : ('a * 'b) t =
  let len1, len2 = (length s1, length s2) in
  if len1 != len2 then raise (Invalid_argument "ArraySeq.zip")
  else tabulate (fun i -> (s1.(i), s2.(i))) len1

let split s i = (Array.sub s 0 i, Array.sub s i (Array.length s - i))

let scan (f : 'a -> 'a -> 'a) (b : 'a) (s : 'a array) =
  let u = Array.copy s in
  let n = Array.length s in
  let rec iter i x =
    if i = n then ()
    else
      let y = f x u.(i) in
      u.(i) <- y;
      iter (i + 1) y
  in
  iter 0 b;
  u

let filter (pred : 'a -> bool) (s : 'a t) : 'a t =
  let matches = map pred s in
  let num_matches =
    map_reduce (fun b -> if b then 1 else 0) ( + ) 0 matches
  in
  let a = Array_handler.get_uninitialized num_matches in
```

```
    let rec iter i next =
      if next = num_matches then ()
      else if matches.(i) then (
        a.(next) <- s.(i);
        iter (i + 1) (next + 1))
      else iter (i + 1) next
    in
    iter 0 0;
    a
end
```