Chunked Tree Sequences

Rohan Yadav

Carnegie Mellon University : School of Computer Science

Table of Contents

Introduction

Functional programs naturally describe transformations over data to solve problem instances, and can compact complicated program logic into a few calls to simple primitives. Additionally, functional programming very naturally describes parallelism, as these primitives can be implemented in a parallel fashion, and key features like immutability allow for easy reasoning about run time and correctness of code running in parallel. We will focus on the sequence data structure, and show work towards a fast parallel implementation of the data structure using chunked tree sequences. We will also compare implementations of the chunked structure with a standard array based implementation in Standard ML.

1 Sequences

1.1 Background

A sequence is mathematically defined as function from $\{0, 1, ..., n-1\} \to \alpha$, or a collection of elements of type α, ordered by index, where we can access elements by their index in the sequence. We can parametrize a wide variety of programs by expressing the algorithms in terms sequences and sequence operations, and easily reason about the cost and correctness of the algorithms given a certain sequence operation.

1.2 Interface

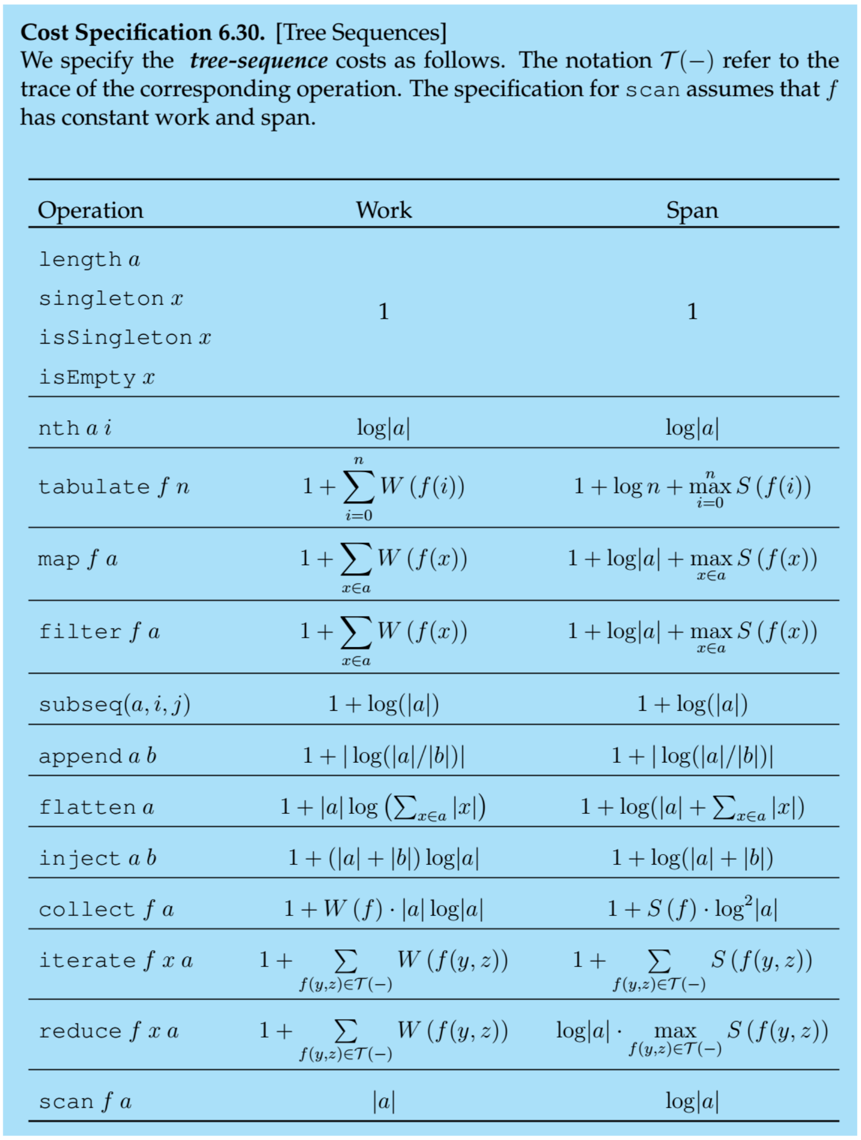
Throughout this section, we will denote the types of the following sequence functions with a ML type syntax.
The current sequence library supports the following operations, along with a few other operations :

- length : α seq $\to$ int
- nth : α seq $\to$ int $\to$ α
- empty : α seq
- singleton : $\alpha \to \alpha$ seq
- subseq : α seq $\to$ int $\to$ int $\to$ α seq
- append : α seq $\to$ α seq $\to$ α seq
- tabulate : (int $\to \alpha$) $\to$ int $\to$ α seq
- map : $(\alpha \to \beta) \to \alpha$ seq $\to \beta$ seq
- reduce : $(\alpha \times \alpha \to \alpha) \to \alpha \to \alpha$ seq $\to \alpha$
- iterate : $(\beta \times \alpha \to \beta) \to \beta \to \alpha$ seq $\to \beta$
- scan : $(\alpha \times \alpha \to \alpha) \to \alpha \to \alpha$ seq $\to$ (α seq $\times\alpha$)
- filter : ($\alpha \to$ bool) $\to \alpha$ seq $\to \alpha$ seq
- merge : ($\alpha \times \alpha \to$ order) $\to$ (α seq $\times\alpha$ seq) $\to \alpha$ seq

The details of these operations, and more can be found at
`http://www.cs.cmu.edu/~15210/docs/sig/sequence/SEQUENCE.html`.

1.3 Cost Semantics

The baseline implementation of sequences uses the Standard ML mutable Array. Arrays allow for the expected cost semantics of sequences, since they allow for constant time access into the array. We will also discuss the cost semantics of using tree bases sequences, which use internal tree data types to create a tree structured representation of a sequence. Since the sequence is structured as a tree, we lose the constant time element look up for a logarithmic time look up, which changes the cost semantics of the structure. The costs for the chunked tree sequences are the same as normal tree sequences, but differ by constant factors, as the chunks of elements change the depth of the tree and add serial base cases to some algorithms.



Cost Specification 6.30. [Tree Sequences]
We specify the ***tree-sequence*** costs as follows. The notation $\mathcal{T}(-)$ refer to the trace of the corresponding operation. The specification for `scan` assumes that f has constant work and span.

Operation	Work	Span
`length` a `singleton` x `isSingleton` x `isEmpty` x	1	1
`nth` a i	$\log\|a\|$	$\log\|a\|$
`tabulate` f n	$1 + \sum_{i=0}^{n} W(f(i))$	$1 + \log n + \max_{i=0}^{n} S(f(i))$
`map` f a	$1 + \sum_{x \in a} W(f(x))$	$1 + \log\|a\| + \max_{x \in a} S(f(x))$
`filter` f a	$1 + \sum_{x \in a} W(f(x))$	$1 + \log\|a\| + \max_{x \in a} S(f(x))$
`subseq`(a, i, j)	$1 + \log(\|a\|)$	$1 + \log(\|a\|)$
`append` a b	$1 + \|\log(\|a\|/\|b\|)\|$	$1 + \|\log(\|a\|/\|b\|)\|$
`flatten` a	$1 + \|a\| \log\left(\sum_{x \in a} \|x\|\right)$	$1 + \log(\|a\| + \sum_{x \in a} \|x\|)$
`inject` a b	$1 + (\|a\| + \|b\|) \log\|a\|$	$1 + \log(\|a\| + \|b\|)$
`collect` f a	$1 + W(f) \cdot \|a\| \log\|a\|$	$1 + S(f) \cdot \log^2\|a\|$
`iterate` f x a	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$
`reduce` f x a	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$\log\|a\| \cdot \max_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$
`scan` f a	$\|a\|$	$\log\|a\|$

2 Chunked Tree Sequences

2.1 Chunking Technique

The idea of chunking is to store a set, or chunk, of data, rather than a single element. When applied to tree based sequence representations, instead of storing a single element at a leaf of the tree, we store a chunk of elements, with size

at most equal to some configurable parameter `chunksize`. This technique still shares the same asymptotic properties as a normal tree sequence, except now the run time costs have constant factors dependent on this `chunksize` parameter. Even though the asymptotic behavior of this implementation is the same as tree sequences, when we are trying to get optimum performance out of parallel machine, constant factors are extremely important, and chunked sequences offer some advantages over array based implementations.
First, chunked tree sequences offer implicit granularity control. Parallel programs can distribute work among cores and switch to a sequential baseline when recursion hits the chunk level in the tree. This property allows for very natural implementations of sequence primitives, as well as easy use in divide and conquer algorithms.
Using chunked sequences can allow for a purely functional implementation, which makes reasoning about the implementation easier, and allow for the implementer to not have to worry about large amounts of data copying to preserve immutability from the outside.
Using chunked sequences can incur less memory overhead and contention than an array implementation, as when a change is made to the sequence, as if the whole sequence is needed to be remade, then one large block allocation does not need to made, but many smaller allocation requests. Additionally, if only part of the sequence is changing, the entire sequence doesn't need to be recomputed, only the chunks that are affected.

2.2 Key Algorithms

We will start by describing how simpler primitives like nth, map and reduce are implemented, before moving on to more complicated primitives like scan and merge.

Datatype Consider we have a datatype like the following :

```
type chunked_seq = Empty
                 | Chunk of chunk_type
                 | Node of int * chunked_seq * chunked_seq
```

Where the chunk could be implemented as any mutable or immutable block of memory, and the first element of the node type is the length of the sequence.

Tabulate To create a tree sequence like this, we can use an algorithm like this

```
fun tabulate f n =
  let
    fun tab (s, l) =
      if l <= chunksize then Chunk.tabulate (fn i => f(s + i)) l else
      let
```

```
          val half = l div 2
          val (L, R) = (tab (s, half) || tab (s + half, l - half))
        in
          Node(L,R)
        end
    in
      tab (0, n)
    end
```

This algorithm splits up the given index into the distinct chunks, and when it gets down to the chunk level it actually allocates the chunk for the sequence.

Indexing To take the nth element of the sequence, we use the following algorithm :

```
  fun nth Empty i = raise OutOfBounds
    | nth Chunk c i = Chunk.nth i
    | nth Node (len, l, r) i =
      if i < length l then nth l i else nth r (i - length l)
```

It is clear why if the tree is relatively balanced this algorithm takes logarithmic time, to look up an element, but a large constant factor faster than having to recurse all the way down to a single element, as we can consider the chunk structure to have constant time element look up.

Map Implementing map is very intuitive on the chunked tree sequences, and is and example of the inbuilt granularity that comes with using chunked sequences.

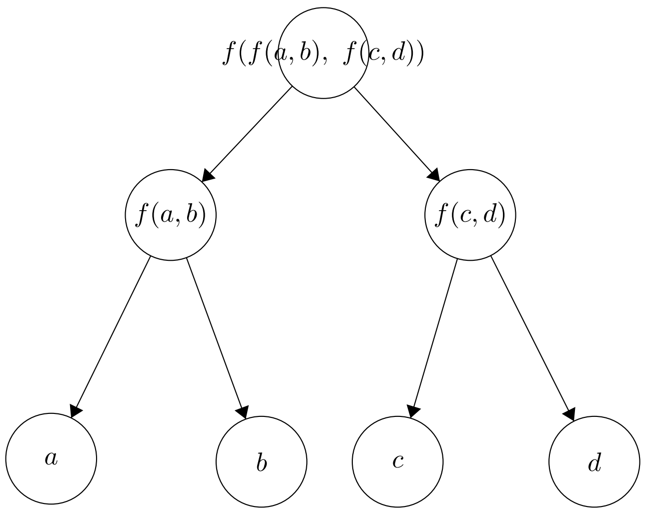
```
    fun map f Empty = Empty
      | map f Chunk c = Chunk.map f c
      | map f Node(i, l, r) =
        let
          val (l', r') = (map f l || map f r)
        in
          Node(i, l', r')
        end
```

Ideally, we can implement Chunk.map to be a serial mapping operation, and this implementation can easily distribute work to each active core, and then compute the serial map.

Reduce We can implement reduce in a very similar way, allowing good use of a serial algorithm at the chunk level :

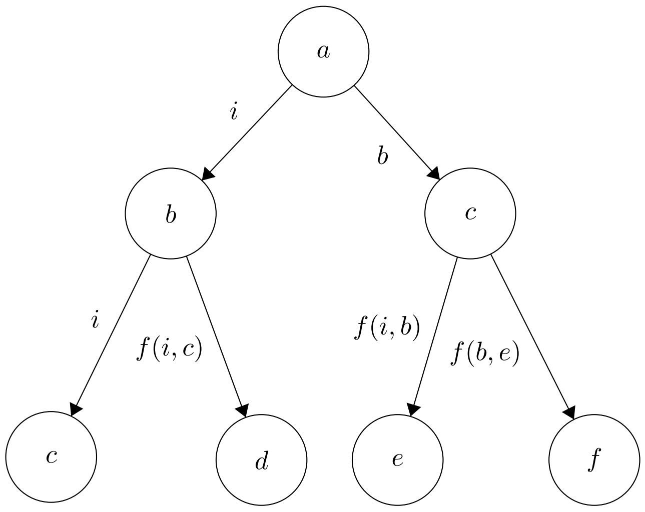
```
fun reduce f b Empty = b
  | reduce f b Chunk c = Chunk.iterate f b c
  | reduce f b Node(_, l, r) =
    let
      val (r1, r2) = (reduce f b l || reduce f b r)
    in
      f(r1, r2)
    end
```

Scan Now we can discuss more complicated primitives like scan, and merge. These algorithms are traditionally implemented in a way that is efficient when accessing arbitrary elements is fast, in-place writes to memory are fast, and very few passes over the data are made.
To implement an efficient scan on chunked sequences, we appeal to the idea that we can do a serial, in place scan over each chunk, and then propagate the results from each scan onto the rest of the sequence. To be clearer, if our sequence consists of chunks `A`, `B`, and `C`, if we scan over each and get `(A', a)`, `(B', b)`, and `(C', c)`, the resulting chunks would be `A'`, `map (fn x => f(a, x)) B'`, and `map (fn x => f(a,f(b,x)) C'`. This intuition can give us a 2 step algorithm to implement scan - an up sweep phase, where we scan over each chunk in the tree, and communicate the results up, and a down sweep phase, where we propagate partial results down to every chunk in the sequence.
If we have a tree with 4 chunks, we can model this up sweep phase as the following :



We push the reduction function down the tree until we hit a chunk, where we scan perform an iterative scan, and end up with the partial results of a, b, c, and d. Then, we construct a tree that mirrors the original structure of the tree, but stores these reduced values for easy access.

We then begin the down sweep phase, where we push the correct combined value down the tree, and perform a mapping operation at the chunk level.



The basic algorithm is given some base case c, we pass c to our left child, and pass f(c, reduced value of left child) to our right child to give each chunk the amount it needs to be mapped over by.
An alternate algorithm is to just perform a scan on each chunk in parallel, and construct a sequence with the results of each. Then we can scan over this created sequence, and propagate the results down to each chunk. Both algorithms perform similarly in practice.

Merge Now, we can discuss a parallel merging operation on these chunked sequences. What we do is take the middle element of the first sequence. Next, we do a binary search to find out what index this would be in the second sequence. Then we can just split that sequence at that index, and recursively merge those halves with the halves of the first sequence. This algorithm is the same as the algorithm for performing a parallel merge on arrays.

Originally, our first algorithm was more complicated than this divide and conquer solution. What we did was, first take the last element of every chunk in the first sequence. Then, do a bulk binary search to find the index of each in the second sequence. With these indices, we can find a a sequence of subsequences of the second array, such that the elements in that subsequence fit within the range of elements in each chunk. Then, we just recursively merge each chunk with this subsequence, and flip the order of the arguments to merge, so that if these subsequences are very large, they will be split up into chunks as well. This algorithm seemed like it would work well, but upon testing it was quite slow, and did not scale well. The overhead of creating all of the separate subsequences and the bulk binary searches were obstacles in getting good performance from this implementation.

3 Implementation Details

We implement these chunked sequences with the methods detailed above, and use the inbuilt MLton Vector structure to represent the actual chunks. Vectors are a functional array supported by MLton, that allows for constant time updates, but doesn't support constant time updates, as it is immutable.
We balance these trees in a way that preserves the invariant that the length of one subtree is not greater than some α * length of other subtree. We enforce this invariant during all sequence functions that change the structure of the tree, such as operations like append and subsequence.

One problem that typically arises with chunked representations like these is that the chunks can become very sparse. The implementation of our chunked sequence structure guarantees that this does not happen. The only two ways of creating a sequence are with the `tabulate` and `append` methods. `tabulate` forces every chunk created to be as full as possible, never leaving around any sparse interior chunks. The only other way to make a new sequence with a different structure is to use the `append` method, which at every level checks to see if the two trees that are being appended can fit in single chunk. If they can, it compresses into a single chunk, regardless of whether the two trees were chunks or tree structures themselves. Lastly, whenever we consider merging two chunks, if one chunk is less than half full, we redistribute the data between the two chunks the make sure that no chunk is below half full. Therefore, no sequence structure created by our library can have any chunks that are less than half empty. The reason why this is stressed is due to the fact that the paradigm while using these sequences is to fun a parallel algorithm on the tree structure, and have a sequential base case once all processors have had their work divided up. If some chunks are not very full, there will be a large amount of workload imbalance between the cores, which heavily limits performance of parallel code.

The full implementation can be found in the file `MkChunkedTreeSequence.sml`. Some operations like update and inject are still unimplemented, and some more complicated operations like collect are implemented inefficiently to just finish the implementation.

4 Benchmarks

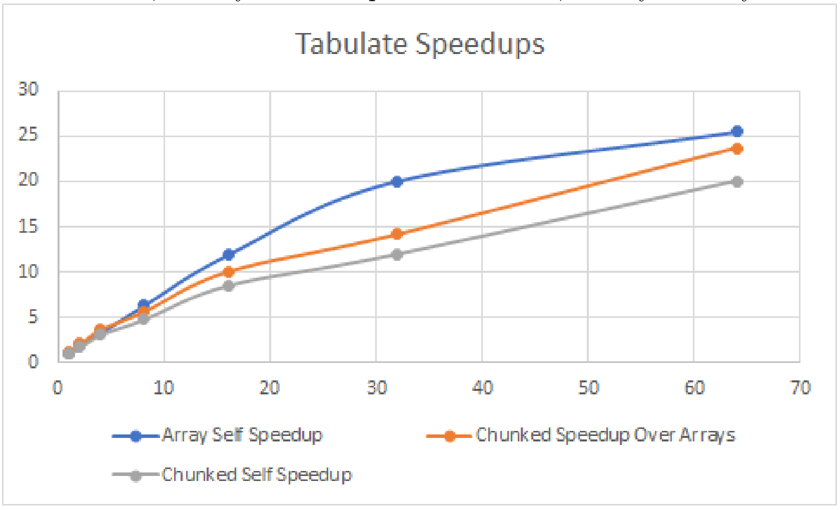
4.1 Data Presentation

We will benchmark the chunked tree sequences against our most efficient array based implementation of sequences. A_n is the time the array implementation takes on n cores. C_n is the time that the chunked implementation takes on n cores. We present the benchmarks with 3 variables : $\frac{A_1}{A_n}$, $\frac{C_1}{C_n}$, and $\frac{A_1}{C_n}$. These represent the self speedups the arrays experience, the self speedup the chunked sequences experience, and the actual speedup over the arrays on 1 core the

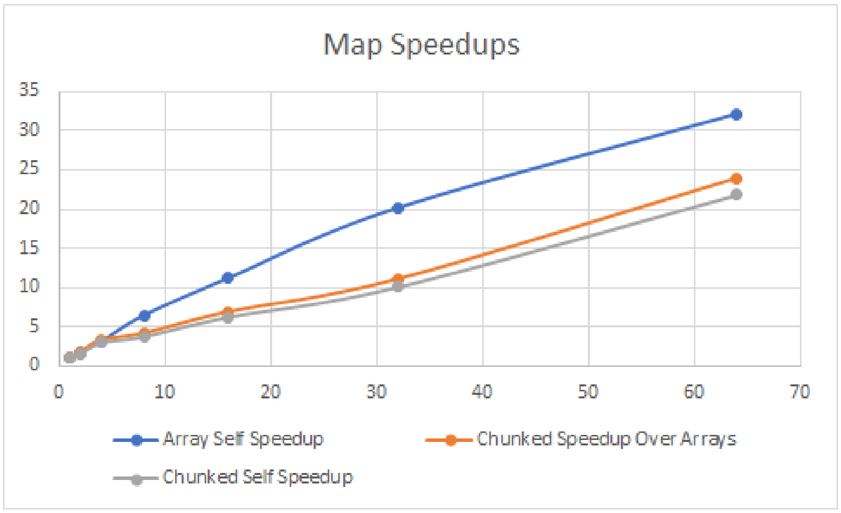
chunked sequences experience. These measurements let us know how both implementations scale to high core counts, as well as what the actual speedup of the chunked sequences over the array sequences are.
The chunk size used for these experiments was 2000. We chose this chunk size by experimenting, and noting that as we lowered the chunk size, performance fell on both low and high core counts, and if we increased the chunk size, performance fell on high core counts (not enough parallelism). Additionally, all results have been timed without garbage collection time factored in, as mlton-spoonhower runs with a sequential garbage collector that becomes a large bottleneck at high core counts. Research is being done towards parallel garbage collection for SML. Additionally, these benchmarks were compiled with the `mlton-spoonhower` compiler, and invoked with the runtime arguments
`./exec @MLton number-processors <n> alloc-chunk 256K - <extra arguments>`.
The measurements were taken on an Intel machine with 72 cores (144 hyperthreads), running at 2.40 GHz, and 1 TB of ram.

4.2 Results

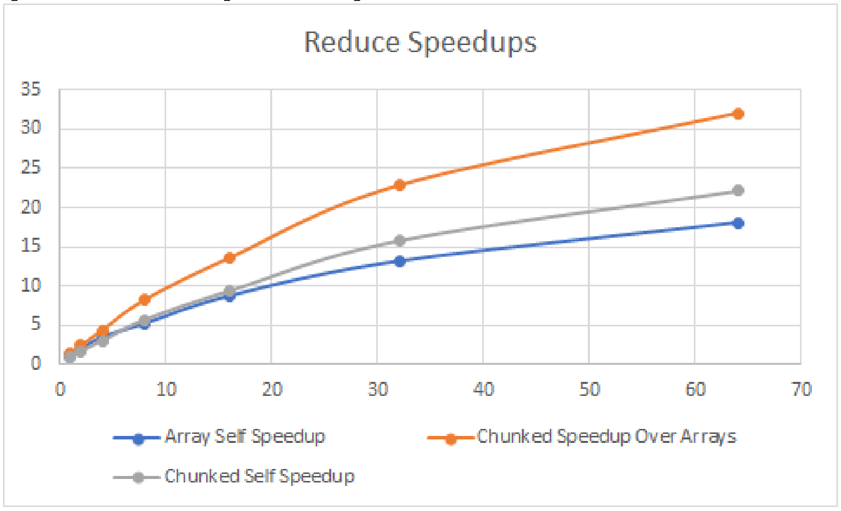
Tabulate scales well, but as it is a trivially parallel computation, but is bandwidth bound, as very little computation is done, mainly memory transfers.



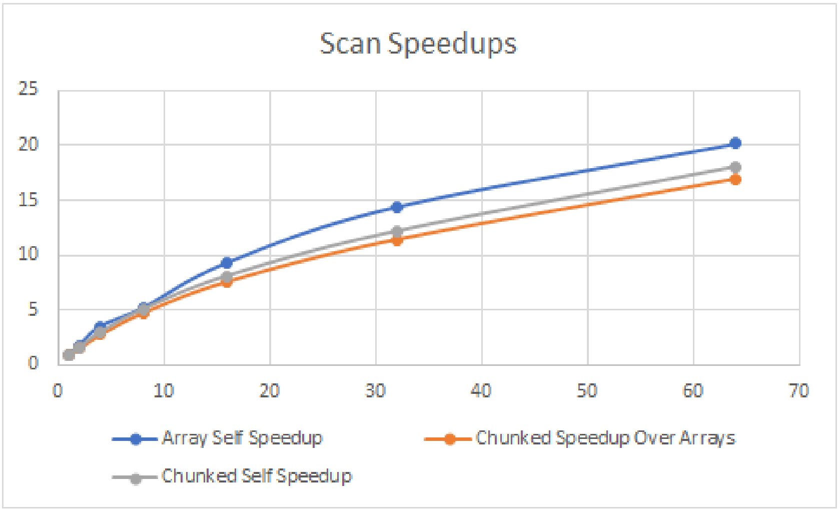
Map is inherently a very similiar computation to tabulate, and thus scales the same way.



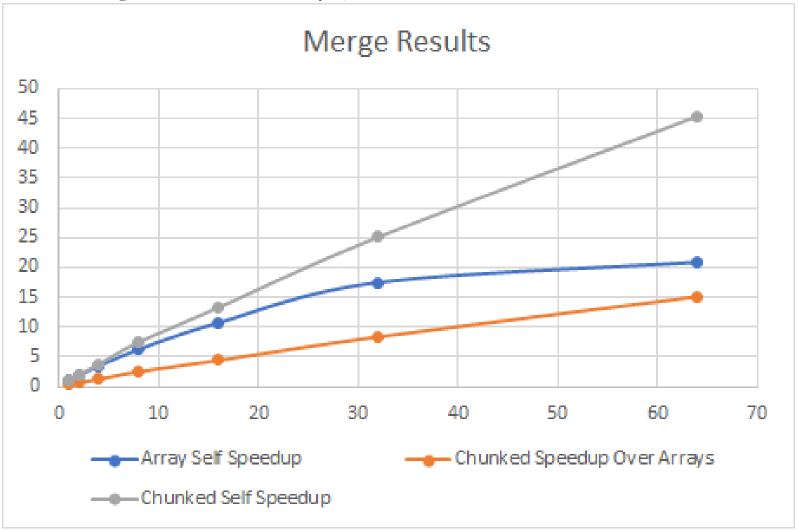
Reduce performs very well on chunked tree sequences, as the divide and conquer nature of the problem is perfect for the tree structure.



Scan experiences good speedup. While still being beaten out by the arrays, the chunked scan offers competitive performance for an algorithm that seems to be very more suited for an array based implementation.



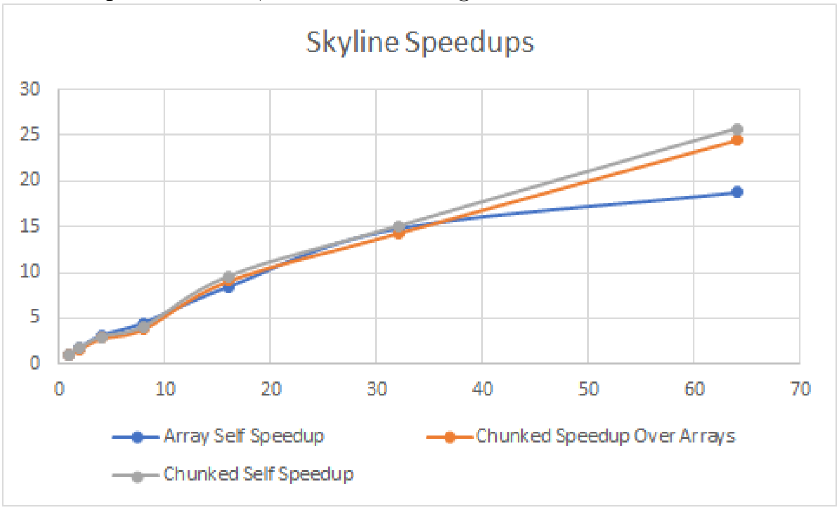
Merge on chunked sequences experiences a large slowdown on a single core when compared to the arrays, but scales better as core count increases.



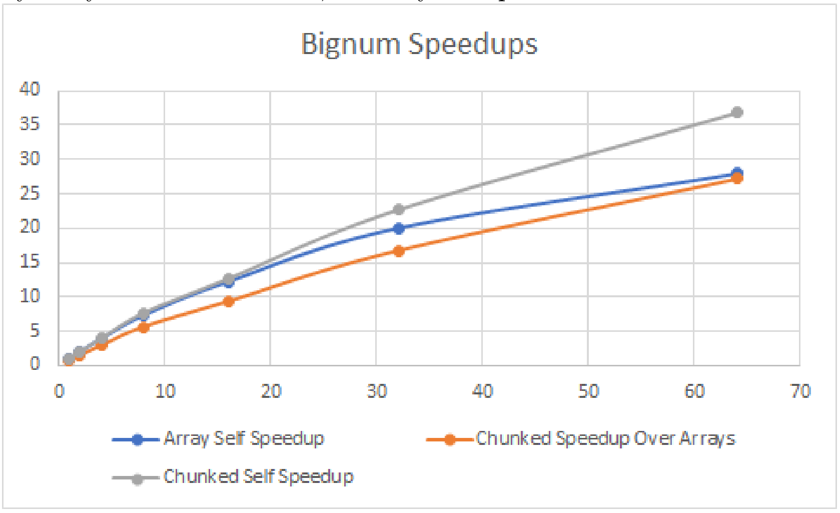
Since mergesort is basically just multiple calls to merge, mergesort follows the same speedup characteristics as merge.

Additionally, we tried to run the chunked sequences on the parentheses matching algorithm from 15-210, and expected to see performance that beat the array based implementation, since the algorithm is essentially a reduce. However, the performance of the chunked sequences were atrocious, almost a order of magnitude slower than the arrays on a single core, as well as not scaling well at all. We are unable to explaoin this behavior and are very puzzled.

Skyline performs very well with chunked tree sequences, beating out an array based implementation, as well as scaling better with core counts.



Bignum's performance scales well with chunked sequences, but is beaten out by arrays at all core counts, but stays competitive.



5 Future Work

Our future work is to extend the chunking idea to a structure called clustered trees. The idea for this structure is to have a tree structure of the sequence still, except instead of having a chunk, the tree will hold clusters, which are arrays that each point to a chunk. The idea for this is to allow another level of parallelism : a divide and conquer algorithm at the top, a parallel-for at the clusters, and a serial algorithm at the chunks. Due to the good result we saw from the chunked sequences, we hope to see performance increases with this data structure as well. Preliminary testing with these kind of structures show improved performance relative to both arrays and chunked sequences on operations like reduce. Additionally, this work presented revealed issues in the current memory system of the mlton runtime. Future work would go towards building an allocation system that better supported large amounts of concurrent allocations.