Columbia | Engineering
The Fu Foundation School of Engineering and Applied Science

COMS W4995 002: Parallel Functional Programming Fall 2021

Project Report

Professor Stephen A. Edwards

PowerList

Yash Datta

UNI yd2590

December 21, 2021

Contents

1 Introduction

J. Misra [1], has introduced **powerlist**, a new recursive data structure that permits concise and elegant description of many data parallel algorithms like Prefix-sum, Batcher's sorting schemes, FFT etc. Similar to a list structure, the base case of powerlist is a list of one element. Longer power lists are constructed from the elements of 2 powerlists, p and q, of the same length using 2 operators described below.

- $p \mid q$ is the powerlist formed by concatenating p and q. This is called **tie**.
- $p \bowtie q$ is the powerlist formed by successively taking alternate items from p and q, starting with p. This is called **zip**.

Further, both p and q are restricted to contain similar elements.
Following additional operators are necessary to express algorithms in terms of powerlists:

- $p \oplus q$ is the powerlist obtained by applying the binary scalar operator $\oplus$ on the elements of p and q at the same position in the 2 lists.
- L^* is the powerlist obtained by shifting the powerlist L by one. The effect of shifting is to append a 0 to the left and discard the rightmost element.

Note that 0 is considered the left identity element of $\oplus$, i.e. $0 \oplus x = x$.
In the following examples, elements of powerlist are enclosed within angular brackets.

- $\langle 0 \rangle \mid \langle 1 \rangle = \langle 0\ 1 \rangle$
- $\langle 0 \rangle \bowtie \langle 1 \rangle = \langle 0\ 1 \rangle$
- $\langle 0\ 1 \rangle \mid \langle 2\ 3 \rangle = \langle 0\ 1\ 2\ 3 \rangle$
- $\langle 0\ 1 \rangle \bowtie \langle 2\ 3 \rangle = \langle 0\ 2\ 1\ 3 \rangle$

There are many applications of this structure for parallel algorithms, some of which are:

- Prefix-sum (scan)
- Batcher sort
- Rank sort
- Fast Fourier Transform

In this project, I have developed a haskell module to demonstrate the use of powerlist in the following algorithms:

- **scan**: Several variations of scan algorithms have been developed, which are primarily of 2 types
  - Simple Prefix Sum
  - Ladner Fischer [1]
- **sort**: A Batcher merge sort scheme has been implemented for sorting using powerlist.

2 Project Description

This haskell project has been developed as a benchmark utility for running and comparing different parallel algorithms using powerlist. The project uses stack for building and creates 2 executables:

- powerlist-exe: for executing and analyzing each of the different algorithms individually
- powerlist-benchmark: for benchmarking each of the algorithm functions by executing them multiple times. This uses criterion package, hence all command line options of criterion can be used.

The project source code including detailed benchmarks are hosted on github here.

2.1 Usage

To run each of the algorithms, use the powerlist-exe executable, which supports 2 commands **scan** and **sort**:

```
> stack exec powerlist-exe -- scan
Usage: powerlist-exe scan (-a|--algo ALGONAME) (-s|--size INPSIZE)
                          [-c|--csize CHUNKSIZE]
  Run Scan Algorithm

Available options:
  -a,--algo ALGONAME        Supported Algos:
                            [SPS,SPSPL,SPSPLPar1,SPSPLPar2,SPSPLPar3,LDF,LDFPar,
                            SPSUBVecPLPar,LDFUBVecPLPar,LDFChunkUBVecPLPar]

  -s,--size INPSIZE         Size of array in terms of powers of 2 on which to run
                            scan
  -c,--csize CHUNKSIZE      Size of chunks for parallelization
  -h,--help                 Show this help text
```

and

```
> stack exec powerlist-exe -- sort
Usage: powerlist-exe sort (-a|--algo ALGONAME) (-s|--size INPSIZE)
                          [-c|--csize CHUNKSIZE]
  Run Sort Algorithm

Available options:
  -a,--algo ALGONAME        Supported Algos: [DEFAULT,BATCHER]
  -s,--size INPSIZE         Size of array in terms of powers of 2 on which to run
                            sort
  -c,--csize CHUNKSIZE      Size of chunks for parallelization
  -h,--help                 Show this help text
```

For example to run simple prefix sum algorithm using powerlist on an array of input size 2^5:

```
> stack exec powerlist-exe -- scan --algo SPSPL --size 5
5984
```

To run the parallel version of the same algorithm using powerlist, on 8 cores and generate eventlog file for threadscope analysis:

```
> stack exec powerlist-exe -- scan --algo SPSPLPar1 --size 20 +RTS -N8 -ls
192154133857304576
```

To run the benchmark for the same algorithm, use powerlist-benchmark executable:

```
> stack exec powerlist-bench -- main/scan/par/nc/SPSPLPar1 +RTS -N8
benchmarking main/scan/par/nc/SPSPLPar1
```

```
time                 1.506 s    (1.316 s .. 1.576 s)
                     0.996 R^2   (0.994 R^2 .. 1.000 R^2)
mean                 1.393 s    (1.324 s .. 1.448 s)
std dev              77.82 ms   (70.55 ms .. 79.28 ms)
variance introduced by outliers: 19% (moderately inflated)
```

Input Output

The input to **scan** algorithm is generated as a simple list/vector of length 2^d from 1 to d. The **scan** algorithm when run simply ouputs the sum of the elements of the prefix sum array. So for example for input $[1, 2, 3, 4]$, prefix sum array $= [1, 3, 6, 10]$, and sum of the elements of this array $= 20$.

The input to **sort** is generated as a reverse list/vector of length 2^d from 2^d down to 1. The **sort** algorithm when run simply outputs the last element of the sorted array. This is done to prevent I/O from affecting algorithm performance in individual runs. Note that the benchmark utility benchmarks the functions directly, so is not affected by any IO.

Here d is the value of the supplied –size param.

2.2 Powerlist in Haskell

I have used 2 different implementations of powerlists, one using *List* and the other using *Data.Vector.Unboxed*

2.2.1 Powerlist as List

The List implementation of powerlist is straightforward and allows to implement the required operators easily:

```haskell
-- Using simple list here as it would be most performant
type PowerList a = [a]
```

The *tie* function is same as ++ of List in haskell, but *zip* is a bit different which is shown below:

```haskell
zip :: PowerList a -> PowerList a -> PowerList a
{-# INLINE zip #-}
zip [] [] = []
zip xs ys = Prelude.zip xs ys >>= \(a, b) -> [a, b]
```

There is an analogous *unzip* function that is required for deconstructing the input powerlist into 2 smaller powerlists.

```haskell
unzip :: PowerList a -> (PowerList a, PowerList a)
unzip =
  snd .
  foldr
    (\x (b, (xs, ys)) ->
       ( not b
       , if b
           then (x : xs, ys)
           else (xs, x : ys)))
    (False, ([], []))
```

The L^* (shift) operator is implemented as below:

```haskell
-- Right shift and use zero
```

```
rsh :: a -> PowerList a -> PowerList a
rsh zero xs = zero : init xs
```

2.2.2 Powerlist as Unboxed Vector

Unboxed Vectors are more memory friendly. We can further reduce memory usage by converting to mutable vectors and modifying the data in place, instead of creating more copies. But the implementation of operators in terms of Vectors is a bit more involved:

```
import qualified Data.Vector.Split as S
import qualified Data.Vector.Unboxed as V
import qualified Data.Vector.Unboxed.Mutable as M

type PowerList a = V.Vector a

tie :: V.Unbox a => PowerList a -> PowerList a -> PowerList a
tie = (V.++)

zip :: (V.Unbox a, Num a) => PowerList a -> PowerList a -> PowerList a
{-# INLINE zip #-}
zip xs ys =
  V.create $ do
    m <- M.new n
    write m 0
    return m
  where
    n = V.length xs + V.length ys
    write m i
      | i < n = do
        M.unsafeWrite m i (xs V.! (i `div` 2))
        M.unsafeWrite m (i + 1) (ys V.! (i `div` 2))
        write m (i + 2)
      | otherwise = return ()

zipWith ::
     (Num a, V.Unbox a)
  => (a -> a -> a)
  -> PowerList a
  -> PowerList a
  -> PowerList a
{-# INLINE zipWith #-}
zipWith op xs ys =
  V.create $ do
    m <- V.thaw xs
    write m ys 0
    return m
  where
    k = V.length xs
    write m y i
      | i < k = do
        curr <- M.unsafeRead m i
        M.unsafeWrite m i (op (y V.! i) curr)
        write m y (i + 1)
      | otherwise = return ()

```

```
unzip :: V.Unbox a => PowerList a -> (PowerList a, PowerList a)
unzip k = (b, c)
  where
    b = V.ifilter (\i _ -> even i) k
    c = V.ifilter (\i _ -> odd i) k
```

The $zipWith$ is nothing but $\oplus$ operator described previously. Parallel versions of these operators have also been implemented, by splitting the input into chunks and running the operator in parallel on the chunks, and later combining the results.

2.3 Algorithms

As mentioned previously, many different versions of some algorithms, have been implemented with several optimizations to parallelize them effectively. The different algorithms and techniques used are described below.

2.3.1 Simple Prefix Sum

Prefix sum or scan is used to generate a prefix sum array from the input array, by summing all the elements of the array upto each element. So for example the prefix sum for input array $[1, 2, 3, 4]$ is given by $[1, 3, 6, 10](1 = 1, 3 = (1 + 2), 6 = (1 + 2 + 3), 10 = (1 + 2 + 3 + 4))$ This is nothing but $scanl1$ in haskell.

```
Prelude> scanl1 (+) [1, 2, 3, 4]
[1,3,6,10]
```

The following variations of this simple algorithm have been implemented:

- **SPSPL** : A sequential prefix sum using powerlist, to demonstrate equivalence.
- **SPSPLPar1** : A parallel implementation of SPSPL, with the Eval Monad, first attempt.
- **SPSPLPar2** : More optimized parallel implementation of SPSPL, with the Eval Monad.
- **SPSPLPar3** : Only evaluate in parallel till certain depth, then fall back to scanl1.
- **SPSUBVecPLPar** : A variation of **SPSPLPar3** using Unboxed Vectors.

These are described further below.

2.3.1.1 SPSPL

Powerlist allow a simple prefix sum function:

$$
\begin{aligned}
&sps\ \langle x\rangle = \langle x\rangle\\
&sps\ L = (sps\ u)\ \bowtie\ (sps\ v)\\
&\quad where\ u\ \bowtie\ v = L^*\ \oplus\ L
\end{aligned}
$$

which translates beautifully into haskell code:

```
import qualified Powerlist as P

sps :: Num a => (a -> a -> a) -> P.PowerList a -> P.PowerList a
sps _ [] = []
sps _ [x] = [x]
sps op l = P.zip (sps op u) (sps op v)
  where (u, v) = P.unzip $ P.zipWith op (P.rsh 0 l) l
```

This sequential implementation is presented to show the usefulness of powerlists and to benchmark parallel algorithms. The powerlist implementation used for this and other algorithms is backed by the *List* data structure of haskell, for its simplicity and flexibility.

2.3.1.2 SPSPLPar1

This was the first attempt at parallelizing SPSPL using the Eval monad. The algorithm is naturally recursive and divides the input into 2 equal sized arrays on which the SPSPLPar1 can be called recursively.

```
parSps1 :: (NFData a, Num a) => (a -> a -> a) -> P.PowerList a -> P.PowerList a
parSps1 _ [] = []
parSps1 _ [x] = [x]
parSps1 op l =
  runEval
    (do (u, v) <- rseq $ P.unzip $ P.zipWith op (P.rsh 0 l) l
        u' <- rparWith rdeepseq (parSps1 op u)
        v' <- rparWith rdeepseq (parSps1 op v)
        rseq $ P.zip u' v')
```

The list implementation of powerlist has been used here. There is some bottleneck when we try to deconstruct the list into 2 halves, where several computations are being done linearly. Other problem is that we are creating a lot of intermediate lists, which require GC.

2.3.1.3 SPSPLPar2

In this variation, further optimization is done by using below techniques:

- We can parallelize the *unzip* operation that corresponds to deconstructing the list by observing that it is just creating 2 lists by elements from odd and even places in the input list. This is achieved by 2 simple functions *odds* and *evens*.
- We can parallelize $zipWith$ operation that corresponds to $\oplus$ operator by breaking the input lists into chunks and calling $zipWith$ on the corresponding chunks of the 2 input lists and concatenating the output from each such call.
- There is a clever observation that since after right shift with 0 we are trying to run a $zipWith$ operation, we can simply prepend 0 to the list and run the $zipWith$, since $zipWith$ will automatically only run the operation on elements at the same position in both lists and ignore the extra last element in the list where 0 was added. This results in elimination of right shift operation.

```
odds :: [a] -> [a]
odds [] = []
odds [x] = [x]
odds (x:_:xs) = x : odds xs

evens :: [a] -> [a]
evens [] = []
evens [_] = []
evens (_:y:xs) = y : evens xs
```

2.3.1.4 SPSPLPar3

This version further improves the runtime by recursing only till a certain depth, thereby reducing the total number of sparks generated. We revert to $scanl1$ function for input arrays smaller than 2^d length, where d is the depth parameter, which is set to 5.

2.3.1.5 SPSUBVecPLPar

This algorithm is another implementation of **SPSPLPar3** but uses powerlist implementation using Unboxed Vectors.

```haskell
import qualified UBVecPowerlist as UVP

parSpsUBVec ::
    (NFData a, UV.Unbox a, Num a)
  => (a -> a -> a)
  -> Int
  -> Int
  -> UVP.PowerList a
  -> UVP.PowerList a
parSpsUBVec _ _ _ l
  | UV.length l <= 1 = l
parSpsUBVec op cs d l
  | d > 4 =
    runEval
      (do k <- rseq $ UVP.shiftAdd l
          u <- rpar (UVP.filterOdd k)
          v <- rpar (UVP.filterEven k)
          _ <- rseq u
          u' <- rparWith rdeepseq (parSpsUBVec op cs (d - 1) u)
          _ <- rseq v
          v' <- rparWith rdeepseq (parSpsUBVec op cs (d - 1) v)
          UVP.parZip (rparWith rdeepseq) cs u' v')
parSpsUBVec op _ _ l = UV.scanl1 op l
```

This is expected to be faster because:

- Unboxed Vectors are more memory friendly.
- We introduce some additional operations like $shiftAdd$ and $filterUsing$ which directly execute the shift and add operation using mutable vectors.

2.3.2 Ladner Fischer

Another algorithm due to Ladner and Fischer can be implemented using powerlist as follows:

$$\begin{aligned} ldf\ \langle x\rangle &= \langle x\rangle \\ ldf(p \bowtie q) &= (t^* \oplus p) \bowtie t \\ &\quad where\ t = ldf(p \oplus q) \end{aligned}$$

And here is the equivalent sequential implementation in haskell:

```haskell
ldf :: Num a => (a -> a -> a) -> P.PowerList a -> P.PowerList a
ldf _ [] = []
ldf _ [x] = [x]
ldf op l = P.zip (P.zipWith op (P.rsh 0 t) p) t
  where
    (p, q) = P.unzip l
    pq = P.zipWith op p q
    t = ldf op pq
```

Again since the algorithm is naturally recursive over half the input array elements, it can be parallelized using the techniques used for SPS.

2.3.2.1 LDFPar

This is the parallel implementation of LDF algorithm, using the eval monad.

```
parLdf ::
     NFData a
  => Num a =>
       (a -> a -> a) -> Int -> Int -> P.PowerList a -> P.PowerList a
parLdf _ _ _ [] = []
parLdf _ _ _ [x] = [x]
parLdf op cs d l
  | d > 4 =
    runEval
      (do p <- rpar (odds l)
          q <- rpar (evens l)
          _ <- rseq p
          _ <- rseq q
          pq <- rseq (P.parZipWith rdeepseq cs op p q)
          t <- rparWith rdeepseq (parLdf op cs (d - 1) pq)
          k <- rseq (P.parZipWith rdeepseq cs op (0 : t) p)
          rseq $ P.zip k t)
parLdf op _ _ l = sequentialSPS op l
```

We can again use all the previous improvements introduced in parallel versions of SPS algorithms (SPSPLPar1, SPSPLPar2, SPSPLPar3) to further optimize parLdf.

2.3.2.2 LDFUBVecPLPar

This algorithm is another implementation of LDFPar. As with **SPSUBVecPLPar** this algorithm uses Unboxed Vector as the powerlist implementation.

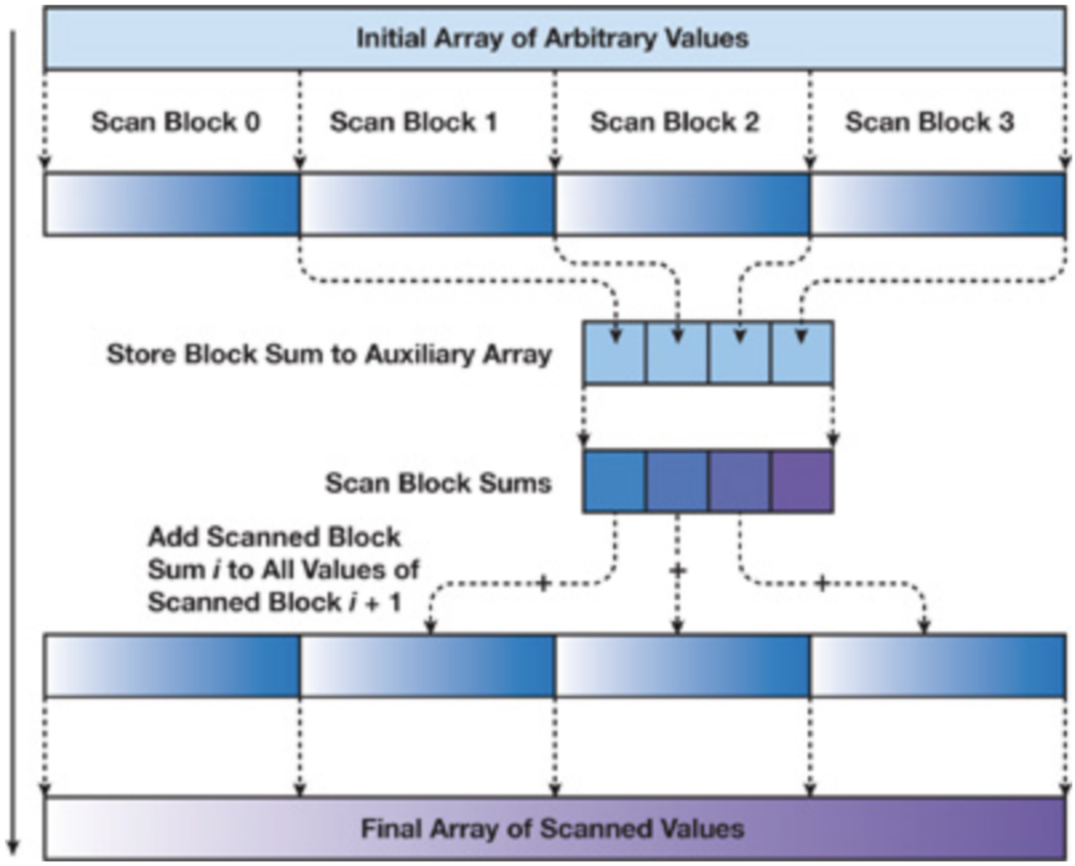
```
import qualified UBVecPowerlist as UVP

parLdfUBVec ::
     (NFData a, UV.Unbox a, Num a)
  => (a -> a -> a)
  -> Int
  -> Int
  -> UVP.PowerList a
  -> UVP.PowerList a
parLdfUBVec _ _ _ l
  | UV.length l <= 1 = l
parLdfUBVec op cs d l
  | d > 4 =
    runEval
      (do p <- rpar $ UVP.filterOdd l
          q <- rpar $ UVP.filterEven l
          _ <- rseq p
          _ <- rseq q
          pq <- UVP.parZipWith (rparWith rdeepseq) op cs p q
          t <- rpar (parLdfUBVec op cs (d - 1) pq)
          k <- rseq $ UVP.shiftAdd2 t p
          UVP.parZip (rparWith rdeepseq) cs k t)
parLdfUBVec op _ _ l = UV.scanl1 op l
```

It has the same advantages as the SPSUBVecPLPar algorithm above. Note the use of *shiftAdd*2 and *filterOdd* and *filterEven* functions that use mutable vectors and hence might consume less memory.

2.3.2.3 LDFChunkUBVecPLPar

This algorithm uses another flavor of **LDFUBVecPLPar** where we first split the input into chunks, then run **LDFUBVecPLPar** over each of these chunks and then combine the results using a technique due to Bleloch [2].

The diagram below shows how the split and combine works. Basically we divide the input into chunks and call our function on each of them in parallel. Then we store the total sum of each chunk (last element) into another auxiliary chunk. We then scan this auxiliary chunk generating an array of chunk increments that are added to all the elements in their respective chunks.



Advantages of this technique are as follows.

- Since we use Unboxed Vector, splitting into chunks takes time proportional to the number of chunks.
- Chunk size controls parallelism of the algorithm, making it more scalable than previous implementations.

Here is the implementation:

```
parLdfChunkUBVec ::
    (NFData a, UV.Unbox a, Num a)
  => (a -> a -> a)
  -> Int
  -> UVP.PowerList a
  -> UVP.PowerList a
parLdfChunkUBVec _ _ l
```

```
  | UV.length l <= 1 = l
parLdfChunkUBVec ops cs l = runEval $ parLdfChunkVec' ops chunks
  where
    n = UV.length l
    chunkSize = 2 ^ cs
    chunks = S.chunksOf chunkSize l
    parLdfChunkVec' ::
         (NFData a, UV.Unbox a, Num a)
      => (a -> a -> a)
      -> [UVP.PowerList a]
      -> Eval (UVP.PowerList a)
    parLdfChunkVec' _ [] = return UV.empty
    parLdfChunkVec' op vChunks = do
      resChunks <- parList rseq (parLdfUBVecNC op cs <$> vChunks)
      res <- rseq $ UV.concat resChunks
            -- Get last element of each block
      lastelems <- parList rdeepseq (UV.last <$> resChunks)
      lastScan <- rseq (UV.fromList $ sequentialSPS op lastelems)
      rseq $
        UV.create $ do
          m <- UV.thaw res
                -- Not sure how to parallelise here!
          mergeChunks (n - 1) (UV.tail $ UV.reverse lastScan) m
          return m
    mergeChunks i lastScan m
      | i > chunkSize = do
        let ad = UV.head lastScan
        go m chunkSize i ad 0
        mergeChunks (i - chunkSize) (UV.tail lastScan) m
      | otherwise = return ()
    go m chs start v i
      | i < chs = do
        curr <- M.unsafeRead m (start - i)
        M.unsafeWrite m (start - i) (curr + v)
        go m chs start v (i + 1)
      | otherwise = return ()
```

Here **parLdfUBVecNC** is the implementation of **LDFUBVecPLPar** without parallelizing (using chunks) the secondary operators like $zipWith$ since we already break the input into chunks at the start. I was unable to implement the addition of increments from auxiliary chunk to corresponding chunks in parallel using a mutable vector. Hence this implementation is still not optimal.

2.3.3 Batcher Merge Sort

This is another application of powerlist where a simple sorting scheme is given by:

$$sort\ \langle x\rangle = \langle x\rangle$$
$$sort(p \bowtie q) = (sort\ p)\ merge\ (sort\ q)$$

We could use any *merge* function here to merge the 2 sorted sub-lists. The Batcher scheme [1] to merge 2 sorted lists can be expressed in terms of powerlist as the below infix operator *bm*

$$\langle x \rangle \ bm \ \langle y \rangle = \langle x \rangle \ \updownarrow \ \langle y \rangle$$
$$(r \ \bowtie \ s) \ bm \ (u \ \bowtie \ v) = (r \ bm \ v) \ \updownarrow \ (s \ bm \ u) \qquad where \ p \ \updownarrow \ q = (p \ min \ q) \ \bowtie \ (p \ max \ q)$$

Here, a comparison operator $\updownarrow$ has been used which is implemented as the $minMaxZip$ function in haskell code. The operator is applied to a pair of equal length powerlists, p, q, and it creates a single powerlist by setting the $2i^{th}$ element to $p_i \ min \ q_i$ and setting the $(2i+1)^{th}$ element to $p_i \ max \ q_i$, where p_i and q_i are the i^{th} elements of each of the 2 lists.

Here is the $minMaxZip$ function for powerlist of vectors

```
minMaxZip :: (V.Unbox a, Ord a) => PowerList a -> PowerList a -> PowerList a
minMaxZip xs ys =
  V.create $ do
    m <- M.new n
    write m 0
    return m
  where
    n = V.length xs + V.length ys
    write mv i
      | i < n = do
        let p = xs V.! (i `div` 2)
        let q = ys V.! (i `div` 2)
        M.unsafeWrite mv i (p `min` q)
        M.unsafeWrite mv (i + 1) (p `max` q)
        write mv (i + 2)
      | otherwise = return ()
```

This gives us the following batcher merge sort implementation in haskell

```
batcherMergeSort :: (Ord a, V.Unbox a) => P.PowerList a -> P.PowerList a
batcherMergeSort l
  | V.length l <= 1 = l
batcherMergeSort l = sortp `batcherMerge` sortq
  where
    sortp = batcherMergeSort p
    sortq = batcherMergeSort q
    p = P.filterOdd l
    q = P.filterEven l

batcherMerge ::
     (Ord a, V.Unbox a) => P.PowerList a -> P.PowerList a -> P.PowerList a
batcherMerge x y
  | V.length x == 1 = V.fromList [hx `min` hy, hx `max` hy]
  where
    hx = V.head x
    hy = V.head y
batcherMerge x y = P.minMaxZip rv su
  where
    rv = r `batcherMerge` v
    su = s `batcherMerge` u
    r = P.filterOdd x
    v = P.filterEven y
```

```
    s = P.filterEven x
    u = P.filterOdd y
```

We use all the techniques used in the previous scan algorithms to come up with this parallel sort algorithm:

```
1  parBatcherMergeSort ::
2       (NFData a, Ord a, V.Unbox a) => Int -> P.PowerList a -> P.PowerList a
3  parBatcherMergeSort _ l
4    | V.length l <= 1 = l
5  parBatcherMergeSort d l
6    | d > 10 =
7      runEval
8        (do p <- rpar $ P.filterOdd l
9            q <- rpar $ P.filterEven l
10           _ <- rseq p
11           sortp <- rparWith rdeepseq (parBatcherMergeSort (d - 1) p)
12           _ <- rseq q
13           sortq <- rparWith rdeepseq (parBatcherMergeSort (d - 1) q)
14           parBatcherMerge d sortp sortq)
15 parBatcherMergeSort _ l = V.fromList $ defaultSort $ V.toList l
16
17 parBatcherMerge ::
18      (Ord a, V.Unbox a)
19   => Int
20   -> P.PowerList a
21   -> P.PowerList a
22   -> Eval (P.PowerList a)
23 parBatcherMerge d x y
24   | d > 10 = do
25     r <- rseq $ P.filterOdd x
26     v <- rseq $ P.filterEven y
27     rv <- parBatcherMerge (d - 1) r v
28     s <- rseq $ P.filterEven x
29     u <- rseq $ P.filterOdd y
30     su <- parBatcherMerge (d - 1) s u
31     rparWith rdeepseq $ P.minMaxZip rv su
32 parBatcherMerge _ x y = rseq (merge x y)
```

The *merge* function call at line 32 is the sequential *merge* of mergesort algorithm implemented using mutable vectors. Again this is used to reduce the number of spark generated, as this algorithm is already highly parallel.

3 Benchmark

The benchmark results of various algorithms are listed in this section. Various combinations of chunk sizes and input size were tried, together with threadscope analysis of individual runs.

3.1 Setup

All benchmarks were performed on an 8 core Intel i9-9900K CPU @ 3.60 GHZ (32G) running Debian 11 (Bullseye).

3.2 Results

The summary results for all the algorithms is listed first, followed by the details about the best ones. Detailed results for each of the algorithm can be viewed on github here.
All results listed below are for arrays / lists of input length 2^{20}.

3.2.1 Scan

Following table summarizes the results for SPS algorithm variations:

Algo Name	Num Cores	ChunkSize	Runtime (ms)	Improvement
SPSPL	1	-	5232	-
SPSPLPar1	8	-	1506	3.47X
SPSPLPar2	8	256	1483	3.52X
SPSPLPar3	8	512	1397	3.74X
SPSUBVecPLPar	8	1024	520.3	10.05X

Following table summarizes the results for LDF algorithm variations:

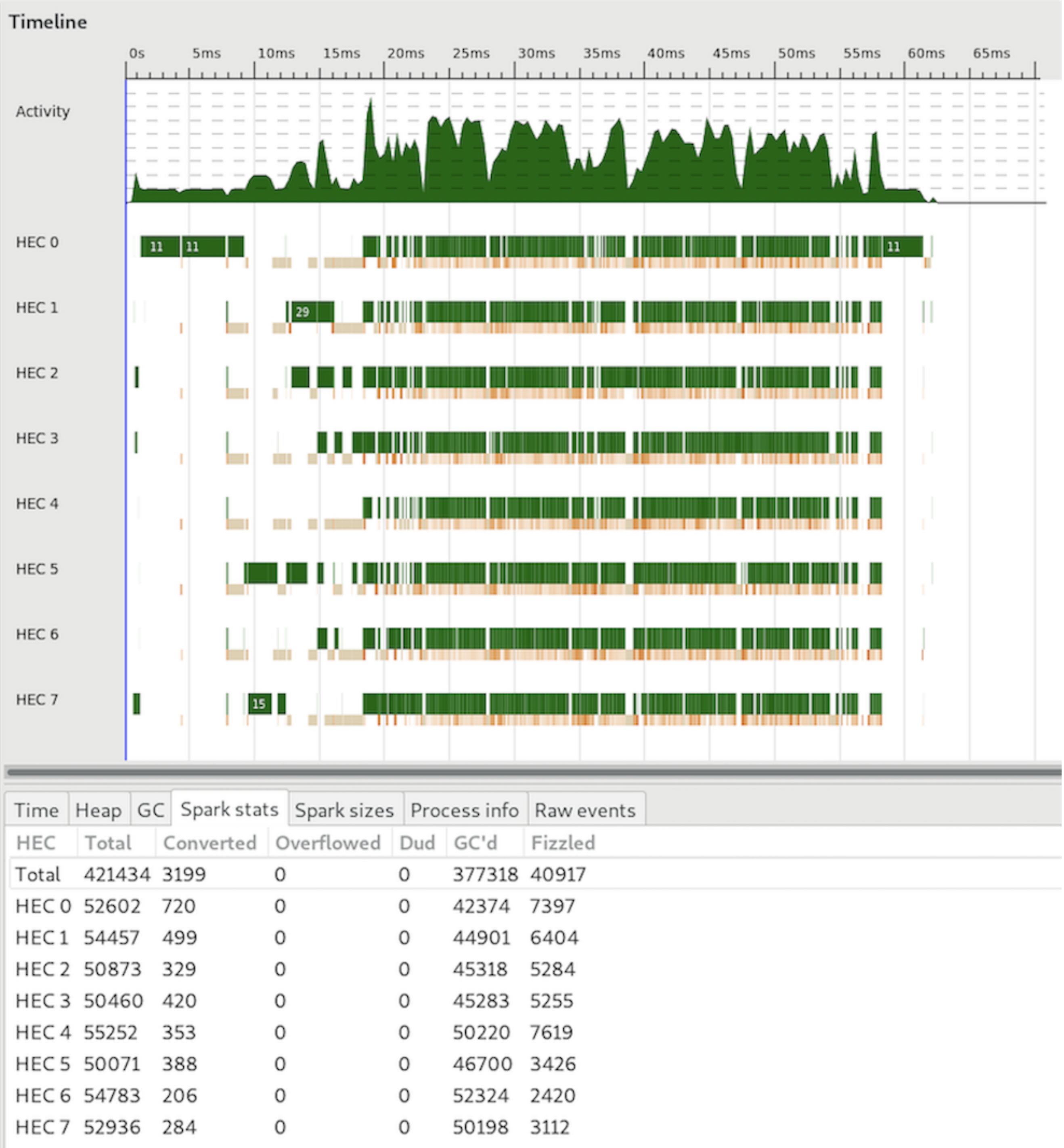
Algo Name	Num Cores	ChunkSize	Runtime (ms)	Improvement
LDF	1	-	490.7	-
LDFPar	8	512	392.1	1.25X
LDFUBVecPLPar	8	1024	171.4	2.86X
LDFChunkUBVecPLPar	8	2^{10}	97.94	5.03X

We discuss the results of SPSUBVecPLPar, LDFUBVecPLPar and LDFChunkUBVecPLPar in further detail.

SPSUBVecPLPar

SPSUBVecPLPar is a significant improvement over SPS and also the list based parallel implementations of SPS, that is SPSPLPar1, SPSPLPar2 and SPSPLPar3. This can be attributed to using much less memory due to the use of Unboxed Vectors and mutable vectors for helper functions.
We experiment with many different chunk sizes, and 1024 performs best, as can be seen in the below graph from criterion html output:

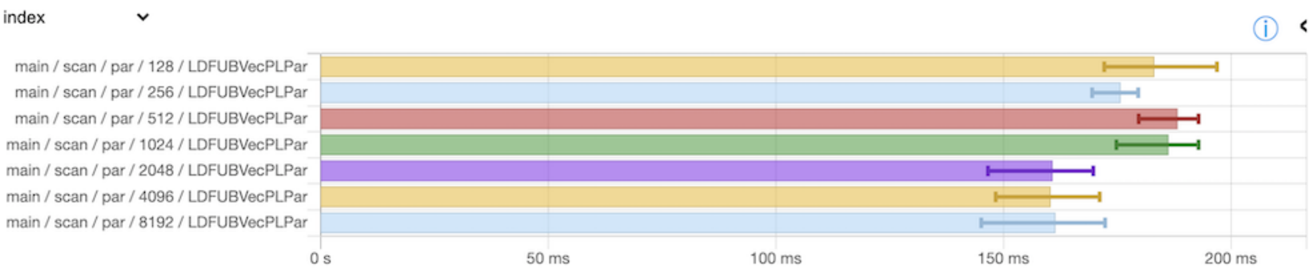
Here are the threadscope results, as we can see there are too many sparks generated:



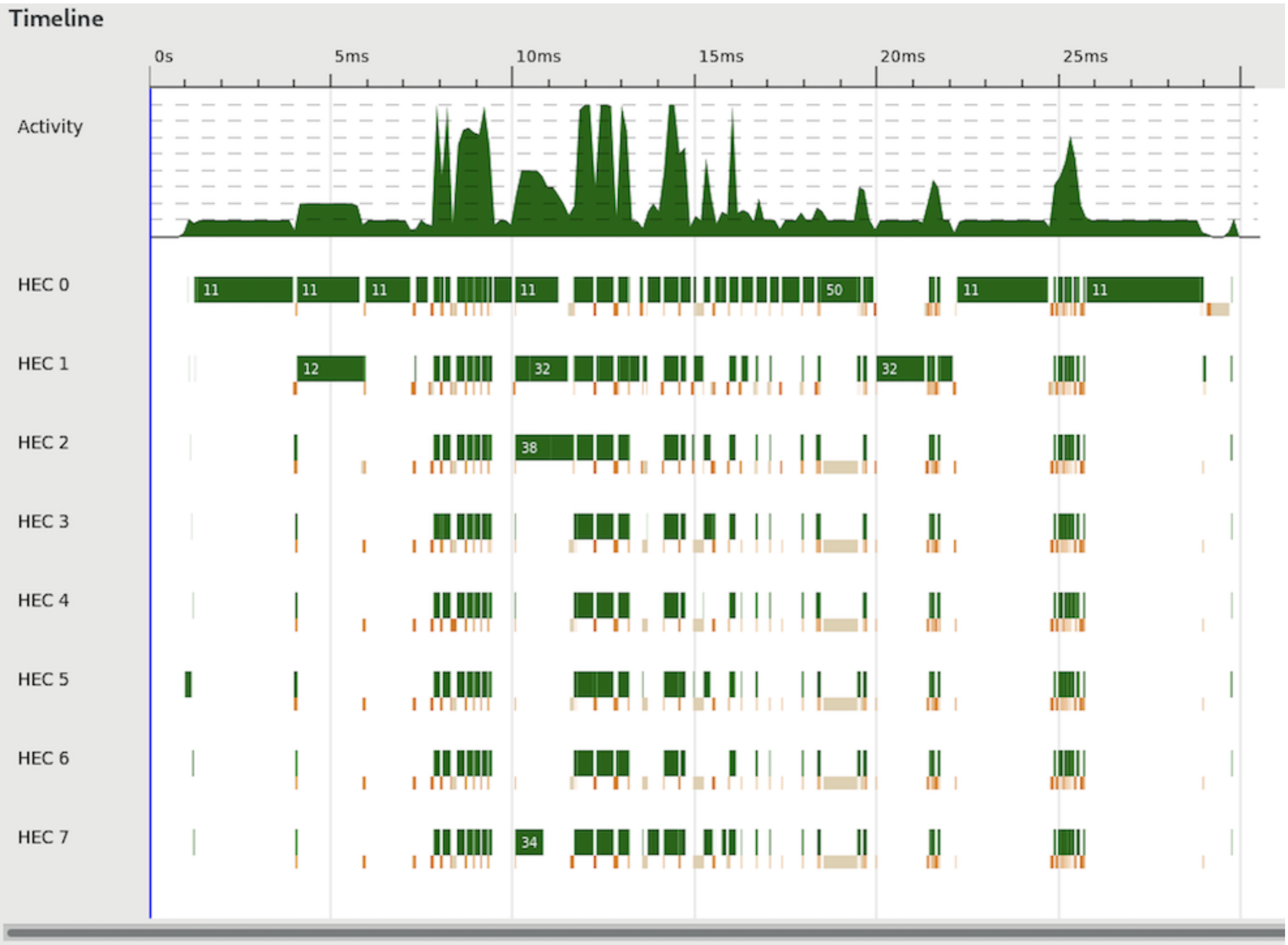
HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	421434	3199	0	0	377318	40917
HEC 0	52602	720	0	0	42374	7397
HEC 1	54457	499	0	0	44901	6404
HEC 2	50873	329	0	0	45318	5284
HEC 3	50460	420	0	0	45283	5255
HEC 4	55252	353	0	0	50220	7619
HEC 5	50071	388	0	0	46700	3426
HEC 6	54783	206	0	0	52324	2420
HEC 7	52936	284	0	0	50198	3112

LDFUBVecPLPar

LDFUBVecPLPar performs even better, since the algorithm itself has better run time. As before, here is the variation with different chunk sizes, again 1024 works best:



Here are the threadscope results, load looks well balanced:

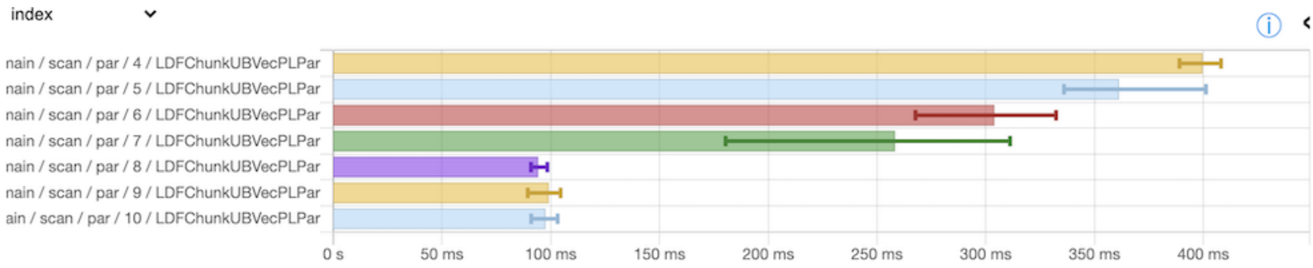


Time | Heap | GC | Spark stats | Spark sizes | Process info | Raw events

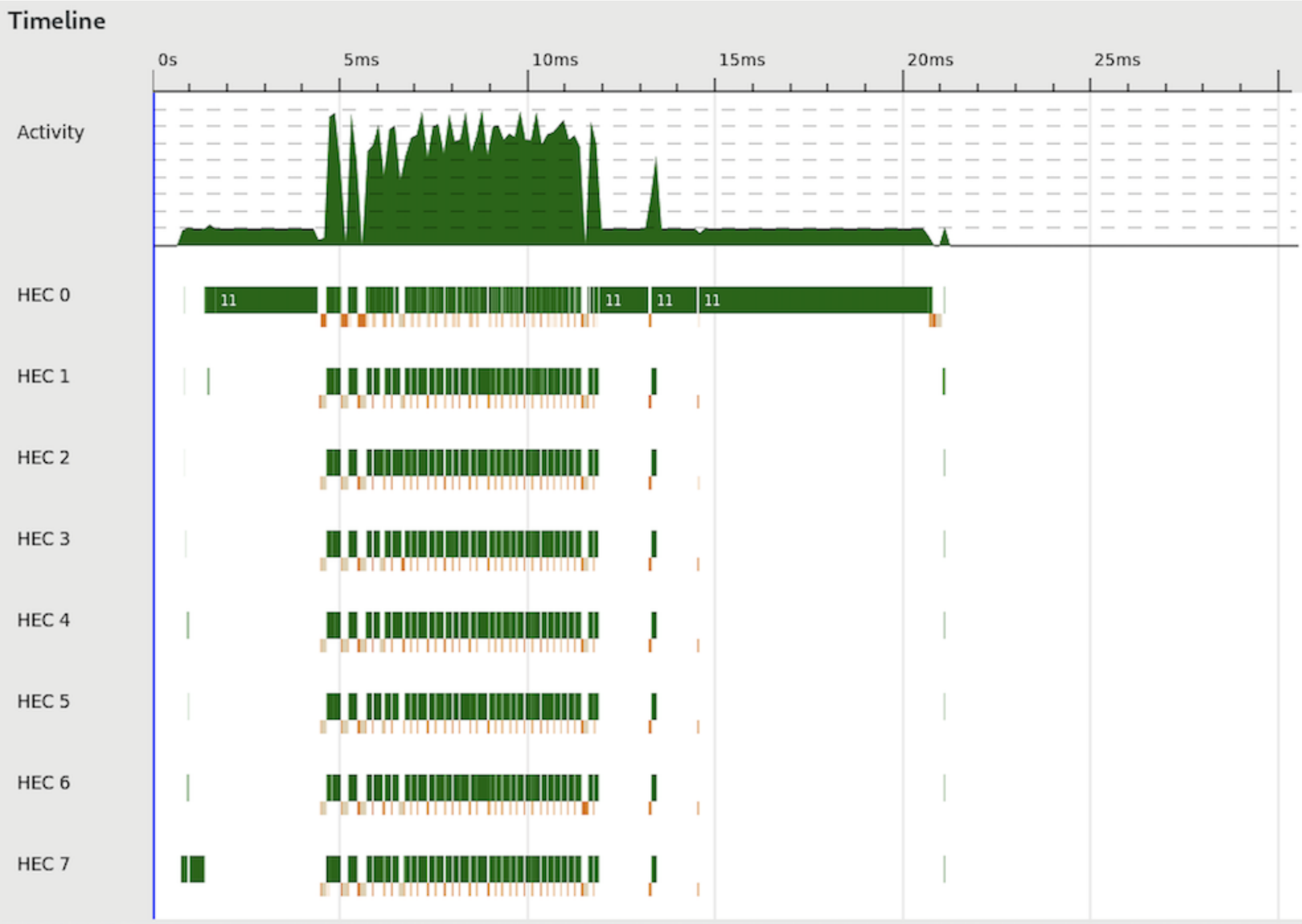
HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	4166	3675	0	0	473	18
HEC 0	2512	847	0	0	344	13
HEC 1	899	489	0	0	129	1
HEC 2	119	373	0	0	0	0
HEC 3	115	364	0	0	0	0
HEC 4	112	355	0	0	0	0
HEC 5	129	398	0	0	0	0
HEC 6	129	404	0	0	0	0
HEC 7	151	445	0	0	0	4

LDFChunkUBVecPLPar

LDFChunkUBVecPLPar performs the best with large enough chunk size, as we split the input from the top. Note that here chunksize is equal to 2^x where x is the number shown in the graph below:



Here are the threadscope results:



Time | Heap | GC | Spark stats | Spark sizes | Process info | Raw events

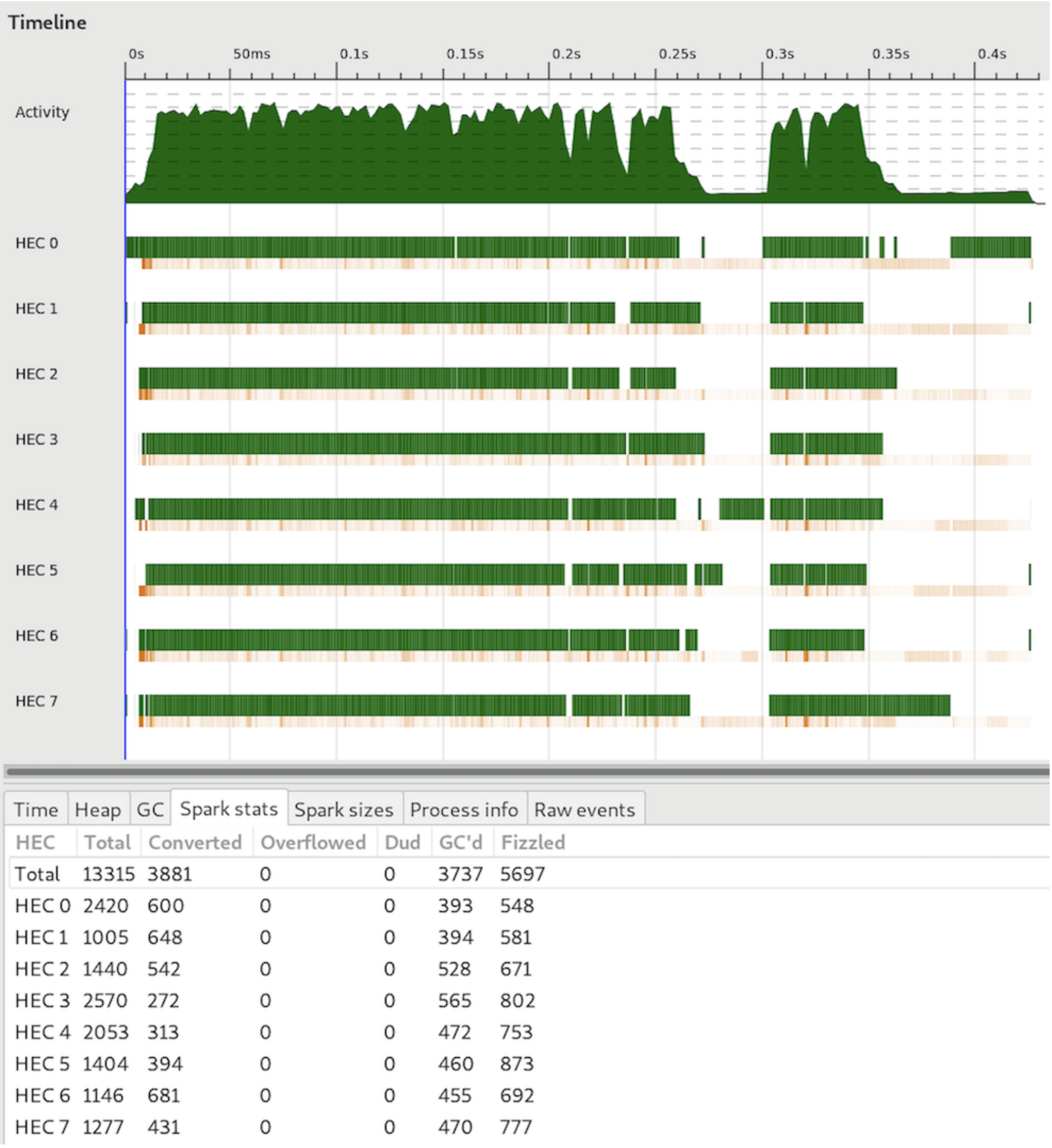
HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	37024	1076	0	0	6293	29655
HEC 0	5368	41	0	0	3237	23
HEC 1	4604	160	0	0	336	4544
HEC 2	4616	153	0	0	442	4317
HEC 3	4752	154	0	0	428	4451
HEC 4	4700	151	0	0	463	4494
HEC 5	4636	149	0	0	461	4190
HEC 6	4732	154	0	0	456	4402
HEC 7	3616	114	0	0	470	3234

Due to the linear implementation of processing auxiliary chunk, we see parallelism limited to part of the run. We might be able to further improve the run time, if the last phase is implemented in parallel over single mutable vector.

3.2.2 Sort

Algo Name	Num Cores	Runtime (ms)	Improvement
BATCHER	1	3929	-
BATCHER	8	1721	2.28X

The batcher sort, even though being a highly parallel algorithm, does not perform that well compared to a more traditional sort like quicksort. The obvious reason I could think of was because of all the copying and merging of intermediate arrays that is needed during the merge phase. Here is the threadscope analysis that shows how well batcher sort parallelizes:



HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	13315	3881	0	0	3737	5697
HEC 0	2420	600	0	0	393	548
HEC 1	1005	648	0	0	394	581
HEC 2	1440	542	0	0	528	671
HEC 3	2570	272	0	0	565	802
HEC 4	2053	313	0	0	472	753
HEC 5	1404	394	0	0	460	873
HEC 6	1146	681	0	0	455	692
HEC 7	1277	431	0	0	470	777

4 Conclusion and Future Work

To summarize, we can say that powerlist provides a new abstraction to come up with recursive and parallel algorithms for several different use cases. The algorithms using powerlist parallelize very well, as can be seen from threadscope captures. It is a challenge though to scale these parallel algorithms since they are recursive in nature, which inherently requires splitting and merging of input, thereby needing more memory. Simple iterative implementations of scan are difficult to beat with such algorithms.

We have several possibilities to extend this work.

- Explore other powerlist application algorithms like FFT.
- Exploit the commutative laws of scalar functions over powerlist operators to further parallelize the implemented algorithms.
- Try to use parallel libraries like massive that support nested parallelism.
- Experiment with RTS GC settings.

References

[1] J. Misra, "Powerlist: A structure for parallel recursion," *ACM Trans. Program. Lang. Syst.*, vol. 16, p. 1737–1767, nov 1994.

[2] G. E. Blelloch, "Prefix sums and their applications," 1990.

Appendices

A Code Listing

app/Main.hs

```haskell
module Main where

import CLParser
  ( Command(Scan, Sort)
  , Opts(Opts)
  , ScanAlgo(LDF, LDFChunkUBVecPLPar, LDFPar, LDFUBVecPLPar, SPS,
        SPSPL, SPSPLPar1, SPSPLPar2, SPSPLPar3, SPSUBVecPLPar)
  , SortAlgo(BATCHER, DEFAULT)
  , parseArgs
  )
import Scan
  ( ldf
  , parSps1
  , runParLdf
  , runParLdfChunkUBVec
  , runParLdfUBVec
  , runParScan2
  , runParScan3
  , runParSpsUBVec
  , runScan
  , sequentialSPS
  , sps
  )
import Sort (runBatcherSort, runDefaultSort)

main :: IO ()
main = run =<< parseArgs

run :: Opts -> IO ()
run opts =
  case opts
    -- Run parallel prefix sum with powerlist
        of
    Opts (Scan SPSPLPar1 n _) -> putStrLn $ runScan parSps1 n
    Opts (Scan SPSPLPar2 n c) -> putStrLn $ runParScan2 c n
    Opts (Scan SPSPLPar3 n c) -> putStrLn $ runParScan3 c n
    -- Run prefix sum via ldf algo (sequential)
    Opts (Scan LDF n _) -> putStrLn $ runScan ldf n
    -- Run parallel prefix sum via ldf algo
    Opts (Scan LDFPar n c) -> putStrLn $ runParLdf c n
    -- Run sequential prefix sum without powerlist
    Opts (Scan SPS n _) -> putStrLn $ runScan sequentialSPS n
    -- Run sequential prefix sum with powerlist
    Opts (Scan SPSPL n _) -> putStrLn $ runScan sps n
    -- Run parallel prefix sum using unboxed vecpowerlist
    Opts (Scan SPSUBVecPLPar n cs) -> putStrLn $ runParSpsUBVec cs n
    Opts (Scan LDFUBVecPLPar n cs) -> putStrLn $ runParLdfUBVec cs n
    Opts (Scan LDFChunkUBVecPLPar n cs) -> putStrLn $ runParLdfChunkUBVec cs n
    -- Run sort
```

```
  Opts (Sort DEFAULT n cs) -> putStrLn $ runDefaultSort cs n
  Opts (Sort BATCHER n cs) -> putStrLn $ runBatcherSort cs n
```

bench/Main.hs

```
module Main where

import Utils
  ( generateList
  , generateReverseList
  , generateReverseUVec
  , generateUVec
  )

import Criterion.Main (bench, bgroup, env, nf, defaultConfig, defaultMainWith)

import Control.DeepSeq (force)

import Scan
  ( ldf
  , parLdf
  , parLdfChunkUBVec
  , parLdfUBVec
  , parSps1
  , parSps2
  , parSps3
  , parSpsUBVec
  , sequentialSPS
  , sps
  )
import Sort (defaultSort, parBatcherMergeSort)

import qualified Data.Vector.Unboxed as V
import Criterion.Types (Config, resamples)

baseConfig :: Config
baseConfig = defaultConfig {
           resamples = 20
       }

setUpEnv :: IO (V.Vector Int, [Int], V.Vector Int, [Int])
setUpEnv = do
  let scanInpUV = force generateUVec 20
  let scanInpL = force generateList 20
  let sortInpUV = force generateReverseUVec 20
  let sortInpL = force generateReverseList 20
  return (scanInpUV, scanInpL, sortInpUV, sortInpL)

main :: IO ()
main =
  defaultMainWith baseConfig
    [ env setUpEnv $ \ ~(scanInpUV, scanInpL, sortInpUV, sortInpL) ->
        bgroup
          "main"
```

```
        [ bgroup
            "scan"
            [ bgroup
                "par"
                [ bgroup "nc" [bench "SPSPLPar1" $ nf (parSps1 (+)) scanInpL]
                , bgroup
                    "4"
                    [ bench "LDFChunkUBVecPLPar" $
                      nf (parLdfChunkUBVec (+) 4) scanInpUV
                    ]
                , bgroup
                    "5"
                    [ bench "LDFChunkUBVecPLPar" $
                      nf (parLdfChunkUBVec (+) 5) scanInpUV
                    ]
                , bgroup
                    "6"
                    [ bench "LDFChunkUBVecPLPar" $
                      nf (parLdfChunkUBVec (+) 6) scanInpUV
                    ]
                , bgroup
                    "7"
                    [ bench "LDFChunkUBVecPLPar" $
                      nf (parLdfChunkUBVec (+) 7) scanInpUV
                    ]
                , bgroup
                    "8"
                    [ bench "LDFChunkUBVecPLPar" $
                      nf (parLdfChunkUBVec (+) 8) scanInpUV
                    ]
                , bgroup
                    "9"
                    [ bench "LDFChunkUBVecPLPar" $
                      nf (parLdfChunkUBVec (+) 9) scanInpUV
                    ]
                , bgroup
                    "10"
                    [ bench "LDFChunkUBVecPLPar" $
                      nf (parLdfChunkUBVec (+) 10) scanInpUV
                    ]
                , bgroup
                    "128"
                    [ bench "SPSPLPar2" $ nf (parSps2 (+) 128) scanInpL
                    , bench "SPSPLPar3" $ nf (parSps3 (+) 128 20) scanInpL
                    , bench "LDFPar" $ nf (parLdf (+) 128 20) scanInpL
                    , bench "LDFUBVecPLPar" $
                      nf (parLdfUBVec (+) 128 20) scanInpUV
                    , bench "SPSUBVecPLPar" $
                      nf (parSpsUBVec (+) 128 20) scanInpUV
                    ]
                , bgroup
                    "256"
                    [ bench "SPSPLPar2" $ nf (parSps2 (+) 256) scanInpL
                    , bench "SPSPLPar3" $ nf (parSps3 (+) 256 20) scanInpL
                    , bench "LDFPar" $ nf (parLdf (+) 256 20) scanInpL
```

```
                    , bench "LDFUBVecPLPar" $
                      nf (parLdfUBVec (+) 256 20) scanInpUV
                    , bench "SPSUBVecPLPar" $
                      nf (parSpsUBVec (+) 256 20) scanInpUV
                    ]
                , bgroup
                    "512"
                    [ bench "SPSPLPar2" $ nf (parSps2 (+) 512) scanInpL
                    , bench "SPSPLPar3" $ nf (parSps3 (+) 512 20) scanInpL
                    , bench "LDFPar" $ nf (parLdf (+) 512 20) scanInpL
                    , bench "LDFUBVecPLPar" $
                      nf (parLdfUBVec (+) 512 20) scanInpUV
                    , bench "SPSUBVecPLPar" $
                      nf (parSpsUBVec (+) 512 20) scanInpUV
                    ]
                , bgroup
                    "1024"
                    [ bench "LDFUBVecPLPar" $
                      nf (parLdfUBVec (+) 1024 20) scanInpUV
                    , bench "SPSUBVecPLPar" $
                      nf (parSpsUBVec (+) 1024 20) scanInpUV
                    ]
                , bgroup
                    "2048"
                    [ bench "LDFUBVecPLPar" $
                      nf (parLdfUBVec (+) 2048 20) scanInpUV
                    , bench "SPSUBVecPLPar" $
                      nf (parSpsUBVec (+) 2048 20) scanInpUV
                    ]
                , bgroup
                    "4096"
                    [ bench "LDFUBVecPLPar" $
                      nf (parLdfUBVec (+) 4096 20) scanInpUV
                    , bench "SPSUBVecPLPar" $
                      nf (parSpsUBVec (+) 4096 20) scanInpUV
                    ]
                , bgroup
                    "8192"
                    [ bench "LDFUBVecPLPar" $
                      nf (parLdfUBVec (+) 8192 20) scanInpUV
                    , bench "SPSUBVecPLPar" $
                      nf (parSpsUBVec (+) 8192 20) scanInpUV
                    ]
                ]
            , bgroup
                "seq"
                [ bench "LDF" $ nf (ldf (+)) scanInpL
                , bench "SPSPL" $ nf (sps (+)) scanInpL
                , bench "SPS" $ nf (sequentialSPS (+)) scanInpL
                ]
            ]
        , bgroup
            "sort"
            [ bench "BATCHER" $ nf (parBatcherMergeSort 20) sortInpUV
            , bench "DEFAULT" $ nf defaultSort sortInpL
```

```
            ]
        ]
    ]
```

src/CLParser.hs

```haskell
{-# LANGUAGE RecordWildCards #-}

module CLParser where

import Options.Applicative
  ( CommandFields
  , Mod
  , Parser
  , ReadM
  , auto
  , command
  , customExecParser
  , eitherReader
  , fullDesc
  , header
  , help
  , helper
  , hsubparser
  , info
  , long
  , metavar
  , option
  , prefs
  , progDesc
  , short
  , showHelpOnEmpty
  , value
  )

data RunType
  = Sequential
  | Parallel

data ScanAlgo
  = SPS
  | SPSPL
  | SPSPLPar1
  | SPSPLPar2
  | SPSPLPar3
  | LDF
  | LDFPar
  | SPSUBVecPLPar
  | LDFUBVecPLPar
  | LDFChunkUBVecPLPar
  deriving (Show, Enum)

data SortAlgo
  = DEFAULT
```

```
  | BATCHER
  deriving (Show, Enum)

newtype Opts =
  Opts
    { cmd :: Command
    }

-- Add more features here in the future
data Command
  = Scan ScanAlgo Int Int
  | Sort SortAlgo Int Int

parser :: Parser Opts
parser = Opts <$> hsubparser (scanCommand <> sortCommand)
  where
    sortCommand :: Mod CommandFields Command
    sortCommand =
      command "sort" (info sortOptions (progDesc "Run Sort Algorithm"))
    sortOptions :: Parser Command
    sortOptions =
      Sort <$>
      option
        sortAlgoReader
        (long "algo" <>
         short 'a' <>
         metavar "ALGONAME" <> help ("Supported Algos: " ++ show [DEFAULT ..])) <*>
      option
        auto
        (long "size" <>
         short 's' <>
         metavar "INPSIZE" <>
         help "Size of array in terms of powers of 2 on which to run sort") <*>
      option
        auto
        (long "csize" <>
         short 'c' <>
         metavar "CHUNKSIZE" <>
         value 64 <> help "Size of chunks for parallelization")
    sortAlgoReader :: ReadM SortAlgo
    sortAlgoReader =
      eitherReader $ \arg ->
        case arg of
          "DEFAULT" -> Right (DEFAULT)
          "BATCHER" -> Right (BATCHER)
          _ -> Left ("Invalid Algo")
    scanCommand :: Mod CommandFields Command
    scanCommand =
      command "scan" (info scanOptions (progDesc "Run Scan Algorithm"))
    scanOptions :: Parser Command
    scanOptions =
      Scan <$>
      option
        scanAlgoReader
        (long "algo" <>
```

```
        short 'a' <>
        metavar "ALGONAME" <> help ("Supported Algos: " ++ show [SPS ..])) <*>
      option
        auto
        (long "size" <>
         short 's' <>
         metavar "INPSIZE" <>
         help "Size of array in terms of powers of 2 on which to run scan") <*>
      option
        auto
        (long "csize" <>
         short 'c' <>
         metavar "CHUNKSIZE" <>
         value 64 <> help "Size of chunks for parallelization")
    scanAlgoReader :: ReadM ScanAlgo
    scanAlgoReader =
      eitherReader $ \arg ->
        case arg of
          "SPS" -> Right (SPS)
          "SPSPL" -> Right (SPSPL)
          "SPSPLPar1" -> Right (SPSPLPar1)
          "SPSPLPar2" -> Right (SPSPLPar2)
          "SPSPLPar3" -> Right (SPSPLPar3)
          "LDF" -> Right (LDF)
          "LDFPar" -> Right (LDFPar)
          "SPSUBVecPLPar" -> Right (SPSUBVecPLPar)
          "LDFUBVecPLPar" -> Right (LDFUBVecPLPar)
          "LDFChunkUBVecPLPar" -> Right (LDFChunkUBVecPLPar)
          _ -> Left ("Invalid Algo")

parseArgs :: IO Opts
parseArgs =
  customExecParser (prefs showHelpOnEmpty) $
  info
    (helper <*> parser)
    (fullDesc <>
     progDesc "powerlist" <>
     header "A program to run algorithms using powerlist abstraction")
```

src/Powerlist.hs

```
module Powerlist where

import Control.Parallel.Strategies

-- Using simple list here as it would be most performant
type PowerList a = [a]

tie :: PowerList a -> PowerList a -> PowerList a
{-# INLINE tie #-}
tie = (++)

zip :: PowerList a -> PowerList a -> PowerList a
{-# INLINE zip #-}
```

```
zip [] [] = []
zip xs ys = Prelude.zip xs ys >>= \(a, b) -> [a, b]

parZip :: Strategy a -> Int -> PowerList a -> PowerList a -> PowerList a
{-# INLINE parZip #-}
parZip strategy cs as bs = Powerlist.zip as bs `using` parListChunk cs strategy

zipWith :: Num a => (a -> a -> a) -> PowerList a -> PowerList a -> PowerList a
{-# INLINE zipWith #-}
zipWith = Prelude.zipWith

parZipWith :: Num a => Strategy a -> Int -> (a -> a -> a) -> [a] -> [a] -> [a]
{-# INLINE parZipWith #-}
parZipWith strategy cs z as bs =
  Powerlist.zipWith z as bs `using` parListChunk cs strategy

unzip :: PowerList a -> (PowerList a, PowerList a)
unzip =
  snd .
  foldr
    (\x (b, (xs, ys)) ->
       ( not b
       , if b
           then (x : xs, ys)
           else (xs, x : ys)))
    (False, ([], []))

{-
unzip = Prelude.unzip . splt
  where splt []      = []
        splt (x:y:xs) = (x, y) : splt xs
        splt _        = error "Malformed powerlist"
-}
-- Right shift and use zero
rsh :: a -> PowerList a -> PowerList a
{-# INLINE rsh #-}
rsh zero xs = zero : init xs
```

src/Scan.hs

```
module Scan where

import Control.DeepSeq (NFData)
import Control.Parallel.Strategies
  ( Eval
  , parList
  , r0
  , rdeepseq
  , rpar
  , rparWith
  , rseq
  , runEval
  )
import Utils (generateList, generateUVec)
```

```
import qualified Data.Vector.Split as S
import qualified Data.Vector.Unboxed as UV
import qualified Data.Vector.Unboxed.Mutable as M
import qualified Powerlist as P
import qualified UBVecPowerlist as UVP

--------------------------------------------------------------------------------
-- Sequential SPS is nothing but haskel's scanl1
--------------------------------------------------------------------------------
sequentialSPS :: (a -> a -> a) -> [a] -> [a]
sequentialSPS = scanl1

--------------------------------------------------------------------------------
-- Sequential SPS using powerlist, works for lists with power of 2 length
--------------------------------------------------------------------------------
sps :: Num a => (a -> a -> a) -> P.PowerList a -> P.PowerList a
sps _ [] = []
sps _ [x] = [x]
sps op l = P.zip (sps op u) (sps op v)
  where
    (u, v) = P.unzip $ P.zipWith op (P.rsh 0 l) l

--------------------------------------------------------------------------------
-- Parallel SPS Version1 using powerlists
--------------------------------------------------------------------------------
parSps1 :: (NFData a, Num a) => (a -> a -> a) -> P.PowerList a -> P.PowerList a
parSps1 _ [] = []
parSps1 _ [x] = [x]
parSps1 op l =
  runEval
    (do (u, v) <- rseq $ P.unzip $ P.zipWith op (P.rsh 0 l) l
        u' <- rparWith rdeepseq (parSps1 op u)
        v' <- rparWith rdeepseq (parSps1 op v)
        rseq $ P.zip u' v')

runScan ::
     ((Int -> Int -> Int) -> P.PowerList Int -> P.PowerList Int)
  -> Int
  -> String
runScan f inp = show $ sum $ f (+) $ generateList inp

odds :: [a] -> [a]
odds [] = []
odds [x] = [x]
odds (x:_:xs) = x : odds xs

evens :: [a] -> [a]
evens [] = []
evens [_] = []
evens (_:y:xs) = y : evens xs

parSps2 ::
     NFData a
  => Num a =>
```

```
      (a -> a -> a) -> Int -> P.PowerList a -> P.PowerList a
parSps2 _ _ [] = []
parSps2 _ _ [x] = [x]
parSps2 op cs l =
  runEval
    (do k <- r0 $ P.parZipWith rdeepseq cs op (0 : l) l
        u <- rpar (odds k)
        v <- rpar (evens k)
        _ <- rseq u
        u' <- rparWith rdeepseq (parSps2 op cs u)
        _ <- rseq v
        v' <- rparWith rdeepseq (parSps2 op cs v)
        rseq $ P.zip u' v')

{-
  Parallel till certain depth, for arrays of size <= 2^4, use sequentialSPS
-}
parSps3 ::
     NFData a
  => Num a =>
      (a -> a -> a) -> Int -> Int -> P.PowerList a -> P.PowerList a
parSps3 _ _ _ [] = []
parSps3 _ _ _ [x] = [x]
parSps3 op cs d l
  | d > 4 =
    runEval
      (do k <- r0 $ P.parZipWith rdeepseq cs op (0 : l) l
          u <- rpar (odds k)
          v <- rpar (evens k)
          _ <- rseq u
          u' <- rparWith rdeepseq (parSps3 op cs (d - 1) u)
          _ <- rseq v
          v' <- rparWith rdeepseq (parSps3 op cs (d - 1) v)
          rseq $ P.zip u' v')
parSps3 op _ _ l = sequentialSPS op l

runParScan2 :: Int -> Int -> String
runParScan2 cs inp = show $ sum $ parSps2 (+) cs $ generateList inp

runParScan3 :: Int -> Int -> String
runParScan3 cs inp = show $ sum $ parSps3 (+) cs inp $ generateList inp

runParLdf :: Int -> Int -> String
runParLdf cs inp = show $ sum $ parLdf (+) cs inp $ generateList inp

runParSpsUBVec :: Int -> Int -> String
runParSpsUBVec cs inp =
  show $ UV.sum $ parSpsUBVec (+) cs inp $ generateUVec inp

runParLdfUBVec :: Int -> Int -> String
runParLdfUBVec cs inp =
  show $ UV.sum $ parLdfUBVec (+) cs inp $ generateUVec inp

runParLdfChunkUBVec :: Int -> Int -> String
runParLdfChunkUBVec cs inp =
```

```haskell
  show $ UV.sum $ parLdfChunkUBVec (+) cs $ generateUVec inp

--------------------------------------------------------------------------------
-- Ladner Fischer Algorithm
--------------------------------------------------------------------------------
ldf :: Num a => (a -> a -> a) -> P.PowerList a -> P.PowerList a
ldf _ [] = []
ldf _ [x] = [x]
ldf op l = P.zip (P.zipWith op (P.rsh 0 t) p) t
  where
    (p, q) = P.unzip l
    pq = P.zipWith op p q
    t = ldf op pq

{-
  A parallel version of LDF
-}
parLdf ::
     NFData a
  => Num a =>
       (a -> a -> a) -> Int -> Int -> P.PowerList a -> P.PowerList a
parLdf _ _ _ [] = []
parLdf _ _ _ [x] = [x]
parLdf op cs d l
  | d > 4 =
    runEval
      (do p <- rpar (odds l)
          q <- rpar (evens l)
          _ <- rseq p
          _ <- rseq q
          pq <- rseq (P.parZipWith rdeepseq cs op p q)
          t <- rparWith rdeepseq (parLdf op cs (d - 1) pq)
          k <- rseq (P.parZipWith rdeepseq cs op (0 : t) p)
          rseq $ P.zip k t)
parLdf op _ _ l = sequentialSPS op l

--------------------------------------------------------------------------------
-- SPS and LDF using powerlist unboxed vector implementation
--------------------------------------------------------------------------------
parSpsUBVec ::
     (NFData a, UV.Unbox a, Num a)
  => (a -> a -> a)
  -> Int
  -> Int
  -> UVP.PowerList a
  -> UVP.PowerList a
parSpsUBVec _ _ _ l
  | UV.length l <= 1 = l
parSpsUBVec op cs d l
  | d > 4 =
    runEval
      (do k <- rseq $ UVP.shiftAdd l
          u <- rpar (UVP.filterOdd k)
          v <- rpar (UVP.filterEven k)
          _ <- rseq u
```

```haskell
        u' <- rparWith rdeepseq (parSpsUBVec op cs (d - 1) u)
        _ <- rseq v
        v' <- rparWith rdeepseq (parSpsUBVec op cs (d - 1) v)
        UVP.parZip (rparWith rdeepseq) cs u' v')
parSpsUBVec op _ _ l = UV.scanl1 op l

parLdfUBVec ::
     (NFData a, UV.Unbox a, Num a)
  => (a -> a -> a)
  -> Int
  -> Int
  -> UVP.PowerList a
  -> UVP.PowerList a
parLdfUBVec _ _ _ l
  | UV.length l <= 1 = l
parLdfUBVec op cs d l
  | d > 4 =
    runEval
      (do p <- rpar $ UVP.filterOdd l
          q <- rpar $ UVP.filterEven l
          _ <- rseq p
          _ <- rseq q
          pq <- UVP.parZipWith (rparWith rdeepseq) op cs p q
          t <- rpar (parLdfUBVec op cs (d - 1) pq)
          k <- rseq $ UVP.shiftAdd2 t p
          UVP.parZip (rparWith rdeepseq) cs k t)
parLdfUBVec op _ _ l = UV.scanl1 op l

--------------------------------------------------------------------------------
-- Chunk the input itself, hybrid of ldf and Blelloch
--------------------------------------------------------------------------------
parLdfUBVecNC ::
     (NFData a, UV.Unbox a, Num a)
  => (a -> a -> a)
  -> Int
  -> UVP.PowerList a
  -> UVP.PowerList a
parLdfUBVecNC _ _ l
  | UV.length l <= 1 = l
parLdfUBVecNC op d l
  | d > 2 =
    runEval
      (do p <- rpar $ UVP.filterOdd l
          q <- rpar $ UVP.filterEven l
          _ <- rseq p
          _ <- rseq q
          pq <- rparWith rdeepseq $ UVP.zipWith op p q
          t <- rpar (parLdfUBVecNC op (d - 1) pq)
          k <- rseq $ UVP.shiftAdd2 t p
          rseq $ UVP.zip k t)
parLdfUBVecNC op _ l = UV.scanl1 op l

parLdfChunkUBVec ::
     (NFData a, UV.Unbox a, Num a)
  => (a -> a -> a)
```

```haskell
  -> Int
  -> UVP.PowerList a
  -> UVP.PowerList a
parLdfChunkUBVec _ _ l
  | UV.length l <= 1 = l
parLdfChunkUBVec ops cs l = runEval $ parLdfChunkVec' ops chunks
  where
    n = UV.length l
    chunkSize = 2 ^ cs
    chunks = S.chunksOf chunkSize l
    parLdfChunkVec' ::
         (NFData a, UV.Unbox a, Num a)
      => (a -> a -> a)
      -> [UVP.PowerList a]
      -> Eval (UVP.PowerList a)
    parLdfChunkVec' _ [] = return UV.empty
    parLdfChunkVec' op vChunks = do
      resChunks <- parList rseq (parLdfUBVecNC op cs <$> vChunks)
      res <- rseq $ UV.concat resChunks
          -- Get last element of each block
      lastelems <- parList rdeepseq (UV.last <$> resChunks)
      lastScan <- rseq (UV.fromList $ sequentialSPS op lastelems)
      rseq $
        UV.create $ do
          m <- UV.thaw res
              -- Not sure how to parallelise here!
          mergeChunks (n - 1) (UV.tail $ UV.reverse lastScan) m
          return m
    mergeChunks i lastScan m
      | i > chunkSize = do
        let ad = UV.head lastScan
        go m chunkSize i ad 0
        mergeChunks (i - chunkSize) (UV.tail lastScan) m
      | otherwise = return ()
    go m chs start v i
      | i < chs = do
        curr <- M.unsafeRead m (start - i)
        M.unsafeWrite m (start - i) (curr + v)
        go m chs start v (i + 1)
      | otherwise = return ()
```

src/Sort.hs

```haskell
module Sort where

import Control.DeepSeq (NFData)
import Control.Parallel.Strategies
  ( Eval
  , rdeepseq
  , rpar
  , rparWith
  , rseq
  , runEval
  )
```

```haskell
import Data.List (sort)
import Utils (generateReverseList, generateReverseUVec)

import qualified Data.Vector.Unboxed as V
import qualified Data.Vector.Unboxed.Mutable as M
import qualified UBVecPowerlist as P

runDefaultSort :: Int -> Int -> String
runDefaultSort _ inp = show $ last $ defaultSort $ generateReverseList inp

runBatcherSort :: Int -> Int -> String
runBatcherSort _ inp =
  show $ V.last $ parBatcherMergeSort inp $ generateReverseUVec inp

defaultSort :: Ord a => [a] -> [a]
defaultSort = sort

--------------------------------------------------------------------------------
-- Sequential Impl for demonstration
--------------------------------------------------------------------------------
batcherMergeSort :: (Ord a, V.Unbox a) => P.PowerList a -> P.PowerList a
batcherMergeSort l
  | V.length l <= 1 = l
batcherMergeSort l = sortp `batcherMerge` sortq
  where
    sortp = batcherMergeSort p
    sortq = batcherMergeSort q
    p = P.filterOdd l
    q = P.filterEven l

batcherMerge ::
     (Ord a, V.Unbox a) => P.PowerList a -> P.PowerList a -> P.PowerList a
batcherMerge x y
  | V.length x == 1 = V.fromList [hx `min` hy, hx `max` hy]
  where
    hx = V.head x
    hy = V.head y
batcherMerge x y = P.minMaxZip rv su
  where
    rv = r `batcherMerge` v
    su = s `batcherMerge` u
    r = P.filterOdd x
    v = P.filterEven y
    s = P.filterEven x
    u = P.filterOdd y

--------------------------------------------------------------------------------
-- Parallel Impl
--------------------------------------------------------------------------------
parBatcherMergeSort ::
     (NFData a, Ord a, V.Unbox a) => Int -> P.PowerList a -> P.PowerList a
parBatcherMergeSort _ l
  | V.length l <= 1 = l
parBatcherMergeSort d l
  | d > 10 =
```

```haskell
    runEval
      (do p <- rpar $ P.filterOdd l
          q <- rpar $ P.filterEven l
          _ <- rseq p
          sortp <- rparWith rdeepseq (parBatcherMergeSort (d - 1) p)
          _ <- rseq q
          sortq <- rparWith rdeepseq (parBatcherMergeSort (d - 1) q)
          parBatcherMerge d sortp sortq)
parBatcherMergeSort _ l = V.fromList $ defaultSort $ V.toList l

parBatcherMerge ::
     (Ord a, V.Unbox a)
  => Int
  -> P.PowerList a
  -> P.PowerList a
  -> Eval (P.PowerList a)
--batcherMerge strategy d cs x y | V.length x == 1 = rseq $ V.fromList [hx `min` hy, hx `max` hy]
--    where hx = V.head x
--          hy = V.head y
parBatcherMerge d x y
  | d > 10 = do
    r <- rseq $ P.filterOdd x
    v <- rseq $ P.filterEven y
    rv <- parBatcherMerge (d - 1) r v
    s <- rseq $ P.filterEven x
    u <- rseq $ P.filterOdd y
    su <- parBatcherMerge (d - 1) s u
    rparWith rdeepseq $ P.minMaxZip rv su
parBatcherMerge _ x y = rseq (merge x y)

merge :: (Ord a, V.Unbox a) => P.PowerList a -> P.PowerList a -> P.PowerList a
merge a b =
  V.create $ do
    v <- M.new nm
    go 0 0 v
    return v
  where
    n = V.length a
    m = V.length b
    nm = n + m
    go i j v
      | (i + j) < nm = do
        let ai = a V.! i
        let bj = b V.! j
        if (j == m) || (i < n && ai <= bj)
          then do
            M.unsafeWrite v (i + j) ai
            go (i + 1) j v
          else do
            M.unsafeWrite v (i + j) bj
            go i (j + 1) v
      | otherwise = return ()
```

src/UBVecPowerlist.hs

```haskell
{-# LANGUAGE FlexibleContexts #-}

module UBVecPowerlist where

import Control.Parallel.Strategies (Eval, Strategy, parList, rdeepseq, rseq)

import qualified Data.Vector.Split as S
import qualified Data.Vector.Unboxed as V
import qualified Data.Vector.Unboxed.Mutable as M

type PowerList a = V.Vector a

tie :: V.Unbox a => PowerList a -> PowerList a -> PowerList a
{-# INLINE tie #-}
tie = (V.++)

zip :: (V.Unbox a, Num a) => PowerList a -> PowerList a -> PowerList a
{-# INLINE zip #-}
--zip xs ys = V.generate (V.length xs + V.length ys) (\i -> if even i then xs V.! (i `div` 2) else ys
    V.! (i `div` 2))
zip xs ys =
  V.create $ do
    m <- M.new n
    write m 0
    return m
  where
    n = V.length xs + V.length ys
    write m i
      | i < n = do
        M.unsafeWrite m i (xs V.! (i `div` 2))
        M.unsafeWrite m (i + 1) (ys V.! (i `div` 2))
        write m (i + 2)
      | otherwise = return ()

parZip ::
     (V.Unbox a, Num a)
  => Strategy (PowerList a)
  -> Int
  -> PowerList a
  -> PowerList a
  -> Eval (PowerList a)
{-# INLINE parZip #-}
parZip strategy cs as bs = do
  inp <- rseq $ Prelude.zip ac bc
  lists <- parList strategy (writePar <$> inp)
  rdeepseq $ V.concat lists
  where
    ac = S.chunksOf cs as
    bc = S.chunksOf cs bs
    writePar (a, b) = UBVecPowerlist.zip a b

zipWith ::
     (Num a, V.Unbox a)
  => (a -> a -> a)
  -> PowerList a
```

```haskell
  -> PowerList a
  -> PowerList a
{-# INLINE zipWith #-}
zipWith op xs ys =
  V.create $ do
    m <- V.thaw xs
    write m ys 0
    return m
  where
    k = V.length xs
    write m y i
      | i < k = do
        curr <- M.unsafeRead m i
        M.unsafeWrite m i (op (y V.! i) curr)
        write m y (i + 1)
      | otherwise = return ()

parZipWith ::
     (Num a, V.Unbox a)
  => Strategy (PowerList a)
  -> (a -> a -> a)
  -> Int
  -> PowerList a
  -> PowerList a
  -> Eval (PowerList a)
{-# INLINE parZipWith #-}
parZipWith strategy op cs as bs = do
  inp <- rseq $ Prelude.zip ac bc
  lists <- parList strategy (writePar <$> inp)
  rdeepseq $ V.concat lists
  where
    ac = S.chunksOf cs as
    bc = S.chunksOf cs bs
    writePar (a, b) = UBVecPowerlist.zipWith op a b

unzip :: V.Unbox a => PowerList a -> (PowerList a, PowerList a)
unzip k = (b, c)
  where
    b = V.ifilter (\i _ -> even i) k
    c = V.ifilter (\i _ -> odd i) k

filterUsing :: V.Unbox a => (Int -> Int) -> PowerList a -> PowerList a
filterUsing op l =
  V.create $ do
    m <- M.new n
    write m 0
    return m
  where
    nl = V.length l
    n = nl `div` 2
    write m i
      | i < n = do
        M.unsafeWrite m i (l V.! op i)
        write m (i + 1)
      | otherwise = return ()
```

```haskell

calculateEvenInd :: Int -> Int
calculateEvenInd = (* 2)

calculateOddInd :: Num a => a -> a
calculateOddInd i = (i * 2) + 1

filterOdd :: V.Unbox a => PowerList a -> PowerList a
filterOdd = filterUsing calculateEvenInd

filterEven :: V.Unbox a => PowerList a -> PowerList a
filterEven = filterUsing calculateOddInd

-- Right shift and use zero, does not perform well as cons is O(n)
rsh :: V.Unbox a => a -> PowerList a -> PowerList a
{-# INLINE rsh #-}
rsh zero xs = V.cons zero $ V.init xs

shiftAdd :: (V.Unbox a, Num a) => PowerList a -> PowerList a
shiftAdd l =
  V.create $ do
    m <- V.thaw l
    go (V.length l - 1) m
    return m
  where
    go ind mv
      | ind > 0 = do
        prev <- M.unsafeRead mv (ind - 1)
        curr <- M.unsafeRead mv ind
        M.unsafeWrite mv ind (prev + curr)
        go (ind - 1) mv
      | otherwise = return ()

shiftAdd2 :: (V.Unbox a, Num a) => PowerList a -> PowerList a -> PowerList a
shiftAdd2 r l =
  V.create $ do
    m <- V.thaw l
    go (V.length l - 1) m
    return m
  where
    go ind mv
      | ind > 0 = do
        curr <- M.unsafeRead mv ind
        M.unsafeWrite mv ind ((r V.! (ind - 1)) + curr)
        go (ind - 1) mv
      | otherwise = return ()

addPairs :: (V.Unbox a, Num a) => PowerList a -> PowerList a
addPairs l =
  V.create $ do
    m <- M.new n
    addPairs' m 0
    return m
  where
    n = V.length l `div` 2
```

```
    addPairs' mv i
      | i < n = do
        M.unsafeWrite mv i (l V.! (2 * i) + (l V.! (2 * i + 1)))
        addPairs' mv (i + 1)
      | otherwise = return ()

minMaxZip :: (V.Unbox a, Ord a) => PowerList a -> PowerList a -> PowerList a
minMaxZip xs ys =
  V.create $ do
    m <- M.new n
    write m 0
    return m
  where
    n = V.length xs + V.length ys
    write mv i
      | i < n = do
        let p = xs V.! (i `div` 2)
        let q = ys V.! (i `div` 2)
        M.unsafeWrite mv i (p `min` q)
        M.unsafeWrite mv (i + 1) (p `max` q)
        write mv (i + 2)
      | otherwise = return ()

parMinMaxZip ::
     (V.Unbox a, Ord a)
  => Strategy (PowerList a)
  -> Int
  -> PowerList a
  -> PowerList a
  -> Eval (PowerList a)
{-# INLINE parMinMaxZip #-}
parMinMaxZip strategy cs as bs = do
  inp <- rseq $ Prelude.zip ac bc
  lists <- parList strategy (writePar <$> inp)
  rdeepseq $ V.concat lists
  where
    ac = S.chunksOf cs as
    bc = S.chunksOf cs bs
    writePar (a, b) = UBVecPowerlist.minMaxZip a b
```

src/Utils.hs

```
module Utils where

import qualified Data.Vector.Unboxed as V

{-
  Generates a list from 2^n to 1
-}
generateReverseList :: Int -> [Int]
generateReverseList n = [2 ^ n,2 ^ n - 1 .. 1]

{-
  Generates a list from 2^n to 1
```

```haskell
-}
generateList :: Int -> [Int]
generateList n = [1 .. 2 ^ n]

{-
  Generate Unboxed Vector with values from 1 to 2^n
-}
generateUVec :: Int -> V.Vector Int
generateUVec n = V.generate (2 ^ n) (+ 1)

{-
  Generate Unboxed Vector with values from 2^n to 1
-}
generateReverseUVec :: Int -> V.Vector Int
generateReverseUVec n = V.generate (2 ^ n) (\i -> 2 ^ n - i)
```

src/VecPowerlist.hs

```haskell
module VecPowerlist where

import qualified Data.Vector as V
import qualified Data.Vector.Mutable as M

-- Using simple list here as it would be most performant
type PowerList a = V.Vector a

tie :: PowerList a -> PowerList a -> PowerList a
{-# INLINE tie #-}
tie = (V.++)

zip :: PowerList a -> PowerList a -> PowerList a
{-# INLINE zip #-}
zip xs ys =
  V.generate
    (V.length xs + V.length ys)
    (\i ->
       if even i
         then xs V.! (i `div` 2)
         else ys V.! (i `div` 2))

--zip _ _ = error "Non similar powerlists"
zipWith :: Num a => (a -> a -> a) -> PowerList a -> PowerList a -> PowerList a
{-# INLINE zipWith #-}
zipWith = V.zipWith

unzip :: PowerList a -> (PowerList a, PowerList a)
unzip k = (b, c)
  where
    b = V.ifilter (\i _ -> even i) k
    c = V.ifilter (\i _ -> odd i) k

-- Right shift and use zero, does not perform well as cons is O(n)
rsh :: a -> PowerList a -> PowerList a
{-# INLINE rsh #-}
```

```
rsh zero xs = V.cons zero $ V.init xs

shiftAdd :: Num a => PowerList a -> PowerList a
shiftAdd l =
  V.create $ do
    m <- V.thaw l
    go (V.length l - 1) m
    return m
  where
    go i mv
      | i > 0 = do
        prev <- M.unsafeRead mv (i - 1)
        curr <- M.unsafeRead mv i
        M.unsafeWrite mv i (prev + curr)
        go (i - 1) mv
      | otherwise = return ()

shiftAdd2 :: Num a => PowerList a -> PowerList a -> PowerList a
shiftAdd2 r l =
  V.create $ do
    m <- V.thaw l
    go (V.length l - 1) m
    return m
  where
    go i mv
      | i > 0 = do
        curr <- M.unsafeRead mv i
        M.unsafeWrite mv i ((r V.! (i - 1)) + curr)
        go (i - 1) mv
      | otherwise = return ()

addPairs :: Num a => PowerList a -> PowerList a
addPairs l =
  V.create $ do
    m <- M.new n
    addPairs' m 0
    return m
  where
    n = V.length l `div` 2
    addPairs' mv i
      | i < n = do
        M.unsafeWrite mv i (l V.! (2 * i) + (l V.! (2 * i + 1)))
        addPairs' mv (i + 1)
      | otherwise = return ()
```

test/Spec.hs

```
import Test.Hspec

import Utils ( generateList, generateUVec, generateReverseUVec )

import qualified Scan
import qualified Data.Vector.Unboxed     as UV
import qualified Sort
```

```
import qualified Sort

main :: IO ()
main = hspec $ do
    describe "Scan.sps" $ do
        it "correctly calculates prefix sum" $ do
            Scan.sps (+) (generateList 2) `shouldBe` [1, 3, 6, 10]

    describe "Scan.parSps2" $ do
        it "correctly calculates prefix sum" $ do
            Scan.parSps2 (+) 2 (generateList 2) `shouldBe` [1, 3, 6, 10]

    describe "Scan.parSps3" $ do
        it "correctly calculates prefix sum" $ do
            Scan.parSps3 (+) 10 6 (generateList 6) `shouldBe` scanl1 (+) [1..2^6]

    describe "Scan.ldf" $ do
        it "correctly calculates prefix sum" $ do
            Scan.ldf (+) (generateList 6) `shouldBe` scanl1 (+) [1..2^6]

    describe "Scan.parldf" $ do
        it "correctly calculates prefix sum" $ do
            Scan.parLdf (+) 10 6 (generateList 6) `shouldBe` scanl1 (+) [1..2^6]

    describe "Scan.parSpsUBVec" $ do
        it "correctly calculates prefix sum" $ do
            Scan.parSpsUBVec (+) 10 6 (generateUVec 6) `shouldBe` UV.fromList (scanl1 (+) [1..2^6])

    describe "Scan.parLdfUBVec" $ do
        it "correctly calculates prefix sum" $ do
            Scan.parLdfUBVec (+) 10 6 (generateUVec 6) `shouldBe` UV.fromList (scanl1 (+) [1..2^6])

    describe "Scan.parLdfUBVecNC" $ do
        it "correctly calculates prefix sum" $ do
            Scan.parLdfUBVecNC (+) 6 (generateUVec 6) `shouldBe` UV.fromList (scanl1 (+) [1..2^6])

    describe "Scan.parLdfChunkUBVec" $ do
        it "correctly calculates prefix sum" $ do
            Scan.parLdfChunkUBVec (+) 2 (generateUVec 8) `shouldBe` UV.fromList (scanl1 (+) [1..2^8])

    describe "Sort.batcherMergeSort" $ do
        it "correctly sorts the input vector" $ do
            Sort.batcherMergeSort (generateReverseUVec 8) `shouldBe` UV.fromList [1..2^8]

    describe "Sort.parBatcherMergeSort" $ do
        it "correctly sorts the input vector" $ do
            Sort.parBatcherMergeSort 8 (generateReverseUVec 8) `shouldBe` UV.fromList [1..2^8]
```