



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	2
2	Project Description	2
2.1	Usage	3
2.2	Powerlist in Haskell	4
2.2.1	Powerlist as List	4
2.2.2	Powerlist as Unboxed Vector	5
2.3	Algorithms	6
2.3.1	Simple Prefix Sum	6
2.3.1.1	SPSPL	6
2.3.1.2	SPSPLPar1	7
2.3.1.3	SPSPLPar2	7
2.3.1.4	SPSPLPar3	7
2.3.1.5	SPSUBVecPLPar	8
2.3.2	Ladner Fischer	8
2.3.2.1	LDFPar	9
2.3.2.2	LDFUBVecPLPar	9
2.3.2.3	LDFChunkUBVecPLPar	10
2.3.3	Batcher Merge Sort	11
3	Benchmark	13
3.1	Setup	13
3.2	Results	14
3.2.1	Scan	14
3.2.2	Sort	18
4	Conclusion and Future Work	19
	Appendices	20
A	Code Listing	20

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 outputs 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:

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

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

```
1  zip :: PowerList a -> PowerList a -> PowerList a
2  {-# INLINE zip #-}
3  zip [] [] = []
4  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.

```
1  unzip :: PowerList a -> (PowerList a, PowerList a)
2  unzip =
3  snd .
4  foldr
5  (\x (b, (xs, ys)) ->
6  ( not b
7  , if b
8  then (x : xs, ys)
9  else (xs, x : ys)))
10 (False, ([], []))
```

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

```
1  -- Right shift and use zero
```

```

2 rsh :: a -> PowerList a -> PowerList a
3 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:

```

1 import qualified Data.Vector.Split as S
2 import qualified Data.Vector.Unboxed as V
3 import qualified Data.Vector.Unboxed.Mutable as M
4
5 type PowerList a = V.Vector a
6
7 tie :: V.Unbox a => PowerList a -> PowerList a -> PowerList a
8 tie = (V.++)
9
10 zip :: (V.Unbox a, Num a) => PowerList a -> PowerList a -> PowerList a
11 {-# INLINE zip #-}
12 zip xs ys =
13     V.create $ do
14         m <- M.new n
15         write m 0
16         return m
17     where
18         n = V.length xs + V.length ys
19         write m i
20             | i < n = do
21                 M.unsafeWrite m i (xs V.! (i `div` 2))
22                 M.unsafeWrite m (i + 1) (ys V.! (i `div` 2))
23                 write m (i + 2)
24             | otherwise = return ()
25
26 zipWith ::
27     (Num a, V.Unbox a)
28     => (a -> a -> a)
29     -> PowerList a
30     -> PowerList a
31     -> PowerList a
32     {-# INLINE zipWith #-}
33 zipWith op xs ys =
34     V.create $ do
35         m <- V.thaw xs
36         write m ys 0
37         return m
38     where
39         k = V.length xs
40         write m y i
41             | i < k = do
42                 curr <- M.unsafeRead m i
43                 M.unsafeWrite m i (op (y V.! i) curr)
44                 write m y (i + 1)
45             | otherwise = return ()
46

```

```

47 unzip :: V.Unbox a => PowerList a -> (PowerList a, PowerList a)
48 unzip k = (b, c)
49   where
50     b = V.ifilter (\i _ -> even i) k
51     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:

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

which translates beautifully into haskell code:

```

1 import qualified Powerlist as P
2
3 sps :: Num a => (a -> a -> a) -> P.PowerList a -> P.PowerList a
4 sps _ [] = []
5 sps _ [x] = [x]
6 sps op l = P.zip (sps op u) (sps op v)
7   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.

```

1 parSps1 :: (NFData a, Num a) => (a -> a -> a) -> P.PowerList a -> P.PowerList a
2 parSps1 _ [] = []
3 parSps1 _ [x] = [x]
4 parSps1 op l =
5   runEval
6     (do (u, v) <- rseq $ P.unzip $ P.zipWith op (P.rsh 0 l) l
7         u' <- rparWith rdeepseq (parSps1 op u)
8         v' <- rparWith rdeepseq (parSps1 op v)
9         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.

```

1 odds :: [a] -> [a]
2 odds [] = []
3 odds [x] = [x]
4 odds (x:_:xs) = x : odds xs
5
6 evens :: [a] -> [a]
7 evens [] = []
8 evens [_] = []
9 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.

```

1 import qualified UVecPowerlist as UVP
2
3 parSpsUBVec ::
4   (NFData a, UV.Unbox a, Num a)
5   => (a -> a -> a)
6   -> Int
7   -> Int
8   -> UVP.PowerList a
9   -> UVP.PowerList a
10 parSpsUBVec _ _ _ 1
11 | UV.length l <= 1 = 1
12 parSpsUBVec op cs d l
13 | d > 4 =
14   runEval
15     (do k <- rseq $ UVP.shiftAdd l
16         u <- rpar (UVP.filterOdd k)
17         v <- rpar (UVP.filterEven k)
18         _ <- rseq u
19         u' <- rparWith rdeepseq (parSpsUBVec op cs (d - 1) u)
20         _ <- rseq v
21         v' <- rparWith rdeepseq (parSpsUBVec op cs (d - 1) v)
22         UVP.parZip (rparWith rdeepseq) cs u' v')
23 parSpsUBVec op _ _ 1 = 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:

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

And here is the equivalent sequential implementation in haskell:

```

1 ldf :: Num a => (a -> a -> a) -> P.PowerList a -> P.PowerList a
2 ldf _ [] = []
3 ldf _ [x] = [x]
4 ldf op l = P.zip (P.zipWith op (P.rsh 0 t) p) t
5   where
6     (p, q) = P.unzip l
7     pq = P.zipWith op p q
8     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.

```

1 parLdf ::
2   NFDData a
3   => Num a =>
4     (a -> a -> a) -> Int -> Int -> P.PowerList a -> P.PowerList a
5 parLdf _ _ _ [] = []
6 parLdf _ _ _ [x] = [x]
7 parLdf op cs d l
8   | d > 4 =
9     runEval
10      (do p <- rpar (odds l)
11          q <- rpar (evens l)
12          _ <- rseq p
13          _ <- rseq q
14          pq <- rseq (P.parZipWith rdeepseq cs op p q)
15          t <- rparWith rdeepseq (parLdf op cs (d - 1) pq)
16          k <- rseq (P.parZipWith rdeepseq cs op (0 : t) p)
17          rseq $ P.zip k t)
18 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.

```

1 import qualified UBVecPowerlist as UVP
2
3 parLdfUBVec ::
4   (NFDData a, UV.Unbox a, Num a)
5   => (a -> a -> a)
6   -> Int
7   -> Int
8   -> UVP.PowerList a
9   -> UVP.PowerList a
10 parLdfUBVec _ _ _ l
11   | UV.length l <= 1 = l
12 parLdfUBVec op cs d l
13   | d > 4 =
14     runEval
15      (do p <- rpar $ UVP.filterOdd l
16          q <- rpar $ UVP.filterEven l
17          _ <- rseq p
18          _ <- rseq q
19          pq <- UVP.parZipWith (rparWith rdeepseq) op cs p q
20          t <- rpar (parLdfUBVec op cs (d - 1) pq)
21          k <- rseq $ UVP.shiftAdd2 t p
22          UVP.parZip (rparWith rdeepseq) cs k t)
23 parLdfUBVec op _ _ l = UV.scanl1 op l

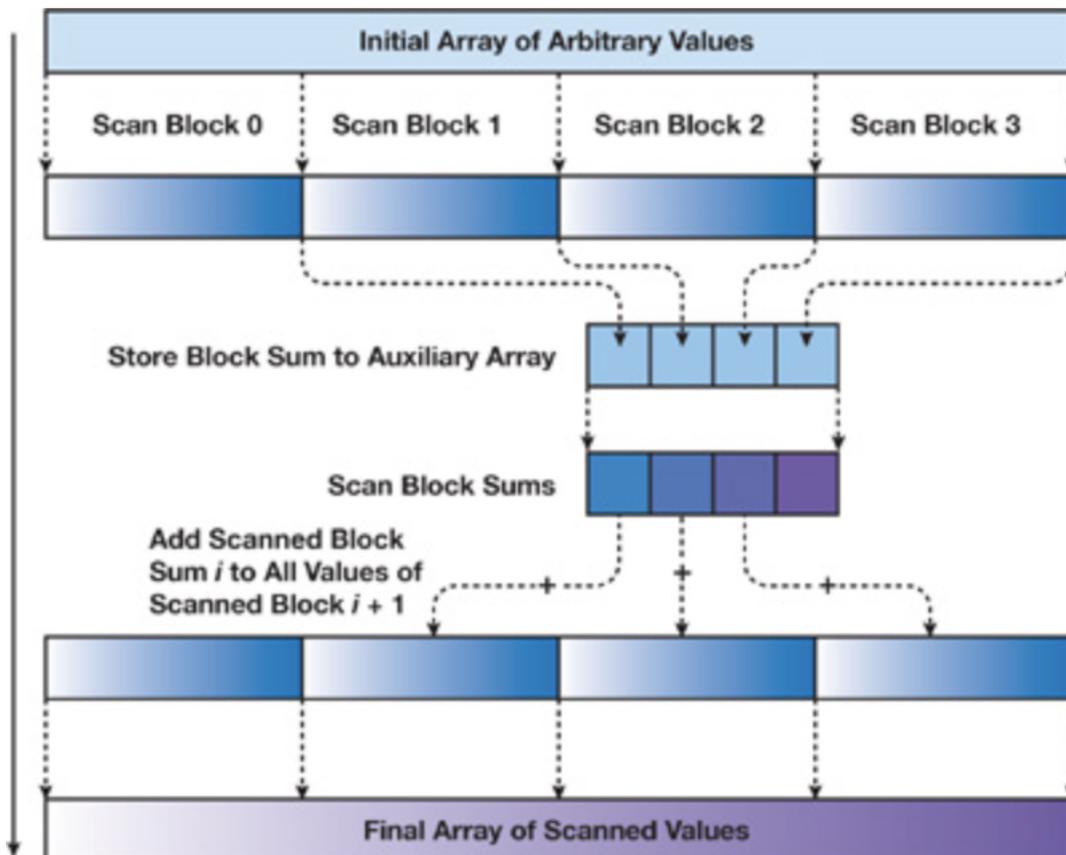
```

It has the same advantages as the SPSUBVecPLPar algorithm above. Note the use of *shiftAdd2* 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:

```

1 parLdfChunkUBVec ::
2   (NFData a, UV.Unbox a, Num a)
3 => (a -> a -> a)
4 -> Int
5 -> UVP.PowerList a
6 -> UVP.PowerList a
7 parLdfChunkUBVec _ _ 1

```

```

8 | UV.length l <= 1 = 1
9 parLdfChunkUBVec ops cs l = runEval $ parLdfChunkVec' ops chunks
10 where
11   n = UV.length l
12   chunkSize = 2 ^ cs
13   chunks = S.chunksOf chunkSize l
14   parLdfChunkVec' ::
15     (NFData a, UV.Unbox a, Num a)
16   => (a -> a -> a)
17   -> [UVP.PowerList a]
18   -> Eval (UVP.PowerList a)
19   parLdfChunkVec' _ [] = return UV.empty
20   parLdfChunkVec' op vChunks = do
21     resChunks <- parList rseq (parLdfUBVecNC op cs <$> vChunks)
22     res <- rseq $ UV.concat resChunks
23     -- Get last element of each block
24     lastelems <- parList rdeepseq (UV.last <$> resChunks)
25     lastScan <- rseq (UV.fromList $ sequentialSPS op lastelems)
26     rseq $
27       UV.create $ do
28         m <- UV.thaw res
29         -- Not sure how to parallelise here!
30         mergeChunks (n - 1) (UV.tail $ UV.reverse lastScan) m
31         return m
32   mergeChunks i lastScan m
33   | i > chunkSize = do
34     let ad = UV.head lastScan
35     go m chunkSize i ad 0
36     mergeChunks (i - chunkSize) (UV.tail lastScan) m
37   | otherwise = return ()
38   go m chs start v i
39   | i < chs = do
40     curr <- M.unsafeRead m (start - i)
41     M.unsafeWrite m (start - i) (curr + v)
42     go m chs start v (i + 1)
43   | 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:

$$\begin{aligned}
 \text{sort } \langle x \rangle &= \langle x \rangle \\
 \text{sort}(p \bowtie q) &= (\text{sort } p) \text{ merge } (\text{sort } q)
 \end{aligned}$$

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*

$$\langle x \rangle \text{ bm } \langle y \rangle = \langle x \rangle \updownarrow \langle y \rangle$$

$$(r \bowtie s) \text{ bm } (u \bowtie v) = (r \text{ bm } v) \updownarrow (s \text{ bm } u) \quad \text{where } p \updownarrow q = (p \text{ min } q) \bowtie (p \text{ max } q)$$

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 $2i^{\text{th}}$ element to $p_i \text{ min } q_i$ and setting the $(2i + 1)^{\text{th}}$ element to $p_i \text{ max } q_i$, 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

```

1 minMaxZip :: (V.Unbox a, Ord a) => PowerList a -> PowerList a -> PowerList a
2 minMaxZip xs ys =
3   V.create $ do
4     m <- M.new n
5     write m 0
6     return m
7   where
8     n = V.length xs + V.length ys
9     write mv i
10    | i < n = do
11      let p = xs V.! (i 'div' 2)
12          q = ys V.! (i 'div' 2)
13          M.unsafeWrite mv i (p 'min' q)
14          M.unsafeWrite mv (i + 1) (p 'max' q)
15          write mv (i + 2)
16    | otherwise = return ()

```

This gives us the following batcher merge sort implementation in haskell

```

1 batcherMergeSort :: (Ord a, V.Unbox a) => P.PowerList a -> P.PowerList a
2 batcherMergeSort l
3   | V.length l <= 1 = l
4 batcherMergeSort l = sortp 'batcherMerge' sortq
5   where
6     sortp = batcherMergeSort p
7     sortq = batcherMergeSort q
8     p = P.filterOdd l
9     q = P.filterEven l
10
11 batcherMerge ::
12   (Ord a, V.Unbox a) => P.PowerList a -> P.PowerList a -> P.PowerList a
13 batcherMerge x y
14   | V.length x == 1 = V.fromList [hx 'min' hy, hx 'max' hy]
15   where
16     hx = V.head x
17     hy = V.head y
18 batcherMerge x y = P.minMaxZip rv su
19   where
20     rv = r 'batcherMerge' v
21     su = s 'batcherMerge' u
22     r = P.filterOdd x
23     v = P.filterEven y

```

```
24 s = P.filterEven x
25 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.

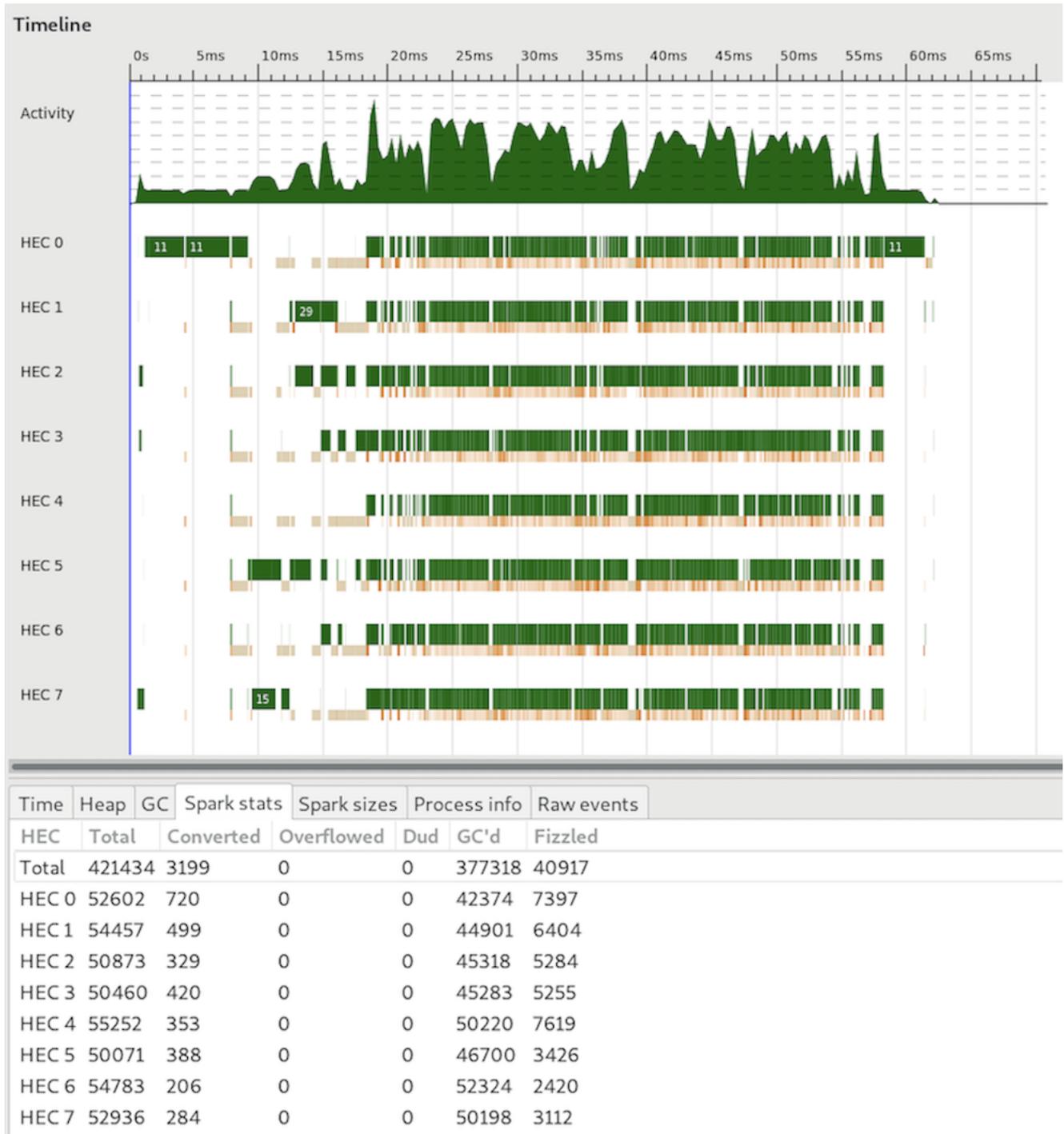
SPSUBVecPLPar

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:

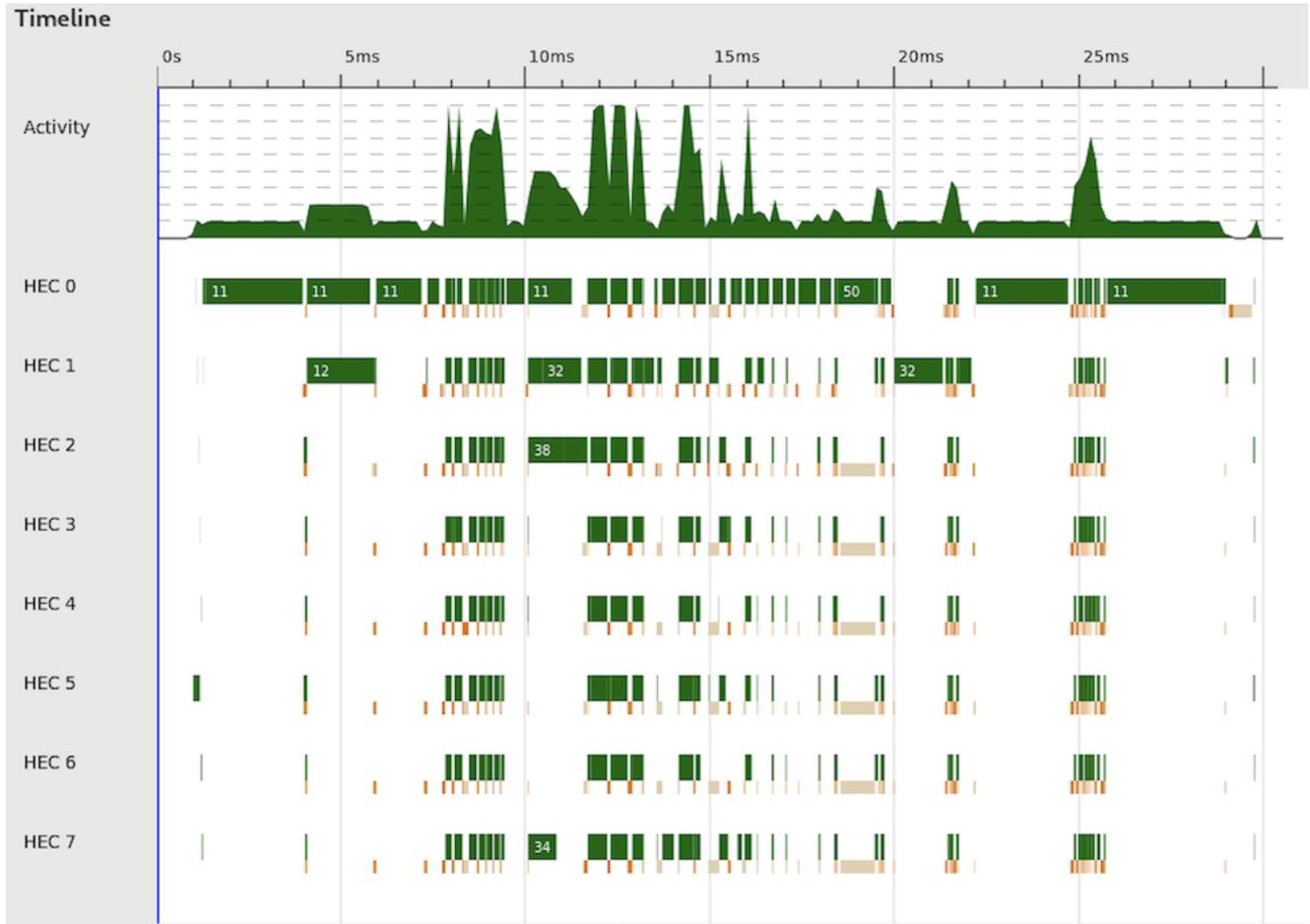


LDFUBVecPLPar

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:



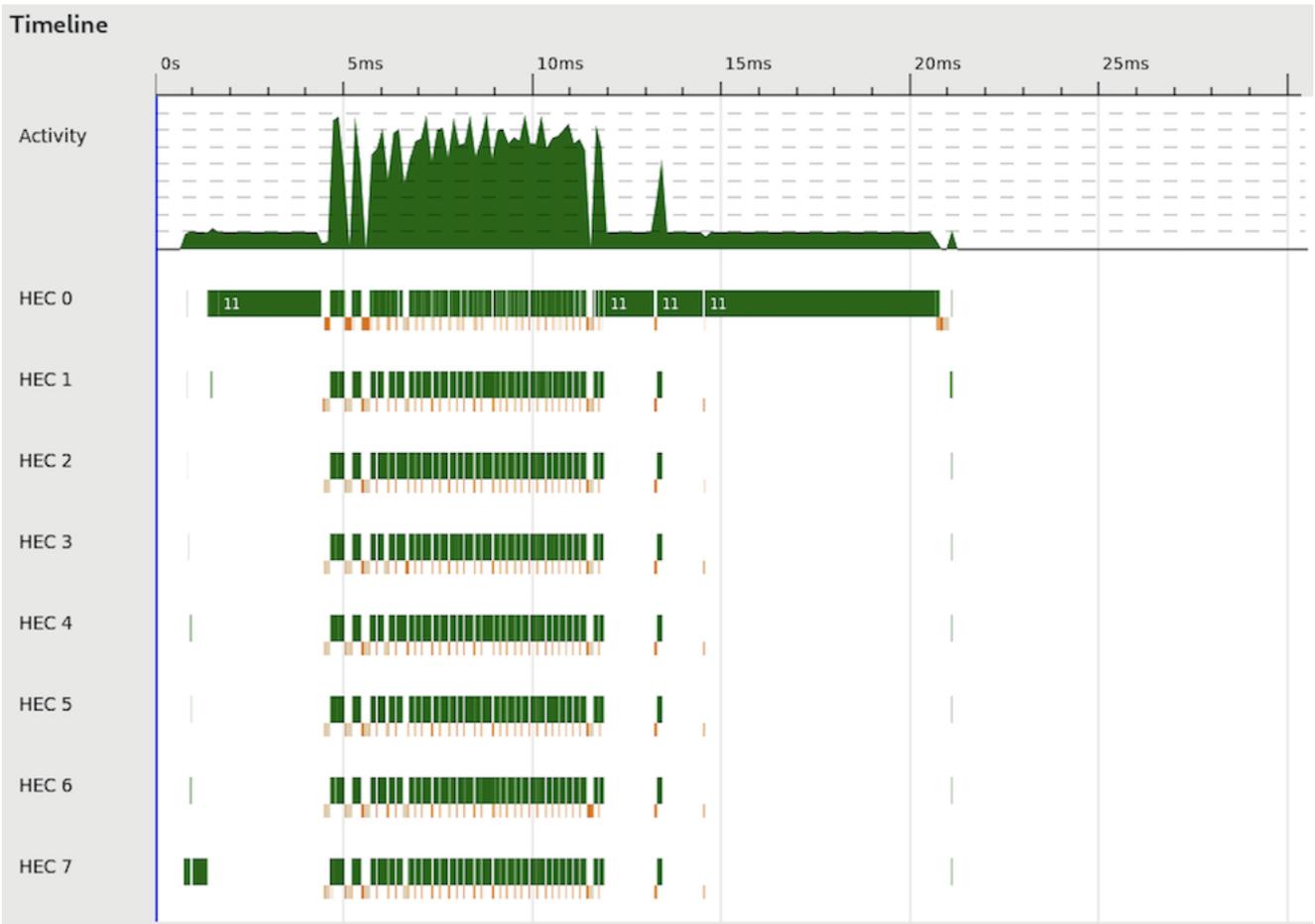
Time	Heap	GC	Spark stats	Spark sizes	Process info	Raw events
HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	4166	3675	0	0	473	18
HEC 0	2512	847	0	0	344	13
HEC 1	899	489	0	0	129	1
HEC 2	119	373	0	0	0	0
HEC 3	115	364	0	0	0	0
HEC 4	112	355	0	0	0	0
HEC 5	129	398	0	0	0	0
HEC 6	129	404	0	0	0	0
HEC 7	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:



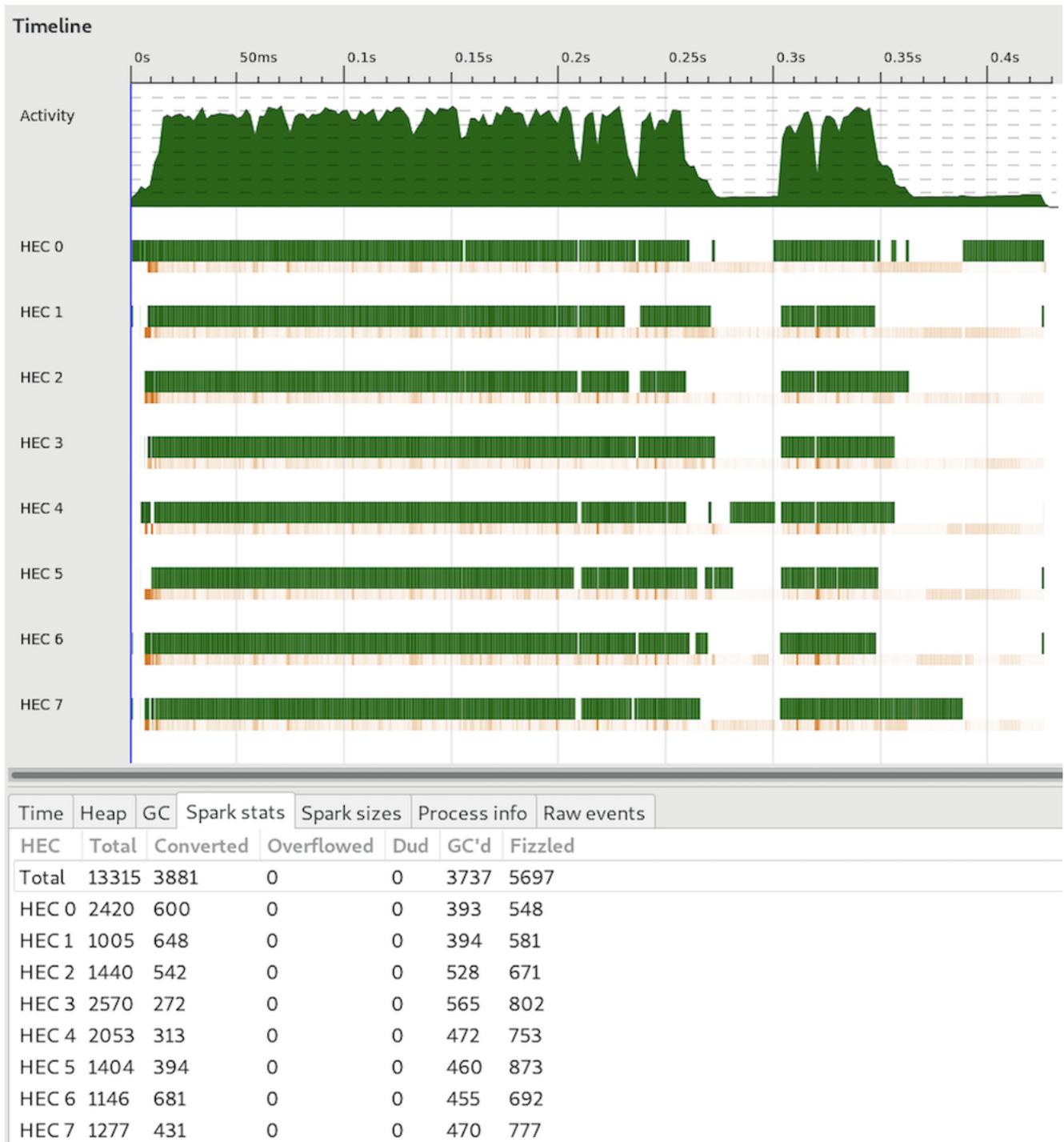
Time	Heap	GC	Spark stats	Spark sizes	Process info	Raw events
HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	37024	1076	0	0	6293	29655
HEC 0	5368	41	0	0	3237	23
HEC 1	4604	160	0	0	336	4544
HEC 2	4616	153	0	0	442	4317
HEC 3	4752	154	0	0	428	4451
HEC 4	4700	151	0	0	463	4494
HEC 5	4636	149	0	0	461	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:



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

```

1  module Main where
2
3  import CLParser
4      ( Command(Scan, Sort)
5        , Opts(Opts)
6        , ScanAlgo(LDF, LDFChunkUBVecPLPar, LDFPar, LDFUBVecPLPar, SPS,
7                  SPSPL, SPSPLPar1, SPSPLPar2, SPSPLPar3, SPSUBVecPLPar)
8        , SortAlgo(BATCHER, DEFAULT)
9        , parseArgs
10      )
11  import Scan
12      ( ldf
13        , parSps1
14        , runParLdf
15        , runParLdfChunkUBVec
16        , runParLdfUBVec
17        , runParScan2
18        , runParScan3
19        , runParSpsUBVec
20        , runScan
21        , sequentialSPS
22        , sps
23      )
24  import Sort (runBatcherSort, runDefaultSort)
25
26  main :: IO ()
27  main = run ==<< parseArgs
28
29  run :: Opts -> IO ()
30  run opts =
31      case opts
32      -- Run parallel prefix sum with powerlist
33      of
34      Opts (Scan SPSPLPar1 n _) -> putStrLn $ runScan parSps1 n
35      Opts (Scan SPSPLPar2 n c) -> putStrLn $ runParScan2 c n
36      Opts (Scan SPSPLPar3 n c) -> putStrLn $ runParScan3 c n
37      -- Run prefix sum via ldf algo (sequential)
38      Opts (Scan LDF n _) -> putStrLn $ runScan ldf n
39      -- Run parallel prefix sum via ldf algo
40      Opts (Scan LDFPar n c) -> putStrLn $ runParLdf c n
41      -- Run sequential prefix sum without powerlist
42      Opts (Scan SPS n _) -> putStrLn $ runScan sequentialSPS n
43      -- Run sequential prefix sum with powerlist
44      Opts (Scan SPSPL n _) -> putStrLn $ runScan sps n
45      -- Run parallel prefix sum using unboxed vecpowerlist
46      Opts (Scan SPSUBVecPLPar n cs) -> putStrLn $ runParSpsUBVec cs n
47      Opts (Scan LDFUBVecPLPar n cs) -> putStrLn $ runParLdfUBVec cs n
48      Opts (Scan LDFChunkUBVecPLPar n cs) -> putStrLn $ runParLdfChunkUBVec cs n
49      -- Run sort

```

```

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

bench/Main.hs

```

1  module Main where
2
3  import Utils
4    ( generateList
5    , generateReverseList
6    , generateReverseUVec
7    , generateUVec
8    )
9
10 import Criterion.Main (bench, bgroup, env, nf, defaultConfig, defaultMainWith)
11
12 import Control.DeepSeq (force)
13
14 import Scan
15    ( ldf
16    , parLdf
17    , parLdfChunkUBVec
18    , parLdfUBVec
19    , parSps1
20    , parSps2
21    , parSps3
22    , parSpsUBVec
23    , sequentialSPS
24    , sps
25    )
26 import Sort (defaultSort, parBatcherMergeSort)
27
28 import qualified Data.Vector.Unboxed as V
29 import Criterion.Types (Config, resamples)
30
31 baseConfig :: Config
32 baseConfig = defaultConfig {
33     resamples = 20
34 }
35
36 setUpEnv :: IO (V.Vector Int, [Int], V.Vector Int, [Int])
37 setUpEnv = do
38   let scanInpUV = force generateUVec 20
39       scanInpL = force generateList 20
40       sortInpUV = force generateReverseUVec 20
41       sortInpL = force generateReverseList 20
42   return (scanInpUV, scanInpL, sortInpUV, sortInpL)
43
44 main :: IO ()
45 main =
46   defaultMainWith baseConfig
47     [ env setUpEnv $ \ ~(scanInpUV, scanInpL, sortInpUV, sortInpL) ->
48       bgroup
49         "main"
  
```

```

50 [ bgroup
51     "scan"
52     [ bgroup
53         "par"
54         [ bgroup "nc" [bench "SPSPLPar1" $ nf (parSps1 (+)) scanInpL]
55         , bgroup
56             "4"
57             [ bench "LDFChunkUBVecPLPar" $
58               nf (parLdfChunkUBVec (+) 4) scanInpUV
59             ]
60         , bgroup
61             "5"
62             [ bench "LDFChunkUBVecPLPar" $
63               nf (parLdfChunkUBVec (+) 5) scanInpUV
64             ]
65         , bgroup
66             "6"
67             [ bench "LDFChunkUBVecPLPar" $
68               nf (parLdfChunkUBVec (+) 6) scanInpUV
69             ]
70         , bgroup
71             "7"
72             [ bench "LDFChunkUBVecPLPar" $
73               nf (parLdfChunkUBVec (+) 7) scanInpUV
74             ]
75         , bgroup
76             "8"
77             [ bench "LDFChunkUBVecPLPar" $
78               nf (parLdfChunkUBVec (+) 8) scanInpUV
79             ]
80         , bgroup
81             "9"
82             [ bench "LDFChunkUBVecPLPar" $
83               nf (parLdfChunkUBVec (+) 9) scanInpUV
84             ]
85         , bgroup
86             "10"
87             [ bench "LDFChunkUBVecPLPar" $
88               nf (parLdfChunkUBVec (+) 10) scanInpUV
89             ]
90         , bgroup
91             "128"
92             [ bench "SPSPLPar2" $ nf (parSps2 (+) 128) scanInpL
93               , bench "SPSPLPar3" $ nf (parSps3 (+) 128 20) scanInpL
94               , bench "LDFPar" $ nf (parLdf (+) 128 20) scanInpL
95               , bench "LDFUBVecPLPar" $
96                 nf (parLdfUBVec (+) 128 20) scanInpUV
97               , bench "SPSUBVecPLPar" $
98                 nf (parSpsUBVec (+) 128 20) scanInpUV
99             ]
100        , bgroup
101            "256"
102            [ bench "SPSPLPar2" $ nf (parSps2 (+) 256) scanInpL
103              , bench "SPSPLPar3" $ nf (parSps3 (+) 256 20) scanInpL
104              , bench "LDFPar" $ nf (parLdf (+) 256 20) scanInpL

```

```

105     , bench "LDFUBVecPLPar" $
106     nf (parLdfUBVec (+) 256 20) scanInpUV
107     , bench "SPSUBVecPLPar" $
108     nf (parSpsUBVec (+) 256 20) scanInpUV
109   ]
110   , bgroup
111     "512"
112     [ bench "SPSPLPar2" $ nf (parSps2 (+) 512) scanInpL
113     , bench "SPSPLPar3" $ nf (parSps3 (+) 512 20) scanInpL
114     , bench "LDFPar" $ nf (parLdf (+) 512 20) scanInpL
115     , bench "LDFUBVecPLPar" $
116     nf (parLdfUBVec (+) 512 20) scanInpUV
117     , bench "SPSUBVecPLPar" $
118     nf (parSpsUBVec (+) 512 20) scanInpUV
119   ]
120   , bgroup
121     "1024"
122     [ bench "LDFUBVecPLPar" $
123     nf (parLdfUBVec (+) 1024 20) scanInpUV
124     , bench "SPSUBVecPLPar" $
125     nf (parSpsUBVec (+) 1024 20) scanInpUV
126   ]
127   , bgroup
128     "2048"
129     [ bench "LDFUBVecPLPar" $
130     nf (parLdfUBVec (+) 2048 20) scanInpUV
131     , bench "SPSUBVecPLPar" $
132     nf (parSpsUBVec (+) 2048 20) scanInpUV
133   ]
134   , bgroup
135     "4096"
136     [ bench "LDFUBVecPLPar" $
137     nf (parLdfUBVec (+) 4096 20) scanInpUV
138     , bench "SPSUBVecPLPar" $
139     nf (parSpsUBVec (+) 4096 20) scanInpUV
140   ]
141   , bgroup
142     "8192"
143     [ bench "LDFUBVecPLPar" $
144     nf (parLdfUBVec (+) 8192 20) scanInpUV
145     , bench "SPSUBVecPLPar" $
146     nf (parSpsUBVec (+) 8192 20) scanInpUV
147   ]
148 ]
149 , bgroup
150   "seq"
151   [ bench "LDF" $ nf (ldf (+)) scanInpL
152   , bench "SPSPL" $ nf (sps (+)) scanInpL
153   , bench "SPS" $ nf (sequentialSPS (+)) scanInpL
154   ]
155 ]
156 , bgroup
157   "sort"
158   [ bench "BATCHER" $ nf (parBatcherMergeSort 20) sortInpUV
159   , bench "DEFAULT" $ nf defaultSort sortInpL

```

```

160     ]
161   ]
162 ]
  
```

src/CLParser.hs

```

1  {-# LANGUAGE RecordWildCards #-}
2
3  module CLParser where
4
5  import Options.Applicative
6    ( CommandFields
7    , Mod
8    , Parser
9    , ReadM
10   , auto
11   , command
12   , customExecParser
13   , eitherReader
14   , fullDesc
15   , header
16   , help
17   , helper
18   , hsubparser
19   , info
20   , long
21   , metavar
22   , option
23   , prefs
24   , progDesc
25   , short
26   , showHelpOnEmpty
27   , value
28   )
29
30  data RunType
31    = Sequential
32    | Parallel
33
34  data ScanAlgo
35    = SPS
36    | SPSPL
37    | SPSPLPar1
38    | SPSPLPar2
39    | SPSPLPar3
40    | LDF
41    | LDFPar
42    | SPSUBVecPLPar
43    | LDFUBVecPLPar
44    | LDFChunkUBVecPLPar
45    deriving (Show, Enum)
46
47  data SortAlgo
48    = DEFAULT
  
```

```

49 | BATCHER
50 deriving (Show, Enum)
51
52 newtype Opts =
53   Opts
54   { cmd :: Command
55     }
56
57 -- Add more features here in the future
58 data Command
59   = Scan ScanAlgo Int Int
60   | Sort SortAlgo Int Int
61
62 parser :: Parser Opts
63 parser = Opts <$> hsubparser (scanCommand <> sortCommand)
64 where
65   sortCommand :: Mod CommandFields Command
66   sortCommand =
67     command "sort" (info sortOptions (progDesc "Run Sort Algorithm"))
68   sortOptions :: Parser Command
69   sortOptions =
70     Sort <$>
71     option
72       sortAlgoReader
73       (long "algo" <>
74         short 'a' <>
75         metavar "ALGONAME" <> help ("Supported Algos: " ++ show [DEFAULT ..])) <*>
76     option
77       auto
78       (long "size" <>
79         short 's' <>
80         metavar "INPSIZE" <>
81         help "Size of array in terms of powers of 2 on which to run sort") <*>
82     option
83       auto
84       (long "csize" <>
85         short 'c' <>
86         metavar "CHUNKSIZE" <>
87         value 64 <> help "Size of chunks for parallelization")
88   sortAlgoReader :: ReadM SortAlgo
89   sortAlgoReader =
90     eitherReader $ \arg ->
91       case arg of
92         "DEFAULT" -> Right (DEFAULT)
93         "BATCHER" -> Right (BATCHER)
94         _ -> Left ("Invalid Algo")
95   scanCommand :: Mod CommandFields Command
96   scanCommand =
97     command "scan" (info scanOptions (progDesc "Run Scan Algorithm"))
98   scanOptions :: Parser Command
99   scanOptions =
100     Scan <$>
101     option
102       scanAlgoReader
103       (long "algo" <>

```

```

104     short 'a' <>
105     metavar "ALGONAME" <> help ("Supported Algos: " ++ show [SPS ..]) <*>
106 option
107 auto
108 (long "size" <>
109  short 's' <>
110  metavar "INPSIZE" <>
111  help "Size of array in terms of powers of 2 on which to run scan") <*>
112 option
113 auto
114 (long "csize" <>
115  short 'c' <>
116  metavar "CHUNKSIZE" <>
117  value 64 <> help "Size of chunks for parallelization")
118 scanAlgoReader :: ReadM ScanAlgo
119 scanAlgoReader =
120  eitherReader $ \arg ->
121  case arg of
122    "SPS" -> Right (SPS)
123    "SPSPL" -> Right (SPSPL)
124    "SPSPLPar1" -> Right (SPSPLPar1)
125    "SPSPLPar2" -> Right (SPSPLPar2)
126    "SPSPLPar3" -> Right (SPSPLPar3)
127    "LDF" -> Right (LDF)
128    "LDFPar" -> Right (LDFPar)
129    "SPSUBVecPLPar" -> Right (SPSUBVecPLPar)
130    "LDFUBVecPLPar" -> Right (LDFUBVecPLPar)
131    "LDFChunkUBVecPLPar" -> Right (LDFChunkUBVecPLPar)
132    _ -> Left ("Invalid Algo")
133
134 parseArgs :: IO Opts
135 parseArgs =
136  customExecParser (prefs showHelpOnEmpty) $
137  info
138    (helper <*> parser)
139    (fullDesc <>
140     progDesc "powerlist" <>
141     header "A program to run algorithms using powerlist abstraction")

```

src/Powerlist.hs

```

1 module Powerlist where
2
3 import Control.Parallel.Strategies
4
5 -- Using simple list here as it would be most performant
6 type PowerList a = [a]
7
8 tie :: PowerList a -> PowerList a -> PowerList a
9 {-# INLINE tie #-}
10 tie = (++)
11
12 zip :: PowerList a -> PowerList a -> PowerList a
13 {-# INLINE zip #-}

```

```

14 zip [] [] = []
15 zip xs ys = Prelude.zip xs ys >>= \(a, b) -> [a, b]
16
17 parZip :: Strategy a -> Int -> PowerList a -> PowerList a -> PowerList a
18 {-# INLINE parZip #-}
19 parZip strategy cs as bs = PowerList.zip as bs 'using' parListChunk cs strategy
20
21 zipWith :: Num a => (a -> a -> a) -> PowerList a -> PowerList a -> PowerList a
22 {-# INLINE zipWith #-}
23 zipWith = Prelude.zipWith
24
25 parZipWith :: Num a => Strategy a -> Int -> (a -> a -> a) -> [a] -> [a] -> [a]
26 {-# INLINE parZipWith #-}
27 parZipWith strategy cs z as bs =
28     PowerList.zipWith z as bs 'using' parListChunk cs strategy
29
30 unzip :: PowerList a -> (PowerList a, PowerList a)
31 unzip =
32     snd .
33     foldr
34         (\x (b, (xs, ys)) ->
35             ( not b
36               , if b
37                 then (x : xs, ys)
38                   else (xs, x : ys)))
39         (False, ([], []))
40
41 {-
42 unzip = Prelude.unzip . splT
43   where splT []      = []
44         splT (x:y:xs) = (x, y) : splT xs
45         splT _        = error "Malformed powerlist"
46 -}
47 -- Right shift and use zero
48 rsh :: a -> PowerList a -> PowerList a
49 {-# INLINE rsh #-}
50 rsh zero xs = zero : init xs

```

src/Scan.hs

```

1 module Scan where
2
3 import Control.DeepSeq (NFData)
4 import Control.Parallel.Strategies
5   ( Eval
6     , parList
7     , r0
8     , rdeepseq
9     , rpar
10    , rparWith
11    , rseq
12    , runEval
13    )
14 import Utils (generateList, generateUVec)

```

```

15
16 import qualified Data.Vector.Split as S
17 import qualified Data.Vector.Unboxed as UV
18 import qualified Data.Vector.Unboxed.Mutable as M
19 import qualified Powerlist as P
20 import qualified UVecPowerlist as UVP
21
22 -----
23 -- Sequential SPS is nothing but haskel's scanl1
24 -----
25 sequentialSPS :: (a -> a -> a) -> [a] -> [a]
26 sequentialSPS = scanl1
27
28 -----
29 -- Sequential SPS using powerlist, works for lists with power of 2 length
30 -----
31 sps :: Num a => (a -> a -> a) -> P.PowerList a -> P.PowerList a
32 sps _ [] = []
33 sps _ [x] = [x]
34 sps op l = P.zip (sps op u) (sps op v)
35   where
36     (u, v) = P.unzip $ P.zipWith op (P.rsh 0 l) l
37
38 -----
39 -- Parallel SPS Version1 using powerlists
40 -----
41 parSps1 :: (NFData a, Num a) => (a -> a -> a) -> P.PowerList a -> P.PowerList a
42 parSps1 _ [] = []
43 parSps1 _ [x] = [x]
44 parSps1 op l =
45   runEval
46     (do (u, v) <- rseq $ P.unzip $ P.zipWith op (P.rsh 0 l) l
47         u' <- rparWith rdeepseq (parSps1 op u)
48         v' <- rparWith rdeepseq (parSps1 op v)
49         rseq $ P.zip u' v')
50
51 runScan ::
52   ((Int -> Int -> Int) -> P.PowerList Int -> P.PowerList Int)
53   -> Int
54   -> String
55 runScan f inp = show $ sum $ f (+) $ generateList inp
56
57 odds :: [a] -> [a]
58 odds [] = []
59 odds [x] = [x]
60 odds (x:_:xs) = x : odds xs
61
62 evens :: [a] -> [a]
63 evens [] = []
64 evens [_] = []
65 evens (_,y:xs) = y : evens xs
66
67 parSps2 ::
68   NFData a
69   => Num a =>

```

```

70     (a -> a -> a) -> Int -> P.PowerList a -> P.PowerList a
71 parSps2 _ _ [] = []
72 parSps2 _ _ [x] = [x]
73 parSps2 op cs l =
74   runEval
75     (do k <- r0 $ P.parZipWith rdeepseq cs op (0 : l) 1
76         u <- rpar (odds k)
77         v <- rpar (evens k)
78         _ <- rseq u
79         u' <- rparWith rdeepseq (parSps2 op cs u)
80         _ <- rseq v
81         v' <- rparWith rdeepseq (parSps2 op cs v)
82         rseq $ P.zip u' v')
83
84 {-
85   Parallel till certain depth, for arrays of size <= 2^4, use sequentialSPS
86 -}
87 parSps3 ::
88   NFDData a
89   => Num a =>
90     (a -> a -> a) -> Int -> Int -> P.PowerList a -> P.PowerList a
91 parSps3 _ _ _ [] = []
92 parSps3 _ _ _ [x] = [x]
93 parSps3 op cs d l
94   | d > 4 =
95     runEval
96       (do k <- r0 $ P.parZipWith rdeepseq cs op (0 : l) 1
97           u <- rpar (odds k)
98           v <- rpar (evens k)
99           _ <- rseq u
100          u' <- rparWith rdeepseq (parSps3 op cs (d - 1) u)
101          _ <- rseq v
102          v' <- rparWith rdeepseq (parSps3 op cs (d - 1) v)
103          rseq $ P.zip u' v')
104 parSps3 op _ _ l = sequentialSPS op l
105
106 runParScan2 :: Int -> Int -> String
107 runParScan2 cs inp = show $ sum $ parSps2 (+) cs $ generateList inp
108
109 runParScan3 :: Int -> Int -> String
110 runParScan3 cs inp = show $ sum $ parSps3 (+) cs inp $ generateList inp
111
112 runParLdf :: Int -> Int -> String
113 runParLdf cs inp = show $ sum $ parLdf (+) cs inp $ generateList inp
114
115 runParSpsUBVec :: Int -> Int -> String
116 runParSpsUBVec cs inp =
117   show $ UV.sum $ parSpsUBVec (+) cs inp $ generateUVec inp
118
119 runParLdfUBVec :: Int -> Int -> String
120 runParLdfUBVec cs inp =
121   show $ UV.sum $ parLdfUBVec (+) cs inp $ generateUVec inp
122
123 runParLdfChunkUBVec :: Int -> Int -> String
124 runParLdfChunkUBVec cs inp =

```

```

125 show $ UV.sum $ parLdfChunkUBVec (+) cs $ generateUVec inp
126
127 -----
128 -- Ladner Fischer Algorithm
129 -----
130 ldf :: Num a => (a -> a -> a) -> P.PowerList a -> P.PowerList a
131 ldf _ [] = []
132 ldf _ [x] = [x]
133 ldf op l = P.zip (P.zipWith op (P.rsh 0 t) p) t
134 where
135     (p, q) = P.unzip l
136     pq = P.zipWith op p q
137     t = ldf op pq
138
139 {-
140   A parallel version of LDF
141 -}
142 parLdf ::
143     NFDData a
144     => Num a =>
145     (a -> a -> a) -> Int -> Int -> P.PowerList a -> P.PowerList a
146 parLdf _ _ _ [] = []
147 parLdf _ _ _ [x] = [x]
148 parLdf op cs d l
149 | d > 4 =
150     runEval
151     (do p <- rpar (odds l)
152         q <- rpar (evens l)
153         _ <- rseq p
154         _ <- rseq q
155         pq <- rseq (P.parZipWith rdeepseq cs op p q)
156         t <- rparWith rdeepseq (parLdf op cs (d - 1) pq)
157         k <- rseq (P.parZipWith rdeepseq cs op (0 : t) p)
158         rseq $ P.zip k t)
159 parLdf op _ _ l = sequentialSPS op l
160
161 -----
162 -- SPS and LDF using powerlist unboxed vector implementation
163 -----
164 parSpsUBVec ::
165     (NFDData a, UV.Unbox a, Num a)
166     => (a -> a -> a)
167     -> Int
168     -> Int
169     -> UVP.PowerList a
170     -> UVP.PowerList a
171 parSpsUBVec _ _ _ l
172 | UV.length l <= 1 = l
173 parSpsUBVec op cs d l
174 | d > 4 =
175     runEval
176     (do k <- rseq $ UVP.shiftAdd l
177         u <- rpar (UVP.filterOdd k)
178         v <- rpar (UVP.filterEven k)
179         _ <- rseq u

```

```

180     u' <- rparWith rdeepseq (parSpsUBVec op cs (d - 1) u)
181     _ <- rseq v
182     v' <- rparWith rdeepseq (parSpsUBVec op cs (d - 1) v)
183     UVP.parZip (rparWith rdeepseq) cs u' v')
184 parSpsUBVec op _ _ l = UV.scanl1 op l
185
186 parLdfUBVec ::
187     (NFData a, UV.Unbox a, Num a)
188 => (a -> a -> a)
189 -> Int
190 -> Int
191 -> UVP.PowerList a
192 -> UVP.PowerList a
193 parLdfUBVec _ _ _ l
194 | UV.length l <= 1 = 1
195 parLdfUBVec op cs d l
196 | d > 4 =
197     runEval
198     (do p <- rpar $ UVP.filterOdd l
199         q <- rpar $ UVP.filterEven l
200         _ <- rseq p
201         _ <- rseq q
202         pq <- UVP.parZipWith (rparWith rdeepseq) op cs p q
203         t <- rpar (parLdfUBVec op cs (d - 1) pq)
204         k <- rseq $ UVP.shiftAdd2 t p
205         UVP.parZip (rparWith rdeepseq) cs k t)
206 parLdfUBVec op _ _ l = UV.scanl1 op l
207
208 -----
209 -- Chunk the input itself, hybrid of ldf and Blelloch
210 -----
211 parLdfUBVecNC ::
212     (NFData a, UV.Unbox a, Num a)
213 => (a -> a -> a)
214 -> Int
215 -> UVP.PowerList a
216 -> UVP.PowerList a
217 parLdfUBVecNC _ _ l
218 | UV.length l <= 1 = 1
219 parLdfUBVecNC op d l
220 | d > 2 =
221     runEval
222     (do p <- rpar $ UVP.filterOdd l
223         q <- rpar $ UVP.filterEven l
224         _ <- rseq p
225         _ <- rseq q
226         pq <- rparWith rdeepseq $ UVP.zipWith op p q
227         t <- rpar (parLdfUBVecNC op (d - 1) pq)
228         k <- rseq $ UVP.shiftAdd2 t p
229         rseq $ UVP.zip k t)
230 parLdfUBVecNC op _ l = UV.scanl1 op l
231
232 parLdfChunkUBVec ::
233     (NFData a, UV.Unbox a, Num a)
234 => (a -> a -> a)

```

```

235 -> Int
236 -> UVP.PowerList a
237 -> UVP.PowerList a
238 parLdfChunkUBVec _ _ 1
239 | UV.length l <= 1 = 1
240 parLdfChunkUBVec ops cs l = runEval $ parLdfChunkVec' ops chunks
241 where
242   n = UV.length l
243   chunkSize = 2 ^ cs
244   chunks = S.chunksOf chunkSize l
245   parLdfChunkVec' ::
246     (NFData a, UV.Unbox a, Num a)
247   => (a -> a -> a)
248   -> [UVP.PowerList a]
249   -> Eval (UVP.PowerList a)
250   parLdfChunkVec' _ [] = return UV.empty
251   parLdfChunkVec' op vChunks = do
252     resChunks <- parList rseq (parLdfUBVecNC op cs <$> vChunks)
253     res <- rseq $ UV.concat resChunks
254     -- Get last element of each block
255     lastelems <- parList rdeepseq (UV.last <$> resChunks)
256     lastScan <- rseq (UV.fromList $ sequentialSPS op lastelems)
257     rseq $
258       UV.create $ do
259         m <- UV.thaw res
260         -- Not sure how to parallelise here!
261         mergeChunks (n - 1) (UV.tail $ UV.reverse lastScan) m
262         return m
263     mergeChunks i lastScan m
264     | i > chunkSize = do
265       let ad = UV.head lastScan
266           go m chunkSize i ad 0
267       mergeChunks (i - chunkSize) (UV.tail lastScan) m
268     | otherwise = return ()
269     go m chs start v i
270     | i < chs = do
271       curr <- M.unsafeRead m (start - i)
272       M.unsafeWrite m (start - i) (curr + v)
273       go m chs start v (i + 1)
274     | otherwise = return ()

```

src/Sort.hs

```

1 module Sort where
2
3 import Control.DeepSeq (NFData)
4 import Control.Parallel.Strategies
5 ( Eval
6   , rdeepseq
7   , rpar
8   , rparWith
9   , rseq
10  , runEval
11  )

```

```

12 import Data.List (sort)
13 import Utils (generateReverseList, generateReverseUVec)
14
15 import qualified Data.Vector.Unboxed as V
16 import qualified Data.Vector.Unboxed.Mutable as M
17 import qualified UVecPowerList as P
18
19 runDefaultSort :: Int -> Int -> String
20 runDefaultSort _ inp = show $ last $ defaultSort $ generateReverseList inp
21
22 runBatcherSort :: Int -> Int -> String
23 runBatcherSort _ inp =
24   show $ V.last $ parBatcherMergeSort inp $ generateReverseUVec inp
25
26 defaultSort :: Ord a => [a] -> [a]
27 defaultSort = sort
28
29 -----
30 -- Sequential Impl for demonstration
31 -----
32 batcherMergeSort :: (Ord a, V.Unbox a) => P.PowerList a -> P.PowerList a
33 batcherMergeSort l
34   | V.length l <= 1 = l
35 batcherMergeSort l = sortp `batcherMerge` sortq
36   where
37     sortp = batcherMergeSort p
38     sortq = batcherMergeSort q
39     p = P.filterOdd l
40     q = P.filterEven l
41
42 batcherMerge ::
43   (Ord a, V.Unbox a) => P.PowerList a -> P.PowerList a -> P.PowerList a
44 batcherMerge x y
45   | V.length x == 1 = V.fromList [hx `min` hy, hx `max` hy]
46   where
47     hx = V.head x
48     hy = V.head y
49 batcherMerge x y = P.minMaxZip rv su
50   where
51     rv = r `batcherMerge` v
52     su = s `batcherMerge` u
53     r = P.filterOdd x
54     v = P.filterEven y
55     s = P.filterEven x
56     u = P.filterOdd y
57
58 -----
59 -- Parallel Impl
60 -----
61 parBatcherMergeSort ::
62   (NFData a, Ord a, V.Unbox a) => Int -> P.PowerList a -> P.PowerList a
63 parBatcherMergeSort _ l
64   | V.length l <= 1 = l
65 parBatcherMergeSort d l
66   | d > 10 =

```

```

67   runEval
68     (do p <- rpar $ P.filterOdd l
69         q <- rpar $ P.filterEven l
70         _ <- rseq p
71         sortp <- rparWith rdeepseq (parBatcherMergeSort (d - 1) p)
72         _ <- rseq q
73         sortq <- rparWith rdeepseq (parBatcherMergeSort (d - 1) q)
74         parBatcherMerge d sortp sortq)
75 parBatcherMergeSort _ l = V.fromList $ defaultSort $ V.toList l
76
77 parBatcherMerge ::
78   (Ord a, V.Unbox a)
79 => Int
80 -> P.PowerList a
81 -> P.PowerList a
82 -> Eval (P.PowerList a)
83 --batcherMerge strategy d cs x y | V.length x == 1 = rseq $ V.fromList [hx 'min' hy, hx 'max' hy]
84 --   where hx = V.head x
85 --         hy = V.head y
86 parBatcherMerge d x y
87 | d > 10 = do
88   r <- rseq $ P.filterOdd x
89   v <- rseq $ P.filterEven y
90   rv <- parBatcherMerge (d - 1) r v
91   s <- rseq $ P.filterEven x
92   u <- rseq $ P.filterOdd y
93   su <- parBatcherMerge (d - 1) s u
94   rparWith rdeepseq $ P.minMaxZip rv su
95 parBatcherMerge _ x y = rseq (merge x y)
96
97 merge :: (Ord a, V.Unbox a) => P.PowerList a -> P.PowerList a -> P.PowerList a
98 merge a b =
99   V.create $ do
100     v <- M.new nm
101     go 0 0 v
102     return v
103   where
104     n = V.length a
105     m = V.length b
106     nm = n + m
107     go i j v
108     | (i + j) < nm = do
109       let ai = a V.! i
110           bj = b V.! j
111           if (j == m) || (i < n && ai <= bj)
112             then do
113               M.unsafeWrite v (i + j) ai
114               go (i + 1) j v
115             else do
116               M.unsafeWrite v (i + j) bj
117               go i (j + 1) v
118     | otherwise = return ()

```

src/UBVecPowerlist.hs

```

1  {-# LANGUAGE FlexibleContexts #-}
2
3  module UBVecPowerlist where
4
5  import Control.Parallel.Strategies (Eval, Strategy, parList, rdeepseq, rseq)
6
7  import qualified Data.Vector.Split as S
8  import qualified Data.Vector.Unboxed as V
9  import qualified Data.Vector.Unboxed.Mutable as M
10
11 type PowerList a = V.Vector a
12
13 tie :: V.Unbox a => PowerList a -> PowerList a -> PowerList a
14 {-# INLINE tie #-}
15 tie = (V.++)
16
17 zip :: (V.Unbox a, Num a) => PowerList a -> PowerList a -> PowerList a
18 {-# INLINE zip #-}
19 --zip xs ys = V.generate (V.length xs + V.length ys) (\i -> if even i then xs V.! (i `div` 2) else ys
20     V.! (i `div` 2))
21 zip xs ys =
22   V.create $ do
23     m <- M.new n
24     write m 0
25     return m
26   where
27     n = V.length xs + V.length ys
28     write m i
29       | i < n = do
30         M.unsafeWrite m i (xs V.! (i `div` 2))
31         M.unsafeWrite m (i + 1) (ys V.! (i `div` 2))
32         write m (i + 2)
33       | otherwise = return ()
34
35 parZip ::
36   (V.Unbox a, Num a)
37   => Strategy (PowerList a)
38   -> Int
39   -> PowerList a
40   -> PowerList a
41   -> Eval (PowerList a)
42 {-# INLINE parZip #-}
43 parZip strategy cs as bs = do
44   inp <- rseq $ Prelude.zip ac bc
45   lists <- parList strategy (writePar <$> inp)
46   rdeepseq $ V.concat lists
47   where
48     ac = S.chunksOf cs as
49     bc = S.chunksOf cs bs
50     writePar (a, b) = UBVecPowerlist.zip a b
51
52 zipWith ::
53   (Num a, V.Unbox a)
54   => (a -> a -> a)
55   -> PowerList a

```

```

55 -> PowerList a
56 -> PowerList a
57 {-# INLINE zipWith #-}
58 zipWith op xs ys =
59   V.create $ do
60     m <- V.thaw xs
61     write m ys 0
62     return m
63   where
64     k = V.length xs
65     write m y i
66       | i < k = do
67         curr <- M.unsafeRead m i
68         M.unsafeWrite m i (op (y V.! i) curr)
69         write m y (i + 1)
70       | otherwise = return ()
71
72 parZipWith ::
73   (Num a, V.Unbox a)
74 => Strategy (PowerList a)
75 -> (a -> a -> a)
76 -> Int
77 -> PowerList a
78 -> PowerList a
79 -> Eval (PowerList a)
80 {-# INLINE parZipWith #-}
81 parZipWith strategy op cs as bs = do
82   inp <- rseq $ Prelude.zip ac bc
83   lists <- parList strategy (writePar <$> inp)
84   rdeepseq $ V.concat lists
85   where
86     ac = S.chunksOf cs as
87     bc = S.chunksOf cs bs
88     writePar (a, b) = UVecPowerlist.zipWith op a b
89
90 unzip :: V.Unbox a => PowerList a -> (PowerList a, PowerList a)
91 unzip k = (b, c)
92   where
93     b = V.ifilter (\i _ -> even i) k
94     c = V.ifilter (\i _ -> odd i) k
95
96 filterUsing :: V.Unbox a => (Int -> Int) -> PowerList a -> PowerList a
97 filterUsing op l =
98   V.create $ do
99     m <- M.new n
100    write m 0
101    return m
102   where
103     nl = V.length l
104     n = nl `div` 2
105     write m i
106       | i < n = do
107         M.unsafeWrite m i (l V.! op i)
108         write m (i + 1)
109       | otherwise = return ()

```

```

110
111 calculateEvenInd :: Int -> Int
112 calculateEvenInd = (* 2)
113
114 calculateOddInd :: Num a => a -> a
115 calculateOddInd i = (i * 2) + 1
116
117 filterOdd :: V.Unbox a => PowerList a -> PowerList a
118 filterOdd = filterUsing calculateEvenInd
119
120 filterEven :: V.Unbox a => PowerList a -> PowerList a
121 filterEven = filterUsing calculateOddInd
122
123 -- Right shift and use zero, does not perform well as cons is O(n)
124 rsh :: V.Unbox a => a -> PowerList a -> PowerList a
125 {-# INLINE rsh #-}
126 rsh zero xs = V.cons zero $ V.init xs
127
128 shiftAdd :: (V.Unbox a, Num a) => PowerList a -> PowerList a
129 shiftAdd l =
130   V.create $ do
131     m <- V.thaw l
132     go (V.length l - 1) m
133     return m
134   where
135     go ind mv
136       | ind > 0 = do
137         prev <- M.unsafeRead mv (ind - 1)
138         curr <- M.unsafeRead mv ind
139         M.unsafeWrite mv ind (prev + curr)
140         go (ind - 1) mv
141       | otherwise = return ()
142
143 shiftAdd2 :: (V.Unbox a, Num a) => PowerList a -> PowerList a -> PowerList a
144 shiftAdd2 r l =
145   V.create $ do
146     m <- V.thaw l
147     go (V.length l - 1) m
148     return m
149   where
150     go ind mv
151       | ind > 0 = do
152         curr <- M.unsafeRead mv ind
153         M.unsafeWrite mv ind ((r V.! (ind - 1)) + curr)
154         go (ind - 1) mv
155       | otherwise = return ()
156
157 addPairs :: (V.Unbox a, Num a) => PowerList a -> PowerList a
158 addPairs l =
159   V.create $ do
160     m <- M.new n
161     addPairs' m 0
162     return m
163   where
164     n = V.length l `div` 2

```

```

165     addPairs' mv i
166     | i < n = do
167         M.unsafeWrite mv i (1 V.! (2 * i) + (1 V.! (2 * i + 1)))
168         addPairs' mv (i + 1)
169     | otherwise = return ()
170
171 minMaxZip :: (V.Unbox a, Ord a) => PowerList a -> PowerList a -> PowerList a
172 minMaxZip xs ys =
173     V.create $ do
174         m <- M.new n
175         write m 0
176         return m
177     where
178         n = V.length xs + V.length ys
179         write mv i
180         | i < n = do
181             let p = xs V.! (i `div` 2)
182                 q = ys V.! (i `div` 2)
183             M.unsafeWrite mv i (p `min` q)
184             M.unsafeWrite mv (i + 1) (p `max` q)
185             write mv (i + 2)
186         | otherwise = return ()
187
188 parMinMaxZip ::
189     (V.Unbox a, Ord a)
190     => Strategy (PowerList a)
191     -> Int
192     -> PowerList a
193     -> PowerList a
194     -> Eval (PowerList a)
195 {-# INLINE parMinMaxZip #-}
196 parMinMaxZip strategy cs as bs = do
197     inp <- rseq $ Prelude.zip ac bc
198     lists <- parList strategy (writePar <$> inp)
199     rdeepseq $ V.concat lists
200     where
201         ac = S.chunksOf cs as
202         bc = S.chunksOf cs bs
203     writePar (a, b) = UVecPowerlist.minMaxZip a b

```

src/Utils.hs

```

1 module Utils where
2
3 import qualified Data.Vector.Unboxed as V
4
5 {-
6     Generates a list from 2^n to 1
7 -}
8 generateReverseList :: Int -> [Int]
9 generateReverseList n = [2 ^ n, 2 ^ n - 1 .. 1]
10
11 {-
12     Generates a list from 2^n to 1

```

```

13 -}
14 generateList :: Int -> [Int]
15 generateList n = [1 .. 2 ^ n]
16
17 {-
18   Generate Unboxed Vector with values from 1 to 2^n
19 -}
20 generateUVec :: Int -> V.Vector Int
21 generateUVec n = V.generate (2 ^ n) (+ 1)
22
23 {-
24   Generate Unboxed Vector with values from 2^n to 1
25 -}
26 generateReverseUVec :: Int -> V.Vector Int
27 generateReverseUVec n = V.generate (2 ^ n) (\i -> 2 ^ n - i)

```

src/VecPowerlist.hs

```

1 module VecPowerlist where
2
3 import qualified Data.Vector as V
4 import qualified Data.Vector.Mutable as M
5
6 -- Using simple list here as it would be most performant
7 type PowerList a = V.Vector a
8
9 tie :: PowerList a -> PowerList a -> PowerList a
10 {-# INLINE tie #-}
11 tie = (V.++)
12
13 zip :: PowerList a -> PowerList a -> PowerList a
14 {-# INLINE zip #-}
15 zip xs ys =
16   V.generate
17     (V.length xs + V.length ys)
18     (\i ->
19       if even i
20         then xs V.! (i `div` 2)
21         else ys V.! (i `div` 2))
22
23 --zip _ _ = error "Non similar powerlists"
24 zipWith :: Num a => (a -> a -> a) -> PowerList a -> PowerList a -> PowerList a
25 {-# INLINE zipWith #-}
26 zipWith = V.zipWith
27
28 unzip :: PowerList a -> (PowerList a, PowerList a)
29 unzip k = (b, c)
30   where
31     b = V.ifilter (\i _ -> even i) k
32     c = V.ifilter (\i _ -> odd i) k
33
34 -- Right shift and use zero, does not perform well as cons is O(n)
35 rsh :: a -> PowerList a -> PowerList a
36 {-# INLINE rsh #-}

```

```

37 rsh zero xs = V.cons zero $ V.init xs
38
39 shiftAdd :: Num a => PowerList a -> PowerList a
40 shiftAdd l =
41   V.create $ do
42     m <- V.thaw l
43     go (V.length l - 1) m
44     return m
45   where
46     go i mv
47       | i > 0 = do
48         prev <- M.unsafeRead mv (i - 1)
49         curr <- M.unsafeRead mv i
50         M.unsafeWrite mv i (prev + curr)
51         go (i - 1) mv
52       | otherwise = return ()
53
54 shiftAdd2 :: Num a => PowerList a -> PowerList a -> PowerList a
55 shiftAdd2 r l =
56   V.create $ do
57     m <- V.thaw l
58     go (V.length l - 1) m
59     return m
60   where
61     go i mv
62       | i > 0 = do
63         curr <- M.unsafeRead mv i
64         M.unsafeWrite mv i ((r V.! (i - 1)) + curr)
65         go (i - 1) mv
66       | otherwise = return ()
67
68 addPairs :: Num a => PowerList a -> PowerList a
69 addPairs l =
70   V.create $ do
71     m <- M.new n
72     addPairs' m 0
73     return m
74   where
75     n = V.length l `div` 2
76     addPairs' mv i
77       | i < n = do
78         M.unsafeWrite mv i (l V.! (2 * i) + (l V.! (2 * i + 1)))
79         addPairs' mv (i + 1)
80       | otherwise = return ()

```

test/Spec.hs

```

1 import Test.Hspec
2
3 import Utils ( generateList, generateUVec, generateReverseUVec )
4
5 import qualified Scan
6 import qualified Data.Vector.Unboxed as UV
7 import qualified Sort

```

```

8 import qualified Sort
9
10 main :: IO ()
11 main = hspec $ do
12     describe "Scan.sps" $ do
13         it "correctly calculates prefix sum" $ do
14             Scan.sps (+) (generateList 2) 'shouldBe' [1, 3, 6, 10]
15
16     describe "Scan.parSps2" $ do
17         it "correctly calculates prefix sum" $ do
18             Scan.parSps2 (+) 2 (generateList 2) 'shouldBe' [1, 3, 6, 10]
19
20     describe "Scan.parSps3" $ do
21         it "correctly calculates prefix sum" $ do
22             Scan.parSps3 (+) 10 6 (generateList 6) 'shouldBe' scanl1 (+) [1..2^6]
23
24     describe "Scan.ldf" $ do
25         it "correctly calculates prefix sum" $ do
26             Scan.ldf (+) (generateList 6) 'shouldBe' scanl1 (+) [1..2^6]
27
28     describe "Scan.parLdf" $ do
29         it "correctly calculates prefix sum" $ do
30             Scan.parLdf (+) 10 6 (generateList 6) 'shouldBe' scanl1 (+) [1..2^6]
31
32     describe "Scan.parSpsUBVec" $ do
33         it "correctly calculates prefix sum" $ do
34             Scan.parSpsUBVec (+) 10 6 (generateUVec 6) 'shouldBe' UV.fromList (scanl1 (+) [1..2^6])
35
36     describe "Scan.parLdfUBVec" $ do
37         it "correctly calculates prefix sum" $ do
38             Scan.parLdfUBVec (+) 10 6 (generateUVec 6) 'shouldBe' UV.fromList (scanl1 (+) [1..2^6])
39
40     describe "Scan.parLdfUBVecNC" $ do
41         it "correctly calculates prefix sum" $ do
42             Scan.parLdfUBVecNC (+) 6 (generateUVec 6) 'shouldBe' UV.fromList (scanl1 (+) [1..2^6])
43
44     describe "Scan.parLdfChunkUBVec" $ do
45         it "correctly calculates prefix sum" $ do
46             Scan.parLdfChunkUBVec (+) 2 (generateUVec 8) 'shouldBe' UV.fromList (scanl1 (+) [1..2^8])
47
48     describe "Sort.batcherMergeSort" $ do
49         it "correctly sorts the input vector" $ do
50             Sort.batcherMergeSort (generateReverseUVec 8) 'shouldBe' UV.fromList [1..2^8]
51
52     describe "Sort.parBatcherMergeSort" $ do
53         it "correctly sorts the input vector" $ do
54             Sort.parBatcherMergeSort 8 (generateReverseUVec 8) 'shouldBe' UV.fromList [1..2^8]

```