COMS 4995 W: Parallel Functional Programming
Optimizing Parallel Sorting

Jay Karp, Benjamin Flin - jlk2225, brf2117
December 23, 2020

1 Abstract

This paper presents an exploration into three main sequential sorting algorithms and their parallel counterparts. These include merge sort, quicksort, as well as bitonic sort. While bitonic sort is an inherently parallel algorithm, merge sort and quicksort on their own are not conventionally parallelized. In this paper we attempt to demonstrate both the potential performance gains of parallelized sorting in haskell, as well as the cleaner, easier to write and simpler code required to write these parallelizations in a purely functional programming language. Finally, we present our own hybrid sorting algorithm which is built upon the best features from the different sorts which we experimented with.

2 Introduction and Problem Statement

Sorting is one of the most fundamental concepts in computer science. It is one of the first topics many introductory classes teach to new computer science students, and yet at the same time it is so foundational that many of the systems we rely on daily are built on the concept of fast, efficient sorting. This includes things such as web indexing, supercomputing, and database management.

With the possibile slowing of Moore's law in the near future, it is also essential that we look at alternative methods to push these important algorithms even further. Because of this we decided to implement three main sorting algorithms using Haskell: merge sort, quicksort, and bitonic sort. Although it would probably be faster to write these sorts in a lower level language like C, we believe that the functional paradigm allows for much easier and more concise data parallelism.

After moving these three main sorts from their equivalent sequential and imperative counterparts, we further pushed the limits of sorting in Haskell to implement some of the different data parallelism models. Finally, we present 'hybrid' sort, which is composed of the best features from all of the sequential, and parallel sorts that we implemented.

3 Implementation

In this section we will discuss the specific design patterns and implementations which we used in our sorting implementations. This includes the general Haskell data structures which we used, as well as a discussion of both the sequential and parallel algorithms which we implemented. Along with this we will provide analysis of these algorithms imperative counterparts and justifications for our design decisions.

3.1 Haskell Data Structures

In our initial testing, our implementations were based on haskell's default lists. However, we quickly realized that all of our implementations would be better suited by a different data structure. This led us towards multiple alternatives, such as regular `Data.Arrays` and `Data.Array.REPA`. Ultimately we decided on implementing all of our solutions using Haskell's `Data.Vector`, as it provided us with a boxed data structure that has both mutable and immutable variants as well as `O(1)` indexing.

The other main data structure that we used during our testing is the `instance NFData Dumb` data structure. This data structure simply contained an int, but used a comparison function with a time cost proportional to its value. This allowed us to test the granularity of our algorithms to some extent, as we could simulate a large load without many elements in our unsorted sequence.

3.2 Sequential

After deciding on our use of vectors, our first step was to convert the pseudocode of sequential imperative code into sequential functional code in haskell. To do this we followed the following pseudocode for Merge Sort, Quicksort, and Bitonic Sort.

```
MergeSort(arr, left, right)
   if (left > right) do
   end
   mid = (left+right)/2
   MergeSort(arr, left, mid)
   MergeSort(arr, mid+1, right)
   Merge(arr, left, mid, right)
end
```

```
QuickSort(arr, low, high)
   if (low < high) do
      pi = partition(arr, low, high)

      QuickSort(arr, low, pi - 1)
      QuickSort(arr, pi + 1, high)
   end
end
```

```
BitonicSort(arr, low, cnt, dir)
  if (cnt > 1) do
    k = cnt / 2
    BitonicSort(a, low, k, 1)
    BitonicSort(a, low+k, k, 0)
    BitonicMerge(a,low, cnt, dir)
  end
end
```

While this step was not extremely challenging, it did come with its own problems. The first problem that we ran into was efficient memory use. Our initial implementations made use of Haskell's `Data.Vector.slice`, which creates a copy of a subset of an initial vector. We very quickly realized that this was creating an excessive number of vector copies and using an extreme amount of memory. This was substantially hindering performance and pushed us to further look into `Data.Vector.Mutable`. This exposed multiple extremely useful functions for in place data modification, specifically,

```
(1) read   :: PrimMonad m => MVector (PrimState m) a -> Int      -> m a
(2) write  :: PrimMonad m => MVector (PrimState m) a -> Int      -> a   -> m ()
```

```
(3) modify :: PrimMonad m => MVector (PrimState m) a -> (a -> a) -> Int -> m ()
(4) swap   :: PrimMonad m => MVector (PrimState m) a -> Int      -> Int -> m ()
```

This provided us with the lower level immutable operations, such as `O(1)` reading as well as writing and swapping vector values in place.

One change we made to the sequential versions of the algorithms was a small change to the default implementation of Merge Sort. While looking at the Haskell `Data.List.sort` implementation, which is a Merge Sort, we realized that they check for subsections of the list that are already sorted in ascending or descending order. If they are in ascending order, then the algorithm skips the rest of that subsection of the sort. If they are in descending order, then flip all of the elements in the vector. This saves a surprising amount of time and also makes best case sorts even faster.

The only other aspect of the sequential sorts which we had to implement was a function fillBitonic (`fillBitonic :: a -> V.Vector a -> V.Vector a`). We came to the realization that because of the parallel nature of the bitonic sorting network we needed to fill our input to our sequential bitonic sort with enough elements so that the size of the sorted vector is a power of 2. To do this we filled the sorted array with empty min values (such as `0` or `""`).

3.3 Parallel

Now that we had the sequential implementations finished, we could focus more specifically on the parallel implementations and optimizations. Most of our parallelism was done using Control.Parallel.Strategies, with the exception of bitonicPar, which used Control.Monad.Par.

3.3.1 Merge Sort

For our parallel merge sort implementation, we split our list into each into a sequence of sorted Vectors sequentially, and then recursively merge all of these Vectors by diving our sequence of Vectors in half and merging each half in parallel. For the top few levels we ran a special merge procedure called compare-exchange described below:

1. Create two tasks (sparks), each containing both lists to merge
2. Have one task compute the lower half of the merged lists, and the other compute the upper half.
3. Concatenate the resulting halves from each task into one list

This procedure was advantageous in the first few levels in our recursive merge, where there are many elements and less can be done in parallel. When merging was done deeper in the call stack, since there were less elements to merge per spark, but many more sparks, so this procedure swapped with our fast sequential one.

3.3.2 Quick Sort

Running quicksort in parallel involved use of the following procedure:

1. Split the sequence evenly among n tasks.
2. Choose a pivot and give it to each task.
3. Have each task partition its sequence into a pair of lists, one with elements less than the pivot, and the other with elements greater than the pivot.
4. Split the tasks into two halves. Have the first half of the tasks give their upper list to the second half and vice versa.
5. Concatenate the lists so we end up with n lists, with $\frac{n}{2}$ lists less than the pivot and $\frac{n}{2}$ greater than the pivot.
6. Recurse on each half in parallel and then concatenate the lists.
7. After $log(n)$ recursions, our base case is to use sequential sort.

Running this procedure in parallel gave good results, however, some time was lost in steps 3 and 4, as well as some time lost in concatenation.

3.3.3 Bitonic Sort

Bitonic sort was the only sort where we used `Control.Monad.Par.IO` instead, since we needed to do in-place swaps in memory, otherwise much time would be lost in copying and memory management associated with Vectors. The only way to do these in-place swaps in parallel is to run it in the `IO` monad, rather than the `ST s ()` monad since `ST` has knowledge of its sequential context in s. Parallelizing bitonic sort was a challenge, as our naive attempt to do each swap in parallel (as one would normally do in a sorting network) resulted in fine-grain parallelism, with enormous amounts of tasks taking very little time. Instead of parallelizing each swap, we instead batched computations by parallelizing recursive calls, with depth limiting.

3.3.4 Hybrid Sort

Hybrid sort was simply a combination of sequential quicksort and the merge operation with the compare-exchange procedure. We divided the list evenly among n tasks and then ran a sequential quicksort in each task in parallel. These lists were then concatenated using the parallel merge procedure used in the parallel merge sort algorithm. This scheme gave very good results as it had low granularity and very little communication between tasks.

4 Evaluation

4.1 Settings

We ran our experiments on an *Intel Core i7 3700k 3.50 GHz processor*.

4.2 Benchmarks

We performed many different benchmarks to test our different sorting implementations, however, we settled on two to discuss in this paper. First, we ran 128 trials of sorting a random permutation of the dictionary. Second, we ran 32 trials of sorting a uniformly distributed list of integers. To compare the speed of all of the algorithms we used 2^{20} integers, however, to compare the speed up when increasing the number of cores on our parallel implementations, we ran this sort on 2^{18} integers.

4.3 Results

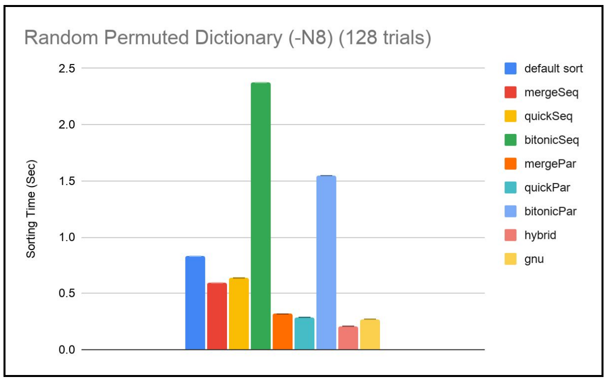


Figure 1

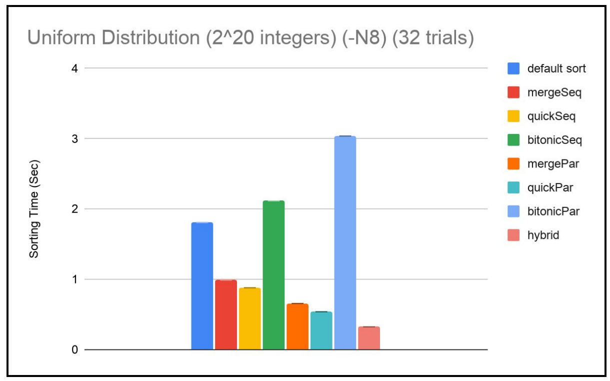


Figure 2

Our experiments show that in almost every case our parallel algorithms ran faster than our sequential implementations. The only exception is that parallel Bitonic sort was slower then its sequential counterpart when sorting integers. One other important thing to note is that our hybrid sort implementation was the fastest sorting algorithm in all of the different tests which we ran. Along with this, our implementation was 4x faster than Haskell's sequential implementation of `Data.List.sort` as well as ~25% faster than GNU sort on a random dictionary. GNU sort is completely written C and coming anywhere near this low level api was an unexpected result.[1]

4.4 Performance Analysis

4.4.1 Parallel Merge Sort

[1] We did not have access to any sort of GNU sort api, which means that these metrics are slightly unfair. GNU sort reads files in parallel and because of the lack of an api, it is difficult to time only the sort. We assume that are sort is pretty similar in time to GNU sort's implementation.

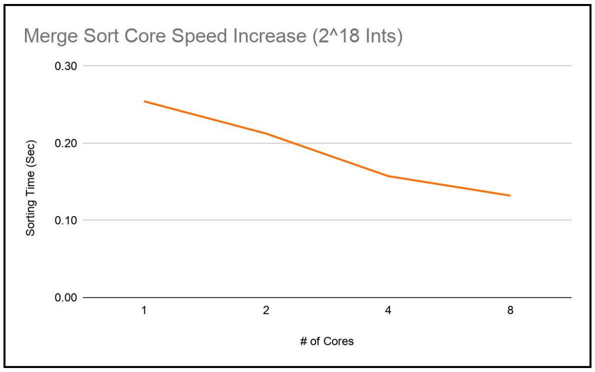
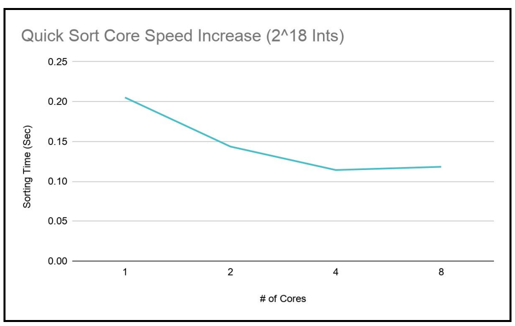


Figure 3

Figure 4: Eventlog for Parallel Merge Sort

This figure shows around a 2x speedup with 8 cores. Threadscope reveals the compare and exchange procedure happening on two cores at the end.

4.4.2 Parallel Quicksort



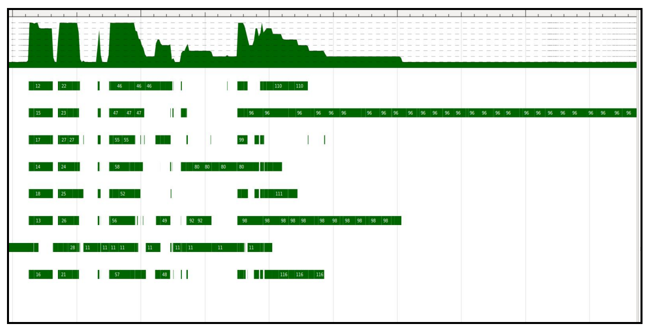
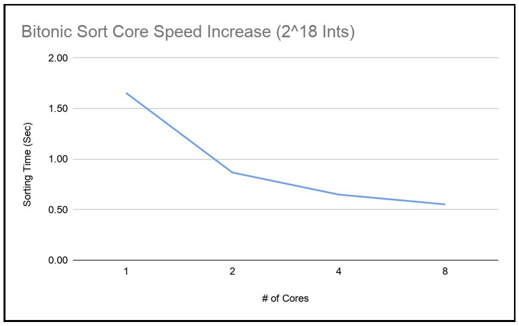


Figure 5

Figure 6: Eventlog for Parallel Quicksort

The figure shows around a 2x speedup with 8 cores. Threadscope reveals that the concatenation of the lists at the end limited our speedups.

4.4.3 Parallel Bitonic Sort



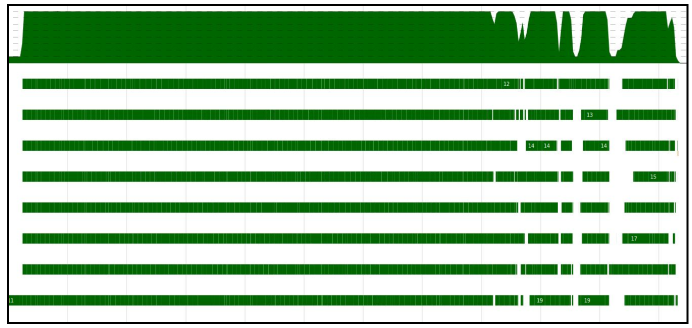
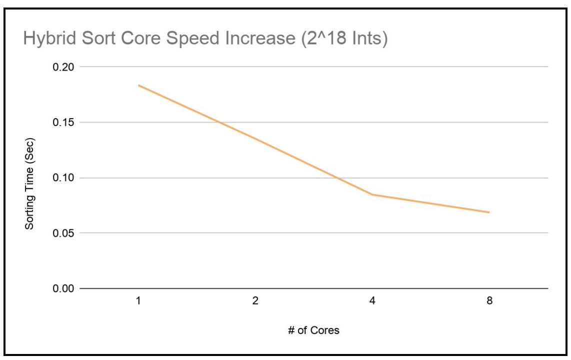


Figure 7

Figure 8: Eventlog for Parallel Bitonic Sort

Bitonic sort is very evenly distributed across 8 cores, however, it still remains slower than the rest. This resulted in around a 4x speedup.

4.4.4 Hybrid Sort



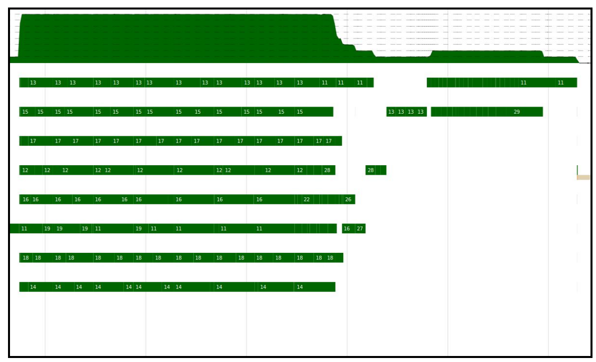


Figure 9

Figure 10: Eventlog for Hybrid Sort

The figure shows around a 3-4x speedup. Threadscope shows each quicksort being divided evenly among the 8 cores and then the compare and exchange sequence happening at the end.

5 Conclusions and Future Work

GNU sort is 4544 lines of C code. Our hybrid haskell sort implementation was ~30 lines. While we cannot give exact speeds of the two implementations, the fact these two parallel sorts were in close in speed is impressive. The ability to provide both high level API's for data parallelism and low level API's for data manipulation speaks to the robustness of parallel functional programming in Haskell. While there were challenges such as handling inter-task communication and memory mutability, our code ended up being relatively efficient, fast and clean.

One of the largest failures which we ran into during our project was the parallelization of Bitonic sort. Bitonic sort is optimized for parallel execution, because each compare and swap operation can be done at the same time. This would be extremely efficient if you had $\frac{n}{2}$ cores where $n = \textit{size of list}$ but because we only had 8 cores, our program seemed to spend more time parallelizing and scheduling computations than it did evaluating them. In our future work, Bitonic sort could be greatly improved by running the algorithm using Haskell GPU support for CUDA. While this would introduce different issues such as memory synchronization, it would overcome many of the problems that Bitonic sort faces when running on the CPU.

6 Usage

The program can be simply built using `stack build` from within the root directory. It can then be run with `stack exec -- par-sort-exe [OPTIONS]`. Running without options, or with either the `-h` or `-help` flag yields the following help text

```
Usage: parsort
-s [   default    | bitonicSeq | mergeSeq   | quickSeq   | bitonicPar |
       mergePar   | quickPar   | hybrid ]
-i [file]
-z [size]
Available Options:
       -s, -sort     Specifies which sort to perform
       -i, -input    [...file] Provides a pointer to a file with new-line separated
                     values to sort
       -z, -size     Specifies a uniformly distributed random array of 2^[size]
                     elements
       -h, -help     Shows the usage information
```

7 References

[1] 11.4 Mergesort
[2] Lecture 12: Parallel quicksort algorithms
[3] Bitonic Sort: Overview

A Code List

Main.hs

```haskell
module Main where

import             Control.DeepSeq     (NFData)
import             Data.List           (sort)
import             Data.String         (fromString)
import qualified Data.Vector           as V
```

```haskell
import           Lib                 (bitonicPar, bitonicSeq, hybridPar,
                                      mergePar, mergeSeq, quickPar, quickSeq,
                                      readLines, shuffle, time, timeIO)
import           System.Console.GetOpt
import           System.Environment  (getArgs)
import           System.Exit         (die)

data Arg
    = Sort String                    -- -s
    | Input String                   -- -i
    | Size String                    -- -z
    | Help                           -- --help
    deriving (Eq, Ord, Show)

options :: [OptDescr Arg]
options = [
    Option ['s'] ["sort"] (ReqArg Sort "") "default | bitonicSeq | mergeSeq | quickSeq | bitonicPar | mergePar | quickPar | hybrid"
    , Option ['i'] ["input"] (ReqArg Input "") "File path"
    , Option ['z'] ["size"] (ReqArg Size "") "2^z size of array to be sorted"
    , Option ['h'] ["help"] (NoArg Help) "Print this help message"
    ]


runFromArgs :: [String] -> IO ()
runFromArgs args = case opt of
    Help:_ -> die usage
    (Sort s):(Input f):_ -> do
          v <- readLines f
          if s == "bitonicSeq" || s == "bitonicPar" then
          runBitonic s (fromString "") v
          else
          runSort s v
    (Sort s):(Size z):_ -> do
          let n = read z :: Int
          v <- shuffle $ V.enumFromN (1 :: Int) (2^n)
          if s == "bitonicSeq" || s == "bitonicPar" then
          runBitonic s 0 v
          else
          runSort s v
    _ -> die usage
    where
    (opt,_,_) = getOpt Permute options args

usage :: String
usage = "Usage: parsort -s [default | bitonicSeq | mergeSeq | quickSeq | bitonicPar | mergePar | quickPar | hybrid] -i [file] -z [size]"
```

```haskell

runSort :: (NFData a, Ord a) => String -> V.Vector a -> IO ()
runSort "default"      v = time "Default Sort" (sort $ V.toList v)
runSort "quickSeq"   v = time "Sequential Quicksort" (quickSeq v)
runSort "mergeSeq"   v = time "Sequential Merge Sort" (mergeSeq v)
runSort "hybrid"       v = time "Parallel Hybrid Sort" (hybridPar v)
runSort "quickPar"   v = time "Parallel Quick Sort" (quickPar v)
runSort "mergePar"   v = time "Parallel Merge Sort" (mergePar v)
runSort _ _            = die usage

runBitonic :: (NFData a, Ord a) => String -> a -> V.Vector a -> IO ()
runBitonic "bitonicSeq" a v = time "Sequential Bitonic Sort" (bitonicSeq a v)
runBitonic "bitonicPar" a v = timeIO "Parallel Bitonic Sort" (bitonicPar a v)
runBitonic _ _ _              = die usage

main :: IO ()
main = getArgs >>= runFromArgs
```

Lib.hs

```haskell
module Lib (
    module Sequential,
    module Parallel,
    module Utils,
) where

import            Parallel
import            Sequential
import            Utils
```

Utils.hs

```haskell
{-# LANGUAGE DeriveGeneric #-}
module Utils
    (
    fillBitonic,
    readLines,
    shuffle,
    Dumb,
    time,
    timeIO
    )
    where

import            Control.DeepSeq     (NFData, force)
```

```haskell
import           Control.Monad       (forM_)
import qualified Data.ByteString     as B
import           Data.Time.Clock     (diffUTCTime, getCurrentTime)
import           Data.Vector         ((!))
import qualified Data.Vector         as V
import qualified Data.Vector.Mutable as M
import           GHC.Generics        (Generic)
import           System.IO           (IOMode (ReadMode), hIsEOF, withFile)
import           System.Random       (randomRIO)

shuffle :: V.Vector a -> IO (V.Vector a)
shuffle v = do
    let n = V.length v - 1
    js <- V.forM (V.enumFromTo 0 n) $ \i -> randomRIO (i, n)
    return $ V.create $ do
    o <- V.thaw v
    forM_ [1..n] $ \i -> M.swap o i (js!i)
    return o


fillBitonic :: a -> V.Vector a -> V.Vector a
fillBitonic a v = V.create $ do
    o <- V.thaw v
    let l = V.length v
    let n = 2 ^ (ceiling (logBase 2 (fromIntegral l) :: Double) :: Int) - l
    o' <- M.grow o n
    p <- M.replicate n a
    M.copy (M.slice l n o') p
    return o'

readLines :: String -> IO (V.Vector B.ByteString)
readLines filename = withFile filename ReadMode ((V.fromList <$>) . getLines)
    where
    getLines handle = do
          eof <- hIsEOF handle
          if eof then
          return []
          else
          (:) <$> B.hGetLine handle <*> getLines handle

newtype Dumb = Dumb Integer deriving (Generic, Show)

instance Eq Dumb where
    (Dumb 0) == (Dumb 0) = True
    (Dumb _) == (Dumb 0) = False
    (Dumb 0) == (Dumb _) = False
    (Dumb x) == (Dumb y) = Dumb (x-1) == Dumb (y-1)
```

```haskell

instance Ord Dumb where
    (Dumb x) <= (Dumb 0) = x <= 0
    (Dumb 0) <= (Dumb y) = y >= 0
    (Dumb x) <= (Dumb y) = Dumb (x-1) <= Dumb (y-1)

instance Num Dumb where
    (+) (Dumb a) (Dumb b) = Dumb $ a + b
    (*) (Dumb a) (Dumb b) = Dumb $ a * b
    abs (Dumb a) = Dumb $ abs a
    fromInteger = Dumb
    negate (Dumb a) = Dumb $ -a
    signum (Dumb a) = Dumb $ signum a

instance NFData Dumb

time :: (NFData a) => String -> a -> IO ()
time msg a = do
    start <- getCurrentTime
    let a' = force a
    end <- a' `seq` getCurrentTime
    putStrLn $ msg ++ ": " ++ show (diffUTCTime end start)

timeIO :: NFData a => [Char] -> IO a -> IO ()
timeIO msg io = do
    start <- getCurrentTime
    a <- io
    end <- force a `seq` getCurrentTime
    putStrLn $ msg ++ ": " ++ show (diffUTCTime end start)
```

Sequential.hs

```haskell
module Sequential (bitonicSeq, mergeSeq, quickSeq) where

import           Control.Monad       (when)
import           Data.Vector         ((!))
import qualified Data.Vector         as V
import qualified Data.Vector.Mutable as M
import           Utils               (fillBitonic)

bitonicSeq :: Ord a => a -> V.Vector a -> V.Vector a
bitonicSeq = (bitonic .) . fillBitonic

bitonic :: Ord a => V.Vector a -> V.Vector a
bitonic v = V.create $ do
```

```
    o <- V.thaw v
    bitonicSort' o 0 (V.length v) True
    return o
    where
        bitonicSort' o low cnt dir =
            when (cnt > 1) $ do
                let k = cnt `div` 2
                bitonicSort' o low k True
                bitonicSort' o (low + k) k False
                bitonicMerge o low cnt dir
        bitonicMerge o low cnt dir =
            when (cnt > 1) $ do
                let k = cnt `div` 2
                loopSwap o low low k dir
                bitonicMerge o low k dir
                bitonicMerge o (low+k) k  dir
        loopSwap o low i k dir =
            when (i < low + k) $ do
                compareAndSwap o i (i+k) dir
                loopSwap o low (i+1) k dir
        compareAndSwap o i j dir = do
            oi <- M.read o i
            oj <- M.read o j
            when (dir == (oi > oj)) $ M.swap o i j

mergeSeq :: Ord a => V.Vector a -> V.Vector a
mergeSeq = merge . runs

runs :: Ord a => V.Vector a -> V.Vector (V.Vector a)
runs x = V.create $ do
    o <- M.new (V.length x)
    runs' 1 x 0 o
    where
        runs' i v k o
            | i < V.length v =
                if v!(i-1) <= v!i then
                    asc (i-1) i k o
                else
                    dsc (i-1) i k o
            | otherwise = return $ M.slice 0 k o
        asc s i k o =
            if i < V.length x && x!(i-1) <= x!i then
                asc s (i+1) k o
            else do
                M.write o k (V.slice s (i-s) x)
                runs' (i+1) x (k+1) o
        dsc s i k o =
```

```haskell
            if i < V.length x && x!(i-1) > x!i then
                dsc s (i+1) k o
            else do
                M.write o k (V.reverse $ V.slice s (i-s) x)
                runs' (i+1) x (k+1) o

merge :: Ord a => V.Vector (V.Vector a) -> V.Vector a
merge v = (!0) $ V.create $ do
    o <- V.thaw v
    mergeAll (V.length v) o
    return $ M.slice 0 1 o
    where
        mergeAll k o
            | k == 1 = return ()
            | otherwise = do
                k' <- mergePairs 0 k o
                mergeAll k' o
        mergePairs i k o
            | i < k - 1 = do
                oi <- M.read o i
                oip1 <- M.read o (i+1)
                M.write o (i `div` 2) (merge2 oi oip1)
                mergePairs (i+2) k o
            | i == k - 1 = do
                oi <- M.read o i
                M.write o (i `div` 2) oi
                return $ k `div` 2 + 1
            | otherwise = return $ k `div` 2

merge2 :: Ord a => V.Vector a -> V.Vector a -> V.Vector a
merge2 a b = V.create $ do
    v <- M.new (V.length a + V.length b)
    a' <- V.thaw a
    b' <- V.thaw b
    go a' b' 0 0 v
    return v
        where go a' b' i j v
                | i < V.length a && j < V.length b = do
                    ai <- M.unsafeRead a' i
                    bj <- M.unsafeRead b' j
                    if ai <= bj then do
                        M.unsafeWrite v (i+j) ai
                        go a' b' (i+1) j v
                    else do
                        M.unsafeWrite v (i+j) bj
                        go a' b' i (j+1) v
                | i < V.length a = do
```

```haskell
                    ai <- M.unsafeRead a' i
                    M.unsafeWrite v (i+j) ai
                    go a' b' (i+1) j v
                | j < V.length b = do
                    bj <- M.unsafeRead b' j
                    M.unsafeWrite v (i+j) bj
                    go a' b' i (j+1) v
                | otherwise = return ()


quickSeq :: Ord a => V.Vector a -> V.Vector a
quickSeq x = V.create $ do
    x' <- V.thaw x
    quickSort' x' 0 (V.length x - 1)
    return x'
    where
    quickSort' v low high
        | low < high = do
            i <- partition v low high
            quickSort' v low (i - 1)
            quickSort' v (i + 1) high
        | otherwise = return ()
    partition v low high = do
        i <- go (low - 1) low
        M.swap v (i+1) high
        return $ i + 1
        where
            go i j
                | j < high = do
                    vj <- M.read v j
                    pivot <- M.read v high
                    if vj < pivot then do
                        M.swap v (i+1) j
                        go (i+1) (j+1)
                    else
                        go i (j+1)
                | otherwise = return i
```

Parallel.hs

```haskell
module Parallel (bitonicPar, mergePar, hybridPar, quickPar) where
import                  Control.DeepSeq           (force)
import                  Control.Monad             (when)
import                  Control.Monad.IO.Class
import                  Control.Monad.Par.Class
import                  Control.Monad.Par.IO
```

```haskell
import           Control.Parallel.Strategies
import           Data.List.Split           (chunksOf)
import           Data.Vector               ((!))
import qualified Data.Vector               as V
import qualified Data.Vector.Mutable       as M
import qualified Data.Vector.Split         as S
import           Sequential                (quickSeq)
import           Utils                     (fillBitonic)

bitonicPar :: (NFData a, Ord a) => a -> V.Vector a -> IO (V.Vector a)
bitonicPar = (bitonic .) . fillBitonic

bitonic :: Ord a => V.Vector a -> IO (V.Vector a)
bitonic v = do
    o <- V.thaw v
    runParIO $ bitonicSort' o 0 (V.length v) True (0 :: Integer)
    V.freeze o
    where
          bitonicSort' o low cnt dir l =
          when (cnt > 1) $ do
                let k = cnt `div` 2
                if l < 7 then do
                a <- spawn $ bitonicSort' o low k True (l+1)
                b <- spawn $ bitonicSort' o (low + k) k False (l+1)
                get a
                get b
                else do
                bitonicSort' o low k True (l+1)
                bitonicSort' o (low + k) k False (l+1)
                bitonicMerge o low cnt dir l
          bitonicMerge o low cnt dir l =
          when (cnt > 1) $ do
                let k = cnt `div` 2
                loopSwap o low low k dir
                if l < 7 then do
                a <- spawn $ bitonicMerge o low k dir (l+1)
                b <- spawn $ bitonicMerge o (low+k) k dir (l+1)
                get a
                get b
                else do
                bitonicMerge o low k dir (l+1)
                bitonicMerge o (low+k) k dir (l+1)
          loopSwap o low i k dir =
          when (i < low + k) $ do
                loopSwap o low (i+1) k dir
                liftIO $ compareAndSwap o i (i+k) dir
          compareAndSwap o i j dir = do
```

```haskell
        oi <- M.read o i
        oj <- M.read o j
        when (dir == (oi > oj)) $ M.swap o i j

hybridPar :: (NFData a, Ord a) => V.Vector a -> V.Vector a
hybridPar v = merge $ V.fromList $ parMap rdeepseq quickSeq chunks
    where
    n = V.length v
    chunks = S.chunksOf (n `div` 32) v

quickPar :: (NFData a, Ord a) => V.Vector a -> V.Vector a
quickPar x = runEval $ quickPar' chunks
    where
    quickPar' :: (NFData a, Ord a) => [V.Vector a] -> Eval (V.Vector a)
    quickPar' [] = return V.empty
    quickPar' [v] = rpar (quickSeq v)
    quickPar' (v:vs) = do
        let p = V.head v
        vs' <- parList rdeepseq (V.partition (<p) <$> (V.tail v:vs))
        lower <- parList rdeepseq (fst <$> vs')
        upper <- parList rdeepseq (snd <$> vs')
        lower' <- parList rdeepseq (filter (not . null) $ V.concat <$> chunksOf 2 lower)
        upper' <- parList rdeepseq (filter (not . null) $ V.concat <$> chunksOf 2 upper)
        lower'' <- quickPar' lower'
        upper'' <- quickPar' upper'
        rpar ((lower'' `V.snoc` p) V.++ upper'')
    n = V.length x
    chunks = S.chunksOf (n `div` 32) x

mergePar :: (NFData a, Ord a) => V.Vector a -> V.Vector a
mergePar = merge . runs

runs :: Ord a => V.Vector a -> V.Vector (V.Vector a)
runs x = V.create $ do
    o <- M.new (V.length x)
    runs' 1 x 0 o
    where
        runs' i v k o
        | i < V.length v =
            if v!(i-1) <= v!i then
            asc (i-1) i k o
            else
            dsc (i-1) i k o
        | otherwise = return $ M.slice 0 k o
        asc s i k o =
```

```
        if i < V.length x && x!(i-1) <= x!i then
            asc s (i+1) k o
        else do
            M.write o k (V.slice s (i-s) x)
            runs' (i+1) x (k+1) o
        dsc s i k o =
        if i < V.length x && x!(i-1) > x!i then
            dsc s (i+1) k o
        else do
            M.write o k (V.reverse $ V.slice s (i-s) x)
            runs' (i+1) x (k+1) o


merge :: (NFData a, Ord a) => V.Vector (V.Vector a) -> V.Vector a
merge x = runEval (merge' (0::Integer) x)
    where
    merge' l v
        | n > 1 = do
        a' <- merge' (l+1) a >>= if l < 15 then rpar else rseq
        b' <- merge' (l+1) b >>= if l < 15 then rpar else rseq
        if l < 1 then merge2Par a' b' else return $ merge2 a' b'
        | otherwise = return $ v!0
        where
        n = V.length v
        a = V.slice 0 (n `div` 2) v
        b = V.slice (n `div` 2) (n - n `div` 2) v

merge2 :: Ord a => V.Vector a -> V.Vector a -> V.Vector a
merge2 a b = V.create $ do
    v <- M.new (V.length a + V.length b)
    a' <- V.thaw a
    b' <- V.thaw b
    go a' b' 0 0 v
    return v
        where go a' b' i j v
            | i < V.length a && j < V.length b = do
                ai <- M.unsafeRead a' i
                bj <- M.unsafeRead b' j
                if ai <= bj then do
                M.unsafeWrite v (i+j) ai
                go a' b' (i+1) j v
                else do
                M.unsafeWrite v (i+j) bj
                go a' b' i (j+1) v
            | i < V.length a = do
                ai <- M.unsafeRead a' i
                M.unsafeWrite v (i+j) ai
```

```haskell
                          go a' b' (i+1) j v
                    | j < V.length b = do
                          bj <- M.unsafeRead b' j
                          M.unsafeWrite v (i+j) bj
                          go a' b' i (j+1) v
                    | otherwise = return ()

merge2Par :: (NFData a, Ord a) => V.Vector a -> V.Vector a -> Eval (V.Vector a)
merge2Par a b = do
   l <- rpar (force lower)
   u <- rpar (force upper)
   return (l V.++ u)
   where
          n = V.length a + V.length b
          h = n `div` 2
          third (_,_,x) = x
          lower = third <$> V.postscanl' accumLower (0,0,undefined) (V.enumFromN (0::Integer) h)
          accumLower (i, j, _) _
          | i < V.length a && j < V.length b =
                if a!i <= b!j then
                      (i+1, j, a!i)
                else
                      (i, j+1, b!j)
          | i < V.length a = (i+1, j, a!i)
          | otherwise = (i, j+1, b!j)
          upper = V.reverse $ third <$> V.postscanl' accumUpper (V.length a - 1,V.length b - 1,undefined) (V.enumFromN (0::Integer) (n-h))
          accumUpper (i, j, _) _
          | i > 0 && j > 0 =
                if a!i >= b!j then
                      (i-1, j, a!i)
                else
                      (i, j-1, b!j)
          | i > 0 = (i-1, j, a!i)
          | otherwise = (i, j-1, b!j)
```