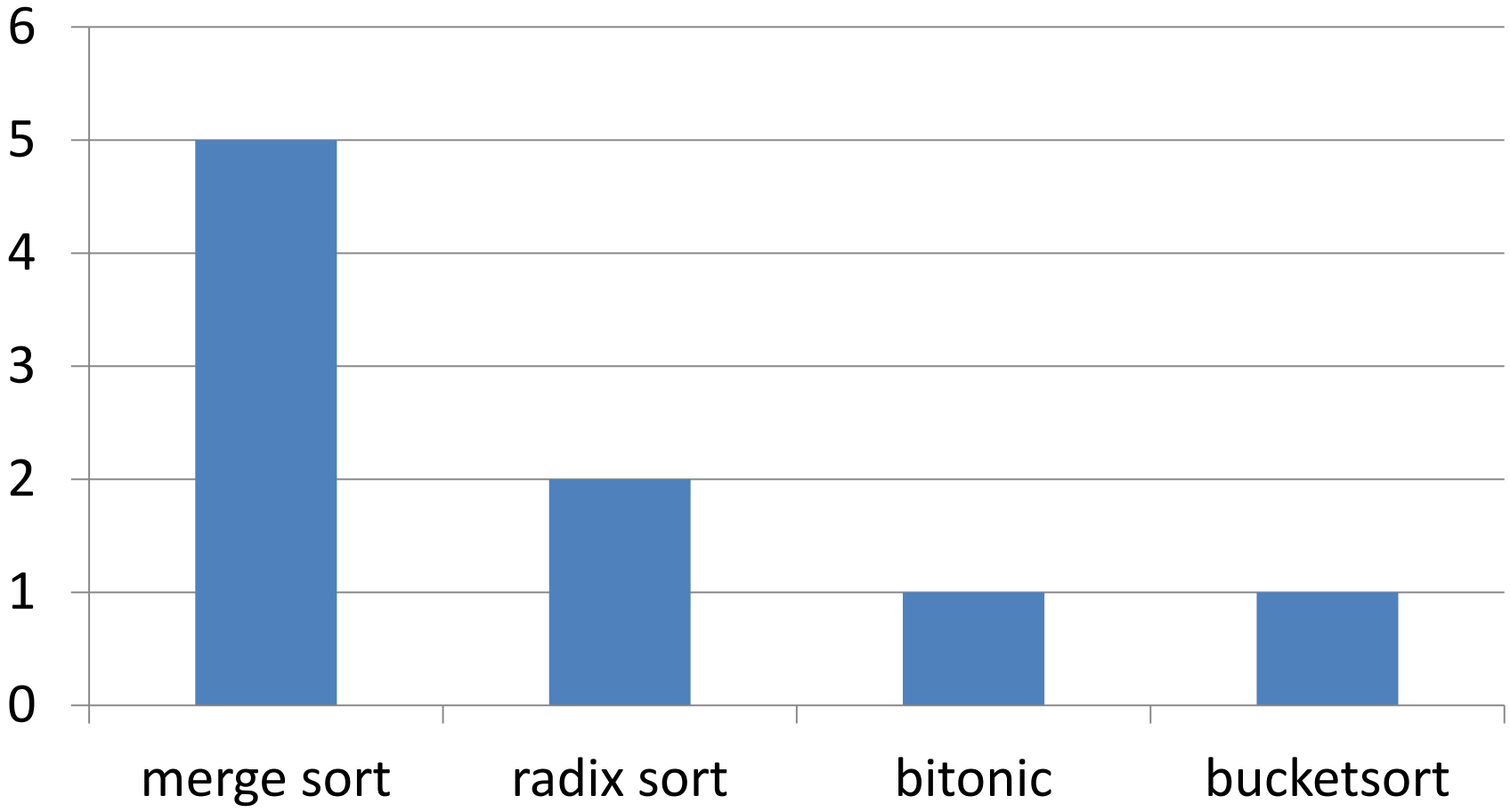
The Parallel Sorting Challenge

The Entries

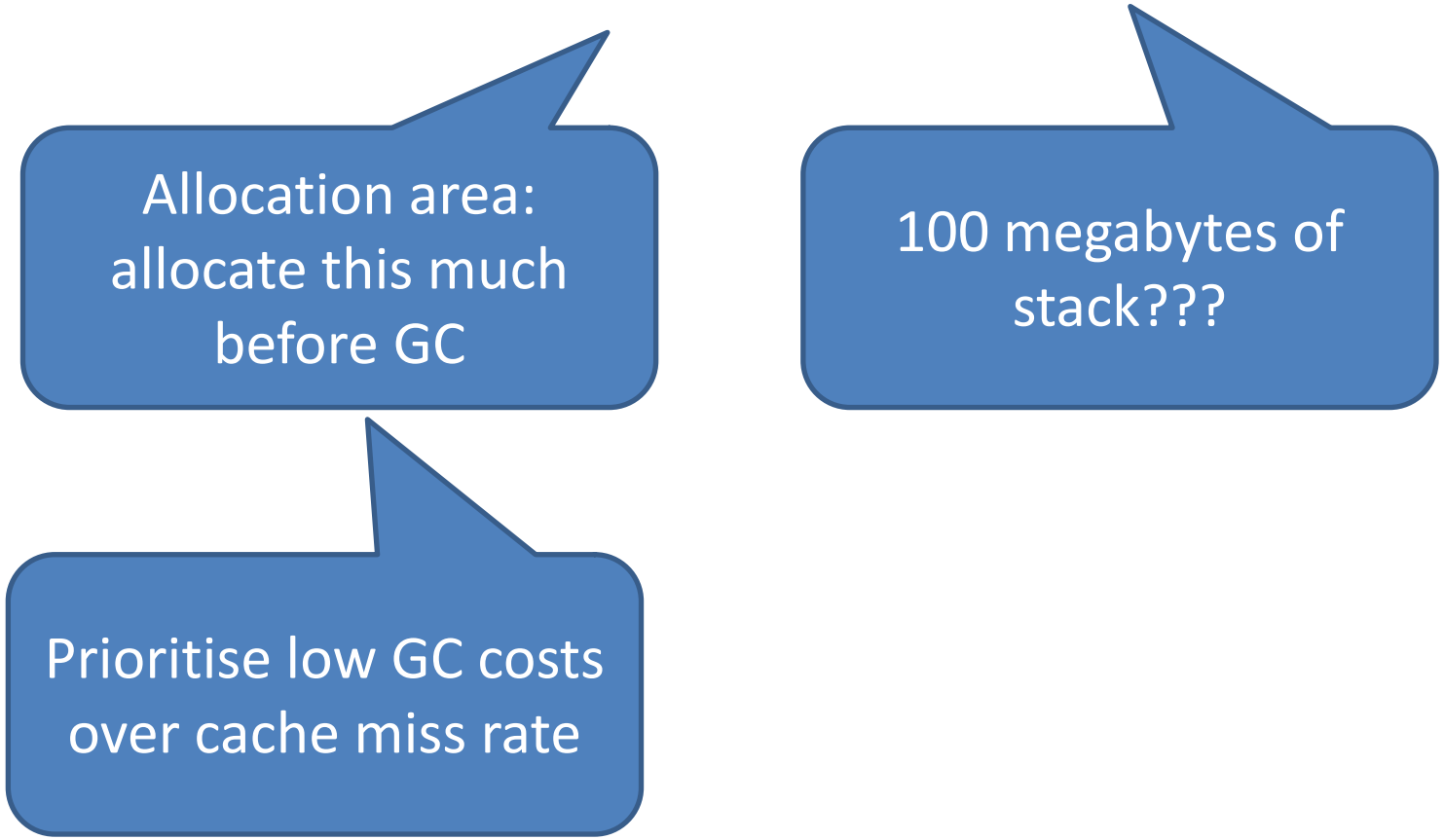
- 7 teams
- 9 participants
- 9 sorters
- 16 entries (differing depths)
- 2 using strategies

The Algorithms

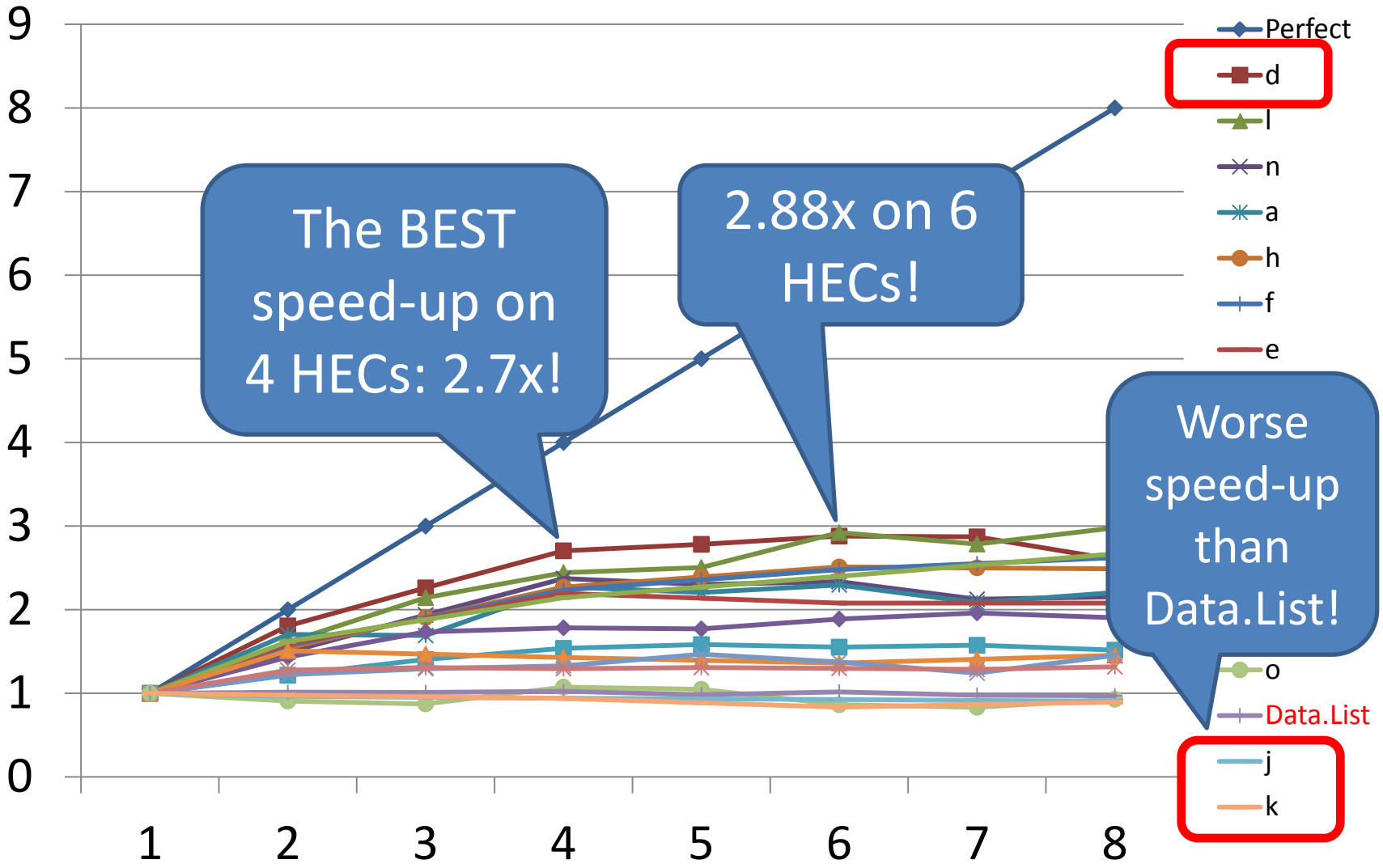


The Flags

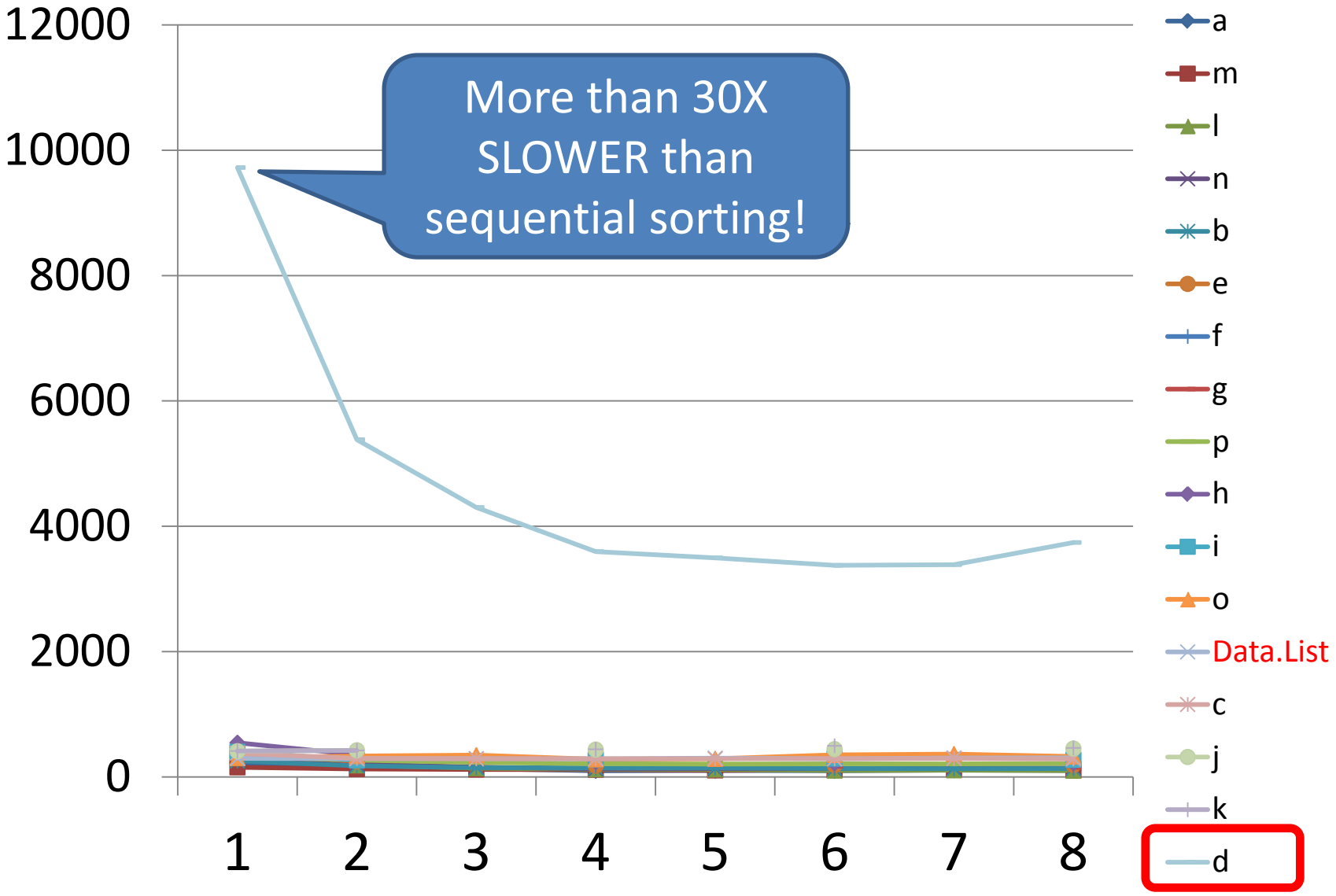
+RTS –A128M –K100M



The Speed-Ups



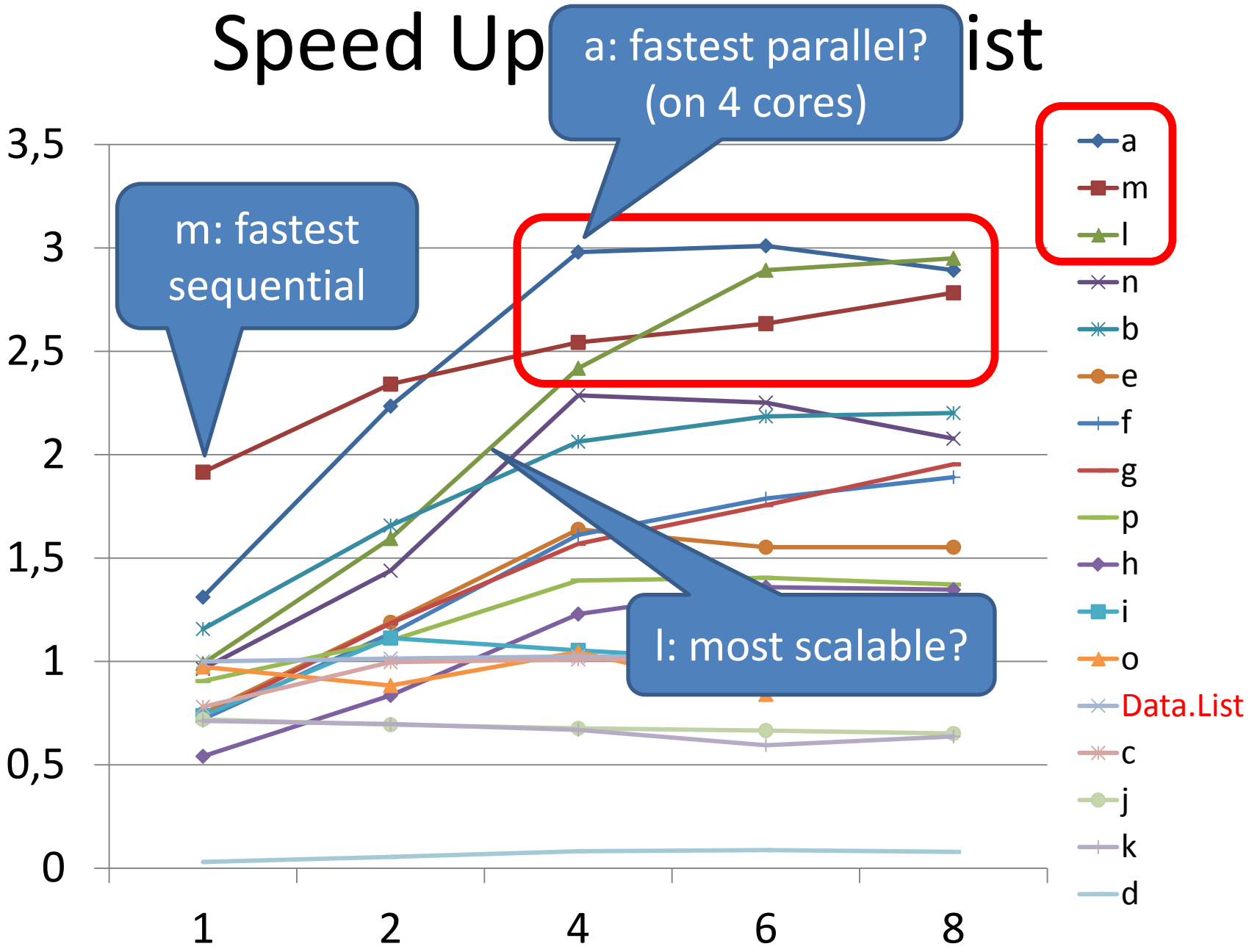
The Speeds


12000
10000
8000
6000
4000
2000
0
1
2
3
4
5
6
7
8
More than 30X SLOWER than sequential sorting!
a
m
l
n
b
e
f
g
p
h
i
o
Data.List
c
j
k
d

The Bitonic Sorter

- Very interesting algorithm
- Highly parallelisable
- Data independent control
  ➔ excellent for hardware implementation
- Worse complexity ☹

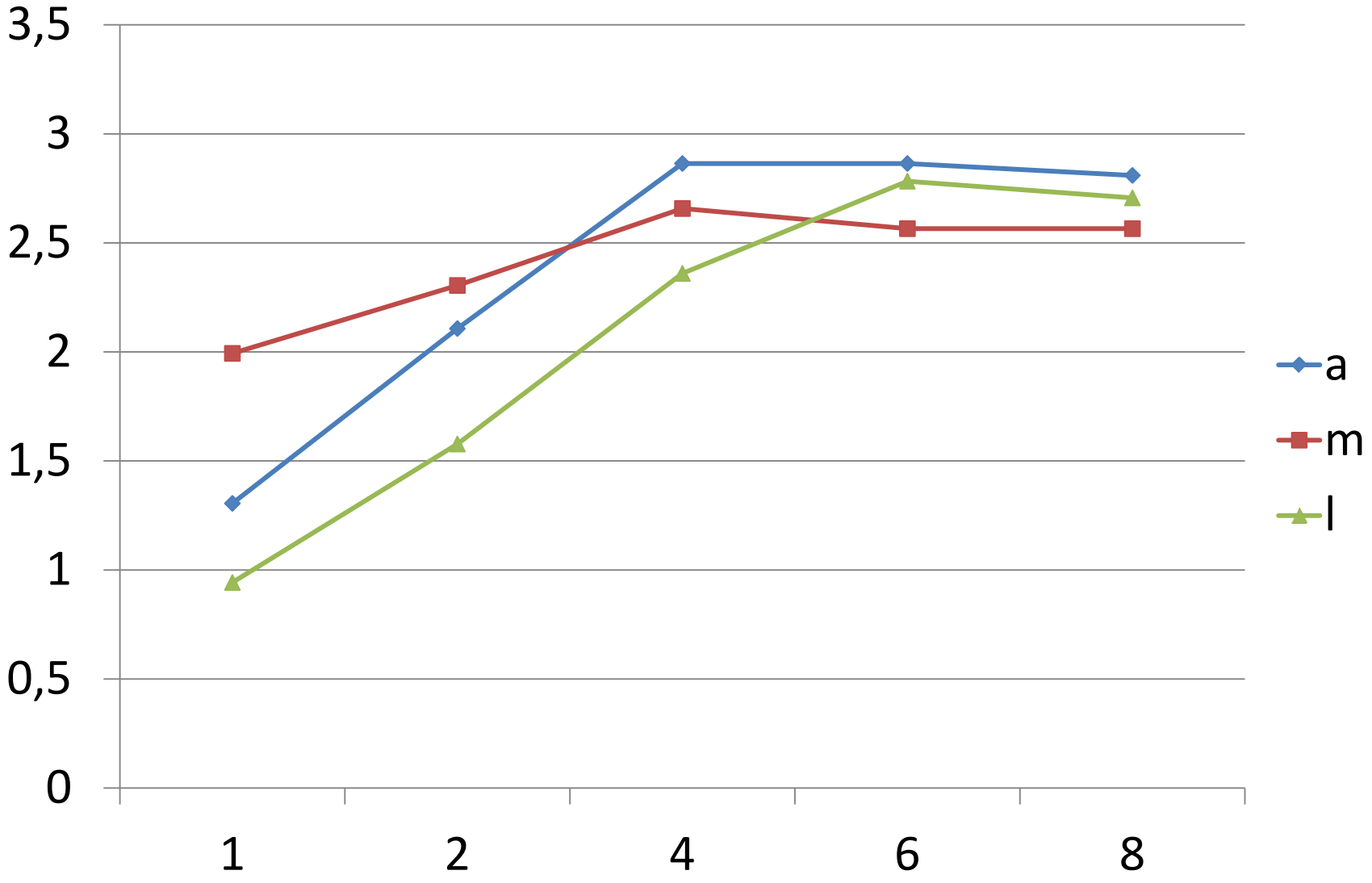
Speed Up ist


a: fastest parallel? (on 4 cores)
m: fastest sequential
l: most scalable?
3,5
3
2,5
2
1,5
1
0,5
0
1
2
4
6
8
a
m
l
n
b
e
f
g
p
h
i
o
Data.List
c
j
k
d

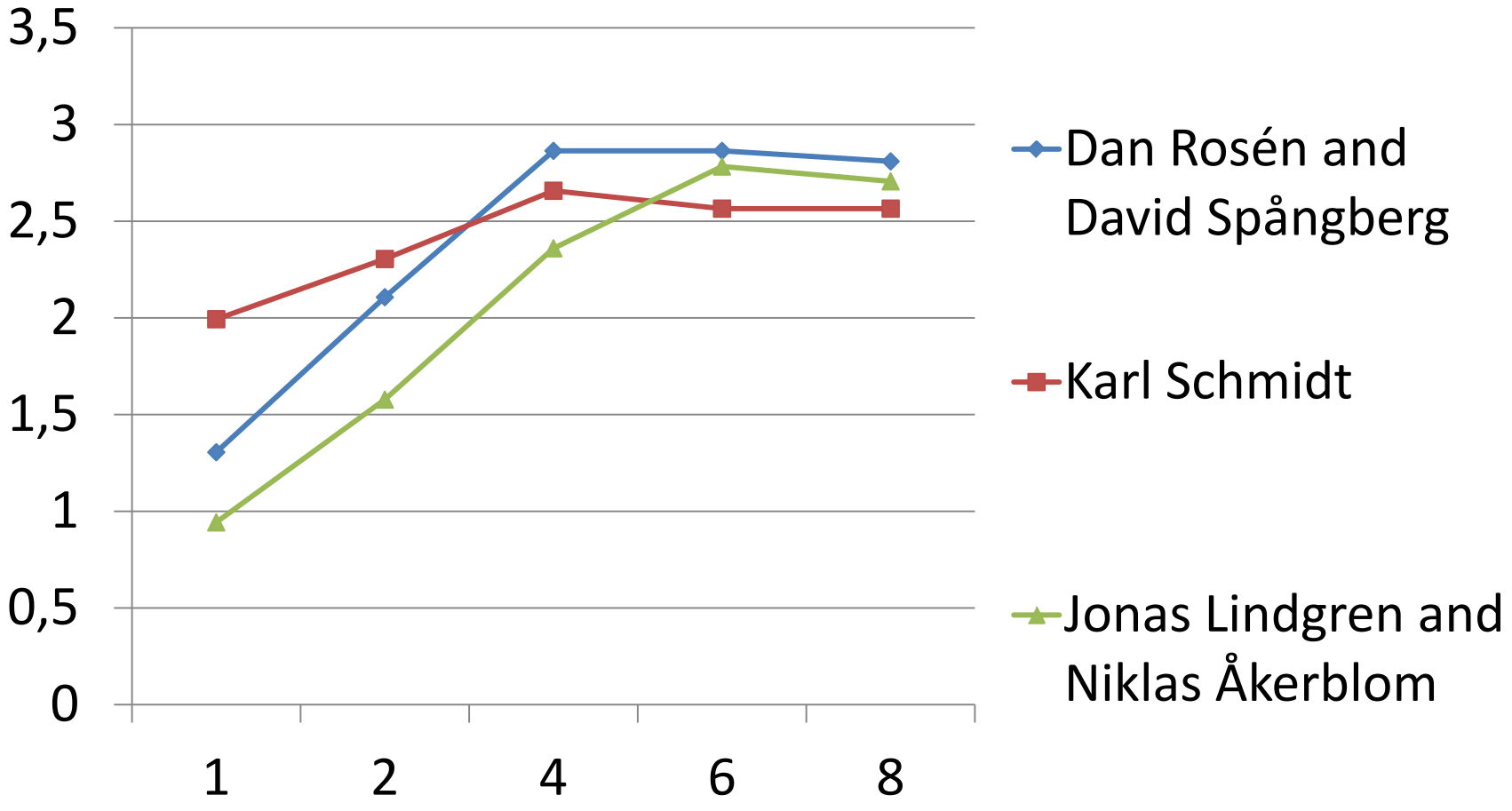
More careful benchmark

- 1000 samples
- Highest priority process
  - Try to minimise interference from other OS tasks

Speed Up over Data.List



The Winners!


3,5
3
2,5
2
1,5
1
0,5
0
1
2
4
6
8
Dan Rosén and David Spångberg
Karl Schmidt
Jonas Lindgren and Niklas Åkerblom

Dan Rosén and David Spångberg

```
-- Variation on radix sort.
bsort' :: forall a . (NFData a,Bits a,Ord a) => [a] -> [a]
bsort' xs = DL.toList (bucket t xs)
  where
    t :: Int
    t = 2

    bits :: Int
    bits = bitSize (undefined :: a)

    swap (a,b) = (b,a) -- uncurry (flip (,))

    bucket :: Int -> [a] -> DL.DList a
    bucket 0 xs = let s = sort xs in rnf s `pseq` DL.fromList s
    bucket d xs = let (u,l) = (if d == t then swap else id)
                              (partition (`testBit` (bits - t - 1 + d)) xs)
                      us = bucket (d-1) u
                      ls = bucket (d-1) l
                  in  us `par` ls `pseq` (ls `DL.append` us)

bsort'IntegerCheating :: [Integer] -> [Integer]
bsort'IntegerCheating xs = map toInteger
                         $ (bsort' :: [Int] -> [Int])
                         $ map fromInteger
                         $ xs
```

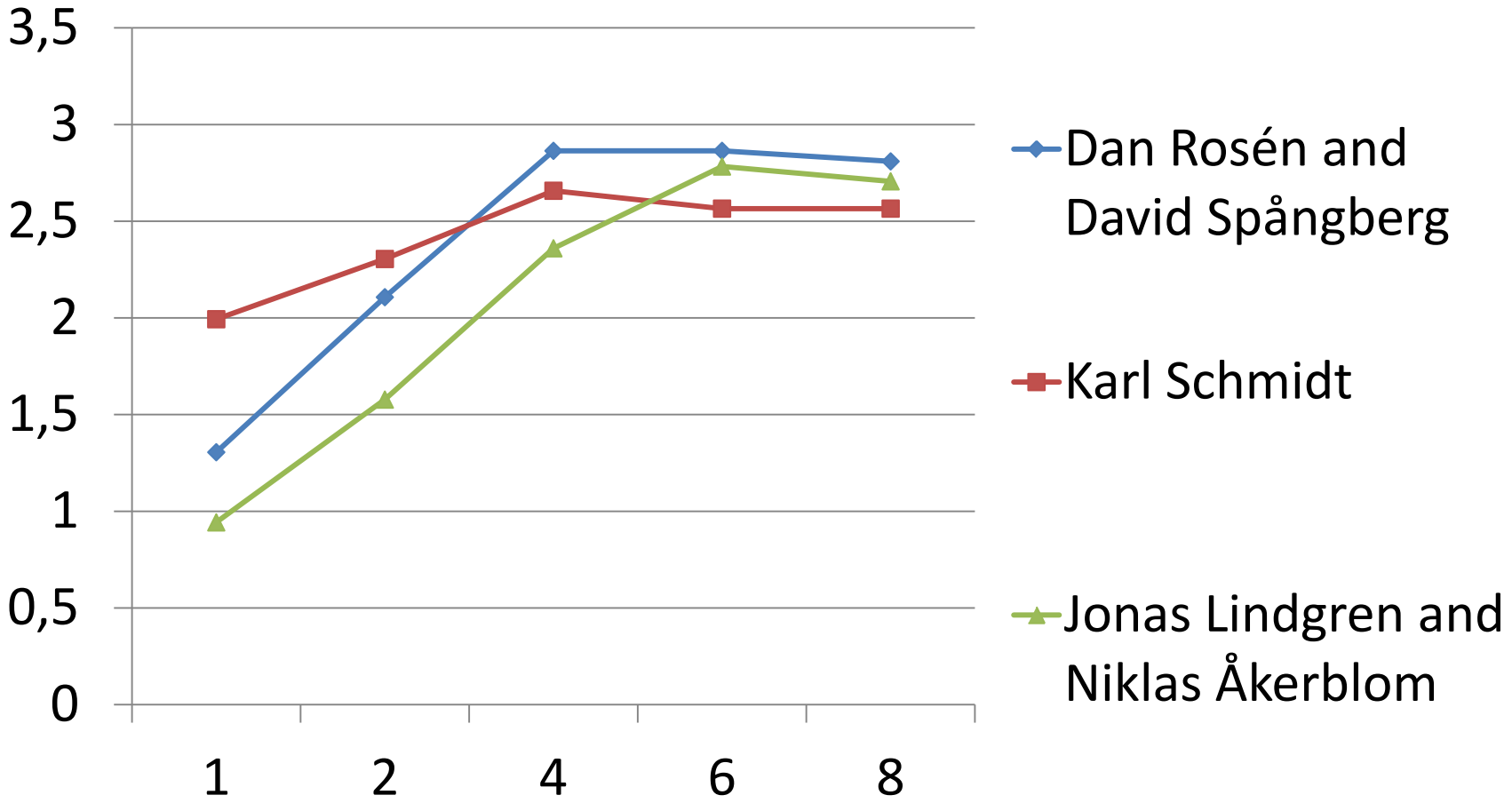
Karl Schmidt

```
bucketsort :: [Integer] -> [Integer]
bucketsort = pbucketsort buckets minmax Data.List.sort
  where minmax  = (toInteger (minBound::Int), toInteger (maxBound::Int))
        buckets = 8*1024

pbucketsort n (xmin,xmax) sort xs = pconcat pdepth bounds $ fmap sort
buckets
  where range      = xmax - xmin + 1
        n'         = min n range    -- bound number of buckets
        bucketSize = range `div` n'
        index i    = i `div` bucketSize
        bounds     = ( xmin `div` bucketSize, xmax `div` bucketSize )
        buckets    = bucketize index bounds xs
        pdepth     = floor $ log $ fromIntegral $ 5*numCapabilities

bucketize index bounds xs =
  accumArray (flip (:)) [] bounds $ map (\x -> (index x,x)) xs

pconcat depth (amin,amax) arr
  | amin == amax =  arr ! amin
  | depth <= 0   = (arr ! amin) ++ (pconcat depth (amin+1, amax) arr)
  | otherwise    = runEval $ do -- divide, parallelize, concatenate
    let amid = (amax + amin) `div` 2
    ys <- rpar `dot` rdeepseq $ pconcat (depth-1) (amid+1, amax) arr
    xs <- rseq                $ pconcat (depth-1) (amin  , amid) arr
    return $ xs ++ ys
```

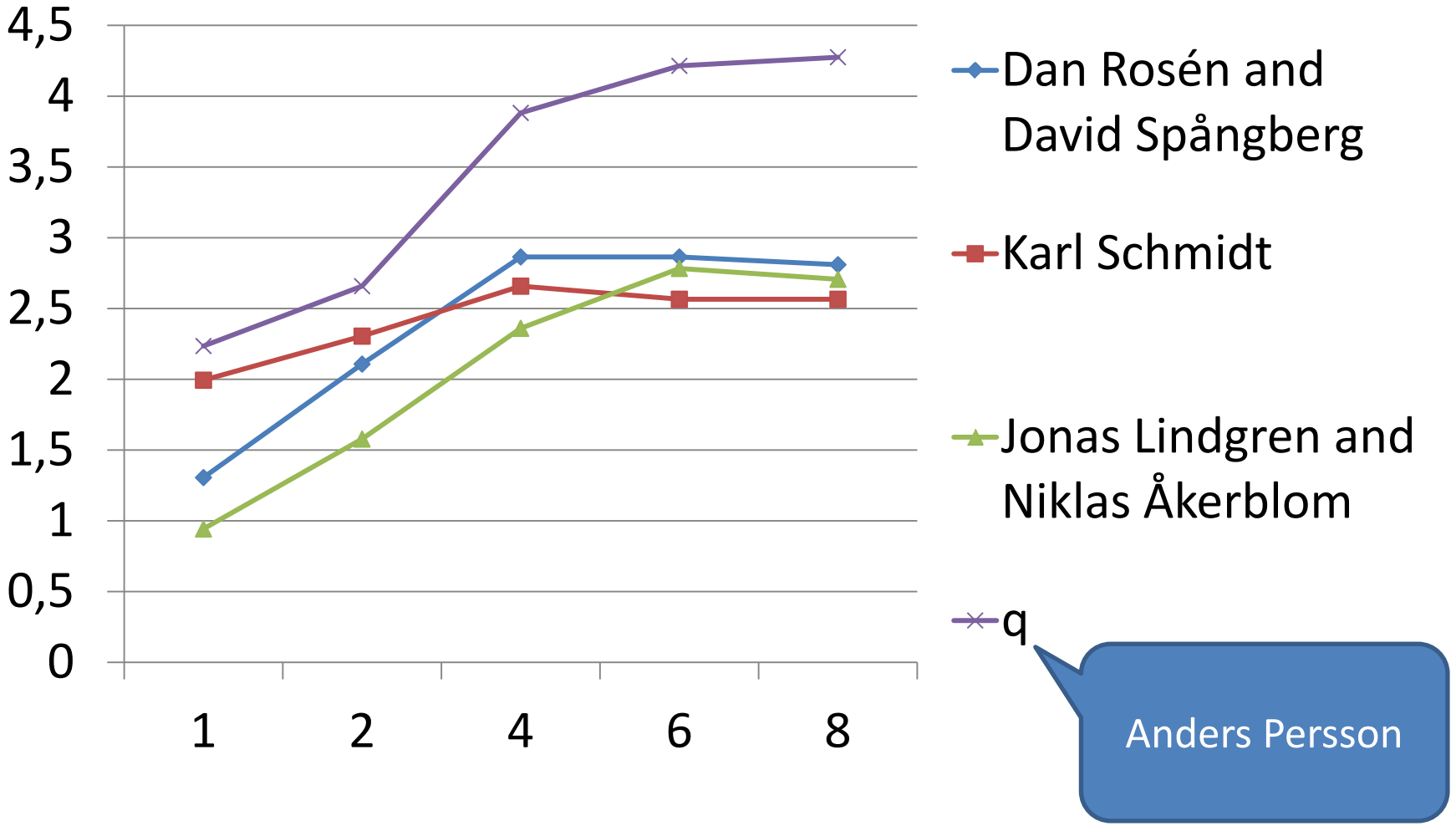
Jonas Lindgren and Niklas Åkerblom

```
sort :: (NFData a, Ord a)=> [a] -> [a]
sort xs = mpsort (xs,11)

msort [] = []
msort (x:[]) = [x]
msort xs = merge (msort p1) (msort p2)
        where (p1,p2 ) = splitAt ( div (length xs) 2) xs

mpsort ([], _) = []
mpsort ((x:[]), _ )= [x]
mpsort (xs, 0) = msort xs
mpsort (xs, n) = par (rnf p2res) (pseq (rnf p1res) (merge p1res p2res))
        where {p2res = mpsort (p2, (n-1))
              ;p1res = mpsort (p1, (n-1))
              ;(p1,p2) = splitAt ( div ((length xs)+1) 2) xs
              }

merge [] [] = []
merge [] xs = xs
merge xs [] = xs
merge (x:xs) (y:ys) = case x<y of
                          True -> x:merge xs (y:ys)
                          False ->y:merge (x:xs) ys
```

The Winners!


3,5
3
2,5
2
1,5
1
0,5
0
1
2
4
6
8
Dan Rosén and David Spångberg
Karl Schmidt
Jonas Lindgren and Niklas Åkerblom

The Mysterious Q



Anders Persson

```
qsortkpd :: Integer -> [Integer] -> [Integer]
qsortkpd limit xs = go limit xs
  where
    go _ []     = []
    go _ [x]    = [x]
    go _ [x,y]  = if x > y then [y,x] else [x,y]
    go 0 xs     = qsort3 xs
    go d (p:xs) = rnf g `par` e `par` l `pseq` (l ++ e ++ g)
      where
        l = go (d-1) lesser
        e = equal
        g = go (d-1) greater
        (lesser, equal, greater) = DL.foldl' part ([], [p], []) xs
        part (!l, !e, !g) x =
          case compare x p of
            LT -> (x:l,   e,   g)
            GT -> (  l,   e, x:g)
            EQ -> (  l, x:e,   g)
```

Anders Persson ctd

```
qsort3 xs = qcat xs []
  where
    qcat (x:xs) zs = part x xs zs [] [] []
    qcat []     zs = zs
    qapp (x:xs) zs = x:qapp xs zs
    qapp []     zs = zs
    part x []     zs !a !b !c = qcat a $ qapp (x : b) $ qcat c zs
    part x (y:ys) zs !a !b !c =
        case compare y x of
                  LT -> part x ys zs (y:a) b c
                  EQ -> part x ys zs a (y:b) c
                  GT -> part x ys zs a b (y:c)
```

Well done, all!