Group 16
Joel Olausson & Sebastian Hermansson

We are running our tests on an 8th gen i7 laptop CPU and using 4 cores.

Assignment 1

Below you can see the benchmarking done on all jackknife functions that implement the different types of parallel maps. Only sJackknife and pmJackknife actually manage to outperform the sequential Jackknife function. Likely this is due to how the computer handles sparks depending on which parallelization techniques are used.

```
benchmarking jackknife
time                 386.8 ms   (371.3 ms .. 400.3 ms)
                     1.000 R²   (0.999 R² .. 1.000 R²)
mean                 421.8 ms   (407.5 ms .. 440.6 ms)
std dev              22.12 ms   (3.548 ms .. 29.46 ms)
variance introduced by outliers: 19% (moderately inflated)

benchmarking pJackknife
time                 425.3 ms   (366.2 ms .. 457.9 ms)
                     0.998 R²   (0.994 R² .. 1.000 R²)
mean                 437.2 ms   (428.8 ms .. 442.2 ms)
std dev              8.320 ms   (3.069 ms .. 11.47 ms)
variance introduced by outliers: 19% (moderately inflated)

benchmarking rJackknife
time                 428.9 ms   (419.6 ms .. 446.6 ms)
                     1.000 R²   (1.000 R² .. 1.000 R²)
mean                 427.7 ms   (424.9 ms .. 429.9 ms)
std dev              2.715 ms   (33.34 μs .. 3.238 ms)
variance introduced by outliers: 19% (moderately inflated)

benchmarking sJackknife
time                 341.6 ms   (332.2 ms .. 349.0 ms)
                     1.000 R²   (1.000 R² .. 1.000 R²)
mean                 344.2 ms   (342.7 ms .. 346.0 ms)
std dev              2.105 ms   (907.3 μs .. 2.813 ms)
variance introduced by outliers: 19% (moderately inflated)

benchmarking pmJackknife
time                 365.9 ms   (75.56 ms .. 535.9 ms)
                     0.934 R²   (0.779 R² .. 1.000 R²)
mean                 442.5 ms   (394.8 ms .. 469.7 ms)
std dev              45.29 ms   (5.182 ms .. 58.01 ms)
variance introduced by outliers: 23% (moderately inflated)
```

Below it is shown how jackknife run with our rmap implementation compares to the given parMap from the Strategies library. As is shown our rmap does not perform as well as parMap. If we compare parJackknife with the sJackknife implementation from the previous image it seems much more similar in perfomance but both of them perform significantly better than sJackknife.

```
benchmarking rJackknife
time                 431.1 ms   (369.8 ms .. 466.8 ms)
                     0.998 R²   (0.996 R² .. 1.000 R²)
mean                 456.1 ms   (437.2 ms .. 485.8 ms)
std dev              29.25 ms   (5.332 ms .. 37.39 ms)
variance introduced by outliers: 19% (moderately inflated)

benchmarking parJackknife
time                 358.7 ms   (351.2 ms .. 366.2 ms)
                     1.000 R²   (1.000 R² .. 1.000 R²)
mean                 354.2 ms   (351.1 ms .. 356.0 ms)
std dev              3.042 ms   (722.1 μs .. 4.096 ms)
variance introduced by outliers: 19% (moderately inflated)
```

Assignment 2

We implemented a mergesort and a sum function on our higher-order divide and conquer algorithm.

The mergesort had a little worse performance than the sequential merge sort before we set granularity. Through testing granularity by setting the base case list size to various numbers we determined that our best performance was achieved by setting it to 8. Below you can see the sequential as well as the parallel version. With optimal granularity, the divide-and-conquer implementation performs better than the sequential.

```
benchmarking mergesort
time                 3.315 ms   (3.254 ms .. 3.362 ms)
                     0.998 R²   (0.997 R² .. 0.999 R²)
mean                 3.451 ms   (3.399 ms .. 3.525 ms)
std dev              213.1 μs   (152.7 μs .. 323.3 μs)
variance introduced by outliers: 40% (moderately inflated)

benchmarking pmergesort
time                 2.276 ms   (2.240 ms .. 2.331 ms)
                     0.995 R²   (0.991 R² .. 0.998 R²)
mean                 2.259 ms   (2.221 ms .. 2.295 ms)
std dev              126.9 μs   (104.6 μs .. 155.8 μs)
variance introduced by outliers: 40% (moderately inflated)
```

The sum performed terribly without any granularity control.

```
benchmarking sum
time                 39.55 μs   (38.97 μs .. 40.35 μs)
                     0.998 R²   (0.996 R² .. 0.999 R²)
mean                 40.89 μs   (40.27 μs .. 41.68 μs)
std dev              2.558 μs   (2.113 μs .. 3.040 μs)
variance introduced by outliers: 67% (severely inflated)

benchmarking psum
time                 2.515 ms   (2.442 ms .. 2.601 ms)
                     0.996 R²   (0.993 R² .. 0.999 R²)
mean                 2.430 ms   (2.406 ms .. 2.466 ms)
std dev              94.86 μs   (73.79 μs .. 143.9 μs)
variance introduced by outliers: 23% (moderately inflated)
```

Here we can see it performing about 64 times worse than the regular sequential sum function. The overhead introduced by creating sparks and the required logic for divide and conquer is causing the computer to spend most of its time on things completely unrelated to actually adding numbers together.

```
benchmarking sum
time                 46.80 μs   (46.35 μs .. 47.32 μs)
                     0.998 R²   (0.997 R² .. 0.999 R²)
mean                 48.67 μs   (47.95 μs .. 50.18 μs)
std dev              3.349 μs   (2.568 μs .. 4.351 μs)
variance introduced by outliers: 70% (severely inflated)

benchmarking psum
time                 485.1 μs   (472.0 μs .. 498.4 μs)
                     0.996 R²   (0.992 R² .. 0.999 R²)
mean                 475.5 μs   (470.6 μs .. 481.2 μs)
std dev              18.63 μs   (13.06 μs .. 27.87 μs)
variance introduced by outliers: 33% (moderately inflated)
```

By introducing granularity where the normal sum function is run on lists that are smaller than 100 we can see a large improvement. Likely because the computer spends much less time on divide-and-conquer logic and spark creation and can instead use its time to actually add numbers.

Assignment 3

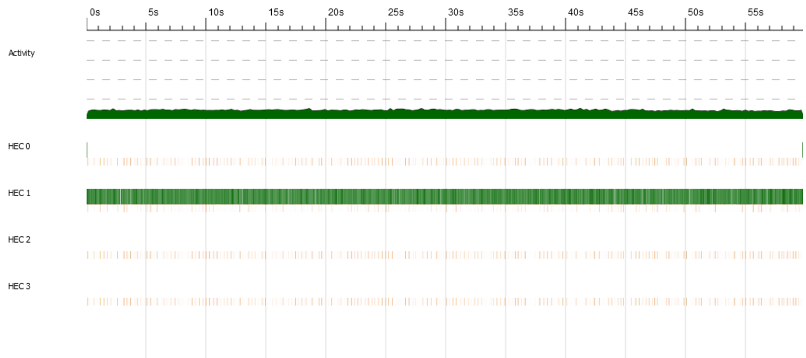
1

In short, parBuffer is a function that takes two arguments, an integer, let us call it “n”, and a strategy, that we will call “s”. The function causes evaluation on the inputs to be done with a limitation on available sparks to n at any one time. This makes the program less resource intensive on memory due to fewer sparks being on a waiting list and avoids too much parallelism which creates unnecessary overhead.

2

We chose to parallelize the sudoku solver. Essentially what we did was replace the map function with parallelized ones utilizing parBuffer, parList and parListChunk as well as a sequential one for comparison. We are able to see significant differences in how these map implementations handle the lists they are working with and how they subsequently handle sparks.

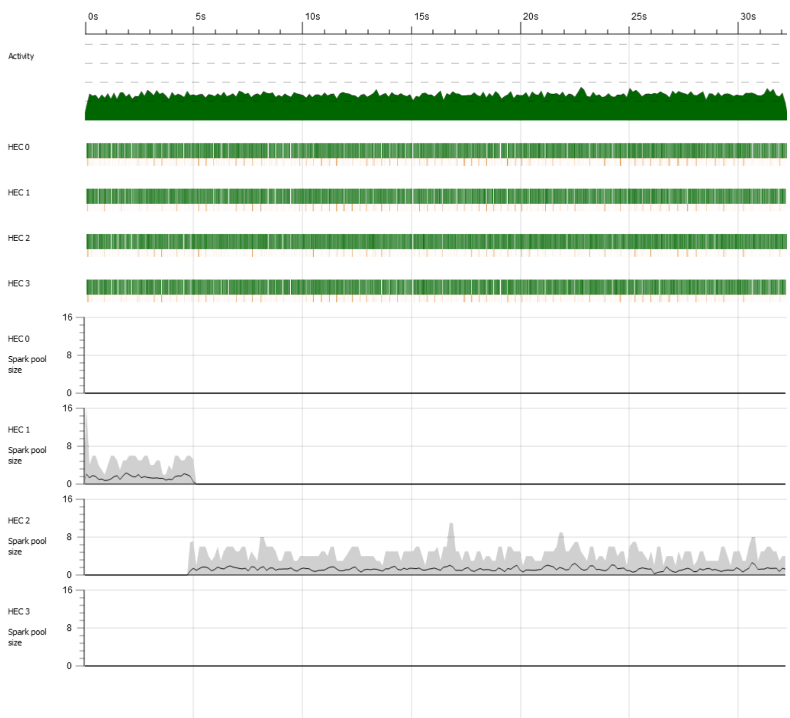
Sudoku running the regular map can be seen in threadscope only utilizing a single core. In total, it takes about a minute to run.



Total time:	59.752s
Mutator time:	58.753s
GC time:	998.917ms
Productivity:	98.3% of mutator vs total

Maximum heap size:	20.0	MiB	20,971,520	bytes
Maximum heap residency:	240.5	KiB	246,240	bytes
Total allocated:	22.1	GiB	23,763,203,544	bytes
Allocation rate:	385.7	MiB/s	404,461,548	bytes per second (of mutator time)
Maximum slop:	76.1	KiB	77,944	bytes

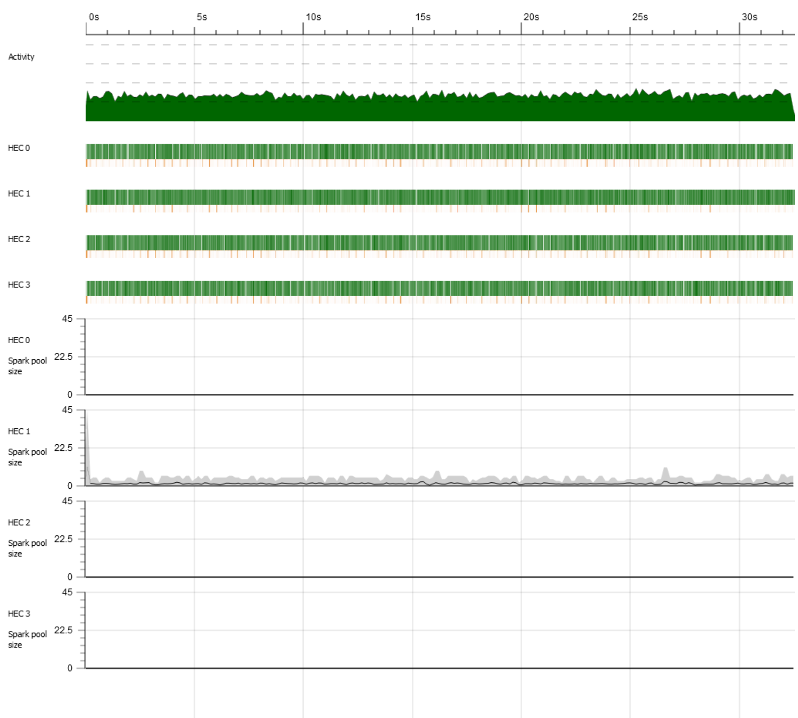
The sudoku using the parBuffer version of the map runs on all cores in parallel, however, the amount of work being done in parallel is rather low, as can be seen from the height of the activity reading. We can also see the spark pool which in this case shows that for the first 5 seconds HEC1 is the only one with sparks in its spark pool. After, that is instead the case for HEC2. This only happens for the version that restricts the buffer to 20 sparks. During the whole process, the number of sparks never exceeds 20. The execution time is about half that of the sequential map.



Total time:	32.711s
Mutator time:	31.858s
GC time:	852.924ms
Productivity:	97.4% of mutator vs total

Maximum heap size:	21.0	MiB	22,020,096	bytes
Maximum heap residency:	437.1	KiB	447,560	bytes
Total allocated:	22.5	GiB	24,188,599,744	bytes
Allocation rate:	724.1	MiB/s	759,261,851	bytes per second (of mutator time)
Maximum slop:	90.8	KiB	92,936	bytes

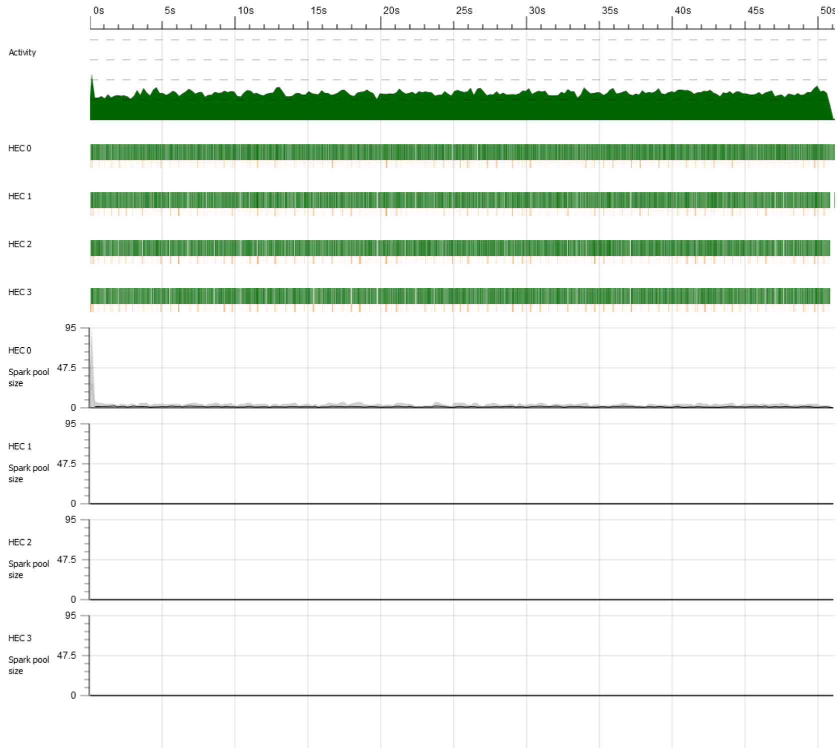
With the buffer limited to 50, we see similar results in productivity and execution time. However, the number of sparks is consistently lower throughout the execution except for right in the beginning.



Total time:	32.259s
Mutator time:	31.525s
GC time:	733.374ms
Productivity:	97.7% of mutator vs total

Maximum heap size:	21.0	MiB	22,020,096	bytes
Maximum heap residency:	508.8	KiB	520,984	bytes
Total allocated:	22.5	GiB	24,185,332,016	bytes
Allocation rate:	731.6	MiB/s	767,168,275	bytes per second (of mutator time)
Maximum slop:	95.1	KiB	97,352	bytes

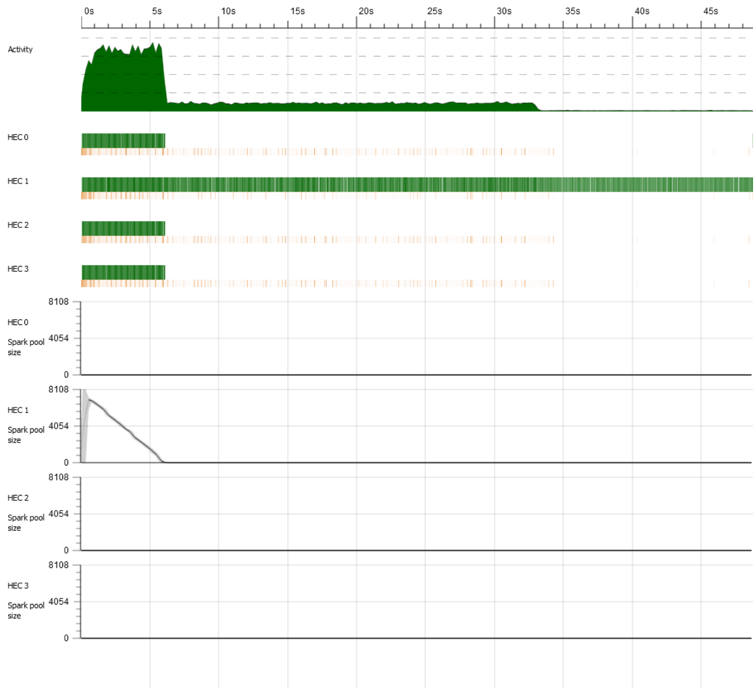
For the 100-sized buffer there is once again a very similar execution time. The number of sparks once again rises quickly at the start but then ends up consistently even lower than that of the 50-sized buffer.



Total time:	32.921s
Mutator time:	32.170s
GC time:	751.265ms
Productivity:	97.7% of mutator vs total

Maximum heap size:	21.0	MiB	22,020,096	bytes
Maximum heap residency:	620.1	KiB	634,976	bytes
Total allocated:	22.5	GiB	24,183,406,656	bytes
Allocation rate:	716.9	MiB/s	751,749,020	bytes per second (of mutator time)
Maximum slop:	106.4	KiB	108,976	bytes

The parList version of the map performs much worse. It starts creating many sparks and then resolves the sparks over time which is why we can see that the number of sparks goes down. This uses much more memory at a time than the parBuffer version which would be bad if we were limited. Unline the buffered version this one overflows. It also stops running in parallel after around 7 seconds which is likely why it performs worse.

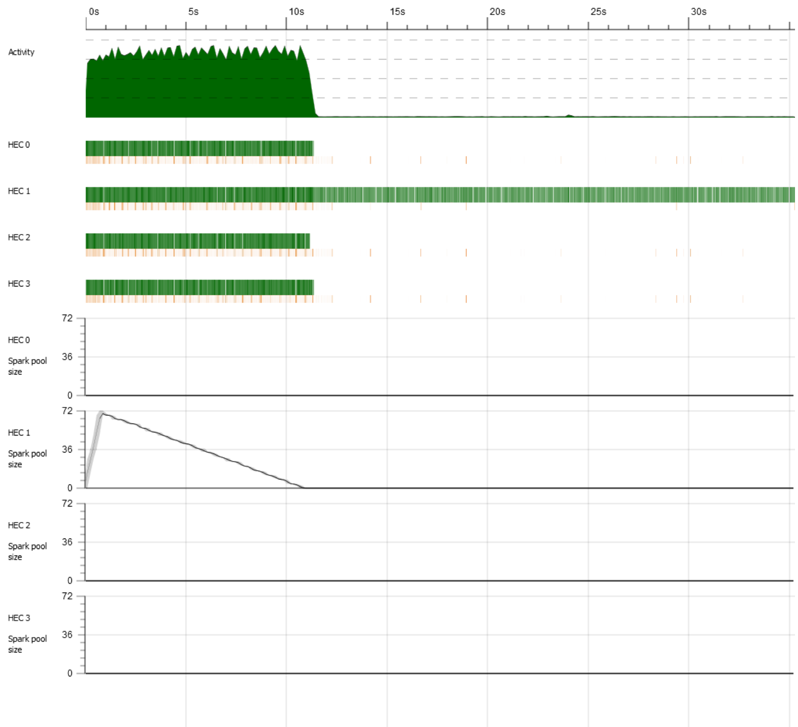


Total time:	48.026s
Mutator time:	46.855s
GC time:	1.171s
Productivity:	97.6% of mutator vs total

Maximum heap size:	117.0	MiB	122,683,392	bytes
Maximum heap residency:	32.3	MiB	33,863,440	bytes
Total allocated:	22.2	GiB	23,856,540,816	bytes
Allocation rate:	485.6	MiB/s	509,159,396	bytes per second (of mutator time)
Maximum slop:	1.4	MiB	1,444,504	bytes

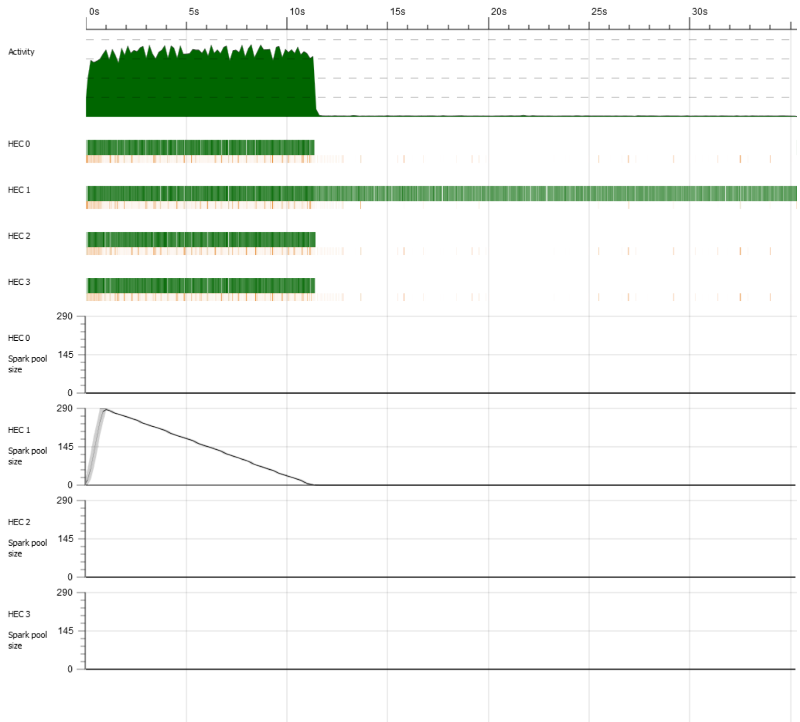
HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	16000	7623	185	0	0	0
HEC 0	0	2204	0	0	0	0
HEC 1	16000	1033	185	0	0	0
HEC 2	0	2163	0	0	0	0
HEC 3	0	2223	0	0	0	0

For parListChunk the performance is very similar to that of parBuffer. For chunks set to 50, 100, and 200 neither the execution time, heap sizes nor any other factors really change. This could be because the list that is being worked on is too small for there to be a large difference between these list sizes. Similar to parList this version creates many sparks in the beginning and then resolves them but it continues running in parallel for 10 seconds which leads to better performance.



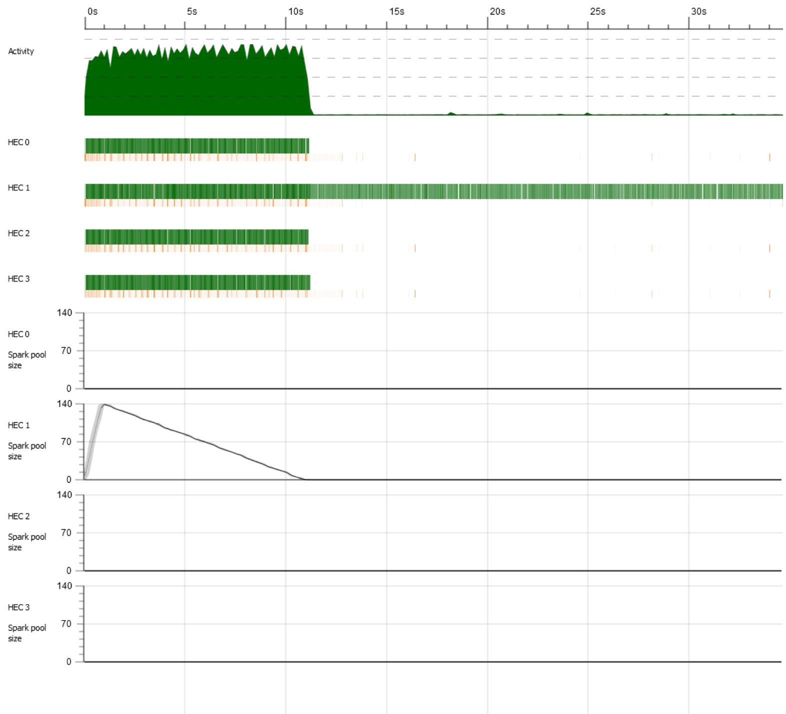
Total time:	34.870s
Mutator time:	34.128s
GC time:	742.729ms
Productivity:	97.9% of mutator vs total

Maximum heap size:	115.0	MiB	120,586,240	bytes
Maximum heap residency:	31.5	MiB	33,064,712	bytes
Total allocated:	22.2	GiB	23,850,764,016	bytes
Allocation rate:	661.6	MiB/s	693,695,453	bytes per second (of mutator time)
Maximum slop:	2.3	MiB	2,403,104	bytes



Total time:	35.183s
Mutator time:	34.382s
GC time:	800.409ms
Productivity:	97.7% of mutator vs total

Maximum heap size:	115.0	MiB	120,586,240	bytes
Maximum heap residency:	31.3	MiB	32,844,824	bytes
Total allocated:	22.2	GiB	23,850,495,888	bytes
Allocation rate:	656.4	MiB/s	688,257,471	bytes per second (of mutator time)
Maximum slop:	2.5	MiB	2,605,432	bytes

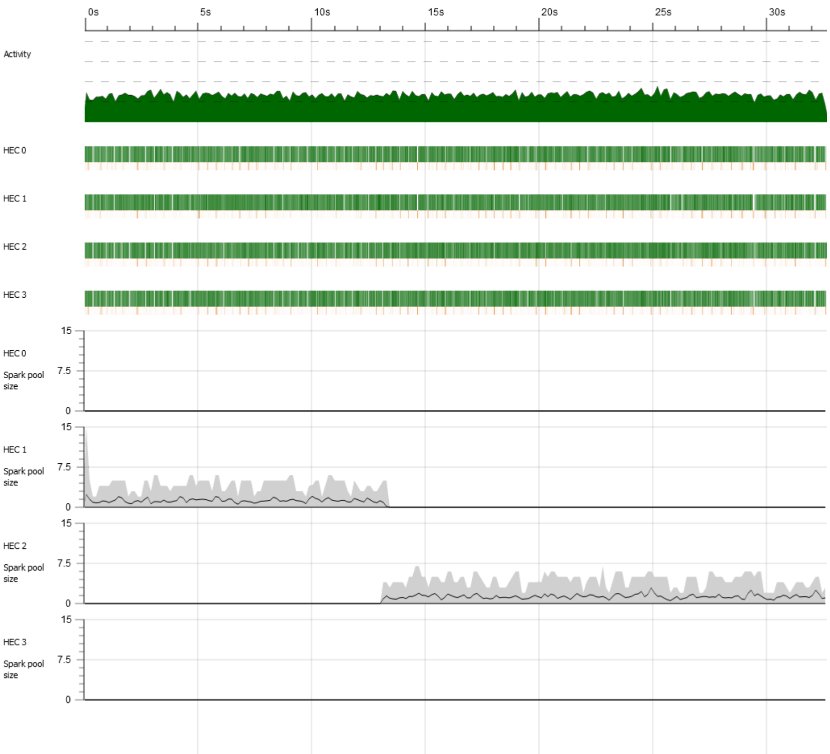


Total time:	35.496s
Mutator time:	34.653s
GC time:	842.150ms
Productivity:	97.6% of mutator vs total

Maximum heap size:	115.0	MiB	120,586,240	bytes
Maximum heap residency:	31.4	MiB	32,907,424	bytes
Total allocated:	22.2	GiB	23,850,179,232	bytes
Allocation rate:	666.5	MiB/s	698,850,197	bytes per second (of mutator time)
Maximum slop:	2.4	MiB	2,544,240	bytes

3

The chunked and buffered version of map runs similarly to that of parBuffer. It has a relatively continuous buffer size. Its heap size is much smaller than the parListChunk version, more similar to parBuffer. It has the advantage of avoiding too much parallelization that creates unnecessary overhead. A larger list chunk helps with performance in our testing. Activity is rather low throughout, however.



Total time:	32.642s
Mutator time:	31.851s
GC time:	791.423ms
Productivity:	97.6% of mutator vs total

Maximum heap size:	21.0	MiB	22,020,096	bytes
Maximum heap residency:	599.0	KiB	613,424	bytes
Total allocated:	22.5	GiB	24,189,055,064	bytes
Allocation rate:	724.3	MiB/s	759,452,382	bytes per second (of mutator time)
Maximum slop:	92.5	KiB	94,752	bytes