Parallel N-Queens in Haskell

Stephen A. Edwards, Columbia University

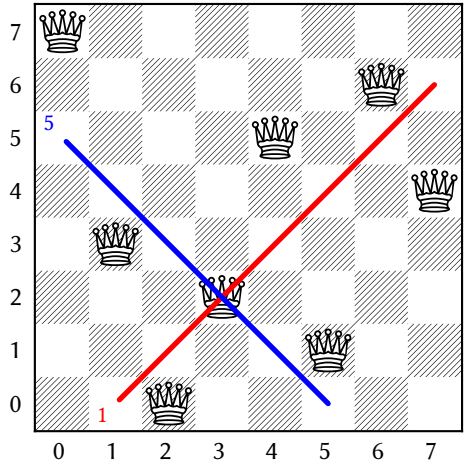
September 5, 2025

Our goal will be to calculate the number of solutions for a board of a particular size. We'll adopt a backtracking depth-first search algorithm due to Niklaus Wirth[1] that adds one queen per column in any row that isn't already occupied and not under diagonal threat from an existing queen.

This eliminates I/O bottlenecks since the input and output of the algorithm is each a small integer.

Wirth represents a partial solution as the row of the queen in each column, the set of occupied rows, the set of occupied diagonal rows that go up to the right, and the set of occupied diagonals that go down to the right.

We will denote the column ("file") with i and the row ("rank") with j. A queen at column i and row j is on up-diagonal $i-j$, whose values range from -7 to 7; the same queen is on an down-diagonal $i+j$, whose values range from 0 to 14. At right, the queen at $(i, j) = (3, 2)$ is on up-diagonal $3-2 = 1$ (red) and on down-diagonal $3 + 2 = 5$.



1 Imports

This is a Literate Haskell file (`.lhs`) which is a LaTeX file that `ghc` can also compile. It only sees the source code, which needs to begin with import directives:

```
import System.Environment(getArgs)
import System.Exit(die)
import qualified Data.Set as Set
import qualified Data.IntSet as IS
import Control.Parallel.Strategies(using, parList, rseq)
-- import Debug.Trace
```

[1] Niklaus Wirth. *Algorithms + Data Structures = Programs.* Prentice Hall, 1976. Section 3.5

2 Platforms

I ran my implementation on three different computers with Intel CPUs to test its platform sensitivity. These are the '3820, an older desktop, the '9700, a newer desktop, and the '4214, a server.

	'3820	**'9700**	**'4214**
Intel Product Line	Core	Core	Xeon Silver
Model	i7-3820	i7-9700	4214
Total Threads	8	8	48
Cores	4	8	12
Threads/Core	2	1	2
Sockets	1	1	2
Frequency (GHz)	3.6	3.0	2.2
Released	2012	2019	2019
Technology (nm)	32	14	14
Socket	LGA2011	LGA1151	LGA3647
Single-thread Rating*	1746	2774	1776
L1 caches†	32K × 4	32K × 8	32K × 24
L2 cache	256K × 4	256K × 8	1M × 24
L3 cache	10M	12M	16.5M × 2
Memory	64G	64G	152G
Speed (GT/s)	1.6	2.1	2.1
Channels‡	4	2	8

*From https://www.cpubenchmark.net
†Instruction and Data caches separate; numbers across all sockets
‡Present on motherboard

While the older '3820 has the clock rate advantage, threads on the '9700 have twice the available cache memory because it does not use simultaneous multithreading (Intel's Hyperthreading, which runs multiple threads per core). The '3820 was a higher-end chip than the '9700, but the '3820 is seven years older. One key difference is the number of pins, which is driven by the number of external memory channels and hence bandwidth.

The '4214 is a typical server: slow, but scales up. It contains two processor chips, each with four external memory channels (the chips have six; my motherboard only exposes four) that can be accessed by both chips. Its clock rate is slower, but its two chips hold processors equivalent to six '3820s and can support up to 2 TB of memory, compared to 128 GB on the '3820 and '9700.

3 Sequential with lists

Our first sequential implementation of the backtracking algorithm will use simple lists for both the row of each queen as well as the sets of diagonals, which we will pass as arguments.

```
nqueens :: String -> Int -> Int
nqueens "seqlist" n = sum $ map (helper [] [] []) [0..n-1]
  where
    helper rows updiags downdiags row
      | row      `elem` rows      ||
        updiag   `elem` updiags   ||
        downdiag `elem` downdiags = 0  -- Can't put it here
      | column == (n - 1)         = 1  -- It fits; we're done
      | otherwise = sum $ map (helper (row      : rows)
                                      (updiag   : updiags)
                                      (downdiag : downdiags)) [0..n-1]
      where column   = length rows -- Number we've already placed
            updiag   = column - row
            downdiag = column + row
```

n	Solutions	Time (s)	Memory (K)
8	92	0.01	8180
9	352	0.01	8180
10	724	0.05	8180
11	2680	0.22	8180
12	14 200	1.22	8308
13	73 712	7.36	8948
14	365 596	47.95	10 100

'9700, lists, No optimization

Time and memory statistics collected with `/usr/bin/time -f "%C %e %M"`

Not surprisingly, enabling optimization provided a noticable, fairly uniform speedup (roughly a factor of two).

Reassuringly, these solution counts match those on the Wikipedia Page[a].

The '9700 platform is the fastest; the other two are roughly 50% slower.

[a] https://en.wikipedia.org/wiki/Eight_queens_puzzle

n	Solutions	Time (s)	Memory (K)
8	92	0.01	4980
9	352	0.01	7412
10	724	0.03	8308
11	2680	0.13	8316
12	14 200	0.71	8180
13	73 712	4.41	8192
14	365 596	28.97	8436

'9700, lists, `-O2`

n	Solutions	Time (s)	Memory (K)
12	14 200	1.07	8312
13	73 712	6.61	8196
14	365 596	43.55	8204

'3820, lists, `-O2`

n	Solutions	Time (s)	Memory (K)
12	14 200	1.06	7680
13	73 712	6.48	7680
14	365 596	42.57	8064

'4214, lists, `-O2`

4 Sequential with Sets

While convenient and efficient for insertions, linked lists are slow to search. Instead, we will use the tree-based `Data.Set` containers for the three sets.

```
nqueens "seqset" n = sum $ map (helper Set.empty Set.empty Set.empty) [0..n-1]
  where
    helper rows updiags downdiags row
      | row      `Set.member` rows      ||
        updiag   `Set.member` updiags   ||
        downdiag `Set.member` downdiags = 0  -- Can't put it here
      | column == (n - 1)               = 1  -- It fits; we're done
      | otherwise = sum $ map (helper (row      `Set.insert` rows)
                                      (updiag   `Set.insert` updiags)
                                      (downdiag `Set.insert` downdiags)) [0..n-1]
      where column   = Set.size rows -- Number we've already placed
            updiag   = column - row
            downdiag = column + row
```

n	Solutions	Time (s)	Memory (K)
8	92	0.01	4980
9	352	0.01	7412
10	724	0.03	8308
11	2680	0.13	8316
12	14 200	0.71	8180
13	73 712	4.41	8192
14	365 596	28.97	8436

'9700, lists, `-O2`

n	Solutions	Time (s)	Memory (K)
8	92	0.01	5364
9	352	0.01	8180
10	724	0.02	8308
11	2680	0.06	8308
12	14 200	0.3	8308
13	73 712	1.68	8436
14	365 596	10.32	8820

'9700, `Set`, `-O2`

n	Solutions	Time (s)	Memory (K)
12	14 200	1.07	8312
13	73 712	6.61	8196
14	365 596	43.55	8204

'3820, lists, `-O2`

n	Solutions	Time (s)	Memory (K)
12	14 200	0.4	8140
13	73 712	2.3	8228
14	365 596	13.91	8616

'3820, `Set`, `-O2`

n	Solutions	Time (s)	Memory (K)
12	14 200	1.06	7680
13	73 712	6.48	7680
14	365 596	42.57	8064

'4214, lists, `-O2`

n	Solutions	Time (s)	Memory (K)
12	14 200	0.44	7680
13	73 712	2.45	7680
14	365 596	14.82	8448

'4214, `Set`, `-O2`

Moving from lists to the `Set` data structure gave nearly a 3× speedup on all three platforms.

5 Sequential with IntSets

Haskell's `Data.Set` only relies on a key being a member of the `Ord` class, but our sets only contain small integers. The `Data.IntSet` container is specially tailored to integer keys, so we will try them instead.

```
nqueens "seqiset" n = sum $ map (helper IS.empty IS.empty IS.empty) [0..n-1]
  where
    helper rows updiags downdiags row
      | row      `IS.member` rows      ||
        updiag   `IS.member` updiags   ||
        downdiag `IS.member` downdiags = 0  -- Can't put it here
      | column == (n - 1)              = 1  -- It fits; we're done
      | otherwise = sum $ map (helper (row      `IS.insert` rows)
                                      (updiag   `IS.insert` updiags)
                                      (downdiag `IS.insert` downdiags)) [0..n-1]
      where column   = IS.size rows -- Number we've already placed
            updiag   = column - row
            downdiag = column + row
```

n	Solutions	Time (s)	Memory (K)
8	92	0.01	5364
9	352	0.01	8180
10	724	0.02	8308
11	2680	0.06	8308
12	14 200	0.3	8308
13	73 712	1.68	8436
14	365 596	10.32	8820

'9700, `Set`, `-O2`

n	Solutions	Time (s)	Memory (K)
8	92	0.01	4596
9	352	0.01	5748
10	724	0.01	8308
11	2680	0.03	8308
12	14 200	0.14	8308
13	73 712	0.75	8308
14	365 596	4.53	8308

'9700, `IntSet`, `-O2`

Moving from `Set` to `IntSet` gave about another 2× speedup.

Now, in preparation for working on a parallel implementation, let's check that switching to the multithreaded runtime system (by compiling with `-threaded`) does not affect the runtimes.

n	Solutions	Time (s)	Memory (K)
8	92	0.01	4596
9	352	0.01	5748
10	724	0.01	8308
11	2680	0.03	8308
12	14 200	0.14	8308
13	73 712	0.75	8308
14	365 596	4.53	8308

'9700, `IntSet`, `-O2`

n	Solutions	Time (s)	Memory (K)
8	92	0.01	4724
9	352	0.02	6004
10	724	0.04	8436
11	2680	0.05	8436
12	14 200	0.14	8436
13	73 712	0.77	8436
14	365 596	4.62	8436

'9700, `IntSet`, `-N1`, `-O2 -threaded -rtsopts`

list

```
 2,226,634,760 bytes allocated in the heap
       494,736 bytes copied during GC
        72,496 bytes maximum residency (2 sample(s))
        30,152 bytes maximum slop
             6 MiB total memory in use (0 MiB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0       532 colls,     0 par    0.002s   0.002s     0.0000s    0.0000s
Gen  1         2 colls,     0 par    0.000s   0.000s     0.0001s    0.0001s

MUT     time    4.512s  (  4.509s elapsed)
GC      time    0.002s  (  0.002s elapsed)
Total   time    4.514s  (  4.520s elapsed)

Alloc rate    493,545,911 bytes per MUT second
```

Set

```
 3,223,147,848 bytes allocated in the heap
     2,600,568 bytes copied during GC
        76,208 bytes maximum residency (2 sample(s))
        30,152 bytes maximum slop
             6 MiB total memory in use (0 MiB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0       772 colls,     0 par    0.005s   0.005s     0.0000s    0.0001s
Gen  1         2 colls,     0 par    0.000s   0.000s     0.0001s    0.0001s

MUT     time    1.675s  (  1.674s elapsed)
GC      time    0.005s  (  0.005s elapsed)
Total   time    1.680s  (  1.680s elapsed)

Alloc rate    1,924,563,566 bytes per MUT second
```

IntSet

```
   945,127,512 bytes allocated in the heap
       314,640 bytes copied during GC
        72,808 bytes maximum residency (2 sample(s))
        30,152 bytes maximum slop
             6 MiB total memory in use (0 MiB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0       225 colls,     0 par    0.001s   0.001s     0.0000s    0.0000s
Gen  1         2 colls,     0 par    0.000s   0.000s     0.0001s    0.0001s

MUT     time    0.760s  (  0.760s elapsed)
GC      time    0.001s  (  0.001s elapsed)
Total   time    0.762s  (  0.770s elapsed)

Alloc rate    1,243,275,249 bytes per MUT second
```

’9700, 13 × 13, `-O2 -threaded -rtsopts`, `-N1 -s`

Here, garbage collection overhead appears very modest, but there are interesting differences among the three implementations. While faster than using lists, using `Set` allocated nearly 50% more memory, needed to copy nearly 5× as much during GC, and ran GC on the nursery (`Gen 0`) nearly 50% more times. The total GC time nearly doubled.

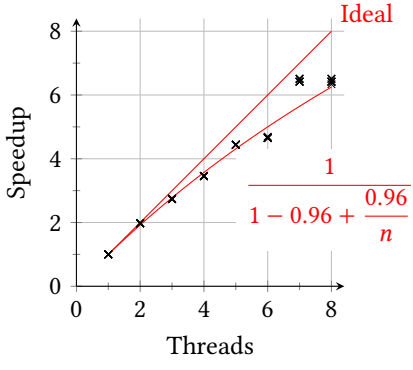
The `IntSet` implementation allocated far less memory, ran GC fewer times, and took less time overall.

Note that in all cases, garbage collection on the main heap (`Gen 1`) ran only twice. This algorithm does not generate long-lived garbage.

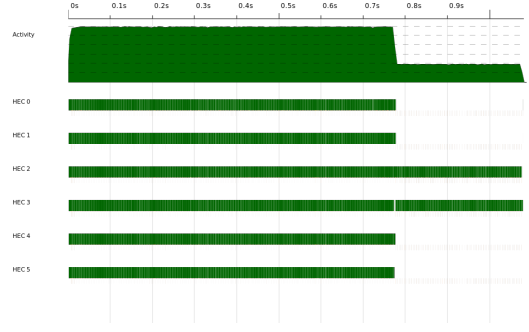
7 Parallel 1

First, we will simply evaluate the first column's choices with `'using' parList rseq`.

```
nqueens "pariset1" n = sum (firstcol `using` parList rseq)
  where
    firstcol = map (helper IS.empty IS.empty IS.empty) [0..n-1]
    helper rows updiags downdiags row
      | row      `IS.member` rows      ||
        updiag   `IS.member` updiags   ||
        downdiag `IS.member` downdiags = 0  -- Can't put it here
      | column == (n - 1)              = 1  -- It fits; we're done
      | otherwise = sum $
         map (helper (row      `IS.insert` rows)
                     (updiag   `IS.insert` updiags)
                     (downdiag `IS.insert` downdiags)) [0..n-1]
      where column   = IS.size rows -- Number we've already placed
            updiag   = column - row
            downdiag = column + row
```



'9700, 14 × 14, `IntSet`

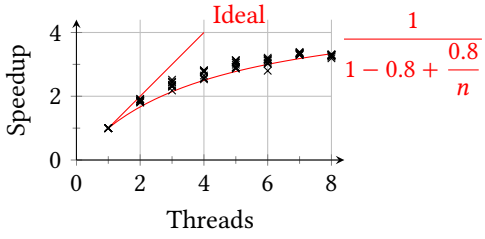


'9700, 14 × 14, `IntSet`, `-N6`

To verify the anomaly at 6 threads wasn't a sampling artifact, I ran each test 10 times and plotted each of them; the elapsed times were remarkably consistent.

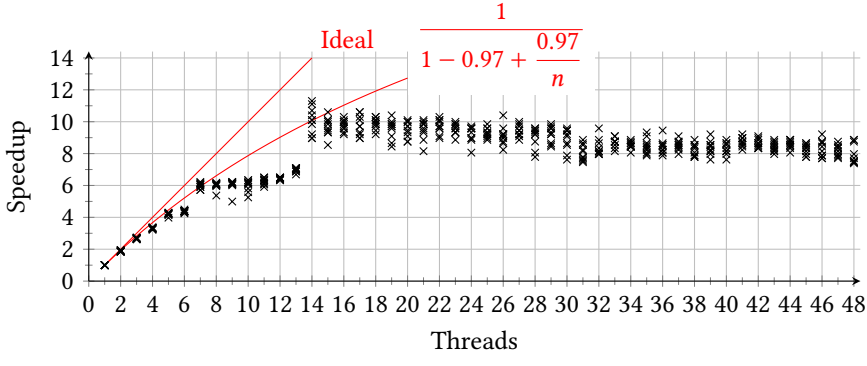
The Threadscope graph on the right confirms my suspicions about load balancing. While the workload for each guessed row is likely similar, since we are only creating 14 sparks, with 6 threads, we run the first 6, then the second 6, then have two "left over."

I ran this experiment on the other two platforms. On the ’3820, the maximum speedup (and the level of parallelism, according to Amdahl) decreased substantially.



’3820, 14×14, `IntSet`

But the results on the ’4214 showed even more anomalies:



’4214, 14×14, `IntSet`

The anomaly at 6 threads exhibited on the ’9700 remains, but now there is performance plateau from 7 to 13 threads, then a big jump at 14, after which the speedup actually *decreases*.

Discretization is usually to blame for such non-smooth scaling. Here, the big jump at 14 is simply due to this algorithm only producing 14 sparks; parallel resources beyond that are simply left idle; the slight slowdown after 14 might be due to the increasing cost of synchronizing more threads.

The conclusion is that we are not supplying the 48-thread machine with enough parallelism.

8 IntSets 2

In preparing to address the load balancing problem by consolidating the two *map* operations to parallelize multiple columns, I inadvertantly sped up the algorithm by another 30%. This sequential code runs over 13× faster than the version that used lists.

```
nqueens "seqiset2" n = count (IS.empty, IS.empty, IS.empty)
 where
  count (rows, ups, downs) | column == n = 1
                           | otherwise   = sum $ map count boards
   where
    column = IS.size rows -- Column we're trying to add
    boards = [ ( row  `IS.insert` rows,   -- Next board adds queen at row
                 up   `IS.insert` ups,    -- Record occupied diagonals
                 down `IS.insert` downs )
             | row <- [0..n-1],           -- Consider each possible row
               let up = column - row      -- up-diagonal number
                   down = column + row,   -- down-diagonal number
               row  `IS.notMember` rows,  -- Row may not be occupied
               up   `IS.notMember` ups,   -- Diagonals may not be occupied
               down `IS.notMember` downs ]
```

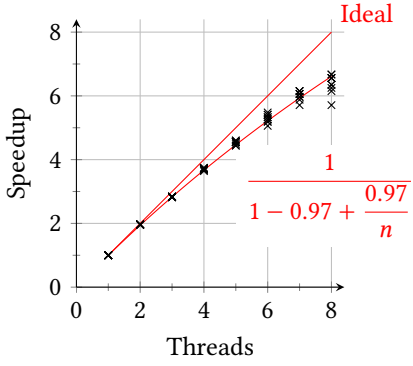
n	Solutions	Time (s)	Memory (K)
8	92	0.01	4724
9	352	0.02	6004
10	724	0.04	8436
11	2680	0.05	8436
12	14 200	0.14	8436
13	73 712	0.77	8436
14	365 596	4.62	8436

'9700, `14 seqiset, -N1`

n	Solutions	Time (s)	Memory (K)
8	92	0.01	4724
9	352	0.01	5748
10	724	0.01	8436
11	2680	0.03	8436
12	14 200	0.11	8436
13	73 712	0.56	8436
14	365 596	3.34	8564

'9700, `14 seqiset2, -N1`

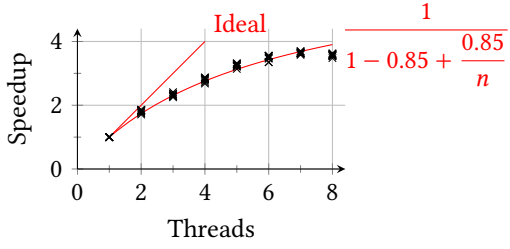
```
nqueens "pariset2" n = count (0 :: Int) (IS.empty, IS.empty, IS.empty)
 where
  count r (rows, ups, downs)
    | column == n = 1
    | r < 2 = sum (map (count (r+1)) boards `using` parList rseq)
    | otherwise   = sum $ map (count (r+1)) boards
   where
    column = IS.size rows -- Column we're trying to add
    boards = [ ( row  `IS.insert` rows,   -- Next board adds queen at row
                 up   `IS.insert` ups,    -- Record occupied diagonals
                 down `IS.insert` downs )
             | row <- [0..n-1],           -- Consider each possible row
               let up = column - row      -- up-diagonal number
                   down = column + row,   -- down-diagonal number
               row  `IS.notMember` rows,  -- Row may not be occupied
               up   `IS.notMember` ups,   -- Diagonals may not be occupied
               down `IS.notMember` downs ]
```



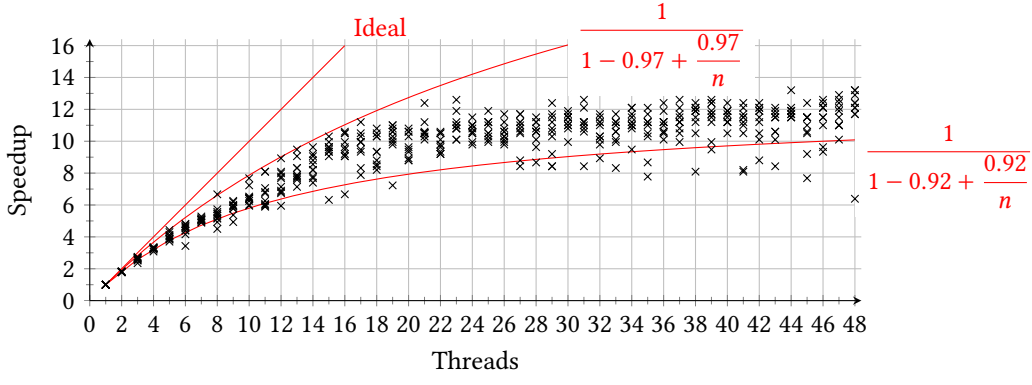
’9700, `14 pariset2`

Parallelizing the first two columns’ searches eliminated the anomaly at six threads (the curve is now smooth) also made it slightly more parallel.

With 8 threads on a 14×14 board, 140/170 sparks were converted; the rest fizzled.



’3820, `14 pariset2`



’4214, `14 pariset2`

On the '9700, increasing the nursery on `nqueens 14 pariset2 +RTS -N8` unexpectedly slowed things down. The default is a 4MB nursery (`-A5M`); increasing it to 64MB slowed things down around 9%.

```
 9,082,812,792 bytes allocated in the heap
     4,135,736 bytes copied during GC
       452,664 bytes maximum residency (3 sample(s))
       101,488 bytes maximum slop
            36 MiB total memory in use (0 MiB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0       292 colls,   292 par    0.028s   0.008s     0.0000s    0.0002s
Gen  1         3 colls,     2 par    0.001s   0.000s     0.0001s    0.0002s

Parallel GC work balance: 67.71% (serial 0%, perfect 100%)
```

`-A4M`

```
TASKS: 18 (1 bound, 17 peak workers (17 total), using -N8)

SPARKS: 170 (139 converted, 0 overflowed, 0 dud, 0 GC'd, 31 fizzled)

INIT    time    0.002s  (  0.001s elapsed)
MUT     time    5.038s  (  0.661s elapsed)
GC      time    0.029s  (  0.009s elapsed)
EXIT    time    0.001s  (  0.009s elapsed)
Total   time    5.070s  (  0.680s elapsed)

Alloc rate    1,802,932,180 bytes per MUT second

Productivity  99.4% of total user, 97.2% of total elapsed
```

```
 9,074,235,752 bytes allocated in the heap
       598,384 bytes copied during GC
       420,488 bytes maximum residency (2 sample(s))
        95,608 bytes maximum slop
           524 MiB total memory in use (0 MiB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0        16 colls,    16 par    0.003s   0.001s     0.0001s    0.0003s
Gen  1         2 colls,     1 par    0.000s   0.000s     0.0001s    0.0001s

Parallel GC work balance: 71.22% (serial 0%, perfect 100%)
```

`-A64M`

```
TASKS: 18 (1 bound, 17 peak workers (17 total), using -N8)

SPARKS: 170 (137 converted, 0 overflowed, 0 dud, 2 GC'd, 31 fizzled)

INIT    time    0.006s  (  0.005s elapsed)
MUT     time    5.801s  (  0.749s elapsed)
GC      time    0.004s  (  0.001s elapsed)
EXIT    time    0.001s  (  0.005s elapsed)
Total   time    5.812s  (  0.760s elapsed)

Alloc rate    1,564,269,276 bytes per MUT second

Productivity  99.8% of total user, 98.5% of total elapsed
```

Increasing the nursery size did reduce garbage collection overhead: the nursery had to be collected only 16 times instead of 289, reducing the number of bytes copied during garbage collection as well as the total time spent performing it (26 to 3 ms), but mutator time increased 62 ms.

This difference is mysterious: There's a small increase (about 3 ms) in start-up time for the 64MB nursery, but overall, the smaller nursery looks like it should be slower because it's being interrupted much more frequently by garbage collection, yet it takes less time to reach its "ramp-down" phase where the various parallel sparks complete and the results are aggregated. Both runs take about the same amount of time to complete this.

FIXME: Consider vectors, using a Word64

FIXME: heap profiling: FIXME: stack ghc – -o nqueens-prof -O2 -Wall -rtsopts -threaded -prof -fprof-auto nqueens.lhs FIXME: ./nqueens-prof 13 seqiset +RTS -hy -N1 -i0.025 FIXME: hp2ps -c nqueens-prof.hp FIXME: About 40% stack, 40% ARR_WORDS (unboxed vectors), and 20% other stuff

FIXME: larger nursery on ford

FIXME: better quantitative analysis of load balancing

FIXME: Heuristic to decide when to run sequential

FIXME: pariset3 to depth of 3

10 Top-Level

Here is the top-level *main* function, which reads the board dimensions and runs the appropriate algorithm.

```
nqueens _ _ = 0 -- Undefined mode

main :: IO ()
main = do
  args1 <- getArgs
  case args1 of
    [nstr, mode] -> print $ nqueens mode (read nstr)
    _ -> die $ "Usage: nqueens n mode"
```