Parallel N-Queens in Haskell

Stephen A. Edwards, Columbia University
https://github.com/sedwards-lab/pfp-nqueens

November 5, 2025

This is intended as a reference project for Columbia's *Parallel Functional Programming* class,[1] which uses Haskell's parallel evaluation strategies, which are described in Marlow's book.[2] This project starts from a good "parallelization candidate"—the n-queens problem—and, through improvements guided by measurements, ultimately gave a 12× parallel speedup. Run time on one machine on a 14×14 board went from 47.5 s to 0.7 s, nearly a 70× speedup. Roughly 10× was due to sequential improvements and 7× due to parallelism.

I selected the n-queens problem because I expected it would lead to successful parallelization. It is not I/O-limited because its input (the board size) and output (number of solutions) were both integers and a modest appetite for runtime memory. This worked: the speedup from parallelization peaked at about 12× on a 48-core server, which Amdahl's law[3] puts at about 95% parallel. The magnitude of the work demanded by this algorithm can be easily adjusted by changing the board size, which makes it convenient to set the runtime of a particular experiment to be long enough to avoid time measurement noise yet short enough to be practical to run repeatedly. It also did not require a substantial corpus of input data to be either obtained or synthesized.

After selecting the algorithm, I created a series of implementations, each faster than the last, by running experiments and adjusting the implementation to produce an improvement. I started with a sequential implementation and twice improved the central data structure used to track the solution. Then, I parallelized the algorithm, first parallelizing the choices for only the first column, then for the first two columns after seeing load balancing issues.

While reducing total elapsed time is important, scalability—how well the implementation is able to take advantage of additional parallel resources—is the deeper, more challenging metric. To monitor it, I routinely plot the speedup (elapsed time of a parallel implementation running on a single thread divided by the elapsed time of a parallel implementation running on multiple threads) as a function of the number of threads. An ideal speedup would be a straight line; Amdahl's law says a task whose parallel fraction is $0 \leq p \leq 1$ speeds up by $1/(1 - p + p/n)$ given n parallel resources.

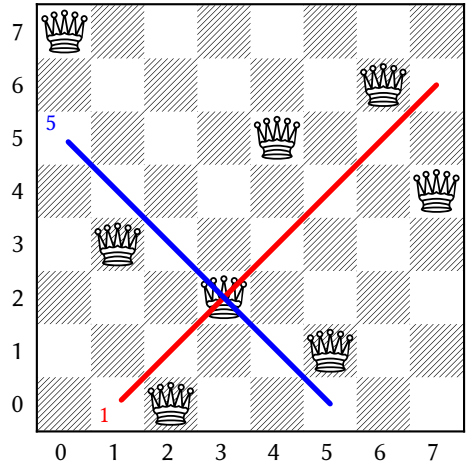
[1] https://www.cs.columbia.edu/~sedwards/classes/2025/4995-fall/
[2] Simon Marlow. *Parallel and Concurrent Programming in Haskell.* O'Reilly, 2013. https://simonmar.github.io/pages/pcph.html
[3] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In Proc. Spring Joint Computer Conference (AFIPS), 1967. pp. 483–485. https://doi.org/10.1145/1465482.1465560

1 N-Queens

N-Queens is a classical problem[4] that asks how many ways there are to arrange n queen pieces on an $n \times n$ chessboard such that none may capture the other, i.e., no two pieces are in the same row, column, or diagonal. I will adopt the backtracking depth-first search algorithm due to Niklaus Wirth[5] that adds one queen per column in any row that isn't already occupied and not under diagonal threat from another queen.

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 $3 + 2 = 5$.



2 The nqueens Executable

This report is a Literate Haskell file (`.lhs`), which is a LaTeX file that GHC can also compile. GHC only sees the source code, which begins 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)
```

I will put our various implementations into a single executable (compiled three ways) that takes two arguments: the board size and a string representing which implementation to use. The *main* function reads these arguments and dispatches the *nqueens* function.

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

-- nqueens mode board-dimensions -> number-of-solutions
nqueens :: String -> Int -> Int
```

[4] Dating from 1850. See W. W. Rouse Ball. *Mathematical Recreations and Essays.* Macmillan, 1905, 4th ed. https://www.gutenberg.org/files/26839/26839-pdf.pdf

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

3 Platforms

I ran my implementations on three different computers with Intel CPUs to test its platform sensitivity. These are the '3820 desktop from 2012, the '9700 desktop from 2019, and '4214 server, also from 2019.

	'3820	'9700	'4214
Intel Product Line	Core i7	Core i7	Xeon Silver
Model	i7-3820	i7-9700	4214
Year	2012	2019	2019
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
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	10 MB	12 MB	16.5 MB × 2
Memory	64 GB	64 GB	152 GB
Memory Speed (GT/s)	1.6	2.1	2.1
Memory 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 primarily 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 provides slots for 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.

4 Sequential with Lists

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

```
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	0.08	8192
9	352	0.02	8164
10	724	0.05	8164
11	2680	0.22	8064
12	14 200	1.21	8320
13	73 712	7.39	8932
14	365 596	47.5	9856

'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.[6]

The '9700 platform is the fastest; the other two are roughly half as fast.

n	Solutions	Time (s)	Memory (K)
8	92	0.01	4836
9	352	0.03	7552
10	724	0.04	8164
11	2680	0.13	8292
12	14 200	0.72	8192
13	73 712	4.36	8192
14	365 596	28.81	8064

'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`

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

5 Sequential with Sets

While convenient and efficient for insertions, linked lists are slow to search. Instead, I 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	4836
9	352	0.03	7552
10	724	0.04	8164
11	2680	0.13	8292
12	14 200	0.72	8192
13	73 712	4.36	8192
14	365 596	28.81	8064

'9700, lists, `-O2`

n	Solutions	Time (s)	Memory (K)
8	92	0.01	5248
9	352	0.02	8192
10	724	0.04	8292
11	2680	0.06	8292
12	14 200	0.3	8164
13	73 712	1.66	8292
14	365 596	10.24	8832

'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.

6 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; I will try it.

```
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	5248
9	352	0.02	8192
10	724	0.04	8292
11	2680	0.06	8292
12	14 200	0.3	8164
13	73 712	1.66	8292
14	365 596	10.24	8832

'9700, `Set`, `-O2`

n	Solutions	Time (s)	Memory (K)
8	92	0.01	4452
9	352	0.01	5888
10	724	0.01	8292
11	2680	0.03	8192
12	14 200	0.14	8292
13	73 712	0.75	8164
14	365 596	4.51	8292

'9700, `IntSet`, `-O2`

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

Now, in preparation for working on a parallel implementation, I will verify 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	4452
9	352	0.01	5888
10	724	0.01	8292
11	2680	0.03	8192
12	14 200	0.14	8292
13	73 712	0.75	8164
14	365 596	4.51	8292

'9700, `IntSet`, `-O2`

n	Solutions	Time (s)	Memory (K)
8	92	0.1	4608
9	352	0.02	5988
10	724	0.04	8292
11	2680	0.04	8420
12	14 200	0.14	8420
13	73 712	0.76	8292
14	365 596	4.61	8420

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

7 Garbage Collection Statistics for the Sequential Implementations

list

```
 2,226,634,208 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    0.0001s
Gen  1         2 colls,     0 par    0.000s   0.000s     0.0001s    0.0001s

MUT     time    4.511s  (  4.513s elapsed)
GC      time    0.002s  (  0.002s elapsed)
Total   time    4.514s  (  4.520s elapsed)

Alloc rate    493,649,598 bytes per MUT second
```

Set

```
 3,223,147,296 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    0.0000s
Gen  1         2 colls,     0 par    0.000s   0.000s     0.0001s    0.0001s

MUT     time    1.663s  (  1.670s elapsed)
GC      time    0.005s  (  0.005s elapsed)
Total   time    1.670s  (  1.680s elapsed)

Alloc rate    1,937,668,355 bytes per MUT second
```

IntSet

```
   945,126,960 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    0.764s  (  0.768s elapsed)
GC      time    0.001s  (  0.001s elapsed)
Total   time    0.765s  (  0.770s elapsed)

Alloc rate    1,237,164,940 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.

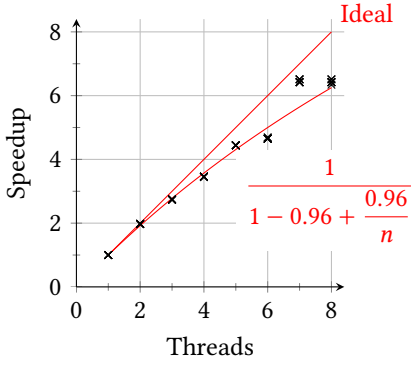
The `IntSet` implementation allocated far less memory, ran GC fewer times, and took less time overall.

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

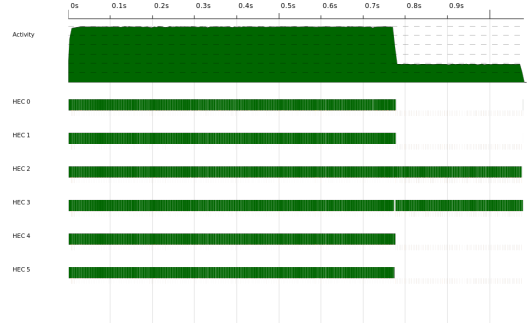
8 Parallel 1

First, I will just evaluate the first column's choices with `'using' parList rseq` and do the rest sequentially.

```
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 pariset1`

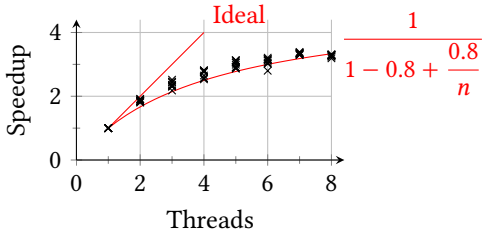


'9700, `14 pariset1 -N6`

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.

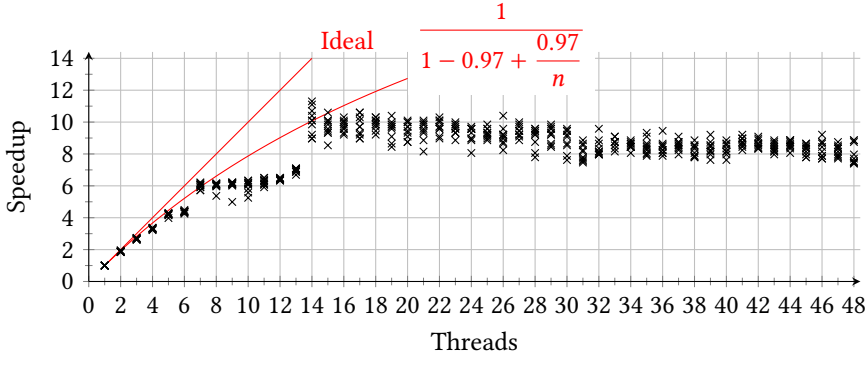
The Threadscope graph on the right confirmed my suspicions that load balancing was to blame for the anomaly. While the workload for each row is likely similar, since we are only creating 14 sparks, with 6 threads, we run the first 6, then the second 6, then have two "left over."

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 pariset1`

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



’4214, `14 pariset1`

The anomaly at 6 threads exhibited on the ’9700 remains, but now there is a 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 jumps. 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 this implementation is simply not supplying the 48-thread machine with enough parallelism.

9 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	0.1	4608
9	352	0.02	5988
10	724	0.04	8292
11	2680	0.04	8420
12	14 200	0.14	8420
13	73 712	0.76	8292
14	365 596	4.61	8420

'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`

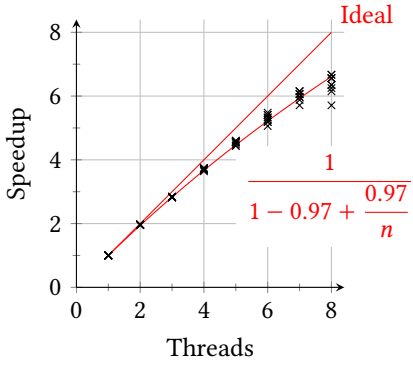
10 Parallel IntSets 2

My second parallel implementation searches the first two columns in parallel, then does the rest sequentially.

```
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 ]
```

On the 8-thread ’9700, 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.

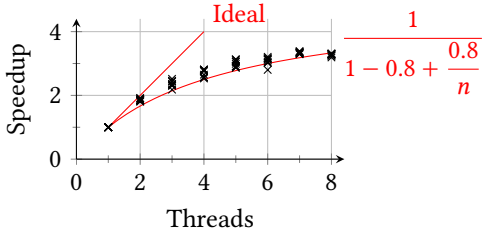


’9700, `14 pariset2`

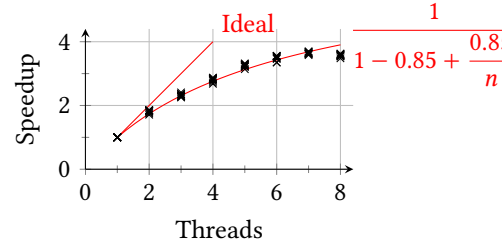
n	Solutions	Time (s)	Memory (K)
8	92	0.01	6244
9	352	0.01	10 628
10	724	0.01	22 788
11	2680	0.02	35 004
12	14 200	0.03	38 176
13	73 712	0.13	38 680
14	365 596	0.68	39 092

’9700, `pariset2 -N8`

On the ’3820, the parallelization improved slightly.

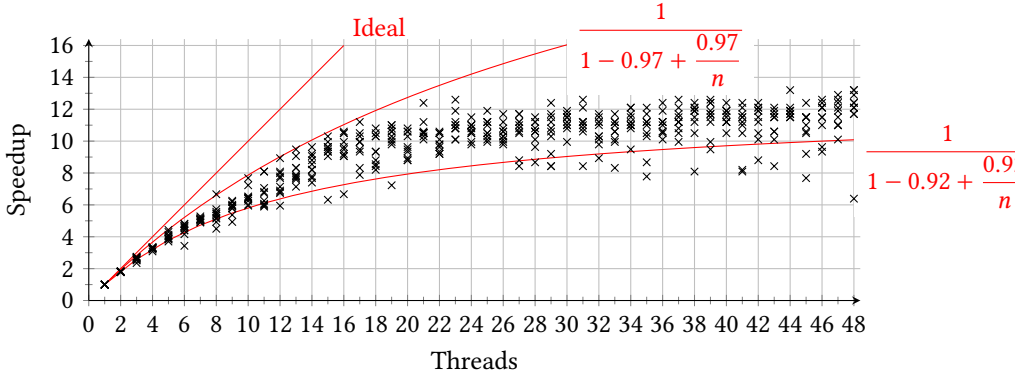


’3820, `14 pariset1`



’3820, `14 pariset2`

The most noticable improvement was on the 48-core ’4214, which saw smooth performance increases up to about 21 threads, after which it plateaued.



’4214, `14 pariset2`

On the ’9700, increasing the nursery with 8 threads unexpectedly slowed things down. The default is a 4 MB nursery (`-A4M`); increasing it to 64 MB 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 292, reducing the number of bytes copied during garbage collection as well as the total time spent performing it (28 to 3 ms), but mutator time increased 88 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.

11 Conclusion

I was happy with a 70× speedup and about 97% parallelizable on an 8-thread machine, so I stopped here. The one final bit of code is the default case for the *nqueens* function, which is faster than anything presented above, albeit with an incorrect result.

```
nqueens _ _ = 0 -- Unknown mode
```

There are many more things that could be done. Sequential optimization ideas include using *Word64*s for the sets, which might be faster than even *IntSet*. Compiling with the *-prof* flag to enable heap profiling could lead to still better data structures and a reduced memory footprint. Keeping the memory footprint small is always a good idea because it makes better use of the caches and reduces the load on the garbage collector.

On the '9700, increasing the nursery size actually slowed things down, but I did not test this on the other two machines. There may be additional benefits to be had with different nursery sizes.

While there was a serious load balancing issue when there were only 14 parallel sparks, it seemed acceptable when I moved to 170 sparks. There might still be improvements to be had.

The current best implementation can make use of 20 threads, but it plateaus after that, even though it seems like it should be embarrassingly parallel. It would be an interesting, if difficult, exercise to figure out why and further improve things. For example, would generating more sparks improve the dynamic load balancing?

12 Running experiments and producing graphs

It's always worth automating the process of running an experiment because you always need to run it multiple times. Each graph and table of results in this report was generated with a two-stage process that

1. ran one of the *nqueens* executables one or more times with different parameters, collecting the runtime and memory usage into a text *.out* file; and

2. processed the "raw" experimental data in the *.out* file with one of four *awk* scripts into a *.tex* file containing a formatted table, or into a *.dat* file plotted using the LaTeX *pgfplots* package.

Running all the experiments is time-consuming and I rarely had access to all three machines simultaneously, so I wrote the *nqueens* bash script to invoke them on command. Running *nqueens* either runs all the experiments for a particular host or runs some subset of them, useful during development. Although the generated *.out* files are generated automatically, they are costly to reproduce, so I checked them into the *git* repository.

Converting the raw experimental data to forms suitable for LaTeX, by contrast, is faster and often needed many times for the same experimental data while I was developing the document, so I gave this responsibility to the *Makefile*.

While this system worked, and the split between running the experiments "manually" and formatting them automatically worked well, adding a new table of results required adding consistent rules to the *nqueens* script, the *Makefile*, and the *nqueens.lhs* source file. In retrospect, I should have put something in the *nqueens.lhs* file that would generate rules for the script and *Makefile*, consolidating each experiment's specification in a single place.