COMS W4995 Parallel Functional Programming: SeedCracker Final Report

A Parallel Minecraft Seed Reverse Engineering Tool

Federick Gonzalez (fag2113), Justin Chen (jbc2186)

22 December 2021

Abstract

We implemented a Minecraft slime chunk-based seed reverse engineering tool in Haskell. We parallelized using various methods, achieving approximately 6x speedup on a 6-core processor. We achieved similar performance results with chunking using both Strategies with `parList` and the `Par` Monad, though overhead starts to dominate with larger numbers of chunks/sparks on `parList`. We also include other poorer parallelization techniques on two different sequential algorithms to demonstrate the impact of the sequential implementation on the parallelization.

1 Background

Minecraft is the best-selling video game of all time. The original game was written in Java in 2009 by Mojang, with development continuing through its official release in 2011 and up to the present day. The game is set in a blocky, procedurally generated 3D world, in which the player is free to explore, extract and farm resources, build, and (in multiplayer servers) interact with other players.

The procedural generation of this world is governed by a 64-bit "world seed." If the player does not manually enter in a seed, it is randomly generated. Although the seed is normally accessible to the player, the seed may be inaccessible for various reasons, such as the world file being lost, the world being on a multiplayer server, or the world belonging to a streamer or Mojang developer and as a result only accessible through video (or even just screenshots). Thus, whether out of nostalgia for

an old now-lost world, a desire to cheat, or as a love project from a fanbase, there is significant motivation to reverse engineer world seeds.

Brute forcing the 64-bit seed would take thousands of years, and would require knowledge of the game logic, so at first it would seem that reverse engineering the seed is an impossible task. However, Minecraft is written in Java. As a result, it has been fully decompiled, allowing for a complete look at the game code. From this, we know that Minecraft uses Java's Random class, which is not very secure. Java Random uses a linear congruential generator (LCG) with a modulus of 2^{48}:

$$seed_{n+1} = (25214903917 \cdot seed_n + 11) \bmod 2^{48}$$

Thus, although it takes a 64-bit seed, only the lower 48 bits are used for most generation. We therefore can attack only these 48 bits independently of the other 16. Furthermore, because randomly generated seeds also use Java Random (by generating two 32-bit integers and combining them together), for these seeds the 48-bit seed actually corresponds to only a single world seed on average, and can be extended to the 64-bit seed nearly instantly. If using a manually entered seed, the search space is larger, but can be brute-forced using other generation features which do use the upper 16 bits.

Although a brute-force attack on a 48-bit integer is feasible (and would take on the order of hours or days), we can further narrow down the search space by taking advantage of Java Random's poor parity. Current state of the art solutions involve combining various structure generation features to find the world seed, including an in-game client mod created by KaptainWutax in 2019 which calculates the seed in real time. One early technique involves the use of slime chunks; this algorithm will be the focus of this project.

2 Implementation Details

2.1 Explanation of Slime Chunk-Based Algorithm

The slime-based approach to seed reverse engineering was an early proof-of-concept, developed in 2014 by Tim Goddard and presented at the New Zealand security conference Kiwicon. An equivalent algorithm was also independently developed by Badel2 and released in 2017.

Slimes are hostile creatures which spawn underground. Their spawning locations

are determined by a grid of "chunks" in the world; chunks where slimes can spawn are called "slime chunks." When generating any given chunk, the game generates a seed by adding the world seed to a number generated from the chunk coordinates; if the random number generated from this seed is divisible by 10, then the chunk is a slime chunk:

```
Random rnd = new Random(seed + chunkVal) ^ 0x3ad8025f);
return rnd.nextInt() % 10 == 0;
```

Our general approach involves collecting slime chunk coordinates, and filtering all possible seeds by those which which generate a number which is divisible by 10 in said slime chunks. Because $10^{15} > 2^{48}$, 15 slime chunks should be theoretically sufficient to identify a seed (though with such a low number, the possibility of collisions is high).

Furthermore, we can optimize this solution by noting that all multiples of 10 are even. We can therefore filter only those seeds which will generate an even number. It turns out that when using Java Random's LCG, the lower 18 bits determine whether the final bit in the resulting number is 0 or 1, and therefore we can first reduce the search space by filtering the lower 18 bits. Each slime chunk essentially cuts the search space in half; with approximately 18 slime chunks, it becomes possible to find a single 18-bit suffix.

In that case, we are left with only $48 - 18 = 30$ bits left to brute-force, which will provide us with anywhere from a handful to several hundred seeds. This takes on the order of seconds to complete. If we provide fewer slime chunks, this leaves us with a larger search space and thus a longer running time, allowing us to therefore adjust the computational difficulty of the problem as necessary.

Finally, we can extend the 48-bit seed to a list of 64-bit seeds which can be generated by Java Random. When generating 64-bit seeds, Java Random actually generates two 32-bit seeds directly in order and concatenates them together. Since the second 32-bit number is determinable by the first number, and because the LCG only works in 48 bits, this essentially restricts the number of 64-bit seeds to 48 bits. As a result, by working backwards, we can extend every 48-bit seed to on average one 64-bit full seed.

In a practical scenario, we would imagine that a player goes around finding locations of slimes and marking those chunk coordinates down. This is then used as input into our program. The slime chunks cannot be recovered from the world file

except in the trivial case where we simply look at the world seed stored on the save file. It is also not practical to use the absence of a slime chunk to help with the reverse engineering, both because it would not help with the pre-filtering step, and also because it would be very easy for the player to mark down a false negative simply because they have not seen any slimes there yet. If there are multiple valid seeds, the player can filter down further by finding more slime chunks, or they can manually check the seeds for a match.

2.2 Sequential Implementation

We based our algorithms on C code by Tim Goddard, available on GitHub. The chunk coordinates are first read from file and converted to "chunk values" using some addition and multiplication. Care needs to be taken to the placement location and kind of type conversions used in order to replicate Java/C integer overflow behavior. This is done in `calcChunkVal`.

The attack takes three steps. First, in `calcLowerBitSeries`, we pre-filter the lower 18 bits of the seed based on whether it will lead to an even number or not (the reasoning for this is explained above) for the provided chunks. This is done using a filter on all possible 18-bit lower seeds, checking whether all the given chunk values lead to an even number based on the slime generation code for that seed.

Second, we filter on whether a seed will generate slime chunks for all the given slime chunks. We check all possible 30-bit partial seeds by combining them with the pre-filtered lower 18 bits and checking whether that 48-bit seed will generate the provided slime chunks. We first did this using a map and filter on a large lazy list, which is seen in `calcSlimeSeedsNaive`. We later implemented a recursive solution, seen in `calcSlimeSeeds`. This looped through all possible 30-bit numbers and cons'd the matching ones together. The lazy list solution was more optimized when running sequentially, but as we will see in the Performance section, the recursive solution is parallelizable without too much overhead.

Third, in `expand48To64Random`, we extend the 48-bit seeds to the possible 64-bit seeds generatable by Java Random. As explained above, this is on average a one-to-one mapping. We also make use of a filter on a large list to accomplish this.

To enable all of this functionality, we also implemented some helper functions. `randomNext` reimplements Java's LCG for the first two filtering steps, while

`randomReverse` is used to work backwards on the third extension step. The first two steps involve running Minecraft's slime chunk logic, which seeds the LCG using a combination of the world seed and the chunk value, and then performs some bit operations on it. We do this for each provided chunk and seed.

2.3 Parallelization

The primary limiting factor of this seed cracking algorithm is the brute force attack on the remaining possible seeds after the filtering on the lower 18 bits. The sequential implementation takes around 45 seconds, and the other steps take under a second combined to complete. Thus, this attack can be parallelized by partitioning the search space. We first implemented a naive partitioning technique using `parBuffer`. We then used "chunking," similar to Simon Marlow's K-means solution, to split up the search space into a user-input-defined number of blocks. We tested this on both `parList` (using `rseq` and `redeepseq`) and the `Par` Monad's `parMap`.

More details on the rationale and results of this parallelization are provided in the Performance section below.

3 Performance Measurements

Performance measurements were taken on a 2018 MacBook Pro with a 6-core 2.6 GHz Intel Core i7 CPU. We generated list of slime chunks from a real Minecraft seed and used that for our input; this is available at `vals.txt`.

3.1 Sequential Solutions

`calcSlimeSeedsNaive`, which performs a map and filter on a lazy list, took approximately 39 seconds to run. `calcSlimeSeeds`, which recursively loops on all possible values and builds them into a list, took about 45 seconds to run. This is likely because map and filter are well-optimized. However, combining the results together for `calcSlimeSeedsNaive` on all possible pre-filtered seed combinations required a custom `concatMap'` that performed strict evaluation in order to prevent overhead from dominating completely.

3.2 Naive `parBuffer` Parallelization

For illustrative purposes, we performed a naive parallelization using `parBuffer`, directly on the map and filter operations. This created $2^30 = 1,073,741,824$ sparks, causing the overhead to dominate. We experienced a 5x slowdown as the program took 225 seconds to complete (Fig. 1).

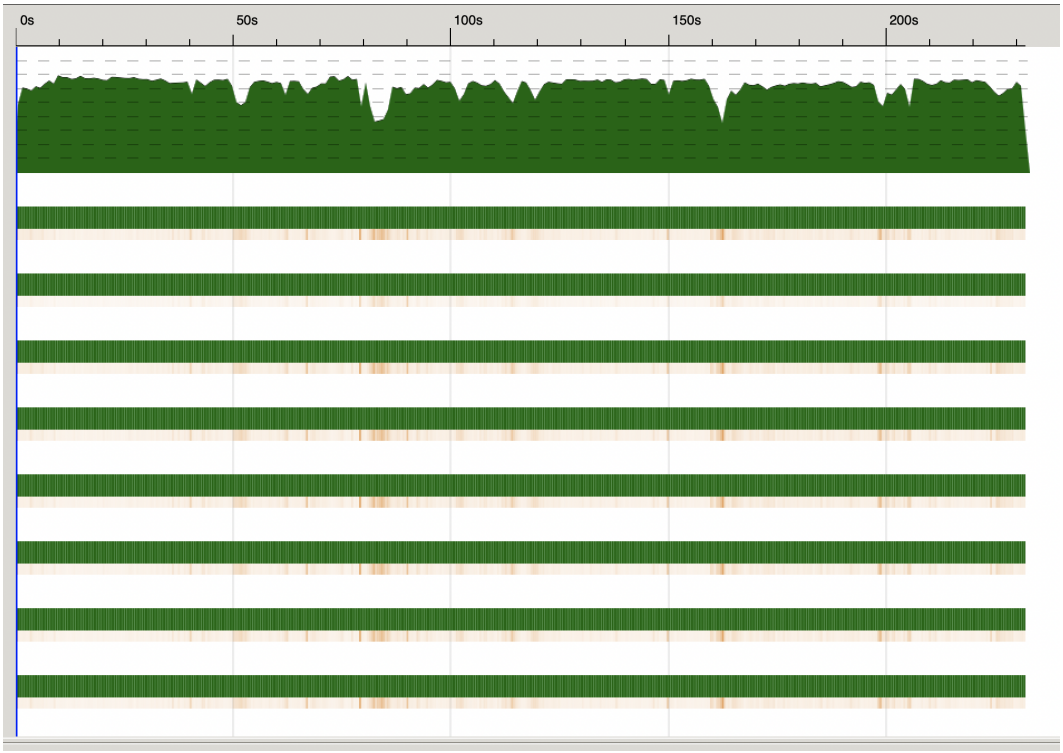

Figure 1: Naive `parBuffer` parallelization. Note the long running time.

3.3 Naive Parallelization with Chunking

Clearly, there needed to be fewer sparks. We took Marlow's parallel K-means as inspiration and performed chunking. We broke list up into blocks (to avoid confusion with slime chunks, we do not call them chunks), but since they were large lazy lists, we used our own list creation instead of directly splitting the large list. (This is also why a `parListChunk` directly on the map and filter would not work; the program spends the bulk of the time managing the large lazy list.) We then ran the sequential algorithm for each block before combining them back together (that last step took a trivial amount of time so there was no need to parallelize it).

However, with the naive solution (involving map and filter), this greatly strained the garbage collector. In fact, the majority of the time was spent garbage collecting; only approximately 40% of the time was actually productively running the algorithm (Fig. 2). As a result, overhead leads to worse running times than the sequential solution until approximately 64 blocks; the program is ultimately able to produce a 25% speedup compared to the sequential version, but that is not that much faster overall.

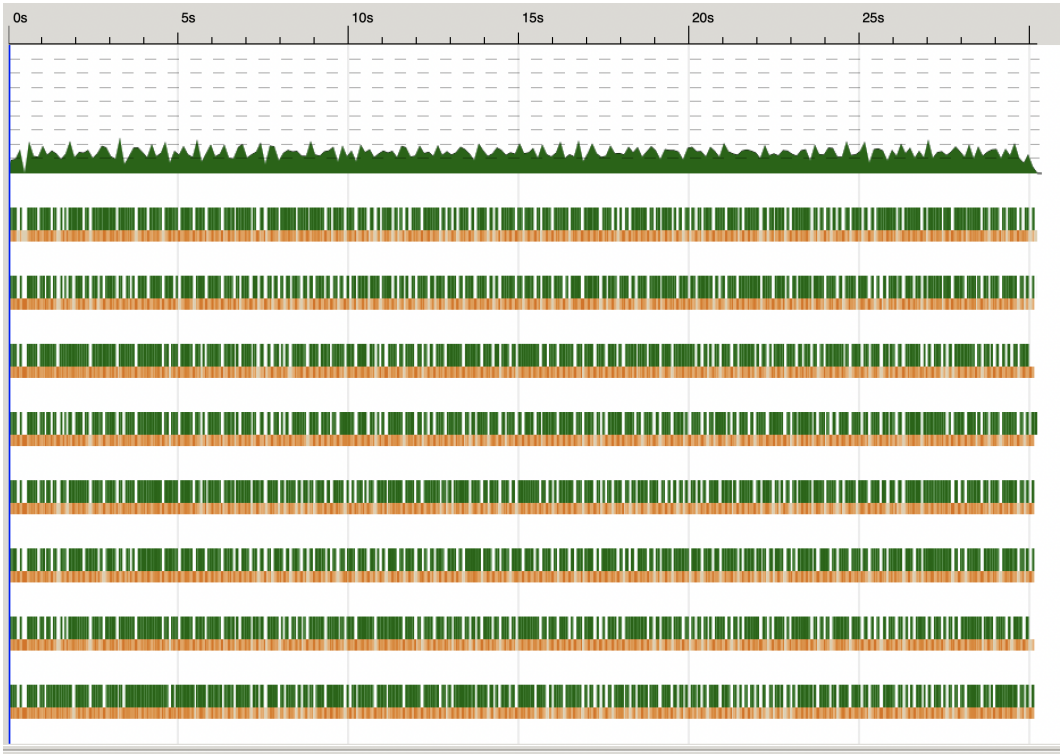

Figure 2: Naive `parList` parallelization as viewed on ThreadScope. Note the massive garbage collection.

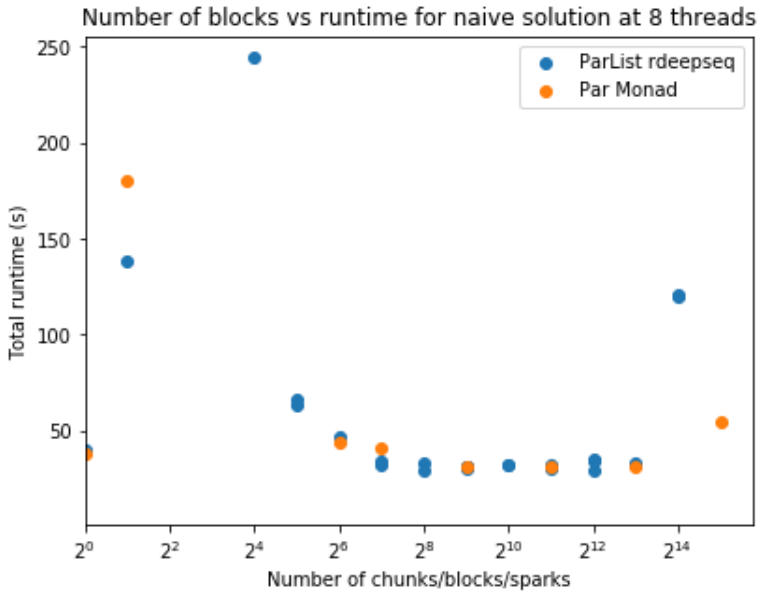

Figure 3: Running times for naive `parList` and `Par` Monad parallelization. Note the massive overhead costs at small block sizes.

3.4 Parallelizing the Recursive Solution

The issues with the massive garbage collection overhead motivated our switch over to a recursive implementation of `calcSlimeSeeds`, which is described in the Sequential Implementation section above. Using this, we get a dramatic speedup to approximately 7 seconds. An analysis on ThreadScope shows work being distributed evenly across all threads (Fig. 4). This demonstrates the inefficiencies of the large lazy list approach when it comes to attempting to parallelize, and the need for a sequential algorithm that will not have such overhead, even when parallelizing. With this solution, we are getting a near-optimal speedup: a 6x speedup on a 6-core processor.

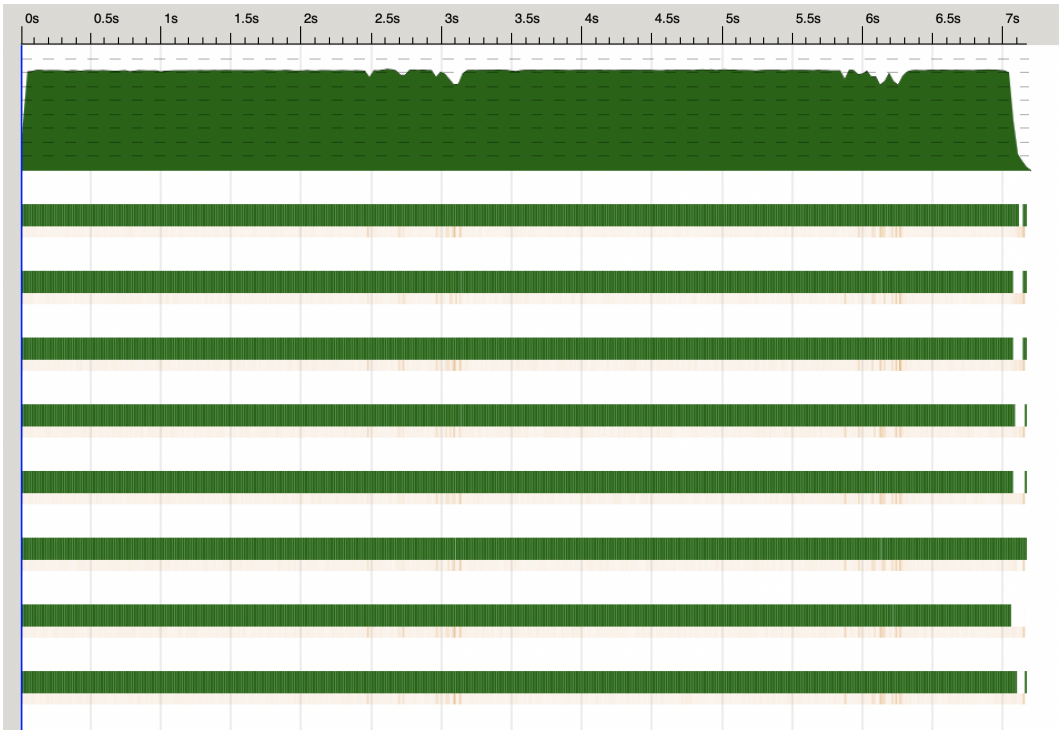

Figure 4: `Par` Monad parallelization as viewed on ThreadScope. Note the efficient and even distribution of work.

3.5 Strategies vs. `Par` Monad

We get nearly the same efficiency and speedup when using Strategies compared to the `Par` Monad. Interestingly, unlike with our `ParList` solution, overhead does not dominate for the `Par` Monad even at very large block numbers (Fig. 5). We speculate that this is due to the `Par` Monad's implementation of parallelization, which uses a load balancing scheduler instead of creating discrete sparks, and as such does not lead to the same overhead that creating tens of thousands of sparks would.

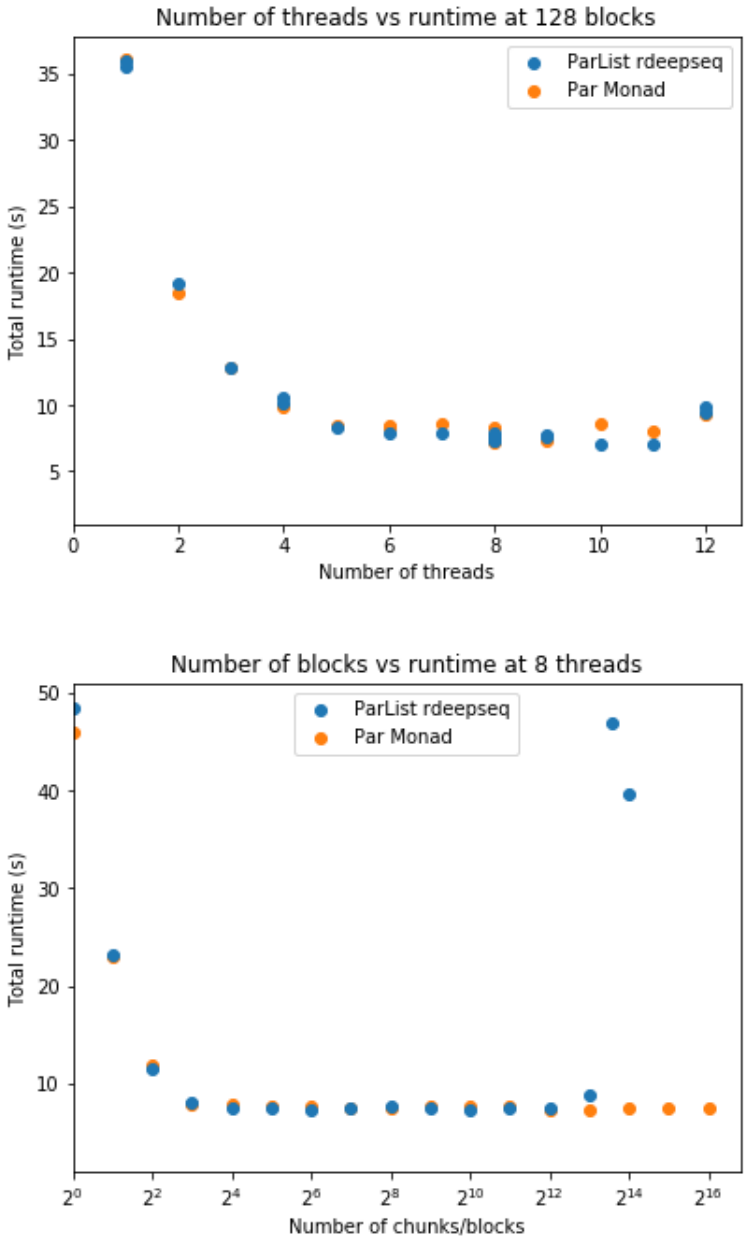

Figure 5: Running times for `parList` and `Par` Monad parallelization for different thread counts and block counts. Note how overhead begins to dominate at low and high block numbers, except the `Par` Monad handles large block numbers well.

4 Future Work

Given more time, we would like to further investigate the possibilities for optimizing the sequential algorithm. This may give us additional speedup, as the original C code is faster than ours even when running sequentially, so there is room for improvement. We would also like to add support for biome-based seed reverse engineering for the remaining 16 bits, instead of inferring based on Java Random's vulnerabilities; this would allow for more accurate seed cracking and work for non-randomly generated seeds. Finally, we would like to investigate if it is possible to create an efficiently parallelizable solution using library functions instead of direct recursion. Map and filter on a large lazy list don't work well due to the overhead, but there may be some list building functions which may work.

5 Source Code

Cracker.hs

```
module Cracker(calcChunkVal, calcLowerBitSeries, expand48To64Random, calcSeq,
calcParListBlocks, calcSeqNaive, calcParListBlocksNaive, calcParBufferNaive,
calcParMonadBlocks, calcParMonadBlocksNaive) where
import Data.Bits(Bits(shiftR, shiftL, xor, (.&.), (.|.)))
import Data.Int(Int32, Int64)
import Data.Word(Word32, Word64)
import Data.List(foldl')
import Control.Parallel.Strategies(using, parList, rdeepseq, parBuffer)
import Control.Monad.Par(runPar)
import Control.Monad.Par.Combinator(parMap)

-- adapted from / inspired by https://stackoverflow.com/q/43033099
concatMap' :: Foldable t => (a -> [b]) -> t a -> [b]
concatMap' f = reverse . foldl' (flip ((++) . f)) []

mask48Bit :: Word64
mask48Bit = 1 `shiftL` 48 - 1

calcChunkVal :: (Int32, Int32) -> Int64
calcChunkVal (x, z) = xsqv + xv + zsqv + zv
    where xsqv = fromIntegral (xw * xw * 0x4c1906)
          xw   = fromIntegral x :: Word32
          xv   = fromIntegral $ x * 0x5ac0db
          zsqv = fromIntegral (z * z) * 0x4307a7
          zv   = fromIntegral $ z * 0x5f24f
```

```haskell
randomNext :: Word64 -> Word64
randomNext seed = (seed * 0x5deece66d + 0xb) .&. mask48Bit

randomReverse :: Word64 -> Word64
randomReverse seed = ((seed - 0xb) * 246154705703781) .&. mask48Bit

matches :: Int64 -> Int64 -> Bool
matches seed chunkVal = doShift random == 0 where
    random = fromIntegral ((seed + chunkVal) `xor` 0x5e434e432) .&. mask48Bit

checkEven :: Int64 -> Int64 -> Bool
checkEven seed chunkVal = even $ doShift random where
    random = fromIntegral ((seed + chunkVal) `xor` 0x5e434e432) .&. mask48Bit

doShift :: Word64 -> Word64
doShift random
    | bits - val + 9 < 0 = doShift nextRandom
    | otherwise = val
    where nextRandom = randomNext random
          bits = nextRandom `shiftR` 17
          val = bits `mod` 10

calcLowerBitSeries :: [Int64] -> [Int64]
calcLowerBitSeries chunkVals = filter (flip all chunkVals . checkEven)
                               [0 .. 1 `shiftL` 18 - 1] :: [Int64]

calcSlimeSeedsNaive :: [Int64] -> [Int64] -> Int64 -> [Int64]
calcSlimeSeedsNaive chunkVals seeds lowerBits
    = filter (flip all chunkVals . matches)
        $ map ((.|. lowerBits) . (`shiftL` 18)) seeds

calcSlimeSeedsParBuffer :: [Int64] -> [Int64] -> Int64 -> [Int64]
calcSlimeSeedsParBuffer chunkVals seeds lowerBits
    = filter (flip all chunkVals . matches)
    (map ((.|. lowerBits) . (`shiftL` 18)) seeds `using` parBuffer 100 rdeepseq)
    `using` parBuffer 100 rdeepseq

calcSlimeSeeds :: [Int64] -> (Int64, Int64) -> Int64 -> [Int64]
calcSlimeSeeds chunkVals (seed, endSeed) lowerBits
    | seed == endSeed = []
    | all (matches fullSeed) chunkVals = fullSeed : restSeeds
    | otherwise = restSeeds
    where fullSeed = (seed `shiftL` 18) .|. lowerBits
          restSeeds = calcSlimeSeeds chunkVals (seed + 1, endSeed) lowerBits
```

```
expand48To64Random :: Int64 -> [Int64]
expand48To64Random seed = map ((.|. lowerInt) . (`shiftL` 32) . subtract offset)
    $ filter ((upperPartial ==) . (.&. mask16Bit)) $ map
    (fromIntegral . (`shiftR` 16) . randomReverse . fromIntegral . (middle .|.))
    [0 .. mask16Bit]
    where mask16Bit = 1 `shiftL` 16 - 1
          lowerInt = seed .&. (1 `shiftL` 32 - 1)
          middle = lowerInt `shiftL` 16
          offset = (seed .&. (1 `shiftL` 31)) `shiftR` 31
          upperPartial = (seed `shiftR` 32 + offset) .&. mask16Bit

calcSeqNaive :: [Int64] -> [Int64] -> [Int64]
calcSeqNaive chunkVals
    = concatMap' (calcSlimeSeedsNaive chunkVals [0 .. 1 `shiftL` 30 - 1])

calcParBufferNaive :: [Int64] -> [Int64] -> [Int64]
calcParBufferNaive chunkVals
    = concatMap' (calcSlimeSeedsParBuffer chunkVals [0 .. 1 `shiftL` 30 - 1])

calcParListBlocksNaive :: Int64 -> [Int64] -> [Int64] -> [Int64]
calcParListBlocksNaive numBlocks chunkVals lowerBits
    = concat (map (flip concatMap' lowerBits . calcSlimeSeedsNaive chunkVals)
        blocks `using` parList rdeepseq)
    where
        step = 1 `shiftL` 30 `quot` numBlocks
        blocks = map (\i -> [i .. i + step - 1]) [0, step .. 1 `shiftL` 30 - 1]

calcParMonadBlocksNaive :: Int64 -> [Int64] -> [Int64] -> [Int64]
calcParMonadBlocksNaive numBlocks chunkVals lowerBits = concat $ runPar
    $ parMap (flip concatMap' lowerBits . calcSlimeSeedsNaive chunkVals) blocks
    where
        step = 1 `shiftL` 30 `quot` numBlocks
        blocks = map (\i -> [i .. i + step - 1]) [0, step .. 1 `shiftL` 30 - 1]

calcSeq :: [Int64] -> [Int64] -> [Int64]
calcSeq chunkVals
    = concatMap (calcSlimeSeeds chunkVals (0, 1 `shiftL` 30 - 1))

calcParListBlocks :: Int64 -> [Int64] -> [Int64] -> [Int64]
calcParListBlocks numBlocks chunkVals lowerBits
    = concat (map (flip concatMap lowerBits . calcSlimeSeeds chunkVals) blocks
        `using` parList rdeepseq)
    where
        step = 1 `shiftL` 30 `quot` numBlocks
        blocks = map (\i -> (i, i + step - 1)) [0, step .. 1 `shiftL` 30 - 1]
```

```
calcParMonadBlocks :: Int64 -> [Int64] -> [Int64] -> [Int64]
calcParMonadBlocks numBlocks chunkVals lowerBits = concat $ runPar
    $ parMap (flip concatMap lowerBits . calcSlimeSeeds chunkVals) blocks
    where
        step = 1 `shiftL` 30 `quot` numBlocks
        blocks = map (\i -> (i, i + step - 1)) [0, step .. 1 `shiftL` 30 - 1]
```

Main.hs

```
module Main where

import Cracker
import System.Environment(getArgs)
import Control.Monad(unless)
import System.Exit(die)
import Data.List(intercalate)
import Data.Ix(inRange)

main :: IO ()
main = do
    let types = ["seq", "parList", "parBuffer", "parMonad"]
    args <- getArgs
    unless (inRange (2, 4) (length args) && args !! 1 `elem` types) $
        die $ "Usage: <filename> (" ++ intercalate "|" types ++
            ") <num-blocks> [naive]"
    let numBlocks = case args of
            _ : _ : num : _ -> read num
            _ -> 128
    chunkCoords <- case args of
        filename : _ -> do
            text <- readFile filename
            return $ map (((\[x, z] -> (x, z)) . map read) . words) (lines text)
        _ -> return []
    let chunkVals = map calcChunkVal chunkCoords
    putStrLn "Calculated the following chunk values from the provided \
        \chunk coordinates:"
    print chunkVals
    let lowerBits = calcLowerBitSeries chunkVals
    putStrLn "Calculated the following valid lower 18 bits:"
    print lowerBits
    let seeds = case args of
            [_, "parList", _, "naive"] ->
                calcParListBlocksNaive numBlocks chunkVals lowerBits
            [_, "seq", _, "naive"] -> calcSeqNaive chunkVals lowerBits
            [_, "parMonad", _, "naive"] ->
```

```
                calcParMonadBlocksNaive numBlocks chunkVals lowerBits
            _ : "parList" : _  ->
                calcParListBlocks numBlocks chunkVals lowerBits
            _ : "seq" : _ -> calcSeq chunkVals lowerBits
            _ : "parBuffer" : _  -> calcParBufferNaive chunkVals lowerBits
            _ : "parMonad" : _ ->
                calcParMonadBlocks numBlocks chunkVals lowerBits
            _ -> [0]
    putStrLn "Calculated the following valid 40-bit seeds:"
    print seeds
    let fullSeeds = concatMap expand48To64Random seeds
    putStrLn "Calculated the following valid 64-bit randomly generatable seeds:"
    print fullSeeds
```