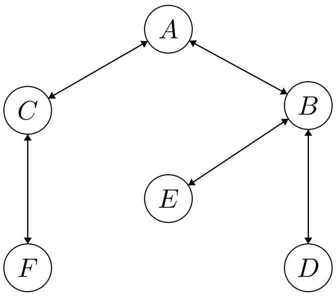
Minimum Vertex Cover

Andre Mao (am5994) Tony Giannini (aug2102)
Minh Hien Tran (mt3854)

December 18, 2024

1 Introduction

Our project examines a quintessential NP-Complete problem: Minimum Vertex Cover. A **vertex cover** of a graph is a subset of the graph's vertices such that every edge in the graph is incident to at least one vertex in this subset. The smallest subset of vertices that is a vertex cover is the **minimum** vertex cover. For example, if we consider the following graph, the minimum vertex cover is B, C, as all edges of the graph have at least one endpoint in this set. Our project builds on the ideas set out in [1].



Being an NP-complete problem, this cannot be solved in polynomial time using any known algorithm. Given a graph $G = (V, E)$, we want to find the set of vertices V' that form the minimum vertex cover. A brute-force solution sequentially examines all subsets of increasing size as possible MVCs. Our goal is to parallelize this brute-force solution.

2 GHC Options

All programs were tested using the `-threaded, -rtsopts`, and `-O2` options.

3 Graph Generation

For graph generation, we created a graph generator to generate a random graph with n integer vertices and m undirected edges. The resulting graph is stored in a file as an edge list in the `/data` folder. The generator can be used to generate a graph with n vertices and m edges by running the command `$ stack exec graph-generator n m`. The source code for this graph generator can be found in `/src/GraphGenerator.hs`. There is also a function in this module, *fileToAdjList*, that takes an edge file and returns an adjacency list. The adjacency list is represented as a 1-indexed *Data.Vector* [2] *IntSets* [3]. To find the neighbors of a node, you can index into the vector (in constant time) using the node's value. A node's neighbors are stored in an *IntSet*. The first step in all of our solutions is calling *fileToAdjList* with the desired edge file. This contributes to the inherently sequential portion of each algorithm.

4 Sequential Solutions

This code was developed and tested on an Apple M1 Pro machine with 8 cores and threads.

4.1 Sequential Version 1 — DFS Subset Generation

The source code can be found in `/src/SequentialV1.hs`.

We begin investigating a sequential brute-force solution as described in the introduction. We start with the most naïve approach. Here, we generate all subsets of V of size 1 and then check if any of these subsets form a minimum vertex cover. If we find no solution, then we generate all subsets of size 2 and check again. We repeat this process for increasing subset sizes until we find a vertex cover. For every size k, we generate a list of lists (*Data.List*) that stores all subsets using a recursive DFS approach. For each node in the graph, we either include or exclude it in the subset. When we include, we solve the problem for a k that is 1 smaller and cons this element to each subset generated from solving that smaller problem. When we exclude, we solve the problem for the same k (without this element being a choice) and don't attach anything to those results. We pool subsets generated from both options during each call, using this as the return value. We concatenate the list of subsets from the *include decision* with the list from the *exclude decision*, which is rather inefficient since list concatenation in Haskell is linear in its first argument. Each subset is checked to see if it satisfies the condition by cycling through every edge in the graph and verifying that one of the endpoints is included in the subset. Thus, the checking process does not use any substantial new memory. This solution runs in $O(n2^n)$.

The key functions in this implementation are as follows:

```
genSubsets :: [a] -> Int -> [[a]]
genSubsets _ 0 = [[]]
genSubsets [] _ = []
genSubsets (x:xs) k = map (x:) (genSubsets xs (k-1)) ++ genSubsets xs k
```

Generates all subsets of a given size k

```
verifyVertexCover :: [Int] -> Vector IntSet -> Bool
verifyVertexCover chosen adj =
    let isCovered u v = u ‘elem‘ chosen || v ‘elem‘ chosen
    in all (\u -> IS.foldl’
        (\acc v -> acc && isCovered u v) True (adj ! u)) [1..n]
```

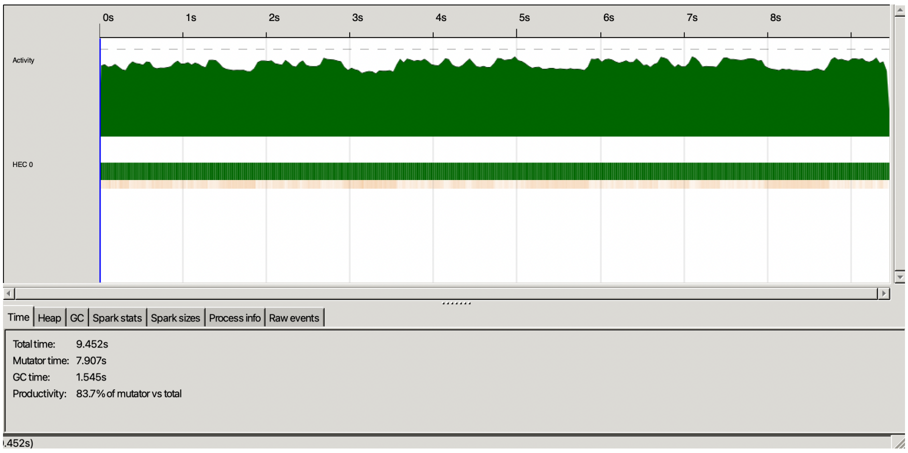
Checks if a subset of vertices covers all edges

```
solve :: Vector IntSet -> [Int]
solve adj =
    let vertices = [1..V.length adj - 1]
    in search 1 vertices
  where
    search size vs =
        case filter (‘verifyVertexCover‘ adj) (genSubsets vs size) of
            []      -> search (size + 1) vs
            (sol:_) -> sol
```

Iteratively generates and tests subsets of increasing sizes to find a minimal vertex cover.

In this approach, we use a lot of memory as we are always storing all subsets of a certain size. This leads to the Threadscope graph shown in 1. With a 26 node, 38 edge graph, the total time this algorithm takes is 9.452s. Productivity is at 83.7%. To replicate these results, run `$ stack test :sv1 --ta "+RTS -l -RTS"`.

The MVC is $[1, 2, 4, 6, 8, 10, 11, 13, 15, 17, 19, 21, 23, 26]$. There are 14 nodes, which indicates that all subsets of sizes from 1 to 14 were generated and checked.
Note that in the Threadscope graph, there is quite a lot of garbage collection. On one core, the generation and storing of subsets is very memory inefficient.



Total time: 9.452s
Mutator time: 7.907s
GC time: 1.545s
Productivity: 83.7% of mutator vs total

Figure 1: Sequential Brute Force

4.2 Sequential Version 2 - Efficient Subset Generation

The source code can be found in `/src/SequentialV2.hs`.

Another sequential solution attempts to use compact, non-lazy data structures to solve this problem with minimal operations. Here, we use a *Seq* (Haskell's name for a non-lazy deque). This data structure allows for O(1) removal and addition at each end. We start by putting all subsets of size 1 on the deque along with their respective covers and extension positions. Elements on the deque are tuples of the form (*subset*, *cover*, *idx*). A *cover* is all edges that are incident to at least one node in the *subset* and *idx* is the index from which we can add nodes to this subset to form larger new subsets (this avoids repeat work).

The whole idea here is that subsets are built from other smaller subsets, and their respective edge covers are extensions of covers from smaller subsets. The algorithm works as follows: we pop from the front of the deque and we see all the larger subsets that can be formed from this subset. If any of these solve the problem, we return the MVC. Otherwise, each larger subset is added to the end of the deque. Both subsets and edge covers are stored using IntSets (compact and non-lazy).

```
encodeEdge u v
    | u < v     = u * 10000 + v
    | otherwise = v * 10000 + u
```

Encodes an edge as a single integer for efficiency.

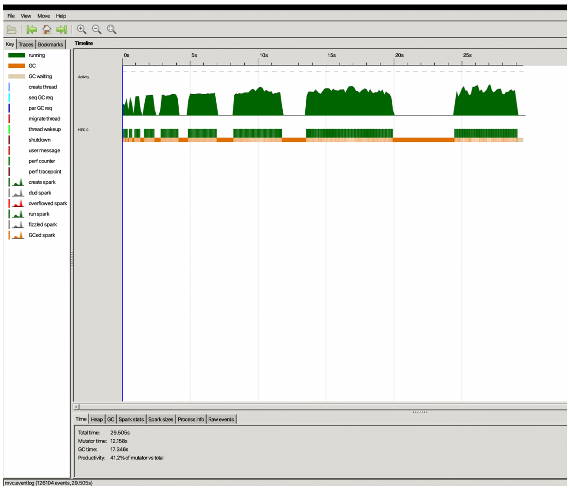
```
processVertex subsets v subset cover =
    let newSubset = IS.insert v subset
        newCover = IS.foldl’ (\acc u -> IS.insert (encodeEdge v u) acc)
             cover (adjList V.! v)

    in if IS.size newCover == m
       then ((Seq.singleton (newSubset, newCover, v + 1)), n + 1)
       else ((subsets Seq.|> (newSubset, newCover, v + 1)), v + 1)
```

Forms new subsets by adding a vertex and extending the cover with corresponding edges.

```
enqueueNextSubsets (subsets, idx) subset cover
    | idx > n = subsets
    | otherwise =
        enqueueNextSubsets
            (processVertex subsets idx subset cover) subset cover
```

Adds all possible extensions of a subset to the queue.

Even though we store more data at a time, we attempt to beat the former solution's runtime by having fewer operations. We hope that the compact data structures will offset the need for storing more state here. The code was run with the same 26-node, 38-edge graph as before, but the results were disappointing. As can be seen in 2, garbage collection killed us. The reason for the long pauses that increase in length as time goes by is that the deque we are working with only ever grows. When a subset is popped from the front, many are added to the back. This data moves into the *old generation* heap sector, and when a major GC event occurs, the *old generation* heap must be processed. This processing time gets longer and longer as this sector of the heap grows. This took 29.5 seconds. To replicate these results, run `$ stack test :sv2 --ta "+RTS -l -RTS"`.



```
Total time:     29.505s
Mutator time:   12.158s
GC time:        17.346s
Productivity:   41.2% of mutator vs total
```

mvc.eventlog (126104 events, 29.505s)

Figure 2: Sequential Deque

Although operations were minimized, the fact that more state had to be stored for each subset led to a slower solution. Since the first sequential solution was more efficient and didn't have as much of a GC issue, we focused on parallelizing it first.

5 Parallel Solutions

There are two parts of the first sequential algorithm that lend themselves to parallelization: subset generation and subset cover verification. Our first parallel approaches focus on parallelizing these components with different techniques to balance memory usage and runtime. All following tests are run on the same 26-node, 38-edge graph from the sequential analysis.

This code was developed and tested on an Apple M1 Pro machine with 8 cores and threads.

5.1 Parallel Version 1 — Depth-Bounded Parallel Subset Generation with Chunk-Based Parallel Subset Verification

The source code can be found in `/src/ParallelV1.hs`.

Our first attempt at parallelization combined a few different ideas to tackle the two components of the problem. In this attempt, we use the same approach and data structures as described in *Sequential Version 1*. We tweaked the subset generator to use **par** and **pseq** to generate subsets recursively in parallel up to a certain depth. We added a fallback mechanism to return to sequential mode when exceeding this depth. Pseq is used to force evaluation of the *exclude the current element* call. The list concatenation operator, which is strict in its first argument, is used to force evaluate the *include the current element* call. As described in *Sequential V1*, list concatenation is used because we return a list of subsets of size k. We concatenate subsets generated by including the first element and those generated without it. *Depth* was hard-coded as 3 here because this was being tested on an 8-core machine. We figured a depth of 3 would lead to 8 parallel tasks (2^3) because we have 2 recursive calls (include and exclude) in our generation function. We later found out that a larger depth was more optimal with an approach like this because the tasks are uneven. We also created a parallel verification process where the current list of subsets of a certain size is divided into 1000-element segments (we guessed this would lead to decent load-balancing and were proven correct). Parlist with rdeepseq (to force parallel evaluation) is used to map the verification function over each chunk of the subsets. If a subset is found to be a vertex cover, it is included in the output of the call to parlist and returned after the processing of the current batch of subsets concludes. Due to the nature of increasing subset sizes, the subset returned will be an MVC.

```
genSubsetsParallel :: Int -> [a] -> Int -> [[a]]
genSubsetsParallel depth xs k
    | k == 0      = [[]]
    | null xs     = []
    | depth <= 0  = genSubsets xs k -- Fallback to sequential
    | otherwise   =
        case xs of
            (x:xs') ->
                let withX = map (x:) (genSubsetsParallel (depth-1) xs' (k-1))
                    withoutX = genSubsetsParallel (depth-1) xs' k
                -- ++ is strict in the first argument, so we have both children
                in withX `par` (withoutX `pseq` (withX ++ withoutX))
            [] -> [] -- Surpress warning
```

Recursively generates subsets of size k in parallel with depth control fallback

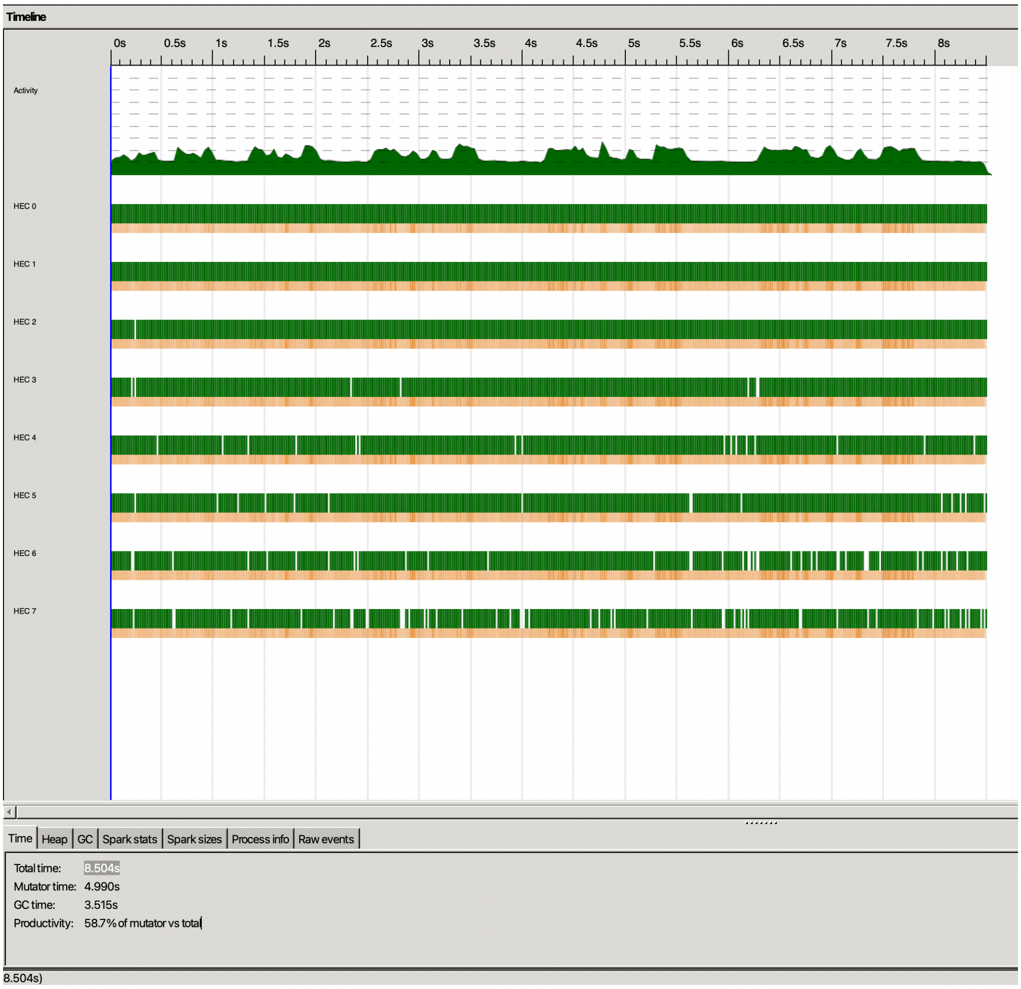
```
solve :: Vector IntSet -> [Int]
solve adj =
    let n = V.length adj - 1
        vertices = [1..n]
        depth = 3 -- Control parallel recursion depth
        chunkSize = 1000 -- Chunks of subsets to verify in parallel
    in search 1 vertices depth chunkSize
    where
        search size vs depth chunkSize =
            let subsets = genSubsetsParallel depth vs size
                results = map checkChunk (chunk chunkSize subsets)
                            `using` parList rdeepseq
            in case concat results of
                []      -> search (size+1) vs depth chunkSize
                (sol:_) -> sol
```

Finds a minimal vertex cover by recursively generating subsets and verifying them in parallel

This parallelization approach proved very costly, as shown in the Threadscope graph in 3. Taking 8.504s at 58.7% productivity on 8 cores, this parallel approach performed only marginally better than the sequential version it's adapted from. Note that nearly half of the algorithm's running time is used for garbage collection, as evidenced by the productivity statistic, orange bars, and low utilization of cores. The reason for so much more GC time here than in the sequential version is that there are 8 times as many nurseries and when one gets full a synchronization between threads must occur so data can be moved into the main heap or be freed. More threads mean more nurseries, more frequent triggering of GCs, and more synchronization overhead during collections. These results can be replicated by running `$ stack test :pv1 --ta "+RTS -l -N8 -RTS"`.

5.2 Parallel Version 1.5 — Increased Nursery Size

We wanted to reduce the amount of GC time in our program, since that was our bottleneck. To do so, we increased the nursery size to 128M by using the **-A128M** flag when executing the test. It is actually recommended for parallel programs to increase nusery size in official RTS documentation



```
Total time:     8.504s
Mutator time:   4.990s
GC time:        3.515s
Productivity:   58.7% of mutator vs total
8.504s)
```

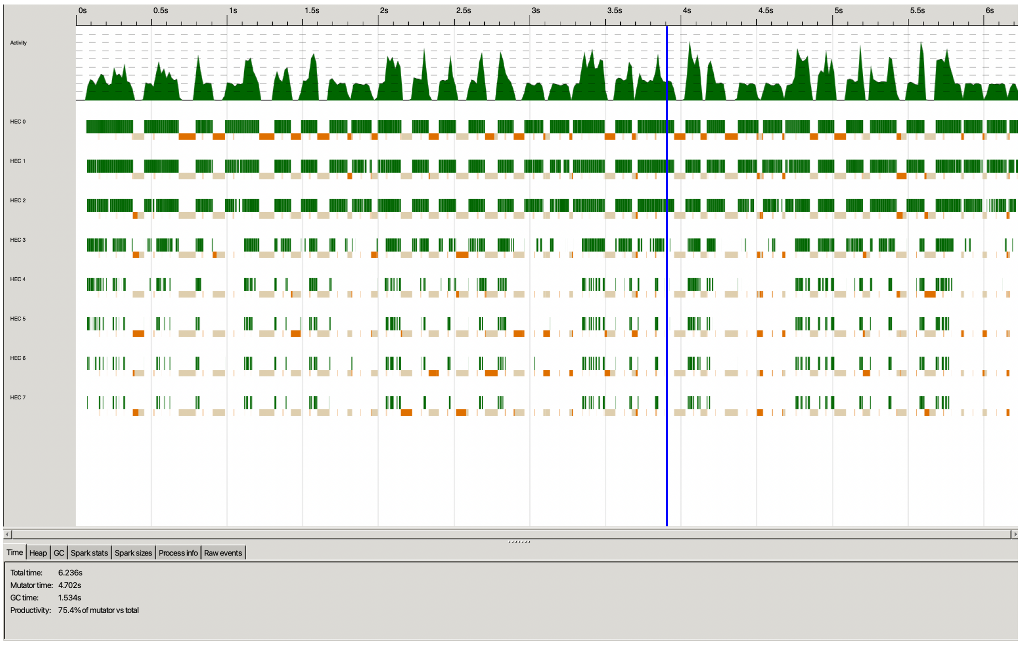
Figure 3: Parallel v1

[4], so we will be doing this from now on. The default size is 512kb. As a result, the algorithm finished on 8 cores in 6.236s with 75.4% Productivity. These results can be replicated by running `$ stack test :pv1 --ta "+RTS -l -N8 -A128M -RTS"`.
Note that while the running time has improved, the same amount of garbage collection has to occur. The speedup comes from the fact that garbage collections are less frequent so the overhead of synchronization is less (GC halts all threads from program execution). By increasing the nursery size, we see much higher core utilization, but the high peaks and low valleys indicate that garbage collection is still much too common for effective parallelization to be in use. This is confirmed when we zoom in and see that while one core is garbage collecting the other cores are sitting around and doing nothing. The Threadscope graph is presented in 4.

5.3 Parallel Version 2 — Cached Subset Generation

The source code can be found in `/src/ParallelV2.hs`.

When thinking about our GC problem, we noted that all new subsets were generated from scratch, causing many of the same subsets to be regenerated repeatedly. Our recursive DFS function creates all subsets of size k - 1 to generate all subsets of size k. Then when we ask for subsets of size k + 1, all subsets of sizes k and k - 1 are generated again. We came up with a new approach that used a Map to store all subsets of size 1 to generate size 2, all subsets of size 2 to generate size 3, etc. We kept our verification component the same but the subset generation process now went back to being done sequentially. This made the code simpler, and was a good starting point to see if we could improve memory efficiency. The generation function takes a lazy map (*Data.Map*) that



```
Total time:     6.236s
Mutator time:   4.702s
GC time:        1.534s
Productivity:   75.4% of mutator vs total
```

Figure 4: Parallel v1 with -A128M Flag

maps a tuple: (elementsToChooseFrom *Data.List*, subsetSize *Int*) to a list of subsets (*Data.List*) of subsetSize. Subsets are generated in the same DFS manner (choice of including first element in our elementsToChooseFrom or not), but we pass the cache from function call to function call and always return the updated cache and the results. For example, when we are looking for subsets of size 4, every time we make the 'include' choice, we can tack on the current element to the cached result for subsets of size 3. By doing this, we achieved a much faster, but far less memory efficient version of the algorithm. This version took 4.1s at 45.4% productivity, with a maximum heap residence of 1.8 GiB. The Threadscope graph is presented in 5. These results can be replicated by running `$ stack test :pv2 --ta "+RTS -l -N8 -A128M -RTS"`

```
genSubsetsCache :: (Ord a) => [a] -> Int -> Map.Map ([a], Int) [[a]] ->
    ([[a]], Map.Map ([a], Int) [[a]])
genSubsetsCache _ 0 cache = ([[]], cache)
genSubsetsCache [] _ cache = ([], cache)
genSubsetsCache list n cache
  | Map.member (list, n) cache = (cache Map.! (list, n), cache)
  | otherwise =
      let (x:xs) = list
          (withX, cache1) = genSubsetsCache xs (n - 1) cache
          (withoutX, cache2) = genSubsetsCache xs n cache1
          result = (map (x:) withX ++ withoutX)
          updatedCache = Map.insert (list, n) result cache2
      in (result, updatedCache)
```

Generates subsets of size n from the given list and uses a cache to store and reuse subsets of smaller sizes.

We note that while the algorithm's speed has increased, the terribly low productivity and high maximum heap residency don't make this a great option for parallelization. Most of the time during garbage collection, there are cores sitting around and doing nothing so parallelizing this algorithm is not worth it. The idea here was that preventing the regeneration of so many subsets would decrease memory usage and garbage collection time. Despite our cache preventing subset regeneration, its values are massive and there are many of them. These values need to be frequently processed by the garbage collector during major collections.

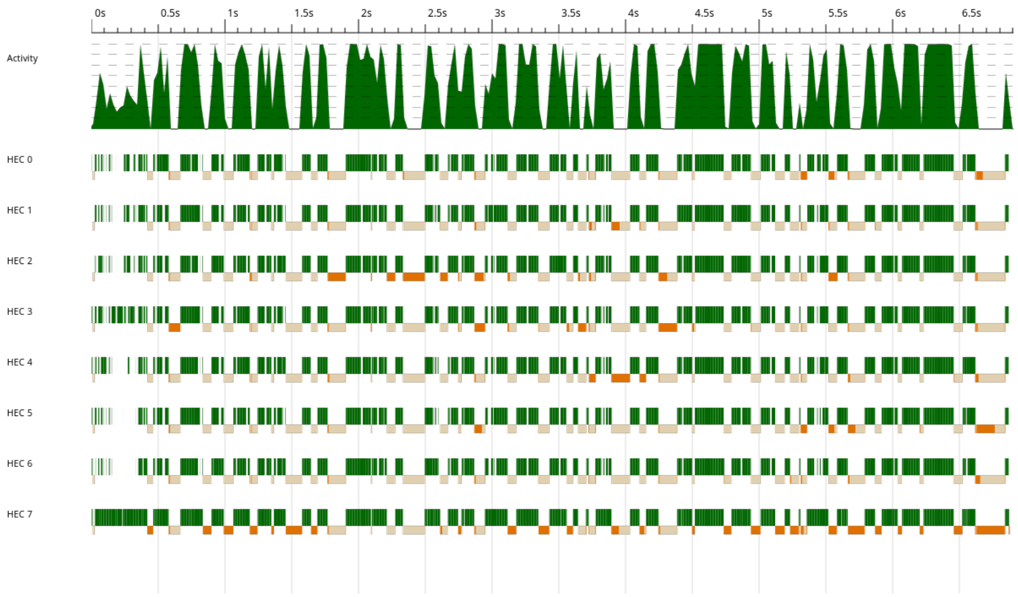


Figure 5: Parallel v2

6 Memory-Efficient Solutions

This code was developed and tested on an Apple M3 Pro with 12 cores and threads. The same graph was used.

6.1 Parallel Version 3 — Top-Down DFS Subset Generation

One of the biggest issues with our last approaches was that we always stored all subsets of a certain size and then scanned them to check if any solved our problem. What if we could check a whole batch of subsets of a size while only storing a small amount of subsets at a time?

In all previous versions of our subset generation process, we tack on nodes to recursive results to produce subsets of the target size. This means that we acquire our subsets after the call stack rolls up. Much state is returned from recursive calls (many subsets of smaller sizes). The idea here is that instead of tacking the current node onto the recursive result during our *include* decision, we can just pass it onto the recursive call in another function parameter. This way the the full subsets will be available at the leaf nodes in our recursive decision tree. When a leaf node or full subset of size k is reached deep in the recursion process, it can just be checked right then and there. This eliminates the need to separate subset generation from checking and storage of many subsets at a time. There only needs to be as many subsets around as there are recursive threads. These subsets accumulate elements until they reach the target size. This approach combined with some other slight optimizations led to a fast, memory efficient solution that responded very well to parallelism and had no GC problem. Below are the key functions. This algorithm is an adaptation of *Parallel V1*.

```
verifyVertexCover :: IS.IntSet -> V.Vector IS.IntSet -> Bool
verifyVertexCover chosen adj =
    let n = V.length adj - 1
        isCovered u v = u ‘IS.member‘ chosen || v ‘IS.member‘ chosen
        checkEdges u = IS.foldr (\v acc -> acc && isCovered u v)
            True (adj V.! u)
    in all checkEdges [1..n]
```

This is the first time *verifyVertexCover* has been altered since it was written for *Sequential V1*. Now instead of storing a subset with a list we are using an *IntSet*. This data structure is more compact than a list, leading to faster membership checks. Also, instead of foldl' for checking if a set of edges are covered, we are using foldr. This is because foldr short-circuit evaluates. When an edge is found not be covered, the foldr expression immediately evaluates to false. [5]

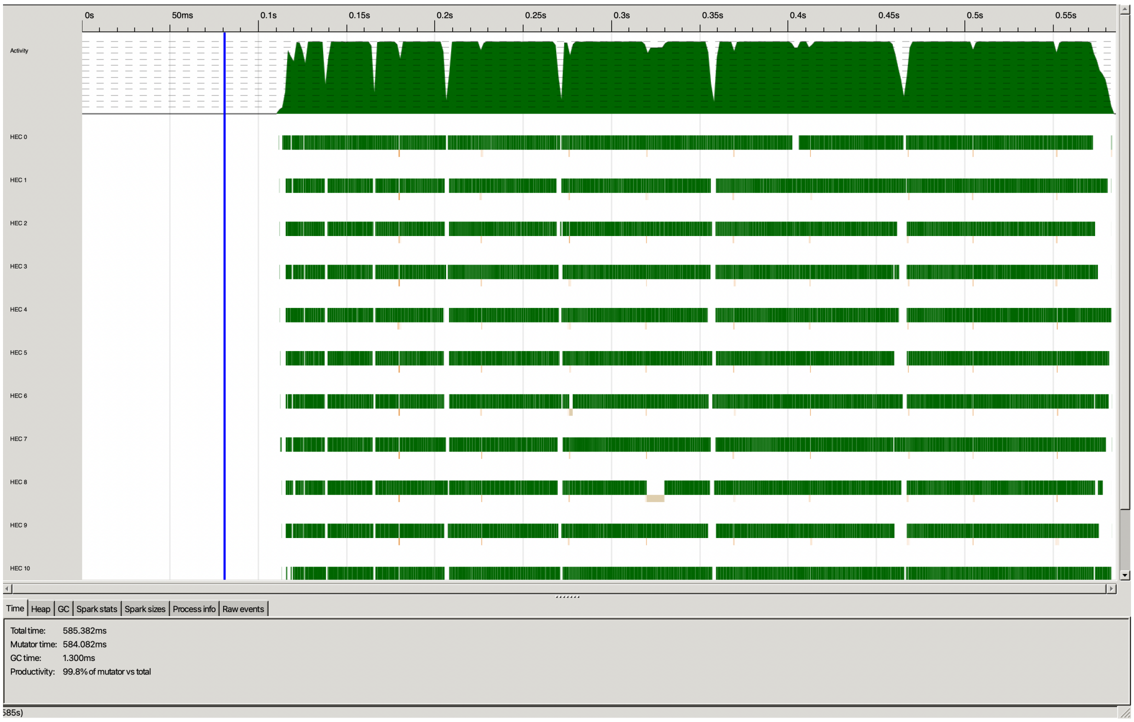
```
genAndCheckPar :: Int -> V.Vector IS.IntSet -> Int -> [Int] ->
    IS.IntSet -> IS.IntSet
genAndCheckPar depth adj k xs chosen
    | k == 0 =
       -- Base case: we have a complete subset of size we wanted
       if verifyVertexCover chosen adj
       then chosen
       else IS.empty
    | null xs = IS.empty
    | depth == 0 =
       -- Fallback to sequential if no more parallel depth allowed
       genAndCheckSeq adj k xs chosen
    | otherwise =
       case xs of
           (x:xs') ->
           let -- Try including x
               chosenWith = genAndCheckPar (depth - 1) adj (k-1) xs'
                   (IS.insert x chosen)
               -- Try excluding x
               chosenWithout = genAndCheckPar (depth - 1) adj k xs'
                   chosen
           in
           -- Evaluate chosenWith in parallel, then force without
           -- Force evaluation of chosenWith with null check
           -- We only want one solution, we discard chosenWithout if
           -- chosenWith not null
           chosenWith `par` (chosenWithout `pseq`
               (if not (IS.null chosenWith)
               then chosenWith
               else chosenWithout))
        [] -> IS.empty -- Surpress warning
```

genAndCheckPar is really just a slightly modified version of *genSubsetsParallel* from *Parallel V1*. As stated previously, we are now storing subsets with *IntSets*. Our generation function now takes one more parameter *chosen* which accumulates the current subset being generated. When we drain *k*, this means *chosen* now contains a subset of size k that we can check and return if an mvc. We only want one mvc, so if the *include* branch of our recursion yeilds a result, the *exclude* branch result is ignored. The same depth mechanism as discussed in *Parallel V1* is used.

```
solve :: V.Vector IS.IntSet -> Int -> [Int]
solve adj depth = solve' 1
    where
       n = V.length adj - 1
       nodes = [1..n]
       solve' size =
           let sol = genAndCheckPar depth adj size nodes IS.empty
           in if IS.null sol
              then solve' (size + 1)
              else IS.toList sol
```

Merging the subset generation and checking process allows for a much more simplistic *solve* function than in the other parallel versions. We simply iterate through subset sizes until *genAndCheckPar* returns something. There is no more scanning through the results of many parallel calls that check chunks of subsets for answers. This makes this approach very well suited for parallelism. We allow the user to specify *depth* as well to see what level of parallel granularity works best on their system.

The source code can be found in `/src/ParallelV3.hs`. We now solve the MVC problem for the same graph as before in 585ms on a 12 core machine with *depth* 8 as seen in 6. Productivity is at 99.8%. We see a gap before execution starts as a result of the -A128M option. This, we presume, is because a larger heap has to be allocated. These results can be replicated by running `$ stack test :pv3 --ta "+RTS -l -N8 -A128M -RTS" --ta "8"`.



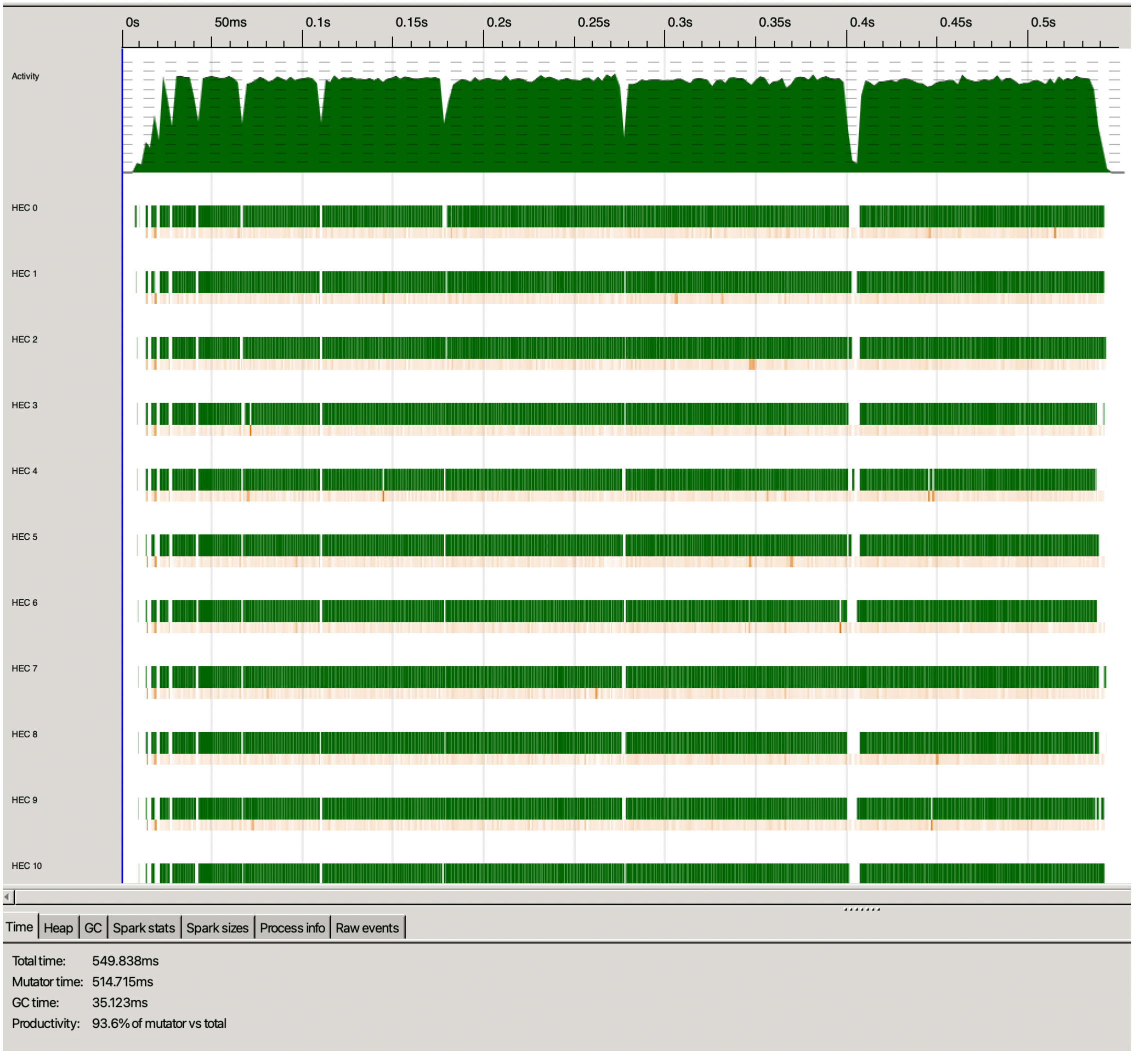
```
Total time:     585.382ms
Mutator time:   584.082ms
GC time:        1.300ms
Productivity:   99.8% of mutator vs total
```

Figure 6: Parallel v3

The -A128M option still benefits use here, even though there is not much state to be stored at any given time. This is because a thread, when continuously passing new values to the recursive function, will still fill a default nursery more often then we would like. This can be confirmed from 7, where we leave out this RTS option. We achieve slightly less productivity here, but we do save about 40ms.

We figured out that our algorithm worked fastest on the 12 core machine it was being run on when a depth of 8 was used. 8 is the speedup graph that allowed us to come to this conclusion. We benefit from having many more parallel tasks than cores because the tasks are uneven. A subset of size k can be generated very deep in the recursion process if the *exclude* decision keeps being made. We go from 2.884s at 0 depth to 0.523s at depth 8. We make use of all cores with the -N12 option here.

At depth 8, we achieve excellent load balancing with some spark fizzling / garbage collection. Here are the spark statistics. The stats in 9 go with the graph in 7 .



```
Total time:     549.838ms
Mutator time:   514.715ms
GC time:        35.123ms
Productivity:   93.6% of mutator vs total
```

Figure 7: Parallel Version 3 Default Nursery

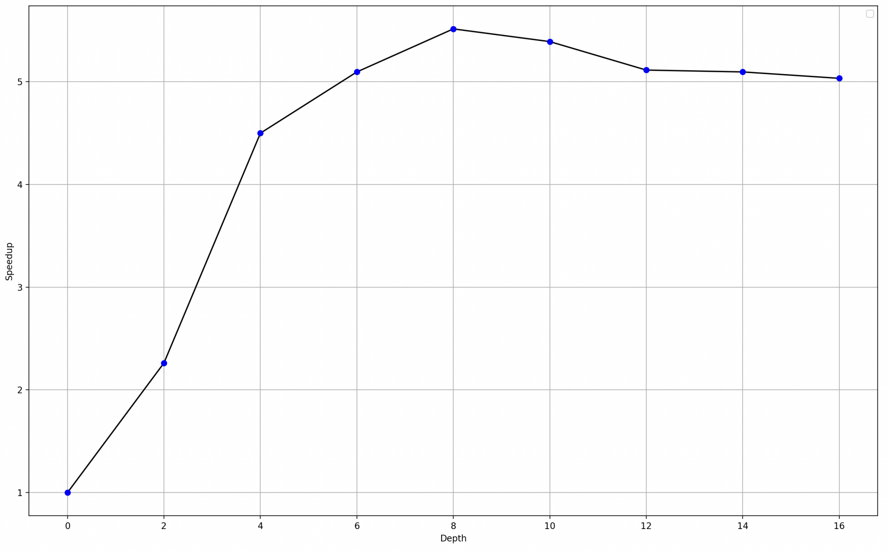
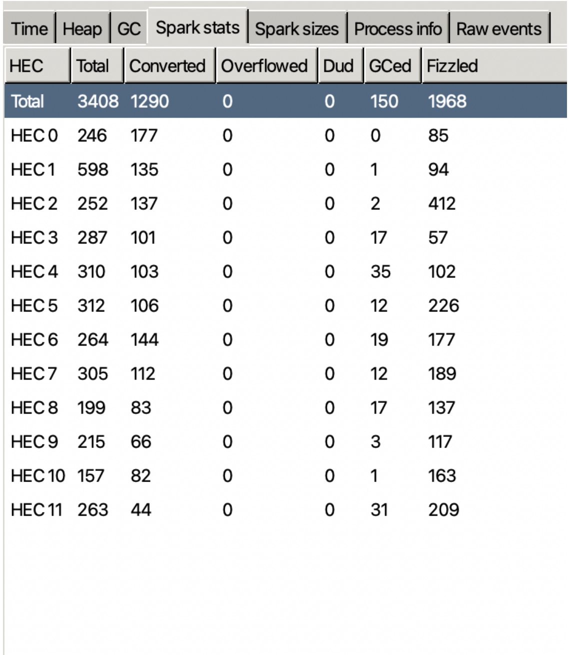


Figure 8: Parallel v3 Depth Speedup



HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	3408	1290	0	0	150	1968
HEC 0	246	177	0	0	0	85
HEC 1	598	135	0	0	1	94
HEC 2	252	137	0	0	2	412
HEC 3	287	101	0	0	17	57
HEC 4	310	103	0	0	35	102
HEC 5	312	106	0	0	12	226
HEC 6	264	144	0	0	19	177
HEC 7	305	112	0	0	12	189
HEC 8	199	83	0	0	17	137
HEC 9	215	66	0	0	3	117
HEC 10	157	82	0	0	1	163
HEC 11	263	44	0	0	31	209

Figure 9: Parallel V3 Spark Stats

In 10 we can see how our execution time speeds up as we increase the number of threads with the -N RTS option. We keep *depth* fixed at 8. We achieve a bit over a 6x speedup on 12 cores. We go from 3.369s to 0.529s. Note that *depth* 0 with 12 cores (2.884s), which corresponds to no spark creation, performs better than depth 8 with 1 core. In both cases we are only making use of one thread of execution, but when *depth* 8 is with 1 core, the overhead of spark creation is not offset by parallel execution.

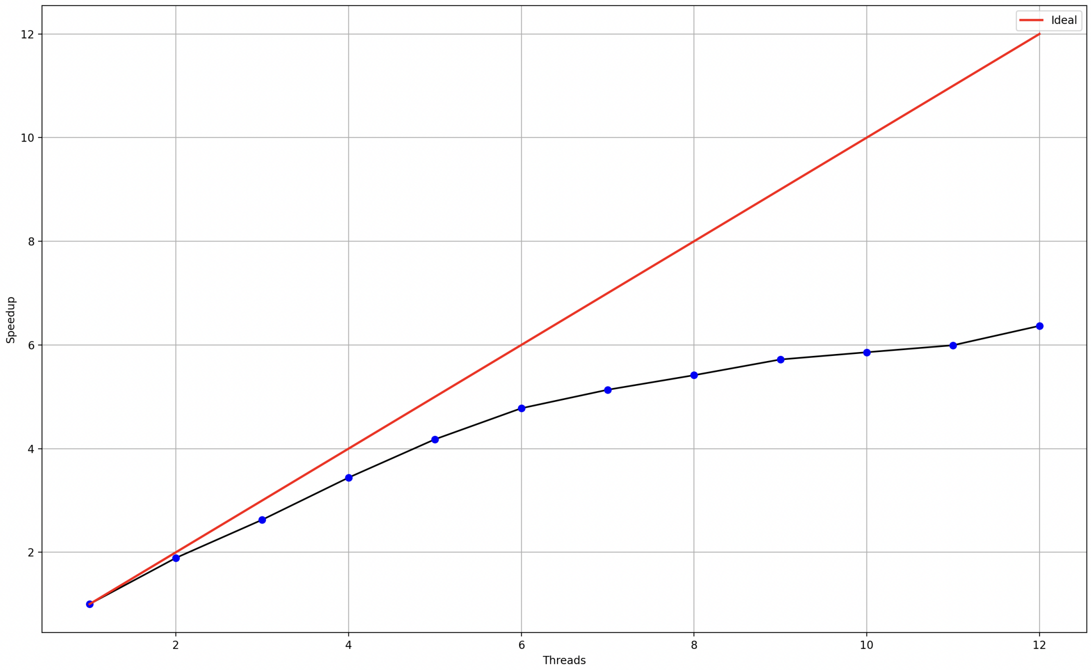


Figure 10: Parallel V3 Thread Speedup

In 11 we plot P (the parallel fraction of our algorithm as a function of N (the amount of threads) and S (the speedup achieved with N threads) using this rearranged form of Amdahl's Law.

$$P = \frac{\left(1 - \frac{1}{S}\right)}{\left(1 - \frac{1}{N}\right)} \tag{1}$$

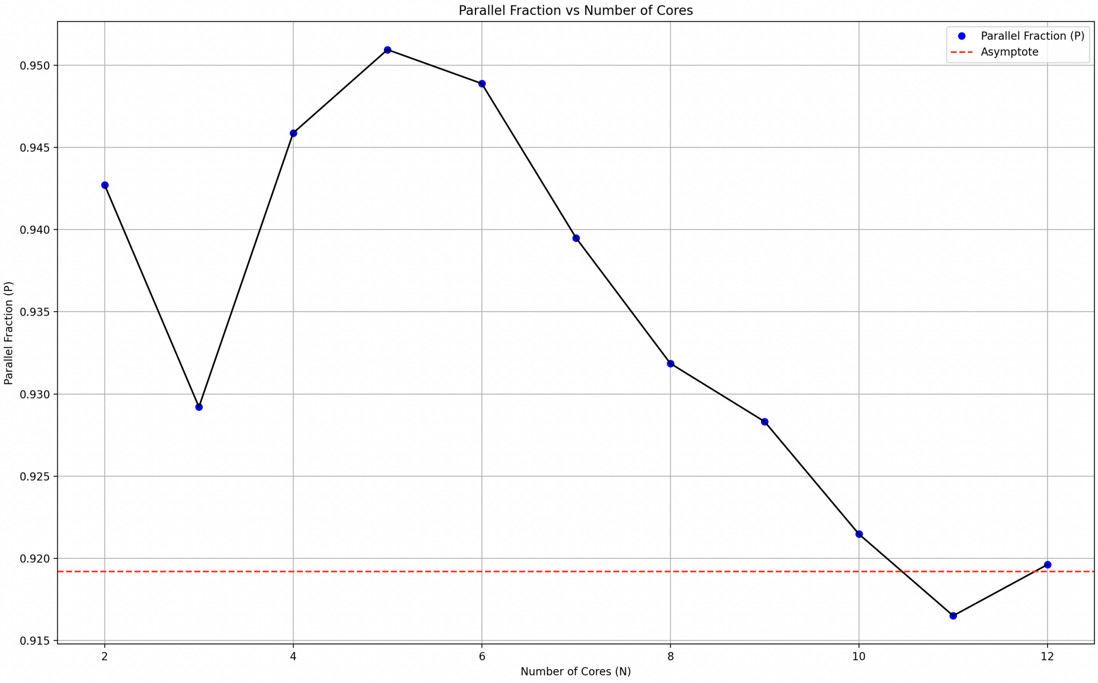


Figure 11: Parallel V3 Amdahl

We create a crude estimation of the horizontal asymptote by averaging the three highest thread counts. From this we can see that around 92% of this algorithm is parallel.

6.2 Parallel Version 4 (Bonus Solution) — Combinatorial Subset Generation

The first deque-based sequential solution we tried has a massive GC problem. That being said, it's quite easy to parallelize this by processing portions of the deque in parallel to generate the next deque. For example, let us say we have all subsets of size 3 on the deque. We can have a task that takes a look at some of these subsets and extends each one of these to form subsets of size 4. This task will also involve checking to see if any newly generated subsets solve the MVC. Once all tasks finish processing their respective portions of the deque, the new subsets each task generated can be glued together to form the next deque to process. We only process the next deque if no solutions were found. This is a good idea, but it still has the issue of needing to have all subsets of a ceratin size in memory at the same time. We also know this will create a garbage collection problem.

What if we can traverse segments of subsets of a certain size as discussed above without storing all subsets of a certain size? Furthermore, what if we only generate subsets of a certain size through mutating subsets of the same size?

We start by defining some helper functions in `src/SubsetTools.hs`.

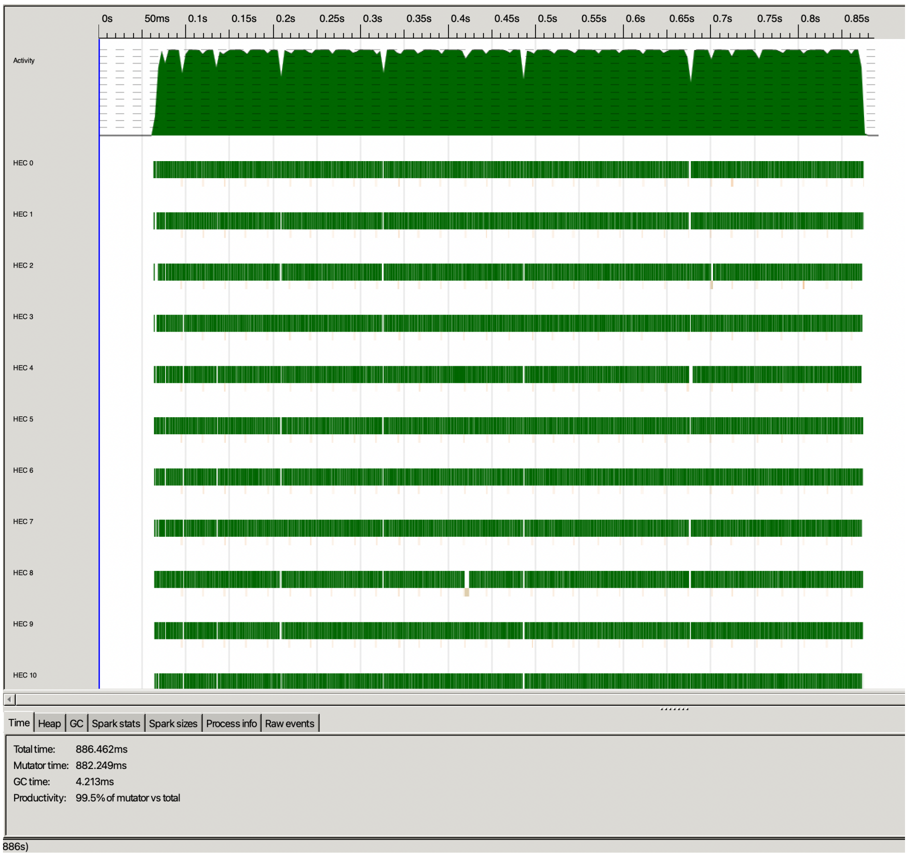
nthSubsetIO constructs the 0-indexed num-th subset of size k from 1..n directly into a mutable vector in ascending order. This code finds the lexicographically num-th k-subset from an n-element set (1 through n). It uses binomial coefficients (choose) to skip over entire ranges of combinations without enumerating all of them. Starting from the smallest element (1) it checks how many combinations start without including that element. If skipping that element accounts for fewer than num remaining combinations, it subtracts that count and moves to the next element without choosing it. Otherwise, the target combo must include that element and so it includes it and proceeds to select the rest. By iteratively applying this logic, it directly constructs the num-th combination. [6]

The helper function *next*, takes a subset of size k, and produces the next subset of size k by finding the first element that can be increased by 1 from the end of the subset without colliding with another value in the subset and increasing it. When the increment operation occurs, all elements after the increased element are replaced with the numbers directly after the increased element in ascending order. These helpers work with mutable vectors, and thus they are in the IO monad.

The *solve* function in parallel v4 calculates n choose k for increasing k and splits the resulting number into intervals based on the parameter *chunkSize*. The larger *chunkSize* is, the less parallelism we have. Each interval is fed to a function called *processRange* by parMapM. This creates a parallel task for each range. k and the starting value are passed into *processRange*. With this data, *nthSubsetIO* is able to calculate the num-th k-element subset. Once *processRange* has its starting subset, it can check if there exists an MVC and call the *next* helper to generate and check again until it checks its entire range. If it finds an answer, it returns it to parMapM's results.

This method allows us to process segments of subsets of a certain size without storing all subsets of a certain size. Furthermore, unlike in PV3, we can now generate subsets of a certain size without having to build them from subsets of smaller sizes for the most part. This in theory should make this a much more efficient solution. However, since this is an NP complete problem, we never have to deal with substantially large subsets. It wouldn't make sense to test this problem with n larger than 30. In Parallel V3, we used a few recursive calls to generate the subsets. The overhead of this segmentation/combinatorics technique actually leads to it performing worse than PV3. Most of time, generating a subset in PV4 involves mucking around with an already existing mutable vector. However, subsets at the start of ranges require a substantial amount factorial operations to be computed. As a result, getting the subset for the start of a range turns out to be an N^2 operation given the repeated factorials. Furthermore, the result of each range evaluation has to be scanned when parMapM returns. We can see in 12 that this solution runs about 300ms slower than PV3. With the ideal chunkSize (10000) this ran at 886ms. That being said, it is still ferociously parallel and experiences essentially the same speedup as PV3 as seen in 14.

The source code can be found in `/src/ParallelV4.hs`. These results can be replicated by running `$ stack test :pv4 --ta "+RTS -l -N8 -A128M -RTS" --ta "10000"`.



Total time: 886.462ms
Mutator time: 882.249ms
GC time: 4.213ms
Productivity: 99.5% of mutator vs total

886s)

Figure 12: Parallel v4

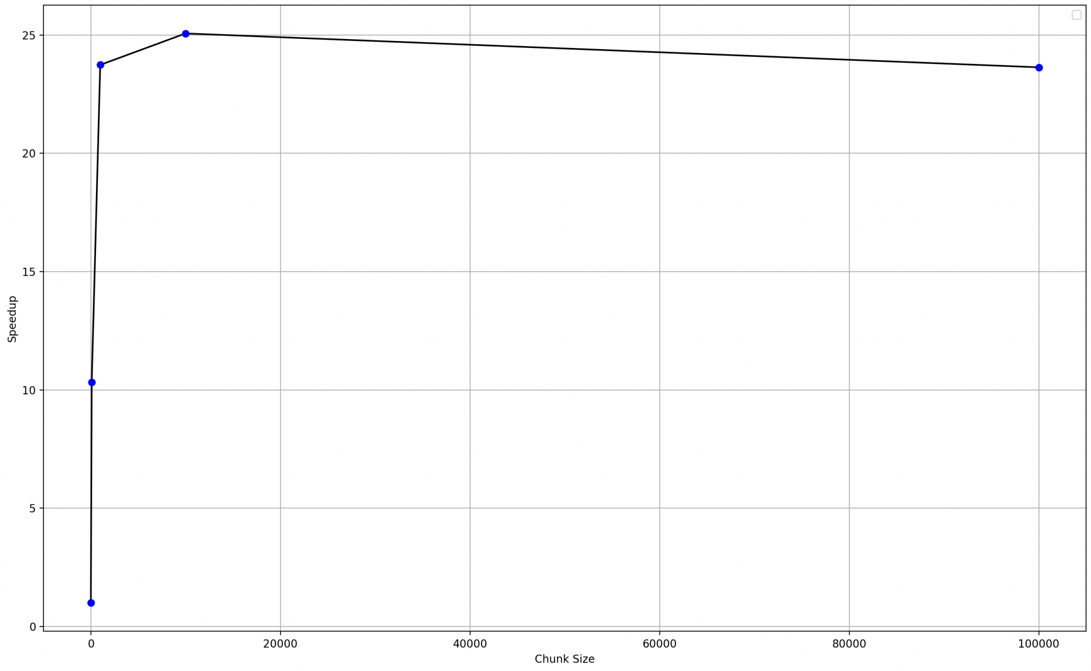


Figure 13: PV4 Chunk Size Speedup

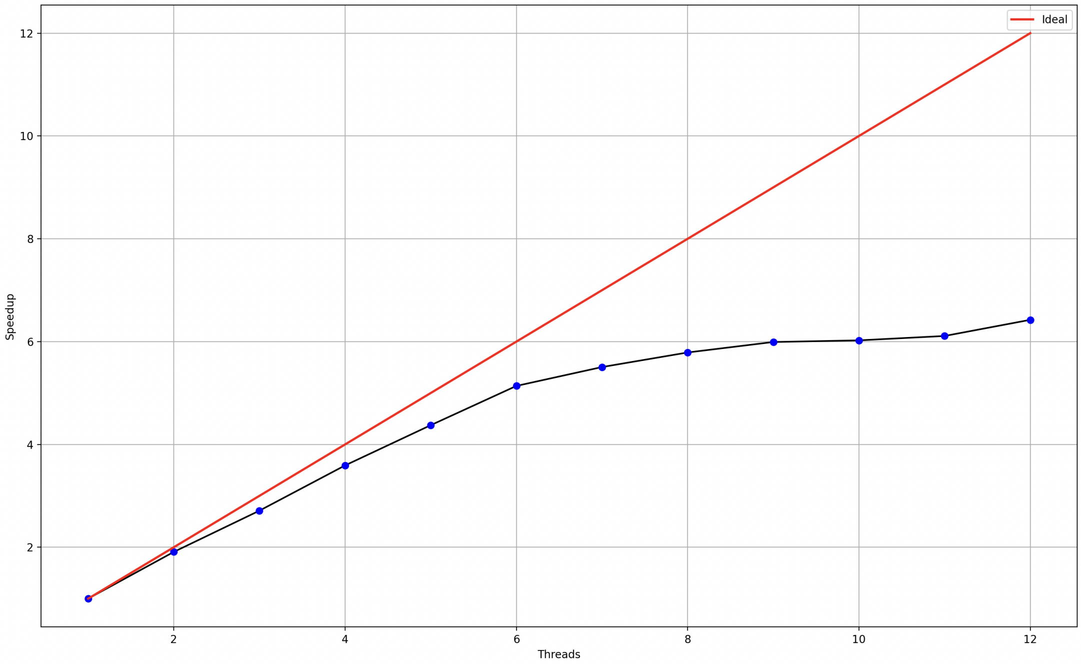


Figure 14: PV4 Thread Speedup

One final note about $ParMapM$:

This monad doesn't create sparks. The Par monad uses a pool of worker threads produced by GHC's runtime system (tasks). Jobs are fed into a work-stealing scheduler managed by the monad-par library. This scheduler distributes the jobs across the available threads. [7] Sparks are not used by the Par monad. Everytime you throw another core into the mix, a thread for execution and a GC thread are created. [8]

-N1 4 tasks
-N2 6 tasks
-N3 8 tasks
-N4 10 tasks
...
-N12 26 tasks

7 References

[1] https://github.com/sedgwickc/VertexCoverSearch
[2] https://hackage.haskell.org/package/vector
[3] https://hackage.haskell.org/package/containers-0.7/docs/Data-IntSet.html
[4] https://ghc.gitlab.haskell.org/ghc/doc/users_guide/runtime_control.html
[5] https://wiki.haskell.org/Foldr_Foldl_Foldl'
[6] https://en.wikipedia.org/wiki/Combinatorial_number_system
[7] https://hackage.haskell.org/package/monad-par
[8] https://simonmar.github.io/publications/multicore-ghc.pdf
[9] https://hackage.haskell.org/package/base-4.21.0.0/docs/System-IO-Unsafe.html
[10] https://hackage.haskell.org/package/base-4.20.0.0/candidate/docs/Data-Char.html
[11] https://hackage.haskell.org/package/base-4.21.0.0/docs/Control-Monad-ST.html
[12] https://hackage.haskell.org/package/base-4.20.0.0/candidate/docs/Data-IORef.html
[13] https://hackage.haskell.org/package/random-1.2.1.3/docs/System-Random.html
[14] https://hackage.haskell.org/package/base-4.21.0.0/docs/System-IO.html
[15] https://hoogle.haskell.org/?hoogle=readMaybe
[16] https://www.microsoft.com/en-us/research/wp-content/uploads/2009/09/ghc-parallel-tuning2.pdf
[17] https://wiki.haskell.org/GHC/Memory_Management