COMS4995 Parallel Functional Programming Project
Maximal Clique Problem with Bron–Kerbosch Algorithm

Xuezhen Wang, Songheng Yin

Uni: ww2604, sy3079

1 Overview

The objective of this project is to develop an implementation of the Bron-Kerbosch algorithm in Haskell, specifically designed to identify all maximal cliques in any given undirected graph. This implementation will comprise both a sequential version and a variety of parallelized variants. The key goal is to conduct a thorough performance analysis of these versions, with a focus on evaluating the impact and efficacy of parallelism in the context of this algorithm.

2 Problem Formulation

2.1 Definitions

In an undirected graph $G = (V, E)$. A **clique** C is a subset of the vertices, $C \subseteq V$, such that every two distinct vertices are adjacent, that is, the induced subgraph by C is a complete graph $K_{|C|}$.

A **maximal clique** is a clique that cannot be extended by including one more adjacent vertex, that is, a clique which does not exist exclusively within the vertex set of a larger clique. A **maximum clique** of a graph is a clique such that there is no clique with more vertices. Moreover, the clique number $\omega(G)$ of a graph G is the number of vertices in a maximum clique in G. Note the difference between maximal clique and maximum clique: a maximum clique is always maximal, the converse is not always true.

2.2 Problem Description

The clique decision problem asking for if a clique of size k exists in the given graph. It was one of Richard Karp's original 21 problems shown NP-complete in Cook [1971] and Karp [1972]. Our task, a variant of the clique decision problem, is to list **all** the **maximal** cliques given an undirected graph G. It is easy to know the problem is impossible to be done in polynomial running time since it can derive the answer to the clique decision problem trivially. Therefore, it is perfectly suitable to assess the effectiveness of parallelism as it is not an IO bounded problem.

2.3 Algorithm

The Bron–Kerbosch algorithm, designed by Bron and Kerbosch [1973], is an enumeration algorithm for finding all maximal cliques in an undirected graph.

Algorithm 1 Bron–Kerbosch Algorithm

```
1:  function BronKerbosch(R, P, X)
2:      if P and X are both empty then
3:          Report R as a maximal clique                ▷ A maximal clique is found
4:      end if
5:      for each vertex v in P do
6:          BronKerbosch(R ∪ {v}, P ∩ N(v), X ∩ N(v))   ▷ Explore extensions of R including v
7:          P ← P \ {v}                                 ▷ Remove v from potential clique extensions
8:          X ← X ∪ {v}                                 ▷ Add v to excluded set for this recursion level
9:      end for
10: end function
```

The Bron-Kerbosch algorithm uses three sets R, P, and X to find maximal cliques in an undirected graph:

1. R (Reported Clique): This set starts empty and grows as the algorithm progresses. It represents the current clique being constructed. When both P and X are empty, R is a maximal clique and is reported as such.
2. P (Potential Nodes): This set contains vertices that are connected to all vertices in R and might be included in the clique. These are potential candidates to be added to R. The algorithm iteratively moves vertices from P to R to explore the expansion of the current clique.
3. X (Excluded Nodes): This set also starts empty and contains vertices that have been considered for inclusion in R and found not to lead to a maximal clique (in the current path of the search). It helps to avoid re-examining the same vertex within the same recursive call.

3 Haskell Implementation

3.1 Deliverables

Basically, our compiled program has the following functionality:

- Usage: gen <mode> <args>..<args> <outfile>
- Usage: compute <readpath> <writepath> <mode> (<mode arg>)
- Usage: test <inputfile> <outputfile>

where:

- In **generation mode (gen)**, the program aims to generate a graph into an output file. The generated file for the graph contains two integers m and n at the first line, which represents the number of vertices and the number of edges. In the following m lines, it contains a_i and b_i where $0 \leq i < m$ for each line representing the vertices connected by i^{th} edge. There are two modes for graph generation, i.e., random and kclique, which take different arguments.
- In **computation mode (compute)**, the program aims to take an input file path for graph in the format as specified above, an output file path for maximal cliques and a specified mode to compute all the maximal cliques. For the output file format, each line contains a set of numbers indicating the vertex indices for the vertices in the clique. Here we provide 5 modes, namely, **sequential mode (seq)**, **naive parallel mode (par_naive)**, **basic parallel mode (par_basic)**, **chunked parallel mode (par_chunked)** and **depth-limited parallel mode (par_depthLimited)**.
- In **testing mode (test)**, the program takes a file path for input graph and the file path for the corresponding output cliques to output Valid or Invalid.

For each of these functionalities, we will introduce them in detail in the later sections.

3.2 Data Structures

To enhance the conciseness and clarity of our code, we focus on the meticulous development of three core components: the **Vertex type**, the **Clique type**, and the **Graph class**. This approach ensures that our code is not only streamlined but also easily interpretable. The corresponding code is defined in **DataStructures.hs**

```
-- Basic Data Structures
type Vertex = Int
type Clique = [Vertex]
class Graph graph where
    num_vertices :: graph -> Int
    num_edges :: graph -> Int
    vertices :: graph -> [ Vertex ]
    is_connected :: graph -> Vertex -> Vertex -> Bool
    neighbours :: graph -> Vertex -> [ Vertex ]
```

3.3 Graph Generation

The project involves the implementation of graph generation functionalities. Two distinct algorithms were developed to create graphs with varying properties. This section outlines the methodologies and justifications for these algorithms.

1. Algorithm 1: Straightforward Random Generation
    - **Methods:** The first algorithm employs a straightforward random generation technique. It begins by iterating over each pair of vertices. For each pair, an edge is added with a 50% probability, effectively creating a random graph.
    - **Observation:** However, an issue was observed with this method. Due to the randomness of edge creation, the size of the largest clique (a subset of vertices where every two distinct vertices are adjacent) within the generated graphs tends to be relatively small. This limits the application of the algorithm in scenarios where larger cliques are necessary for analysis or simulation.
2. Algorithm 2: Clique-Ensured Graph Generation
    - To overcome the limitations of the first algorithm, a second method was introduced with a specific focus on the generation of a large clique within the graph. This algorithm guarantees the presence of a clique of at least size k, which is a desirable property for certain types of graph analysis. The random connections among the remaining vertices preserve the stochastic nature of the graph while ensuring the existence of a substantial clique.

- **Methods:** The steps for this algorithm are as follows:
  (a) Initially, a clique of size k is created by making the first k vertices fully connected.
  (b) For the remaining n-k vertices, the previous random connection method is applied to add edges.
  (c) Finally, the indices of the vertices are remapped to ensure uniformity and to avoid predictable patterns in the graph structure.

Both algorithms serve distinct purposes within the project. The straightforward random generation provides a basic approach to create random graphs, while the clique-ensured method offers a more structured graph with guaranteed properties. The corresponding code is in **GenerateGraph.hs**

3.4 Sequential Implementation

The original Bron-Kerbosch algorithm maintains a state of three sets of vertices, i.e. the R, P, X sets as defined in the algorithm. For an undirected graph $G = (V, E)$. The initial state will be $(\emptyset, V, \emptyset)$, and the Bron-Kerbosch algorithm stops when $P = \emptyset \wedge X = \emptyset$. The core segment in sequential version code is listed below. The function `bronKerbosch` takes 3 parameters: R, P, X in order. And the function `restrictVertices` is used to find the neighborhood, which is used in `exploreCandidates` to modify the Bron-Kerbosch state.

The complete code can be found in **SeqBK.hs**

```
getMaximalCliques :: Graph graph => graph -> [Clique]
getMaximalCliques graph = bronKerbosch [] ( vertices  graph)  []  where
  bronKerbosch :: Clique -> [Vertex] -> [Vertex] -> [Clique]
  bronKerbosch  partialClique  candidateVertices  excludedVertices
      | null  candidateVertices  && null excludedVertices = [ partialClique ]
      | otherwise = exploreCandidates candidateVertices excludedVertices
    where
      exploreCandidates  ::  [Vertex]  -> [Vertex] -> [Clique]
      exploreCandidates  []  _ = []
      exploreCandidates  (currentVertex  :  remainingCandidates) currentExcluded =
          bronKerbosch (currentVertex  :  partialClique )
                       (remainingCandidates ` restrictVertices ` currentVertex)
                       (currentExcluded ` restrictVertices ` currentVertex) ++
          exploreCandidates remainingCandidates (currentVertex  :  currentExcluded)

       restrictVertices   ::  [Vertex]  -> Vertex -> [Vertex]
       restrictVertices   curvertices  vertex  =  filter  (is_connected  graph  vertex)  curvertices
```

3.5 Naive Parallel Implementation

Upon closer examination of the algorithm, the most direct approach to parallelizing it involves introducing a "spark" — a lightweight thread—for each recursive call initiated. Essentially, within each iteration of the for loop, we initiate a spark for the corresponding recursion before proceeding with the next iteration. To implement this parallelism, it is remarkably simple: one need only alter two lines within the code originally written for sequential execution, i.e. the `exploreCandidates` function. This minor modification has the potential to significantly enhance the performance by leveraging concurrent computation. Due to its simplicity, we call it naive parallel implementation. The modified section is attached below and the full implementation can be found in **ParBK_Naive.hs**.

```
exploreCandidates  ::  [Vertex]  -> [Vertex] -> [Clique]
      exploreCandidates  []  _ = []
      exploreCandidates  (currentVertex  :  remainingCandidates) currentExcluded =
          let newCliques = bronKerbosch (currentVertex  :  partialClique )
                                        (remainingCandidates ` restrictVertices ` currentVertex)
                                        (currentExcluded ` restrictVertices ` currentVertex)
          in newCliques `par` (exploreCandidates remainingCandidates (currentVertex  :  currentExcluded)
          `pseq` (newCliques ++ exploreCandidates remainingCandidates (currentVertex  :  currentExcluded)))
```

3.6 Lazy Data Structure + Strategy Implementation

3.6.1 Inspiration

Inspired by the idea introduced in the lecture where **Parallelism = Lazy Data Structure + Strategy**,

- "Build a lazy data structure representing the computation"
- "Apply a Strategy that traverses the computation"

Our goal is to encapsulate the algorithm within a lazy data structure, a design that inherently enables deferred computation. By doing so, we can then apply a variety of strategies to this data structure without necessitating any alteration to the algorithm's core logic. This approach provides the flexibility to optimize performance and resource management dynamically, according to the demands of the execution context, all the while maintaining the algorithm's integrity.

3.6.2 Computation Tree as the Lazy Data Structure

The choice of data structure is pivotal, and our analysis reveals that the most effective representation of each recursive call is as a node within a computation tree. This structure mirrors the algorithm's process, where each node encapsulates a distinct state of computation.

To elucidate this concept, let us consider a specific example. Refer to Figure 1, where both the graph and its corresponding computation tree are depicted you [2021]. The computation tree is constructed iteratively: each time the BronKerbosch recursive function is called, we capture and store the current state—denoted by the variables R, P, X—at that node. Subsequent iterations within the function that trigger further recursive calls result in the creation of new branches in the tree. This process continues until a maximal clique is discovered, at which point the tree culminates in a node representing this complete subgraph.

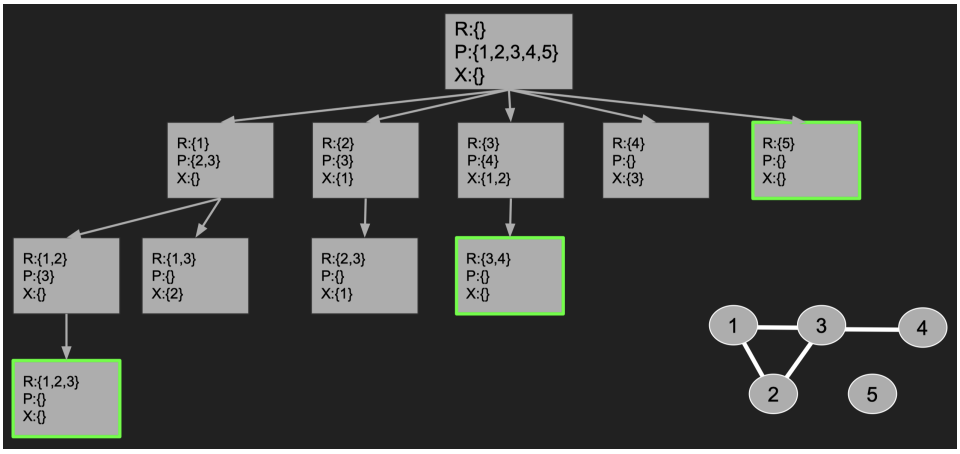


Figure 1: Illustration of Computation Tree

3.6.3 Haskell Data Structure for Computation Tree

Consequently, we have implemented the lazy data structures to represent the computation tree in Haskell, as demonstrated in the following code snippet. The full implementation can be found in **ParBK_DataStructure.hs**

```
type ComputationState = (Clique, [Vertex], [Vertex])
data ComputationTree = Branch ComputationState [ComputationTree] | Node Clique
getMaximalCliques :: Graph graph => graph -> ComputationTree
```

Notice the signature change in `getMaximalCliques` where it returns a `ComputationTree` instead of `[Clique]`. In order to extract cliques from the computation tree. We define the following function in **Util.hs**.

```
extractClique :: ComputationTree -> [Clique]
extractClique (Branch _ xs) = concat (map extractClique xs)
extractClique (Node x) = [x]
```

3.6.4 Basic Parallel Strategy

Similar to the naive parallel strategy, we introduce a "spark" for each recursive call initiated. The full code can be found in **ParBK_Strategy.hs**.

```
strategy_basic :: Strategy ComputationTree
strategy_basic (Branch state xs) = fmap (Branch state) (parList strategy_basic xs)
strategy_basic (Node clique) = fmap Node (rdeepseq clique)
```

However, these are some notable problems for this strategy. Specifically:

- The creation of sparks is occurring at such a rapid rate that it leads to an overflow of the spark pool.
- The sparks being generated are too small, rendering them ineffective for the intended parallel performance gains.

To mitigate these challenges, we propose the following two strategies.

3.6.5 Chunked List Strategy

In this strategy, the idea is to reduce the number of sparks, by chunking the lists. We specify the `chunkSize` as a parameter for this strategy. It determines how many elements of the list are grouped together in each chunk. In principle, the choice of chunk size can significantly impact performance. Too small a size might not fully utilize the benefits of parallelism, while too large a size might lead to uneven distribution of work across cores. The full code can be found in **ParBK_Strategy.hs**.

```
strategy_chunked :: Int -> Strategy ComputationTree
strategy_chunked chunkSize (Branch state xs) = fmap (Branch state) (parListChunk chunkSize (strategy_chunked chunkSize) xs)
strategy_chunked _ (Node clique) = fmap Node (rdeepseq clique)
```

3.6.6 Depth-limit Strategy

This strategy limits the parallelism to a certain depth in the tree, which applies different strategies based on the `depth` parameter. If the depth is greater than zero, it applies the `strategy_depthLimited` recursively to each child in the list `xs`, but with `depth-1`. This means it will go one level deeper into the tree. `parList` is used here, which applies the strategy in parallel to each element of the list `xs`. The parallel processing will continue until the specified depth is reached. If the depth is zero or less, it applies the strategy to the children without parallelism. `evalList` is used to sequentially evaluate each element of the list `xs` with the given strategy. This ensures that beyond the specified depth, the computation proceeds in a non-parallel manner. The full code can be found in **ParBK_Strategy.hs**.

```
strategy_depthLimited :: Int -> Strategy ComputationTree
strategy_depthLimited depth (Branch state xs)
    | depth > 0 = fmap (Branch state) (parList (strategy_depthLimited (depth - 1)) xs)
    | otherwise = fmap (Branch state) (evalList (strategy_depthLimited depth) xs)
strategy_depthLimited _ (Node clique) = fmap Node (rdeepseq clique)
```

3.6.7 Something is Missing? Parallel Extraction!

Everything appears to be in place for the experiment to commence. We have already conducted numerous trials at this stage, as detailed in Section 4.4. Unfortunately, the outcomes have not met our expectations. Upon thorough analysis, to be discussed later, we identified that the bottleneck hindering parallelism is the sequential extraction algorithm `extractClique`. This algorithm's need to traverse an exponentially large computation tree is the primary issue. To address this, we have developed modified parallel extraction methods specifically for each parallel strategies introduced above, i.e., `extractCliquePar`, `extractCliquePar_Depth`, `extractCliquePar_Chunk`, which is presented in **Util.hs**.

```
extractCliquePar :: ComputationTree -> [Clique]
extractCliquePar (Branch _ xs) = concat (parMap rdeepseq extractCliquePar xs)
extractCliquePar (Node x) = [x]

extractCliquePar_Depth :: Int -> ComputationTree -> [Clique]
extractCliquePar_Depth depth (Branch _ xs)
    | depth > 0 = concat (parMap rdeepseq (extractCliquePar_Depth (depth - 1)) xs)
    | otherwise = concatMap (extractCliquePar_Depth depth) xs
extractCliquePar_Depth _ (Node x) = [x]

extractCliquePar_Chunk :: Int -> ComputationTree -> [Clique]
extractCliquePar_Chunk numChunks (Branch _ xs) =
    concat $ parMap rdeepseq (concatMap (extractCliquePar_Chunk numChunks)) (chunkTrees xs numChunks)
extractCliquePar_Chunk _ (Node x) = [x]

chunkTrees :: [ComputationTree] -> Int -> [[ComputationTree]]
chunkTrees trees numChunks = chunksOf chunkSize trees
  where chunkSize = max 1 (length trees `div` numChunks) -- Calculate chunk size, avoid division by zero
```

3.7 Testing

In order to test the correctness of the algorithm, we provide **Test.hs**, in which it defines the following function:

```
-- Main function to validate algorithm output
validateAlgorithm :: String -> String -> IO ()
```

The function takes two file paths as inputs: one for the input graph and another for the output cliques. It then performs an evaluation based on these inputs, yielding an output of either "Valid" or "Invalid". The process involves two primary checks: firstly, it verifies whether all the cliques listed in the output file form fully connected subgraphs as per the input graph. If this condition is met, it proceeds to assess whether these cliques are maximal. A clique is

considered maximal if it is not a subset of a larger fully connected subgraph in the input graph. If both conditions are satisfied, the function outputs "Valid"; otherwise, it declares the result as "Invalid".

4 Experiment Result

4.1 Test Graphs

We randomly generated 2 graphs as input:

1. **Graph A**: A graph of 200 vertices. Note that the probability of having at least one k-clique for a random graph is close to 1 for small k and close to 0 for large k. Moreover, the dividing point for "small" and "large" is

$$\frac{2\log n}{\log 1/p}$$

where p is the edge probability, n is the number of vertices. So almost all of the cliques in this graph will have sizes smaller than 15. Therefore, the computation tree constructed is in some sense "balanced" meaning that it is shallow but the branches distributed evenly (See Figure 2).

2. **Graph B**: A graph of 50 vertices, whose subgraph set includes K_{25}, i.e. at least a clique of 25 vertices. Therefore, the computation tree constructed is in some sense "imbalanced" meaning that it has a rather deep branch for the big clique (See Figure 2).

4.1.1 Illustration of computation tree Structure for Two Graphs

(a) Graph A (b) Graph B

Figure 2: computation tree Structure for Two Graphs

4.2 Sequential Baseline

It costs the sequential algorithm 21.066 seconds to compute all maximal cliques on graph A, and 21.16 seconds on graph B. This will be our baseline for all the following experiments.

4.3 Naive Parallelism

The detailed performance is listed in the table:

Cores	Sparks				Time (sec)
	Total	Converted	GC'ed	Fizzled	
1	-	-	-	-	21.066
2	8152240	11585	1432905	5798057	14.59
4	8093761	15275	1739294	5385841	8.023
8	8321491	42042	1936186	5672354	5.712

Table 1: Eventlog on graph A

Cores	Sparks				Time (sec)
	Total	Converted	GC'ed	Fizzled	
1	-	-	-	-	21.16
2	34215634	167	413291	460132	8.450
4	34897999	1222	1566001	3127131	5.829
8	37677284	8452	11122405	9161273	5.649

Table 2: Eventlog on graph B

We have observed a marked improvement in the running time compared to the sequential baseline algorithm. Additionally, as shown in Table 1 and Table 2, there is a significant rise in the conversion ratio of sparks, with more

sparks being successfully converted out of the total generated. This improvement is likely attributed to the increased number of cores allocated to the task, enhancing the conversion efficiency of sparks in the pool. Moreover, as shown in Figure 3 for -N8, all assigned cores are utilized without too much idleness.

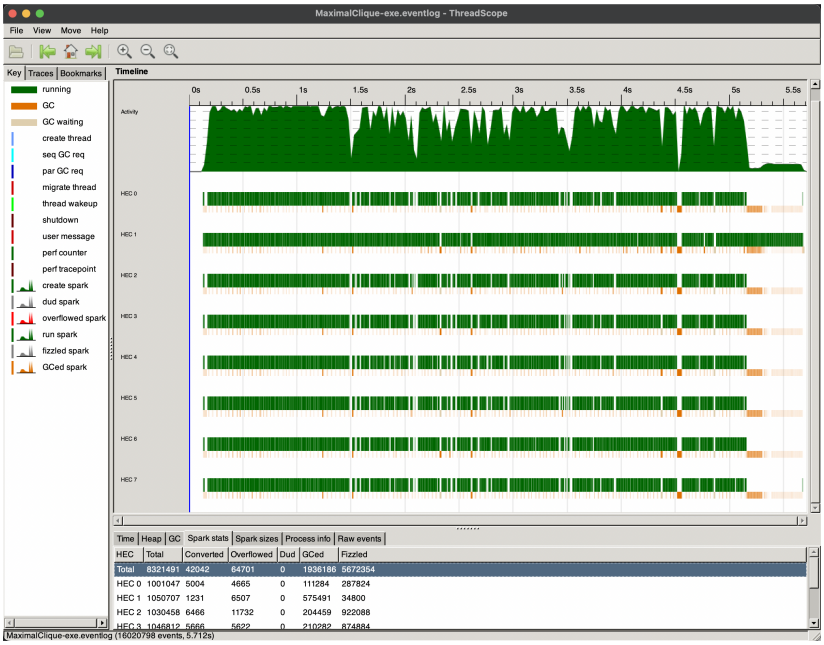


Figure 3: `-N8` Naive Eventlog

4.4 First Stage (Sequential Extraction)

4.4.1 Basic Parallel Strategy

The execution times for the basic parallel strategy on Graph A are 26.552, 23.692, and 23.530 seconds when using the parameters `-N2, -N4, -N8`, respectively. Interestingly, these times are similar to those of the sequential algorithm, a finding that diverges from expectations. Typically, one would anticipate performance akin to the naive parallel algorithm, where similar parameters generally yield comparable results. However, a closer examination of Figure 4 reveals that after an initial phase, the total computational activity approximates that of a single-core utilization. This observation aligns with our previous discussion: the sequential `extractClique` method emerges as a limiting factor in the algorithm's efficiency. Consequently, the addition of more cores does not significantly enhance performance, due to this bottleneck.

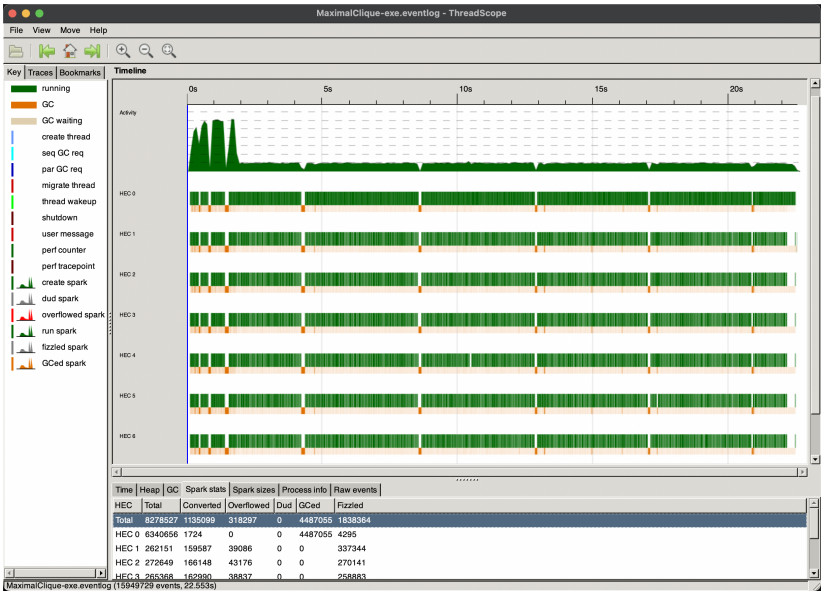


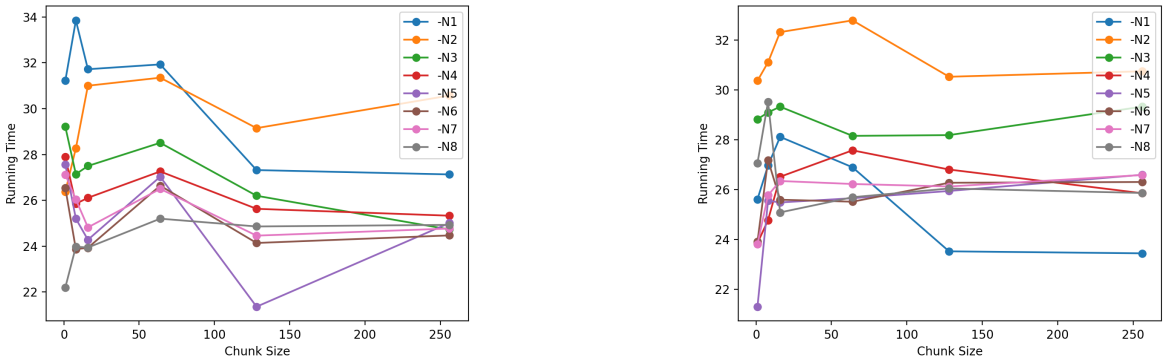
Figure 4: `-N8` Basic Parallel Eventlog

4.4.2 Chunked List Strategy

Figure 5 demonstrates the correlation between the running time and the chunk size of sparks. Despite the process still being constrained by the sequential extraction phase which makes it noisy, several interesting observations emerge by analyzing the plot.

1. The impact of varying chunk sizes and the number of cores on running time depends on the specific structure of the graph. For instance, in the case of Graph A, an optimal running time is achieved using a `chunkSize` of `128` combined with `-N5`. Conversely, for Graph B, the best performance is observed with a `chunkSize` of `1` and `-N5`.

2. As the number of cores increases, the performance curves for different chunk sizes tend to converge. This phenomenon can be attributed to the increased parallelization, which in turn makes the sequential extraction phase more prominent in determining the overall running time.



(a) Graph A (b) Graph B

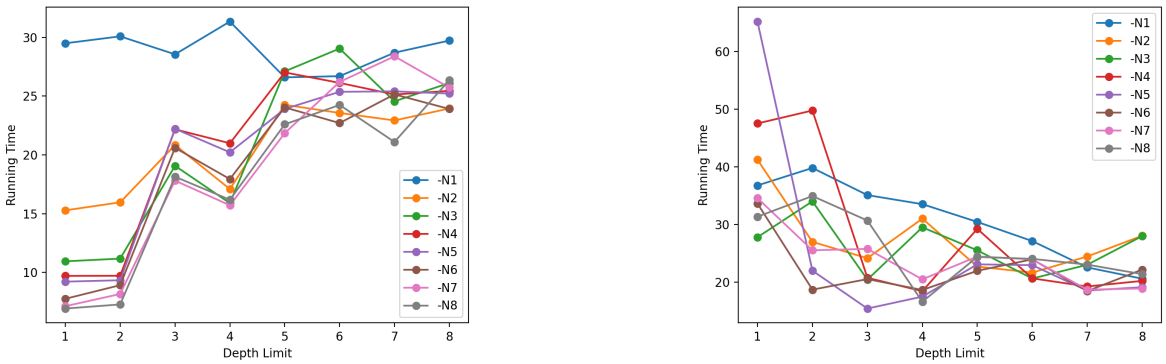
Figure 5: Chunked List Algorithm Eventlog

4.4.3 Depth-limit Strategy

Notice that by limiting the depth of the tree, we get a significant reduction in the running time. Figure 6 illustrates the relationship between running time and the parallelism depth limit. Still, despite the process still being constrained by the sequential extraction phase which makes it noisy, several interesting observations emerge by analyzing the plot.

1. Similar to the chunk size strategy, the optimal strategies and core configurations are influenced by the structure of the graph. For Graph A, peak performance is attained with a `Depth Limit` of 1 and the `-N8` setting. In contrast, Graph B achieves its best performance with a `Depth Limit` of 3 and `-N5`.

2. An interesting observation for Graph A is that a smaller `Depth Limit` proves to be more effective, whereas for Graph B, a larger `Depth Limit` is preferable. This distinction can be understood by referring to the computation tree illustrations in Section 4.1.1. Graph A's computation tree is balanced, suggesting that an even distribution of computation across branches is optimal, thus necessitating a smaller `Depth Limit` to avoid generating excessive sparks for each branch. Conversely, Graph B's tree is unbalanced with more computation concentrated in specific branches, requiring a larger `Depth Limit` to generate sufficient sparks and fully leverage parallelism in these heavier branches.



(a) Graph A (b) Graph B

Figure 6: Depth-limit Algorithm Eventlog

4.5 Second Stage (Parallel Extraction)

The plots now exhibit much more clear plots compared to those in the first stage, as highlighted in the previous section. Furthermore, there's a notable reduction in running time when contrasted with the first stage. Let's proceed to meticulously analyze the plots about the 'chunked list strategy' and the 'depth-limit strategy' in greater detail.

4.5.1 Basic Parallel Strategy

We conduct a comparative analysis between Figure 7 from the second stage and Figure 4 from the first stage. It is evident that the "straight line" representation, indicative of the use of only one "equivalent" core in the latter part of the process, particularly during the extraction phase, is significantly reduced. This observation substantiates our prior hypothesis that it is the sequential extraction that causes this "straight line" and demonstrates the efficacy of parallelizing the extraction method.

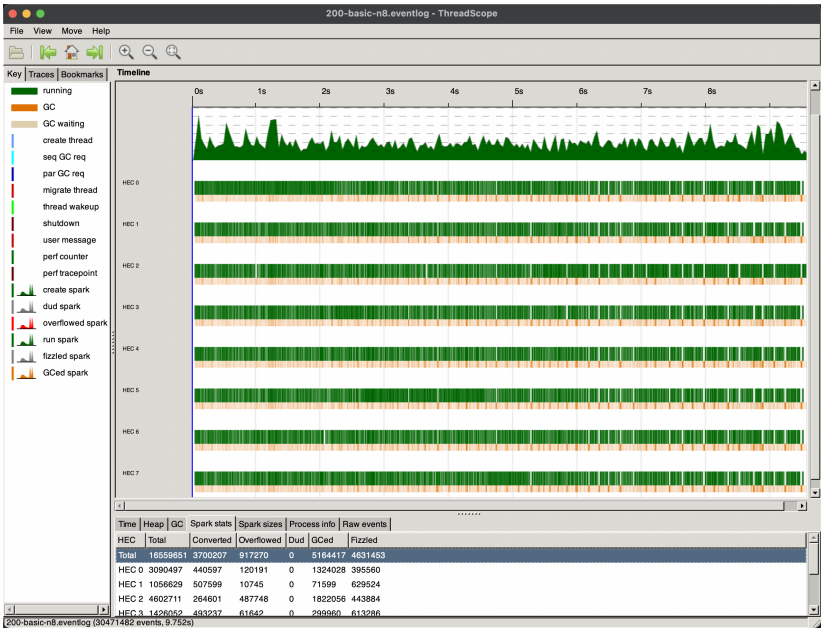


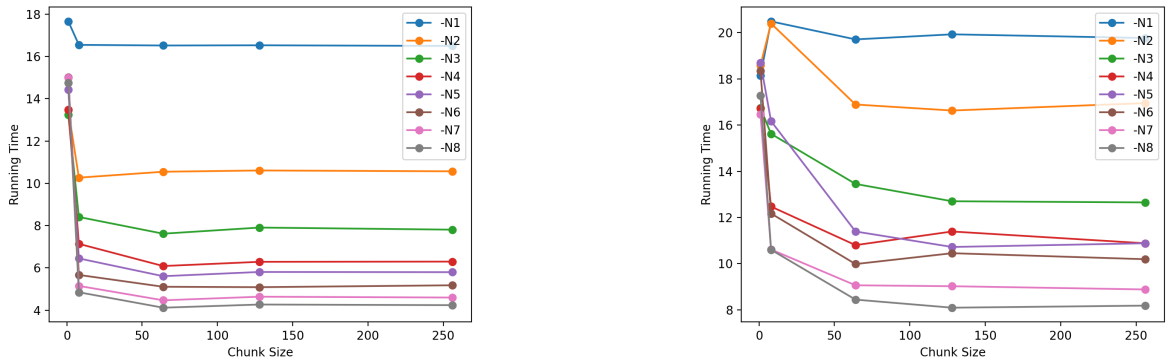
Figure 7: `-N8` Basic Parallel Eventlog

4.5.2 Chunked List Strategy

1. It is observed that as the number of cores increases, the running time monotonically decreases. However, the rate of reduction in running time diminishes with the addition of more cores. This trend is attributed to the increased overhead associated with generating a greater number of sparks. As an example, from Table 3's first and second rows, we can see that the conversion rate improves as the `chunkSize` increases from `1` to `128`.

2. A noticeable decrease in running time is evident when the `chunkSize` is augmented from `1` to larger values. This observation aligns with expectations, as excessively small sparks may be ineffective, and generating an excessive number of sparks can lead to spark pool overflow.

3. Upon further increasing the `chunkSize`, a stabilization in running time is observed across all lines. This phenomenon likely indicates that the maximum branching factor of the computation tree has been reached.



(a) Graph A (b) Graph B

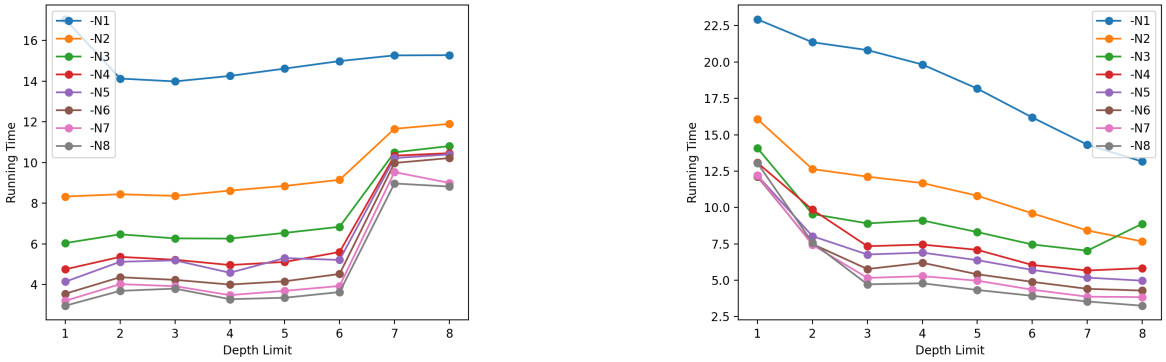
Figure 8: Chunked List Algorithm Eventlog

4.5.3 Depth-limit Strategy

1. In alignment with the analysis presented in Section 4.4.3, modifying the `Depth Limit` exerts contrasting effects on the two graphs, a particularly intriguing phenomenon. Specifically, for Graph A, a smaller `Depth Limit`

proves to be more effective. In contrast, for Graph B, an increase in the `Depth Limit` leads to a reduction in running time. The underlying rationale mirrors that previously discussed, and can be further comprehended by referencing the computation tree illustrations in Section 4.1.1. For Graph A, characterized by a balanced computation tree, an even distribution of computation across branches is ideal, necessitating a smaller **Depth Limit** to prevent the generation of superfluous sparks for each branch. In contrast, Graph B's computation tree is unbalanced, with a heavier concentration of computation in certain branches, hence requiring a larger **Depth Limit** to produce enough sparks to effectively utilize parallelism in these more laden branches.

2. Analogous to the chunked list strategy, an increase in the number of cores leads to enhanced speed. However, the rate of this increase diminishes owing to the associated overhead. As an example, from Table 3's third and fourth rows, we can see that the conversion rate improves a lot as the `Depth Limit` decreases from `5` to `1`.



(a) Graph A

(b) Graph B

Figure 9: Depth-limit Algorithm Eventlog

Cores: -N4, Graph: A	Sparks				Converted Ratio
	Total	Converted	GC'ed	Fizzled	
Chunk Size: 1	12064844	1113217	8906017	1103620	9.23%
Chunk Size: 128	12064718	3461267	10461195	319810	28.69%
Depth Limit: 1	400	246	13	141	61.5%
Depth Limit: 5	7434200	717987	4918254	622945	9.66%

Table 3: Comparison for Spark Conversion

5 Future Directions

We list several future directions to further explore on this project based on current limitations.

1. Our computational framework, which extensively employs list concatenation in parallel strategies, exhibits inefficiencies. A prospective direction for future development involves refining these methodologies to enhance efficiency.

2. A novel approach could be devised by amalgamating the depth limit restriction strategy with the chunked list strategy, potentially yielding a more effective solution.

3. Another avenue for exploration is to program the generation of cliques in a deterministic order, guided by a set of pre-established rules. This approach could potentially lead to more predictable and optimized outcomes.

4. The results presented in this report were obtained from simulations conducted on an M1 Mac, which may not provide stable outcomes. For future endeavors, we aim to test our project on a low-balanced server, executing multiple iterations and then averaging the results to acquire a more stable and reliable measurement of running time.

6 Code

We have included the link to our GitHub repository in this report for the convenience of readers seeking future reference. `https://github.com/William-WANG2/4995-PFP`

References

Maximal Clique Enumeration: Bron-Kerbosch Algorithm. https://www.youtube.com/watch?v=j_uQChgo72I, 2021. [Online; accessed 01-Dec-2023].

Coenraad Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph, 1973. URL https://dl.acm.org/doi/10.1145/362342.362367.

Stephen A. Cook. The complexity of theorem-proving procedures, 1971. URL https://dl.acm.org/doi/10.1145/800157.805047.

Richard M. Karp. Reducibility among combinatorial problems, 1972. URL https://web.archive.org/web/20110629023717/http://www.cs.berkeley.edu/~luca/cs172/karp.pdf.