Parallelization of Dictionary Problem using OpenMP

Wynn Kaza
University of Michigan (Undergraduate CE)
Ann Arbor, United States
wynnkaza@umich.edu

***Abstract*—In the study of network design, transportation routing, and memory resource allocation, new algorithms and challenges are being researched daily to optimize modern designs and find ways to reduce the run time of current algorithms. In this area of technology, a popular problem is the ”Feedback Vertex Set” (FVS), which is a challenging optimization task. This paper builds on a subset of the FVS problem which looks at finding the minimum number of words that can be used to define all the words in the dictionary. Refactoring the original algorithm for C++, I utilize OpenMP and various parallelization techniques to achieve parallel efficiency over the original serial algorithm.**

***Index Terms*—parallelism, OpenMP, minimum feedback vertex set, dictionary problem**

I. Introduction

With the introduction of modern processors with multiple cores and threads, achieving good performance has expanded beyond writing time efficient code. Based on the application, many modern developers have to focus on writing parallel code that can take full advantage of the underlying hardware. However, dealing with parallel code can introduce new obstacles such as worker balancing, inaccurate code, segmentation faults, among many other issues. In this paper, I look at the parallelization of the Dictionary Problem, and focus on a method which allowed for multiple threads to run the algorithm at the same time, even with the serial restrictions imposed by the graph and priority queue mentioned later.

A. Feedback Vertex Set

The FVS algorithm involves the identification of the smallest set of vertices in a graph that can be removed, turning the graph into a directed acyclic graph (DAG). Another way to look at this problem involves finding the smallest amount of elements that cause the graph to have a loop or cycle. A variation of this algorithm, labelled the minimum feedback vertex set (MFVS) is NP-hard due to the difficulty in finding an exact minimal solution. However, a lot of modern research has been developed into this algorithm to discover heuristics and approximation algorithms to reduce the time complexity of MFVS in return for a almost perfect minimum feedback vertex set. This algorithm is important in many fields of study. In network design, identifying which vertices cause feedback loops aid in the development of efficient communication networks. In chip routing, MVFS is used to find optimal circuit routing, as circuits have to be modeled as a DAG. In graph mining, DAG are used to find patterns within the graphs. The pseudo-code for a basic version of FVS is shown in **Algorithm 1**.

Algorithm 1
Feedback Vertex Set

```
$Input$ : a graph $G$
$Output$ : a feedback vertex set
Function findFeedbackVertexSet(G)
Initialize the empty set $feedbackVertexSet$
while hasCycles do
  $maxDegree$ = -1
  $vertexToRemove$ = NULL
  for $vertex$ in $graph$ do
    if degree($vertex$) $>$ $maxDegree$ then
      $maxDegree$ = degree($vertex$)
      $vertexToRemove$ = $vertex$
    end if
  end for
  removeVertex($graph$, $vertexToRemove$)
  add $vertexToRemove$ to $feedbacKVertexSet$
end while
return feedbackVertexSet =0
```

In the simplified version of the FVS algorithm, it takes a graph as input and validates whether the graph has a cycle on ever iteration of the while loop. In the case that the graph does include a cycle, it will find the maximum degree within the graph and remove that vertex from the graph. This would not produce the MFVS, as there is a possibility that it would remove a high degree node that doesn't include any cycles. However, it explains the overarching idea behind the FVS algorithm, and demonstrates how the final result from this algorithm would produce a DAG.

B. Dictionary Problem

The Dictionary Problem is a subset of the FVS algorithm, and the focus of this paper. This paper is inspired by *Noel Garcia* of UCSD, and his work on solving this problem in GoLang [5]. The goal of the dictionary problem is to find a subset of the words in the dictionary that can be used to define every word in the dictionary. For example, imagine that there are two words defined in the dictionary. Firstly, **sword** defined as *a weapon with a long metal blade, used for thrusting or striking*. Secondly, **blade** defined as *the flat cutting edge of a weapon*. In this scenario, because within the definition of sword, the word blade is used, we could remove blade and simply replace it with the words within its definition. This would mean that **sword** would now be defined as *a weapon with a long metal the flat cutting edge of a weapon, used for thrusting or striking*. If this process is applied recursively, more and more words can be removed from the definitions of the words in the dictionary. Based on this explanation, the relationship between this algorithm and the original FVS algorithm becomes apparent. Recursive definitions of words can be viewed similarly to cycles within a graph, and within the scope of this problem, they are viewed as the same. Therefore, by removing recursive definitions of words in the dictionary, we are removing cycles and attempting to find what number of words removed would result in a acyclic graph. The format of the graph can be found in *Figure 1*.

Some points of interest in the graph construction are discussed below. Firstly, all vertexes are only defined once. For example, even if the word *a* is used multiple times in the sentences, there only exists one vertex with *a* as its key. Furthermore, we can recognize that this graph is a directed graph, with every vertex having a incoming list, representing the vertexes that have incoming edges into that vertex, and a outgoing list, representing the vertexes that have outgoing edges from that vertex. This will be more important when calculating the priority of elements later in the paper.

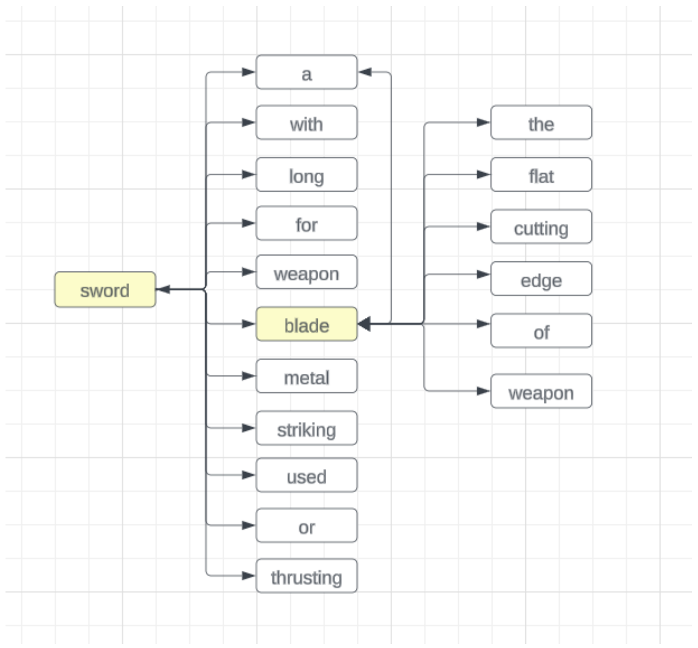


Fig. 1: Graph Construction and Connections

II. Related Work

A. A contraction algorithm for finding small cycle cutsets

Levy and Low present a new algorithm for identifying a cutset of the graph using graph contractions [1]. Their research presents an algorithm capable of finding the cutset within $O(|E|log(V)$. This is achieved by a combined usage of parallel edges, efficient edge shift in the contraction, and calculating vertex in and out-degree. Because each edge is only able to be shift $log(V)$ times, it follows that the final runtime for all edges would be $O(|E|log(V)$. Furthermore, compared to two other cutset algorithms, it is able to find more minimal cutsets.

B. Parallel Heap: An optimal parallel priority queue

Deo and Prasad present a novel priority queue that allows with p processors for the removal and insertion of $\theta(p)$ items, each within $log(n)$ times, where n represents the number of items within the priority queue [2]. This is achieved by maintaining nodes of p items, which helps to simplify the removal of nodes from the priority queue. Insertion is achieved by a top-down approach from the root down to the target node.

III. Dictionary Problem Algorithm

A. Dictionary Problem Implementation

When initially constructing the graph structure defined above, the graph included 110300 vertexes (one for each word), and 879439 edges. Beyond the graph, there is two other data structures that the algorithm uses. A *priority queue* to map vertices with high degrees to higher priority, and a *priority queue hash map* to supplement the priority queue. The serial implementation of the algorithm is shown in **Algorithm 2**.

ALGORITHM 2
Dictionary Problem

```
$Input$ : graph $G$, priority queue $PQ$
$Output$ : FVS $delNodes$
Function solveDictionaryProblem(G)
Remove vertexes w.o. any incoming edges
while $G$.size() $> 0$ do
    $vertex$ = highest priority in $G$
    delete $vertex$ from $G$
    $deleteList = vertex.outList$
    initialize $removalCount$
    while $removalCount > 0$ do
        for $Vertex\ V$ : $deleteList$ do
            if $v.inList$ == 0 then
                delete $v$ from $G$
                add $v.outList$ to $deleteList$
                $removalCount$++
            end if
        end for
    end while
    $delNodes$.add($key$)
end while
return $delNodes$ =0
```

The first part of the algorithm will remove any edges without incoming edges. Essentially, the database that I used for solving this algorithm has some words without definition. In this case, that vertex does not have any connections, and will therefore be removed and not counted towards the solution. The outer while loop will run this algorithm until every vertex in the graph has been removed, either by the outer while-loop, in which case it will be a part of the solution, or by the inner while-loop, which means that it was removed due to the removal of a different vertex. On each iteration of the outer while-loop, a vertex is selected and removed from the graph. This is the vertex with the current highest priority. Priority is defined by the size of the outgoing edge list. The outgoing edge list was used to determine priority within the PQ because the overarching goal of the algorithm is to remove the most vertex with each pop of the priority queue. By choosing the vertex with the most outgoing connections, it will have the highest probability to remove supplement vertexes through its propagation. Next, after the vertex is chosen, a list is created of every vertex that the selected vertex defines. Removal count represents the number of elements that were removed on that iteration of the for-loop. Therefore, while there exists any element that has been removed on that iteration of the code, the while-loop will continue to run. This concept is shown in *Fig 2.*. Finally, for every vertex in the list, if there are no incoming edges, as is the case when all incoming connections have been removed previously, then that vertex is deleted. Any vertex deleted in this fashion is not included in the removal nodes, as it is by definition simply the result of a removal of a vertex higher up the priority.

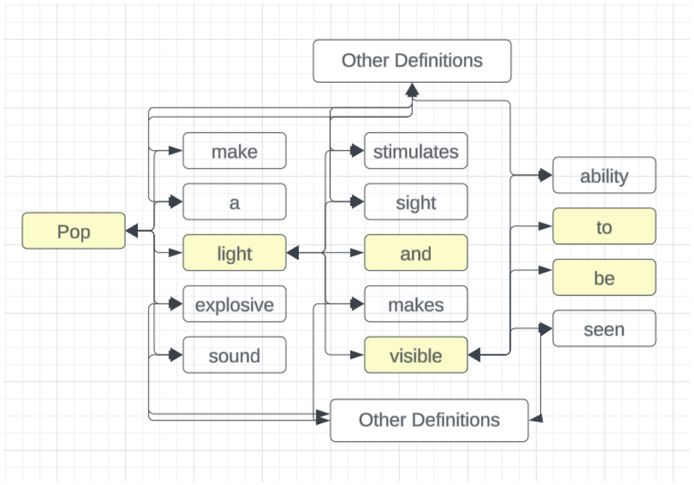


Fig. 2: How graph vertexes propagate deletions

As shown in *Fig 2.*, with the deletion of the word *Pop*, many subsequent words (light, and, visible, to, be) would also be deleted as their only incoming edge comes from *Pop*. In the figure, *Other Definitions* simply represents other words that a vertex could have a connection to.

B. Difficulty of Parallelism

As shown in **Algorithm 2**, the difficulty of the parallelization lies within the nature of the search. As each iteration of the outer loop depends on taking the top element of the priority queue, it means that every iteration of the outer loop would be limited serially. However, without parallelization of the outer loop, the efficiency of the algorithm will take a significant hit. Furthermore, in the scenario where there are parallel accesses to the priority queue, each of the threads would have to work independently to remove vertexes from the graph. This means that the inner loop would have to be sequential, as multiple updates to the graph cannot be happening in parallel, therefore limiting the efficiency of parallelism that this could provide. Another factor to consider is how the removal of one element can effect another element. For example, assume that the second highest priority element, while having many outgoing edges, only has one incoming edge. In this case, if two threads are running concurrently, and the top two elements are popped in parallel, the second element would be added to the final deleted notes list. However, in a sequential runtime, this element would not be considered, as the highest priority element would have removed it from the graph. This example is demonstrated in *Fig 3*.

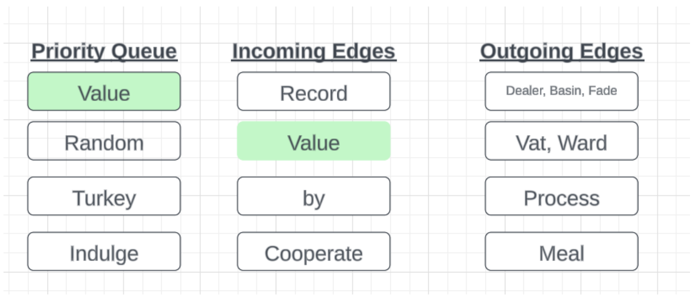


Fig. 3: Priority Queue Removals

In a serial implementation, the removal of *Value* (highest priority vertex) would have caused the removal of *Random*, as its only dependency is on *Value*. However, if the priority queue pops multiple elements in parallel, *Random* would be considered independent of *Value* and not be added in the final solution. This would produce an incorrect final result.

C. Development of Algorithm

To develop this algorithm, I tried to focus on the main difficulty in adding parallelism to this algorithm. While there were some sections that could be improved with a simple OpenMP construct because they were embarrassing parallel, I decided to ignore these functions to not invalidate the final timings of the algorithm (reported faster timing from parallelism of the embarrassingly parallel sections). To develop this algorithm, I looked at the parts of the program that were taking up the most runtime from the whole FVS algorithm. This was the *pop* and $popList$ function, where the inner details are mentioned in **Algorithm 2**. About 66% of the algorithm runtime was for the inner while-loop. Therefore, basing my logic off of Amdahl's law, I decided that this would be the best place to add speedup. To develop an algorithm, I thought about how to allow for multiple vertexes to all index the graph independently, without the need for a critical section. Beyond the buckets, I was forced to think of a way to develop a algorithm that would allow for multiple elements to be popped from the priority queue at the same time. Because of the previously mentioned limitations from graph connections, this was not a simple conversion. To achieve this, I thought of different ways which allowed for threads to be executed out of order, but still present the correct results at the end. Having a hardware background, I thought about how something such as a processor can execute instructions out-of-order but at the end, they are able to maintain correct ordering of the elements. I tried to implement a similar concept, but software design should not be built around this concept, which will be illustrated in *Results and Analysis*.

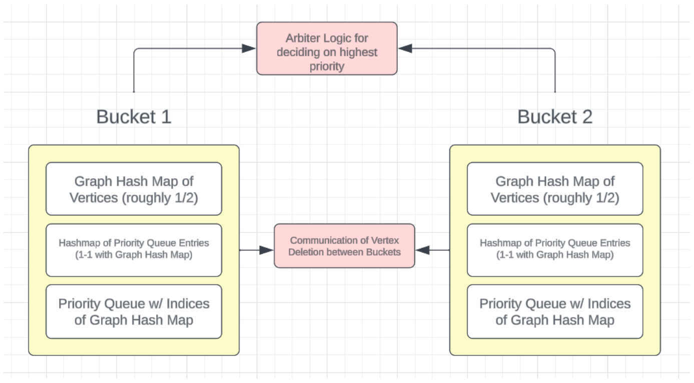
D. Attempted Solutions

In this section, I discuss various attempts to parallelize the code. One of these implementations was removed due to the massive software overhead, and the other implementation did not work due to a majority of the code being contained within critical sections. Overall, these ideas were not used in the final solution due to their inability to work, but the concepts were applied for the final solution. I included this section because I found some of these attempted solutions interesting, and worthwhile of a discussion about parallelism.

- **Individual PQ**: My first idea for implementation was to construct X buckets, where X was the current number of threads used in OpenMP. In this scenario, I could then perform individual operations on each of the buckets, and focus on removing the instances of cross-bucket communication that may be introduced. To develop this algorithm, I firstly split my graph into X buckets which was organized by a double hash of their key. As this method was included in the final method I decided upon, this part is explained in more detail in the next subsection(*Solving Parallelism*). Next, based on the elements in each of the buckets, I would create a priority queue and priority queue hash map for each of the buckets. To confirm that the correct priority was maintained, I wrote an algorithm that would maintain the differences in priority between each of the priority queues. This was done by having a priority queue that included X items, and using a critical section to pop the top element from the priority queue, and then add another vertex from the bucket it was popped from. Code for the arbiter logic is show in **Algorithm 3** below. If the highest priority item was already in use by a thread, the program would stall until that thread was finished. There is no reason to sync up between different threads, as all threads would stall whenever two elements map to the same bucket. However, the issue with this approach was that there was no speedup. This follows with the underlying logic of the code, as while the algorithm splits the original graph size into smaller thread-friendly computations, there is still the issue of the serial nature of the PQ. Furthermore, the addition of the constant checks and introduction of the overhead from OpenMP also slowed down the code. This design implementation is show in *Fig. 4.*. While *Fig. 4.* only demonstrates up to 2 buckets, this algorithm was tested and scaled for up to 16 buckets with correct results displaying. Additionally, I save the current state of in-process vertexes as the top element of the priority queue is being popped, to remove the possibility of the error coming from *Fig. 3.*. In the scenario that the only incoming edges into a vertex are all in the ”in-process” vector, that vertex will not be included in the output of the program. This is to make sure that it would match the behavior of a serial program attempting the same thing. This will be covered in more depth in the next section, as I used a similar implementation in my final approach.

ALGORITHM 3
PQ Arbiter

```
0: Input: PQ of size (# of threads)
0: Input: Buckets of size (# of threads)
0: Input: inProgress vector
0: function ARBITER
0:    $wait$ = boolVector(# of threads)
0:    #pragma omp parallel num_threads(numThreads)
   shared(PQ, wait, inProgress)
0:    while size(PQ) do
0:       #pragma omp critical
0:       if size(PQ)   0 then
0:          $vertex\ V$ = PQ.pop()
0:          inProgress.append(V)
0:          if size(buckets[hash(V)]) $> 0$ then
0:             PQ.append(nextElement(buckets[hash(V)]))
0:          end if
0:          wait[hash(V)] = true
0:       end if
1:
1:       while wait[hash(V)] = true do
1:       end while
1:       algo(inProgress)
2: wait[hash(V)] = false
2:    end while
2: end function=0
```


Arbiter Logic for deciding on highest priority
Bucket 1
Bucket 2
Graph Hash Map of Vertices (roughly 1/2)
Hashmap of Priority Queue Entries (1-1 with Graph Hash Map)
Communication of Vertex Deletion between Buckets
Priority Queue w/ Indices of Graph Hash Map
Graph Hash Map of Vertices (roughly 1/2)
Hashmap of Priority Queue Entries (1-1 with Graph Hash Map)
Priority Queue w/ Indices of Graph Hash Map
Fig. 4: Separating algorithm into buckets

- **One Element Parallelization:** In this implementation, I removed the concepts of buckets and returned to the serial version (one graph, one PQ, one PQHashMap). To parallelize this, I ignored the outer for-loop in **Algorithm 2.** and simply focused on the parallelization of the inner for- loop. After the top element is removed from the priority queue, a OpenMP construct is called on each element of the outgoing list (similar to a BFS). Because communication with the PQ is all done serially, there is no need to worry about adding a incorrect element to the deleted nodes vector. Similarly, continuing with the BFS concept, as all threads are removing vertex connections from the same ”layer”, there is no reason to provide synchronization between layers. When all threads have finished running their portion of the search space, the next element can be serially popped from the PQ and the same implementation can continue. There were a couple major issues with this implementation. Firstly, because of the removal of the buckets, there was only one graph. This means that every modifying operation on the graph (update, delete) was required to be done in a critical section. Furthermore, the local $deletion$ vector and $removalCount$ variable described in **Algorithm 2** were also required to be in a critical section. Therefore, this forced most of the algorithm to be run within a critical section, causing an increase in the runtime due to the overhead of using OpenMP constructs. A diagram for this approach is described in *Fig. 5.*. I believe that the reasoning above is the main reason to the failure of this implementation, on top of the simplicity of the implementation as it is similar to solving an ”embarrassing parallel” problem. The final issue is purely speculative, and is an assumption about this approach. Bench marking of the size of the $deleteList$ compared to the number of processors would have to be tested to confirm the validity of this statement. However, I believe there is an issue with the speedup from more threads. Because the thread-level parallelism suffers based on the number of elements in the deleteList, on the tail end of vertexes where the number of outgoing edges is smaller than the number of threads available, this algorithm heavily suffers from a massive under-utilization of threads. However, I am not sure how to improve on this design as this is an issue with the underlying algorithm and the locations where it is required to by synchronized, and not an issue with the parallelization of the algorithm.

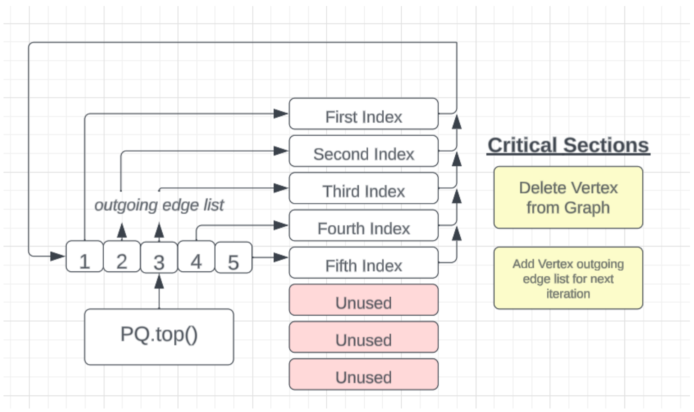


Fig. 5: BFS-Inspired Implementation w/ 8 threads

I also looked into lock-free data structures design such as the implementation of a parallel heap by N. Deo and S. Prasad [2] mentioned above. However, one issue with the paper was that it would do $\theta(P)$ insertions/deletions, but it couldn’t do less than that. Furthermore, I could not figure out a parallel solution which allowed for the rest of the algorithm to run freely without critical sections, even if the highest priority element was chosen correctly for each of the threads(refer to *Fig. 3.*). Therefore, I decided against the implementation of a lock-free data structure, and instead stuck with locking data structures as mentioned above in the *Attempted Solutions* sections, and also in the Solving Parallelism section below.

E. Solving Parallelism

The final implementation of the parallel algorithm combines many of the ideas discussed in the previous section.

Buckets: The central concept that I took advantage of was the idea of splitting the graph into smaller graphs, which I call buckets. These buckets provide a simple way for me to interface with the graph using different threads, without having to worry about race conditions or other issues that may rise from multiple threads all indexing the same graph at the same time. Similar to *Fig. 4.*, the final design includes a bucket with a graph hash map. To create this hash map, I utilized a 2D hash map, where the outer hash map was set to a fixed size of x, which as listed above is the number of threads. The inner hash map represented the same structure as the original graph. Therefore, for every vertex, it was double hashed to find where it belong within the overall structure. Comparing the sizes of the individual buckets, running on 2, 4, 8, 16 threads, the percentage difference between the smallest and largest buckets were respectively, 2.88%, 4.155%, 6.12%, 6.43%. Therefore, it appears as the trend is for the buckets to become more imbalanced with a increase in threads. However, because the differences in the workload of each bucket is not extreme, I do not believe that work-in-balancing would be a large factor in the solution of this algorithm. Unlike the previously mentioned attempted solutions, I decided to return to using a single PQ and PQHashMap due to overhead runtime costs of splitting the PQ and running the algorithm on each iteration. Instead, a global vector is used to maintain the current vertex in circulation. This vector behaves very similar to a queue. The method of implementation that I went for is similar to the design behind the pipeline stages of an Out-of-Order Processor. At the beginning state of the algorithm, vertexes are kicked off of the priority queue, sequentially and in-order. At the completion of a vertex, they are again listed within the vertex. However, only elements at the start of the vertex are able to be added to the final deleted nodes list. This is demonstrated in more detail within *Fig. 6.*. The idea behind this is to allow for multiple threads to run, but remove the case scenario that something similar to *Fig. 3.* could have. Looking at **Algorithm 2.**, this parallizes everything inside of the outside while loop. Synchronization happens when a element is added to $delNodes$. Even if the final thread has begun its iteration (the thread that will remove the final element of G), threads will continue to be generated because there is no way for the algorithm to know this. Therefore, when the conditional case of the outer while loop is hit, I break out of the while loop and kill the remaining threads (the information from the remaining threads is useless).

Example: Let's assume we have five vertexes (1, 2, 3, 4, 5). The complete graph is display in *Fig. 7.*. Assume that the arrows represent the outgoing edges (ex: vertex 1 would have three outgoing edges, and vertex 5 would have 2 incoming edges). Also assume that all 5 vertexes map to different graph hash buckets. Because priority is provided is in order of the highest number of outgoing edges, the PQ would be listed as 1, 3, 4, 5, 2 or a similar variation for the vertexes with ties. When vertex 1 is removed from the graph, it is added to a global vector. Next, vertex 3 would be removed from the graph. This would also be added to the global vector. This would cause an issue because the only incoming edge to vertex 3 is vertex 1, which means that it should have been removed when one finished running its algorithm. Therefore, this is where the global vector is utilized. If at any point during the runtime of vertex 1, it indexes into vertex 3, and it notices that vertex 3 only has one incoming edge, which would be vertex 1, it will invalidate vertex 3 in the global vector. This means that it will not be added to the deleted nodes vector at the end of the program because it has already been included for a previous vector. However, I was unable to figure out a way to stop the thread running vertex 3 after this occurs, so the thread will continue to run, essentially replicating the work done by vertex 1 after that point. Even if the thread for vertex 3 runs ahead of the thread for vertex 1, it will later become invalidated. This is because elements of the global vector are only read to the deleted nodes vector in-order. Therefore, until vertex 1 finishes, it would be impossible for vertex 3 to write to that vector. Essentially, each vertex of the global vector would do a forward search of all elements with a higher index for each of its searches.

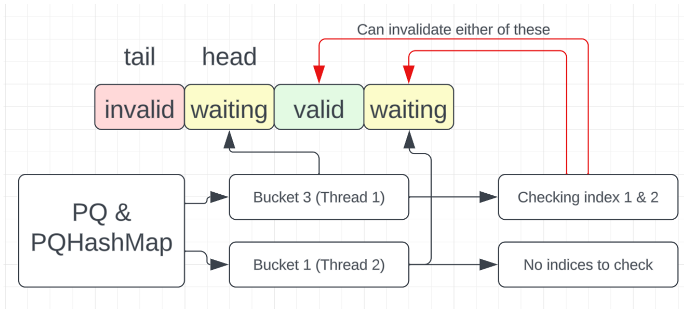


Fig. 6: Global Vector w/ 4 threads

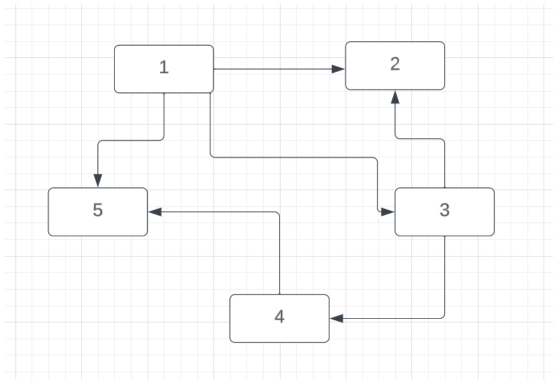


Fig. 7: Graph Example

While the design structure may be seen as inefficient, the size of the global vector is not $O(n)$. As the size of the global vector is limited to the number of threads, it's maximum size would be 16, when running the software at 16 threads. The global vector is implemented as a circular queue, and each node can hold a enum of three types {valid, invalid, waiting}. If the head is on a $valid$ vertex, it will add it to the deleted nodes, invalidate that index, and move the head to the next index. If it is on a invalid vertex, it will also move the head to the next index as long as the head is smaller than the tail. When the head is on a waiting vertex, it will pause there until that vertex either becomes invalidated or validated. Whenever a thread starts running the algorithm on a vertex, that index will go from the invalid state to the waiting stage. The tail represents the position where the next running thread would go.

modLen Optimization: One of the core functions within my code is the $modlen$ function which tracks the number of edges that a vertex has. For example, running $modlen(vertex->outList)$ would return the number of outgoing edges that the vertex has. While this may be seen as something easy to track, because the graph is being updated dynamically, this is bound to change. Whenever a vertex is deleted, the overarching algorithm does not go through every other vertex in the graph and remove all instances of it in the incoming and outgoing edges. Therefore, another vertex may have a reference to a previously deleted vertex, which would cause a segmentation fault. This is the main point behind modLen: to handle deleted vertexes. The original algorithm is very simple (loop through all elements, check if it still exists), and therefore don't count deleted vertices. However, the issue with the original algorithm is that it does not remove a node after counting it. That means that every time $modLen$ is called, it will loop through a previously deleted node. Therefore, I improve on this design by implementing a feature similar to path compression for the union disjoint set. Whenever a list is passed into the function, and a index of the list is empty, it will be removed from that list. Therefore, for any future iteration of the loop, that vertex will not be covered. This implementation works for both serial and parallel versions as the nodes being removed are nodes that should have already been removed. Furthermore, even if a race condition occurs, it will only mean that it would cause extra iterations of the for-loop and not a problem with the underlying state of the vectors.

F. Discoveries

While working on this project, I was able to learn more about OpenMP and the different constructs it provides. In this section, I will describe some of the interesting constructs that I learned about for future reference.

- **Range-Based For Loops**: As my code involves a lot of indexing with graphs and vertices, most of the code uses range-based for loops for indexing through the vertices. However, OpenMP currently does not provided any constructs with dealing with range-based for loops. Looking into this more, for OpenMP 4.0, it appears that the reasoning behind this is that it does support the syntax from C++11, which is when range-based for loops were initially defined in. Looking more into this issue, I discovered that OpenMP 5.0 does allow for range-based for loops to be utilized, however as my code was running on GCC 11, my version of OpenMP supported up to OpenMP 4.5, and only some of the features of OpenMP 5.0. One workaround I found to this solution is to utilize a specific single-producer tasking pattern, as shown in **Algorithm 4**, however, due to the overhead of this variation, it would not be worthwhile to implement this design.
- **Understanding Private and Shared Variables**: For this project, there were many sections that I opted to use private or shared variables within my OpenMP constructs, however I did not fully grasp the concept behind these variables. My main takeaway from my research is that OpenMP makes certain assumption about your variables based on where they are placed in relation to the OpenMP constructs. Variables declared outside a parallel version will be implicity *Shared* and variables declared inside would be implicitly *Private*. This follows along with the convention of a function. Essentially, the private keyword should be used when a variable is outside the declaration of the OpenMP construct, but you want each thread to store its own copy of the variable. It appears that the *Shared* construct does not have much benefit, but can be useful for documentation or to explicitly declare the variable as *Shared* in the scenario it may get changed to private somewhere else in the code through a *default(private)* construct.

ALGORITHM 4
For-Ranged For Loop for OpenMP 4.0

```
#pragma omp parallel
{
    #pragma omp single
    {
        for (auto& $var$ : $container$)
        {
            $pragma omp task
            {
                compute($var$);
            }
        }
    }
}
=0
```

IV. EVALUATION AND VERIFICATION

A. Verification

To verify the correctness of the serial C++ implementation of the GoLang version of the algorithm, I looked at the final results from my C++ implementation compared to the GoLang implementation. This was done by checking the nodes that were included in the final *deleted nodes list* mentioned above, and confirming which vertexes were removed in both algorithms. This was also confirmed by validating that the size of the *deleted nodes list* were of the same length in both algorithms. Similarly, a similar method of validation was utilized to confirm that the parallel implementation produced the correct results. I compared the results of the parallel implementation compared to my C++ serial implementation of the algorithm, and confirmed whether the *deleted nodes* matched between both versions.

B. Evaluation

To evaluate this algorithm, I looked at many factors. Firstly, I looked at the different sizes of the buckets, and compared the balancing of the nodes within the buckets. Secondly, I evaluated and timed multiple parts of the algorithm, such as graph construction, priority queue initialization, running the FVS, and splitting the graph to understand how each part of the algorithm effected the overall runtime of the code. I did not include timing information for the initial reading of the data from the JSON files, or for the verification algorithm run at the end. Overall, it was significantly noticeable that the majority of the application runtime was within the running of the FVS, and the extra work to set up the ability to parallelize the algorithm had a minuscule effect on the overall runtime. This is covered more in **Results and Analysis**

V. Results and Analysis

In this section, I will cover the results of running this algorithm on different number of threads(1, 2, 4, 8, 16) and also against the baseline serial implementation. I decided to run both the parallel algorithm at 1 thread, and the serial algorithm due to the significant changes in the code base to allow for the parallel algorithm to run.

A. Runtime of Bucket Splitting and Balancing of Buckets

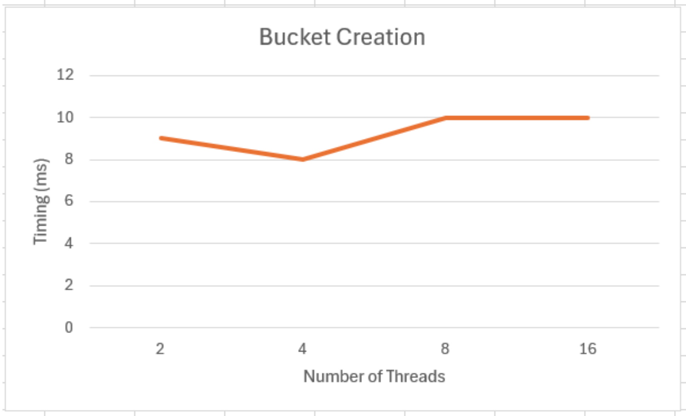


Fig. 8: Bucket Runtime

Looking at the graphs *Fig. 8* and *Fig. 9*, we can recognize that the number of threads has little impact on the runtime of the bucket creation. Furthermore, compared to the overall timing of the algorithm, the bucket creation takes up a minimal amount. Similarly, looking at the bucket sizes, we can see they are relatively balanced with little discrepancy between them.

B. Graph Size per Iteration

Looking back at **Algorithm 2**, we can see how the outer while loop tracks the current sizing of the graph.

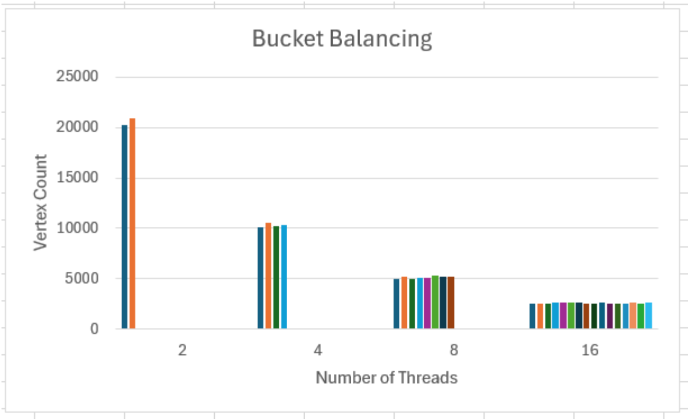


Fig. 9: Bucket Sizes

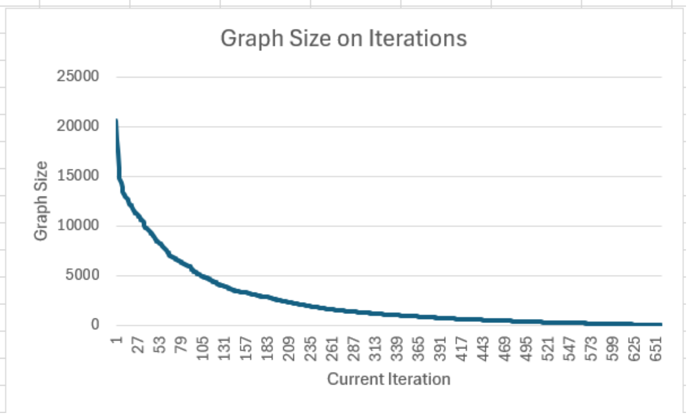


Fig. 10: Graph Sizing

Fig. 10 displays how the graph size changes every iteration. This is a snapshot taken after whenever a thread ”retires”. Essentially, we can recognize that the number of work from a PQ pop is getting increasing less as the search space decreases. This means that the problem will move towards being more sequential. In the start of the algorithm, many vertexes are disconnected allowing for the opportunity of the vertexes in the global vector to not collide. However, as the search space gets smaller, the chance of two vertexes in the global vector colliding get higher, to the point that the algorithm is essentially serial at the tail end. However, I do not believe this is a massive problem as work per PQ pop is much less towards the tail end.

C. Runtime of the FVS

Looking at *Fig. 11*, I have placed two lines illustrating the runtime of the algorithm with the serial implementation at different thread counts (obviously all the same), and the parallel implementation at different

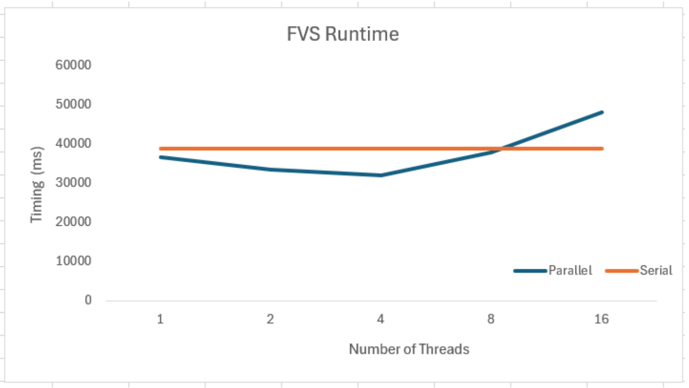


Fig. 11: Overall FVS Runtime

thread counts. I decided on only running up to 16 threads with OpenMP because the runtime begins to increase exponentially from 16 threads and beyond. Looking at the graph, we can also recognize that the speedup of the parallel version at one thread is faster than the serial implementation. My assumption behind this is due to the reformatting of my code. Changes to the modLen function, alongside changes to various other functions could have provided enough of a small speedup compared to the original algorithm.

My assumption for the increase in timing from 8 threads and beyond is due to the $O(X)$ search that the global vector has to do for every vertex it searches. As the global vector increases in size, it is perfectly reasonable for the search time to increase. I could not find a solution around improving this system. Another issue is that some functions dealing with the number of buckets has to iterate through all the buckets. This issue doesn't happen in the serial code because there are only one copies of all the data structures. As the number of buckets increase, the time it takes to run these functions increase.

The serial algorithm ran at *38839ms* and the fastest version of the parallel algorithm (4 threads) ran at *31938ms*. Comparing the improvement in time efficiency between these two, I obtain a speedup of 19.5%. Compared to the upper-limit speedup of perfectly parallel code (400%), this speedup is minuscule compared to the increase in hardware required.

As explained in the previous step, this algorithm is not a linear speedup. While there is a small relation between the number of threads vs the speed-up of the algorithm, there is no direct relationship between the two.

VI. Conclusion and Future Work

Through this paper, the implementation and strategies used for the parallelization of the Dictionary Problem are explained. While not all of the attempted solutions proposed successful results, the final resulting algorithm was able to achieve a moderate speedup over the original algorithm. Overall, as this problem is a DP program, finding a successful solution which removes thread contention is difficult, and I am happy with the results of achieving some form of speedup. In future works, there would be three new factors that I would like to achieve.

- **GPU-Based Programming**: I do not believe that a GPU implementation of this algorithm would be that efficient as the program is already taking a significant performance hit at a small number of threads, compared to the thousands of threads that are available on a GPU. Furthermore, due to the limited memory architecture of a GPU, I do not believe that the memory requirements for each thread would be able to work well and efficiently within the GPU architecture. However, it would still be an interesting problem to implement and verify how the current parallelization implementation works within a GPU. To study more in-depth on this topic, a understanding of modern graph mining implementations would have to be reviewed.
- **Different Datasets** Looking at the original calculations for the number of vertexes and edges (110300 vertexes, 879439 edges), this graph can be considered to be about sparsely connected. In the future, if it would be possible to find a matching dataset, it would be interesting to see the the effective speedup of the algorithm on a densely connected graph. I believe that such a graph would achieve a worse speedup due to the higher possibility of work inbalancing that could occur between different buckets.
- **Work Stealing/Sharing** The current implementation of the graph does not contain any concept of work stealing or sharing. That means that when a thread is done with it's work, it will wait idle until all threads are done with their own work. Having bench-marked OpenMP constructs for work-load imbalance for research previously, I have found that the work imbalance to be extremely low, and difficult to find a motivation to optimize the current static scheduler that is used within OpenMP compared to defining my own scheduling algorithm for the threads to utilize. Therefore, I am unsure how

much speedup a work stealing/sharing optimization would provide.

References

[1] H. Levy and D. W. Low, "A contraction algorithm for finding small cycle cutsets," vol. 9, no. 4, pp. 470–493, Dec. 1988, doi: https://doi.org/10.1016/0196-6774(88)90013-2.

[2] N. Deo and S. Prasad, "Parallel heap: An optimal parallel priority queue," The Journal of Supercomputing, vol. 6, no. 1, pp. 87–98, Mar. 1992, doi: https://doi.org/10.1007/bf00128644.

[3] A. Stivala, P. J. Stuckey, M. Garcia de la Banda, M. Hermenegildo, and A. Wirth, "Lock-free parallel dynamic programming," Journal of Parallel and Distributed Computing, vol. 70, no. 8, pp. 839–848, Aug. 2010, doi: https://doi.org/10.1016/j.jpdc.2010.01.004.

[4] C. Tadonki, "OpenMP Parallelization of Dynamic Programming and Greedy Algorithms," 2020. Accessed: Jan. 21, 2024. [Online]. Available: https://www.cri.ensmp.fr/classement/doc/A-733.pdf

[5] N. Garcia, "n6garcia/Directed-Feedback-Vertex-Set-Algorithm," GitHub, Sep. 22, 2023. https://github.com/n6garcia/Directed-Feedback-Vertex-Set-Algorithm/tree/master (accessed Jan. 21, 2024).