Final Project Report

Date	@November 29, 2022
Course	Parallel Computing
Finished?	☑
Link to Code	https://github.com/nymhyde/Parallel-Computing-Project
Tags	project writing

Parallel Depth-First Search for Directed Acyclic Graphs


```
# -----------------------------------------
#          << GROUP MEMBERs >>
# -----------------------------------------
# 1. Patel Dhruv (patel4db@mail.uc.edu)
# 2. Trexler Jackson (trexlejt@mail.uc.edu)
# -----------------------------------------
```

1. Introduction

Depth-First Search (DFS) is a common algorithm often used as a building block for topological sort, connectivity and planarity testing, among many other applications. We will be implementing the work-

efficient parallel algorithm for the DFS traversal of directed acyclic graphs (DAGs). We will be reporting the speedup of this parallel algorithm (in comparison with the serial implementation).

- Let a graph $G = (V, E)$ be defined by its vertex $V = \{1, 2, 3, ..., n\}$ and edge $E = \{(i_1, j_1), (i_2, j_2), (i_3, j_3), ..., (i_m, j_m)\}$ sets, with $|V| = n$ and $|E| = m$. The sequential lexicographic Depth-First Search (DFS) algorithm was proposed in [4].
- The DFS traversal problem requires us to compute : parent information, pre-order (start time) and post-order (end time) for every node in G, as shown in Figure 1.

2. Literature Survey

The lexicographical DFS problem can be subdivided into two major sets of algorithms:

1. Finding the Pre-order (start time) and the Post-order (end time) for every node in the graph G.
2. Conversion from a DAG to a Directed Tree (DT).

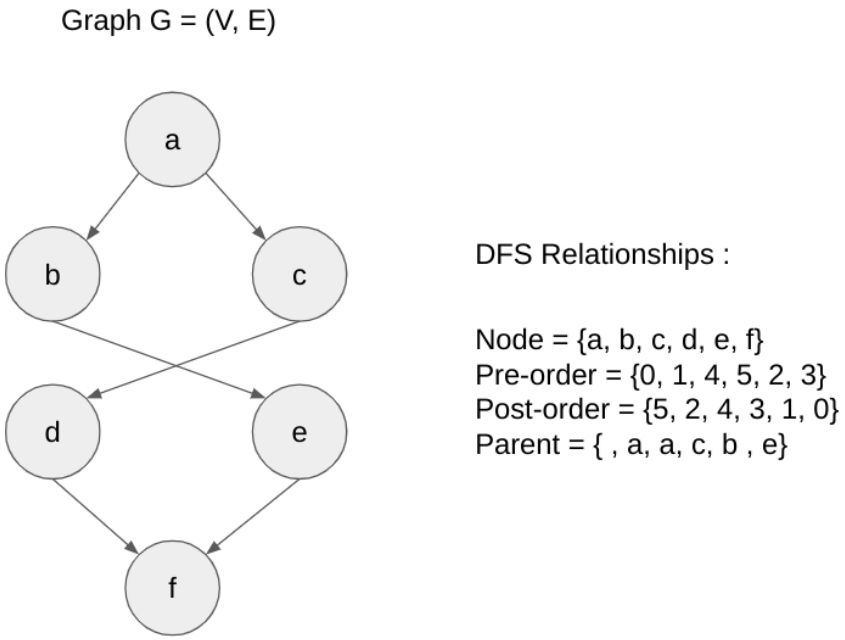


Figure 1 : Graph and associated DFS relationships

- Before we proceed to discuss on how the paper [3] suggests to find the pre-order and post-order, we need to establish some math.

Definition 1 - Let ς_p and ζ_p denote the number of nodes reachable under and including node p, where if a sub-graph is reachable from k multiple parents then its nodes are counted once and k times, respectively.

Lemma 1 - For a DT, where each node has a single parent, $\varsigma_p = \zeta_p$ is simply the sub-graph size, while for a DAG we have $\zeta_p \geq \varsigma_p$.

- For example, in Figure 1 we have $\zeta_p = 7$ and $\varsigma_p = 6$, because we double counted the node f in the former case.

Note that the aforementioned lemma follows directly form the definition of a DT.

- Also, the following recursive relationship could be observed :

$$\zeta_p = 1 + \sum_{i \in C_p} \zeta_i$$

where, C_p is ordered set of children of p.

It so seems that it's easy to compute ζ_p (recursive relation), compared to ς_p can be expensive to compute [3]. Let $\bar{\zeta_l}$, where l is an index of the exclusive prefix sum list, of the list ζ_i, where $i \in C_p$.

$$\bar{\zeta_l} = \sum_{i<l,\ i \in C_p} \zeta_i$$

Having established that, the values of ζ_p can be computed for a DT or DAG by traversing the graph bottom-up, from leafs to roots, as shown in Algorithm 1.

```
<< Algorithm - 1 :: Sub-Graph Size (bottom-up traversal) >>
# ------------------------------------------------------- #

Initialize all sub-graph sizes to 0.
Find leafs and insert them into queue Q.
while Q != {Φ} do
  for node i in Q do in parallel
    Let P_i be set of parents of i and queue C = {Φ}
    for node p in P_i do in parallel
      Mark p outgoing edge (p, i) as visited
      Insert p into C if all outgoing edges are visited
    end for
  end for

  for node p in C do in parallel
    Let C_p be an ordered set of children of node p
    Compute a prefix-sum on C_p, obtaining zeta_p (use lexicographic ordering of elements in C_p)
  end for

  Set Q = C for the next iteration
end while
```

Having computed ζ_p and $\bar{\zeta_p}$ for each node p in the Graph G the paper now expresses the pre and post-order times as their functions.

Theorem 1 - Let ζ_i be the sub-graph size for node i in a DT and $\bar{\zeta_i}$ be the corresponding prefix-sum value. Then,

$$preorder(p) = k \ + \ \tau_p$$

$$postorder(p) = (\zeta_p \ - \ 1) \ + \ \tau_p$$

where, $\Re_{r,p} = \{ \ r, i_1, i_2, ..., i_{k-1}, p\}$ and, $\tau_p = \sum_{l \in \Re_{r,p}} \bar{\zeta_l}$

Corollary 1 - Let a path from root r to node p be an ordered set of nodes $\Re_{i,p} = \{ \ r, i_1, i_2, ..., i_{k-1}, p\}$. Then, τ_p can be obtained recursively,

$$\tau_p = \tau_{i_{k-1}} + \bar{\zeta}_p$$

The above algorithm works well when the input is a DT. So, we now discuss how to convert any DAG to a DT.

```
<< Algorithm - 2 :: Pre-Order and Post-Order (top-down traversal) >>
# ------------------------------------------------------------------ #

Initialize pre and post-order of every node to 0.
Find roots and insert them into Queue Q.

while Q != {Φ} do
  for node p in Q do in parallel
    Let pre = pre-order(p) and post = post-order(p)
    Let C_p be a set of children of p and Queue P = {Φ}
    for node i in C_p do in parallel
      Set pre-order(i) = pre + zeta_i
      Set post-order(i) = post + zeta_i
      Mark i incoming edge (p, i) as visited
      Insert i into P if all incoming edges are visited
    end for

    Set pre-order(p) = pre + depth(p)
    Set post-order(p) = post + (zeta_p - 1)
  end for

  Set Q = P for the next iteration

end while
```

Definition 2 - Let $\Re_{r,p} = \{r, i_1, i_2, ..., i_{k-1}, p\}$ and $\Omega_{r_p} = \{r, j_1, j_2, ..., j_{k-1}, p\}$ be two paths of potentially different length to node p. We say that path $\Re$ has the first lexicographically smallest node and denote it by $\Re_{r,p} < \Omega_{r,p}$.

when during the pair-wise comparison of the elements in the two paths going from left-to-right the path $\Re_{r,p}$ has the lexicographically smallest element in the first mismatch.

- For example, in Figure 1 the two paths to node f are

$$\Re_{r,p} = [a, b, e, f]$$

$$\Omega_{r,p} = [a, c, d, f]$$

Clearly, $\Re_{r,p} = [a, b, e, f] \leq \Omega_{r,p} = [a, c, d, f]$ as $b \leq c$ lexicographically.

Theorem 2 - Let $\Re_{r,p} = \{r, i_1, i_2, i_3, ..., i_{k-1}, p\}$ and $\Omega_{r,p} = \{r, j_1, j_2, j_3, ..., j_{k-1}, p\}$ be two paths of potentially different length to node p. If $\Re_{r,p} < \Omega_{r,p}$ then $\Re_{r,p}$ is the path taken by DFS traversal.

- Putting it all together, the parallel DFS algorithm can be written as Algorithm 4.

3. Analysis

3.1 Complexity Analysis

The following definitions, lemmas and theorems establish the fact that the aforementioned algorithm is a work-efficient one.

Definition 3 - Let k_i be the number of elements insert into the queue at iteration i and $k = max_i\ k_i$ in Algorithm 1-3.

Definition 4 - Let d_i denote the degree of node i and $d\ = max_i\ d_i$ denote the maximum degree in DAG.

Definition 5 - Let δ_i denote the minimum depth of node i and $\delta\ = max_i\ \delta_i$ denote the diameter in a DAG.

Definition 6 - Let η_i denote the maximum depth of node i and $\eta\ = max_i\ \eta_i$ denote the length of the longest path in a DAG.

```
<< Algorithm - 3 :: Compute DFS - parent using Path (top-down traversal) >>
# ------------------------------------------------------------------------ #

Initialize path to {Φ} and parent to -1 for every node.
Find roots and insert them into queue Q.

while Q != {Φ} do
  for node p in Q do in parallel
    Let C_p be a set of children of p and queue P = {Φ}
    for node i in C_p do in parallel
      Let Q_r_i be existing and R_r_i be a new path.(R_r_i is a concatenation of path to p and node i)
      if R_r_i <= Q_r_i then
        Set Q_r_i = R_r_i
        Set parent (i) = p
      end if
      Mark i incoming edge (p, i) as visited
      Insert i into P if all incoming edges are visited
    end for

  end for
  Set Q = P for the next iteration

end while
```

```
<< Algorithm - 4 :: Parallel DFS  (Path) >>
# ---------------------------------------- #

Let Graph G = (V, E) and its adjacency matrix A

Run Algorithm 3
Run Algorithm 1
Run Algorithm 2

Resulting in parent, pre-order and post-order for every node in G
```

Lemma 2 - The parallel prefix sum of n numbers, can be computed in $O(log\ n)$ steps. Also, the algorithm performs $O(n)$ work.

Lemma 3 - Let $n\ =\ min(n_1, n_2)$, then identifying the first left-to-right pair of digits in two sequences of n_1 and n_2 numbers can be performed in $O(log\ n)$ steps, by doing $O(n)$ work.

Lemma 4 - The queue can be implemented such that parallel insertion and extraction of n numbers, can be performed in $O(log\ n)$ and $O(1)$ steps, respectively. Also, the algorithm

performs $O(n)$ work.

Theorem 3 - Algorithm 1 takes $O(\eta(log\ d + log\ k))$ steps and performs $O(m\ +\ n)$ total work to traverse a DAG. The number of processors $t \leq m + n$ actively doing work varies at each step of the algorithm. Here η is the length of the longest path in DAG, d is maximum degree in DAG and k is the maximum number of elements inserted into a queue.

Theorem 4 - Algorithm 2 takes $O(\eta\ logk)$ steps and performs $O(n)$ total work to traverse a DAG. The number of processors $t \leq n$ actively doing work at each step of the algorithm.

Here, η is the length of the longest path in DAG and k is the maximum number of elements inserted into a queue.

Theorem 5 - Algorithm 3 takes $O(\eta(log\eta + logk))$ steps and performs $O(\eta m + n)$ total work to traverse a DAG. The number of processors $t \leq \eta d + n$ actively doing work varies at each step of the algorithm. Here, η is the length of longest path in DAG, d is the maximum degree in DAG and k is the maximum number of elements inserted into a queue.

Corollary 2 - Path based DFS takes $O(\eta(logd + logk + log\eta))$ steps and performs $O(m + n + \eta m)$ total work to traverse a DAG. The number of processors $t \leq m + n + \eta d$ actively doing work varies at each step of the algorithm. Here η is the length of longest path in DAG, k is the maximum number of elements inserted into a queue and d is maximum degree in DAG.

Note that in practice $\eta\ m \rightarrow m$ because the data structure for storing the path detects the same “parent” block during comparisons, which implicitly eliminates additional work. Also, recall that $d,\ k,\ \eta \leq n$ in a DAG and therefore the Path-based algorithm takes no more than $O(\eta\ logn)$ steps and performs $O(m\ +\ n)$ total work.

3.2 Parallel Analysis

The Algorithms 1, 2 and 3 spawn multiple threads wherever **do in parallel** is mentioned. Apart from this the prefix sum calculation in line 13 of Algorithm 1 will also run in parallel.

3.2.1 Task Dependency Graphs

We subdivide the dependency graph of the parallel DFS traversal algorithm into three meta-tasks : the Algorithms 1, 2 and 3, and attach their dependency graphs separately.

(1) Algorithm 1

- Level A - In this level, the sole task is to initialize all sub-graph sizes to 0 and finding the leaf nodes and initializing the queue Q with them.
- Level B - All tasks in this level execute parallely for all the elements of Q, where in each task B_i involves initialization of the queue C accessing the parent list of the node (P_i). Notice that, there will be $|Q| = q$ tasks in this level.
- Level C - All tasks in this level execute parallely for all the parents of the ith node initialized in B_i where in each task C_i involves marking the edges between the parent and it’s child as visited; then, enqueuing the parents which have all their outgoing edges marked as visited into a queue C. Notice that, the number of tasks in this level will be $|P_i|\ *|Q|$.

- Level D - Wait for all the tasks in level C to finish updating the list C, for further computation.
- Level E - Wait for all the tasks in level B to finish their execution.
- Level F - All tasks in this level execute parallely for all the elements p of the list C accessing it's children C_P. Each meta-task in this level F_i represents the dependency graph of the exclusive parallel-prefix sum. Notice that, the number of tasks in this level will be $|C|$.
- Level G - Wait for all the tasks in level E to finish their execution. Then, reinitialize Q as C.

this block is repeated for every iteration of the outer while loop of the Algorithm 1 which is executed maximum η number of times.

(2) Algorithm 2

- Level A - In this level, the sole task is to initialize all sub-graph sizes to 0 and finding the root nodes and initializing the queue Q with them.
- Level B - All tasks in this level execute parallely for all the elements of Q, where in each task B_i involves initialization of preorder, postorder and accessing children C_p of the node p. Notice that, there will be $|Q| = q$ tasks in this level.
- Level C - All tasks in this level execute parallely for all the children of the ith node initialized in B_i where in each task C_i involves calculation of preorder and postorder, along with, marking the edges between the child and it's parent as visited; then, enqueuing those children whose all incoming edges are marked visited into queue P. Notice that, the number of tasks in this level will be $|C_p| * |Q|$.
- Level D - Wait for all the tasks in level C to finish updating the list C. Then, compute the final preorder and postorder value of the nodes in queue Q. Notice that, the number of nodes in this level will be $|Q|$ itself.
- Level E - Wait for all the tasks in level D to finish their execution. Then, reinitialize Q as P.

This block is repeated for every iteration of the outer while loop of the Algorithm 2 which is executed maximum η number of times.

(3) Algorithm 3

- Level A - In this level, the sole task is to initialize all paths to {ϕ}, parents to -1 and finding the root nodes and initializing the queue Q with them.
- Level B - All tasks in this level execute parallely for all the elements of Q, where in each task B_i involves accessing children C_p of the node p. Notice that, there will be $|Q| = q$ tasks in this level.
- Level C - All tasks in this level execute parallely for all the children of the ith node initialized in B_i where in each task C_i involves marking the edges between the parent and it's child vs visited; then, enqueuing the children which have all their incoming edges marked as visited into queue P. Each meta-task in this level C_i represents the dependency graph of lexicographic path comparison function, which is same as the dependency graph for parallel-prefix sum - as mentioned in *Lemma 3*. Notice that, the number of tasks in this level will be $|C_P| * |Q|$.
- Level D - Wait for all the tasks in level C to finish updating the list P, for further computation.

- Level E - Wait for all the tasks in level D to finish their execution. Then, reinitialize Q as P.

this block is repeated for every iteration of the outer while loop of the Algorithm 3 which is executed maximum η number of times.

(4) Algorithm 4

- This is the consolidated task-dependency graph of the parallel DFS traversal, where each meta-task represents the dependency graphs of the Algorithms mentioned within them.

4. Data - Structures Used

4.1 Serial Algorithm

```
typedef vector<int> vi;
struct compressed_sparse_column
{
    vi data;
    vi row;
    vi column;
    vi index_column;
};
```

```
struct graph
{
    compressed_sparse_column* dataset;
    int vertices;
    int edges;
};
```

4.2 Parallel Algorithm

```
struct compressed_sparse_column
{
    int* data;
    int* row;
    int* column;
    int* index_column;
    int* index_row_start;
    int* index_row_end;
};
```

```
struct graph
{
    copressed_sparse_column* dataset;
    bool* roots;
    bool* leaves;
    bool* singletons;
    int vertices;
    int edges;
};
```

5. Implementation of Algorithm

The datasets [1] used were available in Matrix Market Exchange Formats which were then converted to the standard CSC (Compressed Sparse Column) format; which concatenates all non-zero entries of the matrix in column major order and records the starting position for the entries of each column. In addition to the standard CSC format we also store the window within which the parent information of a node lies, this allows us to easily traverse and access information associated with outgoing and incoming edges.

- Assuming that storing a node requires $\beta = 4$ (or 8) bytes. Then, the sequential DFS algorithm uses approximately $(m + 5n)\beta$ bytes to store the input DAG, queue, parent, pre-order and post-order output arrays.
- The parallel DFS algorithm only requires an additional $(m + kn)\beta$ bytes for its data structures. Here, the first term in the sum corresponds to ζ, while the last term corresponds to the path data structure storage, where k is the block size.
- After storing the data in the CSC format, we start with the actual kernel computations where we invoke all the kernels as mentioned below.
  - **5.1 Serial Algorithm**
    - graph* read_data(string_file)
    - void DFS_VISIT(graph* dataset_graph, int vertex, int* color, int* parents, int* discovery_time, int* finish_time)
    - void DFS(graph* dataset_graph, int* color, int* parents, int* discovery_time, int* finish_time)
  - **5.2 Parallel Algorithm**
    - __host__ graph* read_data(const char* file)
    - __device__ char* my_strcpy(char* dest, char* src)
    - __device__ char* my_strcat(char* dest, char* src)
    - __device__ int my_strcmp(char* a, char* b)
    - __device__ char* my_itoa(int number, char* str)
    - __device__ int exclusive_prefix_sum(int* zeta_tilde, int* zeta)
    - __global__ void calculate_exclusive_prefix_sum(bool c, int* zeta, int* zeta_tilde, graph* dataset_graph)
    - __global__ void subgraph_size(int* zeta, int* zeta_tilde, graph* dataset_graph)
    - __global__ void pre_post_order(int* depth, int* zeta, int* zeta_tilde, graph* dataset_graph)
    - __global__ void dag_to_dt(char **global_path, graph* dataset_graph)
    - __global__ void final_kernel(int* depth, int* zeta, int* zeta_tilde, int* parent, char** global_path, graph* dataset_graph)
- The code files have been attached for further reference.

6. Optimizations

- We use the following techniques to optimize our CUDA code.
  1. Path pruning.
  2. CUDA streams for asynchronous memory transfer.
  3. Inline functions for faster execution.
  4. Use of pinned memory.
  5. Use of arithmetic shift operators.
  6. Used Structure of Arrays instead of Array of Structures for storing the dataset in CSC format.
  7. Extensively used __constant__ memory (where ever possible)

7. Evaluation

- Since there are no existing codes available for the parallel DFS traversal algorithms, we would be reporting the speedup in comparison with the sequential DFS algorithm.
- First, to check for correctness of our implementation, we would run it on some small toy test case. Then, we would move onto bigger datasets (shown in Table 1) [1] and report the running time (of both sequential and serial implementations) and the speedup.
- When necessary, we would create DAGs based on these general graphs by dropping the back edges, in other words, only considering the lower triangular part of the adjacency matrix these graphs.
- We aim to compare the following implementations based on their speedup :
  - The sequential DFS algorithm
  - The optimized version of the CUDA implementation of the parallel algorithm

Table 1 - Sample DIMACS graphs / adjacency matrices

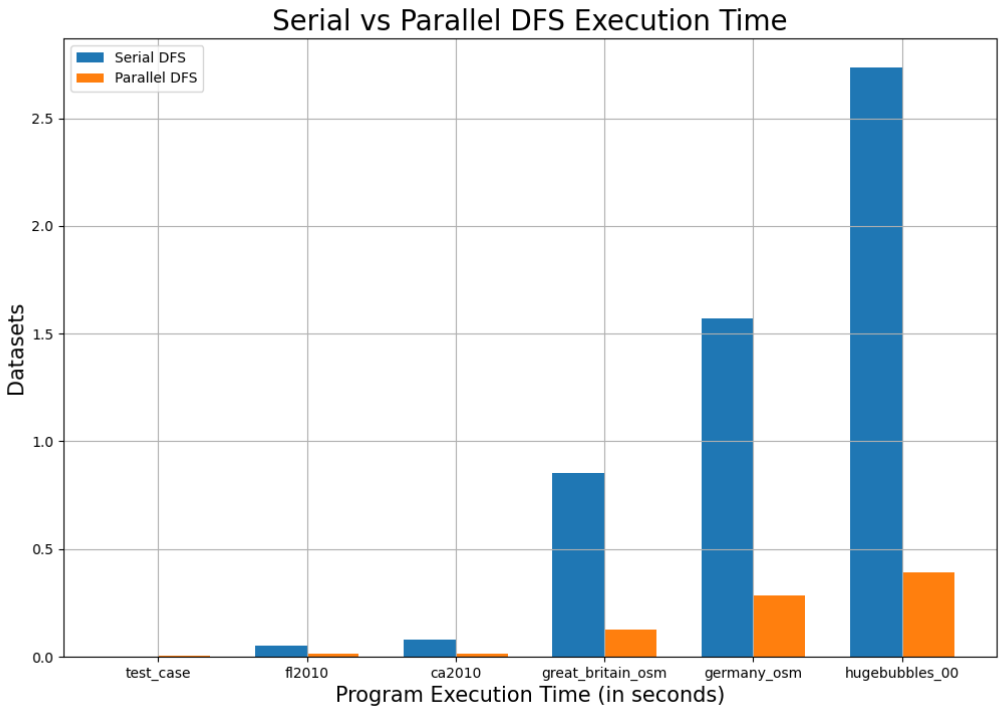
Dataset	**n**	**m**	**d**	η
test case	6	6	2	4
hugebubbles-00	18318144	30144175	3	6724
fl2010	484482	1270757	121	105
ca2010	710146	1880571	119	125
great-britain-osm	7733823	8523976	7	7076
germany-osm	11548846	12793527	12	8890

Table 2 - Serial vs Parallel Execution Times

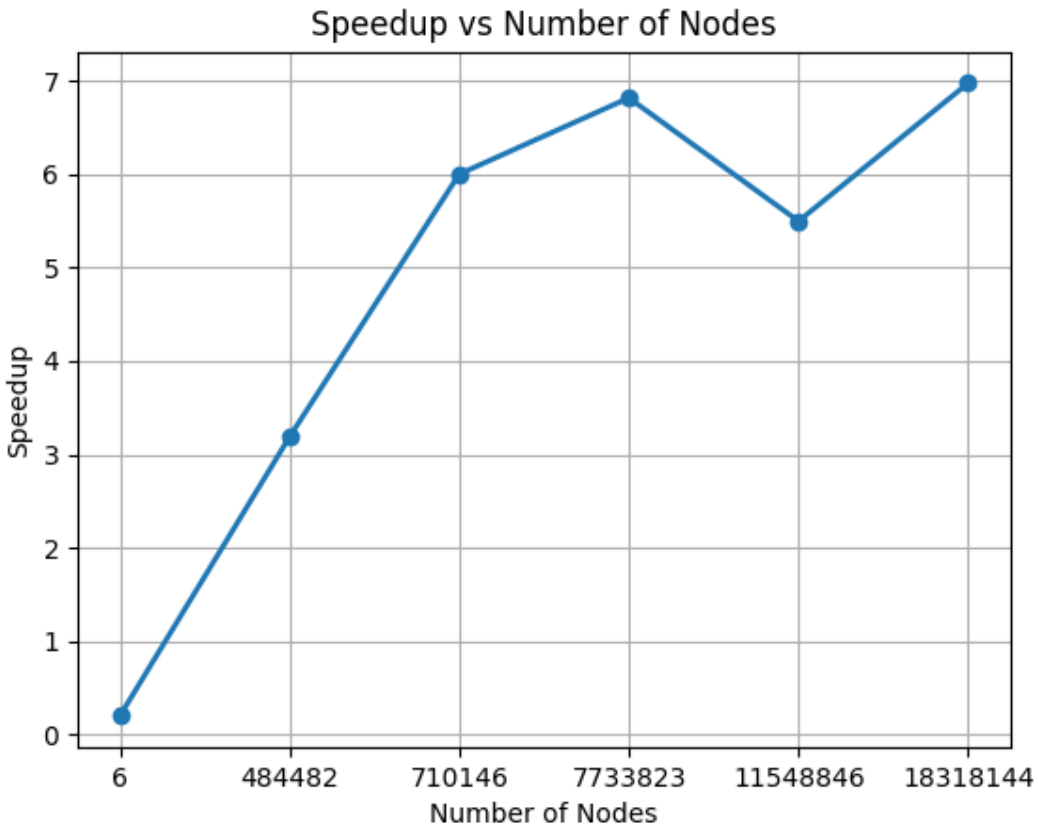
Dataset	Number of nodes	DFS (sec)	Parallel DFS (sec)	Speed Up
test case	6	0.001	0.005	0.2
fl2010	484482	0.051	0.016	3.1875
ca2010	710146	0.078	0.013	6
great-britain-osm	7733823	0.852	0.125	6.816
germany-osm	11548846	1.572	0.286	5.497
hugebubbles-00	18318144	2.734	0.392	6.974

8. Results

Plot 1 : Serial vs Parallel Algorithms execution times



Plot 2 : Variation of speed up with increase in the graph size



9. Conclusion

- We see that for smaller graphs the serial algorithm performs well, which can be due to the kernel launch, idling, communication or resource contention overheads. For larger datasets the speed-up, generally, increases (as evident from the plots above).

Appendix

- Task dependency graphs for Algorithms 1 through 4 are shown as under.

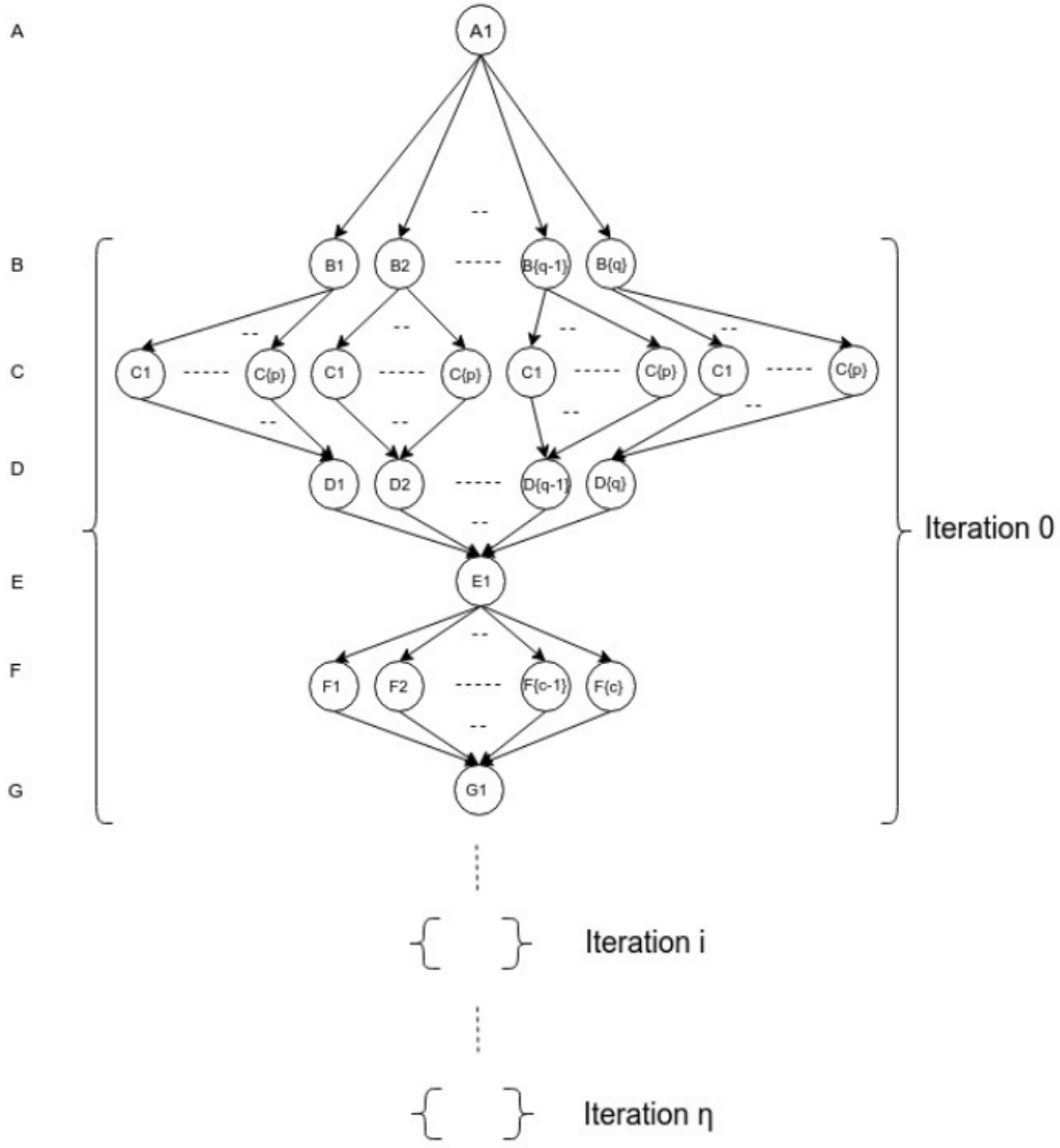


Figure - Task dependency graph for Algorithm 1

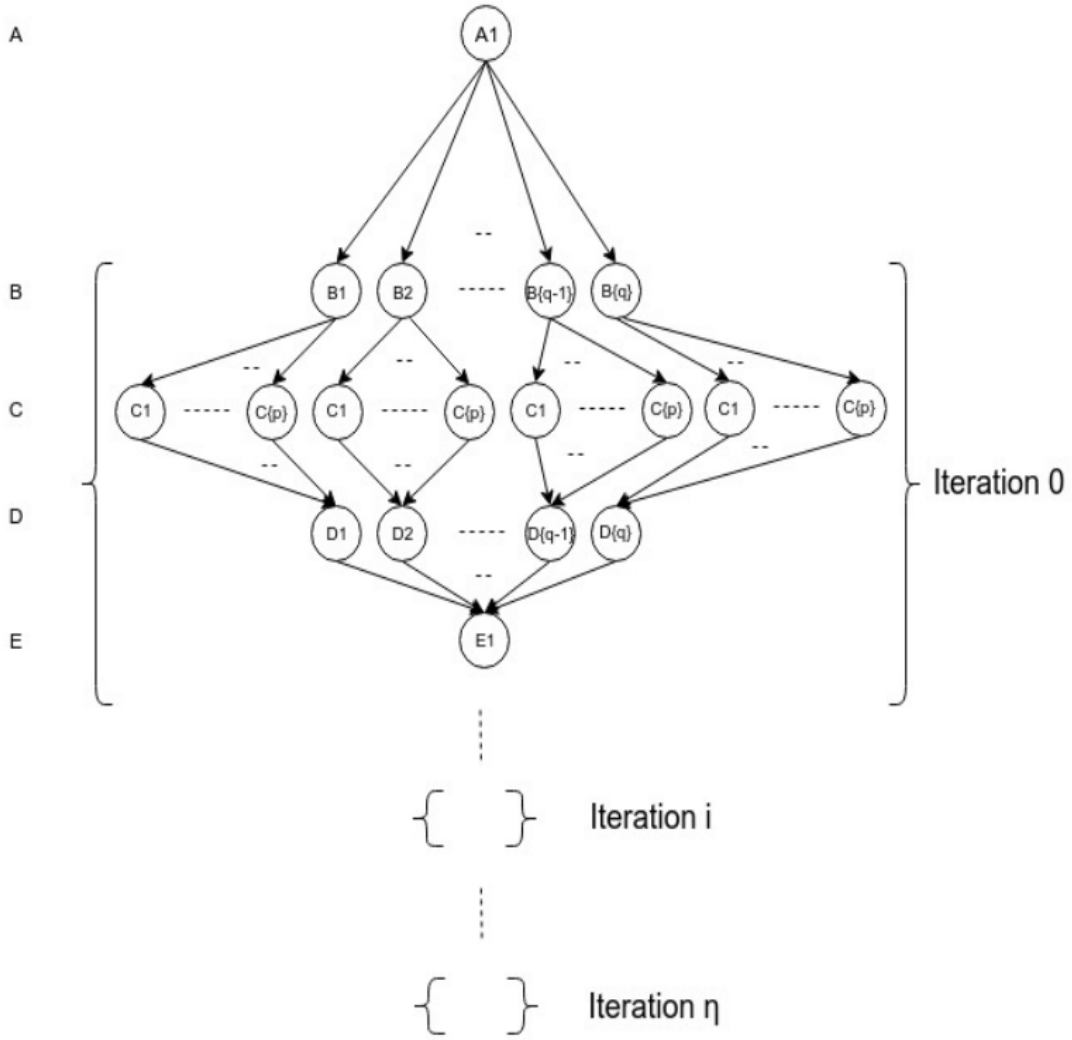


Figure - Task dependency graph for Algorithm 2

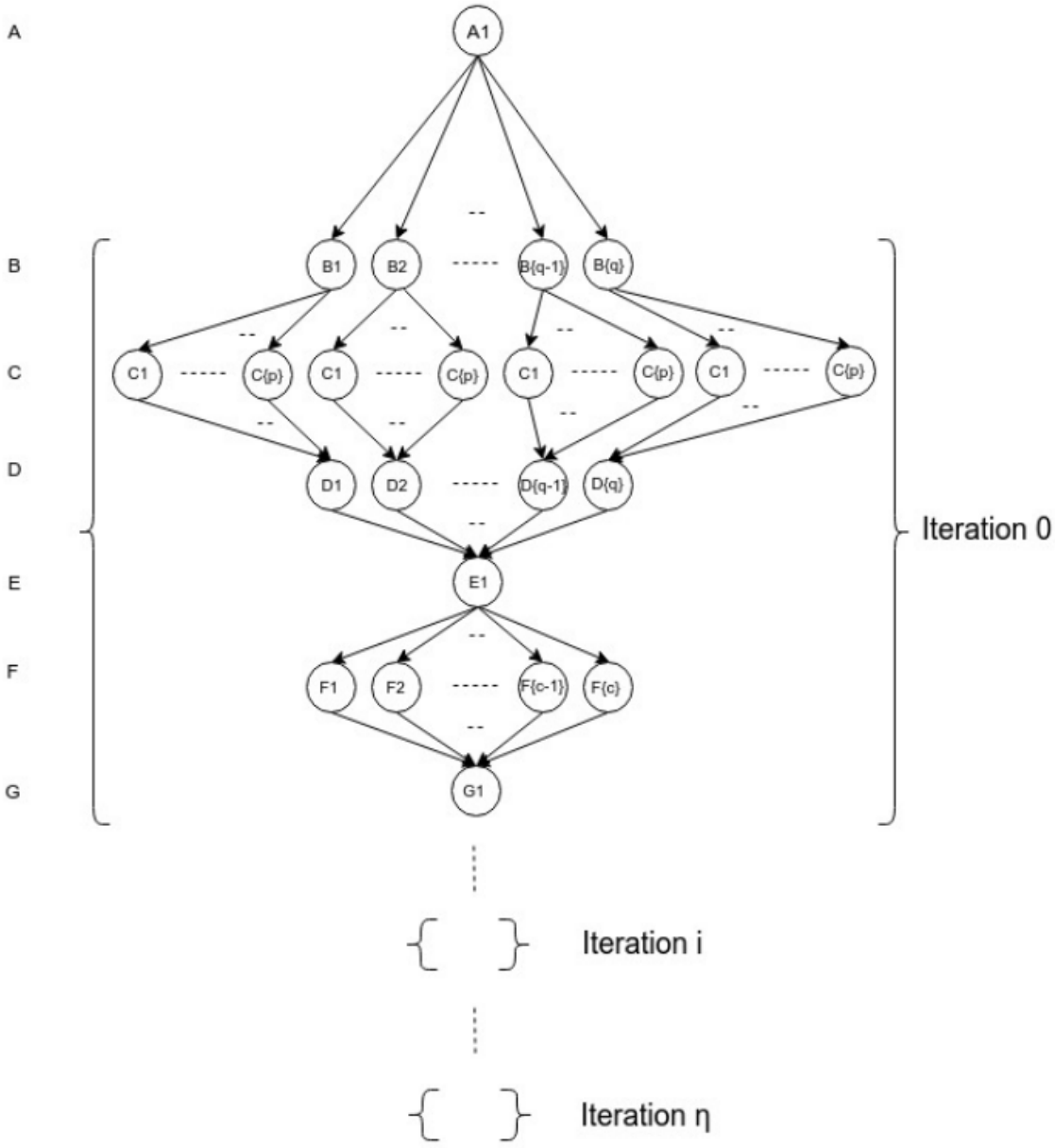


Figure : Task dependency graph for Algorithm 3

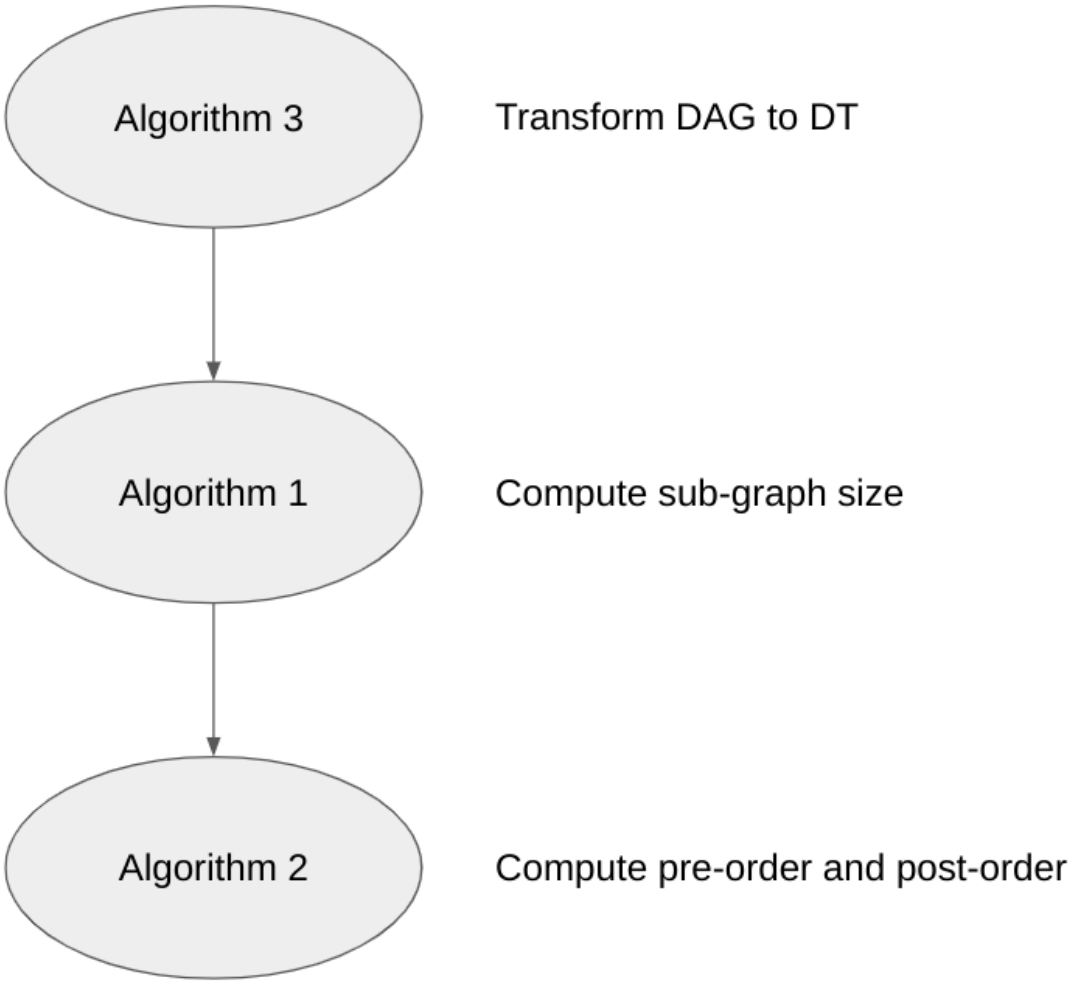


Figure - Task dependency graph for Algorithm 4

References

1. UF Sparse Matrix Collection. 2015. UF Sparse Matrix Collection Computations, version 11. (2015). Retrieved 10-Jun-2015 from https://www.cise.ufl.edu/research/sparse/matrices/list_by_dimension.html
2. Maxim Naumov, Alysson Vrielink, and Michael Garland. 2017. *Parallel Depth-First Search for Directed Acyclic Graphs*. NVIDIA Technical Report NVR-2017-001. NVIDIA, Santa Clara, CA.
3. Maxim Naumov, Alysson Vrielink, and Michael Garland. 2017. *Parallel Depth-First Search for Directed Acyclic Graphs*. In Proceedings of the Seventh Workshop on Irregular Applications : Architectures and Algorithms (*IA3 '17*). NVIDIA, Santa Clara, CA, Article 4.
4. R. E. Tarjan. 1972. Depth-first Search and Linear Graph Algorithms. (*SIAM* Comput. 1). Article 7, 146-160 pages.

Link to Github

https://github.com/nymhyde/Parallel-Computing-Project