we can only identify one level of nodes and mark them as non-garbage. However, for garbage collection, if we want to identify a certain path of useful reference in less amount of time(in other words we need to reach the end of more paths quickly), applying the BFS or Parallel BFS will not work, since both of which proceed the traversal by level.

Given the cases mentioned above, a loosely ordered Parallel Depth-Frist Search can enhance whole traversal performance to a great extent.

2. Background

2.1. Parallel unordered Depth-First Search

Depth-first Search in Graph is inherently sequential , presented by Aggarwal et al [3], it is the spanning tree of DFS must have only one branch, all node must be connected through that branch, those strictly ordered DFS has to accord to the lexicographical ordering property that requires the out-edges of a node in order, as a result, only one of the nodes in the frontier will be visited all other nodes will be put in to a stack and remain unvisited until all the sub-nodes of the first node have already been visited. But for the Parallel Garbage collection, this kind of rule doesn't have much help, because, we only need to reach the final nodes of a given path as soon as possible to mark all node in that path, if we do the conventional strictly ordered Depth-First Search on DAC, another branch will not be visited until the visiting of the first branch has already done, also, the backtrack of traditional DFS can also incur a lot of overhead, can also be time consuming. As a result, reasonably splitting the frontier node into different part, then apply two or more distinct Depth-first Traversals in each thread will be much more helpful. One of good practice is proposed by Mike et al [2]. There are several major contribution of their

- *Weighted-Splittable Frontier*: All nodes in this fron-tier have their own weight corresponding to its out-degree, when doing graph traversing, the frontier data structure in each single thread can be splitted into two in logarithm time based on its weight. Frontier splitting can help to achieve load balancing among threads to avoid thread hungry.
- *Communication Scheme*: *Acquire and Reply*, in short, for thread A who has no work to do will send the `Acquire` request to one of other thread called thread B who has work to do, if A got rejected, it will send `Acquire` to another thread to request work, and so on until thread A gets approved. Oth-erwise, if the `Acquire` request gets approved by thread B, then thread B will split its frontier by half, and share another half to the thread A. Now thread A has some work to do. What is worth notice is that this communication scheme is not one-to-all pattern, instead, it is a peer-to-peer pattern, the communication happends only among two individual threads, no shared resource is used in this case. so for each communication, when A send the `Acquire` request, thread A also tells the targeted thread B the address of A's frontier address, this can be counted as highly efficient, no shared resource will be used, race condition can be avoid.
- *Lazy Splitting*: Instead of actively splitting and shar-ing its frontier among all other idle threads in order to cread high parallism, processors create parallelism only if there is a demand for parallelism. Specifi-cally, PDFS algorithm splits a frontier only if another processor requests work from it. Two rules have been applied to control the degree of parallism: 1) its frontier contains more than K edges, or (2) it has locally processed more than K edges since the last work transfer (and it has at least one edge to send). The thread will split its frontier if and only if either of the two rules have been met, otherwise, just reject the incoming request, and work on its own.

3. Graph Representation Format

In graph storage, there are several types of graph representation formats in memory.

3.1. Adjcent list

An array of linked lists is used. Size of the array is equal to number of vertices. An entry array[i] represents the linked list of vertices adjacent to the ith vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.

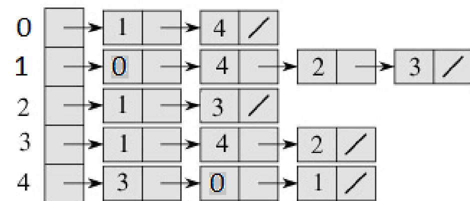

Figure 1. Adjacent List

3.2. Adjacency matrices

Adjacency matrices an adjacency matrix is a $|V| \times |V|$ matrix, where the cell [i, j] in the matrix indicates the existence of edge between the node i and node j, in the unweighted graph, a binary format of matrix can be used just to indicate whether there is an edge or not. In addition, if we want to use the weighted edge of graph, for each cell in the graph using the specific value of weight instead of binary value. Here's the adjacency matrix for a simple social network graph:

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	0	1	0	1	0
1	1	0	0	0	1	0	1	0	0	1
2	0	0	0	0	1	0	1	0	0	0
3	0	0	0	0	1	1	0	0	1	0
4	0	1	1	1	0	1	0	0	0	1
5	0	0	0	1	1	0	0	0	0	0
6	1	1	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	1
8	1	0	0	1	0	0	0	1	0	0
9	0	1	0	0	1	0	0	1	0	0

Figure 2. Adjacency matrix

3.3. Compressed Sparse Row (CSR)

Here in order for easy use, we simplify the CSR graph in-memory format to only two arrays, the one for the *begin* array, the other is *CSR* array. For the begin array, it is a 1-dim array, where each item in the array represent the beginning position of the adjacent nodes(neighbors) in the CSR array, e.p. for node with index i in the begin array, all of its neighbors are stored within the index range from (beg[i], beg[i+1]) in the CSR array. the begin array starts from 0, and its length equals to the number of nodes in the whole graph, while the number of elements in the CSR array contains all of the neighbors which number is equal to number of edges in the graph, in directed graph. Here is an example in given simple graph in form of CSR format along with its begin 3and CSR array4.

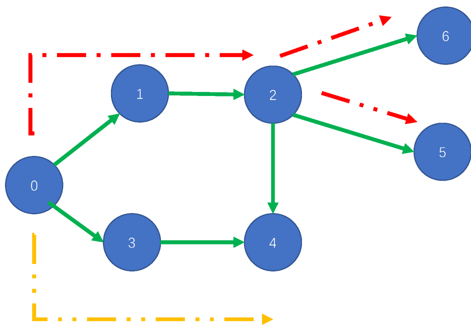

Figure 3. Begin Array

4. Bool Visited Array

The bool visited array is the key data structure in nearly all graph traversal algorithm. It has two functions: first, for each element in this array, it has only two status, *visited* or *unvisited*, it will bookkeep the node whether it has been visited or not; second, by using check this array we

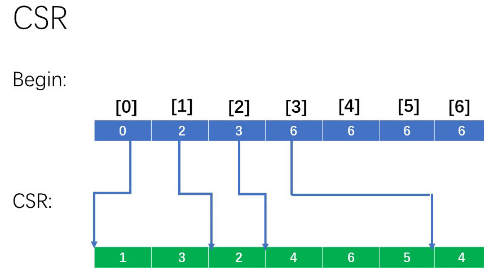

Figure 4. CSR Array

can get to know whether the traversal has already been completed, if all value has been set to visited true, thus the traversal will terminate. however for the second function for termination detection is rarely used in practice, since it requires additional amount of work to be done while the normal graph traversal in progress, thus overhead for those operation can not be ignored. There are much better ways for termination detection applied.

Another key issue in multi-thread environment with regarding to the bool status array is the access control. Since there are not only one thread doing read and write on the same data structure, as a result, this is a critical resource shared by all threads, thus it needs protection for visiting in order to guarantee the consistency and correctness of its value. In tradition way, it is not surprise for us to come up with the mutex for protection, or simply apply critical section around the value-operation statement, however, this can incur a lot of overhead, since only one of the thread will have the right to access or change the resource value, while all other thread are busy waiting for permission, it is wasteful and inefficient, which can even cause deadlock or thread-hungry.

One of state of art good solutions to address this problem is to atomic bool array in this case, this can be achieved by using the `std::atomic<T>`, and specify the T to bool type, this just one element in the array, for whole visited array `std::atomic<bool> *visited` can point to the atomic bool array. There are namely two advantages by using this structure, first, since atomic operation can be imple-ment by very basic hardware instruction, it can be very fast to do this operation comparing to other high-level methods. Second, atomic operation can be lock-free, it can be a good way to avoid race-condition in multi-thread environment, thus, we have no need to worry about consistency and correctness of the program execution. Third, busy waiting can be mitigated to a large extent. Quite different from the above ways to protect the shared resource, using an atomic array, at any time, if two or more thread access or change different elements in the array, that is allowed, however, for the previous methods, the whole bool array will be locked even if one thread is doing some modification to only one element in the array. There is a brief comparison of this two mechanism in following figure 55.

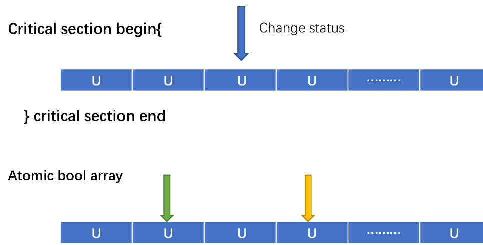

Figure 5. Tradition bool array *Vs.* Atomic bool array

5. Data Structure for Frontier

5.1. Bag

Bag data structure for implementing a dynamic unordered set. We first describe an auxiliary data structure called a pennant. We then show how bags can be implemented using pennants, and we provide algorithms for BAGCREATE, BAG-INSERT, and BAG-UNION.

A pennant is a tree of 2k nodes, where k is a nonnegative integer. Each node x in this tree contains two pointers x. left and x. right to its children. The root of the tree has only a left child, which is a complete binary tree of the remaining elements. Two pennants x and y of size 2k can be combined to form a pennant of size 2k+1 in O(1) time using the following PENNANTUNION function, which is illustrated in Figure.

A bag is a collection of pennants, no two of which have the same size. A bag S using a fixed-size a rray S [0. . r], called the backbone, where 2r+1 exceeds the maximum number of elements ever stored in a bag. Each entry S[k] in the backbone contains either a null pointer or a pointer to a pennant of size 2k.

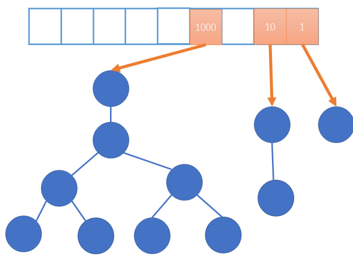

Figure 6. Bag

6. Thread communcation & Scheduling

6.1. Thread Communication Model

A good thread communication model has been proposed here.

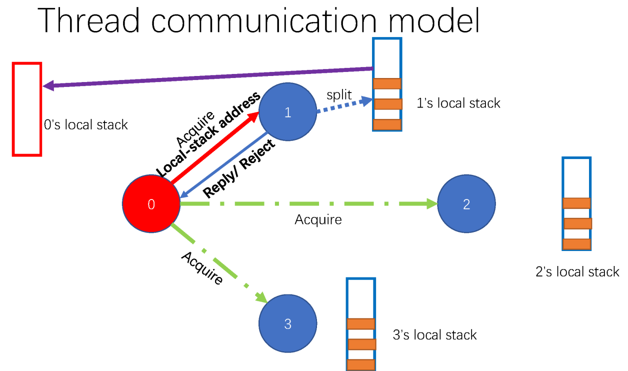

Figure 7. Thread Communication Model with callback

In this approach, there is no shared resource being used, it apply just *Peer-to-Peer* communication model, the basic logic of this model is simple, if one of thread named A run out of work, for each time it will send out the `request` message as well as its frontier address to one of other worker named B, if B is doing its own job or just split its own frontier a little bit early, B will send `reject` message to A, otherwise, B will directly split its frontier to the frontier address of A, and A will resume to work again. In another situation,if A gets `reject` message, A will send the same message again to another core, A will iteratively go through this procedure time and again until it gets frontier to work on.

An alternative way to do here is to use the shared resource like a global frontier. The cores who out of frontier will go directly to get the frontier node from the global frontier, if at that time the global frontier is empty, this core will wait until there comes up some frontier. The cores who have excess number of frontier will split the frontier and push its frontier to the global frontier in order for others to use. This kind of model can be more simple than the one mentioned above, however, the major drawback here is that at any time, only one cores could be able to add or get the frontier from the global frontier, while all others need to wait. This is highly inefficient, especially when thread number is large, all of the traversal will not be proceed if the threads need to share or get its frontier. here is a figure below show how it work.

6.2. Traversal using Task

Here, we present a parallel unordered Depth-First Search by using the task features from the OpenMp(available in OpenMp 3.0+), compared to the DFS using the stack, the idea is simple, for each neighbor of a given node, it will create a task for a new traversal started at this neighbor. This newly created task may be processed immediately or deferred until the next scheduling point. One thing we need to notice is that we use the untied feature from task, which means the task will not be bind to the certain

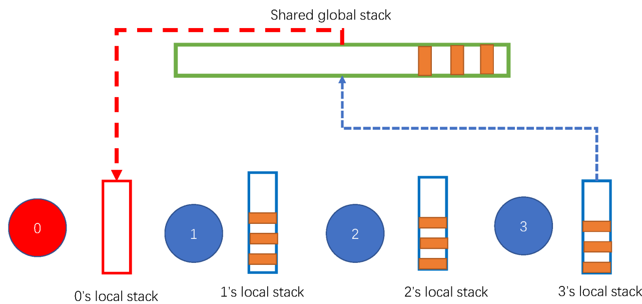

Figure 8. Shared Frontier Model

processor, it can be processed at any available processor.

Also there is one thing we need to care about, the cost of creating a task can overweight benefit of parallelism, here is an example of comparison between these two below,

It is not hard to see that the figure above can achieve

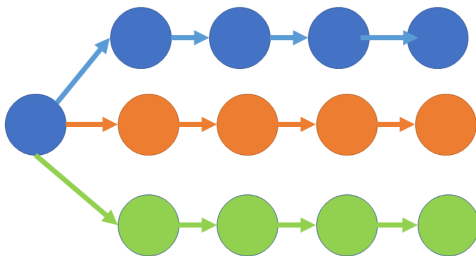

Figure 9. Optimal parallel

a very high parallel since the three distinctive branches almost have no intersection among each other. At the very beginning, if we create three task corresponding to each neighbors of the source vertex, the three task can achieve relatively high parallelism, and there is no need to use the synchronize among the threads who actually execute those tasks.

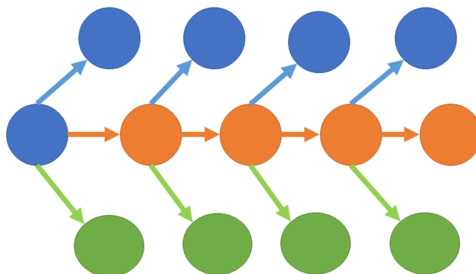

Figure 10. Anti parallel

in the real world the graph is highly irregular, we choose a extreme scenario just as the figure a bove, i n this case, if each time when we walk through the "back-bone" of the graph(the middle path) of the graph, we still do creating the task operation as ever, in this case, the task load is very small, for each task, only one vertex need to be visited, this is not desirable, the parallelism is low, and cost of creating task is high.

6.3. Work-stealing Scheduling

In the modern operating system, for the work-stealing scheduler, there are two kinds of abstraction, where threads correspond to workers, work deque corresponds to workers stack. When a subroutine is spawned, the subroutines acti-vation frame containing its local variables is pushed onto the bottom of the deque. When it returns, the frame is popped off the bottom. When a Worker runs out of work, it becomes a thief and steals the top frame from another victim workers deque.

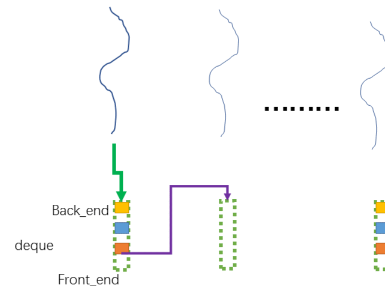

Figure 11. work-stealing

7. Experiment and Implementation

7.1. OpenMp

OpenMp [1] OpenMP (Open Multi-Processing) is an application programming interface (API) which supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran.

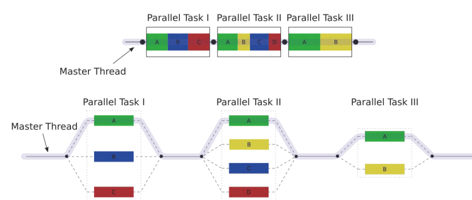

Figure 12. Fork & Join

In OpenMp, a master thread (a series of instructions executed consecutively) forks a specified n umber o f slave threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

7.2. Early version of using controller

We propose a early version of parallel unordered Depth-First Search, where a center controller has been used to manage several tasks, such as the *global frontier empty detection*, *traversal termination detection*. The detailed pro-cedure are put into Algorithm 1.

Algorithm 1 Global Shared Frontier Model

```
1:  Master:
2:  while traversal_completion ≠ True do
3:     if global_frontier is empty then
4:        v = get_an_unvisited_vertex()
5:        global_shared_frontier ←v
6:     else
7:        CONTINUE
8:     end if
9:  end while
10:
11:
12: Worker:
13: while traversal_completion ≠ True do
14:    private_local_stack ←global_frontier.get_frontier()

15:    source_vertices ←private_local_stack.pop()
16:    Depth_First_Search(source_vertex)
17: end while
```

Algorithm 2 Depth First Search

```
1: Depth_First_Search(Source_Vertices)
2: while private_local_stack ≠ empty do
3:    neighbors =vertices.get_neighbors(Source_Vertex)
4:    for all vertex :neighbors do
5:       if vertex is not visited then
6:          private_local_stack ←vertex
7:       end if
8:    end for
9: end while
```

7.3. Task dividing version

Here is the later version algorithms using OpenMp Task construct to create a task for each neighbors of the source vertex. The detailed procedure are shown in Algorithm 3 and 4.

8. Result Analysis

In this experiment, I mainly use the synthesized graph in doing test, with the nodes number the power of two, and the average out-degree for each node about 4 for the below figure shows the quantity variation in number of nodes and the number of edges.
We carefully configure the parallel unordered DFS from sc15 paper, and run it on my own laptop, the hardware

Algorithm 3 Task Dividing

```
1: DFS(Source_Vertex)
2: while private_local_stack ≠ empty do
3:    neighbors =vertices.get_neighbors(Source_Vertex)
4:    for all vertex :neighbors do
5:       if vertex is not visited then
6:          CREATE_TASK DFS(vertex)
7:       end if
8:    end for
9: end while
```

Algorithm 4 Main Iteration

```
1: DO IN PARALLEL
2: for all vertex :all_vertices do
3:    if vertex is not visited then
4:       DFS(vertex)
5:    end if
6: end for
```

is Intel Core i7-3537U, 2.0hz, dual cores, 8GB memory, with SSD, its performance of PDFS speedup are showned in the following figure.

For the first step, we need to find whether all of the nodes in the graph have been visited, so we need to evaluate the accessibility of the graph algorithm. The figure showing below visualize the accessibility of the algorithm.

Now, we can start to run the benchmark on the machine, this tool set takes relative long time to make the graph, and then run the several algorithms on those synthesized graph, we can see from the figure t hat i n 4 out of 8 full testing, the parallel unordered DFS speed up outperformances all other state of art algorithms, the compared baseline algorithms is serial DFS, Ligra graph processing library, Cong PDFS and Parallel Breadth-First Search, approximately 2.0x speedup better than the second fast algorithm.

8.1. Scalability

9. Conclusion

In the paper, we carefully check out the parallel un-ordered Depth-First Search Algorithm, and its practical ap-

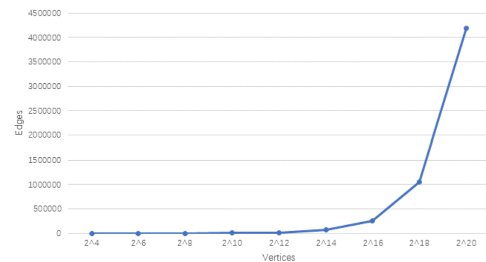

Figure 13. vertices to edges in synthesized graph

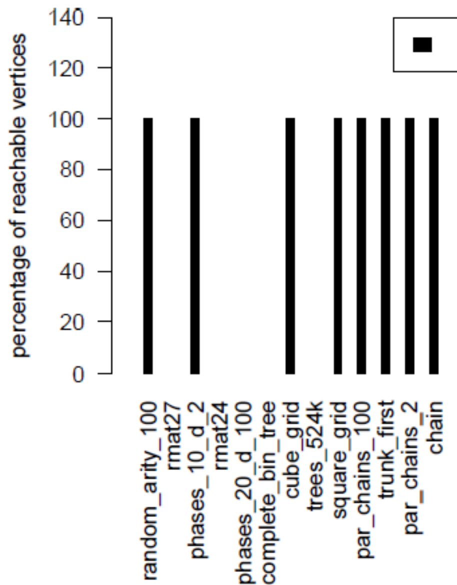

Figure 14. vertices reachability percentage

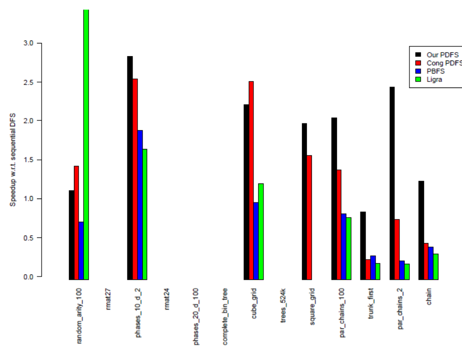

Figure 15. Speedup Comparison

plication. Before we take a look at algorithm, we first check the efficient data structure which could support this algorithm, like the bag frontier data structure in Parallel BFS [6], and weighted splittable frontier in Parallel Unordered DFS [2], then the work-efficient thread communication model as well as the scheduling pattern have been presented. To conclude, strictly ordered DFS is inherently sequential, turns out to be P-complete, however, a loosely ordered DFS can be found, and it application in parallel highly efficient Garbage Collection will be promising.

Reference

[1] Openmp. http://www.openmp.org/. 5

[2] Umut A Acar, Arthur Charguraud, and Mike Rainey. A work-efficient algorithm for parallel unordered depth-first search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 67:1–67:12. ACM, 2015. 2, 7

[3] A. Aggarwal and R. Anderson. A random nc algorithm for depth first search. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 325–334, New York, NY, USA, 1987. ACM. 2

[4] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. 1

[5] Christine H. Flood, David Detlefs, Nir Shavit, and Xiaolan Zhang. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 21–21, Berkeley, CA, USA, 2001. USENIX Association. 1

[6] Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 303–314, New York, NY, USA, 2010. ACM. 1, 7