Top-Down Parallel BFS

Muktikanta Sa

1 Introduction

We present a parallel top-down breadth-first search(BFS) algorithm shown in Algorithm 1. It is based on the algorithms[3, 4, 10]. The BFS is defined as, given a query vertex $v \in V$, output each vertex $u \in V - v$ reachable from v. The collection of vertices happens in a *BFS order*: those at a distance d_1 from v is collected before those at a distance $d_2 > d_1$. When the BFS algorithm terminates, it produces the *BFS tree* with root vertex v. In each step of the top-down approach, each vertex initially stored in a queue, called as a *current frontier*, checks all of its adjacent vertices are already visited or not. It transforms all unvisited vertices to a new queue, called as a *next frontier*, and marked all vertices as visited. This *frontier expansion* design is smartly captured by the sparse matrix-sparse vector multiplication(SpMSpV)[1]. The SpMSpV is formalized as $y \leftarrow Ax$, where x the input vector and it represents the current frontier, A, be the matrix and it represents the graph, and y be the output vector, and it represents the next frontier. Figure 1 depicts the BFS implementation with matrix vector multiplication.

2 Breadth-first search

A BFS algorithm implementation is shown in Algorithm 1, in Lines 1 to 42, begins by initializing the `dist` array(Lines 4-6). Then it initializes the source vertex's distance to zero (Line 7) and inserts that source vertex to the current frontier x. At each step from Lines 13 to 41, we iterate over only nonzeros values in the vector x. All these vertices processed in parallel(Lines 18-30). For each nonzero value of x, it reads the corresponding column and row values from matrix A(see Line 19 and 20). Then it checks the vertices are marked visited or not. If not visited, then it marked visited and insert into next frontier y. The lines from 23-27 is the critical section. After finishing all vertices in the x and before processing next frontier y, all threads have to reach the barrier(Line 31). Then the vectors x and y are swapped, and their corresponding sizes are being set. A single thread does this. The process is repeated

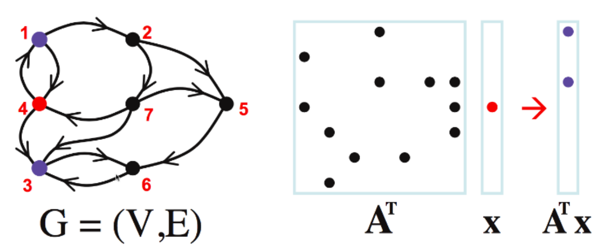

Figure 1: BFS implementation with matrix vector multiplication. Repeated multiplication gives multiple BST steps. The Figure is based on Kepner and Gilberts book [7].

Algorithm 1 Top-down Parallel BFS

```
Input: cols, rows                                                ▷ CSR data structures
Input: n, m                                        ▷ n: # of vertices, m: # edges
Output: bfs tree
procedure TDPBFS(cols, rows, srcv)          ▷ BFS staring from source vertex srcv
    int dist[] ← new int[n];
     int level ← 0;
    for (i ← 0 to n-1 in parallel) do
        dist[i] ← -1                          ▷ All vertices are -1 distance from srcv
    end for
    dist[srcv] ← level;                                ▷ init the srcv distance to 0
    int x[] ← new int[n];                                           ▷ Source vector
    int y[] ← new int[n];                                           ▷ Output vector
    int x_size ← 0;                                            ▷ Source vector size
    int y_size ← 0;                                            ▷ Output vector size
    x[x_size++] ← srcv ;                                     ▷ init the source vertex
    while (x_size > 0) do
        ——-Executed by single thread————
        {
        level ← level + 1;
        }
        for (i ← 0 to x_size -1 in parallel) do
            int nAdj ← rows[x[i]+1] - rows[x[i]] ;           ▷ number of adjacency
            int nAdjlist[] ← cols[rows[x[i]];                      ▷ cols index array
            for (j ← 0 to nAdj-1) do
                ——-critical section————
                if (dist[nAdjlist[j]] == -1) then                 ▷ Check visited or not
                    // y[nAdjlist[j]] ← y[nAdjlist[j]] OR nz[j] AND x[i]   ▷ nz[] : nonzero values
                    dist[nAdjlist[j]] ← level;              ▷ Set the distance from source
                    y[y_size++] ← nAdjlist[j];          ▷ Push the vertex into the new queue
                end if
                ——End critical section———
            end for
        end for
        ———barrier————
        //swap x and y and set the sizes accordingly
        ——-Executed by single thread————
        {
         temp ← x;
         x ← y;
         y ← temp;
         x_size ← y_size;
         y_size ← 0;
        }
    end while
end procedure
```

until all reachable vertices are traversed.

3 Related Work

The first parallel BFS was proposed by Gazit et al. [5]. They used the repeated squaring matrix method based on min-plus semiring. This method is not suitable for the graph like social networks, biological, communication networks, and several other graphs as these graphs are large in real-world.

A sparse graph can be viewed as a sparse matrix, and linear algebraic computations can be model as BFS like sparse matrix-vector multiplication [9]. So to improve the performance of BFS algorithm, one can adopt this. There have been several works has been done on large the graph to implement parallel BFS based on level-synchronous top-down and bottom-up approach [2, 8].

Harish and Narayanan[6] and Yang et al. [11] presented different parallel graph algorithms, including BFS on GPU. Several works can found both in theory and practice on developing fast parallel BFS algorithms both for distributed and shared-memory architecture.

4 Results

We conducted our experiments on a system with Intel(R) Xeon(R) E5-2690 v4 CPU having 56 cores with clock speed 2.60GHz. There are 2 logical threads for each core and each having private cache memory L1-64K and L2-256K. The L3-35840K cache is shared across the cores. The system has 32GB of RAM and 1TB of hard disk. It runs on a 64-bit Linux operating system. The metric for evaluation is the traversed edges per second (TEPS). For graph generation, we used the Ligra [a]. Some piece of code is taken from [b].

Observations: The plots in Figures [c] clearly depict the performance we get by increasing the number of threads did not scale. This is due to the critical section execution from Lines 23 to 27. In the future, we plan to devolve new approaches to increase the scaling.

References

[1] Ariful Azad and Aydin Buluç. A work-efficient parallel sparse matrix-sparse vector multiplication algorithm. *CoRR*, abs/1610.07902, 2016.

[2] Scott Beamer, Aydin Buluç, Krste Asanovic, and David A. Patterson. Distributed memory breadth-first search revisited: Enabling bottom-up search. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, Cambridge, MA, USA, May 20-24, 2013*, pages 1618–1627, 2013.

[3] Aydin Buluç and Kamesh Madduri. Parallel Breadth-first Search on Distributed Memory Systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 65:1–65:12, 2011.

[a] https://github.com/jshun/ligra/tree/master/utils
[b] https://www.cs.rpi.edu/ slotag/classes/FA17/index.html
[c] https://docs.google.com/spreadsheets/d/13J8_pULHLmtK7SqpX5j-i7Ii5ciab-2bUwnd5WIQqqc/edit?usp=sharing

[4] Paul Burkhardt. Optimal Algebraic Breadth-First Search for Sparse Graphs. *CoRR*, abs/1906.03113, 2019.

[5] Hillel Gazit and Gary L. Miller. An improved parallel algorithm that computes the bfs numbering of a directed graph. *Inf. Process. Lett.*, 28(2):61–65, June 1988.

[6] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pages 197–208, 2007.

[7] Jeremy Kepner and John R. Gilbert, editors. *Graph Algorithms in the Language of Linear Algebra*, volume 22 of *Software, environments, tools*. SIAM, 2011.

[8] Ion Stoica, Dawn Song, Raluca Ada Popa, David A. Patterson, Michael W. Mahoney, Randy H. Katz, Anthony D. Joseph, Michael I. Jordan, Joseph M. Hellerstein, Joseph E. Gonzalez, Ken Goldberg, Ali Ghodsi, David Culler, and Pieter Abbeel. A berkeley view of systems challenges for AI. *CoRR*, abs/1712.05855, 2017.

[9] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2007.

[10] Carl Yang, Aydin Buluç, and John D. Owens. Implementing Push-Pull Efficiently in GraphBLAS. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pages 89:1–89:11, 2018.

[11] Xintian Yang, Srinivasan Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus: Implications for graph mining. *Proc. VLDB Endow.*, 4(4):231–242, January 2011.