Proceedings of the 2nd International Conference on Software Engineering and Machine Learning
DOI: 10.54254/2755-2721/74/20240443

The influence of Distributed BFS algorithms in reducing the computational complexity

Jinge Wang

Computing Science, Macao Polytechnic University, Macao, 999078, China

yinci8541740706@126.com

Abstract. Breadth-first search (BFS) is a cornerstone of algorithmic algorithms for complex graphs. In the worst case, BFS's linear complexity is determined by the number of edges and vertices, yet a recently discovered bottom-up method can reduce this complexity to the number of vertices in the best-case scenario, which is usually much less than the number of edges. The conventional top-down approach, however, always takes the same amount of time as the lowest case. To determine the direction, directional algorithms combine top-down and bottom-up methods, adapting between them as the boundary expands. Although bottom-up methods are not always the best option, their use in combination with top-down methods produces efficient algorithms. The main focus of this research is to improve search efficiency and reduce complexity by implementing a version of top-down algorithm optimisation. A scalable distributed-memory parallelisation technique is presented in this study, which has achieved speedups that are ten times faster than the previous purely toplevel approach. In previous studies, it has been demonstrated that this technique is superior to a 1D decomposition method.

Keywords: Distributed BFS algorithms, computational complexity, complex graph.

1. Introduction

Exploring a tree or graph is accomplished by the Depth-first search algorithm.DFS can be applied to detect and eliminate duplicate content to improve resource utilisation and reduce duplication rates.

As an example, in a search engine, depth-first search (DFS) can be applied to crawling and indexing web pages. When a search engine crawler visits a web page, it applies depth-first search to traverse all the links on the page and add them to the queue to be crawled. In this way, the crawler avoids revisiting the same web page repeatedly, thus saving resources [1-2].

In addition, depth-first search (DFS) can also be used for text de-duplication. For example, when dealing with large amounts of text data, repeated paragraphs or sentences may be encountered. At this time, the DFS approach can be used to detect and eliminate these duplicates. Specifically, each paragraph or sentence can be considered as a node and DFS can be used to traverse all possible paths. If duplicate content is found on a path, it can be removed.Depth First Search (DFS) is widely used in reducing duplicates and saving resources. It can assist us to increase efficiency when dealing with large amounts of data and at the same time reduce unnecessary duplication of effort.

A technique for navigating graphs in a variety of graph algorithms, Breadth-First Search (BFS), is a widely employed approach. Due to its sequential nature, depth-first search (DFS) is becoming less important in the context of parallel graph algorithms. As a result, BFS is increasingly being favoured

DOI: 10.54254/2755-2721/74/20240443

as an alternative. This is particularly true for fast parallel graph algorithms. Even when DFS is considered to be the most efficient sequential algorithm for certain tasks, such as finding closely connected components. Exploring the graph |(G)| systematically, BFS is able to identify all vertices accessible from a given source vertex |(s)|. In the worst-case scenario, however, BFS must traverse each edge within the connected part to reach every vertex. Therefore, the most advantageous approach for worst-case analysis is to carry out a level-synchronous traversal with each outgoing edge of the current frontier (the set of vertices found at the actual level) being inspected.

Uncovering various similarities in small-world graphs with low diameter [3], the level-synchronous method reveals many real-world characteristics such as those of social interactions and brain structure [4].The top-down algorithm, which is overly cautious and inefficient in practical situations due to its maximum number of operations, assuming the worst case, can be seen as such: let |(v)| be a node that is |(d)| hops away from the source, and can be reached by |(x)| nodes, where the distance between d-1 and x'| is equal to x|. In plain words, any of the |(x')| vertices has the potential to be the parent of |(v)|. Although there should be only one examination of the |(x')| incoming edges of |(v)|, the top-down algorithm makes |(x'-1)| additional checks, not being able to exploit this knowledge. However, if a significant portion of the neighbours of |(v)| are reachable in |(d-1)| hops, the situation is different. By examining its incoming edges, the source |(v)| can swiftly locate a parent through hops. By incorporating a bottom-up search approach, the BFS algorithm [5] drastically improves on the top-down method by reducing the number of edge examinations. However, implementing this approach on distributed memory presents several difficulties. To begin with, the bottom-up method necessitates swift tests to ascertain the boundary's membership in order to locate a neighbour. However, the size of the frontier exceeds the memory capacity of each processor. Second, once a parent is found, the search for parents for each vertex must be sequential. This is to avoid checking unnecessary edges. Parallelising the search for a vertex's parents completely may lead to redundant work that counteracts the performance advantages, as it may not be possible to complete the search as soon as a parent is found. We modify the two-dimensional graph partitioning method to tackle the first challenge, reducing the amount of boundary that necessitates local replication for each processor. The second obstacle is addressed by utilising sycyclic shifts, which achieve an optimal balance between parallelism and work. This research presents a technique for bottom-up search in distributed memory that is paralleled.

2. Breadth-First Search

We analyse the sequential flavours of both top-down and bottom-up breadth-first search (BFS) algorithms prior to elucidating how we applied our parallel methodology. Algorithm 1 demonstrates that a sequential queue can be used to achieve top-down BFS with level synchronisation. This algorithm keeps track of the parent-child relationship for each vertex, resulting in a " width first spanning tree " starting from the starting vertex, denoted by s. The potential for a vertex v, d-1 hops away from its root, to be the parent of another vertex v, d hops away, is present. The time complexity of this algorithm is directly proportional to (n+m), where n is the number of vertices and m is the number of edges in a graph G=(V,E). The performance of this algorithm remains constant regardless of the best- or worst-case scenario, as it inspects each connected component from the starting point.The performance of this algorithm is the same for both the best- and worst-case scenarios, as it examines every connected component from the initial search point. The valuable insight gained from the basic approach is that many edge examinations are unnecessary due to endpoints already visited.The bottom-up strategy, as Algorithm 2 states, shifts the responsibility of exploration from parents to children, instead of the traditional approach of each vertex in the frontier exploring all its neighbours at each step, identifying those unvisited as children and then adding them to the frontier. Each unvisited vertex (parent[u]=1) then investigates its neighbours to determine if any of them are part of the frontier at each step. If so, the examination of the neighbours (lines 6 through 10) can terminate early since they are valid parents. The early termination function stops as soon as a valid parent is found, making the inner loop more efficient by running it sequentially. Generally speaking,

DOI: 10.54254/2755-2721/74/20240443

the bottom-up method works best when the frontier makes up a significant portion of the arc. At its peak, a high-performance BFS employs the bottom-up approach for the middle stages, while the top-down method is employed at the outset and conclusion of the search. This approach will result in a BFS that is optimised in terms of direction, with each step executing the BFS in the direction that requires the least amount of effort.

3. Analyse

3.1. Sequential top-down BFS algorithm

Algorithm 1 Sequential top-down BFS algorithm

```
    function bfs(graph, start) {
    let visited = new Set();
    let queue = [start];

    while (queue.length > 0) {
      let node = queue.shift();
      if (!visited.has(node)) {
        console.log(node);
        visited.add(node);
        let neighbors = graph[node];
        for (let neighbor of neighbors) {
          if (!visited.has(neighbor)) {
            queue.push(neighbor);
          }
        }
      }
    }
  }

  let graph = {
    A: ['B', 'C'],
    B: ['A', 'D', 'E'],
    C: ['A', 'F'],
    D: ['B'],
    E: ['B', 'F'],
    F: ['C', 'E'],
  };

  bfs(graph, 'A');
```

3.2. Sequential bottom-up BFS algorithm

Algorithm 2 Sequential bottom-up BFS algorithm

```
  function bfs(graph, start) {
    const visited = new Set();
    const queue = [start];

    while (queue.length > 0) {
      const currentNode = queue.shift();
      if (!visited.has(currentNode)) {
        console.log(currentNode);
        visited.add(currentNode);
```

DOI: 10.54254/2755-2721/74/20240443

```
        const neighbors = graph[currentNode];
        for (const neighbor of neighbors) {
          if (!visited.has(neighbor)) {
            queue.push(neighbor);
          }
        }
      }
    }
  }

  const graph = {
    A: ['B', 'C'],
    B: ['A', 'D', 'E'],
    C: ['A', 'F'],
    D: ['B'],
    E: ['B', 'F'],
    F: ['C', 'E'],
  };

  bfs(graph, 'A');
```

Algorithm 3 presents the top-down BFS algorithm with 2D partition, using sparse vectors to implement f and t. The portion of the vector belonging exclusively to the P(i, j)th processor and having no duplicates is denoted as vij, and the portion of the vector belonging to all processors in the ith processor row P(i, :) (replicated) is denoted as vi. Similarly, the part of the vector common to all processors in the ith row P(i, :) (replicated) is denoted vi.

The algorithm consists of four key steps.

A collective allgather operation across column of processors is employed to create the present boundary of vertices on each processor (line 6).Local exploration: Investigate the neighbouring areas of the vertices and merge them within a limited framework (line 8). In essence, this process involves multiplying a sparse matrix by a sparse vector in a specific mathematical structure. In this construction, the minimum value is obtained by each scalar addition, while the second value is obtained by each scalar multiplication.

Fold: By implementing a collective All-to-All operation across the processor row, the recently discovered adjacencies (line 11) are swapped. In this stage, the first pair item (the vertex id that wished to be found) is used as a key to potentially combine updates from different processors targeting the same vertex.

Local update: Update the distances a well as the parents of unvisited vertices (line 12). The remaining entries from the candidate parents are used to create the new frontier.

The 2D algorithm, in contrast to the 1D one, communicates with one processor dimension at a time. If Expansion takes place on one processor, Folding will take place on the other. By employing in-node multithreading, both 1D and 2D algorithms can be advanced by permitting one MPI process per chip instead of per core, thus decreasing the amount of communication participants. Extensive studies comparing the 1D and 2D approaches show that, with or without in-node multithreading, the communication costs of evaluating the 2D method are lower than those of the corresponding 1D method [6]. Additionally, the study demonstrates that in-node multithreading enhances performance beyond that by reducing network contention.

Implementing a bottom-up BFS on a cluster with distributed memory presents several difficulties that are absent in the shared memory situation. The speed increase of the scheme relies on the sequential execution of the inner loop and the execution of fast bound membership tests. A bitmap that can often be stored in the final cache tier is a viable option for executing these membership tests for the frontier on a single compute node. However, for optimal performance in a distributed implementation, it is essential

DOI: 10.54254/2755-2721/74/20240443

to have swift frontier membership checks that don't necessitate traversing the network to determine if a vertex is in the frontier. A 2D decomposition is highly advantageous in representing a selection of vertices that can be the originators of edges to a processor, as storing the entire frontier in the memory of a CPU is not feasible.This subset can be represented using a dense vector for easy access, and it is small enough to fit in the processor's memory [6-7]. Additionally, by compressing the dense format using a bitmap and considering that the frontier tends to occupy a significant portion of the graph during the bottom-up steps, it is possible to avoid using more memory than a sparse vector. However, the 2D decomposition also makes it more challenging to sequentially execute the inner loop.

The parallel analysis of a vertex's nestings is due to the fact that the edges of each vertex are distributed across multiple processors. The advantage of the bottom-up route is that it effectively leads to the end.

4. Conclusion

This study presents a new approach to distributed memory parallel breadth-first search (BFS) direction-optimisation algorithms. It is a combination of an innovative bottom-up search step with a scalable two-dimensional method. The algorithm is evaluated extensively on both artificial and real graphs of varying sizes and degrees using tens of thousands of cores. The new technique significantly outperforms an existing scalable top-down algorithm in data distribution and parallelism management.The top-down approach yields comparable performance to the direction-optimized method, yet with a magnitude of processors that are far fewer.The focus of future work will be on the further improvement and fine-tuning of the algorithm.

References

[1] Tarjan, R. E. (1972). "Depth-first search and linear graph algorithms." SIAM Journal on Computing, vol. 1, no. 2, pp. 146-160.

[2] McLendon III, W., Hendrickson, B., Plimpton, S. J., & Rauchwerger, L. (2005). "Finding strongly connected components in distributed graphs." Journal Parallel Distributed Computing, vol. 65, no. 8, pp. 901-910.

[3] Watts, D. J. (1999). "Networks, dynamics and the small-world phenomenon." American Journal of Sociology, vol. 105, no. 2, pp. 493-527.

[4] Bassett, D. S., & Bullmore, E. (2006). "Small-world brain networks." The Neuroscientist, vol. 12, no. 6, pp. 512-23.

[5] Beamer, S., Asanovic, K., & Patterson, D. A. (2012). "Direction-optimizing breadth-first search." Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC).

[6] Buluç, A., & Madduri, K. (2011). "Parallel breadth-first search on distributed memory systems." International Conference for High Performance Computing, Networking, Storage and Analysis (SC).

[7] Yoo, A., Chow, E., Henderson, K., McLendon, W., Hendrickson, B., & Catalyurek, U. V. (2005). "A scalable distributed parallel breadth-first search algorithm on BlueGene/L." Conference on High Performance Computing (SC).