PARALLEL IMPLEMENTATION OF BREADTH-FIRST SEARCH

Yauhen Klimiankou, Lukas Strebel, Stephanie Christ

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

1. INTRODUCTION

A graph is one of the most powerful and widely used abstract data types, because it is convenient for the representation of a wide range of real-world objects in computer applications. Most graph applications and appropriate algorithms involve graph traversals, during which knowledge about the graph is updated as each vertex is visited. Depth-first search (DFS) and Breadth-first search (BFS) are two basic strategies for graph traversal and searching in graphs. While DFS starts the graph traversal at the root and explores as far as possible along each branch before backtracking, BFS in contrast inspects all the neighbouring vertices starting from the root and then for each of those neighbour vertices in turn, it inspects their neighbour vertices which were unvisited, and so on, level by level. The set of vertices with the same minimal distance from the root vertex is called a level in terms of BFS.

BFS is one of the most basic graph algorithms and a foundation for a wide range of more specific algorithms used for tasks such as finding all vertices within one connected component of the graph, collection copying in garbage collection algorithms, finding the shortest path between two specified vertices, testing a graph for bipartiteness, mesh numbering, computation of the maximum flow in a flow network etc. Most notable examples of its industrial applications are navigation systems for finding the shortest path between two specified destination points on a road network, finding the shortest route to a specified host in a computer network or the shortest route to a host with specified properties, web indexing performed by web crawlers used by web search engines to maintain their search database in the actual state and social interconnections investigation on the base of social networks.

At this time, the information technology industry is experiencing a great shift introduced by mass migration to multi-core processors and emergence of many-core computer systems (up to 120/240 physical/logical cores in theory and 60/120 physical/logical cores in practice on Intel platform, and up to 64 physical cores on AMD platform) and coprocessors (computation accelerators) like Intel Xeon Phi (up to 61 physical cores). All these wide-spreading and emerging computer systems are examples of shared memory architectures (SMA). Wide dissemination of computer systems with shared memory architecture and trends indicating that the development of such systems is the main direction of the further performance improvement of computer systems on one hand, and the widespread use of BFS for a wide range of applications on the other hand, make the question about the most efficient and scalable variant of parallel version of BFS in the environment of such kind of systems relevant to this time.

In this paper, we describe an experimental study of different variants and approaches of parallel BFS algorithms based on OpenMP for SMA computer systems and the design, implementation and evaluation of the variant with which we finally came up as the most promising. We refer to this variant as optimistic parallel BFS. According to our experimental results, it significantly outperforms all other approaches in the environments of AMD SMA platform and Intel Xeon Phi accelerator. Our experiments and analysis of its results highlighted that the efficiency of BFS algorithm variants heavily depend on the properties of the underlying environment (hardware platform, operating system, compiler), as well as on the properties of the graph to which it is applied. We conclude also that the cost of synchronization which is usually used for preserving consistency can be too high, but can in some cases be eliminated.

It is important to note that the optimistic parallel BFS can't be considered as a universal optimal variant of BFS for all kinds of the SMA computer systems. Instead it must be considered as a source of a more general approach to the implementation of BFS for the environment of a particular kind of SMA system. To achieve the most optimal approach for a particular computer system, investigation of different variants of parallel BFS and the most suitable optimizations must be done for the environment of the system of interest, because our results tell us that the efficiency of BFS itself and optimizations used in it heavily depend on the target environment.

2. BACKGROUND: BREADTH-FIRST SEARCH

In this section, we will give a brief overview over the idea of breadth-first search and its sequential asymptotic runtime cost, as well as a short discussion of different graphs and graph properties.

Breadth-first search. Breadth-first search (BFS) is a graph traversal algorithm which starts at a source vertex and either travels until it finds a specific vertex or until it has explored all connected vertices. In the first step, all vertices adjacent to the source are explored and stored in some data structure (called frontier or next) as well as marked as visited. In the second step, the newly visited vertices become the new sources (called neighbours or current) from which the search continues by repeating this step. Doing a traversal this way assures that all the nodes at the same distance to the source are explored before any vertices with greater distance can be explored. All the vertices in the same distance to the source are called level.

The desired output of a BFS can differ depending on where it is applied. With minimal modifications BFS could deliver a predecessor map, where every vertex points to only one parent, or a distance map, where the distance to the source is stored for every vertex. Since the predecessor map is not necessarily unique, we choose to return a distance map as our output to make verification of correctness simpler.

The sequential version of a BFS can be implemented using a single queue and has a theoretical asymptotic runtime of $\mathcal{O}(|V| + |E|)$, where $|V|$ is the number of vertices and $|E|$ the number of edges in the connected graph.

Graphs. As previously mentioned graphs are a widely used abstract data type, which depending on their source and use can have widely different properties which in turn account for strong variations in the performance of BFS and other graph algorithms.

To account for this we used real world graphs and specific synthetic graphs provided by the University of Florida Sparse Matrix Collection [1], notably some graphs from the DIMACS10 challenge, in our test and experimental cases. The mentioned synthetic graphs are mostly so called Kronecker graphs [2], which are generative network graphs that obey the main static network patterns observed in real networks. Especially the small-world property is worth mentioning since it provides a reasonable assumption that a given node in a network has a large enough neighbourhood so that a parallel BFS can be useful.

3. RELATED WORK

BFS is not an easy candidate for parallelisation. It is inherently memory intensive and has pure spacial locality which introduces significant performance loss on today's computer systems with the growing gap between CPU performance, memory performance and increasing memory latencies. As a result, scalability of a parallel version of BFS will be bound by the performance characteristics of the memory subsystem of the target computer system. Nevertheless, intensive use of BFS in the wide range of the applications create a high interest in the most efficient and scalable versions of parallel BFS. As a result, a number of papers were published in this field.

Beamer et al.[3] report a different algorithm to deal with the performance issues encountered when designing a BFS algorithm. The proposed hybrid algorithm combines the usual top-down approach with a new bottom-up part. In the bottom-up part, a level is processed by searching a parent for all unvisited vertices, where a parent is only valid if it is a neighbour of the unvisited vertex. This is advantageous for small-world graphs because it saves accesses and data processing when a large fraction of the vertices are in the frontier. To get optimal results, a hybrid algorithm is proposed where a heuristic switching criteria controls the use of top-down or bottom-up step depending on the size of the frontier and a predicted size of the next frontier. Yasui et al.[4] describe an implementation of such a hybrid algorithm for Kronecker and R-MAT graphs, as well as a detailed description of the heuristic switching parameters.

Berrendorf[5] describes a technique to avoid atomic operations in a generalized scenario. The scenario is given as an if-statement followed by some operations that change a state, where multiple threads might execute the predicate and execute the operations afterwards. The operations need to change the state to the same value if executed multiple times, otherwise there exists a race condition, i.e. the change of the distance of a visited vertex to the value of the level or the addition of a vertex to the next frontier. The trade-off is that doing a BFS this way can result in additional work, since any unvisited vertex may get added multiple times.

In our final proposed technique we go the way of avoiding atomic operations and synchronization within levels at all. The only synchronization point left in our proposed algorithm is an implicit barrier between the processing of different levels of the graph.

In addition to this, we have compared a number of different approaches based on different synchronization primitives and techniques and load balancing strategies.

4. DESIGN AND IMPLEMENTATION

The goal of our efforts is to achieve the most efficient and scalable algorithm of parallel BFS for Shared Memory Architectures.

4.1. Possible directions of performance boosting

There are three main directions of performance boosting in the way of making BFS parallel.

Improvement of cache utilization. It is not a secret that due to the big gap between the performance of CPU subsystems and the memory subsystem performance of modern computer systems, as well as latencies of memory access, make the performance of the algorithms highly sensitive to the provided CPU cache utilization. Introduction of multicore computer systems made the problem even more complex because we are now forced to think about careful splitting of the active data set between all available processors to avoid cache misses introduced by cache lines flip-flopping. Unfortunately, in the case of the BFS, we are forced to work with a data set with inherently pure spatial locality. Due to this, this way of optimization was rejected.

Load balancing improvement. Goal of this approach is to achieve highest possible level of overall utilization of all available processor elements by avoiding the idle time of CPUs introduced by waiting for the arrival of a new job. It is one of the promising approaches, because the classical approach to BFS parallelisation relies on a sequential graph level processing where each level is processed in parallel. As a result, all CPUs that finished their work are forced to wait until the processing of the current level will be accomplished by other still busy processors. Nonetheless, this direction was rejected in favour of the synchronization avoidance.

Avoidance of synchronization. Synchronization is an expensive but necessary component of almost all parallel algorithms. It is expensive in both dimensions, as synchronization usually takes many CPU cycles (in absolute numbers), which reduces scalability. Our main design direction was figuring out of the cheapest scheme of synchronization while still providing consistency. Ideally, we would like to eliminate synchronization at all.

4.2. Different approaches

We implemented many different algorithms and multiple variants for most of them. They return a distance map from one source vertex to all reachable vertices in the graph.

All our implementations are based on OpenMP for synchronization.

Top-down. Most of our approaches are based on a simple top-down algorithm. The idea is to do a level-synchronous traversal of the graph by keeping record of the vertices in the current as well as those in the next level ("frontier" and "neighbour") in two data structures of the same type. We then set the frontier to the current neighbours after each level. This leads to an implicit barrier, as the threads that finish earlier have to wait on the others before proceeding to the next level. We experimented with different data structures and synchronization methods.

The naive variant of the top-down algorithm uses a global standard vector for the frontier to allow easy dynamic splitting between the threads in each level. This balances the load between threads, however it has a significant overhead because it relies on a critical section (`OMP critical`) for checking whether the vertices were already visited and subsequently inserting them into the neighbour data structure.

To improve the naive implementation, the first approach to get rid of the critical section as it produced a lot of overhead. We attempted this by using an atomic, the built-in `__sync_val_compare_and_swap` (CAS), to atomically check whether a vertex was visited and setting the correct distance. In addition to the atomic, this variant uses a local neighbourhood data structure (standard vector) to be able to do without a critical section. Only at the end of each level a lock (`omp_lock_t`) is needed to combine the local neighbourhoods to a global one, which can then be distributed between the threads for the next level. This is an idea adapted from Berrendorf [5].

An extended version of the algorithm before does a non-atomic check if a vertex visited before each CAS. This lets us treat the cases where the vertex had already been visited without any synchronization.

To compare all synchronisation approaches we also looked into concurrent data structures to solve the synchronisation problem of the algorithm i.e. a concurrent queue to handle the neighbour and frontier queue. The design and implementation of such a data structure is a subject of its own and beyond the scope of this paper. Therefore instead of focusing on designing or implementing such a concurrent queue we used the one implemented in the Intel Threading Building Blocks library (Intel TBB version 4.3). We found that using two concurrent queues for the neighbour and frontier and swapping them between levels in the same way as in the above mentioned implementations diminishes the time saved by not using explicit synchronisation significantly. To perform better we implemented a version that uses two concurrent queues that instead of swapping just switch roles from frontier to neighbour and vice versa after each level. This way the swapping in the serial part of the implementation is replaced by a few if-statements while essentially doing the same algorithm.

4.3. Optimistic BFS algorithm design

Our proposed algorithm was designed for the environment of OpenMP and due to this a employs data-parallelism approach. It accepts two input parameters: the graph description and the index of the root node. The graph description is a list of lists, in which each top-level list denotes one of the graph vertices and the bottom level list enumerates all neighbours of the appropriate vertex, thus denoting all

edges connected to it. The algorithm returns a distance map for the specified root node.

The core of the proposed algorithm is classical. It is a level by level sequential top-down walking through the graph, where in each iteration we discover all unvisited neighbours of the vertices in the current level.

The algorithm utilizes $4 \cdot number\_of\_vertices$ bytes of memory allocated in 4 equal chunks:

1. x

Algorithm 1 Optimistic BFS

Input:
Adjacency list: $AF = \{AF_k\}$
Source node: s
Reference to distance map: $distance = \{distance_k\}$

```
1:  n ← size (AF)                                   ▷ Number of vertices
2:  currLevel_k ← False, ∀k ∈ n
3:  nextLevel_k ← False, ∀k ∈ n
4:  visited_k ← False, ∀k ∈ n

5:  currLevel_s ← True
6:  visited_s ← False
7:  distance_s ← 0

8:  stop ← False

9:  while stop = False do
10:     stop ← True
11:     for all v ∈ V do in parallel
12:         if currLevel_v = True then
13:             currLevel_v ← False
14:             for all w ∈ AF_v do
15:                 if visited_w = False then
16:                     distance_w ← distance_v + 1
17:                     nextLevel_w ← True
18:                     visited_w ← True
19:                     stop ← False
20:     swap (currLevel, nextLevel)
21:     nextLevel_k ← False, ∀k ∈ n
```

5. EXPERIMENTAL RESULTS

Experimental setup. We run experiments on three different platforms.

The first platform is the EULER cluster which is operated by the HPC Group of ETH. We had access to one node with a 12-core Intel Xeon E5-2697v2 processors (2.7 GHz nominal, 3.0-3.5 GHz peak). As it supports hyperthreading, we ran our algorithms with up to 24 threads. On this platform, we used the gcc compiler with the -O2 flag.

We also ran our algorithms on an Intel Xeon Phi with 61 cores. On this platform, we used the Intel icc compiler, with the -O2 flag as well.

The last platform is an AMD FX-8350 (4GHz x8, 8Gb, W2k3).

Results.

6. CONCLUSIONS

7. REFERENCES

[1] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. Issue 1, pp. 1:1 – 1:25, 2011.

[2] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani, "Kronecker graphs: An approach to modeling networks," *J. Mach. Learn. Res.*, vol. 11, pp. 985–1042, Mar. 2010.

[3] Scott Beamer, Krste Asanovi, and David A Patterson, "Searching for a parent instead of fighting over children: A fast breadth-first search implementation for graph500," 2011.

[4] Y. Yasui, K. Fujisawa, and K. Goto, "Numa-optimized parallel breadth-first search on multicore single-node system," in *Big Data, 2013 IEEE International Conference on*, Oct 2013, pp. 394–402.

[5] R. Berrendorf, "A technique to avoid atomic operations on large shared memory parallel systems," *International Journal on Advances in Software*, vol. 7, no. 1 & 2, pp. 197–210, 2014.