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. Optimistic BFS algorithm design

Our proposed algorithm was designed for the environment of OpenMP and due to this 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.

One of the core design decision related to memory management. Classical approach employs dynamic data structures like lists and queues which are used for level member nodes. This approach is purely suitable for efficient parallelism, because it explicitly introduce synchronization point and can introduce implicit synchronization point via memory chunks allocation/deallocation. Instead we propose to use raw memory chunks allocated and freed only once in the prologue and epilogue of the algorithm. Our approach utilize $4 \cdot number\_of\_the\_node$ bytes of memory allocated in 4 equal chunks:

1. Flags array tracking visited nodes.
2. Flags array holding all nodes of the current level.
3. Flags array used for tracking of the next level nodes.
4. Flags array used for resetting.

After the end of each level processing the last three flag arrays exchange they roles by simple and efficient pointers swapping in accordance with the next rotation scheme: current level $\rightarrow$ resetting $\rightarrow$ next level $\rightarrow$ current level. As a result each level processing started with set of current level nodes collected during previous level processing and clear map of next level nodes.

Byte per node memory allocation scheme was chosen instead of classical bit masks because it allows to simplify memory access patterns and eliminate synchronization which would be required otherwise. Bit mask update via OR operation includes two elementary memory access operations, because this operation is an example of read-modify-write operation. As a result it introduces source of inconsistency in the concurrent environment. This consistency can be easily fixed by using of the lock prefix (atomic operations on Linux and interlocked operations on Windows), but this will mean one more point of synchronization which we are trying to eliminate.

Level processing performs in parallel using for cycle of the OpenMP. This cycle introduce the only synchronization point of the proposed algorithm via implicit barrier at the end of cycle.

Each iteration of the parallel for cycle containst two conditional actions: making a step on the graph walking and clearing of the clearing flags array. Last one action is made only by threads processing the node at the start of the cache line and clear whole cache line at once. This policy reduces the level of false cache sharing. Stepping through the graph performs only for nodes specified by the flags array of the current level of the walking. For such nodes algorithm algorithm updates the distance map, flags array of visited nodes and flags array for the next level of all unvisited neighbor nodes .

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. EXPERIMENTS

We present experimental results to demonstrate the performance gains that can be realized with Optimistic BFS. There are three experimental testbeds used in the experiments. In all cases scalability was measurements was limited by the number of physical processing elements present in the test environment (including processing elements delivered by Hyper-Threading technology). Testbeds used includes:

- **Euler**. Intel Xeon E5-2697v2 processor (2.7 GHz nominal, 3.0-3.5 GHz peak, HT enabled) with 12 physical and 24 logical cores. GNU GCC compiler with -O2 flag.
- **Einstein**. Intel Xeon Phi Coprocessor 7120 (1.238 GHz base frequency, 1.333 GHz max turbo frequency) with 61 physical cores. 16 Gb of memory. Intel ICC compiler Version 15.0.0.090 Build 20140723 (with -O2 flag). Running on Linux version 2.6.32-431.el6.x86.64.
- **AMD**. AMD FX-8350 (4GHz nominal, 4.2GHz peak) with 8 physical cores. 8 Gb of memory. Microsoft Windows Server 2003 with PAE enabled. Microsoft Visual Studio 2008 Professional with Microsoft C++ compiler 9.00.30729.01 with full optimization enabled.

The experiments include two different graphs:

- **Million**. A huge real world graph includes 1 million nodes and ∼3 million edges (weakly connected graph).
- **DIMACSKRON**. A Kronecker graph used in the DIMACS10 Challenge. Consists of ∼500k nodes and ∼21 million edges.

5.1. Different approaches

We implemented many different algorithms and multiple variants for most of them. All of them return a distance map from one source vertex to all reachable vertices in the graph. All our implementations are based on OpenMP for synchronization. Most of our approaches are based on a simple top-down algorithm.

Top-down naive. 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.

Top-down CAS. 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].

Top-down if-CAS. 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.

Top-down non-atomic.

Doing the visited check from the previous versions in a non atomic way adds additional work but no harmful race condition. This implementation uses a local neighbourhood and (`omp_lock_t`) to add them together, which means it still has to use synchronisation in constrast to the optimistic approach.

Non-level synchronous. Very different approach to the BFS problem. Ignoring the implicit level barrier allows threads to manage their own queues without synchronisation but adds the need to check if any visited node was visited from a different thread in a later level.

TBB Concurrent Queue. To compare all synchronisation approaches we also looked into concurrent data structures to solve the synchronisation problem of the algorithm. 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 ourselves we used the one implemented in the Intel Threading Building Blocks library (Intel TBB version 4.3). Our implementation uses two concurrent queues that switch roles from frontier to neighbour and vice versa after each level to avoid having a costly swap operation in the sequential part between levels.

Optimistic. The optimistic version refers to our best implementation described in detail in section 4 ”Design and implementation”. Optimistic modified refers to a different way of initializing and managing the flag arrays.

5.2. Experimental results

In the following plots resulting from our experiments not all version or approaches mentioned before appear on each of the test systems. The reason behind this is that some are too slow on one test environment while doing okay on an other one. Versions omitted from a plot can therefore be assumed slower then any of the shown algorithms.

In the experiments for every number of threads multiple BFS started from different source nodes are averaged. These measured average times as well as their standard deviation is shown in each plot. The goal of these experiments

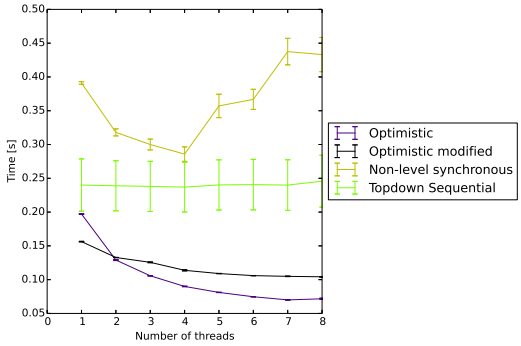

Fig. 1. Performance of our BFS implementations on the AMD test environment operating on the Million graph.

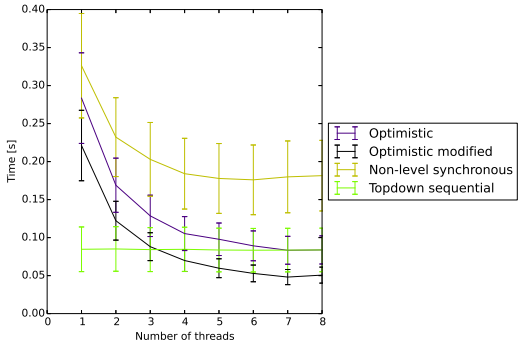

Fig. 2. Performance of our BFS implementations on AMD operating on the DIMACSKRON graph.

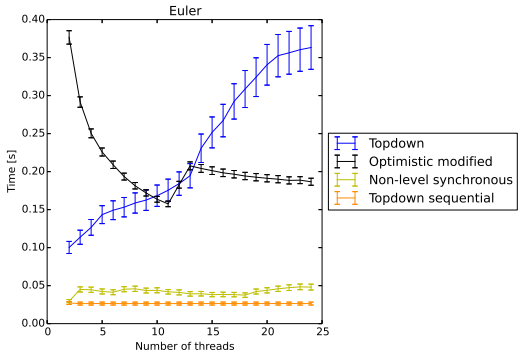

Fig. 3. Performance of our BFS implementations on Euler operating on the Million graph.

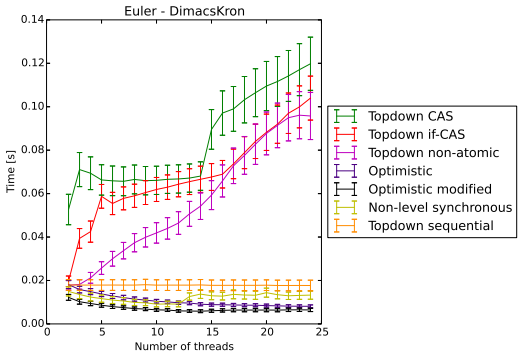

Fig. 4. Performance of our BFS implementations on Euler operating on the DIMACSKRON graph.

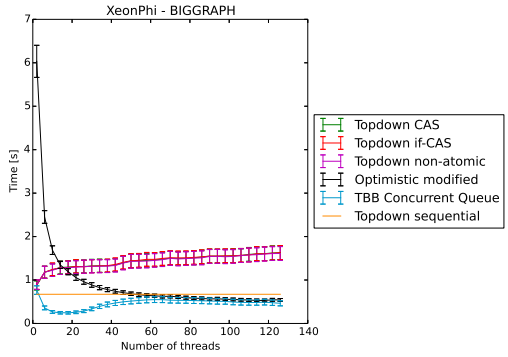

Fig. 5. Performance of our BFS implementations on Einstein operating on the Million graph.

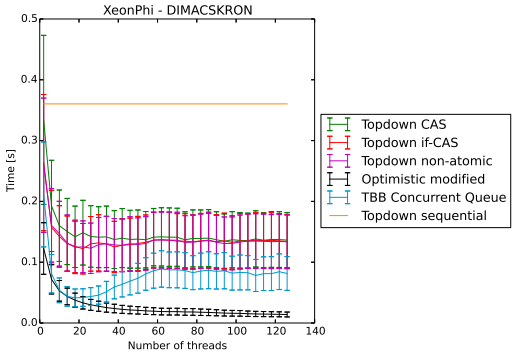

Fig. 6. Performance of our BFS implementations on Einstein operating on the DIMACSKRON graph.

is to find the best performing algorithm and to see how different approaches behave in the case of strong scaling i.e. increasing the number of threads for the same problem size (same graph and same starting nodes).

First we take a look at the AMD system. In Fig. 1 we can observe the irregular and bad performance of the non-level synchronous approach on the AMD system and especially for the Million graph. In constrast the Optimistic versions behave quite nicely and scale in an expected way. Running the Kronecker graph on the same system results in quite different results shown in Fig. 2. While the general behaviour of all shown algorithms seems okay, the cost of doing the BFS in parallel actually makes even the best version slower than the sequential solution up to 3 threads. With more than 3 threads we can see some speedup.

Switching to the Euler system with a higher number of maximal threads we observed quite surprising results in Fig. 3. The biggest surprise is that for the Million graph none of the implemented algorithms manages to be better than a simple sequential run. Reasons for this might be the small number of edges in this graph limiting the potential for parallel execution to a point where the cost for doing any synchronisation or the additional work created to circumvent synchronisation on this system is too great.

Supporting this theory is the fact that in Fig. 4 we can see that for a more connected graph a few algorithms are an improvement in comparison to a sequential run. The main result from this experiment is the observation that any version that relies on `OMP critical`, even if only to manipulate the used data structure, does not perform in a useful way. As a minor detail the jump in the Non-level synchronous version in this plot or more clearly the jump for the Optimistic modified version in Fig. 3 between 12 and 13 threads can be explained as the switch from physical cores to hyperthreading.

Lastly on the Einstein system where the Intel TBB library was available we can compare the performance of a concurrent datastructure approach with our Optimistic algorithm. In Fig. 5 we can notice that on the Einstein system it requires a lot more threads for the Optimistic approach to become useful. On the otherhand the TBB concurrent queue performs well for a limited number of threads after which it loses performance with increasing number of threads.

The same observation though in a less extreme way can be made for the Kronecker graph in Fig. 6 where the TBB concurrent queue is much sooner outperformed by the Optimistic algorithm.

6. DISCUSSION

We now provide a analysis of the results of experimental evaluation of Optimistic BFS and other variants of parallel BFS used during our study to achieve additional insight on

the base of the conducted study. We would like to note that the study can't be considered as complete, and additional experiments and analysis need to be conducted to achieve clear understanding of the all properties of the proposed algorithm. Particularly, weak scaling experiments need to be conducted and dependencies of BFS performance on the target environment properties and target graph properties should be evaluated.

Using the results of our experiments, we determined that the performance of the particular BFS implementation heavily depends on two major groups of factors. One of the very general conclusion from the our study is that performance characteristics of the memory intensive parallel algorithms depends heavily on the characteristics of the target environment. By the target environment we mean underlying hardware architecture, operating system and compiler. To understand what component of the environment plays major role in the affection on the algorithm performance characteristics additional experiments need to be conducted. But what we can see is that even on so similar hardware platforms like based on AMD and Intel implementations of IA-32 architecture algorithm can demonstrate significantly different behaviour, what is surprising. What we can expect is that parallel BFS algorithm optimal for the all SMA architectures can not exist at all and to achieve the best results on a particular environment algorithm must be developed and evaluated specially for this particular environment. Simple code reuse may not work if performance is really have matter.

This analysis also highlighted the dependency of the performance characteristics of the parallel variants of BFS on the characteristics of the target graph. It is not clear what exact characteristics of the target graph and how exactly affects the performance of Optimistic BFS. But detailed understanding of these dependencies and knowledges of the characteristics of typical graphs used in particular application can help in development of the most performant algorithm for this particular application.

In general, we can conclude that the most promising ways of the parallel BFS optimisation is optimisation of synchronization used and improvement of load balancing. The best way of synchronization optimisation is its avoidance, because even cheapest synchronization methods like atomic instruction can be too expensive on practice.

7. SUMMARY AND CONCLUSION

In this paper we present the design, implementation, and evaluation of Optimistic BFS, an parallel version of breadth-first search for shared memory architectures.

We demonstrate through experiments that this algorithm and approach on synchronization avoidance in many cases outperforms not only approaches relying on cheap synchronization methods, but also approaches focused on improvement of load balancing. In fact, we observe that the even cheap synchronization methods can be very expensive for performance and scalability of algorithm. Almost complete avoidance of synchronization results in a significant improvement of performance and determinism of BFS.

In our experimental evaluation, we found that the level of optimality of BFS algorithms heavily depends on characteristics of target environment, consisting of CPU architecture, operating system and compiler used, and on characteristics of the target graph. In other words for different target environment and graphs different variants of parallel BFS can be optimal.

In future work, it would be interest to figure out what component of the Euler test environment is a main source of performance degradation of Optimistic BFS and thus find for which set of target environments it would be most optimal. It also would be interesting to figure out the relationships of the performance provided by optimistic BFS depending of different graph characteristics and thus figure out for what class of graphs it would be most optimal. Furthermore, it will be beneficial to evaluate weak scaling properties of Optimistic BFS.

8. 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.