Level-Synchronous Parallel Breadth-First Search Algorithms For Multicore and Multiprocessor Systems

Rudolf Berrendorf and Matthias Makulla
Computer Science Department
Bonn-Rhein-Sieg University
Sankt Augustin, Germany
e-mail: rudolf.berrendorf@h-brs.de, mathias.makulla@h-brs.de

***Abstract*—Breadth-First Search (BFS) is a graph traversal technique used in many applications as a building block, e.g., to systematically explore a search space. For modern multicore processors and as application graphs get larger, well-performing parallel algorithms are favourable. In this paper, we systematically evaluate an important class of parallel BFS algorithms and discuss programming optimization techniques for their implementation. We concentrate our discussion on level-synchronous algorithms for larger multicore and multiprocessor systems. In our results, we show that for small core counts many of these algorithms show rather similar behaviour. But, for large core counts and large graphs, there are considerable differences in performance and scalability influenced by several factors. This paper gives advice, which algorithm should be used under which circumstances.**

***Index Terms*—parallel breadth-first search; BFS; NUMA; memory bandwidth; data locality**

I. Introduction

BFS is a visiting strategy for all vertices of a graph. BFS is most often used as a building block for many other graph algorithms, including shortest paths, connected components, bipartite graphs, maximum flow, and others [1]. Additionally, BFS is used in many application areas where certain application aspects are modelled by a graph that needs to be traversed according to the BFS visiting pattern. Amongst others, exploring state space in model checking, image processing, investigations of social and semantic graphs, machine learning are such application areas [2].

Many parallel BFS algorithms got published (see Section III for a comprehensive overview including references), all with certain scenarios in mind, e.g., large distributed memory systems with the message passing programming model [3], GPU's (Graphic Processing Unit) with a different parallel programming model [4], or randomized algorithms for fast, but possibly sub-optimal results [5]. Such original work often contains performance data for the newly published algorithm on a certain system, but often just for the new approach, or taking only some parameters in the design space into account [6] [7]. To the best of our knowledge, there is no rigid comparison that systematically evaluates relevant parallel BFS algorithms in detail in the design space with respect to parameters that may influence the performance and/or scalability and give advice which algorithm is best suited for which application scenario. In this paper, BFS algorithms of a class with a large practical impact (level-synchronous algorithms for shared memory parallel systems) are systematically compared to each other.

The paper first gives an overview on parallel BFS algorithms and classifies them. Second, and this is the main contribution of the paper, a selection of level-synchronous algorithms relevant for the important class of multicore and multiprocessors systems with shared memory are systematically evaluated with respect to performance and scalability. The results show that there are significant differences between algorithms for certain constellations, mainly influenced by graph properties and the number of processors / cores used. No single algorithm performs best in all situations. We give advice under which circumstances which algorithms are favourable.

The paper is structured as follows. First, a BFS problem definition is given. Section III gives a comprehensive overview on parallel BFS algorithms with an emphasis on level synchronous algorithms for shared memory systems. Section IV prescribes algorithms in detail that are of concern in this paper. Section V describes our experimental setup, and, in section VI, the evaluation results are discussed, followed by a conclusion.

II. Breadth-First Search Graph Traversal

We are interested in undirected graphs $G = (V,E)$, where $V = \{v_1,...,v_n\}$ is a set of vertices and $E = \{e_1,...,e_m\}$ is a set of edges. An edge e is given by an unordered pair $e = (v_i, v_j)$ with $v_i, v_j \in V$. The number of vertices of a graph will be denoted by $|V| = n$ and the number of edges is $|E| = m$.

Assume a connected graph and a source vertex $v_0 \in V$. For each vertex $u \in V$ define $depth(u)$ as the number of edges on the shortest path from v_0 to u, i.e., the edge distance from v_0. With $depth(G)$ we denote the depth of a graph G defined as the maximum depth of any vertex in the graph *relative to the given source vertex*. Please be aware that this may be different to the diameter of a graph, the largest distance between *any* two vertices.

The problem of BFS for a given graph $G = (V,E)$ and a source vertex $v_0 \in V$ is to visit each vertex in a way such that a vertex v_1 must be visited before any vertex v_2 with $depth(v_1) < depth(v_2)$. As a result of a BFS traversal, either the level of each vertex is determined or a (non-unique) BFS spanning tree with a father-linkage of each vertex is created. Both variants can be handled by BFS algorithms with small modifications and without extra computational effort. The

 ISBN: 978-1-61208-339-1

problem can be easily extended and handled with directed or unconnected graphs. A sequential solution to the problem can be found in textbooks based on a queue where all non-visited adjacent vertices of a visited vertex are enqueued [1]. The computational complexity is $O(|V|+|E|)$.

III. Parallel BFS Algorithms and Related Work

We combine in our BFS implementations presented later in Section IV several existing algorithmic approaches and optimization techniques. Therefore, the presentation of related work has to be intermingled with an overview on parallel BFS algorithms itself.

In the design of a parallel BFS algorithm, different challenges might be encountered. As the computational density for BFS is rather low, BFS is memory bandwidth limited for large graphs and therefore bandwidth has to be handled with care. Additionally, memory accesses and work distribution are both irregluar and data-dependent. Therefore, in large NUMA systems (Non-Uniform Memory Access [8]) data layout and memory access should respect processor locality. In multicore multiprocessor systems, things get even more complicated, as several cores share higher level caches and NUMA-node memory, but have distinct and private lower-level caches.

A more general problem for many parallel algorithms including BFS is a sufficient load balance when static partitioning is not sufficient. Even when an appropriate mechanism for load balancing is deployed, graphs might only supply a limited amount of parallelism. This aspect especially affects the popular level-synchronous approaches for parallel BFS we concentrate on later.

In BFS algorithms, housekeeping has to be done on visited / unvisited vertices with several possibilities how to do that. A rough classification of algorithms can be achieved by looking at these strategies. Some of them are based on special container structures where information has to be inserted and deleted. Scalability and administrative overhead of these containers are of interest. Many algorithms can be classified into two groups: *container centric* and *vertex centric* approaches.

A. Container Centric Approaches

The emphasis in this paper is on level-synchronous algorithms where data structures are used, which store the current and the next vertex frontier. Generally speaking, these approaches deploy two identical containers (*current* and *next*) whose roles are swapped at the end of each iteration. Usually, each container is accessed in a concurring manner such that the handling/avoidance of synchronized access becomes crucial. Container centric approaches are eligible for dynamic load balancing but are sensible to data locality on NUMA systems. Container centric approaches for BFS can be found in some parallel graph libraries [9] [10].

For level synchronous approaches, a simple list is a sufficient container. There are approaches, in which each thread manages two private lists to store the vertex frontiers and uses additional lists as buffers for communication [3] [11]. This approach deploys a static one dimensional partitioning of the graph's vertices and therefore supports data locality. But this approach completely neglects load balancing mechanisms. The very reverse would be an algorithm, which focuses on load balancing. This can be achieved by using special lists that allow concurrent access of multiple threads. In contrast to the thread private lists of the previous approach, two global lists are used to store the vertex frontiers. The threads then concurrently work on these lists and implicit load balancing can be achieved. Concurrent lock-free lists can be efficiently implemented with an atomic compare-and-swap operation.

It is possible to combine both previous approaches and create a well optimized method for NUMA architectures [6] [7] (this paper came too late to our knowledge to include it in our evaluation). Furthermore, lists can be utilised to implement container centric approaches on special hardware platforms as graphic accelerators with warp centric programming [4].

Besides strict FIFO (First-In-First-Out) and relaxed list data structures, other specialized containers may be used. A notable example is the *bag* data structure [12], which is optimized for a recursive, task parallel formulation of a parallel BFS algorithm. This data structure allows an elegant, object-oriented implementation with implicit dynamic load balancing, but which regrettably lacks data locality.

B. Vertex Centric Approaches

A vertex centric approach achieves parallelism by assigning a parallel entity (e.g., a thread) to each vertex of the graph. Subsequently, an algorithm repeatedly iterates over all vertices of the graph. As each vertex is mapped to a parallel entity, this iteration can be parallelised. When processing a vertex, its neighbours are inspected and if unvisited, marked as part of the next vertex frontier. The worst case complexity for this approach is $O(n^2)$ for degenerated graphs (e.g., linear lists). This vertex centric approach might work well only, if the graph depth is very low.

A vertex centric approach does not need any additional data structure beside the graph itself and the resulting *level-/father*-array that is often used to keep track of visited vertices. Besides barrier synchronisation at the end of a level iteration, a vertex centric approach does with some care not need any additional synchronisation. The implementation is therefore rather simple and straightforward. The disadvantages of vertex centric approaches are the lacking mechanisms for load balancing and graphs with large depth (e.g., a linear list).

But this overall approach makes it well-suited for GPU's where each thread is mapped to exactly one vertex [13] [14]. This approach can be optimized further by using hierarchical vertex frontiers to utilize the memory hierarchy of a graphic accelerator, and by using hierarchical thread alignment to reduce the overhead caused by frequent kernel restarts [15].

Their linear memory access and the possibility to take care of data locality allow vertex centric approaches to be efficiently implemented on NUMA machines [16]. Combined with a proper partitioning, they are also suitable for distributed systems, as the overhead in communication is rather low.

 ISBN: 978-1-61208-339-1

C. Other Approaches

The discussion in this paper concentrates on level-synchronous parallel BFS algorithms for shared-memory parallelism. There are parallel algorithms published that use different approaches or that are designed for other parallel architectures in mind. In [5], a probabilistic algorithm is shown that finds a BFS tree with high probability and that works in practice well even with high-diameter graphs. Beamer et.al. [17] combines a level-synchronous top-down approach with a vertex-oriented bottom-up approach where a heuristic switches between the two alternatives; this algorithm shows for small world graphs very good performance. Yasui et.al. [18] explores this approach in more detail for multicore systems. In [19], a fast GPU algorithm is introduced that combines fast primive operations like prefix sums available with highly-optimized libraries. A task-based approach for a combination of CPU/ GPU is presented in Munguia et.al. [20].

For distributed memory systems, the partitioning of the graph is crucial. Basically, the two main strategies are one dimensional partitioning of the vertices and two dimensional edge partitioning [3]. The first approach is suited for small distributed and most shared memory systems, while the second one is viable for large distributed systems. Optimizations of these approaches combine threads and processes in a hybrid environment [21] and use asynchronous communication [22] to tolerate communication latencies.

D. Common extensions and optimizations

An optimization applicable to some algorithms is the use of a bitmap to keep track of visited vertices in a previous iteration [6]. The intention is to keep more information on visited vertices in a higher level of the cache hierarchy.

Fine-grained tuning like memory prefetching can be used to tackle latency problems [7] (but which might produce even more pressure on memory bandwidth).

Besides implicit load balancing of some container centric approaches, there exist additional methods. One is based on a logical ring topology [23] of the involved threads. Each thread keeps track of its neighbour's workload and supplies it with additional work, if it should be idle. Another approach to adapt the algorithm to the topology of the graph monitors the size of the next vertex frontier. At the end of an iteration, the number of active threads is adjusted to match the workload of the coming iteration [11].

IV. Evaluated Algorithms

In our evaluation, we used the following parallel algorithms, each representing certain points in the described algorithm design space for shared memory systems, with an emphasis on level-synchronous algorithms:

- `global`: vertex-centric strategy as described in Section III-B, with parallel iterations over *all vertices on each level* [16]. As pointed out already, this will only work for graphs with a very low depth.
- `graph500`: OpenMP reference implementation in the Graph500 benchmark [9] using a single array list with atomic Compare-And-Swap (CAS) and Fetch-And-Add accesses to insert chunks of vertices. Vertex insertion into core-local chunks is done without synchronized accesses. Only the insertion of a full chunk into the global list has to be done in a synchronized manner. All vertices of a full chunk get copied to the global array list.
- `bag`: using OpenMP [24] tasks and two bag containers as described in [12]. This approach implicitly deploys load balancing mechanisms. Additionally, we implemented a Cilk++ version as in the the original paper that didn't perform better than the OpenMP version.
- `list`: deploys two chunked linear lists with thread safe manipulators based on CAS operations. Threads concurrently remove chunks from the current node frontier and insert unvisited vertices into private chunks. Once a chunk is full, it is inserted into the next node frontier, relaxing concurrent access. The main difference to `graph500` ist that vertices are not copied to a global list but rather a whole chunk gets inserted (updating pointers only). There is some additional overhead, if local chunks get filled only partially.
- `socketlist`: extends the previous approach to respect data locality and NUMA awareness. The data is logically and physically distributed to all NUMA-nodes (i.e., CPU sockets). Each thread primarily processes vertices from its own NUMA-node list where the lists from the previous approach are used for equal distribution of work. If a NUMA-node runs out of work, work is stolen from overloaded NUMA-nodes [6].
- `bitmap`: further refinement and combination of the previous two approaches. A bitmap is used to keep track of visited vertices to reduce memory bandwidth. Again, built-in atomic CAS operations are used to synchronize concurrent access [6].

The first algorithm is vertex-centric, all others are level-synchronous container-centric in our classification and utilize parallelism over the current vertex front. The last three implementations use a programming technique to trade (slightly more) redundant work against atomic operations as described in [25]. `socketlist` is the first in the algorithm list that pays attention to the NUMA memory hierarchy, `bitmap` additionally tries to reduce memory bandwidth by using an additional bitmap to keep track of the binary information whether a vertex is visited or not.

V. Experimental Setup

In this section, we specify our parallel system test environment, describe classes of graphs and chosen graph representatives in this classes.

A. Test Environment

We used in our tests different systems, the largest one a 64-way AMD-6272 Interlagos based system with 128 GB shared memory organised in 4 NUMA nodes, each with 16 cores (1.9 GHz). Two other systems are Intel based with 2 NUMA nodes each (Intel-IB: 48-way E5-2697 Ivy bridge EP at 2.7 GHz,

Intel-SB: 32-way E5-2670 Sandy-Bridge EP at 2.6 GHz). We will focus our discussion on the larger Interlagos system and discuss in section VI-C the influence of the system details.

B. Graphs

It is obvious that graph topology will have a significant influence on the performance of parallel BFS algorithms. We used beside some real graphs synthetically generated pseudo-random graphs that guarantee certain topological properties. R-MAT [26] is such a graph generator with parameters a, b, c influencing the topology and clustering properties of the generated graph (see [26] for details). R-MAT graphs are mostly used to model scale-free graphs. We used in our tests graphs of the following classes:

- Graphs with a very low average and maximum vertex degree resulting in a rather high graph depth and limited vertex fronts. A representative for this class is the road network `road-europe`.
- Graphs with a moderate average and maximum vertex degree. For this class we used Delaunay graphs representing Delaunay triangulations of random points (`delaynay`) and a graph for a 3D PDE-constraint optimization problem (`nlpkkt240`).
- Graphs with a large variation of degrees including few very large vertex degrees. Related to the graph size, they have a smaller graph depth. For this class of graphs we used a real social network (`friendster`), link information for web pages (`wikipedia`), and synthetically generated Kronecker R-MAT graphs with different vertex and edge counts and three R-MAT parameter sets. The first parameter set named 30 is $a = 0.3, b = 0.25, c = 0.25$, the second parameter set 45 is $a = 0.45, b = 0.25, c = 0.15$, and the third parameter set 57 is $a = 0.57, b = 0.19, c = 0.19$. The default for all our RMAT-graphs is the parameter set 57; the graphs with the suffix `-30` and `-45` are generated with the corresponding parameter sets.

All our test graphs are connected, for R-MAT graphs guaranteed with $n-1$ artificial edges connecting vertex i with vertex $i+1$. Some important graph properties for the graphs used are given in table I. For a general discussion on degree distributions of R-MAT graphs see [27].

TABLE I: CHARACTERISTICS FOR SOME OF THE USED GRAPHS.

graph name	$\|V\| \times 10^6$	$\|E\| \times 10^6$	degree avg.	degree max.	graph depth
delaunay (from [28])	16.7	100.6	6	26	1650
nlpkkt240 (from [29])	27.9	802.4	28.6	29	242
road-europe (from [28])	50.9	108.1	2.1	13	17345
wikipedia (from [29])	3.5	45	12.6	7061	459
friendster (from [30])	65.6	3612	55	5214	22
RMAT-1M-10M	1	10	10	43178	400
RMAT-1M-10M-45	1	10	10	4726	16
RMAT-1M-10M-30	1	10	10	107	11
RMAT-1M-100M	1	100	100	530504	91
RMAT-1M-100M-45	1	100	100	58797	8
RMAT-1M-100M-30	1	100	100	1390	9
RMAT-1M-1G	1	1000	1000	5406970	27
RMAT-1M-1G-45	1	1000	1000	599399	8
RMAT-1M-1G-30	1	1000	1000	13959	8
RMAT-100M-1G	100	1000	10	636217	3328
RMAT-100M-2G	100	2000	20	1431295	1932
RMAT-100M-3G	100	3000	30	2227778	1670
RMAT-100M-4G	100	4000	40	3024348	1506

VI. Results

In this section, we discuss our results for the described test environment. Performance results will be given in *Million Traversed Edges Per Second* $MTEPS := m/t/10^6$, where m is the number of edges and t is the time an algorithm takes. MTEPS is a common metric for BFS performance [9] (higher is better). In an undirected graph representing an edge internally with two edges (u,v) and (v,u) only half of the internal edges are counted in this metric.

In the following discussion on results, we distinguish between different views on the problem. It is not possible to show all our results in this paper in detail (3 parallel systems, 35 different graphs, up to 11 thread counts, 32/64 bit versions, different compilers / compiler switches). Rather than that, we summarize results and show only interesting or representative aspects in detail.

On large and more dense graphs, MTEPS values are generally higher than on very sparse graphs. The MTEPS numbers vary between less than 1 and approx. 3,500, depending on the graph. This is due to the fact that in denser graphs many visited edges do not generate an *additional* entry (and therefore work) in a container of unvisited vertices. This observation is not true for `global`, where in all levels all vertices get traversed.

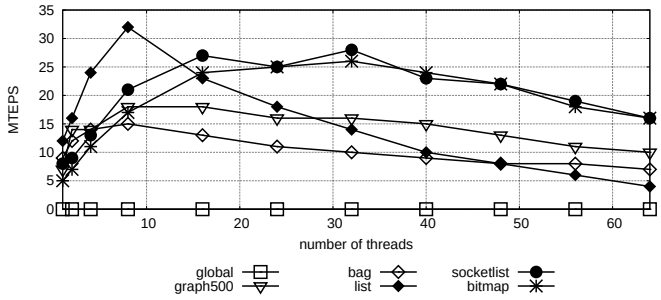
A. Graph Properties and Scalability

In terms of scalability, parallel algorithms need enough parallel work to feed all threads. For graphs with limitating properties, such as small vertex degrees or small total number of vertices / edges, there are problems to feed many parallel threads. Additionally, congestion in accessing smaller shared data structures arise. For such graphs (road network, the delaunay graph and partially small RMAT-graphs), for *all* analysed algorithms performance is limited or even drops as soon the number of threads is beyond some threshold; on *all* of our systems around 8-16 threads. Figure 1a shows the worst case of such an behaviour with `road-europe`. Figure 2 shows different vertex frontier sizes for 3 graphs.

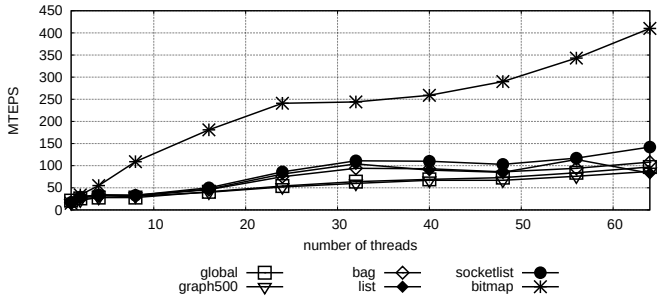
For large graphs and/or high vertex degrees (all larger R-MAT graphs, `friendster`, `nlpkkt240`), the results were quite different from that and all algorithms other than `global` showed on nearly all such graphs and with few exceptions a continuous but in detail different performance increase over all thread counts (see detailled discussion below). Best speedups reach nearly 40 (`bitmap` with `RMAT-1M-1G-30`) on the 64-way parallel system.

B. Algorithms

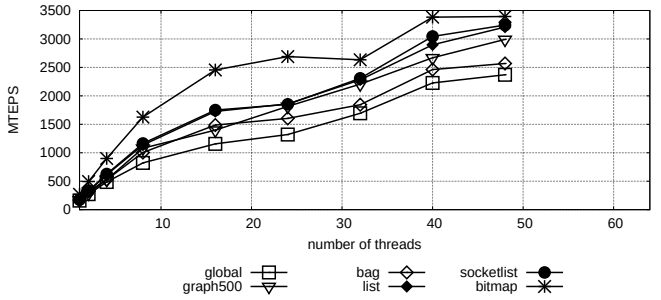
For small thread counts up to 4-8, all algorithms other than `global` show with few exceptions and within a factor of 2 comparable results in absolute performance and behaviour. But, for large thread counts, algorithm behaviour can be quite



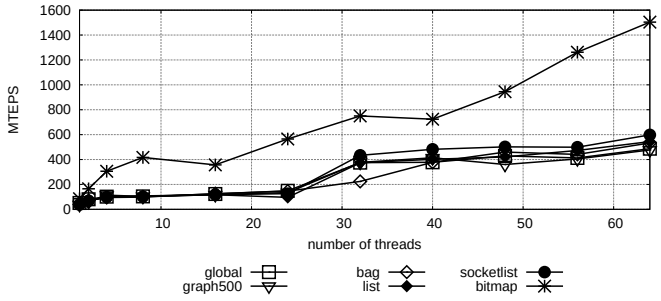
(a) Limited scalability with `road-europe` graph.



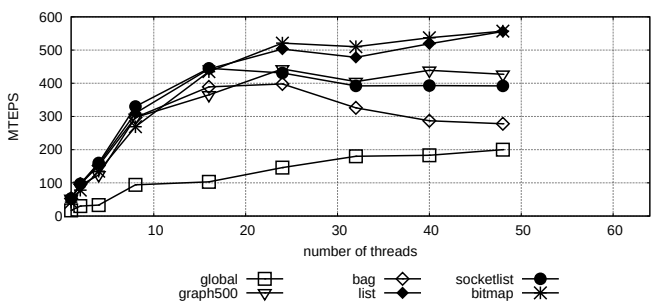
(b) Memory bandwidth optimization with `bitmap` for `friendster` graph.



(c) Similar principal behaviour for dense graphs with a small depth (`RMAT-1M-1G-30` on Intel-IB, 32 bit indices).



(d) `RMAT-1M-1G-30` on AMD system with 64 bit indices.



(e) Intel-IB system with `wikipedia` graph.

Fig. 1: Selected performance results.

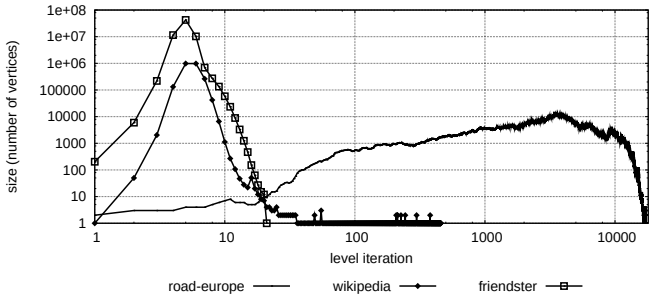


Fig. 2: Dynamic sizes of some vertex frontiers (and potential parallelism).

different. We concentrate, therefore, the following discussions on individual algorithms primarily on large thread counts.

The algorithm `global` has a very different approach than all other algorithms which can be also easily seen in the results. For large graphs with low vertex degrees, this algorithm performs extremely poor as many level-iterations are necessary (e.g., factor 100 slower for road graphs compared to the second worst algorithm; see Figure 1a). The algorithm is only competitive on the systems we used if the graph is very small (no startup overhead with complex data structures) and the graph depth is very low resulting in only a few level-iterations (e.g., less than 10).

The `graph500` algorithm uses atomic operations to increment the position where (a chunk of) vertices get to be inserted into the new vertex front. Additionally, all vertices of a local chunk get copied to the global list (vertex front). This can be fast as long as the number of processors is small. But, as the thread number increases, the cost *per atomic operation* increases [25], and therefore, the performance drops often significantly relative to other algorithms. Additionally, this algorithm does not respect data/NUMA locality on copying vertices which gets a problem with large thread counts.

Algorithm `bag` shows only good results for small thread counts or dense graphs. Similar to `graph500`, this algorithm is not locality / NUMA aware. The bag data structure is based on smaller substructures. Because of the recursive and task parallel nature of the algorithm, the connection between the allocating thread and the data is lost, destroying data locality as the thread count increases. Respecting locality is delegated solely to the run-time system mapping tasks to cores / NUMA nodes. Explicit affinity constructs as in the newest OpenMP version 4.0 [24] could be interesting for that to optimize this algorithm for sparser graphs or many threads.

The simple `list` algorithm has good performance values for small thread counts. But for many threads, `list` performs rather poor on graphs with high vertex degrees. Reasons are implementation specific the use of atomic operations for insert / remove of full/final chunks and that in such vertex lists processor locality is not properly respected. When a thread allocates memory for a vertex chunk and inserts this chunk into the next node frontier, it might be dequeued by another thread in the next level iteration. This thread might be executed on a different NUMA-node, which results in

 ISBN: 978-1-61208-339-1

remote memory accesses. This problem becomes larger with increasing thread/processor counts.

The `socketlist` approach improves the list idea with respect to data locality. For small thread counts, this is a small additional overhead, but, for larger thread counts, the advantage is obvious looking at the cache miss and remote access penalty time of current and future processors (see all figures).

The additional overhead of the `bitmap` algorithm makes this algorithm with few threads even somewhat slower than some other algorithms. But the real advantage shows off with very large graphs and large thread counts, where even higher level caches are not sufficient to buffer vertex fronts. The performance difference to all other algorithms can be significant and is even higher with denser graphs (see Figures 1b, 1c, and 1d).

C. Influence of the system architecture

As described in Section V, we used in our tests different systems but concentrate our discussions so far on results on the largest AMD system. While the principle system architecture on Intel and AMD systems got in the last years rather similar, implementation details, e.g., on cache coherence, atomic operations and cache sizes are quite different.

While the Intel systems were 2 socket systems, the AMD system was a 4 socket system, and that showed (as expected) more sensibility to locality / NUMA. Hyperthreading on Intel systems gave improvements only for large RMAT graphs. Switching from 64 to 32 bit indices (which restricts the number of addressable edges and vertices in a graph) showed improvements due to lower memory bandwidth requirements. These improvements were around 20-30% for all algorithms other than `bitmap`.

VII. Conclusions

In our evaluation for a selection of parallel level synchronous BFS algorithms for shared memory systems, we showed that for small systems / a limited number of threads all algorithms other than `global` behaved almost always rather similar, including absolute performance.

But using large parallel NUMA-systems with a deep memory hierarchy, the evaluated algorithms show often significant differences. Here, the NUMA-aware algorithms `socketlist` and `bitmap` showed constantly good performance and good scalability, if vertex fronts are large enough. Both algorithms utilise dynamic load balancing combined with locality handling, this combionation is a necessity on larger NUMA systems.

References

[1] R. Sedgewick, Algorithms in C++, Part 5: Graph Algorithms, 3rd ed. Addison-Wesley Professional, 2001.

[2] C. Wilson, B. Boe, A. Sala, K. Puttaswamy, and B. Zhao, "User interactions in social networks and their implications," in Eurosys, 2009, pp. 205–218.

[3] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A scalable distributed parallel breadth-first search algorithm on BlueGene/L," in ACM/IEEE Supercomputing, 2005, pp. 25–44.

[4] S. Hong, S. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in 16th ACM Symp. PPoPP, 2011, pp. 267–276.

[5] J. D. Ullman and M. Yannakakis, "High-probability parallel transitive closure algorithms," SIAM Journal Computing, vol. 20, no. 1, 1991, pp. 100–125.

[6] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in ACM/IEEE Intl. Conf. HPCNSA, 2010, pp. 1–11.

[7] J. Chhungani, N. Satish, C. Kim, J. Sewall, and P. Dubey, "Fast and efficient graph traversal algorithm for CPUs: Maximizing single-node efficiency," in Proc. 26th IEEE IPDPS, 2012, pp. 378–389.

[8] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 5th ed. Morgan Kaufmann Publishers, Inc., 2012.

[9] Graph 500 Comitee, Graph 500 Benchmark Suite, http://www.graph500.org/, retrieved: 08.03.2014.

[10] D. Bader and K. Madduri, "SNAP, small-world network analysis and partitioning: an open-source parallel graph framework for the exploration of large-scale networks," in 22nd IEEE Intl. Symp. on Parallel and Distributed Processing, 2008, pp. 1–12.

[11] Y. Xia and V. Prasanna, "Topologically adaptive parallel breadth-first search on multicore processors," in 21st Intl. Conf. on Parallel and Distributed Computing and Systems, 2009, pp. 1–8.

[12] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in Proc. 22nd ACM Symp. on Parallelism in Algorithms and Architectures, 2010, pp. 303–314.

[13] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in 14th Intl. Conf. on High Performance Comp., 2007, pp. 197–208.

[14] P. Harish, V. Vineet, and P. Narayanan, "Large graph algorithms for massively multithreaded architectures," IIIT Hyderabad, Tech. Rep., 2009.

[15] L. Luo, M. Wong, and W. Hwu, "An effective GPU implementation of breadth-first search," in 47th Design Automation Conference, 2010, pp. 52–55.

[16] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," in Intl. Conf. on Parallel Architectures and Compilation Techniques, 2011, pp. 78–88.

[17] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in Proc. Supercomputing 2012, 2012, pp. 1–10.

[18] Y. Yasui, K. Fujusawa, and K. Goto, "NUMA-optimized parallel breadth-first search on multicore single-node system," in Proc. IEEE Intl. Conference on Big Data, 2013, pp. 394–402.

[19] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in Proc. PPoPP. IEEE, 2012, pp. 117–127.

[20] L.-M. Munguìa, D. A. Bader, and E. Ayguade, "Task-based parallel breadth-first search in heterogeneous environments," in Proc. HiPC 2012, 2012, pp. 1–10.

[21] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in Proc. Supercomputing, 2011, pp. 65–79.

[22] H. Lv, G. Tan, M. Chen, and N. Sun, "Understanding parallelism in graph traversal on multi-core clusters," Computer Science – Research and Development, vol. 28, no. 2-3, 2013, pp. 193–201.

[23] Y. Zhang and E. Hansen, "Parallel breadth-first heuristic search on a shared-memory architectur," in AAAI Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications, 2006, pp. 1 – 6.

[24] OpenMP API, 4th ed., OpenMP Architecture Review Board, http://www.openmp.org/, Jul. 2013, retrieved: 08.03.2014.

[25] R. Berrendorf, "Trading redundant work against atomic operations on large shared memory parallel systems," in Proc. Seventh Intl. Conference on Advanced Engineering Computing and Applications in Sciences (ADVCOMP), 2013, pp. 61–66.

[26] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in SIAM Conf. Data Mining, 2004, pp. 442 – 446.

[27] C. Groër, B. D. Sullivan, and S. Poole, "A mathematical analysis of the R-MAT random graph generator," Networks, vol. 58, no. 3, Oct. 2011, pp. 159–170.

[28] DIMACS, DIMACS'10 Graph Collection, http://www.cc.gatech.edu/dimacs10/, retrieved: 08.03.2014.

[29] T. Davis and Y. Hu, Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices/, retrieved: 08.03.2014.

[30] J. Leskovec, Stanford Large Network Dataset Collection, http://snap.stanford.edu/data/index.html, retrieved: 08.03.2014.

 ISBN: 978-1-61208-339-1