Evaluating Parallel Breadth-First Search Algorithms for Multiprocessor Systems

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

***Abstract*—Breadth-First Search is a graph traversal technique used in many applications as a building block, e.g., to systematically explore a search space or to determine single source shortest paths in unweighted graphs. For modern multicore processors and as application graphs get larger, well-performing parallel algorithms are favorable. In this paper, we systematically evaluate an important class of parallel algorithms for this problem and discuss programming optimization techniques for their implementation on parallel systems with shared memory. 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 performance behavior. But, for large core counts and large graphs, there are considerable differences in performance and scalability influenced by several factors, including graph topology. 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

Breadth-First Search (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] [2] [3]. Additionally, BFS is used in many application areas where certain application aspects are modeled 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 [4].

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 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 [2] [3] [5]. The computational complexity is $O(|V|+|E|)$. Level-synchronous BFS algorithms work in parallel on all vertices of one level and have a barrier synchronization [6] before the work for the next level is launched.

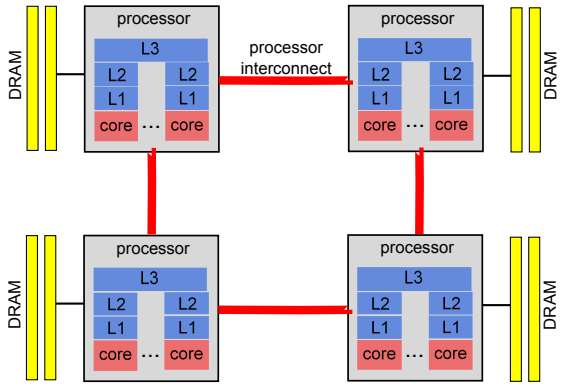


Fig. 1: Principial Structure of a 4-socket Multiprocessor Multicore NUMA system.

Large parallel systems with shared memory are nowadays organized as multiprocessor multicore system in a Non-Uniform Memory Access topology [7] (NUMA; see Fig. 1). In such systems multiple processors (usually with multiple cores) are connected by a fast interconnection network, e.g., QuickPath Interconnect (QPI) on Intel systems [8] or HyperTransport (HT) on AMD systems [9]. All processors / cores share a common address space in a DRAM based main memory. The interesting aspect is that this main memory / address space is distributed to the NUMA nodes. This has consequences for the performance aware programmer as accessing data on processor i that resides on DRAM chips that are assigned / close to processor i is faster than accessing data residing in DRAM chips that are assigned to a different / further away processor. Additionally, the coherence protocol may invalidate cached data in the cache of one processor because another processor modifies the same data. As a consequence, for a programmer

often a global view on parallel data structures is necessary to circumvent performance degradation related to coherence issues.

Many parallel BFS algorithms got published (see Section II for a comprehensive overview including references), all with certain scenarios in mind, e.g., large distributed memory parallel systems using the message passing programming model [10] [11] [12], algorithms variants that are tailored to Graphic Processing Units (GPU) using a different parallel programming model [13] [14] [15], or randomized algorithms for fast, but possibly sub-optimal results [16]. 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 [17] [18]. 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 favorable.

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

II. Related Work and Parallel BFS Algorithms

We combine in our BFS implementations presented later in Section III 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 irregular and dependent on the data / graph topology. Therefore, in large NUMA systems data layout and memory access should respect processor locality [19]. 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 (see Fig. 1 for an illustration).

A more general problem for many parallel algorithms including BFS is a sufficient load balance of work to parallel threads when static partitioning is not sufficient, e.g., distributing statically all vertices in blocks over available cores. Even if an appropriate mechanism for load balancing is deployed, graphs might only supply a limited amount of parallelism. As the governing factor that influences workload and parallelism is the average vertex degree of the traversed graph, especially graphs with a very low average vertex degree are challenging to most algorithms. This aspect notably affects the popular level-synchronous approaches for parallel BFS we concentrate on later. We discuss this aspect in Section V.

The output of an BFS algorithm is an array `level` of size n that stores in `level[v]` the level found for the vertex v. This array can be (mis-)used to keep track of unvisited vertices, too, e.g., initially storing the value -1 in all array elements marking all vertices as unvisited. We discuss in Section II-D other possibilities. As discussed, in BFS algorithms housekeeping has to be done on visited / unvisited vertices as well as frontiers 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.

Important for level-synchronous algorithms is the notion of a level and correlated to that a (vertex) frontier. Fig. 2 explains that notion on an example graph and three level iterations of a level-synchronous BFS algorithm. Starting with a given vertex v_0 this vertex makes up the initial vertex frontier and gets assigned the level 0. All unvisited neighbors of all vertices of the current frontier are part of the next frontier and get the current level plus 1. Such a level iteration is repeated until no more unvisited vertices exist. As can be seen for the example graph in Fig. 2, housekeeping has to be done whether a vertex is visited or not. And working on several vertices of the same level in parallel may lead to a situation where several threads may detect in parallel that a vertex is unvisited. Therefore, care has to be taken to handle such situations, e.g., with synchronization or handling unsynchronized accesses in a appropriate way [20]. As explained before, we concentrate our discussion on connected graphs. To extend BFS for graphs that are not connected, another BFS traversal can be started if one BFS traversal stops with any then unvisited vertex as long as unvisited vertices exist.

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*)

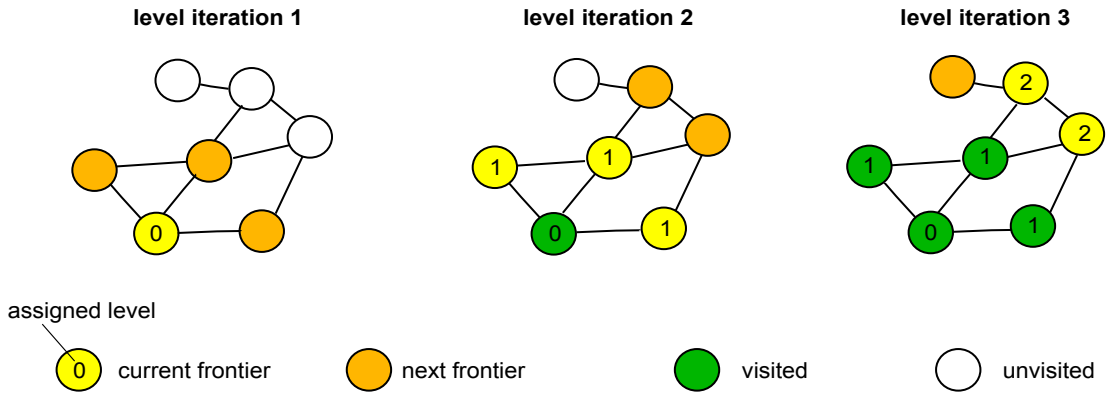


Fig. 2: Vertex levels and frontiers.

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 accesses 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 [21] [22] [23] [24].

For level synchronous approaches, a simple list (for example an array based 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 [10] [25] [26]. Each vertex is associated with a certain thread, which leads to a static one dimensional partitioning of the graph's vertices. When one thread encounters a vertex associated to another thread while processing the adjacent vertices of its local vertex frontier, it adds this foreign vertex to the communication buffer of the owning thread. After a barrier synchronization each thread processes the vertices contained in the communication buffers of each foreign thread. As the number of threads increases, an increased number of communication buffers must be allocated, limiting the scalability of this approach. Due to the one dimensional partitioning data locality can be utilized.

In contrast 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 [17] [18]. While a global list for the current vertex frontier supplies fundamental work balance, it completely ignores data locality. In the NUMA optimized approach each vertex is assigned to one memory bank, leading to a one dimensional vertex partitioning among the sockets / NUMA nodes. As described above communication buffers are managed for each socket that gather foreign vertices. The local vertex frontier for one socket is a concurrent list, which is processed in parallel by the threads belonging to this socket.

Furthermore, lists can be utilized to implement container centric approaches on special hardware platforms as graphic accelerators with warp centric programming [13] [27]. Instead of threads, warps (groups of threads; [15]) become the governing parallel entities. With warp centric programming, BFS is divided into two phases: SISD and SIMD. In a SISD phase, each thread of a warp executes the same code on the same data. These phases are used to copy global vertex chunks to warp local memory. Special hardware techniques support efficient copying on warp level. In the SIMD phase each thread executes the same statements but on different data. This is used to process the adjacent vertices of one vertex in parallel. Because the workload is split into chunks and gradually processed by the warps, fundamental work balancing is ensured. To avoid ill-sized chunks another optimization may be applied: deferring outliers [13], which will be discussed in a later section.

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 [28] [29], 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 or rather leaves it solely to a thread runtime system. A *bag* is an array of a special kind of binary trees that holds the vertices of the current and the next vertex frontier. One can insert new nodes into a *bag*, split one *bag* into two almost equal sized *bags* and unify two *bags* to form a new *bag*. All operations are designed to work with minimal complexity. When beginning a new BFS level iteration one *bag* forms the entire vertex frontier for this iteration. The initial *bag* is then split into two parts. The first part is used for the current thread and the second is used to spawn a new thread (or parallel task). A thread splits and spawns new tasks until its input *bag* is smaller than a specified threshold. In this case the thread processes the vertices from

its partial *bag* and inserts the next frontier's vertices into a new *bag*. When two threads finish processing their *bags*, both results are unified. This is recursively done until only one *bag* remains, which then forms the new vertex frontier.

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 parallelized. When processing a vertex, its neighbors are inspected and if unvisited, marked as part of the next vertex frontier. The worst case complexity for this approach is therefore $O(n^2)$ 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 synchronization at the end of a level iteration, a vertex centric approach does with some care not need any additional synchronization. The implementation is therefore rather simple and straightforward. The disadvantages of vertex centric approaches are the lacking mechanisms for load balancing and graphs with a large depth.

But this overall approach makes it well-suited for GPU's where each vertex is mapped to exactly one thread [30] [31]. 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 [32].

Their linear memory access and the possibility to take care of data locality allow vertex centric approaches to be efficiently implemented on NUMA machines [27]. Combined with a proper partitioning, they are also suitable for distributed systems, as the overhead in communication is rather low. But as pointed out already above, this general approach is suited only for graphs with a very low depth.

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 [16], 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. [33] 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. [34] explores this approach in more detail for multicore systems. In [35], a fast GPU algorithm is introduced that combines fast primitive operations like prefix sums available with highly-optimized libraries. A task-based approach for a combination of CPU/ GPU is presented by Munguia et al. [36].

A BFS traversal can be implemented as a matrix-vector product over a special semi-ring [37]. Matrix-vector multiplication is subject to elaborated research and one could profit from highly optimized parallel implementations when implementing BFS as a matrix-vector product.

Additionally, there are (early) algorithms that focus more on a principial approach while ignoring important aspects of real parallel systems [38] [39].

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 [10]. 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 [37] and use asynchronous communication [40] to tolerate communication latencies: one dedicated communication thread per process is used to take care of all communication between the different processes. The worker threads communicate via multiple producer / single consumer queues with the communication thread [40]. Each worker writes the non-local vertices to the proper queue while the communication thread monitors these queues. When a queue is full, the communicator sends its contents to the associated process. This reduces the communication overhead at the end of an iteration. Scarpazza discusses in [41] optimizations for the Cell/B.E. processor. Pearce [42] discusses techniques to use even NAND based Flash memories for very large graphs that do not fit into main memory.

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 [17] instead of (mis-)using the level information to keep track of unvisited vertices (e.g., `level[v]` equals -1 for an unvisited vertex *v*). The intention is to keep more information on visited vertices in a higher level of the cache hierarchy as well as to reduce memory bandwidth demands. Bitmaps can be used to optimize container as well as vertex centric approaches.

Fine-grained tuning like memory pre-fetching can be used to tackle latency problems [18] (but which might produce even more pressure on memory bandwidth).

Most container centric approaches work on vertex chunks instead of single vertices [28]. This reduces container access and synchronization overhead.

To avoid unequal workloads on different threads another optimization is to defer outliers [13] that could slow down one thread and force all others to go idle and wait for the busy thread at the end of an iteration. Usually, outliers are vertices with an unusual high vertex degree. When a thread encounters an outlying vertex, this vertex is inserted into a special container. This way processing outliers is deferred until the end of an iteration, avoiding unequal distribution of workload among the threads.

Besides implicit load balancing of some container centric approaches, there exist additional methods. One is based on

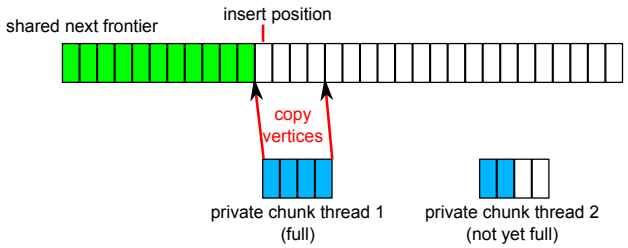


Fig. 3: Inserting vertices of a full chunk into the global list of a vertex frontier with algorithm `graph500`.

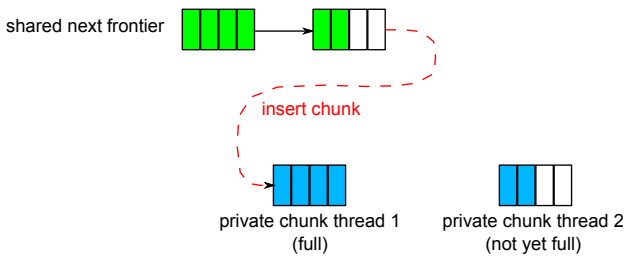


Fig. 4: Inserting vertices of a full chunk into the global list of a vertex frontier with algorithm `list`.

a logical ring topology [43] of the involved threads. Each thread keeps track of its neighbor's workload and supplies it with additional work, if it should be idle. A single control thread is present but the load balancing is done by the worker threads so the controller does not become a communication bottleneck. A downside of this optimization is that shifting work between threads may destroy data locality and affect overall performance.

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 [25]. Like the previous optimization this does not go well with data locality for NUMA architectures or distributed systems because vertices owned by an inactive process or thread would have to be moved to some active unit.

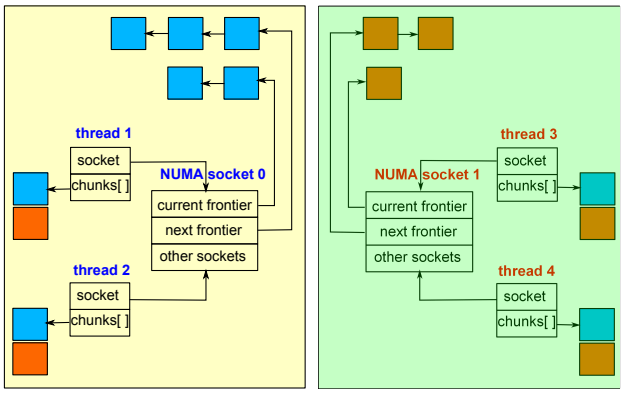


Fig. 5: Modeling a multicore NUMA architecture in software with algorithm `socketlist` (example for a 2-socket 2-core system).

III. 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 II-B, with parallel iterations over *all vertices on each level* [27]. The vertices are distributed statically in blocks to all threads. As pointed out already, this will only work for graphs with a very low depth. All data is allocated in parallel to meet the NUMA first touch policy and take care of data locality. First touch strategy [44] means that the processor that executes the initial access to a part of a data structure (usually with a granularity of a 4 KB page of the virtual address space) allocates that part of the data structure in the DRAM that is assigned to that NUMA node. This is the default allocation strategy in most operating systems including Linux. The distribution of data and work is therefore to split the vertices into blocks and assign each block to a different thread. Then, the data allocation as well as the work on that data is distributed.
- `graph500`: OpenMP reference implementation in the Graph500 benchmark [21] 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-/thread-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 (atomically increasing a write index). All vertices of a full chunk get copied to the global array list. See Fig. 3 for a two thread example.
- `bag`: using OpenMP [45] tasks and two bag containers as described in [28]. This approach implicitly deploys load balancing mechanisms. Because of its parallel task based divide and conquer nature it does not take data locality into account (but leaves it solely to the thread runtime system). The original *bag* data structure is extended by a so called *hopper* [28]. This additional structure serves as a vertex cache to reduce the amount of insert operations on the main structure. In addition to the OpenMP implementation we implemented a Cilk+ version [46] as in the the original paper that did not show any significant differences to the OpenMP version with respect to performance.
- `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, the chunk is inserted into the next node frontier, relaxing concurrent access. The main difference to `graph500` is that vertices are not copied to a global list but rather a whole chunk gets inserted (updating pointers only). And another difference to `graph500` is related to that the distribution of vertices of a frontier to

threads is chunk-based rather than vertex based. There is some additional overhead, if local chunks get filled only partially. See Fig. 4 for a two thread example.

- `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., processor sockets). Each thread primarily processes vertices of the current vertex frontier from its own NUMA-node list where the lists from the previous level are used for equal distribution of work. If a NUMA-node runs out of work, work is stolen from overloaded NUMA-nodes [17]. A newly detected vertex for the next vertex frontier is first inserted into a thread local chunk. Instead of using only one buffer chunk per NUMA node, *n* chunks are used per NUMA node by each thread where *n* is the number of NUMA nodes in the system. This simplifies NUMA-aware insertion of a chunk into the correct part of the next frontier. See Fig. 5 for a four threads in a 2-socket 2-core system.
- `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 [17].

The first algorithm `global` 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 [20]. `socketlist` is the first container-centric algorithm in the 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.

Some of the algorithms work with chunks were the chunk size is an algorithm parameter. For those algorithms we used a chunk size of 64 vertices. Another algorithm parameter in the threshold value in `bag`; we used 1,000 for that parameter. We did some pre experiments with these algorithm parameters using different graphs and different system and found that these chosen values were reasonable / best for most / many test instances.

IV. Experimental Setup

Beside the choice of the parallel algorithm itself certain parameters may influence the performance of parallel BFS, possibly dependent on the algorithm used. We are interested in relative performance comparisons between the different algorithms but also in their scalability for an increasing degree of parallelism. The latter aspect is of particular interest as future processors / systems will have more parallelism than todays' systems and that available hardware parallelism needs to be utilized efficiently by programs / algorithms. Therefore, scalability is a major concern. Large parallel systems with different architectures are used in the evaluation to examine the influence of the degree of parallelism and main system aspects.

Furthermore, the graph topology will likely have a significant influence on performance. Level-synchronous algorithms in general need on *each* level enough parallelism to feed all available threads. On the other side, if there is significantly more parallelism available than the number of threads, this parallelism has to be managed. Therefore, the available parallelism in each BFS level as well as the distribution of available parallelism over all levels is of interest when evaluating such algorithms.

In this section, we specify our parallel system test environment, describe classes of graphs and chosen graph representatives in this classes. The algorithms are implemented in C and partially C++ using the OpenMP parallel programming model [45]. Porting this to other parallel programming models utilizing thread-based shared memory parallelism like the new thread functionality in the recent language standards for C [47] and C++ [48], or using similar thread-based programming models like PThreads [49] or Cilk+ [46] should be rather straightforward.

To be able to handle also very large graphs, 64 bit indices were used throughout all tests unless otherwise stated. A discussion on using 32 bit indices (which can reduce memory bandwidth demands significantly) for graphs that are limited to roughly 4 billion vertices / edges is done in Section V-C.

A. Test Environment

Today, any mainstream system with more than one processor socket is organized as a NUMA system. Fig. 1 has shown the principal system architecture of such a system, in the example shown for a 4 socket system. On such systems, especially data-intensive algorithms have to respect distributed data allocation [44] [50] and processor locality [19] in the execution of an algorithm. Beside the performance characteristics of the memory hierarchy known from modern processors [7], in a NUMA system an additional penalty exists if a core accesses data that is not cached by a private L1/L2 cache of this core or the shared L3 cache of the corresponding processor, and the data resides in that part of the distributed main memory that is allocated on a different NUMA socket.

We used in our tests parallel systems (see Table I for details) that span a spectrum of system parameters, mainly the degree of parallelism, NUMA topology, cache sizes, and cycle time. The largest system is a 64-way AMD-6272 Interlagos based system with 128 GB shared memory organized in 4 NUMA nodes, each with 16 cores. An additional AMD based system with 4 NUMA nodes but fewer core count was used, too. Two other systems are Intel based systems with only 2 NUMA nodes each. We will focus our discussion on the larger Interlagos system and discuss in Section V-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

TABLE I: SYSTEMS USED.

name	Intel-IB	Intel-SB	AMD-IL	AMD-MC
processor:				
manufacturer	Intel	Intel	AMD	AMD
CPU nodel	E5-2697	E5-2670	Opteron 6272	Opteron 6168
architecture	Ivy Bridge	Sandy Bridge	Interlagos	Magny Cours
frequency[GHz]	2.7	2.6	2.1	1.9
last level cache size [MB]	30	20	16	12
system:				
memory [GB]	256	128	128	32
number of CPU sockets	2	2	4	4
n-way parallel	48	32	64	48

used some larger real graphs from the DIMACS-10 challenge [51], the Florida Sparse Matrix Collection [52], and the Stanford Large Dataset Collection [53]. Additionally, we used synthetically generated pseudo-random graphs that guarantee certain topological properties. R-MAT [54] is such a graph generator with parameters a, b, c influencing the topology and clustering properties of the generated graph (see [54] for details). R-MAT graphs are mostly used to model scale-free graphs. The graph `friendster` and larger RMAT-graphs could not be used on all systems due to memory requirements. 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 $a = 0.57, b = 0.19, c = 0.19$.

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 are given in Table II. For a general discussion on degree distributions of R-MAT graphs see [55].

V. 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 elapsed time in seconds an algorithm takes. MTEPS is a common metric for BFS performance [21] (higher is better). To give an idea on the elapsed time, for example an MTEPS value of 2000 for the graph `RMAT-1M-1G-57` with 1 million vertices and 1 billion edges corresponds to 250 milliseconds execution time for the whole graph traversal on the system Intel-SB. 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.

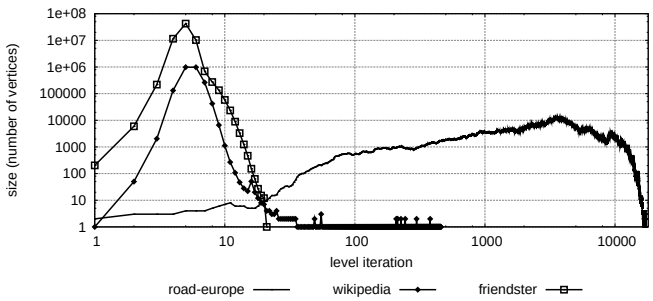


Fig. 6: Dynamic sizes of some vertex frontiers and potential parallelism.

On large and more dense graphs, MTEPS values are generally higher than on very sparse graphs. 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 the algorithm `global`, where in all levels all vertices get traversed. The MTEPS numbers for the graphs and systems used vary between less than 1 and approx. 3,500, depending on the graph, system and algorithm.

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 (4 parallel systems, 26 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.

A. Graph Properties and Scalability

In terms of scalability, parallel algorithms need enough parallel work to feed all threads. For graphs with limiting properties, such as very 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.

Fig. 6 shows relevant vertex frontier sizes for 3 selected graphs showing different characteristics. The x axis gives the level of BFS traversal, the y axis shows the corresponding

TABLE II: CHARACTERISTICS OF THE USED GRAPHS.

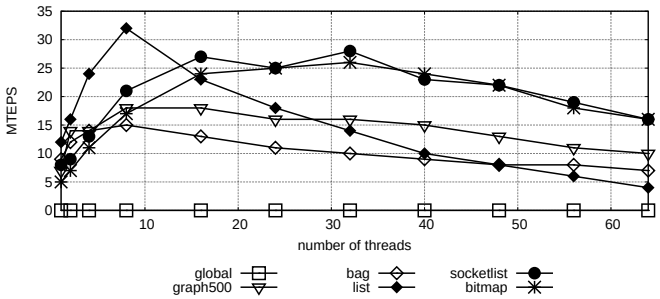
graph name	$\|V\| \times 10^6$	$\|E\| \times 10^6$	degree avg.	degree max.	graph depth
delaunay (from [51])	16.7	100.6	6	26	1650
nlpkkt240 (from [52])	27.9	802.4	28.6	29	242
road-europe (from [51])	50.9	108.1	2.1	13	17345
wikipedia (from [52])	3.5	45	12.6	7061	459
friendster (from [53])	65.6	3612	55	5214	22
RMAT-1M-10M-30	1	10	10	107	11
RMAT-1M-10M-45	1	10	10	4726	16
RMAT-1M-10M-57	1	10	10	43178	400
RMAT-1M-100M-30	1	100	100	1390	9
RMAT-1M-100M-45	1	100	100	58797	8
RMAT-1M-100M-57	1	100	100	530504	91
RMAT-1M-1G-30	1	1000	1000	13959	8
RMAT-1M-1G-45	1	1000	1000	599399	8
RMAT-1M-1G-57	1	1000	1000	5406970	27
RMAT-100M-1G-30	100	1000	10	181	19
RMAT-100M-1G-45	100	1000	10	37953	41
RMAT-100M-1G-57	100	1000	10	636217	3328
RMAT-100M-2G-30	100	2000	20	418	16
RMAT-100M-2G-45	100	2000	20	85894	31
RMAT-100M-2G-57	100	2000	20	1431295	1932
RMAT-100M-4G-30	100	4000	40	894	15
RMAT-100M-4G-45	100	4000	40	180694	31
RMAT-100M-4G-57	100	4000	40	3024348	1506
RMAT-100M-8G-30	100	8000	40	1807	15
RMAT-100M-8G-45	100	8000	40	371454	21
RMAT-100M-8G-57	100	8000	40	6210095	1506

size of the vertex frontier for this level. The frontier size for `friendster` has a steep curve (i.e., there is soon enough parallelism available) that remains high for nearly all level iterations. The frontier size for `wikipedia` start similar, but has for the later level iterations only few vertices per frontier left, which restricts *any* level-synchronous BFS algorithm in utilizing parallelism in this later level iterations. And the worst case shown is the graph `road-europe` where the frontier size never exceeds more than roughly 10,000 vertices. Working on 10,000 vertices with 64 threads means that every thread has not more than roughly 150 vertices to work on, and the computational density for BFS is rather low.

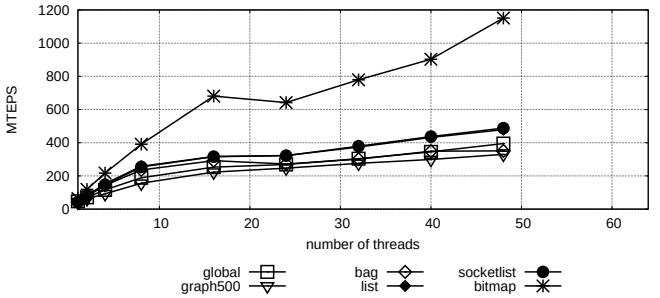
For graphs with such limiting properties (road network, the delaunay graph and partially small RMAT-graphs), for *all* analyzed algorithms performance is limited or even drops as soon the number of threads is beyond some threshold; on *all* of our systems around 8-16 threads. Fig. 7a shows the worst case of such an behavior with `road-europe`. For graphs with such properties, other algorithms different to a level-synchronous approach should be taken into account, e.g., [16].

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 Fig. 7b for an example and the detailed discussion below). Best speedups reach nearly 40 (`bitmap` with `RMAT-1M-1G-30`) on the 64-way parallel system.

For denser graphs with a very low depth often all algorithms show a very similar behavior and even absolute performance, even with an increasing number of threads. See Fig. 8 for an example.



(a) Limited scalability with graph `road-europe` on system AMD-IL.



(b) Continuous performance increase with more threads for graph `friendster` on system Intel-IB.

Fig. 7: Differences in Scalability.

B. Algorithms

For small thread counts up to 4-8, all algorithms other than `global` show with few exceptions and within a factor of

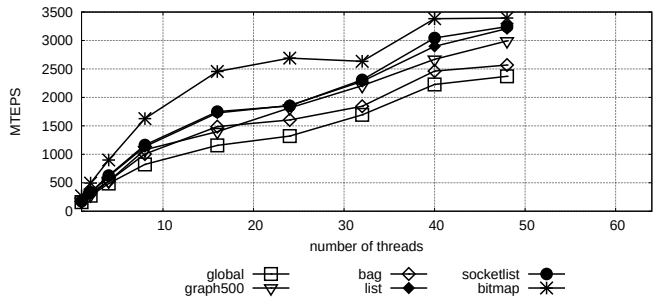
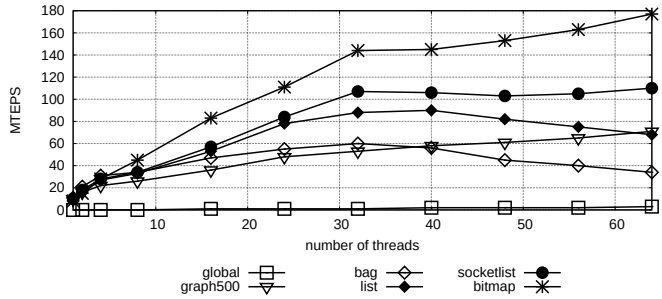
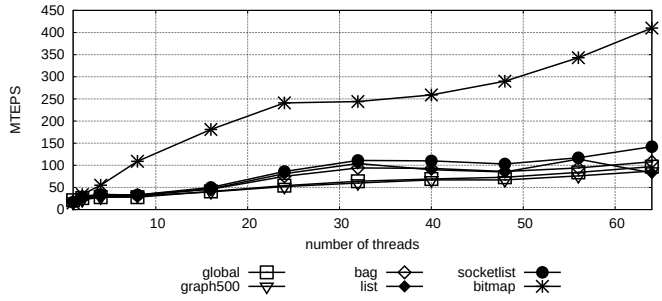


Fig. 8: Similar principal behavior for dense graphs with a small depth like on the graph RMAT-1M-1G-30 on system Intel-IB with 32 bit indices.



(a) Benefit of NUMA awareness for graph RMAT-100M-1G-57 on system AMD-IL.



(b) Memory bandwidth optimization with algorithm bitmap for graph friendster on system AMD-IL.

Fig. 9: Benefits of clever memory handling.

2 comparable results in absolute performance and principal behavior. Therefore, for small systems / a low degree of parallelism the choice of algorithm is not really crucial. But for large thread counts, algorithm behavior can be quite different. Therefore, we concentrate the following discussion on individual algorithms primarily on large thread counts (more than 8).

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 the road graph compared to the second worst algorithm; see Fig. 7a for an example of that behavior. The algorithm is only competitive on the systems we used if the graph is very small and the graph depth is very low resulting in only a few level-iterations, e.g., less than 10. See Fig. 8 for an example.

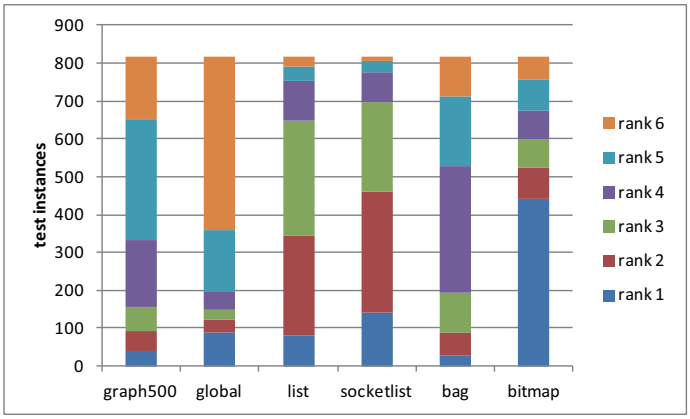
The graph500 algorithm uses atomic operations to increment the position where (a chunk of) vertices get to be inserted into the new vertex front. Additionally, all vertices of a local chunk get copied to the global list (vertex front). This can be fast as long as the number of processors is small. But, as the number of threads increases, the cost *per atomic operation* increases [20], and therefore, the performance drops often significantly relative to other algorithms. Additionally, this algorithm does not respect data / NUMA locality on copying vertices from a local chunk to a global list, which gets a serious 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, often destroying data locality as the thread count increases. Respecting locality is delegated solely to the run-time system mapping tasks to cores / NUMA nodes. In principle, it is often a good idea to delegate complex decisions to runtime systems. But in this case the runtime system (without modifications / extensions) has not enough knowledge about the whole context that would be necessary to further optimize the affinity handling in a more global view. Explicit affinity constructs as in the latest OpenMP version 4.0 [45] could be interesting for that to explicitly optimize this algorithm for sparser graphs or many threads instead of leaving all decisions to the runtime system.

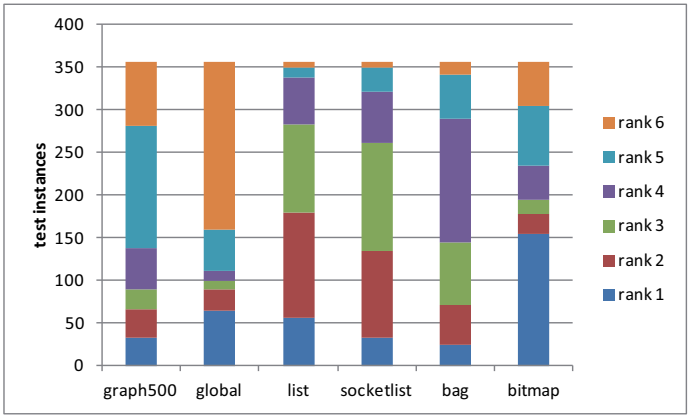
The simple list algorithm has good performance values for small thread counts. But for many threads, list performs rather poor on graphs with high vertex degrees. Reasons are implementation specific the use of atomic operations for insert / remove of full / final chunks and that in such vertex lists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

The socketlist approach improves the list idea with respect to NUMA aware data locality at the expensive of an additional more complex data structure. For small thread counts, this is an additional overhead that often does not pay off but on the other side also does not drop performance in a relevant way. 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 Fig. 9).

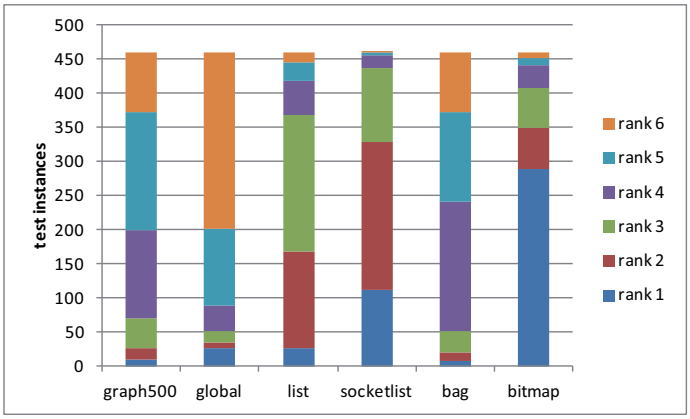
The additional overhead of the bitmap algorithm makes this algorithm with only a few threads even somewhat slower than some other algorithms (but again not in a relevant way). But the real advantage shows off with very large and dense graphs and large thread counts, when even higher level caches are not



(a) Algorithm ranking for all test instances.



(b) Algorithm ranking for a thread count up to 8.



(c) Algorithms ranking for a thread count 16 and more.

Fig. 10: Relative performance ranking of algorithms.

sufficient to buffer vertex fronts and memory bandwidth gets the real bottleneck. The performance difference to all other algorithms can then be significant and is even higher with denser graphs (see Figs. 8, 9a, 9b).

In the above discussion performance observations on algorithms were given in a general way but explicitly shown only for selected examples. Fig. 10 shows summarizing statistics on the algorithms for all graphs on all systems and all thread counts. The six algorithms of interest are ranked for each problem instance (graph, system, thread count) on their relative performance to each other with a rank from 1 to 6. An algorithm ranked first for a problem instance was the best performing algorithm for that problem instance. A rank says nothing about the absolute difference between two algorithms for a problem instance. The difference between two performance numbers might be rather small while for another problem instance the performance of the algorithm ranked first might be significant higher then the performance of the second ranked algorithm.

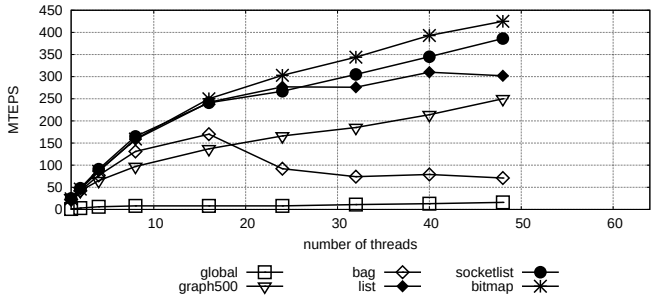
Fig. 10 shows three histograms: for all test instances, for thread counts up to 8 (i.e., small parallel systems), and for threads counts of 16 and more (i.e., large parallel systems). As a remark, the results for very different graphs topologies are summarized in these diagrams such that for example for very small graphs algorithms with a startup overhead to generate complex data structures have a disadvantage and may be ranked lower. As can be clearly seen, the algorithms that optimize memory accesses (awareness of NUMA topology, memory bandwidth reduction) show best results. This is especially true for many threads and for the two systems with 4 NUMA nodes (see for example Figs. 7b and 9b for this observation). The algorithm `bitmap` was ranked first or second in more than 75% of all problems instances with large threads counts and roughly 50% even for small threads counts. Also, evidently the algorithm `global` is worst in more than halve of all instances.

C. Influence of the system architecture

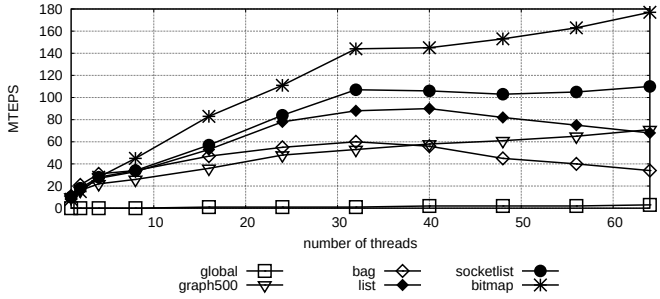
As described in Section IV, we used in our tests different systems but concentrated our discussions so far on results on the largest AMD-IL system. While the principle system architecture on Intel and AMD systems got in the last years rather similar, implementation details, e.g., on cache coherence, atomic operations, cache sizes, and the micro-architecture are quite different [56] [57].

While the Intel systems are 2 socket systems, the AMD systems are 4 socket system, and the latter systems showed (as expected) more sensibility to locality / NUMA. Fig. 11 shows this sensibility for the same graph on a 4 socket system and on a 2 socket system. While on the 4 NUMA node system the `bitmap` algorithm has more advantages than all other algorithms, on the less sensitive 2 NUMA node system the performance difference of the algorithms to all other algorithms is less.

Hyper-Threading on Intel systems gave improvements only for large RMAT graphs. There is a choice of using 32 bit indices (i.e., the data type **`int`** or **`unsigned int`**) or 64 bit indices (i.e., the data type **`long`** or **`unsigned long`**). Using a 32 bit index limits the number of vertices / edges to not more than roughly 4 billion. On the other side, a 32 bit index requires less memory bandwidth, which is rather preciously for a BFS algorithm with very low computational density. Comparing 32 bit ro 64 bit results showed as expected performance improvements due to lower memory bandwidth requirements

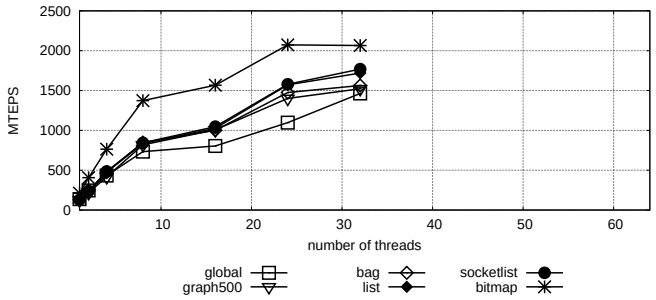


(a) 2 NUMA nodes for graph RMAT-100M-1G-57 on system Intel-IB.

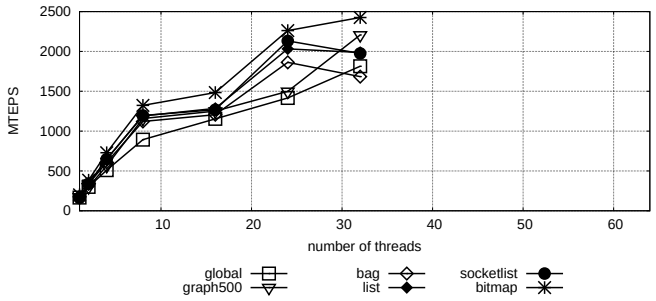


(b) 4 NUMA nodes for graph RMAT-100M-1G-57 on system AMD-IL.

Fig. 11: Difference in NUMA nodes.



(a) 64 bit indices for graph RMAT-1M-1G-30 on system Intel-SB.



(b) 32 bit indices for graph RMAT-1M-1G-30 on system Intel-SB.

Fig. 12: Switching from 64 to 32 bit indices.

and fitting more information in caches. These improvements were around 20-30% for all algorithms other than bitmap (with a less bandwidth pressure than the other algorithms). Fig. 12 shows an example for this improvement. The absolute improvement is for example for 32 threads roughly 1800 MTEPS with 32 bit indices compared to roughly 1500 MTEPS with 64 bit indices. The advantage of the algorithm bitmap compared to all other algorithms decreases as the pressure on memory bandwidth in decreased with 32 bit instead of 64 bit indices.

VI. 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. Therefore, for very small systems the choice of parallel BFS algorithm other than global is not crucial.

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 utilize dynamic load balancing combined with locality handling, this combination is a necessity on larger NUMA systems. Especially on larger and more dense graphs, the bitmap shows often significant performance advantages over all other algorithms (approx. 75% of all test instances ranked first or second). Using 32 bit indices for smaller graphs instead of 64 bit indices reduces the benefit of this algorithm compared to the others.

The level-synchronous approach should be used only if the graph topology ensures enough parallelism on each / on most levels for the given system. Otherwise, *any* level-synchronous BFS algorithm has problems to feed that many threads.

Acknowledgements

The system infrastructure was partially funded by an infrastructure grant of the Ministry for Innovation, Science, Research, and Technology of the state North-Rhine-Westphalia.

We thank the anonymous reviewers for their careful reading and their helpful comments and suggestions on an earlier version of the manuscript.

References

[1] R. Berrendorf and M. Makulla, "Level-synchronous parallel breadth-first search algorithms for multicore- and multiprocessors systems," in *Proc. Sixth Intl. Conference on Future Computational Technologies and Applications (FUTURE COMPUTING 2014)*, 2014, pp. 26–31.

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

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[4] C. Wilson, B. Boe, A. Sala, K. P. Puttaswamy, and B. Y. Zhao, "User interactions in social networks and their implications," in *Proc. 4th ACM European Conference on Computer Systems (Eurosys)*, 2009, pp. 205–218.

[5] J. G. Siek, L.-Q. Lee, and A. Lumbdsdaine, *The Boost Graph Library*. Addison-Wesley Professional, 2001.

[6] P. S. Pacheco, *An Introduction to Parallel Programming*. Burlington, MA: Morgan Kaufman, 2011.

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

[8] Intel, *Intel® Quickpath Interconnect Maximizes Multi-Core Performance*, http://www.intel.com/technology/quickpath/, retrieved: 22.11.2014.

[9] Hypertransport Consortium, http://www.hypertransport.org/, retrieved: 22.11.2014.

[10] 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 Conf. on Supercomputing*, 2005, pp. 25–44.

[11] F. Checconi, J. Willcock, and A. R. Choudhury, "Breaking the speed and scalability barriers for graph exploration on distributed-memory machines," in *Proc. Intl. Conference on High-Performance Computing, Networking and Storage and Analysis (SC'12)*, 2012, pp. 1–12.

[12] Message Passing Interface Forum, "MPI: A message-passing interface standard, version 3.0," Tech. Rep., Jul. 2012.

[13] S. Hong, S. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proc. 16th ACM Symp. Principles and Practice of Parallel Processing (PPoPP)*, 2011, pp. 267–276.

[14] D. Li and M. Becchi, "Deploying graph algorithms on GPUs: an adaptive solution," in *Proc. 27nd Intl. Symp. on Parallelism and Distributed Computing (IPDPD2013)*. IEEE, 2013, pp. 1013–1024.

[15] *CUDA Toolkit Documentation v6.0*, Nvidia, http://docs.nvidia.com/cuda/, 2014, retrieved: 22.11.2014.

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

[17] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *ACM/IEEE Intl.Conf. for High Performance Computing, Networking, Storage and Analysis (HPCNSA)*, 2010, pp. 1–11.

[18] 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 Intl. Parallel and Distributed Processing Symposium*. IEEE, 2012, pp. 378–389.

[19] A. Agarwal and A. Gupta, "Temporal, processor and spatial locality in multiprocessor memory references," *Frontiers of Computing Systems Research*, vol. 1, pp. 271–295, 1990.

[20] 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.

[21] Graph 500 Comitee, *Graph 500 Benchmark Suite*, http://www.graph500.org/, retrieved: 22.11.2014.

[22] 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.

[23] N. Edmonds, J. Willcock, A. Lumsdaine, and T. Hoefler, "Design of a large-scale hybrid-parallel graph library," in *Intl. Conference on High Performance Computing Student Reserarch Symposium*. IEEE, Dec. 2010.

[24] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. Principles and Practice of Parallel Processing*. IEEE, 2013, pp. 135–146.

[25] 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.

[26] D. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2," in *35th Intl. Conf. on Parallel Processing*, 2006, pp. 523–530.

[27] 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.

[28] 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.

[29] T. B. Schardl, "Design and analysis of a nondeterministic parallel breadth-first search algorithm," Master's thesis, MIT, EECS Department, Jun. 2010.

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

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

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

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

[34] 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.

[35] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *Proc. Principles and Practice of Parallel Processing*. IEEE, 2012, pp. 117–127.

[36] L.-M. Munguìa, D. A. Bader, and E. Ayguade, "Task-based parallel breadth-first search in heterogeneous environments," in *Proc. Intl. Conf. on High Performance Computing (HiPC 2012)*, 2012, pp. 1–10.

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

[38] H. Gazit and G. L. Miller, "An improved parallel algorithm that computer the BFS numbering of a directed graph," *Information Processing Letters*, vol. 28, pp. 61–65, Jun. 1988.

[39] R. K. Gosh and G. Bhattacharjee, "Parallel breadth-first search algorithms for trees and graphs," *Intern. Journal Computer Math.*, vol. 15, pp. 255–268, 1984.

[40] 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, pp. 193–201, 2013.

[41] D. Scarpazza, O. Villa, and F. Petrini, "Efficient breadth-first search on the Cell/BE processor," *IEEE Trans. Par. and Distr. Systems*, pp. 1381–1395, 2008.

[42] R. Pearce, M. Gokhale, and N. M. Amato, "Scaling techniques for massive scale-free graphs in distributed (external) memory," in *Proc. Intl. Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2013, pp. 825–836.

[43] 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.

[44] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP*. Cambridge, MA: The MIT Press, 2008.

[45] *OpenMP Application Program Interface*, 4th ed., OpenMP Architecture Review Board, http://www.openmp.org/, Jul. 2013, retrieved: 22.11.2014.

[46] *Intel® Cilk™Plus*, https://software.intel.com/en-us/intel-cilk-plus, retrieved: 22.11.2014.

[47] *ISO/IEC 9899:2011 - Programming Languages – C*, ISO, Genf, Schweiz, 2011.

[48] *ISO/IEC 14882:2011 Programming Languages – C++*, ISO, Genf, Schweiz, 2011.

[49] IEEE, *Posix.1c (IEEE Std 1003.1c-2013)*, Institute of Electrical and Electronics Engineers, Inc., 2013.

[50] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*. San Francisco: Morgan Kaufman Publishers, 2001.

[51] DIMACS, *DIMACS'10 Graph Collection*, http://www.cc.gatech.edu/dimacs10/, retrieved: 22.11.2014.

[52] T. Davis and Y. Hu, *Florida Sparse Matrix Collection*, http://www.cise.ufl.edu/research/sparse/matrices/, retrieved: 22.11.2014.

[53] J. Leskovec, *Stanford Large Network Dataset Collection*, http://snap.stanford.edu/data/index.html, retrieved: 22.11.2014.

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

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

[56] Intel, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, 2014.

[57] Advanced Micro Devices, *Software Optimization Guide for AMD Family 15h Processors*, Jan. 2012.