A Technique to Avoid Atomic Operations on Large Shared Memory Parallel Systems

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

***Abstract*—Updating a shared data structure in a parallel program is usually done with some sort of high-level synchronization operation to ensure correctness and consistency. The realization of such high-level synchronization operations is done with appropriate low-level atomic synchronization instructions that the target processor architecture provides. These instructions are costly and often limited in their scalability on larger multi-core / multi-processor systems. In this paper, a technique is discussed that replaces atomic updates of a shared data structure with ordinary and cheaper read/write operations. The necessary conditions are specified that must be fulfilled to ensure overall correctness of the program despite missing synchronization. The advantage of this technique is the reduction of access costs as well as more scalability due to elided atomic operations. But on the other side, possibly more work has to be done caused by missing synchronization. Therefore, additional work is traded against costly atomic operations. A practical application is shown with level-synchronous parallel Breadth-First Search on an undirected graph where two vertex frontiers are accessed in parallel. This application scenario is also used for an evaluation of the technique. Tests were done on four different large parallel systems with up to 64-way parallelism. It will be shown that for the graph application examined the amount of additional work caused by missing synchronization is neglectible and the performance is almost always better than the approach with atomic operations.**

***Index Terms*—atomic operation, CAS, scalability, shared memory, redundant work, parallel work queue, parallel Breadth-First Search**

I. Introduction

Updating a shared data structure in a parallel program as for example the state check/update whether a vertex in a graph is visited or not [1] is usually done on an application level with some sort of high-level atomic update operation. In OpenMP [2] this could be realized lock-protected, in a critical section, or if syntax allows with an atomic pragma construct. Pthread-related API's (Application Programming Interface) [3] [4] [5] can use for example a mutex variable to protect access to a shared data structure and to ensure mutual exclusion of parallel threads. A general discussion on using different synchronization constructs in parallel and concurrent programming can be found in [6] and [7].

The implementation of such a high-level synchronization operation itself is done by the compiler or inside a runtime system often with one or even more atomic instructions (Compare-And-Swap, Atomic-Add, Fetch-And$-\Phi$, Test-And-Set, etc.) of the underlying processor architecture [8] [9] [10]. The general problem with atomic instructions of type read-modify-write is that these are rather costly compared to ordinary memory accesses and not really scalable on larger systems [11] [12] (see also Section IV for investigations on that). The time for *one* such atomic operation increases significantly under contention as the number of cores in a multi-core / multi-processor system gets larger. Therefore, frequently accessing shared data structures with atomic operations imposes a severe performance problem, especially on large parallel systems.

The use of such synchronized updates on shared data guarantees correct operations on the data when using multiple threads. But this strict enforcement is often not really necessary. An example is a work queue, where working threads insert new work items and idle threads remove such items to be worked on. But for certain algorithmic scenarios (e.g., within a certain program phase), a work item may be inserted even multiple times without violating the *overall* correctness of the algorithm, but only causing additional redundant work to be done as the same work item may exist multiple times in a queue. In such cases, the costly synchronized access could be completely removed and replaced with cheaper non-atomic accesses, but eventually introducing additional work to be done if work items get inserted multiple times.

An example for such a scenario is a Breadth First Search (BFS) for undirected graphs (see Section III for details). Many of the published parallel BFS algorithms iterate over a vertex frontier where the vertices of the current vertex frontier determine, which unvisited vertices are part of the following vertex frontier. In this scenario, adding a vertex twice in such a frontier generates more work to be done in the next level iteration but does not influence the correctness of the algorithm (see Section V for details). Another, more general scenario is the development of asynchronous algorithms [13] [14].

In this paper, a general optimization strategy is introduced that replaces costly atomic modifiers with cheaper read/write accesses. Necessary conditions are defined that need to be met to apply the technique. The motivation for this optimization technique and the evaluation is done using a concrete parallel BFS algorithm on large shared memory multi-core multi-processor systems with up to 64 cores. Factors are discussed that influence the amount of potential additional work and whether this additional work without any synchronized access trades off against the traditional synchronized access to a work

queue doing exactly the amount of work that is necessary.

The paper is organized as follows. After the introduction, the paper starts with an overview of related work. After this, a brief overview is given on parallel BFS algorithms; level synchronous BFS is an example where the new approach can be applied and will also be used in the evaluation of the new technique. In Section IV, evidence is given that certain atomic operations including Compare-And-Swap have scalability problems on larger systems. In Section V, the new approach is introduced first for a special scenario, a generalization of the technique follows in Section VI. After that, the experimental setup is pointed out, and then the new approach is evaluated in detail. The paper ends with a summary.

II. Related Work

There are several papers on certain aspects on the optimization of synchronization constructs in a wider sense. This includes, amongst others, reducing the number of consecutive mutex locks/unlocks [15] in a program and compiler optimizations for read/write barriers [16]. Furthermore, there are advanced synchronization techniques trying to minimize synchronization costs including RCU (Read-Copy-Update) [17], special monitors [18] and read-writer optimizations [19].

An interesting general approach to handle possible concurrent accesses to shared data structures is the concept of transactional memory (original concept paper [20]). This approach has some similarities with the approach introduced in this paper as both are optimistic: do a read-modify-write operation without a critical section and react only is something went wrong. The idea with transactional memory as well as in the new approach discussed later in this paper is that the bad thing happens rather seldom. Transactional memory detects the problem and, depending on the Application Program Interface (API) in use, rolls back the whole transaction and restarts the operation. The technique proposed in this paper instead ignores the problem (and does not even detect the problem) and has more work to do in the remaining execution of the algorithm. Transactional memory is implemented on a hardware level in recent processors (IBM processors PowerPC for BlueGene/Q [21] and System z [22]; Intel Haswell [23]).

Lock-free and wait-free data structures are often proposed as a way to reduce/avoid synchronization problems (priority inversion, deadlock) that may occur using mutual exclusion or other blocking synchronization constructs. Starting with the fundamental idea by Leslie Lamport [24], many papers followed on certain aspects, e.g., [25] [26] [27] [28] [29] [30] [31] [32] [33] [34]; see [35] for a comprehensive view. As lock- and wait-free data structures are internally often realized with (as will be shown, non-scalable) atomic Compare-And-Swap operations, similar ideas as presented in this paper might be interesting in that area, too.

In [36], a parallel graph algorithm for the construction of a spanning tree is discussed using mutual exclusion and lock-free data structures. The authors discuss the problem of overlapping work and how to ensure that every work item (newly visited vertex of the graph) is handled by one thread only using atomic Test-And-Set operations. In our paper, it is proposed the other way to allow that a work item may be handled by more than one thread and avoiding even a test-and-set operation that is still necessary in [36].

Reference [6] gives an overview of different aspects on related topics. [37] shows a similar benign race as ours in a parallel BFS algorithm, but without analyzing the influence of that in detail.

III. Parallel Algorithms for BFS

Parallel level-synchronous BFS algorithms will be used as a motivating application as well as in the evaluation of the new idea that is introduced later in detail. Therefore, a short introduction to parallel BFS follows. Breadth-First Search 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 single-source shortest paths, minimum spanning tree, connected components, bipartite graphs, maximum flow, and others [38] [39]. 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 [40].

In the application scenario used for the examination, undirected graphs $G = (V,E)$ are of interest, 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 $depth(G, v_0)$ the depth of a graph G is denoted defined as the maximum depth of any vertex in the graph *relative to the given source vertex* v_0. Please be aware, that this may be different to the diameter of a graph, the largest distance between *any* two vertices.

The problem of Breadth First Search 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 [38] [39]. The computational complexity is $O(|V + |E|)$.

If one tries to design a parallel BFS algorithm, different challenges might be encountered. As the computational density of BFS is rather low, BFS is bandwidth limited for large graphs and therefore memory bandwidth has to be handled

with care. For a similar reason in cache coherent NUMA systems (Non-Uniform Memory Access [41]), 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 private lower-level caches.

```
1:  function BFS(graph g, vertex source)
2:    var
3:      d, distance vector of size |V|. Initial values: ∞
4:      current,next, vertex container. Initially empty
5:    end var
6:    d[source] ← 0
7:    current.insert(source)
8:    while current is not empty do
9:      for all v in current do
10:       for all neighbours w of v do
11:         old = CompareAndSwap(d[w],∞,d[v]+1)
12:         if old == ∞ then
13:           next.insert(w)
14:         end if
15:       end for
16:     end for
17:     Barrier
18:     swap current with next
19:   end while
20:   return d
21: end function
```

Fig. 1: Parallel BFS with an atomic Compare-And-Swap-operation

In BFS algorithms housekeeping has to be done on visited / unvisited vertices with several possibilities how to do that. Some of them are based on special container structures for vertex frontiers where information has to be inserted and deleted. Scalability and administrative overhead of these containers are of interest. Generally speaking, these approaches deploy two identical containers (current frontier, next frontier) whose roles are swapped at the end of each level iteration. Fig. 1 shows this in a rather straightforward formulation with an atomic Compare-And-Swap (CAS) operation in an inner loop (line 11) to detect and update unvisited vertex neighbors. In this atomic operation, a vertex w is checked whether it is visited already ($d[w] \neq \infty$), and if not, marks the vertex as visited. Based on this knowledge, only an unvisited vertex gets inserted into the next vertex frontier. After all vertices in the current container are visited, all threads wait at a barrier before work on the next container / frontier gets started (level iteration). This version can be further optimized using chunked lists for every thread. The insert operation of a new vertex into a thread-local chunk can be done in a non-atomic way. But the construction of a global list from thread-local chunks (i.e., the insertion of each chunk into a global list) must still be done in a synchronized way. But as this is done only if a chunk gets full, this is not the critical operation of this algorithm but the detection of whether a vertex is visited or not in line 11. 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 [42] [43]. Reference [44] contains an overview and evaluation of several parallel BFS algorithms.

For level synchronized 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 [45] [46]. This approach deploys a static one dimensional partitioning of the graph's vertices and therefore supports data locality.

Level synchronous algorithms are quite easy to understand and often to realize, too. With certain additional optimizations performance is often very good [44]. This type of BFS algorithm is used as an instance of the problem scenario where the newly proposed technique can be applied. It should be mentioned, that for certain classes of graphs (e.g., high diameter graphs) parallel algorithms exist that perform better than the level synchronous approach [47] [48] [49] [50].

TABLE I: PROCESSORS AND SYSTEMS USED

name	Intel-IB	Intel-SB	AMD-IL	AMD-MC
processor:				
manufacturer	Intel	Intel	AMD	AMD
CPU name	E5-2697	E5-2670	Opteron 6272	Opteron 6168
architecture	Ivy Bridge	Sandy Bridge	Interlagos	Magny Cours
frequ.[GHz]	2.7	2.6	2.1	1.9
system:				
memory [GB]	256	128	128	32
# CPU sockets	2	2	4	4
n-way parallel	48	32	64	48

IV. Performance Problem of Atomic Read-Modify-Write Operations on Large Parallel Systems

Atomic operations in a higher level parallel API for shared memory systems as mutual exclusion, atomic update, locks, Compare-And-Swap etc. [6] [7] [51] [52] are usually mapped on shared memory systems to atomic instructions that the underlying processor architecture provides [8] [9] [10]. Some of these atomic instructions are by itself rather costly compared to a simple memory access if no contention exists. For exampe, embedded in a function call and executed by one thread on an Intel E5-2697 CPU, an atomic CAS operation on a 64 bit data type takes 7 ns compared to 3 ns that a compound non-atomic read-test-write operation takes. Table I describes in detail and names the systems used in the following.

But under contention, if multiple threads concurrently access a shared state with such instructions, the cost *per operation* increases significantly for certain types of atomic operations. This is especially true for read-test/modify-write type operations like Compare-And-Swap and Fetch-And-Add. And the contention penalty gets higher the more CPU sockets a system has [11] [12]. Fig. 2 shows the cost for one 64-bit atomic Compare-And-Swap operation (of type read-modify/test-write) on different shared memory systems dependend on the number of threads utilised. The number of threads used does not exceed the degree of hardware parallelism a system under

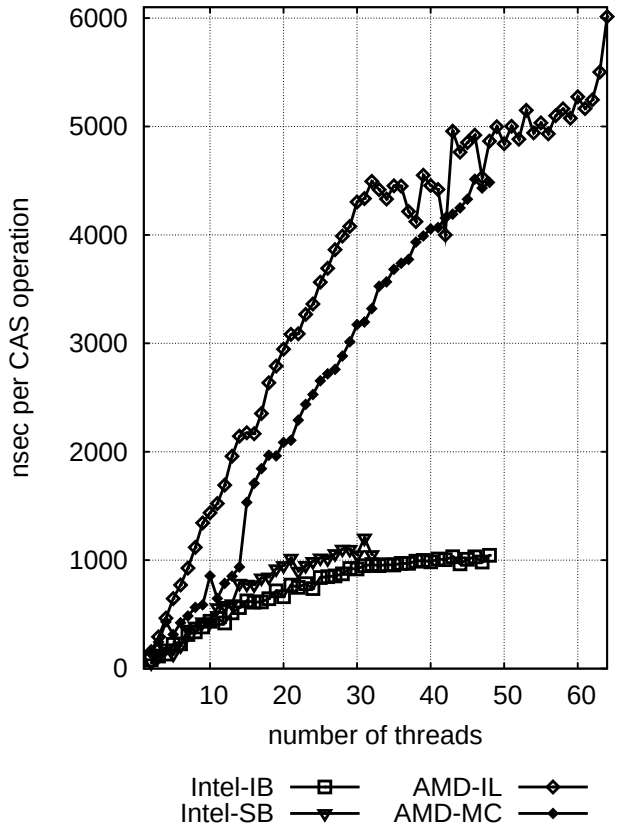


Fig. 2: Cost per Compare-And-Swap operation on different parallel systems.

consideration has. Therefore, every thread is mapped (in an operating system specific manner) to a different core of the system and is always runnable (ignoring operating system effects like interrupt handling). In this test, p threads do in parallel in a loop $n = 1,000,000$ atomic CAS operations each. The test was executed on parallel systems of different generations of processors, processor manufacturers, degree of parallelism, and number of processor sockets. Similar results can be seen for other atomic read-test/modify-write operations, too, e.g., atomic Fetch-And-Add. As can be easily seen from the results, the overhead on larger systems with four sockets (the AMD-based systems used) is significantly higher than on smaller systems with two sockets (the Intel-based systems used). The minimum time for a CAS operation is between 7 and 22 ns without contention on the systems used, the time gets as high as approx. 1,000 ns per operation on the two socket systems and up to approx. 6,000 ns on the four socket systems under heavy contention. This means that performance problems with atomic operations hurt on larger systems with more sockets even more than on smaller systems.

V. Alternative to Atomic Accesses

In this section, a technique is proposed to replace the costly and non-scalable atomic accesses with cheaper non-atomic accesses. First, the technique will be motivated on the concrete BFS application introduced in Fig. 1. It will be shown, that under certain conditions atomic operations can be traded against additional work and therefore traded against additional overhead. Then a discussion follows, what factors influence this possible overhead. And finally, in Section VI, a generalization is given under what circumstances the technique can be applied generally.

A. Optimizing BFS

Looking at the formulation of the parallel BFS algorithm in Fig.1, an atomic CAS-Operation is used in line 11 to check whether the neighbour vertex w is unvisited ($d[w] = \infty$), and if so, replace the depth-value of w with the depth value of the current vertex v incremented by one. And if the neighbour vertex w was unvisited, additionally insert w into the next vertex frontier. The replacement of the value ∞ by a non-∞ value (the depth value) marks the vertex as visited. The CAS operation guarantees, that every vertex is inserted exactly once into a vertex frontier (detection and mark of visitedness). Without the atomic operation, a race condition [53] exists on $d[w]$. Replacing the critical operation with a non-atomic code results in Fig. 3 (only relevant parts are shown here).

```
1: for all neighbours w of v do
2:   if d[w] = ∞ then
3:     d[w] = d[v] + 1
4:     next.insert(w)
5:   end if
6: end for
```

Fig. 3: Parallel non-atomic BFS (relevant part)

The code of interest is in lines 2 and 3, that was previously guarded by the CAS-operation. There are two possibilities when executing this code in parallel:

1) Between the read access $d[w]$ in line 2 and the completion of the write access in line 3 no other thread accesses $d[w]$. In this case, there is *no problem* with this version, the vertex w is inserted exactly once in a vertex frontier as before. But see additionally the discussion of the appropriate memory model below.
2) More than one thread detects for a certain vertex w_x that w_x is unvisited (i.e., $d[w_x] = \infty$) before any of the other threads can change the $d[w_x]$ to some visited value. In this case, the vertex w_x gets inserted twice or even more into the next vertex frontier.

The insert operation in line 4 has to be done with care as this might be done concurrently by multiple threads. As this has to be handled in all version similar and is not really critical in all versions of discussion, this is not further discussed here. It is important to state that even the second case produces *no wrong results* as *any thread* that detects that $d[w_x]$ is unvisited, writes into $d[w_x]$ in the next step the value $d[v]+1$ that is *the same value for all threads* in one level iteration. Therefore, correctness is guaranteed in our scenario even if multiple threads concurrently detect that the same vertex is unvisited. But, as stated above, in such a case the vertex w_x is inserted twice or even more into the next vertex frontier and due to

that, generates more and redundant work in the next level iteration. Working later on a vertex multiple times is again no correctness problem. The resulting depth values for all vertices are the same in a level-synchronous algorithm, independend on how many times a vertex gets worked on.

Inspecting the generated assembler code for lines 2 and 3 of the code given in Fig. 3, the read access to $d[w]$ in line 2 (i.e., a load instruction) and the write access to $d[w]$ in line 3 (i.e., a store instruction) are nearby instructions in the code sequence. These inspections were done for different compilers (GNU gcc, Intel icc, PGI pgcc) and it was found that this observation is more or less invariant of the compiler used for the given code sequence (and it would be curious if this observation would not hold for this code example). With an assumption, that a thread is not suspended during execution, the time window between the two instructions is therefore rather small (few cycles in practice). This assumption will be mostly true for many real scenarios where parallel programs get executed, e.g., running OpenMP programs on a dedicated system with not more threads than processor cores available.

Another aspect in this discussion is the memory consistency model in use [54] [55] [56] [57]. In a strict memory consistency model, it is guaranteed, that a read operation always retuns the value of the last write operation to that memory location. But todays, all memory consistency models in practical use (e.g., [5] [4] [2] [58]) are rather relaxed and the compiler may buffer the value of $d[w]$ in a register, a processor core may buffer that value in write buffers, or the new value is not propagated between different processors soon, etc. This can enlarge the time window for problems substantially even under the assumption made above that a thread is not suspended. A programmer may insert an appropriate flush operation before line 2 and after line 3 such that all threads / processors are forced to read / write $d[w]$ to / from main memory in the corresponding operation. But dependend on the implementation of such a flush-operation, this could lead to substantial additional overhead as this is for example done inside an inner loop iteration in the application example given.

B. Factors Influencing Additional Work

The question of interest is now, whether the relaxation using non-atomic modifications to $d[w]$ as given in Fig. 3 (which surely is faster than a CAS-operation as explained in Section IV) pays off. Due to the fact that without an atomic CAS operation a vertex might get inserted multiple times into the next vertex frontier, the question is whether the amount of work to be done might be increased substantially. The amount of additional work to be done will be influenced generally speaking by:

1) the problem time window in relation to the time threads spend in non-critical code. This is influenced by the generated code sequence and implemented consistency model as discussed above.
2) the number of threads in use, i.e., the number of concurrent parties.
3) the problem data influencing access collisions, i.e., in our case the topology of the graph (vertex degrees, shared neighbours)

The larger the time window is that another thread may see the vertex in question as unvisited, and the more threads are participating, and the more vertices have connections to the unvisited vertex, the higher the probability that additional work is generated.

VI. Generalization of the Technique

Although the motivation of the technique was given here in the context of a parallel BFS algorithm, the technique itself is not specific to BFS and can be generalized. Therefore, the suggestion is to replace costly atomic operations with cheaper simple load/store operations without influencing the correctness of the algorithm but probably doing more / redundant work. The hypothesis is that especially on large shared memory systems with many concurrent threads this technique pays off.

Looking in a more abstract way on the suggested technique, there is predicate $p : X \to \{true, false\}$ for some set X. If the result of the predicate is true, there is a state-change operation $c : X \to X$ for the same $x \in X$ that the predicate was applied to, followed by some operation $f : Y \to Y$ for some set Y. The generalized application scenario is therefore given in Fig. 4.

```
1: if p(x) then
2:     c(x)
3:     f(y)
4: end if
```

Fig. 4: Generalized Scenario

As multiple threads might execute the predicate p, multiple threads might detect the same true condition and therefore execute $c(x)$ and $f(y)$ subsequently and also redundantly. As a consequence, the operation c and f must be both idempotent to ensure that executing the state-change function f multiple times does not influence the correctness. A function $g : A \to A$ is called *idempotent* if $g \circ g = g$, i.e., $\forall a \in A : g(g(a)) = g(a)$.

For the BFS example, the p-Operation is the test $d[w] = \infty$ and the c-operation changes the state of $d[w]$ to the same value if executed multiple times for the same vertex. The f-operation is the insertion of the vertex into the next vertex front. The c-operation as well as the f-operation meet the requirements for the two operations, respectively. As $d[w]$ is assigned the same value, this is an idemponent operation. And the multiple insertion of a vertex into the next vertex frontier is also (semantically) a idemtpotent operation, because the *result* of working on the next vertex frontier will be the same, independent how many times a vertex gets inserted into a frontier.

VII. Experimental Setup

In this section, the technique introduced in Section V is systematically examined with the concrete scenario of parallel BFS. Three factors were identified that may influence the

performance of an application using the new technique due to additionally generated work. All factors are examined in detail.

A. Algorithm Versions

The general algorithmic approach for parallel BFS chosen for this discussion was already given in Fig. 1 in an easy to formulate version. The concrete realization to handle vertex fronts was done for this evaluation with chunked array based lists where each thread inserts a new vertex unprotected into a thread-private chunk of size 128. If such a chunk gets filled, the chunk is inserted into a global list in a protected way. The insertion of a whole chunk into the global list is done in all algorithm versions examined with one atomic operation. But the influence of that atomic operation is rather small as only whole chunks get inserted and not single vertices.

In the first version named *atomicBFS*1 (see Fig. 5), every thread uses a CAS operation as described in Fig. 1 to detect unvisited vertices and updates them accordingly. This guarantees, that every vertex is inserted exactly once in a vertex frontier. But on the other side, *every* check is done atomically even on vertices that were visited already, even in any previous level iteration. This is a save but somewhat naïve implementation.

This last aspect can be optimized for many program kernels easily with a standard optimization technique for parallel programs in prefixing the expensive CAS-operation with a normal read operation followed by the CAS-operation *only*, if the test was successful, i.e., a test-and-test-and-set operation (see Fig. 6). This technique has significant advantages if vertices get visited many times (e.g., graphs with high vertex degrees). Then, only the first visit must be atomic, all other accesses would detect that the vertex is visited already. This optimization technique is also used in the OpenMP reference implementation of the Graph500 benchmark [42] for BFS. This optimized version is named *atomicBFS*2. In this version, all vertices already visited are no longer handled with a CAS operation. The discussion of the performance effect of this optimization is given later in detail.

The third approach named *nonatomicBFS* does not use atomic operations for the detection of unvisitedness, but rather uses the technique proposed (see Fig. 7). Therefore, a vertex may be inserted more than once in the next vertex frontier. The main difference to the other two versions is therefore that the detection of an unvisited vertex and the subsequent update to a visited state is no longer done atomically but rather with simple read/write accesses including the possibility of multiple insertions of a vertex as multiple threads may see concurrently a vertex as unvisited. Further algorithmic optimizations different to that discussed here and a general overview of parallel BFS algorithms can be found in another paper [44]. There is also shown, that there are better but more complex algorithms for the parallel BFS problem for large shared memory systems. In this paper, only the discussion atomic operations vs. redundant work is of interest, and therefore the *relative* comparism of the introduced three versions is sufficient for that.

To filter out unrelated effects, all test runs were repeated 5 times and the best result out of these 5 results was taken as the final result of a test.

```
1: for all neighbours w of v do
2:    old = CompareAndSwap(d[w], ∞, d[v] + 1)
3:    if old == ∞ then
4:       next.insert(w)
5:    end if
6: end for
```

Fig. 5: Kernel for *atomicBFS*1

```
1: for all neighbours w of v do
2:    if d[w] ≠ ∞ then
3:       old = CompareAndSwap(d[w], ∞, d[v] + 1)
4:       if old == ∞ then
5:          next.insert(w)
6:       end if
7:    end if
8: end for
```

Fig. 6: Kernel for *atomicBFS*2

```
1: for all neighbours w of v do
2:    if d[w] == ∞ then
3:       d[w] = d[v] + 1
4:       next.insert(w)
5:    end if
6: end for
```

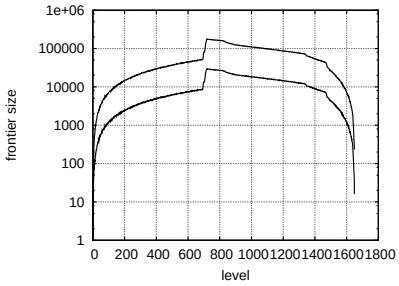
Fig. 7: Kernel for *nonatomicBFS*

B. Factors Influencing Overhead

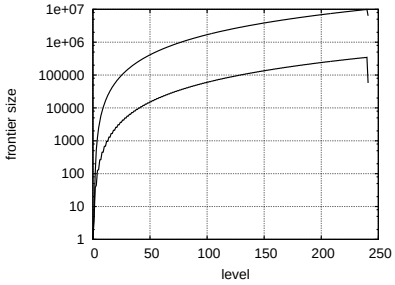
As discussed already in Section V, the first factor influencing the probability of multiple insertions is the time window related to the time spent in non-critical code. Although the BFS algorithm has only few instructions between the read and write operation on the critical data, there is not much work to do in the non-critical part (just the insert operation) executing the critical part with high frequence and therefore increasing the probability for collisions. Therefore, BFS is an example for a rather problematic algorithm in this sense.

The second factor influencing the probability of double insertion is the degree of parallelism. Different parallel systems were used in the tests as described already in Table I. The largest one is a 64-way AMD-6272 Interlagos based system with 128 GB shared memory organised in 4 NUMA nodes (i.e., 4 CPU sockets). A second AMD-based systems has also four sockets, while the remaining two systems are two-socket systems (see Table I for details and names to refer to a specific system).

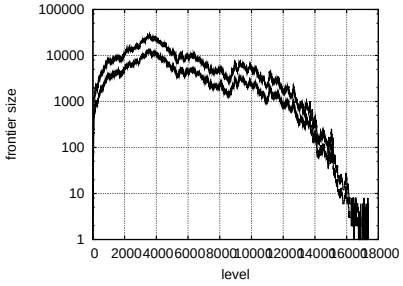
The third factor influencing additional work is the probability of a data collision, i.e., in the given application the probability that two vertices with the same depth share a common unvisited neighbor in the graph. Only unvisited neighbours leed to an atomic operation in version *atomicBFS*2 and to



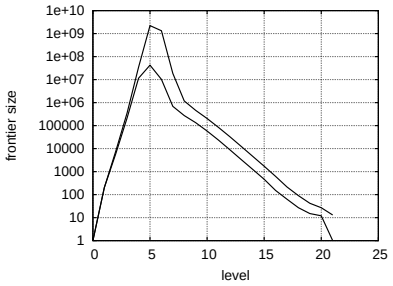
(a) Frontiers for delaunay



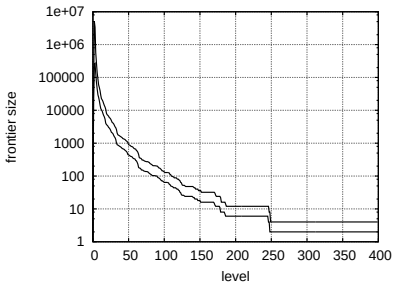
(b) Frontiers for nlpkkt240



(c) Frontiers for road-europe

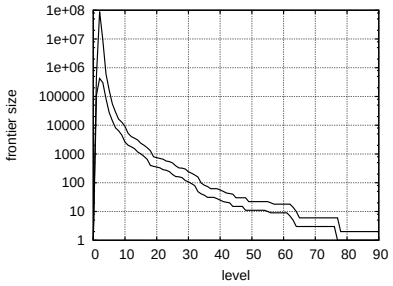


(d) Frontiers for friendster

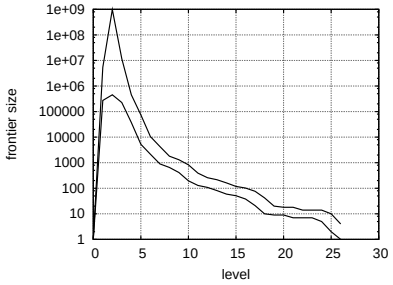


(e) Frontiers for R-1M-10M-57

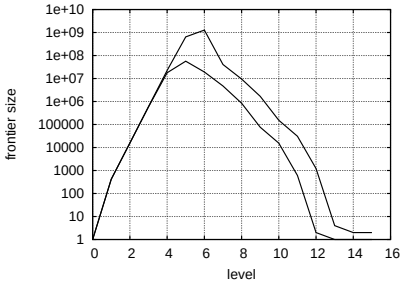
Fig. 8: Vertex and edge frontier sizes of selected graphs (part 1). Upper curve is edge frontier, lower curve is vertex frontier.



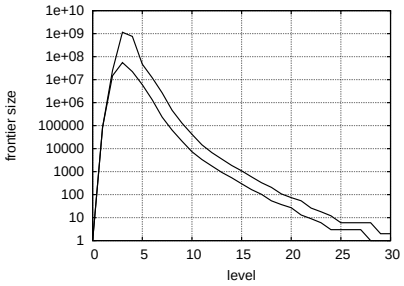
(a) Frontiers for R-1M-100M-57



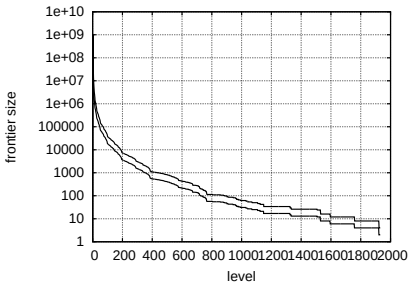
(b) Frontiers for R-1M-1G-57



(c) Frontiers for R-100M-2G-30



(d) Frontiers for R-100M-2G-45



(e) Frontiers for R-100M-2G-57

Fig. 9: Vertex and edge frontier sizes for selected graphs(part 2). Upper curve is edge frontier, lower curve is vertex frontier.

TABLE II: CHARACTERISTICS FOR USED GRAPHS

graph name	$\|V\| \times 10^6$	$\|E\| \times 10^6$	degree avg.	degree max.	graph depth
delaunay [59]	16.7	100.6	6	26	1650
nlpkkt240 [60]	27.9	802.4	28.6	29	242
road-europe [59]	50.9	108.1	2.1	13	17345
friendster [61]	65.6	3612	55	5214	22
R-1M-10M-30	1	10	10	107	11
R-1M-10M-45	1	10	10	4726	16
R-1M-10M-57	1	10	10	43178	400
R-1M-100M-30	1	100	100	1390	9
R-1M-100M-45	1	100	100	58797	8
R-1M-100M-57	1	100	100	530504	91
R-1M-1G-30	1	1000	1000	13959	8
R-1M-1G-45	1	1000	1000	599399	8
R-1M-1G-57	1	1000	1000	5406970	27
R-100M-1G-30	100	1000	10	181	19
R-100M-1G-45	100	1000	10	37935	41
R-100M-1G-57	100	1000	10	636217	3328
R-100M-2G-30	100	2000	20	418	16
R-100M-2G-45	100	2000	20	85494	31
R-100M-2G-57	100	2000	20	1431295	1932
R-100M-4G-30	100	4000	40	894	15
R-100M-4G-45	100	4000	40	180694	31
R-100M-4G-57	100	4000	40	3024348	1506

a possible double-insertion in version *nonatomicBFS*. This factor is mainly influenced in the given scenario by the graph topology / degree distribution. To examine this influence, several large graphs were used from different application domains including real graphs from social networks, road networks, optimization problems, and triangulation graphs. The graph instances were taken from then DIMACS-10 challenge [59], the Florida Sparse Matrix Collection [60], and the Stanford Large Dataset Collection [61]. The graph `friendster` and larger RMAT-graphs could not be used on all systems due to memory requirements. Additionally, synthetically generated pseudo-random graphs were used that guarantee certain topological properties. R-MAT [62] is such a graph generator with parameters a,b,c influencing the topology and clustering properties of the generated graph (see [62] for details). R-MAT graphs are mostly used to model scale-free graphs. For the evaluation tests, graphs of the following classes werde used:

- 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, Delaunay graphs representing Delaunay triangulations of random points (`delaunay`) and a graph for a 3D PDE-constraint optimization problem (`nlpkkt240`) are used.
- 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, a real social network (`friendster`), and synthetically generated Kronecker R-MAT graphs are used, the later 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 named 57 is $a = 0.57, b = 0.19, c = 0.19$.

All 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 II. For a general discussion on degree distributions of R-MAT graphs see [63].

C. Factors Influencing Performance and Scalabilty

To interpret results in the following section, frontier sizes during the level-synchronous execution of the BFS algorithm will be given in relation to the level number as an additional information (see Figs. 8 and 9). The *edge frontier size* gives the number of outgoing edges from vertices in the current frontier, i.e., the number of vertex *candidates* that have to be checked for inclusion into the next frontier. On the other side, the *vertex frontier size* gives the number of unique vertices that get inserted into the next vertex frontier, i.e., the vertex was checked, found unvisited, and then sucessfully inserted. The edge frontier size is therefore the amount of checks to be done (in algorithm version *atomicBFS*1 with a CAS operation, in the other versions by a simple read operation), and the vertex frontier size is the amount of actual insertions into the next frontier (in version *atomicBFS*2 as part of the CAS, in version *nonatomicBFS* with a simple write). The edge frontier size is always at least as large as the vertex frontier size. In the figures, the edge frontier size is always the upper curve.

Setting this frontier information in relation to the performance numbers, a large difference between edge frontier size and vertex frontier size in a level iteration means that many atomic checks were made in version *atomicBFS*1 that did not lead to an unvisited neighbor vertex / insert operation. On the other side, if the difference between vertex and edge frontier size is small, the difference between the two atomic algorithm versions should be less as most of the atomic operations are executed in both atomic versions.

Figs. 8 and 9 show frontier sizes during each level. Please be aware that the y-axis has a logarithmic scale. The higher a number for the vertex frontier is, the more parallel work is available. A frontier size of 1,000 or even less on a parallel system with 64 threads all working in parallel on this problem means a severe performance limitation.

All BFS algorithms introduced here are limited for large graphs by memory bandwidth demands, especially when using many threads. This means that for many large graphs and using many threads, the effects under discussion here may be hidden by memory bandwidth restrictions [44]. Additionally, if there is not enough parallelism available (small vertex frontier at any level iteration), performance is again limited in all versions using many threads. In such situations, algorithms *atomicBFS*2 and *nonatomicBFS* will most likely perform very similar. In Section VIII-C, statistical filters are used to handle this in the discussion.

VIII. Evaluation Results

In the following, performance results are discussed comparing the three algorithm versions on the different parallel systems and with different input data as specified in Section VII. Not all results can be shown here in detail. Rather, the influence of the stated factors is discussed, results are summarized where reasonable, and overall statistics are presented. Additionally, the amount of overhead for *nonatomicBFS* is discussed. Performance number for BFS will be given as a rate Million Traversed Edges per Second (MTEPS), a usual measure for BFS performance [42] (the higher, the better).

A. Absolute Performance Results

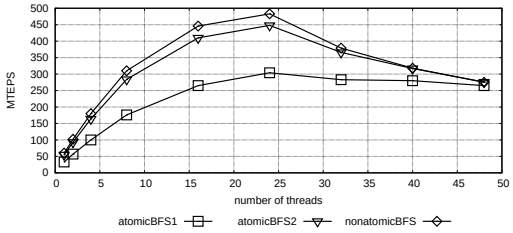
Figs. 10 - 13 show absolute performance results for the three algorithm versions of investigation on the different parallel systems using selected data sets. The performance limitation or even degradation using many threads (especially with graph `road-europe`) is caused by the limited parallelism (see Figs. 8a and 8c for that) or memory bandwidth restrictions. More sophisticated BFS algorithms that are out of the scope of this paper can handle that more efficient. Details on that can be found in [44].

For all graphs shown other than `road-europe` the algorithm version performing worst is the algorithm version *atomicBFS*1 as with *every* access to $d[w]$ in the relevant code section an atomic CAS operation is executed. This is true on all systems used and with nearly all thread counts. The performance difference to the other two algorithm versions is very high, if many of the atomic operation were done unneccesarily, i.e., a vertex of investigation was visited already before. This is the case if the difference between edge and vertex frontier is large. And the performance difference is large as long as no other effects (limited parallelism, memory bandwidth restriction) superimpose this effect. Both other algorithm versions use no CAS at all (*nonatomicBFS*) or only if a vertex has been seen in a non-atomic pre-test as unvisited (*atomicBFS*2). This behaviour clearly underpins the central message that atomic operations should be avoided whenever possible.

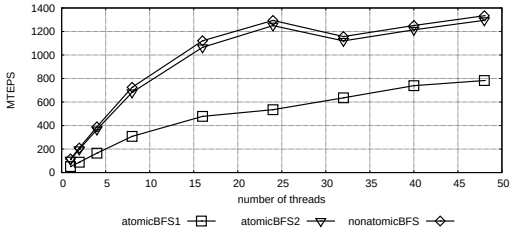
B. Relative Improvements

As algorithm version *atomicBFS*1 has in most configurations severe performance limitations, a closer look will be done on the other two version only: *atomicBFS*2 using CAS only if necessary and *nonatomicBFS* using the introduced technique of avoiding atomic operations at all.

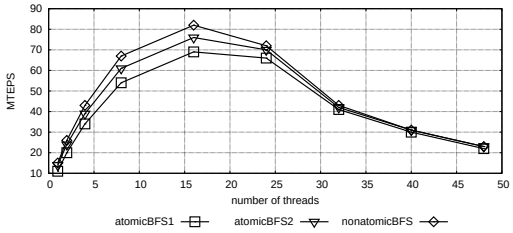
Fig. 14 shows relative performance improvements in percent of algorithm version *nonatomicBFS* without atomic operations relative to algorithm version *atomicBFS*2 with (already optimized) atomic updates. A positive value means that the non-atomic version performs better than *atomicBFS*2. The figures show that for many threads the difference between the two versions of discussion is rather small (and then often below the accuracy of measurement). The reason for that was given already: with this rather simple BFS algorithm versions, for many threads and large graphs, memory bandwidth and/or limited parallelism is the limiting factor, and not the atomic



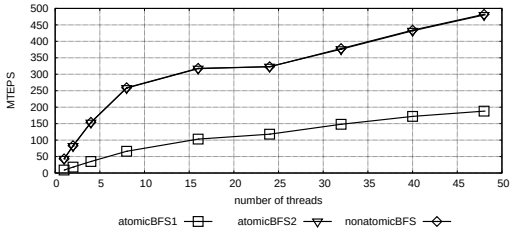
(a) delaunay



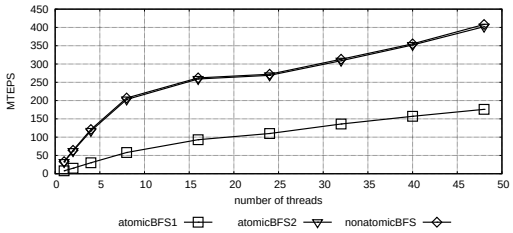
(b) nlpkkt240



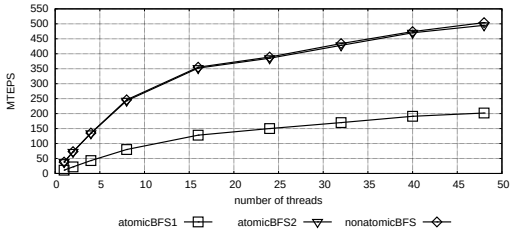
(c) road-europe



(d) friendster

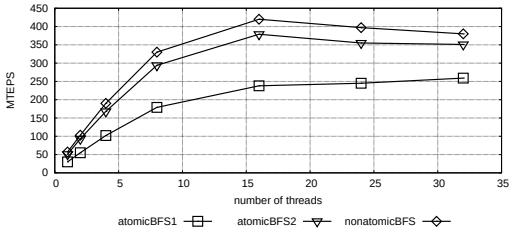


(e) R-100M-2G-45

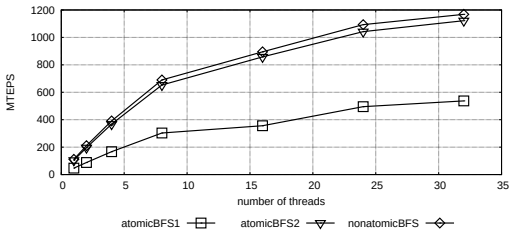


(f) R-100M-2G-57

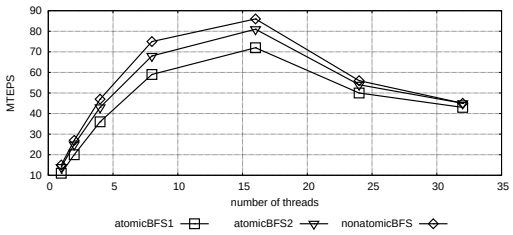
Fig. 10: Performance data on system Intel-IB.



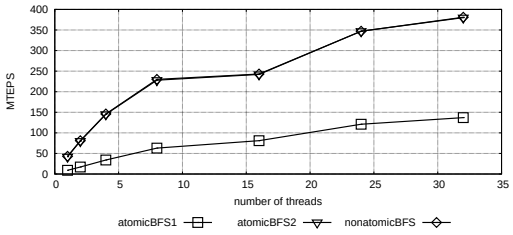
(a) delaunay



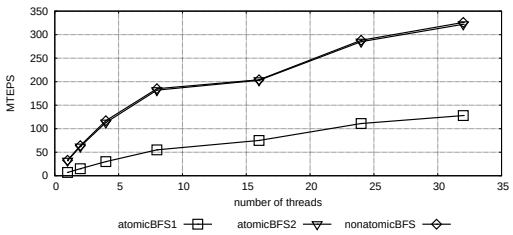
(b) nlpkkt240



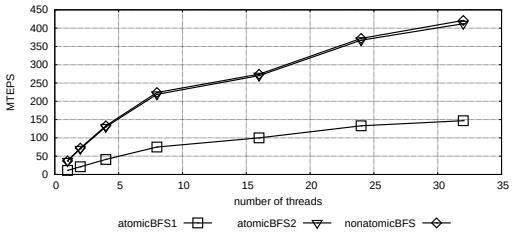
(c) road-europe



(d) friendster

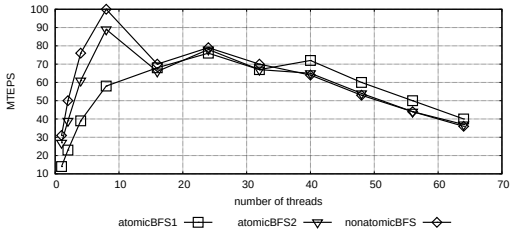


(e) R-100M-2G-45

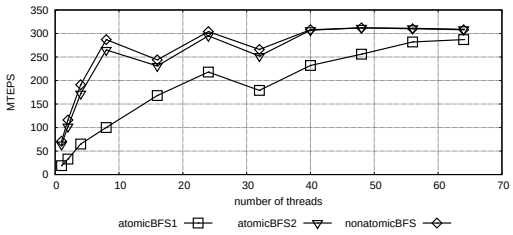


(f) R-100M-2G-57

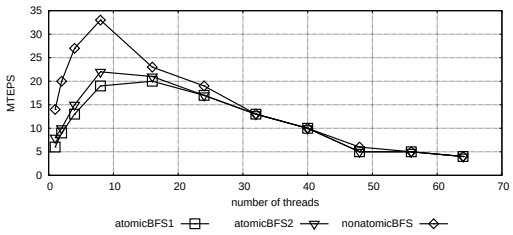
Fig. 11: Performance data on system Intel-SB.



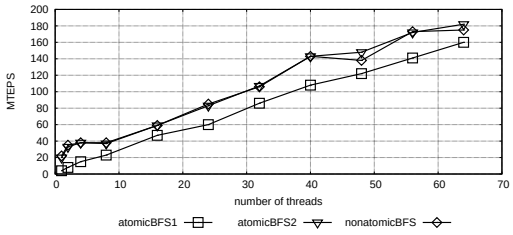
(a) delaunay



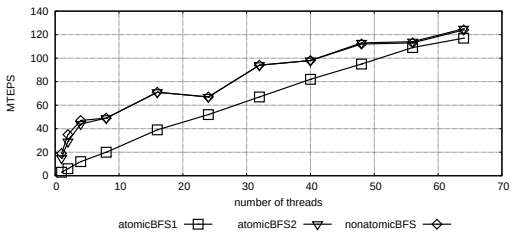
(b) nlpkkt240



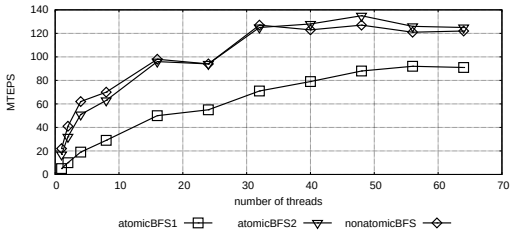
(c) road-europe



(d) friendster

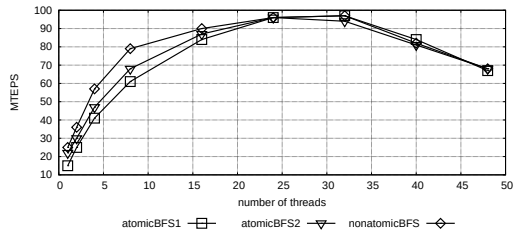


(e) R-100M-2G-45

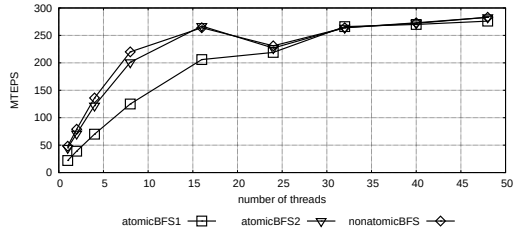


(f) R-100M-2G-57

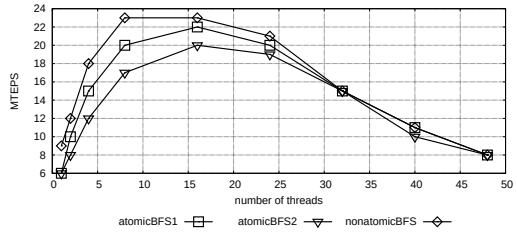
Fig. 12: Performance data on system AMD-IL.



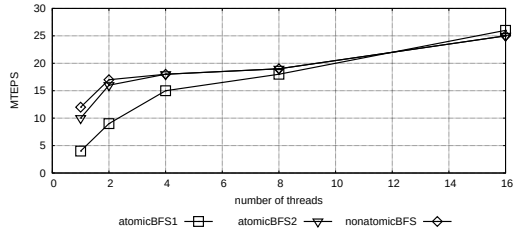
(a) delaunay



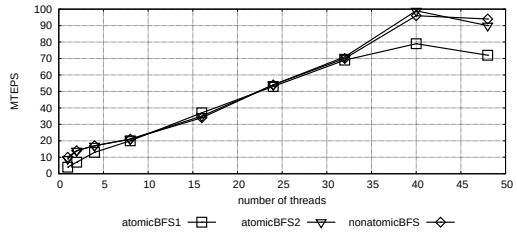
(b) nlpkkt240



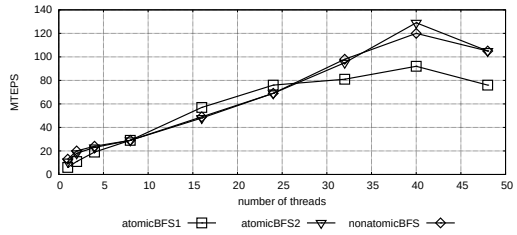
(c) road-europe



(d) friendster



(e) R-100M-2G-45



(f) R-100M-2G-57

Fig. 13: Performance data on system AMD-MC.

accesses. But the smaller the number of threads used, the higher is the difference between the two version in the favour of *nonatomicBFS*. This is because the atomic accesses are the main performance problem, and not memory bandwidth or limited parallelism. Often, the single thread case has the largest relative improvement for the non-atomic version. This is mostly related to the additional overhead for atomic operations (see Section IV) compared to a cheaper non-atomic read and write operation.

On the two AMD systems with 4 CPU sockets the non-atomic version shows more improvements than on the 2-socket Intel systems. The reason for that is the higher overhead for CAS operations on systems with more sockets (see Section IV and especially Fig. 2).

C. Statistical Analysis

In Section VII, the test setup was described. Tests were done on 4 different systems, with different thread numbers depending on the available parallelism of a system, and different input graphs. In total, 761 configurations were tested. So far, selected results were presented in detail. In this section, statistics on all results summarize the performance results.

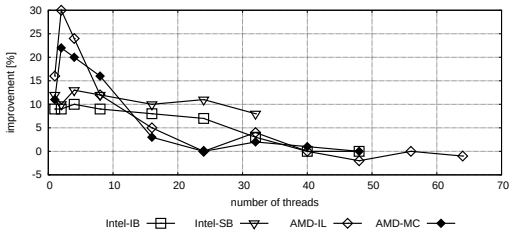
Although all tests were repeated several times (see Section VII for details), there are variations in runtime caused by several and partly non-deterministic factors in a complex parallel system. To leave these artefacts out of the discussion, we define a difference between two results that is below 3% as within the accuracy of measurement and therefore not related to the discussion here.

Fig. 15 shows a statistical analysis of all 761 results. Five classes of test results are shown, given for each system:

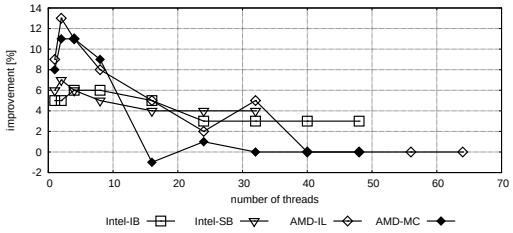
- < -10: where the atomic version *atomicBFS*2 performoms significantly better (better than 10%) than *nonatomicBFS* (in total 2 out of 761 instances)
- $-10 <= x < -3$: where the atomic version *atomicBFS*2 performoms better than *nonatomicBFS* (in total 20 instances)
- $-3 <= x < 3$: where differences are within the accuracy of measurement (in total 406 instances, most of them with large threads counts)
- $3 <= x < 10$: where the non-atomic version *nonatomicBFS* performoms better than *atomicBFS*2 (in total 220 instances)
- $>= 10$: where the non-atomic version *nonatomicBFS* performoms significantly better than *atomicBFS*2 (in total 113 instances)

It can be seen that large improvements (more than 10% performance increase) are mainly on the 4-socket systems while the 2-socket systems show increases that mostly lie below 10%.

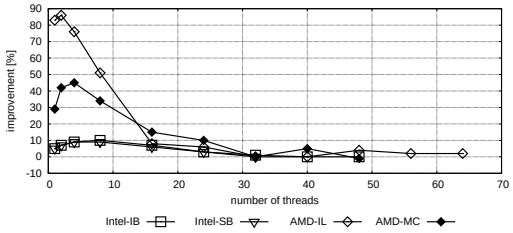
Summarizing the results over all systems, in approx. 2.9% of all test cases the new approach performs worse than *atomicBFS*2, in approx. 43.8% of all test cases the new approach performs better, and in the rest of the test cases the two versions performed rather similar (less than 3% difference). As can be easily seen from Figs. 10 - 13, this is



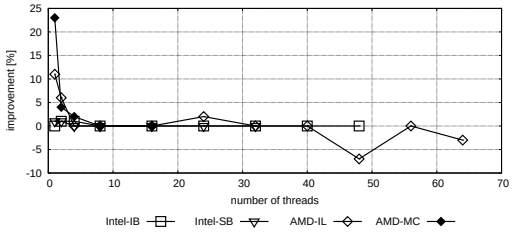
(a) delaunay



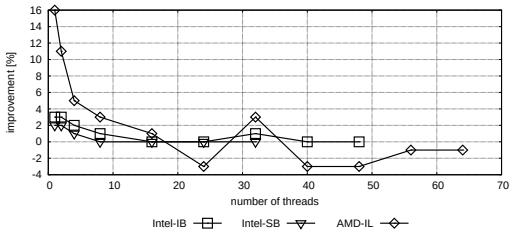
(b) nlpkkt240



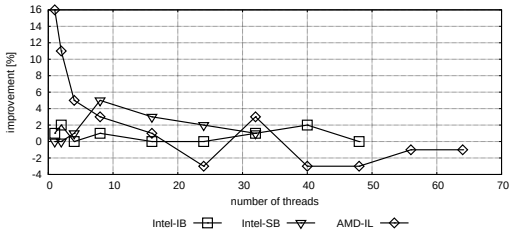
(c) road-europe



(d) friendster



(e) R-100M-4G-45



(f) R-100M-4G-57

Fig. 14: Relative improvements of non-atomic version *nonatomicBFS* compared to optimized atomic version *atomicBFS*2.

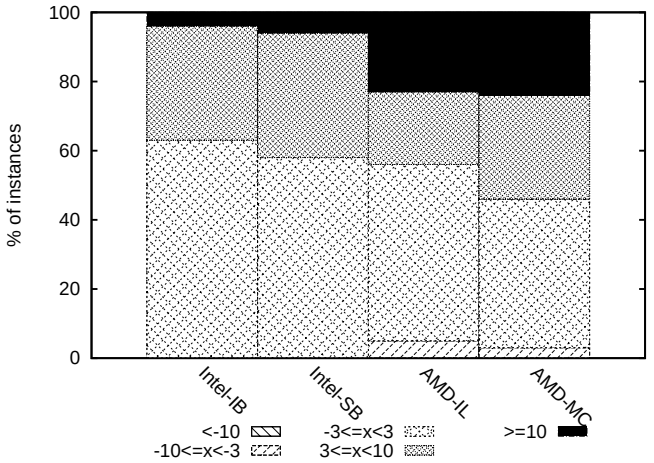
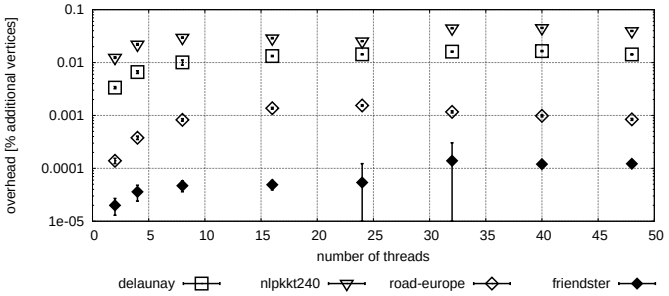
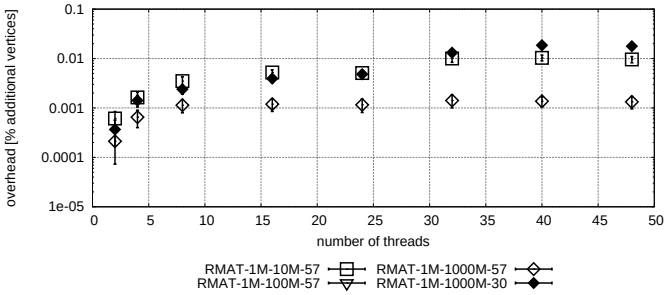


Fig. 15: Classes of performance improvements between *nonatomicBFS* and *atomicBFS*2. $x > 0$ are test instances, where *nonatomicBFS* was faster than *atomicBFS*2).



(a) Percentage of additional vertices (non RMAT graphs)



(b) Percentage of additional vertices (selected RMAT graphs)

Fig. 16: Overhead in percentage of additional vertices for selected graphs on the system Intel-IB, shown with error bars.

often the case when using many threads, where other effects then superimpose the discussion atomic / non-atomic.

D. Additional Overhead

The newly introduced technique avoids atomic operations in exchange with a probability of more work to be executed. To examine the amount of additional work, it was measured how many vertices get inserted multiple times in version *nonatomicBFS*, which is proportional to the additional and redundant work that is generated. The general factors influencing that were discussed already in Section V-B.

Fig. 16 shows exemplarily for the system Intel-IB the overhead for selected graphs including the worst case for that system. The program was run for this test 100 times. Shown in the figure is the average overhead in percent and the standard deviation as an error bar, for a specific number of threads respectively.

As can be seen, the probability increases slightly with more threads, but still this overhead is for the scenario used negligible. Even with 48-fold concurrency, there are very rare situations that lead to multiple insertions. In all tests executed – on all systems, with all number of threads, with all data sets, with different compilers – the additional overhead was in a neglectible range for the application used. In all tests the overhead in percent of *additional* vertices to be handled was always below 0.1 %, and most times even far less than that.

The maximum overhead seen on the system Intel-IB was 0.047 percent, or in absolute numbers, instead of 27,993,600 vertices to be inserted, with $nonatomicBFS$ 29,314,002 vertices were inserted. The difference in this worst case that happened was therefore 13,204 additional vertices. For RMAT-graphs, the difference was even always below 500 additional vertices in absolute values.

IX. Conclusions

In this paper, it was proposed (in parallel programs and within certain scenarios) to replace costly atomic update operations on shared data structures with simple read-write updates. If the correctness of the algorithm is not affected by this change, this leads to an algorithm variant that does not need atomic operations to update the shared data structure. This program variant still works correctly, but on the other side, it may generate more and redundant work to be done.

As an example for such a scenario, a parallel BFS algorithm was used where the atomic detection and update of univisited neighbour vertices was replaced with simple non-atomic read-/write updates. The results for this application show, that the non-atomic version has a huge performance improvement in many situations compared to a straightforward implementation with atomic accesses ($atomicBFS1$). And even compared to an already optimized version using atomic operations only if necessary, the proposed new technique has comparable or many times better performance (approx. 43,9% of all test instances). Especially larger parallel systems with more CPU sockets benefit.

The reason for the performance boost was the avoidance of atomic operations. The additional overhead, which the technique may introduce, was in the BFS application neglectible.

To apply the technique in general, it was shown what requirements must be fulfilled: the test and update operations must be idempotent.

The mainstream transactional memory hardware implementations introduced in recent processors generations (e.g., Intel Haswell) use a different approach. But similar to the approach introduced in this paper, this is an optimistic approach, too, as only the conflict case has to be handled, and not every access. It would be rather interesting to compare these two alternatives with relevant scenarios.

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. Matthias Makulla did most of the implementation work on several parallel graph algorithms including an initial version of the ones used in this paper.

References

[1] 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.

[2] *OpenMP Application Program Interface*, 4th ed., OpenMP Architecture Review Board, http://www.openmp.org/, Jul. 2013, retrieved: 08.03.2014.

[3] IEEE, *Posix.1c (IEEE Std 1003.1c-2008)*, Institute of Electrical and Electronics Engineers, Inc., 2008.

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

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

[6] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Burlington, MA: Morgan Kaufmann, 2008.

[7] M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, second edition ed. Harlow, England: Pearson Eduction, 2006.

[8] *Intel©64 and IA-32 Architectures Software Developer's Manual*. Intel Press, 2013, vol. 2: Instruction Set Reference.

[9] *AMD64 Architecture Programmers Manual*. Advanced Micro Devices, 2013, vol. 3: General-Purpose and System Instructions.

[10] *ARM v8 Instruction Set Architecture*. ARM Limited, 2013.

[11] 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*, 2010, pp. 1–11.

[12] P. E. McKenney, *Synchronization and Scalability in the Macho Multicore Era*, http://www2.rdrop.com/∼paulmck/scalability/paper/ MachoMulticore.2010.08.09a.pdf, retrieved: 27.01.2014.

[13] G. M. Baudet, "Asynchronous iterative methods for multiprocessors," *Journal of the ACM*, vol. 25, no. 2, pp. 226–244, Apr. 1978.

[14] M. Wu, "Asynchronous algorithms for shared memory machines," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[15] P. Diniz and M. Rinard, "Synchronization transformations for parallel computing," in *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1997, pp. 187–200.

[16] D. Novillo, R. Unrau, and J. Schaeffer, "Optimizing mutual exclusion synchronization in explicitly parallel programs," in *Proc. 5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, 2000, pp. 128–142.

[17] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole, "User-level implementations of read-copy update," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 2, pp. 375– 382, Feb. 2012.

[18] D. Dice, "Implementing fast Java monitors with relaxed locks," in *Proc. JavaTM Virtual Machine and Technology Symposium*, Monterey, 2001, pp. 79–90.

[19] S. Haldar and K. Vidyasankar, "Constructing 1-writer multireader multivalued atomic variables from regular variables," *Journal of the ACM*, vol. 42, no. 1, pp. 186–203, 1995.

[20] M. Herlihy and J. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proc. 20th Intl. Symposium on Computer Architecture*, 1993, pp. 289–300.

[21] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silbera, and M. Michael, "Evaluation of Blue Gene/Q hardware support for transactional memories," in *Proceedings of the 21st International Conference on Parallel architectures and compilation techniques (PACT'12)*. New York, NY: ACM, 2012, pp. 127–136.

[22] C. Jacobi, T. Siegel, and D. Greiner, "Transactional memory architecture and implementation for IBM system z," in *Proceedings of the IEEE/ACM 45th Annual Intl. Symposium on Microarchitecture*, 2012, pp. 25–36.

[23] J. Reinders, *Transactional Synchronization in Haswell*, http://software.intel.com/en-us/blogs/2012/02/07/ transactional-synchronization-in-haswell, 2012, retrieved 30.01.2014.

[24] L. Lamport, "Concurrent reading and writing," *Journal of the ACM*, vol. 20, no. 11, pp. 906–811, 1977.

[25] J. Aspnes and M. Herlihy, "Wait-free dats structures in the asynchronous PRAM model," in *Proc. 2nd Annual Symposium on Parallel Algorithms and Architectures (SPAA-90)*, Crete, Greece, Jul. 1990, pp. 240–349.

[26] M. Herlihy, "Wait-free synchronization," *ACM Trans. Programming Languages and Systems*, vol. 13, no. 1, pp. 124–149, 1991.

[27] A. LaMarca, "A performance evaluation of lock-free synchronization protocols," in *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, Los Angeles, CA, Aug. 1994, pp. 130–140.

[28] V. Lanin and D. Sasha, "Concurrent set manipulation without locking," in *Proceedings of the 7thACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Austin, TX, Mar. 1988, pp. 211–220.

[29] P. Tsigas and Y. Zhang, "A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems," in *Proceedings of the 13th Annual Symposium on Parallel Algorithms and Architectures (SPAA-01)*, Crete, Greece, Sep. 2001, pp. 134–143.

[30] G. Barnes, "Wait-free algorithms for heaps," University of Washington, Seattle, WA, Tech. Rep. TR-94-12-07, 1994.

[31] J. Valois, "Lock-free data structures," Ph.D. dissertation, Rensselaer Polytechnic Institute, Troy, NY, May 1995.

[32] ——, "Lock-free linked lists using compare-and-swap," in *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, Ottawa, Canada, 1995, pp. 214–222.

[33] H. Sundell and P. Tsigas, "Scalable and lock-free concurrent dictionaries," Chalmers University, Gteborg, Sweden, Tech. Rep. 2003-10, 2003.

[34] H. Sundell, "Efficient and practical non-blocking data stccutures," Ph.D. dissertation, Chalmers University, Gtebord, Sweden, 2004.

[35] K. Fraser and T. Harris, "Concurrent programming without locks," *IEEE Trans. Computers*, vol. 25, no. 2, 2007.

[36] G. Cong and D. A. Bader, "Designing irregular parallel algorithms with mutual exclusion and lock-free protocols," *Journal of Parallel and Distributed Computing*, no. 66, pp. 854–866, 2006.

[37] 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.

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

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

[40] 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.

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

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

[43] 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.

[44] R. Berrendorf and M. Makulla, "Level-synchronous parallel breadth-first search algorithms for multicore- and multiprocessors systems," in *submitted for publication*, 2014.

[45] 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.

[46] 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.

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

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

[49] 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.

[50] 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.

[51] G. Taubenfeld, *Synchronization Algorithms and Concurrent Programming*. Harlow, Essex: Pearson Education Limited, 2006.

[52] C. Hoare, "Monitors: An operating system structuring concept," *Comm. ACM*, vol. 17, no. 10, pp. 549–557, 1974.

[53] A. Silberschatz, J. B. Galvin, and G. Gagne, *Operating System Concepts*, 8th ed. John Wiley & Sons Inc, 2008.

[54] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Comm. ACM*, vol. 21, no. 7, pp. 558 – 565, Jul. 1978.

[55] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, pp. 66–76, Dec. 1996.

[56] M. Dubois, C. Scheurich, and F. A. Briggs, "Synchronization, coherence, and event ordering in multiprocessors," *IEEE Computer*, pp. 9–21, Feb. 1988.

[57] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Proc. 17th Intl. Symposium on Computer Architecture*. IEEE, 1990, pp. 15–26.

[58] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 3rd ed. Addison Wesley, 2005.

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

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

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

[62] 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.

[63] 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.