Trading Redundant Work Against 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. However, underlying synchronization instructions in a processor architecture are costly and rather limited in their scalability on larger multi-core/multi-processors systems. In this paper, we examine work queue operations where such costly atomic update operations are replaced with non-atomic modifiers (simple read+write). In this approach, we trade the exact amount of work with atomic operations against doing more and redundant work but without atomic operations and without violating the correctness of the algorithm. We show results for the application of this idea to the concrete scenario of parallel Breadth First Search (BFS) algorithms for undirected graphs on two large NUMA shared memory system with up to 64 cores.**

Keywords*—*atomic instructions, redundant work, parallel BFS

I. Introduction

Updating a shared data structure in a parallel program as for example an insert operation on a work queue is usually done on an application level with some sort of high-level atomic update operation (e.g., in OpenMP [1] lock-protected, atomic operation, etc.; see [2] [3] for a general discussion). The implementation of such a high-level synchronization operation itself is done by the compiler or inside a runtime system with one or even more atomic instructions (atomic-add, test-and$-\Phi$, compare-and-swap, etc.) of the underlaying processor architecture. The general problem with such atomic instructions is that they are rather costly compared to an ordninary memory access and not really scalable on larger systems [4] [5] (see also section IV for our own investigations on that). The time for *one* such atomic instruction increases significantly under contention as the number of cores in a multi-core/multi-processor system gets larger.

As the use of such synchronized updates on shared data guarantees correct operations on that data, this strict enforcement is often not really necessary. An example is a work queue, where working threads insert new items and idle threads remove 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 correctness of the algorithm, but only causing additional redundant work to be done. In such cases, the costly synchronized access can be completely removed for the cost of eventually additional work to be done.

An example for such a scenario is a Breadth First Search (BFS) for undirected graphs (see section III for details). Most of the published parallel BFS algorithms iterate over a vertex frontier where the vertices of the current vertex frontier insert new unvisited vertices to 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. Another, more general scenario is the development of asynchronous algorithms [6].

In this paper, we examine such a general strategy for a concrete parallel BFS algorithm on large shared memory multi-core multi-processor systems with up to 64 cores. We examine, what the factors are that influence the amount of additional work, what the amount of additional work is, and whether this additional work without any synchronized access to the work queue trades off against the traditional sychronized access to a work queue doing exactly the amount of work that is necessary.

The paper is organized as follows. After this introduction, we start with an overview of related work, followed by a brief overview on parallel BFS algorithms. After that, we present our new approach, describe our experimental setup, and then evaluate the new approach against the traditional way.

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 consecuting mutex lock/unlocks [7] in a program and compiler optimizations for read/write barriers [8]. Furtheron, there are advanced synchronization techniques trying to minimize synchronization costs including RCU (Read-Copy-Update) [9], special monitors [10], read-writer optimizations [11], and specialized lock-free data structures (e.g., [12]). [2] gives an overview of different aspects on related topics. [13] shows a similar benign race as ours in a parallel BFS algorithm, but without analyzing the influence of that.

An interesting general approach to handle possible concurrent accesses to shared data structures is the concept of transactional memory (original paper [14]). This approach has some similarities with our approach 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 our approach is that the bad thing happens rather seldom. Transactional memory detects the problem and (depending on the API in use) rolls back the whole transaction and restarts the operation. We instead ignore the problem (and do not even detect the problem) and have more work to do in the future.

III. Parallel Algorithms for BFS

In our application scenario for the examination, we are interested in undirected graphs $G = (V,E)$ where V is a set of vertices $v_1,...,v_n$ and E is a set of edges $e_1,...,e_m$. 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 source vertex.

The problem of Breadth First Search (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. 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 ccNUMA systems, 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: $\infty$
 4:     $current, next$, vertex container. Initially empty
 5:   end var
 6:   $d[source] \leftarrow 0$
 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:         $old = CompareAndSwap(d[w], \infty, d[v]+1)$
12:         if $old = \infty$ 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 CAS-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 version 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 is checked wether 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 visitedness in line 11. Container centric approaches are eligible for dynamic load balancing but are sensible to data locality on ccNUMA systems. Container centric approaches for BFS can be found in some parallel graph libraries [15] [16]. [17] 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 [18] [19]. This approach deploys a static one dimensional partitioning of the graph's vertices and therefore supports data locality.

IV. Alternative To Atomic Accesses

Atomic operations in a higher level parallel API for shared memory systems as mutual exclusion, atomic update, locks, compare-and-swap etc. are usually mapped on shared memory systems to atomic instructions that the underlying processor architecture provides. These atomic instructions are by itself rather costly if no contention exists. But if multiple threads concurrently access a shared state with such instructions, the cost *per operation* increases significantly. Fig. 2 shows the cost for one lock/unlock-operation (`omp_set_lock/omp_unset_lock`) in OpenMP on a shared memory system dependend on the number of processor cores utilised. In this test, p processors do in a loop n lock/unlock-operation with an empty function call between that. The test was executed on a large 64 core AMD based system. Other systems show a similar behaviour.

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 child 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 CAS operation guarantees, that every vertex is inserted exactly once into a vertex frontier (detection and mark

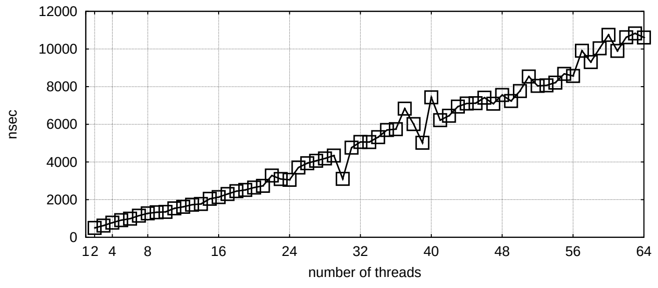


Fig. 2: Cost per lock/unlock on a large AMD-based system.

of visitedness). Without the atomic operation, a race condition exists on $d[w]$. Replacing the critical operation with a non-atomic code results in Fig. 3 (only relevant parts are shown).

```
1: for all neighbours $w$ of $v$ do
2:    if $d[w] = \infty$ 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 line 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 acces in line 3 no other thread accesses $d[w]$. In this case (and an appropriate memory model discussed above) there is *no problem* with this version, the vertex w is inserted exactly once in a vertex frontier as before.
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.

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 equal for all threads in one level iteration. Therefore, correctness is guaranteed in our scenario. 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.

Looking at the generated assembler code (and this is more or less invariant of the compiler used), 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. 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, 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. In a strict memory consistency model, it is guaranteed, that the write operation is visible to other threads immediately after this operation. But todays, all memory consistency models in practical use (e.g., [20] [1]) 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 of the used parallel API 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 done inside an inner loop iteration.

The question we are interested in 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) pays off as we might increase the work to be done substantially. The amount of additional work to be done will be influenced generally speaking mainly by:

1) problem time window (influenced by the generated code sequence and implemented consistency model) in relation to the time threads spend in non-critical code
2) the number of threads in use (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.

Although we state this here in the context of a parallel BFS algorithm, the discussion is a general discussion on the technique itself and not specific to BFS. We propose to replace costly atomic operations with probably redundant work but with cheaper simple load/store operations without modifying the correctness of the algorithm. The hypothesis is, that especially on large shared memory systems with many concurrent threads this technique pays off.

V. Experimental Setup

In this section, we describe the test setup to systematically compare the two alternatives (atomic accesses vs. redundant work) in the concrete scenario of a parallel BFS. The general algorithmic approach for parallel BFS chosen for this discussion was already given in Fig.1. We optimized this algorithm to work on chunked array based lists where each thread inserts a new vertex into a thread-private chunk. If such a chunk gets filled, the chunk is inserted into a global list. The insertion of a chunk into the global list is done in all algorithm versions with one atomic operation. But the influence of that atomic operation is neglectible.

TABLE I: CHARACTERISTICS FOR USED GRAPHS

graph name	$\|V\|$	$\|E\|$	degree avg.	degree max.	graph depth
RMAT-1M-1G	10^6	10^9	1,000	599,399	8
RMAT-1M-10M	10^6	10^7	10	4,726	16
Streets-Europe	$50,912,018$	$108,109,320$	2.123	13	17,345

In the first version named *atomicBFS*1, 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 a previous level iteration.

This last aspect can be optimized easily with a standard optimization technique in prefixing the expensive CAS-operation with a normal read operation followed by the CAS-operation only, if the test was sucessful (i.e., a test-and-test-and-set operation). This technique is also done in the OpenMP reference implementation of the Graph500 benchmark [15] for BFS. We name this version *atomicBFS*2. In this version, all vertices already visited are no longer handled with a CAS operation. We discuss the performance effect of this optimization later.

The third approach (named *nonatomicBFS*) does not use atomic operations for the unvisited-detection, but rather the code shown in Fig. 3. Therefore, a vertex may be inserted more than once in the next vertex frontier. The main difference to the other 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 a vertex as unvisited concurrently. Further algorithmic optimizations different to that discussed here and a general overview of parallel BFS algorithms can be found in another paper [17]. There is also shown, that there are better but more complex algorithms for the parallel BFS problem. But as we are only interested in this paper in the discussion of atomic operations vs. redundant work, the *relative* comparism of the introduced three versions is sufficient for that.

As we discussed already in section IV, 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. 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. We used in our tests different parallel systems. The largest one is a 64 core AMD-6272 Interlagos based system with 128 GB shared memory organised in 4 NUMA nodes, each with 16 cores (1.9 GHz). Another system is a 2-way Intel E5-2670 system with 128 GB main memory and 16-way parallelism (including 2-way Hyperthreading).

The third factor is the probability of a data collision, i.e., two vertices having a common neighbor in the graph. Only unvisited neighbours leed to an atomic operation in version *atomicBFS*1. This factor is mainly influenced in our scenario by the graph topology / degree distribution. We used several large graphs from different application areas. Besides real graphs we used also synthetically generated pseudo-random graphs that guarantee certain topological properties. Due to the limited space in this paper, we will show only results for a street graph (Streets-Europe) and two R-MAT graphs with parameters a,b,c influencing the topology, degree distribution, and clustering properties of the generated graph. See [21] for details on RMAT-graphs and [22] for a general discussion on degree distributions for R-MAT graphs. We used as RMAT parameters in the results shown $a = 0.45, b = 0.25, c = 0.15$. After graph generation, we introduced artificial edges to get a connected graph. Table I shows some important properties of the graphs used.

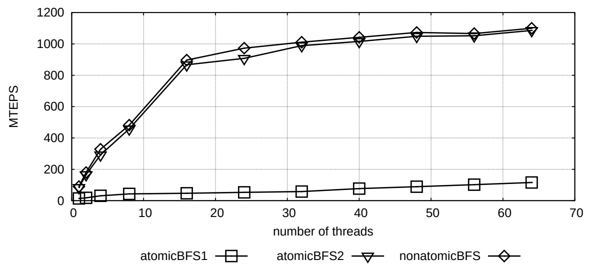
VI. RESULTS

Fig. 4 and Fig. 5 show performance results for the three versions of investigation on the two different parallel systems using different data sets. The performance is given as a rate Million Traversed Edges per Second (MTEPS), a usual measure for BFS performance (the higher, the better). The relative performance degradation Fig.4b and 4c in all versions with higher thread numbers is caused by memory bandwidth restrictions. Details on that can be found in [17].

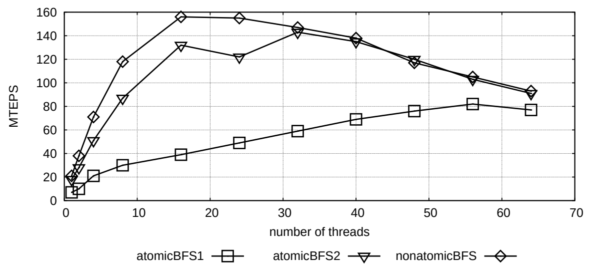
The unoptimized atomic version *atomicBFS*1 is in all tests slower than the other two versions as with *every* access to $d[w]$ in the relevant code section an atomic operation is executed. The performance difference to the other versions is very high, if many of the atomic operation were done unneccesarily, i.e., a vertex of investigation was visited already before (e.g., Fig. 4a and Fig.5a). For the two atomic versions, most times the optimized second atomic version *atomicBFS*2 is much better due to the prefixed test done with a normal read operation.

But the best version out of the three is the version *nonatomicBFS* using our proposed technique without any atomic operation in the code section of investigation. The difference to the better atomic version *atomicBFS*2 is rather small if there is a lot of vertex sharing (e.g., vertices have high degrees). In that case, vertices may get visited very often and only the first visit leads to a CAS operation in version *atomicBFS*2 (see again Fig. 4a and Fig.5a). On the other side, the difference between the non-atomic version *nonatomicBFS* and *atomicBFS*2 is quite high, if update operations are done more frequently on vertex visits, as for example in sparse graphs with small vertex degrees (Fig.4b, 4c, 5b, 5c).

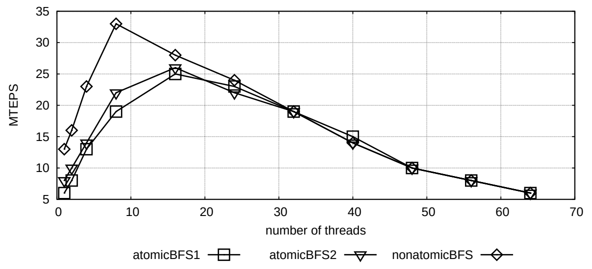
To further examine these results, we determined frontier sizes during each level iteration. The *edge frontier size* gives the number of outgoing edges from vertices in the current frontier, i.e., the number of vertex candidates that have to be checked for inclusion into the next frontier. On the other 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 with a CAS, in version *nonatomicBFS* with a simple write). Fig. 6



(a) Performance for RMAT-1M-1000M

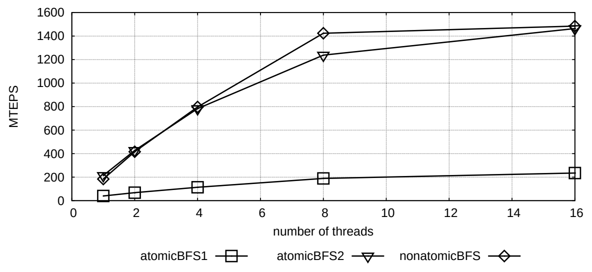


(b) Performance for RMAT-1M-10M

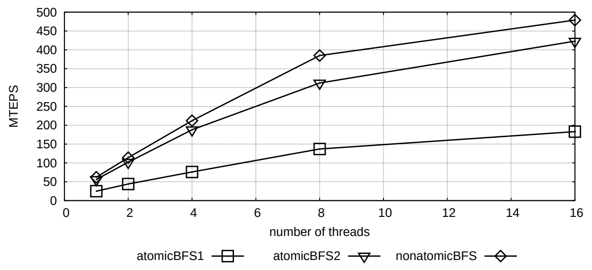


(c) Performance for Streets-Europe

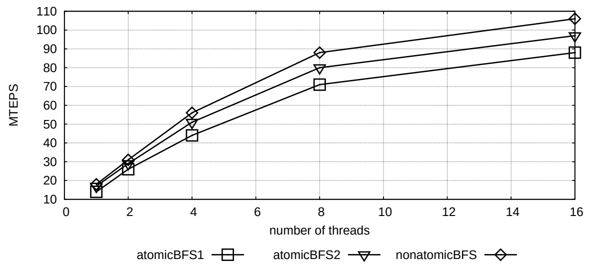
Fig. 4: Performance data on AMD-based system with 64x parallelism.



(a) Performance for RMAT-1M-1000M



(b) Performance for RMAT-1M-10M



(c) Performance for Streets-Europe

Fig. 5: Performance data on Intel-based system with 16x parallelism.

TABLE II: PERCENTAGE OF VERTICES THAT GET INSERTED MULTIPLE TIMES.

number of threads	min. percentage	median	max. percentage
2	0.000012	0.000030	0.000049
4	0.000002	0.000014	0.000026
8	0.000004	0.000013	0.000027
16	0.000002	0.000018	0.000039
24	0.000006	0.000020	0.000037
32	0.000010	0.000022	0.000035
40	0.000010	0.000022	0.000037
48	0.000010	0.000026	0.000035
56	0.000012	0.000027	0.000055
64	0.000016	0.000029	0.000051

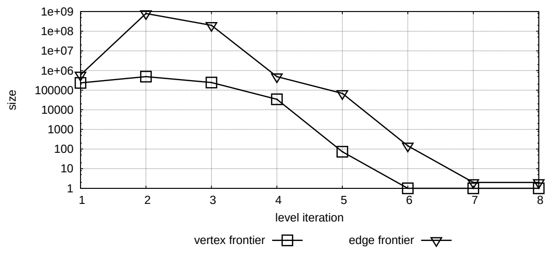
shows frontier sizes during each level iteration. Setting this information in relation to the performance numbers, a large difference between edge frontier size and vertex frontier size in a level iteration means that many atomic checks were made in version *atomicBFS*1 that didn't lead to an unvisited neighbor vertex / insert operation. On the other side, if the difference between vertex and edge frontier size is small, the difference between the three versions is less, as the amount of critical operations is rather small compared to all operations executed.

Furtheron, we measured how many vertices get inserted multiple times in version *nonatomicBFS*, i.e., the additional and redundant work that is generated. The factors influencing that were discussed already. We show results only for the largest system and for the street-graph, this is the most problematic test instance where the probability for double insertion is highest. In Tab. II we show the overhead in percentage of vertices inserted more than once, i.e., leading to redundant and more work. As can be seen, the probability increases slightly with more threads, but still this overhead is for our scenario negligible (also in all other tests with different data sets not shown here). Even with 64-fold concurrency, there are very rare situations that lead to multiple insertions. The maximum overhead value is 0.000055 percent or absolutely seen, instead of 50,912,018 vertices to be inserted, with *nonatomicBFS* 50,912,046 vertices were inserted, the difference is 28.

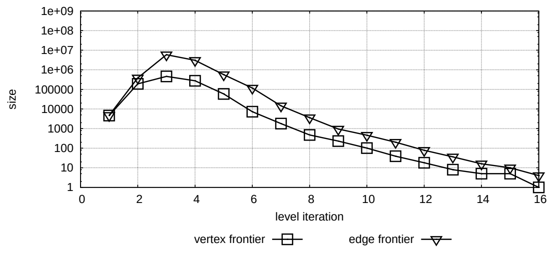
VII. CONCLUSIONS

We propose 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 any atomic operations. This algorithm variant still works correctly, but on the other side, it may generate more and redundant work to be done.

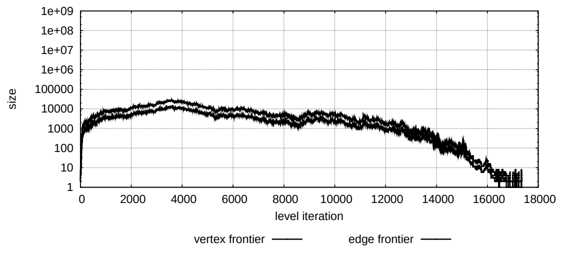
As an example for such a scenario, we used a parallel BFS algorithm where the atomic detection and update of



(a) Frontiers for RMAT-1M-1000M



(b) Frontiers for RMAT-1M-10M



(c) Frontiers for Streets-Europe

Fig. 6: Vertex and edge frontier sizes.

univisited neighbour vertices was replaced with simple non-atomic read/write updates. The results show, that for this scenario the non-atomic version has a huge performance improvement in many situations compared to a straightforward implementation with atomic accesses (*atomicBFS*1). And our version has most times a performance improvement of up to 50% compared to an optimized atomic version (*atomicBFS*2) that uses atomic accesses only if necessary. The higher the frequency of atomic operations, the greater the advantage is. Our proposed technique delivers in *all tests* equal or better performance results within the error of measurement than any of the versions with atomic operations.

The upcoming mainstream transactional memory hardware implementations (e.g., Intel Haswell) use a different approach. But similar to our approach, 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] “OpenMP application program interface,” OpenMP Architecture Review Board, http://www.openmp.org/, 2011, retrieved: 6,2013.

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

[3] M. Ben-Ari, Principles of Concurrent and Distributed Programming. Harlow: Addison-Wesley, 2006.

[4] 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.

[5] P. E. McKenney, “Synchronization and scalability in the macho multicore era,” http://www2.rdrop.com/∼paulmck/scalability/paper/MachoMulticore.2010.08.09a.pdf, 2010, retrieved: 6,2013.

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

[7] 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.

[8] D. Novillo, R. C. 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.

[9] 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, 2012, pp. 375– 382.

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

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

[12] K. Fraser and T. Harris, “Concurrent programming without locks,” IEEE Transactions on Computers, vol. 25, no. 2, 2007, pp. 1 – 44.

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

[14] 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.

[15] Graph 500 Comitee, “Graph 500 benchmark suite,” http://www.graph500.org/, retrieved: 6, 2013.

[16] 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.

[17] R. Berrendorf and M. Makulla, “Parallel breadth first search algorithms for multicore- and multiprocessor systems,” in submitted for publication, 2013.

[18] 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.

[19] 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. 1–10.

[20] ISO/IEC 14882:2011 Programming Languages – C++, ISO, Geneva, Switzerland, 2011.

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

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