International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:6, No:1, 2012

Concurrency without Locking in Parallel Hash Structures used for Data Processing

Ákos Dudás and Sándor Juhász

***Abstract*—**Various mechanisms providing mutual exclusion and thread synchronization can be used to support parallel processing within a single computer. Instead of using locks, semaphores, barriers or other traditional approaches in this paper we focus on alternative ways for making better use of modern multithreaded architectures and preparing hash tables for concurrent accesses. Hash structures will be used to demonstrate and compare two entirely different approaches (rule based cooperation and hardware synchronization support) to an efficient parallel implementation using traditional locks. Comparison includes implementation details, performance ranking and scalability issues. We aim at understanding the effects the parallelization schemes have on the execution environment with special focus on the memory system and memory access characteristics.

***Keywords*—**Lock-free synchronization, mutual exclusion, parallel hash tables, parallel performance

I. Introduction

PARALLELIZATION is becoming the de-facto solution for performance enhancement of all types of algorithms. Large scale data processing is a perfect candidate for parallelization, as the most time consuming steps execute simple transformations on large amounts of data (e.g. [1] claims preprocessing to be responsible for 80% of the execution time of data mining process), thus little space is left for algorithmic optimizations.

Speeding up an algorithm requires the analysis and the understanding of the factors that determine its performance. Algorithmic step count, data movement costs, memory layout and usage pattern of data structures all play an important role in the runtime behavior of an application. Our main objective is to offer different methods to make better use of widespread state-of-the-art computer architectures that contain multiple processors (and/or processor cores) by identifying the main performance factors and examining their effects when various parallel approaches are applied to solve the same data transformation problem.

The motivation of this work comes from a large scale web log processing project [2] which required the recoding of multiple large data fields with limited domain to a more compact representation format. The essential of the task was to build a large lookup table containing up to several tens of millions of key-value pairs. The code table was built on the fly, that is, the data structure empty in the beginning was continuously expanded when any unknown field content (key) that was encountered on the input. Any existing (long) input content was to be replaced by a (short) output value that was assigned to the key at the first encounter. The result of the process was a new data file more fitting for further processing as it carried the same information content as the original web log, but only had a fraction of its size.

For increased efficiency the lookup table was implemented by a custom hash table that only supported insertion of new elements and looking up the values belonging to specific keys (no entry modification or table reorganizations were required). For this reason in the further part we focus our effort on speeding up these two operations, although it is worth mentioning that hints for efficient parallel implementations of the missing operations can be found in [7, 8, 9].

The fact that the hash table is used in data processing is relevant for two reasons. First of all, the vast amount of data that goes through the hash table forms our goal to use as few indirections in the storage structure as possible; this, however, clashes with the concept of one of the synchronization solutions we present. Secondly, the size of the table, and namely, the number of buckets in the table makes the use of locks unfavorable due to their overhead of consuming memory; memory which could be used in the caches for data storage instead.

We present and analyze different methods for allowing and speeding up parallel access to hash tables. This practical example is used for demonstrating the main ideas and outlining the performance effects of the different approaches. Both the aforementioned issues, namely indirections in the storage structure, and memory overhead of the parallelization solutions will be studied through the performance evaluation of the methods.

The rest of the paper is organized as follows. In Section II we identify the performance factors of parallel algorithms running on current desktop architectures and present some recent works using locks in concurrent hash tables. Section III presents our arguments against the use of locks in general. Section IV provides the implementation details of the lock-free, non-blocking hash table variants that allow several threads to cooperate without using traditional synchronization mechanisms such as locks, semaphores or barriers. In Section V measurement results are presented and analyzed to evaluate the different implementation approaches. Section VI summarizes our findings and gives a brief overview of our findings.

The authors are with the Department of Automation and Applied Informatics, Budapest University of Technology and Economics, 1117 Budapest, Magyar tudósok krt. 2. QB-207, Hungary. E-mail: {akos.dudas, juhasz.sandor}@aut.bme.hu.

II. Related Works

A. Performance Factors of Sequential Implementations

The most important factor that determines the performance of an algorithmic its mathematical complexity. This particular aspect is not in our scope now, because hash tables have O (1) complexity for accessing an item. The access time being a constant means that it is independent of the size or of number of elements in the hash table, and with the right choice of parameters (size of the hash table, nearly ideal hash function) the average access path can be kept nearly as low as a single step [3].

In data- and memory intensive applications the second most important factor is the memory access characteristics of the algorithm. Lookup tables require low computation power; their performance is rather dominated by the memory access times than the number of instructions completed to find the element corresponding to the key. Current architectures bridge the performance gap between current CPUs and their main memory by a multilevel cache, thus in practice the performance of data intensive application is always measured in the number of the slow memory accesses (cache misses). [4] was the first to mention the importance of caches when adjusting the parameters of hash tables, while [5] provide in detailed study about the cache performance when using various hash functions along with different collision avoidance methods. [10] proposed a method to combine cache awareness and reliability by grouping buckets into cache lines. In our previous work [6] we provided an extensive study explaining how the structure (memory layout) and parameters of the hash table should be chosen for optimal lookup performance. These results will be referenced and used during the presentation of the implementation details.

B. Parallel Execution and Mutual Exclusion

In parallel environments (such as multi-CPU and multi-core systems) further speedup can be achieved by taking advantage of the execution environment's capabilities. Preparing a sequential algorithm for parallel executing has many approaches. In order to ensure that threads executing in parallel provide the same result as the sequential approach, at some critical points it must be made sure that only one thread has access to certain parts of the data structures to maintain their integrity. This kind of mutual exclusion is usually enforced by the use of locks.

Several parallel implementations of hash tables are available that use look based synchronization. A single hash table level look can easily become a bottleneck, thus several method were developed to overcome this difficulty. Larson et al. in [11] use two lock levels, there is one global table level lock, and there is one separate lightweight lock (a flag) for each bucket. The high level lock is just used for setting the bucket level flags and released right afterwards. This ensures a fine grained mutual exclusion (concurrent operations on bucket level), but needs only one real lock for the implementation. It was shown by [12] that in case of non-extensible hash tables simple reader-writer locks can provide a good performance in shared memory multiprocessor systems. More efficient implementation like [8] use a more sophisticated locking scheme with a smaller number of higher level locks (allocated for hash table sections including multiple buckets) allowing concurrent searching and resizing of the hash table.

III. A Case For Avoiding Locks

Locks can be implemented purely by software in theory, however all modern architecture provide hardware support in form of atomic bitwise *test-and-set* and word-size *compare-and-swap* operations guaranteeing that no interruption, or any other bus operation initiated by other processors or bus controllers will occur between the read and write part of operation. Hardware locking can be used directly, or through a wrapping layer provided by the execution environment or the operating system (providing extra services like waiting queues or thread state control yielding for other threads until the critical section becomes available).

The first problem with locks is that since they are the means of communication between threads and the processors the threads execute on, they must always be up-to-date, meaning that they cannot be cached. The same memory location is periodically updated by various threads forcing all processors to purge the particular cache line from their caches. The next time the lock is tested, the data is read from the system memory directly resulting in a cache miss. This is an important consideration when locks are used; they are expensive to check and modify and the cost is a cache miss at all times.

The second problem is the actual level of parallelism under the surface. The use of locks does not provide parallelism; it does exactly the opposite. Threads are forced to wait if another one is still working the critical section. The threads can wait actively (i.e. continuously polling the state of the locks by spin-waiting) or passively giving up their time slice until the lock becomes available. Either solution has additional costs, such as wasting computing power by active waiting, or involving the operating system scheduler in the other case.

The amount of time lost at waiting can be reduced by creating multiple finer-grained critical regions. In case of hash tables this means that instead of locking the whole table we apply the lock on smaller regions of an open hash table, or on bucket or bucket groups of a bucket hash table. As number of locks increases collisions become less and less probable, providing better performance and scalability. Unfortunately using high number of locks is often not supported by the run-time environment (we may just have a few thousands), and they have a relatively large memory footprint. Even if we use only one bit, we cannot create an array of locks because of the effect of "false sharing." False sharing hints on the shared use of the same cache line by multiple independent locks, where the modification of one lock will not only effect that single bit, but it will purge all other unrelated neighboring bits (for a 64 byte long cache line 511 other locks) form the cache of the concurrent processors. We can solve this problem by assigning a complete cache line to each lock, or merging the lock with data items (adding one extra byte for each lock) thus spreading locks all over the memory resulting in higher memory allocation.

In general all lock-based algorithms suffer from the drawbacks of blocking synchronization such as deadlocks, long and undefined delays and priority inversions which is especially true when using extensible hash tables [9].

Using locks is not really convenient: threads can hinder the execution of each other, extra memory is required, data structure should be reorganized (to place the lock) and in cases of unfortunate implementations the interaction of multiple locks can stall the whole processing (deadlock situations). A great amount of research efforts has been made in the literature to develop non-blocking synchronization methods. In the following section we present two ideas that give a kind of workaround of the lock-based mutual exclusion particularly applicable to hash tables.

IV. Implementing A Concurrent Hash Tables

This section presents the concurrent implementations of the hash table. We start from a single threaded, optimized variant, which then will be altered according to three different parallelization schemes: traditional mutual exclusion with locking, lock-free with hardware atomic operations, and lock-free rule-based cooperation.

A. Optimized Hash Table

Parallel performance optimization begins with an optimized sequential hash table. As discussed in detail in Section II the performance is significantly affected by the memory characteristics. Multiple works as well as our experiences [5, 6, 10, 11] show that the number cache misses in the search path should be a primary concern. We have found [6] that the best choice in this case is a hash structure that uses no indirections for storing the items if there is no collision (see Fig.1). This is a bucket hash table using chaining for resolving collisions. The most important benefit of this structure is that the first item in each bucket can directly be accessed. (This property will be relevant when discussing the CAS solution in Section IV.C.)

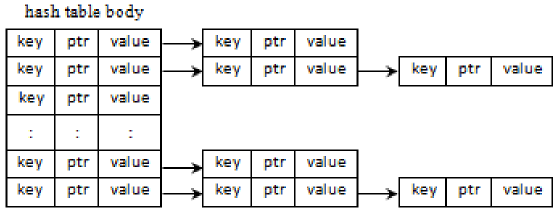

Fig. 1 Memory layout of hash table we use

Each item consists of the key, a pointer to the next item, and the stored value associated with the particular element. The key and the pointer are placed next to each other to increase the chance of being in the same cache line beneficial when following the bucket chain. The hash table body is compact, no cache line alignment or padding is used to reduce memory footprint.

Another import factor for good performance is short search path. This can be guaranteed by a uniform hash function and the right choice of the table size. Although this hash table is a kind of bucket hash implementation the average bucket size should be kept as low as 1, which is achieved by choosing the size of the hash table about 20-40% bigger, than the number of items it should hold. In this working point the compact structure functions rather like an open key hash table with special list-based collision avoidance mechanism.

B. Traditional Locking

Little effort is required for a simple locking implementation. A single table-level lock sufficiently solves the concurrency problem, but it basically serializes all accesses to the table easily resulting in performance loss instead of gain. On the other hand using too many locks (i.e. one for each bucket) it is a waste of memory space: if we place a lock to each bucket, than threads only collide when trying to access the same bucket at once (practically never), but we have to maintain tens of millions of locks in the memory.

Based on previous research [8, 11, 12] and verified by our experiments region locks are the best choice. We chose to set up 1024 locking regions (Fig.3.a) from consecutive hash table items.

C. CAS Based Implementation

Lock free parallel implementations usually use atomic compare-and-swap (CAS) operations instead of explicit locking. This method is founded on the fact that all modern computer architectures provide hardware support for atomic combined read-write operations (e.g. the CMPXCHG instruction in thex86architecture) that allow the non-destructive manipulation of a single machine word.The basic idea is that we should construct algorithms where it is sufficient to manipulate a single machine word to achieve the necessary result. In practice the manipulated value is a pointer (which has the length of one machine word), thus in the first step we create the composite data structure in the memory according to the current situation, than we try to move it in place with the conditional CAS operation. If the compare condition does not hold true anymore (the memory location has been modified by another thread) than the swap operations fails. We adapt the structure to the new condition and we try the insertion again.

With this method complete lock-free data structures and algorithms can be build, as it was done with linked list by Michael in [12], whose works was further extended to resizable hash tables by in [9].When creating a CAS based implementation (see Fig. 3.b) the data structure manipulation consist of a series of pointer adjustments. For this reason we have to modify the data structure we use (see Fig. 2). This works exactly the same way as the previous variant except for the data items not being embedded into the body of the hash table.This extra indirection in the structure clashes with our goal to store the items accessible without indirection from the table body, and is expected to be accountable for an increased number of cache misses.

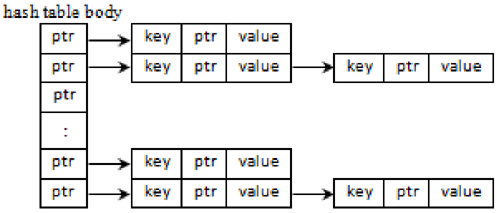

Fig. 2 Memory layout of hash table adopted to be used with CAS

The main advantage of this approach is high scalability using hardware supported optimistic concurrency handling, while its price is the development effort put into finding and implementing an algorithm which produces the right results with applying the pointer manipulations one by one in the right order. When using this approach we expect less waiting due to the omission of locks, but at a cost of increased cache misses that originate from the memory layout modification required.

D. Rule-Based Cooperation

The last method for implementing a lock-free hash table is what we call *rule-based cooperation*. The main idea is to do a kind a reverse work allocation compared to the traditional lock based scenario where each thread is capable of executing any task and while doing so they protect their working area with lock to avoid the interference with other tasks. In the reverse case we specialize our threads by assigning individual work areas to them that do not overlap (separate functions, pipeline stages, spatial domains or graph branches). In this case a specific service requests can be served by just one thread, and it is up to the right selection of domains to provide the load balancing.

The selection rules are usually based on data decomposition as functional decomposition generally does not provide free scalability and we easily reface the problem of critical sections on the level of organizing the control. According to our knowledge this kind of cooperation is never mentioned in the literature of shared memory algorithms, but the idea is not unknown in distributed systems, where there low cost shared memory synchronization is not available, thus the coarser grained cooperation between the nodes is maintained by directed point-to-point messages or implicit work allocation rules (e.g. distributed file servers, horizontally partitioned data bases, documents groups allocated to separate web servers).

In practice with *n* threads the original hash table is divided into *n* regions (sub-hash tables, see Fig. 3.c) where each thread is responsible for exactly one region. As the thread is the owner of its region no locking is required, as nobody else is allowed to reach the data inside.

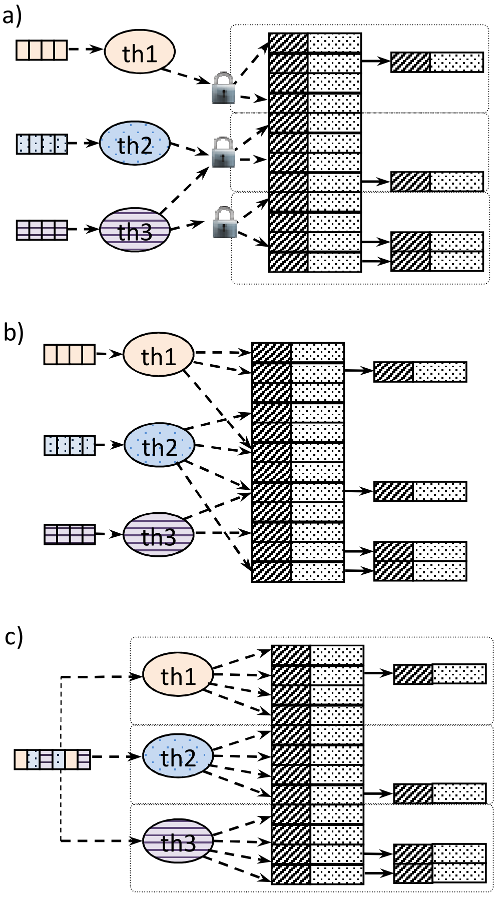

Fig. 3 Concurrent hash table implementation: a) hash table with section locks, b) CAS based cooperation, and c) rule-based task separation

The threads share a common input and output region which can be addressed directly (with the index of the input or the output element). Each threads reads each input element (no locking is needed, this is just a read only access for everyone) and based on a rule they decide whether to process that specific element or to move to the next one (Figure 3.c). The rule is constructed in such a way that it chooses exactly one thread for processing the element, and that thread is responsible for creating the output (no locking required as writing as each element is written by one single thread).

In the case of lookup tables the rule should be constructed in a way that ensures the elements with the same key always go to the same thread. In our implementation we used a simple modulo n rule, which was applied the hash function of the key (the same hash function is used as the one used inside the sub-hash tables for placing the elements). That is, if the threads are numbered between 0 and *n-1* then thread *m* (0 <*m*<*n*) selects itself for processing if *hash (key) mod n* equals to *m*. We should use complex hash functions in both the external decision and the internal placement as we seek for providing a highly uniform load distribution between the threads and between the table slots as well. The advantage of sharing the same hash function is that it needs to be calculated only once when handling an item. The complex external decision function has its disadvantage as well as all threads need to calculate it for each item just to find out whether is belongs to their scope or not. We expect to see higher instruction count on behalf of this approach.

In the case of rule based workload distribution there is no need for locking neither at software nor at hardware level, no special instruction are required, no additional cache misses appear and implementation is relatively simple. These benefits come at the cost of increased computation as all thread has to apply the rule to all items which is just a kind of synchronization overhead for *n-1* out of the *n* threads.

V. Measurements And Results

In this section we compare the performance of above mentioned three implementation types (high-level section lock, CAS, rule-based co-operation) considering different number of threads and various workload types.

The measurements were executed on an Intel Core i7-2600 CPU (3.6 GHz, 4 core and Hyper Threading) with 8 GB system memory and Windows 7 operating system. The hash tables and every locking mechanism were implemented in C++ with careful manual optimization and compiled by Microsoft Visual Studio 2010 in default release build mode.

Each test scenario reports the average of 5 executions. We measured the execution time, the number of last level cache misses (8 MB L3 cache shared by all cores) and the number of executed instructions.

The first type of workload (see Figure 4) has 50% insert and 50% lookup operations, while the second (see Figure 5) consists of 10% insert and 90% lookup operations. (Please note that *operations* in this context are searches/inserts in the table, while the *instructions* are executed by the CPU).

What we are interested in is on one hand the performance which is measured in terms of operations (lookup/insert) per millisecond (the higher the better) and the reason behind the performance differences. We measure the number of operations per 1000 CPU instructions and number of operations for each dozen cache misses. The scales are of practical choice for the visibility. Both are better when higher (i.e. more lookups completed by the same number of CPU instructions).

The lock-based solution and the CAS method have their peak performance at 8 threads achieving 2.9 and 3.5-4.25 speedup over the baseline while the cooperation method is at its the best with 7 threads with a speedup of 2.3-2.5.

The lock-based solution dominates the lock-free solutions up to about 8 concurrent threads, which is exactly the number of (virtual) cores in the CPU.

The CAS solution has mostly the same performance. We also see that it increased cache miss count (the operations per 10 cache misses is lower) which is due to the extra indirection in the data structure. It also has the best instruction count (locks do spin-waiting consuming instructions, cooperation calculates more hash functions). The biggest gain is the scalability.

The cooperation solution behaves unevenly with different number of threads, which is due to the implicit load balancing of the chosen hash function which performs the data decomposition. It has the worst utilization of CPU instructions (lowest count of lookups/inserts completed by the same amount of CPU instructions). Since the hash function is calculated by all threads, it goes to waste for all but one of them. It also seems that this solution is not really cache friendly. Since the threads are not controlled or synchronized it seems that they have uneven loads and work in different regions of the system memory, which puts more strain on the system.

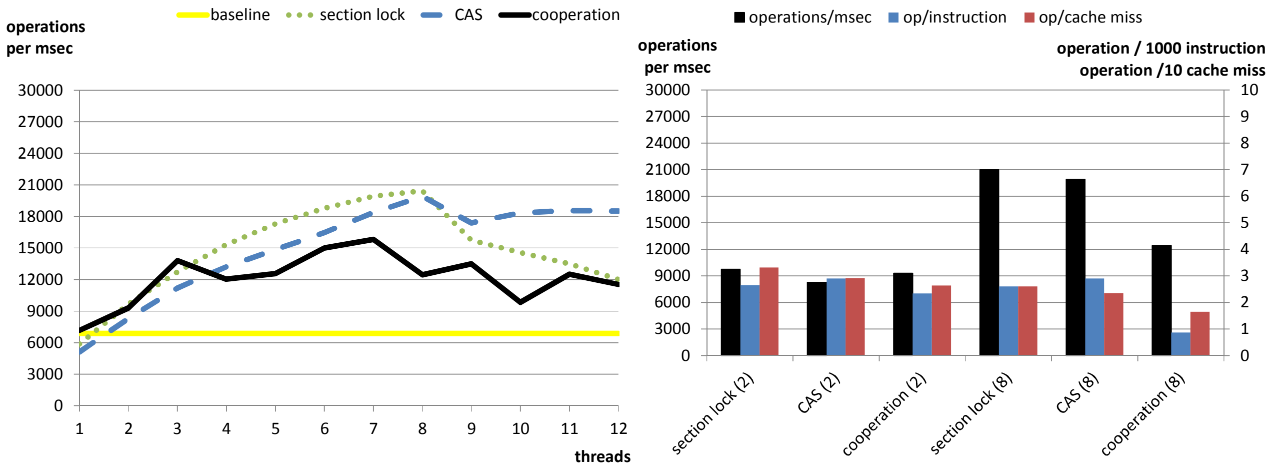

Fig. 4 Operations per millisecond for various number of threads using the four different schemes (left); and the same for 2 and 8 threads with the corresponding number of operations per instructions and cache misses (right). The workload consist of 50% insert and 50% lookup operations

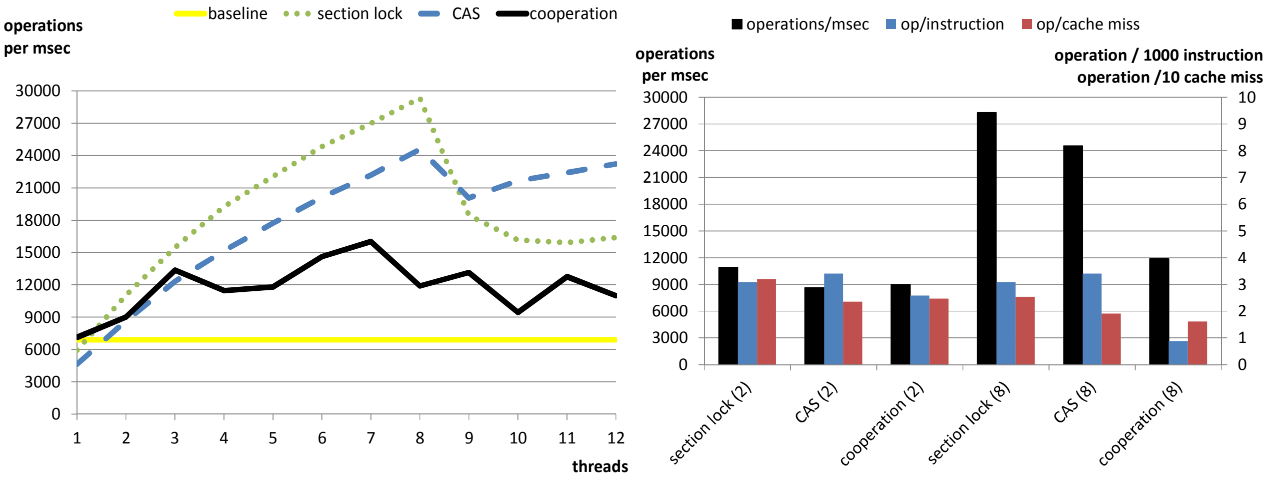

Fig. 5 Operations per millisecond for various number of threads using the four different schemes (left); and the same for 2 and 8 threads with the corresponding number of operations per instructions and cache misses (right). The workload consist of 10% insert and 90% lookup operations

VI. Conclusion

This paper presented two lock-free implementations of hash tables. Our hypothesis is that locking solutions are not practical, since they have limited applicability (constraint on the number of locks), require invasive modification of the algorithms (placement of the locks, memory concerns), threads can hinder each other's performance, and lastly, it is prone to faulty or crashed threads.

It can be said that really good performance can be achieved with locks, but it must also be taken into consideration that the locks we used are implemented as assembly level bit-test-and-set based solutions allowing for very low overhead and unlimited number of locks. The use of locks also potentially hinders executing, while lock-free solutions are incapable of causing deadlocks.

The lock-free solutions this paper examined included a well-known technique of using the CAS primitive, which provides a good alternative to locking, but required changes in the storage structure, which causes more cache misses and results in more complex algorithm where extra manual labor was put into the implementation and verification. The biggest advantage of this approach is its scalability.

The rule-based cooperation with data decomposition is an idea borrowed from distributed systems and applied to shared memory parallel algorithms. The performance of this approach is limited by the increased amount of instructions all threads need to perform redundantly, but has a nice feature of being applicable without any modification to the algorithms and data structures.

Acknowledgment

This project is supported by the New Hungary Development Plan (Project ID: TÁMOP-4.2.1/B-09/1/KMR-2010-0002).

References

[1] S. Ansari, R. Kohavi, L. Mason, and Z. Zheng, "Integrating E-Commerce and Data Mining: Architecture and Challenges," in IEEE International Conference on Data Mining, 2001, pp. 27-34.

[2] S. Juhász, and R. Iváncsy,"Tracking Activity of Real Individuals in Web Logs,"*International Journal of Computer Science*, Vol. 2, No. 3, pp. 172-177, 2007.

[3] W. Litwin,"Linear hashing: A new tool for file and table addressing," In Proceedings of the Sixth International Conference on Very Large Data Bases, New York, pp. 212-223, 1980.

[4] M. Mitzenmacher,"Good Hash Tables & Multiple Hash Functions," Dr. Dobbs Journal, No. 336, pp. 28-32, May 2002.

[5] G. L. Heileman and W. Luo,"How caching affects hashing," In Proceedings of the 7th Workshop on Algorithm Engineering and Experiments, Vancouver, Canada, pp. 141-154, 2005.

[6] S. Juhász and Á. Dudás, "Optimising Large Hash Tables for Lookup Performance," In proceedings of the IADIS International Conference Informatics 2008, Amsterdam, The Netherlands, pp. 107-114, 2008.

[7] M. Greenwald, "Two-handed emulation: How to build non-blocking implementations of complex data-structures using DCAS," In Proceedings of the 21st ACM Symposium on Principles of Distributed Computing. ACM, New York, pp. 260–269, 2002.

[8] D. Lea, "Hash table util.concurrent.ConcurrentHashMap, revision 1.3, in JSR-166, the proposed Java Concurrency Package", 2003, http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent/

[9] O. Shalev and N. Shavit, "Split-ordered lists: Lock-free extensible hash tables,"*Journal of the ACM* 53(3): pp. 379-405, 2006.

[10] A. Sachedina, M. A. Huras and K. K. Romanufa,"Resizable cache sensitive hash table," US Patent number: 7085911, 2003.

[11] P.-A. Larson, M. R. Krishnan,and G. V. Reilly, "Scaleable hash table for shared-memory multiprocessor system," US Patent number: 6578131, 2003.

[12] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," In Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, New York, pp. 73–82, 2002.