information

MDPI

Article

Algorithms for Optimization of Processor and Memory Affinity for Remote Core Locking Synchronization in Multithreaded Applications

Alexey Paznikov and Yulia Shichkina *

Department of Computer Science and Engineering, Saint Petersburg Electrotechnical University "LETI", Saint Petersburg 197022, Russia; apaznikov@gmail.com

* Correspondence: strange.y@mail.ru; Tel.: +8-981-963-6645

Received: 11 December 2017; Accepted: 16 January 2018; Published: 18 January 2018

Abstract: This paper proposes algorithms for optimization of the Remote Core Locking (RCL) synchronization method in multithreaded programs. We propose an algorithm for the initialization of RCL-locks and an algorithm for thread affinity optimization. The algorithms consider the structures of hierarchical computer systems and non-uniform memory access (NUMA) in order to minimize the execution time of multithreaded programs with RCL. The experimental results on multi-core computer systems show the reduction of execution time for programs with RCL.

Keywords: remote core locking; RCL; synchronization; critical sections; scalability

1. Introduction

Currently available computer systems (CS) [1] are of large scale and include multiple architectures. These systems are composed of shared memory multi-core compute nodes (SMP, NUMA systems) equipped with universal processors, as well as specialized accelerators (graphics processors, massively parallel multi-core processors). The number of processor cores exceeds 10^2–10^3. Effective execution of parallel programs on these systems is a significant challenge. System software must consider the large scale, multiple architectures and hierarchical structure.

Parallel programs for multi-core CS with shared memory are multithreaded, in most cases. Software tools must ensure linear speedup with a large amount of parallel threads. Thread synchronization while accessing shared data structures is one of the most significant problems in multithreaded programming. The existing approaches for thread synchronization include locks, lock-free algorithms and concurrent data structures [2], and software transactional memory [3].

The main drawback of lock-free algorithms and concurrent data structures, despite of their good scalability, is the limited application scope and severe complexity related to the parallel programs development [2,4,5]. Furthermore, the development of lock-free algorithms and data structures includes the problems, connected with the memory release (ABA problem) [6,7], poor performance and restricted nature of atomic operations. Moreover, the throughput of lock-free concurrent data structures is often similar to their corresponding lock-based data structures.

Software transactional memory nowadays has a variety of issues, connected with large overheads for execution of transactions and multiple aborts of transactions. Moreover, the existing transactional memory implementations restrict the operations set inside the transactional section. Consequently, transactional memory does not ensure sufficient performance of multithreaded programs, and is, for now, not commonly applied in real applications.

The conventional approach of using lock-based critical sections for synchronization in multithreaded programs is still the most widespread in software development. Locks are simple to use (compared with lock-free algorithms and data structures) and, in most cases, ensure acceptable performance. Furthermore,

most existing multithreaded programs utilize a lock-based approach. Thereby, development of scalable algorithms and software tools for lock-based synchronization is urgent, today.

Lock scalability depends on access contention of threads accessing shared memory areas and the locality of references. Access contention arises when multiple threads simultaneously access a critical section, protected by one synchronization primitive. In terms of hardware, this leads to a huge load being applied to the data bus, as well as cache memory inefficiency. The cache memory locality is meaningful when a thread inside a critical section accesses the shared data previously used on another processor core. This case leads to cache misses, and a significant increase in the time required for critical section execution.

The main approaches for scalable lock implementations are CAS spinlocks [8], MCS-locks [9], Flat combining [10], CC-Synch [11], DSM-Synch [11], Oyama lock [12].

For the analysis of existing approaches, we have to consider the critical section's execution time. The time t for a critical section execution comprises the time t_1 for the execution of a critical section's instructions and the time t_2 for the transfer of lock ownership. In the existing lock algorithms, the time required for the transfer of lock ownership is determined by global flag access (CAS spinlocks), context switches and awakening of the thread executing the critical section (PThread mutex, MCS-locks, etc.), or global lock capture (Flat Combining). The time for execution of the critical section's instructions depends substantially on the time t_3 of global variables access. Most existing locking algorithms do not localize access to shared memory areas.

Existing research includes methods for localization access to cache memory [10,13–15]. The works [14,15] are devoted to the development of concurrent data structures (linked lists and hash tables) on the basis of critical section execution on dedicated processor cores. The paper [13] proposes a universal hardware solution, which includes the set of processor instructions for transferring the ownership to a dedicated processor core. Flat Combining [10] refers to software approaches. Flat Combining realizes execution of critical sections by server threads (all threads become server by turns). However, the transfer of lock server ownership between threads being executed on different processor cores leads to a performance decrease, even at insignificant access contention. In addition to this, all these algorithms do not support thread locking inside critical sections, including active waiting and operation system core locking.

Ownership transfer involves the overheads due to context switches, global variable cache loading from RAM, and activation of the thread executing the critical section. Critical section execution time depends heavily on the reference localization of global variables. Common mutual exclusion algorithms assume frequent context switches, which leads to the exclusion of shared variables from cache memory.

This paper considers the Remote Core Locking (RCL) method [16,17], which assumes the execution of critical sections by dedicated processor cores. RCL minimizes the execution time of existing programs, thanks to critical path reduction. This technique assumes the replacement of high-load critical sections in existing multithreading applications with remote functions, which call for its execution on dedicated processor cores. In this case, all of the critical sections protected by one lock are executed by a server thread running on the dedicated processor core. In this way, all critical sections are executed on that particular thread, and the overheads for the transfer of lock ownership are negligible. RCL also reduces critical sections' instruction execution time. The minimization is achieved by localization of data accessed within a critical section in the cache memory of RCL-server's core. Localization minimizes cache misses. Each working thread (client-thread) uses dedicated cache-line, which is used for critical section data storage and active waiting. This schema reduces access contention.

The current implementation of RCL has several drawbacks. Firstly, there is no memory affinity in NUMA systems. Computer systems with non-uniform memory access (NUMA) (Figure 1) are currently widespread. These systems are compositions of multi-core processors, each of which relates directly to the local segment of the global memory. A processor (subset of processors) with their local

memory constitutes a NUMA-node. Interconnection between processors and local memory addresses is performed through busses (AMD HyperTransport, Intel Quick Path Interconnect). The address to local memory is performed directly, the address to remote NUMA-nodes is more costly, because it requires the use of an interprocessor bus. In multithreaded programs, the latency of RCL-server addresses to the shared memory areas is essential for the execution time of a program. Memory allocation on NUMA-nodes, which are not local the RCL-server, leads to severe overheads when RCL-server accesses the variables allocated on the remote NUMA-nodes.

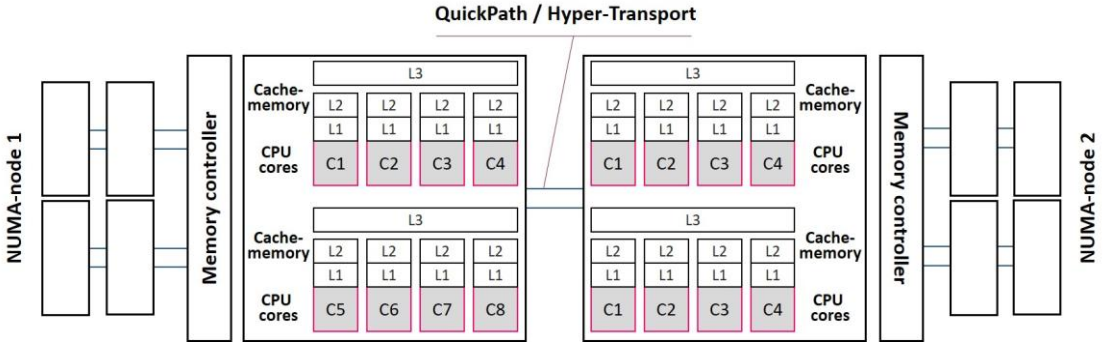

Figure 1. Structure of NUMA-systems.

RCL also has no mechanism for automatic selection of processor cores for the server thread and the working threads while considering the hierarchical structure of the computer system and existing affinities. Processor affinity greatly affects the overheads caused by localization of access to global variables. Therefore, user threads should be executed on the processors cores, located "closer" to the processor core of the RCL-server. In existing RCL implementation, users have to manually choose the affinity for all threads. In so doing, they should consider existing affinity of the RCL-server and working threads. Thus, the development of tools for the automation of this procedure constitute a current problem.

This work proposes an algorithm for RCL-lock initialization that recognizes memory affinity to NUMA-nodes and RCL-server affinity to processor cores, as well as an algorithm for the sub-optimal affinity of working threads to processor cores. These algorithms take into account the hierarchical structure of multi-core CS and non-uniform memory access in NUMA-systems in order to minimize the execution time of critical sections.

2. Multi-Core Hierarchical Computer System Model

Let there be multi-core CS with shared memory, including N processor cores: $p = \{1, 2, \ldots, N\}$. The computer system has a hierarchical structure, which can be described as a tree, comprising L levels (Figure 2). Each level of the system is represented by individual types of structural element in the CS (NUMA-nodes, processor cores and multilevel cache-memory). We introduce the following notation: c_{lk}—the number of processor cores possessed by children of an element $k \in \{1, 2, \ldots, n_l\}$ of level $l \in \{1, 2, \ldots, L\}$; $r = p(l, k)$ – the first direct parent element $r \in \{1, 2, \ldots, n_{l-1}\}$ for an element k, located on the level l; m—the number of NUMA-nodes of the multi-core CS; $j = m(i)$—the number $j \in \{1, 2, \ldots, m\}$ of NUMA-nodes containing a processor core i; q_i—the set of processor cores belonging to the NUMA-node i.

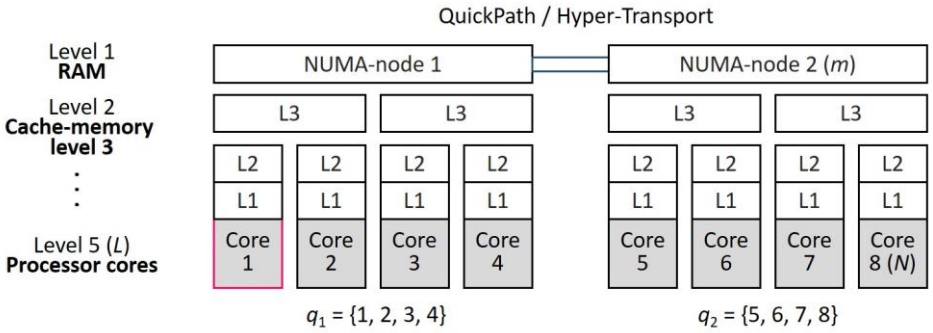

Figure 2. An example of the hierarchical structure of a multi-core CS, $N = 8$, $L = 5$, $m = 2$, $c_{23} = 2$, $p(3; 4) = 2$, $m = 2$, $m(3) = 1$.

3. RCL Optimization Algorithms

For memory affinity optimization and RCL-server processor affinity optimization we propose the algorithm RCLLockInitNUMA for the initialization of RCL-locks (Algorithm 1). The algorithm takes into account non-uniform memory access, and is performed during the initialization of the RCL-lock.

Algorithm 1. RCLLockInitNUMA.

```
1:  /* Compute the number of free cores on CS and nodes. */
2:  node_usage[1, . . . , m] = 0
3:  nb_free_cores = 0
4:  for i = 1 to N do
5:      if IsRCLServer(i) then
6:          node_usage[m(i)] = node_usage[m(i)] + 1
7:      else
8:          nb_free_cores = nb_free_cores + 1
9:      end if
10: end for
11: /* Try to set memory affinity to the NUMA-node. */
12: nb_busy_nodes = 0
13: for i = 0 to m do
14:     if node usage[i] > 0 then
15:         nb busy nodes = nb busy nodes + 1
16:         node = i
17:     end if
18: end for
19: if nb_busy_nodes = 1 then
20:     SetMemBind(node)
21: end if
22: /* Set the affinity of RCL-server. */
23: if nb_free_cores = 1 then
24:     core = GetNextCoreRR()
25: else
26:     n = GetMostBusyNode(node_usage)
27:     for i = 1 to q_n do
28:         if not IsRCLServer(i) then
29:             core = i
30:             break
```

```
31:         end if
32:     end for
33: end if
34: RCLLockInitDefault(core)
```

In the first stage of the algorithm (lines 2–10), we compute the number of processor cores which are not busy by RCL-server and the number of free processor cores on each of NUMA-nodes. Subsequently (lines 12–18), we compute the summary number of NUMA-nodes with the RCL-servers running on it. If there is only one such NUMA-node, we set the memory affinity to this node (lines 19–21).

The second stage of the algorithm (lines 23–34) includes a search for sub-optimal processor cores, and the binding of the RCL-server to it. If there is only one processor core in the system which is not busy by RCL-server, we set the affinity of the RCL-server to the first next (occupied) processor core (lines 23–24). One core is always kept free, on which to run working threads. If there is more than one free processor core in the system, we search for the least busy NUMA-node (line 26), and set the affinity of the RCL-server to the first free core in this node (lines 27–32). The algorithm concludes with a call of the default function of the RCL-lock initialization with obtained processor affinity (line 34).

For the optimization of working thread affinity, we propose the heuristic algorithm RCLHierarchicalAffinity (Algorithm 2). The algorithm takes into account the hierarchical structure of multi-core CS in order to minimize the execution time of multithreaded programs with RCL. This algorithm is executed each time a parallel thread is created.

Algorithm 2. RCLHierarchicalAffinity.

```
1:  if IsRegularThread(thr_attr) then
2:      core_usage[1, ..., N] = 0
3:      for i = 1 to N do
4:          if IsRCLServer(i) then
5:              nthr_per_core = 0
6:              l = L
7:              k = i
8:              core = 0
9:              /* Search for the nearest processor core. */
10:             do
11:                 /* Find the first covering parent element. */
12:                 do
13:                     c_lk_prev = c_lk
14:                     k = p(l; k)
15:                     l = l − 1
16:                 while clk = clk_prev or l = 1
17:                 /* When the root is reached, increase the minimal count
18:                     of threads per one core. */
19:                 if l = 1 then
20:                     nthr_per_core = nthr_per_core + 1
21:                     obj = i
22:                 else
23:                     /* Find the first least busy processor core. */
24:                     for j = 1 to c_lk do
25:                         if core_usage[j] ≤ nthr_per_core then
26:                             core = j
27:                             break
28:                         end if
```

```
29:                         end for
30:                     end if
31:                 while core = 0
32:                 SetAffinity(core, thr_attr)
33:                 core_usage[core] = core_usage[core] + 1
34:                 return
35:             end if
36:         end for
37: end if
```

In the first stage of the algorithm (line 1), we check whether the thread is an RCL-server. For each common (working) threads, we search for all of the RCL-servers (lines 3–4) being executed in the system. When the first processor core with an RCL-server is found, this core becomes the current element (lines 6–7), and for this element we search for the nearest free processor core for the affinity of the created thread (lines 4–35). At the beginning of the algorithm, we consider that a processor core with no affined threads is a free core (line 5).

In the first stage of the core search, we find the first covering element of hierarchical structure. The covering element contains the current element and some any other processor cores except current element (lines 12–16). When the uppermost element of the hierarchical structure is reached, we increment the minimal number of threads per core (the free core is now the core with a greater number of threads running on it) (lines 19–21). When the covering element is found, we search for the first free processor core in it (lines 24–29), and set the affinity of the created thread to it (line 32), whereby, for this core, the number of threads executed on it is increased (line 33). After the affinity of the thread is set, the algorithm is finished (line 34).

We implemented RCLLockInitNUMA and RCLHierarchicalAffinity algorithms in a library that utilizes the RCL method. We expect that the use of a modified RCL library will reduce the time required for critical section execution and critical section throughput for existing multithreaded programs that use RCL.

The algorithms utilize non-uniform memory access and the hierarchical structure of the system (caching overheads) and access contention, so the effect of using the algorithms depends on the type of the system (NUMA or SMP) on which the multithreaded programs are executed, the structure of the system (number of hierarchical levels, existence of shared caches), the number of working threads, existing affinities of the threads (RCL-server thread, working threads and a thread which allocates memory), and the patterns of access to the memory of data structures. Thus, the efficiency of the algorithms has to be evaluated based on these factors.

4. Experimental Results

For the evaluation of the algorithms, it is necessary to use shared memory systems with uniform and non-uniform memory access (NUMA and SMP). While RCLLockInitNUMA is focused on NUMA systems, RCLHierarchicalAffinity can be used with both system types. Benchmarks should consider different access patterns to the memory of the data structures. The evaluation should be done for different numbers of working threads (taking into account the number of processor cores in the system).

The experiments were conducted on the multi-core nodes of the computer clusters Oak and Jet in the multicluster computer system of at the Center of Parallel Computational Technologies at the Siberian State University of Telecommunications and Information Sciences. The node of the Oak cluster (a NUMA system) includes two quad-core Intel Xeon E5620 processors (2.4 GHz, with sizes of cache-memory of 32 KiB, 256 KiB, and 12 MiB for levels 1, 2 and 3, respectively) and 24 GiB of RAM (Figure 3a). The ratio of rate of access to local and remote NUMA-nodes is 21 to 10. The node of the Jet cluster (a SMP system) is equipped with a quad-core Intel Xeon E5420processor (2.5 GHz, with sizes of cache-memory of 32 KiB and 12 MiB for levels 1 and 2, respectively) and 8 GiB of RAM (Figure 3b).

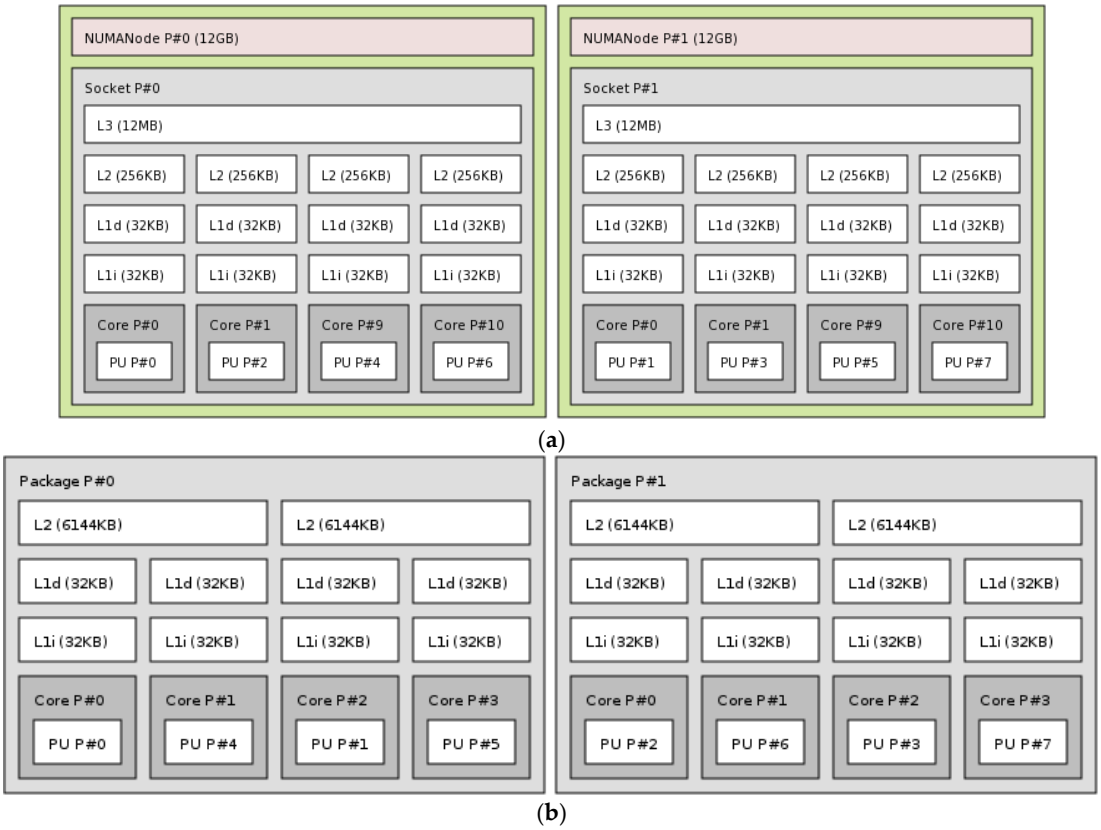

Figure 3. Hierarchical structure of the computing nodes of clusters Oak and Jet. (**a**) Computer node of cluster Oak; (**b**) Computer node of cluster Jet.

The operating systems GNU/Linux CentOS 6 (Oak) and Fedora 21 (Jet) are installed on the computing nodes. The compiler GCC 5.3.0 was used.

Using the described systems, we aim to study how non-uniform memory access (Oak node) affects the efficiency of the algorithms. Additionally, we will determine how efficiency is affected by the shared cache (L3 in the Oak node and L2 in the Jet node). We expect the algorithms will be more efficient in the NUMA system with the deeper hierarchy.

We developed a benchmark for the evaluation of the algorithms. The benchmark performs iterative access to elements of integer arrays of length $b = 5 \times 10^8$ elements inside the critical section, organized with the RCL. The number of operations is $n = 10^8/\mathrm{p}$. As a pattern of operations in the critical section, we used an increment of the integer variable by 1 (this operation is executed at each iteration). We used three memory access patterns:

- sequential access: on each new iteration, choose the element that follows the previous one;
- strided access (interval-based access): on each new iteration, choose the element for which the index exceeds the previous one by $s = 20$;
- random access: on each iteration, randomly choose the element of the array.

It is well known that these access patterns are the most common in multithreaded programs. We plan to study how the memory access pattern affects the efficiency of the proposed algorithms; and according to the results of the experiments, we will provide recommendations for multithreading programming practice.

The number p of parallel threads was varied between 2 and 7 (7—number of processor cores on the computer nodes which are not busy by RCL-server) in the first experiment, and between 2 and 100 in the second one. Here, we aim to evaluate the efficiency of the algorithm depending on

the number of working threads (and how this number relates to the number of processor cores). We expect the efficiency of RCLLockInitNUMA does not depend on the number of threads, while RCLHierarchicalAffinity will be more effective when all the threads (including RCL-server) are bound to the processor cores of one NUMA node.

The throughput $b = n/t$ of the critical section was used as an indicator of efficiency (here, n is the number of operations, and t is the time of benchmark execution). Note that throughput is the most common efficiency indicator in studies on synchronization techniques.

We compared the efficiency of the algorithms for the initialization of RCL-lock; RCLLockInitDefault (current RCL-lock initialization function) and RCLLockInitNUMA. Additionally, we compared the affinity of the threads obtained by the algorithms RCLHierarchicalAffinity with other arbitrary affinities.

In the experiments for the evaluation of RCL-lock initialization algorithms, we used the thread affinity depicted in Figure 4. Additionally, we conducted the experiment without fixing the affinity of the working threads in order to study how fixed affinity affects the efficiency of the algorithms for lock initialization.

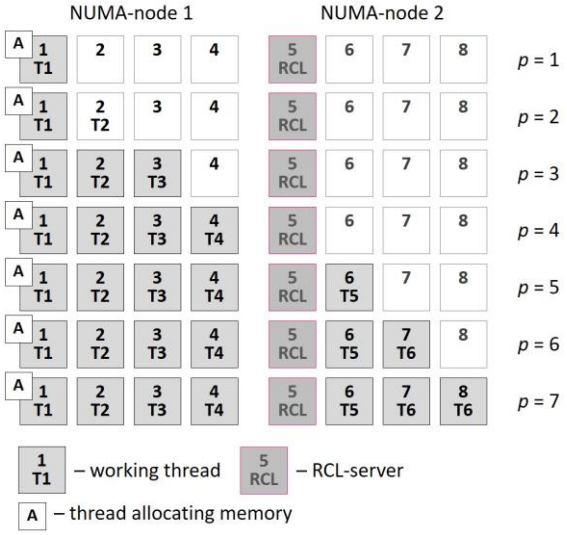

Figure 4. Thread affinity to the processor cores in the experiments, $p = 2, \ldots, 7$.

Figure 5 depicts the dependence of the critical section's throughput b on the number p of working threads. We can see that the algorithm RCLLockInitNUMA minimizes the throughput of the critical section for random access and strided access to the elements of the test array by 10–20%. We can explain this by the fact that, in these access patterns, data is not cached in the local cache of processor core on which RCL-server is running. Therefore RCL-server addresses the RAM directly, whereby the access rate depends on the data location in the local or remote NUMA-node. The effect is perceptible when the number of threads is near to the number of processor cores (Figure 5a,b) and exceeds it (Figure 5c), and does not change significantly when the number of threads changes. The fixed affinity of threads to processor cores (Figure 5a) does not significantly affect the results.

Figures 6 and 7 represent the experimental results obtained for different affinities for the benchmark. Here, the number p of working threads does not exceed the number of cores, because the affinity makes sense only in those conditions. The results show that the algorithm RCLHierarchicalAffinity significantly increases critical section throughput. The effect of the algorithms depends on the number of threads (up to 2.4 times at $p = 2$, up to 2.2 times at $p = 3$, up to 1.3 times at $p = 4$, up to 1.2 times at $p = 5$) and on the access pattern (up to 1.5 times for random access, up to 2.4 times for sequential access and up to 2.1 times for strided access). In the case of a small number of cores, RCL-server and working threads do not share cache memory or the memory of one NUMA node; therefore, the overheads due to cache misses or memory access via NUMA bus are significant.

As we expected, the efficiency of the RCLHierarchicalAffinity decreased when the number of threads was greater than the number of processor cores on the NUMA node. This is due to the large overheads when the RCL-server uses the NUMA bus to access memory that was previously accessed on a thread which has been affined to another NUMA-node.

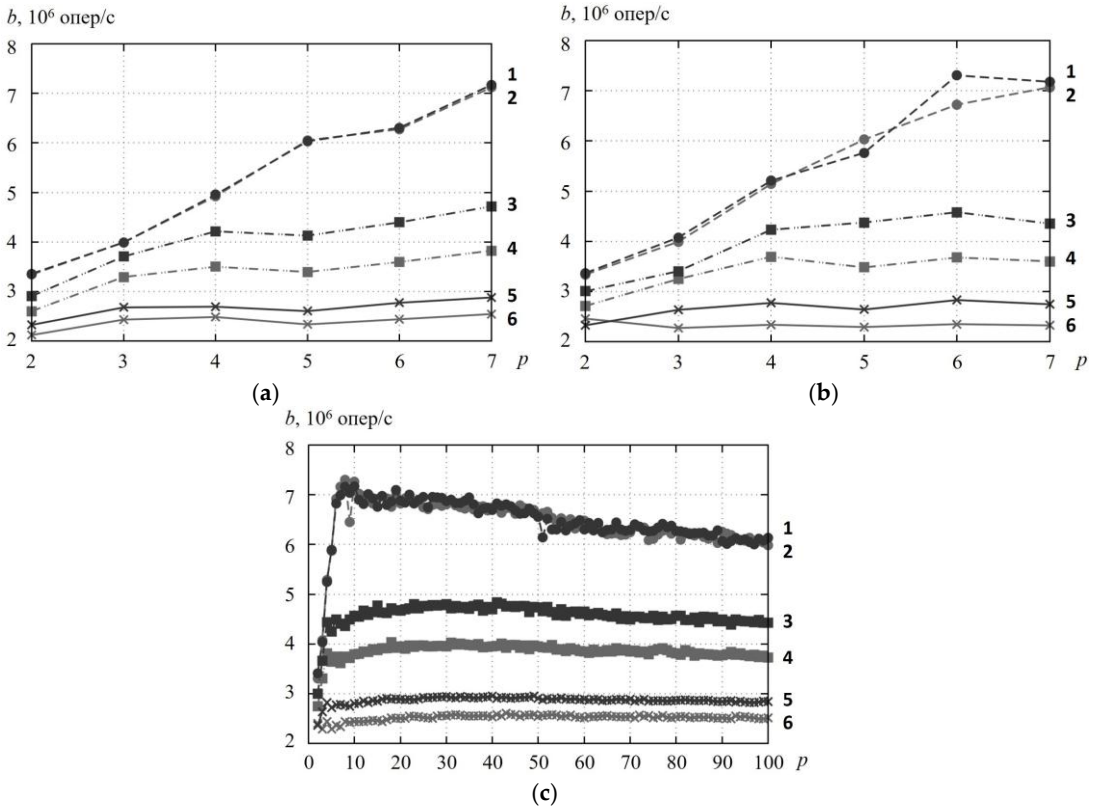

Figure 5. Efficiency of the algorithms for RCL-lock initialization on the Oak cluster. (**a**) $p = 2, \ldots, 7$, thread affinity corresponds to Figure 4; (**b**) $p = 2, \ldots, 7$, no thread affinity; (**c**) $p = 2, \ldots, 100$. 1—RCLLockInitNUMA, sequential access, 2—RCLLockInitDefault, sequential access, 3—RCLLockInitNUMA, strided access, 4—RCLLockInitDefault, strided access, 5—RCLLockInitNUMA, random access, 6—RCLLockInitDefault, random access.

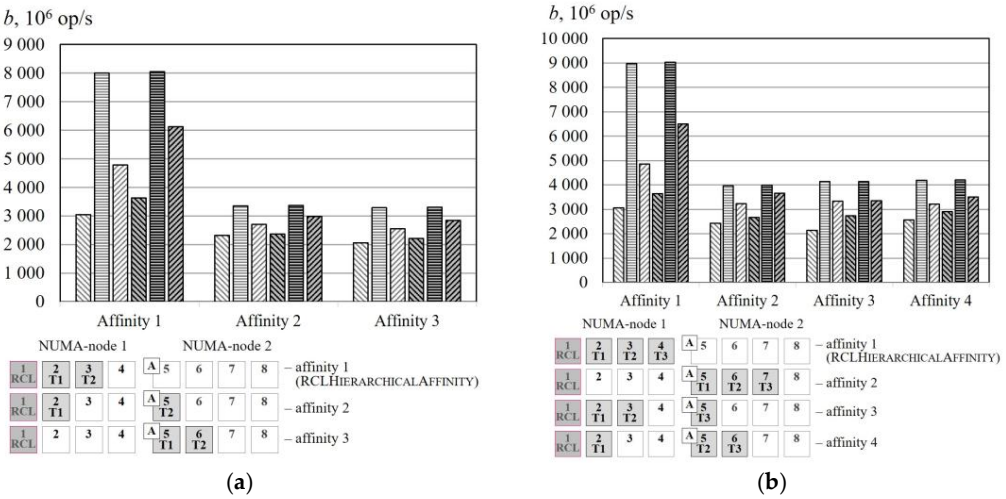

Figure 6. *Cont.*

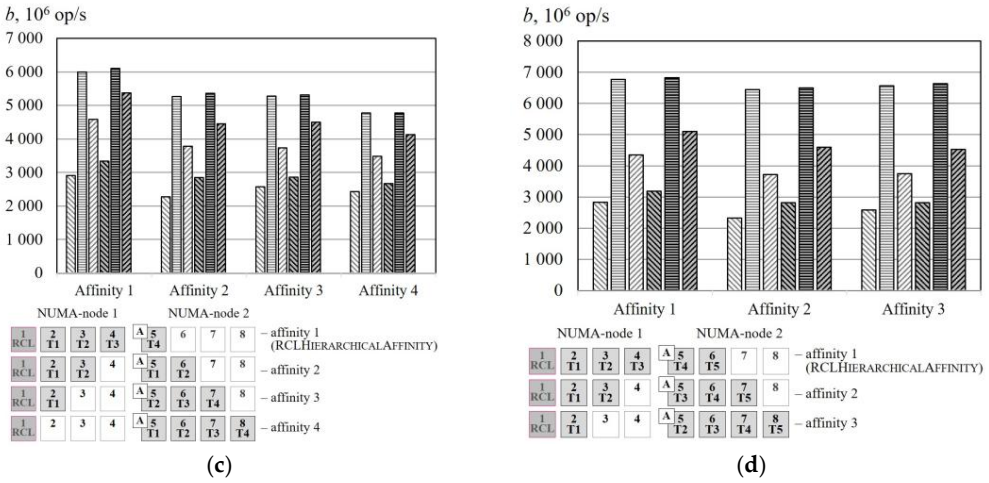

Figure 6. Thread affinity efficiency comparison, Oak cluster (NUMA system). (**a**) p = 2; (**b**) p = 3; (**c**) p = 4; (**d**) p = 5. ▧—RCLLockInitDefault, random access, ▤—RCLLockInitDefault, sequential access, ▨—RCLLockInitDefault, strided access, ▩—RCLLockInitNUMA, random access, ▬—RCLLockInitNUMA, sequential access, ▨—RCLLockInitNUMA, strided access. —working thread, —RCL-server, —thread allocating the memory.

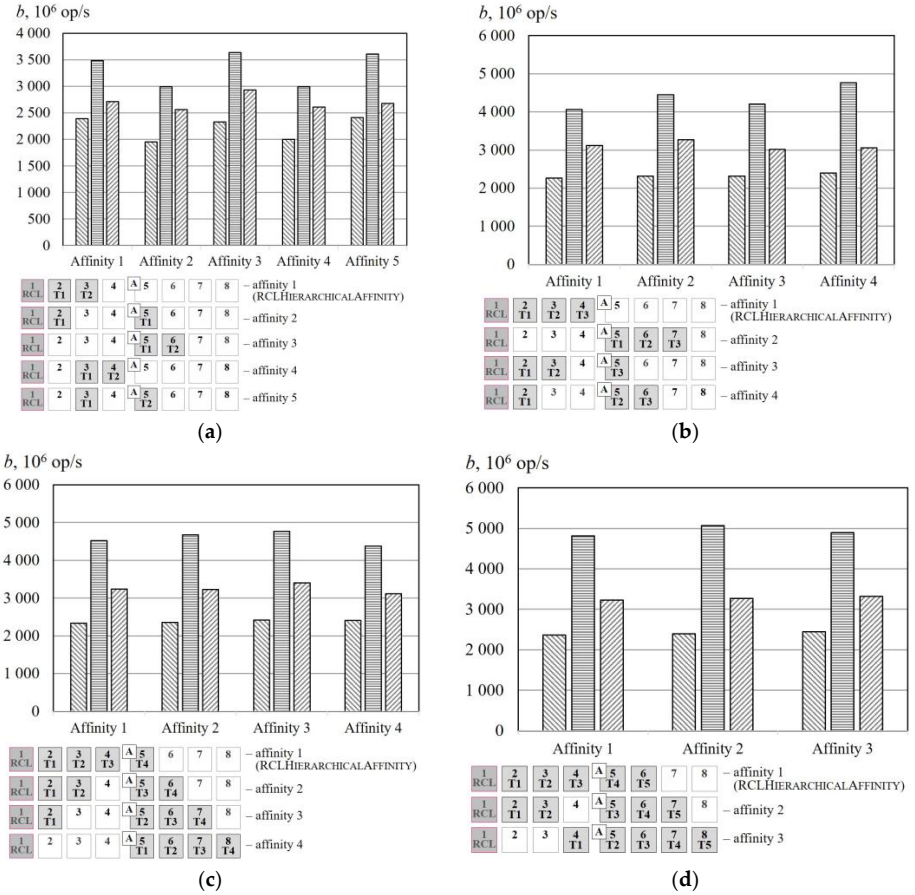

Figure 7. Thread affinity efficiency comparison, Jet cluster (SMP system). (**a**) p = 2; (**b**) p = 3; (**c**) p = 4; (**d**) p = 5. ▧—RCLLockInitDefault, random access, ▤—RCLLockInitDefault, sequential access, ▨—RCLLockInitDefault, strided access, ▩—RCLLockInitNUMA, random access, ▬—RCLLockInitNUMA, sequential access, ▨—RCLLockInitNUMA, strided access. —working thread, —RCL-server, —thread allocating the memory.

Critical section throughput for benchmark execution on the node of the Jet cluster (Figure 7) insignificantly varies for different thread affinities. This is explained by the lack of a shared (for processor cores of one processor) cache, and uniform access to memory (SMP-system).

5. Conclusions

Algorithms for optimization of the execution of multithreaded programs based on Remote Core Locking (RCL) were developed. We proposed the algorithm RCLLockInitNUMA for initialization of RCL-lock while taking into account the non-uniform memory access in multi-core NUMA-systems, and the algorithm RCLHierarchicalAffinity for sub-optimal thread affinity in hierarchical multi-core computer systems.

The algorithm RCLLockInitNUMA increases the throughput of the critical sections of multithreaded programs with random access and strided access to the elements of arrays on a NUMA systems by an average of 10–20%. Optimization is achieved by means of the minimization of the number of addresses to remote memory NUMA-segments. The algorithm RCLHierarchicalAffinity increases the throughput of the critical section by up to 1.2–2.4 times for all access templates on NUMA computer systems. The algorithms realize the affinity while taking into account all of the hierarchical levels of multi-core computer systems.

The developed algorithms are realized as a library and can be used to minimize existing multithreaded programs based on RCL.

Acknowledgments: The paper has been prepared within the scope of the state project "Initiative scientific project" of the main part of the state plan of the Ministry of Education and Science of the Russian Federation (task No. 2.6553.2017/8.9 BCH Basic Part).

Author Contributions: A.P. and Y.S. conceived and designed the experiments; A.P. performed the experiments; A.P. and Y.S. analyzed the data; A.P. and Y.S. wrote the paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Khoroshevsky, V.G. Distributed programmable structure computer systems. *Vestnik SibGUTI* **2010**, *2*, 3–41.
2. Herlihy, M.; Shavit, N. *The Art of Multiprocessor Programming*; Revised Reprint; Elsevier: Amsterdam, The Netherlands, 2012; p. 528.
3. Herlihy, M.; Moss, J.E.B. Transactional memory: Architectural support for lock-free data structures. In Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, CA, USA, 16–19 May 1993; Volume 21, pp. 289–300.
4. Shavit, N. Data Structures in the Multicore Age. In *Communications of the ACM*; ACM: New York, NY, USA, 2011; Volume 54, pp. 76–84.
5. Shavit, N.; Moir, M. Concurrent Data Structures. In *Handbook of Data Structures and Applications*; Metha, D., Sahni, S., Eds.; Chapman and Hall/CRC Press: Boca Raton, FL, USA, 2004; Chapter 47; pp. 47-1–47-30.
6. Dechev, D.; Pirkelbauer, P.; Stroustrup, B. Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. In Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), Carmona, Spain, 4–7 May 2010; pp. 185–192.
7. Michael, M.M.; Scott, M.L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, PA, USA, 23–26 May 1996; pp. 267–275.
8. Anderson, T.E. The performance of spin lock alternatives for shared-money multi-processors. *IEEE Trans. Parallel Distributed Syst.* **1990**, *1*, 6–16. [CrossRef]
9. Mellor-Crummey, J.M.; Scott, M.L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst. (TOCS)* **1991**, *9*, 21–65. [CrossRef]

10. Hendler, D.; Incze, I.; Shavit, N.; Tzafrir, M. Flat combining and the synchronization-parallelism tradeo. In Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures, Santorini, Greece, 13–15 June 2010; pp. 355–364.
11. Fatourou, P.; Kallimanis, N.D. Revisiting the combining synchronization technique. In *ACM SIGPLAN Notices*; ACM: New York, NY, USA, 2012; Volume 47, pp. 257–266.
12. Oyama, Y.; Taura, K.; Yonezawa, A. Executing parallel programs with synchronization bottlenecks efficiently. In Proceedings of the International Workshop on PDSIA 99, Sendai, Japan, 5–7 July 1999; pp. 1–24.
13. Suleman, M.A.; Mutlu, O.; Qureshi, M.K.; Patt, Y.N. Accelerating critical section execution with asymmetric multi-core architectures. In *ACM SIGARCH Computer Architecture News*; ACM: New York, NY, USA, 2009; Volume 37, pp. 253–264.
14. Metreveli, Z.; Zeldovich, N.; Kaashoek, M.F. Cphash: A cache-partitioned hash table. In *ACM SIGPLAN Notices*; ACM: New York, NY, USA, 2012; Volume 47, pp. 319–320.
15. Calciu, I.; Gottschlich, J.E.; Herlihy, M. Using elimination and delegation to implement a scalable NUMA-friendly stack. In Proceedings of the Usenix Workshop on Hot Topics in Parallelism (HotPar), San Jose, CA, USA, 24–25 June 2013; p. 17.
16. Lozi, J.P.; David, F.; Tomas, G.; Lawall, J.; Muller, G. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. In Proceedings of the USENIX Annual Technical Conference, Boston, MA, USA, 13–15 June 2012; p. 6576.
17. Lozi, J.P.; Thomas, G.; Lawall, J.L.; Muller, G. *Efficient Locking for Multicore Architectures*; Research Report RR-7779; INRIA: Paris, France, 2011; pp. 1–30.