Optimization of thread affinity and memory affinity for remote core locking synchronization in multithreaded programs for multicore computer systems

Alexey Paznikov
Saint Petersburg Electrotechnical University "LETI", Saint Petersburg, Russia
E-mail: *apaznikov@gmail.com*

Received 28 May 2017; accepted 29 May 2017
DOI https://doi.org/10.21595/vp.2017.18689

Abstract. This paper proposes the algorithms for optimization of Remote Core Locking (RCL) synchronization method in multithreaded programs. The algorithm of initialization of RCL-locks and the algorithms for threads affinity optimization are developed. The algorithms consider the structures of hierarchical computer systems and non-uniform memory access (NUMA) to minimize execution time of RCL-programs. The experimental results on multi-core computer systems represented in the paper shows the reduction of RCL-programs execution time.

Keywords: remote core locking, RCL, synchronization, critical sections, scalability.

1. Introduction

Currently distributed computer systems (CS) [1] are large-scale and include multiple architectures. These systems are composed of shared memory multi-core compute nodes (SMP, NUMA systems) equipped by universal processors as well as specialized accelerators. System software must consider the large scale, multiple architectures and hierarchical structure. Parallel programs for multi-core CS are multithreaded in most cases. The software must ensure linear speedup with a large amount of threads. Thread synchronization while accessing to shared data structures is one of the most significant problem in multithreading programming. The existing approaches for thread synchronization include locks, lock-free algorithms and concurrent data structures [2] and software transactional memory [3].

The main drawbacks of lock-free algorithms and data structures are the limited application scope and high complexity of development of programs [2, 4, 5]. Furthermore, the development of lock-free algorithms and data structures includes the problems, connected with ABA problem [6, 7], poor performance and restricted nature of atomic operations. Software transactional memory nowadays have a variety of issues and does not ensure sufficient performance of multithreaded programs and is not applied in common real applications for now.

Conventional approach for synchronization by lock-based critical sections is still the most widespread. Locks are simple in usage and ensure the acceptable performance. Furthermore, the most of existing multithreaded programs utilizes lock-based approach. Thereby scalable algorithms and software tools for lock-based synchronization is very important today. Lock scalability depends on resolutions of the problems, connected with access contention of threads and locality of references. Access contention arises when multiple threads access to a critical section, protected by one lock. In the terms of hardware, it leads to the huge load to the data bus and cache memory inefficiency.

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]. Some of the existing works also includes the methods for localization access to cache memory [10, 13-15]. The works [14, 15] are devoted to concurrent data structures development (lists and hash tables) based on critical section execution on dedicated processor cores. The paper [13] propose 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 and suggests the server threads (all the threads become server threads by order), which execute critical sections.

This paper considers Remote Core Locking (RCL) method [16, 17]. RCL minimizes execution time of existing programs thanks to critical path reduction. This technique assumes replacement of high-load critical sections in existing multithreading applications to remote functions calls for its execution on dedicated processor cores (Fig. 1).

The current implementation of RCL has several drawbacks. There is no memory affinity for NUMA systems. The latency of RCL-server addresses to the shared memory areas is essential for the execution time of a program. Memory allocation on the NUMA-node, which is not local to the node, on which RCL-server is running, leads to severe overheads when RCL-server addresses to the variables allocated on remote nodes. RCL also has no mechanism of automatic selection of processor cores for server thread and working threads with considering the structure of computer system and existing affinity. The processor affinity greatly effects on the overheads caused by localization of access to the global variables. Therefore, user threads should be executed on the processors cores, located as more "close" to the processor core of RCL-server. In the existing RCL implementation user have to choose processor core for RCL-server and working threads by hands. Thus, the tools for automation of this procedure is the actual problem.

This work proposes the algorithm of RCL-locks initialization, which realizes memory affinity to the NUMA-nodes and RCL-server affinity to the processor cores, and the algorithm of sub-optimal affinity of working threads to the processor cores. The algorithms consider the hierarchical structure of multi-core CS and non-uniform memory access in NUMA-systems to minimize critical sections execution time.

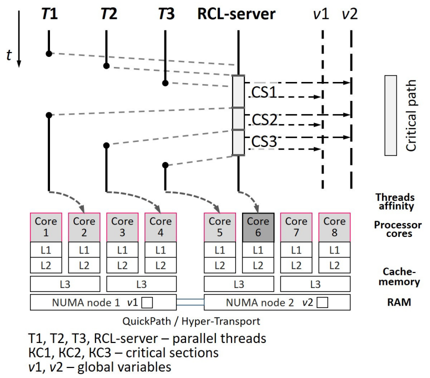

Fig. 1. Remote core locking (RCL)

2. RCL optimization algorithms

Let there is multi-core CS with shared memory, including N processor cores: $P = \{1, 2, \ldots, N\}$. Computer system has hierarchical structure, which can be described as a tree, comprising L levels Fig. 2. Each level of the system is represented by individual type of structural elements of CS (NUMA-nodes, processor cores and multilevel cache-memory). We introduce the notation: c_{lk} – the number of processor cores, owned by childs of an element $k \in \{1, 2, \ldots, n_l\}$ from the 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 multi-core CS; $j = m(i)$ – number $j \in \{1, 2, \ldots, m\}$ of NUMA-node, containing a processor core i; q_i – the set of processor cores, belonging to the NUMA-node i.

For the memory affinity optimization and RCL-server processor affinity optimization we propose the algorithm RCLLockInitNUMA of initialization of RCL-lock Fig. 3(a). The algorithm considers the non-uniform memory access and is performed during the initialization of RCL-lock.

On the first stage (lines 2-10) we compute the number of cores which is not busy by the RCL-server and used on each of NUMA-node. Further (lines 12-18) we compute the summary number of NUMA-nodes with the RCL-servers. If there is only one such node, we set the memory affinity to this node (lines 19-21). The second stage of the algorithm (lines 23-34) includes the search of sub-optimal processor core and the affinity of RCL-server to it. If there is only one not busy by RCL-server processor core in the system, set the affinity of the RCL-server to the next already occupied processor core (lines 23-24). One core is always free to run working threads on it. If there are more than one free processor core in the system, we search the least busy NUMA-node (line 26) and set the affinity of RCL-server to the first free core in this node (lines 27-32). The algorithms are finished by call of function of RCL-lock initialization with selected affinity.

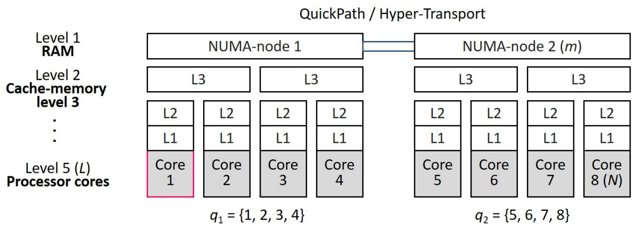

Fig. 2. An example of hierarchical structure of multi-core CS
$N = 8, L = 5, m = 2, c_{23} = 2, p(3;4) = 2, m = 2, m(3) = 1$

RCLLockInitNUMA

```
1   /* Compute the number of free cores. */
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 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)
```

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 core. */
10        do
11          /* Find the first covering element. */
12          do
13            $c_{lk}$_prev = $c_{lk}$
14            k = p(l; k)
15            l = l – 1
16          while $c_{lk}$ = $c_{lk}$_prev or l = 1
17          /* When the root is reached, increase
18             the number of threads per 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 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
```

a *b*

Fig. 3. Algorithms RCLLockInitNUMA and RCLHierarchicalAffinity

For the optimization of thread affinity, we propose the algorithm RCLHierarchicalAffinity Fig. 3(b). The algorithm takes into account the structure of CS to minimize the execution time of programs with RCL. That algorithm is executed each time when parallel thread is created.

On the first stage (line 1) we check if the thread is not RCL-server. For each of thread we search all the RCL-servers (lines 3-4) executed in the system. After that when first processor core with RCL-server is found, this core becomes the current element (lines 6-7) and we search the nearest free processor core for the affinity of created thread (lines 4-35). In the beginning of the algorithm processor core with no attached threads is the free core (line 5). On the first stage of the core search we find first covering element of hierarchical structure. Covering element includes current element and contains more number of processor cores than it (lines 12-16). When the uppermost element of hierarchical structure is reached we increment the minimal number of threads per core (the free core now is the core with more number of attached threads) (lines 19-21). When the covering element is found we search first free core in it (lines 24-29) and set the affinity of created thread to it (line 32). Wherein 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).

3. Experimental results

The experiments were conducted on the nodes of computer clusters Oak and Jet of Center of parallel computational technologies of Siberian state university of telecommunications and information sciences. The node of Oak includes two quad-core processors Intel Xeon E5620. The ratio of rate of access to local and remote NUMA-nodes are 21 to 10. The node of cluster Jet is equipped by quad-core processor Intel Xeon E5420. On the computing nodes, the operating systems CentOS 6 (Oak) and Fedora 21 (Jet) are installed. The compiler GCC 5.3.0 was used.

The benchmark performs iterative access to the elements of integer array of length $b = 5\times10^8$ elements inside the critical sections, organized with RCL. The number of operations $n = 10^8/p$. As the operation in the critical section we used the increment of a variable by 1. As the access patterns three patterns were used: sequential access (on each new iteration choose the element, following the previous one), strided access (choose the element with the index more by $s = 20$ than the previous one), random access (randomly choose the element). Number p of parallel threads was varied from 2 to 7 (7 – number of not busy by RCL-server cores on the computing node) in the first experiment and from 2 to 100 in the second one. The throughput $b = n/t$ of the structure was used as an indicator of efficiency (here t is time of benchmark execution).

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

The Fig. 4 depicts the throughput b of critical section with number p of working threads. We can see that the algorithm RCLLockInitNUMA minimizes by 10-20 % the throughput of critical section at random access to the elements of test array and at strided access. In these access patterns data is not cached in the local cache of processor core, on which RCL-server is running. Therefore RCL-server address to the RAM directly, wherein the access rate depends on data location in local or remote NUMA-node. The effect is perceptible at the number of threads comparable to the number of processor cores Fig. 4(a), as like for greater number of threads Fig. 4(b) and significantly do not change when the number of threads is changing. The fixed affinity of threads to processor cores insignificantly affect to the throughput.

The Fig. 5 represent the experimental results of different affinities for the benchmark. The algorithm RCLHierarchicalAffinity significantly increases critical section throughput. The effect of the algorithms depends on 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 an access pattern (up to 1.5 times at random access, up to 2.4 times at sequential access and up to 2.1 times at strided access).

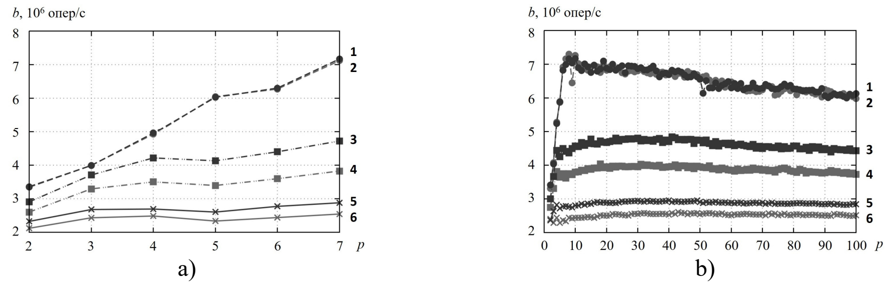

Fig. 4. Efficiency of the algorithms of RCL-lock initalization, cluster Oak. $a - p = 2, \ldots, 7$, $b - p = 2, \ldots, 100$. 1 – RCLLockInitNUMA, sequential access, 2 – RCLLockInitDefault, sequential access, 3 – RCLLockInitNUMA, strided access, 4 – RCLLockInitDefault, strided access, 5 – RCLLockInitNUMA, random access, 6 – RCLLockInitDefault, random access

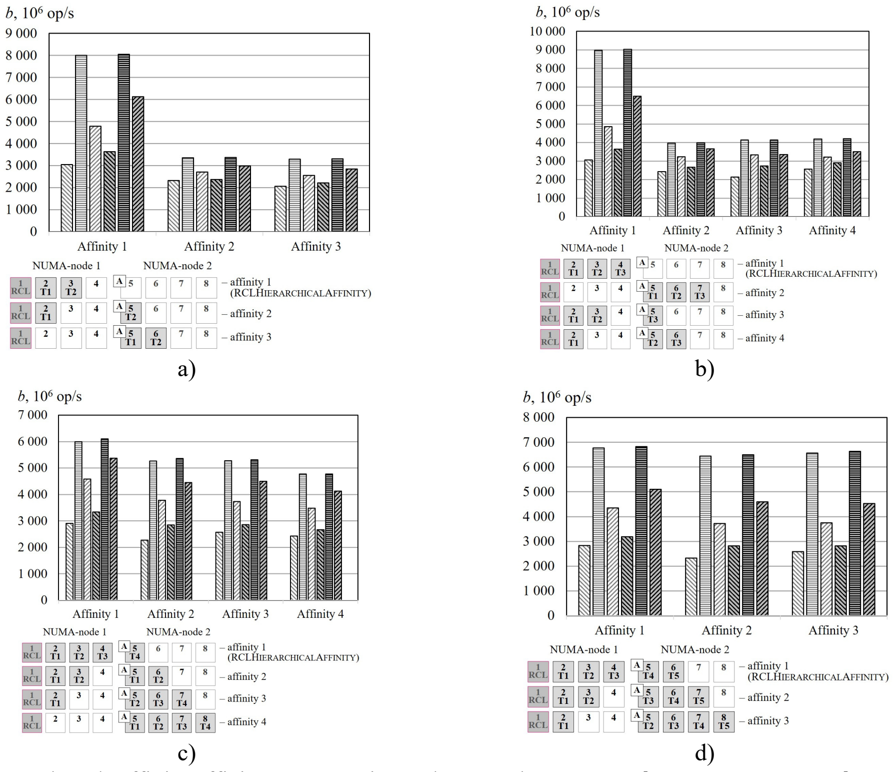

Fig. 5. Threads affinity efficiency comparison, cluster Oak. $a - p = 2$, $b - p = 3$, $c - p = 4$, $d - p = 5$.
▧ – RCLLockInitDefault, random access, ▤ – RCLLockInitDefault, sequential access,
▨ – RCLLockInitDefault, strided access, ▧ – RCLLockInitNUMA, random access,
▤ – RCLLockInitNUMA, sequential access, ▨ – RCLLockInitNUMA, strided access,
2 T1 – working thread, 1 RCL – RCL-server, A – thread allocating the memory

4. Conclusions

We proposed the algorithm RCLLockInitNUMA of initialization of RCL-lock with considering the non-uniform memory access in multi-core NUMA-systems and the algorithm RCLHierarchicalAffinity of sub-optimal thread affinity in hierarchical multi-core computer systems. The algorithm RCLLockInitNUMA increases by 10-20 % at the average the throughput of critical sections of parallel multithreaded programs based on RCL at random access and strided access to the elements of arrays on the NUMA multi-core systems. The optimization is reached by means of minimization of number of addresses to remote memory NUMA-segments. The algorithm RCLHierarchicalAffinity increases the throughput of critical section up to 1.2-2.4 times for all access templates on some computer systems. The algorithms realize the affinity with considering all the hierarchical levels of multi-core computer systems. The developed algorithms are realized as a library and may be used to minimize existing programs based of RCL.

Acknowledgements

The reported study was partially supported by RFBR, research Projects 15-07-02693a, 15-07-00653a, 16-07-00712a, 15-37-20113a. 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 Russian Federation (Task No. 2.6553.2017/BCH Basic Part). The paper is supported by the Contract No. 02.G25.31.0149 dated 01.12.2015 (Board of Education of Russia).

References

[1] **Khoroshevsky V. G.** Distributed programmable structure computer systems. Vestnik SibGUTI, Vol. 2, 2010, p. 3-41.

[2] **Herlihy M., Shavit N.** The Art of Multiprocessor Programming. Revised Reprint, Elsevier, 2012, p. 528.

[3] **Herlihy M., Moss J. E. B.** Transactional memory: architectural support for lock-free data structures. Proceedings of the 20th Annual International Symposium on Computer Architecture, ACM, Vol. 21, Issue 2, 1993, p. 289-300.

[4] **Shavit N.** Data structures in the multicore age. Communications of the ACM, NY, USA, Vol. 54, Issue 3, 2011, p. 76-84.

[5] **Shavit N., Moir M.** Concurrent Data Structures. Handbook of Data Structures and Applications. Chapter 47. Chapman and Hall/CRC Press, 2004, p. 47-30.

[6] **Dechev D., Pirkelbauer P., Stroustrup B.** Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, 2010, p. 185-192.

[7] **Michael M. M., Scott M. L.** Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing, ACM, 1996, p. 267-275.

[8] **Anderson T. E.** The performance of spin lock alternatives for shared-money multi-processors. IEEE Transactions on Parallel and Distributed Systems, Vol. 1, Issue 1, 1990, p. 6-16.

[9] **Mellor-Crummey J. M., Scott M. L.** Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems, Vol. 9, Issue 1, 1991, p. 21-65.

[10] **Hendler D., et al.** Flat combining and the synchronization-parallelism tradeo. 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, ACM, 2010, p. 355-364.

[11] **Fatourou P., Kallimanis N. D.** Revisiting the combining synchronization technique. ACM SIGPLAN Notices, ACM, Vol. 47, Issue 8, 2012, p. 257-266.

[12] **Oyama Y., Taura K., Yonezawa A.** Executing parallel programs with synchronization bottlenecks efficiently. Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications, PDSIA, 1999, p. 1-24.

[13] **Suleman M. A., et al.** Accelerating critical section execution with asymmetric multi-core architectures. SIGARCH Computer Architecture News, ACM, Vol. 37, Issue 1, 2009, p. 253-264.

[14] **Metreveli Z., Zeldovich N., Kaashoek M. F.** Cphash: a cache-partitioned hash table. ACM SIGPLAN Notices, ACM, Vol. 47, Issue 8, 2012, p. 319-320.

[15] **Calciu I., Gottschlich J. E., Herlihy M.** Using elimination and delegation to implement a scalable NUMA-friendly stack. Proceedings of Usenix Workshop on Hot Topics in Parallelism (HotPar), 2013, p. 17.

[16] **Lozi J. P., et al.** Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. USENIX Annual Technical Conference, 2012, p. 6576.

[17] **Lozi J. P., Thomas G., Lawall J. L., Muller G.** Efficient Locking for Multicore Architecture. Research Report RR-7779, INRIA, 2011, pp. 1-30.