Optimization Strategies for Inter-Thread Synchronization Overhead on NUMA Machine

Song Wu, Jun Zhang, Yaqiong Peng, Hai Jin, Wenbin Jiang
Services Computing Technology and System Lab
Cluster and Grid Computing Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
wusong@hust.edu.cn

***Abstract*—Overhead caused by data consistence issue in inter-thread synchronization probably degrades the performance of parallel applications. *Non-Uniform Memory Access* (NUMA), as the mainstream architecture in today's multicore processor, further exacerbates this issue due to the significant overhead incurred by *Remote Memory Reference* (RMR). Therefore, to reduce synchronization overhead, it is important to solve the data consistence issue. In this paper, we classify the overhead into two kinds: (1) overhead incurred by algorithms themselves, and (2) overhead incurred by critical sections. To reduce two kinds of overhead on NUMA machine, we present two optimization strategies called *search and backtrace* (SAB) and *reorder critical section and non-critical section* (RCAN), respectively. In SAB, a server thread tries to search a thread coming from master NUMA node, and designates it as the new server thread. In this way, most of the time, shared data resides in the cache of master NUMA node, resulting in lower overhead caused by data consistence issue in critical section. In RCAN, each thread consecutively posts synchronization requests, followed by consecutively executing non-critical section. In this way, server threads could serve enough requests, resulting in better data locality. We design an algorithm named R-Synch based on SAB, while designing an algorithm named H-STA based on RCAN. Our evaluation with representative synchronization algorithms demonstrates the effectiveness of R-Synch and H-STA.**

***Keywords*-NUMA; data consistence; synchronization; algorithm**

I. Introduction

The prevalence of multicore results in a popularity of parallel programming to improve the application performance. Unfortunately, it is not true for applications that frequently access shared data, because shared data must be accessed mutually exclusive by threads under the aegis of synchronization algorithms. Therefore, a highly efficient synchronization algorithm plays an important role in parallel programming, especially for applications that need significant synchronization.

Processor manufacturers quickly shift from simple bus-based designs to NUMA and *Cache Coherent NUMA* (CC-NUMA) architectures due to the growing size of multicore machines [1]. Typically, a NUMA machine contains several nodes connected by an interconnect. Each node consists of several cores, independent cache, and shared local memory. Accessing of data missed in the local cache can incur off-chip memory access [2] and interconnect traffic which are significantly costly on CC-NUMA machines. According to [1, 3], access by a core to its local memory (e.g. its private or shared last level cache) can be much faster than access to the remote memory located on another node. These features complicate the design of highly efficient synchronization algorithms.

Synchronization technique walks a long way from the traditional simple lock algorithms to the state-of-the-art algorithms such as CC-Synch and H-Synch [4]. Queue locks are proposed to reduce the overall cache coherence traffic by forming queues of threads [5–8], and each thread spins on a separate local memory location [1]. However, queue lock may not work well on CC-NUMA machines because threads executing instructions may alternately come from different NUMA nodes, resulting in non-trivial overhead of cache misses and interconnect communications.

Combining technique [4, 9, 10] is a compelling approach to design lock algorithms by preventing the shared resource from bouncing back and forth among multiple cores. However, it still works not so well on NUMA machines, because communication between threads would incur lots of cache misses. Hierarchical locks originally presented by Radovic [11] is a good idea for designing NUMA-aware lock algorithms [1, 4], but it still faces a challenge that how to make threads coming from the same NUMA node consecutively access shared resource as many as possible.

In summary, queue lock eliminates the hot spot [12] problem that causes the overhead inside algorithm itself; combining lock reduces the overhead occurring in critical section by using combining policy in serving synchronization requests; hierarchical lock tries to reduce the overhead occurring both in the critical section and algorithm itself. This paper presents two more efficient policies to reduce the overhead in critical section and algorithm itself, respectively.

We first present a *search and backtrace* (referred to as SAB) policy. In SAB, a server thread tries to search a thread coming from the master NUMA node, and designates it

978-1-4673-8590-9/15/$31.00 ©2015 IEEE

as the new server thread. In this way, most of the time, shared data reside in the cache of master NUMA node, thus reducing the overhead caused by data consistence issue in critical section. Then, we present a policy called *reorder critical section and non-critical section* (referred to as RCAN). In RCAN, a thread consecutively posts synchronization requests, followed by consecutively executing non-critical section. In this way, a server thread could serve enough requests, thus enhancing data locality.

The main contributions of this paper are listed below.

- We present two optimization strategies, namely SAB and RCAN, to reduce the synchronization overhead incurred by algorithms themselves and critical sections, respectively.
- To make the two strategies into practice, we design an algorithm named R-Synch based on SAB, while designing an algorithm named H-STA based on RCAN.
- We conduct comprehensive experiments to demonstrate the effectiveness of R-Synch and H-STA. R-Synch works better when the main overhead occurs in critical section. H-STA works better when the main overhead occurs in algorithm itself.

II. Motivation and Design

In this section, we first introduce synchronization overhead. Then, we analyze the locations of synchronization overhead. Next, we analyze how such overhead is generated. Finally, we design two optimization strategies to reduce the overhead.

A. Overview of Synchronization Overhead

As shown in Figure 1, there are three situations, where a thread executes a task containing subtasks: Task1, Task2, Task3. In the first situation, a single thread takes $3*T$, $2*T$, $3*T$ to complete Task1, Task2, and Task3, respectively. In the second situation, if there are three threads and the task can be totally parallelized, then the time taken to complete all three subtasks is $2*T$. The third situation is shown in Figure 1(c), which involves the synchronization overhead. In this situation, multiple threads access shared data. To ensure data security, threads must access shared data one by one. As a result, the time taken to complete three subtasks increases to $3*T*a$, $2*T*b$, $3*T*c$, respectively ($a>1$, $b>1$, $c>1$). Therefore, when the synchronization overhead is involved, the efficiency of parallel execution is probably even worse than the serial execution.

B. Locations of Synchronization Overhead

Figure 2 describes the execution overview of a multithreaded application. This multithreaded application contains three kinds of executions as mentioned above. The left and the right of Figure 2 describe the serial execution part. The middle mixes the parallel and synchronization execution, which are the main topic. In the middle of Figure

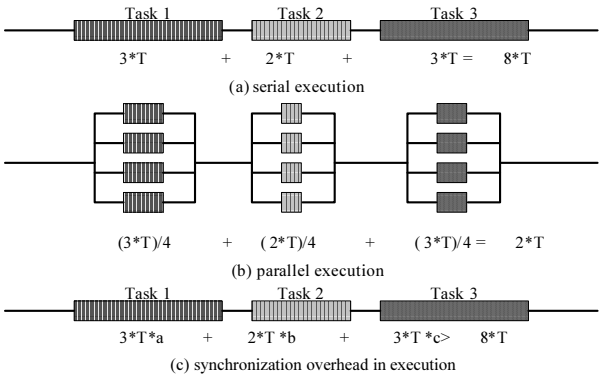


Figure 1. Threads execute a task in three kinds of situations

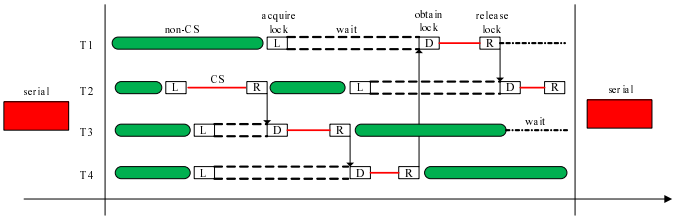


Figure 2. Execution overview of a multithreaded application

2, parallel parts are marked with *non-CS*, and critical section parts are marked with *CS*.

In parallel parts, threads execute simultaneously without interacting with each other. In critical section, threads must first try to acquire the lock. If a thread successfully acquires and obtains the lock, it enters into the critical section and operates the shared data. The thread will release the lock after leaving the critical section. Actually, it is hard for a thread to acquire and obtain the lock because other threads also compete for the lock. So a thread must pay some efforts between acquiring and obtaining the lock. This effort means the overhead caused by the process of acquiring and obtaining the lock. We call this overhead as synchronization overhead inside the algorithm because the size of the overhead depends on how a programmer designs the algorithm.

Once entering into a critical section, threads access or operate shared data, causing some overhead due to the data consistence issue. This overhead affects the time for a thread to complete the critical section work. If the time gets longer, it will increase the difficulty for other threads to enter into the critical section [13]. We call this overhead as synchronization overhead inside critical section.

C. Data Consistency Issue

As analyzed above, there are two locations that cause the synchronization overhead: (1) synchronization overhead inside algorithm itself, and (2) synchronization overhead inside critical section. Two kinds of synchronization overhead are

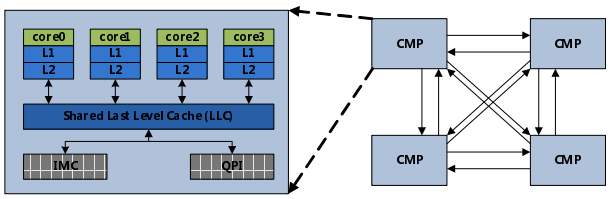


Figure 3. Overview of NUMA architecture.

caused by data consistency issue, especially on NUMA machines. Figure 3 is an overview of NUMA architecture. On the left of the figure, there is a CMP (*Chip Multiprocessors*) chip containing four cores. On the right of the figure, four CMP chips make up the NUMA architecture, where each CMP is a NUMA node and has its private/shared cache. As multiple caches can hold the same memory location, data consistency issue arises.

When data is accessed by a thread, it will enter into the corresponding cache line. Therefore, the shared data accessed by multiple threads will enter into several cache lines and these cache lines will spring over every NUMA node. Some cache lines may hold the latest valid data, while some cache lines may hold invalid data. At this time, data consistency issue arises. Cache coherence protocol is used to maintain data consistency due to that multiple cache lines hold the same memory locations. When the data accessed by a thread is not hit in the cache, it will cause an off-chip memory access (or RMR as mentioned before). The off-chip memory access indicates an access to the memory or a cache line on other NUMA nodes. An off-chip memory access is several times slower than the access to the local cache [3]. Two kinds of synchronization overhead are mainly caused by such off-chip memory accesses. To decrease the synchronization overhead, it is necessary to decrease two kinds of off-chip memory accesses.

To investigate two kinds of overhead, we conduct four experiments on state-of-the-art lock algorithms, namely CC-Synch and H-Synch [4]. The results are shown in Figure 4.

The first experiment simulates a Fetch&Multiply object coming from [4], and the experimental results are coincident with [4]. Clearly, H-Synch outperforms CC-Synch because H-Synch uses a hierarchical NUMA-aware policy to reduce off-chip memory accesses. In this experiment, two kinds of overhead are horse and horse.

To investigate the overhead caused by critical section, we conduct two other tests and the experimental scheme comes from [11, 14], where threads in critical section modify each element of a shared vector. As shown in Figure 4(b), the results are still coincident with the conclusion of the original paper. When we increase the size of the shared vector, as shown in Figure 4(c), the performance of CC-Synch and H-Synch becomes approximately the same. This is because the overhead caused by critical section outweigh the benefits

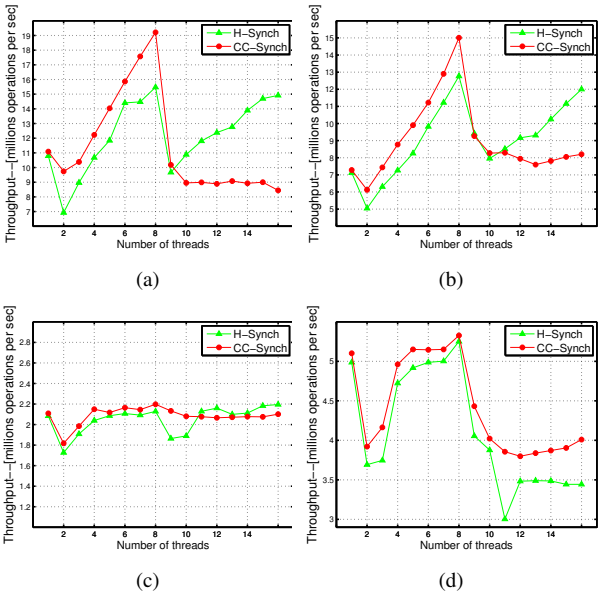


Figure 4. Average throughput of H-Synch and CC-Synch while running four motivating experiments. X-axis represents the number of threads, Y-axis represents throughput in millions operations per second.

brought by H-Synch.

In the last experiment, we make a little changes that a thread in critical section modifies a fixed number of elements that scatter across the shared vector. As shown in Figure 4(d), CC-Synch outperforms H-Synch in this experiment, which is out of our expectation.

Summary. In the first two experiments, as the overhead occurring inside algorithm itself is the main overhead, the hierarchical policy used by H-Synch brings performance promotion. In the last two experiments, we change the size or the access pattern of the shared data, making the overhead occurring in critical section become the main overhead. At this time, the hierarchical policy does not help H-Synch much. Therefore, we have two points to improve the performance of synchronization algorithms. One is to reduce off-chip memory accesses inside algorithm itself, while another is to reduce off-chip memory accesses inside the critical section.

D. Optimization Strategies

To reduce two kinds of synchronization overhead, we present two optimization strategies: SAB and RCAN.

1) Reducing off-chip memroy accesses occurring in critical section: SAB is proposed to reduce overhead occurring in critical section. In critical section, multiple threads access the same shared data, and thus several cache lines may hold the data. If only one cache of a NUMA node holds the shared data, the off-chip memory accesses will be significantly reduced. If threads access shared data coming from the same NUMA node (we refer to this NUMA node as master NUMA node), then the shared data would only enter into

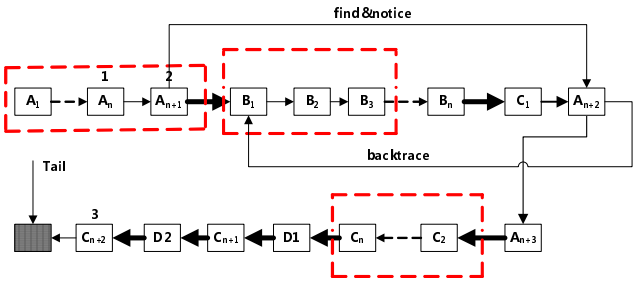


Figure 5. Overview of SAB.

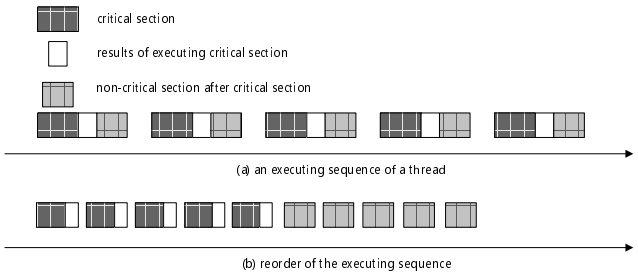


Figure 6. Overview of RCAN.

the cache of the same NUMA node. As a result, the data consistency issue is basically solved.

We use Figure 5 to describe SAB policy. In the figure, *A*, *B*, *C*, and *D* represents four different NUMA nodes respectively, and the corresponding subscript number represents different threads of the NUMA node. SAB is based on a technique called combining [4]. In combining, the thread obtaining the lock is called as combiner. When a combiner completes its request, it will continue to serve the requests of other threads, then the combiner releases the lock and chooses a new combiner.

Now we detail the working mechanism of SAB. Threads that want to execute critical section insert its request node to the linked list, and then spin on the locked field waiting to be served by the combiner or be designated as the new combiner. To be simplicity, we assume A is the master NUMA node and the thread at the head of the list is the combiner. When the combiner has served some number of requests and stops, it will face three situations that are marked with a number on top of the node. If the combiner stops at A_n, as the next thread A_{n+1} comes from the same NUMA node, A_{n+1} will be designated as the new combiner. If the combiner stops at C_{n+2}, as there are no active threads in the list, the combiner writes something to the node indicating that there are no active requests at present.

The last and most complicate situation occurs when the combiner stops at A_{n+1}, as the next thread B_1 comes from a different NUMA node, the combiner travels the linked list and tries to find a thread of the same NUMA node. If one such thread is found, then the old combiner designates it as the new combiner and tells it the backtrace position from which it should start to serve the requests in the next execution round. If not finding one, the combiner either simply designates the thread next to it as the new combiner or does the same as in the second situation.

In this way, the shared data almost resides in the cache of master NUMA node, and threads accessing the shared data also come from master NUMA node. When threads access the shared data, the shared data are already in the local cache in most cases, resulting in less off-chip memory accesses occurring in critical section.

2) Reducing off-chip memory accesses occur in algorithm itself: Figure 6(a) shows an execution sequence of a thread. For simplicity, we assume that there is only one critical section in the code executed by threads. Therefore, we can use the critical section as a boundary to divide the code into three sections. The first: *Critical Section* (CS for short), the second: *Result of Critical Section* (ROCS for short) and the last: *The Parallel Part* (TPP for short).

We propose a method called RCAN that can accelerate the velocity of posting requests for a single thread. According to combinatorics, there are totally three kinds of relationships between CS and TPP (here we regard CS and ROCS as a whole). Namely CS vs. CS, TPP vs. TPP, CS vs. TPP. We first analyze the three relationships under the condition of multiple threads by observing the execution behavior of multiple threads, then we try to apply the feature to a single thread. First, synchronization requests posted by threads can be served by combiner in sequence generally or not, and we maintain the default sequence semantics for a single thread. Second, the parallel parts do not access shared resources, so they can be executed simultaneously, and thus the execution order of TPPs does not matter for a single thread. Finally, as to CS and TPP, the parallel parts must be executed according to the results of critical section, that is to say, the parallel parts must be executed after the corresponding critical section.

According to the analysis, we can reorder the execution sequence in Figure 6(a) and get a new execution sequence as shown in Figure 6(b). By doing this, we can accelerate the velocity of posting requests for a single thread. Hence the combiner can consecutively serve enough requests resulting in enhanced data locality. This is because inter-thread communications caused by synchronization mostly occur in the same NUMA node, so as to reduce the off-chip memory accesses occurring in algorithm itself.

III. Applying SAB and RCAN to Synchronization Algorithms

In this section, we show how to employ SAB and RCAN to design synchronization algorithms. Based on SAB policy we design a synchronization algorithm called R-Synch. Based on RCAN policy, we design a synchronization

algorithm called H-STA. When overhead in critical section dominates, R-Synch works better. When overhead in algorithm itself dominates, H-STA works better.

A. R-Synch

Before describing R-Synch algorithm, we first introduce some data structures. Each thread has a request node consisting of several fields: (1) *arg* is used to store arguments and results; (2) *locked* indicates whether the lock is held; (3) *completed* indicates whether the request has been served; (4) *node* represents the serial number of the corresponding NUMA node; (5) *btr* indicates the backtrace position; (6) *next* points to the next node. For simplicity, some details are omitted from Algorithm 1.

Algorithm 1 Pseudocode for R-Synch

```
1:  struct request node{arg, pid, locked, completed, node, btr, next}
2:  function APPLYOP(request req, ...)
3:      my_new_node = SWAP(tail, my_current_node);
4:      my_new_node → next = my_current_node;
5:      some other work like parameter setting;
6:      while my_new_node → locked do/*busy waiting*/
7:          ;
8:      end while
9:      if my_new_node → completed then
10:         return my_new_node → arg;
11:     end if
12:     if master = −1 then
13:         master = my_new_node → node;
14:     end if
15:     p = my_new_node;
16:     if p → btr ≠ null then
17:         tmp = p;
18:         p = p → btr;
19:         tmp → btr = null;
20:     end if
21:     while p → next ≠ null and help_num ≤ max do
22:         temp_next_node = p → next;
23:         serve the request which is stored in node p;
24:         p → completed = true;
25:         p → locked = false;
26:         p = temp_next_node;
27:     end while
28:     if p → node = null or p → node = master then
29:         p → locked = false;
30:     else
31:         p1 = p;
32:         while p → next ≠ null and p → node ≠ master do
33:             p = p → next;
34:         end while
35:         if p → node = master then
36:             p → btr = p1;
37:             p → locked = false;
38:         else
39:             p1 → locked = false;
40:         end if
41:     end if
42:     return my_new_node → arg;
43: end function
```

When a thread has a synchronization request, it inserts its request node to the tail of the linked list by using an atomic SWAP operation (lines 3-4). Then, it spins on locked field of my_new_node until the lock is released by the combiner. Once released, the thread decides what to do next by checking the value of *completed* field. If the filed is true, it means that its request has been served by the combiner. Then, the thread will return; If the field is false, the current thread becomes the new combiner.

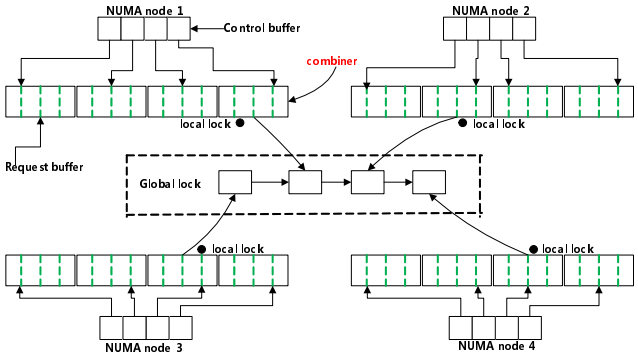


Figure 7. Overview of H-STA.

Once becoming combiner, the thread will set the value of $master$ according to its initial value. Then, the new combiner starts to work by first checking the *btr* field. If *btr* is not NULL, then it sets the backtrace position to the value of *btr*. The combiner travels the linked list of requests, and serves its own request and a predefined number of requests of other threads (lines 21-27). When it completes a request, it sets the *locked* field and *completed* field of the corresponding thread to false and true, respectively. Once the combiner completes its work, it chooses the next new combiner according to the SAB policy (lines 28-41). If the next thread comes from the same NUMA node, or there is no active thread, it simply writes a false value to *locked* field of the next thread node or dummy node. Otherwise, it will travel the linked list to find a thread of the same node to be designated as the new combiner and tell it the backtrace position. If not finding an appropriate thread, it can only designate the thread next to it in the list as the new combiner.

B. H-STA

H-STA is a hierarchical version of RCAN as shown in Figure 7. We assume there are totally four NUMA nodes. For each NUMA node, there is a request buffer and control buffer. A request buffer contains several request nodes. Each request node consists of several slots which implements STA. Each slot is defined as a struct with three fields: (1) *arg* is used to store arguments of critical section or the results of it; (2) *pid* is used to distinguish threads in a NUMA node; (3) *completed* is used to identify whether there is an active request waiting to be served.

Each control buffer contains several nodes of a size equal to the number of cores in that NUMA node, and each node is also defined as a struct with several fields:

(1) *_up* and *_low* presents the upper and lower bound of the corresponding request node in the request buffer, respectively; (2) *_combiner_index* is used by combiner; (3) *_thread_index* is used by common threads.

The lock policy is a hierarchical version. As shown in Figure 7, there are multiple local locks, and each lock is for a NUMA node. Besides, the policy contains a global CLH lock. In each NUMA node, threads compete for the local lock, the winner becomes combiner of that node. Then, all the local combiners compete for the global lock. Only the winner owns the right to access shared resource.

Thread posts synchronization requests by writing essential information in the corresponding slots. When a thread has an request, it first judges whether the *_thread_ index* field reaches the end of the corresponding request node. If not, it posts this request in the slot of the request node and increases the value of *_thread_index* by one. Otherwise, the thread will compete for the local lock. If failing in acquiring the lock, the thread will wait until its requests are served or the lock is released. In the former situation, the thread will execute the corresponding parallel parts and then returns. In the latter situation, it will try to acquire the lock. If succeeding in acquiring the lock, the thread becomes the combiner of this NUMA node. Then, it will try to acquire the global lock repeatedly until it succeeds. Then, it will travel each slot of the control buffer. According to *_up* and *_low* fields, the combiner will find the corresponding request node and check each slot of it. If there are active requests that have not been served, combiner serves them. When the combiner completes its work, it will execute its parallel parts according to the results stored in *arg* fields.

Of course, not all programs can be divided according to RCAN policy, such as nested critical section. In future work, we will extend the applying range of RCAN policy.

IV. Evaluation

In this section, we evaluate R-Synch and H-STA by comparing them with other state-of-the-art synchronization algorithms. We begin with an introduction of the hardware and software platform, followed by a description of the experiment methodology. Then, we test the algorithms with microbenchmarks that are widely used in the literature. Finally, we further investigate the performance of R-Synch and H-STA on more complex concurrent objects, namely shared stack.

A. Platform

We evaluate R-Synch and H-STA on a CC-NUMA machine consisting of two Intel Xeon E5-2670 processors. Each processor contains eight cores. Each core has a 32KB L1 private data/instruction cache, and a 256KB L2 private data cache. All cores within a processor share a fast 20MB L3 data cache. To avoid the bottlenecks in memory allocation, the Hoard memory allocator [15] is used.

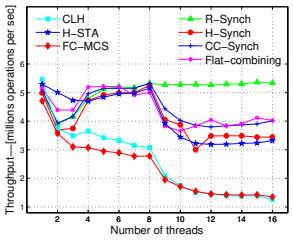


Figure 8. Average throughput of each implementation when running R-vector microbenchmark.

B. Experimental Methodology

To evaluate the performance of R-Synch and H-STA, we compare them with several state-of-the-art synchronization algorithms, including H-Synch [4], CC-Synch [4], Flat-combining [10], FC-MCS [1], and CLH [7, 8]. In all experiments, each algorithm executes totally 10^7 times operations for different values of n, and n is the number of current active threads. Besides, we assume that the size of data accessed to serve the requests is smaller than the size of cache, and test the cache misses for each experiment.

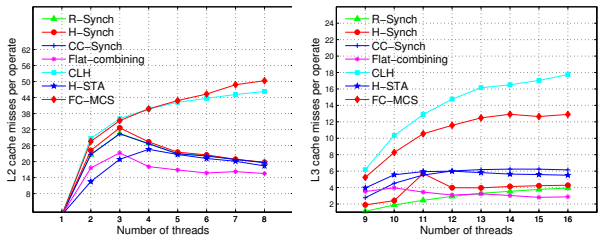
C. Microbenchmarks

We test all the algorithms by using two microbenchmarks that are widely used in the literature. The first is a modified microbenchmark from [11]. As described in Section II, we make a little changes to it. For simplicity, we call it as R-vector. The second is a simulated shared Fetch&Multiply object used in [4].

1) R-vector: The throughput of each algorithm for R-vector is shown in Figure 8. When the number of thread is less than eight, the four combining based algorithms achieve approximately the same performance. When threads are across multiple NUMA nodes, R-Synch outperforms all other algorithms. To be specific, R-Synch achieves up to 1.38 times higher throughput than CC-Synch. The performance of Flat-combining is close to CC-Synch, and they are a little slower than H-Synch and H-STA. Also, R-Synch significantly outperforms CLH and FC-MCS by a factor up to 4.21.

Although FC-MCS is NUMA-aware, it performs worse on machine with small clusters of cores. Experiments in [4] proves that FC-MCS performs well on machine with large clusters of cores. When threads reside in one NUMA node, there is no interconnect communication and RMRs. Therefore, all combining based algorithms achieve approximately the same performance. For CLH and FC-MCS, combining is not used to serve requests. Therefore, every request may be applied by a different thread, resulting in a higher L1/L2 cache miss than other four algorithms (as shown in Figure 9).

When threads are spread across multiple NUMA nodes, circumstance becomes complex, due to the issues of off-



(a) (b)

Figure 9. L2 cache misses per operation and L3 cache misses per operation.

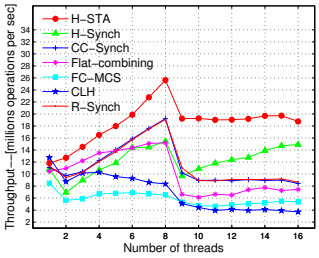


Figure 10. Average throughput of each implementation while simulating a Fetch&Multiple object.

chip memory accesses and interconnect communication. The better performance of R-Synch can be explained by the lower L3 cache miss as shown in Figure 9(b). However, when the number of threads is beyond 13, R-Synch does not have the lowest L3 cache misses. In section II, we classify the locations that may occur overhead into two kinds. The first is inside the algorithm itself, which is caused due to communication between combiner and other threads here. The second is inside critical section. In this experiment, the latter is more costly. According to the SAB policy used in R-Synch, we know that overhead incurred in critical section could be effectively reduced. Therefore, R-Synch performs better when multiple NUMA nodes are involved.

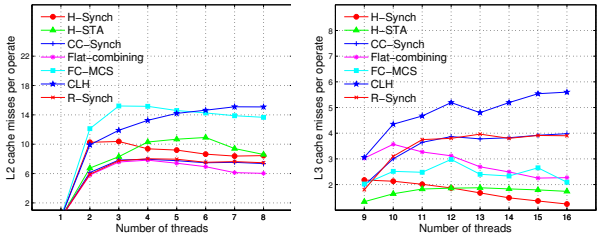
2) Fetch&Multiply: Figure 10 depicts the performance of each algorithm when simulating a Fetch&Multiply object. H-STA scales significantly better than all other algorithms. When multiple NUMA nodes are involved, we will mainly concentrate on the performance differences among all the algorithms. As a whole, H-STA outperforms H-Synch by a factor up to 1.98. The performance of CC-Synch and Flat-combining are close, and they are all overtaken by H-STA by a factor up to 2.15. Again, CLH and FC-MCS are the slowest algorithms.

H-STA and H-Synch are the two fastest algorithms, because they have lower cache misses (as shown in Figure 11). When the number of threads is beyond 13, L3 cache misses of H-STA is a little higher than H-Synch. However, H-STA still performs better than H-Synch. This is because H-STA has a much lower L2 cache misses (as shown in Table I).

Table I
L2 CACHE MISSES PER OPERATE. THE FIRST ROW REPRESENTS THE NUMBER OF THREADS.

	9	10	11	12	13	14	15	16
FC-MCS	13.5	13.1	13.2	15.2	13.4	13.4	14.9	13.9
H-Synch	8.7	9.6	9.4	9.3	9.4	9.1	8.9	8.8
H-STA	6	6.1	6.4	6.6	6.5	6.8	6.8	6.8

Although FC-MCS has a litter lower L3 cache misses than CC-Synch, it is still outperformed by CC-Synch in terms of throughput. This is because FC-MCS has a much higher L2 cache misses (as shown in Table I).



(a) (b)

Figure 11. L2 cache misses per operation and L3 cache misses per operation.

D. Shared stack

Stack is a data structure with a wide range of use. For example, inter-thread communication is heavily based on accessing such data structure [4]. Therefore, we further investigate the performance of each algorithm by applying them to shared stacks.

As shown in Figure 12, HSTA-Stack achieves the best performance, followed by H-Stack (H-Synch). Throughput of FC-Stack (Flat-combining), CC-Stack (CC-Synch), and R-Stack (R-Synch) are approximately the same. Again, CLH-Stack and FCMCS-Stack are the slowest implementation of shared stack. Both HSTA-Stack and H-Stack employ a hierarchical policy to reduce RMRs and interconnect communication, resulting in enhanced data locality, which is proven by L3 cache miss curve in Figure 13(b).

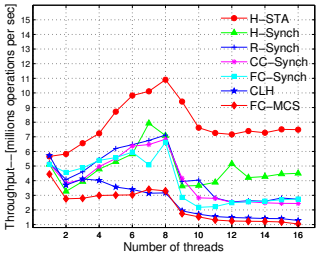
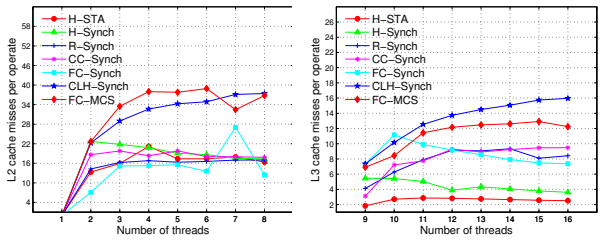


Figure 12. Average throughput of each implementation when apply them on shared stack.



(a) (b)

Figure 13. L2 cache misses per operation and L3 cache misses per operation.

V. Relate Work

The combining technique has been studied for decades. To the best of our knowledge, the earliest combining technique is proposed by [12] to construct a software combining tree for decreasing memory contention. Another combining based synchronization algorithm in [9] is presented later. However, it suffers lots of contention in posting requests and may cause unbounded RMRs. A hardware technique called ACS is proposed in [13], which uses an asymmetric faster core to execute critical sections. Sim [16] and Flat-combining [10] are two highly efficient implementations of combining technique, which are proven to significantly outperform fine-grained thread synchronization.

Hierarchical technique is presented to mainly deal with issues caused by NUMA architectures, such as RMRs and interconnect communications among multiple NUMA nodes. To the best of our knowledge, HBO [11] is the first hierarchical technique that encourages threads from the same NUMA node to acquire the lock consecutively for reducing interconnect communication and achieving strong data locality. However, HBO is a test-and-test-and-set lock assisted with a backoff scheme, which are known to incur lots of invalidation traffic. FC-MCS [1] is another highly efficient hierarchical locks that outperforms all previous NUMA-aware or none NUMA-aware locks. Nevertheless, FC-MCS performs poorly on machines with small clusters of cores due to the difficulty in building long local list of requests. H-Synch [4] is the fastest lock algorithm that employs both combining and hierarchical technique. Unlike FC-MCS, H-Synch works well on machines with small clusters of cores.

VI. Conclusion

Synchronization overhead limits the performance of parallel applications. This paper analyzes the locations that incur the overhead and presents two strategies to reduce the overhead: SAB and RCAN. SAB tries to reduce the overhead occurring in critical section, while RCAN tries to reduce the overhead occurring in algorithm itself. Finally, we show how to use the two strategies to design synchronization algorithms. We use SAB to design an algorithm called R-Synch, and use RCAN to design an algorithm called H-STA. Experiments show our strategies effectively reduce the overhead in critical section and algorithm itself, respectively.

Acknowledgements

This work was supported by National Science Foundation of China under grant No.61232008, National 863 Hi-Tech Research and Development Program under grant No. 2014AA01A302 and No.2015AA01A203, the Fundamental Research Funds for the Central Universities under grant No.2015TS067.

References

[1] D. Dice, V. J. Marathe, and N. Shavit, "Flat-combining numa locks," in *Proc. SPAA'11*, 2011, pp. 65–74.

[2] Y. Zhang, M. Kandemir, and T. Yemliha, "Studying inter-core data reuse in multicores," in *Proc. SIGMETRICS'11*, 2011, pp. 25–36.

[3] D. Hackenberg, D. Molka, and W. E. Nagel, "Comparing cache architectures and coherency protocols on x86-64 multicore smp systems," in *Proc. MICRO'09*, 2009, pp. 413–422.

[4] P. Fatourou and N. D. Kallimanis, "Revisiting the combining synchronization technique," in *Proc. PPoPP'12*, 2012, pp. 257–266.

[5] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 6–16, 1990.

[6] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 9, no. 1, pp. 21–65, 1991.

[7] T. S. Craig, "Building fifo and priority-queueing spin locks fromatomic swap," Department of Computer Science, University of Washington, Tech. Rep., 1993.

[8] P. S. Magnusson, A. Landin, and E. Hagersten, "Queue locks on cache coherent multiprocessors," in *Proc. IPPS'94*, 1994, pp. 165–171.

[9] Y. Oyama, K. Taura, and A. Yonezawa, "Executing parallel programs with synchronization bottlenecks efficiently," in *Proc. PDSIA'99*, 1999, pp. 182–204.

[10] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff," in *Proc. SPAA'10*, 2010, pp. 355–364.

[11] Z. Radovic and E. Hagersten, "Hierarchical backoff locks for nonuniform communication architectures," in *Proc. HPCA'03*, 2003, pp. 241–252.

[12] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie, "Distributing hot-spot addressing in large-scale multiprocessors," *IEEE Transactions on Computers*, vol. 36, no. 4, pp. 388–395, 1987.

[13] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," in *Proc. ASPLOS'09*, 2009, pp. 253–264.

[14] V. Luchangco, D. Nussbaum, and N. Shavit, "A hierarchical clh queue lock," in *Proc. Euro-Par'06*, 2006, pp. 801–810.

[15] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," in *Proc. ASPLOS'00*, 2000, pp. 117–128.

[16] P. Fatourou and N. D. Kallimanis, "A highly-efficient wait-free universal construction," in *Proc. SPAA'11*, 2011, pp. 325–334.