Distributed Computing Design Methods for Multicore Application Programming

Qian Yu, Tong Li, Zhong Wen Xie, Na Zhao, Ying Lin
School of Software, Yunnan University, Kun Ming, 650091, China
Key Laboratory in Software Engineering of Yunnan Province
yuqian2001cn@163.com

Abstract—**In order to solve the serial execution caused by multithreaded concurrent access to shared data and realize the dynamic load balance of tasks on shared memory symmetric multi-processor (multi-core) computing platform, new design methods are presented. By presenting multicore distributed locks, multicore shared data localization, multicore distributed queue, the new design methods can greatly decrease the number of accessing the shared data and realize the dynamic load balance of tasks. For illustration, design scheme of multicore task manager of server software are given by using new design methods. Results shows the new design methods reduce the number of access shared resources, partially resolve the serial execution of cooperative threads and realize the dynamic task balance of server software, which validate the superiority of this approach.**

Keywords- multithread concurrency; multicore distributed lock; multicore shared data localization;multicore distributed queue.

I. INTRODUCTION

Shared memory symmetric multiprocessor is the mainstream platform of multicore computing. The architectures mainly focus on exploiting thread-level parallelism rather than instruction-level parallelism. Traditional multithread concurrent access mechanisms have lock-based access models [1]-[4] and lock-free access model [5]-[8]. Exclusive locks [1] only allows one thread to enter mutual section at the same time, other shared threads are be blocked and queued to access the resource. Conditional lock [2] allows a thread to wait or wake other threads when a particular condition is satisfied, which doesn't provide no-starvation implementation on common data structures. Conditional lock reduces the number lock is used, much better performance than the mutually exclusive lock. Fich [3] puts forward ways to mutually exclusive access to the share register. Hierarchical lock [4] is to enable more users to concurrently access to database resources. The shared resource protected by coarse lock contains the resource protected by fine-grained lock. Accessing shared resource at lower lever causes the data contend.

Lock-free concurrent access implements by atomic operations CAS (compare and swap). Lock-free concurrent access better performance than locking concurrency, but Lock-free-based programming is difficult to grasp. The method is very complex and causes deadlock.

The efficiency by solving a recursive equation that depends on the distribution of task service times and the expected number of tasks needed to be synchronized, efficiency decreases with an increase in the number of processors [10]. Eliminating barrier synchronization [11] for compiler-parallelized codes on software distributed shared memory. In parallel computing, task time that it takes for a processor to complete a task using local synchronization [12] approaches an exact limit as the number of processors in the cycle approaches infinity. Under global synchronization, however, the time is unbounded, increasing logarithmically with the number of processors. Local barriers and predictive barriers scheduling for reducing synchronization overhead are presented in the simulation of message-passing multi-computers [13].

However, multicore environment is different from multicomputer in which we mainly focus on communication and data migration of global data.

In multi-core environment if thread concurrent access to shared resources using coarse-grained locks, threads must be queued waiting for a shared resource and the serial execution. No more thread executing in system leads to the CPU core be idle and in a State of hunger.

The paper presents multicore distributed lock, the design method of multicore distributed data structure and multicore shared data localization in order to reduce the times of lock operation and decrease synchronization queuing times of waiting for shared resource.

The paper is structured as follows. The second section presents multicore distributed lock to avoid blocked multithread queuing caused by using centralized lock. The third section introduces the method of multicore shared data localization. The fourth section introduces t design scheme of multicore task manager of server software. The fifth section analyses the running result of task manager.

II. MULTICORE DISTRIBUTED LOCK

In order to analyze the performance of multi-threaded concurrent access mechanism, two concepts of thread granularity[9] (expressed by $T_{granularity}$) and lock granularity (expressed by $Key_{granularity}$) are defined. Thread granularity is the ratio of effective computing time in thread and thread computing time, that is, $T_{granularity} = (t_{inside}+t_{parallel}) / t_{lockunlock}$, t_{inside} expresses running time of operations inside lock, $t_{lockunlock}$ expresses running time of lock operations, $t_{parallel}$ expresses parallel computing time of operation outside lock. Thread granularity is proportional to the effective computing time inside thread and inversely proportional to lock operation time inside thread. The larger the effective

© the authors

computing within thread, the greater the thread granularity, so applications that execute in parallel, less break down the number of threads. Lock granularity [5] is used to measure the size of a shared resource. It is associated with a shared resource data and the operation of shared data. There are coarse grained lock and fine-grained lock.

This paper defines lock granularity ($Key_{granularity}$), which is the ratio of t_{inside} and $t_{lockunlock}$, that is, $Key_{granularity} = t_{inside}/t_{lockunlock}$. Operation time is small in coarse-grained lock, but lock contend time in order to waiting for shared resource becomes longer. The smaller fine-grained lock protects shared-resource, the shorter lock contend time. Fine-grained lock can decrease the time queued shared resources.

As $t_{lockunlock}$ is a constant in multicore environment, and therefore make the lock granularity smaller lock, can only reduce the amount of internal calculations. In fine-grained-lock-based access model, the lock internal computing only relates to the shared resources, the non-shared resource computing must be moved to the lock external computing; if the lock is the large calculation of the shared resource, it can be decomposed into smaller shared resource calculation to set fine-grained-lock, but also can reduce the probability of each shared resource synchronization queued by thread.

Multi-core distributed lock synchronous mechanism is to break large sharing data down into small blocks, and to set fine-grained lock on each small piece of shared data, avoiding creating a multithreaded queue caused by multiple threads to synchronize access to a lock or use atomic operations to access the same variable. As shown in Figure 1, No. 1 thread, No. 2 thread, No. 3 thread, No. 4 thread share large data block by synchronizing access.

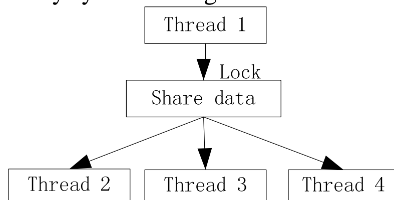

Figure 1. shared resource using Centralized lock

To makes thread use more fine-grained locks, shared data operations need to be split into more small operation blocks for which set a lock. Thread of shared same block data are divided into a group, in which threads contend for same lock. Any two threads which are not in same group do not occur lock contend. It is multicore distributed lock policy.

As shown in Figure 2, six threads to synchronize access to shared data are broken down into three chunks, and six threads are divided into three groups. No.1 thread and No.2 thread, No.3 thread and No.4 thread, No.5 thread and No.6 thread respectively synchronize access to data block 1, data block 2 and data block 3.

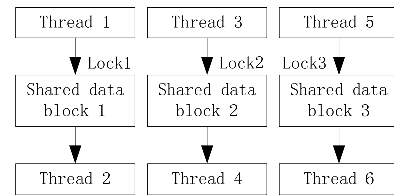

Figure 2. thread contend in group by multicore distributed lock

However, under normal circumstances, only part of the threads can grouped to synchronize access to same shared data, others which cannot be in group because they need to synchronize access to different shared data. As shown in Figure 3, No.1 thread need access to all shared data, however, any two threads of No.2, No.3, No.4 threads are independent. Each of three threads must synchronize access to shared data block respectively with No.1 thread.

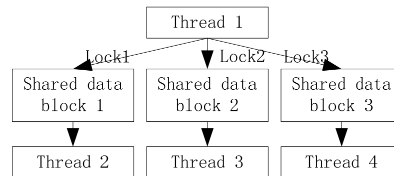

Figure 3. partial threads contend in group by multicore distributed lock

Sometimes while shared data can be divided into multiple small data block, multiple threads must access all of small data block, which threads cannot group. This paper take measure to random access any data block by using distributed lock, so there does not exists synchronization contend as long as each moments the thread access different of data block. As shown in Figure 4, thread No.1or 2or 3 to randomly access the shared data blocks NO.1or 2or 3. The performance of this distributed lock is less than grouping threads synchronize access to shared resources, but better than the performance of centralized synchronization for accessing data.

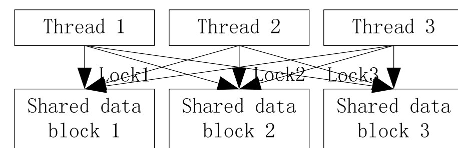

Figure 4. random competition share resources using multicore distributed lock

III. THREADS SHARED DATA LOCALIZATION

The cache locality of sequential programs has been improved relying upon hints provided at the time of thread creation to determine a thread execution order likely to reduce cache misses [14].

Breaking the data into the thread-private data, there does not exist data competition between threads, so thread calculation does not require locks and do not need to use atomic operations when the thread data is localized. No synchronization of threads running fast, the shared data for

© the authors

those existing data competition issues in multi-core processor systems, should try to break it down into the thread-private data, which can reduce the calculation access to shared resources. This is multicore distributed lock competition mode with local computing.

As shown in Figure 5, group synchronization threads share data, and each thread has a private data for local computing, which can reduce the number of access thread locks so as to improve the parallel degree of threads in multicore environment.

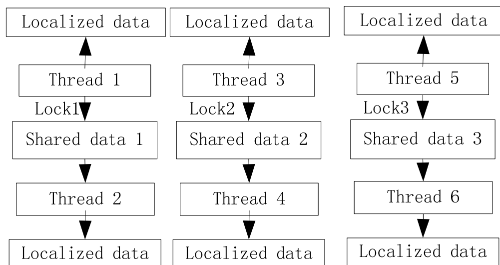

Figure 5. threads contend in group with localized data

Shared bulk data can be extracted as local data of thread, which design method can reduce the access number of thread synchronization data and increase the parallelism of threads in multi-core environment.

IV. DESIGN SCHEME OF MULTICORE TASK MANAGER OF SERVER SOFTWARE

Task manager of server software is a producer-consumer model implemented by programming interface provided by the operating system.

As shown in Figure 6, Traditional design mode is that producers in the server accept the clients' incoming requests and assign a thread to handle them, while a consumer get and handle a task from shared queue. When producer put the task into shared queue, consumers don't get the task from it and must wait for producer release the access right of the shared queue. Any two of consumers must exclusively access the shared queue, which lead to some CPU core idle.

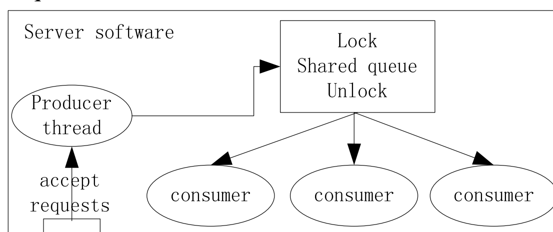

Figure 6. traditional design mode of server software

In order to resolve the above problem, we present a new design scheme for multicore application which has three strategies. The first strategy is to let threads contend in group. There are two groups of producer threads and consumer threads in this example. The second strategy is to change the single shared queue into distributed shared queue which is two-level or multilevel queue. Distributed lock is used on distributed shared queue to ensure multi-threads synchronously access shared sub-queue. The third strategy is to add each consumer with a local queue. The data in local queue is private data of each consumer thread which data can be bulk get from distributed shared queue in order to reduce the number of access the shared data.

The new design method has three key components which are multicore distributed shared queue with distributed lock, thread pool and local queue of each consumer respectively.

As shown in Figure 7, distributed queue realizes mutually access the shared sub-queue and dynamic load balance, only when producer and one consumer or any two consumers access the same sub-queue, they must mutually access, which reduce the probability of queue for shared resources. In others circumstances, producer and consumer can run in parallel. Every consumer has a local queue into which consumer get tasks in bulk from shared queue in order to greatly reduce the number of access shared resource.

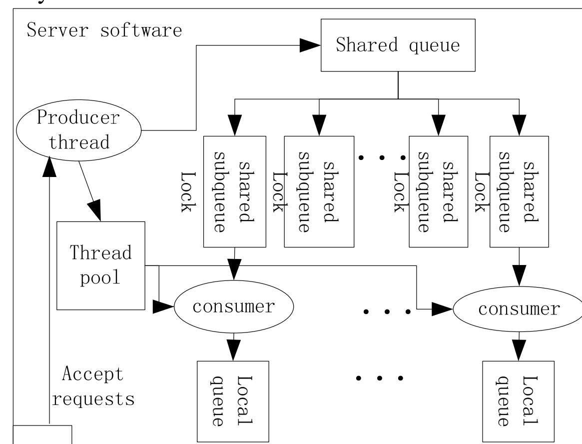

Figure 7. threads contend in group with localized data, distributed queue and distributed lock

V. RUNNING RESULTS ANALYSIS

Implementation of server software has three functions: producer producing tasks and putting them into shared sub-queue, consumer bulk getting tasks into its local queue form shared queue and taking tasks from its local queue to implement them one by one, tracking each thread handling tasks processes as well as load per thread.

The server software has six components which are Thread Pool, Distribute Queue, Local Queue, Share Queue, Task Manager, Scheduler respectively. Thread Pool component can produce the consumer threads. Distribute Queue component sets the number of local queue according to the number of threads and gets bulk tasks from shared sub-queues into the local queue from which thread can get task and implement. Component of Task Manager creates distributed queue, local queue and thread pool preparing for scheduling task. Scheduler tracks all actions of server software from creating producer thread and consumer threads to putting the tasks into shared queue of distributed queue by producer thread, from getting the tasks from shared queue and putting them into local queue to pop the task to handle it.

© the authors

As shown in the figure 8, by tracking the actions of scheduling tasks, all the tasks has been present which each consumer thread has completed.

As shown in the figure 9, by giving the total calculation completed by each thread, dynamic task scheduling balance is realized by those new distributed design methods.

```
thread id 0 total sum task 5
->0->1->4->6->7
thread id 1 total sum task 1
->9
thread id 2 total sum task 3
->2->5->3
thread id 3 total sum task 1
->8
```

Figure 8. result of tracking thread

```
thread id 0 total sum task 268
thread id 1 total sum task 240
thread id 2 total sum task 239
thread id 3 total sum task 253
```

Figure 9. results of load balance scheduling tasks on each thread

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China under Grant No. 60963007, No. 61262024 and No. 61262025; Scientific Research Fund of Yunnan Provincial Department of Education under Grant No.2012C108 and No. 2010Y250; Education Innovation Fund of Software School of Yunnan University under Grant No.2010EI13 and No.2010EI14; Talents training mode of creative experiment project of Ministry of education under Grant No. X3108005; Science Foundation of Yunnan Province under Grant No. 2010CD026, No.2012FB118; Science Foundation of Yunnan Education Department under Grant No. 2012Y257; Science Foundation of Key Laboratory of Software Engineering of Yunnan Province under Grant No. 2011SE09.

REFERENCES

[1] DIJKSTRA, E. W. “Solution of a problem in concurrent programming control,” Communication ACM 8, 9 ,Sept. 1965, 569.

[2] Leslie Lamport, “A fast mutual exclusion algorithm,”[J] ACM Transactions on Computer Systems, Vol. 5 No. 1, 1987, pp. l-11.

[3] Fich, Faith; Hendler, Danny; Shavit, Nir. “On the inherent weakness of conditional synchronization primitives,” Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing, PODC July 25 – 28,2004: St. John's, Newfoundland, Canada. New York, NY: ACM Press. pp. 80–87.

[4] J. N. Gray, R. A. Lorie, and G. R. Putzolu. “Granularity of locks in a shared data base,” In Proceedings of the 1st International Conference on Very Large Data Bases (VLDB '75). ACM, New York, NY, USA, 1975,pp.428-451.

[5] Campbell, M.D. and Holt, R.L. “Lock-Granularity Analysis Tools in WR4-MP,” [J]. Software, IEEE,10(2),1993,pp.66-70.

[6] Danny Hendler and Nir Shavit. “Non-blocking steal-half work queues,” In Proceedings of the twenty-first annual symposium on Principles of distributed computing. ACM, New York, NY, USA, 2002,pp. 280-289.

[7] Kogan, Alex; Petrank, Erez. “Wait-free queues with multiple enqueuers and dequeuers,” Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2011, pp.12-16.

[8] Kogan, Alex; Petrank, Erez. "A methodology for creating fast wait-free data structures," Proceedings of the 17ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New Orleans, LA: ACM Press. 2012, pp. 141-150.

[9] Zhou Wei-ming. Multi-core computing and programming [M]. Wuhan:Huazhong University of Science & Technology Press,2009.

[10] Chang C S, Nelson R. “Bounds on the speedup and efficiency of partial synchronization in parallel processing,” [J]. Systems Journal of the Association for Computing Machinery, January 1995,42(1).

[11] Han H, Tseng C, Keleher P. “Eliminating barrier synchronization for compiler-parallelized codes on software DSMs,” [J]. International Journal of Parallel Programming,26(5), 1998, pp.591-612.

[12] Julia L, Quentiin F. Stout. “A performance analysis of local synchronization,” [C]. In Proceedings of the 18th ACM Symposium on Parallelism in Algorithms and Architectures, July 2006,254-260.

[13] Legedza U, Weihl W. “Reducing synchronization overhead in parallel simulation,” [C]. In Workshop on Parallel and Distributed Simulation, 1996,pp.86-95.

[14] James Philbin, Jan Edler, Otto J. Anshus, Craig C. Douglas, and Kai Li.. “Thread scheduling for cache locality,” SIGOPS Oper. Syst. Rev. 30, 5. 1996, pp.60-71.

© the authors