Volume 3, Issue 3, March 2013 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Multithreaded Java Applications Performance Improvement

Amit Kumar Mangal
M.Tech (CSE) (Research Scholar)
MAHAMAYA TECHNICAL UNIVERSITY

Today every thing is going distributed, for this many server applications, such as Web servers, application servers, database servers, file servers, and mail servers, maintain worker queues and thread pools to handle large numbers of short tasks that arrive from remote sources. In general, a "worker queue" holds all the short tasks that need to be executed, and the threads in the thread pool retrieve the tasks from the worker queue and complete the tasks. Since multiple threads act on the worker queue, adding tasks to and deleting tasks from the worker queue needs to be synchronized, this introduces contention in the worker queue.
General Terms---Work Queue, Reduce Contention, Multithreading, Work stealing, and Thread Pool.

Keywords— Performance Improvement, Reduce Contention, Multithreading, Work stealing, and Thread Pool.

I. INTRODUCTION

Traditional approach of managing concurrent resources was creating individual thread for each new request generated from any server request but this approach create lots of threads which are difficult to manage. Approach with single work queue with thread pool, in this all threads shared a common work queue which store all new tasks generated from requests and all threads checks this common work queue for new task arrivals if work queue has tasks to accomplish then there is no problem but when common work queue have no task to do then all threads wait on common work queue. That generates contention because when new task arrive there is no master to decide which thread to execute the next task. Currently above mentioned two methods are used to manage large scale applications which are working on multithreaded environment. These issues degrade multithreaded java applications performance in manifolds. 1) As no of threads increase their controlling gets maximum available resources that make application response slow. 2) If no task to do then it will take work queue in deadlock which makes application closed. 3) Threads management becomes complex. Due to these problems new approach is required to manage multithreaded java application performance improvement. For performance improvement each thread should acquired its own worker queue for tasks to be managed by the each thread. This will be helpful in resolving contention problem that generally occurred in multithreaded java applications with this individual thread have to manage its own worker queue tasks so it will only be waiting for its own task queue for new task that will never in contention because no other task is trying to access this queue.

II. RELATED WORKS

Currently in java multithreaded applications for managing task common work queue model is used which is shared by all the threads active at a time in the system. In this approach all tasks from remote sources are put in common work queue and whenever task is to be executed it is retrieved from the common work queue. This is explained in Fig1.

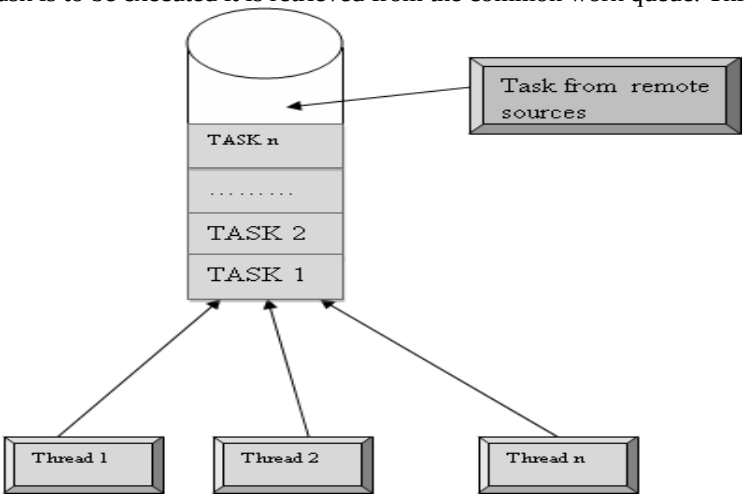

Fig 1:- ***Common Worker Queue.***

III. OWN WORK

Today, with the advent of faster multi processors on multicores, it becomes a challenge for software applications to utilize the underlying cores effectively. (For example, IBM's Power7, Oracle's UltraSPARC, and Intel's Nehalem are multicore processors capable of running multiple threads.). This will be helpful in upcoming contention and overhead problems that are generally in multiprocessors environment. This will also help in upcoming resource chunks problems.

For using multicore power of processors there are various solutions available for overcoming the contention in the common worker queue approach:

- Using lock-free data structures
- Using concurrent data structures with multiple locks
- Maintaining multiple queues to isolate the contention

We are explaining how to maintain multiple task queues—a queue-per-thread approach—to isolate the contention means in this approach tasks comes from remote sources put in threads queue, as shown in Fig 2.

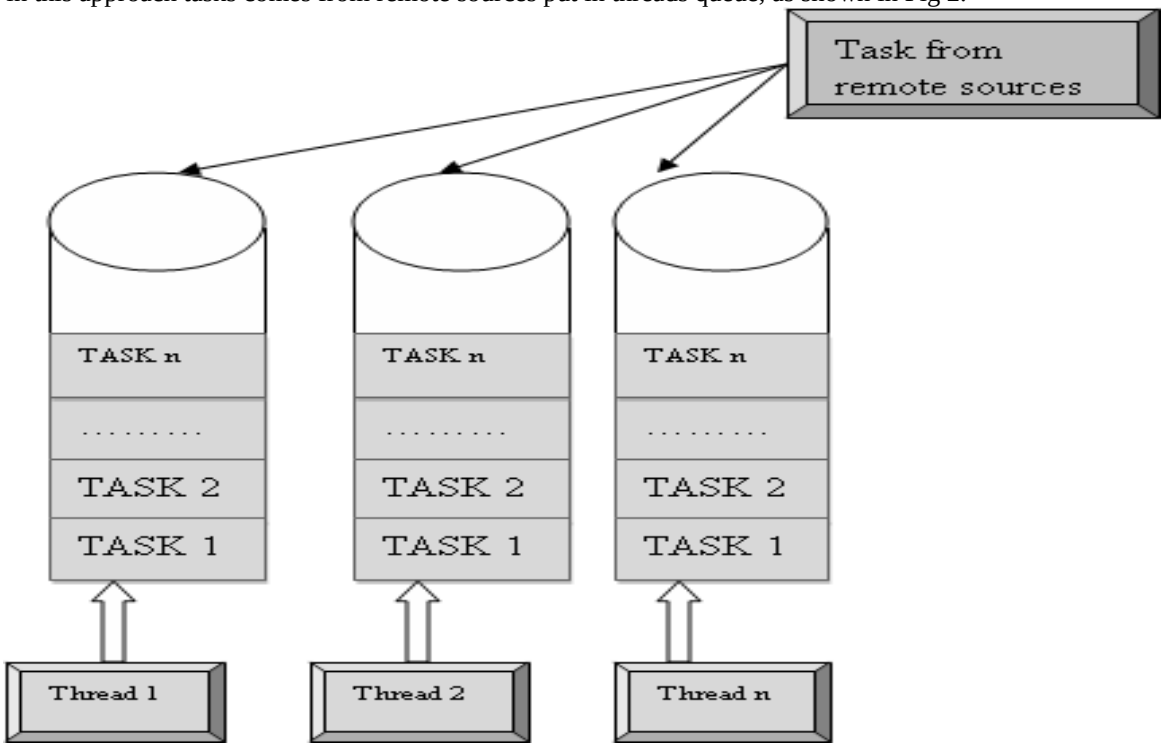

Fig:-2 ***Queue-per-Thread Queue***

Above explained approach each and every thread has their own worker queues that can get tasks from their assigned specific queues. This is not possible to get any other queues tasks for execution. Due to this approach isolates contention when retrieving tasks because there is no one to compete with. This guarantees that threads will not be in a sleeping state if there are tasks in the worker queue, which utilizes the cores effectively.

Although the queue-per-thread approach greatly reduces the contention, it does not guarantee that the underlying cores are used effectively all the time, for example, what happens if a couple of queues get emptied long before other queues? This is a common situation, and in this case, only a few threads execute the tasks whereas other threads (emptied queues threads) wait for the new tasks to arrive. This can happen due to following:

- Unpredictable nature of the scheduling algorithm
- Unpredictable nature of the incoming tasks (short versus long)

A solution for the above problem is *work stealing*. Work stealing is about to that one thread can steal work from another queue when thread finds that its own work queue is empty. This process ensures that all the threads (and, in turn, the cores) are busy all the time and make resource utilization at most.. Fig 3 shows a scenario where Thread 2 steals a work from Thread 1's queue because thread2 own queue is empty while thread 1 is have a lot to do. This process is helpful when resource utilization is to be achieved Work stealing can be implemented with standard queues, with implementation of dequeue drastically reduces the contention and starvation involved in stealing the work as well as it also resolves deadlock situations:

- Only the worker thread accesses the head of its own dequeue, so there is never contention for the head of a dequeue.
- The tail of the dequeue is accessed only when a thread runs out of work. There is rarely contention for the tail of any threads dequeue either.

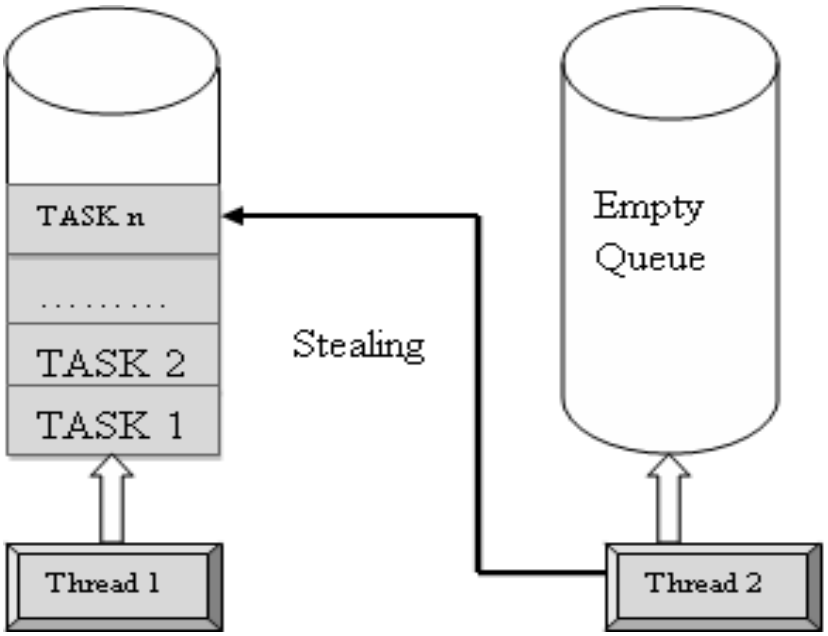

Fig3:- ***Queue-per-Thread Queue with Work Stealing***

IV. TEST CASE

In order to identify advantages and studied the behavior of the above mentioned approaches, we developed a test matrix for the three approaches mentioned. The test basically creates a lot of 10 x 10 matrixes that will generate multiplication tasks and executes them using the three approaches. The test defines the following classes:

- FirstClass: A class that initiates, starts, and coordinates various elements of the test.
- WorkProviderThread: A thread that creates a lot of 10 x 10 matrix multiplication tasks and queues them.
- Task: A class that defines a 10 x 10 matrix multiplication.
- WorkBenchQueue: An interface that defines a set of methods that any worker queue must implement.
- WorkerQueueFactory: A factory class that returns the workQueue object based on the queue type.
- SimpleWorkQueue: A class that defines a simple worker queue and initiates a set of threads. This depicts the first queue type mentioned (common worker queue).
- MultiWorkQueue: A class that isolates the contention by defining multiple workers queues (one per thread) and depicts the second queue type.
- WorkStealingQueue: A class that isolates the contention by defining multiple queues and steals work when it finds one of its thread's queues is empty. This depicts the third queue type mentioned.

The test can be executed by specifying the queue type, number of threads, and number of tasks.

V. EXPERIMENTAL RESULTS

I had evaluated the worker queue implementation performance in different os type architectures and the results are very positive. Initially, we evaluated the performance on an AMD Opteron box, which had eight core processors and ran Linux, and we found that performance for queue type 3 was improved by 11 to 18.7% over queue type 1, depending on the load, as shown in Fig 4.

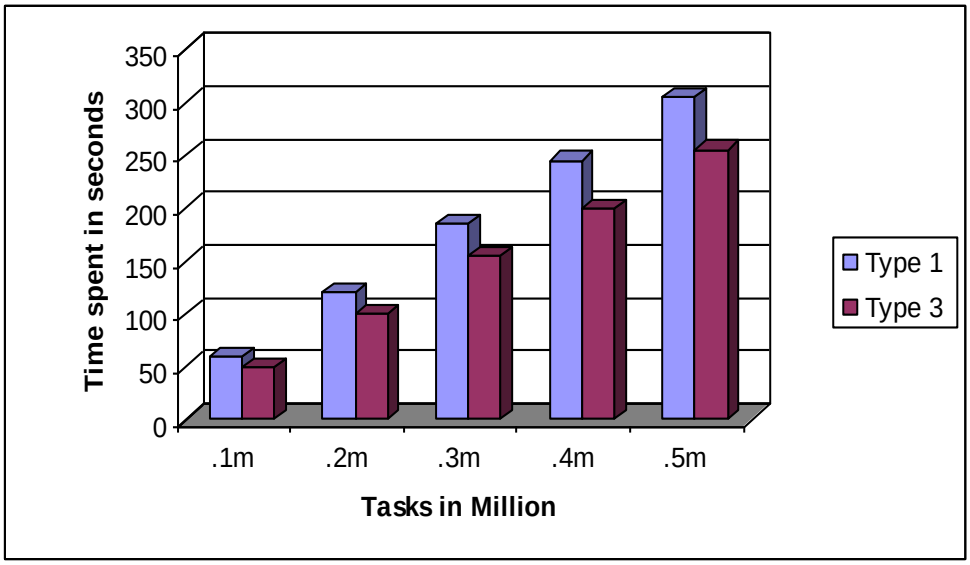

Fig 4. Performance Comparison between Type 1 and Type 3 Queues on Linux AMD Opteron System with Eight Core Processors.

[Disclaimer: This is not to compare any products or claim performance for any product; it is just to showcase the advantage of the techniques proposed in this paper, which are purely the authors' views.]

We also evaluated the performance in a Linux power system that had four dual-core processors (Power4), and we found that performance was improved by 11 to 16.5% for the same load, as shown in Figure 5.

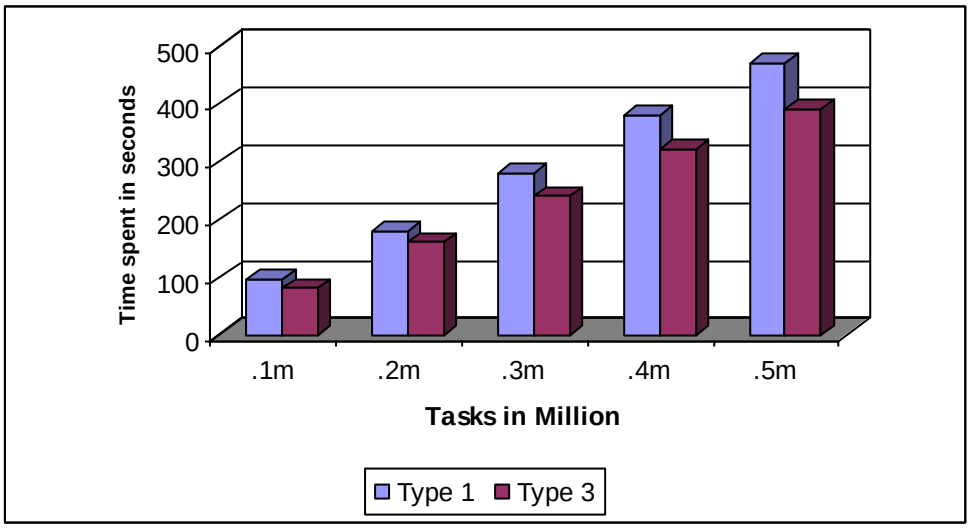

Fig 5. Performance Comparisons Between Type 1 and Type 3 Queues on Linux System with Four Dual-Core Power Processors.

As shown in Fig 4 and Fig 5, we varied the tasks from 0.1 million to .5 million and measured the performance in seconds. The outcome of our experiment clearly indicates that a great amount of contention is created in queue type 1 and it can be eliminated by creating multiple queues and stealing work.

VI. SUMMARY

This paper demonstrated the performance degradation involved in the common worker queue approach and then isolated the contention by creating one queue per thread. This paper also demonstrated, through a simple benchmark, why work stealing is important and how it improves the overall performance of an application. This paper is about to improve performance in multithreaded system by implementing queue based system that can improve the things.

ACKNOWLEDGMENTS

I would like to express my special thanks of gratitude to my guide Mr. Ashwani sir who gave me the golden opportunity to write this important paper on the topic Multithreaded Java Applications Performance Improvement. He provides me the directions to show my interest in java and also helped me in doing a lot of Research and I came to know about so many new things, I am really thankful to them.

REFERENCES

[1] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer. Java Concurrency in practice.
[2] Scott Oaks & Henry Wong. Java Threads.
[3] Pertrick Killelea. Web Performance Tuning.

.

