Performance Analysis of Load Balancing Queues in User Level Runtime Systems for Multi-Core Processors

Vikranth B
CVR College of Engineering/Information Technology Department, Hyderabad, India
Email: b.vikranth@cvr.ac.in

Abstract: **The speed of single processors is limited by the speed of light or speed of electron. Hence, processor manufacturers are packing multiple low speed processors or cores on to the same chip which are called multi-core processors. The number of processors on single chip is gradually increasing. Though multi-core processors are similar to Symmetric Multi-Processors (SMP), there are notable differences between them like shared last level cache. Operating systems consider these multi-core processors as SMP, and apply the suitable methods for task-scheduling and load balancing. But these strategies cannot fully explore the details of multi-core processors. The software also must be written to take the advantage of these multi-core processors. In this context, user level runtime systems evolved with a task as primitive concurrency construct. These tasks created by the program during runtime are added to the queues at user level runtime.**
In this paper, we analyze the performance of various user level queues and their contention using Java.

Index Terms—**Concurrency, Task, Work stealing, Centralized queue, Multi queue Double ended queue.**

I. Introduction

The goal of parallelism is to maximally exploit the number of CPU cores present at hardware level. The goals of parallelism are:

- Increasing throughput
- Reduce latency
- Reduce power consumption
- Scale depending on number of cores (CPUs).
- Prevent loss of data locality.

Since the operating systems consider multi core processors as SMPs, the user level scheduler has to take care of scheduling the tasks created by the user to schedule in an optimal fashion [1]. The common practice by popular operating systems in case of multiprocessors is maintaining a single queue or multiple queues per processor. The processes or threads get added to these queues dynamically during the runtime and are scheduled on to the CPUs. Rear end of the queue is used for adding work load and front end of the queue is used for popping the work load and be executed. But the smallest execution unit in operating systems like Linux is a process or a thread (in Linux, process and thread are created using fork() and pthread_create() respectively). But both of these calls involve clone() system call of kernel. Using clone system call involves much overhead making it heavy weight. Though thread creation is of less overhead than a process creation overhead, it is still considerable load on performance involving:

- system call overhead involving trap
- kernel level data-structure access on every operation.

Because of the above mentioned disadvantages of native thread API, modern approach of parallel programming runtime systems evolved from kernel level to user level.

These user level runtime systems are popular and became a De-facto standard for parallel programming [1]. Popular parallel runtimes such as Open MP, Cilk and TBB follow this approach for their runtime implementation. These runtime systems introduce a new scheduling entity called task which is considered to be even lighter in weight than thread since it is completely maintained at user level.

These runtime systems, provide API calls to create and maintain tasks. The programmer has to follow a sequence of API calls to implement parallel programs.

1. init():Initialize the runtime
2. spawnTask(): Create tasks where ever parallel activity is to be done
3. join():Wait for the tasks to complete.
4. release():Free the runtime.

init(): During the initialization of user level runtime system, a pool of native threads(pthread on Linux) is created and a single queue or multiple queues are created. These threads which get created during initialization of runtime are called worker threads or workers. Since it being a one time duty, during the initialization of runtime, every time a parallel execution entity is to be created, we need not enter the kernel level

spawnTask(): spawning a task allocates enough memory for task body and its parameters and this task object is added to the queue.

join(): All coarse grain or fine grain tasks have to wait until all the remaining tasks of that level have been completed at the joining points.

release(): It is the last call to the runtime which does join operation on native threads of thread pool and deallocation of queue objects which were created during the init().

In this paper, our focus is on studying the impact of the queue data structures which are used by worker threads to queue the tasks created by the programmer. We implemented three types of queues:

- Centralized or single queue
- Multiple queues without work stealing
- Multiple queues with work stealing

doi:10.32377/cvrjst1116

In this paper, the load balancing strategies are implemented and evaluated their performance by using a Matrix multiplication benchmark. To the best of our knowledge this is not ever studied in previous literature. Section II describes various types of queues used in user level runtime systems. Section III describes the experimental setup and result analysis.

II. Types Of Load Balancing queues

A. Single Queue

In this approach, during the initialization of runtime system, a single global queue data structure is created. This global queue is responsible for queuing the tasks created using *taskSpawn()*. This global queue is shared among all worker threads. Every worker thread is bound to a hardware level core or processor. If hyper threading is enabled at BIOS setup, the number of worker threads can be equal to the number of hyper threads. All these worker threads attempt to perform a dequeue operation on this global queue to get a task object when a worker becomes available. Once it is successful in dequeue operation a worker gets a task object and worker invokes its task body execution on its associated core.

Single queue approach is the simplest mechanism of implementing user level runtime system. When a worker thread is ready to execute a task, it attempts to dequeue a task from the global queue. This operation is a critical section and the worker must acquire a lock before this operation and release the lock after the operation. But it may suffer from the following disadvantages which may effect the overall performance of the parallel application.

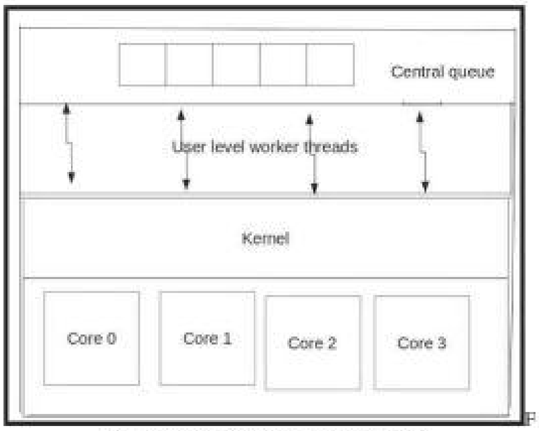


Figure 1: Centralized Queue Architecture

- Since all workers access a single queue worker threads may suffer from contention.
- It being a centralized approach, if worker threads become available at same instance, they have to compete to access the global queue. This may effect overall throughput of parallel task.
- Locality of the queue may cause cache performance isolation problem and false sharing.

B. Multi Queue

To overcome the main disadvantage of single queue approach, if a separate queue is associated with every worker thread, that is multi queue approach. Hence it is the first step for distributed load balancing. The tasks created by the programmer are added to separate queues associated with individual worker threads [2]. This approach guarantees transparent load balancing with the constraint that all the tasks are of equal duration. Since individual worker thread has access to its own queue, they need not contend on dequeue operation.

But this multi queue approach is not effective when all tasks are of variant duration. If tasks are of different

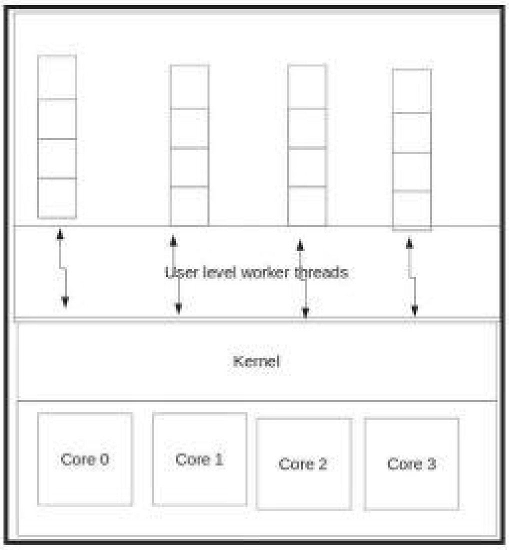


Figure 2: Multi Queue Architecture

durations, a queue associated with short duration jobs may become empty where as a queue with big duration tasks is overloaded.

C. Work stealing Queue

Work stealing is a distributed dynamic load balancing technique [3]. In this strategy, the thread pool maintains separate work queue per worker thread with stealing ability. When a work queue becomes empty it can attempt to steal a task from other queues. Since one end of the queue is accessible by its worker thread and other end is to add tasks. These queues need an additional feature of dequeuing from both ends in case of stealing by other worker. Double ended queues are used to implement this feature since it allows task deletion from both ends [4]. One end is accessed by the associated worker thread and other end open for stealing by other worker threads when they become empty. The tasks are created by application during runtime using the taskSpawn() or similar construct. The task gets added at the tail of a queue associated with each

doi:10.32377/cvrjst1116

processor. When execution reaches a fork point such as a spawn or parallel loop, one or more new tasks are created and put on a queue. The main strategies by which idle workers find new tasks are:

- Find a task from its own work queue
- Distributed work queues with randomized stealing

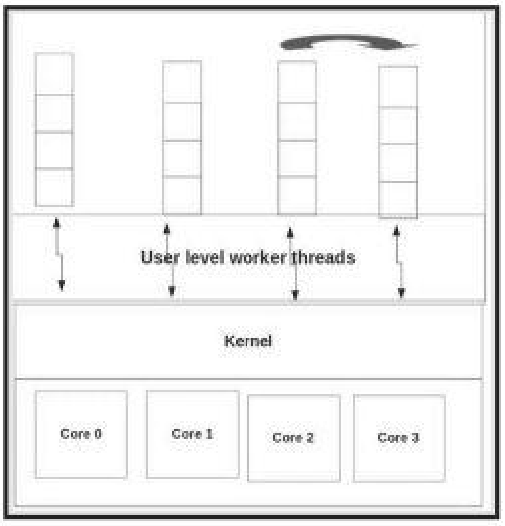


Figure 3: Work Stealing Queue Architecture

Since work stealing queues allow balancing of load across queues, it is more effective than multi queue approach.

III. Experimental Results and Analysis

To illustrate the performance of different queue approaches at user level runtime systems, we implemented different types of queues in Java. The worker threads are plain Java threads which implement *java.lang.Runnable* interface. Three types of worker queues are implemented as blocking queues. Task construct is provided to the parallel programmer simply in the form of an interface. If the programmer has to implement his task body he can simply implement the interface and override the run() method of the interface. To make the job of testing different types of workloads, the user can choose an option at command line interface. The relationships among the classes are presented in Figure 4.

To evaluate the performance of the above mentioned three approaches, we implemented Square Matrix multiplication benchmark program. In this parallel implementation each task is created to find a single element of the product matrix (i.e. if the given two matrices A and B are 512 X 512 matrices, a total of 2^{18} tasks are created during execution. The intension behind taking large sized matrices is to overload the queues with task objects and test how effectively load balancing is done across the queues. The benchmark is executed on

IBM x3400 server with 4 core (8 cores if hyper threading is enabled) Xeon E5-2401 and Linux kernel version 3.16.2.200. The number of worker threads taken in our experimental set up is equal to the number of cores (4) by disabling hyper-threading.

It can be observed from the execution times presented in Table1 that work stealing queue approach gives better performance than Centralized queue and Multi worker queue approaches. Though the difference of execution times between multi worker queue approach and work stealing is little, the difference is gradually significant for bigger matrix size inputs. The main difference is due to the stealing approach to balance the load across queues which is not addressed in plain multi queue approach. As stated in introduction section, centralized queue approach gives poor performance due to contention among the worker threads to access the single global queue. The difference of execution time is spent on saving the critical section code to access the global queue.

TABLE I

Execution Times Of Various Worker Queue Approaches

Matrix Size	Execution time in milliseconds		
	Centralized queue	Multi queue	Work stealing queue
128	2	1	1
256	7	6	5
512	15	14	14
1024	22	21	20
2048	53	47	42
4096	153	150	141

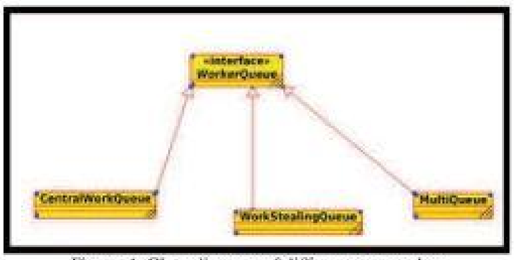
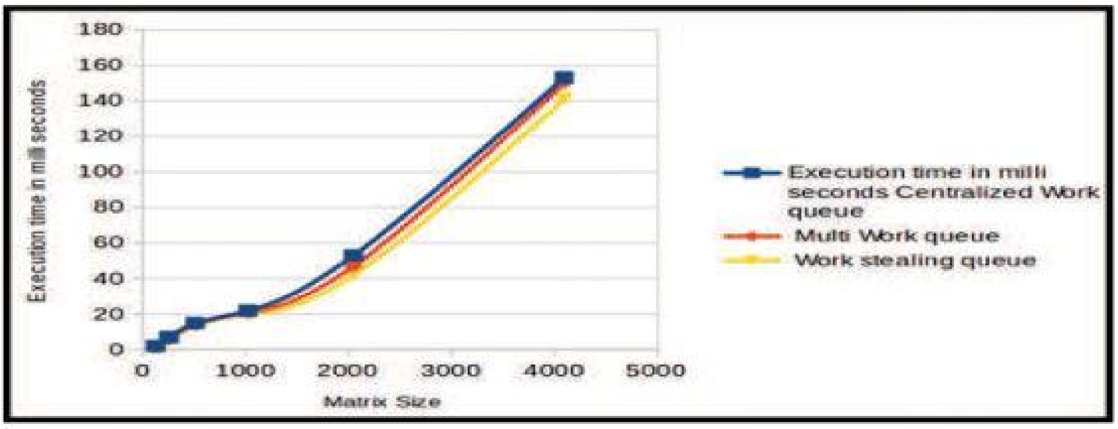


Figure 4: Class diagram of different approaches



IV. Conclusions

In this paper, we the effect of different load balancing queues on performance of task parallel applications. It is observed that work stealing queue approach performs better when compared to centralized queue approach and multi queue approach.

References

[1] Blagodurov, Sergey, and Alexandra Fedorova.*"User-level scheduling on NUMA multicore systems under Linux." 2011* s.l. : Proceedings. of Linux Symposium., 2011.

[2] P.E. Hadjidoukas, G.Ch. Philos, V.V. Dimakopoulos, *"Exploiting fine-grain thread parallelism on multicore architectures"*, .. 2009, Scientific Programming,, pp. Vol. 17, No. 4, Nov., pp. 309–323.

[3] Faxén, Karl-Filip.*"Wool-a work stealing library"*. s.l. : ACM SIGARCH Computer Architecture News, 2009. 36.5 : 93-100.

[4]Hendler, Danny, et al *"A dynamic-sized nonblocking work stealing deque"*.. s.l. : ACM, 2005.

doi:10.32377/cvrjst1116