Performance Enhancement of Multicore Processors using Dynamic Load Balancing

Ajay Tiwari
School of Computer Science
Devi Ahilya University (DAVV)
Indore, India
e-mail: tiwariajay8@yahoo.com

***Abstract*—** Introduction of Multi-core Architecture has opened new area for researchers where dynamic load balancing can be applied to distribute the work load among the cores. Multi-core Architecture provides hardware parallelism through cores inside CPU. Its increased performance and low cost as compared to single-core machines, attracts High Performance Computing (HPC) community. The paper proposes a user level dynamic load balancing model for multi-core processors using Java multi-threading and use of Java I/O framework for I/O operations.

Keywords*-*Multithreading;dynamic load balancing; multicore;

*****

I. Introduction

Nodes of the distributed computing environment, comprising multi-core processors are becoming more popular than traditional Symmetric Multiprocessor (SMP) computing nodes. Both scientific and business applications can be benefited from multi-core processors [3]. Due to large difference in the architecture of single-core and multi-core processors, the existing dynamic load balancing techniques cannot be directly applicable in DCE comprising multi-core processing elements. The existing DLB techniques for distributed computing environment viz. cluster, grid and cloud, distribute and balance the load among the nodes whereas for DCE comprising multi-core processing elements, a two stage load balancing is required: in first stage DLB among the nodes and in its second stage among the cores of the nodes.

From 1994 to 1998, CPU clock speeds rose by 300% and it was expected by the processor manufacturers that in near future processors clock speed will reach up to 10.0 Ghz and processors would be capable of processing one trillion operations per second. However it was observed that with the increase of clock speed, processors consume more power and generate more heat. This extra heat generation became barrier to speed acceleration of CPU. Therefore from 2007 to 2011, maximum CPU clock speed raised from 2.93 GHz to 3.9 GHz i.e. an increase of 33%. Later on, the improvement in the processor's performance was observed with the invention of multi-core processors. As shown in Figure 1, in multi-core architecture, processes are executed on more than one core of the processor, each having restrained clock speed which provides hardware parallelism and is named as Chip Multi Processing (CMP). Prior to CMP, HPC community used Symmetric Multi Processors (SMP). The main drawback of SMP is that, processors of SMP communicate through motherboard whereas lying on the same die, cores of CMP communicate through faster cache. Clusters using multi-core nodes are more popular and due to its better cost-to-performance ratio draw the attention of scientific institutions and business organizations. One of the major challenges for multi-core nodes is running parallel applications and tasks to cores mapping such that each core of the computing resources are efficiently utilized [1].

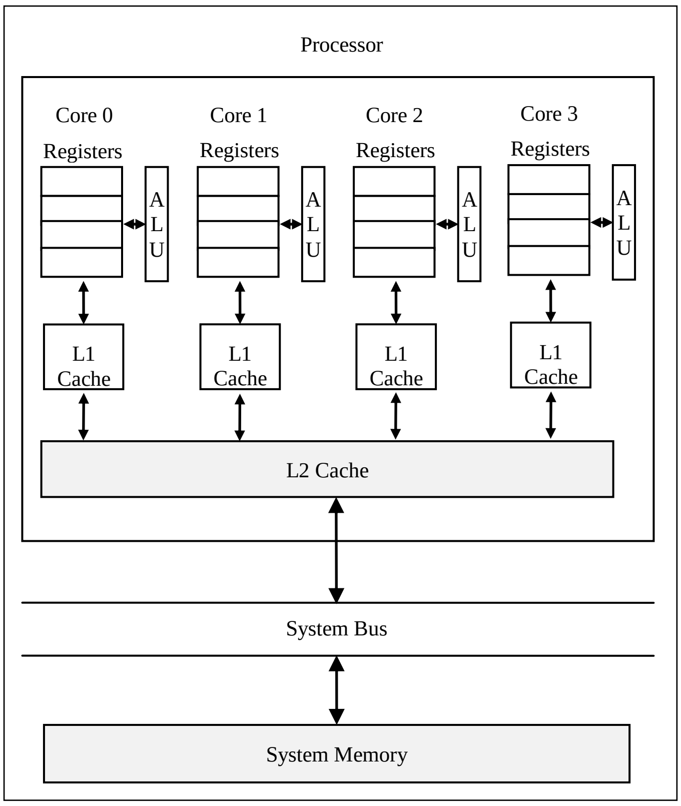

Figure 1 Symmetric Multicore Processor

Multi processing and multi-core processing elements provide parallelism at hardware level. However, parallel processing cannot be achieved without the support of parallelism at software level. In software, parallelism can be achieved through Instruction-Level Parallelism (ILP) or Thread-Level Parallelism (TLP). ILP works on the machine-instruction and helps the processors to split the instructions into sub-instructions and re-order the instructions & sub-instructions as per need. TLP is a boon for multi-core processors where different threads are executed on different cores to achieve parallelism. Some examples of TLP are as follows:

- Mail-server allows reading of e-mails and downloading the material simultaneously using separate threads.
- Computer game software applies physics, AI and graphics using separate threads.

The advanced technique of integrating registers produces complex multi-core processors ranging from symmetric multi-core to asymmetric multi-core processors. In symmetric multi-core processor, all cores present in a die are identical whereas cores of an asymmetric multi-core also preset in a same die but are of different design and different capabilities. Multi-threading based applications make best use of symmetric multi-core processors whereas asymmetric multi-core processors are used by special purpose programs viz. video games, home theatre etc. Two barriers which deteriorate the performance of multi-core processors: I/O operations and unequal distribution of task among the threads which results workload imbalance among the cores.

In proposed model, Java multi-threading is used to distribute the workload evenly among the cores and Java I/O framework reduces the gap between processor performance and I/O performance.

II. Related Work

Several approaches have been proposed in literature to address the issue of utilization of multi-core processors using load balancing at core level. The load distribution is performed based on observed behavior of application. In these types of load balancing strategies, an algorithm is tailored to improve performance of a particular parallel application and is not suitable for general purpose parallel applications.

Lea proposed Java Fork/Join framework to support divide-and-conquer problems. This framework is easy to use and consists of splitting the task into independent subtasks via fork operation, and then joining all subtasks via a join operation. The performance primarily depends on garbage collection, memory locality, task synchronization, and task locality. The framework is made up of a pool of worker threads, a fork/join Task, and queues of tasks. The worker threads are standard (“heavy”) threads. The framework also uses work-stealing algorithm which consists of, from an empty queue of a worker thread, popping a task belonging to a non-empty queue of another worker thread. When the subtask size is smaller than the threshold, subtask is executed serially. But, sometimes it is very difficult to determine the threshold [7] [10].

Zhong presented algorithms for both inter-node and intra-node load balancing based on performance models, previously developed for each node. Performance models of each core in a multi-core processor are created. However, as cores compete for shared resources and affect the performance of each other dynamically, the performance model of an individual core may not be realistic. In most of the models, including the model for heterogeneous clusters, authors treat all the nodes of the cluster as having equal computing capabilities, whereas in practice, cluster may consists of nodes having different computing capabilities [14] [4]].

Wang pointed out two common problems of utilizing processors under multi-core architecture, namely processors waiting for IO operation to finish and load balancing among cores. In order to exploiting multi-core processing power of, he proposed a multi-core load balancing model using Java framework. Wang considered priority of processes at the core level, which reduces the performance of the algorithm [13]. Hofmeyr presented a load balancing technique designed specifically for parallel applications running on multi-core systems. Instead of balancing run queue length, author’s algorithm balanced the time for which a thread has executed on “faster” core and “slower” core [6].

To exploit multi-core architecture, a major challenge is to convert single threaded applications to multithreading codes [9]. Hybrid programming paradigms have been reported in several published work that mainly experimented on SMP cluster. IBM SP systems are used by Cappello & Daniel to compare NAS parallel benchmarks on SMP cluster. Authors also presented a study of communication and memory access patterns in the cluster [2]. Hit rates of L1 and L2 cache are studied by Taylor & Wu on multi-core cluster by using ‘National aeronautics and space administration Advanced Supercomputing (NAS)’ parallel benchmarks SP and BT [12].

In the related literature, we have observed that either the researchers have improved I/O bottleneck or parallelism. The proposed model tries to improve both I/O bottleneck and parallelism simultaneously.

III. Proposed Framework

In the past, the only way to deal with divide-and-conquer problems was the use of low level threads by native methods. The number of threads created by native methods were equal to number of tasks i.e. there is one-to-one correspondence between number of tasks and number of threads which results poor performance as some threads are of heavy weight while others are light weight. One slice thread and 10 slices thread are treated in similar way at the time of allocation to the cores of multi-core processor which results in load imbalance at core level. Similarly, there is a large difference between processor’s speed and I/O speed. A processor or a core has to wait a lot till the completion of I/O operations.

The paper addresses both these problems by using Java Fork/Join framework for parallel applications and Java New Input-Output (NIO) framework to speed up I/O operations for reducing the weighting time of the cores. The Java Fork/Join framework was included in JDK 1.7 API and Java NIO framework was introduced in JDK1.4.

A. Fork/Join Framework

Fork/Join framework is a classical way of solving divide-and-conquer problems. Lea has introduced Java Fork/Join framework to deal with parallel programming through high level threads with the goal to minimize execution time by exploiting parallelism [7] [15]. The framework can be described as follows:

a) Partition into Sub-Problems: The problem is broken up into manageable sub-problems where each sub-problem should be as independent as possible (one of the important decomposition principles).
b) Create Subtasks: The solution to each sub-problem is found as a Runnable task.
c) Fork Subtasks: The subtasks are handed over to pool of worker threads where pool size depends on the number of cores of a multi-core node.
d) Join Subtasks: Compose the solution of the subtasks belonging to a worker thread. Repeat the step for all the worker threads.

e) Compose Solution: Integrate the solution provided by the worker threads.

As stated above, the framework balances the load among multi-cores in step (c) where tasks are handed over to worker threads. The load balancing is achieved by serializing more than one light weight tasks to a worker thread whereas a heavy task is taken care by an individual worker thread.

B. *Java NIO Framework*

Java New Input/Output (NIO) is an alternative to standard java stream based I/O API. Although Java NIO comprises of number of classes and components, its core components are Channel, Buffer and Selector. Java NIO works with channels and buffers instead of byte streams and character streams [11].

A Channel is similar to streams, with the difference that Channels read data into Buffers and Buffers write data into Channels. To handle file and network I/O, there are several Channels provided by Java viz. FileChannel, DatagramChannel, SocketChannel, ServerSocketChannel etc. It is known that a buffer is a block of memory into which data is written as well as read from. Java NIO, wrap these memory blocks in a Buffer object, where a set of methods allowed easy handling of buffers. In Java NIO, a selector is an object that can monitor multiple channels for events viz. connection opened, data arrived etc. and allows a single thread to handle multiple Channels (connections) [5].

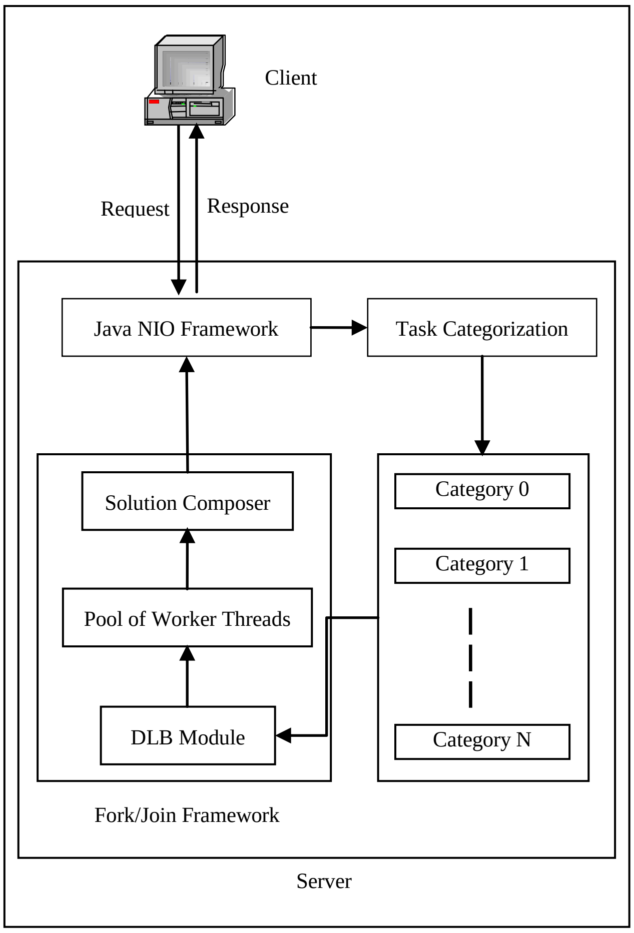

Figure 2 Proposed Multicore Server Load Balancing Architecture

C. *Proposed Server Load Balancing Architecture*

As shown in Figure 2, this section discusses the proposed multi-core server architecture where we have combined Java NIO and Fork/Join framework. The working of the model is as follows:

- Java NIO framework is used to handle network I/O and file I/O which minimizes the multi-core waiting time.
- Requests are categorized into classes viz. data-intensive, computation-intensive etc. to help in splitting the tasks as well as serialization of light weight tasks.
- Fork/Join framework is used to exploit multi processing where a load balancer serializes light weight tasks to worker threads for even workload distribution among the multi-cores. Finally, the response is sent back to the client through NIO framework.

The proposed system considers symmetric multi-core nodes where all the cores are identical. Though the proposed framework is not able to solve all the problems, it utilizes multi-cores in such a way that heavy tasks and light weight tasks complete their execution almost simultaneously and improves the performance of overall system.

D. *Proposed Load Balancing Algorithm*

Algorithm considers the following assumptions:

- The server has *n* cores $C_1, C_2, \dots, C_n$.
- Q_i is the queue attached with core C_i.
- W_i is the worker thread which executes the tasks of Q_i. queue.
- There are *m* tasks $T_1, T_2 \dots T_m$ in the system.

Task T_i can be divided into *k* parallel subtasks $T_{i1}, T_{i2} \dots T_{ik}$ where *k* may vary from task to task.

a) For all tasks T_i, distribute parallel subtasks $T_{i1}, T_{i2} \dots T_{ik}$ into all task queues such that each queue has k/n parallel subtasks.
b) Worker thread W_i executes the tasks of queue Q_i on core C_i.
c) Worker threads use adoptive migration techniques to migrate a process from a heavy loaded queue to their own queue. The technique works as:
   i) For a lightly loaded system it uses pull based or receiver initiated technique to pull the tasks from a longer queue.
   ii) For a moderately or heavy loaded system the sender initiated technique is being used where a worker thread searches least loaded queue and request for process migration.

IV. EXPERIMENT AND RESULT ANALYSIS

To compare the performance of proposed work, various experiments are executed using proposed DLB algorithm without external load balancing. The experiments cover Matrix Multiplication, Merge Sort and Fibonacci [10]. For matrix multiplication, we used iterative method, where merge sort, uses divide-and-conquer algorithm which divides the list into two equal sub lists until the sub list is reduced to two elements. Fibonacci is a compute intensive algorithm to find the sum of natural numbers. For each experiment, number of subtasks executed on each core and total execution time with and without DLB is collected [13]. The Experiments are performed in IBM System X-3200 M3 Server 7328-I6S, Intel Xeon E 3430 (Quad Core) 8 MB Cache 1333 MHz 2.4 GHz, 8 GB Ram running Linux Fedora 11 with Java 1.7.0 runtime environment. The experiments are repeated five times and results are averaged.

TABLE I NUMBER OF PROCESSES

Experiments	Core 0	Core 1	Core 2	Core 3
Fibonacci	9837	8698	9119	10510
Merge Sort	14376	16504	14682	16869
Matrix Mul	23327	19430	21465	16320
LB Fibonacci	9321	9698	9631	9514
LB Merge Sort	16012	15131	15705	15583
LB Matrix Mul	19785	20641	20149	19967

TABLE II PROCESSES EXECUTION TIME IN MILLISECONDS

Experiments	With DLB	Without DLB
Fibonacci	5478	6183
Merge Sort	7932	9319
Matrix Mul	9163	11063

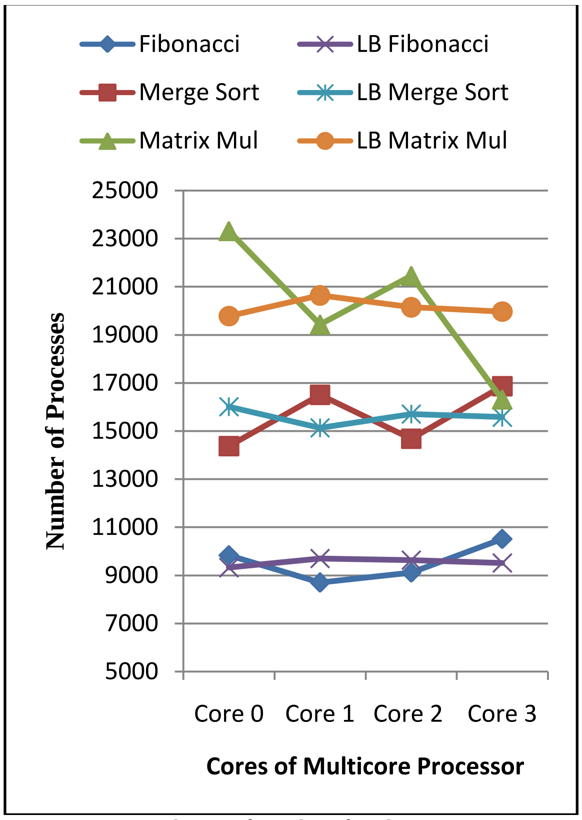

Figure 3. Distribution of Number of Tasks in various Cores.

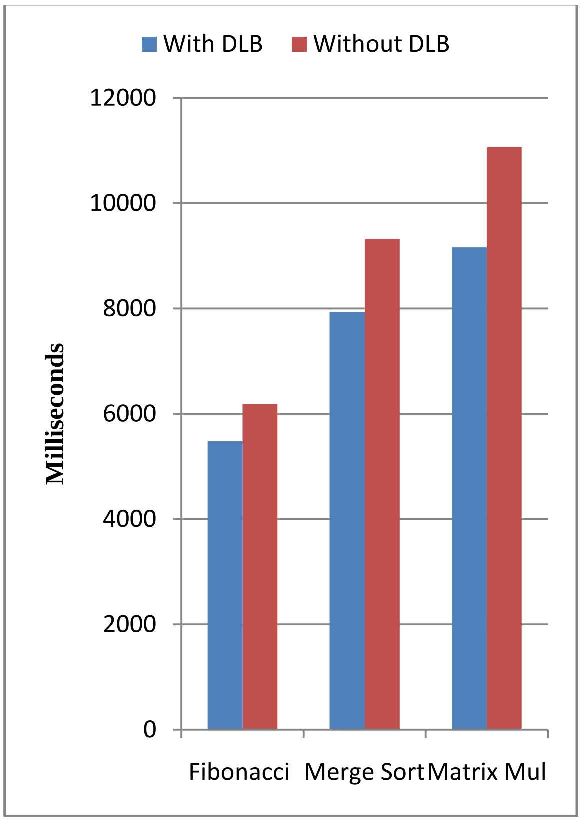

Figure 4. Comparison of Execution Time With and Without DLB.

The Fibonacci function is executed for fifty arguments and merge sort is used to sort twenty five thousands integers to test the performance of the proposed model. Tables I and Figure 3 depicts the number of processes that are distributed among the cores. Figure 3 shows that the processes are distributed evenly when DLB algorithm is being used. On the other hand, in random distribution method, uneven distribution can be observed. The distribution of tasks of smaller size is similar to the proposed model as in the case of Fibonacci and Merge sort but for larger tasks like Matrix multiplication, a large difference can be observed in distribution of processes by using both methods as shown in Figure 3. Table II and Figure 4 show the completion time of the various experiments using proposed DLB model and using random distribution method. Approximately 13%, 17% and 21% increase of throughput of the system using proposed model over random distribution method for Fibonacci, Merge Sort and Matrix multiplication experiments respectively has been observed. Therefore, the proposed model is useful for larger tasks.

V. CONCLUSION

Multiple cores are effective for data parallel applications where same code can run through multiple threads on different sets of data as well as for functionally decomposed computation intensive tasks where each task run in parallel on different cores [8]. Design of dynamic load balancing algorithms for multi-core processor based DCE is more complex than DCEs having uni-core processor based nodes.

The paper proposes an adaptive load balancing model for symmetric multi-core nodes where all the cores of a node are identical. The model used two frameworks for I/O as well as for parallel programming. Experimental results show that the proposed model is feasible for large tasks and all the processes finish almost at the same time which indicates the overall performance of the multi-core processors.

REFERENCES

[1] S. Akhter and R. Jason, “Multi-Core programming: increasing performance through software multi-threading,” Machine Press, Beijing, China, 2007, pp. 12-13.

[2] F. Cappello and D. Etiemble, “MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks,” IEEE conference on Supercomputing, Nov. 2000, pp. 12-23.

[3] J. Chen, W. Watson and W. Mao, “Multi-threading performance on commodity multi-core processors,” 9th International Conference on High Performance Computing, March 2007, pp. 1-8.

[4] J. Dummler, T. Rauber and G. Runger, “Scalable computing with parallel tasks,” 2nd Workshop on Many-Task Computing on Grids and Supercomputers, Portland, Nov. 2009, pp. 141-148.

[5] R. Hitchens, “How to build a scalable multiplexed server with NIO,” JavaOne Conference, 2006.

[6] S. Hofmeyr, C. Iancu and F. Blagojevic, “Load balancing on speed,” ACM Symposium on Principles and Practice of Parallel Programming, Bangalore, India, Jan 2010, pp. 147-158.

[7] D. Lea, “A Java Fork/Join Framework,” In JAVA’00, 2000, pp. 36-43.

[8] A. Mamidala, “MPI collectives on modern Multicore clusters: performance optimizations and communication characteristics,” 10th IEEE International Conference on Cluster, Cloud and Grid Computing, Melbourne, May 2010.

[9] M. Parsons, “The challenge of Multicore: a brief history of a brick wall,” Technical Report, Available online at: http://www.epcc.ed.ac.uk.

[10] A. Senghor and K. Konate, “A Java Fork-Join framework-based parallelizing compiler for dealing with divide and conquer algorithm,” Journal of Information Technology Review, Vol. 4, No 1, Feb 2013, pp. 1-12.

[11] R. Standtke and U. Nitsche, “Java NIO framework-introducing a high-performance i/o framework for java,” In Proceedings of the Internationl Conference on Software and Data Technologies (ICSOFT 2008), Porto, Portugal, 2008.

[12] V. Taylor and X. Wu, “Performance characteristics of hybrid MPI/OpenMP implementations of NAS parallel benchmarks SP and BT on large-scale Multicore clusters,” International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems , Vol. 38, No. 4, March 2011.

[13] Y. Wang and G. Qin, “A Multicore load balancing model based on Java NIO,” Indonesian Journal of Electrical Engineering, Vol.10, No.6, Oct 2012, pp. 1490-1495.

[14] Z. Zhong, V. Rychkov and A. Lastovetsky, “Data partitioning on heterogeneous Multicore platforms,” IEEE International Conference on Cluster Computing, Sept. 2011, pp. 580-588.

[15] Z. Zhou, “Understanding the JVM: advanced features and best practices,” Machine Press, Beijing, China, 2011, pp. 336-337.