A Multicore Load Balancing Model Based on Java NIO

Yang Wang*, Guofeng Qin

The CAD research center, Tongji University, 4800 Caoan Road, Jiading District, Shanghai, China

*corresponding author, e-mail: pingpong006@163.com*, gfqinq@yahoo.com.cn

Abstract

First this paper points out two common problems of utilizing processors under multicore architecture, namely processors waiting for IO operation to finish and load balancing among cores. Then it makes an analysis of the reasons for them. In order to fully exploit multicore processors, this paper proposes a multicore load balancing model based on the Java NIO framework which offers a solution to above problems. This model mainly illustrates a task scheduling algorithm which uses a parallel computing framework, Java Fork/Join. At last, experiments and performance analysis prove the effectiveness of this model in utilizing the multicore processors. Although the model is constructed under the architecture of Java language, it can be extended to other languages without much being changed.

Keywords: Java NIO, Fork/Join, multicore, load balancing, task scheduling

1. Introduction

As multicore processor is becoming so common nowadays, more and more servers have started to use the multicore architecture. But there exists some problems in utilizing the multicore processors. First, it is known to us that the traditional blocking I/O operation is much slower than the CPU’s processing speed, thus making CPU waiting simultaneously instead of doing any practical jobs[1]. Second, tasks are distributed unevenly to threads which leads to workloads unbalancing among cores[2]. These problems affect the overall utilization and performance of parallel processors.

To solve the problems mentioned this paper proposes a multicore load balancing model based on the Java NIO (Java New I/O) framework. The model utilizes both the high-performance non-blocking I/O in Java NIO framework and parallel processing ability of multicore processors. Besides, load balancing is achieved.

2. Java NIO

To keep CPU from waiting I/O simultaneously, the model uses Java NIO Framework which was included in JDK1.4 API. Java NIO was designed to provide access to the low-level operations of modern operating systems and the intent is to facilitate an implementation that can directly use the most efficient operations of the underlying platform[3]. It makes up for the traditional way of I/O, namely blocking I/O by providing buffer-based multichannel non-blocking I/O methods in Java language.

Java NIO makes use of the Reactor design pattern and events are selected by the selector at set intervals. When a non-blocking method is dealing with I/O operations it might return without waiting for the I/O to finish, thus achieving higher performance and greater parallel processing capacity. Figure 1 shows a sample of how Java NIO works. A ServerSocketChannel is first registered to the selector as an acceptor or a request handler. Then the acceptor handles the client requests passed from the selector and spawns SocketChannels to receive or send data. The Selector polls all channels and dispatches client requests or I/O operations at set intervals.

3. Multicore Load Balancing

In this section, we will discuss multicore load balancing in a user-level way rather than a kernel way. 3.1 illustrates the type of load balancing involved, and 3.2 introduces the Java Fork/Join framework which is the basis of the model proposed later. In 3.3, we propose the
multicore load balancing model and at last in 3.4 an overall framework combing Java NIO and the model is given.

![Diagram of Java NIO](image)

**Figure 1** How Java NIO works

### 3.1 Cause of Multicore Load Unbalancing

This paper focuses on the type of multicore load unbalancing caused by tasks distributed unevenly to multi-threads.

First, from the version of JDK1.2 Java thread is implemented by native thread, that means how operating system supports native thread decides the implementation of JVM threads. Both the Windows version and the Linux version of Sun JDK use a 1:1 mapping to implement Java thread[4], in other words, a Java thread is mapped to a LWP (light-weight process) and could be regarded as a native thread. This paper is based on Linux and can be extends to Windows or other similar operating systems.

Second, after comparing different combinations of CPU workload factors Kunz pointed out that the number of tasks in running queue is the best factor for evaluating the CPU workload in Linux[5], and actually the Linux kernel put it in use. In Linux, a task is a thread. As is mentioned, a Java thread is a native thread or a kernel thread, therefore, a 1-slice Java thread and a 10-slices Java thread are equivalent when scheduled to cores by the kernel. Thus multicore load unbalancing emerges.

When developing an application under multicore architecture, distributing tasks evenly to cores by using multi-threading techniques is an effective way to enhance system performance[6]. This paper proposes a task scheduling model to achieve the goal.

![Diagram of Fork/Join Framework](image)

**Figure 2** Fork/Join Framework
3.2 Fork/Join Framework

The proposed model is based on the Java Fork/Join framework introduced in JDK1.7[7]. Fork/Join framework is a classic way of dealing with parallel programming. Though it could not solve all the problems, it is able to utilize multicores and make them cooperate to finish heavy tasks within its Applicable scope.

The theory of Fork/Join framework can be described as follows: If a task could be divided into some subtasks and the results are easily to be obtained to combine these subtasks, then the task is fit to be solved by Fork/Join framework. Figure 2 shows how it works. The task above is dependent on the subtasks below. The request will not get the result of Task 0 until all the subtasks return. Other problems related to parallelism such as load balancing and synchronization could also be solved under this framework.

3.3 Task Scheduling Model

3.3.1 Task Parallelization and Serialization

When dealing client requests server applications are likely to handle both heavy tasks and light-weight tasks. If one heavy task is distributed to one thread, it would cause load unbalancing. Analogously, too many light-weight tasks that each occupies a thread would lead to high system cost[2]. Thus tasks executed in threads should be controlled in a suitable size. Figure 3 shows a way of task size controlling by parallelizing heavy tasks and serializing light-weight (LW) tasks.

![Diagram](3.3.2 Task Scheduling Algorithm)

As light-weight task Serialization is relatively easy to attain, it just waits for enough tasks to arrive and fetches a thread from the thread pool to execute the tasks. Thus we mainly discuss the heavy task parallelization here. The task scheduling algorithm is based on Java Fork/Join framework and makes an improvement of classic task scheduling algorithm.

We have definitions below: (1) The server has n processors $P_1, P_2, \ldots, P_n$, and each processor $P_i$ has a running task queue $Q_i$ which the subtasks run on. (2) There are also $n$ worker threads $W_1, W_2, \ldots, W_n$, and $W_i$ executes the tasks on $Q_i$. (3) There are $k$ tasks $T_1, T_2, \ldots, T_k$, whose priority ranges strictly from high to low. $T_i$ could be divided into $m$ parallel subtasks $t_1, t_2, \ldots, t_m$ ($m$ may differ for different tasks) which inherit the priority from their parent task by fork operation.

The steps of the algorithm are the following:

1. For task $T_i$, the parallel subtasks $t_1, t_2, \ldots, t_m$ are distributed evenly to all task queues, so each queue has $m/n$ parallel tasks. Tasks with higher priorities enqueue earlier than other tasks. Thus the distribution of parallel tasks is obtained and is presented in Figure 4.
II. If $Q_i$ is not empty, $W_i$ dequeues a task from it to execute; else go to Step III.

III. $W_i$ searches the system for the processor which has the longest task queue, locates the task that has the highest priority and migrates it to $Q_i$. Go back to Step II.

IV. If the migration in Step III fails, $W_i$ tries to migrate lower-priority tasks or search for other processors to repeat the migration. If all these fail and $Q_i$ is still empty, $W_i$ blocks and waits for new task to awake.

![Figure 4 Distribution of Subtasks on Multicores](image)

On one hand because of the FIFO task queue structure, later high-priority tasks enqueue after earlier low-priority tasks, which assures the chance of execution of low-priority tasks. On the other hand task migration get high-priority tasks executed as soon as possible.

3.4 Server Load Balancing Architecture

This section combines Java NIO and the proposed task scheduling algorithm to obtain an overall server load balancing architecture. Some implementation details are presented as follows:

a. Task type: generally task type determines task size and how tasks are handled-parallelized or serialized. Thus when receiving a task, the server judges its task type first to decide what to do next. For example, a file access task or a mathematical calculation task has more work to do than a form submitting task and had better to be parallelized.

b. Message priority queue: in order to handle messages with different priorities, a data structure of message priority queue is introduced to classify the messages. When these queues are polled, more messages in high-priority queues would be chosen and forked.

c. I/O: in the architecture Java NIO is used in both network I/O and native I/O to minimize the time of waiting for processors.

d. CPU affinity: as a worker thread takes charge of a processor’s queue tasks, it is likely to be bound to the processor. But Java language does not provide CPU affinity, thus we use JNI (Java Native Interface) and set CPU affinity of worker thread by invoking a low-level C language dynamic library.

The architecture is presented in Figure 5. The load balancing module implements the proposed task scheduling algorithm and is the key to multicore load balancing and system performance.

4. Experiment and Analysis

In this section, an experiment is done to test the task scheduling algorithm in load balancing and utilizing multicore processors. It compares the results of task execution with one core, two cores and four cores. We choose three types of classic tasks for testing, they are the Fibonnaci program with argument 40(Fib), quick sorting of 20,000 integers(Sort) and multiplication of 512-bit and 512-bit integer(Mul). All three types have the same priority. For
each type of task, number of subtasks executed on cores and the total execution time was gathered. The tasks are executed 10 times and the results are averaged which are presented in Table 1, Table 2 and Figure 6.

![Diagram](image)

**Figure 5. Server Load Balancing Architecture**

**Table 1. Number of subtasks on cores**

<table>
<thead>
<tr>
<th>1 Core</th>
<th>2 Cores</th>
<th>4 Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU0</td>
<td>CPU0</td>
<td>CPU0</td>
</tr>
<tr>
<td>CPU1</td>
<td>CPU1</td>
<td>CPU1</td>
</tr>
<tr>
<td>CPU2</td>
<td>CPU2</td>
<td>CPU2</td>
</tr>
<tr>
<td>CPU3</td>
<td>CPU3</td>
<td>CPU3</td>
</tr>
<tr>
<td>Fib</td>
<td>35421</td>
<td>17635</td>
</tr>
<tr>
<td>Sort</td>
<td>66670</td>
<td>33020</td>
</tr>
<tr>
<td>Mul</td>
<td>87381</td>
<td>43912</td>
</tr>
</tbody>
</table>

**Table 2. Execution time of tasks (unit: milliseconds)**

<table>
<thead>
<tr>
<th></th>
<th>1 Core</th>
<th>2 Cores</th>
<th>4 Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fib</td>
<td>6300</td>
<td>5895</td>
<td>4496</td>
</tr>
<tr>
<td>Sort</td>
<td>9788</td>
<td>8772</td>
<td>6857</td>
</tr>
<tr>
<td>Mul</td>
<td>11302</td>
<td>10129</td>
<td>7994</td>
</tr>
</tbody>
</table>

![Chart](image)

**Figure 6. Execution time of tasks**
Experiment environment: Intel Core i7 Q720 1.6GHz, Linux 2.6.32, JDK 1.7. We use VMware Workstation 8.0 as the running environment to configure the number of cores. Table 1 shows that subtasks are almost evenly distributed to cores. From Table 2 we can see that with the number of cores increasing execution time of each type of task decreases, but it does not have a strictly inverse relationship with number of cores. According to Figure 6, by comparing the task size, it shows that the larger the task size is, the model achieves a greater utilization of multicore processors which get a higher percentage of time reduction. Therefore we suggest applying the model when problems are large enough to make the most of multicore processors.

5. Conclusion

This paper proposes a multicore load balancing model based on Java NIO framework. The model uses Java NIO to keep CPU from waiting simultaneously, and task parallelization and serialization to utilize multicores resources. The vital task scheduling algorithm bases on Fork/Join framework, and focuses on parallel task scheduling while leaving task division to the framework. By experiment, the algorithm is tested and proved to be feasible and efficient. Although the paper is Java-based, the model is fit for other languages, provided frameworks with similar functions are available.

References