Parallel Design Patterns for multi-core using Java
Jayshree Chaudhari
Student of ME (Computer Engineering), Shah and Anchor Kutchhi Engineering College, Mumbai, India.

ABSTRACT
Multi-core have become a norm, but software deployed on such systems are not suited to exploit the power available at their disposal. This paper presents how the parallel programming design patterns can be adapted for performance benefits on multi-core CPUs. Java 7 provides only Fork and Join parallel design pattern, which is efficient for recursive programming. But other scenario like simple loops, dynamic load balancing, scientific calculations, and other parallel design patterns proves to be handy. I have implemented design patterns not available in Java 7, as API's and compared core utilization and time taken with respect to corresponding sequential implementation for Monte Carlo algorithm.

Keywords: Parallel Design Pattern, Single Process Multiple Data, Master Worker, Loop Parallelism, Shared Queue, Distributed Array.

I. INTRODUCTION
Up to early 2000’s hardware manufacturers were successful in doubling clock frequency of micro-processors every 18 months in line with Moore’s law. Micro-processor manufacturers realized need of multi-core processor due to increasing gap between processor and memory speed, the limit of instruction level parallelism as well as high power consumption [1, 10].

Multi-core micro-processor delivers high performance through multi-processor parallel processing. Running the sequential program of the job only gives the performance on single core, leaving the other core on the chip wasted [10]. To optimize use of multi-core processor, program needs to be re-written in parallel using parallel programming. Parallel programming is a methodology of dividing a large problem in to smaller ones and solving smaller problems concurrently [11].

The initial response to the multi-core was parallel programming using multi-threaded programming. However, writing parallel and concurrent programs in the multi-threading model is difficult. Parallel design patterns rescues programming community from intricacies of writing a parallel program and allows focusing on business logic.

Until Java 5, writing a parallel program was writing multi-threaded program, which was painful as development community had to deal with thread synchronization and pitfalls of parallel programming. Java 5 and Java 6 provided java.util.concurrent packages, which introduced set of API’s providing development community a powerful concurrency build blocks. Java 7 further enhanced support for parallel programming by providing Fork and Join Design Pattern API.

Other powerful programming design patterns like SPMD, Master Worker, Loop Parallelism, Distributed Array and Shared Queue are still not present [2]. I have implemented this design patterns as API and compared time taken and core utilization compared with corresponding sequential implementation for Monte Carlo algorithm.

II. PARALLEL DESIGN PATTERN
Design patterns are a recurring solution to a standard problem. Programming design patterns are modeled on successful past programming experiences. These patterns once modeled, can then be reused in other programs. Typically, hand coding a program from scratch i.e. multi-threaded results in better execution time performance, but may consume immense amounts of time and errors [9].

Design patterns for parallel programming makes life of programmer easy by addressing to recurring problems like concurrent access to data, communicating between parallel programs, synchronization, dead locks etc. A parallel design pattern includes Single Program Multiple Data (SPMD), Master-Worker, Loop Parallelism, Shared Queue and Distributed Array.

A. SPMD & Distributed Array
In SPMD Design pattern, all Unit of Executions (UEs) execute a Single Program in parallel each having its own set of data and unique identifier. SPMD pattern is mostly used in scientific applications having complex compute intensive algorithms, which needs to be executed in parallel without changing the program. The data might be different between UEs, or slightly different computations might be needed on a subset of UEs, but for the most part each UE will carry out
similar computations. Distributed Array decomposes the data either vertically or horizontally into number of sets [1, 3].

Figure 1 shows high level diagram for implementation of SPMD along with Distributed Array. Master process decomposes and distributes data across predefined number of UEs (4 UE’s considered in figure 1) and assigns unique identifier for each UE to differentiate behavior. Data is decomposed among four UE’s using Distributed Array pattern. The results of UEs are combined by the master process.

**Figure 1: Single Program Multiple Data & Distributed Array Pattern [1]**

**B. Master Worker & Shared Queue**

A Single master process sets up a pool of worker processes and a bag of tasks. Workers execute concurrently by repeatedly removing a task from bag of tasks and processing it. Shared Queue shares data efficiently between concurrently executing UEs. Master Worker design pattern dynamically balance the work on a set of tasks among the Unit of Executions (UEs) [1].

Figure 2 shows high level diagram for Master Worker and Shared Queue. Master process will create a bag of task using Shared Queue and will create predefined number of UEs (4 UE’s are shown in figure 2) which will act as workers [6]. This UEs/Workers removes task repeatedly from Shared Queue one by one concurrently and process it till the queue is empty or termination condition is reached. Master will then combine the results.

**Figure 1 : Master Worker & Shared Queue [1]**
C. Loop Parallelism

Loop Parallelism transforms compute intensive loops of sequential program into a parallel program, where-in different iterations of the loop are executed in parallel. In most of the languages, loops are executed sequentially. If numbers of iterations are high and compute intensive, then time taken is higher and core utilization is poor. In such cases it makes sense to execute iterations in parallel threads on multiple UEs [1, 5].

Figure 3 shows the high level diagram of Loop Parallelism, here the iteration of for loop are divided equally among three UEs. Numbers of UEs are predefined before the execution starts.

![Loop Parallelism Design](image)

III. RELATED WORK

- In the book “Pattern of Parallel Programming” by Timothy G. Mattson and Beverly A. Sanders and Berna L. Massingill introduced a complete, highly accessible pattern language using design spaces that helps any experienced developer “think parallel” and write effective parallel code. Design spaces provides development community medium to write complex parallel program using parallel design patterns. Supporting structure design space defines patterns like Master Worker, SPMD, Loop Parallelism, Distributed Array and Shared Queue/Data [1].
- Microsoft .NET framework 4 included Parallel Pattern Library (PPL), Task Parallel Library (TPL) and Parallel Language-Integrated Query (PLINQ), which are commonly used with C++ and C#. On top of these libraries .NET 4 frameworks added design patterns like Parallel Loop, Fork and Join, Shared Data and other patterns like Map Reduce and Producer Consumer [7].
- Java added Fork and Join design pattern build on top of executor framework in its version 7[8].

IV. METHODOLOGY

Water fall development model is used to develop API in Java for design patterns like Master Worker, SPMD and Loop Parallelism [4]. Development is done on top of existing Java concurrency API and using eclipse development environment. For testing API test classes are created along with corresponding sequential implementation. CPU Core utilization is monitored using Window task manager.

Following environment was considered for testing the patterns:
- Processor Details : Intel® Core™ i5-4310U CPU @ 2.00GHz
- No of Cores : 4
- RAM : 4 GB
- OS : Windows 7 Service pack 1 64-bit
- JVM: JDK 7
- CPU Utilization: Windows 7 Resource monitor
- Algorithm: Monte Carlo

This test can also be carried out on other operating system which can host Java Virtual Machine (JVM) and having multi-core CPU.
V. OBSERVATION OF DIFFERENT DESIGN PATTERN

### TABLE 1: TIME TAKEN BY DESIGN PATTERNS

<table>
<thead>
<tr>
<th>Number of UE’s</th>
<th>Fork and Join</th>
<th>SPMD &amp; Distributed Array</th>
<th>Master Worker &amp; Shared Queue</th>
<th>Loop Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>20.122</td>
<td>22.241</td>
<td>21.540</td>
<td>22.307</td>
</tr>
<tr>
<td><strong>4</strong></td>
<td><strong>8.836</strong></td>
<td><strong>8.814</strong></td>
<td><strong>8.884</strong></td>
<td><strong>9.046</strong></td>
</tr>
<tr>
<td>6</td>
<td>9.45</td>
<td>9.110</td>
<td>9.6</td>
<td>9.552</td>
</tr>
</tbody>
</table>

Figure 4: Core Utilization Sequential  
Figure 5: Core Utilization Fork & Join (4 UE’s)  
Figure 6: Core Utilization SPMD (4 UE’s)  
Figure 7: Core Utilization Master Worker (4 UE’s)
Following are observations based on test results:

- Sequential program mostly uses one core of CPU as shown in figure 4.
- With increase in number of UE’s cores utilization improves and time taken reduces as observed in table 1.
- Approx. 60% reduction in time taken by design patterns compared with corresponding sequential implementation as shown in table 1.
- Time taken by design patterns reduces drastically up to 4 UE’s and thereafter starts increasing slowly.
- Master Worker & Shared Queue, SPMD & Distributed Array and Loop Parallelism API developed are having comparable time reduction as observed in table 1 and core utilization compared with existing Fork and Join on same machine as shown in figures 5, 6, 7 and 8.
- Time taken for execution differs each time program is executed for both sequential and Patterns API developed.

VI. CONCLUSIONS

Sequential Programs are not able to take advantage of advancement in hardware architecture of multi-core. To take advantage of multiple cores, programs need to be written in parallel by writing multi-threaded program. Writing a multi-threaded program brings with it its own challenges like concurrent access to data, communicating between parallel programs, synchronization, deadlocks etc. Design pattern addresses challenges associated with writing multi-threaded program.

The observations show that all the API are getting approximate 60% reduction in time taken and 100% cores utilization compared with equivalent sequential program for Monte-Carlo algorithm. Increase in number of processing units the core utilization better till number of UEs are equivalent to number of cores, after that the time taking increases as there is overhead for core swapping between UEs.

In Future these patterns can be extended to other programming languages. This pattern can also be used as distributed parallel programming.

REFERENCES


