CEJVM: “Cluster Enabled Java Virtual Machine”

M.U. Janjua, M. Yasin, F. Sher, K. Awan, I. Hassan
Faculty of Computer Science & Engineering, GIK Institute, Topi, Pakistan
umar_janjua@yahoo.com, mmyasin@giki.edu.pk

Abstract

CEJVM is a cluster enabled Java Virtual Machine, which executes in a distributed fashion among collaborating nodes of a dedicated cluster. It extends Java’s multithreading mechanism to parallel computing paradigm by transparent migration of independent application threads modeled in master worker paradigm. The goal is to obtain improved performance for computationally-intensive multi-threaded Java programs without modifying traditional JVM’s code, Java language or compiler implementation. Deploying a master worker relationship among the nodes in the cluster, CEJVM monitors, packs, transports and resurrects the Java Threads on “Master JVM” and helper “Worker JVMs”.

Currently, we have created a prototype that runs pure Java applications on local Ethernet based cluster of Win2K computers. We have achieved appreciable speedup for ANN training program written in Java (with independent threads) on CEJVM without any modification to the programs source code.

1. Introduction

Java Virtual Machine (JVM) [4] is a stack-based software-computing machine, which understands and executes a defined set of platform neutral instructions called “Byte Code”. A Java language compiler converts the Java source code into the “Byte Code”, which can be interpreted and executed over any other platform where JVM interpreter is available. Such “Byte Code” interpretation makes a Java program independent of specific platforms.

Java language allows the programmers to initiate multiple paths of execution inside the program, called as Java Threads. On a single processor platform, multiple Java threads share the same processor by executing in different scheduled time slices. The multi-threaded Java application can obtain improved performance when one thread is involved in I/O bound task, and other proceeds for CPU bound task. For a multi-processor platform, the benefit is obvious as each thread can run its individual task on individual processor and thus the total time for multi-threaded java applications would be less. There are various multiprocessor platforms ranging from on-board tightly coupled SMP machines to loosely coupled clusters. By cluster, we mean a collection of computers, which collaboratively perform an intensive task by sharing individual resources (memory, file system, processors, I/O space) over the network. We focus on dedicated clusters of uni-processor machines where constituent nodes perform the assigned computationally intensive task only.

This paper presents our approach in leveraging JVM’s multi-threading mechanism to cluster computing paradigm. We provide a Java programming platform, which takes advantage of the cluster to improve the performance of computationally intensive multi-threaded application by transparently migrating Java threads over different nodes in the cluster. At present, we do not address the traditional problems of data synchronization, and load balancing among dynamically changing number of threads across cluster. A comprehensive survey on the above issues by Nuttall [10] can be referred.

Currently, there are different approaches that extend Java for parallel computing paradigm. We categorize them in two groups- firstly, those which have modified traditional JVM or Java compiler, and secondly, those which have not done any modification but rely on top level framework libraries. CJVM by Yariv Ardor [1] has altered the implementation of the JVM by changing its "Byte Code" and class loading. This approach has the downside that it is specific to particular JVM implementation and hence not portable. Another approach described as JESSICA [2] employs a Distributed Shared Memory in addition to changing JVM implementation. JPVM [5] provides framework-oriented solution, which requires programmers to use specific libraries. RMI [6] requires an extra overhead on java programmers to explicitly mange remote servers and clients and marshal objects.

2. CEJVM’s Approach

CEJVM enables the cluster of traditional JVMs to provide a software platform for parallel execution of Java application threads on individual nodes. Each node in the
cluster has traditional JVM deployed on it, where only one node impersonates the Master role and others do the Worker roles. The Master node, over which the original Java application is executed, controls and directs the Worker nodes in the cluster. The Worker nodes collaborate with Master node by receiving Java threads from the Master node, executing them over the traditional JVM, and transporting them back to Master on completion.

We do not provide complete Single System Image SSI proposed by KaiHwang and Hai Jan [3] of the cluster to the Java application. On the other hand, we aim to provide thread transparency- whereby threads created by application migrate and execute on other nodes in cluster without requiring Java application’s explicit intervention or Java programmers’ knowledge.

In the next sections, we discuss how CEJVM transparently leverages traditional JVM’s multi-threading to cluster computing paradigm, what programming paradigm is offered to the java programmer and how CEJVM sets apart from contemporary approaches.

2.1. Programming Paradigm

CEJVM offers master/worker parallel programming paradigm to the java programmer, where the master thread distributes the computationally intensive tasks into smaller sub-tasks among worker threads, then waits for the workers threads to complete the allocated tasks, and finally, accumulates the result from sub-results of workers at their completion.

![Multithreaded Java Application](image)

Currently, our CEJVM implementation supports mutually independent worker threads i.e. except for the data given at the start from the Master, they do not depend (no data sharing and messages during run method execution) for the completion of the task assigned to them. We have conveniently modeled several applications from the field of neural network training, digital image processing and matrix operation according to the paradigm. (See section 3.)

2.2. Thread Migration

To transparently migrate Java threads, CEJVM uses two standard technologies— JVMDI [7] and JNI [8]. JVMDI is a two-way programming interface which notifies its clients of interesting occurrences inside JVM executing application in the form of events like “Thread Creation”, “Method Entry”, “Method Exit”, “Attribute Modification” etc. The JVMDI client can also query and control the application through many different functions either in response to events or independent of them. CEJVM incorporates its sub-components as in-process JVMDI client with traditional JVM running original Java application. In this way, CEJVM receives thread specific events with minimal intrusion. Out of a large set of possible notification events, we listen only to selected thread specific events like “Run method entry”, “Run method exit”. In this way, the number of interrupts to the application, as well as the corresponding overhead is minimized.

JNI is a standard interface, which allows creating, manipulating and updating Java Objects inside JVM through a language other than Java like C. It is through this interface that CEJVM creates Java Threads, and accesses or updates their member fields with in traditional JVM, both on Worker and Master node.

2.2.1. Migration Level. We envisage Java Thread object existing at three levels of abstraction-machine, JVM, and application level. At the machine level, the thread context comprises machine stack, register values and memory pages. Thread migration at such a level will introduce operating system and machine specific dependencies of the like of process structure and register architecture. At JVM level, through JVMDI [7], we can retrieve stack contents (java stack frames, instruction pointers, etc), which constitute Thread Context, from the running JVM. At the application level—the approach followed by CEJVM, we propose to capture the thread object instance by copying all its fields through JVMDI, packing it and transporting it to the destination node, where it is resurrected as a thread instance by JNI method call, and resumed again. This approach has the advantage that it is independent of the underlying platform architecture and interoperable with all JVM implementations that support JVMDI and JNI.

In order to meet the above challenge, the CEJVM establishes following thread management mechanism.
2.2.2. Thread Detection. When a java application creates a Java Thread Object, it is not instantly executed unless its \texttt{.start} method is called, which in turn calls \texttt{.run} method. The thread object exists as a normal Java object, which may receive messages from other objects. After invoking the start function, the thread object instance, depending on the implementation of JVM, is mapped to an underlying system’s thread. Through JVMDI interface, CEJVM monitors the invocation of run method for threads created inside the JVM. When Method Entry event for the run method is notified, the Master node considers that thread to be a potential candidate for migration. On the other hand, at the worker node, when the “Method Exit” event for the run method is notified, the Worker transports it back to the Master node.

2.2.3. Thread Selection. Along with worker and master threads created explicitly by the java programmer in the application, JVM also contains other system threads running as daemons. These system threads include “garbage collector”, which runs intermittently to de-allocate unreferenced objects from the memory, and the “signal dispatcher”, which manages signals among the threads. Another is the JVM loader thread that calls the java application's “public static void main” function. Also included are CEJVM specific threads running in the process space of the JVM. These system threads are not migrated by CEJVM, as there is no benefit at all. Instead, keeping them at the respective JVM where they are created, CEJVM benefits from the automatic garbage collection. CEJVM differentiates between system threads spawned by the JVM and the application threads launched by the programmer. Only application worker threads at Master node are migrated from the master node to the Worker node and vice-versa.

2.2.4. Thread Distribution. This mechanism uses a simple load balancing technique. After a particular interval of time, the Master node periodically determines the number of threads executing at Worker nodes and maintains a count per worker. When a Java thread becomes candidate for migration at Master node, it is transported to the Worker node that has least number of Java threads.

2.2.5. Thread Packing and Unpacking. Once a particular Java thread is identified as a candidate for migration, we pack the thread in a special format called as Thread Packed Format (TPF). We define TPF as a collection of member objects, and member primitive fields of the Java thread instance. For primitive fields, we store the member’s type and its value directly. For a member object field, the member objects’ member fields are stored. The thread packing algorithm works in a recursive manner on Java Thread instance. Given its thread ID, it moves along its inheritance hierarchy, if any, from the base “\texttt{java.lang.Thread}” class to the derived class. At each level, it stores the member fields and member objects. Finishing with higher level super class, it descends down copying objects and primitive fields till it reaches the instance’s thread class, where finally the TPF for the given thread instance is manufactured.

Whenever a thread object is migrated, a global identifier is created which is uniform across all traditional JVMs. It is through this global Thread ID that we recognize and associate the copies of that particular Thread Object at the Master node as well as Worker Node. In Java, objects are accessed through reference variables. It is possible that the different references may point to the same java object allocated in the memory and hence, the same java object may be packed more than once. CEJVM employs special checks to ensure that, despite multiple references to the same object, the thread packing algorithm does not copy one object more than once. Such checks are based on the fact that each java object, allocated in the JVM memory, has a unique object ID.

The Thread Unpacking algorithm works the opposite way. Given the TPF, it resurrects the Java thread object by copying all its fields in a new Java Thread object at the destination node. The present implementation packs Java Threads created as subclasses of the “Thread” class, not the one created by implementing “Runnable” interface.

2.3. Comparison with other works

Unlike cJVM [1] and JESSICA [2], we have not modified the implementation of JVM. Contrary to our migration at the application level, JESSICA migrates threads at the JVM level in terms of java stack frames and instruction pointers. Neither do we convert java’s serial code into the parallel code form as done by Aart J.C. Bik and Dennis [9] by restructuring Java compiler, nor we introduce new parallel constructs at the Java language. Also, we do not enforce the usage of any specific library framework package classes like JPVM [5] and RMI [6]. Our approach is unique in the sense that neither it modifies the JVM implementation nor it makes the JVM, the application and the programmer aware of thread migration. The same multi-threaded Java program that runs on CEJVM can unchangeably run on traditional JVM.
3. Performance Evaluation

In this section, we present our experiments to verify and demonstrate CEJVM approach. We have obtained substantial speedup gains for computationally intensive Java applications. Also, we report the costs and overheads incurred and added by CEJVM and have found them negligible for computationally intensive multi-threaded Java applications. We have tested the applications on a cluster consisting of three nodes (one master, two workers) 300Mhz Pentium computers with 128 MB RAM interconnected through a switch over 10Mbits connection line. Each computer had a Java 1.3 SDK installed on it.

Clearly, the CEJVM approach incurs overheads when it distributes threads over the cluster. Thread Pack time (TPt) is cost of time incurred by CEJVM in retrieving the Thread Object from JVMs memory address space and transforming it to Thread Packed format. Thread Transport time (TTt) is the time incurred in transporting TPF from one node to other. Hence, the total thread migration time of transferring the thread from one node and resuming it at the target node is the sum of packing thread on original node, transporting it over the network, unpacking it inside the target JVM address space and resuming it. Also, since CEJVM management daemon thread run in the same process space of the traditional JVM, they introduce their own administrative overheads. Since, we are targeting computationally intensive multi-threaded Java applications where worker threads spend most of the time in computational work, such overheads become negligible and do not nullify or minimize the speedup obtained by the distributed execution of threads in parallel over individual nodes in the cluster.

Fig 2 shows a linear relationship between Thread object size with respect to TTt and TPt. The thread object size is normalized with respect to the size of the empty Thread Class instance. We increase the size of the thread objects by adding primitive data types and objects of other classes as member variables inside its class declarations.

4. Conclusion and Future work

The paper presented an approach to improve the performance of computationally intensive multithreaded Java application by transparent migration of threads in a cluster where independent threads execute in parallel. We have concluded that using JVMDI and JNI, the threads can be migrated transparently without changing JVM, Java language, Java compiler or imposing third-party library frameworks. CEJVM is interoperable with all JVM implementations that support JVMDI and JNI.

Having completed the independent worker thread migration, the next step is to leverage thread synchronization primitives so that different threads across cluster can handle shared data correctly. This involves intercepting the traditional JVM’s synchronization primitive “monitor” calls through JVMDI and extending it for cluster environment. Also, we would like to relieve CEJVM of master-worker paradigm dependency in order to benefit the class of multithreaded applications modeled in peer-to-peer relationship. The ability of the CEJVM to handle dynamically changing threads would involve implementing an intelligent dynamic load balancing scheme, where Worker JVM can also migrate threads to other nodes.

5. References


[9] “Automatically Exploiting Implicit Parallelism in Java” Aart J.C. Bik and Dennis B. Gannon, Lindley Hall Computer Science Dept, Indiana University, Bloomington, Indiana ,USA