M-JavaMPI: A Java-MPI Binding with Process Migration Support

Ricky K.K. Ma, Cho-Li Wang, and Francis C.M. Lau
Department of Computer Science and Information Systems
The University of Hong Kong

Presented by: Cho-Li Wang
Outline of Presentation

Introduction
- Why Java MPI Binding?
- Our Research Objectives
- Our Approach
- Java Virtual Machine Debugger Interface

M-JavaMPI System Architecture
- Java Process State Capturing and Restoring
- Restorable MPI Communication

Performance Evaluation

Related Works

Conclusions and Future Works
Introduction: Why Java+MPI?

Java

- Emerging as a major language for distributed and parallel programming.
- Almost for all platforms: Sun’s J2SE, J2EE, J2ME.
- But...Client-Server Model, No SPMD
  - Sockets and the Remote Method Invocation (RMI)
  - Both communication models are optimized for client-server programming, whereas the parallel computing world is mainly concerned with "symmetric" (peer-to-peer) communication, occurring in groups of interacting peers.
Introduction: Why Java+MPI?

Message Passing Interface (MPI)

- **Standard message-passing communication library** (Has been implemented on many parallel machines).
- Directly supports the **Single Program Multiple Data (SPMD)** model of parallel computing.
- **Natural model** on distributed-memory machines such as clusters.
- Possible to do **special problem partitioning, initial assignment of application data to machines, and intelligent runtime data movement** to achieve high performance.
Java MPI Binding: Existing Solutions

Direct bindings to the native MPI library

- mpiJava [Baker, et. al., 1998] :
  - through JNI wrappers to native MPI software

- JavaMPI [Mintchev: 1997] :
  - through JNI wrappers to native MPI software
  (wrappers were automatically generated by a special-purpose code generator)
Java MPI Binding: Existing Solutions

- MPI libraries entirely written in Java.
  - JMPI : [MPI Software Technology : 1997]
  - Jmpi : [Dincer: 1998]
  - MPIJ : [DOGMA project : 1999]
  - PJMPI: [Tong et. al.: 2000]
  - MPJ : [MPI Software Technology : 2000]
Discussion: Java MPI Binding

Direct Java-MPI binding
- (O) Efficient MPI communication through calling native MPI methods
- (X) Low-level conflicts between the Java runtime and the interrupt mechanisms used in MPI implementations

Pure Java implementation
- (O) Provides a portable MPI implementation
- (X) MPI communication is less efficient
Our Research Objectives

Application Fault-tolerant:
- Many scientific applications run for a very long time (days or even months at a time).
- System failures (e.g., hardware or network failures) can be expected to occur during the run of applications.
- The system aborts the job early because of a planned downtime.

Dynamic Load Balancing:
- Computation patterns of irregularly structured problems can not be expected in the algorithm design phase.
- Most programmers are lack of skills to design efficient algorithms in message passing programming.
- Time-shared computing environment.
Our Approach

- **Fault Tolerance and Dynamic Load-balancing**
  - Transparent Java process migration without programmer’s involvement or modification of their codes.
  - Automatic message redirection and communication handoff

- **High Portability**
  - No modification of OS, JVM, and MPI
  - Java Virtual Machine Debugger Interface (JVMDI)

- **Efficient Messaging Support (MPI) for Java**
  - Minimize the overheads for binding MPI with Java
  - Avoid low-level resource conflicts between MPI and JVM
Java Virtual Machine Debugger Interface (JVMDI)

- **Standard interface for JVM:**
  - Define standard services that a JVM must provide for debugging.
  - Available since Java 2.

- **Enough support to capture Java process state:**
  - Able to obtain runtime information of threads, stack frames, local variables, classes, objects and methods.
  - It can be used to control threads, set local variables, receive notification of events.
M-JavaMPI Overview

Java Debugger Interface (JVMDI)
- Used to capture execution context
- Eager(all) strategy to reduce residual dependency

Object serialization
- Java process context is saved in a platform-independent format

Exception handler inserted at pre-processing
- Cope with the migration layer to restore the processes

Client-server based Java-MPI interface
- Provides restorable MPI communications
A Layered View of M-JavaMPI

- **Java MPI program** (Java bytecode)
  - Preprocessing layer
    - (Insert exception handlers)
  - Java-MPI API
  - Java API
- **Migration layer**
  - (Save and restore process)
- **JVMDI** (Debugger interface in Java 2)
- **JVM**
- **Restorable MPI layer**
- **Native MPI**
- **OS**
- **Hardware**

- **Bytecode rearrangement and introduction of special local variables**
- **Provide MPI wrapper for Java program**
- **Save and restore process. Process and object information are saved and restored by using object serialization, reflection and exception through JVMDI**
- **Java .class files are modified by inserting an exception handler in each method of each class. The handler is used to restore the process state.**
- **Debugger interface in Java 2. Used to retrieve and restore process state**
- **Provide restorable MPI communication through MPI daemons**
- **Support low-latency and high-bandwidth data communication**
Migration Granularity

At the Java source code level

Migration can only happen after the complete execution of all Java bytecode corresponding to a single Java source code line.

- Migration is postponed until the end of the executing Java source line
- Similarly for a migration request that is received in the middle of the execution of a native method
State Capturing and Restoring

1. **Program code**: re-used in the destination nodes.
2. **Data**: captured and restored using the object serialization mechanism.
3. **Execution context**: captured by using JVMDI and restored by the exception handlers which are inserted during the pre-processing of bytecode.

**Eager(all) strategy**: For each frame, local variables, referenced objects, the name of the class and class method, and program counter are saved using object serialization
public class A {
    int a;
    char b;
    ...
}

public class A {
    try {
        ...
    } catch (RestorationException e) {
        a = saved value of local variable a;
        b = saved value of local variable b;
        pc = saved value of program counter when the program is suspended
        jump to the location where the program is suspended
    }
}
Restorable MPI Layer

- MPI daemon run on each node of the cluster to support message passing between distributed java processes.
- IPC between Java program and MPI daemon in the same node is done through *shared memory* and *semaphores*.
LEGENDS

- Migration events
- Event triggers

**Source Node**

- suspend user process
- capture process state
- JVM and process quit
- notify MPI daemon of the completion of capturing

- send migration request

**Destination Node**

- MPI daemon (source node)
- MPI daemon (destination node)
- migration client (destination node)

- start an instance of JVM with JVMDI client
- send notification of the readiness of captured process data
- Sending notification of the readiness of captured process data

**Process migration steps**

1. Suspend user process
2. Capture process state
3. JVM and process quit
4. Notify MPI daemon of completion of capturing
5. Send migration request
6. Broadcast migration info. to all MPI daemons
7. Send buffered messages
8. Notify MPI daemon of completion of capturing
9. JVM and process quit
10. Start an instance of JVM with JVMDI client
11. Send notification of the readiness of captured process data
12. Send notification message (and captured process data if central file system is not used)
13. Process is restarted and suspended

**Restoration of execution state**

- Migrated process is restored
- Source Node
- Destination Node
Performance Evaluation

Experimental Setting

- PC Cluster
  - 16-node cluster
  - Pentium II 300 MHz with 128MB of memory
  - Linux 2.2.14 with Sun JDK 1.3.0 + MPICH
  - Connected by 100Mb/s fast Ethernet

- All Java programs were executed without JIT compilation mode enabled
Bandwidth: PingPong Test

- Native MPI (C+MPI): 10.5 MB/s
- Direct Java-MPI binding: 9.2 MB/s
- Restorable MPI layer: 7.6 MB/s
Latency: PingPong Test

- **Native MPI (C+MPI):** 0.2 ms
- **Direct Java-MPI binding:** 0.23 ms
- **Restorable MPI layer:** 0.26 ms
Migration Cost: capturing and restoring objects

Minimum Overhead:
- Capturing: 54 ms (JVMDI)
- Restoring: 1 ms (Reception)
Migration Cost: capturing and restoring frames

(Empty Java frame: No local variables are defined in each frame)
Performance Evaluation

Application Performance

- **PI Calculation** (computational intensive)
- **Recursive ray-tracing** (computational intensive)
- **NAS integer sort** (comp. + comm. intensive)
- **Parallel SOR** (comp. + comm. intensive)
Execution Time of PI and Ray-tracing with and without migration layer
(Debugging Mode vs. Interpretation Mode; Binding Overhead)
## Execution time of NAS program with different problem sizes

<table>
<thead>
<tr>
<th>Problem size (no. of integers)</th>
<th>Time (sec) : Direct Java-MPI Binding (Interpretation Mode)</th>
<th>Time (sec) : M-JavaMPI (Debugging Mode)</th>
<th>Slowdown introduced by M-JavaMPI (in %)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Comp</td>
<td>Comm</td>
</tr>
<tr>
<td>Class S: 65536</td>
<td>0.023</td>
<td>0.009</td>
<td>0.014</td>
</tr>
<tr>
<td>Class W: 1048576</td>
<td>0.393</td>
<td>0.182</td>
<td>0.212</td>
</tr>
<tr>
<td>Class A: 8388608</td>
<td>3.206</td>
<td>1.545</td>
<td>1.66</td>
</tr>
</tbody>
</table>

No noticeable overhead introduced in the computation part when running in **debugging mode** using 2 nodes; while in the communication part, an overhead of about **10-20%** was induced.
Execution time of SOR

Execution time of SOR on an array of size 256x256 (Process migration adds extra 2-3 sec).

<table>
<thead>
<tr>
<th>No. of nodes</th>
<th>No migration (sec)</th>
<th>One migration (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1013</td>
<td>1016</td>
</tr>
<tr>
<td>2</td>
<td>518</td>
<td>521</td>
</tr>
<tr>
<td>4</td>
<td>267</td>
<td>270</td>
</tr>
<tr>
<td>6</td>
<td>176</td>
<td>178</td>
</tr>
<tr>
<td>8</td>
<td>141</td>
<td>144</td>
</tr>
</tbody>
</table>

• Execution time of SOR using different numbers of nodes with and without migration layer (No Migration)
**Average Process Migration Time**

<table>
<thead>
<tr>
<th>Applications</th>
<th>Average migration time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PI</td>
<td>2 sec</td>
</tr>
<tr>
<td>Ray-tracing</td>
<td>3 sec</td>
</tr>
<tr>
<td>NAS</td>
<td>2 sec</td>
</tr>
<tr>
<td>SOR</td>
<td>3 sec</td>
</tr>
</tbody>
</table>

**Mainly dominated by the startup time of JVM and loading time of the Java process in the destination node**
Dynamic Load Balancing

A Simple Test:

- SOR program was executed using 6 nodes with one of the nodes executing a computationally intensive program.

- Without migration: 319s.

- With migration: 180s.
Related Works: Fault Tolerance Support for MPI

- **CoCheck MPI** (Technische Universität München):
  - Restart the virtual machine every time a node failure occurs (Heavy Penalty)

- **MPI-TM** (Mississippi State U.):
  - MPI with task migration.

- **LA-MPI** (ACL at LANL):
  - Support end-to-end network fault-tolerant message passing without aborting the application.

- **MPI/FT** (MPI Software Technology: 2000)
Related Works: Java Process/Thread Migration

- **JESSICA (HKU:1999):**
  - Java Thread Migration in interpretation mode. Modification of JVM.

- **JESSICA2 (HKU:2002):**
  - Java Thread Migration in JIT compiler mode. Modification of JVM.

- **MERPATI:**
  - Entire run-time information of the Java virtual machine (JVM).

- **Checkpointing Java (University of Tennessee):**

- **Jthread (Utah):**
  - Thread migration based on the Voyager framework.

- **Mobile Agent related:** Brakes, JavaGo, Class File Translation (U. of Tokyo), MOBA, MobileThread (Inria), etc.
Java Process/Thread Migration

**Bytecode instrument:**
- Insert code into programs, which can be done manually, or via some pre-processors.

**JVM Extension:**
- Make thread state accessible from Java programs. Non-transparent to applications. Modifications of JVM are required

**Checkpoint the whole JVM process:**
- Very powerful but heavy penalty

**Modification of JVM:**
- Totally transparent to the applications, efficient but very difficult to implement -- JESSICA and JESSICA2
Conclusions

M-JavaMPI’s Main Features:
- JVMDI is used to capture execution states
- Exception handler is used to restore process state
- Restorable MPI is provided for transparent message redirection and communication handoff.

- Acceptable migration overheads for long-run scientific applications.
- Dynamic load balancing with the support of process migration -- Good for inexperienced programmers
- A good base for achieving fault tolerance
- Simple!! No need to modify OS, JVM and MPI
Future Works

1. M-JavaMPI in JIT compiler mode
2. Develop system modules for automatic dynamic load balancing
3. Develop system modules for effective fault-tolerant supports
4. M-JavaMPI on the Grid ??