# 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*
*Email: {kk1ma, clwang, fcmlau}@csis.hku.hk*

## Abstract

*Several Java bindings to the Message Passing Interface (MPI) software have been developed for high-performance parallel Java-based computing with message-passing in the past. None of them however addressed the issue of supporting transparent Java process migration for achieving dynamic load distribution and balancing. This paper presents a middleware, called M-JavaMPI, that runs on top of the standard JVM to support transparent Java process migration and communication redirection. The middleware allows Java processes to freely and transparently migrate between machines to achieve load balancing, and migrated processes can continue communication with other processes using MPI. The method we use to achieve process migration is to capture execution context and restoring the execution context at the Java bytecode level using the Java Virtual Machine Debugger Interface (JVMDI). Post-migration interprocess communication is enabled via a restorable Java-MPI API. Tests using a 16-node cluster have shown that our mechanism yields considerable performance gain through migration.*

**Keywords:** *process migration, MPI, JVMDI, message passing, M-JavaMPI, load balancing, Java, cluster computing, parallel computing*

## 1. Introduction

The Message Passing Interface (MPI) is a widely adopted communication library for parallel and distributed computing. Although the existing MPI standard specifies language bindings for only Fortran, C and C++, there has been effort to provide MPI also for Java [9,10,11,13,14].

Existing approaches to MPI for Java can be grouped into two main types: (1) native MPI bindings where the some native MPI library is called by Java programs through Java wrappers [9,10,11], and (2) pure Java implementations [13,14]. The native MPI binding approach provides efficient MPI communication through calling native MPI methods. Conflicts could arise on the use of system resources such as signals between the MPI library and the JVM. The pure Java implementation approach on the other hand can provide a portable MPI implementation since the whole MPI library is rewritten in Java, but the MPI communication would be relatively less efficient since Java operates at a higher level.

In this paper, we propose a cluster middleware, called *M-JavaMPI* (“M” stands for “migration”), to support location-transparent Java process communication using MPI. An MPI wrapper is provided to allow Java programs to link to the native MPI library to support efficient message passing among distributed Java processes. Our approach is different from previous solutions, as our implementation follows a client-server message redirection model. This approach makes it possible to avoid conflicts on the use of system resources between the native MPI library and the JVM.

Another important feature of *M-JavaMPI* is its support of preemptive Java process migration. To achieve high performance and robustness in parallel Java computing in distributed environments, process migration is an attractive feature. Such a feature enables dynamic load distribution and balancing. Unbalanced loading has been found to greatly affect the performance of applications. Process migration can also help those long-running applications by relocating them at suitable times to prevent interruption due to system activities or the execution of other applications. It also can help relocate processes closer to data they need to access.

In order to migrate a Java thread or process, essential process context and execution state information need to be copied from the source node to the destination node. Java supports code mobility through platform-independent bytecode, the customizable Java class loader, and the object serialization mechanism [20]. The Java language however does not provide mechanisms for inspecting, saving and restoring Java execution context.

In the past, many different approaches have been studied and developed to capture and restore execution context. For example, execution context can be captured by inserting code into the program, which can be done manually [17,18,21,22], or via some pre-processor [3,4,5,6,23]. These methods tend to incur significant overhead during execution, even when no migration actually occurs. Execution context can also be captured by extending the JVM to make thread state accessible from Java programs. But modification of the JVM can be difficult [1,7,8]. Execution context can also be captured by checkpointing the whole JVM process, which requires some special checkpoint facility [24].

Unlike other existing Java process migration solutions, we use the Java built-in debugging interface, JVMDI, to capture Java process execution state. As JVMDI is a standard interface, our approach is potentially more portable than existing solutions and the implementation is less complex since no modification of the JVM is necessary. To fully support the mobility of the communicating Java processes, the MPI component is designed to allow communication channels to be re-constructed automatically during program execution. We call this a “restorable” MPI layer.

The rest of the paper is organized as follows. In section 2, we show an overview of the proposed middleware. Section 3 discusses our mechanism on saving and restoring execution state of programs. Section 4 describes the restorable MPI layer. Performance results and evaluation are given in section 5. Related works are presented in section 6. Finally, we conclude our work in section 7.

## 2. The M-JavaMPI System Architecture

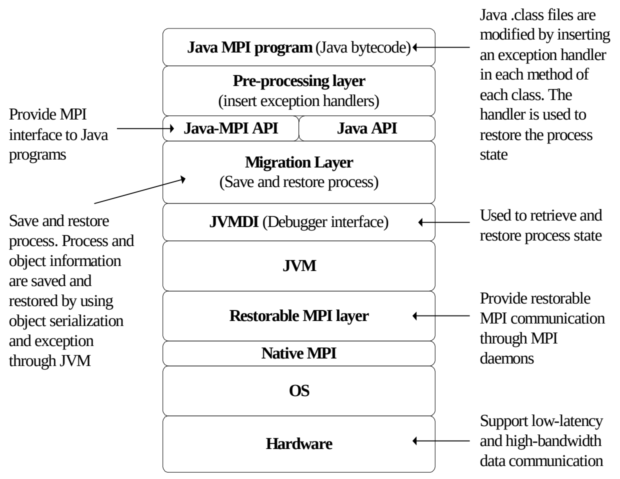
Java MPI program (Java bytecode)
Pre-processing layer (insert exception handlers)
Java-MPI API
Java API
Migration Layer (Save and restore process)
JVMDI (Debugger interface)
JVM
Restorable MPI layer
Native MPI
OS
Hardware
Provide MPI interface to Java programs
Save and restore process. Process and object information are saved and restored by using object serialization and exception through JVM
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
Used to retrieve and restore process state
Provide restorable MPI communication through MPI daemons
Support low-latency and high-bandwidth data communication

**Fig. 1. The layered design of M-JavaMPI**

Fig. 1. shows the layered design of the M-JavaMPI middleware. The middleware consists of several layers, including the *Pre-processing layer*, the *Java-MPI API layer*, the *Migration layer*, and the *Restorable MPI layer*.

The **Pre-processing layer** is used to modify the bytecode of the Java application before the bytecode is passed to the JVM for execution. “Restoration functions” are inserted into the application. These functions react to the migration layer to restore the Java stacks and resume execution during migration. In order to avoid the overhead due to added code during normal execution, these functions are added as exception handlers in the program, in the form of encapsulated “try-catch” statements. These “try-catch” blocks will run only when a *restoration exception* occurs during the restoration of process.

As we do not want to modify the JVM, some special treatment of the Java bytecode is needed. This includes re-arrangement of bytecode during pre-processing and the addition of local variables to avoid having to retrieve operand stacks from the JVM, which is only possible by modifiying the JVM. Details are discussed in Section 3.3.

The **Java-MPI API layer** provides MPI calling interfaces to Java programs. We opted for a modular, client-server design of a message redirection mechanism for migrated Java processes. The Java-MPI API layer acts as a client which sends MPI-related messaging requests to the MPI daemon (a server) in the same node in the **Restorable MPI layer**. The MPI daemon is responsible for delivering messages on behalf of the Java process. Communication channels are re-constructed automatically after migration. This allows Java processes to communicate with each other after migration as if no migration has occurred.

The **Migration layer** performs two main tasks: (1) to capture and save the execution state of the migrating process in the source node, and to restore the execution state of the migrated process in the destination node; (2) to cooperate with the Restorable MPI layer to reconstruct the communication channels of the parallel application.

In M-JavaMPI, the granularity of migration is at the Java source code level. It means that migration can only happen after the complete execution of all Java bytecode corresponding to a single Java source code line, and before the execution of the next Java source code line. If a migration request is received in the middle of executing a Java source code line, the migration will be delayed until the end of execution of the current source code line; similarly for a migration request that is received in the middle of the execution of a native method.

This source-code-level granularity simplifies the design of the migration mechanism by eliminating the need to save operand stacks which are usually non-empty in the middle of the execution of a source line. It also avoids the need to save machine-dependent process state information which is present during the execution of a native method.

One of our design goals is to avoid modifying the JVM, so that the resulting system can then be as portable as any ordinary Java program. In order to achieve this goal, we need to make use of existing Java functions for capturing and restoring process states. In M-JavaMPI, we use the Java built-in interface, JVMDI, to capture Java process execution states. Potential migrating points are set at the address of the first bytecode of any Java source line which can be obtained from any Java classfile by using the debugging interface. To enable this feature, we need only to compile Java programs with the debugging option switched on.

## 3. Process State Capturing and Restoring

Java has provided portable bytecode and dynamic class loading to allow Java programs to be executed in different platforms. Besides, Java offers the *object serialization* mechanism which can store objects in a portable format. This allows objects to be saved and restored across different platforms. Java however does not provide functions for saving and restoring process states. In this section, we discuss our approach to capturing and restoring Java process state information.

### 3.1. Java Virtual Machine Debugger Interface

The Java Virtual Machine Debugger Interface (JVMDI) [19] is a native interface available for the JVM since Java 2, and is used typically by debuggers. It defines the standard services that a JVM must provide for debugging. There are ways to inspect the state and to control the execution of applications. Using JVMDI, we can obtain the runtime information of threads, stack frames, local variables, classes, objects, and methods. In addition, JVMDI can be used to control threads, to set local variables, and to receive notifications of events such as method exit/entry and frame pop-up. JVMDI is called by the JVMDI client running in the same virtual machine as the application program being debugged. The application would run continuously if no debugging requests are received.

### 3.2. State Capturing using JVMDI

We make use of JVMDI to capture process states. This can be done much more easily than other existing approaches. All the actions performed by JVMDI clients are transparent to the application.

When starting JVM, a JVMDI client is started as well. Migration is initiated and carried out by the JVMDI client. When migration is ready to occur, the client suspends the execution of the application. Then it sends a message to the local MPI daemon to notify it of the migration. After that, it inspects and saves all the Java stack frames created by the migrating Java process. For each frame, local variables, referenced objects, the name of the class and the class method, and the program counter need to be saved using *object serialization*. After the saving, the captured data are sent to the destination node.

### 3.3. State Restoring using Exception

Although JVMDI provides functions for inspecting the execution state of a program, there are not enough functions for re-establishing the execution context, such as the frame stack and the program counter. We therefore resort to pre-processing to add restoration capability to the application to interact with the migration layer to perform restoration of parallel Java processes.

During pre-processing, bytecode is modified in two ways: (1) bytecode rearrangement and introduction of special local variables; (2) insertion of restoration functions in the form of exception handlers. As data in the operand stack are JVM-dependent and no functions are provided by JVMDI to extract and rebuild operand stacks, it is hard to capture and restore operand stacks. The approach we take is to do away with the need to save operand stacks. Our design makes sure that all operand stacks are empty at the time of migration. This is achieved through source-code-level granularity and bytecode rearrangement. Note that the operand stack of the current frame is always empty immediately after the completion of the execution of a Java source code line. For the operand stacks of all the frames other than the current frame, bytecode rearrangement is performed to make these operand stacks always empty during migration. Consider the following statement:

```
y = f(x)+g(x);
```

When migration takes place during the evaluation of *g(x)*, the intermediate value, i.e., the value of *f(x)*, is stored in operand stack. This value needs to be captured during migration, and restored after migration. To handle such runtime generated intermediate values, the original Java code is transformed so that these values are saved in some specially created local variables, instead of the operand stack. For this example, the original Java code line is transformed to

```
tmp1 = f(x);
tmp2 = g(x);
y = tmp1+tmp2;
```

The above transformation involves rearrangement of the bytecode and the creation of additional local variables.

Apart from bytecode rearrangement, restoration functions are inserted as exception handlers to cope with the migration layer to perform restoration. Exception handlers are inserted in each of the methods. The exception handlers catch and react to *restoration exceptions*. Inside these exception handlers, local variables of the called methods are pre-set with the saved information, and a "jump" command is issued to branch to the position saved during capturing. To illustrate, here is a program fragment before pre-processing:

```
public class A {
  int a;
  char b;
  …
}
```

After pre-processing, the program fragment becomes:

```
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
  }
}
```

In the destination node, before receiving a notification indicating the completion of the capturing process, an instance of the process would be created. A breakpoint is set at the start of the *main()* function so that when the instance is created, the breakpoint is caught right away. The migration layer in the destination node will wait for the notification of the completion of the capturing process from the MPI daemon.

When the notification is received by the MPI daemon, the MPI daemon will send a notification message to the migration layer. Then the migration layer will throw a Restoration Exception to the newly created instance of the process. The exception is caught by the Restoration Exception Handler where local variables of the method are restored to the saved values. A "branch" command is then performed to jump to the last executed location of the current frame. This action is repeated for each frame of the program until the last frame is re-established. Then the program will execute again from the last executed position.

## 4. Restorable MPI Communication

### 4.1. Client-Server Message Redirection Model

The Restorable MPI layer is based on a client-server model. This layer consists of MPI daemons and a Java-MPI communication API for Java programs. The Java-MPI communication API is the interface for parallel Java processes to send requests to MPI daemons. An MPI daemon runs on each node of the cluster to support message passing between distributed Java processes. The MPI daemon is responsible for sending messages and receiving messages on behalf of the calling Java program in the same node. The Java program and the MPI daemon in the same node communicate through shared memory and semaphores.

In order to provide efficient MPI communication, communication between nodes is done using the native MPI library. Instead of linking the Java program directly with the native MPI library, the native MPI library is linked by the MPI daemon such that MPI communication is used exclusively by MPI daemons in different nodes for their communication. This approach requires no modification of the existing MPI library.

### 4.2. Process Migration in Action

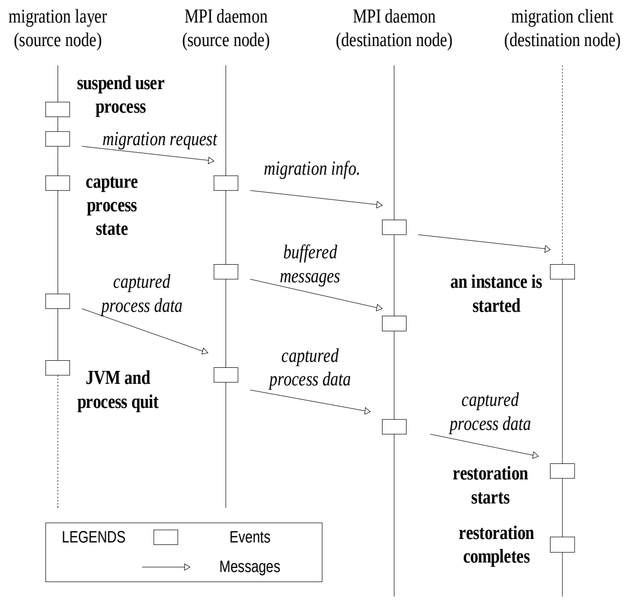

**Fig. 2. Process migration steps in M-JavaMPI**

Fig. 2. illustrates the steps involved in process migration. Initially, MPI daemons are started on all the nodes of the cluster. The MPI daemon in a node covered by the running parallel program will be responsible for all the MPI communication of its MPI client. MPI daemons on the idle nodes will wait for migration requests from MPI daemons of other nodes. Message forwarding will be performed through cooperation of the MPI daemons if a Java client has been migrated.

When a migration starts to occur, the migration layer at the source node suspends the execution of the Java process. It then sends a migration request to its local MPI daemon. Then, it starts capturing the execution state of the migrating process. Captured execution state is saved in a "dump file" in the file server. After that, the migrating process and the JVM, including the migration layer, in the source node stop running. Only the MPI daemon continues to run.

The MPI daemon waits for migration requests, if any, from other MPI daemons, and performs forwarding of messages to the migrated process. Sequence numbers are used to keep messages in order. Message forwarding will only serve those messages that were sent before the migration occurred and have not been received by the migrated process. After migration, when all “old” messages have been received, new messages are sent directly from the source node to the destination node according to the mapping table which records the actual locations of the nodes. No forwarding will take place afterwards. That means no residue dependency will be introduced by the communication.

## 5. Performance Evaluation

Execution of a parallel program can be divided into two parts: computation and communication. The introduction of our middleware may have an impact on the performance of both the computation and communication parts. The computation part could be affected by the state-capturing and state-restoring actions and the use of JVMDI, while the communication part could be affected by the restorable MPI communication mechanism. We evaluated the performance of our Java MPI API, and the state-capturing and restoring mechanism. We also carried out benchmark testing using several Java application programs. We divided our evaluation into three parts: evaluation of the performance of the restorable MPI layer, evaluation of the performance of the state-capturing and restoring mechanism, and evaluation of the performance of the system as a whole.

The experiments were conducted on a 16-node cluster. Each node is a 300MHz Pentium II PC with 128MB of memory, running Linux 2.2.14 with Sun JDK 1.3.0. The nodes are connected by a 100Mb/s fast Ethernet switch. All Java programs were executed in interpreted mode.

### 5.1. Java MPI API

A *pingpong* test was conducted to study the communication performance of the restorable MPI communication layer. In this test, messages of various sizes were sent back and forth between processes. To ensure that anomalies in message timings were minimized, the pingpong was repeated 64 times for each message size.

Fig. 3. shows the communication bandwidth attained for different message sizes using different communication interfaces. The bandwidths of using the native MPI library with a C program, and direct Java-MPI binding with a Java program were also measured for comparison.

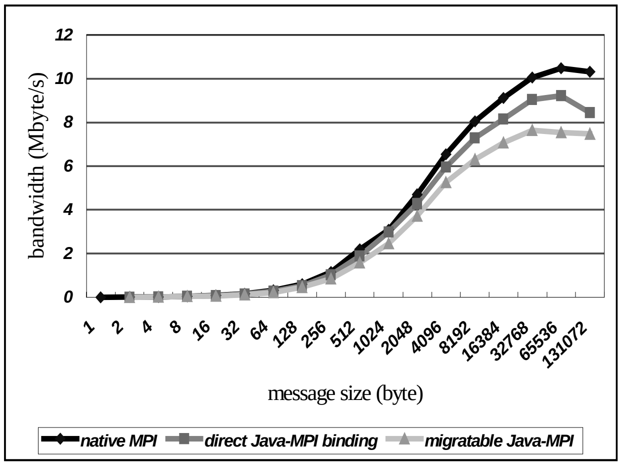

**Fig. 3. Performance comparisons of using different communication mechanisms**

Among the different communication mechanisms, the performance of the native MPI is the best, with a bandwidth of 10.5 Mbytes/s, followed by the performance of direct Java-MPI binding, with a bandwidth of 9.2 Mbytes/s. The peak bandwidth of communication using our restorable MPI layer is 7.6 Mbytes/s, which is 17% less than the direct binding.

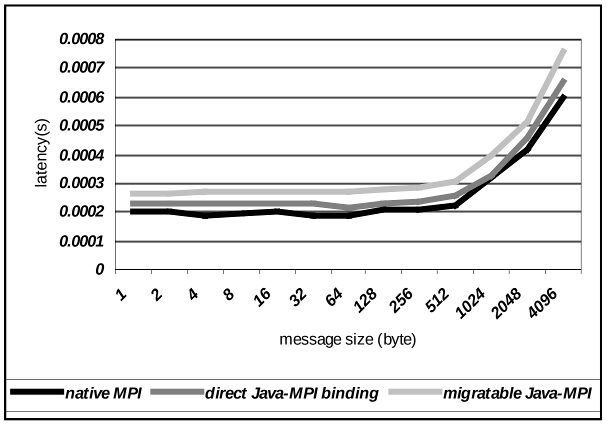

**Fig.4. Comparisons of latencies for small messages using different communication mechanisms**

Fig. 4 compares the latencies of communication for small messages. The minimum latency of the native MPI and the direct Java-MPI binding are 0.2ms and 0.23ms respectively. The minimum latency of the restorable MPI mechanism is 0.26ms.

### 5.2. State-capturing and State-restoring

The migration cost equals to the sum of time spent in capturing state in the source node, the time spent in restoring state in the destination node, and the time spent in starting the JVM and loading the program in the destination node. The time spent in capturing state can be further divided into two parts: time spent in capturing the objects and time spent in capturing the frames. Similarly, the time spent in restoring state can be further divided into two parts: time spent in restoring the objects and time spent in restoring the frames. The times spent in capturing and restoring both objects and frames are shown in Figures 4 and 5 respectively.

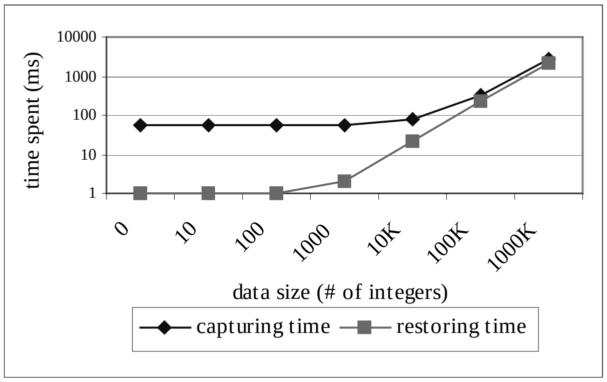

**Fig.5. Time spent in capturing and restoring objects**

Fig. 5 shows the time needed in capturing and restoring objects of different sizes. In this test, objects that were used are arrays of integers. The data size of an integer is 4 bytes. The minimum overheads in capturing and restoring objects are 54 and 1 ms respectively. The capturing time is about 0.7 μs/bytes and the restoring time is about 0.5 μs/bytes.

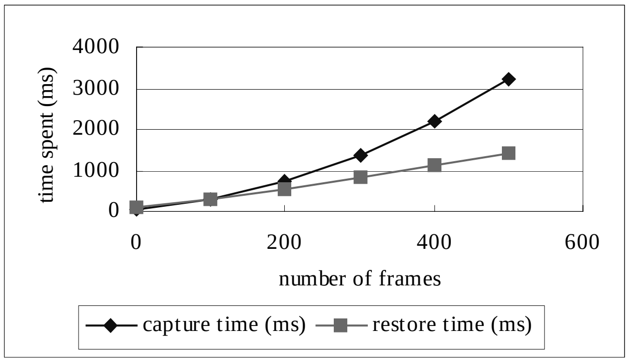

**Fig.6. Time spent in capturing and restoring frames**

Fig.6 shows the time needed in capturing and restoring frames. In this test, no local variables were defined in each frame. Hence, the measured time is the minimum overhead in capturing and restoring different number of frames.

## 5.3. Application Performance

Four parallel applications were used for evaluating the system. These were PI calculation, recursive ray-tracing, NAS integer sort and parallel SOR. The PI calculation and recursive ray-tracing programs are computationally intensive. The NAS integer sort and the parallel SOR are both computationally and communication intensive.

### 5.3.1. Overhead of M-JavaMPI

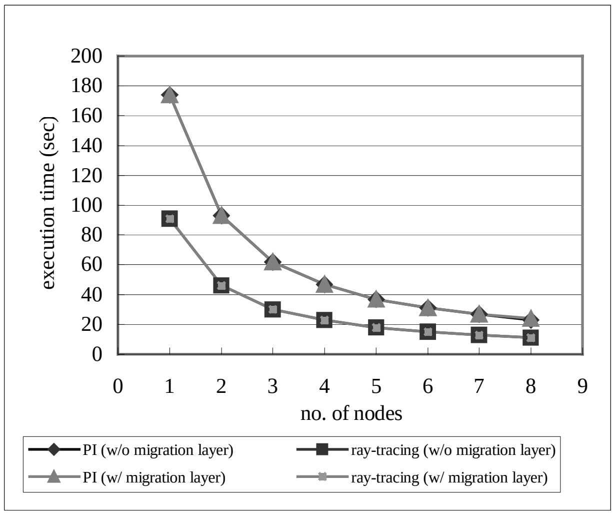

**Fig. 7. Time spent in calculating PI and Ray-tracing with and without the migration layer**

We first ran the programs in an evenly loaded environment to evaluate the overhead of the middleware during normal execution (i.e., no migration). Fig. 7 shows the time spent in the PI calculation and the ray-tracing program using different numbers of nodes with and without migration layer. For PI calculation, the two curves overlap with each other. Similar situation was observed in the two curves of the ray-tracing case. These results indicate that if there is no migration, the execution time of parallel Java applications is not affected at all by the presence of M-JavaMPI. The migration layer did not introduce any noticeable overhead in the execution of the tested Java applications.

| Problem size (no. of integers) | Time used (sec) in environment without M-JavaMPI | | | Time used (sec) in environment with M-JavaMPI | | | Overhead introduced by M-JavaMPI (in %) | |
|---|---|---|---|---|---|---|---|---|
| | Total | Comp | Comm | Total | Comp | Comm | Total | Comm |
| Class S: 65536 | 0.023 | 0.009 | 0.014 | 0.026 | 0.009 | 0.017 | 13% | 21% |
| Class W:1048576 | 0.393 | 0.182 | 0.212 | 0.424 | 0.182 | 0.242 | 7.8% | 14% |
| Class A: 8388608 | 3.206 | 1.545 | 1.66 | 3.387 | 1.546 | 1.840 | 5.6% | 11% |

**Table 1. Time spent in NAS program in different environments**

Table 1. shows the comparison of the performance of the NAS program with M-JavaMPI enabled and disabled. Two nodes were used for this test.

From the breakdown of the execution time, it can be seen that there is no noticeable overhead introduced in the computation part; while in the communication part, an overhead of about 10-20% was induced.

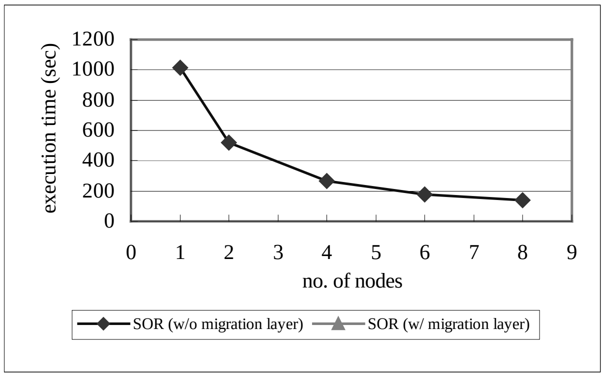

**Fig. 8. Time spent in executing SOR using different numbers of nodes with and without migration layer**

Fig.8 shows the time spent in executing the SOR application using different numbers of nodes with and without the migration layer respectively. The two curves showing the time spent in SOR with and without the migration layer nearly overlap exactly with each other. This shows that the migration layer does not introduce any noticeable overhead.

### 5.3.2. Cost of Migration

| Applications | Average migration time |
|---|---|
| PI | 2 |
| Ray-tracing | 3 |
| NAS | 2 |
| SOR | 3 |

**Table 2. Time spent in migration for different applications**

Table 2 shows the time spent in migration for different applications. A large part of the migration time was spent in starting the JVM and loading the program in the destination node.

Take the SOR program as an example. The execution of the program was repeated using six nodes in an unevenly loaded environment with one of the nodes executing a computationally intensive program. With no migration, the execution time of the program was 319s. The execution of the program was repeated then in the same environment. This time, however, shortly after the program had started, the process in the heavily loaded node was migrated to an idle node. The execution time came out to be 180s. This shows that considerable performance gain can be achieved by using the migration facility of our system.

## 6. Related Work

There are systems, such as JESSICA [1], Ara [6], and among others [5,7], that provide state-capturing and restoring of Java programs, but these systems need to modify the JVM. Some work [4,23] has been done to allow state-capturing and restoring via pre-processing of bytecode. Our approach is different from this in that M-JavaMPI uses pre-processing only to add code for state-restoring but not for capturing execution state. Besides, in their approaches, code is added to change the original program flow in order to do state-capturing and restoring. This could translate into considerable amount of overhead during runtime. In our approach, code is inserted as exception handlers which will only be executed during restoring. Some researchers have used exception throwing facility [3,8]. They use exception handlers to capture process states whereas we use them for state-restoring.

Several research projects [9,10,11,12,13,14] have provided Java bindings for MPI. All of them however did not include any restorable message-passing communication feature. Among them, mpiJava [11] and JavaMPI [9] use direct binding of Java programs and the MPI library. We use a client-server redirection model instead to avoid the instability of such a binding due to conflicts on the use of system resources. Moreover, our Java-MPI layer is “MPI-implementation-independent,” which makes our system more portable.

## 7. Conclusion

We use JVMDI to capture execution states, to be restored using exception handlers. The restorable MPI component provides for restorable and transparent message-passing communication for migrated processes. As JVMDI is a standard interface, our approach is potentially more portable and suitable for a heterogeneous environment, and there is no need to modify the JVM. Pre-processing is done to allow exceptions to be used to restore processes with no significant penalty inflicted on normal executions that require no migration.

Future work items include supports for processes with multiple threads, modification of the debugging interface to support M-JavaMPI in JIT mode, incorporation of a cluster monitoring system, called ClusterProbe [25], to support runtime workload detection, introduction of a system module for automatic dynamic load balancing, and one for post-migration I/O operations.

## Acknowledgement

This research was supported in part by a Hong Kong RGC grant with code HKU 7030/97E.

## References

[1] M.J.M. Ma, C.L. Wang, F.C.M. Lau, ``JESSICA : Java-Enabled Single-System-Image Computing Architecture," *Journal of Parallel and Distributed Computing*, Vol. 60, No. 10, October 2000, pp. 1194-1222

[2] T. Lindholm and F. Yellin. "The Java Virtual Machine Specification". Addison-Wesley, 1996.

[3] S. Funfrocken. "Transparent Migration of Java-based Mobile Agents (Capturing and Reestablishing the State of Java Programs)". *Proceedings of the Second International Workshop on Mobile Agents* (MA'98), pp.26-37

[4] M. Dahm. "Byte Code Engineering". *Proceedings JIT'99*, 1999.

[5] M.Ranganthan, A. Acharya, S. D. Sharma and J Saltz. "Network-aware Mobile programs". *Proceedings of the USENIX Annual Technical Conference*, Anaheim, California, 1997.

[6] H. Peine and T. Stolpmann. "The architecture of the Ara platform for mobile agents". *Proceedings of the Second International Workshop of Mobile Agents* (MA'97), 1997.

[7] S. Bouchenak. "Pickling threads state in the Java system". *Proceedings of the third European Research Seminar on Advances in Distributed Systems* (ERSADS'99), 1999.

[8] T. Sekiguchi, H. Masuhara, and A. Yonezawa. "A simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation". *Coordination Languages and Models*, Volume 1594 of LNCS, pages 211-226, Springer-Verlag, April, 1999

[9] S. Mintchev. "Writing Programs in JavaMPI". TR MAN-CSPE-02, Univ. of Westminster, London, UK, 1997

[10] Sava Mintchev and Vladimir Getov. "Towards portable message passing in Java: Binding MPI" Technical Report TR-CSPE-07". University of Westminster, School of Computer Science, Harrow Campus, July 1997.

[11] M. Bake. "mpiJava: A Java interface to MPI". *1st UK Workshop on Java for HKCN*, 1998.

[12] B. Carpenter, V. Getov, G. Judd, T. Skjellum, G. Fox. "MPI for Java". TR JGF-TR-03, Java Grande Forum, 1998.

[13] Tong WeiQin, Ye Hua, Yao WenSheng. "PJMPI: pure Java implementation of MPI". *Proceedings of the 4th International Conference on High Performance Computing in the Asia-Pacific Region*, 2000.

[14] K. Dincer. "A Ubiquitous Message Passing Interface Implementation in Java: jmpi". Proceedings of 13th International and 10th Symposium on Parallel and Distributed Processing, 1999.

[15] L.M. Silva, V. Batista, P. Martins, G. Soares. "Using mobile agents for parallel processing". Proceedings of the International Symposium on Distributed Objects and Applications, 1999.

[16] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, Songnian Zhou. "Process Migration". HP Labs, AT &T Labs-Research, TOG Research Institut, EMC, University of Toronto and Platform Computing.

[17] Danny B. Lange and Daniel T. Chang. "IBM Aglets Workbench: A White Paper". IBM Corporation.

[18] "Voyager Core Package Technical Overview" ObjectSpace Inc. 1997

[19] Javasoft. "Java Virtual Machine Debugger Interface". http://java.sun.com/j2se/1.3/docs/guide/jpda/jvmdi-spec.html

[20] Javasoft. "Java Object Serialization". http://java.sun.com/j2se/1.3/docs/guide/serialization/index.html

[21] M. Straber, J.Baumann and F. Hohl. "Mole – A Java based Mobile Agent system". Special Issues in Object Oriented Programming, pp.301-308, 1997

[22] Danny Lange and Mitsuru Oshima. "Programming and Deploying Java Mobile Agents with Aglets". Addison-Wesley, 1998

[23] Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen and Pierre Verbaeten. "Portable Support for Transparent Thread Migration in Java". In Proceedings of International Symposium on Agent Systems and Applications/Mobile Agents (ASA/MA'2000), September 2000, Zurich, Suisse.

[24] Jon Howell. "Straightforward Java Persistence Through Checkpointing". Proceedings of the 3rd International Workshop on Persistence and Java, 1998, pp.322-334.

[25] Zhengyu Liang, Yudong Sun, and Cho-Li Wang, "ClusterProbe: An Open, Flexible and Scalable Cluster Monitoring Tool," *The First International Workshop on Cluster Computing* (IWCC'99), pp. 261-268, Dec. 2-3, 1999.