Exploring Performance Improvement for Java-based Scientific Simulations that use the Swarm Toolkit

Xiaorong Xiang and Gregory Madey
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556 USA

Abstract

There has been growing interest for using the Java programming language in scientific and engineering applications. This is because Java offers several features, which other traditional languages (C, C++, and FORTRAN) lack, including portability, garbage collection mechanism, built-in threads, and the RMI mechanism. However, the historic poor performance of Java stops it from being widely used in scientific applications. Although research and development on Java resulting in JIT compilers, JVM improvement, and high performance compilers, have been done much for runtime environment optimization and significantly speeded up the Java programs, we still believe that creating hand-optimized Java source code and optimizing code for speed, reliability, scalability, and maintainability are also crucial during program development stage. Natural organic matter (NOM), a mixture of organic molecules with different types of structure and composition, micro-organisms, and their environment form a complex system. The NOM simulator is an agent-based stochastic model for simulating the behaviors of molecules over time. The simulator is built using Java programming language and the Swarm agent-based modeling library. We analyze the NOM simulation model from several aspects: runtime optimization, database access, objects usage, parallel and distributed computing. The NOM simulation model possesses most of characteristics which general scientific applications have. These techniques and analysis approaches can be generally used in other scientific applications. We expect that our experiences can help other developers using Java/Swarm to find a suitable way of achieving higher performance for their applications.

1 Introduction

C, C++ and FORTRAN have traditionally been used for modeling scientific applications. Since the Java programming language was introduced by Sun Microsystem in the mid-1990s, there has been growing interest for using it in scientific and engineering applications. The reason for this is that Java offers several features, which other traditional languages lack. Scientists use various platforms (such as Windows, Unix and MacOs) for their scientific studies. It is impossible to deploy an application written in C, C++, or FORTRAN languages from one platform to the other without rebuilding the application or changing the code. One of the most attractive features of Java is its portability, “write once, run anywhere.” Java runtime environment (JVM) provides an automatic garbage collection feature that reduces the burden of explicitly managing memory for programmer. The Java built-in threads implementation and Java's Remote Method Invocation (RMI) mechanism make it easy for parallel computing and distributed computing.

Although there are many attractive features provided by the Java language and the J2EE architecture makes Java a potential language for scientific applications, performance remains a prime concern for program developers using Java. The portability and memory management implementation in JVM impose a penalty on the performance. Ashworth (1999) [1] discussed several issues related to the use of Java for high performance scientific applications.

However, much research has been done to reduce the performance gap between Java and other programming languages. This research includes Just-in-Time (JIT) compilers that compile the byte code into native code on-the-fly just before execution, adaptive compiler technology (Sun's HotSpot VM), third-party optimizing compilers that compile the Java source code to the optimized bytecode, and high performance compilers that compile the Java source to native code for a particular architecture (IBM High-Performance Compiler for Java for RS6000 architecture). Bull (2001) [2] rewrote the Java Grande Benchmarks in C and FORTRAN. Bull also compared the performance between these languages in different Java runtime environments on different hardware platforms. The results demonstrate that the performance gap is quite small on some platforms.

The runtime environment optimization is an important aspect to study in order to achieve high performance. Determining and understanding the factors that affect the performance of scientific applications from a software engineering perspective and identifying and eliminating the bottlenecks that limit scalability at the software development stage are also necessary for high performance scientific applications.

In this paper, several approaches for analyzing and improving the performance of a particular scientific application, the NOM simulation model, have been described. The NOM simulation model is intended to simulate NOM behavior over time in a complex system. Such behavior includes transport through soil pores, adsorption to or desorption from mineral surfaces, and interactions with each other. The simulation model was built using the Java programming language and the Swarm library. The NOM simulation model is a typical distributed, stochastic scientific application that uses an agent-based modeling approach. It generates a substantial data set that must be stored in a remote database, able to be manipulated for producing useful information. The application simulates the behavior of a large number of molecules. An application of the NOM simulation model needs to run for a long time, often for days. Several aspects of the simulation model have been analyzed, including runtime optimization, database access, objects usage, and parallel and distributed computing.

The NOM simulation model possesses the characteristics which typical scientific applications have. These techniques and analytical approaches can generally be used in other scientific applications. We expect that our experiences can help other scientific application developers to find a suitable way of tuning and achieving higher performance for their applications.

2 Profiling tool

OptimizeIt is a Java J2EE performance tuning tool developed by Borland company. It is a commercial tool and the trial version can be downloaded from their Web site [1]. OptimizeIt can display the information about heap allocation, garbage collection, active threads, and class load in the form of graphs. The CPU sampling information is also displayed in a tree structure. These data can be exported to a formatted text file (HTML). Information about object allocation and deallocation can be viewed in a user selected order. OptimizeIt is a powerful tool that detects memory leaks and CPU performance bottlenecks in Java applications. The screen images shown in Appendix C reflect some of the features of OptimizeIt while it was used for profiling the NOM simulation model.

3 Data Structure

Selecting the appropriate data structure is important in a scientific application. Different data structures have significant impacts on the performance. No single data structure is appropriate for all situations. The best way to decide which type of data structure to use in an application is to create some benchmarks that reflect how the structure is used in the application.

In the NOM simulation model, each molecule object needs to be held in a collection. Individual molecule objects are accessed randomly at each time step. They can also be added or removed from this collection. A List data structure is a suitable choice to store, retrieve, and manipulate molecule objects. Java 2 platform provides two types of List structures: ArrayList and LinkedList. The ArrayList is a random access list, and the LinkedList is a sequential access list. The same operation has very different performance characteristics for different types of List. Position access is an operation used to access objects through its index in the List. Position access is a linear-time operation in the sequential access list, and it is a constant-time operation in a random access list. However, the remove operation for the random access list is more expensive than for sequential access list [3].

An algorithm for manipulating random access lists can produce quadratic behavior when it is applied to sequential access lists. In the NOM simulation model, in order to randomize the order of accessing the molecules in a List, the List needs to be shuffled at the beginning of each time step. The collections class in Java 2 platform provides a shuffle method to randomize the order of objects in the List. The shuffle algorithm used in the implementation has a linear time complexity for random access lists and

[1]http://www.borland.com/optimizeit/

quadratic complexity for sequential access lists. In order to avoid the expensive operation that would result from shuffling a sequential access list, the shuffle method converts the list into an array before executing shuffling and converts the shuffled array back into the list.

Three benchmarks have been created in order to test the performance, using different List structures and operations. These benchmarks reflect exactly how the data structures have been used in the NOM simulation model. In benchmark A, the MoleculeList is implemented using ArrayList, objects are accessed using get(), and MoleculeList is randomized using shuffle(). In benchmark B, MoleculeList is implemented using LinkedList, objects are accessed using get(), and shuffle() is used. In benchmark C, MoleculeList is implemented using LinkedList, objects are accessed using iterator, and shuffle() is used. In each benchmark, the add and remove operations are measured, and the overall performance is measured to reflect the NOM application behavior.

Figure 1 shows the relationship between different types of List with different operations. It also illustrates the behavior of different operations when the List size is doubled. In the NOM application, molecules are uniformly distributed on the grid. Doubling the grid size, therefore, will double the molecule number and the List size.

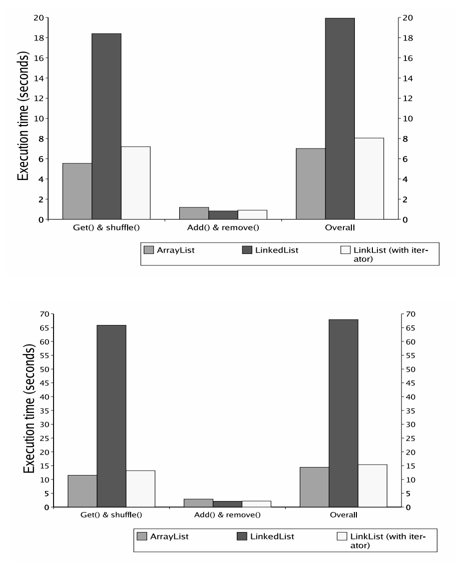

Figure 1: Performance comparison of different operations for ArrayList and LinkedList, the simulation runs for 500 time steps. Top: grid size is 400 X 300. Bottom: grid size is 800 X 300

The overall performance gain for ArrayList has been offset by the add and remove operations. By doubling the grid size, the time for randomly accessing the LinkedList increases more than 6 times. For this particular application, the ArrayList is the best choice for implementing the MoleculeList. It can get a maximum 2.8 speedup in this benchmark with grid size 400 X 300. When the grid size becomes large, the speedup increases.

4 Object reuse

Objects play a vital role in Object-Oriented programming, C++, and Java. The Java Virtual Machine (JVM) can automatically manage memory using the “garbage collection” mechanism. Java developers can allocate objects as necessary without considering deallocation, while C++ programmers must manually specify where an object in the heap is to be reclaimed by coding a “delete” statement. Excessively creating objects not only increases the memory footprint and CPU time for garbage collection, it also increases the possibility of memory leak.

Sosnoski (1999) [4] showed that the time for the object allocation in a Java program running on the JVM is 50 percent longer than one using the C++ code. This is caused by the overhead of adding internal information to help in the garbage collection process when allocating objects in heap.

Different JVMs, different versions of Sun JVM and IBM JVM, use various techniques to automatically discover objects when they are no longer being referred to and to recycle the memory periodically. Despite the automatic nature of the garbage collection process, a potential disadvantage is that it adds an overhead that can affect program performance, even in some JVMs such as Sun HotSpot JRE, where the garbage collection job runs in a separate thread.

Although the JVM is responsible for reclaiming the unused memory, Java programmers still need to put effort into making it clear to the JVM what objects are no longer being referenced [5]. For example, in some situations, a programmer needs to set the object into “null” manually in order to dereference the object. Although the memory leaks that are common in C++ are less likely to happen in Java, they can still occur due to poor design or simple coding errors.

An elegant way of reducing the overhead of objects created and destroyed and of improving the performance is object reuse. It can also reduce the probability of potential memory leaks. In order to reuse a certain type of object, several steps need to be followed:

- Isolating object

  Due to the overhead of object reuse, such as managing the object pools, only objects that need to be created and destroyed frequently are compatible with this technology. For a large scale application, using a powerful profiling tool is necessary to help developers detect this kind of object. OptimizeIt has been chosen in the development process of the NOM simulation model.

  In the NOM simulation model, at each time step, a certain number of molecules can enter into the system or leave the system. The molecule objects need to be created and destroyed frequently and they are candidates for potential reuse.

- Optimizing object size

  The size of objects not only has an effect on the memory footprint but also on the CPU time. It is worthwhile to estimate the size of a particular object and the number of instances for a given class in memory. A trade off can be made by either reducing the object size or keeping the precision.

- Reinitializing object

  Although Sun's HotSpot VMs radically improved the performance of object allocation and garbage collection, object allocation and instantiation still has a significant cost, especially when the object size is small.

  A micro-benchmark is created for testing the time of creation or re-initialization of the *Molecule* objects in the NOM simulation model. In this benchmark, 1000 *Molcule* objects are created, then reinitialized to their original state. The average creation time for these *Molecule* objects is 53.85 million seconds, while the average reinitializing time is 3.9 million seconds.

- Creating object pool

  In order to reuse certain types of objects, these objects need to be kept in a collection in the memory, the so-called object pool. An object pool is used to store a free list of objects. Generally, an object pool can be implemented using Vector, Linked List, ArrayList, or a raw array. Selecting a suitable type of data structure depends on the type of operations used for managing the pool.

  In the NOM simulation model, the object pool has been implemented as a First-In-First-Out (FIFO) queue. When a new molecule object needs to be created, the method first needs to check the object pool. If there is an available object in the pool, the object is removed from the queue and reinitialized. If there is no object available, a new object is created. When a molecule object leaves the system, it is added at the end of the queue. The data structure chosen for the object pool implementation is LinkedList.

Object reuse is a simple and elegant way to conserve memory and enhance speed. By sharing and reusing objects, processes or threads are not slowed down by the instantiation and loading time of new objects, or the overhead of excessive garbage collection.

5 Database connection and database query

The database design and optimization plays a critical role in evaluating the performance and scalability of scientific applications. Most modern databases, called database management systems (DBMS), are built for speed and scalability. They are distributed systems that consist of a number of concurrently running processes, threads, and various machines that work together in an efficient way to deliver fast and scalable data. The basic purpose for a database is to store, manage, and access the data on one or several distributed machines.

For an application written in the Java programming language, communication to a database is accomplished through a Java Database Connectivity (JDBC) driver, with all database Input/Output via SQL (Structured Query Language). Database vendors provide their own JDBC drivers that conform to the common Java API defined by Sun Microsystems. Using JDBC allows developers to change database location, port, and database vendors with minimal changes in code.

Database query operations include data retrieval, insertion, updating, and deletion, as well as table creation and alteration. In a Web-based scientific application, data retrieval, insertion, updating, and deletion are the four most frequently used operations. JDBC offers several advanced techniques that allow the programmer to write high performance queries [6]. These techniques are presented to show the significant impact on performance and scalability.

- Connection Pooling

  Establishing a connection to a database is a time-consuming process, especially for a short query. It is therefore reasonable to reuse the connection object that connects to the same database repeatedly. Connection pooling is a runtime object pool that can be used to cache connections. The connection pool is normally used in a Web-based application connected to a database that a number of clients frequently access. Database vendors and application server vendors provide connection pooling utilities for managing the connection between Web applications and databases in an efficient and reliable way.

- Prepared statements

  Query processing is a process for resolving a SQL query. It can be broken down into three basic phases: query parsing, query plan generation and optimization, and plan execution [6]. The query parsing phase is a syntax-checking process for the string-based SQL query that ensures the query statement is legal. If a SQL statement or a set of similar SQL statements needs to be executed repeatedly, the cost for the parsing process can be reduced by caching the previously parsed queries. JDBC provides this function with a PreparedStatement object. Unlike the Statement object, which needs to be sent to a database for parsing each time, the PreparedStatement is compiled in advance and can be executed as many times as needed. The speedup of a query varies according to the type of query (SELECT, INSERT, UPDATE, DELETE) or the complexity of the query (more than one table involves).

- Batch updates

  For a large scale scientific application, the application and the database normally reside on physically distributed machines. Substantial network latency can lead to very inefficient query execution. JDBC provides the batch update approach that can decrease the network latency effect by executing a number of queries in one network round trip. The query processing, however, is not necessarily faster when using the batch update approach instead of the PreparedStatment. The trade off comes in two forms, (1) batch updates can only be combined with a Statement object and (2) the size of the data needs to be sent from the client to the server in one large round trip. The batch updates approach offers more benefits when the network connection is slow.

- Transaction management

  In SQL terms, a transaction is a series of operations that must be executed as a single logical unit. When a connection is created using JDBC, the database is set in autocommit mode and each SQL statement is treated as a separate transaction. In order to allow two or more statements to be grouped into a transaction, the autocommit mode needs to be disabled using *Connection.setAutoCommit(false)*. Treating a group of operations as a transaction is a safe way to guarantee the integrity of a database. For example, if a bank customer wishes to transfer funds from a savings account to a checking account, the two update operations need to be sent. If these two calls are treated as two separate transactions, then one is successful and the other fails due to the network or other factors. This results in data that is not consistent in the database.

  Explicitly commit a group of SQL statements is not only a safe approach, but also has large performance impact because the overhead of the commit operation is reduced. In the NOM model, all the data and information pertaining to the reacted molecules in the system are stored in the database at the end of each time step. The insertions for the data of all the reacted molecules at each time step are treated as one transaction to ensure data consistency in the system.

The NOM core simulation engine connects to the Oracle database using the JDBC thin driver. Figure 2 shows the performance comparison for data insertions using five approaches. The simulation runs for 96 time steps, with a total of 1782 insertions.

The benchmark consists of five cases. In case 1, each insertion for each reacted molecule was treated as a single transaction with a Statement object. In case 2, each insertion for each reacted molecule was treated as a single transaction with a PreparedStatement object. In case 3, a group of insertions for every reacted molecule in one time step is treated as a single transaction with Statement object. In case 4, a group of insertions for every reacted molecule in one time step is treated as a single transaction with PreparedStatement object. In case 5, a batch updates approach is applied.

Case 4, the PreparedStatement object with transaction management has the best performance in the NOM application. It offers 3.03 times speedup relative to case 1 in this benchmark.

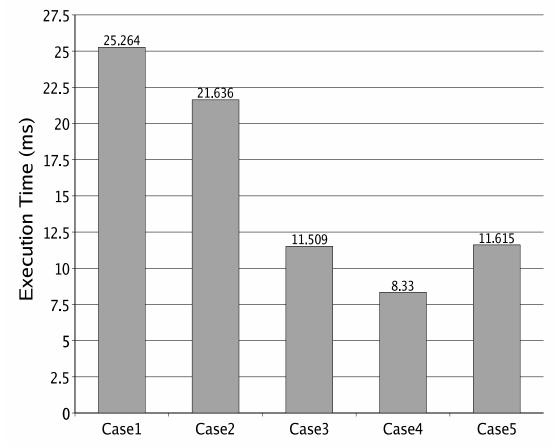

Figure 2: Comparison of data insertions using five approaches in NOM simulation model

6 Parallel data output with Java threads

Querying a database is an I/O bound process, especially when the database sever and the application server are on different machines. This is a common architecture design in large-scale scientific applications. For several independent queries, it is more efficient to make use of idle CPU cycles and have these queries executed in parallel instead of one after the other. For database queries that only involve the data read from a database, the parallelism can be easily accomplished by using the Java built-in threads and connection pooling technologies.

Large-scale scientific applications are not only I/O bound processes, but also CPU bound processes. If the application server has multiple processors, multithreading can be used to overlap the computation and communication. More specifically, parallelism can be achieved by overlapping the computation and the I/O. There are, however, trade offs between the simplicity of programming and the performance. When an application not only involves the data-read but also involves the data-write, several programming issues need to be considered to prevent the deadlock and the race conditions.

In the NOM simulation model, large amounts of data need to be written in the database at each time step. The average time for one record insertion using PreparedStatement object with transaction management is 4.7 milliseconds as shown in previous section. In order to parallelize the data write to the database, a buffer (FIFO queue) is allocated and an extra thread is created. While the computational thread adds the object to the queue, the I/O thread removes the object from the queue and writes the data to the database. If there are no objects in the queue, the data-writing thread executes busy waiting. If the number of objects in the queue is equal to the queue size, the computation thread waits. In order to safely add to and remove from the queue, all the accesses to the queue are synchronized.

The NOM simulation model has been tested and run for various time steps, from 96 time steps to 579 time steps. Refer to Figure 3.

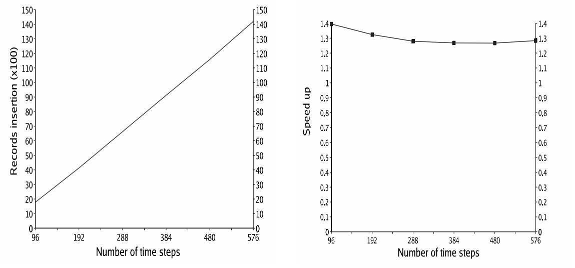

Figure 3: Overlap the computation and I/O using Java threads. Left figure shows that the number of data insertion over the time steps. Right figure shows that the speedup of using a separate thread for data output.

By using a separate thread for data output, there is average 1.3 speedup relative to the single threads model for the NOM application. In the NOM simulation model, the time for computation is more than the time for I/O at each time step. If the computation takes much less time than the I/O, we can consider using more I/O threads to execute data writing.

7 Choosing JVM

Various Java Virtual Machines (JVM), such as IBM JVM and Sun HotSpot JVM, have been implemented in conforming to the Java Virtual Machine Specification [7]. The runtime and the compiler are two main parts in the Sun HotSpot architecture. The compiler portion in a JVM translates the bytecode into native machine instructions in order to improve execution speed. The runtime portion in a JVM includes a bytecode interpreter, memory management, garbage collection, and a low-level task handler [3].

Sun MicroSystem implements two types of HotSpot JVM, Client VM and Server VM, to meet different requirements for different applications. Compared with server side programs, client side programs often require a smaller RAM footprint and have a faster start-up time. These two HotSpot JVM share the same runtime portion, but the main difference between them is found in their compiler technologies. The Server VM contains a highly advanced adaptive compiler that includes many of the optimization technologies which are used in C++ compilers [3].

The application of the NOM simulation model has been benchmarked using two different runtime modes of Sun JVM 1.4.1_01 on Redhat Linux 8.0 for 500 time steps and 1500 time steps. Figure 4 shows that as the grid size and the time steps increase, Server VM offers higher performance than Client VM. The results shows that choosing different JVMs can produce significant differences in the performance.

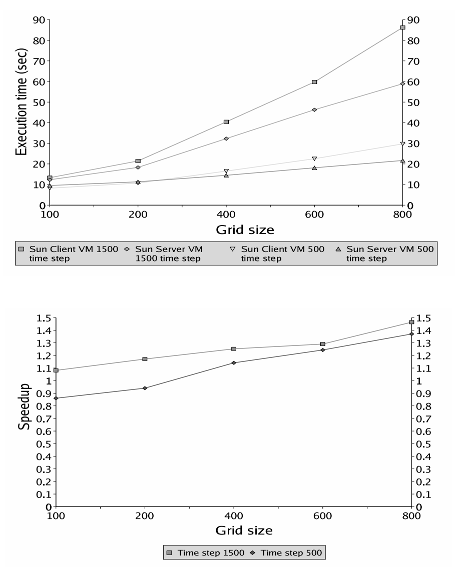

Figure 4: Performance comparison for a NOM application run in Sun Client VM and Sun Server VM with different grid sizes.

Ladd (2003) [8] showed benchmark results for both Sun and IBM JVM with versions 1.3.1 and 1.4.1_01 on a Linux platform. According to Ladd, the JVM version 1.3.1 has a higher performance level than version 1.4.1 and the IBM JVM has a better performance level than Sun JVM. Choosing the appropriate JVM for a particular scientific application also involves the consideration of the hardware architecture and the operating system.

8 High performance compilers

Traditionally, a Java program is compiled into bytecode using a compiler and a Java Virtual Machine (JVM) is needed to read in and interpret the bytecode.

Jikes[2], a faster compiler developed by IBM, can translate Java source code into bytecode 10 times faster than the `javac` from Sun JDK 1.4.1_01. Although it does not necessarily generate a faster running code, it can speed up the development process for large applications.

GCJ[3], the GNU Compiler for the Java language, can compile the Java source code into either the bytecode or the native code. GCJ has been integrated into GCC. Bothner (2003) [9] discussed the advantages, features, and limitations of GCJ in detail. Ladd (2003) [8] did several benchmarks using GCJ on the Linux platform and showed a performance gain of GCJ over other JVMs. The GCJ compiler is a project under development and several limitations still exist. For example, GCJ can not compile the Java program with `swing`. Using GCJ, the Java application with Swarm library can be compiled into native code, but it does not improve the performance.

There are many other runtime environment optimizers and high performance compilers. AlphaWorks is a high performance compiler for Java from IBM alphaWorks. It can be used on OS/2, AIX and Windows NT platform. JOVE [4] also can only be used on Windows machines. TowerJ environment [5], developed by Tower Technology Corporation, is another example of high performance compilers. It is available for Solaris and Linux platforms.

9 Scalability

For large scale scientific applications written in Java, the scalability can be improved using the parallelism programming model. The parallelism model can run with a single JVM in a shared memory multiprocessor system or with multiple JVMs in a distributed memory system. The Java built-in threads mechanism is a convenient method for parallelism implementation in shared memory environments. However, for large scale applications that require large memory and CPU time, distributing the application on multiple JVMs in a distributed memory system with some

[2]http://www-124.ibm.com/developerworks/oss/jikes/
[3]http://gcc.gnu.org/java/
[4]http://www.instantiations.com/jove/
[5]http://www.towerj.com

message passing mechanisms for inter-VM communication is a suitable way to address the requirements.

The standard Java libraries, thread class, is appropriate for using in the parallel programming paradigm in a single JVM environment. Since most scientific applications are CPU bound, in order to avoid the context switching to achieve best performance, the number of threads should match the number of processors in the hardware architecture. Additionally, the thread creation and destroying should be avoided by creating and managing a thread pool.

OpenMP [10], an open standard for shared memory directives, defines directives for FORTRAN, C, and C++. OpenMP provides a portable and scalable model that offers a simple and flexible interface for developing parallel applications in shared memory systems. JOMP [11] provides a set of OpenMP-like directives and library routines for supporting shared memory parallel programming in Java. It uses Java threads as the underlying parallel model and is most useful for parallelizing scientific applications at the loop level.

For distributed computing, Java provides a communication mechanism using sockets and the RMI (Remote Method Invocation). Java RMI [12] is a message passing paradigm based on the RPC (Remote Procedure Call) mechanism. RMI is primarily intended for use in the client-server model instead of the peer-to-peer communication model. On the other hand, the explicit use of sockets is too low-level to be used to develop a parallel application [13].

In C, C++, and FORTRAN, an explicit message passing interface (MPI) [14] standard has been defined for supporting communication of an application in cluster environments. MPI is the most widely-used standard for inter-process communication in high-performance computing. MPICH [15], LAM MPI[16] are two successful examples of portable MPI implementations in traditional languages. Programming with MPI is relatively straightforward because it supports the single program multiple data (SPMD) model of parallel computing, wherein a group of processes cooperate by executing identical program images on local data values.

In 1998, a group of researchers of the Java Grande Forum worked on a specification MPI-like application programming interface for message passing in Java (MPJ) [17]. The current implementations of the MPJ can be separated in two ways: as a wrapper to existing native MPI libraries or written in pure Java. NPAC's mpiJava [18] is an example of the wrapper approach using Java Native Interface (JNI) to execute a native call. The JMPI project [19] implements message passing with Java RMI and object serialization. The jmpi [20] is built upon the JPVM system. MPIJ [21] is a Java based implementation of MPI integrated with DOGMA (Distributed Object Group Metacomputing Architecture). The MPJ implementation in pure Java is usually slower than wrapper implementations for existing MPI libraries, but pure Java implementations are more reliable, stable, and secure [20]. Getov (2001) [13] did an experiment that used the IBM High Performance Compiler for Java (HPCJ), which generates native code for the RS6000 architecture, to evaluate the performance of MPJ on an IBM SP2 distributed-memory parallel machines. The results show that when using such a compiler, the MPJ communication components are as fast as those of the MPI.

9.1 Parallel implementations

The scalability of the NOM simulation model involves two aspects, the required total number of simulation time steps and the grid size. Two parallelism programming models are implemented for the NOM simulation model. The Java thread version is implemented using built-in Java threads and runs on a single JVM. The distributed memory model is implemented by using mpiJava library with LAM MPI.

In the sequential implementation model for simulating the NOM complex system, the behavior of individual molecules is simulated using the agent-based modeling approach. Molecules reside in cells on a 2D grid and individual molecules can be transported through the soil medium via water flow. At each time step, molecules can move from one cell to the other when a random event occurs. The time for finishing a simulation is largely determined by the number of molecules in the system and the time steps.

9.1.1 Java threads version

In order to parallelize the programming model, the original grid has been equally separated into two subset grids (e.g., a 800X300 grid is separated into two 400X300 grids). Two threads are created, each thread has its own grid object to place molecules and a collection to hold molecules. In each time step, the computation for individual molecules is executed on the two threads concurrently.

When one molecule moves across the boundary, the molecule is removed from the grid and placed into a local buffer in the current thread. At the end of the time step, a Barrier is used to synchronize these two threads at this point. After all the threads reach this state, one of the threads executes the exchange operation by maintaining the state of two grids and two *MoleculeLists*. Threads have been synchronized before this thread finishes the setting of boundary condition. Figure 5 shows the design.

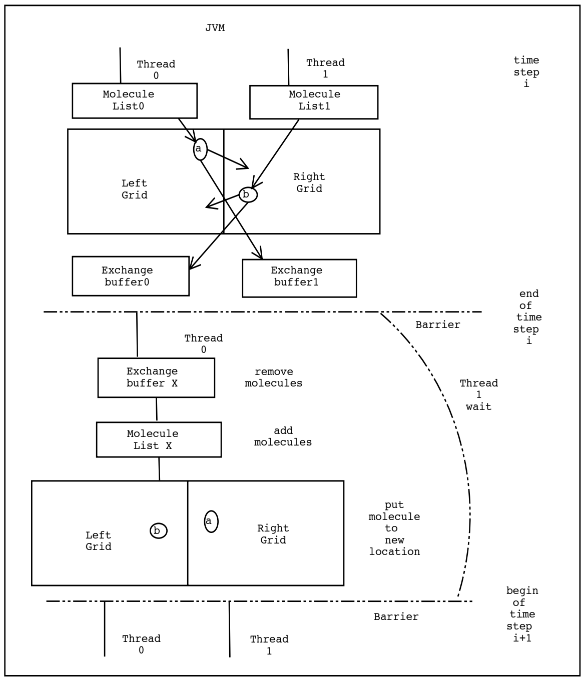

Figure 5: The design for parallelism of NOM simulation model using Java threads.

9.1.2 MPJ version

In the distributed memory model, each processor runs its own copy of code. The ModelSwarm object contains a subset of the original grid. These subsets of the grid have equal size in order to ensure the computational balance of each node in the cluster. The basic design model is similar to the Java thread model. Each node in the cluster machine processes its own computation of the subset of the grid. When a molecule crosses the boundary, it is added into the local buffer. MPI.COMM_WORLD.Barrier is used to synchronize these processes at the each time step. Molecule objects that cross the boundary are sent to their neighbor grids using blocking send and receive modes, MPI.COMM_WORLD.Sendrecv, MPI.COMM_WORLD.Send, and MPI.COMM_WORLD.Recv. Figure 6 shows this design. The molecule list and grid on each machine are updated after all the sending and receiving are finished.

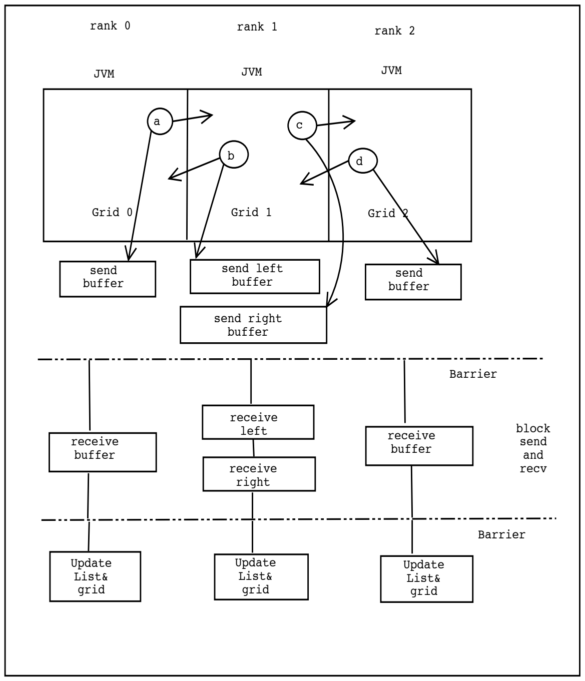

Figure 6: The design for parallelism of NOM simulation model using MPJ.

9.2 Performance results

In order to measure the performance of the parallel implementations, a Linux cluster has been built. The cluster consists of 4 PCs. Each PC has dual 650 MHz Intel processor running the RedHat Linux 8.0 Operating System. The Java codes are implemented and compiled using SUN's 1.4.1_01 Java Development Kit and executed on SUN's Java Virtual Machine.

Figure 7 presents the results for the sequential version and Java thread version for the NOM application that runs for 1500 time steps with various grid size. Both versions ran on a single dual Intel computer.

The experiments for parallelism of the NOM simulation model using the distributed memory model are made on a cluster of four PCs. LAM MPI 6.5.9 and mpiJava have been used to build the execution environment. The NOM application runs on 2 and 4 machines.

Figure 8 shows that the performance comparison between the sequential programming model and the MPJ model that ran on 4 nodes in the cluster. These two models both ran for 500 and 1500 time steps. This figure shows that the communication between nodes and the grid and *MoleculeList* maintenances offset the performance gained by distributing the job to different computers when the problem size is small.

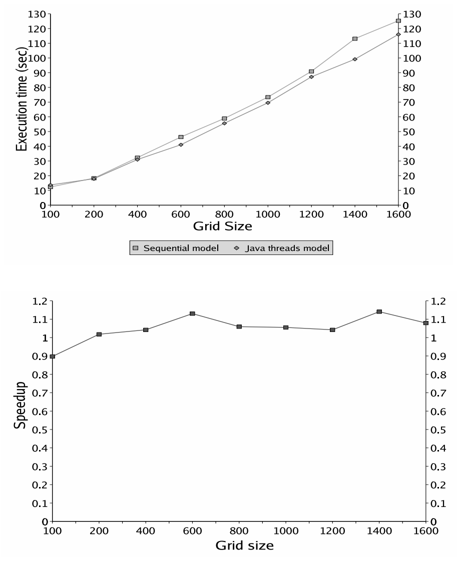

Figure 7: Compare the performance between sequential programming model and Java thread model in the NOM application (one thread vs. two threads on a single dual CPU computer).

As the problem size grows (larger grid size or longer time steps), however, a performance improvement appears by distributing the job to. Compared to the ideal performance gain of a factor of 4, the efficiency is low.

Figure 9 illustrates that the job has been distributed on four nodes in the cluster machines and ran for 500 time steps. There is no synchronization between nodes. Instead of sending the molecules which cross the boundary to other nodes, they are wrapped to the other side of this subset grid. This figure shows that as the grid size increases the speedup is closer to the ideal linear speedup of 4.

10 Conclusion

In this paper, several approaches for exploring the performance and scalability improvement of a typical scientific application, the NOM simulation model, has been presented. These approaches are summarized in Table 1. The speedup varied by the problem size and different situations. The speedup shown in the table came from the benchmarks that were described in the previous sections. All the numbers of speedup listed in the table show the approximate highest performance gain in each benchmark.

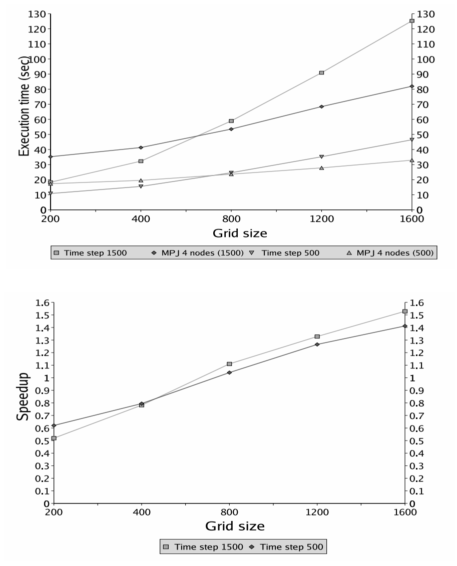

Figure 8: Comparison of the performance between sequential programming model and MPJ model that runs on 4 machines for 1500 time steps.

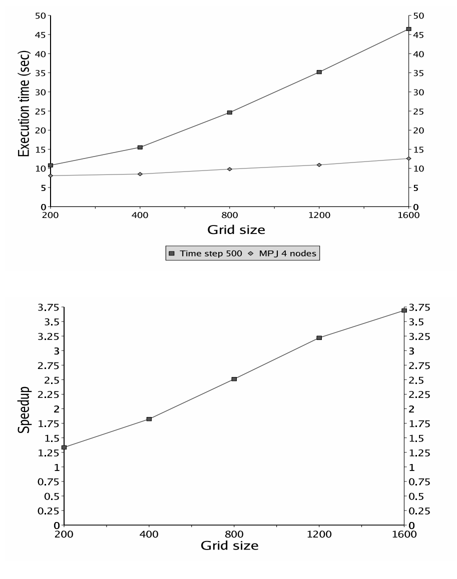

Figure 9: Comparison of the performance between sequential programming model and MPJ model on 4 machines without synchronization and molecule exchange.

Table 1: Summary of the performance improvements

Approaches	**Speedup**	**Comments**
Data structure	2.8	using ArrayList provides higher overall performance than using LinkedList in the benchmark.
Object reuse	-	reduce the memory footprint, the garbage collection cycle, and the overhead of object allocation and deallocation. The performance gain is relatively small compare to the overall computation time in the NOM simulation model.
JDBC	3	JDBC tuning can effect the performance of data I/O.
Parallel data output	1.3	overlap the computation and I/O using Java threads can improve the performance.
Java runtime	1.4	Sun Server VM has higher performance than Sun Client VM for large-scale scientific applications.
Java threads model	1.1	performance of Java threads model largely depends on the JVM implementation and how efficient the operating system handle the threads.
MPJ model	1.5	distributing the job to 4 nodes in a cluster has relative larger performance gains when problem size is big and the communication between nodes is minimized. However, it is still much lower than the ideal performance gain (4).

Besides the approaches that listed in the Table, using a native code compiler that can compile the Java source code to native code can increase the performance for some applications.

Program profiling is a crucial step for high performance computation in Java-based applications. Two major aspects, CPU time and memory usage, need to be monitored.

Selecting the appropriate runtime environment for a particular application is important. Besides the JVMs that have been evaluated here, IBM JVM is also valuable to be investigated.

Using Java built-in threads to parallelize the Java applications is a convenient approach. How much performance can be gained from this parallism depends on the JVM implementation and the efficiency of the operating system handling the Java threads. The experiments that we did are on a dual CPU PC with Linux operating system. It is valuable to extend these experiments to Windows operating system or Solaris operating systems.

Distributing jobs on multiple machines in a cluster environment is an efficient approach for large size problems. However, the communication among nodes and the grid and list maintenances offset the performance gain. In order to avoid this overhead, instead of synchronizing all the processes at each time step, they can be synchronized at every 5 time steps or more. For this particular application, it is also possible to distribute the job on multiple machines using MPJ and combine the results at the end of the simulation in the database. There is no communication at all after the job is distributed. However, the validation is necessary for these two approaches.

References

[1] Mike Ashworth. The potential of Java for high performance applications. In *The First International Conference on the Practical Application of Java*, pages 19–33, 1999.

[2] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman. Benchmarking Java against C and FORTRAN for scientific applications. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 97–105, June 2001.

[3] Steve Wilson and Jeff Kesselman. *Java platform performance strategies and tactics*, chapter Appendix B. Addison-Wesley, 2000.

[4] Dennis M. Sosnoski. Java performance programming, part 1: Smart object-management

saves the day. *Java World*, Nov 1999. http://www.javaworld.com/javaworld/jw-11-1999/jw-11-performance.html.

[5] Steve Wilson and Jeff Kesselman. *Java platform performance strategies and tactics*, chapter Appendix A. Addison-Wesley, 2000.

[6] Greg Barish. *Building Scalable and High-Performance* J*ava Web Applications using* J2EE *Technology.* Addison-Wesley, 2002.

[7] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, Second Edition.* Addison-Wesley, 1999.

[8] Scott Robert Ladd. Benchmarking compilers and languages for ia32. http://www.coyotegulch.com/reviews/almabench.html, 01 2003.

[9] Per Bothner. Compiling Java with GCJ. *Linux Journal*, Jan 2003. http://www.linuxjournal.com/article.php?sid=4860.

[10] OpenMP Architecture Review Board. OpenMP C and C++ application programming interface. Technical report, OpenMP Architecture Review Board, 1998. Available from http://www.openmp.org.

[11] Mark Bull and Mark Kambites. JOMP–An OpenMP-like interface for Java. In *ACM 2000 Java Grande Conference*. ACM, 2000. Available from http://www.epcc.ed.ac.uk/research/jomp.

[12] Sun Microsystems. Java remote method invocation specification. Technical report, Sun Microsystems, 1998. Available at: http://java.sun.com/products/jdk/rmi/.

[13] V. Getov, G. von Laszewski, M. Philippsen, and I. Foster. Multiparadigm communications in Java for grid computing. *Commication of the ACM*, 44(10):118–125, 2001.

[14] MPI. http://www-unix.mcs.anl.gov/mpi/.

[15] MPICH: A portable implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich/.

[16] LAM/MPI parallel computing. http://www.lam-mpi.org/.

[17] Bryan Carpenter, Vladimir Getov, Glenn Judd, Anthony Skjellum, and Geoffrey Fox. MPJ: MPI-like message passing for Java. *Concurrency: Practice and Experience*, 12(11), 2000.

[18] Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. mpiJava: An objected-oriented Java interface to MPI. In *International Workshop on Java for parallel and Distributed Computing, IPPS/SPDP 1999*, April 1999.

[19] Steven Morin, Israel Korean, and C. Mani Krisha. JMPI: Implementing the message passing standard in Java. In *Internatinal Parallel and Distributed Processing Symposium: IPDPS 2002 Workshops*. IEEE, 2002.

[20] Kivanc Dincer. Ubiquitous message passing interface implementation in Java: jmpi. In *13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing.* IEEE, 1999.

[21] Glenn Judd, Mark Clement, and Quinn Snell. Dogma: Distributed object group management architecture. In *Concurrency: Practice and Experience*, volume 10. ACM 1998 Workshop on Java for High-Performance Network Computing, 1998.