EXPLORING IMPROVEMENT FOR JAVA-BASED SCIENTIFIC SIMULATIONS

Jayanti Kumari[#1]
[#]*Faculty, Department of Computer Application*
Yogoda Satsanga Mahavidyalaya – Centre for Vocational Studies
Ranchi, Jharkhand, India
[1]Email: jayantikumari@gmail.com

***Abstract*—Application of Java programming language in scientific and engineering applications is growing day by day. Java offers several features like portability, garbage collection mechanism, built-in threads, and the RMI mechanism. Historically, Java is known for poor performance in scientific applications, but recent researches in Java has resulted in JIT compilers, JVM improvement, and high performance compilers. Runtime situation optimization has significantly enhanced the speed of execution of java programs. In fact hand-optimized Java source code for speed, dependability, scalability, and maintainability are also critical during program development stage. In this work an agent-based simulation model is built using Java programming language and the Swarm Simulation library. The performance of this simulation model is analyzed from several aspects viz., runtime optimization, database access, object usage, parallelization and distributed computing. This simulation model possesses most of characteristics which general scientific simulations have. These techniques and analysis approaches can also be commonly used in other scientific simulations using Java.**

***Keywords*— programming, reliability, simulations, mechanism, optimization, presentation**.

I. INTRODUCTION

C, C++ and FORTRAN have traditionally been used for modeling scientific applications. Since the Java programming language was introduced by Sun Microsystems in the mid-1990s, there has been growing importance 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 characteristic that reduces the burden of explicitly managing memory for programmer. The Java built-in threads performance 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.

Conversely, 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 Hot Spot VM), third party optimizing compilers that compile the Java source code to the optimized byte code, 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 Bench marks 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 relatively small on some platforms. The runtime environment optimization is an important aspect to study in order to achieve best performance. Determining and understanding the factors that affect the performance of scientific applications from a software engineering point of view and identifying and eliminating the bottlenecks that limit scalability at the software development stage is also necessary for best performance scientific applications. In this paper, several approaches for analyzing and improving the presentation of a particular scientific simulation, that is used to study Natural Organic Matter (NOM), has been described. The NOM simulation was built using the Java programming language and the Swarm library from Santa Fe Institute[3]. 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 and able to be manipulated for producing helpful information. The application simulates the behavior of a big 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 has been analyzed, including runtime optimization, database access, objects usage, and parallel and distributed computing. The NOM simulation model possesses the uniqueness which typical scientific applications have.

That 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 proper way of tuning and achieving higher performance for their applications.

II. Related Works

More and more implementations of Java based simulation environment and toolkits [4] [5] [6] [3] indicate that Java language and Java-based technologies has been widely and will be continue used in simulation modeling and implementations. Habitually, a Java program is compiled into byte code using a compiler and a Java Virtual Machine (JVM) is needed to read in and interpret the bytecode.GCJ1, the GNU Compiler for the Java language, can compile the Java source code into either the byte code or the local code. GCJ have been integrated into GCC. Bothner(2003) [7] discussed the advantages, features and boundaries 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 boundaries 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 local code, but it does not recover the performance. There are several other runtime environment optimizers and high performance compilers. Alpha Works are a high performance compiler for Java from IBM alpha Works. That can be used on OS/2, AIX and Windows NT platform. JOVE 2 also can only be used on Windows machines. Tower J environment 3, developed by Tower Technology Corporation, is another example of high performance compilers. It is available for Solaris and Linux platforms. For big scale scientific applications written in Java, the scalability can be improved using the programming model. The parallelism model can run through 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 technique 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 message passing mechanisms for inter-VM communication are a suitable way to address the requirements. The standard Java libraries, thread class are 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 equivalent the number of processors in the hardware architecture. Furthermore, the thread creation and destroying should be avoided by creating and managing a thread pool. Open MP [9], an open standard for shared memory directives, defines directives for FORTRAN, C, and C++. Open MP provides a portable and scalable model that offers a simple and flexible interface for developing parallel applications in shared memory systems. JOMP [10] provides a set of Open MP-like directives and library routines for sustaining shared memory parallel programming in Java. It uses Java threads as the underlying parallel model and is best 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 [11] is a message passing paradigm based on the RPC (Remote Procedure Call) mechanism. RMI is primarily deliberate 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 build up a parallel application [12].

In C, C++, and FORTRAN, an explicit message passing interface (MPI) [13] standard has been defined for supporting communication of an application in group environments. MPI are the most widely used standard for inter-process communication in high-performance computing. MPICH [14] and LAM MPI[15] are two successful examples of manageable MPI implementations in traditional languages. Programming with MPI are relatively straightforward because it supports the single program multiple data (SPMD) model of parallel computing, where in a group of processes assist 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) [16]. The current implementations of the MPJ can be separated in two ways: as a wrapper to accessible native MPI libraries or written in pure Java. NPAC's MPI Java [17] are an example of the wrapper approach using Java Native Interface (JNI) to execute a native call. The JMPI project [18] implements message passing with Java RMI and object serialization. The JMPI [19] are built upon the JPVM system. MPIJ [20] are a Java based implementation of MPI integrated with DOGMA (Distributed Object Group Meta computing Architecture). The MPJ implementation in pure Java is usually slower than binding implementations for existing MPI libraries, but pure Java implementations are more reliable, stable, and secure [19]. Getov (2001) [12] did an experiment that used the IBM High Performance Compiler for Java (HPCJ), that generates native code for the RS6000 architecture, to evaluate the performance of MPJ on 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. Optimize It is a Java J2EE performance tuning device developed by Borland company. That is a commercial device and the trial version can be downloaded from their Web site 4. Optimize It can display the information about heap allocation, garbage collection, active threads and class load in the form of graphs. The CPU variety information is also displayed in a tree structure. These data can be exported to a format text file (HTML). Information about object allocation and deal location can be viewed in a user selected order. Optimize It is a powerful device that detects memory leaks and CPU performance bottlenecks in Java applications.

III. SIMULATION MODEL

Natural organic matter (NOM) is a mixture of heterogeneous molecules that come from animal and plant material in the natural environment. It plays a crucial role in ecological and bio-geochemical processes such as the evolution of soils, the transport of pollutants, and the global biochemical and geochemical cycling of elements [21]. NOM, a prevalent constituent of natural waters, are highly reactive with mineral surface [21]. But NOM is transported through soil pores by water, it can be adsorbed onto or desorbed from mineral surfaces. Sorption of NOM is an important consideration in the handling of drinking water. The evolution of NOM over time from originator molecules to mineralization is an important research area in a wide range of disciplines, with biology, geochemistry, ecology, and soil science and water resources. NOM, micro-organisms, and their environment form a complex system. The global phenomenon of a complex system can often be observed by simulating the dynamic behavior of individual components and their connections in the system. Composite systems often have emergent properties. The evolution of NOM over time from precursor molecules to concluding mineralization involves various molecular transformations. These transformations grip chemical reactions, adsorption, aggregation and physical transport in soil, ground, or surface waters. In order to provide scientists a test bed for their theoretical analysis and experimental results for NOM study, a Web-based simulator was built. We expect that the system will help scientists good understand the NOM composite system by providing them with information for predicting the properties of the NOM system over time. The NOM simulation model is built upon the J2EE architecture running on a circulated cluster. That cluster has multiple dual processor PCs running Red hat Linux 8.0 and Windows 2000 operating systems. These machines in the cluster include a HTTP server, simulation servers, database servers, a reports server, and a data mining server.

The NOM simulation system is an agent-based stochastic model that can model NOM, mineral surfaces, and microbial connections near the surface of the soil. In the stochastic model of the evolution of NOM in separate time and space, NOM are presented as a big number of discrete molecules with varying chemical and physical properties. Individual molecules can be transported through the soil medium via water flow, adsorb on the soil particle surfaces, and react with other molecules, micro-organisms, and the surroundings. Individual molecules are represented as agents with their own complex structures. These molecules move along with the water flow and react with each other in a 2D discrete grid, i.e. a rectangular lattice composed of multiple cells. Every molecule can occupy at most one cell and every cell can host at most one molecule. During the execution of the simulation, every molecule may move to another location, experience adsorption to or desorption from a particular site, and chemically react with other molecules. At the beginning of a simulation, simulation parameters are read from the database into the simulation engine. The total number of molecules in the simulation can be determined by specifying the separate grid size and the molecule density. The simulation time, defined by the user, is divided into a very big number of equal, discrete, and independent time steps. In every time step, every molecule in the system is chosen in random order and the behavior of this individual are determined by a set of rules.

IV. SIMULATION ANALYSIS

We analyzed the performance for this simulation model using the Optimize it tuning device.

A. Data Structure

Selecting the appropriate data structure is important in a scientific function. Different data structures have significant impacts on the

presentation No single data structure is suitable for all situations. The best way to decide that type of data structure to use in an function is to create some benchmarks that reflect how the structure is used in the function. In the NOM simulation model, every molecule object needs to be held in a collection. Individual molecule objects are accessed randomly at every time step. They can also be added or removed from this group. Java 2 platform provides two types of List structures: Array List and Linked List. The Array List is a random access list, and the Linked List 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 during 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. But, the remove operation for the random access list is more expensive than for sequential access list [22].An algorithm for manipulating random access lists can produce quadratic performance 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 every time step. The collections class in Java 2 platform provides a shuffle technique to randomize the order of objects in the List. The shuffle algorithm used in the implementation has a linear time complication for random access lists and quadratic complication for sequential access lists. In order to avoid the expensive operation that would result from shuffling a sequential access list, the shuffle technique converts the list into an array before executing shuffling and converts the shuffled array back in the list. Three benchmarks have been produced in order to test the performance, using different List structures and operations. These benchmarks return exactly how the data structures have been used in the NOM simulation model. In benchmark A, the Molecule List is implemented using Array List, objects are accessed using get(), and Molecule List is randomized using shuffle(). In benchmark B, Molecule List is implemented using Linked List, 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 every benchmark, the add and remove operations are measured, and the overall presentation is measured to reflect the NOM function behavior.

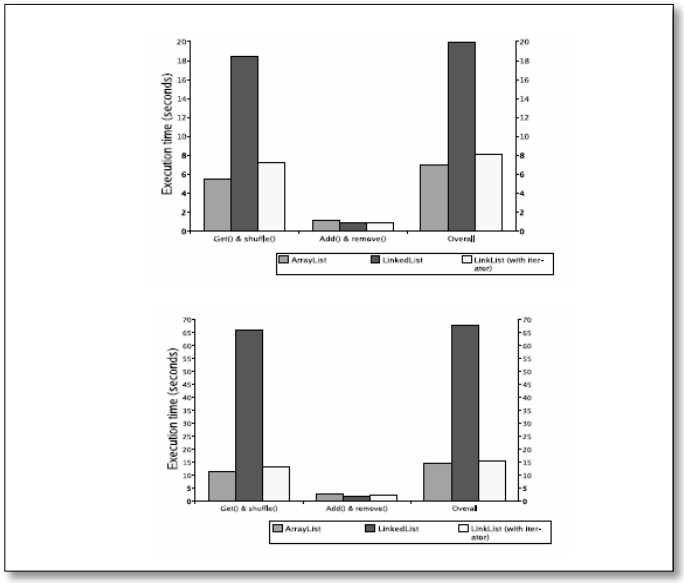

Fig1: presentation comparison of different operation for Array List & Linked List, the Simulation runs for 500 time steps. Top: Net size is 400X300. Bottom Net size is 800 X 300.

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 function, molecules are uniformly distributed on the grid. Doubling the grid size, therefore, will double the molecule number and the List size. The overall performance gain for Array List has been balance by add and remove operations. By doubling the grid size, the time for randomly accessing the Linked List increases more than 6 times. For that particular function, the Array List is the best choice for implementing the Molecule List. That can get a maximum 2.8 speedup in this benchmark with

grid size 400 X 300. When the grid size becomes big, the speedup increases.

B. Object reuse

Objects play a very important role in Object-Oriented programming, C++, and Java. The Java Virtual Machine (JVM) can automatically control memory using the "garbage collection" device. Java developers can allocate objects as necessary without considering de-allocation, while C++ programmers must manually specify where an object in the heap is to be broken by coding a "delete" statement. Excessively creating objects not only enhances the memory footprint and CPU time for garbage collection, it also enhances the possibility of memory leak. Sosnoski (1999) [23] 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. That 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 a variety of techniques to automatically discover objects when they are no longer being referred to and to recycle the memory periodically. In spite of the automatic nature of the garbage collection process, a potential disadvantage is that it adds an overhead that can change program presentation, even in some JVMs such as Sun Hot Spot 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 [24]. For example, in some situations, a programmer needs to set the object into "null" manually in order to dereference the object. While 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 produced 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 produced and destroyed frequently are compatible with that technology. For a big scale function, using a powerful profiling device is necessary to help developers detect this kind of object. Optimize It has been selected in the development process of the NOM simulation model. In the NOM simulation model, at every time step, a definite number of molecules can enter into the system or leave the system. The molecule objects need to be produced and destroyed frequently and they are candidates for potential reuse.

• Optimizing object size

The size of objects not only has an end product 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. An exchange can be made by either reducing the object size or keeping the accuracy.

• Reinitializing object

Although Sun's Hot Spot VMs radically enhanced the presentation 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 formation or re-initialization of the *Molecule* objects in the NOM simulation model. In this benchmark, 1000 *Molcule* objects are produced, then reinitialized to their original state. The average construction time for these *Molecule* objects is 53.85 milliseconds, while the average reinitializing time is 3.9 milliseconds.

• Creating object pool

In order to reuse certain types of objects, that 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, Array List, or a raw array. Selecting a suitable type of data structure depends on the type of operations used for supervision 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 produced, the technique 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 produced. 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 Linked List. Object reuse is a simple and elegant way to conserve memory and increase 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.

C. Database Connection and Database

Query For a function written in the Java programming language, communication to a database can be 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. JDBC offers several advanced techniques that allow the programmer to write high presentation queries [25]. These methods are presented to show the significant impact on presentation and scalability.

• 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 [25]. 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 Prepared Statement object. Unlike the Statement object, which needs to be sent to a database for parsing each time, the Prepared Statement 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 big scale scientific function, the function and the database normally reside on physically distributed equipment. 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, yet, is not necessarily faster when using the batch update approach instead of the Prepared Statement. 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 produced using

JDBC, the database is set in auto commit mode and every SQL statement is treated as a separate transaction. In order to allow two or more statements to be grouped into a transaction, the auto commit mode needs to be disabled using *Connection. Set Auto Commit (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. Therefore results in data that is not consistent in the database. Explicitly committing a group of SQL statements is not only a safe approach, but also has big presentation 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 is stored in the database at the end of every time step. The insertions for the data of all the reacted molecules at every time step are treated as one transaction to ensure data regularity in the system. The NOM core simulation engine connects to the Oracle database using the JDBC thin driver. Figure 2 shows the presentation 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, every insertion for every reacted molecule was treated as a single transaction with a Statement object.

In case 2, every insertion for every reacted molecule was treated as a single transaction with a Prepared Statement 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 Prepared Statement object.

In case 5, a batch updates approach is applied.

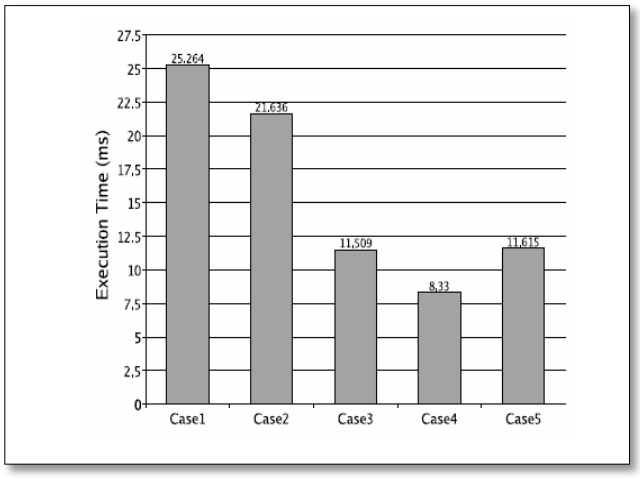

Fig2: Comparison of data insertions using five approaches in NOM Simulation model.

Comparison of data insertions using five approaches in NOM simulation model Case 4, the Prepared Statement object with transaction management has the best presentation in the NOM application. It offers 3.03 times speedup relative to case 1 in this benchmark.

D. Parallel Data Output with Java Threads

Large-scale scientific applications always involve large data sets output. It is not only CPU bound processes, but also I/O bound processes, especially when the database server and the function server are on different machines. That is a common architecture design in large-scale scientific applications. If the simulation server has multiple processors, multithreading can be used to overlap the calculation and communication. Most 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, big amounts of data need to be written in the database at each time step. The average time for one record insertion using Prepared Statement 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**.**

By using a separate thread for data output, there is average 1.3 speedup relative to the single threads model for the NOM application.

E. 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 [26]. Sun Micro System implements two types of Hot Spot JVM, Client VM and Server VM, to meet different requirements for different function Compared with server side programs, client side programs often require a smaller RAM footprint and have a faster start-up time. These two Hot Spot JVM shares the same runtime portion, but the main differ-

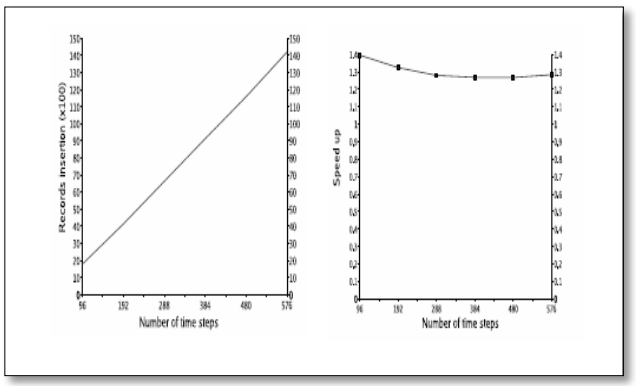

Fig3: overlap the computation and I/O using Java threads. Left Fig shows that the number of data insertion over the time steps. Right Fig shows that the speed up of using a separate thread for data out-put.

Difference between them are 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 [22]. The application of the NOM simulation model has been benchmarked using two different runtime modes of Sun JVM 1.4.1 01 on Red hat Linux 8.0 for 500 time steps and 1500 time steps. Figure 4 shows that as the net size and the time steps increase, Server VM offers higher performance than Client VM. The result shows that choosing different JVMs can produce significant differences in the presentation.

Ladd (2003) [8] showed benchmark results for both Sun and IBM JVM with versions 1.3.1 and 1.4.1 01on a Linux platform. According to Ladd, the JVM version 1.3.1 has a higher presentation level than version1.4.1 and the IBM JVM has a best presentation level than Sun JVM. Choosing the appropriate JVM for a particular scientific function also involves the consideration of the hardware architecture and the operating system.

F. Scalability

The scalability of the NOM simulation model involves two aspects, the required total number of simulation time steps and the net 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 MPI Java 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 every 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.

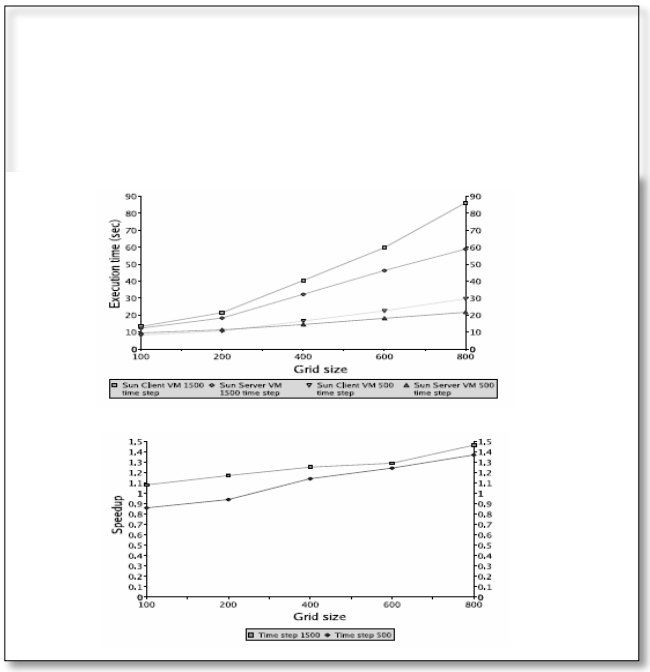

Fig4: Presentation component for NOM application run in Sun client VM and Sun Server VM with different net sizes.

G. Java Threads Version

In order to parallelize the programming model, the original net has been equally separated into two subset grids (e.g., a 800X300 grid is separated into two 400X300 grids). Two threads are created every thread has its own net 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 net and placed into a local buffer in the current thread. At the end of the time step, a hurdle 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 nets and two *Molecule Lists*. Threads have been synchronized before that thread finishes the setting of boundary.

H. Performance Results

In order to measure the presentation 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 Red Hat Linux 8.0 Operating System. The Java codes are implemented and compiled using Java Development Kit and executed on SUN's Java Virtual Machine**.** MPI Java has been used to build the execution atmosphere. The NOM function runs on 2 and 4 machines.

The presentation 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 net and *Molecule List* maintenances offset the presentation gained by distributing the job to different computers when the problem size is small.

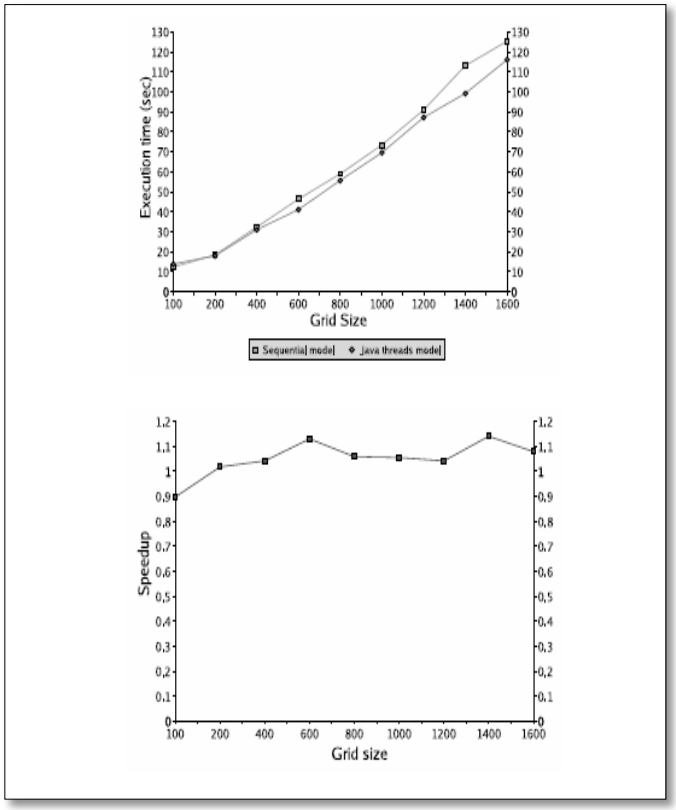

Fig 5:- Compare the performance between Sequential programming model and Java thread model in the NOM Application (one thread Vs two threads on a single dual CPU computer)

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

V. CONCLUSION

In this paper, several approaches for exploring the presentation and scalability of development of a typical scientific function "the NOM simulation model" have been presented. These approaches are summarized as follows:

Data structure 2.8.1 using Array List provides higher overall presentation than using Linked List in Benchmark object reuse - reduce the memory footprint, the garbage collection cycle, and the overhead of object allocation and deal location. The presentation gain is relatively small compare to the overall computation time in the NOM simulation model. JDBC 3.1 JDBC tuning can affect the presentation of data I/O. Parallel data output 1.3.1 overlap the computation and I/O using Java threads can improve the presentation. Java runtime 1.4.1 Sun Server VM has higher presentation than Sun Client VM for large-scale scientific applications. Java threads model 1.1.1 presentation of Java threads model largely depends on the JVM implementation and how efficient the operating system handle the threads. MPJ model 1.5.1 distributing the job to 4 nodes in a cluster has relative larger presentation gains when problem size is big and the communication between nodes is minimized. However, it is still much lower than the ideal presentation gain (4).

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

Program Profiling is applied for overall performance gain. Number of CPU cycles and memory usage, need to be monitored. Appropriate runtime environment for a particular function is very important for better result. Besides JVMs that is evaluated here, IBM JVM too needs to be investigated. Java built-in threads needed to parallelize the Java applications is found to be convenient. The performance gain by parallelism 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. We can extend these experiments to Windows operating system or Solaris operating systems too. Distributing jobs on multiple machines in a cluster environment is an efficient approach for large size problems. But, the communication among nodes and the net and list maintenances *offset* the

Presentation gain. 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. Hence, we can distribute the job on multiple machines using MPJ and combine the results at the end of the simulation in the database.

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] Swarm development group. http://www.swarm.org.

[4] Yung-Hsin Wang and Szu-Hsuan Ho. Implementation of a devs-javabean simulation envi-ronment. In *The 34th Annual Simulation Symposium* pages 333–338, Seattle, Washington, 2001.

[5] Peter H.M. Jacobs, Niels A. Lang, and Alexander Verbraeck. D-sol: A distributed

java based discrete event simulation architecture. In *The 2002 Winter Simulation Conference*, pages 793–800, 2002.

[6] Repast. http://repast.sourceforge.net/.

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

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

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

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

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

[12] 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.

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

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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. [20] 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.

[21] Steve Cabaniss. Modeling and stochastic simulation of nom reactions, working paper, July 2002.

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

[23] 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.

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

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

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