Developing a Practical Parallel Multi-pass Renderer in Java and C++

— Toward a Grande Application in Java —

Hitoshi YAMAUCHI[*]
Denkituusin University, GSIS
Tokyo, Japan
yamauchi@archi.is.tohoku.ac.jp

Atusi MAEDA[†]
Denkituusin University, GSIS
Tokyo, Japan
maeda@is.uec.ac.jp

Hiroaki KOBAYASHI
Tohoku University, GSIS
Sendai, Japan
koba@archi.is.tohoku.ac.jp

ABSTRACT
In the area of parallel processing, performance has been the primary goal, and historically, parallel software writers paid less attention to software portability. However, as software is becoming more complicated, costs for developing and maintaining parallel applications are rapidly increasing. Reusable and portable software is certainly needed even in the parallel processing area. Java appeared on the scene under the slogan of "Write once, run anywhere", advertising portability as its largest advantage. Java Grande Forum was established to achieve two goals; portability and high-performance.

Current Forum discussions seem to concentrate on optimization of Java programs, elements of numerical libraries, message passing interface for Java, etc. Few implementations of practical applications are presented so far. To find out obstacles in writing Grand Challenge applications in Java, empirical studies on developing large and practical applications in Java are strongly desired.

As an example of practical distributed parallel applications, we have implemented two versions of a parallel multi-pass rendering system. One version is written in C++ and the other is written in Java. The multi-pass rendering method is a combination of radiosity and ray-tracing methods. These implementations, about 56,000 lines in total, are publicly available at
`http://www.archi.is.tohoku.ac.jp/research/cg/`. These two programs are based on the identical algorithm and are directly comparable in terms of performance and efficiency in software development. Experimental results on Sun Enterprise with JDK 1.2.1 and gcc 2.7.2 which is used only for compiling message passing library show that compared to the C++ version, the performance of the Java version is about three to five times slower and requires approximately four to seven times more memory space. We also discuss some problems encountered in developing practical parallel distributed applications in Java.

*Current affiliation: Max-Planck-Institut für Informatik, Saarbrücken, Germany

†Current affiliation: University of Tsukuba, Science Information Processing Center

ACM 2000 Java Grande San Francisco, California, June 3-4, 2000

1. INTRODUCTION
Early researchers on parallel computers paid virtually no attention to program portability. Since parallel processing is a promising approach to solve a wide variety of complex or huge problems, the primary concern of parallel processing research is to achieve high-performance. However, it is hard to develop high-performance software, while keeping their portability. In high-performance software developments, special hardware supports are often assumed. Dedicated programming languages or programming environments depending on particular parallel platforms are fairly common.

On the other hand, as software becomes getting more complicated and larger, costs for development and maintenance of programs are getting more and more expensive. In contrast, performance improvement of commodity products makes high-performance hardware widely available at lower cost. As a result, it is not a cost-effective idea to modify programs whenever they are ported on new platforms. In other words, although performance is still a matter of greatest importance, software portability is also becoming a crucial issue.

New standards for Fortran, especially standardization of communication libraries such as MPI [16], and standardization for parallelizing compiler directives such as OpenMP [18] are typical examples of efforts to solve this portability problem. Enhancement of existing languages, however, cannot completely remove environment-dependent part of the system. It is still very hard to pursue higher performance and platform independence at the same time.

In this situation, Java appeared as an ultimately portable language. At first, applications in Java for numerical computing were impractical because early Java programs ran quite slow. However, its high portability was so attractive that some eager attempts were made to exploit the capability of Java in the field of scientific computing [7]. Activities of Java Grande Forum are the most noticeable movement among them. In Java Grande Forum, problems and their possible solutions in developing numerical computation applications called Grande Applications are being discussed.

Java is a new language and is still making continued progress in its sophistication. Research interests tend to be directed to optimization of serial, relatively small programs. Of course, researches on parallel Java applications are being performed, though, practical and large-scale parallel applications in Java are not yet ready for prime time.

In this paper, we present a large scale parallel application written in Java. We believe it is particularly valuable to obtain empirical knowledges and experiences in building and using practical applications in Java in this immature stage.

We have implemented photo-realistic image synthesis programs based on the integration of radiosity and ray-tracing in both Java and C++. The program is parallelized for execution on a message-passing parallel computer. Both of implementations use the identical algorithm and suitable for direct performance comparison between the two languages.

The rest of the paper is organized as follows. In Section 2, we will show the brief description of a parallel multi-pass rendering algorithm which we implemented here. In Section 3, we describe implementations of the parallel multi-pass rendering method in Java and C++. In Section 4, we will show the experimental results of both implementations and also discuss problems for developing Grande Applications in Java. Section 5 concludes the paper with a summary and some directions for further investigations.

2. A PARALLEL MULTI-PASS RENDERING METHOD: INTEGRATION OF RADIOSITY AND RAY-TRACING

2.1 Multi-pass rendering Method

The *radiosity method* and the *ray-tracing method* are algorithms that generate an image by constructing a virtual space on a computer and simulating propagation of light among objects modeled inside the space. The principal formula is called *the rendering equation*, which is given by [12] as follows.

$$\begin{aligned} I(x,x') &= g(x,x')\ [\ \epsilon(x,x') + \\ &\qquad \int_{\Omega} \rho(x,x',x'')\ g(x',x'')\ I(x',x'')\ dx''] \end{aligned}$$

Here, $I(x,x')$ is the amount of light energy propagated from point x' to point x, $g(x,x')$ is the geometry term representing visibility from point x' to x, $\epsilon(x,x')$ is the amount of light energy radiated from x' to x, and $\rho(x,x',x'')$ is the bidirectional reflection distribution function that denotes the ratio of rays from x'' to x via x'.

To generate photo-realistic images, we must solve the rendering equation as accurate as possible. Applications based on these algorithms include lighting and landscape simulations that require physically accurate calculations. Recently, they are also used in computer games, arts and so on. Due to physically accurate computations involved, however, these algorithms are known to be very time-consuming, and this feature prevents photo-realistic image synthesis based on the rendering equation from practical use.

Of the two algorithms above, ray-tracing [23] may be more popular as a method for generating photo-realistic images. What we call "ray-tracing algorithm" uses a set of a screen and a viewpoint to generate an image, and traces rays from the viewpoint to light sources with reflection and/or refraction with objects. This method is actually referred to as *backward ray-tracing* since rays are traced from a viewpoint instead of light sources. In practice, this method does not have significant advantages unless rays are traceable toward light sources. In other words, surfaces need to be assumed as specular elements in a modeled environment. If diffuse surfaces are encountered during the backward tracing, further tracing of the ray becomes difficult since diffuse surface does not preserve the state of a ray prior to reflection. In short, backward ray-tracing is an approximation of the rendering equation that assumes reflection coefficient as mostly specular.

In ray-tracing, it is difficult to simulate bleeding, which is an effect of ray propagation among diffuse surfaces. To compute the effect of bleeding, the radiosity method [5] was proposed. In the radiosity method, objects are divided into small discrete pieces of planes called *patches*. Diffuse reflective propagation of light energy among patches are computed. Concretely, an entire environment is sampled by ray-tracing with setting a viewpoint on each patch and putting a small screen called hemi-cube in front of it. Light energy that affected a patch is gathered and accumulated on it from the whole environment, and reflection energy is radiated again into the environment after multiplying the patch's reflectivity. The sampling process is repeated until radiated energy becomes lower than a predetermined threshold. In this way, the radiosity method computes transport of energy among diffuse surfaces, and an image is generated by translating energies of visible patches into intensities. Since the radiosity method samples energy of light against the entire environment, it can approximate the rendering equation by assuming that reflection on patches is diffuse reflective.

The multi-pass rendering method [24] is an algorithm that combines both of the two methods to precisely capture light propagation among both diffuse and specular surfaces. In this paper, the implementations of our parallel renderer are based on the multi-pass rendering method.

2.2 Parallelizing Multi-pass rendering Method

The multi-pass rendering method is a combination of the radiosity method and the ray-tracing method. The radiosity method computes propagation of light energy in an environment mainly consisting of diffuse surfaces, while the ray-tracing method calculates propagation of rays in an environment that mainly contains specular surfaces. In the multi-pass rendering method, the propagation of rays among diffuse surfaces is computed by radiosity first, and the equilibrium of light energy in the diffuse environment is obtained.

After that, rays propagated through specular surfaces are calculated by ray-tracing. In the ray-tracing phase, the results of radiosity computation are used for global illumination calculations. In this way, the multi-pass rendering method has a capability to handle global diffuse reflection by radiosity and specular reflection/refraction by ray-tracing, and can generate photo-realistic images.

However, the radiosity and ray-tracing methods are both known as time-consuming algorithms. Since the multi-pass rendering method is a serial combination of these two, it consumes even more time. Some attempts for parallelizing this method have been proposed to accelerate the computations of the multi-pass rendering method.

In radiosity, a visibility between any patches called *a form factor* is computed at the time of energy exchange. Several methods to exploit parallelism in this form factor computation are well known [19, 21, 3]. These methods make use of the fact that it is possible to test the visibility for each patch independently when testing which patches exchange energy each other. In standard radiosity methods, a virtual viewpoint with a virtual screen is set on each patch with radiating energy. For each pixel of the virtual screen, there exists an opportunity of independent computations.

In ray-tracing, it is widely known that computations of the rays that pass through each pixel can be computed independently and thus can be performed in parallel [1].

Therefore, both of parallel radiosity and parallel ray-tracing basically take advantage of the same source of parallelism; the independence of sampling at the time of visibility test.

In conventional parallelizing methods, basically no knowledge about distance between objects is used for testing existence of interaction of light between the objects. When one object is selected, the whole environment has to be scanned to search objects interacting with the selected one. Space subdivision methods, e.g. octree [9], were proposed to make scanning range small. In ray-tracing, a ray issued from a viewpoint or an object with reflection searches an interacting object. In space subdivision methods where a whole space is subdivided into subspaces, the searching area is limited from the whole space to some small set of subspaces along the ray. Similarly, the radiosity method can employ a space subdivision method for visibility test to restrict the scanning range as in ray-tracing.

In radiosity, radiation light from one object influences a lot of the other objects since the method treats diffuse reflection. Therefore, each object accesses a wide range of the environment in the radiosity method. In ray-tracing, it is hard to predict the direction of rays due to reflection and refraction. Therefore, access patterns of rays to the environment become highly irregular [1]. Therefore, in both methods, an object may access the other objects in the environment to exchange energy. As a result, when a ray's starting point and its direction are decided, the objects searching area can be squeezed by the space subdivision method. However, as we cannot know the information about the searching area before calculation, each processor of a parallel system must be able to access the whole environment to find intersecting objects visible on a screen. This means that the all processors share the one environment.

If this type of parallel algorithms is implemented on a parallel processing system with a shared memory, memory conflicts will occur frequently on the shared memory as the number of processing nodes increases, and it is hard to achieve linear speedups in the calculations. On the other hand, this could be implemented on a distributed memory parallel computer with a message-passing mechanism, where objects are distributed to each local memory. Since this type of parallel processing causes a large global communication overhead, we cannot expect to achieve linear speedups when the number of processing nodes are increased. Another method based on broadcasting was proposed [6]. In this method, a node called "host" broadcasts a ray's information to all the processing nodes with objects' information. However, the host computer will become the bottleneck of the system performance. We classify these problems into a "object-sharing problem" since these problems are caused due to logically shared objects' information.

We thought that the object-sharing problem is caused due to the fact that the conventional parallel image synthesis algorithms usually extract parallelism from standard sequential image synthesis algorithms. The computation models of sequential algorithms assume the same access time to any address like the RAM (Random Access Machine) [14]. However, on practical parallel processing machines, access time is not constant usually. A remote memory access leads to a higher cost than an access to local memory, and memory access conflicts on a shared memory will worsen this situation.

To solve the object-sharing problem, we noticed that the rendering equation holds in each subspace even if an object space is subdivided. Therefore, we subdivided a space to subspaces regularly and distribute them among parallel processing nodes. Ray propagation is simulated through inter-processor communications. Each processor calculates ray-object interaction within allocated subspaces. In this paper, we call this scheme the object-space parallel processing model.

This method is very simple, but the performance gain of parallel processing will be marginal unless objects are placed regularly or randomly in a scene. The reasons are:

- The occupation ratio of objects in a space is very low in usual scenes. For example, it is a special case that object occupies more than one-half of a space when we design an office or house. Most part of a room has no objects as we can see in an actual room.
- There is strong object coherency along horizontal and vertical directions because of the effect of gravity. We can easily see some examples like a floor in buildings, ceilings, walls, and pillars. In addition, objects like furniture do not exist in the air suddenly, and they are on a floor and a wall. We can also find such examples in the natural world (e.g. trees).

In the end, we can easily see that objects hold a little volume in a space and they are not uniformly distributed. Therefore, parallel processing using simple subspace division and simple allocation cannot achieve a good performance except that objects are regularly distributed or randomly distributed. It may sound inconsistently that both the randomly distribution and the regularly distribution of objects can balance the system load. However, the random distribution means that there is no bias in objects' distribution, and therefore, random allocation of subspaces among processors can also balance the computational loads of the processors as well as the case of regular distribution of objects in a space.

Therefore, when we try to render usual scenes, we must deal with biased distribution of objects in a space. Adaptive space division is one of the solutions to solve this problem. However, it is difficult to allocate such irregular subspace to processing nodes systematically. Moreover, traversing adaptively subdivided subspaces for a large number of rays leads to a high computational costs than regularly subdivided subspaces. That causes another new problem.

In [13, 26], we proposed a static load-balancing scheme based on the object-space parallel processing model. In this static load-balancing scheme, a space was subdivided more finely, compared to the number of processing nodes, and some hashing functions were used to allocate subspaces to processing nodes randomly. At the same time, traverse costs from subspace to subspace was kept low.

Two kinds of network topologies were considered in our implementation of the static load-balancing scheme. In [13], ring, mesh and 3D torus topologies were taken into account. Network topologies with constant node distance, like a multi-stage network, was assumed in [26].

In this paper, we implemented this parallel multi-pass rendering method based on the object space parallel processing model in both Java and C++, and examined them in terms of execution time and memory space.

3. IMPLEMENTATION OF THE PARALLEL MULTI-PASS RENDERING ALGORITHM

We implemented parallel multi-pass rendering programs named **mpi2C++** and **mpi2Java**. Libraries and languages that are used for these two implementations are as follows.

- **mpi2C++** is written in C++ and uses MPI (Message Passing Interface) in C as a communication library.
- **mpi2Java** is implemented in Java and uses MPI in C as a communication library. A wrapper library called `mpiJava` [2], which enables Java programs to call MPI library written in C, was modified and used in this implementation.

C/C++ compiler and Java compiler we used for implementations are gcc version 2.7.2 and JDK 1.2.1 production release, respectively. We adopted `mpich`[17] version 1.1.2 as the MPI library. The `mpiJava` version is 1.2beta with some modification by the authors.

mpi2C++ contains an implementation of a parallel volume rendering method in addition to the multi-pass rendering of radiosity and ray tracing, and the Java version can process multi-pass rendering only. Thus, although we cannot directly compare the code size of the Java version with that of the C++ version, we have counted the total number of source code lines of two programs using UNIX command **wc**. The source code sizes were 31996 lines in **mpi2C++** and 23833 lines in **mpi2Java**. The Java program can use Java's rich set of standard libraries and tend to require less code size. For example, the current implementation in Java uses class `Map` (`HashMap`) to keep track of object information. When we had implemented the C++ version, however, the stable implementation of class libraries such as STL had not been readily available. Therefore, we had to implement a table management code by ourself, which resulted in an increase in code size.

On the other hand, some fundamental classes are not available in Java class libraries. There is an implementation of `Stack` based on class `Vector`, but class `Queue` is not included in the libraries. We first implemented a `Queue` using class `Vector`, but in this implementation, a dequeue operation takes $O(n)$ time where n is the number of elements in a queue. This is the wrong implementation. Current version uses `Queue` based on ring buffers developed by the authors, and this takes only $O(1)$ time.

4. EXPERIMENTAL RESULTS AND DISCUSSION

4.1 Performance evaluation parameters

Table 2 shows the scene parameters of our experiments. The number of triangle polygons are 100,000 to 220,000 in these test scenes. The original geometry data can be obtained from Radiance web site [25]. The experiments are carried out on a Sun Enterprise E3500 (UltraSPARC-II 336 MHz $\times$ 8) running under SunOS 5.6 with 1.5 GB memory. Although this computer is a multi-processor machine with a shared memory, our implementation does not use shared memory primitives and each process uses a message passing library for communications. Therefore, the address space of each process is completely separated from the other processes. As a Sun Enterprise has multiple memory modules and we expect different process uses different memory module, the object-sharing problem will be avoided under this situation.

In our implementations, transmission of ray's information through ray-packets is needed to synthesize images. Also in our implementation, ray-packets are buffered when they are sent or received, and users can control the buffer size.

Table 1 shows a point-to-point communication performance of the system using the ping-pong scheme [10]. Throughputs are improved as buffer size increases, however, the saturation of performance improvement is found when the size of buffer exceeds 2^{10}KB. Moreover, because buffers with too large size stall ray propagation in the parallel multi-pass rendering system, proper buffer size exists. From our preliminary experiments, we found a suitable buffer size of around $64(=2^6)$KB, and this buffer size is adopted in the experiments.

Table 1: Throughput of Point to Point Communication (MB/sec) on a Sun Enterprise with mpich

Buffer size(KB)	2^2	2^4	2^6	2^8	2^{10}	2^{12}
Throughput (MB/sec)	17	39	60	67	73	75

Figure 1 shows the rendering images created by **mpi2Java**. When a scene and the number of processing nodes are fixed, identical inputs are given to both **mpi2Java** and **mpi2C++** for comparison of the Java implementation with C++ implementation.

4.2 Results

Figures 2 and 3 show the results in elapsed time in each parallel rendering implementation. We use the `System.currentTimeMillis()` function for the **mpi2Java** and the `gettimeofday()` function for the **mpi2C++** to measure the elapsed time. Since both functions can get the current time of day, we can measure the *"real"* processing time including all kinds of overheads. In this measurement, we use three scenes to evaluate performance of both implementations on 1, 2, 4 and 8 nodes.

As both **mpi2Java** and **mpi2C++** are based on the same algorithm and have almost similar configuration of source codes, we think that the comparison between two implementations is meaningful. From the experiments, the elapsed time of the Java with JIT version was 3 to 5 times longer than that of the C++ version. In addition, the Java version without JIT takes longer elapsed time in comparison with the Java with JIT version. In the end, the Java implementation without JIT takes 9 to 25 times longer CPU cycles than the C++ implementation.

We also examined the memory requirement for the execution of the Java and C++ versions by using the unix **ps** command. Table 3 shows the memory consumption in scene *"office"*. This table includes source code size and execution image size : binary image size of C++ and class file size of Java. *RPS (Relative process size)* in the table is calculated using the following equation.

$$\text{RPS} = \frac{\text{process size of Java implementation}}{\text{process size of C++ implementation}}$$

The average dynamic memory consumption sizes are calculated under the same number of nodes and the same scene. According to our results, Java's static bytecode density is four times higher than the C++ binary code density. Although Java bytecodes are usually translated into machine codes by JIT at runtime, there are some processors (e.g. `PicoJava`[15]) that can directly execute Java bytecodes as native codes. On such platforms, the size of a Java class file directly relates to the code size at runtime. Even though Java is profitable in terms of static code size density, the dynamic execution image size of the Java version is huge, and this is a serious disadvantage of this Java implementation. The Java version consumes 4 to 7 times more memory space in comparison with the C++ version.

In addition, we measured elapsed time of reading modeling data into the memory in order to examine I/O performance.

Table 3: Memory Requirement of mpi2C++ and mpi2Java per node (Office) and their RPS (Relative Process Size)

# of Nodes	1	2	4	8	code size
C++ (MB)	58	34	27	19	1.6 (MB)
Java (MB)	247	233	129	80	0.4 (MB)
RPS	4.3	6.6	4.8	4.2	—

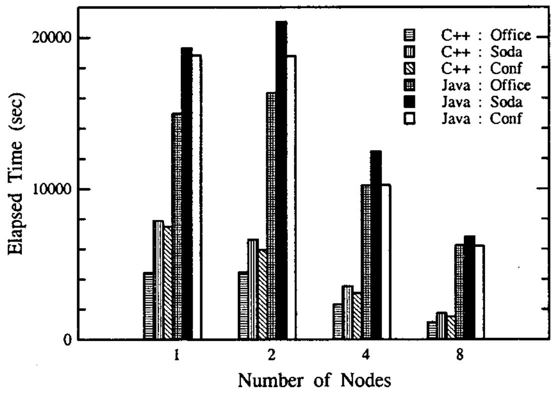


Figure 2: Elapsed Time of Parallel Radiosity

The results are shown in Table 4. We found that the UTF conversion filter works when reading an ASCII file. We also measured this effect. Experimental results show that the overhead of the UTF conversion filter reaches 10 to 20% of execution time for reading. If you do not want to use the UTF conversion filter, you specify encoding "ASCII" at class constructor `InputStreamReader` in your code Explicitly, or set the locale environment to "C". (i.e., type `setenv LANG C` in the csh environment.) It is a good idea to set the encoding ASCII if you can assume an input file is always in an ASCII format.

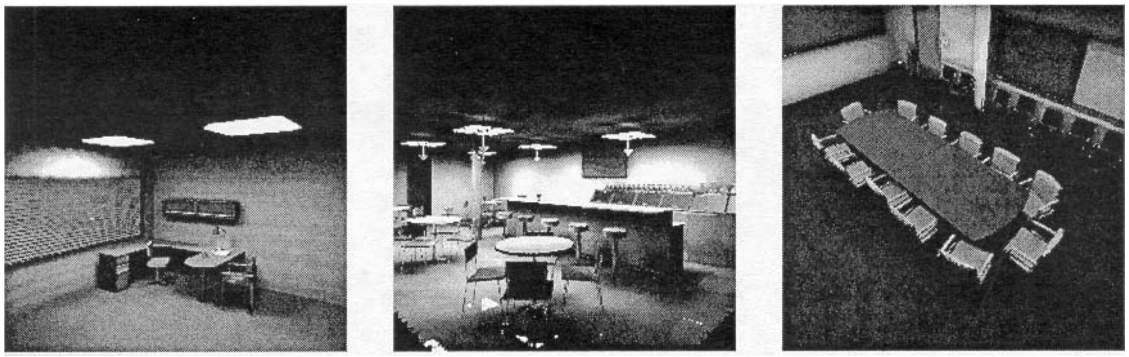
In the experiments, the execution time of the radiosity method is 25 times longer than that of the ray-tracing method. The behaviors of both the Java and C++ programs have almost a similar tendency in changing the number of nodes and varying scenes. In other words, we cannot find any qualitative differences between two implementations. Remarkable differences between them are in only quantitative aspects like execution time and quantities of memory consumption.

4.3 Discussion and Future Work

Through our measurements with Sun JDK, differences in performance and memory efficiency in the case when the same programmers implemented same algorithm is demonstrated. Although behaviors of small Java programs had been studied to some extent, results on large scale programs are hardly seen in the literature. In this paper, a practical application was implemented in two languages; Java and C++, and comparison of the behaviors of the two programs

Table 2: Parameters of Test Scenes

Test Scene	(a) Office	(b) Soda Shop	(c) Conference Room
Number of Patches	102,824	133,668	226,621
Number of Subspaces	64 x 64 x 64 (= 262,144)		
Hemi-Cube Resolution (pixels/top surface)	40 x 40		
Max Number of Reflections	3		
Screen Size (pixels)	512 x 512		
Number of Sampling Rays Per Pixel	4		
Ray Coalescing Factor	256, 512		
Radiosity Energy Convergence Tolerance	83 %	75 %	84 %



(a) Office (b) Soda Shop (c) Conference Room

Figure 1: Test Images

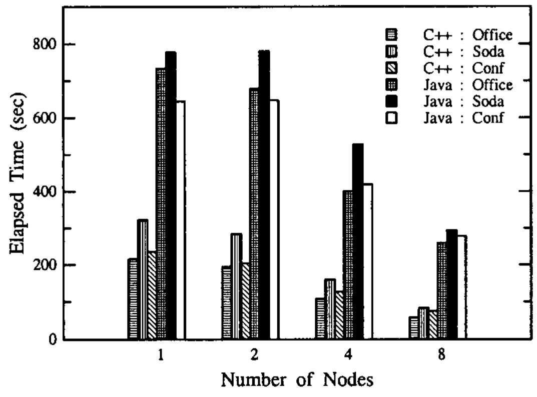


Figure 3: Elapsed Time of Parallel Ray-Tracing

Table 4: Elapsed Time of Data Read

Scene		(a) Office	(b) Soda	(c) Conf
Size of data file (MB)		7.6	10.7	19.1
C++ (sec)		5.6	7.4	12.6
Java (sec)	JIT	28.6	36.3	63.5
	no JIT	263.2	352.1	614.8
	UTF & JIT	34.0	39.4	71.2
	UTF & no JIT	315.3	417.2	726.4

are made. In terms of processor performance and I/O performance, Java implementation takes three to five times more execution time in comparison with the C++ implementation. When JIT is turned off, the Java version requires 9 to 25 times more elapsed time.

Our message passing library for Java version is based on mpiJava which is implemented by using the JNI (Java Native Interface) [22] for the MPI library. In most of cases of using JNI protocol, each element of a data array must be copied to an array of communication buffer instead only informing the start pointer of a data array to the con munication system. This data copy processing will be a overhead. It is a natural question that the overhead caus the main difference of performance between Java version an C++ version. However, figures 2 and 3 also show the ca of the one node. In our implementation, the communic tion library is not called when the number of nodes is on In such a case, the difference of elapsed time between C+ and Java version is around three. Therefore, the overhea of calling communication library is not dominant elemen However, in C++ version, there is some performance in provement from one node to two nodes. On the other han in Java version, we can see some performance degradatic from one node to two nodes. Accordingly, the overhead calling communication library of Java version seems som what larger than C++ one. We need more detailed obse vation of the overhead of calling communication library i future.

Even when taking high productivity of Java into accoun three-fold performance degradation in numerical computation is almost unacceptable. However, Java Grande Forum is working on extensions of Java language to enhance Java application performance. We are going to introduce these extensions into our implementations and examine their performance.

Java Grande Forum also proposes a number of benchmark programs [11, 4, 20]. One of the benchmark suites called Scimark2.0 [20], for example, consists of applets that execute FFT, Sparse Matrix Multiply, Monte Carlo integration, and SOR method. This benchmark suite, however, completes execution within only one minute on Pentium Pro 266MHz. Presumably, this is because one of the purposes of the suite is to gather as many results as possible from a wide variety of machines. However, with the goal of Java Grande in mind, the scale of the program seems to be too small. The benchmark used in [8] is also small as a Grande application; it is one of the class A programs in NASPara suite that sorts an array of 8M integers. The Ray-Tracing problem in Large Scale Applications category of the benchmark suite ver. 2.0 [11] proposed by Java Grande Forum Application and Concurrency Working Group (JGACWG) contains only 64 objects. In contrast, the number of objects in each scene we used here is in the order of 100,000 to 200,000. The Ray-Tracing benchmark program of JGACWG in Large Scale Applications uses RMI for the distributed ray-tracing method and can be used to figure out common problems in distributed computing with Java, but the size of the problem is too small in the criteria of current CG research as a scene rendering problem. To fill up the lack of large scale problems, we are planning to open the software we used in the experiments to the public and to propose it as one of the benchmarks. We are certain that we must explore a possibility of Java in practical and large-scale problems in the future.

Even though performance is obviously a great concern, memory consumption was also a serious problem. In the numerical computation area, many of applications consume a large amount of memory. Such a large memory consumption as shown in the experiments restricts the applicability of Java in this area. Even if systems with garbage collection may work well when a relatively larger memory space say, two to three times larger memory area than actually live data size is given, 7 times larger memory consumption observed in our experiments is almost hopeless. In many programs including ours, memory requirements effectively limit the size of solvable problems. The cause of this large memory footprint is not clear at present. It may be due to fragmentation, object header or other Java-specific storage overheads, or hidden object references created implicitly inside the class libraries. There is a possibility of memory leak in our version of the communication library, but it is unlikely because memory consumption was stable throughout hours of computation time after rapid growth in startup. We coded carefully to avoid references to unnecessary objects. For example, we explicitly assigned null to the unused indices (i.e. not in the range between head and tail) of arrays that keep objects in ring buffers. Otherwise,the garbage collector cannot reclaim objects pointed to by these array portions. It is surprise for us that memory consumption was so large in spite of these careful coding. Research to save runtime memory consumption might perhaps be given higher priority, since memory consumption is currently more restrictive than speed in large scale computing in Java.

There is a possibility that the Sun's Java implementation holds more large size memory than real use. Then, our observation with unix **ps** command could be misleading. However, our parallel process was sometimes finished because of memory exhausting on the Sun Enterprise. From a practical standpoint, the observation by **ps** command may be one of the first order approximations. Of course, more detailed observation for memory consumption of Java is needed. We hope our implementation helps such a study for a grande application.

5. CONCLUSIONS

Previous work in scientific computations in Java lacks the following viewpoint:

- Comparison of capabilities of Java with those of other language (e.g. C++) through implementation of large scale application by the same programmer.

With regard to this point, comparable performance of Java implementation to C++ implementation is a current target for Java applications to catch up. In addition to pursuit of performance of basic operations such as matrix operations, empirical study on real world applications is essential. For this purpose, we presented an example of practical applications by implementing a distributed parallel radiosity and ray-tracing method in two languages, and compared these two implementations in terms speed and memory requirement. Measurement showed that our Java implementation is approximately three to five times slower in both computation and I/O operations compared to its C++ counterpart.

Benchmark programs currently proposed by Java Grande are relatively small. We believe that our experience provides new insights and contributes to a basis for solving problems that would be encountered when developing large practical applications in Java. Thus, we propose a parallel radiosity method and a parallel ray-tracing method to be included in the future benchmark suite.

It is known that memory consumption of Java programs are large when compared to traditional languages like C++. In our experiments, the difference in memory requirement appeared to be larger than the difference in processing speed. Inefficient memory usage is critical because the amount of memory often determines the maximum problem size that can be solved on a given computer system.

Currently we are planning to carry out some experiments on other platforms in addition to Sun workstations. Besides, to make our system more complete as a practical application benchmark, we are planning to include volume rendering as well as radiosity and ray tracing.

Acknowledgments

This research was partially supported by Grant-in-Aid for encouragement of Young Scientists, the Ministry of Education, Grant No.10780163 and 10780179. We would like to thank the anonymous reviewers for their helpful comments.

6. REFERENCES

[1] D. Badouel, K. Bouatouch, and T. Priol. Distributing data and control for ray tracing in parallel. *IEEE CG & Application*, 14(4):69–77, July 1994.

[2] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and S. Lim. mpiJava: An object-oriented Java interface to MPI. In *Intl. Workshop on Java for Parallel and Distributed Computing IPPS/SPDP*, Apr. 1999.

[3] D. R. baum and J. M. Winget. Real time radiosity through parallel processing and hardware acceleration. *Proceedings of the 1990 symposium on Interactive 3D graphics*, 24(4):67–75, Aug. 1990.

[4] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A methodology for benchmarking Java Grande applications. *Proc. ACM 1999 Java Grande Conference*, pages 81–88, June 1999.

[5] M. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg. A progressive refinement approach to fast radiosity image generation. *SIGGRAPH*, 22(4):75–84, Aug. 1988.

[6] H. N. et al. Links-1 : A parallel pipelined multimicrocomputer system for image creation. *ACM Computer Architecture*, pages 387–394, July 1983.

[7] G. Fox. Java for scientific computing. http://www.npac.syr.edu/users/gcf/javaforcse.html.

[8] V. Getov, S. Flynn-Hummel, S. Mintchev, and T. Ngo. Massively parallel computing in Java. In *Proceedings of MPPM*, pages 112–117, London, Nov. 1997. IEEE Computer Society.

[9] A. S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, pages 160–167, 1984.

[10] K. Hwang and Z. Zu. *Scalable Parallel Computing*. McGraw-Hill, 1997.

[11] Java Grande benchmarking initiative. http://www.epcc.ed.ac.uk/javagrande/. Edinburgh Parallel Computing Centre (EPCC).

[12] J. T. Kajiya. The rendering equation. *Computer Graphics (Proc. SIGGRAPHS)*, 20(4):143–150, 1986.

[13] H. Kobayashi, H. Yamauchi, Y. Toh, and T. Nakamura. A hierarchical parallel processing system for the multipass-rendering method. *IEEE International Parallel Processing Symposium*, pages 62–67, April 1996.

[14] F. T. Leighton. *Introduction to PARALLEL ALGORITHMS AND ARCHITECTURES*. Morgan Kaufmann Publishers, 1992.

[15] H. McGhan and M. O'Connor. Picojava: A direct execution engine for java bytecode. *IEEE Computer*, 31(10):22–30, October 1998.

[16] MPI Forum. *MPI: The Complete Reference*. The MIT Press, 1994.

[17] MPICH - a portable MPI implementation. http://www-unix.mcs.anl.gov/mpi/mpich/.

[18] OpenMP: Simple, portable, scalable SMP programming. http://www.openmp.org/.

[19] D. Paddon and A. Chalmers. Parallel processing of the radiosity method. *Computer-Aided Design*, 26(12):917–927, December 1994.

[20] SciMark2.0. http://math.nist.gov/scimark/.

[21] J. P. Singh and M. L. Anoop Gupta. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, pages 45–55, July 1994.

[22] Sun Microsystems. Java Native Interface. http://java.sun.com/products/jdk/1.2/docs/guide/ibjni/.

[23] W. T. An improved illumination model for shaded display. *CACM*, 23(6):343–349, 1980.

[24] J. R. Wallace, M. F. Cohen, and D. P. Greenberg. A two-pass solution to the rendering equation: A synthesis of ray tracing and radiosity methods. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21(4):311–320, July 1987.

[25] G. J. Ward. The radiance lighting simulation and rendering system. *SIGGRAPH*, pages 459–472, July 1994.

[26] H. Yamauchi, T. Maeda, H. Kobayashi, and T. Nakamura. The object-space parallel processing of the multipass rendering method on the $(M\pi)^2$ with a distributed-frame buffer system. *IEICE Trans. on Info. & Syst.*, E80-D(9):909–918, Sept. 1997.