SingleJava: A Distributed Java Virtual Machine Based on Thread Migration

Jian Su
School of Computer & Computing Science, Zhejiang University City College, Hangzhou, China
Email: suj@zucc.edu.cn

Chong Zhou
School of Computer Science and Technology, Zhejiang University, Hangzhou, China
Email: zhouchonghz@gmail.com

Wenyong Weng
School of Computer & Computing Science, Zhejiang University City College, Hangzhou, China
Email: wengwy@zucc.edu.cn

***Abstract*—A distributed Java virtual machine called SingleJava based on thread migration is proposed in this paper. SingleJava can be used to build a distributed computing framework while keeping itself completely transparent to Java programmers. The main idea underlying is to improve the inevitable Java virtual machine in a Java based software system, i.e., adding a build-in distributed infrastructure to the virtual machine and integrating it into a distributed computing framework. Meanwhile, traditional distributed program are usually difficult to test or debug. SingleJava supports testing and debugging applications on a single node Java virtual machine, such as Oracle Hopspot or IBM J9, and the applications or compiled classes can be deployed directly to the distributed SingleJava infrastructures.**

***Index Terms*—Distributed Computing, Parallel Computing, Java Virtual Machine**

I. Introduction

Distributed computing is an important mean to deal with complex problems which require huge computing power. Java is one of the most popular programming languages since Java applications run not directly on the underlying operation system, but on a Java virtual machine. Obviously, it is meaningful to find ways to extends traditional Java technologies into distributed versions[1-3].

In this paper, we will discuss a distributed Java virtual machine called SingleJava, which is based on thread migration. SingleJava aims to build a distributed computing framework and environment on the basis of Java virtual machine. An interesting feature of SingleJava is the distributed Java virtual machine will be transparent to Java programmers. That means the programmers could code, test, and debug their applications just like on a traditional single node JVM, and finally deploy the applications to a distributed computing environment consisting of a set of SingleJava machines. For well parallelized algorithms, performance will be promoted vastly [4-7].

SingleJava exploits the low coupling feature among thread management structures, distributes tasks by migrating Java threads. When a new Java Thread is started to be scheduled by invoking the Thread.start() method, SingleJava will schedule and distribute the thread to a proper WORKER node.

SingleJava use a distributed Java heap to encapsulate global object accessing operations. Execute engines on different nodes can read or write object fields through a unified interface regardless of whether the object is owned by the node or residents on any other node.

SingleJava also offers a garbage collecting mechanism, as a completion of automate memory management [8-11]. Within a distributed environment, SingleJava uses a garbage collecting scheme with two granularities, Local Collecting and Global Collecting. Local Collecting is caused by a local heap allocating failure and happened to a single node only. While global collecting is caused by user requirement or heap allocating failure again after Local Collecting, and it affects every node in the distributed computing system.

Java language supports multi-thread programming natively. All Java program will run in a thread context, no matter whether the user specifies it explicitly or not. Java language offers synchronization mechanisms for Java programming in multi-thread paradigm. However, threads distributed on multiple machines can not be easily synchronized since these threads are not in a single address space. SingleJava uses a proxy service to synchronize Java threads [13-16].

II. Architecture

SingleJava is a distributed Java Virtual Machine, which tries to build distributed computing system based on the Java Virtual Machine. The distributed virtual machine will be transparent to Java programmers who will just code like on a traditional, single node JVM. And even more, they can test and debug their programme on a

doi:10.4304/jsw.8.11.2917-2924

Hotspot JVM, after everything is settled, deploy their programme classes directly onto SingleJava. The programme will just get run, and come into being a distributed computing system. For well parallelized algorithms, performance will get promoted vastly.

The design gist of SingleJava is to provide a fully transparent distributed infrastructure for the upper level programmers, designed as a generic Java-based distributed computing platform. On the one hand, from writing multithreaded applications, and publishing applications, to the final start of the execution of the application can be completely the same as in the single-node Java virtual machine completely transparent to the upper level. On the other hand, SingleJava seek to support zero-modifying legacy code to be executed directly, write once, compile, directly executed directly. To this end, SingleJava design dynamic code deployment mechanism to manage shared data and use distributed Java heap.

In order to allow the execution of applications on SingleJava completely the same with single-node JVM, SingleJava design dynamic code deployment mechanism. User code simply deploy on the Master node, but does not include other Java class files, all Worker nodes in addition to the Java Foundation Class Library using Java fully dynamic link mechanism, user code migration from the Master node to the Worker nodes automatically in runtime.

SingleJava strive to keep the Java programming model, so to design a distributed Java heap, all nodes like access local Java heap access global objects in a distributed environment. Protect the consistency of the read and write global data synchronization mutex to ensure global consistency and integrity of the object. Syntax, semantics, programming model of Java programs do not need to make any changes, the old Java legacy code can perform direct migration to SingleJava .

SingleJava management mechanism is designed to be concise. A single MASTER node architecture is conducive to the implementation of the system and is responsible for balancing strategy, synchronized global Java heap data, as well as the realization of inter-node thread synchronization mutex. To remission the performance impact to the system to avoid a single MASTER node, SingleJava management separation of management and code execution. The MASTER node only responsible for the management of the cluster, not participate in the execution of the Java code.

SingleJava are tightly coupled distributed computing systems. Analysis showed that communication between the nodes within the system, a communication between the nodes is more frequent, to improve the inter-node request processing throughput, have a critical impact on the overall system performance. To do this SingleJava request processing thread pool technology to improve multi-the Request concurrent node response efficiency, not immediately respond to a request, the introduction of asynchronous response mechanism to further improve performance.

SingleJava uses a single MASTER node to manage the cluster, and multiple WORKER nodes to execute the Java byte codes in a parallelized way. MASTER node is in charge of maintaining the runtime information of WORKER nodes. When the system is initialized, all the nodes of WORKER will register themselves to the MASTER node. MASTER node records their information at runtime, and manages the topology dynamically. The architecture of SingleJava can be described as Figure1.

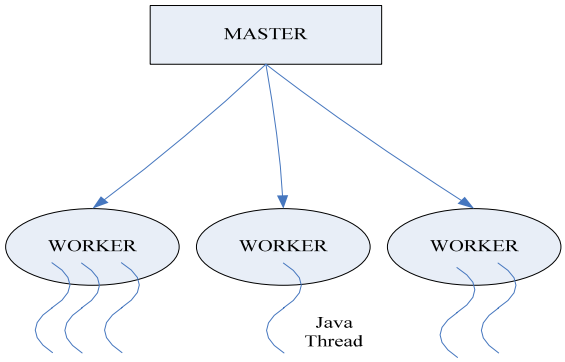

Figure 1. Architecture of SingleJava

III DISTRIBUTE TASKS

At the very beginning of the Java program, MASTER node will read in the main class (the class which contains the main method and is issued through the command line, such as HelloWorld.class, or something like that) from file system, select one WORKER node from the worker list, and send a REQ_MAIN_TASK request wrapping the main class file bytes together with the command line parameters to the WORKER node selected. As soon as the WORKER node receives the request, it will parse the class file bytes, generate a data structure for the class, and invoke the executing engine to execute the main method. The program here by begin to run, anytime latter, when the WORKER node tries to create a new Java thread, which is achieved by invoke the start() method of a class extends Thread. The WORKER node will send a REQ_NEW_THREAD request to MASTER node, which wraps the 32-bit identifier of the new thread object. MASTER node selects a WORKER node from the worker list, and sends a REQ_THREAD_TASK request to the selected WORKER node wrapping the 32-bit id of the new thread object. As the new selected WORKER node receives the request, it will load the class corresponding to the object first, and execute the run() method. Above process can be illustrated as the following Figure 2.

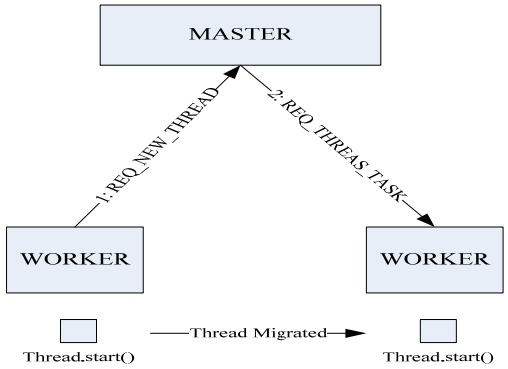

Figure 2. Thread Migration

IV. Work Load Balancing

SingleJava use the threads running on one single node as the basic metric of work load. When selecting WORKER node to migrate a new thread, the threads running on the node will be listed as the first consideration, the node with less thread running will be more preferred.

Beside this, MASTER will prefer a WORKER that has finished task more recently. The request processing function of WORKER node will work as a wrapper of Java method Thread.run(), and when the thread is finished, the wrapper will report to the MASTER. MASTER records the last thread finish time of every node, and decrease the running thread number of the corresponding node.

V. Global Heap Accessing

There are two kinds of variables in Java programs, they are local variables and instance/class fields. Local variable is private to Java threads, and their life cycle is limited within the memory of one single node. In contrary, instance/class fields should be able to get accessed globally, that means any node of SingleJava may access the object/class which residents on and owned by some other node.

SingleJava use a 32-bit global reference to identify a Class/Object uniquely, which is allocated by MASTER node. Any node can access to the class/object fields and class method/field definitions with the global reference across different nodes. Class reference is allocated by MASTER node when the class is loaded, while an object reference is allocated by MASTER node when the object is instanced.

The so called distributed Java heap is created to let any node to access any object/class freely. The architecture of the distributed Java Heap can be described as Figure 3.

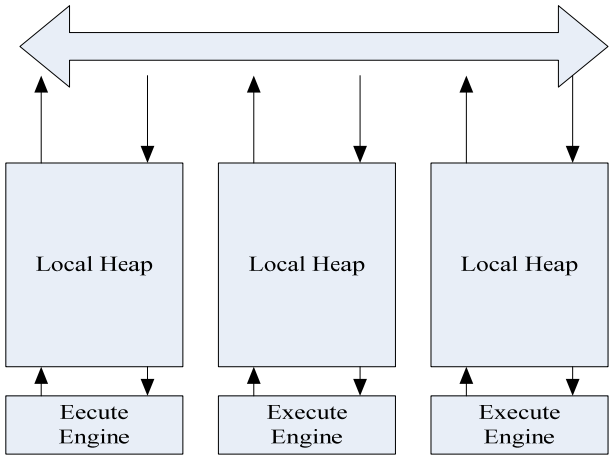

Figure 3. Distributed Java Heap

The distributed Java heap use a write-invalidate protocol to keep the consistency of object or class data [6][7]. At any time, on any WORKER node, an object must be one and only one of the following states:

- Accessed in read-only mode by one or more nodes
- Read and written by a single node (object owner)
- Not existed

The three states can transfer to each other in the specified event:

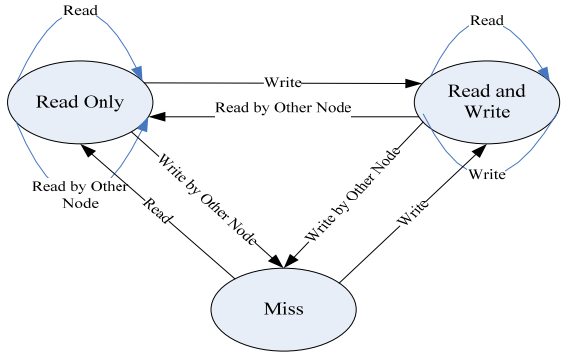

Figure 4. The Three States of Objects and their Duplicates

VI. Threads Synchronization

Threads distributed on multiple machines can not be synchronized like in a single address space. SingleJava uses a proxy service to synchronize Java threads. Let us take wait/notify mechanism which is offered by Java Programming Language for example to illustrate the details.

First of all, the WORKER node on which there is a thread want to wait on some object will send a REQ_WAIT_ON request to MASTER node, which contains the 32-bit id of the object on which the thread would like to wait. MASTER node will look into it's object table to find out which WORKER node owns this object. Then MATER will send a REQ_WAIT_ON request to the object owner which contains the 32-bit object id and information about the waiting Java thread and node. As soon as the owner receives the request, it

will add a structure contain the information about the waiting Java thread and node into the wait set corresponding to the object specified by the 32-bit id. Until now, the wait process is finished. Lately another thread will issue a notify process, it must send a REQ_NOTIFY_THREAD request to MASTER first, again, the MASTER will look into it's object table to find out the owner, then send a REQ_NOTIFY_THREAD request to the owner. As soon as the owner receives the request, it will select one Thread structure from it's wait set, according to the information in there, send a REQ_NOTIFY request back to the first WORKER node, then the request listening thread on there will act as a proxy to notify the waiting thread to resume running.

VII. Garbage Collecting

Any Java Virtual Machine must offer a garbage collecting mechanism to the upper level programmer [8] [9], as a completion of automate memory management. There are two class opportunities for garbage collecting, the user issued, which happens when the Java Program invokes the System.gc() method and the heap allocate failure caused, which is happened when the Virtual Machine is trying to allocate memory on the main memory, but failed. The Garbage Collecting mechanism of SingleJava is based on reference counting [10] [11], and divided into two granularities: Local Collecting and Global Collecting.

Local Collecting is caused by a local heap allocating failure and happen to a single WORKER node only. Local Collecting collects object read-only duplicates in local heap. While global collecting is caused by user requirement or heap allocating failure again after Local Collecting, and will cause a garbage collecting participated by all nodes.

In Global Collecting, MASTER node will request all WORKER nodes to suspend all their running Java threads, bringing the VM into a WORLD STOP state first. After that, WORKER nodes begin to do a reference counting, and reply all the marked objects back to MASTER. Collected every WORKER's reply, MASTER will make out a union from all the marked object sets reported by WORKER nodes. According the global marked object list, MASTER will authorize the owner of unmarked object to do finalize and garbage collecting. The last step of Global Collecting is to resume all threads suspended by garbage collecting and unfreeze the world.

VIII. Performance Evolution

We use Virtual Box Version 4.2.4 r81684 to set up our experiment nodes. Eight experiment nodes with the same configuration is constructed to build experiment systems. Configuration of experiment node can be denoted as Figure 5.

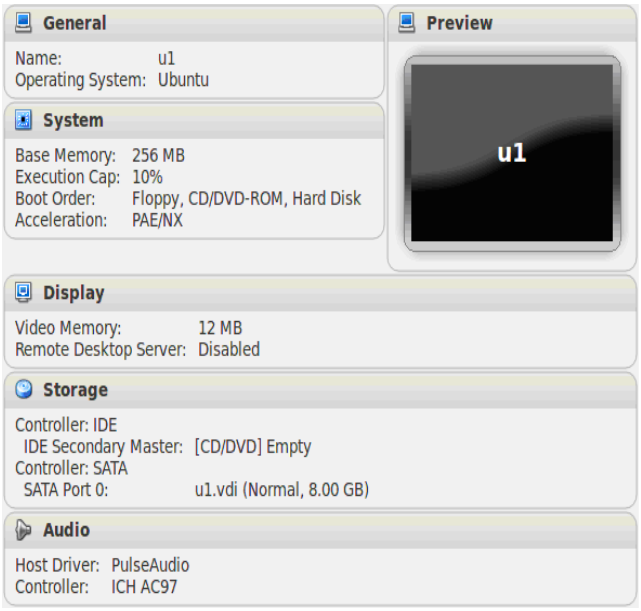

Figure 5. Configuration of Experiment Nodes

We constructed two systems for experiment, Sys-I is our target for performance evolution of single Java, and Sys-Ⅱ is used as a reference.

TABLE I.

CONFIGURATION OF EXPERIMENT SYSTEMS

Sofware Layer	Experiment Systems	
	Sys-I	Sys-Ⅱ
Legend in Diagram		
Java Library	GNU Classpath	JRE 5 System Library
Java Virtual Machine	SingleJava：1 MASTER and 7 WORKERs	JRE 5 Standard Edition HopSpot
Operating System	Ubuntu Lucid Lynx	Ubuntu Lucid Lynx
Hardware Platform	8 Experiment nodes	8 Experiment nodes

We use Multi-threaded Benchmarks of Java Grande Forum Benchmark Suite [12] [13] to evaluate the performance of SingleJava.

The test is divided into two parts, part one is the testing of core algorithms. The first item is called Series, which is designed to calculate the first N Fourrier Coefficients of function with in interval from 0 to 2. Performance is evaluated by average time costing of case resolving (Figure 6) and coefficients output per second (Figure 7).

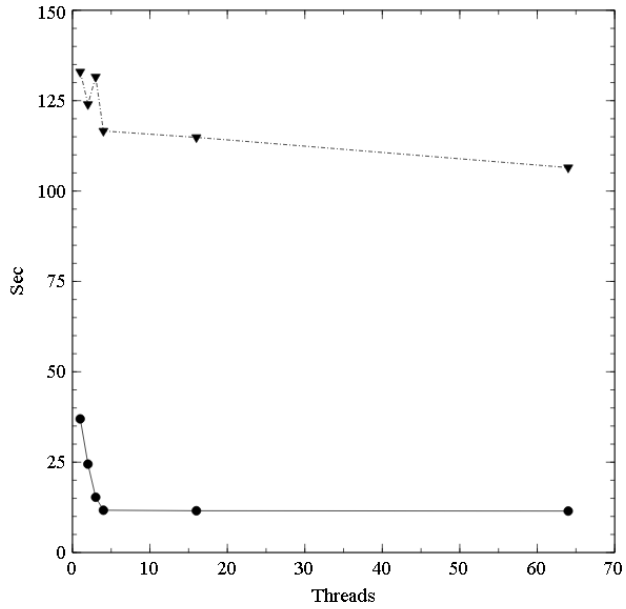

Figure 6. Series test, evaluated in seconds

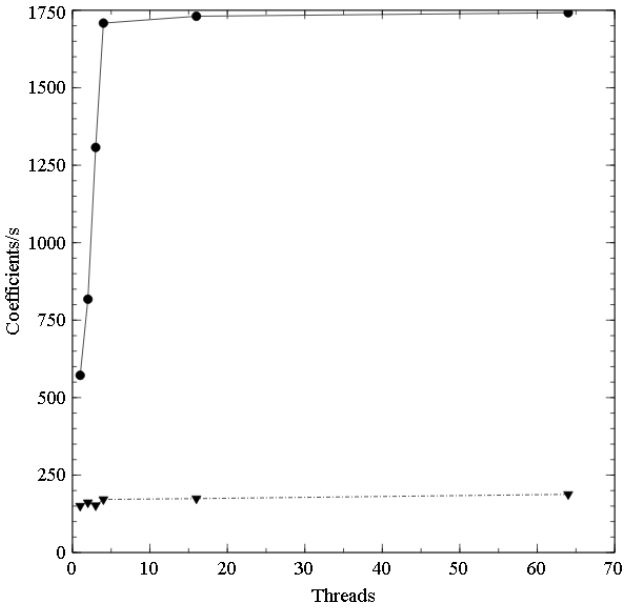

Figure 7. Series test, evaluated in coefficients per second

As observed, Sys-I is 20 times faster in both case solving and coefficients output.

The second test in part 2 is called LUFact. This test solves an N×N linear system using LU factorization followed by a triangular solve. This is a Java version of the well known Linpack benchmark. Performance is evaluated by average time costing of case resolving(Figure 8) and float-point calculating times per second(Figure 9).

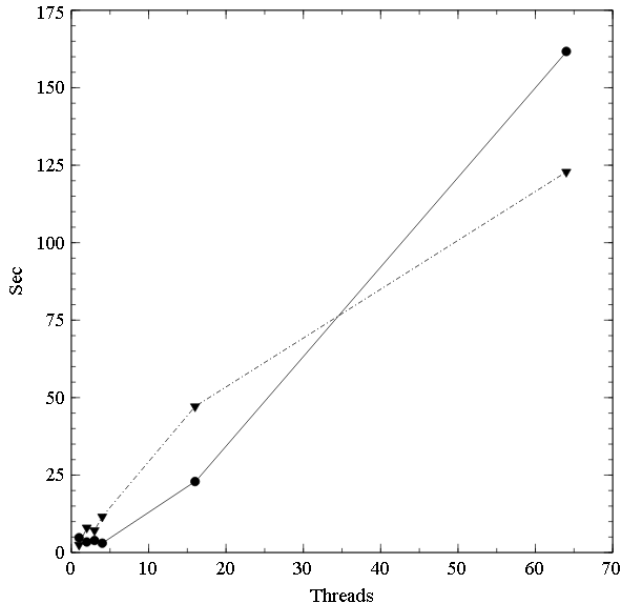

Figure 8. LUFact test, evaluated in seconds

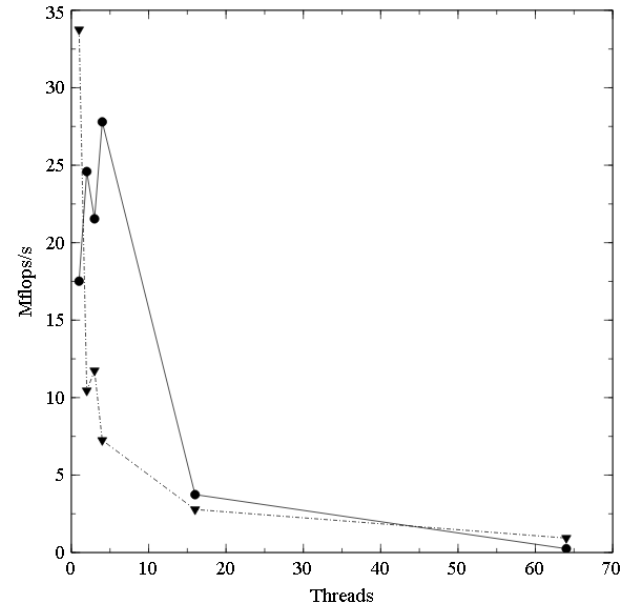

Figure 9. LUFact test, evaluated in Mflops per second

As observed, the performance of Sys-I is preferred when thread number is less than 40, when thread number beyond, too much thread synchronize caused performance loss is also significant.

The third test of part 1 is called SOR which performs 100 iterations of successive over-relaxation on an N×N grid. Performance is evaluated by average time costing of case resolving(Figure 10) and iteration times per second(Figure 11).

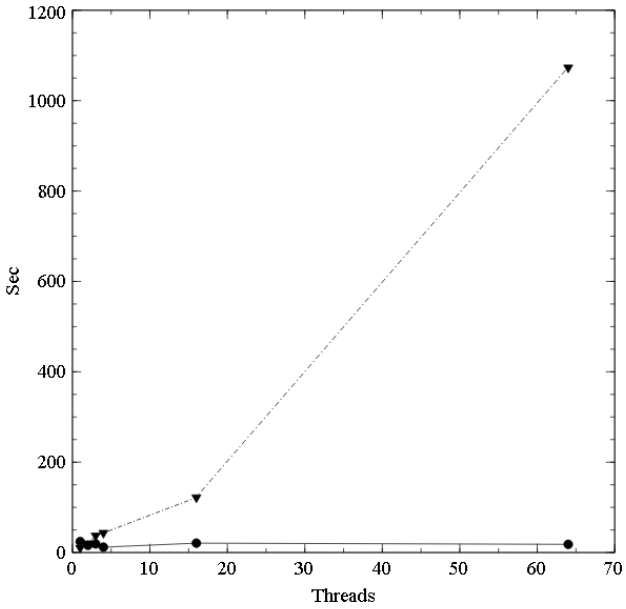

Figure 10. SOR test, evaluated in seconds

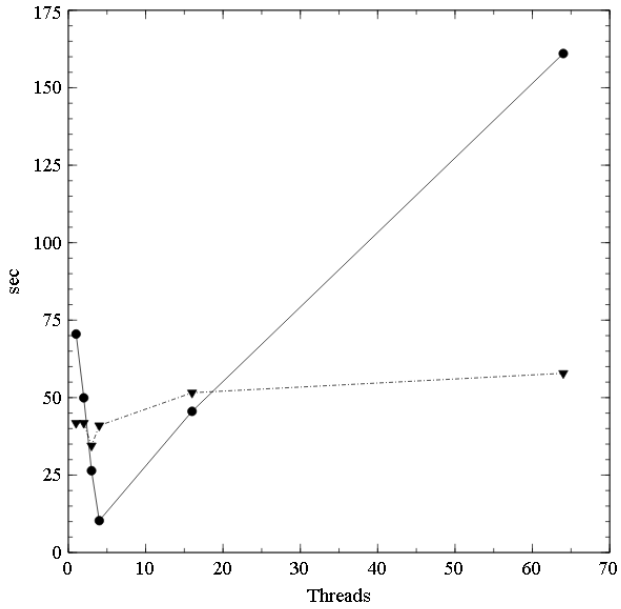

Figure 12. MolDyn test, evaluated in seconds

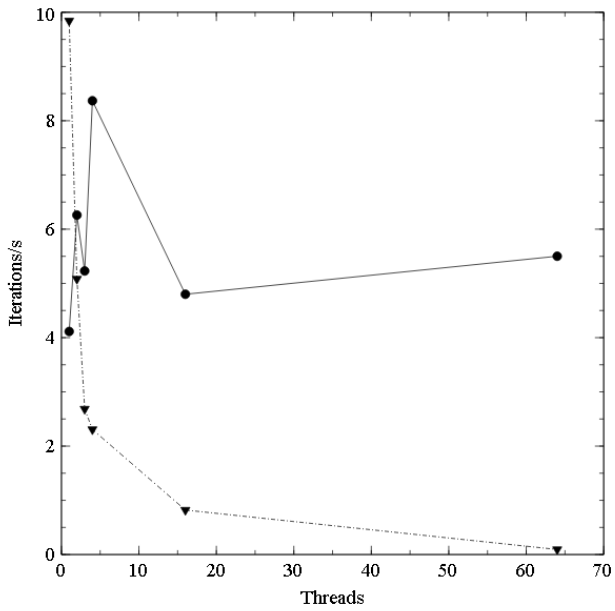

Figure 11. SOR test, evaluated in iterations per second

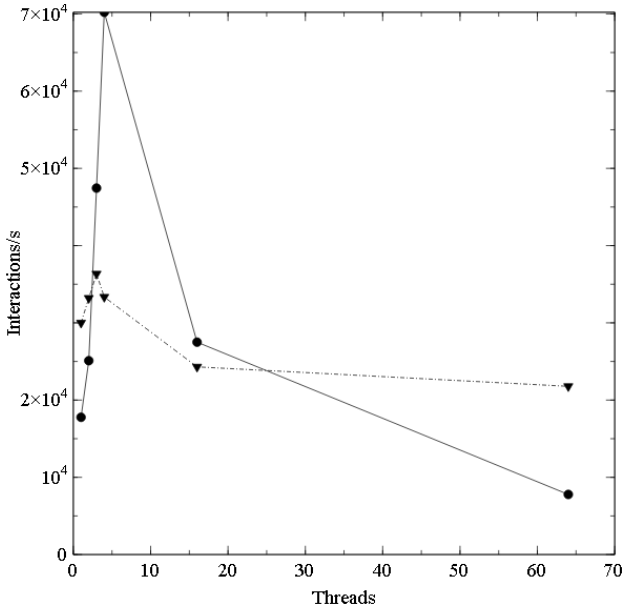

Figure 13. MolDyn test evaluated in interactions per second

As observed, Sys-I is performed better in this test. As limited parallel ability the average time costing of case resolving of Sys-Ⅱ is grown almost in a linear way while Sys-I kept steady.

Part 2 is a direct emulation of Grande Applications, for compatibility consideration, the I/O and UI is omitted as a highlight on the performance of Java Executing Environment.

The first test is called MolDyn, which is an N-body code modeling particles interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions. Performance is evaluated by average time costing of case resolving(Figure 12) and force calculation times per second(Figure 13).

As observed, Sys-I is performing excellent in this well parallelized computing intensive task. With carefully selected thread number, Sys-I can perform two times better than Sys-Ⅱ.

The second test of Part 2 is called MonteCarlo, which is a financial simulation, using Monte Carlo techniques to price products derived from the price of an underlying asset. Performance is evaluated by average time costing of case resolving(Figure 10 Left) and sample output per second(Figure 10 Right).

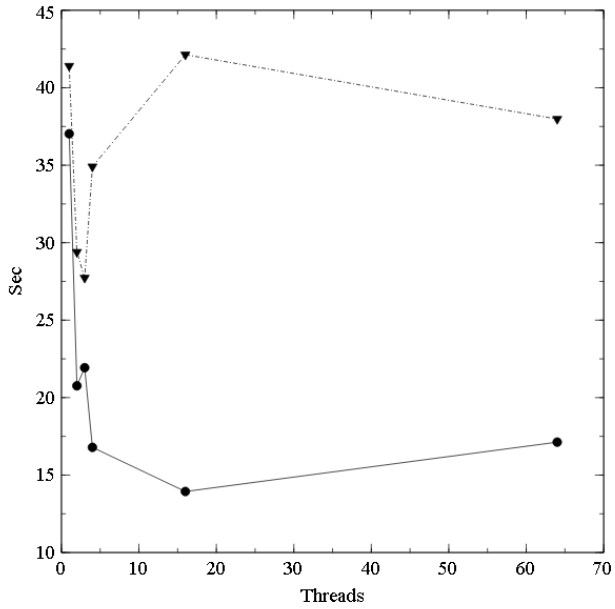

Figure 14. Monte Carlo test, evaluated in seconds

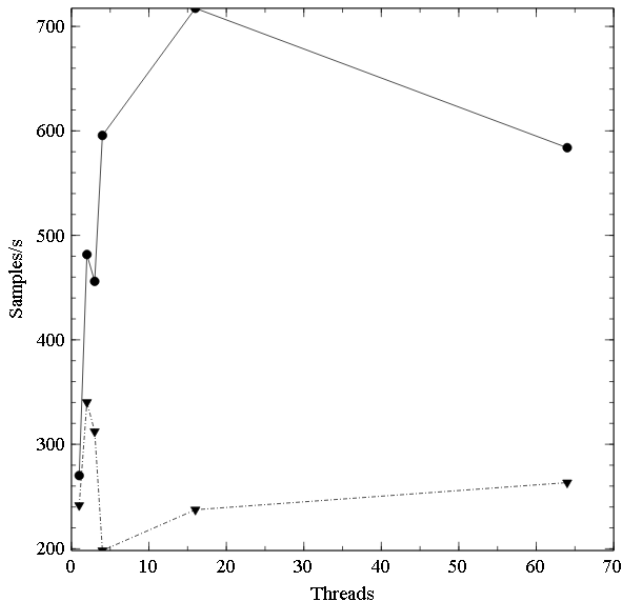

Figure 15. Monte Carlo test, evaluated in samples per second

As observed, in a best situation as tested, Sys-Ican produce 7 times more samples than Sys-Ⅱ.

The last test of part 2 is called RayTracer, which measures the performance of a 3D raytracer. Performance is evaluated by average time costing of case resolving (Figure 16) and pixels rendered per second(Figure 17).

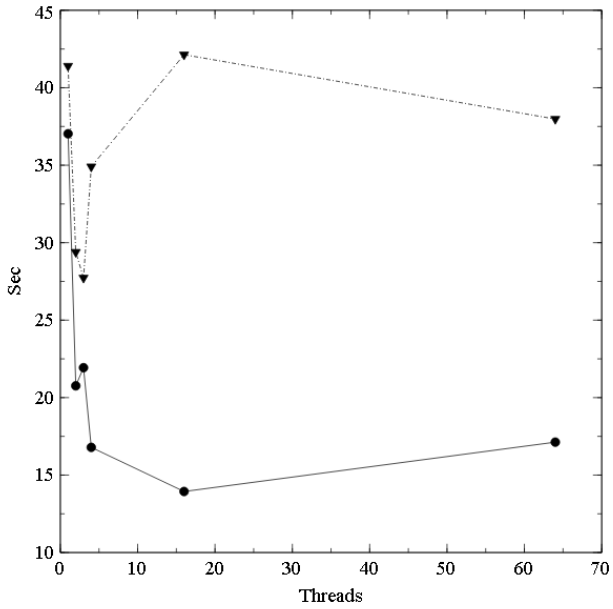

Figure 16. RayTracer test, evaluated in seconds

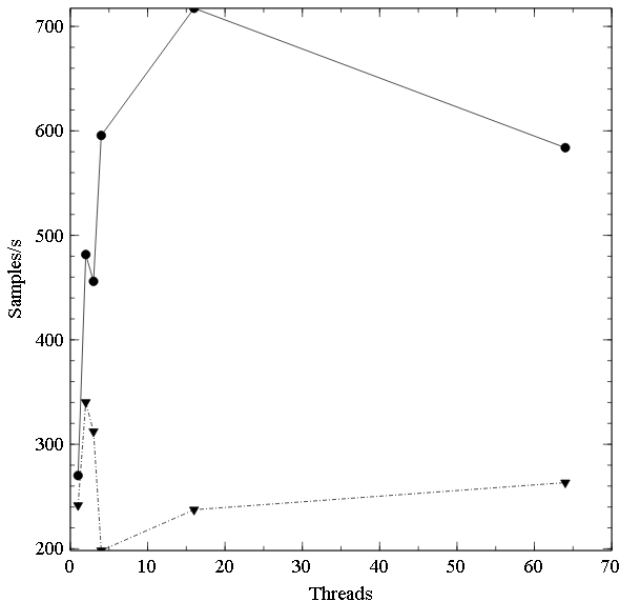

Figure 17. RayTracer test, evaluated in samples per second

As observed, the speed of scene rendering and throughput of pixels of Sys-Iare both two times better than Sys-Ⅱ.

IX. CONCLUSION

This paper proposes a thread migration approach to build distributed Java virtual machine. It is an interesting attempt to make use of the inevitable Java Virtual Machine in Java based software systems to build a transparent distributed computing framework.

This paper uses a Thread Migration approach to distribute tasks among different nodes, which is achieved by hijack internal native implement of Thread.start() method of Java Virtual Machine. Making use of the low coupling among different thread thread management

structures of in Java Virtual Machine, this paper also implemented a Distributed Java Heap for global object fields' accessing, and a distributed Garbage Collecting mechanism. Another consideration needs to be taken is how to synchronize Java Threads on different nodes, we implemented a local proxy based service to achieve that.

Finally, we evaluated the performance of our system prototype SingleJava using the Multi-threaded Benchmarks of Java Grande Forum Benchmark Suite. A performance analysis is also provided.

ACKNOWLEDGMENT

The work of this paper is supported by the Science and Technology Program of Zhejiang Province (No.2012C33078).

REFERENCES

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

[2] T. D. Brauna, H. J. Siegelb, N. Beckc, L. L. Bölönid, M. Maheswarane, A. I. Reutherf, J. P. Robertsong, M. D. Theysh, B. Yaoi, D. Hensgenj, and R. F. Freundk, "A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems", Journal of Parallel and Distributed Computing, vol. 61, no.6, pp.810-837, 2001.

[3] Yu-Kwong Kwok, and Ishfaq Ahmad, "Efficient Scheduling of Arbitrary Task Graphs to Multiprocessors Using a Parallel Genetic Algorithm", Journal of Parallel and Distributed Computing, vol.47, no.1, pp.58-77, 1997.

[4] K. Thitikamol, "Thread Migration and Communication Minimization in DSM Systems", In Proceedings of the IEEE, vol.87, no.3, pp.487-497, 1999.

[5] Ayal Itzkovitz, Assaf Schuster, and Lea Shalev, "Thread Migration and its Applications in Distributed Shared Memory Systems", Journal of Systems and Software, vol. 42, no.1, pp.71-87,1998.

[6] J. Archibald, and B. Jean-Loup, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model", ACM Transactions on Computer Systems, vol.4, no.4, pp.273-298, 1986.

[7] N. P. Jouppi, "Cache Write Policies and Performance", ACM SIGARCH Computer Architecture News, vol. 21, no. 2, pp. 191-201, 1993.

[8] Bernard Lang, Christian Queinnec, and Jose Piquer, "Garbage Collecting the World", In Proceeding of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp.39-50, 1992.

[9] Saleh E. Abdullahi, and Graem A. Ringwood, "Garbage Collecting the Internet: A Survey of Distributed Garbage Collection", ACM Computing Surveys, vol. 30, no. 3, pp. 330-373, 1998.

[10] Hans-J Boehm, "The Space Cost of Lazy Reference Counting", In Proceeding of the ACM SIGPLAN-SIGACT Symposium, pp.210-219, 2004.

[11] Y. Levanoni, and E. Petrank, "An On-the-fly Reference-counting Garbage Collector for Java", ACM Transactions on Programming Languages and Systems, vol. 28, no. 1, pp. 1-69, 2006.

[12] Preeti Paranjape-Voditel, and Umesh Deshpande, "A DIC-based Distributed Algorithm for Frequent Itemset Generation", Journal of Software, vol 6, no 2, pp.306-313, Feb 2011.

[13] Brad Long, "A Framework for Model Checking Concurrent Java Components", Journal of Software, vol 4, no 8, pp.867-874, Oct 2009.

[14] Lidong Zhai, Li Guo, Xiang Cui, and Shuhao Li, "Research on Real-time Publish/Subscribe System supported by Data-Integration", Journal of Software, vol 6, no 6, pp.1133-1139, Jun 2011

[15] Xin Chen, Xubin He, He Guo, and Yuxin Wang, "Design and Evaluation of an Online Anomaly Detector for Distributed Storage Systems", Journal of Software, vol 6, no 12, pp. 2379-2390, Dec 2011.

[16] Bing Gao, and Jianpei Zhang, "Density Based Distribute Data Stream Clustering Algorithm", Journal of Software, vol 8, no 2, pp. 435-442, Feb 2013.

Jian Su received the doctor degree in computer science & technology from Zhejiang University in 2003. Currently, he is an Associate Professor at Zhejiang University City College. His research interests include Cloud Computing and Data Mining.

Chong Zhou is a master student of Department of Computer Science at Zhejiang University. His research interests include Cloud Computing and Software Engineering.

Wengyong Weng received the master degree in computer science & technology from Zhejiang University in 2004. Currently, he is an Associate Professor at Zhejiang University City College. His research interests include Software Engineering and Cloud Computing.