International Journal of Advanced Computer Research (ISSN (print): 2249-7277 ISSN (online): 2277-7970)
Volume-3 Number-1 Issue-8 March-2013

Modeling For Runtime Locality Optimizations of Distributed Java Applications Using Dynamic Localization Algorithm

Daya Shankar Verma[1], Abhay Kumar[2] , Amarto Chakrabarty[3] , Haider Banka[4] , Amrita Priyam[5]
Information Technology Department, Government of Jharkhand[1]
Assistant Professor, Dept. of Computer Science, University Polytechnic, BIT Mesra, Ranchi[2]
Manager State Projects, UIDAI, Government of India[3]
Assistant Professor, Dept. of Computer Science and Engineering, ISM, Dhanbad[4]
Associate Professor, Dept. of Computer Science, BIT Lalpur Extn., Ranchi[5]

Abstract

In distributed java environments the locality of objects plays a crucial role in determining the performance, scalability and stability of the overall system. A manual distribution of objects has several drawbacks and requires a series of assumptions which may not be applicable as the system scales. We introduce and demonstrate how a Dynamic Localization Algorithm can be used to place objects in different Java Virtual Machines based on the processing and communication times. Using the metrics obtained by the Locality Optimization Algorithm, we have designed a placement strategy for objects and migrated them to their optimal Java Virtual Machine. For simplicity, we use the problem statements of Matrix Multiplication and show how a repeated execution dynamically selects the optimal Java Virtual Machine. We have used Java Party runtime environment to demonstrate the algorithm.

Keywords

JVM, Java Party, Dynamic Localization Algorithm, DOC.

1. Introduction

Distributed system provides a single coherent environment of computers and software to accommodate remote resource sharing. Depending on the organization of processes, distributed system can be classified into two models: client-server and peer-to-peer. Most distributed systems are based on the client-server model.[1][2][3]

Replicating objects from remote servers to client machines for local executions reduces user response times. Replication also improves system scalability. Moreover, replication enhances the service availability as the clients are able to request services from locally valid replicas even through network connectivity is disrupted.[3][4][8][11]

JavaParty is another companion of Java in this field. JavaParty allows easy porting of multi-threaded Java programs to distributed environments such as clusters. JavaParty extends the capabilities of Java to distributed computing environments to support multiple address space. [7][12][13] All language extensions are automatically transformed back to pure Java. JavaParty code is transformed into regular Java code plus RMI hooks. The resulting RMI portions are fed into the RMI compiler to generate stubs and skeletons. This approach maintains the Java object semantics such that the programmer can use remote objects just like normal Java Objects.[5][6][9][10].

2. Literature Review

Object Replication and Migration can be done randomly or based on a strategy. It becomes very important to have a placement strategy with which the objects would be replicated, migrated or created.[2][3][4] Solely distributing objects and threads over virtual machines is not sufficient for achieving performance gains. Until, JavaParty provides a mechanism to create remote objects on specific nodes of a cluster environment. Such a manual approach has several disadvantages. First, the object distribution is dependent on the specific topology for which the program is compiled. The distribution strategy must be adapted to each target platform. Second, manually specifying the location of every single object creation is tedious. Third, the optimal placement of objects often cannot be determined statically for dynamic applications where the optimal location of objects changes at runtime.[1][7][8][9] The work at hand focuses on the automatic generation of a distribution strategy for remote objects. The generation is based on runtime information of the distributed system. Even if the initial object distribution generated by JavaParty is not optimal, the locality of the application is optimized at runtime.[10][11][12][13]

3. Proposed Algorithm

Locality Optimization Algorithm

Locality Decision in parallel object-oriented languages can be grouped in three categories:

- Let the programmer specify placement and migration explicitly by means of annotations,
- Static object distribution where the compiler tries to predict the best node for a new object, and
- Dynamic object distribution based on a runtime system that keeps track of the call graph.

We discussed the three modes with respect to JavaParty as follows:

Manual Object Distribution

The node of the object to be allocated with the tag @at n. The following code snippet allocates an object of remote class R on the distributed environment.

```
import jp.lang.DistributedRuntime;
public void manualDistribution()
        {
  int noOfRegisteredJVMs = DistributedRuntime.getMachineCnt();
  for (int n = 0; n < noOfRegisteredJVMs; n++)
        {
    /** @at n */
    SomeRemoteObject object = new SomeRemoteObject();}}
```

In the above code snippet, the annotation **/** @ at n */** makes sure that the Object formed by the statement SomeRemoteObject object = **new** SomeRemoteObject();

Gets created in the JVM registered by the ID n Manual Object distribution assumes that the programmer has the idea where the object should execute.

Static object distribution

Although a Java thread cannot migrate, the control flow (called *activity* in the following) can: when a method of a remote object is invoked, the activity conceptually leaves the JVM of the caller and is continued at the callee's JVM where it competes with other activities. Due to time slicing and blocking, competing activities on one JVM decrease the total parallelism. Additional costs are introduced by the remote method invocation itself because of communication latency and bandwidth limitations. Based on a static type analysis, estimates for two values are derived:

- ***work(t, a)*** describes the computing time that activity *t* spends on methods of object *a*, and
- ***cost(t, a)*** describes the communication time that would be necessary if *t* and *a* are not located in the same address space.

Through the placement of object *a*, the computing time of that activity *t* should be maximized in which address space *a* is created. At the same time, the sum of communication cost that is required for those activities t_i assigned to remote virtual machines should be minimized.

We assume an initial setting where all objects are located in a single address space with a single processor such that all method calls are local. Each object *a* can be mapped to an activity *t* in which address space it should be placed:

$$\mathbf{Activity(a) = t \;\square\; maximizes(work(t,a) - \sum_{t_{i \neq t}} cost\,(t_i, a))}$$

Since usually more activities are used than virtual machines are available, several activities must share a virtual machine. The parallelization win of each activity can be estimated by mapping each object to its optimal activity. The parallelization win is computed by the sum of *work (t, a)* for objects *a* which reside in the address space of activity *t* minus the sum of *cost (t, b)* for objects *b* that are placed remotely.

The sum of *work(t, a)* represents the computing time that activity *t* spends in its own address space. This work is done in parallel to other activities if no synchronization mechanisms are used. The time that is spent for communication with other address spaces is represented by the sum of *cost(t, b)* for all objects *b* that are not assigned to activity *t*. Note that we charge the cost of a remote call to the activity that invoked the remote method, not to the activity that actually executes the method call. Activities are assigned to the available virtual machines in decreasing order of their parallelization wins until a single activity has been scheduled to each virtual machine. For each remaining activity, a new parallelization win is computed that accounts for the potential co-location with other activities. The activity is assigned to that group of activities with the highest combined parallelization win. This process is repeated until all activities are scheduled to their optimal virtual machine. The result of the distribution analysis is a mapping of each remote object to the virtual machine on which it should be placed.

Dynamic Object Distribution

The static approach of Object distribution has two disadvantages:

- There is no knowledge about future call graphs and invocation frequencies. This problem is inherent to dynamic approaches but can be softened by using heuristics to predict future behaviour.
- Creation of Objects that cannot migrate often results in a broad redistribution of other objects. In homogeneous clustered environments this can be reduced by avoiding cyclic redistributions of remote objects.

The dynamic approach has an advantage that rather than estimating the value or work and cost they can be measured. Cost and Work needs to be measured in units of time. From Pentium processor onwards a RDTSC (Read Time Stamp Counter) method of measuring cycles is available. Also, with the Java Technology advancement the relationship of underlying hardware to the execution and the runtime environment has been made obsolete. Our cluster would essentially be a collection of Java Virtual Machines and not Physical machines. To compare the time for two or more Java Virtual machines, we would make use of API's that can convert clock units to time units. The time in milliseconds can be retrieved using the System.currentTimeInMillis().

To avoid measurement errors because of concurrency, we assume that the workstations of the cluster are used exclusively for JavaParty. Thus, we assume that those interrupts balance over time such that cycle counting actually reflects the *average* execution time. RMI uses a standard mechanism for communicating with remote objects – stubs and skeletons. We want to measure *work(t, a)* and *cost(t, a)* in order to apply the distribution algorithm. In the context of stubs and skeletons, *work* corresponds to the time that the actual method implementation takes and *cost* corresponds to the time that is required for carrying out the remote call, i.e. marshaling and transmitting parameters and result. For remote object *r*, a stub is instantiated on each node while only one skeleton is instantiated on the node where the implementation of *r* resides. That is, there are *n* stubs and *one* skeleton for each remote object. Basically, our approach is to measure the communication time of a remote call in the stub and the execution time of the implementation in the skeleton by using the RDTSC instruction. We store aggregated *work* and *cost* values in the skeleton.

Exponential Moving Average

An Exponential Moving Average (EMA), sometimes also called an Exponentially Weighted Moving Average (EWMA), applies weighting factors which decrease exponentially. The Figure 1 at below shows an example of the weight decrease.

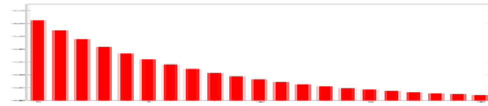

Figure 1: The weight decrease

The degree of weighing decrease is expressed as a constant smoothing factor k, a number between 0 and 1. k may be expressed as a percentage, so a smoothing factor of 10% is equivalent to k = 0.1. A higher k discounts older observations faster. Alternatively, k may be expressed in terms of N time periods, where:

Smoothing Factor (k)= 2 / (N +1)

The observation at a time period t is designated Y_t, and the value of the EMA at any time period t is designated S_t. S_1 is undefined. S_2 may be initialized in a number of different ways, most commonly by setting S_2 to Y_1, though other techniques exist, such as setting S_2 to an average of the first four or five observations. The prominence of the S_2 initialization's effect on the resultant moving average depends on k; smaller k values make the choice of S_2 relatively more important than larger k values, since a higher k discounts older observations faster. The formula for calculating the EMA at time periods $t > 2$ is : $S_t = k * Y_{t-1} + (1-k) * S_{t-1}$

This formula can also be expressed in technical analysis terms as follows, showing how the EMA steps towards the latest data point, but only by a proportion of the difference (each time):

$EMA_{(N)} = EMA_{(N-1)} + k\, [Value_{(N)} – EMA_{(N-1)}]$

Calculation of Costs and Averages

We have used the Moving Average formula for determination of Average Cost and Work as:

Average $Cost_{JVM(X,N)}$ = Average $Cost_{JVM(X,N-1)}$ + k [$Cost_{JVM(X)}$ – Average $Cost_{JVM(X,N-1)}$]

and Average $Work_{JVM(X,N)}$ = Average $Work_{JVM(X,N-1)}$ + k [$Work_{JVM(X)}$ – Average $Work_{JVM(X,N-1)}$]

Where

Smoothing factor (k)= 2 / [NoOfJVM(O) + 1]

and

Average $Cost_{JVM(X,N)}$ = *Average Cost for JVM with ID **X** after Executing **N** objects*

Average $Cost_{JVM(X,N-1)}$ = *Average Cost for JVM with ID **X** after Executing **-1** objects*

Average $Cost_{JVM(X,N)}$ = *Average Cost for JVM with ID **X** after Executing **N** objects*
Average $Cost_{JVM(X,N-1)}$ = *Average Cost for JVM with ID **X** after Executing **-1** objects*
NoOfJVM(O)= *Number of Registered (and Willing) JVM's for Object Type **O***
$Cost_{JVM(X)}$ = *Cost of Executing Latest Network Call on JVM with ID **X***
$Work_{JVM(X)}$ = *Work effort in Executing method on JVM with ID **X***
We need to understand the relationship:
$Cost_{JVM(X)} = Comm_{JVM(X)} + Work_{JVM(X)}$
Where
$Comm_{JVM(X)}$ = *Communication Time for Executing Object **O** on JVM **X***
We have used the above concepts and implemented the same using JavaParty.

Expected Solution
As soon as the Program starts, it should show up the number of registered Java Virtual Machines in a tabular form. The Tablel 1 shows the metrics based on the combination of Problem statement and Java Virtual Machine Number.

Table 1: Metrics on the combination of Problem statement and JVM Number

JVM ID	JVM/Process	No Of Units	Recent Cost	Recent Work	Average Cost	Average Work
0	Matrix Multiplier-JVM#0	0	0.0	0.0	0.0	0.0
1	Matrix Multiplier-JVM#1	0	0.0	0.0	0.0	0.0

As soon as one Object of Matrix Multiplier gets introduced by clicking on Matrix Multiplication method the Table 2 shows the values of recent cost and work incurred while executing it.

Table 2: The values of recent cost and work

JVM ID	JVM/Process	No Of Units	Recent Cost	Recent Work	Average Cost	Average Work
0	Matrix Multiplier-JVM#0	1	235.0	172.0	235.0	172.0
1	Matrix Multiplier-JVM#1	0	0.0	0.0	0.0	0.0

When the Matrix Multiplier gets pressed again, the next object should execute in JVM 1 as it has never been used. This is how the Table 3 should look.

Table 3: Execution in JVM #1

JVM ID	JVM/Process	No Of Units	Recent Cost	Recent Work	Average Cost	Average Work
0	Matrix Multiplier-JVM#0	1	235.0	172.0	235.0	172.0
1	Matrix Multiplier-JVM#1	1	203	156	203	156

There should be another Table 4 which lists how many instances have been created and which Java Virtual Machine are being used.

Table 4: Creation instances in used JVM

ID	Name	JVM	Start Time	End Time	Cost	Work
1	Matrix Multiplier	Matrix Multiplier-JVM #0	Sun Jul 12 00:45:50 IST 2012	Sun Jul 12 00:45:50 IST 2012	235.0	172.0
2	Matrix Multiplier	Matrix Multiplier-JVM	Sun Jul 12 00:47:20 IST 2012	Sun Jul 12 00:47:20	203	156

		#1		IST 2012		

As per this Table 4 the second injection of object resulted in its execution in JVM which got the benefit of doubt that it could be the more efficient one. Now, if a distribution mechanism is not available the third object could execute anywhere. On the other hand if a distribution mechanism is present, it should use the work and cost metrics to decide where will the code execute. Based on the above data, the cost of using JVM#0 is more than the cost of using JVM#1. This means based on Locality Optimization Techniques (LOT) discussed so far, the next object would be optimally placed in JVM#1 and not JVM#0. On clicking the injection button (Matrix Multiplier) again, the JVM Table 5 changes as follows:

Table 5: Object Optimally placed in JVM#1

JVMID	JVM/ Process	No Of Units	Recent Cost	Recent Work	Average Cost	Average Work
0	Matrix Multiplier-JVM#0	1	235.0	172.0	235.0	172.0
1	Matrix Multiplier-JVM#1	2	172	156	182.33	156

and the Table 6 which lists what instance executed and where should be placed.

Table 6: The average cost of JVM#1

ID	Name	JVM	Start Time	End Time	Cost	Work
1	Matrix Multiplier	Matrix Multiplier-JVM#0	Sun Jul 12 00:45:50 IST 2012	Sun Jul 12 00:45:50 IST 2012	235.0	172.0
2	Matrix Multiplier	Matrix Multiplier-JVM#1	Sun Jul 12 00:47:20 IST 2012	Sun Jul 12 00:47:20 IST 2012	203	156
3	Matrix Multiplier	Matrix Multiplier-JVM#1	Sun Jul 12 00:52:30 IST 2012	Sun Jul 12 00:52:30 IST 2012	172	156

Since the average cost of JVM#1 is still less, the 4th object should also be executed in the same Java Virtual Machine. The above algorithm depicted as Figure 2.

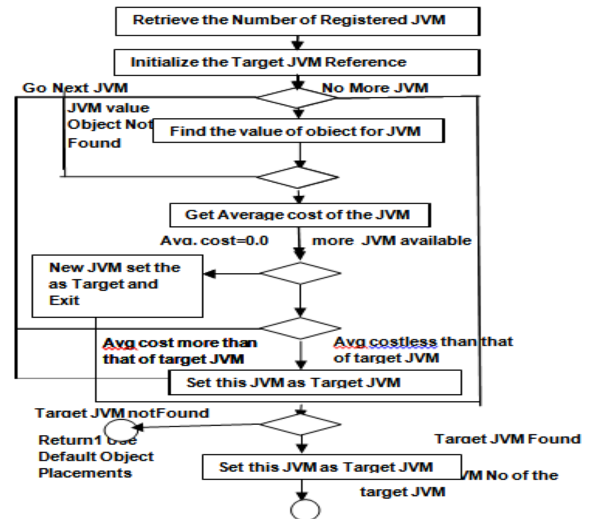

Figure 2: Flow Chart

4. Result Analysis

The system works on a Dynamic Localization scheme. Initially each of the three JVM's got the chance to execute objects. When the fourth object came in, average of JVM 2 was least and it got the chance to execute. As long as the Moving average of JVM2 does not go more than the average of JVM 1, JVM 2 will be the optimal choice and would get the chance to execute Objects.

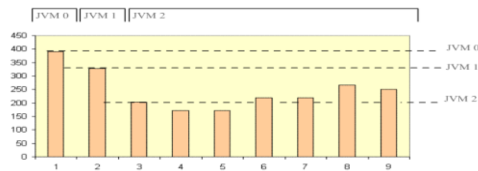

Figure 3: Result Analysis

This Demo can be run again and again for the two family of objects and readings could be verified and put in a graphical form as Figure 3. The system will always use the most optimal Java Virtual machine to execute objects.

5. Conclusion and Future Work

Using the Java Party Distributed Runtime Environment we are able to demonstrate our Dynamic Localization of Distributed Objects. We are able to show how dynamic distribution helps in deciding the best option and helps us to design systems which are much more powerful and much more intelligent. This logic does not include the dimension of concurrency i.e. even of the average of a JVM is very good if objects come in at a very high rate then the principles of parallel processing should supersede and objects should be moved to separate JVM's. This work assumes that the incoming objects are of same atomic workload. However, this may not be true every time. There could be cases when the work could be of same type but different computing overheads. The Locality Optimization Logics could be improved to foresee the possible overhead and then determine the optimal machine. For Objects of varying sizes and overheads the concept of Moving Average will not hold good. This logic could be improved by taking into consideration the health of the Java Virtual Machine as well. Currently if the memory of the JVM is being filled at a very fast rate the logic does not take into account that the JVM could actually crash very soon.

References

[1] M. Factor, A. Schuster, and K. Shagin, "A distributed runtime for Java: yesterday and today", Parallel and Distributed Processing Symposium, 2004.

[2] W. Zhu, C. –L. Wang, and F.C.M. Lau, "JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support", IEEE Fourth International Conference on Cluster Computing, Chicago, USA, September 2002.

[3] Sun Microsystems, "Java Remote Method InvocationSpecification",2003.http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html.

[4] Sun Microsystems, "Java Native Interface", 2003. http://java.sun.com/j2se/1.4.2/docs/guide/jni.

[5] Intel Corp, "Using the RDTSC Instruction for PerformanceMonitoring",1997.http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM.

[6] R. Veldema, R. A. F. Bhoedjang, and H. E. Bal, "Jackal, a compiler based implementation of java for clusters of workstations", Proceedings of PPoPP, 2001.

[7] Abhay Kumar, D.S.Verma, V. Bhattacherjee et. al., "Modeling using K-means clustering algorithm" Recent Advances in Information Technology (RAIT-2012), IEEEXplore ISBN: 978-1-4577-0694-3, Page number: 554 – 558, 15-17 March 2012.

[8] G. Antoniu, L. Bouge, P. Hatcher, M. MacBeth, K McGuigan, and R. Namyst, "The Hyperion system: Compiling multithreaded Java bytecode for distributed execution", Parallel Computing, 2001.

[9] Y. Aridor, M. Factor, and A. Teperman, "Cjvm: a single system image of a JVM on a cluster", Parallel Processing, 1999, pp. 4-11.

[10] T. Fahringer, "JavaSymphony: a system for development of locality-oriented distributed and parallel Java applications", Cluster Computing,2000.

[11] V. Felea, R. Olejnik, and B. Toursel, "ADAJ:a Java Distributed Environment for Easy Programming Design and Efficient Execution", Shedae Informaticae, UJ Press, Krakow, 2004, pp. 9-36.

[12] J.Maassen and R.V. Nieuwpoort, "Fast parallel Java", Master's thesis, Dept. of Computer Science, Vrije Universiteit, Amsterdam, August 1998.

[13] M. Philippsen and M. Zenger, "JavaParty Transparent Remote Objects in Java", Concurrency: Practice and Experience, 1997.

Daya Shankar Verma, M.Tech(CS), MCA, MCSE, born in India, having 10+ years of experience in academic and software industry, completed M.Tech in Computer Science from BIT Mesra, Ranchi and Microsoft Certified Systems Engineer (MCSE) from Microsoft, USA and presently working as Scientific Officer (Programmer) in Information Technology Department, Government of Jharkhand in India. Published paper "Modeling using K-means clustering algorithm" in IEEE xplore. Current research focuses on Data Mining.