2005

Exploiting the Advantages of Object-Based DSM in A Heterogeneous Cluster Environment

Xuli Liu  
*University of Nebraska - Lincoln*, xuliu@cse.unl.edu

Hong Jiang  
*University of Nebraska - Lincoln*, jiang@cse.unl.edu

Leen-Kiat Soh  
*University of Nebraska*, lsoh2@unl.edu

Follow this and additional works at: [http://digitalcommons.unl.edu/cseconfwork](http://digitalcommons.unl.edu/cseconfwork)  
Part of the [Computer Sciences Commons](http://digitalcommons.unl.edu/cseconfwork)

Liu, Xuli; Jiang, Hong; and Soh, Leen-Kiat, "Exploiting the Advantages of Object-Based DSM in A Heterogeneous Cluster Environment" (2005). *CSE Conference and Workshop Papers*. 17.  
[http://digitalcommons.unl.edu/cseconfwork/17](http://digitalcommons.unl.edu/cseconfwork/17)
Exploiting the Advantages of Object-Based DSM in A Heterogeneous Cluster Environment

Xuli Liu, Hong Jiang, Leen-Kiat Soh

Department of Computer Science and Engineering, University of Nebraska – Lincoln
{xuliu, jiang, lksoh}@cse.unl.edu

Abstract

In recent years, increasing effort has been made by the cluster and grid computing community to build object-based Distributed Shared Memory systems (DSM) in a cluster environment. In most of these systems, a shared object is simply used as a data-exchanging unit so as to alleviate the false-sharing problem, and the advantages of sharing objects remain to be fully exploited. Thus, this paper is motivated to investigate the potential advantages of object-based DSM. For example, the performance of a distributed application may be significantly improved by adaptively and judiciously setting the size of the shared-objects, i.e., granularity. This paper, in addition to investigating the advantages of sharing objects, particularly focuses on observing how the performance of a distributed application changes with varied granularity, obtaining the optimal granularity through curve fitting, studying the factors that affect the optimal granularity, and predicting this optimal granularity in a changing runtime environment.

1. Introduction

Parallel/distributed programming has two prevailing communication abstractions—explicit Message Passing (MP) and Distributed Shared Memory (DSM). Due to its programming complexity, MP has been replaced with DSM on multiprocessor computers, such as DASH [7] at Stanford, Alewife [2] at MIT, TreadMarks [6] at Rice, IVY [9] at Yale, etc. Nevertheless, MP remains the dominant programming paradigm on cluster computers. The reason is that the design of DSM on multiprocessor computers cannot be transplanted easily and directly onto cluster computers due to the significant difference in the communication infrastructure between the tightly coupled multiprocessor systems and the loosely coupled cluster systems. For example, compared to multiprocessor computers, cluster computers have much higher communication latency and lack hardware support for replication and consistency control. This provides impetus for researchers to redesign DSM on cluster computers to overcome the associated problems.

The design of DSM on multiprocessor computers is based on either cache lines or memory pages. Although some of the DSM systems on cluster computers are still page-based, e.g., Java/DSM [19], most of them provide object-based DSM, such as JavaParty [13], Jackal [15], Hyperion [12], and cJVM [1]. To the best of our knowledge, most of the current studies on object-based DSM are still in the phase of building research platforms and solving the related implementation issues. In these systems, an object is simply used as a data-exchanging unit similar to a cache line or a memory page in the DSM of multiprocessor computers. Nevertheless, some key studies have been carried out, aiming at improving the efficiency of the object-based DSM. For example, Chang et al. [3] proposed a mechanism to assign different consistency levels appropriate for different shared objects for interactive applications on wide area distributed systems. However, the aforementioned studies focus on either the implementation issues or improving the efficiency of the DSM from some specific aspects such as communication latency and bandwidth requirements. Thus, these studies fall short of systematically addressing the advantages of sharing objects, or attempting to fully exploiting such advantages.

In this paper, in addition to investigating the advantages of sharing objects, we focus on the shared-object size (i.e., granularity) problem. We run the LU benchmark application [18] using various granularities, combined with varied runtime factors. By observing the performance variations, we conclude that an optimal granularity can be obtained through curve fitting. Furthermore, we empirically derive an equation to predict this optimal granularity in a changing runtime environment. We also observed important LU-related properties on clusters, such as the insensitivity of problem size and network bandwidth to the determination of optimal granularity.

The rest of this paper is organized as follows. In Section 2, existing studies in the literature on object-based DSM systems are briefly reviewed. Section 3 studies the
advantages of object-based DSM systems over traditional DSM systems. In Section 4, we briefly introduce our object-based DSM prototype system called PJava [11]. The methodology for obtaining the optimal granularity is presented in Section 5. Finally, we conclude and outline our future work in Section 6.

2. Related work

Due to the portability of Java as well as its parallel features such as Remote Method Invocation (RMI), socket, monitor, and multithread, most of the current DSM studies on cluster computers concentrate on distributed Java Virtual Machine (JVM). Three approaches are available to construct distributed JVM. The first one is to lay the distributed JVM on top of some existing DSM system. For example, Java/DSM [19], built on top of the page-based DSM of TreadMarks [6], provides a virtual shared memory Java programming environment. Unfortunately, Java/DSM inherits the page-based feature from TreadMarks, which is incompatible with the object-based memory model of Java and that prevents Java/DSM from exploiting the advantages of Java objects.

The second approach is based on a cluster-enabled implementation of JVM, of which cJVM [1] is an example. This implementation approach is able to support any pure Java application without requiring any code modification; however, it suffers from the difficulty in keeping up with the evolution of JVM. In cJVM, each node runs a cJVM process to execute part of the application’s Java threads and contains a portion of the Java objects.

The third approach, which is the most widely used by current research projects, is to put another layer on top of JVM and provide the necessary parallel and distributed features. For example, JavaParty [13] provides a virtual object-based shared address space by translating Java-Party code into pure Java code with RMI running on JVM. Hyperion [12] supports parallel execution by distributing Java threads across multiple nodes in a cluster computer and providing an object-based virtual shared memory. Furthermore, in order to speed up the parallel execution, Hyperion provides a Java-bytecode-to-C translator. Jackal [15] provides a fine-grained DSM, which is managed as a collection of regions distributed on multiple cluster nodes. Each region is either an object or a fixed-size partition of an array. PJava [11], an ongoing project by the authors of this paper, provides programmers with a virtual object-based DSM by translating the PJava code into pure Java code and handling the communication among cluster nodes via sockets.

Besides distributed JVM, distributed DSM has been implemented on other platforms. For instance, Seidmann [14] presented a Distributed DSM system as an extension to the Microsoft .NET framework.

Some studies with more focus on improving the performance of distributed DSM systems have been carried out and reported in the literature. With an excessive number of replicas in a system, the benefits obtained from increased locality can be offset by the cost of maintaining consistency among the replicas. Leonardo et al. [8] proposed an adaptive scheme to determine on the fly the number of replicas to be made and the locations to store these replicas. In the distributed JVM system presented by Fang et al. [4], object migration, thread migration, and object pre-fetching are adopted to improve the performance. In order to alleviate contention on the lock applied to a distributed shared object, Wong et al. [17] presented a multi-locking mechanism in an object-based DSM system called Jasmine [16]. In Jasmine, multiple locks are applied to different data sets of a shared object to enhance the access concurrency.

3. The advantages of sharing objects

Some object-based DSM systems have been built and studied. However, these studies have not systematically addressed the advantages of using shared objects in DSM and investigated ways to fully exploit such advantages. This section identifies several such advantages.

First, the data-exchanging unit in a traditional DSM system is either a cache line or a memory page. Therefore, either the problem of insufficient locality due to the small cache line or the problem of false-sharing due to the large page size becomes inevitable, potentially degrading the performance of distributed applications significantly. On the other hand, an object-based DSM system only encapsulates related data into one single object, which effectively alleviates the false-sharing problem while capturing sufficient amount of locality. Furthermore, the size of a shared object could be judiciously adapted to benefit the performance of distributed applications.

Second, the object-oriented programming paradigm, in which the memory is managed based on objects, is widely used in practice by programmers. Thus, object-based DSM can better support the object-oriented programming.

Third, it is important to notice that only related data is encapsulated into one single object. Therefore, the data in one shared object may exhibit very similar access patterns; meanwhile, different shared objects may have different access patterns. Based on these access patterns, varied coherence and/or consistency protocols may be applied to different shared objects. For example, an update-based coherence protocol is suitable for a shared object that is frequently referenced by many but relatively infrequently modified; whereas, a frequently modified but infrequently referenced or shared object prefers an invalidation-based coherence protocol.
Finally, replication is widely used in DSM systems to improve locality, and a corresponding mechanism is required to maintain the consistency among replicas. Based on the fact that different shared objects may have varied requirements on the Quality of Consistency (QoC), it is possible to assign different consistency levels to different shared objects, so as to improve the performance of the distributed application. QoC allows the level of consistency of a shared object to be specified and monitored in our compilation environment, which will be further discussed in Section 4.1.

4. PJava

Instead of building a distributed JVM from scratch, PJava is built on top of JVM, providing users with an object-based distributed JVM. In this section, the compilation and runtime environments of PJava are briefly introduced, and an in-depth description could be found in [11].

4.1. The PJava compilation environment

In PJava, the standard Java language is extended to support parallel programming for distributed applications. Through this programming interface, users are able to use compilation directives to specify how to parallelize a task and partition the data, and to define globally shared objects and local objects, etc. Figure 1 shows a block diagram of the compilation environment of PJava.

![Figure 1. The compilation environment of PJava](image)

From Figure 1, we see that the PJava translator takes the PJava code as input, parallelizes the task and partitions the data, and outputs the pure Java code. During this process, heuristics stored in the knowledge base are referenced so as to determine the proper sharing level and consistency protocol for each shared object, and to choose an optimal shared-object size, and so on.

The information stored in the knowledge base is used to support the translation described above. It includes (1) the access pattern for each shared object, which is used to choose a proper coherence and/or consistency protocol for that object; (2) the consistency level assignment for each shared object, which is determined by the QoC requirements defined by the user; (3) the optimal shared-object size, which is used to guide the data partition process of a parallel task; and (4) the runtime environment factors, e.g., CPU usage and network bandwidth availability. The data in the knowledge base is initially entered and updated by the users, but will later be automatically adjusted based on the feedback obtained from the analysis mechanism. The QoC requirements include factors such as numerical error tolerance level, latency requirement, staleness tolerance level, synchronization requirement, etc. The pure Java code generated by the PJava translator is then compiled by the standard Javac compiler and is executed on the PJava runtime environment. At the same time, the execution information, e.g., the access patterns of each shared object and the CPU usage, is collected and analyzed by the analysis mechanism, and the analysis result is fed back to the knowledge base to adjust and/or update the related information.

In the current phase of this study, most of the components shown in Figure 1 have been implemented and this PJava compilation environment is operational. However, the knowledge base does not exist yet, only a few modules of the analysis mechanism are implemented, and the parallelization of tasks is not fully automated. In the experiments conducted in this paper, the runtime information, e.g., CPU and network bandwidth availabilities, is manually collected and analyzed, and the knowledge is fed into the PJava translator by hand.

The innovation of our PJava compilation environment lies with its use of knowledge base and embedded analysis mechanism; this is advantageous because it adaptively adjusts to optimize the parallelization and resource allocation strategies according to the dynamics of the system and characteristics of the application.

4.2. The PJava runtime environment

Figure 2 shows the fully implemented runtime environment of PJava on the logical head node and on one of the many work nodes. Note that the runtime environment on all work nodes is logically identical. The logical head node accepts from a user the PJava code, generates the corresponding standard Java classes and partitions the data using the heuristics in the knowledge base, and then maps the Java classes and data onto the work nodes. The logical head node is also responsible for returning the execution result to the user. The reason for using a logical head node instead of a fixed physical one is to avoid the single point of failure and the possible bottleneck.

On each work node, a daemon process keeps watching for incoming tasks and data. After getting a task, a corresponding work thread is created to execute this task,
and necessary service threads are spawned to handle the communication with other work nodes. In this object-based DSM environment, a table is maintained on each node to manage the shared objects. Besides the data, a shared object also includes other status information including the object state, the sharing level, the location of the up-to-date copy, etc. Replication is supported in PJava to enhance locality. The Distributed Shared Object Supporting Package (DSO-SP), built on top of the JVM, provides the necessary functionalities to maintain the consistency among replicas [10]. After the task is completed, the execution result is returned to the logical head node, along with such execution information as CPU usage, etc.

Figure 2. The runtime environment of PJava

5. Determining the optimal shared-object size (granularity)

In Section 3, some important advantages of an object-based DSM system are identified. In this section, we focus on one of these advantages—the idea of determining an optimal shared-object size, or object granularity. In a traditional cache-line-based or page-based DSM system, it is not easy to switch between cache-line granularity and memory-page granularity, let alone flexibly and adaptively setting the size of data-exchanging unit. On the other hand, an object-based DSM system is flexible in that regard. This motivates us to investigate ways to determine the optimal shared-object size for data-parallel applications in this paper.

For our investigation, we study the performance of the benchmark application LU [18] with varied shared-object sizes. LU, which factors a dense matrix into the product of a lower triangular matrix and an upper triangular matrix, lies at the heart of many scientific and engineering applications. We observe that an optimal shared-object size does exist and is affected by several runtime environment factors. Based on the observation, we find that a formula could further be obtained through curve fitting to predict this optimal shared-object size under changing runtime environments. Note that what we propose here is a methodology, and through our experiments, show how we conduct our search for a formula that adapts to the environmental factors for the benchmark application LU.

In what follows, we first describe the experimental setup, and then present a theoretical analysis on the relationship between the application performance and the shared-object size. The methodology starts with the observation of the performance variance of LU at different shared-object sizes. We then investigate the factors affecting the optimal shared-object size. Subsequently, curve fitting is used to identify the optimal shared-object size and a formula is derived. Finally, we validate the formula through further testing.

5.1. Experimental Setup

The experiment is carried out using the benchmark application LU from the SPLASH-2 suite. The original benchmark program was in C language, and we rewrote it using Java to make it object-oriented. LU factors a dense matrix into the product of a lower triangular and an upper triangular matrix. Here, the parallel algorithm is briefly described, and more details could be found in [18].

Assuming that the total number of work nodes is $p$ and the size of the matrix to be factorized is $n \times n$, the matrix is partitioned into a set of sub-matrices with size $m \times m$ (i.e., the so-called granularity), and these sub-matrices are mapped to the $p$ work nodes using a 2-D scatter decomposition. Starting from the top-left sub-matrix, the sub-matrices are LUed by its owner along the diagonal of the matrix downwards to the bottom-right. After a diagonal sub-matrix is LUed, all other sub-matrices on its bottom-right starts being processed at once, after which the next diagonal sub-matrix will be dealt with until all of the diagonal ones are processed. Obviously, synchronization exists among the work nodes.

The experimental platform is the Prairiefire Supercomputer at University of Nebraska–Lincoln, a high-performance cluster computer with 128 nodes. Each node has two AMD 2.2 G Opteron (64-bit) processors and 4GB RAM, and the nodes are connected by a Myrinet and a Gigabit Ethernet.

5.2. Theoretical Analysis

Roughly speaking, the total running time of a distributed application consists of three parts, namely, the computation time, communication time, and synchronization time, as shown in Equation (1).

$$T_{total} = T_{comp} + T_{comm} + T_{sync} \quad (1)$$
Intuitively, given a distributed application and an object-based DSM platform, \( T_{total} \) is affected by the shared-object size (i.e., granularity), the problem size, the number of work nodes, the CPU availability, the network-bandwidth availability, and the characteristics of the application such as the parallel algorithm and synchronization. Conceptually, this relationship can be expressed as in the following equation:

\[
T_{total} = f_{total}(g, s, n, c, b, p) \quad (2)
\]

In Equation (2), \( g \) stands for the shared object size (i.e., granularity), \( s \) symbolizes the problem size, \( n \) denotes the number of work nodes, \( c \) is the CPU availability, \( b \) signifies the network-bandwidth availability, and \( p \) describes the application characteristics such as synchronization behavior and parallelism level.

The goal of this research is to study how the application performance is affected by \( g \), and to find the optimal \( g \) so as to minimize \( T_{total} \) of a distributed application. Using \( g^{*} \) to express the optimal \( g \), we have

\[
g^{*} = \arg \min_{s, n, c, b, p} (T_{total}) \quad (3)
\]

Equation (3) means that \( g^{*} \) is the value where \( T_{total} \) is the minimum value under the given set of \( s, n, c, b, \) and \( p \). It is obvious that \( g^{*} \) is dependent on \( s, n, c, b, \) and \( p \), i.e.,

\[
g^{*} = f_{g^{*}}(s, n, c, b, p) \quad (4)
\]

In the following, we study how to obtain and adapt \( g^{*} \) in a changing runtime environment. Note that since \( p \) is an inherent property of an application, it is omitted from the current phase of this research.

5.3. Obtaining \( g^{*} \) through curve fitting

After observing the performance data of LU with different matrix sizes \( (s) \) and different numbers of work nodes \( (n) \) in a dynamic environment, we conclude that the application performance is highly affected by \( g \), and the optimal \( g ^ {*} \), i.e., \( g^{*} \), exists. As an example, Table 1 shows the data obtained when factoring a 3500x3500 matrix on 12 work nodes with full CPU and network availability. Other experiments on different matrix sizes with different number of work nodes under various system workloads had been undertaken and showed similar effects.

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Execution Time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100x100</td>
<td>229.700</td>
</tr>
<tr>
<td>150x150</td>
<td>130.766</td>
</tr>
<tr>
<td>200x200</td>
<td>105.513</td>
</tr>
<tr>
<td>250x250</td>
<td>71.900</td>
</tr>
<tr>
<td>300x300</td>
<td>62.051</td>
</tr>
<tr>
<td>350x350</td>
<td>65.525</td>
</tr>
<tr>
<td>400x400</td>
<td>100.509</td>
</tr>
</tbody>
</table>

The data in Table 1 is shown in Figure 3 as a curve that can be expressed with the formula in Equation (5):

\[
f(x) = 586.6e^{-0.009695x} + 1.309e^{0.01047x} \quad (5)
\]

Let \( df(x)/dx=0 \), and we obtain the optimal value for \( x=299 \). Setting \( g=299 \times 299 \), we run the LU application with the same settings as those in Table 1, and we get \( T_{total}=58.458 \text{(Seconds)} \), which is less than all the sample \( T_{total} \) values shown in Table 1. This hints that the mathematical equation (i.e., Eq. 5) can be used to obtain the optimal granularity size that can be validated by the actual run on the benchmark application of LU.

5.4. Impacts of \( s, n, c \) and \( b \) on \( g^{*} \)

In this subsection, we study the impacts of \( s, n, c \) and \( b \) on \( g^{*} \). First, given full CPU and network bandwidth availability, the changes of \( g^{*} \) is evaluated with multiple matrix sizes \( (s) \) running on different number of work nodes \( (n) \). And then, we evaluate the variation of \( g^{*} \) after taking away some CPU power \( (c) \) and network bandwidth \( (b) \).

5.4.1. The impacts of \( s \) and \( n \) on \( g^{*} \). Varying \( g \), we run LU on different numbers of work nodes \( (n) \) with different matrix sizes \( (s) \). As an example, Figure 4 shows the performance data of LU with different matrix sizes \( (2500x2500, 3000x3000, 3500x3500, \text{and} 4000x4000) \), each with 4 work nodes. Correspondingly, Figure 5 shows the results of the experiment with 12 work nodes. The curves shown in these two figures are obtained by applying shape-preserving interpolation [5] on the sample data. Both Figures 4 and 5 show that

- \( T_{comp} \) increases with \( g \). The reason is that a larger \( g \) causes more cache misses, which leads to more computation time.
- A smaller \( g \) implies a bigger number of shared objects and more frequent communication. Thus, \( T_{comm} \) decreases as \( g \) increases.
- As \( g \) increases, \( T_{sync} \) first decreases and then increases after \( g \) reaches some value. The reason is that when \( g \) is small, a large number of synchronization activities are
Based on the observed characteristics of \( T_{\text{comp}} \), \( T_{\text{comm}} \), and \( T_{\text{sync}} \), \( g^* \) exists for \( T_{\text{total}} \) though it is not very clear.

\( g^* \) is not quite sensitive to \( s \) (i.e., the matrix size in LU). Therefore, \( g^* \) needs not be adjusted when \( s \) changes.

From Figure 6, we see that \( T_{\text{total}} \), \( T_{\text{comp}} \), \( T_{\text{comm}} \), and \( T_{\text{sync}} \) all increase with a reduced \( c \). It is easy to understand the increase in \( T_{\text{comp}} \). The increase in \( T_{\text{sync}} \) is due to the increased waiting time for shared objects that now take longer time to compute. It is important to notice that, despite the increase in \( T_{\text{comp}} \) and \( T_{\text{sync}} \), the shapes and patterns of their curves remain almost identical to the curves with 100% \( c \) availability.

Figure 6 shows that \( T_{\text{comm}} \) increases too. The reason is that CPU is involved in building the socket connections in communication. Therefore, \( T_{\text{comm}} \) increases when less CPU processing power is available. What is more important is that the shape of the communication time curve changes after some CPU power is taken away—suggesting that a larger \( g \) is preferred since a larger \( g \) causes less frequent communication. As a result, \( g^* \) varies from about 200x200 (with 100% \( c \) availability) to around 350x350 (with 60% \( c \) availability). Thus, we conclude that \( g^* \) increases as the CPU contention intensifies, and this change comes solely from the fluctuation in the communication time. Other experiments on different matrix sizes and varied number of work nodes showed the same result.

### 5.4.3. The impact of \( b \) on \( g^* \)

We perform a similar experiment as in Section 5.4.2 by taking away 40% of the network bandwidth, and the result is shown in Figure 7.

From Figure 7, we can see that \( T_{\text{total}} \), \( T_{\text{comp}} \), and \( T_{\text{sync}} \) all increase slightly. The reason is that the program used for consuming the network bandwidth uses a small amount of CPU processing power. As a cluster computer always uses a high-speed network to connect the nodes (e.g., a Myrinet and a Gigabyte Ethernet are used in Prai-
rieﬁre), the contention on the network bandwidth does not noticeably slow down the communication in distributed applications. Actually, the time to set up the socket connections dominates the total communication time if the data size is not too large in each transmission. As a result, \( T_{\text{conn}} \) remains almost unchanged and \( g^* \) remains unchanged as the contention on the network bandwidth gets intense, as shown in Figure 7. Therefore, we conclude that the impact of \( h \) on \( g^* \) can be neglected. Similar experiments on other matrix sizes with different number of work nodes under varied network workloads also claimed the insensitivity of \( g^* \) to the variation of \( b \).

5.5. Adapting \( g^* \) to varying \( c \) for a given \( n \)

In most cluster environments, a user submits his/her application to the system scheduler by specifying the number of work nodes, \( n \), required by the application, among other parameters. In order to ﬁgure out a solution to adjust \( g^* \) with varying \( c \), for a given \( n \), we carry out a set of experiments to observe the changes of \( g^* \) with varied \( c \) values. As an example, Table 2 contains the experimental results on a matrix of size 3500x3500 with 4 work nodes. Note that \( g^* \) in Table 2 is obtained through the curve ﬁtting described in Section 5.3.

<table>
<thead>
<tr>
<th>Percentage of CPU power being taken away</th>
<th>Predicted value</th>
<th>Real value</th>
</tr>
</thead>
<tbody>
<tr>
<td>18%</td>
<td>246</td>
<td>247</td>
</tr>
<tr>
<td>25%</td>
<td>254</td>
<td>255</td>
</tr>
<tr>
<td>38%</td>
<td>275</td>
<td>274</td>
</tr>
</tbody>
</table>

5.6. Methodology

Summarizing the above observations, we conclude that \( g^* \) is sensitive to \( n \) and \( c \), and insensitive to \( s \) and \( b \). That is, when the problem size (\( s \)) and network bandwidth availability (\( b \)) change, \( g^* \) does not need to be adjusted. When the total number of work nodes (\( n \)) changes, it is necessary and reasonable to obtain \( g^* \) again through curve ﬁtting since \( n \) is usually an input parameter by the user. Given a distributed application and ﬁxed number of work nodes, by a user for the very ﬁrst time, it is worthwhile to collect some sample data and perform a curve ﬁtting as done in Section 5.5 to obtain a formula like Equation (6) so as to predict \( g^* \) whenever \( c \) changes. This predictive formula can then be stored in the knowledge base of Figure 1 for optimizing performances of future submissions of similar applications. We re-calculate \( g^* \) each time we re-run a distributed application. For some applications such as online collaborative applications and online gaming, it is possible to adjust \( g^* \) on the ﬂy.

6. Conclusion and Future work
In this paper, we have systematically investigated the advantages of object-based DSM systems, and especially focused on how to adaptively set the optimal shared-object size for a distributed application. Based on our experiments using the benchmark application LU, we have concluded that the optimal shared object size $g^*$ can be obtained through curve fitting. We have also observed that $g^*$ is highly sensitive to the total number of work nodes and CPU contention; whereas, $g^*$ is not quite sensitive to the problem size and network bandwidth contention. We have also speculated that a formula obtained through curve fitting could be used to predict $g^*$ in a dynamic runtime environment.

The methodology proposed in this paper had only been evaluated by one test case, i.e., the SPLASH LU benchmark, and its extent of generality remains to be established. In the future, more experiments need to be carried out on other test cases so as to observe whether comparable results could be achieved. Furthermore, we will implement the adaptive mechanism, completing the analysis component that is part of the overall approach (Figure 1). Moreover, we will investigate in details other advantages outlined in Section 3. In addition to supporting scientific applications, we plan to adapt the infrastructure of PJava to support on-line collaboration applications and collaborative computing.

7. Acknowledgement

This work was supported in part by a seed grant from the National Center for Information Technology in Education, a NSF SBIR Grant (DMI-0441249), and a Chinese NSF 973 Project Grant (2004cb318201). Research Computing Facility (RCF) at University of Nebraska-Lincoln provided the experimental environment—Prairiefire supercomputer, and Dr. David Swanson and Makoto Furukawa kindly supported the experiments. The authors would like to thank Yifeng Zhu and the anonymous reviewers for their comments to this work.

References


