Pleiad: A Cross-Environment Middleware Providing Efficient Multithreading on Clusters

Konstantinos I. Karantasis
High Performance Information Systems Lab
Computer Engineering & Informatics Department
University of Patras
26500 Rio, Greece
kik@hpclab.ceid.upatras.gr

Eleftherios D. Polychronopoulos
High Performance Information Systems Lab
Computer Engineering & Informatics Department
University of Patras
26500 Rio, Greece
edp@hpclab.ceid.upatras.gr

ABSTRACT
The engagement of cluster and grid computing, two popular trends of today’s high performance computation, has formed an imperative need for efficient utilization of the afforded resources. In this paper we present the concept, design and implementation of the Pleiad platform1. Having its origin in the proposition of distributed shared memory (DSM), Pleiad is a cluster middleware that provides shared memory abstraction which enables transparent multithreaded execution across the cluster nodes. It belongs to the new generation of cluster middleware that aside from providing the proof of concept regarding unification of the cluster memory resources, they aim to achieve satisfactory levels of performance and scalability for a broad range of multithreaded applications. First results from the performance evaluation of Pleiad appear emboldening and they are presented in comparison with an efficient implementation of MPI for the Java platform.

Categories and Subject Descriptors
D.1.3 [Software]: Programming Techniques—Concurrent Programming

General Terms
Design, Experimentation, Languages, Performance

Keywords
cluster middleware, shared memory programming, java

1. INTRODUCTION
Since its first implementations, the paradigm of distributed shared memory holds an important position among the models designated for high performance computing in the presence of demanding applications. Trying to achieve a successful compromise between the well established architecture classes of symmetric multiprocessors (SMPs) and distributed systems, modern DSMs are challenged to adapt to the current status of high performance computational environments. The most significant representatives of such environments are contemporary grids. These environments consist of a wide range of platforms that support concurrency, like clusters of SMPs and networks of multicore desktops. Moreover, their diversity is expected to grow even further with the advent of the first commodity mobile and embedded multiprocessor devices. Therefore, heterogeneity is a de facto convention which has to be considered when software is constructed for such non-uniform distributed platforms. Furthermore, being at the early stages of their development, grids are characterized by the absence of sufficient and successful programming models. The Java language, exceeding 10 years of existence already, is considered to be a notable alternative for parallel and distributed computing [7]. The Java platform, available on almost every uniprocessor and multiprocessor computer system, forms an interesting basis for efficient portable and interoperable computing across the heterogeneous grid environments. There are quite a few Java systems that provide abstraction of shared memory across distant nodes and they can be grouped into three main categories according to the implementation level [10]:

- The first category includes all systems that involve implementation at the Java Virtual Machine (JVM) level, providing to the programmer complete transparency. One of the main targets of these systems is to be able to run legacy Java applications unchanged on top of the cluster infrastructure. Many of these systems are also referred to as distributed virtual machines, judging by the impression of a single system image they convey to the Java application developer [23, 3].

- The systems that enforce shared memory consistency during the compilation phase or through bytecode instrumentation belong to the second category. These systems make use of standard JVMs, while aiming at the same time at satisfactory levels of transparency [21, 1].

- Finally, systems implementing shared memory abstraction that is based completely at the Java library belong in the third category. These systems are fully portable and ready to utilize the most recent advances in the Java platform development [11, 2].
In this paper we present Pleiad, our research prototype of a library level, Java-based cluster middleware. Every module of Pleiad has been implemented from scratch using the latest features introduced in the Java platform from version 5 and thereafter. The prototype presented in the current paper is unique in several aspects. Pleiad has the ability to interchange several implementations of its mechanisms during runtime and lay grounds for efficient tuning on every platform that is deployed. Regardless shared memory consistency maintenance, we have implemented both lazy release consistency and scope consistency, enhanced with ownership migration. Pleiad supports multithreading inside every processing node and its synchronization mechanisms are aware of the ability for multithreaded execution. Based solely upon standard Java library methods, Pleiad is fully portable across every Java platform that implements the specification of the Java language version 5 and beyond.

The rest of the paper is organized as follows. In section 2 we refer to the research efforts that pertain most to our system. Section 3 outlines the design concepts of Pleiad and describes the most important mechanisms of the platform. Section 4 provides the first indicative experimental results concerning the performance of Pleiad in comparison with MPI. Finally in Section 5 we draw our conclusions and refer to our future work.

2. RELATED WORK

Numerous research efforts have adopted Java at its early versions to provide cluster middleware, both efficient and easy to program [12, 19]. Nevertheless, besides providing a proof of concept, these efforts can be considered premature nowadays if we take into account the development of the Java platform during the last decade. This is because at its early versions, upon which these efforts were developed, the Java platform lacked many performance related enhancements and in some occasions presented serious inefficiencies [13, 14]. Just to name a few, the absence of non-blocking IO, the lack of Just-In-Time (JIT) compilation and the problematic memory model specification are among the most important issues that were resolved in the recent versions of the Java platform.

In the current paper we consider for comparison only the systems that are implemented at the Java library level and utilize the recent advances on the Java platform. These systems are directly related to Pleiad and include:

JavaSplit. Focused on offering transparent data sharing on top of clusters, JavaSplit, implemented by Factor et al. [11], is a runtime system that comes in conjunction with a Java bytecode compiler. Using instrumentation at bytecode level, JavaSplit succeeds to execute ordinary multithreaded Java applications using its distributed runtime system.

JavaSpaces. Introducet as a specification [20], JavaSpaces defines a programming model based on the Linda language and runtime system. According to Linda’s model, the system constructs the abstraction of a shared space based on the definition of tuples. Although it is mainly destined to operate as an object sharing facility it has been used for computational intensive applications in some cases. Up until now, one free as well as one commercial implementation have been released.

JuxMem. Antoniu et al. [2] present a research prototype that applies a peer-to-peer approach at the communication infrastructure of a library based DSM. JuxMem uses the JXTA peer-to-peer overlay to build a grid middleware, versions of which exist in both Java and C. To the best of our knowledge, JuxMem along with Pleiad are the only systems that support versions of scope consistency.

ProActive. Baduel et al. [5] implement a middleware that is based on the concept of components and utilizes Active Messages [22] to provide shared memory abstraction across distributed platforms. ProActive is implemented using the Java library and provides a complete environment for multithreaded execution across numerous grid or cluster runtime systems.

Finally, closely related to Pleiad and its concepts is a new generation of programming languages targeting parallel and scientific computation on top of physically distributed resources, the Partitioned Global Address Space (PGAS) languages. Among them, X10 [8] appears to have the most common objectives with Pleiad, however, at the moment it supports only multicore execution and a cluster enabled version is expected in 2009.

3. ARCHITECTURAL OVERVIEW

Drawing an inference from previous research efforts regarding systems that provide any kind of shared memory abstraction on top of physically distributed architecture, we observe that no policy or algorithm appears appropriately effective for every application category. For instance, while some consistency protocols respond efficiently to execution patterns that are dominated by read accesses, the same protocols may fail to scale when the application mostly incorporates write accesses.

For this reason, we strongly believe that the performance of modern cluster middleware is highly depended upon their ability to adapt to the application pattern either during execution runtime or with the presence of a short autotuning phase before the execution of the main application [4]. Therefore, the design of the Pleiad cluster middleware, to its whole extent, is infused by the potential of interchangeability between the various implementations of certain mechanisms without the need of recompilation. To achieve this goal, every significant mechanism is described through a well defined interface or abstract class which is backed by numerous alternative implementations. During execution there are hooks on the specific implementation in use, which can change even at runtime, through the use of specific transition methods. The interaction between the various modules, some of which are shown in Fig. 1, is realized exclusively through their well defined interfaces. Although this approach may sacrifice some transient code optimizations, it makes Pleiad one of the very few systems that provide the infrastructure for holistic adaptability of every crucial mechanism at runtime.

The architectural layout of Pleiad is presented in the figure above (Fig. 1). Pleiad is structured in three layers: (i) the Communication layer, comprising the necessary infrastructure for the communicating distributed threads, (ii) the Distributed Memory Management (DMM) layer, incorporating the actual policies, protocols and algorithms that provide shared memory abstraction and (iii) the Application Programming Interface (API) layer, exposing the functionality of Pleiad to the application programmer. In the next sections we describe the most important modules incorporated in every layer in order to obtain shared memory abstraction at object level.
3.1 Communication Layer

This layer holds the necessary modules for the realization of communication between the threads that are distributed over the multiprocessor nodes of the system. Regardless of the network infrastructure being used, the communication layer of Pleiad meets specific requirements in order to lay grounds for low latency data exchange. One of the most important features is asynchronous message handling. Pleiad exploits all the appropriate features introduced by the new I/O implementation in the latest versions of the Java platform. The design of the communication layer was inspired by well know design patterns that describe asynchronous execution. However, it was implemented from scratch to meet the high performance requirements of a cluster middleware. At its current version, Pleiad uses TCP/IP and defines on top of it a custom communication protocol based on message exchange. On each message, the actual data is preceded by or the header and actions are taken so that packet fragmentation is avoided along the network path.

Another important aspect is related to incoming traffic. As opposed to previous Java cluster aware implementations, Pleiad uses connection multiplexing as introduced in Java 1.4.2 and, thus, it is not obliged to dedicate a thread on every different connection. This fact, along with the use of non-blocking data buffers, gives the potential for greater scalability. Therefore Pleiad makes use of a sole service thread on each node, that co-exists with the threads of the multithreaded application that runs on top of Pleiad. This decision in the multicore era is expected to have imperceptible cost. Moreover Pleiad already supports multithreading in terms of server threads. However the default setting, with most clusters still incorporating dual or quad core nodes, is to deploy a single server thread in order to avoid synchronization delays on network resources.

3.2 Distributed Memory Management Layer

Pleiad, implemented using an object oriented language, relies on object sharing to provide shared memory abstraction across the cluster nodes. According to that, the minimum data sharing unit is every single, user defined, object. The sharing of objects is accomplished with the use of replicas for every object that is globally shared. Pleiad also defines an owner for every object - the home node - which, in turn, is responsible to deliver the most recent version of the object. Among the various issues concerning the implementation of shared memory abstraction [18], we identify three as the most critical in the current context: (i) the degree of concurrency, (ii) the consistency protocol and (iii) the synchronization mechanisms. In Pleiad, every one of these pylons constitutes a separate and independent module, which has its own interface and whose implementations are gathered inside the package of distributed memory management.

3.2.1 Concurrency degree

Over the several years of research in the area of distributed shared memory, it has been established that the concurrency degree is defined by the way, i.e. how, the accesses of shared data are allowed to occur at the extent of the distributed system. In its current version, Pleiad supports that at any time can exist multiple concurrent readers in every case and selectively (a) multiple concurrent writers for shared arrays and (b) a single concurrent writer for plain shared objects. Therefore these policies correspond to multiple-readers-multiple-writers (MRMW) access for shared arrays and multiple-readers-single-writer (MRSW) access for shared objects. This choice is justified by the presence of false sharing only in the case of distributed arrays.

In order to properly support the above concurrency modes we need to consider the policy that keeps the replicas updated. In the current context, upon every write, we follow the approach of invalidation for the replicas instead of the approach of the immediate update. Invalidation-based algorithms tend to perform well when the access pattern includes several consecutive writes between two read accesses. Nevertheless, the study of direct update-based versions of our algorithm is in our close interests as well.

3.2.2 Consistency Protocols

Up to the present, several systems that offer shared memory abstraction have been closely connected with a specific consistency protocol. This is due to the impact of these mechanisms on application performance. Pleiad, being part of a new generation of cluster middleware, does not utterly
rely on a specific consistency model. Alternatively, it incorporates mechanisms for dynamic and on-line swapping of consistency models implementations. In this early version of the prototype, we have implemented several variations of two of the most successful consistency protocols that have been proposed in the context of distributed shared memory. These protocols are Lazy Release Consistency (LRC) and Scope Consistency (ScC). As far as we know, Pleiad is the only system to have implemented and evaluated both LRC and ScC, while some systems have selected either to implement relaxed consistency relying solely on one of these protocols or to impose a more implicit consistency scheme by using for instance Active Messages. Firstly, we have implemented LRC which associates the synchronization mechanisms that are used in the whole program with the shared objects. After that we have extended our protocol to identify every scope as it is formulated by the use of locks and barriers and therefore support a ScC protocol.

The initial placement is done in several ways, including round robin, chunk based and random. However Pleiad supports home migration applying either a first touch scheme, when two consecutive writes occur the one after the other, or a threshold based migration scheme. These schemes allow the proper placement of shared objects in the case that they were initially misplaced. The information about this change is not directly propagated to every node of the system, but it is gradually conveyed along with the responses of future requests to the former home node.

3.2.3 Distributed Synchronization

Synchronization, being prerequisite for most shared memory programming models, is a key performance factor for a cluster middleware that provides shared memory abstraction. In the previous years distributed synchronization has gained significant attention as a research topic related to DSM systems. In the current context, we have chosen to implement mechanisms that are based on extensively tested suggestions, since we will be concentrating separately on distributed synchronization in the near future. Therefore, our solution is closely related to the one initially presented by Naimi et al. at [17] and enhanced by Mueller in such a way that it could be applied to middleware that support multithreaded on every node [15, 16]. In contrast to other Java-based systems that support only one or two types of synchronization leaving to the programmer the responsibility to simulate the rest forms of synchronization, Pleiad makes directly available through its API the two mostly used synchronization mechanisms, being mutual exclusion and barrier synchronization. The implementation of every such form of synchronization is discussed shortly below:

Mutual Exclusion. We adopted the approach of a token-based mutual exclusion algorithm. Before entering its critical section, every thread has to acquire the token. We distinguish between the cases of a read and a write request on a per token basis. As the token moves across the distributed system, a path of probable owners is formed. Such paths must be followed every time a thread issues a request for token acquisition. To reduce the path length, a path compression technique is applied, as described by Naimi et al. at [17]. Special care has to be taken in the presence of per-node multithreading. This is because on every request, the token is finally delivered to the requesting node and not to the specific thread of that node. In any case, the token is delivered to a node. Consequently, the contention inside the node is resolved using standard concurrency mechanisms from the Java library. Our implementation is equipped with two versions of the above mentioned mutual exclusion algorithm. The first one is fair with respect to the arrival of lock requests at the lock owner. The second one favors, to some extent, local threads, rather than remote node threads, in an attempt to exploit spatial locality.

Barrier Synchronization. For barrier synchronization we followed the approach of notification broadcasting. It is regarded as the simplest concept in order to implement barrier synchronization and behaves adequately on small scale clusters up to 16 nodes. The notifications are sent on a per node and not on a per thread basis, which means that in order to notify multiple threads on a node to proceed beyond the barrier the notification is only sent to the specific node. At this early stage, broadcasting was preferred not only due to its simplicity but also because the underlying available network did not impose any predefined hierarchical structure. However we are already in the process of incorporating other approaches based on the formation of hierarchical structures as well.

3.3 Application Programming Interface

Every object that is part of Pleiad API extends through inheritance the abstract class PleiadObject. Objects included in the Pleiad API to date, are summarized in Table 1. Considering the possibility of further enhancements, the API aims to preserve a key balance between, simplicity and abstraction as well as powerfulness and fine-grain expression of parallelism respectively.

Since Pleiad is a middleware that depends solely on the Java library, the intervention of the programmer is required at certain points for the access patterns of a multithreaded application to be expressed. Nevertheless, the effort of using the Pleiad API is no greater than the effort encountered when utilizing a collection, e.g. a HashMap or a LinkedList, from the standard library of the Java platform. Altogether, this extra effort is required in two occasions.

<table>
<thead>
<tr>
<th>Table 1: Object types exposed by the Pleiad API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>--------------------------------</td>
</tr>
<tr>
<td>PleiadObject</td>
</tr>
<tr>
<td>SharedObject</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>SharedArray</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>LockObject</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>BarrierObject</td>
</tr>
<tr>
<td>PleiadThread</td>
</tr>
</tbody>
</table>
Firstly, when the programmer has to declare which objects should be shared, as in Pleiad there are no objects that are shared by default. This is not an excessive requirement and it is valid for other systems as well, i.e. most cluster-enabled implementations of OpenMP for which sharing by default is disabled. In Pleiad the declaration of an object as shared is achieved by wrapping a specific java object in a new object of type SharedObject. Once this is accomplished, the Pleiad middleware is responsible for the distribution of the updates on the specific object, and the programmer just needs to specify the accesses to the object with the use of explicit get or set operations. Concurrent access patterns can be applied with the use of the synchronization objects that are available on these arrays.

As far as it concerns the execution spanning across the nodes of the cluster, it is carried out by user defined thread classes that extend the abstract class PleiadThread (possibly an interface in future releases). This way Pleiad supports with great flexibility both SPMD models that resemble MPI execution and MPMD that are closely related to shared memory multithreaded programming.

On Table 2 we demonstrate a sample use of Pleiad API. We sketch how we can achieve a multithreaded hypothetic computation. The presented code is quite close to the actual code that we wrote to implement the kernel applications that are discussed next in the evaluation section.

4. EVALUATION

First of all, in order to evaluate Pleiad on a broad context we have chosen two different cluster platforms to conduct our experiments. Although they have moderate scale, they possess some interesting characteristics. The first cluster is an AMD Athlon cluster, with one dual-core processor on every node and represents a typical case of a commodity cluster, which, of course, will be soon superseded by quad-core and many-core clusters. On the other side the second cluster consists of nodes that include a dual-core Intel Atom processor on each node with hyperthreading enabled. This second cluster can be considered as a representative of embedded, low-power, clusters that will soon start to emerge with the development of more sophisticated and more efficient embedded multi-core processors. The characteristics of both experimental platforms are summarized on Table 3.

Having as first class priority to compare Pleiad with well established systems which comprise comparable characteristics, we have chosen MPJ Express [6] for an implementation of MPI based on the Java platform. A Java based imple-
implementation of MPI was preferred among the various MPI implementations in order to concentrate on a comparison that is not implicitly affected by the differences on the language or the platform level. Nevertheless future comparisons will include systems that are closely related to Pleiad, providing shared memory abstraction on top of physically distributed resources, as well as systems that are implemented outside the Java platform.

The benchmarks that were used to evaluate both systems form two groups. The first group (Fig 2 and 3) includes basic collective operations that require cooperation between the deployed threads and they are traditionally provided by every MPI implementation. These benchmarks, because they involve mostly network communication, they were executed only on the Atom cluster. The second group (Fig 4 and 5) consists of kernels from the Java Grande Forum benchmarks. For the executions with MPJ Express we used the implementations of the benchmarks as they were provided with the source code of MPJ version 0.27 for both the micro kernels and the JGF applications. In Pleiad we obtained the same effect that the collective operations have in MPI, implicitly, using basic get and set operations on SharedArrays and interacting with the underlying consistency protocol using barrier synchronization. The port of the JGF applications to the Pleiad API was based on the original multithreaded version of the benchmarks[9].

As far as it concerns the collective operations we observe that Pleiad provides a far lower latency barrier synchronization mechanisms than MPJ, it performs better on operations Scatter, Gather and Allgather, and it is overtaken by MPJ on operations Broadcast and Alltoall. In order to explain these measurements we need to take into account that Pleiad uses an invalidation based mechanism to keep the copies of shared objects updated. This means that the updates are transferred following a "pull" pattern whenever a

get operation is taking place on the reader threads, rather than being pushed by the writer thread to the readers when the set operation takes place. The use of invalidations is not expected to perform better compared to the use of updates since the produced data are consumed right away after the barrier. This is evident at the Broadcast and the Alltoall benchmarks and is caused by the contention at the owner node. On the other hand, on Scatter, Gather and Allgather benchmarks the readers do not refer to the same object, and therefore the cost of updates is amortized. Concerning MPJ we observe a definite inefficiency in terms of inter-node multithreading, where in the cases of 2, or 4 threads per node the performance degrades a lot, in constrast to Pleiad which seems to respond well when more cores are offered to the applications, even in the presence of the service thread. Of course, because of the relaxed memory model, these updates take place at the moment of barrier synchronization.

The evaluation of the four Java Grande kernels produces analogous results on both experimental platforms. However, the impact of the service thread that executes aside the application threads on Pleiad is greater when intra-node multithreading is exercised at the Athlon cluster. This is more
Figure 4: JGF Section2: Series and SparseMatmult

Figure 5: JGF Section3: MonteCarlo and Raytrace

evident on this specific cluster because of the presence of only two cores. As the experimentation on the Atom cluster shows, this is not expected to further affect multithreading on a many-core system.

Concisely, Pleiad closely follows MPJ at the two out of the four applications, the Series and the Raytrace kernels, while it scales far better on the other two, the SparseMatMul and the MonteCarlo kernels. Concerning the first two benchmarks, this is a satisfactory result since these applications have a well defined and regular pattern that is quite optimized using MPI. In contrast to these kernels, the kernel that implements multiplication on sparse matrices, in the case of MPJ is mandated by an Allreduce operation which computes a sum operation on the application data. Under those circumstances these collective operations have a negative impact, especially in the case of intra-node multithreading. Pleiad, on the other hand, although it suffers low performance at the start mainly because of the big size of the arrays on executions with a few threads, it scales evenly throughout the whole execution series. Concerning the MonteCarlo kernel, while it is quite parallel, the MPJ faces scalability problems because it operates on arbitrary

5. CONCLUSIONS AND FUTURE WORK

This paper has presented the core infrastructure of Pleiad, a modern cluster middleware based on the Java platform. We have demonstrated some indicative experimental results that show that Pleiad equally competes with an efficient Java MPI implementation and under circumstances Pleiad can be proven more efficient than even raw message passing. Concerning its design concepts, Pleiad is one of the few fully portable multithreaded cluster middleware based on the Java platform and has a unique approach on the potential support of adaptability during runtime.

On the forefront of our future targets lies the will to enhance every performance critical mechanism of the presented cluster middleware and explore the limits of the proposed shared memory programming model in the environment of multicore cluster federations, as it will rapidly evolve in the near future. More specifically we soon plan to compare the currently available consistency protocols with an implementation of transactional memory in the context of Pleiad.
Finally in our short term intentions is to make the source code of our prototype publicly available for experimental comparisons via the World Wide Web.

6. REFERENCES


