Is Shared Memory Programming Attainable on Clusters of Embedded Processors?

Konstantinos I. Karantasis and Eleftherios D. Polychronopoulos
High Performance Information Systems Laboratory
Computer Engineering & Informatics Department,
University of Patras
26500 Rio, Greece
1 kik@hpclab.ceid.upatras.gr, 2 edp@hpclab.ceid.upatras.gr
http://pdsgroup.hpclab.ceid.upatras.gr

1. Introduction

In the past the concept of embedded system was limited in small, special-purpose, electronic devices designated to perform one or a few dedicated functions. Later technological advances introduced special purpose processors and real-time operating systems in embedded systems, expanding the notion of embedded to include more complex though more critical applications. In recent years the processor manufacturers introduced new powerful embedded processors, promoting a new feeling of embedded systems. These processors are called to cover the needs that the new era in computing defines. Nowadays the user holds devices that have to be mobile, low-power as well as powerful, equipped with sophisticated software. These include mobile computers, smart phones, palm devices and many others. For that purpose new powerful embedded processors like ARM, XScale, Alchemy have already been introduced. One of the most recent addition include was made last year (2008) with Intel shipping Atom™[1], a new dual-core low-power embedded-processor with hyperthreading on each core, resulting in four (4) threads of execution in each die.

Having all this low-cost, new technology available and the fact that already millions of these types of embedded systems are available worldwide, it would be interesting to investigate the feasibility for these devices to dynamically form small "computing islands", and efficiently support in that way every day applications and even solve classical scientific problems. In this paper we present our Java based middleware platform Pleiad, executing on top of a small embedded cluster. Our cluster is build out of eight (8) Atom™ Dual-core 1.6GHz embedded-processors, on Intel Mini-ITX D945GCLF2 boards. In total our embedded cluster has 32 threads of execution available.

In Pleiad every module 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. Regarding 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 describes the experimental platform and Section 5 provides the first indicative experimental results concerning the performance of Pleiad in comparison with an implementation of MPI in Java, MPJ Express. Finally in Section 6 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 [2], [3]. Nevertheless, besides providing a proof of concept, these efforts are 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 [4], [5]. Just to name a few, the absence of non-blocking IO, the lack of 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. Next we refer to systems that presented realistic results and are mostly related to Pleiad:

JavaSplit. Focused on offering transparent data sharing on top of clusters, JavaSplit, implemented by Factor et al. [6], 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. Introduced as a specification [7], 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. [8] 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. Buduel et al. [9] implement a middleware that is based on the concept of components and utilizes Active Messages [10] 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.

3. Architectural Overview

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.

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. Therefore the design
of Pleiad is concentrated on (a) providing several alternative implementations for every distinct mechanism that forms a well defined module on a certain layer and (b) being able to interchange these implementations even during runtime in a transparent way. In Pleiad this is possible because the interface of every mechanism is decoupled from its implementation. The selection of the appropriate implementation is done during runtime in contrast to most other systems where the implementation is chosen during compile time. 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 basic elements to achieve such a transition in Pleiad is the construction of a factory class that can produce the desired implementation of the mechanism and the provision of the appropriate methods that will designate the interchange between two implementations. Next we describe shortly the key components of every layer.

3.1. Middleware Infrastructure

Forming the basis over the physically distributed resources the Communication layer holds the necessary modules for the realization of communication between the threads that are distributed over the multicore nodes of the system. Connection multiplexing and asynchronous data handling is applied making realizable the deployment of a single service thread in a scalable manner. In that way the CPU resources that are deprived of the application remain at the minimum. Currently a TCP/IP implementation is present in Pleiad mainly due to the available network infrastructure. Nonetheless, the inclusion of higher level constructs like RMI and MPI, or hardware specific protocols will involve implementation of interfaces only at the Network Layer.

Pleiad, implemented using an object oriented language, relies on object sharing to provide shared memory abstraction across the distributed resources. Consequently, the minimum data sharing unit across the system is every single, user defined, object. Therefore, for every object that sharing is enabled, an owner node is defined and replicas are constructed to the other nodes according to the reference pattern on the specific object.

Among the various issues concerning the implementation of shared memory abstraction [11], we identify three as the most critical in the current context: (i) the degree of concurrency, (ii) the consistency model and (iii) the synchronization mechanisms. The degree of concurrency is defining whether multiple threads can concurrently read or modify the state of shared object. In Pleiad multiple readers/single writer access (MRSW) is permitted for single shared objects whereas multiple writers access (MRMW) is achievable for shared arrays. Concerning the memory model of Pleiad, several relaxed consistency models are supported including scope consistency (ScC), release consistency (RC) and lazy release consistency (LRC). These models define that the state of the shared data on the distributed system are valid across synchronization points. These points are available through the third basic mechanism of the DMM layer, the synchronization module. Pleiad supports synchronization of the distributed threads based on barrier synchronization and mutual exclusion using locks.

Table 1. Object types exposed from Pleiad API

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Basic Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>PleiadObject</td>
<td>- abstract base class -</td>
</tr>
<tr>
<td>SharedObject</td>
<td>Object void get( ), set(Object obj)</td>
</tr>
<tr>
<td>SharedArray</td>
<td>void set(int i, Object obj)</td>
</tr>
<tr>
<td>LockObject</td>
<td>void lock( ), unlock( )</td>
</tr>
<tr>
<td>BarrierObject</td>
<td>void await( )</td>
</tr>
<tr>
<td>PleiadThread</td>
<td>void run( )</td>
</tr>
</tbody>
</table>

3.2. 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. 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 initial placement and the distribution of the updates on the specific object. The programmer just needs to specify the accesses to the object with the use of explicit get or set operations.
Table 2. Example use of Pleiad API

```java
public class Paradigm {
    ...
    public void deploy(String[] args) {
        SharedObject x = new SharedObject(new Integer(0));
        int chunk = 200;
        SharedArray v = new SharedArray(1600, chunk);
        LockObject mutex = new LockObject();
        BarrierObject bar = new BarrierObject(numOfThreads);
        for (int i = 0; i < numOfThreads; i++)
            Pleiad.startupThreads(args);
    }
    ...
}
```

(a) Application common class

```java
public class ParadigmThread extends PleiadThread {
    public void run() {
        iStart = size * id;
        iEnd = size * (id + 1);
        for (int i = iStart; i < iEnd; i++)
            num = compute(v.get(i));
        mutex.lock();
        sum = x.get();
        x.set(sum + num);
        mutex.unlock();
        bar.await();
    }
    ...
}
```

(b) Application thread class

Concurrent access patterns can be applied with the use of the synchronization objects that are also summarized on Table 1. In any case the assumptions about the validity of the returned objects must take into account the underlying relaxed memory consistency model that is currently in use by the Pleiad middleware.

A more powerful construct as opposed to the SharedObject is the SharedArray. This kind of object implements the notion of arrays, whose elements are transparently distributed across the nodes of the cluster. As we already mentioned a multiple readers/multiple writers access pattern can be applied concurrently on these arrays. Apart from one-dimensional and two-dimensional arrays, Pleiad supports shared arrays that span over arbitrary number of dimensions.

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 release). 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 reference application benchmarks of Java Grande Forum that are discussed next in the evaluation section.

Table 3. Experimental Platform and Settings

<table>
<thead>
<tr>
<th># nodes / # threads</th>
<th>8 / 32</th>
</tr>
</thead>
<tbody>
<tr>
<td># Cores</td>
<td>16 with HyperThreading enabled</td>
</tr>
<tr>
<td>CPU Type</td>
<td>Intel® Atom™ 330 1.60GHz</td>
</tr>
<tr>
<td>Cache/RAM</td>
<td>512KB / 2048MB</td>
</tr>
<tr>
<td>Network</td>
<td>Gigabit via Gigabit Switch</td>
</tr>
<tr>
<td>OS/JVM</td>
<td>Fedora 9 / Linux Kernel 2.6.25</td>
</tr>
<tr>
<td>JVM</td>
<td>Sun Java 6 (JVM 1.6.0_10)</td>
</tr>
<tr>
<td>Parameters</td>
<td>java -ms1024M -mx2048M</td>
</tr>
</tbody>
</table>

4. Experimental Environment

Our experimental platform consist of a small scale embedded cluster using eight low-power chassis (Table 3). Each chassis is equipped with one Atom™ dual-core hyperthreaded embedded processor which operates at 1.6GHz. Every node has 2GB DDR2 677/533 MHz of RAM and all the nodes are connected using a Gigabit switch. In aggregation the computational power of our cluster comprises 8 processors, 16 cores and 32 kernel threads.

This is to our knowledge one of the first evaluations of scientific, CPU intensive, applications on this new embedded processor. Atom™, the smallest processor these days, uses in-order execution with a relatively big pipeline and aims to achieve acceptable levels of performance for ordinary general-purpose applications, being at the same time considerably low power.

5. Evaluation

In the current context we provide our first, though insightful, comparison of Pleiad with its well established counter-proposal, the message passing interface (MPI) on a clustered environment with the afore-mentioned characteristics. As a reference implementation we have chosen MPJ Express [12] which is a recent and efficient implementation of MPI that relies on the latest advances of the Java platform. A Java based implementation of MPI was preferred 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
Figure 2. Collective operations I

Figure 3. Collective operations II

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 currently presented benchmarks 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. The second group (Fig 4, 5 and 6) consists of kernels from the Java Grande Forum Benchmarks [13]. 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. 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.

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 superseded 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 contrast 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 experimentation with three applications of the Java Grande Forum benchmark suite shows that Pleiad closely follows MPJ at the two out of the three applications, the Series and the Raytrace kernels (Fig. 4 and 6). This is a satisfactory result since these applications have a well
defined and regular pattern that is quite optimized using MPI. Unlike 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 that applications this kernel does not perform well, especially using inter-node multithreading. Pleiad, on the other hand although it starts having lower performance especially because of the big size of the arrays on executions with a few threads, it scales throughout the whole execution series.

6. Conclusions and Future Work

This paper has presented the first evaluation of a modern cluster middleware based on the Java platform on top of a cluster comprising a new generation of embedded processor, the Intel’s Atom™ processors. We have demonstrated some indicative experimental results that show that Pleiad equally competes with an efficient Java MPI implementation and under circumstances can be proven more efficient than raw message passing. Concerning the design of its concepts, Pleiad is one of the few fully portable multithreaded cluster middleware based on the Java platform and has a unique approach concerning 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 non-uniform, embedded and high-end, 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 non-commercial use via the World Wide Web. In that way Pleiad will be free for experimental comparisons with similar research platforms as well as open for valuable comments that will result on further improvements.

References


