Towards an Adaptable Middleware for Parallel Computing in Heterogeneous Environments

João Saramago
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
2829-516 Caparica, Portugal

Diogo Mourão
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
2829-516 Caparica, Portugal

Hervé Paulino
CITI / Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
2829-516 Caparica, Portugal
Email: herve@di.fct.unl.pt

Abstract—The adoption of the multi-core processor design has spawned a wide variety of CPU configurations, namely in the number and nature of the hosted cores. The cluster organization of such CPUs is particularly sensitive to this issue, since applications must be aware of such heterogeneity in order to fully exploit the potential of the underlying hardware. The existing frameworks for cluster computing do not fully address this issue, relying on the programmer to handle many of the concerns associated to the aforesaid heterogeneity, burdening him with non-functional details. To this extent, we propose an adaptable middleware for Java High-Performance Computing in heterogeneous clusters. Adaptability is obtained by trusting the runtime system with the management of thread and data placement, and by equipping it with a layer able to adjust its implementation to the particularities of the underlying software and hardware stacks. In this paper we address both the middleware’s programming model and runtime system, as well as some initial performance results that attest the validity of the approach.

Keywords—High-Performance Computing, Multi-core Architectures, Virtualization

I. INTRODUCTION

The shift from clock to core scaling in CPU design spawned a wide variety of processor architectures and configurations. The major vendors build of base architectures to offer a panoply of configurations with diverse clock speeds, number of cores and memory subsystems (with distinct levels of cache hierarchy and cache sharing).

This status quo has a clear impact on cluster computing, since applications must be aware of such heterogeneity, in order to fully take advantage of the underlying hardware. The programming of such architectures requires different models for intra- and inter-node communications. Although the foreseen scaling of the number of cores cause multi-core architectures to increasingly share characteristics akin to distributed memory environments, the data copy overhead of message-passing communication within a node is still too expensive when compared to the shared-memory alternative.

Some approaches, such as OpenMP/MPI hybrid programming [1], expose all these details to the application developer, forcing him to manage communication and parallelism at both levels. Others, mostly component- or object-oriented, provide higher levels of abstraction, either by encapsulating remote communication in special objects or simply by providing Single System Images (SSI), which hide the distributed nature of the execution environment by presenting a shared-memory model.

The Java language is assuming an particularly important role in this higher level side of the spectrum. Its popularity has been consistently growing, as High-Performance Computing (HPC) is slowly making its way into the mainstream of computing [2]. The performance gap between the execution of Java and native code has been considerably narrowed, allowing for the advantages of the language to shine, namely: cross platform interoperability (a feature very important in this field), and a widely accepted programming model with simple constructs for multi-threading and a comprehensive API for network communication.

This trend fostered the proposal of many Java-based HPC systems, ranging from the implementation of known message-passing libraries, such as the MPI standard with MPIJ [3]; middleware systems, of which we highlight Proactive [4], Parallel Java [5] and MapReduce [6]; language extensions, such as HPJava [7] and Titanium [8], to new Java-inspired concurrent programming languages, such as X10 [9] (and eventually Fortress [10], if a cluster runtime is ever available).

With the exception of Proactive (and Fortress), all the featured programming models converge in one of fundamental issue: no room is left for the compiler to generate processor specific code, nor for the runtime system to adapt itself to the characteristics of the underlying hardware:

- HPJava and Titanium feature a static flat SPMD model that forces the programmer to use other mechanisms to explore intra-node parallelism;
- Parallel Java requires hybrid message-passing/shared-memory programming;
- MapReduce is mostly limited to communication and synchronization free problems;
- X10 burdens the programmer with the scheduling of the tasks, compromising efficient composability and scalability.
The Proactive programming model is more suitable for these environments, namely its use of hierarchical data-parallelism, popularized by Sequoia [11]. Nonetheless, most of this endeavour is performed at system integration level rather than at the programming level.

At runtime, all these systems rely on the JVM for handling platform heterogeneity. The node configuration discrepancies must be identified and managed by the programmer. Moreover, no support is given for heterogeneous nodes, such as the ones that feature Graphic Processing Units (GPUs). It is our opinion that this is a crucial issue nowadays. Work on the addition of GPU support in existing systems and languages [12] have showcased the limitations of overexposing the underlying execution model to the programmer. It is our opinion that these must be handled by the compiler and the runtime system, the programmer should only be responsible for the identification of what should be parallelized and not be burdened with the how.

To this extent we propose ELINA, a Java middleware for the development of parallel applications across a wide range of architectures, namely shared memory machines, such as desktops and laptops; clusters of such machines, and GPUs\(^1\). For this purpose it provides a high-level programming interface grounded in the notion of active object [13] (that we refer to as services) and places, virtual processors that can be mapped into available physical nodes or cores. The concept is similar to the place and locale abstractions found, respectively, in the X10 [9] and Chapel [14] programming languages. These, however, define a place (or locale) as a portion of the partitioned address space plus the set of threads that operate upon that same portion. ELINA does not feature a native partitioned address space. We are closer to the definition of place found in [15], in the sense that a place virtualizes the available processing units, supplying an execution environment for services. Place awareness is not mandatory, to further his level of abstraction the programmer may neglect the absolute and relative mapping of the services into the available resources.

Task and data parallelism is expressed through a set of annotations, similar to OpenMP [16], which is suitable for compiler and runtime system optimizations. For instance, GPU support can be included without requiring novel programming constructions.

This programming model is backed-up by a runtime system whose architecture is inspired in that of an operating systems, in the sense that it not only provides a well defined interface to the programmer, but also specifies an interface for the integration of different technologies, enabling the adaptation of the middleware to the nature of target architecture. This adaptation layer supports multiple levels of application: processor, place, cluster and inter-cluster. Thus, a given configuration may impact the system at different levels.

The following section describes ELINA’s programming model and runtime system. Section III evaluates the system from a performance perspective by resorting to applications of the Java Grande benchmark suite [17]. Section IV compares our work with others that use Java as a tool for HPC. Finally, Section V presents our final conclusions and prospective future work.

II. THE ELINA FRAMEWORK

The ELINA framework was designed to efficiently support the execution of Java applications across heterogeneous execution environments, namely heterogeneous clusters and heterogeneous nodes. For that purpose, it features a modular architecture that delegates functionalities to pluggable modules, which are used to adapt the middleware to the specific nature of the target hardware. The overall architecture is structured as follows:

- **API** - centred around the concepts of active object (service) and place, it is mostly an annotation based interface that induces a declarative programming model for expressing both task and data-parallelism;
- **Core** - concentrates all the technology independent logic of the middleware, and;
- **Technology Abstraction Layer (TAL)** - an adaptation layer that enables the integration of distinct technologies to adjust the system as a whole and its particular instances to the characteristics of the underlying software and hardware stacks.

One of the goals of this work is to provide an architecture independent interface that enables the execution of the same code across multiple architectures, hence providing support for heterogeneous environments. With this objective in mind, ELINA can be used either as a library for running parallel computations on a stand-alone machine or as a middleware that executes multiple concurrent applications on top of a

\(^1\)The GPU support is currently under development
distributing infrastructure, such as a cluster. In the second case, the execution environment is composed by multiple interconnected instances, each defining a different place. Figure 1 illustrates the overall design of the system.

The remainder of this section further details the middleware’s programming model and runtime system.

A. The Programming Model

An application is constructed by composing a set of services that are distributed among the available places, according to a configurable pre-defined scheduling policy.

Services build on the active object design pattern to provide a flexible programming model for both multi-cores and clusters of multi-cores. The edification of inter-service communication on top of method invocation induces a message-passing model suitable for distributed memory environments.

The absence of a global address space constrains the passing of memory references as arguments in the invocation of service operations. This goes against Java’s usual semantics and does not contribute to improve performance in current multi-core processors. The overhead of copying data may still impose an undesirable performance penalty in many applications. Thus, services executing in the same place can pass references to shared objects, implicitly, defining another vessel of communication.

Services can be autonomous, in the sense that they can perform computation autonomously or passive, only executing as a response to an external stimulus (a method invocation). Autonomous services are automatically triggered as soon as the application is deployed, and execute concurrently with the remainder. A special autonomous service, entitled application, serves as an entry point to the execution of an application in a cluster environment, much like a Java main class. Figure 2 illustrates the programmer’s view of the computation.

ELINA’s programming interface is mostly annotation based. Parallelism may be expressed through a collection of Java annotations or by invoking methods on the ServiceProvider class, the base class for service implementation. Services must explicitly declare the interface they are implementing by extending the Service interface. Only the methods that compose such interface are remotely accessible. We are thus not in the presence of a classic SSI.

1) Task Parallelism: Methods of a service interface can be explicitly annotated as tasks that run concurrently within a place. Being active objects, services decouple the invocation from the execution of task annotated methods. Such feature combined with an asynchronous invocation mechanism provides the framework for the concurrent execution of both caller and callee, until the former requires the result computed by the latter.

The method’s annotation is transparent to the invoker, which does not have to modify its calling convention. The code of all @Task annotated methods, and subsequent invocations, are instrumented to introduce implicit futures.

2) Data Parallelism: Data parallelism in ELINA is also available at method level. A method defined as a task in the service’s interface can be further annotated - in its implementation - to employ a Single Operation Multiple Data (SOMD) execution model [18]. This model applies a Single Program Multiple Data (SPMD) approach to the execution of methods, enabling their parallel application to distinct partitions of the input arguments. The paradigm is exposed to the programmer as a three stage process, whose description follows:

Distribute - partitions an input argument to obtain a collection of elements of the same type;
Map - concurrently applies the annotated method to each partition of the input arguments;
Reduce - reduces the result produced by the previous stage to compute the method’s result.

Listings 1 and 2 illustrate, respectively, the specification and implementation of the classic matrix multiplication operation, which is included in a math service. The implementation exemplifies use of the SOMD approach. The classes that embed the desired distribution and reduction policies are supplied as arguments to annotations @DistributionPolicy and @ReductionPolicy, respectively. The former must be applied to a specific argument, while the scope of the latter encompasses the whole method. Distributions and reductions are reified by interfaces Distribution<T> and Reduction<PR>, respectively.

```
public interface MathService extends Service {
    @Task
    int[][] matMult(int[] a, int[] b);
    ...
}
```

Listing 1: A service interface with Task annotated methods
the interface (or interfaces depending on the aggregation behaviour) of the aggregated services, and therefore can itself be aggregated, spawning an hierarchical structure. Currently four aggregation behaviours are supplied:

- **Distribute-Map-Reduce**: applies the SOMD paradigm across multiple instances of a given service, enabling the hierarchical use of this parallelizing strategy. The input arguments are recursively distributed from the top-level aggregation down to the leaf services, and then among the computational resources available at the places hosting such leafs. A DMR aggregation of `MathServiceProvider` (Listing 2) is expressed as follows:

```java
ServiceAggregator.DMR(MathServiceProvider.class, nrOfInstances)
```

and returns a service of type `MathService`. Figure 3 illustrates the hierarchical application of this parallelizing strategy.

- **Service Pool**: a pool of services of a given type. Any invocation targeted at the aggregation is distributed among the aggregated service providers according to a configurable scheduling policy.

- **Partitioned Data Structure**: distributes a key addressable data structure along the aggregated services. A programmable partitioner determines the actual location of the data.

- **Facade**: provides a single entry point for a set of services. A uniform interface permits the retrieval, and subsequent invocation of any of the aggregated services.

4) **Synchronization**: ELINA synchronization mechanisms comprehend atomic blocks, Mesa-style conditions and hierarchical barriers. At the time of this writing, only the first can be expressed through an annotation (`@Atomic`), which can only be applied at method declaration level. Annotations for the remainder operations are still in the work, thus these must be performed through the explicit invocation of methods from the `ServiceProvider` class.

The hierarchical barriers are close to Habanero’s phasers [19]. They allow for multiple tasks to register and synchronize, and for barriers to be hierarchically composed. The latter is particularly useful within service aggregations - service instances located at a given place can synchronize at a local barrier while a global synchronization is issued.

### B. Place Awareness and Affinities

The service to place mapping is controllable through the exposed API by specifying affinity relationships. These steer the scheduling of the application imposing restrictions on the scheduling algorithm. The middleware supports multiple levels of affinities, although currently we only allow affinities between services, and between services and places. The former assures that related services execute at the same place, while the latter simply forces the scheduling of the given service to the specified place. Current work focuses on

![Figure 3: Hierarchical Distribute-Map-Reduce](image)

Listing 2: Application of the SOMD execution model

```java
public class MathServiceProvider implements ServiceProvider {
    @ReductionPolicy(policy=MatrixRed.class)
    public int[][] matMult(@DistributionPolicy(policy=LineDist.class) int[][] a, @DistributionPolicy(policy=ColDist.class) int[][] b) {
        int aRows = a.length;
        int bColumns = b[0].length;
        int[][] result = new int[aRows][bColumns];
        for (int i = 0; i < aRows; i++)
            for (int j = 0; j < bColumns; j++)
                for (int k = 0; k < a[0].length; k++)
                    result[i][j] += a[i][k] * b[k][j];
        return result;
    }
}
```

3) **Service Aggregation**: The hierarchical composition of services is specially aimed at distributed environments. Services can be aggregated according to a pre-established set of behaviours. The process generates a new service that inherits
intermediate levels of affinity, for example, between services and hardware requirements, such as a lower bound for the number of cores or the existence of a particular GPU.

At runtime, a service can discover in which place it is executing, and even explicitly migrate to another one. This provides the tools for the implementation of dynamic adaptation strategies based on locality. Moreover, the place abstraction features place-wide synchronization mechanisms, allowing for multiple services to synchronize themselves at a given place.

C. Runtime System

In this subsection we take a closer look at core and TAL layers, with special emphasis on the latter. The core layer implements the middleware’s logic. It is composed of multiple modules that provide for the deployment and execution of applications. These include managers for applications, services, task execution, communication and synchronization. Each application executes within a sandbox with dedicated class loaders and client connections. This approach facilitates the seamless transition from the stand-alone to the cluster environment by redirecting the application’s standard input, output and error to the client’s console.

The TAL layer enables the adaptation of the upper software layers to the specificity of the target execution environment. For that purpose it supports the plugging of software modules, which sustain the functionalities supplied by the core layer. These modules (or adapters) must comply to a set of Java interfaces that determine the contract between ELINA and external technologies. A functionality driven explanation of the middleware’s configurable behaviours follows:

Task execution: the core layer offloads the execution of tasks to one or more pool of worker threads. The actual implementation of these pools is relegated to the TAL layer, enabling distinct implementations and scheduling policies according to the target hardware.

Communication: the core’s communication module concentrates all inter-place communication, namely communication and synchronization between geographically dispersed services, the migration of services, and load balancing information. The communication technology and protocol are delegated in the TAL layer.

Synchronization: the core layer exposes an abstract factory for the concrete monitor and barrier factories provided by the TAL layer. The relegation of both functionalities to this layer allows, on one hand, the implementation of pessimistic and optimistic approaches for atomic blocks and, on the other hand, the experimentation of multiple barrier implementations, e.g. based on Java’s phasers and cyclic barriers.

Distribution and Reduction: most of this work is relegated to the TAL layer with the motivation of enabling the use of different types of architectures (CPUs, GPUs and clusters). Each stage features its own adapter, permitting the use of distinct hardware in both stages and with that overlap the execution of the map and reduction stages.

Service scheduling: the core layer manages the location of services across the available places according to the scheduling and load-balancing heuristics specified in the TAL layer. Three adapters adjust the middleware’s configuration at this level: scheduling algorithm, load balancing algorithm and node ranking. The latter establishes a rank of the available places that is used by the remainder to determine the services’ location.

The core provides a work load statistic module that can be used to implement the ranking heuristic. The overall location of the services is also influenced by the affinity relations defined in the application (Subsection II-B).

The TAL layer is itself structured in multiple levels, each defining a scope of application: Processor - processor or accelerator level; Place - place (or network node) level; Cluster - cluster level, and; Web - inter-cluster (Web) level; Currently, only levels Place and Cluster are being used, inter-cluster support is on the works.

Examples of adapters with different scopes are the Communication, which provides for cluster-wide (or even inter-cluster) communication, and the MonitorFactory, whose scope is confined to a single place. Some adapters, however, may have instances at multiple levels. An example is the Distribution adapter, which at the cluster level distributes the work amongst all the aggregated service instances, while at the place level it distributes the work amid the local computational resources, e.g. multiple cores (abstracted in a pool of threads) or any supported accelerator. The TaskExecutor follows the same methodology, when in the presence of a service pool aggregation. At cluster level it offloads the task to one of the aggregated service instances, whilst at place level it offloads it to the installed pool of workers.

III. EXPERIMENTS

We are currently porting the Java Grande and NAS benchmark suites to our programming model. This section presents the performance evaluation of three of such applications (from Java Grande), namely:

Crypt - IDEA encryption and decryption on an array of given size (parameter: array size in bytes);

Series - computation of the first given Fourier coefficients of a pre-determined function (parameter: number of coefficients), and;

MonteCarlo - a MonteCarlo approach to a financial simulation (parameter: number of sample time series).

We experimented with work loads of different orders of magnitude, based on the benchmark’s reference parameters.

Table I presents such configurations and the associated sequential execution time (in milliseconds).
All measurements were performed on a cluster composed of eight dual processor nodes: five featuring two Quad-Core AMD Opteron 2376 CPUs at 2.3 GHz and 16 Gigabytes of main memory, and three featuring two Dual-Core Intel Xeon E5504 CPUs at 2.0 GHz and 8 Gigabytes of main memory - all running the 2.6.26-2 version of the Linux operating system. Each of these nodes executed an instance of the middleware, configured to use Java NIO inter-node communication; a thread-pool tailored for data-parallelism (as many workers as cores); hierarchical DMR that, at a place, uses the installed thread-pool; a scheduling policy that favours the Quad-core nodes. The following XML file depicts this configuration.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="sable" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="sable_config.xsd" debug="false">
  ...
  <adapters>
    <communication communication.NIOCom"/communication>
    <synchronization> synchronization.LockBased"/synchronization>
    <barrier> synchronization PhaserBased"/barrier>
    <copy> cloning.DeepCopy"/copy>
    <distrReduce_levels>
      <distrReduce_level level="Node"> collective.DistRed"/distrReduce_level>
      <distrReduce_level level="Cluster"> collective.DistRed"/distrReduce_level>
    </distrReduce_levels>
    <scheduling rank=" scheduling.rank.LexicalRank"> scheduling.RoundRobin"/scheduling>
    <taskExecutor configuration="dataParallel"> TaskExecutor. TaskExecutorPoolThreads"/taskExecutor>
  </adapters>
</config>
```

The speedups obtained for each application in relation to the sequential implementations are depicted in the graphs of Figures 4 and 5. The first showcase the evolution as the number of places increase, while the second presents a more fine-grained, core level, view of those same measurements. Note that the execution environment is heterogeneous: the first five places feature 8 cores, whilst the remainder three only 4.

The work load of the Crypt parameter configurations are quite small, and thus not good candidates for impressive speedups in cluster environments. Nonetheless, classes C and D obtain quasi-linear speedups up to 5 places. From that point on, the inclusion of the less performant nodes reduces speedup, since the overall execution time is dictated by the tasks running in these slower nodes.

The longer execution times of the Series benchmark provides good results in every class, culminating with super-linear speedups in Class C.

MonteCarlo stressed the communication adapter because each sample produces a result. Thus, the class B configuration produces 60000 results, which total more that 300MBytes. The Java serialization overhead associated to the transmission of this data greatly limited the performance gains, which although improving with the increase of the number of places, never outperformed the performance of a single, 8 core, place. We are currently tackling this issue, hence the measurements here presented do not account the emission of this data. In this context, this benchmark essentially measures the scalability of the work distribution strategy, which produced quite good results.

### IV. RELATED WORK

Single System Image (SSI) distributed JVMs automatically enable the execution of Java programs in cluster environments. However, the language does not feature native constructs to express parallelism at this level and thus the programmer is not able to control the locality of the computation nor decompose its problem for parallel execution. cJVM [20], the Cluster JVM, is a reference work in this field. Although, remote method invocation has been optimized, it still imposes a significant overhead when compared to a local invocation.

HPC tailored proposals equip Java with mechanisms to express distributed parallel computations. Parallel Java [5] is a library and a middleware for parallel and distributed computing on SMP nodes and clusters of such nodes. The programming model transposes to Java the concepts OpenMP and MPI, allowing for both the shared-memory and message-passing paradigms. Inter-node communication is akin to MPI, and thus explicit in the code. Moreover, the programmer must program according to the nature of the hardware, instead of delegating such adaptation to the runtime system layer.

The ProActive [4] Java middleware (and GSM, its extension to the Grid) builds from the component-based pro-
Programming model to assemble sets of active objects. The use of active objects hides remote communication behind regular method invocation (much like RMI). Proactive distinguishes itself from the remainder by allowing the hierarchical composition of components (through the Fractal component model). This, however, is done at the integration rather than at the programming level. The components are defined through XML files that specify component interface, composition, and requirements. The focus is fundamentally on the construction of parallel applications by integrating existing software modules. Our work shares some ideas with Proactive, namely the use of active objects and their composition, but removes itself from the port model and its provides/uses binding mechanism. Also, all composition is performed at the programming level.

MapReduce [6] is an software architecture proposed by Google, which has fostered many implementations. It explores data parallelism by applying functions to large data sets, an approach close to our use of Single Operation Multiple Data [18], as will be described in Subsection II-A. This model is suited for processing large data sets, but its file based communication and restrictive programming model limits its range of application. The lack of flexibility of runtime system’s architecture has also been thoroughly exposed, as the proposal of new features and the support of new platforms, such as multicore nodes [21], have required partial or integral re-implementations.

Regarding the systems’ adaptability, both Proactive and the reference MapReduce implementations allow for the parametrization of the scheduling policies. Proactive also features the concept of multi-node task topology, which allows for the association of pre-defined policies, such as best proximity, to steer the scheduling algorithm. This approach is less powerful that our affinity based scheduling, but eventually simpler to use at the moment.

HPJava [7] and Titanium [8] extend the Java language with array programming constructs. HPJava introduces multidimensional arrays, similar to the ones featured in High-Performance Fortran [22], whilst Titanium instantiates the PGAS programming model. Both support distributed and shared memory environments, but, as all array programming languages, they are restricted to the SPMD paradigm. The location of all computation is pre-determined at deployment time and fixed.

Languages X10 [9] and Chapel [14] overcome this limitation by extending the PGAS model with processing unit abstractions (localities) to which code may be shipped and executed. Language constructs allow a) the asynchronous spawning of tasks in localities, b) the distribution of arrays across localities and the subsequent spawning of activities.
to operate over the distributed data, and c) the management of the results of such asynchronous activities.

Data and activity (thread) placement is explicit in the language. This approach is suitable for data parallelism, since a common approach is to distribute the data along the entire partitioned address space. However, when it comes to task parallelism, explicit thread placement limits compositionality, since it forces the API user to be aware if and where the invoked code is placing the computation. This is the result of an over imperative model that requires the programmer to state where everything must be done, instead of a more declarative approach that delegates this work on the runtime system. The work on the porting of X10 to GPUs [12] clearly exposed these limitations. The GPU base programming model is exposed in the language, hence the programmer is aware of the hardware low-level details.

X10 have also experimented with hierarchical parallelism [23] by combining its programming mode with the concepts introduced in Sequoia [11].

As stated in the introduction section, none of these systems cope with node configuration heterogeneity.

V. CONCLUSIONS AND FUTURE WORK

This paper presents an adaptable middleware for Java HPC in heterogeneous environments. An annotation-based programming model allows for the declarative identification of code suitable for task and data parallelism. This modus operandi makes way for delegating heterogeneity on the compiler and runtime system. We are currently focusing on the role of the latter, proposing a modular runtime system that builds on two major building blocks: hierarchical parallelism, both at task and data program decomposition level, and a modular, also hierarchical, adaptation layer.

These allow for the system as a whole, or its particular instances, to adapt to the target hardware or to the application’s characteristics and requirements. An example of the former is the addition of adapters for GPU offloading or for a particular networking technology, and an example of the latter is to adapt a master node (in a master-slave software architecture) subjected to intensive communication on the role of the latter, proposing a modular runtime system that builds on two major building blocks: hierarchical parallelism, both at task and data program decomposition level, and a modular, also hierarchical, adaptation layer.

These allow for the system as a whole, or its particular instances, to adapt to the target hardware or to the application’s characteristics and requirements. An example of the former is the addition of adapters for GPU offloading or for a particular networking technology, and an example of the latter is to adapt a master node (in a master-slave software architecture) subjected to intensive communication by deducing more resources for the handling of such task.

The reported experiments deployed the middleware in an heterogeneous cluster of multi-cores. The work-load was distributed according to the characteristics of the nodes to provide good initial performance results. We are currently porting the JavaGrande and NAS benchmarks to have a more exhaustive benchmark suite.

Future work will focus on the inclusion of GPU support, for which we are envisioning the use of the Rootbeer framework [24]. We are also working on the optimization of the communication adapter, namely the Java serialization overhead, and on the TaskExecutor, to better suite the dynamism of some task parallel applications. Finally, we are working on the ability of the runtime system (backed up by the compiler) to choose between a set of adapters on-the-fly, empowering an extra degree of dynamism.

ACKNOWLEDGEMENT

This work was partially funded by FCT MCTES under PEst-OE/EEI/UI0527/2011, Centro de Informática e Tecnologias da Informação (CITI/FCT/UNL) - 2011-2012.

REFERENCES


