ALiCE

Non-Java Applications Support

Octavian Purdila

February-August 2002

Computer System Laboratory,
School of Computing,
National University of Singapore
Advisor: A/P Teo Yong Meng

Computer Science,
Automatic Control and Computers Faculty,
Politehnica University of Bucharest
Advisor: Prof. Dr. Ing. Nicolae Tapus

Contents

Abstract

ALiCE is a middleware that supports developing of generic grid application development and deployment. It is build in Java using the Sun Microsystems JavaSpaces technology and it is designed with platform independence, scalability, modularity, performance and ease of use in mind.

The need of non-Java, native applications support in ALiCE came from different requirements that users would expect from such a system. One such requirement is maximum performance, as the majority of the applications that will use ALiCE would be computation intensive. It is a fact that because the Java platform uses an *interpreter* it's rather slow than most other languages which are *executed*. Another requirement is that existing applications should be easy to port to ALiCE. A lot of application are developed in either C or C++, and porting these to Java it's not trivial. Also, there are a lot of debugging, profiling, and development tools for languages which are more older than Java.

This document starts with an overview in the grid computing area, continues with an overview of the ALiCE system and then presents the design and features of the non-Java support in ALiCE. Next, two applications developed to test the non-Java support in ALiCE are presented. We continue by presenting the results of various tests performed and finally, conclusions and consideration for future work.

1. Introduction

The goal of ALiCE - Adaptive and scaLable internet-based Computing Engine - is to harness the use of idle cycles for a large number of computers (desktops, mid and high end servers, computer clusters) connected together through Intranet/Internet. The ALiCE's goal place it into the GRID computing area.

GRID computing consists of those "...technologies that enable us to access computing power and resources with the ease similar to electrical power. The computational power grid is analogous to electric power grid. Grid computing allows to couple geographically distributed resources and offers consistent and inexpensive access to resources irrespective of their physical location or access point. It enables sharing, selection, and aggregation of a wide variety of geographically distributed computational resources (such as supercomputers, compute clusters, storage systems, data sources, instruments, people). Thus allowing them to be used a single, unified resource for solving large-scale compute and data intensive computing applications" [20]. GRID computing is trying to do what Internet has done to networking: creating an uniform, easy to access, interoperable technology that will enhance user's life.

The field of GRID computing is quite recent, although both parallel and distributed comuting, areas that GRID computing is related, have a long life already. However, GRID systems tend to be more general and featureful and thus much more complex. These factors explains both the lack of a well structured approach when implementing GRID systems and the progress that this new technology is seeing today. Many consider the GRID as the next technology shift [6], with important implication in the economic and social life [7].

One of the most important requirements for a GRID system is the ability to be able to interconnect a large number of heteregenous systems. In order to achieve this, we placed at the core of ALiCE, our GRID system, the Java platform - a technology that is inherently platform independent. Which allows us to handle heteregenous systems simple and in an elegant manner. However, we wanted to provide more: more performance, more languages from which the programmer to choose when developing ALiCE applications.

This document concentrate on the non-Java support in ALiCE, from the user and programmer point of view but as well presenting the internals. The design and implementation concentrates on supporting native applications (for C and C++ languages, but any other language that produces binary native applications should be easy to add in) for reasons presented further in Section 3.2. However, the ALiCE's architecture permits adding non-native languages, like scripting and interpreted languges.

1.1. GRID systems

1.1.1. Legion

Legion [17], from the University of Virginia, is an object-based grid computing middleware that provides a single address space (distributed shared memory) for all nodes to use as a medium for exchanging objects. Legion is totally distributed. Hence, the system has no centralized node of any kind that function as a manager or a coordinator of the shared memory. Objects are sent from one physical address space to another via message passing.

It is designed for wide area networks and it supports a variety of programming languages, such as s C++, Fortran, Mentat, Parallel Virtual Machine (PVM) and Message Passing Interface (MPI). Though it is claimed to be scalable it is platform dependent on the UNIX platform.

1.1.2. GLOBE

GLOBE [16] is a Java-based grid computing middleware developed at Vrije University, Netherlands. Like Legion, it provides a single address space for nodes to use as a distributed shared memory. Unlike Legion, however, GLOBE nodes keep a local object that functions as a representative of the remote object in its local physical memory. The GLOBE system does not only allow sharing of computational alone, but it allows the sharing of any resource.

1.1.3. Globus

The Globus [15, 5] project at Argonne National Lab focuses on building a toolkit, a middleware, for building grid computing systems. It provides several basic lower- level services that simplify the design of higher level services that serves as a meta computer. These services include naming services, security, and resource management.

Globus is the most well-known grid middleware available today. Compared to Globus, even though is much more lighter, ALiCE has some definitely advantages, like platform independence, ease of use and administration, better control over resources. ALiCE also targets more the home user than the large systems that Globus targets.

1.1.4. Condor

Condor [18], developed at University of Wisconsin, is a grid computing system used harness the idle cycles of computers residing in an Intranet. Condor provides a set of libraries that a C program can link to. Through this library, the program have access to check pointing and remote system call mechanisms. Condor supports job migration and quality of service specifications by allowing the users to specify a list of preferences and requirements. Requirements specify the minimum resource needed to execute the job whereas preferences specify the ideal amount of resources that the job would want to run on. Despite the tremendous advantage that a Condor system can provide, it is limited to NT and UNIX platforms only.

1.1.5. SETI@home

SETI@home [19], is part of the SETI (Search for ExtraTerestrial Intelligence) program that tries to find intelligent patterns in the radio waves received from spaces. It is an application that runs a screen saver on the machines of anyone who is willing to offer the idle cycles of his/her computer to process those signals

1.1.6. Distributed.Net

Distributed.Net [14] is a project that tries to develop specific distributed applications for key-cracking that are working on the same principle as SETI@home. The project had a lot of success, successfully completing the crack of 56-bits DES key, which led to the conclusion that a 56-bits key is too little security.

2. Overview of the ALiCE system

The ALiCE system [1, 2] must support three basic functions: allowing users to submit the applications that they wish to run, allowing users to contribute the computational power of their machines to the ALiCE system and resource management - matching resource demand with available resources.

Basically, we are using a three-tier architecture, consisting of three elements:

the consumer that submits new applications into the system

the resource broker that does resource management and scheduling with a centralized view of the system

the producers that are executing code supplied by the applications

There is an additional component used for data files, *the data server*. This can be the same machine as the resource broker and it's present in the system so that application can have access to an uniform, unique name-space for file storage and access.

2.1. System Components

All the components of the ALiCE system are presented in Figure 2.1 on page 10 and the functionality of each of them is explained in the following sections.

The connection between the components is done through the network, either a Local Area Network if the work environment is a cluster, or the Internet/Intranet if the system is deployed over wide area connections. All the communication is done through JavaSpace and the means of communication will be further detailed later on.

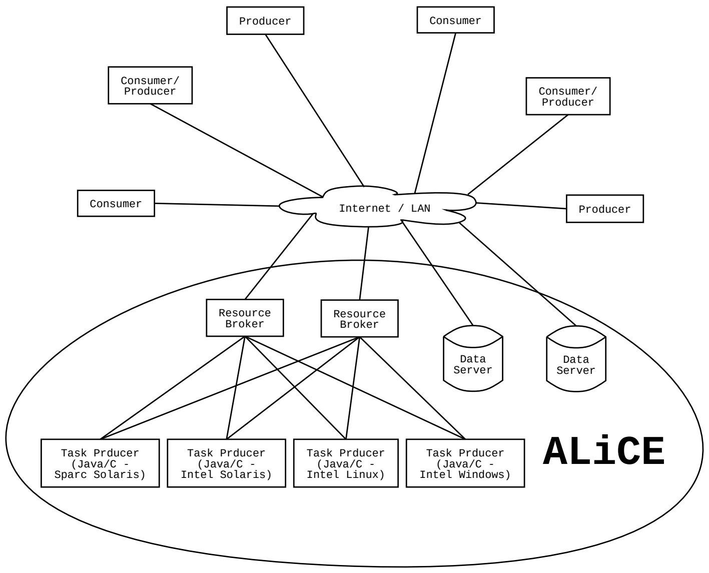

Producer
Consumer
Consumer/ Producer
Consumer/ Producer
Consumer
Internet / LAN
Producer
Resource Broker
Resource Broker
Data Server
Data Server
Task Prducer (Java/C - Sparc Solaris)
Task Prducer (Java/C - Intel Solaris)
Task Prducer (Java/C - Intel Linux)
Task Prducer (Java/C - Intel Windows)
ALiCE

Figure 2.1.: ALiCE components

2.1.1. The Consumer

The consumer is the one submitting the applications to the system. It can be any machine that is connected to the ALiCE system through a LAN or through the Internet and that runs the ALiCE consumer/producer components and GUI (in the end, any machine connected to the Internet can use the ALiCE GRID system). This means that the user will use a GUI to submit a file containing the ALiCE application in a specific form the language that is written in. For non-Java support, this file is a .tar.gz archive which should contain at least one task generator class and one result collector class (see Section 3.1 for more details about the programming model). The task generator is transported inside the ALiCE system in order for tasks to be generated, initialized and sent to be producer. The result collector is executed at the consumer at it receives the results generated by the tasks created by the task generator.

The consumer is also the point from which new protocols and new runtime supports can be added to the whole system. Although the new protocols plug-in support was tested and it works, some tuning is still badly needed, as well as an unified support for using the new protocols and selecting from them for security purposes. The plug-in support for new languages support is not fully deployed as yet.

2.1.2. The Producer

The producer is a machine that has volunteered its idle cycles to run ALiCE applications. The producer will receive tasks from the ALiCE system in the form of .tar.gz archive, will extract objects from the archive and in the end will dynamically load and execute them. The results obtained from each task will be sent so they can be received by the consumer that has originally submitted the application.

The producer and the consumer can actually be the same machine, this being the most usual case, when someone who volunteers to run application from others also wants to run his/hers own applications in ALiCE. In order to support this, the GUI for the system is unified for the producer and for the consumer.

2.1.3. The Resource Broker

The resource broker is the central point of the system. Basically, the only thing it does is scheduling. The scheduling is needed for many reasons. The first is to have a control over resource allocation and usage, since we are dealing with multiple concurrent applications at the same time in the system.

Then there are the objective needs imposed by supporting other languages than Java. In contrast with the portability and platform-Independence of the Java programming language, other languages are platform-dependent and even library-depended. The scheduler should than choose an appropriate platform for the producer that should run that application. There are two types of scheduling done at the resource broker's site: application scheduling and task scheduling.

Even though there are many approaches to scheduling, we choose to have a centralized scheduler, with options to do part of the scheduling distributed by means of pattern-matching when retrieving objects from JavaSpace.

2.1.4. The Task Producer

The task producer is a machine that is part of the ALiCE core (but *needs* not to be so - it can be outside) and it is meant to run the task generator classes of the applications. This will generate tasks, which will be scheduled by the resource broker and than downloaded by the producers directly from the task producer. The separation of this machine from the resource broker (in the previous version, they were running on the same machine) was done for two principal reasons:

- since we are supporting non-Java applications, those applications are platform-dependent and not all of them can be run at the resource broker;

- in order to separate and isolate the central point of ALiCE, the resource broker, from any alien code. Since the task producer runs code submitted by consumers, we don't have total control over what that code does. Even with strongly enforced security and code safety measures, we can't guarantee total security. So the decision was made to run the code on another machine and in this way to achieve *total* safety of the resource broker.

Each task producer is running either Java code (which can be run on any platform), either code compiled for the platform that the task producer offers.

2.1.5. The Data Server

The data server is a machine dedicated for data file storage. Any data file used by an application can be submitted to the data server. From inside any task, the programmer can obtain access to a data file submitted for the application that has generated the task. Through the reference obtained, the task can read or write chunks from that file of any size, from 1 to the size of the whole file.

2.2. Communication between components

In ALiCE, all communication is initiated and supported through JavaSpace. This means that there will be no communication between parts of the system that do not leave a trace in the space. In future development, this will help with implementing a central accounting and monitoring scheme that will be able to register all the communications that take place in the system.

JavaSpace is also used as the mean to synchronize between the components of the system. Since the API of JavaSpace place at our disposal a set of blocking calls, we can use those calls to synchronize different threads running on different machines in the ALiCE grid computing system.

All the objects transfered through JavaSpace in ALiCE fall in one of two categories: file references or messages. Each defines a class of communications. One implies the transfer of application's objects through the network, thus supporting mobile code in our grid system. This is further detailed in Section 2.3. The other one is just for transferring information between components of the system, to advertise capabilities and resources, to synchronize parts of ALiCE and for any other communication purposes.

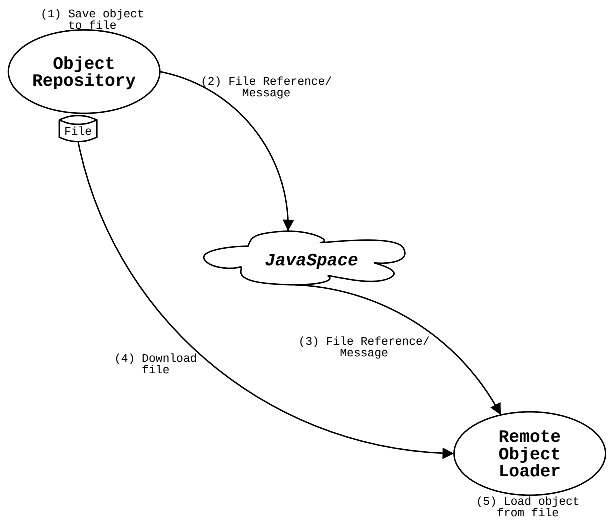


Figure 2.2.: Object Network Transport Architecture

2.3. Object Network Transport Architecture

Object Network Transport Architecture (for short ONTA) is an infrastructure used to freely move live objects over the network, with a great accent on scalability and ability to sustain high workloads in terms of number, size and diversity of objects. Basically, any object that is persistent[1]can be send over the network using ONTA.

The architecture uses a hybrid model between the centralized and distributed models. We choose to have a central point in the system - JavaSpace [2]- so that we can track transfers and maintain security. We use the peer-to-peer model to actually transfer files between the nodes of the system in order to achieve scalability.

ONTA clients and servers use *object/file references* to advertise files/initiate transfers. The references contain information about how to retrieve a particular object needed by an application (the protocol to use, the host location as well as the name of the object), but also other kind of information like authentication information, level of security and information specific to a runtime support. An important thing to mention is that the size of these object references is very small compared to the actual object itself, and thus even if we have a semi-centralized point in the system it will not become a bottleneck.

[1]for Java this means to create a class that implements the Serializable interface; for non-Java a special object format is used, described in Section 3.2

[2]the central point being JavaSpace is not necessarily a single machine; however from a logical point of view, we have a central point

3. Non-Java application support in ALiCE

3.1. Programming model

In order to understand how non-Java support to ALiCE was design it is important to present the programming model that can be used to write applications for ALiCE. An ALiCE application is a distributed application, having multiple tasks running on it's behalf. A tasks is an object that contains both code and data and which has a standard interface that the system can use it to start the tasks run on the system's nodes. The task is the basic unit of the computation model used in ALiCE from the point of view of the parallelism of the application. Tasks can be run on any node of the system. Tasks can be created from within tasks, thus being able to create task dynamically as the application needs them to scale to the size of the problem.

An application has two special tasks: the entry point task and the exit point task. These two special tasks are called in the ALiCE programming model the *Task Generator,* respectively the *Result Collector*, due to historic considerations[1]. While tasks of an application other than the result collector can be executed on any node of the system, the result collector itself will be executed at the client machine since the result collector is the interface between the user and the ALiCE system. However, there is no need for an application to have two separate tasks one for the task generator and one for the result collector. They can both integrated into one task, which of course will be run at the client machine.

Besides tasks, the computation model allows also raw objects (which can also contain both code and data) to be used when developing an application. Such objects are often used to let tasks communicate with each others. Almost every application will use raw objects to

[1] in the first version of ALiCE [2] only the task generator could create tasks, and there was no way a task could exchange information with another, they were just do their work and returned a result to the result collector

send results from tasks to the result collector. As you probably notice the task is a superset of an (raw) object. What is important to stress is that each object (be it task or raw object) contains code and data that is private to that object.

So far we presented the components of the computation model but we still not stated how this components will interact with each other. What operations is the user allowed to do with them. We choose a few such operations in order to keep the model simple. There are just three operations: *new, put, get* . Simple yet powerful. Using these operations you can implement anything. You can apply this operations on any object be it a raw object of a task, from within any object. As one can observe we combined two of the most used models: the object oriented model [12] and the fork/exec model [11].

Even if the meanings of the operations are obvious let's do a quick presentation of them. *new* allows one to create a new object of a specified kind. It can be either a task or a raw object. Creating a new task seems to be equivalent to the creation of a new thread in aoperation multi-thread based systems. However it is not. This is because when you create a new thread the newly created thread shares the memory with all other threads. This is not true in our case. It looks more like a fork and then exec process as the data and code of the newly created object is not shared, is private to that object. There is also a *clone* operation you can perform, which is similar of course to just fork.

The *put* and *get* operations are used send / receive objects to / from another task. The operations are similar with equivalent operations used in other distriuted systems (like the send / receive in MPI and write / take in JavaSpace). However, even if the syntax is similar the semantics is different. Remeber that an object can contain both code and data, so receiving an object A means that a shared object will be loaded in the address space of the object B that received the object A. Even more, because the object is persistent it means the the object A could've been running on some other nodes of the system gathering some data which could be now be made available to the current node.

3.2. Object format

One of the important design decision for non-java application support in ALiCE was in what format the applications will be distributed to the system's nodes. As resulted from the programming model the application is actually a sum of objects interacting. Thus by defining the representation of the programming model object in the real world, we actually choose in what format the applications will be distributed.

To quickly get to the point, some alternatives are: source, native binary, virtual machine binary. Each of these one have many other possible “sub” alternatives. However first of all we had to choose between these three alternatives. For the one which doesn’t want to go through the (probably silly) arguments: we used the native binary approach. You can safely skip through the next two paragraphs.

Using a source approach would made the system unusable for most of the applications, as even if the open source movement has gain momentum the majority of the application are still not open source. Another argument against the source approach was the fact that this will increase the demands on the producers nodes, as the producers would need to have deployed development tools. It would also increase the overhead, as the source would need first to be compiled. So the source approach goes away.

The virtual machine approach was also rejected. The arguments against it were performance related, and the fact that there are already some virtual machines (java virtual machine, p-code virtual machine, .net virtual machine), so if we wanted a virtual machine approach we should probably use an existing one. But the fact is that we already using a virtual machine (the Java virtual machine) in the ALiCE system, so we have checked that. What we wanted was a different approach. Another argument in not using (only) the virtual machine approach is that in the real Internet/Intranet world the number of platforms are not so diverse (well, until we will have embedded devices on-line, like the intelligent fridge ordering another bottle of milk to the grocery store; and even if that happens, we think that those devices would not be the right target for node of a GRID system). We have a sufficient number of similar platforms not to worry about the fact that a binary approach would not have enough nodes of a kind.

So the choice after all was the binary approach. However the quest is not over. Even if we choose a native binary as the form in which the application/parts of the application would flow through the system there are still decision to be made. Modern operating systems have many forms of binaries: static binaries, static libraries, static shared libraries, dynamic shared libraries [3]. Each of these choices have many advantages/disadvantages, but the more important one is that while some are fast but inflexible, others are more flexible but slow.

The choice was dynamic library. It was basically the only option if we were to respect the programming model. The problem is that we stated that any tasks can receive objects from any other tasks (and at any time of it’s lifetime). So after the object gets transported to the destination it will need to be loaded in the process that runs the tasks which is doing a get operation. And since the object contains also code and not only data, the only choice here is to use dynamic libraries.

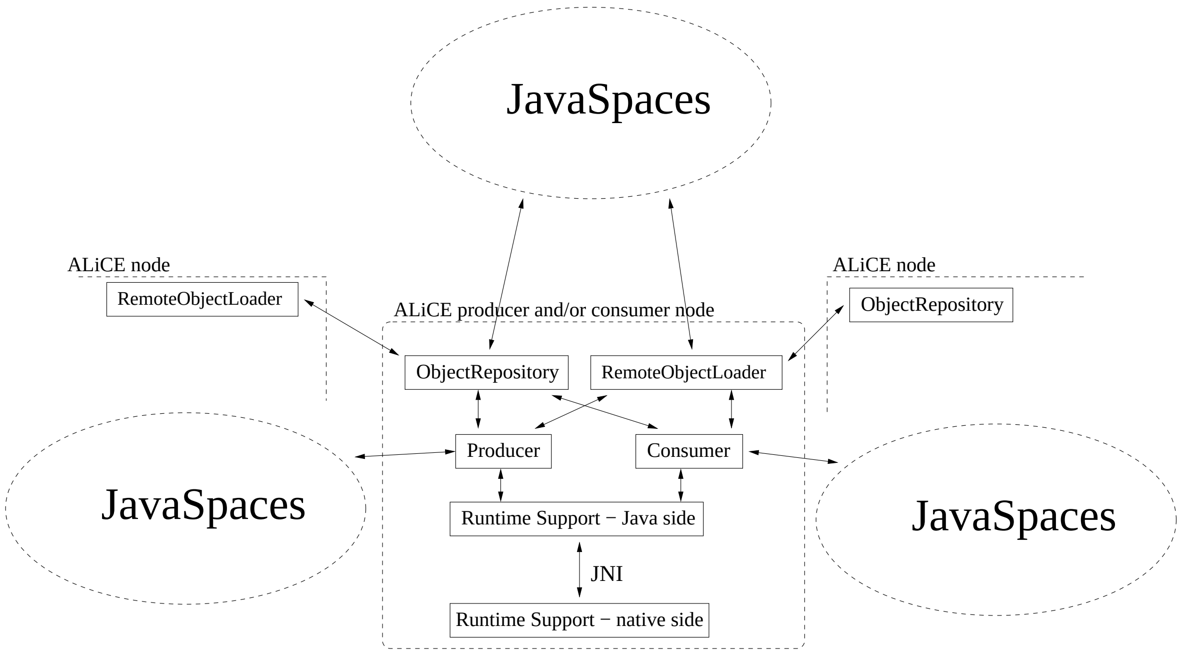


Figure 3.1.: A non-Java applications enabled ALiCE node

3.3. Architecture overview

Non-java support in ALiCE does not change much the architecture of the system. There will still be a resource broker which will coordinate the whole system, consumers which will deliver applications to the system and producers which will run these applications. However some of these components will need to be enhanced.

The first basic need when supporting non-java application in ALiCE is the necessity of a component in the ALiCE system that will actually execute machine specific tasks: the runtime support. Due to the fact that the core of ALiCE is using the Java platform, the runtime support is having two sides: the Java side and the native side. The native side will provide access to the node's resources like CPU, disk, memory by executing non-java applications on the node. The Java side will provide communication with the ALiCE system, bringing applications to be executed, sending results back, sending and receiving system messages.

Both sides of the runtime support will have to run on the producer node and will be able to run applications on a specific platform as defined by the operating system and processor. In order to allow the system to be extensible and auto-upgradeable the system has the capacity of registering new types of runtime support as they are introduced into the system and if needed download them to the producer node.

In order to accomplish it's tasks the native side of the runtime support is using some specific mechanism that will be presented in depth in the following sections. As we've earlier said, one of the most important things that the native runtime support has to do is to execute the application, or more exact to execute tasks of the application. For reasons that we presented earlier tasks will be shipped to the native runtime support in form of a binary, actually a dynamic library. Most modern operating cand handle dynamically loading of such libraries. But there are some other issues in developing a grid system, that make the dynamically loaded libraries insuficient for our needs.

Let's try to see what happens in the system when we run a simple application. Lets say that we just completed our shining brand new matrix multiplication application for ALiCE which computes matrix C as the result of the multiplication between matrix A and B (all of the matrices have N^2 elements). And it looks something like this. We have a tasks generator and result collector collapsed together into one single task. Then we have one more task that actually will perform the multiplication. Lets say that this tasks will compute one single result from the C matrix. So by creating N^2 tasks we will solve our problem (yes I know, it's not the state of the art matrix multiplication algorithm but it is a good/simple example). But no. Because in order to compute the result the task will need one row from matrix A and one column from matrix B; and since we are smart people we did not hard-coded the matrices in the application but let the user specify their elements. Here we have two options: we either create objects that contains the required line and column from the matrices from the TG&RC task and we send them to the computational tasks, either we initialize the computational tasks with the required information before distributing them to the nodes. We argue that the second approach is much better. Think about it: it's faster since the initialization is done locally, on the same machine, it's more natural to the programmer, it's less bandwidth consuming. And as we will see later, this approach will also make possible to develop a system were tasks can be preempted, which will solve some important problems that a grid system faces.

But if we want to use the second approach were facing persistence problems. The binary formats, compilers, linkers and loaders were not design to run a program that initializes some variable to some values, then stop the execution and save the variables that are supposed to be saved in the binary format so that it can be loaded later, possible on another machine. And this is exactly what we need to do. To make things simpler and more portable the implementation is divided in two parts: the system part and the language part. Thus we developed the persistent object which corresponds to the system part and the ALiCE object which corresponds to the language part. The persistent object deals with binary formats and loaders. The ALiCE object deals with compilers and linkers by creating an interface that can be use from a programming language by the user. Both of these will be covered in depth in the following pages.

3.4. The Persistent Object

At some producer node the application is executed and generates and initialize tasks that will be scheduled to producers nodes. This architecture makes use of the fact that the tasks once initialized at some node are able to remain in that state until they reach producers nodes. Such a behavior can be achieved by the use of persistent objects. Persistent objects are also used to transport results. We propose the persistent object in order to allow native code objects to run among different nodes of a system, allowing them to freeze their state while they are transported between nodes.

3.4.1. Overview

As mentioned before, the applications and tasks will be transported between nodes in a dynamic library format. The shared object will contain the code, initialized data, references to libraries, etc. in such a way that it will be able for the runtime-support to load this object in memory and execute it's code. So far this description resemble the dynamic libraries that are implemented on most Unixes and also on the Windows platform (referred as DLL[2]s). We extended these dynamic libraries so that they became persistent objects.

To make them persistent objects means that we have to find a way to save the state of the object's code. This can be achieved by saving the variables, which resume to saving the object's data. As stated by most ABI[3]s of operating systems [4, 3], the object data's can be of three[4] types: initialized data, uninitialized data and dynamic allocated data. For each type of data there is one section in the object that contains it, except for dynamic data. We have the *.bss* section for uninitialized data and the *.data* section for the initialized data. The dynamic allocated data is contained in segments allocated as the process runs therefore we don't need a section to hold this kind of data. The *.bss* section does not contain any data, but just specify the total size of the uninitialized data.

In order to make the object persistent we have two possible approaches: either save

[2]dynamic loaded library
[3]application binary interface
[4]we exclude stack which holds function local variables; Section 6.2 includes some consideration about stack and local variables

the object's normal data, either create a new kind of data (persistent data) which will be hold separately both in memory and shared object's file. To be more flexible we choose the second approach, so that the environment will provide to the programmer both persistent and non-persistent data.

The persistent object will provide to the user two kinds of persistent data: initialized / uninitialized data and dynamically allocated data. Keeping the first kind of data persistence is straight forward: we keep this data in a non standard section *.pdata*, which is allocated in the shared object. Then, when unloading the persistent object, we save the data from memory to the shared object's file. No loading is required, since the system automatically loads that section in memory. However, this requires that we keep a map between *.pdata* section and where it's data is located (which segment in memory). This can be done with little overhead while we are loading the persistent object using system facilities like memory mappings inquires (e.g. for UNIX systems the /proc/self/maps file). In order to keep the persistent variables in the *.pdata* segment each variable needs special directives.[5]

For dynamically allocated data, it's not anymore so simple. Because that address collision could occur when loading multiple persistent objects[6], a mechanism should be provided to allow data relocation (which is provided by the operating system for static data, but not for dynamically allocated data). To solve these problems we propose a solution in which we use our own memory allocation routines. The allocated memory will then be saved to the shared object's file, to the end of the shared object's data and sections. We use this approach for performance, so that we don't have to move sections around while the heap grows/shrinks.

The pointers used with the allocation routines will actually be virtual pointers. When the user will want to access the area pointed by such a virtual pointer, it will have to use a function to translate it into an usual pointer. By keeping the addresses virtual, we are able to relocate the segments which contain dynamically allocated data.

3.4.2. Persistent static variables

The static persistent variables have to be marked so that the implementation can save them. This is done by using the keyword *persistent* after declaring the variable, but before initializing it. The *persistent* keyword is actually a macro defined in this way:

```
#define persistent __attribute__((section(".pdata")))
```

[5]these directives are compiler dependent; so far we support the gcc compiler
[6]multiple persistent shared object can be load in the same process, thus sharing the process's address space

The effect of this statement [8] is that the variable which has this attribute will be placed in the *.pdata* section, section which has the same attributes as the *.data* section (i.e. the following flags are set for the section: CONTENTS, ALLOC, LOAD, DATA).

3.4.3. Persistent dynamically allocated data.

The persistent object's file contains beside the compiler generated sections, the *.pdata* section a *.heap* section and the heap data saved at the end of the shared object's file. The heap section is created when compiling the source, with the following statement declared in the *pso.h* header file:

```
#ifdef __PSO__
static struct pso_heap _pso_heap \
__attribute__ ((section(".heap")))\
__attribute__ ((unused)) = { 0 , };
#endif
```

Therefore, the .heap section contains informations about the heap. However the heap's data is saved right at the end of the file. We choose to save persistent heap data at the end file, and not in a section because the heap size can vary during the lifetime of the persistent object, and by placing the heap in a section we would've been forced to move sections around when the heap would grow or shrink.

The encoded pointers used are actually offsets from the heap base. Each time the heap moves, the *base_correction* field of the *struct pso_heap* (which is kept in the .heap section) is adjusted. The encoded pointer it is transformed into an usable pointer by adding to its value the value of the *base_correction* field to compensate for the movement of the heap.

The allocation algorithm for the persistent heap was inspired by the GNU glibc's memory allocator [10]. We used the same (well, almost the same) memory layout as in the named allocator, then developed a "good enough" algorithm. The persistent heap is allocated through the *mmap* system call into one contiguous segment. It can be easily grown or shrink as needed by using the *mremap* system call.

The whole heap is divided into variable size chunks which are marked either free or allocated. Each chunk contains at the top two words which help moving from this chunk to the next or the previous chunk. The first holds the size (in bytes) of the previous chunk including the two words. The second holds the size (in bytes) of the next chunk, also including the two words. If the least significant bit of the second word is set than the chunk is allocated. Otherwise the chunk is free. By using the LSB bit we have to align the chunks (and thus users

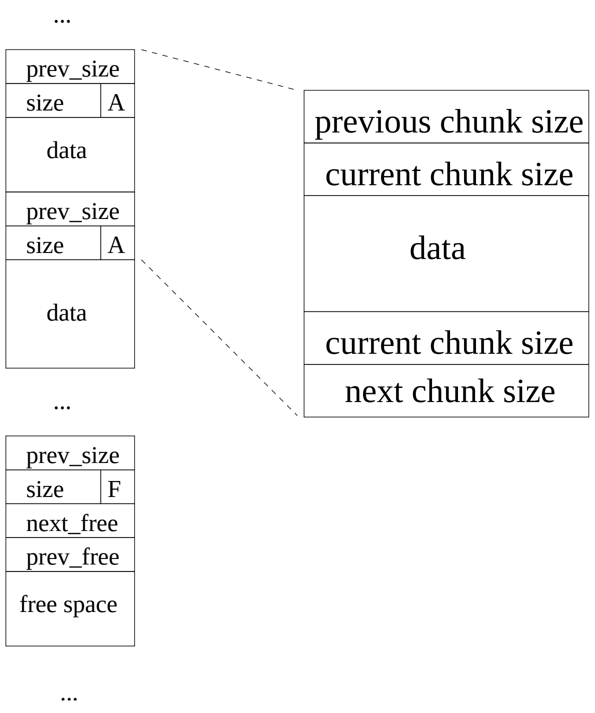


Figure 3.2.: Persistent heap allocator layout

data) to word boundaries. This would anyway be required for some architectures, while for others even if it is not required provides maximum performance.

An allocated chunk has user data following the second word. In a free chunk the first two words of the user data (thus the third and fourth words) are used as pointers to the previous, respectively the next element in the free chunks list.

When the heap is created for the first time (that is either a load time, either at first allocation if at load time there is not enough memory to allocate one memory page) the heap consists of a single free chunk with the size of a page size. When the user allocates memory, some free chunk (more about choosing the chunk in a moment) will be split into two, one marked used and one marked free. As the reader probably figured the free chunks will be kept together in a double linked list. The list will be sorted after the size of the chunks, with the biggest chunk coming first. When the user wants to allocate memory the allocator will take a look at the first (and thus biggest) chunk and see if there is enough space there. If so this chunk will be split in one used and one free chunk, which of course will be moved in the list of free chunks to keep the list sorted. If there is not enough space in the first chunk it means that the heap needs to be grown. In order to be efficient heap grows (or shrinks) in system pages (4KB on i386). The heap can be grown either by resizing the last chunk if it is free, either by creating a new chunk if the last one is used. This kind of allocation algorithm is called “worst fit” [9].

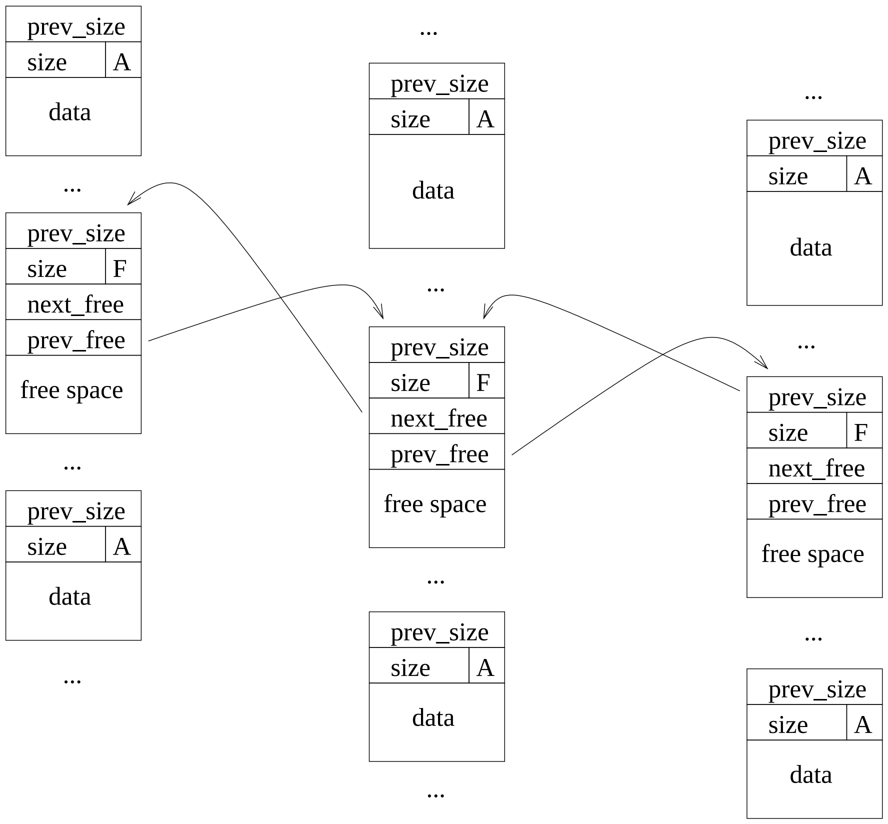


Figure 3.3.: Persisten heap allocator: free chunks list

In order to reduce fragmentation, each times a chunk is freed by the user the allocator tries to collapse adjacent free chunks into one chunk. Also to reduce the size of the heap, if there is space unused at the beginning of the heap segment, the whole segment is logically shifted and thus the free space can be ignored. This works by adjusting the *base_correction* properly. This operation is perform only when the persistent object is unloaded.

3.5. The ALiCE Object

To implement the programming model described we need more than just what we are able to do with the persistent object. Basically one can use the persistent object to do that, but it would be extremely complicated for the user to do that. The ALiCE object is an extension of the persistent object to ease the programmer's job of developing ALiCE applications.

3.5.1. Overview

The basic building block that we used to create an API and runtime support for non java application in ALiCE is the ALiCE object. The ALiCE object has the property of being persistent, as described in the persistent object overview. To manipulate objects, different operations can be performed: creating a new object of a particular kind, cloning an existing object or freeing it. These operations will not interfere with the persistent property of the objects.

The creation of a new object is similar to the process of instantiation a class. We will often use the terms instantiation of an object or creation of a new object intermittently, describing the same process. Cloning an object will also create a new object, but the newly created object will be an exact copy (from the point of view of the object's persistence) of the cloned object.

While the persistent object guarantees that an object is persistent, it makes no guarantees about the persistence of complex structures build from multiple such objects. For example, one object might create instances of other objects and might invoke operations from within those created objects. During it's lifetime the object could be moved around the nodes of the system thus making the whole structure (the object that created instances of other objects plus those instances) inconsistent. However the ALiCE object (which is build on top of the persistent object) guarantees that any structure build from ALiCE objects will be persistent.

In order to move around objects in the ALiCE system, as well as to perform internal system operations, the objects need to be identified in a way in which each object in the system is uniquely identified by it's id. The naming scheme we developed give each object in the system an unique id, even if the objects are created on different nodes of the system. The naming scheme does not use communication between nodes to agree on the names.

3.5.2. Naming scheme

As stated earlier in the overview, objects are named such that each object's name in the system it's unique. To achieve this, each object of an application (run) has to be uniquely identified. The naming scheme we developed allow us to do so, without using an arbiter or using communication between nodes. Each object in the application will have a parent (except the *entry object* and the *pure objects*) which created them. As objects are created they inherit the parent's id and are given by the parent one unique id (by incrementing a persistent counter). Thus naming the object *object name-parent id-id* we achieve our goals, in fact by creating a hierarchy in the objects of an application, based on the parent-child relationship.

The *entry object* of an application is the object that contains the entry point in the application code. This object as well as all *pure objects* have an parent id as well as an id of zero. The *pure objects* are objects that were never instantiated and will never be. They are used to create instances of objects, when the application calls the *alice_new_object* function. The actual instantiation act is done by copying the pure object's file to another file. As stated earlier one object will be saved in the form of a dynamically loaded library, and to keep things simpler we choose to name the file the same as the object.

3.5.3. Object dependencies

The ALiCE object layer is introduced to ensure the persistence of complex structure formed by multiple ALiCE objects. To do that, the library keeps track of the objects that one object uses and stores (in the object) references to them. Such work is often called as maintaining dependencies. Dependencies are important in ALiCE because as one object might be moved from one node to another in the ALiCE system, if it uses other objects, these objects will have to be moved along with that object.

This why we maintain a list of dependencies for each object. The list actually contains the names of the objects (naming was covered in the previous section). Dependencies can be of two types: dynamic and static. The static dependencies are those dependencies that are known right after the code was compiled, before execution. Static dependencies are the result of calling the *alice_new_object* function, and creates a dependency to a pure object. The pure object will be needed on the node that the dependent object will be executed (which at some point might call the *alice_new_object* function), so that a new object of that type could be created.

To find out static dependencies right after the compilation phase, the *alice_new_object* is in fact a macro which beside other things contains a

```
({\
        static const char* alice_static_deps \
        __attribute__ ((unused)) \
        __attribute__ ((sec-
tion(".alice.dep"))) = #name"-0-0";\
})
```

line (yes, it is a single line; from the point of view of the preprocessor). The effect is that a non-global variable is created containing a reference to the name of the object on which this object depends, which is placed in the *.alice.dep* section. The section's beginning is marked by a global variable (*alice_static_deps*), accessible at run-time, and a *NULL* reference at the end of the section. This is one of the reasons behind the rule that the application code has to be contained between the *ALICE_OBJECT* and *ALICE_OBJECT_END* macros. Thus at run-time we have a compact block of static (direct) dependencies, easily accessible.

However, the described mechanism is only half the work done when resolving dependencies. This is because it finds out only direct dependencies. For performance reasons we would like to know all dependencies that are needed for an object to run as soon as possible, be they direct or indirect (objects needed by the objects in the dependency list). That's why after the object is compiled static dependencies have to be resolved for it. This is currently done when the user uses the *alice-pack* utility that takes the entry object as an argument and construct an archive with the needed object files, each object having it's static dependencies resolved. Resolving static dependencies results in creating a dynamically allocated persistent list (since the process of gathering static dependencies is done after compile time – but before run time) containing the static dependencies.

The dynamic dependencies are created at run-time, when calling the *alice_new_object* or *alice_clone_object*. These routines returns a pointer to a *alice_od* structure representing the newly created ALiCE object. From now on, the programmer is expecting to access these objects with the functions from the API, and eventually data/code from this object. But this object might just be moved to another node. And together with it, the system should also move the objects that the programmer might access. Again, the need to maintain dependencies, this time dynamic ones as they are created at run time. As object are created, they can be free with the *alice_free_object* function call. Thus the need to remove dynamic dependencies. As the dynamic dependencies list has to be available on the entire lifetime of the object, it will be saved as a dynamically allocated persistent list.

3.5.4. Platform requirements

Because the ALiCE system will be compose from heterogeneous nodes in terms of hardware architectures, operating systems and languages we had to implement some basic way of specifying what the application will need so that it can run on a node. This is not necessary for Java, since it's motto is "write once, run everywhere", but for non-Java support is a necessity.

The programmer can specify some of the platform requirements when creating the ALiCE object:

os a string identifying the operating system ("linux", "solaris", etc.)

os_features a mask of features that the operating system must have so that the object can run; the features will be defined as macros in the API, but so far no such feature are available

os_version a pair {major version, minor version} for the operating system; in order to run an object on a node, the operating system on the node should be at least the one present in the platform requirements for the object;

proc a string identifying the processor class ("i386", "sparc", "ppc", etc.)

proc_features a mask of features that the processor must have (like mmx, hardware float point, etc.); so far no features are defined

There are also some other platform requirements that are set automatically:

- the runtime support version
- the libraries that this object needs in order to be executed

3.6. The runtime support

As previously said, the runtime support it's the core of non-java support for ALiCE. Because the ALiCE system at it's core it's Java based on one hand, and because we are supporting native binaries on the other hand, the runtime support has a dual nature.

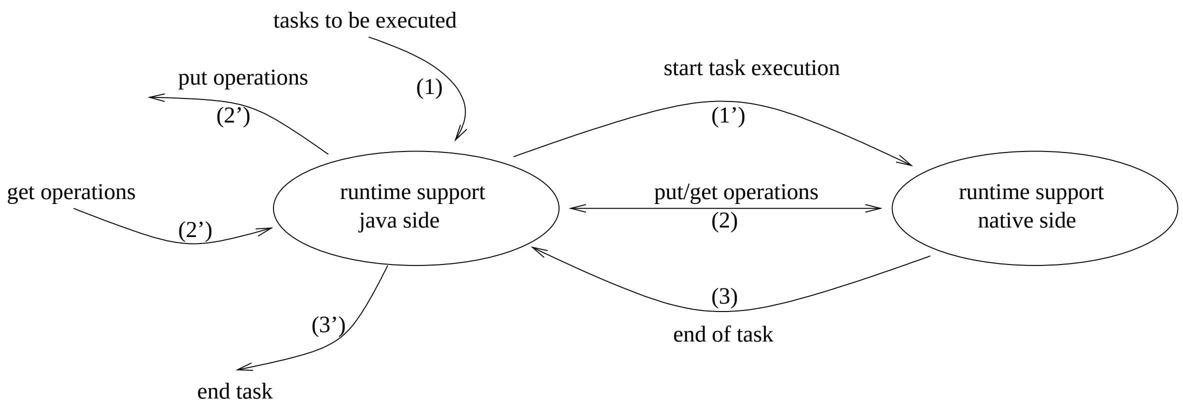


Figure 3.4.: Interactions between the two sides of the runtime support

3.6.1. The native side of the runtime support

The native side of the runtime support is implemented as a shared library which gathers functions used by the ALiCE objects and as well by the Java side of the runtime support to perform certain operations related to the execution of a task.

After loading the library the Java side can directly call functions from this library by using the Java Native Interface. Beside the entry points for calling from the Java side, the library contains the implementation of the functions described in the API which can be called from within one object.

The implementation is thread safe, meaning that two or more threads can arbitrarily call functions from the library. Even more, the library does not use mutexes, as the library does not use global or static variables, so there is no bottle neck in synchronization points. However, the library uses thread specific data to keep pointers to the Java environment and the execution directory. For this purpose we uses pthread (POSIX threads) library. We choose this library because of it's wide portability range.

At a glance, the native side's jobs are:

- packing/unpacking packages
- executing tasks
- talking with the Java side on behalf of the applications (for putting/getting objects to/from the system)

Although the packing/unpacking routines could be implemented in the Java side, we choose to implement them in the native side for performance reasons.

To execute multiple tasks on the system without interfering, each task has it's own directory were the objects/tasks are brought for execution, new objects/tasks are created and packed to send to another node of the system.

Each time an application puts an object into the application space, the system has to perform the following tasks before being able to deliver it:

- lookup static dependencies
- lookup dynamic dependencies
- processing platform requirements and create a common platform requirement valid for all depended objects
- pack the depended files

3.6.2. The java side of the runtime support

The Java side of the runtime support it's mostly acting as an interface between the ALiCE system and the native side of the runtime support, the part that does all the hard work. However there is still some work to do for the Java side too.

One of the more important functions of this side is to check object references to see if the platform requirements are met. Another important function is to keep the mappings between the ALiCE tasks and the native processes and threads that executes the ALiCE's tasks.

3.6.3. Security

There are a lot of security issues when it comes to GRID systems. Some of them are dealing with authentication and communication encryption. Some other are dealing with result checking (how do we know that a result send is not send by a malicious user?). However these are system global security issues, not related to the subject of this document and we will not discuss them here. Instead we will discuss the local security issues. The ones related to the security of the producer nodes, because the code we execute at these nodes is arbitrary. The question is: how do we protect the producer nodes against malicious code? Even more, how do we protect tasks from each other[7]?In order to protect the producer node we first need to understand what resources is the producer given to the alien code. These resource are actually what a program needs in order to be able to be executed:

[7] yes, usually the producer nodes will execute only one task at a time, but if the producer node is multiprocessor we might want to take that advantage and run more tasks in parallel

- memory
- disk access
- CPU time
- network access

These are the obvious ones. There are still a few that are more obscure:

- the number of processes/threads that can be created
- the number of files that can be created/accessed

So by looking at the listed items, we know what we need to protect. Protecting here means that we should control how much of the resources should the alien code have access. Otherwise, if we don't limit the access to resources, the alien code could make the node unusable. Let's say for example that we don't care about how much disk space that the alien code is allowed to use. In this case, one attack could easily be made by creating huge files on the node. Eventually the disk space would be consumed, and important operations could not be performed by the operating system (rotating logs, running certain daemons/services, etc.).

Besides protecting the nodes resources, we need also to protect the other users of the system so that the alien code could not interfere with the jobs that these users are running here. We can group the type of objects belonging to other users of the system in multiple classes like:

- processes/threads
- files

To give an example of what might happen if we have no such protection imagine that the alien code could read files belonging to users of the system. If it could read them, the alien code would easily be able to send them away. Not something that you want to happen to you.

By looking at what modern operating systems offer when it comes to protection we could easily say that to acquire the desired protection, we would need not more than to run the alien code under a special non-privileged user and use limitations of the resources with the means of the operating system (for UNIX systems we could use the *getrlimit()* system call).

There still remains the problem of how to protect multiple tasks running concurrently from each others. This means that we have to protect the processes/threads and files of these tasks. However an implementation which uses threads to run multiple tasks allows one task to access the memory of another task. The correct solution is to isolate the tasks in separate processes. Another problem when trying to protect tasks is to allow access to files only for the owner task. But we can't do this with usually restrictioqns such as file permissions as the the tasks are running under the same user (or if were to create multiple users, how many of these users should we create?). The solution here is to use operating system features in order to limit files access to a per process basis. A possible solution is to use the *chroot()* system call present on most of the modern UNIX-es.

4. Applications

In order to test the current implementation of the non-Java support for ALiCE, several applications were developed and their performance was studied in order to identify flaws in the implementations. The applications we present here are older Java ALiCE applications. We choose them in order to compare the performance of the non-Java and Java support.

4.1. Ray-tracer

This application has been ported from it's initial Java implementation to a C++ non parallel implementation. Then the C++ application was easily wrapped with a thin layer of code in order to be integrated into the ALiCE system. The ALiCE adaptation consists of implementing a render tasks, an ALiCE task that will be run at producers nodes, and a collapsed task containing both the task generator and the result collector that will run at the consumer node.

The application is rendering (ray tracing) a scene. The application was originally developed[1] by Sun Microsystems, and later adapted to run on ALiCE. Porting the application from Java to C++ introduced the problem of what GUI to use. We used the Qt[2] libraries to display results. Also, to allow real-time update of the GUI, the application has two threads[3]: one thread that is concern with running the GUI related stuff, the other one is used to receive results from the producers. This was a good test for our implementation, as we used a lot libraries, interaction with X11 and GUI and multi-threading.

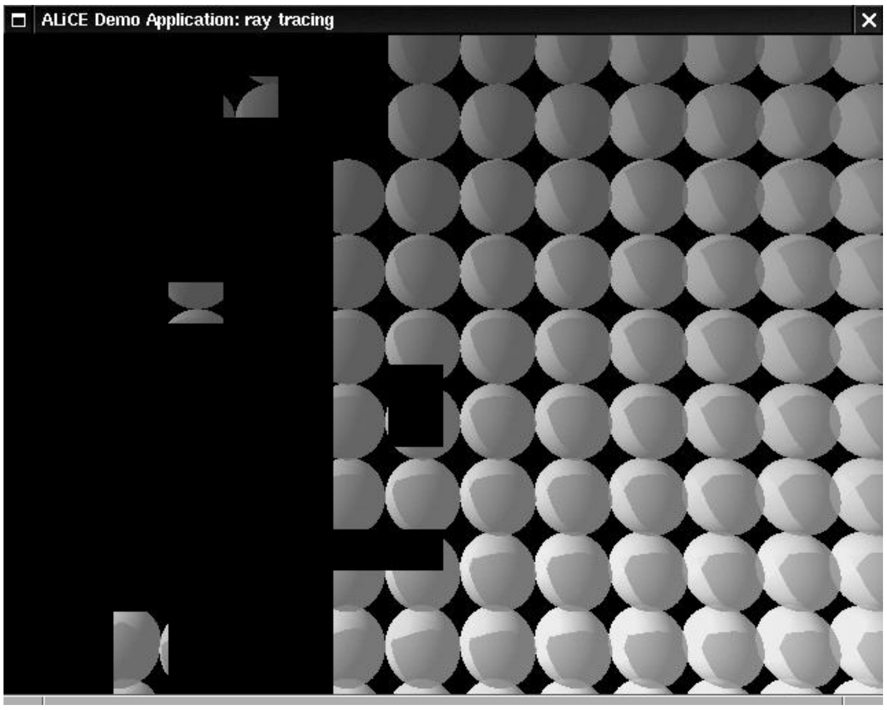
One interesting comment[4] about this application was the ease of how much we boost up the application performance just by using compiler optimizations and profiling to detect bottlenecks. Basically we almost doubled the performance after a few minutes of analyzing the profiling data.

[1]the application is in the public domain

[2]Qt libraries are developed by Troll-Tech and are a used in the de-facto Linux GUI KDE

[3]for threads we used the pthread library, the POSIX conformant Linux library for threads

[4]just to prove how easy is to develop applications for a well established language, one of the reasons we



q

Figure 4.1.: Screenshot of the C++ ray-tracer application running

4.2. Mandelbrot fractal

The other application that we present here is a Mandelbrot fractal generator. It's development used the same path as the other application: ported from Java (this time to C), wrapped in a thin layer of code to use it for ALiCE: a task to compute portions of the fractal, and another one for a collapsed task generator - result collector.

Porting the application to C/Linux also replaced the GUI support form Java/AWT to Qt and added multi-threading support for smooth visualization of the fractal, as parts of it are completed by producers.

choose to implement non-java support

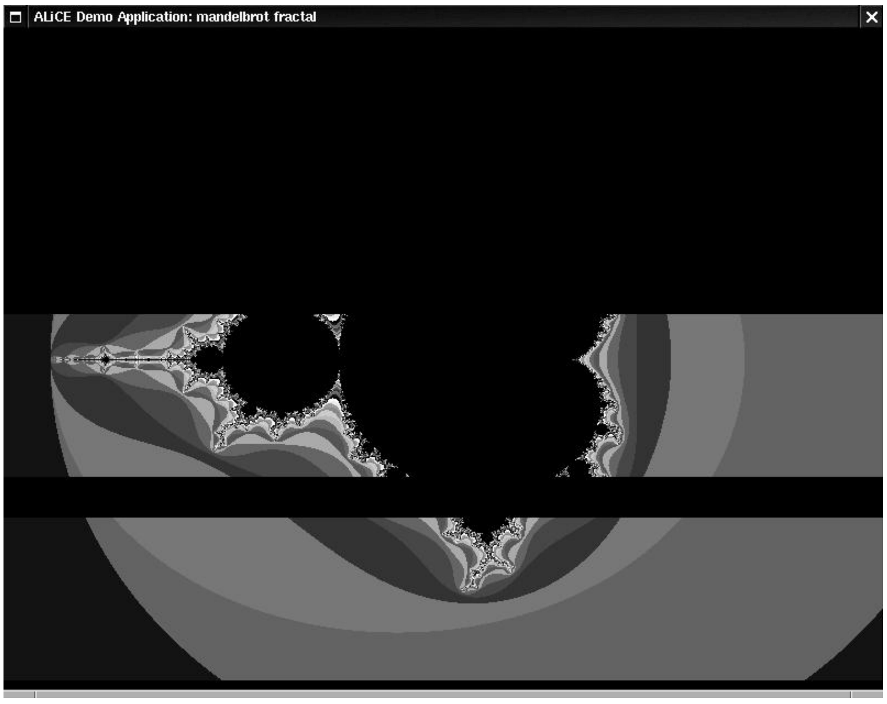


Figure 4.2.: Screenshot of the C Mandelbrot fractal generator running

5. Performance analysis

5.1. The test bed

Our experiments were carried out on a cluster of twenty-four nodes shown in Figure 5.1 on page 36. Sixteen (named ws00 to ws15) are Intel PII 400MHz with 256MB of RAM, and eight (named ws17 to ws24) are Intel PIII 866MHz with 256MB of RAM. These nodes are connected to each other via a 100Mbps switch. All nodes are running RedHat Linux release 7.0 (Guiness) distribution, based on the 2.2.16-22 Linux Kernel.

ALiCE is developed using the Java TM 2 Software Development Kit version 1.3.1 and 1.4.0 and the Jini Starter Kit version 1.2. We used GigaSpace TM Platform 2.0 in our actual stage of development and testing. For development and experiments, we use the Java TM 2 Runtime Environment Standard Edition with the Java TM HotSpot Server and Client Virtual Machines build 1.3.1_03-b03, mixed mode. The HotSpot Server Virtual Machines are used for Resource Broker and Producer nodes. The Consumer nodes make use of the HotSpot Client Virtual Machine. During our test the need to have a machine with more memory to hold GigaSpace raised, so we moved 128 MB of RAM from ws04 to ws21, which held GigaSpace for all our experiments. We also tried to run GigaSpace on a Sun Ultra 30 station, based on an UltraSparc II 266Mhz, but this machine proved to be much to slow in terms of processor power than needed. The conclusion is that if GigaSpace is held on only one machine, that machine should be a powerful machine with lots of memory.

The tests were performed using the ray-tracer application presented in Section 4.1. The scene consisting of a 100 spheres with reflection and transparency properties and some lights of different colors. Shadows were also taken into account. The image size used in the tests was 1024x768.

Before presenting the tests and results we obtained, we should make a few consideration about some of the application's particularities, otherwise the results we obtained might be strange. The ray-tracer application is partitioning the scene to render in such way that each

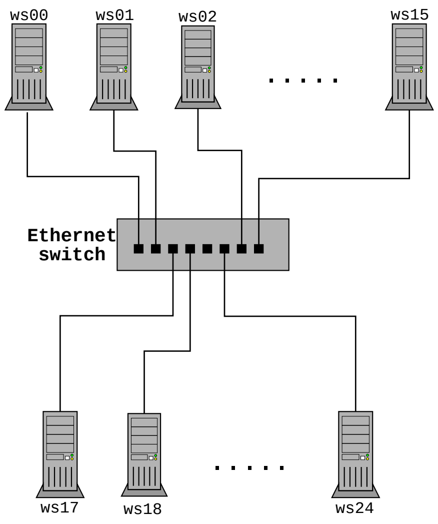


Figure 5.1.: The cluster used for performance analysis

task will compute almost the same number of pixels. This however does not mean that the tasks will have the same amount of work to do, due to the fact that a certain portion of a scene will have more objects, lights, shadows, etc. If the number of tasks is small, the probability that one task will have to do a more intensive computation rises. And if the number of tasks are greater than the number of producer nodes, than the results will depend on the order in which the tasks were submitted to producer (which is not something we control). Also it's important to say that the we obtain the best performance when the number of tasks is a multiple of the number of producer nodes. These explains why the execution time is slightly descending when we rise the number of tasks but the number of tasks is still small.

5.2. Scalability test

In this test we tried to measure the system scalability. It is important to understand that we can't measure the system scalability directly, but only indirect by measuring applications performance. We ran two tests: one with producers running one thread, and one with producers running two threads. The results shows that running producers with 2 threads brings no benefits, but only performance penality (all the machines are single processor). The first six results are obtained by running the tests on the PIII machines, the rest used the six PIII machines and aditional PII machines. The results are presented in Table 5.1 on page 37 and Figure 5.2 on page 37.

Producer nodes	Completion time - 1 thread (sec)	Completion time - 2 threads (sec)
1	268.5	267
2	135.5	137
3	92	94
4	68.5	74
5	59	62.5
6	51	59.5
7	48.5	57
8	46.5	53
9	45.5	53
10	45	56
11	43.5	54
12	43.5	55
13	41	57
14	40.5	53
15	38.5	53
16	42	56.5
17	42.5	58

Table 5.1.: Performance study for system / application scalability; the C++ ray tracing application generated 64 tasks on each run

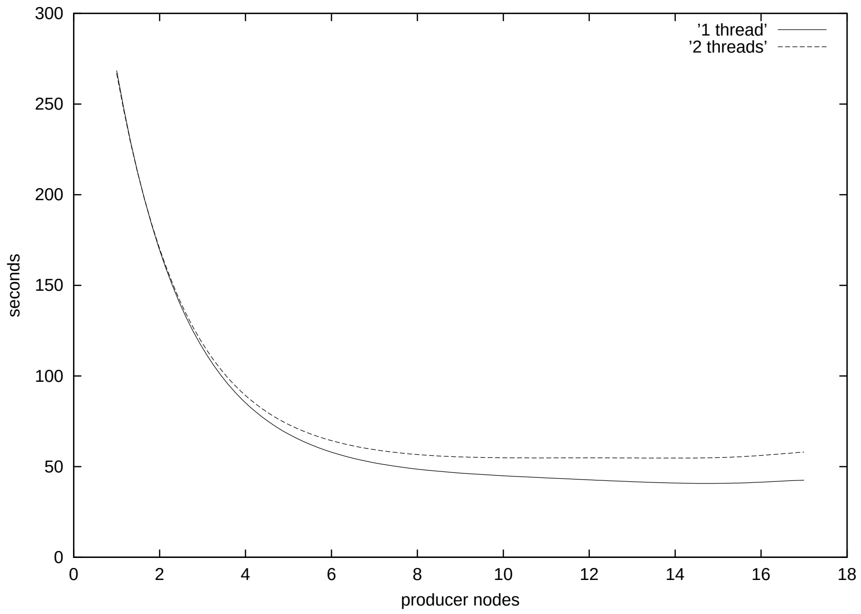


Figure 5.2.: Performance study for system / application scalability; the C++ ray tracing application generated 64 tasks on each run

Tasks	1st run (seconds)	2nd run (seconds)	3rd run (seconds)	Average (seconds)
6	85	72	64	73.66
9	66	63	62	63.66
12	69	73	71	71
15	62	61	63	62.00
24	62	60	59	60.33
30	59	60	61	60.00
48	57	60	61	59.33
88	59	58	61	59.33
130	59	62	60	60.33
234	75	75	74	74.66
494	101	107	106	104.66
875	234	230	240	234.66
1976	612	543	645	612

Table 5.2.: Performance study for a given problem with a fixed number of producer nodes (5 Pentium III at 866Mhz with 256M of RAM) and varying the number of tasks

5.3. Task overhead test

In this test we varied the number of tasks for a problem while keeping the number of producer nodes constant. We used 5 producer nodes, consisting of Pentium II machines at 866MHz and 256M of RAM. The number of tasks were varied from as few as 6 to as many as 1976. The peak performance was obtained for a quite large interval (30-130 tasks). While the fact that we did not obtain peak performance for a number of tasks closed to the number of producers was explained in Section 5.1, the fact that we obtained peak performance while varying the number of tasks to as many as 100 suggests that the ALiCE system overhead is quite small. For result take a look at Table 5.2 on page 38 and Figure 5.3 on page 39.

5.4. Java vs native test

In this test we compared the results for two runs of the same application with the same parameters (number of tasks, same problem size) one developed for Java, and one for C++ (the ray-tracing applications). The results in Table 5.3 on page 40 shows what we've already expected. The non-Java applications and runtime support are running faster, roughly two times faster. Also, as the number of tasks per application is growing the Java runtime-support performance is degrading faster than the non-Java runtime support, as shown in Figure 5.4 on page 40. This is normal, because of the serialization process which is invoked each time an object is saved/loaded. For each save/load the Java runtime-support has to save/load each

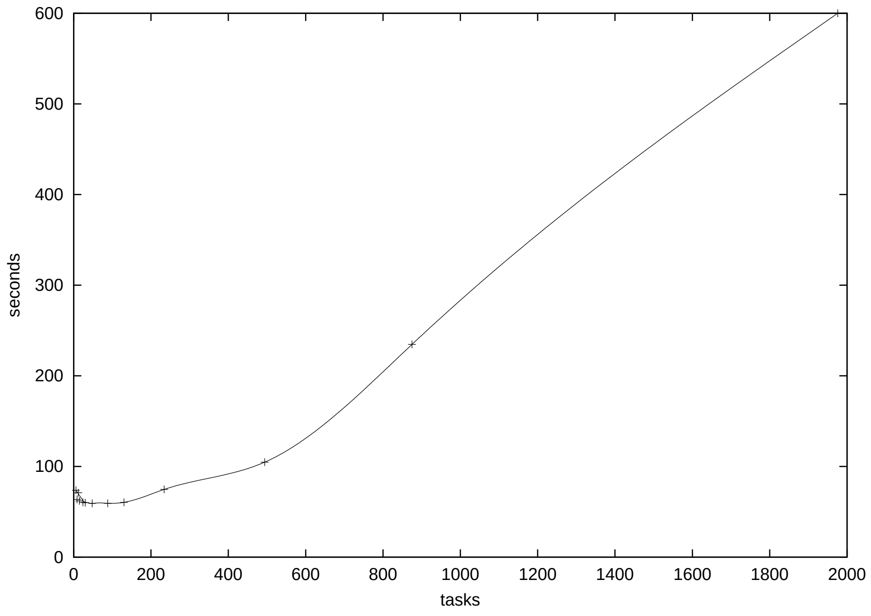


Figure 5.3.: Performance study for a given problem with a fixed number of producer nodes (5 Pentium III machines with 256M of RAM) and varying the number of tasks

field of the object. Which is significantly a much more overhead then approach that we use for the non-Java runtime-support (remember, we save the whole data segment, not each variable in turn).

5.5. Persistent heap allocator test

To test the performance of the persistent heap allocator we used the example provided in Section C as well as the equivalent non-persistent program. Basically the program inserts/removes items from a simple linked list.

As it can be seen the results in Table 5.4 on page 41, and the graphic represention in Figure 5.5 on page 41 and Figure 5.6 on page 42 shows the fact that our allocation algorithm performs better than the glibc's algorithm (for this example!), despite the encoding/decoding pointers overhead (which is insignificant, something that we would expect). This makes us think that the current implementation is good enough for testing the feasibility of the persistent object model.

Number of tasks	C++ execution time (seconds)	Java execution time (seconds)
6	73.66	163
9	63.66	158
12	71	135
15	62.00	141
24	60.33	120
30	60.00	115
48	59.33	107
88	59.33	112
130	60.33	115
234	74.66	117
494	104.66	270
875	234.66	285
1976	612	1230

Table 5.3.: Performance study between the Java and non-Java application and runtime support; we used 5 producer nodes (Pentium III at 866MHz with 256M of RAM)

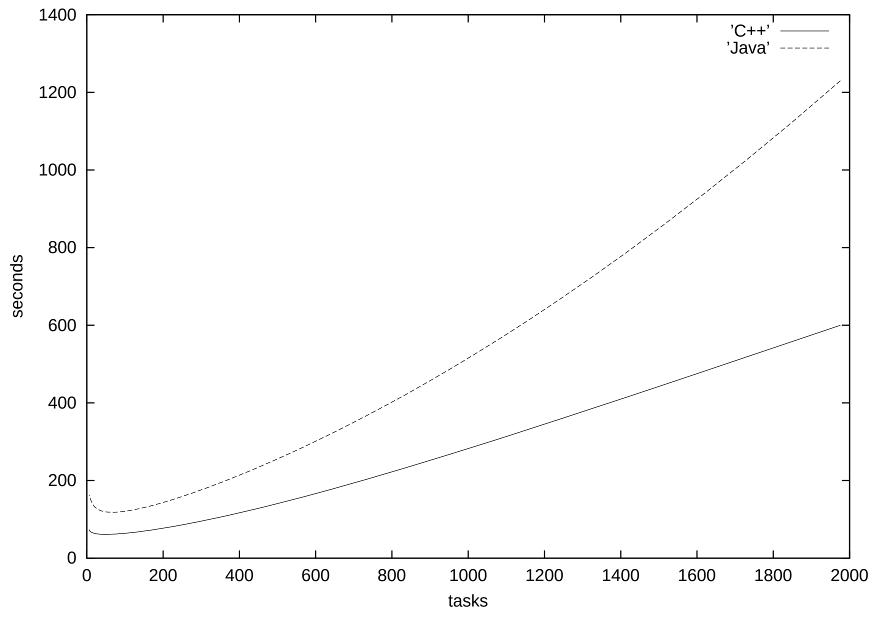


Figure 5.4.: Performance study between the Java and non-Java application and runtime support; we used 5 producer nodes (Pentium III at 866MHz with 256M of RAM)

Operations	PO insert (sec)	GLIBC insert (sec)	PO remove (sec)	GLIBC remove (sec)
1,000,000	0.13	0.16	0.09	0.09
1,500,000	0.18	0.24	0.15	0.14
2,000,000	0.25	0.32	0.19	0.18
2,500,00	0.31	0.42	0.23	0.22
3,000,00	0.37	0.49	0.28	0.27
3,500,00	0.44	0.56	0.33	0.30
…	…	…	…	…
10,000,000	1.21	1.63	0.92	0.88

Table 5.4.: Performance analysis for the persistent heap allocator

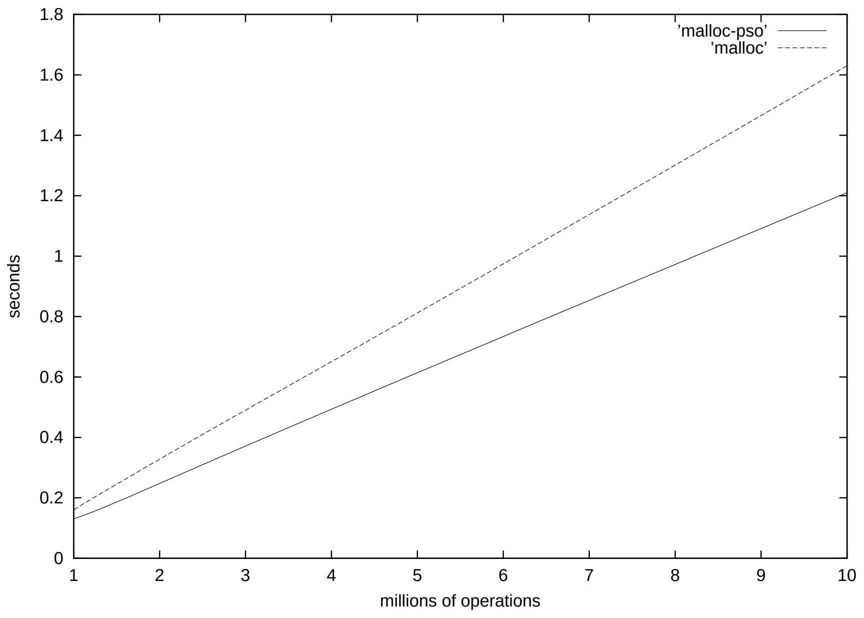


Figure 5.5.: Comparison between the persistent heap allocator and the system heap allocator for *malloc* operations. The comparison was performed while running a program which inserted/removed elements from a simple linked list.

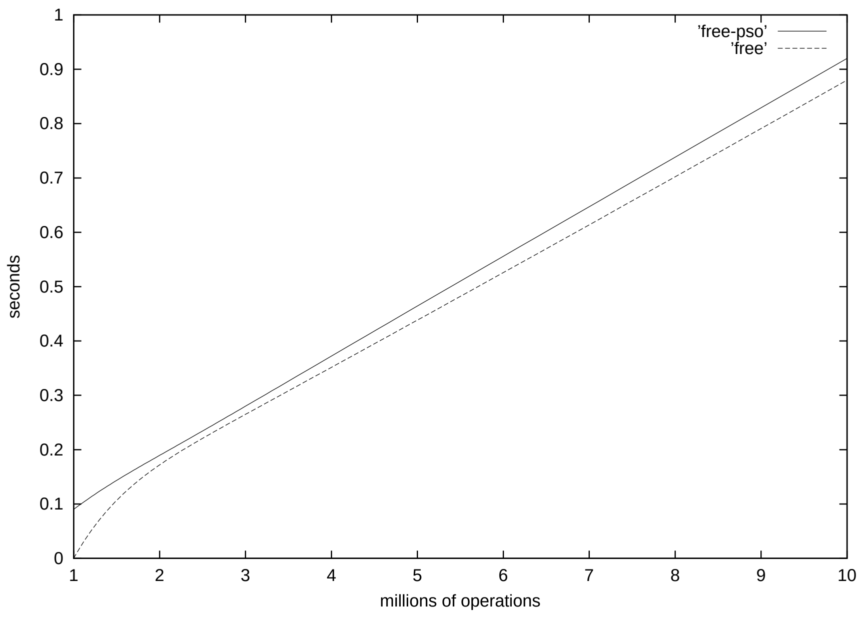


Figure 5.6.: Comparison between the persistent heap allocator and the system heap allocator for *free* operations. The comparison was performed while running a program which inserted/removed elements from a simple linked list.

6. Conclusions

6.1. Summary

Support for non-Java applications in ALiCE was developed in the first place in order to achieve two main benefits: increased performance for computing power hungry application and reduced development time for existing applications. The results obtained from the experiments we performed (see Section 5) gives as resons to believe that we succeded in our task.

In order to implement an easy to use non-Java API for programmers, which will hide the complexity of writing GRID applications, we developed an layer of abstraction built on top of the hosting operating system. In order to be as fast as possible this layer interacts intimate with the operating system. However, the implementation is based as much as possible on POSIX and ISO-C standards features to achieve a high degree of portability. It also tightly interacts with Java through Java Native Interface. Thus, we built a layer sitting between the ALiCE system and the operating system, in order to support native applications in ALiCE.

6.2. Future work

The current ALiCE implementation works nice for a certain kinds of deployment. It is mostly targeted to Intranets which have a few and known number of users that run applications all in the same time and thus in which quality of service requirements are not so important. However the current implementation will not perform well in an environment which has more than a few users[1], with some applications having priority over other ones. That is because once a producer has been assigned a task to run it will run that tasks until completion, even if some more important application (with a higher priority) has tasks that were submitted after are waiting for free producers. One “solution” would be to cancel the execution of

[1]we reffer here to consumer nodes, not producer nodes

problem	how to fix	where to be implemented
pointers	keep pointers relative to segment; compiler generates code to translate and access the pointer; dynamic linker relocates segments base	compiler, dynamic loader
stack relocation	aditional code generated by the compiler for function prologue and epilogue; dynamic linker relocates segments base	compiler, dynamic loader
register relocation	protect code regions that are using absolut pointer values from beeing preempted	controller process
memory layout	3 segments: code+rodata, data+bss+heap and stack	-
task preemption	save the program counter and stack pointer;	controller process
code sharing	separate code and data segments	dynamic loader, linker

Table 6.1.: Various problems and solutions for preemtable and code sharable persistent objects

some low priority tasks. Those tasks would have to be restarted, thus wasting resources. A better approach would be to preempt the low priority tasks, with the possibility to resume the execution later, possibly on another node. *Preemption and migration* is required in order to implement efficient fairness use of the system's resources by concurrent applications.

Another important drawback of the current implementation is the fact that the code is not shared among the same kind of objects, neither when the objects are loaded in memory neither when they are transported. *Code sharing* would make ALiCE faster in operations and less memory and network bandwidth consuming.

Finally, another important drawback of the current implementation is the fact that the programmer has to encode/decode pointers in order to keep the object persistent. *Compiler support for pointer referencing and dereferencing* is not only a nice feature to have, it is in fact necessary if we want to be able to preempt tasks, as we will show further.

The most demanding requirement from those above is preemption of tasks. Since the programmer have no control of when object's code would be preempted, we no longer have to have two type of variables (from the point of view of the object's persistence) like persistent and non-persistent variables. Instead we need all variables to be persistent.

Compiler support for referencing/dereferencing pointers would be vital if we want to support preemption. Since all the variables have to be persistent, all pointers have to be persistent. In order to do that, we need to save the pointers in a way that will be independent of code/data/heap/stack location in the address space. Pointers will be kept in an encoded format (segment:offset) in order to be independent of the location of where the segment is present in the virtual address space. This format depends on what kind of memory layout we will use.

To decide on the memory layout we will use, we have first to get through an overview of the kinds of variables we will support. Each of these kinds of variables are kept together in sections like:

data this section will keep the global and local static variables

bss this section will keep the uninitialized variables

heap this section will keep dynamically allocated memory

stack this section will keep (beside other things like parameters to functions and return addresses) the local variables

When choosing a memory layout of the object's data we basically have to extreme aproaches: either each kind of data is kept in separated segments, either we keep all kinds of data in a single segment. Both of theese aproaches have advantages and disadvantages: while keeping each kind of data in each own segment proves to be very flexible and low on memory overhead, it differs significantly from the ABI used in most of the modern OSes, thus making the implementation more intrusive and less portable. Supporting more segments means using more bits from the pointer for identifying the segment, which in turn leads to having less bits for offset, thus reducing the adresable virtual address space whithin an object. On the other hand, keeping all the data and code of the object in one segment even if it will not harm the ability of the object to access the hole virtual address space, it will impose some penalities. Since the code and data have to be contained in single segment, we will be unable to share code (and also read-only data) among different objects. Keeping the heap and stack together will result in poor performance, as both of these segments could grow, and at some point memory will have to be moved around. This is why we propose that we keep three segments per object: code+rodata, data+bss+heap and stack.

Preemption will also require to save to the context of the object, so that it will be restored when the object will restore execution. The object's context consists of:

- program counter register value
- stack pointer register value

- general and/or other register's values (floating point registers too)

Program counter register and stack pointer register values will need relocation at run-time due to the fact that the stack/code could be loaded at another virtual address. General/other registers would also need relocation if their values would contain pointers. However, it would be difficult or performance degrading to maintain information about registers holding pointers; that is why we present an alternative that we think that it's better.

Another important problem that have to be solved to have persistent preemptive objects is *relocation of the general registers* that contain pointers. Since we save the pointers in a relocatable format it seems that this is not a problem. And in fact it is not. The real problem is that in order to get a reference to / dereference a pointer the compiler might need additional registers/temporary variables on stack to compute an absolute address so that it can fetch the desired value. If the preemption occurs during this critical time things will break. One solution would be to protect such critical code. But as we will probably implement preemption by using signals, that means that we have to block/unblock a certain signal every time we get a reference to / dereference a pointer. Not an option for performance. Another approach would be to mark somewhere the code portions that are critical, and if we are inside one of the critical portions to let continue execution until we exit from it (a few more instructions).

One more important problem that we face when dealing with preemptive and migrational objects is the *stack relocation*. When programs calls routines the return address is saved on stack. With preemptive and migrational objects is possible that the code will be loaded at a different address. Thus, it is necessary to relocate the return addresses from stack to point to the correct location. Although stack relocation is an option, we choose not to approach it, as it is requiring extra information and execution time to do the relocations even if some of them might never be used. The approach we choose is resembling the solution we adopted for relocation of general registers: we keep the return addresses on stack in relative offset from the base of the segment. To do that, the function prologue end epilogue will have to be modified to take this into account: the prologue will have to put on stack the relative offset (as we can't call with relative, but only absolute values) and the epilogue will have to change that back to absolute addresses (as we can't use relative addresses to return, but only absolute).

To support shared code we will also need to change the architecture of the current implementation. Thus, the pure-objects will contain the code and initialized data. The non-pure objects will not contain code at all, but only data (initialized/uninitialized/heap/stack) and some other information needed for various operations plus dependency information (like the pure-object that this non-pure object will depend on). In order to support a shared code architecture, the programs will have to be compiled in *position independent code* [4, 3] format. More, when code will execute on behalf of different objects the context switch will also have to *refill the .GOT*[2] (and maybe *.PLT*[3]*)* [4, 3] so that it will point to correct locations.

In order to support preemption, the most appealing solution seems to use signals. Signals are well supported on most the of modern operating systems. However, the context switching will remain a problem. One possible solution could be to use the *setjmp* and *longjmp* primitives that should be available to platform that are implementing the POSIX standard combined with specific OS/architecture code that will handle the context saving/loading. We choose not to use this approach due to implementation complications (like how are we going to allow executing code after passing the critical sections). We think that using a *ptrace* approach would be better, in terms of performance and ease of implementation. *ptrace* is a system call that is widely available in UNIX systems and that is used for debugging purposes. Using *ptrace* one process is able to control another process in terms of: stooping, starting, peek/poke registers and memory, trapping single instruction execution (single step) and system calls.

Finally, there are some library issues when dealing with the new architecture proposed. Since we are preempting the execution of objects, it might happen that we are preempted while we are executing a library call. We have two options here: either recompile the libraries so that we make them data relocation friendly, or wait until the execution exits the library call.

[2]Global Ofsset Table
[3]Procedure Linking Table

A. Programming templates and API

A.1. Defining and using an ALiCE object

Defining an ALiCE object should be done by using the following template:

```
#define __ALICE_OBJECT__
#include <ALICE/object.h>
ALICE_OBJECT(object_name);
void alice_object_init(void *arg)
{
       /* your code here */
}

void alice_object_load(void)
{
       /* your code here */
}

void alice_object_save(void)
{
       /* your code here */
}

/* more of your code here */
ALICE_OBJECT_END;
```

The *object_name* identifies the class of objects that can be instantiated from the definition of the object. As we present later, the programmers have to use it when creating new objects.

All the code related to ALiCE programming (code that would call functions from ALiCE's API) should be contained between the *ALICE_OBJECT()* and *ALICE_OBJECT_END*.

The *alice_object_init(void *arg)* is a function that must be defined when defining an ALiCE object. This function will be called exactly after a new object is instantiated. Briefly, an instantiation of an object means creating a in-memory copy of the object's definition, as defined by the programmer with the template. A copy is created, so that other instantiations may be possible. The argument to the function will be passed from the *alice_new_object()* function call. The programmer should implement this function so that it will initialize the object. More on what need to be done here is covered in Section A.5.

The *alice_object_load()*, respectively the *alice_object_save()* functions will be called by the runtime when the object is about to be unloaded from memory, respectively after the object was loaded in memory. The programmer should make sure that the state of the object is consistent after exiting from these methods. Considerations about what should these functions accomplish are made in Section A.6.

NOTE: the functions described above should NOT be declared *static*.

struct alice_od* alice_new_object(object_name, void *arg)

This function instantiate a new object of type *object_name*. The init function of the object will be called with the value of *arg* as parameter.

A pointer to a *struct alice_od* (an object descriptor) have to be used by the programmer in order to use this object. Note that the pointer is allocated from the dynamically persistent data, thus it will always be safe to use the object, without the need to concern about the consistency of the *value* this pointer. This value should be kept in a persistent variable. More about persistent variables in Section A.3.

The allocated object will be "linked " with the object that created it until it will be explicitly free, i.e. when the allocator object will be moved to another node of the system the allocated object will also be moved.

struct alice_od* alice_clone_object(struct alice_od *od)

This function creates an exact copy of the object *obj* (as stated by the persistent object overview in Section 3.4.1; e.g. the statically and dynamically persistent data will be the same; and of course the code and other readonly data too). The cloned object will be "linked" with the one which cloned it as presented in the above paragraph.

void alice_free_object(struct alice_od *od)

This function unlinks the object *od* and unload it from memory immediately.

void alice_set_object_data(void *arg)

This function sets an internal field of the object from which this function is called to the value of the *arg* pointer. This value can be read by other objects by calling the *alice_get_object_data* function (see below).

void* alice_get_object_data(struct alice_od *od)

This function retrieves the value stored with the function *alice_set_object_data* presented above. Since this function is used to get data from another object the user must used it with care. If the value represents the value of a pointer, it must be translated by the object which holds it to an usable pointer each time the object is loaded and converted back to a persistent value when the object is saved (see Section A.3). The user might want to use the load/save mechanism presented above to accomplish this (see Section A.6).

extern int alice_put_object(struct alice_od *result, int type)

This function puts an object into the application space. Any object belonging to this application will be able to get this object by using *alice_get_object()* function.

struct alice_od* alice_get_object(int type)

This function obtains an object of type *type* from the application space. Together with the *alice_put_object* this function can be use to communicate between different tasks of an application. It is at least use for sending results from the producer nodes to the result collector.

struct alice_od* alice_current_object()

This function returns an object descriptor of the caller object, i.e. the object which made the call.

A.2. Defining and using an ALiCE task

To define a task use the following template:

```
#define __ALICE_TASK__
#include <alice/object.h>
ALICE_TASK(task_name);
void alice_object_init(void *arg)
{
       /* your code here */
}

void alice_object_load(void)
{
       /* your code here */
}

void alice_object_save(void)
{
       /* your code here */
}
void alice_task_execute(void)
{
       /* your code here */
}
/* more of your code here */
ALICE_TASK_END;
```

The programmer can create new tasks, clone and free them with the same functions as for plain objects.

extern int alice_process_task(struct alice_od *task)

The function hands over the task and its related objects to the system. The system checks for the availability of a function called *alice_task_execute* and if present it schedules it for execution (the resource broker does the scheduling) to a specific producer node of the system.

A.3. Keeping the object persistent

To keep the object persistent, the programmer can make use of global persistent variable as well as persistent dynamically allocated memory.

To make a global variable persistent the programmer has to use the keyword *persistent* after declaring the variable, but before initializing it. For example, if we want to have two persistent variables *var_a* and *var_b*, and one non-persistent variable *var_c* all of them being of type *int* we would use something like:

```
int var_a persistent, var_b persistent = 0, var_c;
```

NOTE: the *persistent* keyword is case sensitive and it's lowered cased. Also, the persistent variables means that the *exact value* is saved before the object will be moved from a node and will restore later, when the object will be loaded on another node.

But just global persistent variables will not be enough to write complex programs. That is why, the programmer can use persistent dynamically allocated data to suit his needs. The following functions are available to dynamically allocate persistent data:

```
void* alice_malloc(unsigned long size);
void alice_free(void *ptr);
void* alice_malloc_decoded(unsigned long size);
void alice_free_decoded(void *ptr);
```

The functions from above looks similar to their operating system counterparts: *free()* and *malloc()*. However, certain restriction are imposed to the programmer in order to make the allocated space persistent. With persistent dynamic data there are two kinds of pointers: usual pointers that can be used to access the allocated data, and encoded pointers that are used to keep the pointers values persistent. Thus, as a good an error-free style of programming, the programmer should keep the values of the pointers encoded, and when they need to access data through a pointer, they should decode the pointer to an usual pointer, and then access the data through the converted pointer. If the encoded pointer is decoded itself, it must be encoded back after the it's job is done.

The *_decoded* functions are used when the pointers passed can be directly used by the user to access data. The other functions are expecting encoded pointers. The *malloc* functions return either an encoded/decoded pointer to the allocated memory area, either *NULL* if the process runs out of memory. The user has access to these functions to convert pointers:

```
ptr_type* pso_encode_ptr(struct pso *pso, ptr_type *ptr)
ptr_type* pso_decode_ptr(struct pso *pso, ptr_type *ptr)
```

where ptr_type is actually the type of the pointer passed to the function, and is dynamically determined.

NOTE: the data is allocated / freed for the object that makes the *allice_free()* and *alice_malloc()* calls.

A.4. Platform requirements

The programmer should specify the platform requirements when defining the object, using the following format:

```
ALICE_OBJECT(name, requirement1:value1, ..., require-
ment n:value n);
```

The requirements are described in Section 3.5.4. For example:

```
ALICE_OBJECT(dummy, os:”linux”, proc:”i386”);
```

will create a new ALiCE object with the name *dummy* and which will need the linux operating system and an i386 class processor to run.

A.5. Object interactions

Altough we tried to automate and make transparent to the user every aspect of keeping the object persistent, the current implementation still not deals with all the posibilities. That is why we provided in the first place the *alice_object_load()* and *alice_object_save()* functions. So that the programmer use them as extreme ways of keeping the object persistent, for some programming language constructions that we do not handle yet.

Since we adopted an object-oriented programming model, only the objects themselves are able to access their data and code. Another object can NOT access data or call functions from another object. Yet, objects needs to interact between them in order to solve complex problems. Using ALiCE API and programming model, one can easily do that. Let’s consider the following structure:

```
struct matrix_op {
        void (*rand)();
        void (*copy)(struct matrix_op*);
        void (*set)(int i, int j, int f);
        int (*get)(int i, int j);
        void (*print)(void);
};
```

which describes the operations of an object that implements a matrix. Now let's consider the following code excerpt:

```
ALICE_OBJECT(matrix);
struct matrix_op matrix_op;
/*
 * more code here implementing the
 *         matrix_rand,
 *         matrix_copy,
 *         matrix_print,
 *         matrix_get and
 *         matrix_set functions
 */
void alice_init_object(void *arg)
{
        matrix_op.rand=&matrix_rand;
        matrix_op.copy=&matrix_copy;
        matrix_op.print=&matrix_print;
        matrix_op.get=&matrix_get;
        matrix_op.set=&matrix_set;
        alice_set_object_data(&matrix_op);
        /* more initialization code here */
}
ALICE_OBJECT_END;
```

Each time an object will be created, the internal field of the object which can be accessed with the *alice_get_object_data()* function, will be set to a pointer to a structure pointing to the operations that the object chooses to make public. Thus the following sequence of code will print the contents of the matrix:

```
struct alice_od *matrix_a;
struct matrix_op *a_op;
a_op=alice_get_object_data(matrix_a);
a_op->print();
```

A.6. Using *alice_object_save()* and *alice_object_load()*

From the previous code excerpts we saw how easy is too interact between objects. However, because of extensive use of the dynamic loading tehniques, the method presented above is not safe. Because the pointer values obtained witht the & operator are not persistent. The current implementation has no means to transform these pointers into persistent ones. That is why one should use the following lines of code to make the previous example safe:

```
void alice_object_load(void *arg)
{
        matrix_op.rand=&matrix_rand;
        matrix_op.copy=&matrix_copy;
        matrix_op.print=&matrix_print;
        matrix_op.get=&matrix_get;
        matrix_op.set=&matrix_set;
        alice_set_object_data(&matrix_op);
}
```

These lines of code will be called each time the object will be loaded in memory, before control it's passed to the program code. Thus, the pointers will point to the right place, as they were initialized right after the object was loaded in memory.

A.7. Limitations and caveats

Although we tried to make the system as complete as posible, there are still some unresolved issues caused manly by rellocations at load time. Because of this, constructs as:

```
const char *string persistent = “VOODOO”;
```

would not work correctly. The programmer should never use such constructs, or the program would crash. This will be worked out in future versions.

A.8. Compilation, packaging and tools

A.8.1. Compilation

The compilation process for an alice object is very simple. Beside the usual compilation steps, you should compile it like a shared object. For the gcc compiler this is done by passing the *-shared* flag to the compiler. There is no need to compile the object in position independent code [3, 2] (e.g. for the gcc compiler use the *-fpic* of *-fPIC* flags) as code sharing is not yet posible to achieve with ALiCE objects.

A.8.2. Packaging

An important tool that you must use if want to develop application for ALiCE is the *alice-pack* utility. It expects either one or two arguments. If you pass only one argument it will expect one ALiCE task which should implement both the functions of a result collector and of a task generator (we call this kind of task a collapsed task generator - result collector). As you've probably guessed when passing two arguments it will expect a task generator and a result collector (in this order!). Based on it's inputs the utility will generate one package for the application. The package it's a *.tar.gz* file containing the task generator, the result collector and all the static dependencies of these files. In the process it will also do the static dependencies for every object in the package.[1]Another important task of the packager is to create a common platform requirements between the static dependent groups of objects[2] from the package. For example if we have two objects that statically depend on each other, and one object has as requirement the linux operating system, and the other no requirement, they will both have linux as a requirement. This is also done to speed up the process of packaging while running the application.

A.8.3. Tools

alice-exec-task is provided so that the user is able to test it's code, before send it over into the ALiCE system. It will expect one parameter, an ALiCE task object file. The utility will load and execute the task. However, since this utility it's not linked (connected) to the ALiCE system, the operations we can perform within the task are limited. Any send/receive operation will fail, causing the task to terminate.

[1]remember that even the direct static dependencies are known right after compilation, the indirect static dependencies are not; in order to speed up dependency lookup when running the application all static dependencies – both direct and indirect, are looked up at this time and saved in the object for future lookups

[2]a static dependent group of objects are a set objects that statically depend on each other, be it direct or indirect

```
$ bin/pso-dheap demo/c/fractal/fractal-0-0 1
heap mmap_addr=0x401cb000 mmap_size=0x1000 first=0x401cb000
last=0x401cb2d8 free=(nil) size=0x2f0
       addr       size     prev_size  used?  next_free  prev_free
001 0x401cb000 0x00000068 0x00000000   yes      (nil)      (nil)
002 0x401cb068 0x00000020 0x00000068   yes      (nil)      (nil)
003 0x401cb088 0x00000020 0x00000020   yes      (nil)      (nil)
...
026 0x401cb2d8 0x00000018 0x00000018   yes      (nil)      (nil)
Total chunks 026, free chunks 000
Total size 0x000002f0, free size 0x00000000
$
```

Figure A.1.: Output from *pso-dheap*

```
$ bin/alice-shdep demo/c/fractal/fractal-0-0
static: task-0-0
static: result-0-0
$
```

Figure A.2.: Output from *alice-shdeps*

pso-dheap takes two arguments, a persistent object file and the level of detail, and prints information and statistics about the persistent heap of the object. The level of detail should be a number between 0 and 3. The level of detail is incremental (means that information that is printed on level 0 is also printed on level 1). The given informations per detail level are:

- 0 - the number of free and used chunks as well as the total size of the heap and how much free bytes are available
- 1 - information about each chunk of the heap (like address, size of the chunk, size of the previous chunk, status of chunk – free or used, and pointers to the next and previous chunks in the list of free chunks)
- 2 - the list of free chunks
- 3 - contents of each used chunk

alice-shdeps is provided so that one can inspect the dependencies of an ALiCE object. It will expect one ALiCE object filename as argument on invocation and will print out the static and dinamic dependencies of the object.

```
$ bin/alice-sharch demo/c/fractal/fractal-0-0
Runtime support version: 0.0
OS: linux
OS features: 0x0
OS version: 0.0
Processor: i386
Processor features: 0x0
Library dependencies:
        /lib/ld-linux.so.2
        /lib/libNoVersion.so.1
        libalice.so
...
```

Figure A.3.: Output from *alice-sharch*

alice-sharch is provided so that one can inspect the platform requirements for an ALiCE object. It will expect one ALiCE object filename as argument on invocation and will print the platform requirements of the object: the runtime support version, the OS, OS version and OS features, the processor and processor features as well as library dependencies.

alice-c-consumer is expecting two or three parameters. If you pass two parameters it will expect first the package file name and then the name of the task generator - result collector collapsed task that should be present in the package. If you pass three parameters it will expect the package filename, the name of the task generator and the name of the result collector (in this order!).

alice-c-producer is expecting two parameters: the number of threads to run a task generator executors and the number of tasks to be run as tasks executors. Note the *alice-c-consumer* and *alice-c-producer* will not work in the same time on the same machine. You should use the GUI for that purpose.

B. A simple example

This is a simple ALiCE application consisting of a distributed matrix multiplication program. The two matrices have NxN elements and N^2 tasks are generated to compute the resulted matrix. Each task will be initialized by the task generator to carry within the two matrices which should be multiplied as well as the position in the result matrix that the task should compute. To demonstrate how simple object oriented tehniques can be used with the ALiCE API, the application makes use of an matrix object.

The result collector

```
#define __ALICE_TASK__
#include <c/task.h>

#include "task.h"
#include "matrix.h"
#include "result.h"

ALICE_TASK(collector);

struct alice_od *a persistent, *b persistent;
int line persistent, column persistent, size persistent;

void alice_object_init(void *arg)
{
}

void alice_task_execute(void)
{
        struct alice_od *m, *rod;
        struct matrix_op *m_op;
        int i, j, size=10;
        struct result *r;

        m=alice_new_object(matrix, (void*)size);
        m_op=alice_get_object_data(m);

        for(i=0; i<size; i++)
                for(j=0; j<size; j++) {
                        rod=alice_get_object(0);
                        r=alice_get_object_data(rod);
                        m_op->set(r->line, r->column, r->result);
                        alice_free_object(rod);
                }

        m_op->print();
}

ALICE_TASK_END;
```

The task generator

```
#include <stdio.h>
#include <stdlib.h>

#define __ALICE_TASK__
#include <c/task.h>

#include "task.h"
#include "matrix.h"

ALICE_TASK(taskgen)


void alice_object_init(void *arg)
{
}

void alice_task_execute(void)
{
        struct alice_od *a, *b, *task;
        struct matrix_op *a_op, *b_op;
        struct task_init ti;
        int size=10, i, j;

        a=alice_new_object(matrix, (void*)size);
        a_op=alice_get_object_data(a);
        b=alice_new_object(matrix, (void*)size);
        b_op=alice_get_object_data(b);
        a_op->rand(); b_op->rand();
        a_op->print(); putchar('\n'); b_op->print();

        for(i=0; i<size; i++)
                for(j=0; j<size; j++) {
                        ti.line=i; ti.column=j; ti.size=size;
                        ti.a=a_op; ti.b=b_op;
                        task=alice_new_object(task, &ti);
                        alice_process_task(task);
                }
}

ALICE_TASK_END
```

The task (header)

```
#ifndef _TASK_H
#define _TASK_H

#include "matrix.h"

struct task_init {
        struct matrix_op *a, *b;
        int line, column, size;
};

#endif /* _TASK_H */
```

The task

```
#define __ALICE_TASK__
#include <c/task.h>

#include "task.h"
#include "matrix.h"
#include "result.h"

ALICE_TASK(task);

struct alice_od *a persistent, *b persistent;
int line persistent, column persistent, size persistent;

void alice_object_init(void *arg)
{
        struct task_init *ti=(struct task_init*)arg;

        line=ti->line; column=ti->column; size=ti->size;

        a=alice_new_object(matrix, (void*)ti->size);
        ((struct matrix_op*)alice_get_object_data(a))->copy(ti->a);

        b=alice_new_object(matrix, (void*)ti->size);
        ((struct matrix_op*)alice_get_object_data(b))->copy(ti->b);
}

void alice_task_execute(void)
{
        int i;
        struct matrix_op *a_op=alice_get_object_data(a);
        struct matrix_op *b_op=alice_get_object_data(b);
        struct result r;
        struct alice_od *rod;

        r.line=line; r.column=column; r.result=0;
        for(i=0; i<size; i++) {
                r.result+=a_op->get(line,i)*b_op->get(i, column);
        }

        rod=alice_new_object(result, (void*)&r);
        alice_put_object(rod, 0);
}


ALICE_TASK_END;
```

The result (header)

```
#ifndef _RESULT_H
#define _RESULT_H

struct result {
        int line, column;
        int result;
};

#endif  /* _RESULT_H */
```

The result

```
#include <stdio.h>
#include <stdlib.h>

#define __ALICE_OBJECT__
#include <c/object.h>

#include "result.h"

ALICE_OBJECT(result);

struct result result persistent;

void alice_object_init(void *arg)
{
        struct result *r=(struct result*)arg;

        result.line=r->line; result.column=r->column;
        result.result=r->result;

        alice_set_object_data(&result);
}


void alice_object_load(void)
{
        alice_set_object_data(&result);
}
```

The matrix (header)

```
#ifndef _MATRIX_H
#define _MATRIX_H

struct matrix_op {
        void (*rand)();
        void (*copy)(struct matrix_op*);
        void (*set)(int i, int j, int f);
        int (*get)(int i, int j);
        void (*print)(void);
};


#endif /* _MATRIX_H */
```

The matrix

```
#include <stdio.h>
#include <stdlib.h>

#define __ALICE_OBJECT__
#include <c/object.h>
#include <c/platform.h>

#include "matrix.h"

ALICE_OBJECT(matrix);

struct matrix_op matrix_op;

int *data persistent, size persistent;

void matrix_rand()
{
        int i, j;
        int *_data;

        _data=pso_decode_ptr(pso, data);
        for(i=0; i<size; i++)
                for(j=0; j<size; j++)
                        _data[j+i*j]=rand()%100;
}

void matrix_copy(struct matrix_op *mop)
{
        int i, j;
        int *_data;

        _data=pso_decode_ptr(pso, data);
        for(i=0; i<size; i++)
                for(j=0; j<size; j++)
                        _data[j+size*i]=mop->get(i, j);
}

void matrix_print(void)
{
        int *_data;
        int i, j;

        _data=pso_decode_ptr(pso, data);
        for(i=0; i<size; i++) {
                for(j=0; j<size; j++)
                        printf(" %3d ", _data[j+size*i]);
                putchar('\n');
        }
}

int matrix_get(int i, int j)
{
        return pso_decode_ptr(pso, data)[j+size*i];
}

void matrix_set(int i, int j, int f)
{
        pso_decode_ptr(pso, data)[j+size*i]=f;
}

void init()
{
        matrix_op.rand=&matrix_rand;
        matrix_op.copy=&matrix_copy;
        matrix_op.print=&matrix_print;
        matrix_op.get=&matrix_get;
        matrix_op.set=&matrix_set;
        alice_set_object_data(&matrix_op);
}

void alice_object_init(void *arg)
{
        int _size=(int)arg;

        size=_size;
        data=pso_malloc(pso, size*size*sizeof(int));

        init();
}

void alice_object_load(void)
{
        init();
}

ALICE_OBJECT_END;
```

C. Heap allocator tester

```
#include <stdio.h>
#include <unistd.h>
#include <sys/times.h>
#include "pso.h"

struct list_t {
        int data;
        struct list_t *next;
};

struct list_t list_t head PERSISTENT = { 0, };
int elem PERSISTENT = 1;

void dump_tms_diffs(const char *str, struct tms *start, struct tms *stop)
{
        long clk_tck=sysconf(_SC_CLK_TCK);


        printf("%s: user time %f, system time %f\n", str,
               (float)(stop->tms_utime-start->tms_utime)/clk_tck,
               (float)(stop->tms_stime-start->tms_stime)/clk_tck);
}

void run()
{
        struct list_t *i, *new_elem;
        struct tms tms_start, tms_stop;
        int j;

        printf("%lx %d\n", pso->heap->base_correction, elem);
        pso_dump_heap(pso->heap, 0);

        times(&tms_start);
        for(i=pso_decode_ptr(pso, head.next), j=elem-1; i != NULL;
          i = pso_decode_ptr(pso, i->next), j--)
                if (i->data != j) {
                        printf("BOOM %p %d %d!!!\n", i, i->data, j);
                        return;
                }

        times(&tms_stop);
        dump_tms_diffs("follow", &tms_start, &tms_stop);

        times(&tms_start);
        for(j=0; j<500000; j++) {
                new_elem=pso_malloc_decoded(pso, sizeof(struct list_t));
                new_elem->next=head.next; new_elem->data=elem++;
                head.next=pso_encode_ptr(pso, new_elem);
        }
        times(&tms_stop);
        dump_tms_diffs("append", &tms_start, &tms_stop);


        for(j=0; j<1000; j++) {
                new_elem=pso_decode_ptr(pso, head.next);
                head.next=new_elem->next;
                pso_free(pso, pso_encode_ptr(pso, new_elem));
                elem--;
        }
        return;
}
```

Bibliography

[1] Radu Niculita. *“ALiCE a Java-based Grid Computing System”*, National University of Singapore & Politehnica University of Bucharest, 2002

[2] Johan Prawira. *“ALiCE, Java-based Grid Computing System*”, Honours Thesis, School of Computing, National University of Singapore, 2001

[3] John R. Levine. *“Linkers & Loaders*”, Morgan Kaufmann Publishers, 2000

[4] The Santa Cruz Operations, AT&T. *“System V Appplication Binary Interface”, Edition 4.1*, 2001

[5] Ian Foster, Carl Kesselman. “*Globus: A Metacomputing Infrastructure Toolkit*”, International Journal of Supercomputing Applications, 1997

[6] Ian Foster, Carl Kesselman. *"The Grid: Blueprint for a New Computing Infrastructure", Chapter 2,* Morgan- Kaufman, 1999.

[7] I. Foster, C. Kesselman, S. Tuecke. “*The Anatomy of the Grid: Enabling Scalable Virtual Organizations*”, International J. Supercomputer Applications, 15(3), 2001

[8] ***, “GNU CC Reference Manual”, C extension, Variable Attributes.*

[9] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. *“Operating System Concepts”, 2nd edition, pages 285-286,* John Wiley & Sons, 2000

[10] Benjamin Chelf. *“Dynamic Memory Allocation”*, Linux Magazine, June 2001

[11] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. *“Operating System Concepts”, 2nd edition, pages 707-710,* John Wiley & Sons, 2000

[12] James C. McKim. *“Object Oriented Concepts”*, paper, 2000

[13] SETI@home: http://setiathome.ssl.berkeley.edu

[14] Distributed.Net: http://www.distributed.net

[15] Globus: http://www.globus.org

[16] The GLOBE Project: http://www.cs.vu.nl/~steen/globe/

[17] Legion: http://www.cs.virginia.edu/~legion

[18] Condor: http://www.cs.wisc.edu/condor

[19] SETI@home: http://setiathome.ssl.berkeley.edu

[20] Grid Computing Info Centre (GRID Infoware): http://www.gridcomputing.com