Jini-Grid

Ben Clifford
School of Mathematical Sciences
Queen Mary and Westfield College

Supervisors:
Scott Telford, EPCC
Martin Westhead, EPCC

Abstract

Jini is a technology that allows networks of heterogeneous services to organise themselves with little human intervention. It provides for fault tolerance and for the automatic transfer of driver code, written in Java, when and where it is needed.
These abilities will be extremely useful, if not essential, in a large scale environment such as the Grid.
In this project, Jini technology is used to share compute servers using the task farm paradigm.
A standard API that all compute servers must implement is defined. Four implementations with different capabilities are presented as well as a selection of clients and helper classes.

2 Introduction

This project aims to use Jini and Java to build a small prototype Grid.

The Grid is a concept that will provide the ability for a user to use some high performance computing resource without knowing exactly where or what it is. One part of The Grid will be the use of remote computation servers, selected either by the user or by some automatic process, based for example on the amount of free processor time or on cost.

Task farms have been chosen as the execution model for this project, as they are quite independent of the underlying parallel architecture and hence are suitable for deployment over a wide range of architectures.

Research has already taken place within EPCC on task farms in Java, and it is upon a previous EPCC-SSP report that I have based my work.[1] That project implemented HPT (Hitachi Parallel Tasks), a remotely accessible task farm system, intended to run Java code primarily on the Hitachi SR2201 using MPI but also capable of running on other MPI systems and on shared memory systems. One of the ideas behind the design was that a client should be able to run with minimal code modification on a workstation for initial development, on larger systems for testing and finally on the Hitachi machine for real-world usage. JiniGrid attempts to extend this to create a Grid.

Java provides cross-platform compatibility between a wide variety of platforms, by providing abstractions of common system facilities such as the file-system, as well as techniques for transferring code and data between disparate systems.

Jini is a collection of classes and concepts that enables clients and services running on a network to discover each other and provides standard protocols for the clients and services to interact once they have done so. It makes heavy use of Java features such as object serialization, remote method invocation (RMI) and bytecode portability.

New clients and servers can be added to a network and they will automatically detect each other and be able to interact, without the need to install new drivers.

Jini provides mechanisms for fault tolerance. Programs must *lease* resources from services - if they die or are disconnected, the lease will eventually expire and the resources will be freed. This stops the various services “filling up” with useless data.

3 A Standard API

All servers must provide a standard API, so that they all appear identical to clients. JiniGrid servers must provide a service object which implements the `TaskFarmService` interface. The API classes, stored in the jinigrid.spec package, are the only classes that all clients and servers must have in common.

The service object will be transmitted to the lookup service and then on to interested clients using object serialization. It should pass on calls to the compute server. It may do this in any way that it pleases to – often RMI is used for this purpose as no explicit coding is necessary, but the object is free to use whatever means it cares to.

The TaskFarmService object will produce on demand another object which implements the TaskFarm interface (below). This provides methods to run a task, register to receive notification of completed or to finish and free resources.

The API also contains classes to transfer bytecode in jar files, carry information about the capabilities of servers such as the number of processors and the libraries that are available, and to symbolise progress events (sent, if required, when a certain number of tasks have been completed).

```
public interface TaskFarm
{
public Vector runTask(Vector tasks,
                               Object globalData,
                                String slaveClass)
               throws RemoteException;

public void finish() throws RemoteException;

public EventRegistration trackProgress(long duration,
     RemoteEventListener rel, MarshalledObject key,
     int reportEvery, boolean sendResults)
       throws RemoteException;
}
```

4 Servers

ServerOne

ServerOne was the first server that I implemented. It was based on the HPT remote compute server code previously written at EPCC.[1] That code needed little modification: firstly a wrapper layer was placed around it to convert the JiniGrid API calls to HPT calls, and secondly, code was inserted to register with Jini lookup services at the same time as starting up.

The API of this remote server formed the basis for the JiniGrid API, although was made more abstract.

The Server uses message passing, either with MPI or with a multi-threaded 100% Java message passing system. It can be used on any architecture that supports either Java threads over multiple processors or MPI.

The code uses the mpiJava binding of Java to MPI.

ServerQueue

ServerQueue takes task farm jobs and submits them into the job-queueing system of the EPCC Lomond service, to be executed by ServerThread.

It is split into two sections. The main section, the queue manager, runs permanent on Lomond front-end. When it receives a job request from a client, it stores it on disk and submits a Worker job into the Lomond queueing system. When this Worker is eventually run, it connects using RMI to the queue manager, retrieves the appropriate job data, spawns ServerThread to execute the job, and sends the results back to the queuemanager, which then forwards them to the client.

It should be easy to convert this to submit jobs to another queueing system or to use a different task farm for the actual execution of the code.

ServerThread

ServerThread is a thread-based server. It has a lower overhead than ServerOne, but cannot be used where threads are not supported over multiple processors (for example, Beowulf machines).

It handles internal communication using shared Vector objects, one to send tasks to the workers and one to collect the results.

It is considerably more light-weight than ServerOne, but can only be used on JVMs where threads will be run on multiple processors, with shared memory.

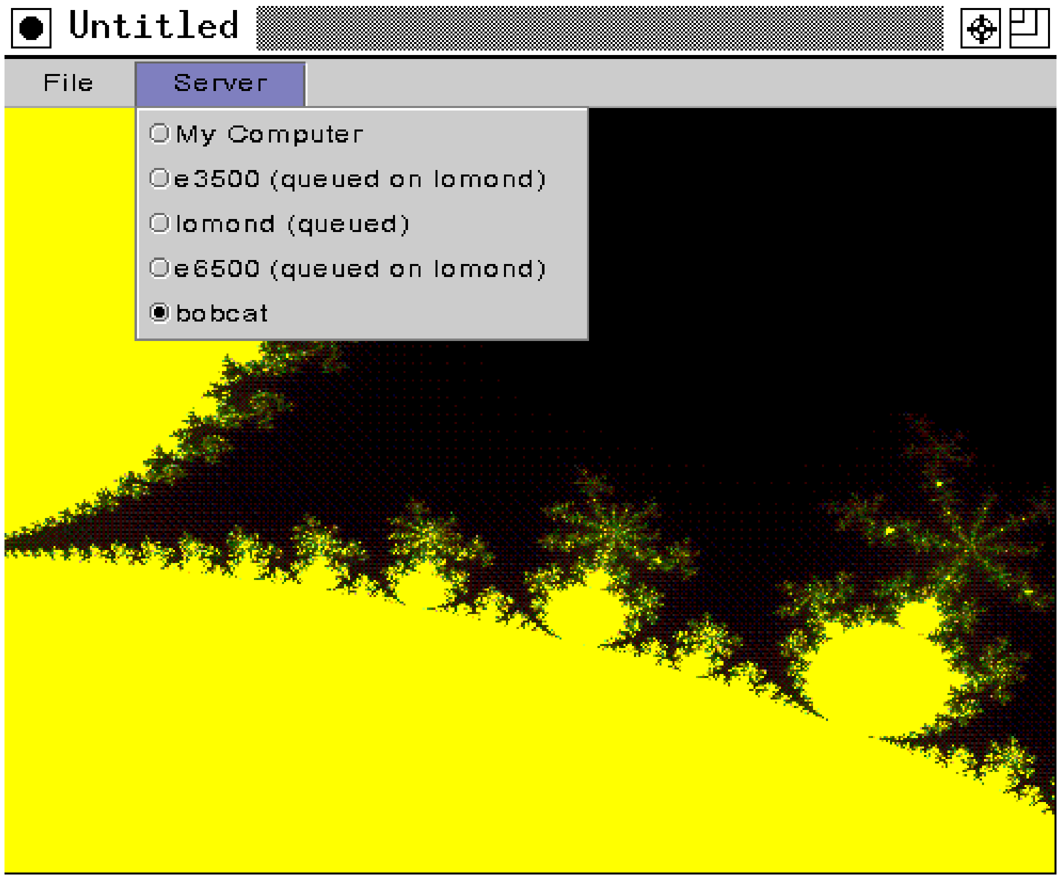
ServerMeta

ServerMeta provides a meta-server, that will take a job and distribute it amongst other available task farms. To the client, this appears exactly like a normal JiniGrid task-farm server.

Using this server, the client can take advantage of all processors on many different machines simultaneously, even if they have different architectures and are running different server implementations.

At present, tasks are distributed evenly between all systems, although some form of metric could determine the proportions between each different component system.

5 A Jini-Grid Client



○ My Computer
○ e3500 (queued on lomond)
○ lomond (queued)
○ e6500 (queued on lomond)
● bobcat

This Fractal Viewer produces a rendering of the Mandelbrot set using JiniGrid task farms.

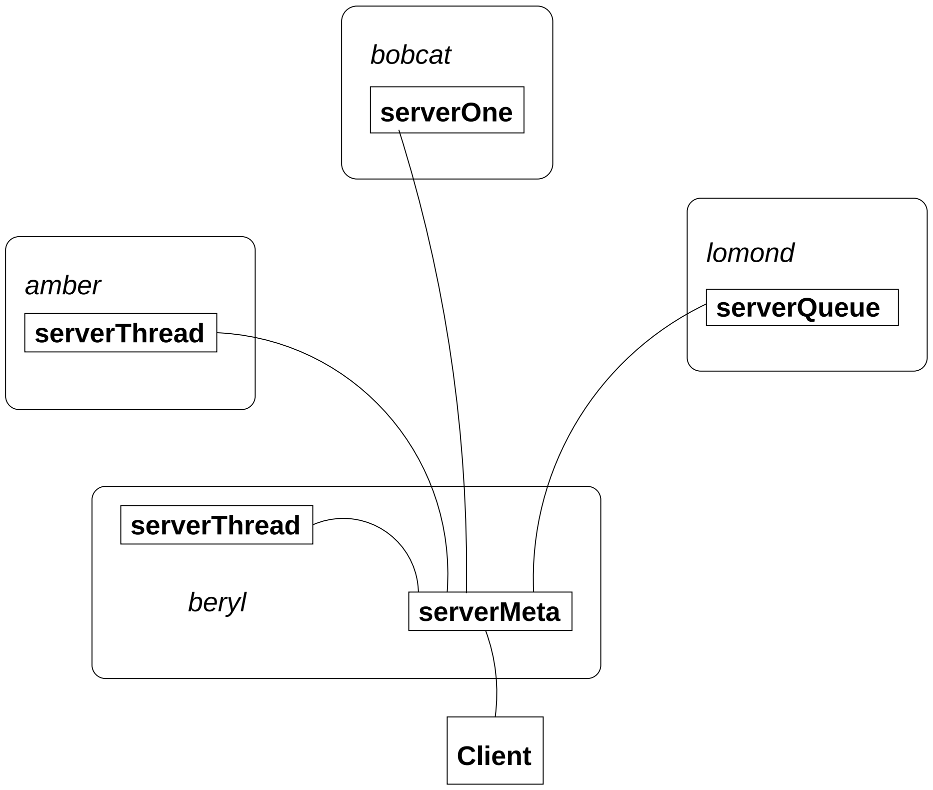
This rendering is very easy to parallelize as a task farm: The plane to be rendered can be trivially broken up into small rectangles, with each rectangle rendered as a separate task.

As each task is completed, a progress event is sent to the client and the rendered tile can be drawn on screen immediately – the on screen image gradually builds up over time.

The user may highlight any section of the image and the display will zoom in to that region and recompute.

The client presents the user with a choice of available compute servers. The user may choose any, or may choose to use the local processor. At any time during the use of the client, the user may change the selected option and from then on the newly selected server will be used.

6 A Grid At EPCC



A small Grid was constructed at EPCC from various machines.

Two HPC machines and two normal unix hosts were set up with the appropriate server software. Additionally the metaserver was run on one of the machines.

The machines have the following specifications:
BOBCAT: Beowulf-class, MPI over Ethernet, running ServerOne.
Lomond: Sun E3000, shared memory, running ServerQueue.
Amber and Beryl: Single processor Sun workstations running Server Thread.

A client could connect to any of the servers individually, or could use the metaserver on beryl to combine together a total of 22 processors over all of the machines.

The Mandelbrot client of the previous slide was used to provide a visual indication of the relative speeds of each server.

References

[1] Fernando Elson Mourão. A Middleware Environment for Parallel Java on the Hitachi SR2201. SSP report, Edinburgh Parallel Computing Centre, September 1998.