EPCC-SS98-10

A Middleware Environment for Parallel Java on the Hitachi SR2201

Fernando Elson Mourão

Abstract

Parallel programming for the masses, seamless metacomputing, are expressions that are becoming common nowadays. They offer the hope that the non specialist programmer in the non too distant future may have the opportunity to program large parallel computers without having to know too much about parallel computing itself and without having to learn some of the complex underlying concepts. In a parallel computing environment one finds that usage is strictly controlled due to the expense and demand for these machines. One needs cheaper environments where programs can be tested and developed. However on networked workstations, a potential development environment, one is exposed to heterogeneity problems. Java already largely solves this problem, but packages for parallel programming still do not exist. This project, one of the modules of a joint project between EPCC and Hitachi Europe, allows a programmer to develop a parallel code on a cluster of workstations, or even in a single computer and later run it on the Hitachi's MPP SR2201 system, without having to modify or even re-compile his code.

Date	Version	Summary
21 August 1998	v0.1	First Draft with outline and small introduction
7 September 1998	v0.2	First full version ready for revision
8 September 1998	v0.3	Second draft for revision
9 September 1998	v1.0	Pre-Release
10 September 1998	v1.1	Final version

1 Introduction

This Summer Scholarship Project is part of a collaborative Java project, between EPCC and Hitachi. It involved the use of Hitachi SR2201 distributed multiprocessor high-performance computer located at EPCC. The main aim of the project was to create and port a set of tools that would allow the execution of bytecode portable multi-threaded Java programs to run on the Hitachi SR2201.

The overall goals of the project can be further broken down to the following steps:

- Porting Sun Microsystems' Java Virtual Machine (JVM) to the SR2201.
- Implementing a set of Java bindings for SR2201's MPI [5].
- Developing a middleware runtime environment whereby an user can execute Java programs on a remote SR2201 from their own desktop client.

The last of these tasks was the one that was implemented during this Summer Scholarship Project. This report gives a brief description of the module and work concerned. The report begins with an overview of the main project, stating the motivations of the EPCC-Hitachi project followed by a general view of the design, details about the implementation and finally a brief set of conclusion that can be drawn from the project.

2 Motivations

In the past parallel computing vendors have introduced their own parallel programming libraries, that very often exposed the system's architecture and were not portable. To solve these problems several organisations and institutions got together and developed standards like MPI [5], or generic parallel programming environments such as PVM [9] so the programmer would have a portable parallel programming interface.

There still exists the problem of potentially long development times and of obtaining usage of parallel machines over this period, because the parallel programming concept is more complex and the machines are expensive implying strict usage policies and the developing tools are not as mature as on serial machines. So, most of the time programmers have no choice but to develop their programs on less expensive machines and only when the program is considered to be completed it is ported to a parallel machine. There it must be recompiled and very often adjustments still have to be made for it to run properly.

With the emergence Java [10] it has become possible to write programs that run across heterogeneous environments. However Java still does not have much support for parallel systems. It was only ot the beginning of August this year that Sun Microsystems announced its new product JINI [11] which seems to take a step towards bringing Java closer to parallel and distributed computing. The product though is still not available, and its success on the high performance computing area depends somewhat on whether Java is able to improve its performance or not. For these reasons MPP machines like the SR2201 are still not a part of the Java revolution.

The EPCC-Hitachi project's initial aim was to allow the programmer to use Java's threads library on the SR2201. Thus, when a program is run on the SR2201, any Java threads would really be parallelised and distributed through out the processors of the SR2201. This initial goals proved to be too ambitious and these have been substituted by the middleware environment.

2.1 What is the SR2201?

The Hitachi SR2201 [3] is a Massively Parallel Processors (MPP) systems that can have up to 2048 processors. Distributed memory is used. Each processor has a peak performance of 0.3 G-FLOPS. A fully configured system would thus have a peak performance of 600 G-FLOPS when using all 2048 processors. The system uses an UNIX-based operating system HI-UX/MPP. A microkernel operating system based on Mach [2] is installed on every processor, and an Unix server works on top of it on special purpose processors. EPCC's model has 8 processors with 256Mb each. The SR2201's parallel development libraries include MPI, PVM and HPF amongst others. There is no official support from Hitachi[1] for Java as yet.

2.2 Why use Java on the SR2201?

Java started off as a language strongly associated with the Internet. Its main strength: portability and the facility of allowing applets to work on a web page as small and simple applications propelled Java to the masses. Nowadays Java is being used for more than merely animating web pages.

Mobile Agents[2] [4] technology has became popular based on Java. Projects like:

- DOGMA [1] – a Java based parallel application environment that allows programs to run on both local networks of workstations as well as "anonymous computers" which can participate in the system via browsers.
- Java Grande [8] – a project forum to create standards for High Performance computing using Java.
- Java for Computational Science and Engineering [7] – research project on Java resources for computer science.
- JavaMPI [12] – one of the many Java bindings for MPI.

These, amongst others, show that the usage of Java for high performance computing is becoming a reality. A parallel Java development environment for the SR2201 thus seems to be doing no more than answering users' needs.

2.3 What users want/ do not want

If the goal is to make programming simple enough for the programmer not to have to worry about the system on which their program will run on then there is obviously a compromise to be made. The ideal situation from the user's point of view would be to program using regular Java threads and subsequently for the system to decide how to execute them. In the case of the SR2201 the threads would be redirected through as many available processors as necessary or possible. This already constitutes a problem if the programmer is not aware of the number of available processors – ideally they should not have to be. They should however not have to use something like MPI to overcome this problem. An intermediate solution would be desirable.

[1] Or at least is not far as the author knows.

[2] Agents: software entities that assist people and act on their behalf.

This was the option taken by the EPCC-Hitachi project. A parallel library that follows the Task-Farm paradigm without exposing too much of the parallelism behind it to the user was chosen to achieve this.

3 Design

The library that has been created is called *Hitachi Parallel Threads* (HPT) and consists of three deployment scenarios: Single JVM, a pure MPP and a Network Client that connects to one or more Compute Servers.

The package's main classes are: `ProxyMaster`, `SlaveDriver`, `ComputeServerStatus`, `ControlTaskRun`, `TaskRunner`, `JarData` and `Slaveable`.

From the classes mentioned above only `ProxyMaster` and `SlaveDriver` are not accessed directly by the user, they are the main classes that run in the compute server. All of the others are visible to the user.

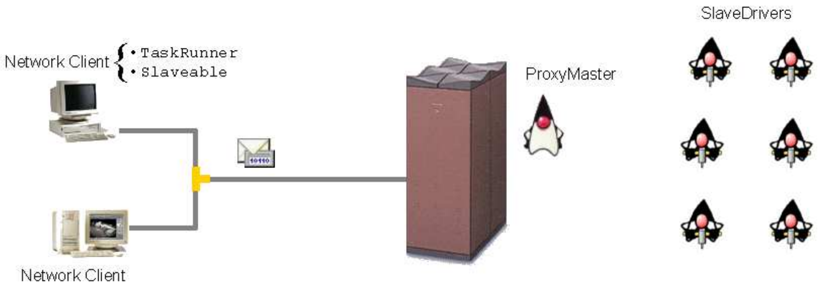

Figure 1: General architecture of the environment.

In order to write a parallel application the user programs the Network Client using `TaskRunner` and the tasks he wants to compute using the `Slaveable` interface. The task has to implement the abstract methods of `Slaveable`. The Network Client has the following structure:

- Instantiate a `TaskRunner` object which makes the connection to the server, and authenticates the client using a private key.
- Request the execution of the tasks by calling `TaskRunner`'s `runSlaveClass` method. This method takes the tasks, initialisation data, control data and a Jar archive with the class' definition and sends it to the Compute Server to be run.
- Finish the use of the Compute Server by calling the `finish` method.

Even though Figure 1 illustrates the Networked version of the HPT package, in all of the different scenarios the programmer, only has to implement his tasks and a client. He does that by extending the `Slaveable` class and the main program (called client on the scenario presented) that uses `TaskRunner`'s methods to compute the tasks. The `JarData` class loads and stores the Jar archive file with the classes to be sent to the Compute Server before a task is run. This is done automatically by the package. The user only has to instantiate an object of `JarData` and pass it to the `TaskRunner` when it requests the execution of those tasks. The `ComputeServerStatus` class is only for the client to receive information about the Compute Server. It includes the information about the actual number of slaves reserved for that client and if it has permission to use the server. The `ControlTaskRun` is used to control the tasks during the computation which, at the moment, only comprises the maximum time that one task is allowed run for.

3.1 Single JVM

Figure 2: Illustration of the Single JVM version.

This version works using a regular single processor computer. The user implements his code with the HPT package and the tasks are run as threads.

If the code is running on an SMP[3] with native thread support, the threads[4] may actually be running on different processors. This however is vendor dependent.

In this scenario the `TaskRunner` is the only active class of the package, it is responsible for launching the threads, managing them and gathering the results.

3.2 Pure MPP

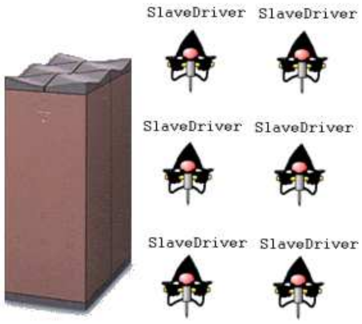

Figure 3: Illustration of the MPP version.

In this scenario all of the processors being used start and end at the same time. One of the processes, the one that calls `TaskRunner`'s methods assumes the role of master and the others are run as slaves. Once more the `SlaveDriver` will receive the tasks to be run. The tasks creation is done on the master and then sent to the slaves to run. When the tasks are all completed the master will gather the results, as is usual in a Master-Slave type paradigm.

[3]Symmetric MultiProcessing.

[4]Silicon Graphics, Sun and Microsoft have JVMs with native threads support for their SMPs.

3.3 Network Client and Compute Server (the SSP project)

The Compute Servers and the Network Clients can be anywhere as long as they are connected through a TCP/IP network. The Network Client program connects to the Compute Server and sends the tasks to be run, when they are finished the results are sent back to the client.

The `ProxyMaster` class manages the Compute Server and the requests of the clients. The `SlaveDriver` is the class that works as a slave of `ProxyMaster` receiving the tasks and running them. The `TaskRunner` class makes the connection between the Network Client and the Compute Server, logging and authenticating the client on the Compute Server, sending the tasks and receiving the results. All of this is done using a direct socket connection to the ProxyMaster.

3.3.1 Dynamic Class Loading

What makes the migration of the tasks from the Network Client to the Compute Server possible is Java's ability to dynamically load classes at runtime, and the possibility of doing it from anywhere as long as all classes related to the one loaded from a given place are loaded using the same class loader. This feature was used in this project in conjunction with Sun's Jar archive system which stores and compresses files into one file. The classes needed to execute the task have to be stored in a Jar file and it is given as a parameter using the `JarData` class. This file has to be generated by the user before the execution of his program. The user therefore has to be familiar with the usage of Jar archives.

When the the Network Client sends a request to run a task first the Jar file is sent to the Compute Server, and only after the actual task objects are sent in a separate message. In this way the classes are loaded automatically when the message is received. At the moment the jar files are stored in a given directory as there is no means to reuse classes that are stored on the Compute Server.

3.3.2 Security

For the user to be able to access the Compute Server an "account" has to be set up and a file containing a private key has to be stored on the user's client machine. This file is used to sign the initialisation message that is sent to the server. It is assumed that this file is handled with the same precautions as any access key to a secure system.

These were the major design problems that had to be solved, its implementation details are described next.

4 Implementation

Only the main implementation issues are referred to in this section, for further details see Appendix A on page 9. The package called `HPT` is the main package. For the different ways of communicating between the master and the slaves in each scenario a different package containing a class called `CommsWrapper`, that extends HPT's `CommsWrapper00` class, is used. The methods used for the communications between the master and the slaves are the same for every different scenario. The different ways of communicating are done through pipes for the Single JVM scenario, and MPI for the other two scenarios. At the moment only the piped communications wrapper is coded. It is placed on a package called `HPT.Pipes`. For each of the targets, only the package has to be changed on the `import` section of some of the classes. Bear in mind that this is not a concern for the final user of the package, this was an option to simplify implementation and maintenance of the code. The loading of the classes was done when the tasks were being read from the sender.

4.1 Dynamic Class Loading

This was achieved using the ability to extend object streams, so when an object is being received and the correspondent class has not been loaded, instead of using the primordial class loader, another one can be used. This allows the use of a class loader that reads the class information from a Jar file previously sent by the Network Client. The `HptObjectInputStream` is the class that overloads the original `ObjectInputStream` thus using the class `JarClassLoader` it reads the task classes from the Jar file when needed. Dynamically loading classes might cause security problems so the users must be authenticated in order to avoid foreign intrusions on the system.

4.2 Security

The security system implemented is based on a digital signature using a public key. On the initialisation message the class `ClientTicket` encapsulates this signature. It has some information about the client signed with the client's private key. When the `ProxyMaster` receives the message it uses the client's public key to verify the signature. The keys are stored and accessed using the `UsersManager` class.

5 Conclusion

The goals of this SSP project were achieved. A fully working version of the class libraries is now available. A simple example that calculates PI has been provided, as a means to test the environment's functionality. The only task of the initial set of goals of the project which was not done was a GUI[5] to monitor the Compute Server status. This was due to some problems in obtaining information from the Compute Server. The `ComputeServerStatus` class, from which the GUI would get the information, is at an early stage of development and necessitates further improvements before this could be done.

This library will potentially allow a softer entry into parallel programming. Based on the Task-Farm paradigm it obviously fits better certain kinds of problems. As is broadly known Java is not best suited for high performance computing, nevertheless efforts are being made to change this. With this library the programmer can develop and test his parallel programs without accessing a parallel computer. Only when bigger problems or faster results are needed a parallel computer has to be used, thus resulting in better resource usage. The library itself is simple to use because it only requires a small set of classes and follows the object oriented concept. It is simple

[5]Graphical User Interface.

enough, for someone acquainted with Java, to program their tasks in a separate object as a worker which receives data and computes it and a master that does the overall coordination.

There are, of course, improvements to the library that can be made at several levels.

Jars caching – the tasks' classes archived on a Jar file are stored on the Compute Server. As the programs are to be used more than once the use of the previously stored classes could improve performance.

Authentication – the authentication is done through a digital signature on initialisation. Security could be improved if a full encryption of the messages were to be performed.

Sandbox models – restricted usage policies, like disk quotas and running times are useful in this kind of environment as mention before.

Accounting – for statistical purposes an accounting system would help the managing of the resources.

GUIs – the monitor GUI previously mentioned still has to be implemented. A GUI interface for the user creation an management would also simplify the administrator's work in user managing.

A The Packages and its Classes

A.1 package HPT.Pipes

This package is the pipes' implementation of the `CommsWrapper` class.

A.2 package HPT

This is the main package where almost all of the classes are. A small description of each of the classes follows.

A.2.1 *Classes specific for communications:*

`MessageConstants` -Interface used to hold constants for the messages' identifier.

`Message` - Base class to build messages to be exchanged between the Network Client and the ProxyMaster and between ProxyMaster and the SlaveDrivers.

All of the following are subclasses of `Message`. There are two sets of messages, the ones sent between the Network Client and the ProxyMaster and the ones sent between the ProxyMaster and the SlaveDrivers.

Messages exchanged between the Network Client and ProxyMaster:

`CMinitMessage` - The initialisation message to be sent to the ProxyMaster. Contains the number of slaves the client requests to use, and the authentication class `ClientTicket`.

`CMinitMessageReply` - The initialisation message reply to be sent from the ProxyMaster to the Network Client. Returns the compute server's status with the number of slaves allocated for this client, the permission and load information in string format at the moment.

`CMjarMessage` - This is the jar message with the classes to be instantiated that is sent from the Network Client to the ProxyMaster. It is sent before the request to run the tasks so the classes can be loaded when the actual tasks are received.

`CMrunSlaveMessage` - The message with the tasks, its initialisation, and control for the server to run.

`CMresultMessage` - This is the result message with the results from the computation that is sent back from the ProxyMaster to the Network Client.

`CMdieMessage` - This is the message that tells the ProxyMaster that the Client has finished.

Messages exchanged between the ProxyMaster and the SlaveDrivers:

`MSclassNameMsg` - This message is for setting the name of the slave class with which to instantiate and execute.

`MSinitAllTasksMsg` - Initialisation data common to all tasks that is broadcasted in the beginning, when a new set of tasks is to be run.

`MStaskMsg` - This contains a given task to be computed by the slave.

MScontrolTaskMsg - Control information to be used by the `SlaveDriver` class with information like the maximum amount of time that one task can run.

MSresultMsg - Message returning the result of a completed task sent from the SlaveDriver to the ProxyMaster.

MSkillSlaveMsg - This message instructs the SlaveDriver to exit from further computations.

A.2.2 *Miscellaneous purpose classes:*

ClientTicket - Class that has the client's information to be granted permission to use the parallel machine.

Dbg - this class is used for debugging purposes only.

CommGroup - Class that stores the group of slaves in a communications group. Also used to distinguish the slaves that belong to a client.

CommsWrapper00 - Base class for wrappers around communication routines. Variations for MPI and I/O pipes would be extended from this class. Uses the class/structure `ObjectOfint` from *JavaMPI*. Fully implemented by Hon Yau.

ComputeServer - This class is the class that starts the compute server. Initialises the system and becomes either a ProxyMaster or a slave This 1st version works as serial and the slaves are threads.

GenerateKeys - The class that generates the encryption key pairs for the authentication. Solely used by the `UsersManager` class.

GenericClassLoader - A generic abstract class loader to be used for loading from multiple sources. Must be subclassed and the abstract method `loadClassRawData()` implemented to provide the raw data of the class.

JarClassLoader - Subclass of `GenericClassLoader` for loading classes from a Jar file.

HptByteArrayOutputStream - New version of `ByteArrayOutputStream` class, where the user can specify the internal buffer to use, and where the user can obtain a reference to that buffer instead of via memory copies. The prefix 'Hpt' is a working title. Fully implemented by Hon Yau.

HptDefaults - Static class with all the common default values for the HPT package.

HptObjectInputStream - Subclass of `ObjectInputStream`. This makes possible for the classes to be loaded automatically by the JVM, using the `JarClassLoader`, when the messages containing the tasks are read trough the object stream.

ProxyMaster - One of the key classes of the third scenario. This is the class that acts as the interface between the Network Client and the Compute Server. Receives the requests from the Client, and puts the slaves running on the parallel machine.

UsersManager - Class that manages the list of users and respective public keys. It is used by `ProxyMaster` to retrieve a user's public key. It is also used by the `CreateUser` class to create a new user and its the key pairs or to erase a user from the users list.

ServeClientState - Class used only by the `ServeClient` class to check the Network Client for consistency on its current state.

ServeClient - Threaded class that `ProxyMaster` launches each time a new client connects to it. This is the class that actually serves as proxy.

SlavesList - Class which maintains a list containing Slaves and their known state. Used to manage how tasks are assigned to each Slave and to each client by using `CommGroup`.

SlaveStatus - Class that stores the slave's current status. Either it is free to be used or it is busy executing a task. Used only by `SlavesList`.

ServerInfo - Class that gathers information of the Server. At the moment executes the UNIX system command `w` to get system info, and stores the number of Network Clients currently using the ComputeServer.

SlaveDriver - Class for instantiating and running a slave class. Each instance runs on one of the nodes and receives orders from `ProxyMaster`, using `CommsWrapper`.

TaskRunner00 - Base class, for running Slave objects amongst a given set of Slave Drivers. `TaskRunner` subclasses this one. Fully implemented by Hon Yau.

A.2.3 *Exceptions*

These are the exception classes used inside the package:

ClientTicketException - Error exceptions thrown by the `ClientTicketClass`.

CommsException - Error exception, for communication calls. With optional field for selecting whether to print debug information via a stack trace.

UsersManagerException - Error exceptions thrown by the `UsersManager`.

A.2.4 *User classes:*

These are the package's classes that the user has access to and has to use thus a more detailed description of them will be given:

TaskRunner - Main class the programmer uses for communicating with the Compute Servers. The program has to start by instantiating a `TaskRunner` object were the constructor's call is the following:

```
public TaskRunner(String computeServerName,
                  String userName,
                  String privateKeyFilename,
                  int nSlaves,
                  ComputeServerStatus csState)
        throws TaskRunnerException
```

Its parameters' meaning are:

`computeServerName` - The hostname of the parallel machine where the ProxyMaster is.

`username` - The user's name for the authentication ticket.

`privateKeyFilename` - the private key's file name.

`nSlaves` - number of slaves the client wants to use, which may not be the number of slaves that it will get.

`csState` - returns the current state of the parallel machine and includes the actual number of slaves the client is allowed to use.

After that, and if initialisation was successful the program may call the `runSlaveClass` method, as many times as needed:

```
public Vector runSlaveClass(Vector tasks,
                            String slaveClassName,
                            Object initAllTasks,
                            Vector taskControls,
                            JarData jar)
              throws TaskRunnerException
```

Its parameters' meaning are:

`tasks` - A `Vector` containing the tasks.

`slaveClassName` - The name of the starting slave class. This class must implement the `Runnable` interface.

`initAllTasks` - Data or work common to all tasks.

`taskControls` - Vector of `ControlTaskRun` objects, for controlling the execution of each task. Must be of the same length as the task vector.

`jar` - The Jar archive object to send to the Compute Server.

And it returns a `Vector` with the results of the computation. When the program finishes using the Compute Server it must call the `finish` method:

```
public void finish(String computeServerName)
            throws TaskRunnerException
```

Which receives a string, `computeServerName` with the name of the Compute Server also used on the constructor.

`TaskRunnerException` - this exception is only used for information purposes, the program can print the information to know more about the problem that caused it. Or it can be cached so the program can make a graceful exit. Its usage is exactly the same as any exception of Java.

`JarData` - This class is for encapsulation of the Jar file. The user instantiates an object and uses the object as a parameter of `TaskRunner`'s `runSlaveClass`. It is of no other use to the client program. So the user only has to know its constructor:

```
public JarData(String filename, String mainClass) throws
IOException
```

Its parameters are:

`filename` - the jar file to be stored.

`mainClass` - the main class' name to be used for instantiation of that class on the Server.

`Slaveable` - Abstract class for the user to implement its tasks so the Compute Server can access their methods. Fully implemented by Hon Yau. The methods the user has to implement are:

`public Slaveable()` - the constructor.

`public abstract void setInitAllTasks( Object initAllTasks)` - common data or work to all slaves.

`public abstract void setTask(Object task)` - data or work specific to that task.

`public abstract void run()` - method to run the task.

`public abstract Object returnResult()` - method trough which the SlaveDriver gets the result from the task to send back to the ProxyMaster.

`ControlTaskRun` - The class to control the task. At the moment it only has a timeout. The user only needs to use the constructor:

```
public ControlTaskRun(String header, int timeout)
```

The `header` string is to store some extra information about the control object, it is for future use. The `timeout` value is the maximum amount of time a task is allowed to run.

`ComputeServerStatus` - Stores information about the compute server. At the moment only has the number of slaves allocated to the client, the permission flag for the client to check if it was allowed to use the server or not, the number of clients using the server and a string with the load of the Server. The user only accesses the get methods:

`public boolean getPermission()` - Returns `true` if the user is allowed to use the server, `false` otherwise.

`public int getNrSlaves()` - Returns the number of slaves allocated to the client.

`public int getNrClients()` - Returns the number of clients using the Compute Server.

`public String getServerLoadInfo()` - Returns a string with information about the load of the Compute Server.

References

[1] Brigham Young University. *DOGMA- Distributed Object Group MetaComputing Architecture*. For more information see: `http://ccc.cs.byu.edu/DOGMA/`.

[2] Carnegie Mellon University. *Mach Operating System* . For more information see: texttthttp://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html.

[3] Hitachi, Ld. *HITACHI SR2201 Massively Parallel Processor*. For more information see: `http://www.hitachi.co.jp/Prod/comp/hpc/eng/sr1.html`.

[4] IBM- International Business Machines Corporation. *IBM Intelligent Agents Home Page*. For more information see: `http://www.networking.ibm.com/iag/iaghome.html`.

[5] MPI Forum. *MPI- Message Passing Interface*. For more information see: `http://www.mpi-forum.org/`.

[6] Patrick Naughton. *The Java HandBook*. Osborne McGraw-Hill, 1996.

[7] NPAC- Northeast Parallel Architectures Center. *Java for Scientific Computing*. For more information see: `http://www.npac.syr.edu/projects/javaforcse/`.

[8] NPAC- Northeast Parallel Architectures Center. *Java Grande Forum*. For more information see: `http://www.javagrande.org/`.

[9] Oak Ridge National Laboratory and the University of Tennessee. *PVM- Parallel Virtual Machine*. For more information see: `http://www.epm.ornl.gov/pvm/pvm_home.html`.

[10] Sun Microsystems, Inc. *Java*, 1998. Fore more information see: `http://www.javasoft.com/`.

[11] Sun Microsystems, Inc. *Jini*, 1998. Fore more information see: `http://www.javasoft.com/products/jini/`.

[12] University of Westminster. *JavaMPI - a Java binding for MPI*. For more information see: `http://perun.hscs.wmin.ac.uk/`.

[13] Hon W. Yau. Design for the Execution of Parallel Threads on the Hitachi SR2201 MPP. EPCC-Hitachi Project Internal.

Acknowledgments:

I would like to thank Hon, my supervisor, for his time and patience supervising me during this project. A special thanks to Mario Antonioletti that took Hon's place on the last week. I would also want to thank EPCC for this great opportunity and all of its staff for their sympathy and friendship.

Fernando Elson Mourão
Informatics Engineering student at the University of Coimbra.
e-mail: elson@dsg.dei.uc.pt

Supervisor:

Dr. Hon W. Yau
Application Consultant at EPCC
Degree in physics from Manchester University. Ph.D in theoretical physics from Edinburgh University.
e-mail: H.Yau@epcc.ed.ac.uk