Dynamic Management and Execution of Parallel Algorithms on a Java Multicomputer

PAUL SAGE AND PETER MILLIGAN
School of Computer Science,
The Queen's University of Belfast, Belfast BT7 1NN,
UNITED KINGDOM

Abstract: - It is clear that writing software for parallel architectures is a non-trivial process. This has encouraged much research in an effort to provide tools to assist parallel software development. However, while these tools may cater for architecture-specific problems, they do little for the concept of parallel software engineering, as the end product is usually neither scaleable nor portable. The introduction of a level of abstraction in the expression of parallel algorithms can elevate the reasoning process above architectural constraints and assist the production of more flexible code. This paper outlines an object-oriented parallel algorithm development paradigm based on a Task and Channel notation, and examines the utilization of Java technologies in the development of a distributed Java Virtual Machine architecture on which algorithms expressed in this notation may be executed and managed dynamically.

Key-Words:Java Parallel Distributed Brokerage Agents *CSCC'99 Proceedings*, Pages:3281-3286

1 Introduction

Producing parallel programs, either by the migration of existing sequential code or by developing new code, is an inherently difficult process made mode complicated by the variety of target architectures available. The rapid developments in networking technologies and the emergence of Java as a widely accepted viable cross-platform language and architecture definition, have clearly contributed to the convergence of parallel and distributed computing with Internet and multimedia technologies. While these developments represent significant and exciting advances, the added complexity involved compounds the problem of producing modular, scaleable and efficient parallel and distributed algorithms that can accommodate the current requirements and retain a respectable shelf life.

In the early days of Computer Science, programmers were restricted to using assembly language or machine code for software development, requiring specialised skills, which were possessed by very few people. The abstract art of sequential programming, as understood by contemporary software engineers, was facilitated through the introduction of compiler tools and high-level programming languages. Sequential programming techniques enable computational problems to be structured, through abstract reasoning, in terms of sequences of operations (selection, repetition, statement, and function) without reference to specific architectural issues. Code written to solve a particular problem could then be easily ported to any architecture that had the required compiler and runtime support.

Thus it can be argued that the development and expression of parallel algorithms can be significantly simplified by adopting a similar abstract approach. The desire for a unified methodology for parallel software engineering is expressed in the work of Milligan [1] and Foster [2]. Milligan proposed a methodology for the description of parallel algorithms that exploited the work of Jackson [3] and expressed the resulting designs in Hoare's CSP [4]. Foster describes the application of Task and asynchronous Channel abstractions in the expression of parallel algorithms, together with the definition of an imaginary parallel architecture on which these algorithms can execute.

Java technologies from Sun Microsystems [2], which was originally developed for embedded systems, has capitalized on the popularity of the Internet to gain increasing attention from distributed application developers. Java TM consists of a programming language, a range of tools to support software development called the Java TM

Development Kit (JDK), and shared-memory architecture definition, the Java TM Virtual Machine (JVM) providing runtime support.

Through its widespread availability (in the form of JVM implementations within all popular web browsers and in JDK's for all popular platforms) Java encourages homogeneity from distinctly heterogeneous components. This, together with its in-built ability to dynamically deliver and execute software components, introduces the potential for the development of a dynamically modifiable parallel architecture using Java technologies. This paper examines the development of Java parallel algorithms using a task and asynchronous channel representation and their dynamic distribution and execution on a parallel architecture based on a number of interconnected Java Virtual Machines.

2 The Programming Model

To provide a suitable abstract parallel programming paradigm the problem reasoning process must be clearly separated from system scheduling activity. The Task and Channel model is well suited to the abstract development of parallel algorithms as Tasks are easily mapped to processors, the algorithm under development need not be concerned with the number of available processors, or their configuration. Tasks in a program may reside on the same or separate processors, and the Task abstraction upholds the concept of modular development, with similarities with the Object-Oriented programming paradigm.

In this computational model a parallel program is defined to be a set of one or more concurrent Tasks, the number of which can vary during execution. A Task encapsulates a sequence of program statements defining sequential activity, and a set of ports or Channels that define its interface to its environment. A Channel is a message queue which enables communication between cooperating Tasks through send and receive operations. The send operation is considered to asynchronous, without delaying the sending Task. The receive operation, on the other hand, is synchronous causing the receiving Task to block until data is available. In addition to being able to send and receive messages, a Task is able to dynamically create and terminate other Tasks. Dynamic connectivity is facilitated through the ability to send channel information as messages between Tasks.

2.1 Task Implementation

In this study the Task abstraction is implemented as a Java class, which inherits functionality from the java.thread.Thread class, defined within the JDK.

Instances of the Thread class (or subclasses) may be created and managed as pseudo-concurrent processes within a Java Virtual Machine. The Task class is the basic building block for a parallel algorithm and represents a process with access to its own local memory. As Java is an Object-Oriented programming language the class mechanism is ideal for encapsulating the concept of a Task, providing information hiding which promotes the notion of discrete behaviour and locality. The inheritance mechanism enables the definition of generic Task state and behaviour governing interaction with the architecture operating system and other Tasks within the program through the Channel abstraction.

The realities of providing connectivity and the need for efficient scheduling is managed exclusively within the definition of class Task and its interaction with the architecture operating system. This leaves the programmer free to express a problem in terms of the task and channel abstractions provided without initial reference to hardware considerations. Furthermore, the class encapsulation mechanism provides a means of insulating a parallel algorithm implementation from inevitable platform developments. While, at present, the system implementation accommodates workstation cluster topologies, it is only a matter of time before the Java TM Virtual Machine can make efficient use of tightly coupled multiprocessor platforms. The Task and Channel paradigm can remain consistent as far as the programmer is concerned with the underlying implementation tuned to suit the runtime environment.

The implementation of a given parallel algorithm involves creating extensions of the Task class and defining associated behaviour by implementing a specific run() method. This defines the entry point for the execution of any Java thread. When an instance of a Task is created its constructor method is automatically called by the system. The constructor of class Task interacts with the host operating system to register its existence. Each new Task created, and subsequently registered, is assigned a unique identifier. When a Task is created (as part of the execution of some other Task) there is no guarantee that it will execute within the same Java Virtual Machine as its parent Task. It is left to the architecture runtime environment to determine when and where the Task

will execute. Therefore, any object handles which are obtained as a result of instance creation cannot normally be used directly for inter-task communication. Object handles may, however, be used to obtain system information relating to a Task such as the assigned process identifier.

2.2 Channel Implementation

The channel abstraction, which is used to provide asynchronous communication between task instances, is implemented by class Channel.

```
class Master extends Task
{  public Master() { … }
   // Master constructor

   public void run()
   {  Slave s = new Slave();
      Channel c1 = requestChannel(s);
      AsyOutStream aos =
c1.getOutputStream();
      // code representing Master task
behaviour
    }
}
class Slave extends Task
{   public Slave() { … } // Slave task
constructor
    public void run()
    {   Channel c2 = acceptChannel() ;
        DataInputStream dis =
c2.getInputStream() ;
        // code representing Slave task
behaviour
    }
}
```

The above code fragments illustrate the steps necessary to establish channel communication between two tasks. When an in-stance of class Master is created and begins executing, it spawns another task, of type Slave. The Master task then requests the creation of a communications channel with the Slave instance by calling the requestChannel() Task method. This method takes one argument, which is a reference to the Slave object. It should be noted, however, that this reference is not used for direct communication. Rather, it is used to obtain the system-assigned process identifier and it is this unique identifier that is then used by the system to establish a connection, ultimately returned to the caller as a Channel object reference.

The runtime behaviour of the Slave instance reflects that of the Master through a call to the acceptChannel() method. This ver-sion of the method, which has no parameters, blocks until a request is made for connection (from any source), returning a reference to a Channel object once established. Potential communication problems, e.g. starvation and deadlock are handled by the Java synchronization mechanism. Bi-directional data streams may then be created in both Master and Slave tasks in the following way:

```
// Master :
AsyOutputStream aos = c1.getOutputStream();

// Slave :
DataInputStream dis = c2.getInputStream();
```

In the example above ***aos*** is an instance of the class AsyOutputStream, which may be used to write data of primitive type in an asynchronous fashion, and ***dis*** is an instance of the java.io class DataInputStream from which primitive data items are read synchronously, blocking the reading task until data is available. The following diagram illustrates the relationships between class definitions and object instances.

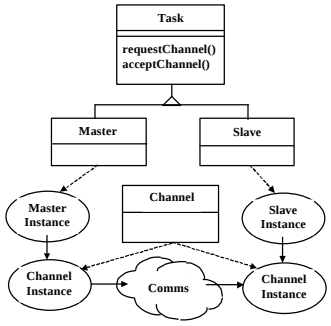

Fig. 1. Task and Channel Relationships.

3 Architecture and Runtime Support

Parallel algorithms developed using the Task and Channel paradigm described above are dynamically distributed and executed on an architecture based on the concept of the Multicomputer [5]. The Multicomputer is constructed from a number of von Neumann computers called nodes linked by an interconnection network. Each node is capable of executing one or more sequential processes operating on local memory. In addition to reading and writing

local memory, each executing process may interact with the operating system to establish message passing connections with other processes within the system. Access to local memory is less expensive than access to remote (within other tasks) memory. Consequently, read and write operations are less expensive than send and receive, the metrics of which depend on the speed of the network, the processor and associated software. The Multicomputer is very similar to a distributed-memory MIMD architecture.

For this study the Multicomputer model is translated to a cluster of Pentium-based PC's networked using standard Ethernet. Each PC is equipped with its own Java TM Virtual Machine and associated system software with deals with the scheduling and management of Tasks.

In reality, a systems software object called the Process Management Agent or PMA governs the distribution of processes between participating processor nodes. The role of the PMA is to interact with the user accommodating execution requests, to manage the creation and dynamic distribution of processes, and to maintain a systems information registry describing the state of an executing algorithm.

In this study, the decision governing where a Task is placed, is made on the basis of a simple load balancing algorithm. It is envisaged that later versions of the PMA will be able to make scheduling decisions based on support from a Knowledge-Based System containing rules associating Task characterisation data with node characterization data. In this way the PMA can assume the role of an "Intelligent Agent".

The PMA has dominion over a number of Process Execution Agents (PEA's) one of which resides on each of the Java TM Virtual Machines participating in the resource. Under the direction of the PMA a given PEA can take delivery of a binary file representing a specific task, create an instance of the task, and schedule its execution. This ability to dynamically load class data and create instances is facilitated through the Java TM class loader mechanism, found in java.system.ClassLoader, employed to a great degree in web browsers which support Java TM Applet technology. Each PEA has management responsibility over processes which have been sent to it, covering binary file downloading and process creation, process registration, process suspension and restarting, and process termination and deletion

The architecture PMA maintains a model of an executing algorithm representing a snapshot of task activity at any given time. This information is held within a Registry object containing details on each task and active channel communications. Instances of the system class TaskData describe individual tasks, the details of which include:

- ***TaskID***: a unique, system-assigned identifier for each task,
- ***Current Status***: loaded, executed, suspended, terminated,
- ***ClassName***: the filename of the Java TM class which was used to create the task instance,
- **HostIP**: the IP address of the host on which the task is executing.

The next section describes the construction, distribution and execution of a simple parallel algorithm on the Java Multicomputer architecture implementation.

4 Example : Mandelbrot Set Generator

The Mandelbrot Set [4], discovered by Benoit B. Mandelbrot who coined the name "fractal" in 1975 from the Latin fractus or "to break", is defined to be the set of all complex numbers c such that:

$$|z[N]| < 2 \qquad (1)$$

for arbitrarily large values of N, where

$$z[0] = 0 \qquad (2)$$

$$z[n+1] = z[n]^2 + c \qquad (3)$$

The Mandelbrot set is usually displayed as an Argand diagram, giving each point a colour which depends on the largest N for which | z[N] | < 2, up to some maximum N which is used for the points in the set (for which N is infinite). These points are tradition-ally coloured black. The Mandelbrot set is the best known example of a fractal, which includes smaller versions of itself and can be explored to arbitrary levels of detail. The image generated through the computation of the Mandelbrot Set is represented by a 2D integer array, with individual array elements defining a single pixel value. As the calculation of an individual pixel value has no bearing on the calculation of other values

in the grid, Mandelbrot generation is a completely parallel (or “embarrassingly” parallel) computation.

A simple SPMD computation of the Mandelbrot Set involves partitioning the target grid representing the image into four smaller arrays. In the example below the overall grid size of 400 x 400 pixels is divided into four areas, each of size 200 x 200. The pixel data for each area is computed by a separate Task, called a Slave. The overall process is controlled by a Master task which:

1. Creates the Slave Tasks, assigning each a portion of the overall grid, and
2. Collects data from each of the Slaves Tasks updating the display.

Each Slave Task has a copy of a method called computePoint() which can calculate the value for a pixel depending on its position. Once a Channel is established between the Master Task and a Slave Task, the Slave opens both a synchronous input stream and an asynchronous output stream.

The Master transmits partitioning information to each Slave, which then proceeds to calculate and return pixel data over the area defined. The runtime behaviour of the Master is defined by the following code:

```
class MasterHelper implements Runnable
{  Slave s ;
   int size = 200 ;
   int xstart, ystart ;
   Display disp ;
   public MasterHelper(int x, int y, Display
d)
   {  xstart=x ; ystart=y ;
      // set grid partition
      disp = d ;
      // reference to graphical display
   }


   public void run()
   {  s = new Slave() ;
      // create a slave task
      Channel c = requestChannel(s) ;
      AsyOutStream aos= c.getOutputStream();
      DataInputStream dis =
                    c.getInputStream() ;
      aos.writeInt(xstart) ;
      aos.writeInt(ystart) ;
      aos.writeInt(size) ;
      for(int I=xstart;I<xstart+size;I++)
         for(int y=ystart;y<ystart+size;y++)
           disp.setPixelValue(dis.readInt());
   }
}

class Master extends Task
{  private Display d;
   public Master()
   {  d= new Display() ;
   }
   public void run()
   { (new MasterHelper(0,0,d)).start() ;
     (new MasterHelper(200,0,d)).start() ;
     (new MasterHelper(0,200,d)).start() ;
     (new MasterHelper(200,200,d)).start() ;
   }
}
```

The Mandelbrot algorithm was tested on a Java Multicomputer architecture configured with up to four Pentium 166 PC’s, each with 32 MB of RAM. The following table highlights the execution results for the Manelbrot algorithm with infinity set to 4 and the number of iterations set to 20, for various architecture configurations.

Table 1. Mandelbrot code execution times for various processor combinations.

Processor/Task Configuration	*Time (seconds)*
One Processor & four Tasks	4.93
Two Processors & four Tasks	3.05
Four Processors & four Tasks	2.00

4 Conclusions

From its initial launch in May 1995 Java TM has increased in popularity within the academic and industrial communities, both as a teaching language and a viable development platform. The strength of Java, and, perhaps its weakness, is closely related to its widespread availability on many platforms. Java is viewed by many as an enabling technology which will change the nature of computing, moving away from the ownership and usage of physical hardware towards the provision of access to services. However, for the developer, programming in Java can raise as many problems as it solves. Rapid developments in Java technologies have produced variant JVM specifications, which can potentially lead to compatibility problems.

That aside, Java provides excellent library support for concurrency (within a single Java Virtual Machine), for socket-based network programming, and graphical user interface (GUI) development. This enables the construction of functionally intricate distributed applications with relative ease in comparison to some other languages such as C++, and, in many ways, adheres much more closely to the to Object-Oriented paradigm. The support for Smalltalk-style Meta-Classes [5], the close relationship with the Internet, and the ability to load code components (local and remote), provides the mechanism for the construction of dynamically modifiable software, which under intelligent control can react to its environment to best achieve the goal of maximum efficiency.

References:

[1] Milligan, Peter, The Synthesis of Parallel Programs, PhD Thesis, The Queen's University of Belfast, 1986.

[2] Foster, Ian, Designing and Building Parallel Programs, Online Document, http://www.mcs.anl.gov/dbpp/, Addison-Wesley Inc., Argonne National Laboratory, and the NSF Center for Research on Parallel Computation.

[3] Jackson, M.A., Information Systems : Modeling, Sequencing and Transformations, IEEE 3rd International Conference on Software Engineer-ing, 1978.

[4] Hoare, C.A.R., Communicating Sequential Processes, C.A.C.M., Vol 21, No.8 pp 666-677, August 1978.

[5] Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA

[6] Snyder, L. Type architectures, shared memory, and the corollary of modest potential. Ann. Rev. Computer. Science., 1:289--317,1986

[7] Mandelbrot, B. Comment j'ai decouvert les fractales, La Recherche (1986), 420-424.

[8] Lewis, Simon, The Art and Science of Smalltalk, Hewlett Packard