Distributed Java Platform with Programmable MIMD Capabilities

T. Keane[1], R. Allen[1], T.J. Naughton[1], J. McInerney[2], and J. Waldron[3]

[1] Department of Computer Science, National University of Ireland, Maynooth, Ireland
[2] Department of Biology, National University of Ireland, Maynooth, Ireland
[3] Department of Computer Science, Trinity College, Dublin 2, Ireland
Corresponding author: tom.naughton@may.ie

Abstract. A distributed Java platform has been designed and built for the simplified implementation of distributed Java applications. Its programmable nature means that code as well as data is distributed over a network. The platform is largely based on the Java Distributed Computation Library of Fritsche, Power, and Waldron. The generality of our system is demonstrated through the emulation of a MIMD (multiple instruction, multiple data) architecture. This is achieved by augmenting the server with a virtual pipeline processor. We explain the design of the system, its deployment over a university network, and its evaluation through a sample application.

1 Introduction

A class of distributed computation systems is based on the client-server model. This class is characterised by (i) clients that instigate all communication and have no knowledge of each other (no peer-to-peer communication), (ii) a server that has little information on, or control of, its clients, and (iii) computations that are insensitive to fluctuations in the number of clients or client failure. Well-known and successful systems in this class include the Great Internet Mersenne Prime Search (GIMPS) [1] and SETI@Home [2]. These systems are usually designed with a single application in mind, and are not generalisable or programmable. A Java distributed computation library (JDCL) [3] was designed to provide a simple general-purpose platform for developers who wish to quickly implement a distributed computation in the context of a SIMD (single instruction, multiple data) architecture. Its aims were to allow developers to abstract completely from networking details and to allow distributed computations to be reprogrammed without requiring any client-side administration. Its attractions included network and platform independence, simplicity of design, and ease of use for developers.

Our contribution has been to continue development of the system, bringing it to a level in terms of functionality and robustness that permits demonstration of a large-scale application. The JDCL was in an early stage of development and required a number of enhancements and modifications to bring it up to such a level. In addition to refining the functionality and efficiency of existing features of the JDCL [3] our system contains enhancements that are in line with the aspirations of its original developers. They include facilitating ease of distribution [the client consists of an initialisation file and a single jar (Java archive) file], and coping with client failure.

N. Guelfi et al. (Eds.): FIDJI 2002, LNCS 2604, pp. 122–131, 2003.

The server is capable of both detecting client failure and redistributing the computational load.

Other enhancements (not aspirations of the original JDCL developers) include adding security to the clients, and expanding the range of applications that the JDCL can support. A security manager has been developed that limits the downloaded task's interaction with the client software and donor machine. The other major enhancement is the system's emulation of a MIMD (multiple instruction, multiple data) architecture. This is explained in Sect. 2. The design of the system is explained in Sect. 3. Section 4 gives a brief overview of how the system is programmed and in Sect. 5 the system is evaluated with an application from the field of bioinformatics.

Java proved to be an ideal language for the development of this system. It was possible to design a straightforward interface to the system: users are required to extend only two classes to completely reconfigure a distributed computation. Furthermore, identical clients (and identical downloaded tasks) could be run on a variety of platforms. Existing programmable distributed environments or libraries range from MPI [4] and PVM [5] to JavaSpaces [6] and the Java OO Neural Engine (Joone) [7].

2 Computational Theory for MIMD Emulation

A major enhancement of our system is its emulation of a MIMD architecture. In order to do this, the server simulates a pipeline processor capable of repackaging and redistributing partial results during a computation. In this section, we give the computational theory of MIMD emulation through client server processing.

Consider an input *X*, and a computation on that input *C*(*X*) that returns some result *r*. We could say that $r = C(X)$. In client-server computing, the server partitions the input data into *n* segments

$$X = \sum_{i=0}^{n-1} x_i \,. \tag{1}$$

such that each transformation $x_i \rightarrow C(x_i) = r_i$ is performed by one of a set of clients. The server reconstructs the original result by combining these partial results

$$r = C(X) = \bigcup_{i=0}^{n-1} C(x_i) \,. \tag{2}$$

where $\cup$ denotes an appropriate combination operation. This is the starting assumption of work related to SPMD (single program, multiple data) computation through functional programming [8]. In pipeline processing, a computation is decomposed into *m* smaller transformations that each acts on the result of the previous transformation, $r = C(X) = c_{m-1}\left(c_{m-2}\left(\cdots c_1\left(c_0\left(X\right)\right)\cdots\right)\right)$, where *X* is the input. A recursive definition of this concept could be written as follows,

$$r_j = \begin{cases} c_0(X) & \text{if } j = 0; \\ c_j(r_{j-1}) & \text{if } j > 0. \end{cases} \tag{3}$$

where $r = r_{m-1}$ can be regarded as the seed to the recursion and defines the final result. The first clause in Eq. (3) is the terminating condition (passing the input to the first transformation) and the second clause describes how the result of any one transformation depends on the preceding transformation. We use the following compact notation to represent the recursive definition of Eq. (3),

$$r = C(X) = \prod_{j=0}^{m-1} c_j(X) \ . \tag{4}$$

where $\prod$ denotes the operation to appropriately pass the results of one transformation to another. Equation (4) describes passing the complete input X to transformation c_0, the result being passed to c_1, and so on. Staying within the pipeline processing paradigm, we could further partition the input into n segments, as described in Eq. (1), and pass each segment in turn through the complete sequence of m transformations. Appropriately combining the partial results at the end of the final transformation, as in Eq. (2), would allow us to write Eq. (4) as

$$r = C(X) = \bigcup_{i=0}^{n-1} \left(\prod_{j=0}^{m-1} c_j(x_i) \right). \tag{5}$$

The advantages of the representation in Eq. (5) include the ability to arbitrarily change the granularity of the data throughput (some transformations may have restrictions on the size or format of their arguments) and to permit parallelisation of the computation. Pipeline computations could possibly be regarded as MISD (multiple instruction, single data).

It is possible to combine both the client-server (SIMD) and pipeline (MISD) models. This is important if we want to allow clients to effect arbitrary transforms rather than each one performing the same c_j. In this case, the server divides the computation as well as the data. It distributes to the clients a description of a transformation c_j as well as a data segment x_i. Since the partitioning shown in Eq. (1) is possible, there will not be any interdependencies between different parts of the data stream. Equations (4) and (2) could therefore be combined as

$$r = C(X) = \prod_{j=0}^{m-1} \bigcup_{i=0}^{n-1} c_j(x_i) \ . \tag{6}$$

which describes transforming all of the data segments with c_j before applying c_{j+1}, and so on. Since Eqs. (5) and (6) describe the same computation, this shows that the order in which each $c_j(x_i)$ is effected is unimportant, as long as one finds the appropriate $(\bigcup, \prod)$ pair. An out-of-order implementation of Eq. (6) is a MIMD computation. Consequently, an MIMD emulator is the by-product of a loosely coupled client-server simulation of a highly structured pipeline processor. This computational theory tells us nothing about how to find an appropriate $(\bigcup, \prod)$ pair, or how efficient the resulting MIMD emulation might be. Sanders [9] has proposed an efficient algorithm to emulate MIMD computations on a synchronous SIMD system. Our asynchronous

system should admit emulation algorithms that are even more efficient because it completely avoids what Sanders calls SIMD overhead [9] (where the globally issued instruction is not required locally). Our system is still susceptible to load imbalance overhead but this problem-dependent issue is inherent to all parallel computing, including MIMD parallelism. Figure 1 shows an abstract model of the system.

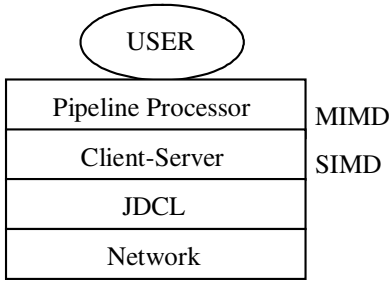

Fig. 1. System layers of abstraction

3 Design of the System

The design mirrors that of Sanders [9] with a number of enhancements inspired by our computational model. The user partitions the MIMD algorithm into multiple independent sequential stages, if possible. Each stage corresponds to a node in a theoretical 'pipeline.' The code corresponding to all stages (the Task) is sent to clients as a compiled Java class. Execution of each of the (one or more) stages then proceeds as a SIMD computation as in [9]. All stages of the pipeline could be 'processing' at the same time if the particular problem allowed. Our system is therefore most efficient at emulating MIMD computations that can be naturally expressed as a pipeline of SIMD computations. The overall system can be subdivided up into three main sections; common modules, server, and client.

3.1 Common Modules

We found that as with many distributed systems, there is a lot of overlap in terms of functionality between client and server. Each system (client and server) can keep two distinct logs: system logs and error logs. The system logs record system events as they happen. In the event of some catastrophic event (e.g. power loss), it may be possible to use these logs at the server to restart a particular problem at the point where it was halted. These logs are an optional feature on the client and are mainly used for debugging purposes.

Communications are performed on our system using Java sockets. We decided to produce one single module for use on the server and client to perform all socket communications. Its main functions are to open and close sockets, send and receive messages, and to terminate communications. The other shared communications module is the basic unit used for communication in the system called the `Message` class. It is be extendable so that a message can contain items such as data, algorithms, information on client status, and so on.

Each data unit that is sent out to be processed by our server has a user defined time limit associated with it. If the results for that unit have not returned to the server within the specified time, it is assumed that the unit has failed. This would normally happen through client failure, e.g. a donor machine being switched off. Additionally, if the client is not finished processing the unit when the time limit expires, it will contact the server and request a time extension. For these purposes, the server and client have been provided with a common timing module.

At the heart of our distributed system is the piece of compiled Java code that is downloaded from the server to each client – the `Task`. This Java class contains the algorithm that is executed over all of the subsequent data units that are received by the client. The user of the system is required to extend this common class. All tasks conforming to this interface will be accepted by the system. Any Java exceptions that occur in the task as it is executing at the client are fed back to the server via the communications protocol.

3.2 Server

The server can be divided up into three main sections (see Fig. 2). The `ServerEngine` is responsible for initialising the server at start-up. It reads in the user defined initialisation parameters via an external text file. After reading all of the initialisation options, it creates the log files, loads and checks the user defined classes (`DataHandler` and `Task`) and then creates the `ConnectionManager`. The `ServerEngine` also acts as the interface between the communications modules and the current running problem on the system. It also manages the lists of data units (pending and expired) that are currently out being processed by clients.

The `ConnectionManager` is responsible for listening on the server socket for new client connections and creating a new `ServerMessageHandler` thread to handle each connection. Each time a new client connection is received, a new Java thread is created that handles the communication on a separate port to the main port used by the server to listen for new connections.

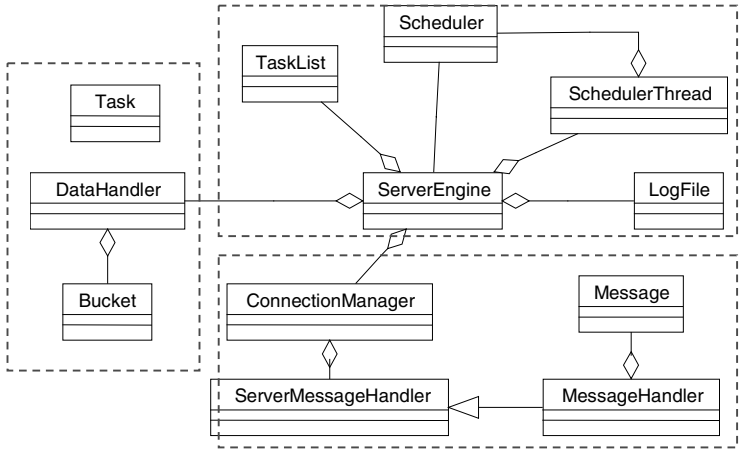

Fig. 2. Server design: the user extends Task (which is sent to the client) and DataHandler

3.3 Client

The client software can be divided up into two main sections (see Fig. 3). The `ClientEngine` is responsible for initialising the client software. Its first task is to read in the server details that are contained in the external text file that is included in the client installation. Once these details have been parsed correctly, the `ClientEngine` starts the security manager that remains in operation for the lifetime of the client. This strictly limits the client's interactions with the donor machine's resources. The final task of the `ClientEngine` is to start the communications section of the client.

The `ClientMessageHandler` is at the centre of the communications section of the system. It manages the communications protocol, receives and initialises the downloaded task and manages the execution of each data unit. Each downloaded algorithm is received by the client as a compiled Java class. By using Java's class loading facilities, the algorithm is subsequently dynamically loaded into the executing client. The `ClientMessageHandler` uses the shared timing module to monitor the time being taken to process each data unit. If a data unit is taking longer than its allotted time to process its data, the client can request a time extension. The client continues in an infinite loop requesting and processing data units from the server. When the server sends a new algorithm to the client, this new algorithm is loaded dynamically and all subsequent data units received are processed using the new algorithm. There are extensive exception handling mechanisms coded into the client so that the client should run continuously until explicitly shut down by the donor of the machine. Full details on the design of the JDCL and its extensions can be found in [3,9].

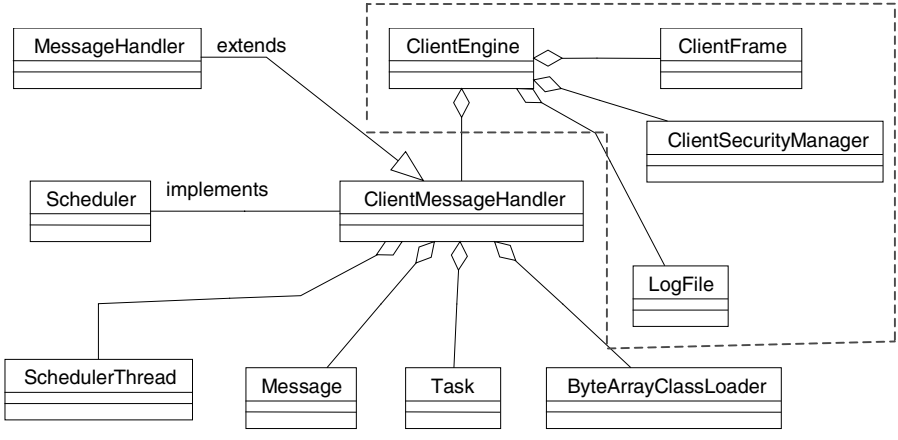

Fig. 3. Client design

4 Programming the System

To program the system with a given problem, there are two java classes that must be extended using the standard Java inheritance mechanisms. These are the DataHandler

class and the Task class. The subclass of DataHandler specifies how to manage the data for the distributed computation. The subclass of Task is where the distributed algorithm is programmed. In order to program a MIMD computation, the user makes use of the Bucket class when designing their DataHandler.

4.1 DataHandler

The main purpose of the extended DataHandler is to manage all of the data relating to the current problem. The first section of the DataHandler is the init() method. The purpose of this method is to initialise whatever data structures are necessary for the overall computation. This can involve things such as setting up readers of files, initialising arrays, assigning variables values, etc. The next section is the getNextParameterSet() method. This is where the pre-processed data units are generated to be sent to the clients. This method is called every time a client requests a data unit to process. The return type is an Object array and since all Java classes inherit from the Object class, this method can return any data type supported by Java. The task running at the client receives this Object array as its data. Therefore, it is usual for the elements of this Object array to be explicitly typecasted at the client. The final section of the DataHandler is the resultsHandler() method.

4.2 Task

The subclass of the Task class describes how the data received by the clients is to be processed. There is only one method that must be extended in the task - the run() method. The pre-processed data that is sent by the server in each data unit is available through the parameterList variable. The general format of the run() method is described below in Fig. 4.

```
public void run(){
   try{
       Object[] array = parameterList;

       <actual algorithm here>

       returnList = new Object[];
       returnList[x] = // results of computation here
       endProcessing();
   }catch( Exception e ){
       exceptionInProcessing( e );
   }
}
```

Fig. 4. General form of run() method

The whole method is encompassed in one large try-catch statement. The purpose of this statement is to handle any exceptions the run() method can generate. All exceptions will be caught and are fed back to the server. The post-processed data is returned via the returnList Object array. At the bottom of the run() method is the endProcessing() which tells the client that the task has finished processing the data.

4.3 Bucket Class

The main purpose of this class is to effect the theoretical pipeline for MIMD computations. This allows the developer to set up the server to act like a pipeline processor (Fig. 5) with several different intermediary stages in the distributed computation.

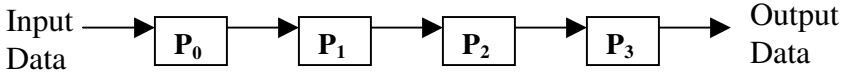

Fig. 5. Pipelined processes, where P_0 through P_4 are the processes

The pipeline is simulated by using the bucket class to represent the storage required at each stage of the pipeline. The information is stored in each bucket by using Java Vectors thus maintaining the type independence of the system. In a MIMD computation, the complete algorithm (including the code for all stages of the pipeline) is sent to each client with a flag being sent with each data unit to indicate which stage the data unit belongs to.

5 Application of System

Strands of DNA can be regarded as strings of base-4 symbols. The nucleotides adenine, guanine, cytosine, and thymine are represented by the symbols A, G, C, and T, respectively. Our application involved building up a picture of the repeated substrings within a DNA strand. We chose the DNA of the tuberculosis bacterium, which contained approximately 5M nucleotides. As well as exact-matched substrings, we also permitted insertions and deletions, up to a maximum in each case, to reflect the fact that slightly different DNA strings can code for the same functionality. In each case, we performed the search on the complete DNA strand and recorded the locations of all repeated substrings of length greater than 13 in a database. A more detailed account of our search algorithm and the results obtained is in preparation [10].

We ran the three distributed algorithms over the aforementioned laboratory of 90 clients and recorded the speedup data shown in Table 1. For these computations we did not have sole use of the laboratory. The number of processors varied over our computation but we noted that at all times at least 40 processors were working for the server. The disparity between speedups (i) and (ii) was due to choosing a work unit size for the former that was too small (thus not making efficient usage of the intra-laboratory network resources). The difference between speedups (ii) and (iii), we believe, simply reflects the uncertainty of resource-availability in a busy university laboratory environment. Taking only the results for insertions and deletions, our system has demonstrated an average speedup of 53 with (assuming a full complement of 90 processors) an efficiency of 59%. For a finer view of this speedup, we conducted an experiment in which the server distributed 100 equal work units among a number of dedicated clients. We varied the number of clients, recorded the computation time each time, and created the speedup plot shown in Fig. 6. This shows that with sole use of the laboratory resources, and for our specialised application, we

could get very close to the theoretical speedup maximum. The theoretical speedup maximum was calculated from $S(p) = w/\lceil w/p \rceil$, where w denotes number of individual work units and where S, speedup, is parameterised by number of processors p.

Table 1. Speedup achieved for each of the three repeated substring search strategies

Search strategy	Single processor	40-90 processors	Speedup
(i) Exact matching	130 hours	28 hours	4.6
(ii) Insertions	1790 hours	31 hours	57.7
(iii) Deletions	1670 hours	35 hours	47.7

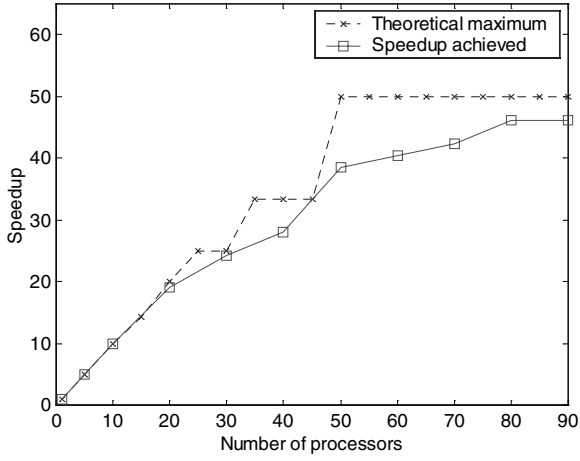

Fig. 6. Evaluation of speedup

6 Conclusion

We have refined the JDCL in terms of efficiency and functionality, including the successful extension of the system to emulate a MIMD architecture. This has allowed us to implement a large-scale bioinformatics application. The system is completely generalisable, and because it is written in Java, the developer interface is simplified to the extension of two classes. Work is ongoing on the next generation of this system. Several new features are to be incorporated into the new system including a multi-problem scheduler, compression, encryption and authentication of all communications, a remote server interface and the migration of all communications to Java RMI. Future work includes performing this type of DNA analysis on other similar size genomes with a view to eventually performing this type of analysis on the human genome.

We gratefully acknowledge assistance from the Department of Computer Science, NUI Maynooth, and technicians M. Monaghan, P. Marshall, and J. Cotter.

The continuation of this research has recently been funded by the Irish Research Council for Science, Engineering and Technology: funded by the National Development Plan.

References

[1] G. Woltman, "Great Internet Mersenne Prime Search," 1996. <http://www.mersenne.org>
[2] SETI@Home - Search for Extraterrestial Intelligence at Home, 1999. <http:// setiathome.ssl.berkeley.edu>
[3] K. Fritsche, J. Power, J. Waldron, "A Java distributed computation library," *Proc. 2nd International Conference on Parallel and Distributed Computing, Applications and Technologies* (PDCAT2001), pp. 236–243, Taipei, Taiwan, July 2001.
[4] Message Passing Interface Forum, "MPI: A message-passing interface standard," *International Journal of Supercomputer Applications and High Performance Computing* 8(3/4), 159–416, 1994.
[5] V. S. Sunderam, "PVM: a framework for parallel distributed computing," *Concurrency: Practice and Experience* 2(4), 315–340, 1990.
[6] Sun Microsystems, "JavaSpaces Technology," 2001. <http://java.sun.com/products/javaspaces>
[7] VA Linux Systems, Inc, "Joone - Java object oriented neural engine," 2001. <http://joone.sourceforge.net/>
[8] F. Loulergue, G. Hains, C. Foisy, "A calculus of functional BSP programs," Science of Computer Programming, 37, 253–277, 2000.
[9] P. Sanders, "Emulating MIMD behaviour on SIMD machines," *International Conference on Massively Parallel Processing Applications and Development*, pp. 313–321, Delft, 1994. Elsevier. (Extended version in "Efficient emulation of MIMD behavior on SIMD machines," Technical Report IB 29/95, Universität Karlsruhe, Fakultät für Informatik, 1995.)
[10] R. Allen, T. Keane, T.J. Naughton, J. McInerney, J. Waldron, “Segmental duplication in Tuberculosis genome,” submitted October 2002.