Distributed computing for DNA analysis

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

We report on extensions to a Java distributed computation library (JDCL) by Fritsche, Power, and Waldron, with application to a problem in the field of bioinformatics. Within our framework the system has been extended to support applications requiring a MIMD (multiple instruction, multiple data) architecture. The system has been evaluated through a DNA pattern matching application over a network of 90 PCs. The user is required to extend only two Java classes to completely configure a distributed computation.

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 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. 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 client software also integrates seamlessly with the donor machines. As a low-priority service it utilises only 'spare' clock cycles so that the inconvenience a donor experiences is minimised. The other major enhancement is the system' s emulation ofa MIMD (multiple instruction, multiple data) architecture. This is explained in Sect. 2. The design of the system is outlined in Sect. 3.

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]. The strength of our system, we believe, is that it takes full advantage of dynamic multiprocessor architectures (such as the Internet itself) in which individual processors work in a completely asynchronous manner and do not have the ability to efficiently support a shared memory.

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. In pipeline processing, a computation is decomposed into m smaller transformations that each acts on the result of the previous transformation, $r = C(X) = c_{m-1}(c_{m-2}(\cdots c_1(c_0(X))\cdots))$, 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 $(\cup,\prod)$ 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 $(\cup,\prod)$ pair, or how efficient the resulting MIMD emulation might be. Sanders [8] 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 [8] (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. The user sees a pipeline processor, which sits on a client-server architecture and uses the JDCL. The standard client-server model is available to the user by constructing a pipeline composed of a single processing stage.

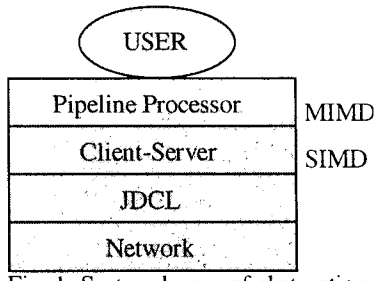


Fig. 1: System layers of abstraction

3 Design of the system

The design mirrors that of Sanders [8] 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 Java class. Execution of each of the (one or more) stages then proceeds as a SIMD computation as in [8]. The server maintains a list (or Bucket) of partial results corresponding to the input to each pipeline stage. As soon as there are sufficient partial results in a bucket (as specified by the user) they are combined by the server and sent to a client along with a token indicating which part of the code to execute (i.e. which stage in the pipeline). The client returns this token so that the server can file the result in the next Bucket in the sequence. As such, 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 server is divided into three main sections. The ServerEngine (see Fig. 2) manages all server-side data structures, classes, and logs. It retrieves all of its parameters from a user-defined initialisation file and is responsible for loading the user-defined classes. The communication section (ConnectionManager, etc.) handles all the communication between the server and client. The user-defined classes (Task and DataHandler) are extended by the user to specify a distributed computation. DataHandler partitions the data to be bundled with copies of Task, sets appropriate flags (if necessary) in Task, and collates the partial results. The Bucket class is available to the user to avail of pipeline processor functionality when

designing DataHandler. The client design (see Fig. 3) can be split into two main parts. The ClientEngine is responsible for such tasks as initialising the software, starting the security manager, log files, and GUI. The ClientMessageHandler is responsible for tasks such as implementing and managing the communications protocol, initialising and managing the downloaded Task, keeping track of all timing functions, and managing the data units from the server. Full details on the design of the JDCL and its extensions can be found in [3,9].

4 Evaluation

It is evident that when simulating a pipeline processor the server is responsible for effecting peer-to-peer communication between neighbouring processing stages, and will undoubtedly be the bottleneck in a MIMD emulation. In fact, finite bandwidth at the server (rather than finite memory) ultimately limits the scalability of the system. This is because the server maintains information only about tasks (completed and uncompleted): space complexity is dependent only on the size of the computation and is independent of the number of clients. Since the server is designed never to initiate communication with a client, an essential practical consideration is sharing the limited bandwidth at the server between clients returning results and clients looking for a new task. This balance is investigated while examining how the server copes with overloading through an empirical evaluation.

A cut-down version of the DNA substring problem (outlined in Sect. 5) was partitioned into 100 work units (each requiring approximately 6 minutes of processing time), and repeatedly solved over different numbers of clients in the range [1,90]. We employed a laboratory of 90 Dell Optiplex GX1 machines (Pentium 600MHz processor, 128Mbytes memory, 6Gbytes storage). Our server resided on a Dell Optiplex GX110 machine (Pentium 1000MHz processor, 256Mbytes memory) with a 10Mbit/s connection to the laboratory. For these tests we had sole use of the processor and network resources. Clients were encoded with two wait times. A 'retry wait' determines how long a client will wait before retrying to connect to the server (if it failed to connect). A 'null wait' determines how long a client will wait before asking for a new task after the server previously informed it that there were currently no outstanding tasks. The smaller the null wait time the quicker clients will come on board when the server does need them. However, unsuccessful requests for tasks could slow the overall computation by blocking clients that are trying to return results.

To simulate in the order of thousands of client connections we handicapped the server at an operating system level by only allowing it a fixed number of socket connections each minute. As the allowable rate of socket connections was reduced, we found that the system's performance became increasingly sensitive to the difference between the null wait time and the processing time required for the task. Figure 4(a) shows plots of processing time against number of clients, for two different null wait times, a fixed retry wait of 10s, and a maximum rate of 15 connections/minute at the server. Figure 4(b) shows the same data plotted to indicate speedup. The plots show that by carefully selecting the null wait time we can strike a balance between pressing clients into service as soon as possible and blocking clients that are trying to return results with requests for new tasks. [Figure 4(b) shows a comparison with the desired linear speedup. Our deviation from this can be partially explained by not configuring the number of tasks to be very much greater than the number of processors.]

5 Application

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 contains approximately 5M nucleotides. As well as exact-matched substrings, we also permitted insertions and deletions, up to a maximum in each case, because slightly different DNA strings can code for the same

functionality. In each case, we searched the complete DNA strand and recorded the locations of all repeated substrings of length greater than 13 in a database. The longest exact-match substring in tuberculosis contains 1526 nucleotides and first appears at index 400211 of its sequenced DNA. A more detailed account of these results 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 as at times some machines were switched off or were booted into operating systems on which a JVM was not installed. We noted, however, 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 task 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%.

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 can be simplified to the extension of two classes. As would be expected, the system is also platform and network independent. Future work includes a scheduler for the server and a selection of client-side configuration options to increase its acceptability among potential donors.

We gratefully acknowledge assistance from the Department of Computer Science, NUI Maynooth, and technicians M. Monaghan, P. Marshall, and J. Cotter. We also thank the anonymous reviewers of this paper for their valuable contributions.

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. <java.sun.com/products/javaspaces>

[7] VA Linux Systems, Inc, "Joone - Java object oriented neural engine," 2001. <http://joone.sourceforge.net/>

[8] P. Sanders, "Emulating MIMD behavior 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, Universitat Karlsruhe, Fakultat fur Informatik, 1995.)

[9] R. Allen, T. Keane, T.J. Naughton, J. Waldron, "A general-purpose distributed computing environment with application to DNA analysis," Tech. Report no. NUIM-CS-TR-2002-03, Department of Computer Science, National University of Ireland, Maynooth, May 2002.

[10] R. Allen, T. Keane, T.J. Naughton, J. McInerney, *BioInformatics*, in preparation.

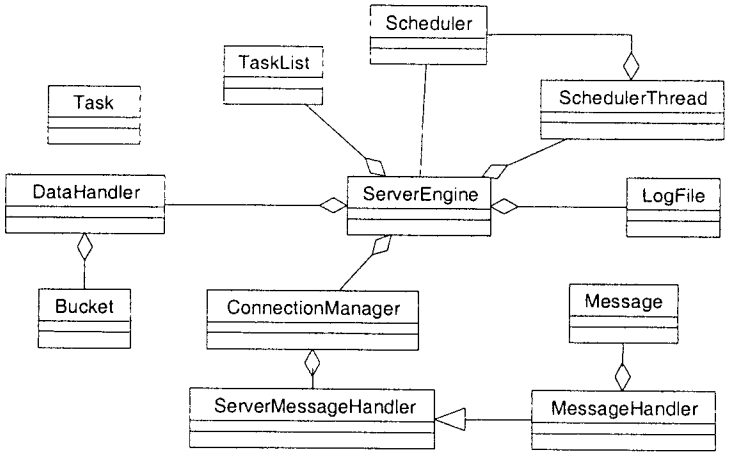


Fig. 2. Server design: the user extends Task (which is sent to the client) and DataHandler.

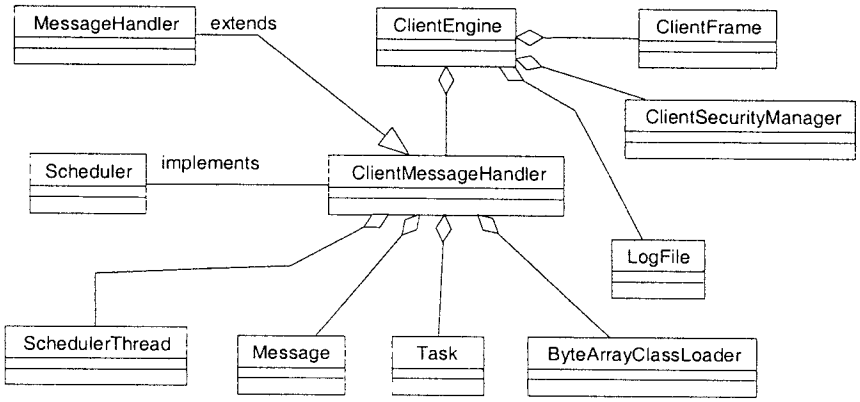
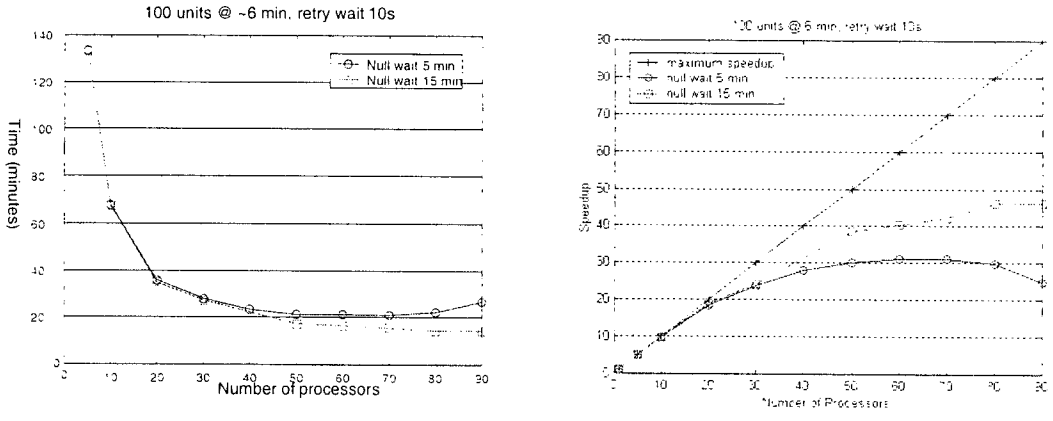


Fig. 3. Client design.



(a) (b)

Fig. 4. Evaluation of overloading on the server: (a) processing time, and (b) speedup.

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

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