Multi-Tiered Distributed Computing Platform

Andrew Page, Thomas Keane, Richard Allen, Thomas J. Naughton
Department of Computer Science
National University of Ireland, Maynooth
County Kildare, Ireland
{andrew.j.page, tom.naughton}@may.ie

John Waldron
Department of Computer Science
Trinity College Dublin
Dublin 2, Ireland

ABSTRACT

A simple programmable Java platform-independent distributed computation system has been developed to exploit the free resources on computers linked together by a network. It is a multi-tiered distributed system model, which is unbounded in principal. The system consists of an n-ary tree of nodes where the internal nodes perform the scheduling and the leaves do the processing. The scheduler nodes communicate in a peer-to-peer manner and the processing nodes operate in a strictly client-server manner with their respective scheduler. The independent schedulers on each tier dynamically allocate resources between jobs based on the constantly changing characteristics of the underlying network. The system has been evaluated over a network of 90 PCs with a bioinformatics application.

1. INTRODUCTION

Currently there are a few notable distributed computing platforms such as SETI@home [1], distributed.net [3], and United Devices [9]. They work on the principle of a user donating their machine to the system so that its free resources can help to process computationally large problems. The widespread success of the Internet has meant that these distributed systems have been able to harness a huge amount of computational resources from the donors' machines, which would otherwise have not been utilised to their full potential. These distributed systems rely on a client-server model, where the distributed system has one server and many clients. In practice it has been used by the SETI@home distributed system very successfully, with up to three million client machines as part of the system [1]. However, since there is only one server (single machine or cluster) for all of the clients there is thus a finite limit on the number of clients the system can handle at any one time, with this limit depending on the network resources and computational resources at the server. A common solution is to increase the bandwidth of the server's Internet connection and to upgrade the power of the server, but this can be expensive.

The Berkley Open Infrastructure Networking Computing (BOINC) [2] is a programmable successor to SETI@home. Although it is programmable, there are a number of preconditions on the types of computations that will be run on the system, which limits its usability. The problems must be appealing, so that users will be interested in donating their free resources to the project. Also, only problems that can be structured such that there is no interdependency between different pieces of data are considered. This is because BOINC retains a client-server architecture, and implements a one-step processing stage. If a computation requires further processing of intermediate results, separate dedicated machines must be used.

Our aim is to design a programmable distributed computing platform that is unrestricted in terms of the type or structure of computations that can be performed. We retain aspects of the client-server model, but introduce peer-to-peer communication within a tree of scheduling nodes that serves to overcome the scalability limitations of employing a single server. Java is used to ensure platform independence for both scheduler nodes and processing nodes.

In Sect. 2, we introduce the multi-tiered model. In Sect. 3, the designs of the main components of the system are presented. Implementation and performance evaluation are discussed in Sect. 4, and we conclude in Sect. 5.

2. OVERVIEW OF THE SYSTEM

The foundations for the multi-tiered distributed computation system were laid in the Java Distributed Computation Library (JDCL) [4] and its extensions [7], which provide an emulated MIMD pipeline processor. The JDCL provided a simple development platform for developers who wished to quickly implement a distributed computation system. It arose out of the need for a platform-independent distributed system that was easy to create, adapted to system changes, and was easy to deploy. Systems such as SETI@home did not address these issues very well and were designed to be platform dependant and for a single purpose only. The JDCL does, however, suffer from similar scalability problems to those of SETI@home in that it has one server (single machine or cluster). The design of the current multi-tiered system aims to address this concern.

PPPJ '03 Kilkenny, Ireland

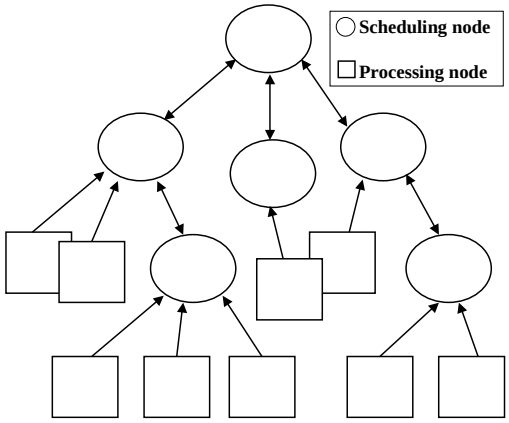

Figure 1: Example topology of multi-tiered distributed computing system.

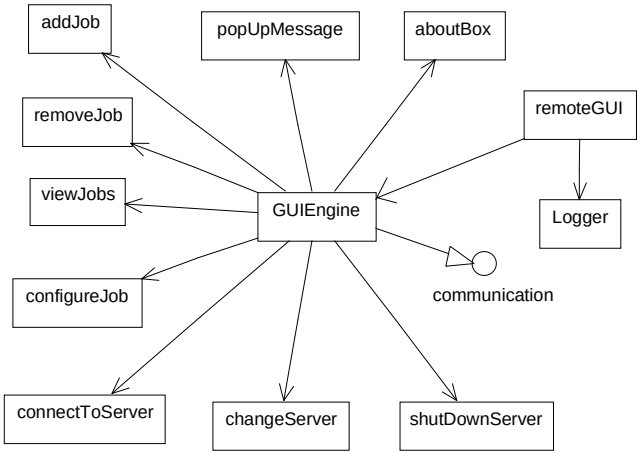

Figure 2: The design of the remote interface.

2.1 Multi-tiered Model

The multi-tiered distributed computing system was created with the intention of utilising the maximum computational resources of the machines under its control, while not placing any limitations on the maximum size of the system. The client-server model alone was not sufficient, so a hybrid model was created that combines the advantages of peer-to-peer and client-server architectures within the one model. The system consists of an n-ary tree of nodes where the internal nodes perform the scheduling and the leaf nodes do the processing, as shown in Fig. 1. The scheduler nodes communicate in a peer-to-peer fashion, which permits top-down reconfigurability, extensibility, and tree balancing. The processing nodes operate in a strictly client-server fashion with their respective scheduler, which promotes donors' trust in the system (anonymity, security) and admits a simple and robust design.

The potential size of the distributed system, in width and depth, is unlimited in principle due to the ability of the system to dynamically add another scheduling node, tier of scheduling nodes, or processing nodes. Also, since the scheduling nodes are distributed, the potential for bottlenecks is reduced, thus improving the performance of the system. The distributed nature of the scheduling nodes means that if one (other than the root) were to fail, the rest of the system could always compensate for the loss of that branch. By having multiple servers it means that the distributed system is a MIMD architecture, as opposed to the previous version, which only emulated a MIMD system using a pipeline processor [7]. Although not an increase in capability, this does increase the sophistication of the algorithms that can be distributed over the system, and therefore increases the user's task in programming a distributed computation. We have attempted to structure the programmers' interface as much as possible in this regard, to find a balance between expressiveness and simplicity. For example, the programmer is required only to extend two classes to fully specify a distributed computation, as explained later in Sect. 3.

2.2 Recursively Splitting up a Problem

Any problem that is run on our distributed system must be parallelised by the user. The system dynamically breaks up the problem into units, according to a user-defined algorithm, at each tier in the system. The system attempts to get the mean processing time of each unit to be equal to the user-defined optimal value. Given a problem, the maximum speedup achievable is limited by the number of atomic units that the problem can be broken into, and the number of clients in the system.

2.3 User-defined Fair Round Robin Scheduling

Each server has its own independent scheduler and decides dynamically how the resources of the system are to be divided up among the jobs in its queue. The scheduler employs strategies based on a user-defined fair round robin scheduling mechanism [6, 5]. The user sets an integer weighting for each problem that specifies the relative proportion of parallel computation time that should be allocated to that problem. The scheduler combines this weighting with the mean unit processing time of each problem to ensure that parallel computation time is allocated in the correct proportions.

3. DESIGN

There are three distinct parts to the multi-tiered distributed computation system. These are the client (processing node), the server (scheduling node), and the remote interface. The operation of server and client is explained in detail in [7]. The user is required to extend two classes to create a problem to run on the system. The DataManager class (in the scheduler) specifies how the problem is to be broken up and the intermediate results put together. The Algorithm class (in the client) specifies the actual computation.

3.1 Remote Interface

The remote interface is a standalone application, which communicates over TCP/IP and allows the administrator to fully control all of the jobs and servers in the system. This is achieved by using SUN Microsystems' Remote Method Invocation (RMI) communications technology. Jobs can be added, removed, configured, and their current state viewed. Likewise servers can be queried and shutdown.

The RemoteGUI object (see Fig. 2) launches and initialises the interface and sets up the logs files, which record all system events and errors. The GUIEngine is the centre of the application. The individual features are self-contained

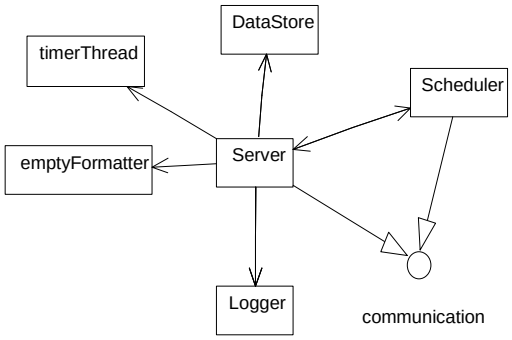

Figure 3: The design of the server.

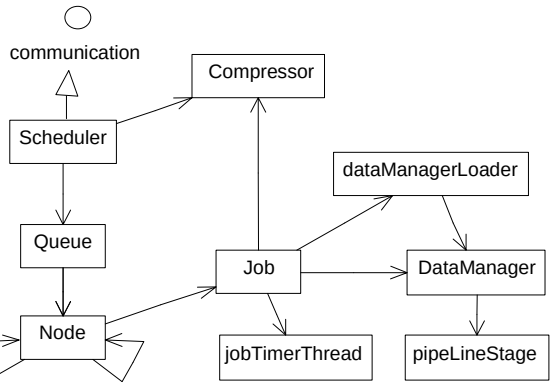

Figure 4: The design of the scheduler.

in their own classes. The users of the system do not need any knowledge of the topology or workings of the system in order to submit jobs and get their processed results back. Any commands sent from the remote interface get propagated to all the servers in the system. The remote interface can connect and disconnect from servers without affecting the distributed system.

3.2 Server

The server (scheduling node), as shown in Fig. 3, is the engine of the entire distributed system, controlling subordinate servers (subordinate scheduling nodes) and clients (processing nodes) and is largely described in [7]. There have been a few modifications to the pervious version to allow it to be a multi-tiered distributed system. A server can be added to the distributed system very easily while the system is running, and likewise can be removed from the system without affecting the stability of the system or causing running jobs to become corrupted. A new server contacts the server above it and synchronises itself with the rest of the system. If a server is removed or fails the rest of the system compensates for this loss by reallocating the work it had in its queue to another part of the system (the server above the failing server will have a record of all jobs in the failing server's queue). This resilience to failure ensures the system can perform in unknown operating conditions.

3.3 Scheduler

The scheduler is a central part of the server and indeed of the entire distributed system. In the multi-tiered distributed system the scheduler was created to allow a limitless number (memory limits aside) of different jobs to run at the same time in parallel on the system.

Each Job object (see Fig. 4) is self contained. It does not depend on anything external to it and is contained in a node in the queue, ensuring the complete encapsulation of each Job in the system. These Job objects pass any requests for units of work or processed results to the user-defined DataManager to process [7]. Clients and servers both get the same units when they request more work, although the client processes the unit of work while the server breaks it up further into more subunits. The generic nature of everything that goes in and out of the Job object allows for maximum flexibly in the design of the topology of the system. Each scheduler is independent of every other in the distributed system, optimising themselves to the computation resources available. If the processing results of a unit sent by a server are not returned within a specific period, the unit is said to expire and the server resends it to the next requesting server/client. Each Job also contains its own timer so that if the unit is close to expiring the server requests an extension from the server above.

4. IMPLEMENTATION

The entire system was designed using objects at a high level, thus when it came to implementation Java was chosen because of its object orientated capabilities and also for its platform-independence and ease of implementation. Almost every operating system and hardware architecture supports a Java Virtual Machine, from desktop PCs to mobile phones. The computational resources available for this system's deployment were based on machines with varying operating systems, such as Microsoft Windows NT/2000/98, GNU-Linux, HP UNIX, and varying hardware architectures such as those of Intel, SUN, and HP.

4.1 Applications

The multi-tiered distributed system has so far been used to analyse tuberculosis and ecoli genomes searching for duplicated patterns. An advantage of this type of distributed system is that the client machines will get upgraded over time, and hence the computational power of the network will grow even if the number of clients remains constant. Thus certain problems, which at the moment seem to require too much computational power, may become computationally feasible over time.

4.2 Performance Evaluation

We carried out performance tests to evaluate the capabilities of the multi-tiered distributed system. We set out to show that adding more servers would increase the capacity of the distributed system. These were carried out in a lab with a dedicated network of 90 PCs connected by Ethernet. Each had a 1 GHz Pentium III with 256 MB of RAM and 20 GB of hard disk space and was connected to a 10 Mb/s Ethernet LAN.

Figure 5 shows the speedup that can be achieved using this system. The problem posed was an analysis of the ecoli genome (five million nucleotides in length) to find duplicated strings within the genome. Based on the speedup data, the average efficency using 94 processors is 77%.

Since a server can handle in the order of thousands of

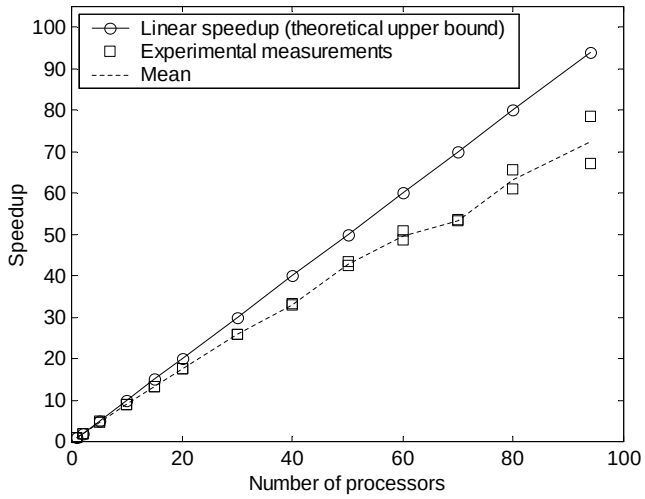

Figure 5: Speedup achieved with a bioinformatics application.

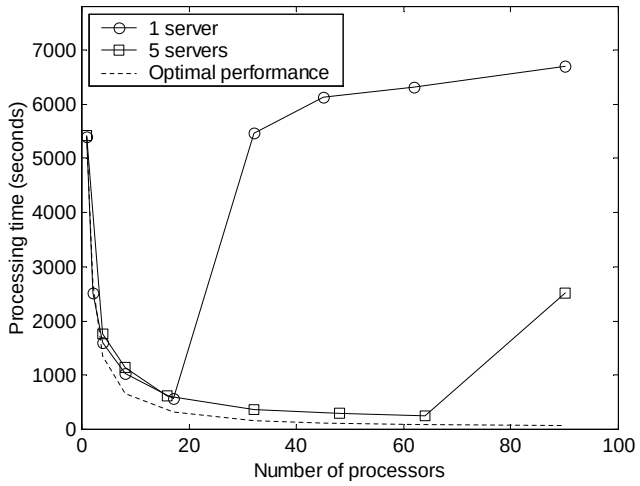

Figure 6: Processing time comparisons with 1- and 5-server distributed computation systems, in the presence of simulated congestion.

clients we had to simulate congestion to demonstrate that the multi-tiered distributed system can increase the number of clients in a system. This was achieved by using Shunra's freeware Nimbus bandwidth throttling software [8]. The bandwidth of each server was constricted to only 14.4 kb/s, and in addition, each unit returned from a server or client was bloated with extra data to make the returned results larger. The size of this bloated data was proportional to the size of the unit. The problem posed to the distributed system was a pattern matching exercise with the tuberculosis genome (four million nucleotides in length) to find all duplicated strings within the genome.

The problem was initially run with 1 server and n clients, which is traditional for the client-server model. Next we ran the problem using 5 servers, arranged as 1 server on top with 4 servers underneath, with n clients. Figure 6 shows the resulting plot for several values of n. This plot shows that after a certain number of clients, network congestion at the server causes the processing time to actually increase. As the number of servers increases, this critical number can be increased. Eventually, the multi-tiered system too reaches its capacity but it can handle many more clients than the single server system. The optimum was calculated assuming linear speedup from the timing with one client.

5. FUTURE WORK

Future improvements to the system will allow for the dynamic rebalancing of the system to improve parallel efficiency, and modification of the scheduling strategy. The next problem to be run on the system will use Pollard-Rho factorisation to test encryption strengths of selected algorithms.

6. ACKNOWLEDGEMENT

Support is acknowledged from the Irish Research Council for Science, Engineering, and Technology, funded by the National Development Plan.

7. REFERENCES

[1] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Massively distributed computing for SETI. *Computing in Science & Engineering*, 3(1):78–83, Feb. 2001.

[2] http://boinc.berkeley.edu.

[3] http://www.distributed.net.

[4] K. Fritsche, J. Power, and J. Waldron. A Java distributed compution library. In *2nd International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT2001)*, pages 236–243, Taipei, Taiwan, July 2001.

[5] S. Hariri, H. Topcuoglu, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parrallel and Distributed Systems*, 13:260–274, Mar. 2002.

[6] S. Kanhere, A. Parekh, and H. Sethu. Fair and efficient packet scheduling using elastic round robin. *IEEE Transactions on Parrallel and Distributed Systems*, 13:324–326, Mar. 2002.

[7] T. Keane, R. Allen, T. J. Naughton, J. McInerney, and J. Waldron. Distributed Java platform with programmable MIMD capabilities. In N. Guelfi, E. Astesiano, and G. Reggio, editors, *Scientific Engineering for Distributed Java Applications*, volume 2604 of *Springer Lecture Notes in Computer Science*, pages 122–131, Feb. 2003.

[8] http://www.shunra.com.

[9] http://www.ud.com.