A Java Distributed Computation Library

Karsten Fritsche[a], James Power[b] & John Waldron[†]

[a] Mathematics, Statistics & Information Systems,
University of Natal, Pietermaritzburg 3209, South Africa.
Email: karsten@alpha.futurenet.co.za

[b] Dept of Computer Science, NUI, Maynooth, Ireland.
Email: James.Power@may.ie Tel: +353-1-7083447

[†] Dept of Computer Science, Trinity College, Dublin 2, Ireland.
Email: jtw_ie@yahoo.co.uk Tel: +353-1-6083685
[†]Corresponding author.

Abstract

This paper describes the design and development of a Java Distributed Computation Library, which provides a simple development platform for developers who wish to quickly implement a distributed computation in the context of an SPMD architecture (Single Program, Multiple Data). The need for this research arose out of the realisation that the currently available distributed computation libraries and systems do not adequately meet certain criteria, such as ease of development, dynamic changes to system behaviour, and easy deployment of distributed software. The proposed solution to this problem was to produce a Java-based distributed computation library which enables developers to use the Java language to quickly and easily implement a distributed computation. The results of experiments conducted using DCL are also presented, as a means of showing that DCL met its design goals.

keywords: Distributed Architecture

1: Introduction

Research in the field of distributed computation has been aimed mainly at developing programming environments, API's and libraries which allow developers to design software which can take advantage of distributed computational resources. This raises serious issues, such as the coordination of computational effort across a network; communication and sharing of resources between processes distributed across a network; the provision of fault tolerance, transparency and other desirable distributed system properties; and the

deployment of software to run on such systems. Many distributed computation environments make the handling of these issues the responsibility of the developer. Due to the complexity of these issues, developing applications using such environments is usually difficult, error-prone, and slow. Attempts to use such systems to solve problems also lead to various administrative difficulties, especially with regards to the deployment of distributed software. For example, many distributed computation systems are based on the client-server model, in which the client computers run purpose-built client software. This client software is responsible for communication with the server in order to coordinate the allocation of processes in the system. Typically, the client software is dedicated to performing only one task or type of task, with very little ability to change its own behaviour. However, the central system administrators may wish to change the task the system performs, or wish to update the existing client software, which will usually involve deploying an updated client application to each of the client computers. This process can be extremely tedious. The problem is made worse in the case where the client computers are not directly under the control of the administrators, but are owned by individuals who willingly take part in distributed computation projects. In such cases, the administrators must notify each participant that an updated client exists and then encourage all the participants to download and install the new client. Thus, much of the administrative burden falls on the client-side. This problem is compounded further for developers when one considers that different versions of the client software need to be written, compiled and tested for each different platform on which the client software is expected to function. Together, these factors make the developer's task many times more complex, and vastly increase development time. They are also not adequately addressed by the currently available distributed computation libraries and environments.

To summarise, the situation of concern addressed in this paper is that the currently available distributed computation environments do not provide a platform for development that is easy to use, generally do not support dynamic changes to system behaviour (especially in terms of updated client behaviour), and do not promote easy deployment of software (due to platform specific issues).

The main goal of this research project is to solve these problems by providing developers with a Java-based library called the Distributed Computation Library (DCL) that can be

used to easily implement a distributed computation. The library should not require the programmer to know specific details about the underlying architecture of the system, but should rather allow the programmer to deal only with high level constructs which make sense in a distributed environment. The library should make changing system behaviour dynamically and deployment of software as easy as possible.

Sub-goals which need to be considered in order to achieve the main goal are:

- to determine what level of abstraction, in the form of high level constructs, are required and on what system models the library should be based in order to make using the library as developer-friendly as possible with respect to a distributed environment.
- to determine what formal classifications have been developed to describe tasks and what criteria determine whether or not a task will be well suited to being implemented in a distributed environment.
- to develop an effective task distribution and communications mechanism using such technologies as Java Remote Method Invocation (JavaRMI), object serialisation and client-server architectures.
- to implement a suitable problem using DCL, and to use this as a means of showing that DCL fulfills its design objectives.

The potential users of this library are developers who wish to implement distributed computations without having to concern themselves with the details of the underlying systems. It is thus assumed that the users of the library will have prior programming experience, specifically in the Java language, as well as familiarity with the Java Runtime Environment (JRE) version 1.2 (or later), and Java compiler tools. It is also assumed that they will be familiar with the main concepts in the field of distributed systems. The library will not be intended as a teaching tool.

It is also assumed that each of the potential host computers will have a Java Virtual Machine (JVM) installed (Java Development Kit version 1.2 or later), and that the network will support the TCP/IP protocols. This is due to limitations of previous Java versions, and limitations imposed by Java's networking API's. These assumptions are not unreasonable (nor overly restrictive), considering the widespread use of the TCP/IP suite of protocols and the ready availability of JVMs for most commonly used hardware and software platforms.

As several authors have noted, there is a growing trend in the IT industry towards an object-oriented paradigm for distributed computing. Furthermore, from a practical point of view, the dominant architecture used when discussing distributed computing is the client-server model due to its ability to conceptually model the way in which distributed software components interact (Umar 1993), although various other models are used for theoretical discussions. This view is supported by the success of applications which use the client-server model, such as SETI@home Sullivan et al (1997), GIMPS, and DCTI (Distributed Technologies Inc., 2000). However, each of these systems fails to meet the criteria of easy deployment of software, since they all require that the owner of the client computers take responsibility for ensuring that the client software is the most recent available version. Also, these systems do not support the ability of executable tasks to migrate across the network. Rather, the executable tasks are bound to their host computers, and it is only the data that migrates across the network. Even the PVM system does not provide a mechanism for tasks to migrate across the network. Although PVM Manchek (1995) provides far more flexibility through its more general MIMD architecture, it does not provide a simple platform for development, since it still requires significant developer effort to achieve effective load balancing and concurrency control. Furthermore, all the systems examined suffer from platform-dependency issues, requiring that different versions of client software be written for different platforms (as is the case with GIMPS, SETI@home, and DCTI), or that executable tasks be written and tested in a variety of programming languages and platforms (as is the case with PVM).

Thus, there is a need for a distributed system which solves the problems of ease of development and easy deployment of software. Based on the research presented above, such a system should be based on a client-server model, using an SPMD architecture. It would need to be platform independent, in order to promote easy deployment of distributed software and to simplify the development process by shielding developers from platform specific issues. The system should also present the developer with a set of simple constructs which can be used to develop a distributed computation within the context of an SPMD architecture.

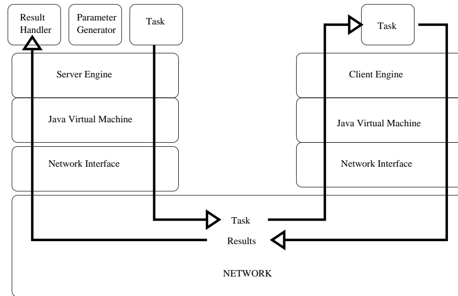

Figure 1. Main subsystems of DCL.

2 System Design

The software described in this document is intended for use by developers who wish to implement distributed computations within the context of an SPMD architecture. The software is in the form of a programming library, called the Distributed Computation Library (DCL), which can be used to distribute developer-defined tasks across a network for processing by remote clients, gather the results of these tasks together, and allow the results to be handled in a developer-defined manner.

The software is written in Java, and thus requires that all developer-defined software components also be written in Java. Furthermore, the Java Runtime Environment (JRE) version 1.2 (or later) must be installed on each machine which is expected to run the software.

2.1 Architectural Design

2.1 Overview of Sub-Systems

The main sub-systems of which the system is composed are shown in Figure 1. Below is a brief description of the responsibilities and functions of these sub-systems.

Java Virtual Machine The JVM must run on both the client and the server sides. The JVM provides the interface between the ClientEngine and ServerEngine sub-systems, and the hardware and networking platforms on which they run. The JVM is also responsible for providing the platform- independent features of the system.

ServerEngine The most important sub-system on the server-side is the ServerEngine. This sub-system runs on top of the JVM, and is able to communicate with clients via the network interface and the network. Its principle responsibility is to manage vari-

ous server-side data structures and components, as well as interacting with the three developer-defined components, namely the ResultHandler, ParameterGenerator and Task (described below). The ServerEngine also manages the communication mechanisms through which transactions between clients and the server will be conducted.

Task The Task defines the basic unit of computation for a specific application, and is developer- defined. The Task is sent to clients on request via the network. It is the only sub-system which can migrate across the network in this fashion. At the client-side, the Task is responsible for executing the algorithm of the application, using parameter sets which are passed to it by its host ClientEngine.

ParameterGenerator The ParameterGenerator sub-system is responsible for generating parameter sets, or work units, which are sent to clients, on request, via the network. The parameter sets are received by the ClientEngine and passed to the local Task sub-system for processing. The ParameterGenerator is developer-defined.

ResultHandler The ResultHandler is responsible for handling result sets, which have been processed by Tasks residing within a remote ClientEngine and have been returned to the ServerEngine. The ResultHandler is developer-defined, and should, at a minimum, record the global result of a specific computation in a fixed storage location, such as a file.

ClientEngine The ClientEngine is the most important sub-system on the client side, and is responsible for maintaining various client-side data structures necessary for its runtime functionality. It is also responsible for requesting the Task sub-system from the ServerEngine in order to create a local copy. The ClientEngine must also request parameter sets from the ServerEngine, which it passes to the local Task for processing. Once the Task indicates that the processing is complete, the ClientEngine is responsible for sending the result set back to the ServerEngine, where it is passed to the ResultHandler, as described above.

Network Interfaces and Network The network interfaces and the network are the medium across which all client-server communication takes place, and are thus critical in determining the overall performance of the system. The network must support the TCP/IP protocols.

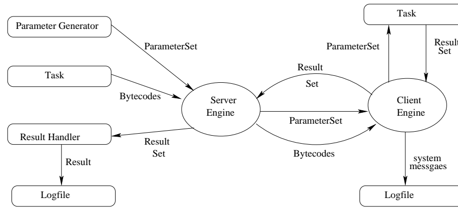

Figure 2. The inputs and outputs, and their interaction with the system.

2.2 System Model

It has already been implied in the above discussion that the overall system structure is based on a client-server model. This model was chosen due to its similarity to the SPMD architecture on which the system is based. Also, the client-server model is well-understood, easy to use, and provides a good conceptual model for the way in which a distributed computation functions in the context of an SPMD architecture (Umar 1993). The inputs and outputs, and their interaction with the system, are shown in Figure 2. The inputs to the system are all developer-defined and are specific to a particular distributed application. They must all be written in Java. The inputs are:

Task This is the executable task which will be distributed to the client computers, and must be in the form of a compiled Java class. This task should conform to the standard interface, in order for it to migrate across the network from the server to the client and still work as intended. The task represents the executable component of a particular distributed computation.

Parameter generator This is the component which generates successive parameter sets which are sent to client computers for processing, and must be in the form of a compiled Java class. The parameter sets represent the data component of a particular distributed computation.

Result handler this is the component which is responsible for handling the results sent back to the server from clients. The manner in which the results are handled is application-specific, and thus the result handler must be developer-defined. This component is also responsible for any application-specific logging which needs to take place (ie: over and above the standard logging facilities provided by the system, which serve a mainly diagnostic purpose). For example, the result handler is responsible for

storing results in an appropriate file.

Initialisation files These are used on both the client and server sides in order to set up various runtime parameters (such as server IP address, default TCP port, and logging options). These files are in the form of plain text files, and must be provided by the developer.

The system produces the following output:

Log files These files indicate any errors or failure in the system (for diagnostic purposes) as well as reflecting the correct functioning of the system (in the form of informational messages).

Result sets These are the raw, unprocessed results that are returned from clients to the server. The server passes these results to the result handler for application-specific processing.

Additional output The developer may embed additional output facilities into the result handler, allowing for customised output of results, for example writing results into a file of a specified format.

2.3 Component Design

The classes involved in the system can be broadly divided into three groups: developer-defined classes; server-side classes; and client-side classes. The latter two groups form the system itself. A brief description of the functions and design of each component is given below.

The ServerEngine class manages various server-side data structures and classes. It represents the program entry point on the server-side. When the ServerEngine is started, it retrieves its runtime parameters from a developer-defined initialisation file. The structure of this file is described in the section 4.3 below. The ServerEngine creates the ErrorLog, SystemLog and ResultsLog, as well as the PendingTasks and ExpiredTasks lists. The ServerEngine must also load the ParameterGenerator, the ResultHandler, and the Task, which are all specified by the developer in the initialisation file. At this point, the ServerEngine allows the ParameterGenerator and ResultHandler to perform any application specific initialisation. This process is described in section 4. The ServerEngine creates

and runs the ConnectionManager, and passes it the necessary information so that it can begin listening for incoming connections. Finally, the ServerEngine creates and runs a SchedulerThread.

The ConnectionManager class is responsible for listening on the server socket for incoming connections, and creating a new ServerMessageHandler to handle each new connection. It is also responsible for closing the server socket when the ServerEngine has determined that the computation is complete. The ConnectionManager must also keep track of all active ServerMessageHandlers so that it can terminate them all in an orderly fashion before closing down itself.

These two classes are responsible for keeping track of the status of all parameter sets currently being processed in the system. They do so by storing each parameter set, together with the time allocated for the completion of that parameter set, in a hashtable indexed by task ID. The task ID is a unique integer identifier which is assigned to each parameter set by the ServerEngine. PendingTasks keeps track of all parameter sets which have been allocated to a client, but for which no results have yet been returned and which have not exceeded their allocated time. ExpiredTasks keeps track of all parameter sets for which the allocated time for completion has elapsed. These parameter sets were allocated to clients which failed to return a result within the allocated time and failed to request a time extension, and are thus assumed to have crashed. When a client requests a parameter set from the server, it will be assigned a parameter set taken from the ExpiredTasks list, if possible. If the ExpiredTasks list is empty, a new parameter set will be generated by the ParameterGenerator, which will be assigned a unique task ID and entered into the PendingTasks list, before being sent to the client. This ensures that the system is able to recover from client crashes, by simply reassigning the parameter sets that a crashed client was working on to a new client.

The ServerMessageHandler class is responsible for communicating with the client, using the protocol described in Section 4. It does this by waiting for the client to request a service, and then sending the appropriate response after having consulted its host ServerEngine, if necessary. All communication is achieved by the transmission and reception of Message objects, which are described below.

The ClientEngine manages various client-side data structures and classes. It represents the

program entry point on the client-side. When the ClientEngine is started, it retrieves its runtime parameters from a developer-defined initialisation file, whose structure is described below. The ClientEngine must also create a SystemLog and an ErrorLog file for logging purposes. Unlike the ServerEngine, the ClientEngine does not create a ResultsLog, since results are only meaningful when they are gathered together at the server-side. The ClientEngine loads a special ClassLoader called the ByteArrayClassLoader. It also creates and runs the ClientMessageHandler.

The ClientMessageHandler class is responsible for communicating with the server, using the protocol described in Section 4. The ClientMessageHandler always initiates communication by requesting a service from the server. When it receives a response, it performs the appropriate action, and possibly requests another service from the server. All communication is achieved by the transmission and reception of Message objects, which are described below.

The ByteArrayClassLoader is a special ClassLoader, which is able to instantiate Java classes based on the Java bytecodes extracted from a Message object. The ByteArrayClassLoader is used by the ClientMessageHandler to dynamically create new instances of the Task objects that the server sends to the client. This mechanism of creating new classes and instantiating instances of those classes is a critical feature of the system, namely that client-side behaviour can change dynamically and under the full control of the server-side, without requiring that a new client component be written for every new kind of computation that a developer wishes to implement. Instead, the ClientMessageHandler can request a copy of the latest version of the Task object from the server and instantiate it locally. In this way, the core of the client side software remains the same from computation to computation, and is able to accommodate any arbitrary developer-defined executable Task object, without change to the main client-side core. This ability of the client to dynamically load Tasks in order to change its behaviour greatly increases its flexibility, and simplifies software distribution.

These classes provide logging facilities which are used by the system to record errors (in the ErrorLog), results (in the ResultsLog) or informational messages (in the SystemLog). They may also be used by the developer to provide additional logging services from within the ResultHandler, ParameterGenerator and Task classes. The format of the log file output can be controlled using the initialisation files (described in section 4) in order to set various

options. For example, output may be prefixed with time and date stamps, or it may be written to a specified file or to standard output, or both.

The SchedulerThread is used to provide scheduling services to classes which need to periodically perform some action. The SchedulerThread does this by setting a flag visible to its owner which indicates that a timeout has occurred, and then alerting its owner using the notify() method call. This forces the owner to wake up, examine the flag, and to perform any necessary actions based on the value of the flag. The length of the timeout used by the SchedulerThread is passed to it by its owner class, and is usually retrieved from an initialisation file. Two classes make use of the SchedulerThread, namely the ClientMessageHandler and the ServerEngine. The ClientMessageHandler must be woken periodically in order to request time extensions from the server. The ServerEngine must be woken periodically in order to examine the PendingTasks list to determine if any Tasks have expired, and if necessary to move them to the ExpiredTasks list.

The ResultHandler is a developer-defined class which is controlled by the ServerEngine, and is responsible for handling result sets that clients send back to the server. The ResultHandler should be extended by the developer to include any application specific functionality with regards to handling results, such as storing results in formatted files. Result sets are passed to the ResultHandler in the form of arrays of Objects, which were generated by a Task residing in a remote client. The exact format and composition of this array of Objects is application specific and defined by the developer.

The ParameterGenerator is a developer-defined class which is controlled by the ServerEngine. Its purpose is to generate successive parameter sets, which are sent to clients for processing by remote Task objects. These parameter sets are in the form of arrays of Objects, whose exact format and composition is application specific and thus defined by the developer.

The Task class represents the executable portion of a computation. Since its exact function is application specific, it must be defined by the developer. However, its overall function is to process parameter sets (which are generated by the ParameterGenerator on the server and sent to the client), and produce result sets (which are sent back to the server to be handled by the ResultHandler). The Task object notifies its host ClientMessageHandler when it has completed processing a parameter set, at which point the ClientMessageHandler can retrieve the result set from the Task and return it to the server.

The Message class forms the basis for all communications between the ClientMessageHandler and ServerMessageHandler. Any request or response is packaged into a Message object, which is then serialised and sent via a TCP socket to the receiving side. Each Message object consists of a message type and a content type, which describe the nature of the Message object and its contents respectively. Each Message also has a content, which is simply an array of Objects, into which the actual data is packed. The Message may also contain a timeout field and a task ID field, which are used when parameter sets or result sets are being transmitted. This allows the MessageHandlers to determine which task ID is being referred to, or what timeout has been assigned to a given task.

3 Design of Experiments and Results

Two experiments were conducted using the DCL library. There were two main reasons for conducting these experiments. Firstly, the experiments were designed to give concrete results regarding the runtime performance of DCL, and to give an indication of how well these results conform to expected results and theoretical predictions. Secondly, the experiments provided a means of examining DCL through the eyes of a typical developer, in other words treating the DCL library as a black box. This gave valuable insights into the effectiveness of the developer interface which DCL presents, and thus gave an idea of how well DCL met its design goals in terms of ease of development.

3.1 Simulated Problem

The aim of this experiment was to examine the runtime performance of the DCL library using a simulated problem, which was distributed to various numbers of clients, in order to establish how well the performance of DCL corresponded to theoretically predicted results.

A Task class, which simulated a computation by sleeping for 20 seconds, was written and compiled. In order to produce a verifiable result, the Task was also designed to add up the integers between two given numbers. The ParameterGenerator for the experiment generated these pairs of numbers such that each pair involved adding up 5 numbers, and the overall computation would result in finding the sum of the numbers from 1 to 100. Thus, there were a total of 20 parameter sets to be processed, with each one taking roughly 20 seconds to complete. This provided a means of predicting the approximate runtime that

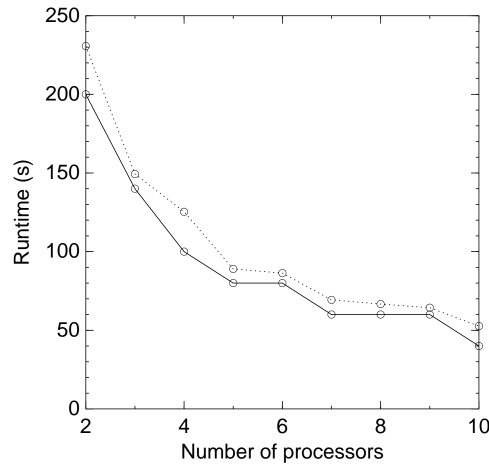

Figure 3. Runtime versus number of processors (solid line expected values).

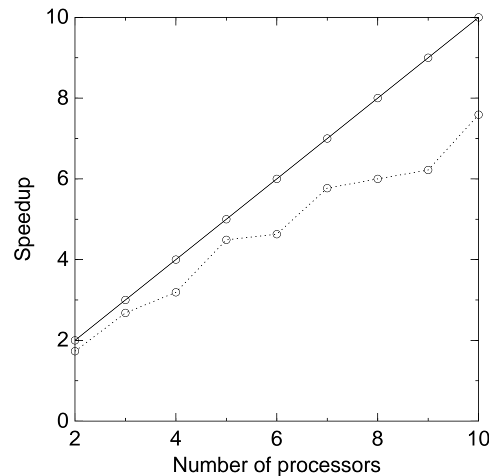

Figure 4. Speedup versus number of processors (solid line theoretical maximum).

could be expected of the system. The experiment was conducted using various numbers of processors, ranging from 2 to 10, with three trials being conducted each time. The results were then averaged over these three trials.

In the graph of runtime versus number of processors (Figure 3), the expected runtime curve should be below the actual runtime curve, since the calculation used to generate the expected runtime curve does not take communications delays into account. However, it can be expected that the two curves will be highly correlated. The runtime curves are also inversely proportional to the number of processors. However, the runtime curves begin to level off beyond 7 processors, with very little runtime performance improvement coming from each additional processor. This will be discussed further below.

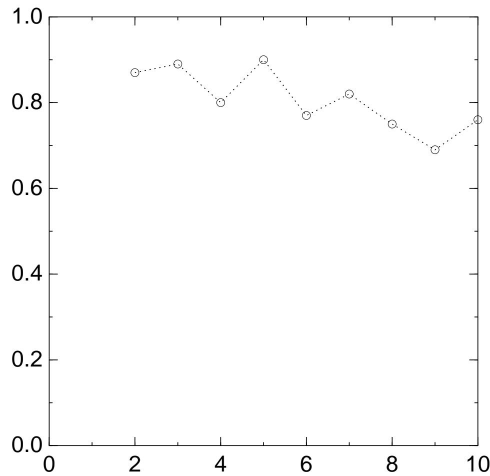

Figure 5. Efficiency versus number of processors.

In the graph of speedup versus number of processors (Figure 4), the curve representing the maximum theoretical speedup that can be attained is a straight line, since it is always equal to the number of processors being used. The actual speedup curve shown in this graph exhibits a high correlation to the maximum theoretical speedup for small numbers of processors. However, as the number of processors increases, the actual speedup curve begins to flatten out, with the difference between the maximum theoretical speedup and the actual speedup becoming larger. This corresponds well with the runtime result discussed above, namely that the incremental benefit of using more than 7 processors becomes increasingly small. This conforms to the predicted theoretical results, specifically Amdahl's Law. This law states that the speedup that can be achieved by a parallel algorithm is limited by the fraction of the algorithm which must be run in sequence. Thus, the algorithm will not be able to benefit as much from each additional processor, but will rather derive less and less benefit from each additional processor. Amdahl's Law predicts that there will come a point where additional processors will produce no benefit whatsoever. In practice, additional processors could even degrade performance, since they naturally consume resources, such as network bandwidth, which could cause the performance of the entire system to suffer.

The graph of efficiency versus number of processors (Figure 5) also shows an interesting trend, namely that efficiency drops as the number of processors rises. This is because processors can not spend all of their time doing useful computational work. Rather, some of their time is spent communicating with other processors. Thus, perfect efficiency cannot be achieved in practice. Instead, it is expected that the efficiency of a system will gradually

drop as the number of processors increases. This is due to the increase in communications overhead that each additional processors necessarily brings to the system, and hence each additional processor results in a drop in overall system efficiency.

3.2 Sorting Problem

The aim of this experiment was to examine the runtime performance of the DCL library using a real world problem, which was distributed to various numbers of clients, in order to establish how well the performance of DCL corresponded to theoretically predicted results. This experiment also provided an opportunity to examine the developer interface of DCL in more detail, and specifically in terms of ease of development.

A Task class, which was able to sort a given set up numbers into ascending order, was written and compiled. The sort algorithm implemented in the Task was a simple bubble sort, which was chosen due to its simplicity. This Task was then distributed to various numbers of client computers. Three files, each containing 400000 randomly generated integers in the range 0 to 1000, were generated. These files were read in by the ParameterGenerator, and broken up into parameter sets of 20000 integers per set, for distribution to the clients. This makes a total of 20 parameter sets. The ResultHandler gathered the result sets in the form of arrays of sorted integers, and once all 20 result sets had been gathered, it performed a final merge sort on the 20 sets to obtain a final sorted list of 400000 integers. A sequential algorithm implementing the same procedure, except performing the individual bubble sorts in sequence instead of in parallel, was also written to form a basis for comparison. The experiment was conducted using various numbers of processors, ranging from 2 to 10, and three trials were conducted each time. Each trial also used a different random number data set, in order to prevent the possibility of the algorithm being unfairly subjected to best or worst case scenarios. The runtimes over the three trials were then averaged. The same data sets were also processed by the sequential algorithm, and again the three trials were averaged.

The results of this experiment proved to be very similar to the results of experiment 1. This was an encouraging sign, since it showed that the system performance agreed with expected theoretical results in simulated and real world situations. Firstly, the graph of runtime versus number of processors (Figure 6) shows a gradual drop in runtime as the number of

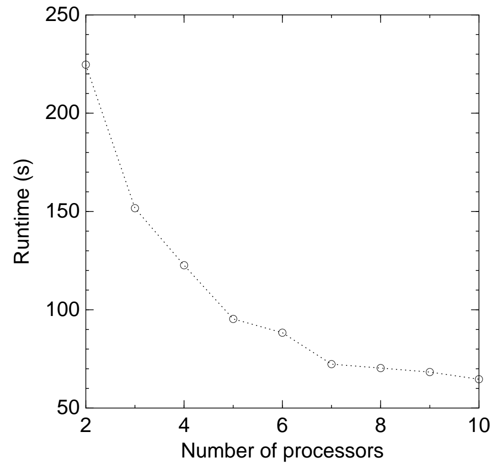

Figure 6. Runtime versus number of processors.

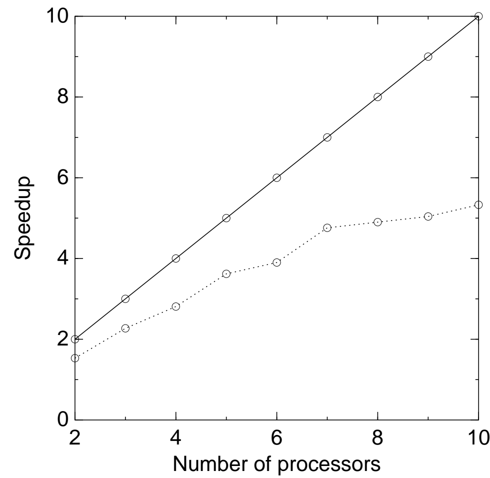

Figure 7. Speedup versus number of processors (solid line theoretical maximum).

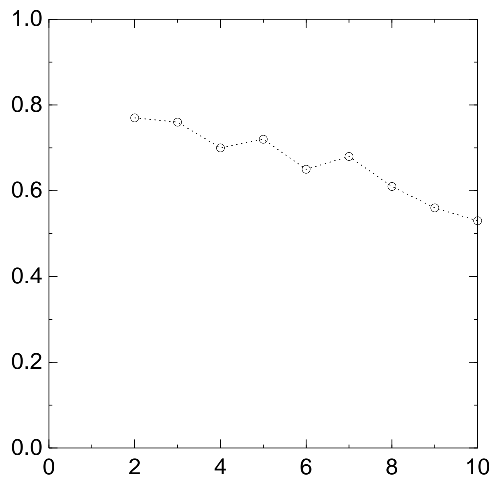

Figure 8. Efficiency versus number of processors.

processors increases, with the incremental runtime improvements becoming less and less with each additional; processor. In the graph of speedup versus number of processors (Figure 7), the difference between actual speedup and maximum theoretical speedup again becomes greater as the number of processors increases. Also, the actual speedup starts to level off after about 7 processors. Finally, the efficiency graph (Figure 8) shows the same gradual decline in efficiency as more processors are added to the system. This is due to the increased communications burden that each additional processors brings to the system, specifically the server, which begins to act as a bottleneck. All of these trends agree well with the theoretically predicted results for a distributed computation.

Interestingly, in both the simulated and the real problem, the speedup curves tended to level off at between 6 and 7 processors. This would indicate that the system exhibits some degree of reliability in terms of performance. However, performance is highly application specific, and thus this result should be seen merely as an accidental correlation between the two experiments.

4 Conclusion

The problem identified in this paper is that the currently available distributed systems and programming libraries do not provide developers with easy platforms for development, do not allow client behaviour to be changed dynamically, and do not promote easy deployment of software. Yet, these libraries and environments are becoming increasingly important in the modern computing environment, with the rapid growth in network technologies. Thus, a programming library which addresses these issues is needed, so that developers may easily implement a distributed computation. Such a library should shield the developer from platform specific issues which may hinder the development process, and should provide a simple and complete set of high level constructs with which a developer can implement a distributed computation. Furthermore, the library must support the automatic update of the client to ensure that the latest possible task available from the server is being executed. This will result in a far simpler software deployment mechanism than systems such as GIMPS, DCTI, and PVM currently support.

The goals of this paper were to develop a library which met those requirements. Specifically, the goals were to produce a library which provided a simple development platform for the

implementation of distributed computations, which allowed client behaviour to be updated dynamically, and which was platform independent to promote easy deployment of software. This resulted in the development of the Distributed Computation Library (DCL).

In Section 3, two experiments were conducted, whose purpose was not only to gather empirical data regarding the runtime performance of DCL, but also to determine to what extent DCL met its design goals. It was concluded that DCL met the goals of the paper in several ways. Firstly, DCL only requires the developer to use three classes in order to develop a distributed computation. The experiments presented in Chapter 4 show that these three classes form a complete set of tools for implementing a distributed computation. The classes are also designed in such a way that the developer is only required to overwrite a maximum of 2 methods in each of these classes in order to produce a valid distributed computation for DCL. Since DCL is entirely written in Java, the developer is also required to use Java, and this brings to DCL the property of platform independence. This not only simplifies the development process by shielding the developer from platform specific issues, but also promotes easier software deployment, since developers no longer have to write different versions of their client software for each platform in the network. Also, due to Java object serialisation and ClassLoaders, the client side components of DCL are able to dynamically load Java class files which have been retrieved from the server, and instantiate local copies of those classes. This means that the client side components of DCL can dynamically change their behaviour, thus ensuring that the clients are always executing the latest possible version of the distributed computation provided by the developer. Thus, DCL has successfully met all the design goals originally specified.

5 References

Ajith Tom, P. and Siva Ram Murthy, C., 1998. Algorithms for Reliability-Oriented Module Allocation in Distributed Computing Systems. Journal of Systems and Software, Vol 40, pp125-138.

Alexandrov, A. D., Ibel, M., Schauser, K.E. and Scheiman, C.J., 1997. SuperWeb: Research Issues in Java-Based Global Computing. Department of Computer Science, University of California.

Blelloch, G. E., 1996. Programming Parallel Algorithms. Communications of the ACM, Vol 39, No 3, pp85-88.

Codenotti, B. and Leoncini, M., 1993. Introduction to Parallel Processing. Addison-Wesley Publishing Company, Great Britain.

Cristian, F., 1996. Synchronous and Asynchronous Group Communication. Communications of the ACM, Vol 39, No 4, pp88-97.

Distributed Computing Technologies Inc., 2000. distributed.net: Node Zero <http://www.distributed.net>

Flynn Hummel, S., Ngo, T., and Srinivasan, H., 1996. SPMD Programming in Java. IBM T.J. Watson Research Center

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V., 1994. PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press

Harold, E. R., 1996. Java Network Programming, O'Reilly & Associates Inc., Sebastopol.

Hayes, B., 1998. Computing Science - Collective Wisdom. American Scientist, March-April.

Kumar, V., Grama, A., Gupta, A., and Karypis, G., 1994. Introduction to Parallel Computing - Design and Analysis of Algorithms. Benjamin/Cummings Publishing Company, Inc., California.

Norton, C. D., Szymanski, B.K. and Decyk, V., 1995. Object-Oriented Parallel Computation for Plasma Simulation. Communications of the ACM, Vol 38, No 10, pp88-100.

Object Management Group, 2000. CORBA White papers. <http://www.omg.org/library/whitepapers.html>

Purao, S., et al., 1997. Effective Distribution of Object-Oriented Applications. Communications of the ACM, Vol 41, No 8, pp100-107.

Raynal, M., 1988. Networks and Distributed Computation - Concepts, Tools, and Algorithms, MIT Press, Cambridge.

Schmidt, D. C. and Fayad, M., 1997. Building Reusable Object-Oriented Frameworks for Distributed Software. Communications of the ACM, Vol 40, No 10, pp85-87.

SETI@home, 2000. Search for Extraterrestrial Intelligence at Home.
<http://setiathome.ssl.berkeley.edu>

Sinha, A., 1994. Client-Server Computing: Current Technology Review. Communications of the ACM, Vol 37, No 4.

Stallings, W., 1996. Computer Organization and Architecture - Designing for Performance 4 ed., Prentice-Hall, New Jersey.

Stout, Q. F., 1992. Ultrafast Parallel Algorithms and Reconfigurable Meshes. Proc DARPA Software Technology Conference, pp181-188.

Sullivan, W. T., Werthimer, D., Bowyer, S., Cobb, J., Gedye, D. and Anderson, D., 1997. A new major SETI project based on Project Serendip data and 100,000 personal computers. Proc. of the Fifth Intl. Conf. on Bioastronomy, No 161.

Tiemeyer, M. P. and Wong, J.S.K, 1998. A task migration algorithm for heterogeneous distributed computing systems. Journal of Systems and Software, Vol 41, pp175-188.

Umar, A., 1993. Distributed Computing and Client-Server Systems, Prentice Hall, Englewood Cliffs.

Wollrath, A., Waldo, J. and Riggs, R., 1997. Java-Centric Distributed Computing. IEEE Micro, Vol 17, No 3.

Woltman, G., 2000. Great Internet Mersenne Prime Search.
<http://www.mersenne.org>