GUCHA: An Internet-based Parallel Computing System Using Java

L. F. Lau, A. L. Ananda, G. Tan, W. F. Wong
School of Computing
National University of Singapore
3 Science Drive 2, Singapore 117543
{laulf, ananda, gtan, wongwf}@comp.nus.edu.sg

Gucha[*] is a system developed for harnessing the immense computational resource available in the Internet for parallel processing. In this paper, existing systems that make use of Java for network parallel computing are presented and categorized. Next, the requirements and goals of an effective parallel computing system in the Internet environment are singled out. These serve as the blueprint for the development of the Gucha system. Its infrastructure and features, namely *ease of use*, *heterogeneity*, *portability*, *security*, *fault tolerance*, *load balancing* and *scalability* are discussed. Lastly, experimental results from the running of several parallel applications in the Gucha environment are presented. Based on the results, the types of parallel applications that would be well suited for running in Gucha are identified.

Keywords: Java, parallel processing, load balancing, task placement, heterogeneous computing system

1. Introduction

The exponential growth of Internet in recent years has linked tens of millions of computers together. The combined computational power of even a small fraction of this pool of machines, which are idle most of the time, is many times more than what a central parallel supercomputer can offer. In fact, there have been many successful attempts to solve complicated mathematics and scientific problems by tapping this gigantic computational resource. However, harnessing the resources of Internet-based computers for parallel computing is by no means an easy task. It introduces new difficulties and problems that have never been addressed by parallel computing in LAN (local area network) environment.

Gucha is developed to provide Internet-based parallel computing with the objective of addressing the difficulties that such a computing environment might impose. In particular, the project aims to create an infrastructure to support parallel computations in an Internet-based

[*] Gucha in Sanskrit language means cluster.

environment and provide programming interface that could enable programmers to develop parallel applications with ease.

In the next section, existing implementations of parallel computing systems using Java are given. Critiques of these implementations follow. In section 3, the design overview of Gucha and the requirements for effective parallel computing in Internet environment are discussed. This is followed by a discussion of the implementation of Gucha in section 4. Next, experimental results based on the running of parallel applications in Gucha environment are presented.

2. Parallel Computing Systems Using Java

Java, a programming language designed from ground up with networking in mind, is set to become one of the most important and widely used languages in the Internet era. Its popularity has surged with the phenomenal growth of Internet. With its cross-platform, secure, object-oriented, and network-centric features, it is ideally suited to address the issues of parallel computing in Internet-based environment.

With the powerful features of Java, it is little wonder that a significant number of systems have been developed to support network parallel programming using Java. Basically, these systems can be classified into the following categories:

2.1 Parallel Computing Systems based on Java Applets

In these systems, applications are broken down into smaller tasks and downloaded, in the form of applets, by client machines volunteering their computational resources. This is simply done by directing the web-browsers at client machines to the URLs of the web servers where the applications are run. After execution at the client machine, the computed results are returned to the server machines where the applets were downloaded. Examples of such systems are Javelin [2], Bayanihan [3,8], Charlotte [1] and DAMPP (Distributed Applet-based Massively Parallel Processing) [7].

The use of applets has allowed any machine, connected to the Internet and equipped with a Java-enabled web browser, to join any ongoing computation with ease. Due to Java’s platform independence, the platform of the participating machine is not an issue at all. Effort required for the setting up of such a parallel computing system is minimal.

However, due to security restrictions of applets (such as being only able to open network connections to the server from which they are downloaded), such systems are only suitable for supporting applications that can be decomposed into non-communicating, coarse-grained and totally independent functional tasks. In short, only *master-slave* style of parallelism can be achieved. In most of the above systems, the web server is the only contact point for volunteers to download the applets. High congestion, in terms of system and network load, may be experienced at the server and it can easily become a bottleneck for the entire computation. Single point of failure and inscalability of such systems are problems not to be dismissed.

2.2 Parallel Computing Systems based on Standalone Java Applications

Systems supporting network parallel computing as standalone Java applications can be sub-categorized into two groups: one that is based on Java Native Methods mechanism and the other that is not. Native methods refer to methods, called by Java programs, but are written in languages other than Java. An example of a system that uses Java Native Methods mechanism is jPVM [6]. By providing native method wrappers around the existing standard PVM routines, jPVM allows Java applications and existing C, C++ and Fortran applications to communicate with one another using the PVM API. Using jPVM, programmers can link existing PVM applications in Java graphical user interface front ends. Another system that uses Java Native Methods mechanism is IceT [4]. However, unlike jPVM, it allows resources on machines to be made available to users who do not have log-on privileges.

Other systems are implemented entirely in Java and make use of the socket interface in the standard API for communication among parallel units. One example is JPVM [5]. It is not interoperable with PVM but provides Java implementation of routines found in the PVM library. Another example is MPIJ [9], a pure Java implementation of Message Passing Interface standard.

Systems implemented as standalone Java applications do not have the same restrictions in network communications as in the case of systems using Java applets. Peer-to-peer communications among participating machines are possible. Those that make use of native methods mechanism also have a clear advantage of software reuse and performance. However, the use of native methods assumes that the underlying platform already has support for the native methods. Consequently, such systems will face portability problems.

Implementations of the above systems, with the exception of IceT, only allow users to utilize resources of those machines where they have access privileges. Usually, certain assumptions of the underlying file systems are also made. Hence, the setting up of such systems for large-scale parallel computations, based on computing resources from volunteers in the Internet, would be an uphill task.

2.3 Weaknesses

The designs of most existing parallel systems are catered for computing in a small network setup, such as LAN. Hence, issues such as scalability, portability, fault tolerance and security are not addressed. These weaknesses will handicap their deployment as parallel computing tools for large-scale computations in Internet-based environment.

3. Gucha: Design Overview

Gucha is a 100 percent pure Java implementation [11] and not based on native methods mechanism. Based on the above classifications, Gucha fits into the second category of standalone Java application. It is designed for Internet-based parallel computing and aims to address the weaknesses of existing implementations mentioned above. Its design is as follows:

3.1.1 Ease of Use

This involves two groups of users – the programmers who develop the parallel applications and the volunteers who contribute their machines for the computations. From a programmer's perspective, the programming interface must be flexible and simple. Developing parallel applications should be done with as much ease as sequential applications. In Gucha, a set of features-rich networking APIs that will shield the programmer from the complexity of the underlying network communications is provided. The collection of heterogeneous machines in the network would appear as a single high-performance parallel machine.

From a volunteer's perspective, contributing his machines to participate in computations should be simple and hassle-free. No technical knowledge of the program to be run should be required on his part. As mentioned earlier, in some Java-based computing systems, to volunteer simply means visiting a web server site using a Java-capable browser. However, due to communication restrictions of applets, such an approach is not adopted in Gucha. Instead, users are required to run a program at their machines to join the pool of volunteers. Such an approach is no more difficult than visiting a website as no other effort is required on the part of the users.

With Java’s code mobility, Gucha supports dynamic loading of applications from the network to be executed at the volunteers’ machines. The loading and execution of applications are transparent to the volunteers. It is very much like the dynamic loading and execution of applets, but without their limitations.

3.1.2 Heterogeneity and Portability

The Internet is a collection of machines of heterogeneous platforms and architectures. It will be an uphill task for the programmer to write codes for different systems and distribute the compiled binaries to the systems. As Java is architecture neutral, Gucha, being a 100% pure Java implementation, is portable and will operate faultlessly on any platform as long as Java JDK1.2 is supported.

3.1.3 Security

This involves two aspects. Firstly, from a volunteer’s perspective, by contributing his machine in a computation, he must have the assurance that it will be protected against malicious attacks (e.g. virus attack). In a LAN setup, this is easily achieved by maintaining users access rights and accounts, so that the network is available only to trusted users. Although the security of volunteer’s machine is enforced to a certain extent, the setting up of such a computing environment would be tedious and introduce problems, such as scalability. In Gucha, no assumptions of access privileges of machines participating in computations and their underlying file system are made.

The second aspect involves computational security. Besides protecting volunteers, the environment must also ensure that the computation itself is secure from malicious attacks from volunteers. Preventing volunteers from *spying* on computations, which may be confidential, is also another concern. In Gucha, enforcing secure computations is a prime concern. To achieve this, cryptographic and authentication mechanisms for network communications among machines in the system are implemented.

3.1.4 Fault Tolerance

An Internet-based computing environment is governed by the following characteristics:

1) Uncertain availability of machines. In the Internet, machines join and leave a computation at the whim of volunteers. There is no control as to when a volunteer should join a computation and how long he should remain committed to it.

2) Unpredictable network. The load of the Internet traffic and faulty network links are the major contributing factors.
3) Unreliability of volunteers. Based on what we have discussed on security, the reliability of the computed results from volunteers' machines is questionable.

The Gucha system is designed to recover from unintentional faults caused by the first two characteristics. It polls machines in the system at regular intervals to check whether they are still "alive". In the event that a volunteer leaves the system abruptly, Gucha is able to off-load the failed computation to another machine such that the overall computed results would not be affected. This implementation will be elaborated in the next section.

Intentional faults caused by the third characteristic are harder to tackle. Known techniques, such as *replication* and *spot-checking* [8], to detect and recover from these faults are inefficient and unreliable. In Gucha, no solution is offered to counter this problem yet. Instead, through tight security enforcement, occurrences of such faults caused by rogue volunteers are minimized, if not avoided.

3.1.5 Network Load Balancing

A system with support for network load balancing should be able to detect congested network links and avoid assigning computations to machines reachable by those links, whenever possible. To achieve this, in Gucha, all the volunteers in the system are polled at regular intervals to measure the network transfer delays. This technique, called *bandwidth probing*, will be discussed in detail in the next section.

3.1.6 Scalability

To tap the computational resources of tens of thousands of machine in the Internet for parallel computing effectively, the scalability of the system is an important issue. The communication architecture (or topology) supported by the system is a deciding factor in its scalability. In Gucha, the architecture adopted is a hybrid between the centralized and decentralized one. The main benefits of this architecture are its abilities to reduce message traffic exchanges (as compared to a decentralized design) and improve fault-tolerance (as compared to a centralized design). An illustration of this communication architecture is given in Figure 1.

3.2 Network Communication Architecture

The design of the network communication architecture of a computing system will directly, as well as indirectly, affect the implementation of the above design. As shown in Figure 1, Gucha adopts hierarchical network communication architecture consisting of four entities: *director*, *coordinator*, *volunteer* and *client*. To prevent the diagram from getting too complicated, the communication links between volunteer and director, and that of client and director, are not shown.

The main function of the director is to provide lookup service for coordinators information in the Gucha system. As the name implies, coordinators serve as the middlemen between volunteers and clients during task distribution. Volunteers are machines that offer their resources for parallel computations and clients are machines that tap these resources.

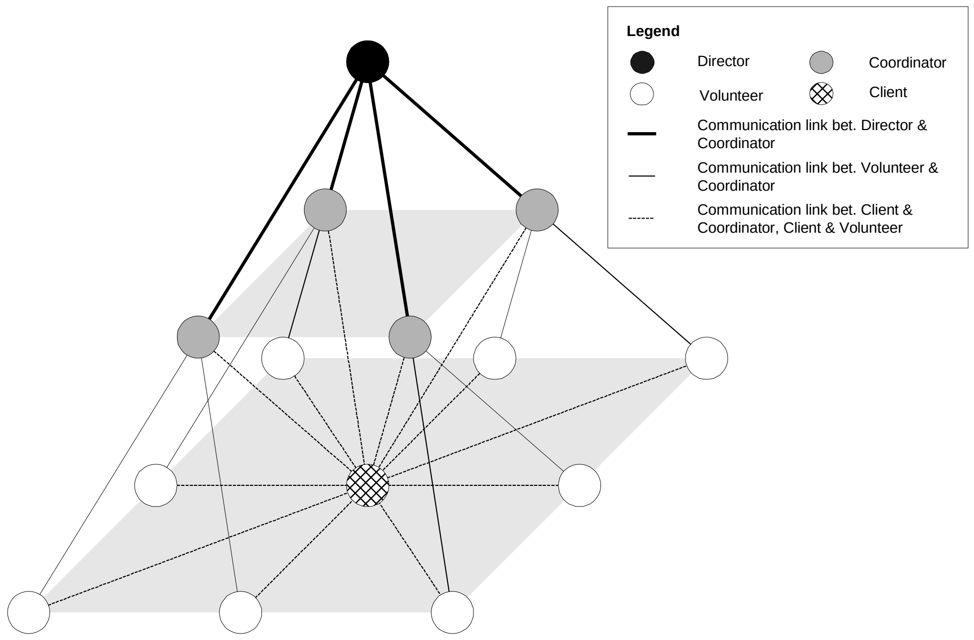
Legend
Director
Coordinator
Volunteer
Client
Communication link bet. Director & Coordinator
Communication link bet. Volunteer & Coordinator
Communication link bet. Client & Coordinator, Client & Volunteer

Figure 1. Network Communication Architecture of Gucha

3.3 Scenario Walkthrough

A typical scenario of how the entities interact with each other is shown in Figure 2. The sequence of interactions is as follows:

(1) When a coordinator is first started up, it will register with a specified director in the Gucha system. During registration, it will supply a password to the director for verification purpose.

(2) On receipt of the registration request, the director will check the information supplied by the coordinator against its database of legitimate coordinators. If the coordinator passes the check, it will be added to the director's list of active coordinators.

(3) When a volunteer is first started up, it will contact the director for information of the active coordinators in the Gucha system.

(4) The director returns the list of active coordinators that have registered with it.

(5) From the list, the volunteer selects one coordinator, by random, for registration. During registration, the hardware information and lease time for volunteering is supplied.

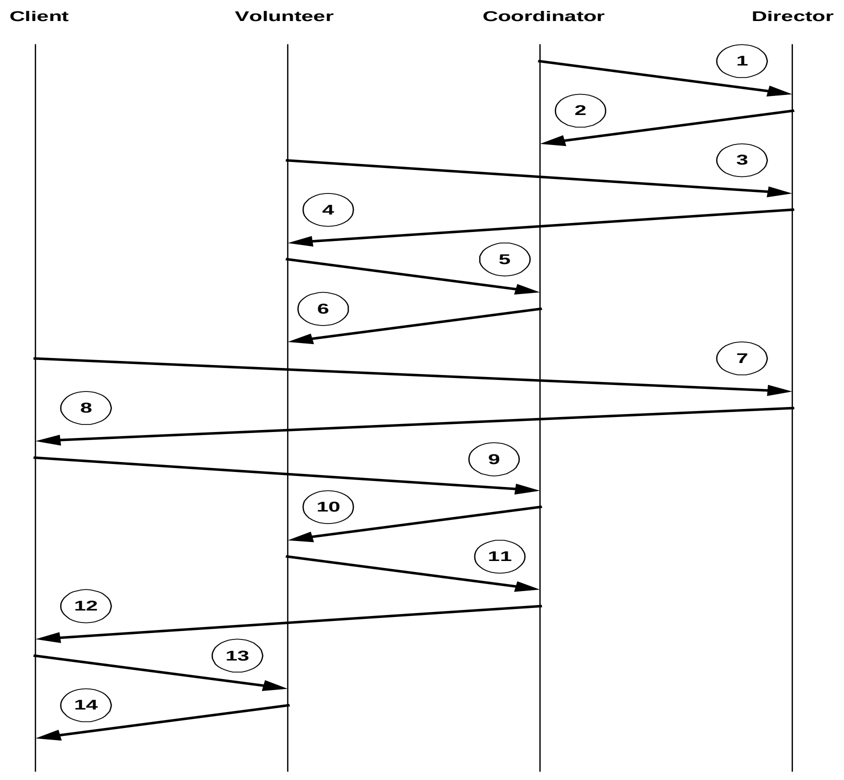

Figure 2. Scenario Walkthrough

(6) On successful registration, the coordinator will add the information supplied by the volunteer to its database of registered volunteers. A random number (called the volunteer-id) will be generated and the volunteer is informed. This id will be used in subsequent communications between the volunteer and the coordinator for identity checking purpose.

(7) When a client is first started up, it will contact the director for information of the active coordinators in the Gucha system.

(8) The director returns the list of active coordinators that have registered with it.

(9) The client will contact the first coordinator, in the returned list from director, for volunteers. In its request, it will supply a password for verification purpose and also the number of volunteers needed. If insufficient volunteers are returned, it will contact other coordinators in the network until all the volunteers required are found.

(10) On receipt of the request for volunteers, the coordinator will check the information supplied by the client against its database of legitimate clients. If the client passes the check, the coordinator will select a list of potential volunteers (based on the number of volunteers requested) for allocation to the client. For each of these volunteers, a random number (called the client-id) is generated. The coordinator will then inform every volunteer of its allocation and the client-id to be used during interactions with the client.

(11) The volunteer will acknowledge the receipt of the allocation notice by sending a confirmation reply to the coordinator.

(12) The coordinator will send the list of allocated volunteers and their corresponding client-ids to the client.

(13) With information received from the coordinator, the client will send task placement requests to the volunteers. Information included in the requests comprises the client-ids, arguments and bytecodes of the application to be run.

(14) The volunteer will check whether the client-id supplied by the client matches with that given by the coordinator earlier. If they match, the volunteer will "class-load" the bytecodes supplied for task execution. Result of the execution is then sent back to the client.

3.4 Constraints

The design of the Gucha system can only achieve master-slave style of parallelism. This implies that volunteers participating in the execution of a parallel application would not be able to communicate with each other during computation. Consequently, Gucha can only handles applications that can be broken down into non-communicating and independent functional tasks. However, as Gucha is a non-applet based implementation, it can be enhanced further to support peer-to-peer communications among volunteers and parallel applications based on tree computing model.

Communication between two entities in Gucha system involves encryption of data at the sending party and decryption by the receiving party. In addition to this, overhead caused by network transfer of data between the parties will also have a negative impact on the overall performance of the application. To offset the impact caused by the overheads, the tasks sent to volunteers during task placement should be of large-grained size.

4. Gucha: Implementation

This section describes the implementation of the Gucha system.

4.1 Scheduling Algorithm

To support network load balancing, Gucha implements a scheduler such that volunteers with the highest capability are selected for task execution. The description of the scheduling algorithm will be based on the *transfer policy*, *information policy* and *placement policy* adopted.

4.1.1 Transfer Policy

This policy decides when and how an attempt should be made to transfer an arriving task to another host. The client will have *n* number of tasks for distribution to volunteers in the network for processing. It will contact a coordinator in the Gucha system for *n* number of volunteers for the processing. The coordinator will search its database of registered volunteers and select *n* (if possible) most capable volunteers for allocation to the client. However, under circumstances when the coordinator is unable to meet (either fully or partially) the client's demand, the client will contact other coordinators in the network for the remaining number of volunteers required. The process repeats itself until all *n* volunteers are found.

4.1.2 Information Policy

This policy is concerned with deciding what kind of information is to be collected, when, where and how it should be collected for scheduling purposes. During registration with the coordinator, a volunteer will supply its hardware information (CPU speed, memory capacity and disk capacity) and the lease time for contributing its resources. The coordinator stores this information in a *VolunteerNode*. The coordinator also maintains a linked list of *VolunteerNode*s (known as the *VolunteerList*) which contains information of all volunteers that have registered with the coordinator. At regular intervals, the coordinator will do *bandwidth probing* on every volunteer by prompting them to send certain length of byte stream over. The network transfer time (called *netDelay*) of each volunteer is measured from the moment that the volunteer is prompted till the moment the byte stream is received by the coordinator. Failure of a volunteer to respond to the coordinator's prompting will signify that it has been disconnected from the network. Consequently, it will be removed from the linked list.

It is clear that information of all the volunteers in the network is not kept centrally in one coordinator. With a distributed design by use of multiple coordinators, single point of failure is avoided. With different clients consulting different coordinators for volunteers, the likelihood of any coordinator becoming a bottleneck is also reduced.

It should also be noted that the coordinators do not exchange information of its volunteers among themselves. This is to reduce communication traffic among the coordinators. The Gucha system is designed for Internet-based computing with large number of participating volunteers. If all the coordinators were to update each other of every change of network status of each of their volunteers, the frequency of communication exchanges will be very high.

4.1.3 Placement Policy

This policy decides the host to which a task should be transferred for execution. Upon receipt of requests from clients for volunteers, the coordinator will select the best volunteers in its *VolunteerList* for tasks placement. Only volunteers with the top *capability* values will be selected. The derivation of capability value for each volunteer is based on *netDelay* (the network delay information collected from bandwidth probing), and the hardware information of the volunteer (collected during registration). The formula used is as shown in Figure 3.

$$\mathbf{CP = W_1{*}CPU_N + W_2{*}MEM_N + W_3{*}DISK_N + W_4{*}NET_N}$$

where CP stands for Capability value of the volunteer

W_1, W_2, W_3, W_4 are weightages of the respective resources. To be decided by coordinator.

NB: W1 + W2 + W3 + W4 = 1

CPU_N, MEM_N, $DISK_N$, NET_N are the normalized values of the CPU speed, memory capacity, disk capacity and network load of the volunteer respectively.

NB: Each value is derived from the normalization of the resource value against the best possible value in its category.

Figure 3. Capability Value Derivation for Volunteer

The *VolunteerList* is sorted, based on the *capability* value of each *VolunteerNode*, in an ascending order. This is to facilitate the selection of the best volunteers during task placement. Reasons for choosing network bandwidth as a parameter for placement policy are as follows:

1) In the Gucha system, the participating hosts may be geographically distributed in different parts of the world. Consequently, the network communication delay among the hosts will be quite significant, as compared to a LAN setup. In view of this, if system load were to be used as a factor for volunteer selection, the load information from the volunteers that arrives at the coordinator may be obsolete and may not truly reflect the load state of the volunteers at that point in time.

2) In the Gucha system, bytecodes of the applications from the client will be transferred to the selected volunteers for execution. This may involve a huge amount of data transfer. If the links to the volunteers are congested or of low bandwidth, the transfer may take up a significant amount of time and hence, defeat the purpose of distributed processing.
3) Although the *netDelay* is a measure of the link status between a volunteer and the coordinator, it can also be viewed as an indicator of the load of the volunteer. A lower loaded host will respond faster than a heavier loaded one during bandwidth probing. Therefore, the lower *netDelay* value of the volunteer recorded at the coordinator will increase its chance of selection for task placement, over its heavier loaded counterparts.

The use of the formulae for deciding the best volunteers for task placement allows the deployment of Gucha in different environments. In the Internet environment, which Gucha is designed for, more emphasis should be put on the weighting for the *netDelay* parameter. However, for deployment in a LAN environment, whereby the network is more stable but participating machines are of varying hardware capabilities, the weightings for system resources (*CPU*, *MEM* and *DISK*) should be increased and that for *netDelay* decreased accordingly.

4.2 Class Loading

The heterogeneous nature of the Internet does not allow Gucha to make any assumption of the underlying file systems and access privileges to systems. As such, to utilize the resources of a volunteer machine, classes (in the form of bytecodes) of the task to be executed will have to be dynamically transferred from the network to the volunteer during run-time. This mobility of codes is achievable in Java by a mechanism called *class loading*.

In Gucha, we have implemented a class loader, called *GuchaClassLoader*, to take care of the loading of task bytecodes, from the client across the network, and dynamically linking them with the run-time system of the volunteer. The *GuchaClassLoader* starts by being a subclass of **java.lang.ClassLoader**. An overview of this implementation is as follows:

- Using the default system class loader, check if the class to be loaded is a system class.
- If it is not, attempt to fetch the class from the classloader’s class repository.
- If failed, get the bytecodes of the class from the *byteCodesTable,* which is sent by the client across the network.
- Perform bytecodes verification and convert the bytecodes into a class object.

With the help of the *GuchaClassLoader*, the volunteer will load all the necessary classes needed for the task execution. Upon completion, the final result obtained is then returned to the client.

4.3 Security

Gucha aims to provide an Internet-based computing environment that ensures security. On top of the existing protection provided by the Java language itself, Gucha offers encrypted communication exchanges among coordinators, volunteers and clients. Every host in Gucha possesses a common *secretkey* that is used for the encryption and decryption of messages. Hence, even when the communication path is tapped, the information that could be viewed by an intruder is useless.

Gucha has also implemented authentication by the use of identification numbers and passwords, which the sending hosts must provide to the receiving hosts. It is only after verification that the identification numbers and passwords match with those at the receiving hosts that the incoming connections are accepted. With this feature, computational resources of volunteers are guarded against unauthorized clients.

4.3.1 Cryptography in Gucha

In Gucha, DES (Data Encryption Standard), a symmetric cipher, is used for the encryption of communication messages. For secure communication in Gucha, a message at the sending party is first stored as a *serializable* object. Before sending out to the receiver, the object is encrypted by making use of the **javax.crypto.SealedObject** utility class in JCE 1.2. The *SealedObject* constructor wraps around the supplied serializable object and encrypts it with a supplied cipher. At the receiving party, the *getObject*() method is used to retrieve the original, unencrypted object with the same cipher.

4.4.1 Programming Model and Developer Interface

One of the objectives of Gucha is to provide a programming interface that can help developers to program parallel applications with as much ease as sequential ones. Figure 4 gives an illustration of how simple developing parallel applications in Gucha could be. The programming model encompasses the following steps:

(1) **Create the environment.** In the background, the client contacts the director for the available coordinators in the Gucha system. An empty linked list (called the *TaskPool*) is also generated for the storage of tasks that will be generated.

(2) **Generate tasks to *TaskPool*.** The number of tasks generated depends on the number of volunteers specified by the developer for execution of the application. The tasks generated, in the form of arguments for the application, are stored in the *TaskPool*. For each task created, the developer will be issued a *task-id*, which is needed in the retrieval of result at stage (4).

```
public static void main (String[] args) {
        ….
 //  (1) Create the environment
  Gucha_Env env = new Gucha_Env(className, appnClassNames,
               "SecretKey.ser", director, password);
  ….
  //  (2) Generate tasks to TaskPool
  for (int i=0; i<numVolunteers; i++) {
     ….
     tidarray[I] = env.AddToTaskPool((Object) task)
  }
  ….
  //  (3) Distribute tasks to volunteers
  env.SendTasks();
  ….
  //  (4) Collect computed results
  for (int i=0; i<numVolunteers, i++) {
    ….
    result[i] = env.getResult(tidarray[i], waitTime);
  }
  ….
   // (5) Stop all unfinished tasks at volunteers
   env.StopTasks();
}
```

Figure 4. Gucha Programming Model

(3) **Distribute tasks to volunteers.** In the background, the client contact the coordinators to get the volunteers needed. It will then transfer the bytecodes of the application and the respective argument to each of the volunteers found by invoking the *TaskPlacer* thread. The client might not be able to get all the volunteers it needs and the existing volunteers allocated might leave the system abruptly without finishing the task allocated. As such, the client will invoke the *VolunteerHunter* thread, which will hunt for new volunteers from the coordinators whenever there is a shortage. It will also invoke the *VolunteerAliveHandler* thread to check, at regular intervals, whether the volunteers allocated with tasks are still alive. Detection of failed volunteers will spin off the whole process of finding new volunteers and transferring of tasks, once allocated to those failed volunteers, to the new volunteers. This recovery mechanism is automatic and transparent to the developers. At this stage, for collecting results back from the volunteers, the client will also invoke the *ResultCollector* thread.

(4) **Collect computed results.** Results computed by the volunteers will be received by the *ResultCollector* thread and stored. The coordinator will also be informed to release allocations of the volunteers held by the client. To retrieve the computed results, the developer will have to supply the necessary *task-id*s issued earlier at stage (2).

(5) **Stop all unfinished tasks at volunteers.** This step is optional as it depends on the application's need. If the developer has acquired the necessary results, it can terminate existing execution of tasks at all volunteers by invoking the *StopTasks()* method. For applications that need to collect back all computed results, such invocation will not be necessary.

5. Gucha Performance

Experiments are conducted to measure the performance of the Gucha system. Results collected from the experiments are based on the running of parallel applications ported to Gucha's platform. In the experiments, a homogeneous collection of 64 Pentium PCs, running WinNT OS, is used to form the volunteers cluster. To show heterogeneity of the Gucha system, machines of different hardware configurations and OS (UltraSPARC, Solaris 2.7 OS) are chosen to form the director, coordinators and client. All the machines are connected to the network via 100Mbps links. The experiments are conducted when the load of all the machines and the network are low.

5.1 RC5 Cracking

RC5 [12] is fast block cipher that uses a parameterized algorithm with a variable block size, a variable key size and a variable number of rounds. Typical choices for the block size will be 32, 64 or 128 bits. The number of rounds can range from 0 to 255 and the key can range from 0 to 2048 bits in size. The encryption routine of RC5 consists of three primitive operations: integer addition, bitwise exclusive-or, and variable rotation. The heavy use of data-dependent rotations and the mixture of different operations provide the security of RC5.

5.1.1 Experimental Results

For this, a sequential RC5 cracking program [13], for uncovering the 32-bits secret key used to encrypt a message, is ported over to Gucha. The program adopts a "brute force" search on all possible keys until it finds the one that decrypts the message. Time ($\mathbf{T_{seq}}$) needed to run the program sequentially on one volunteer is measured.

The Gucha-ised version breaks up the key space to search into smaller ones (of equal size) and distributes them to the volunteers so that the search can be performed in parallel. For this experiment, the program is run with different numbers of volunteers in the network. Time (**T_{dis}**) is measured from the moment the client creates the Gucha environment to the moment when the secret key is found by one of the volunteers and the result returned to the client. The speedup (**T_{seq}/T_{dis}**) for each of above experiments is then computed. Result is as shown in Figure 5.

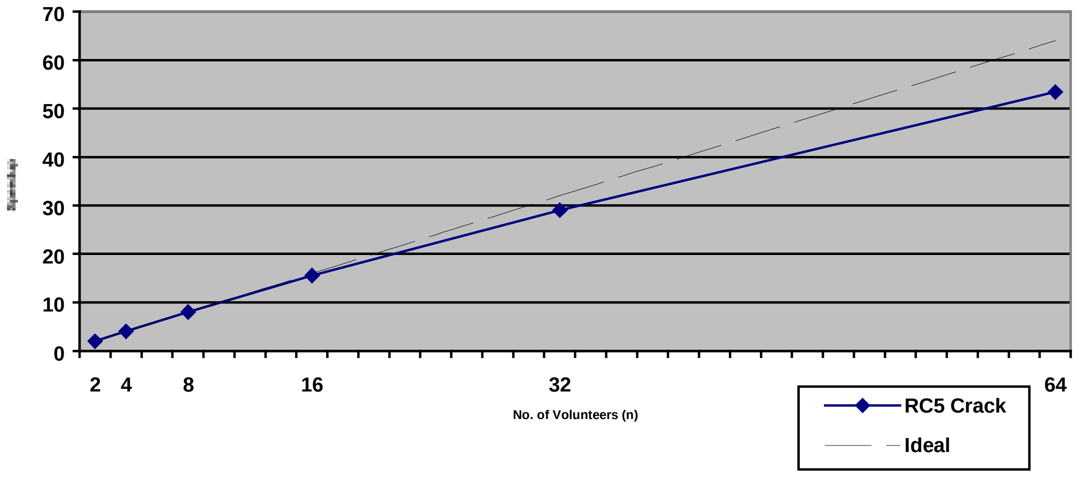

Figure 5. RC5 Cracking Experimental Results

5.1.2 Performance

The speedup curve (in Figure 5) obtained for Gucha-ised parallel cracking is very close to the ideal situation. Characteristics of this application are as follows:

- Highly computational intensive.
- Low bandwidth requirement. The bulk of the network communications among the entities in the system is only during task distribution and result collection stage. In these stages, the amount of data exchanged is usually not high.
- All computations end when the client receives the first result from a volunteer.

5.2 Ray Tracing

Ray tracing is a method that creates photo-realistic synthetic images from a mathematical description of the scene to be generated. Computations involve millions of floating-point operations and consume large amount of processor time even with scenes of modest complexity. Fortunately, each ray in the algorithm is independent of each other, making ray tracing an ideal candidate for parallel processing.

5.2.1 Experimental Results

For our experiments, a sequential Java Ray Tracing program [10] is ported to Gucha. The program is run on a standalone machine to generate a scene of 512 x 512 pixels.

The Gucha-ised version breaks up the scene to be generated into smaller sub-regions and distributes them to the volunteers so that the generation can be performed in parallel. Time is measured from the moment the client creates the Gucha environment to the moment the client collects back all the computed results from the volunteers. Speedup is then computed and results are as shown in Figure 6.

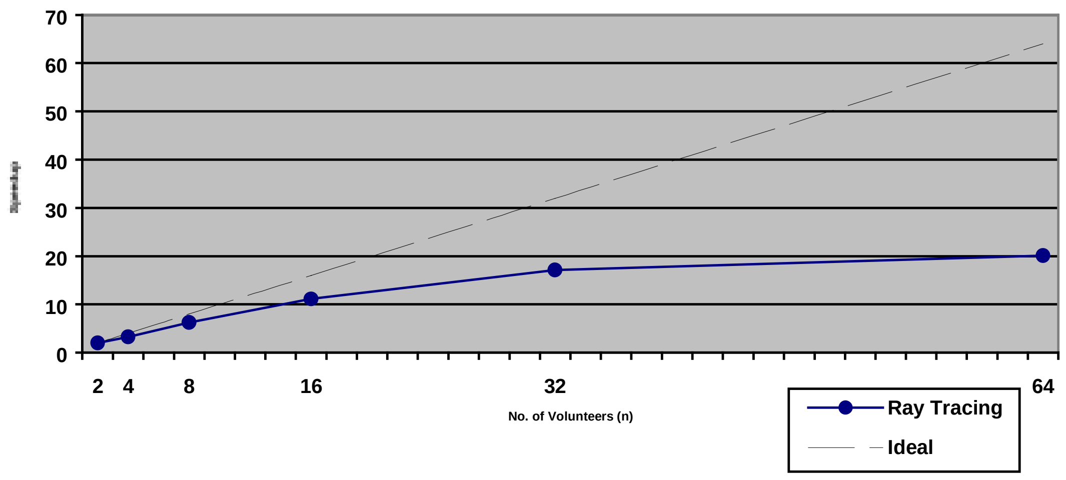

Figure 6. Ray Tracing Experimental Results

5.2.2 Performance

Unlike the previous case, speedup for Gucha-ised Ray Tracing starts to level off with 32 processors. The characteristics associated with this application are:

- Not as computationally intensive as RC5 cracking, basing on the time required for executing the program.
- Low bandwidth requirement. Data exchanges among entities are not too high.
- Computations end only when the client receives results from all volunteers. Hence, overall performance is dependent on the poorest performing volunteer in the system that returns the result.

5.3 Matrix Multiplication

The multiplication of two matrices is one of the most basic operations of linear algebra and scientific computing. The algorithm for performing matrix multiplication is fundamentally very simple and involves a series of multiplication and additions.

5.3.1 Experimental Results

The sequential version is run on a standalone machine to perform matrix multiplication on two square matrices of size 128 x 128.

In our experiments for performing parallel matrix multiplication on the Gucha system, efficiency is not the main objective. Instead, we hope to investigate the impact of transferring large amount of data between entities (in particular, client and volunteers) on the Gucha performance. In the multiplication of matrices A and B to give the result of matrix C, matrix A is broken up into groups of equal-sized rows, based on the number of volunteers that are used. During computations, each sub-matrix A of certain rows and the entire matrix B are transferred to a volunteer to generate the results for the same rows in matrix C. All computed results are sent back to the client to form matrix C. Time is measured from the moment the client creates the Gucha environment to the moment the client collects back all the computed results from the volunteers. Speedup is then computed.

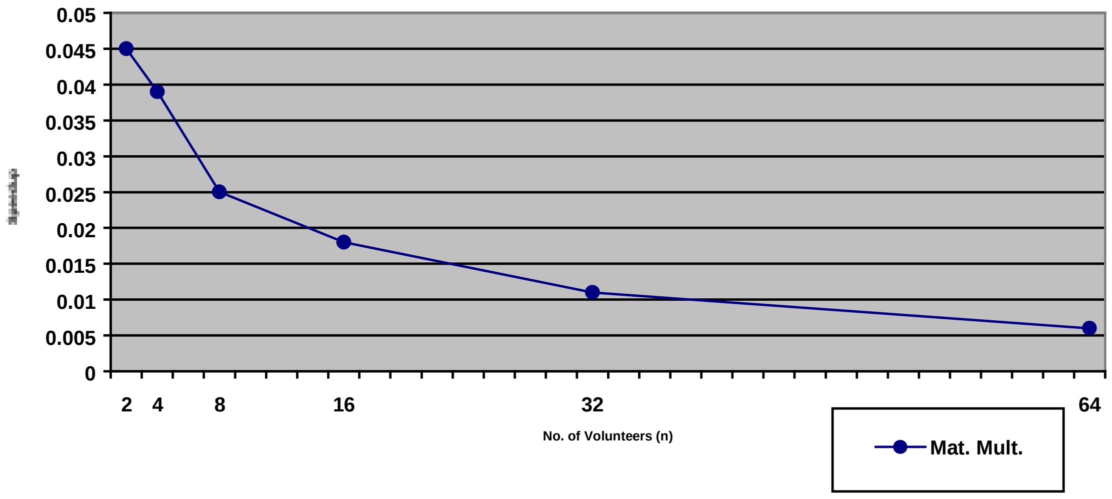

Figure 7. Matrix Multiplication Experimental Results

5.3.2 Performance

From Figure 7, the performance for Gucha-ised matrix multiplication is very poor. Not only there is no performance improvement, the speedup value decreases with the use of more volunteers. Characteristics of this application are:

- Low computation requirement. The size of the matrices used in experiments is not very large.
- High bandwidth requirement. The amount of data for a matrix of size 128 x 128 to be transferred from the client to a volunteer is already 1 Mbits. Hence, as more volunteers are

involved in the computation, the network throughput at the client's end becomes a performance bottleneck.

- Long encryption and decryption time. In the Gucha system, the communication exchanges among the entities are encrypted. As the amount of data involved in the exchanges is huge, time taken for encryption and decryption of the data would be significantly high.
- As in the previous experiment, computations end only when the client receives results from all volunteers.

Experiments are conducted to measure the time for transferring data from client to a volunteer and for the volunteer to perform the necessary decryption, during task allocation. From our measurements, these overheads take up about 25% of the overall processing time.

5.4 Remarks

From the above experiments, the speedup of a Gucha-ised application depends largely on the computation to communication ratio involved. For good speedup results, computational performance gained for executing an application in parallel with a group of volunteers must be able to offset the loss incurred by communications overhead. In general, Gucha is well suited for executing coarse-grained applications (such as RC5 cracking) with high computation to communication ratio.

Applications that complete based on the receipt of the first result from a volunteer would also be better candidates for executing in Gucha's computing environment. In such applications, the waiting for all volunteers to send back results is not necessary and hence, overall performance is independent of the poorest performing volunteer in the system. For Internet-based computing environment, due to the unpredictable nature of the network and volunteers, this factor is especially important. Applications, such as the searching of secret keys in RC5 cracking, are ideal candidates for execution using Gucha.

6 Conclusions

Despite the dynamic and unpredictable nature of Internet as a computational resource, the potential of utilizing it for metacomputing is still great. However, existing Java-based implementations for parallel computing are not designed with Internet-based computing in mind. As such, their deployment in the Internet environment may not be feasible.

Gucha aims to address deficiencies of the existing implementations for Internet-based parallel computations. Its implementation goals are *ease of use*, *heterogeneity*, *portability*, *security*, *fault tolerance*, *load balancing* and *scalability*. To our knowledge, it is the only Internet-based computing system that supports dynamic code execution without the use of applets, network load scheduling and encryption of network communications. Preliminary testing based on the execution of parallel applications developed for the Gucha system gives promising results. However, for good speedup results, Gucha should be used in executing coarse-grained applications with high computation to communication ratio.

References

[1] A. Baratloo, M. Karaul, Z. Kedem, P. Wyckoff, "Charlotte: Metacomputing on the Web", 9th International Conference on Parallel and Distributed Computing Systems (PDCS), 1996. http://www.cs.ucsb.edu/research/superweb/

[2] B. O. Christiansen, P. Cappello, M. F. Ionescu, M. O. Neary, K. E. Schauser, D. Wu, "Javelin: Internet-Based Parallel Computing using Java", http://www.cs.ucsb.edu/research/superweb/home.html

[3] Luis F. G. Sarmenta, S. Hirano, S. A. Ward, "Towards Bayanihan: Building an Extensible Framework for Volunteer Computing using Java, http://www.cag.lcs.mit.edu/bayanihan

[4] P. A. Gray, V. S. Sunderam, "The IceT Framework for Metacomputing", http://www.mathcs.emory.edu/~gray/abstract7.html

[5] A. J. Ferrari, "JPVM: Network Parallel Computing in Java", Technical Report CS-97-29, Dept. of Comp. Sci, Unverisity of Virginia, Charlottesville, VA 22903, USA, http://www.cs.virginia.edu/~ajf2j/jpvm.html

[6] David A. Thurman, "jPVM: The Java to PVM Interface, December 1996, http://www.isye.gatech.edu/chmsr/jPVM/

[7] N. Yalamanchilli, W. Cohen, "Communication Performance of Java based Parallel Virtual Machines", ACM 1998 Workshop on Java for High Performance Network Computing, http://www.cs.ucsb.edu/conferences/java98/program.html

[8] Luis F. G. Sarmenta, "An Adaptive, Fault-tolerant Implementation of BSP for Java-based Volunteer Computing Systems, http://www.cag.lcs.mit.edu/bayanihan

[9] MPIJ 1.1, http://ccc.cs.byu.edu/OnlineDocs/docs/mpij/MPIJ.html

[10] M. Armstrong, Y. Ma, "Java Ray Tracer", http://robotics.eecs.berkeley.edu/~mayi/CS184/

[11] Sun Microsystems, “The 100% Pure Java Initiative White Paper”, http://java.sun.com/100percent/wp.html

[12] R.L. Rivest, “The RC5 encryption algorithm”, CryptoBytes, 1(1): 9-11, 1995

[13] G. Hewgill, “RC5 and Java Toys”, http://www.hewgill.com/rc5/index.html