Overall Introduction to JiniSolve - a Jini Based Grid Computing Framework

Qixin Wang, Xinghua An, Xiaoge Wang, Kehong Wang
Dept. of Computer Sci. & Tech., Tsinghua University
Postal Addr.: P.O.Box 84-93, Tsinghua Univ., Beijing, 100084, P.R.China Tel: +86-(0)10-62789831
Email: wchsh@yahoo.com, {anxh, wang}@tirc.cs.tsinghua.edu.cn
November, 2000

Abstract

People are now interested in sharing the computational capabilities of previously exclusive high performance computer systems across the network. However, the difficulty in cross platform computing and organization over the Internet has hampered such attempts for a long time. Enlightened by Java byte codes' ideal feature of platform independence and Jini infrastructure's high adaptability, we hereby propose JiniSolve -- a new grid computing framework. JiniSolve can support all of the three distributed computing mechanisms -- remote computing, mobile agent computing and code shipping, among which, the support for mobile agent computing is a unique advantage of JiniSolve in comparison to other similar grid computing frameworks. Moreover, JiniSolve is capable of organizing disseminated computational resources over the network asynchronously and autonomously during runtime, which enhances the framework's availability, reliability and ease of use.

Keywords: *Internet Computing, Grid Computing, Mobile Agent, Jini, Java*

1. Introduction

Because of the increasing number of high performance computer systems of various architectures available and the rapid development of internet/intranet infrastructures, people solving computing intensive problems are no longer satisfied with the old computing paradigm of being confined to one local computer system, with solid hardware configurations, preinstalled software packages, and local data. Both the manufacturers and the users of high performance computer systems are interested in the idea of *Grid Computing* [6], i.e., utilizing any desirable computational resources (clusters, mainframes, any architecturally homogeneous or heterogeneous systems) across the network (especially the Internet) to solve problems in a transparent, distributed manner and to make high performance computational resources more efficiently used.

There are fundamentally three types of distributed computing mechanisms [5]: *mobile agent computing* (client code and data runtime migration), *remote computing*, and *code shipping* (server code runtime migration). In mobile agent computing, both the code and data are originally on client's machine. They migrate to the computing machine (server) during run-time to be executed there and finally return the results to the client. In remote computing, data are transferred during runtime from client to server to be processed there. (a typical example of remote computing framework is CORBA). In code shipping, the code is downloaded from server to the client in runtime to process the data there and produce results.

However, many grid computing frameworks, such as NetSolve[5], Ninf[4], RCS[3] etc., can only support remote computing. They have not broken through the barrier of platform differences and hence can not effectively support code migration, which is imperative in implementing mobile agent computing and code shipping computing mechanisms. On the other hand, some existing quasi-grid computing frameworks based on Java and Web technologies, such as Charlotte[2], need too much manual interference in organizing the distributed resources to be regarded as a transparent framework, although they can support code migration to some extend.

In order to solve these deficiencies, we hereby propose the JiniSolve framework, which takes the advantage of Java and Jini to achieve the following goals:

1) Support all of the three distributed computing mechanisms, especially mobile agent computing;

2) Transparent (automatic) and dynamic (runtime) organization of all grid computing resources.

3) Higher efficiency in the utilization of high performance computing systems; better adaptability, reliability, availability and security.

Java[12,13,14] and Jini[15] play key roles in the framework of JiniSolve. Java byte code's well-known feature of platform independence enables JiniSolve to *support runtime code migration and data exchange over heterogeneous platforms* across the network easily, hence breaks through a previously insurmountable barrier in grid computing. Jini technology was firstly proposed for pervasive computing, i.e., to integrate all kinds of electronic devices into Internet. However, in JiniSolve, Jini is playing the role of automatically organizing grid computing resources during runtime, guaranteeing the framework's asynchronism, adaptability, reliability, availability and ease of use.

In the following, we are going to introduce the architecture of JiniSolve, its programming patterns, its advantages and give some primary experimental experiences.

2. Architecture of the Framework

JiniSolve framework consists of six types of entities: *JiniSolve service*, *JiniSolve client*, *lookup service*, *service proxy*, *client compugent* and *service locale*.

JiniSolve clients are those machines who join the JiniSolve community to find computational resources to accomplish certain computational tasks.

The *JiniSolve services* refer to the computational resources available in the JiniSolve community. They can be architecturally heterogeneous high performance mainframes, workstation clusters etc., which join the JiniSolve community to contribute their computational capabilities.

The *lookup service* is a broker service in the JiniSolve community and plays a vital role in running the JiniSolve framework. Lookup services register and maintain the lists of available JiniSolve services in the subnet and their attributes. They tell JiniSolve clients about which computational resources are proper to use. They also help JiniSolve clients to connect with the specific JiniSolve services.

A *JiniSolve service proxy* is a Java software object (with member code and data) instantiated by a specific JiniSolve service to be delivered to its JiniSolve clients in runtime to act as a representative for it. Having got the proxy object, the JiniSolve client can locally call the member methods of the proxy to conduct local computation (i.e., code shipping computing), activate remote computations on the JiniSolve service (i.e., remote computing), or launch client s computational mobile agent -- *compugent* -- to the remote JiniSolve service host to be executed there(i.e., mobile agent computing).

The *client compugent* and *service locale* are important for conducting mobile agent computing in the JiniSolve framework, which are to be introduced in detail in section 2.4.

Each physical host can act as one or several of these entities according to specific contexts. E.g., a mainframe acting as a JiniSolve service can be a JiniSolve client at the same time, using other services in the framework to solve its sub-problems.

2.1. Physical Architecture & Primary Deployment Scenarios

The primary physical architecture of JiniSolve is illustrated in Figure 1. The basic deployment scenarios are as follows.

The central element of running a JiniSolve community is its lookup services, which comply with the *Jini Lookup Service Specifications* [15]. There may be several lookup services running in one JiniSolve community simultaneously in order to provide increased system reliability, availability(e.g., to resist partial failure) and a broader global view of the whole community.

When a JiniSolve service starts up, it joins a specific JiniSolve community and tries to find all nearby lookup services using *Jini Discovery and Join Protocols* [15]. To each lookup service it found, it will make a registration (as shown by arrow 1,2,3 in Figure 1). The registration can include information on the location, proxy type, additional capabilities (other than those defined by its proxy interface), status info of the JiniSolve service etc.

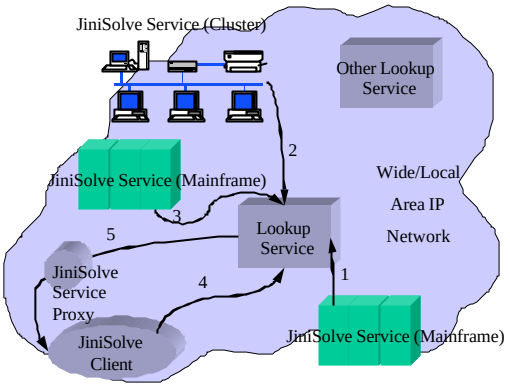

Figure 1. Physical Architecture of JiniSolve

When a client joins a JiniSolve community, it tries to find one or more *lookup services* in network vicinity using *Jini Discovery Protocols* and tries to lookup proper registered JiniSolve services to accomplish its computation (shown by arrow 4 in Figure 1). When a desired *JiniSolve service* is found, the client downloads its proxy (shown by arrow 5 in Figure 1) and dynamically load the proxy into the client's Java runtime so as to utilize the services provided by the *JiniSolve service*. All of these are done according to the *Jini Lookup Protocol* [15].

Because of the deployment of Multi-cast UDP/IP based Jini Discovery/Announcement Protocols, in local area networks, all of the resource organizations are done automatically, totally transparent to human users. However, in wide area network, lookup services will use Unicast TCP/IP connections to communicate with each other. Hence, a few human configurations are needed. To sum up, the organization of distributed resources in JiniSolve is highly automatic and transparent. This is one of the advantages of JiniSolve over similar frameworks such as Charlotte [2].

2.2. API Conventions between Unacquainted Clients and Services

Though the local programs and scientific computing libraries of clients and services can be platform dependent native codes (see section 2.3), the interconnecting modules of JiniSolve entities should be programmed in Java to meet the requirements of platform independence in grid computing.

There are a few types of well-known Java *interfaces* imperative to both client and service side programming.

They act as conventions among entities who want to join the JiniSolve community, either as a client or service, so that any pair of previously unacquainted client and service can communicate with each other during runtime.

The first type of such Java interface is that of the *service proxy's* (a service proxy object itself is not an interface, but it should comply with the definition of some certain types of proxy interfaces and implement all the abstract member methods of them). Before running, a JiniSolve client is unable to know which proxy object of a specific JiniSolve service is going to be got after the lookup. The client side developer will have to define the client' s interaction with future service proxies according to service proxy interface conventions. So defining such universal well-known proxy interfaces is an imperative task in building the JiniSolve framework.

There may be a few well-known service proxy interfaces existing, though not many. Since our JiniSolve framework is aimed at solving scientific computational problems, we choose to define our first well-known proxy interface according to the LAPACK[1] Application Programming Interface(API)s, a well known software package for solving linear algebra problems.

The reasons why we choose LAPACK as the first universal interface in JiniSolve are as follows. First, most scientific computational tasks can boil down to linear algebra problems; second, LAPACK is famous -- many programmers in scientific computing are familiar with its software routines.

The following is an example of our proxy interface:

```
public interface JiniSolveLAPACKServiceProxyInterface
extends Serializable { ……
    /*solve a real system of linear equations A * X = B*/
    public void dgesv (int n, /*number of equations*/
      int nrhs,  /*number of columns of matrix B*/
      double [] a, /*matrix A*/
      int lda, /*leading dimension of matrix A*/
      int [] ipiv, /*pivot indices*/
      double [] b, /*input - matrix B, output - result X*/
      int ldb, /*leading dimension of array B*/
      intW info /*additional result info*/ ); ……
}
```

Thus, a JiniSolve client application developer who want to utilize a JiniSolve service to solve linear algebra problems can program upon the Java abstract member methods (functions) declared in the *JiniSolveLAPACKServiceProxyInterface*. Any downloaded service proxy instance during runtime through the JiniSolve framework will understand these invocations. On the other hand, any JiniSolve service that provides such proxy should implement all the abstract member methods (functions) declared in the interface, either directly in Java, or in the service machine's native code through Java Native Interface (JNI) [13].

In addition to LAPACK, Many other universal proxy interfaces are under consideration and definition.

Besides the service proxy interfaces, there are two other types of Java interfaces that should be well-known to the JiniSolve community (if the community is to support *mobile agent computing* mechanism) -- the Java programming interfaces for *compugents* and *service locales*, which is to be discussed in detail in section 2.4.

2.3. Programmatic Architectures

As mentioned in section 1, JiniSolve framework supports three types of distributed computing mechanisms -- *remote computing*, *code shipping* and *mobile agent computing*. Figure 2 illustrates the programmatic architectures of *remote computing*, which is probably the most commonly used mechanism in practice.

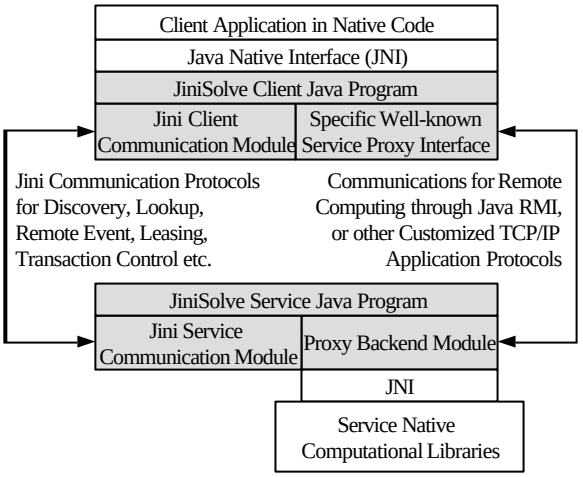

Figure 2. The JiniSolve Programmatic Architectures for Remote Computing

In the figure, the software modules are distinctively divided into two sets - that of client side and of service side. The gray rectangles represent software modules that are encoded as Java byte codes and are interpreted by Java Virtual Machines(JVM); while the white blocks, which are optional in specific implementations, represent software modules encoded in a local host's native binary codes.

The top layer of a JiniSolve client application may be encoded in native binary code. If so, JNI mechanisms [13] are deployed to serve as an interface for the native code to call the underlying *JiniSolve client Java program*.

The client application can also be directly written in Java, discarding the previous two layers. The *JiniSolve client Java program*'s main module is built on two important underlying sub-modules - the *Jini client communication module* and the well-known *JiniSolve service proxy interface*, which will be substituted by a downloaded *service proxy* implementation during runtime.

The *Jini client communication module* provides the JiniSolve client with abilities to participate in the JiniSolve community, which is in fact a Jini community. The process of finding *lookup services*, *JiniSolve services* and fetching *service proxies* are conducted in this module. The development APIs for this module are mainly Jini and Java Development Kit APIs.

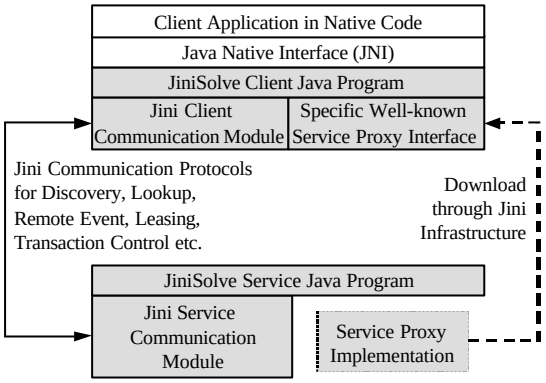

Figure 3. The JiniSolve Programmatic Architectures for Code Shipping

The well-known *JiniSolve service proxy interface* is introduced in section 2.2. It is just a virtual programming interface during compile time, but will be substituted by a *service proxy* during runtime, which provides the actual implementation. The service side developers, rather than the client side developers, are responsible for the various specific implementations of the proxy interface. The implementation usually utilizes Java Remote Method Invocation (RMI) mechanisms [14] to activate the remote Java methods on the service, transmit parameter data and retrieve return values.

For the *JiniSolve service*, the main program should be a Java program. The two underlying software modules are the *Jini service communication module* and the *proxy backend module* (see Figure 2).

The *Jini service communication module* enables the *JiniSolve service* to participate in the JiniSolve community as a Jini service, so that it can find *lookup service*, register, launch proxies to interested client etc.

The *proxy backend module* responds to network requests from proxies of the *JiniSolve service* disseminated across the JiniSolve community. The service proxies that reside on JiniSolve clients use Java RMI or other customized TCP/IP application protocols to invoke methods in this module remotely. The methods in this module will accept the data submitted by proxies, process them and return the results back. This is exactly the mechanism of *remote computing*.

In most cases, software routines in the service host's local native computational libraries are called (through JNI) by the proxy backend module to do the real data processing (see Figure 2). The *service native computational libraries* may usually be well-known scientific computation packages such as LAPACK, ScaLAPACK, etc.

Figure 3 shows the programmatic architecture for *code shipping*, which is much simpler. However, this mechanism may be much less frequently used in real practice, since it can't utilize the computational capability of the server, which is usually a high performance computer. The architecture is similar to that of *remote computing* except that the proxy backend module and its underlying native modules are all discarded. The *service proxy* is exactly the code shipped to client and is invoked locally to the client's host without any future interactions with the service host.

2.4. Mobile Agent Computing Support

One of the unique advantages of JiniSolve framework is its support for the *mobile agent computing* mechanism, whose programmatic architecture is illustrated in Figure 4.

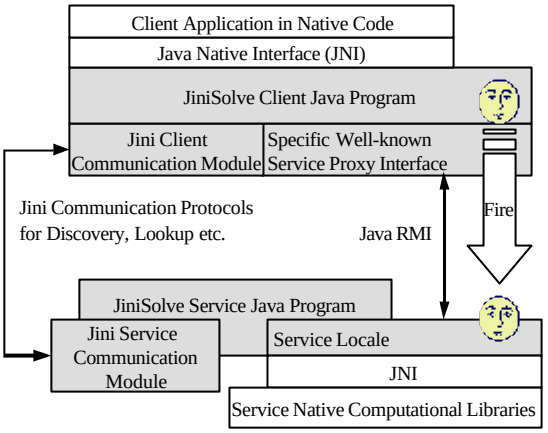

Figure 4. The JiniSolve Programmatic Architecture for Mobile Agent Computing

Under this mechanism, the client first needs to find a service that supports mobile agent computing. After the client has downloaded the proxy of such a service, the client can create a mobile computing agent (the smile face, see Figure 4), which is actually a Java object with program logic and raw data to be processed. We name such mobile agent as *compugent* (or *client compugent*) in our JiniSolve community. Next, certain methods of the service proxy are invoked to "fire" (launch) the compugent onto the remote service host. The fire action is supported by Java RMI mechanisms. The *landing place* on the service host that accepts client compugent is a special software module called *service locale*, which provides software routines for compugent to utilize the local software or hardware resources on the service host.

Compared with that of *remote computing*, the raw data is now encapsulated in *compugent* together with the computational control logic. The *service proxy* does not take the trouble to invoke remote computational methods on the remote service, but rather, all the computational control logic is carried by the compugent onto the service host and carried out there locally.

In order to implement such a computing mechanism, there must be a few conventions between the client and service in programming, i.e., there must be a few well-known interfaces predefined so that client and service programs can be developed independently.

In our prototype JiniSolve framework, we defined three interfaces -- *CompugentInterface*, *LAPACKCompugentWorkstationInterface* and *CompugentWorkstationProxyInterface*, which define the ubiquitous functionality

of *compugent*, LAPACK compatible *service locale* and *service proxies* capable of firing *compugents* respectively. Thus, client and service end developers can write their programs according to these interfaces independently to implement mobile agent computing.

The *CompugentInterface* and *CompugentWorkstation ProxyInterface* are simple and universal. The *LAPACK CompugentWorkstationInterface* is destined to provide functionality defined in LAPACK. Other kinds of *service locale* interfaces may be added in the future.

2.5. JiniSolve Development Kit -- Some Programs

The users of JiniSolve include end users and *client/service developers*. The developers of JiniSolve can be divided into two categories -- the JiniSolve client developers and the JiniSolve service developers. The client developers want to utilize JiniSolve to solve computational problems; while service developers want to publish (or sell) their local system's computational capabilities.

As a middleware provider, we provide certain facility software components, conventions and script definitions to encapsulate the underlying JiniSolve infrastructures, so that client/service developers can concentrate on their own higher level application logic. We name this facility software as *JiniSolve Development Kit* (*JiniSolveDK*, which also includes APIs in JDK and Jini Development Kit).

Following is an example client Java program using *JiniSolveDK*.

```
……
JiniSolveFFTServiceProxyInterface[] proxy;
ServiceTemplate tmpl = new ServiceTemplate(null, {JiniSolveFFT
                        ServiceProxyInterface.class}, null);
ProxyFinder proxyFinder = new ProxyFinder();/* assistant object for
finding proper service proxies from JiniSolve for the client*/
try{
   proxy = proxyFinder.getProxies(tmpl, /*proxy matching template,
                        specifying which type of proxies to find*/
            1,          /*desired quantity of proxies*/
            5000);      /*time out milliseconds*/...
   proxy[0].FFT_2D(matrix, 64); /*invoke FFT on remote service*/
}catch(RemoteException e)……
```

In the above program, *ProxyFinder* is a facility class from *JiniSolveDK* for finding proper service proxies form JiniSolve community for the client. We provide this class to encapsulate all the trivial details of dealing with underlying Jini protocols in finding proper services and their proxies, so that the JiniSolve client end developers can concentrate on their own application logic.

In the above program, the remote FFT transformation are conducted by the proxy in a *blocking* fashion. We also provide APIs for conducting *nonblocking* fashion of Remote Method Invocation, so as to introduce parallelism into client's program.

We provide two useful auxiliary classes -- *ParallelTask* and *Synchronizer* to represent parallel task and parallel execution synchronizer respectively.

In the following program, *p1.parExecuteCompugent (c1, t1)* and *p2.parExecuteCompugent(c2, t2)* both return in a *nonblocking* fashion, leaving actual remote execution on services going on in parallel. The program blocks at the statement of "*synch.waitForSatisfaction()*" until all the parallel tasks (*t1* and *t2*) are finished. Finally, the results of the parallel tasks are retrieved using *t1.getResult()* and *t2.getResult()*.

```
  ……
/* p1, p2 are proxies of two services that can accept compugent*/
  Synchronizer synch = new Synchronizer();
  ……
  ParallelTask t1 = new ParallelTask();
  t1.addSynchronizer(synch);
  t1.setInput(data1);
  XXCompugent c1 = new XXCompugent( .);
  p1.parExecuteCompugent(c1, t1);
  ……
  ParallelTask t2 = new ParallelTask();
  t2.addSynchronizer(synch);
  t2.setInput(data2);
  XXCompugent c2 = new XXCompugent( .);
  p2.parExecuteCompugent(c2, t2);
  ……
  synch.waitForSatisfaction();
  ……
/*access the result of task t1, t2*/
  ……t1.getResult(); ……t2.getResult(); ……
```

From the above program, we can also understand how proxies launch client's mobile agents onto services - e.g., *p1.parExecuteCompugent(c1, t1)* will invoke proxy *p1*'s underlying RMI mechanism to launch compugent *c1* to *p1*'s remote server *in a non-blocking fashion*. A *blocking* version of launching a client compugent is like this:

```
<Result> = proxy.executeCompugent( anyClientCompugentObject );
```

The service end program may be a little bit more complicated. In order to setup a JiniSolve service, the service developer should implement the following programs:

1. Service Proxy: implement specific well-known JiniSolve proxy interfaces to make the local service available to the outside JiniSolve community.
2. Optional Proxy Backend (including Service Locale): to deploy the local system's resources to support remote proxies disseminated across the network.
3. Optional Monitor Daemon: a daemon application listening and analyzing information gathered by all the distributed proxies and the local system, according to service developer's specific strategy, so as to monitor the local system, adjust the status info in lookup services etc.
4. Launch Script: a configuration file telling *ProxyLauncher*(see below) how to launch the service proxy and setup backend (include proxy backend, service local and monitor daemon).

In *JiniSolveDK*, we provide a facility class -- *ProxyLauncher* to encapsulate the JiniSolve infrastructures for launching service proxies onto lookup services. All service developer need to do is to indicate in the *Launch Script* what to launch, what to setup in the service and a few easy to understood instructions on how to launch.

2.6. Performance Concerns

Since the birth of Java, it is continuously being criticized for its poor executing speed, and hence being repelled from the field of scientific computation. However, there are several reasons explaining why Java based JiniSolve framework can enhance the overall performance of client tasks.

Firstly, under most circumstances in JiniSolve, Java merely acts as an intermediate "glue" in combining miscellaneous elements of heterogeneous platforms across the network to constitute the JiniSolve grid computing community, rather than take charge of performing actual scientific computing. Using Java's JNI mechanisms, the real computations are carried out by native codes on JiniSolve service hosts, so the performance is not drastically affected by Java. On the contrary, Java enables users to access more kinds of computational resources that are unavailable before due to platform differences.

Under most conditions, the organizational time cost of Java (JVM and Java object initialization, network communication, method invocation etc.) is negligible when compared to the time cost of real scientific computation conducted in native codes. So whether or not the organizational software module is written in Java or native code is to make not much difference.

Our example experiment in remote-sensing data processing (see section 4) also supports the above conclusion (see Figure 5). In Figure 5, *Trc* represents the total time cost in the remote computing process, while *Tnative* represents the part of time cost spent on native code execution. From the figure, we can easily understand that *Tnative* comprises most part of *Trc*.

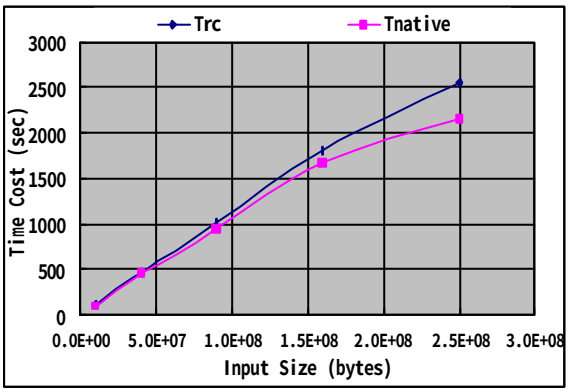

Figure 5 Native Code Computing Time Cost in Remote Computing Mechanism, an Example

Secondly, JiniSolve environment enables its users to deploy a medium grained grid computing pattern. The granary of parallelism in JiniSolve can reach the level of function (i.e., functions of a computing task can be executed in parallelism in the JiniSolve framework). E.g., in executing one single client process, a JiniSolve client can dispatch one function call on one remote super computer and dispatch the other function call to another remote super computer. Thus, it is possible for JiniSolve clients to optimally utilize the specific advantages of different high performance systems in parallel. E.g., a client program may send its function call number 1 to a vector computer where it is the most desirable and function call number 2 to a mesh computer. This will enhance the performance of JiniSolve. Moreover, this also provides a mechanism to utilize the advantages of different system architectures more efficiently. People may no longer need to quarrel over the problem of which high performance computer architecture is better and which is worse.

Thirdly, the support for *mobile agent computing* will also enhance performances under certain circumstances. One of the advantages of mobile agent computing is that by sending computational control logic to the data source, network transmission of intermediate data will be dramatically reduced [11]. In section 4, we will present such an example in remote sensing data processing.

2.7. Reliability, Availability and Security Issues

Based on Jini infrastructure, JiniSolve community can achieve high reliability and availability. The deployment of *Jini Leasing Protocol and Jini Transaction Protocol* protects the framework from the harm of partial network failures. For example, *Jini Leasing Protocol* guarantees that the crashed service's registrations will be cleaned in time so that the lookup services are protected from being bogged down by accumulated "stale" registrations. Meanwhile, Jini infrastructure can easily support service redundancy, specific third-party checkpoint storage service etc., which makes the services provided by JiniSolve highly available.

The security infrastructure of JiniSolve is based on Java2 security mechanisms[12], under which, any Java application (client application, service application or lookup service etc.) can setup a security manager in runtime to carry out preset security policies to specific Java byte codes being executed in the Java runtime environment. For example, a Java application running on host A can deliberately allow or forbid byte codes downloaded from host B to visit host A's local files, set up TCP/IP connections to other hosts etc.

3. Comparison with Similar Frameworks

Table 1 is a comparison of JiniSolve and two other major grid computing frameworks -- NetSolve[5,7] and Legion[8,9,10]. Specific advantages of JiniSolve are indicated in the table with **Bold** characters.

		JiniSolve	NetSolve	Legion
Distributed Computation over Wide Area Network		Supported	Supported	Supported
Distributed Computation over Heterogeneous Platforms		Supported	Supported	Supported
Granary of Code Being Shared		**Function**	Function	Object
Code Shipping Computing Mechanism (Runtime Service Code Migration)		**Supported**	Not Supported	Not Supported
Mobile Agent Computing Mechanism (Runtime Client Code Migration)		**Supported**	Not Supported	Not Supported
Clonable Mobile Agent		**Supported**	Not Supported	Not Supported
Organizational Automaticity of Distributed Entities (Transparency)	Service – Broker	**Fully Automatic**	Manual Assis. Needed	No Brokers
	Client -- Broker		Manual Assis. Needed	No Brokers
	Client – Service		Fully Automatic	Manual
	Broker – Broker	**Fully Automatic (LAN) Manual Config. Needed (WAN)**	Manual Assis. Needed	No Brokers
Organizational Reliability of Distributed Entities	Service – Broker	**Protected by protocols mainly based on time out mechanisms (little network communication)**	Protected by protocols mainly based on periodical polling (heavy network communication)	No Brokers
	Client – Broker			No Brokers
	Client – Service			Protected by certain protocols
	Broker – Broker			No Brokers
Need for Unifying Data Exchange Format across Heterogeneous Platforms		**Unnecessary (using Java data format across the network)**	Needed	Needed
Service Sudden Crash Handling & Rehabilitating		Notify the user by throwing a Java remote exception	Return an error code	?
Parallel Coding Experience Requirements for Users (Developers)		**Not Needed**	Not Needed	Needed
Security Mechanisms (Identification, Encryption, Access Control etc.)		*Digital authentication and data encryption are supported. After identification, client code can access any server resource according to policy configuration preset by the server administrator.*	?	Authentication, access control are supported

? lack of relevant information; *Italic* under implementation or design;

Table 1. Comparison of JiniSolve with NetSolve and Legion

4. Example Applications

In the field of remote-sensing image processing, a geographic place is described by several images of different wave bands, which contain huge information.

Our task is to apply slant transform to the images of each band of a specific geographic place, and evaluate the importance of each band according to our specific strategy (in the example, our strategy is based on entropy). Then the band with low importance is filtered and only the first three bands with the highest importance is reserved to reconstitute a filtered image.

In the past, we can only conduct remote computing, in which, control logic (evaluation strategy) is not transmitted to the server. So the server doesn't know what to do next after having applied slant transform to images of each band. Though merely three of the processed images are useful, it will return all the images back to the client to let it carry out its specific evaluation strategy. Under such mechanism, the network bandwidth is wasted, and unnecessary transmission delay rises. What is more, the client needs to use its own computational resources to carry out its evaluation strategy.

However, a mobile agent computing mechanism is much more desirable for our task:

1. Client finds a remote server capable of slant transform
2. Fire client's compugent to server, which carries our specific evaluation strategy and raw data.
3. Compugent calls slant transform function on the server through the server's service locale.
4. Server returns transformed images of each band to the *local* compugent (*rather to the network*).
5. Compugent evaluates each band according to our specific strategy *locally on the server host*.
6. Compugent selects the images of the best three bands and return them to client through network.

Under such mechanism, the compugent object, which is implemented by client application developer, migrates to the source of intermediate data -- the server, and interacts with it for the client. Compared to the remote computing mechanism, there are two advantages

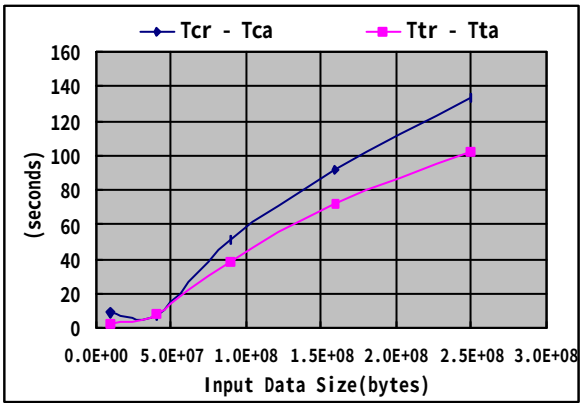

Figure 6. Comparison of Remote Computing & Mobile Agent Computing the Remote-sensing Data Processing

-- firstly, the network transmission of unwanted intermediate data is avoided, which will greatly reduce the latency caused by redundant transmission; secondly, carrying out evaluation strategy on the high performance server computer may be more efficient than doing it on the client platform.

We carried out our design on JiniSolve framework and got the results shown in Figure 6.

In Figure 6, The curve with rhombus dots represents the *deference* between *Tcr* and *Tca*, which are the total time costs of remote computing and mobile agent computing respectively. The curve with square dots indicates the *difference* between *Ttr* and *Tta*, which are the total time costs of data transmission in remote computing and mobile agent computing mechanism respectively.

From Figure 6 we can understand that as the scale of input data is increasing, mobile agent computing becomes more and more superior to remote computing because of the reduction in intermediate data transmission (e.g., when input data size reaches 250MB, the network transmission delay of remote computing is about 2 minutes longer than that of mobile agent computing).

5. Summary and Future Work

Compared to other existing grid computing frameworks, JiniSolve is architecturally innovative. One of the most distinctive advantages of JiniSolve is its support for all 3 types of distributed computation mechanisms, especially the mobile agent computing. Another advantage is its distributed, asynchronous, dynamic organization capability, in which client programs find desirable computing machines on the network automatically in runtime without human interference. Other advantages of Java and Jini are also integrated in the JiniSolve framework. The programmatic architecture of JiniSolve is clear and functional, which provides an ideal infrastructure to build higher-level software applications.

The future work is to be concentrated in the following aspects:

1) Lookup strategy optimization. Various factors, such as service architectural characteristics, workload, network topology, bandwidth etc. are going to be considered by lookup service, so as to dispatch workload more properly.

2) Implement more comprehensive and easy to use developer tools and user interface facilities, such as constructing command line user interface, Java/C/C++/Fortran development libraries, possible script utilities in building user interfaces etc.

3) Better security facilities. On top of the security infrastructures provided by Java2, we want to implement more sophisticated high level and easy to use security facilities, such as account administration, logging etc.

4) Defining more comprehensive sets of well-known service proxy interfaces and service locale interfaces.

5) Further research in fault tolerance issues, such as distributed back up, check point etc.

6) Further research in JiniSolve's distributed transaction issues. Specific problems may require specific distributed transaction arrangements, which may be of great academic or technological research values.

7) Support e-Business, so that service providers can *sell* their computational capability in the JiniSolve grid just like electric power stations sell their electricity in the real world.

6. Acknowledgement

The work was supported by funding from Tsinghua University for the fundamental research and by funding from the Ministry of Education of China for ACI project.

References

[1] E. Anderson, Z. Bai, et al, *LAPACK User's Guide (Second Edition)*, SIAM, Philadelphia, PA, 1995

[2] A. Baratloo et al, Charlotte: Grid computing on the Web, *Proc. of the 9th Parallel and Distributed Computing Systems*, 1996

[3] W. Gander et al, The Remote Computational System, *Lecture Note in Computer Science, High-Performance Computation and Network*, 1067:662-667, 1996

[4] Mitsuhisa Sato, Hidemoto Nakada et al, Ninf: A Network based Information Library for Global World-Wide Computing Infrastructure, *Proc. of Parallel Object-Oriented Methods and Applications (POOMA)*, 1996

[5] Henri Casanova, Jack Dongarra, NetSolve: A Network-Enabled Solver; Examples and Users, *Proc. 7th Heterogeneous Computing Workshop (HCW'98)*, pp. 19-28, 1998

[6] I. Foster et al, The Grid, Blueprint for a New Computing Infrastructure, *Morgan Kaufmann Publishers Inc.*, 1998

[7] Henri Casanova et al, Deploying Fault-tolerance and Task Migration with NetSolve, *to appear in The International Journal on Future Generation Computer Systems.*

[8] M. Humphrey, F. Knabe et al, Accountability and Control of Process Creation in the Legion Metasystem, *Proc. of the 2000 Network and Distributed System Security Symposium (NDSS2000)*, San Diego, Cal, 2000

[9] S. Brian et al, Grid-Based File Access: The Legion I/O Model, *Presented at the 9th IEEE Int'l Symposium on High Performance Distributed Computing*, Pittsburgh, Penn., 2000

[10] S.Andrew et al, The Legion Vision of a Worldwide Virtual Computer, *Communications of the ACM*, v40, No.1, 1997

[11] H. S. Nwana, Software Agents: An Overview, *Knowledge Engineering Review*, 11(3):205-244, 1996

[12] Li Gong, Java Security Architecture (JDK1.2), *http://java.sun.com/products/jdk/1.2/docs/guide/security/security-spec.doc.html*, version 1.0, 1998

[13] Sun Microsystems Inc., Java Native Interface Specification, *http://java.sun.com/products/jdk/1.2/docs/guide/jni/spec/jniTOC.doc.html*, 1997

[14] Sun Microsystems Inc., Java™ Remote Method Invocation Specification, *ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf*, 1998

[15] Sun Microsystems Inc., Jini Specifications (v1.0.1), *http://www.sun.com/jini/specs/*, 1999