A Java-based Programming Environment for Hierarchical Grid: Jojo

Hidemoto Nakada
National Institute of Advanced Industrial
Science and Technology (AIST)
1-1-1 Umezono, Tsukuba, 305-8568, Japan
hide-nakada@aist.go.jp

Satoshi Matsuoka
Tokyo Institute of Technology
2-12-1 Ookayama, Tokyo, 152-8550, Japan
matsu@is.titech.ac.jp

Satoshi Sekiguchi
National Institute of Advanced Industrial
Science and Technology (AIST)
1-1-1 Umezono, Tsukuba, 305-8568, Japan
s.sekiguchi@aist.go.jp

Abstract

Despite recent developments in higher-level middleware for the Grid supporting high level of ease-of-programming, hurdles for widespread adoption of Grids remain high, due to (1) assumption of peer-to-peer connectivity of all Grid nodes, as well as (2) lack of scalable programming and deployment support. We propose a Java-based programming environment for a hierarchically organized Grid named Jojo, that allow seamless utilization of private-addressed clusters. Jojo provides several features, including secure private remote invocation using Globus GRAM and ssh/rsh to privately-addressed nodes in clusters, intuitive message passing API suitable for overlapped execution using multiple threads, and automatic user/system program staging. Using Jojo, users can easily construct and execute parallel distributed applications on the Grid. We show its design and implementation, ppprogramming API, a working example, as well as preliminary performance evaluation results that prove effectiveness of hierarchal execution.

1. Introduction

Recent developments in high speed networking enables collective use of globally distributed computing resources as a huge single problem solving environment, a.k.a. the Grid. Currently, the Globus Toolkit[6] serves as the 'de-facto standard' lower-level infrastructure for the Grid, and on top it, several middleware systems have been proposed to ease the programming effort, including th GridRPC system Ninf-G[11] and the MPICH-G2[8] that implements MPI programming over heterogeneous nodes on the Grid. Such systems have been proven useful to provide ease-of-programming above Globus in a number of real-life application projects and deployments.

One of the shortcomings of such systems is, however, that they assume all the computers in the environment have globally accessible addresses and can communicate directly with each other. This means that they cannot utilize clusters enclosed inside firewalls, with nodes that use private IP addresses. Furthermore, to utilize these systems, users have to install and set up all the necessary middleware as well as user application programs on each and every computer. The resulting hurdles are quite high, and in fact can be considered one of the primary inhibitors towards widespread adoption of the Grid.

To resolve these problems, we propose a Java based programming environment for the Grid, called Jojo, which is suitable for Grids composed of multiple privately-addressed clusters. Jojo provides several features to ease the programmer/users burden, including secure recursive remote invocation via Globus GRAM and ssh/rsh, intuitive message passing API which enables latency hiding in conjunction with the Java's thread feature, and automatic user/system program staging. Using Jojo, users can construct and execute his own parallel distributed application on the Grid with ease. We describes the design and implementation of Jojo, its programming API, configuration file syntax and a working program example. We also show preliminary performance evaluation results, demonstrating that the performance of Jojo is acceptable and hierarchical execution is effective for programs that could exploit such hiearchical topologies of its underlying nodes, such as search problems.

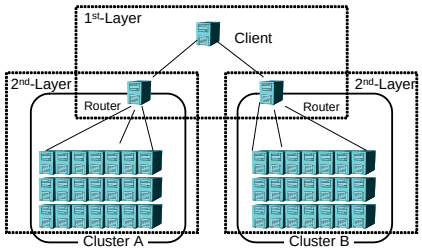

Figure 1. Cluster of clusters.

2. Hierarchical Grid Environment

PC clusters are now serving as the primal computation resources for the Grid environment. In a sense, the underlying execution environment of the Grid can be considered as a large cluster of PC clusters over heterogeneous domains, interconnected by fast networks.

The problem is that, due to security issues and exhaustion of the IP address space, it is becoming increasingly common for individual nodes of clusters to be assigned private IP addresses. As a result, while their individual nodes can initiate connection to outside of the cluster, typically using the NAT (Network Address Translation) feature provided by the router, they cannot accept connections from the outside. The consequence of this is that a node in a cluster cannot directly communicate with a node in other clusters.

As a result, most Grid middleware assuming global node-to-node communication connectivity becomes inoperable. For example, MPICH-G2 will not be able to execute on these kinds of clusters, because it assumes global node-to-node connectivity. Tanaka [10] attempted to resolve this problem with a proxy server called NX-proxy, which works as an application level address translator. While it worked for some Globus-enabled middleware and applications, the appoach was rather ad-hoc, and it was not effective for programs that embed IP addresses in the packets at the application-level. Moreover, maitaining properly-tracked versions of NX-proxy turned out to be quite tedious. As such, there are no versions of NX-proxy that would work over later version of Globus (2.X, and 3.X).

Some middleware such as Ninf-G, have been designed to utilize such clusters. Initially, the Ninf-G client needs to contact the representative cluster node with a Global IP address. Then, Globus GRAM will invoke an executable on a privately addressed node via a backend queueing system. Individual executable can then in turn connect back to the client with appropriate address translation provided by the NAT router.

However, in this configuration, all the connections will eventually be centralized to the client. In order to facilitate a set of clusters with possibly hundreds or even thousands of nodes in total, a single client must single-handedly maintain hundreds to thousands of connections. This solution is obviously not scalable not only due to excessive load on the client resulting from handing hundreds to thousands of data streams, but also due to limitations of file descriptors imposed by the underlying operating systems.

To cope with the problem, we propose to consider the cluster of clusters as a hierarchically composed Grid. The first layer is globally addressed and composed of client nodes as well as ”representative” router nodes of clusters. The second layer is all the privately-addressed cluster nodes as well as the router nodes. We also assume that nodes in a cluster cannot communicate with the outside only indirectly via the router nodes.

Although such a hierarchical structure may seem restrictive, a surprising number of Grid applications will execute and scale well in such an environment; in fact we can take advantage of the underlying configuration to speedup applications: for example, a simple master-worker applications may run faster when the master installed on the router node instead of the remote client, due to the reduced latency between the master and worker. also, for more complicated problems, if we can map the structure of the problem onto the network structure properly, we might attain speed up reducing the communication over low speed networks. [2]

3. The Design of Jojo

Jojo is designed to have the following characteristics:

- Hierarchical architecture that suite the underlying hierarchical structure of typical Grid environments
- Flexible and simple message passing API which takes advantage of the Java thread facility.
- Automatic program staging that ease the burden of installing the system onto servers.

3.1. Architecture

Jojo can be configured in a hierarchical tree form to suite in the hierarchical Grid environment discussed in the previous section. In Jojo, each computer in the system is called a ‘node’. Each node executes its own class on a seperate Java virtual machine and can communicate with its parent node, its child nodes, and its siblings via its parent.

3.2. API Design

There are several existing Java based communication libraries, including RMI[4] from Sun microsystems, JPVM[5] which is a PVM implementation in

Java, and mpiJava[3] which is a MPI implementation in Java. JCluster[1] provides both RMI and MPI interfaces.

RMI provides distributed object model based on synchronous remote message invocation. RMI is easy to use and its model is general and expressive. The down side is that RMI itself does not support parallel execution feature. Users have to explicitly use thread to write parallel programs. MPI/PVM provides a message passing model which uses explicit send and receive. The model is originally meant to work with C and Fortran, which do not have language-integrated thread support. If we were to hide latency by taking advantage of multi-thread feature of Java, this model would not be very suitable due to the necessity of multiple sender/receiver matching, as well as the difficulty of buffer management to achieve thread safety.

Jojo employs a messaging model that can be considered a "hybrid" between distributed object and message passing. In Jojo, just one representing active object runs on each node. The active object can send/receive messages to/from other active objects on respective nodes. Users can specify different representative active object class for each node. Jojo provides explicit "send" method, but no "receive". Messages are automatically handled by a user defined handler when they arrive. To support RPC-like behavior, i.e., "receiving a reply just after sending a request" that happens often in parallel programs, Jojo also provides a set of "call" methods.

The message passing API is carefully designed to enable overlapping of data transfer and execution, taking advantage of Java multithreading. The details of the API will be shown in the next section.

3.3. Invocation of servers

In a Grid environment, we cannot always assume the existence of shared filesystems. Installing all the user programs properly to all the servers that have been brokered can be a heavy burden for users. Jojo automatically stages all the user programs from the client node to all the server nodes.

As a matter of a fact, the core portions of the Jojo system itself is also automatically staged to the servers, avoiding potential user misconfigurations, thereby greatly reducing the burden of installation and maintenance of latest versions of Grid middleware.

4. The System Details of Jojo

4.1. Invocation on Remote Hosts via *RJava*

To reduce the users' burden of installation, Jojo automatically stages not only the users program but also the system program itself. The automatic staging is enabled by the bootstrapping server *RJava*[9] as follows:

```
abstract class Code{
  Node [] siblings;    // Sibling nodes
  Node [] descendants; // child nodes
  Node    parent;      // parent nodes
  Node    self;        // reference to myself
  int     rank;        // order in the siblings

  public void init(Map arg); // initialize
  public void start();       // the body
  public Object handle(Message mes);  // handler
}
```

Figure 2. Code Class

1. The bootstrapping server module is encapsulated in a jar file, which includes a customized class loader which is designed to load classes via network connection.
2. The RJava client sends the jar file to the server and invokes RJava server using the jar file, making connection between client and server.
3. The RJava server invokes the Jojo system classes which are loaded by the customized class loader. The net result is that the Jojo class files are actually loaded from client file system and sent to the server.
4. The Jojo system classes, in turn, invoke the user supplied classes, which are also loaded on the client side and transferred to the server.
5. On the client side, the Jojo system classes are invoked by the RJava client. The user classes are also invoked by the Jojo system classes.

RJava supports Globus GRAM and ssh/rsh as Invocation Protocols. In ssh/rsh mode, the client ships the bootstrap server jar file to the server by using scp/rcp in advance, and then invokes it on a Java VM using ssh/rsh. In the Globus mode, Jojo uses file staging facility which is provided by the GRAM protocol to ship the jar file.

For multiplely-layered invocation, the process described above will repeatedly occur. The class file load requests, issued by the class loader embedded in the bootstrap server, is always delegated to the original client host.

4.2. The Jojo Programming API

To write programs for Jojo, programmers have to "extend" an abstract class called `Code` using supporting classes called `Node` and `Message`.

4.2.1. The Code Class Figure 2 shows the definition of the `Code` class. The class has references to the siblings, descendants, and parent nodes. It also has a reference to itself.

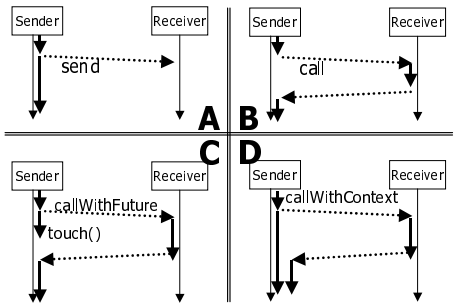

Figure 3. Various Communication Modes

These reference can be transfered to other nodes by embedding them in node-to-node messages.

The programmer has to implement three methods; `init`, `start`, and `handle`. The `init` method is the initialization code for the class. The properties specified by the user as the invocation argument will be passed as the argument for this method. The `start` and the `handle` method will never be called until this method finishes.

The `start` method is the main program for the class. This method will be called by the system, just once.

The `handle` method is the message handler. This method will be called every time when the node receives a message from other nodes. Each message is handled in a dedicated thread to enable overlapped handling of messages. Therefore, however long the time may be required to handle a message, other messages will not be affected. On the other hand, programmers have to take care of racing conditions when handlers access shared data.

If a programmer is not willing to take on this burden, he/she can easily make all the handler invocations serialized by just putting `asynchronous` modifier to this method.

4.2.2. The Node Class The instances of this class are references to the other nodes in the system, and provide communication methods to other nodes. This class implements four types of communication for programmer flexibility.

`void send(Message msg)`
: This method simply sends a `Message` object, described below, to the target node and returns immediately(figure 3:A).

`Object call(Message msg)`
: This method sends a `Message` object, waits for reply from the communication peer, and returns the object to the caller(figure 3:B).

`Future callFuture(Message msg)`
: This method provides future based asynchronous communication: it sends a `Message` object and immediately returns with a `Future` object without waiting for a reply. The caller can access the reply by calling the `touch()` method of the `Future` object. If the reply has already arrived, the method immediately returns, and if not, it blocks waiting for the reply crom the callee (figure 3:C).

`void callWithContext(Message msg, Context context)`
: This method provides another asynchronous communication method: it returns immediately after sending the message. It takes a Object that implements `Context` interface which have `run(Object o)` method. When the reply arrives, the `run` method will be invoked with the reply object as the argument, in a newly created thread (figure 3:D). Please note that proper synchronization is required when the `run` method accesses shared data as mentioned earlier.

4.2.3. The Message Class The `Message` class represents messages transfered among nodes. The class have three fields: the integer `tag` field, the `contents` field with type `Serializable`, and the `from` field whose type is `Node`. `Tag` represent message id number to distinguish message, and can be arbitrary defined by the programmer. The `contents` field stores the message body object. The `from` field represents the message sender node. This field only makes sense at the receiver side, and is automatically instantiated by the system during the transfer of the message.

4.3. Configuring the Jojo system on a Hierarchical Grid

To start up a Jojo program, programmers have to specify the configuration of computers available on the Grid, invocation protocols to be used, and `Code` class to run on each node. Since the configuration of computers can be multi-hierarchical, the configuration file format itself has to be able to express such kind of configuration. Since the Java standard properties file format cannot meet this condition, we define an XML-based configuration schema for the file format.

There are three elements in the schema; `node`, `code`, and `invocation`. The `node` element defines a node, embodying a `host` attribute which specifies the hostname of the node. It also defines a code element, an invocation element (both of which can be empty), and zero or more `node` element in it. `Node` elements in a `node` element stand for computers subject to invocation by the containing node. The `code` element specifies the classname for the code. The `invocation` element specifies invocation protocols and parameters to invoke the code. Additionally, the value of the host attribute allows the attachement of the default keyword. The `node` element definition that has this keyword is treated as the default values for other nodes, allowing con-

cise description of node configurations. A sample working configuration file will be shown in section5.

4.4. File I/O redirection

File I/O is often required during program execution for reading configuration and/or input data files, and writing results and/or logs to files. Jojo provides (almost) transparent access to the file system of the client node, enabling programmers to write their programs without concerns for of the location where the program actually runs, without explicit support of distributed filesystems such as NFS/AFS in the backend clusters (in fact in a true Grid environment NFS not AFS would not appropriately work.)

The use of this facility is simple: merely substitute Jojo classes `RemoteFileReader` and `RemoteFileWriter` instead of Java standard `FileReader` and `FileWriter`. This facility is implemented as files being streamed in real time instead of being staged in/out. As a result, information logged by a server side program will instantly appears on the client file system, enabling realtime monitoring of the server program behavior.

5. Sample Program

Here, we show a master-worker program to calculate PI using the Monte-Carlo method. Figure 4 shows the master and 5 shows the worker. This program performs dynamic load balance using self-guided scheduling; the worker requests a chunk of jobs from the master, solves it, returns the result, and request another chunk. In this program returning a result and requesting jobs are encoded into a single message to simplify the program.

To execute this program, Jojo requires a configuration file and a properties file. The configuration file shown in figure 6 specifies to execute the program with ssh connection, using computers named pad00, pad01, pad02 and pad03. Note that `code` and `invocation` elements are efficiently shared by all the nodes, using the "default" notation.

Properties file for this program is shown immediately below:

```
times=100000
divide=100
```

The attributes in the file specifies Monte-Carlo trial times and division number for load distribution.

The program is executed from the command line in the following manner ,where configuration file and properties file are named `jojo.conf` and `pi.prop`, respectively.

```
> Java silf.jojo.Jojo jojo.conf pi.prop
```

```
public class PiMaster extends Code{
 long times, perNode;
 int  divide;
 boolean done = false;
 long doneTrial = 0, doneResult = 0;

 public void init(Map prop) throws JojoException{
  times =
    Long.parseLong((String)prop.get("times"));
  divide =
    Integer.parseInt((String)prop.get("divide"));
  perNode = times / divide;
 }

 public void start() throws JojoException{
  synchronized (this){
   while (!done){
    try {wait();}
    catch (InterruptedException e) {}
   }
   System.out.println("PI = " +
     (((double)doneResult/doneTrial)*4));
  }
 }

 synchronized public Object
 handle(Message msg) throws JojoException{
  if (msg.tag == PiWorker.MSG_TRIAL_REQUEST){
   long [] pair = (long[])(msg.contents);
   doneTrial  += pair[0];
   doneResult += pair[1];
   if (doneTrial >= times){
     done = true;
     notifyAll();
     return new Long(0);
   } else
     return new Long(perNode);
  } else
   throw new JojoException(
    "cannot handle the message: " + msg);
 }
}
```

Figure 4. Master Program

```
public class PiWorker extends Code{
 static final int MSG_TRIAL_REQUEST = 1;
 Random random = new Random();

 public void start()
 throws JojoException{
  long trialTimes = 0;
  long doneTimes  = 0;
  while (true){
   Message msg =
     new Message(MSG_TRIAL_REQUEST,
                 new long[]{trialTimes, doneTimes});
   trialTimes =
              ((Long)(parent.call(msg))).longValue();
   if (trialTimes == 0) break;
   doneTimes = trial(trialTimes);
  }
 }

 private long trial(long trialTimes){
  long counter = 0;
  for (long i = 0; i < trialTimes; i++){
   double x = random.nextDouble();
   double y = random.nextDouble();
   if (x * x + y * y < 1.0)
    counter++;
  }
  return counter;
 }
}
```

Figure 5. Worker Program

```
<node host="root">
 <code> PiMaster </code>
 <node host="default">
   <code> PiWorker </code>
   <invocation
    javaPath="java"
    rjavaJarPath=
      "/usr/users/nakada/bin/rjava.jar"
    rjavaProtocol="ssh"
    rjavaRsh="ssh"
    rjavaRcp="scp"
   />
  </node>
  <node host="pad00"/>
  <node host="pad01"/>
  <node host="pad02"/>
  <node host="pad03"/>
</node>
```

Figure 6. Configuration file for the sample program

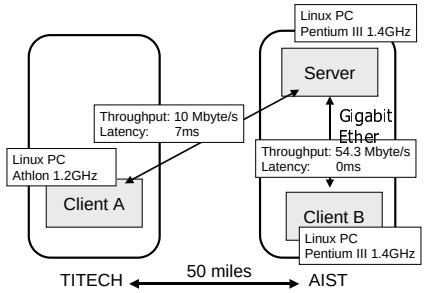

Figure 7. Evaluation Environment for throughput.

6. Preliminary evaluations

To evaluate the basic performance of Jojo, we measured its throughput between nodes. And to validate its layered architecture, we conducted an experiment with the simple master-worker program shown in the previous section.

6.1. Throughput Evaluation

We measured throughput using Globus (hereafter referred to as `GSI`) and ssh(hereafter referred to as `SSH`), in an environment shown in figure 7. We call **WAN** the experiment between TITECH and AIST, whose physical distance is approximately 80kms, and **LAN** the experiment inside AIST.

Figure 8 shows results measured in LAN. SSH and GSI exhibited 12Mbyte/s and 8Mbyte/s. Although reasonably fast, compared to the native Gigabit Ethernet bandwidth we are penalized considerably. Also, GSI exhibited somewhat

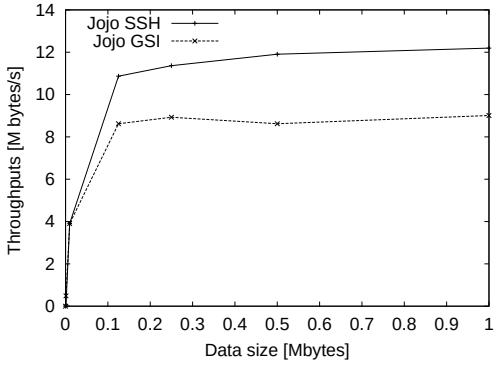

Figure 8. Throughput in LAN Environment

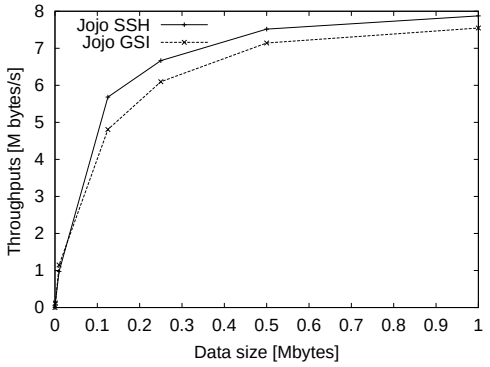

Figure 9. Throughput in WAN Environment

slower performance compared to SSH. We believe that the overhead is mainly due to the encryption/decryption performed in SSH and Globus I/O, as well as stream multiplexing and switching that occurs within RJava.

Figure 9 shows the results measured in WAN. We can see that the difference between GSI and SSH is smaller, and they both exhibit approximately 70% of the 10Mbyte/s bandwidth which is the speed cap between TITECH and AIST. This is because the low speed network is hiding the overhead posed by Jojo and the underlying SSH/GSI.

6.2. Evaluation by Master-Worker program

Here, we validate the effectiveness of the layered architecture, comparing performances on two different setup for a master-worker program.

6.2.1. Evaluation Environment We used the program shown in section 5. We used a PC cluster installed at AIST via the Internet connection from a remote client. Figure 10 shows the cluster performance and network setup.

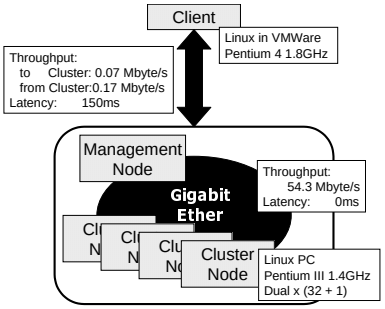

Figure 10. Master-Worker evaluation environment.

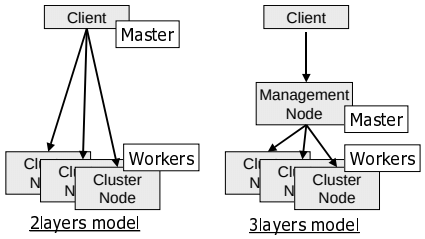

Figure 11. Master-Worker setup.

We set up two experiments; **2-layered** and **3-layered**. For 2-layered setup, we ran the master on the 1st layer and the worker on the 2nd layer. For 3-layered setup, we ran the master on the 2nd layer and the worker on the 3rd layer; the 1st layer does nothing (figure 11).

We used the same PC cluster for workers. In the 3-layered setup, we employed the administration node of the cluster as the 2nd layer node. Since the administration node is directly connected to the same network switch as the cluster nodes, the communication speed between administration node and a cluster node is same as the speed between cluster nodes.

We conducted a Monte-Carlo simulation of 100 million trials, divided into 10,000 jobs of 10,000 trials, in a master-worker setup. 10000 trials take approximately 8ms on a single node. We used ssh for all communication [1].

Figure 12 shows the result. The single-node line shown in the graph represents the performance of a single program execution time on a node of the cluster.

1 Since ssh does not provide single sign-on capability, we cannot use ssh for the connection between the 2nd and the 3rd layer. Here, we cope this problem as follows. In advance, we logged in the second-layer, started up ssh-agent and got the SSH_AUTH_SOCK and SSH_AGENT_PID environment variable values that are required to access the ssh-agent. We gave Jojo these values as the configuration file entry. The Jojo 2nd layer node accessed the ssh-agent with these values and invoked 3rd layer.

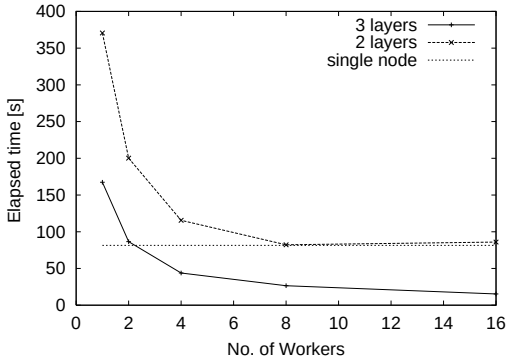

Figure 12. Master-Worker result in 2layer and 3layer.

We can see that 3-layered model is faster than 2-layered model for all the number of nodes, and 2-layered model does not show speed up for nodes larger than 8, while 3-layered model exhibits speedup up to 16 nodes.

6.2.2. Discussion In this program, data transferred between master and worker is just a single long integer. For such types of programs, communication throughput between master and worker does not have big impact on the performance, but latency does. When a worker finishes a job, it requests a new job to the worker and idles till the new job arrives. The idle time depends on the latency between the master and workers, and excessive idle time causes performance degradation.

The program used in this experiment would usually be considered unsuitable for parallel execution on the Grid, since the worker runs just 8ms per each work assignment, far shorter than network latencies in a wide-area environment. This is supported by the low performance in the 2-layered setup experiment. On the other hand, 3-layered setup shows sufficient speedup. This is because the communication between the master and workers occurred only inside the high-speed local area network. This result implies that certain classes of applications usually considered being not suitable for the Grid can be effectively executed on the Grid using multi-layered architecture provided by Jojo.

To conduct these experiments, neither program installation nor daemon program start-up are required for the server side clusters. Only the SSH/RSH daemon or Globus GRAM gatekeeper needs to be running inside each cluster node. This fact supports effectiveness of automatic user/system program shipping provided by Jojo .

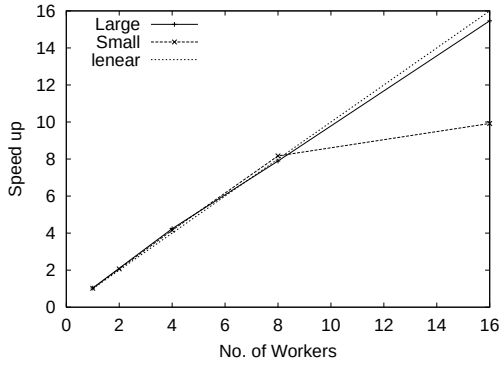

Figure 13. Speed up of protein 3-dimensional structure optmization.

6.3. Evaluation using a Real Application

Here, we show the result of an evaluation using a real application. As the application, we employed protein 3-dimensional structure optmization using NMR spectroscopy. We solved this problem using genetic algorithm according to a literature[7]. We parallelized the genetic algorithm program using Jojo based on master worker style.

As an evaluation platform, we again used the cluster shown in figure 10. For this experiment, we set up all the master and workers on the cluster. We performed two experiments with two proteins; *small*, with 13 residual, and *large*, with 27 resudual.

Figure 13 shows the speedup against the number of workers. With a small protein, speedup saturates for more than 8 workers, in contrast to the case with large protein which scales well up to 16 workers.

7. Conclusion

We proposed a Java based communication library that is suitable for the Grid environment consisting of privately-addressed PC clusters. It allow easy specification of multi-hierarchical communication tree on PC clusters and its utilization to achive effective master-worker computation. It also provides easy-to-use and multi-thread-aware message passing API, suitable to hide latency in the Grid environment.

The evaluation shows that its basic throughput is acceptable, and for master-worker programs multi-hierarchical composition is effective.

For the future work, we will address following issue:

- Validation of scalability. Jojo is designed with scalability in mind, but its scalability is not proven yet. We will conduct experiments using large scale setup with more than 1000 CPUs to confirm scalability on a large Grid to be installed consisting of nodes at AIST and TITECH.
- Fault tolerance. Currently, Jojo does not provide any capability for fault tolerance. If one node dies during computation, the whole system may freeze or fail, just as it is with MPI programs. We will address this issue defining higher level API on top of or with a small extension to the Jojo API.

References

[1] Jcluster. http://vip.6to23.com/jcluster/.

[2] K. Aida, W. Natsume, and Y. Futakata. Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm. In *Proc. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, 2003.

[3] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and S.Lim. mpijava: An object-oriented java interface to mpi. In *International Workshop on Java for Parallel and Distributed Computing*, 1999.

[4] F. Breg, S. Diwan, J. Villacis, J. Balasubramanian, E. Akman, and D. Gannon. Java rmi performance and object model interoperability: Experiments with java/hpc++. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998.

[5] A. J. Ferrari. Jpvm: network parallel computing in java. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998.

[6] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[7] I. Ono, H. Fujiki, M. Ootsuka, N. Nakashima, N. Ono, and S. Tate. Global optimization of protein 3-dimensional structures in nmr by a genetic algorithm. In *Proc. 2002 Congress on Evolutionary Computation*, pages 303–308, 2002.

[8] A. Roy, I. Foster, W. Gropp, N. Karonis, V. Sander, and B. Toonen. MPICH-GQ: Quality-of-Service for Message Passing Programs, November 2000.

[9] Y. Sohda, H. Nakada, H. Ogawa, and S. Matsuoka. Implementation of portable software dsm in java. In *Proc. of JavaGrande 2001*, pages 163–162, June 2001.

[10] Y. Tanaka, M. Hirano, M. Sato, H. Nakada, and S. Sekiguchi. Performance evaluation of a firewall-compliant globus-based wide-area cluster system. In *9th IEEE International Symposium on High Performance Distributed Computing (HPDC 2000)*, pages 121–128, 2000.

[11] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-g: A reference implementation of rpc-based programming middleware for grid computing. *Journal of Grid Computing*, 1(1):41–51, 2003.