Autonomically-Adapting Master-Worker Programming Framework for Multi-Layered Grid-of-Clusters

Hitoshi Aoki
Tokyo Institute of Technology
2-12-1 Ookayama, Tokyo, 152-8550, Japan
hitoshi@smg.is.titech.ac.jp

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

Kouji Tanaka
Waseda University
513 Wasedatsurumaki, Tokyo 162-0041, Japan
k.tanaka@waseda.jp

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

Abstract

Past work on "programming for the grid" for compute-intensive applications have largely focused on two extremes, either loosely-coupled, cycle-scavenging, desktop grid environments such as BOINC or Condor vs. tightly-coupled metacomputing systems such as MPICH-G2 or GridMPI. Neither is really appropriate for class of applications that are middle-tier, e.g., fine-grained branch-and-bound master-worker applications where the running time of individual jobs may range from less than tenth of a second to few minutes, and communication intervals being further fine-grained. Our new grid programming middleware and framework, **Jojo2***, allows efficient programming of fine-grained, hierarchical master-worker applications on a grid-of-clusters environment. During programming, much of the complexities associated with hierarchy and changes in the underlying resources are hidden away or isolated, and the user need not be aware of physical configuration of the resources. Even during execution, new compute resources can be explicitly be added via batch submissions, and are subsequently automatically detected and incorporated in the system. Evaluation of Jojo2 on real master-worker applications proved very positive, both on TSUBAME, a supercomputing cluster with 10,000 nodes, as well as a nationwide testbed involving more than 800 CPUs and a number of 100-node class clusters. In both cases, applications on Jojo2 not only scaled very well, but autonomously adapted to bulk changes in the order of hundreds of underlying resources, both resources being added as well as those going away.*

1. Introduction

Past work on "programming for the grid" for compute-intensive applications have largely focused on two extremes. One is loosely-coupled, cycle-scavenging, desktop grid environments such as BOINC[2] or Condor[11], where compute cycles are largely provided for free, jobs are principally independent and parameter-sweep, where eager resource allocation policies work well, and adaptation to changing requirements of the underlying compute resources are achievable in a fairly natural fashion. The other is tightly-coupled metacomputing systems such as MPICH-G2[6] or GridMPI[4] where resources are often metered and allocated at the beginning of job execution, and fundamentally being difficult to adapt to underlying changes in the resources, if feasible at all. Neither is really appropriate for class of applications that are middle-tier, e.g., fine-grained branch-and-bound master-worker applications where the running time of individual jobs may range from less than tenth of a second to few minutes, and communication intervals being further fine-grained.

Such applications have great possibilities to scale on a multi-layered cluster-of-grid environment, where abundant resources could be provided by grid resource centers in a metered fashion, and medium granularity of the running time and communication made possible by efficient localized communication. The problems then are twofold (1)How to provide a program framework to allow for programming such applications in a cluster-of-grids environment efficiently, independent of

the underlying physical configuration of the grid, and (2) how to allow explicit resource control by the application user as well as the system administrator, such that increase and decrease in the resources according to application progress, available resources, and/or available allocation time. Where, much of the bookkeeping and reconfiguration on the application side would be achieved in an autonomic fashion. In fact (2) is also closely tied to longevity and reliability of long-running applications in such an environment, where resources will go away not only for physical hardware faults, but also software faults as well as non-faulty events such as jobs explicitly being killed on backfill scheduling.

Our new grid programming middleware and framework, Jojo2, allows efficient programming of fine-grained, hierarchical master-worker applications on such grid-of-clusters environment. During programming, the programming model is that of multi-layered master-worker method [1, 5], and much of the complexities associated with such hierarchy and changes in the underlying resources are hidden away or isolated, and the user need not be aware of physical configuration of the resources. Even during execution, new physical compute resources can be explicitly be added via batch submissions, and are subsequently automatically detected and incorporated in the system. Resources that go away are also detected and the application as well as the system reconfigured accordingly. Callback hooks are provided in the API for cases where such adaptations also require manual interventions.

Evaluation of Jojo2 on real master-worker application in genetic programming proved very positive, both on TSUBAME, a supercomputing cluster with 10,000 nodes, as well as a nationwide distributed grid testbed involving with more than 800 CPUs consisting of a federation of 100-node class clusters. In both cases, applications on Jojo2 not only scaled very well, but autonomously adapted to bulk changes in the order of hundreds of underlying resources, both resources being added as well as those going away. The results show that 1) API in Jojo2 is able to allow programmers to write proper handlers to allow a program can adapt to the environmental change, 2) it scales well with more than one thousand CPUs, and 3) it adapts to the changing environment with acceptable overhead

Additional contributions of the paper are as follows:

- We propose a mechanism to automatically configure internal communication channel within a cluster. It uses UDP packet broadcast for component discovery. This allows less configuration effort for the users and on-the-fly addition and removal of nodes to the system.

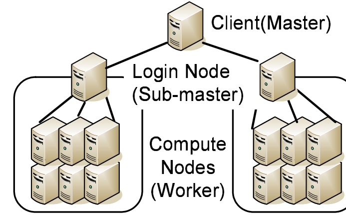

Figure 1. Multi-layered Master-worker mapped on the Typical Grid-of-Clusters

- We propose a programming API that allows programmers to write medium to fine-grained adaptable programs on the grid with minimum effort, by imposing restrictions on message passing and providing a uniform message handler mechanism.

2. Multi-layered Master-Worker on the Hierarchical Grid

Master-Worker computation is a simple yet powerful computation model in which a master manages a queue of jobs and assigns jobs on request from the workers. Large number of computations can be mapped into this model, such as parameter-sweep computations, branch and bound, and genetic algorithms. One notable characteristics of this method is that load balance amongst workers is automatically performed without any effort.

One potential shortcoming in the model is scalability. With huge number of workers, the master which is in charge of distributing jobs could be the bottleneck and make some of the workers starved. To avoid this problem, **Multi-Layered Master-Worker** [1, 5] model was proposed, where submasters are inserted between master and workers. Submasters locally manage local job queues and distribute jobs to downstream workers. Masters communicate with submasters to balance the length of local job queues managed by the submasters. While master-submaster communication tends to be in bulk, submaster-worker communication is more fine-grained and frequent.

On the other hand, grids also have multiple layers. Modern scientific computation Grids are typically composed as a grid-of-clusters, where several clusters, each of them is managed by its own queuing system locally, are connected with high-speed wide-area network. The upper layer of such a Grid is the wide-area network, where latency is high, throughput is relatively low, number of participating nodes is small, and security requirements are severe. The lower layer is the internal network within the cluster, where latency is low,

throughput is high, number of nodes is large, and security requirements are not severe. Between these two layers there are "login nodes" that connect two networks.

We can map the multi-layered Master-worker on to the Multi-Layered Grid as shown in figure 1, i.e.; put submasters on the login nodes and workers on the nodes within the clusters. Thus, we can map the two different natured communications on to the appropriate networks, i.e., less frequent bulk communication between master and submasters will be performed on the high-latency internet, and frequent fine-grained communication between workers and submasters will be mapped onto the fast, low-latency internal network, avoiding starvation comes from job supply delay.

3. Design of Jojo2

3.1. Requirements

The requirements for the middleware to support multi-layered master-worker computation in the large scale Grid environment are as follows:

Autonomic network configuration It has to allow nodes to be added or removed by application users as well as administrators. For that, the middleware have to 1) automatically detect the addition/removal and 2) autonomically establish/shutdown connections

Programming API to allow adaptation It has to provide programming API that allows programmers to write a program that can adapt node joining/leaving.

Allow explict resource controll Explicit resource control by the users and administrators during application execution over multiple production-managed clusters During application execution, users and administrators should be able to add or subtract compute resources at will, with software facilities readily available across multiple clusters such as (different varieties of) batch queuing systems.

3.2. Network Configuration

Jojo2 employs two different network configuration methods, one for external and another for the internal network. For external network, i.e., between master and submasters, it takes straight forward approach. The master invokes submasters and the submasters connect back to the master.[1] The hostnames to execute submasters have to be specified in the configuration file by the users.

1 This describes the case with Globus GRAM. With SSH, it is even simpler; the master invokes submasters with SSH and use the connections as the communication channels.

For internal network, i.e. between submaster and workers, it uses discovery-based network configuration. The workers "discover" a submaster and connect to it. For discovery, we employed UDP packet broadcast. The submaster periodically broadcast a packet that contains contact information to the submaster. The workers listen to the packet and get the information. Note that it is allowed to have several submasters in a single cluster, which is important especially in a large-scale cluster.

Workers can be invoked via local queuing system, such as Condor, SGE, or PBS. It allows users to invoke workers without knowing each node in the system. Note that the worker invocation can be performed via Globus with appropriate job managers. In this case, users do not have to login the cluster at all. The workers can be added at any time.

To detect nodes removal and/or node/network faults, Jojo2 uses heartbeat monitoring. Each component sends heartbeat message periodically each other. When a component does not receive the message from another component for a specific time period, it recognizes the peer has died.

3.3. The programming API

Jojo2 provides users a programming API designed focusing on 1) hierarchical modeling that fits in the hierarchical Grid Environment, 2) callback handles to adapt dynamic node join/leave, and 3) Multicast based message passing structure to be tolerant of node number fluctuation.

Basic Concept In Jojo2, each node runs its own independent program. Each program sends and receives message with each other and perform computation. While Jojo2 allows each single node to have different programs, nodes in a layer tend to share a single program. The API is designed to be as asynchronous as possible so that it allows programmers to write programs in a non-blocking fashion.

Adaptation to the changing Environment While the message passing API is designed to be tolerant of the node join/leave, as discussed below, there are still few things left for the programmers. To write an adaptive program for changing environment, the programmers have to be notified the change and address them. Jojo2 provides handlers that notify the joining/leaving of downstream nodes.

Message Passing In Jojo2, each program that runs on each node is implemented as a class that extends an abstract class provided by the system. The abstract class provides objects that abstract the upstream node and

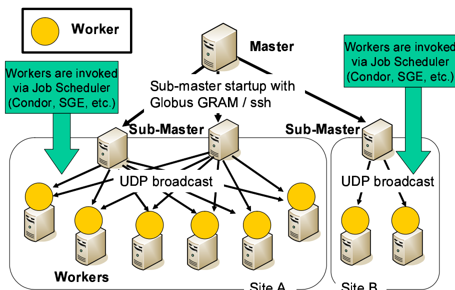

Figure 2. Dynamic Node Configuration

downstream nodes. Invoking "send" methods on the objects will send messages to the corresponding nodes.

In contrast with send, there is no explicit "receive" method. Instead, the arriving messages will invoke handlers, in an independent thread.

To communicate to an upstream node, various message passing modes are supported, since the programmers safely assume that the upstream node is always there. For downstream nodes, however, this may not be the case. Each node might disappear, and new node might join at any time. Sending messages to disappeared nodes will cause exceptions and would otherwise require the programs to deal with the situation explicitly.

To avoid this situation, we allow only the multicast style message passing to the downstream nodes. As the result, the programmer does not have to manage downstream nodes one by one, but rather, downstream nodes appear in the program as one single object. Although this might seem somewhat restrictive in terms of generality in writing parallel programs, in practice we have found that such features a sufficient to describe every master-worker style program we have encountered.

4. Implementation

4.1. Dynamic Node configuration

Jojo2 provides autonomous node discovery and configuration based on the UDP broadcast. Here we describe the steps to construct 3-tiered Grid composed of two clusters (figure 2). We assume that the clusters to be of general production in that it can be accessed via Globus or SSH, and have some queuing system to manage the nodes shared across multiple users and jobs. Each node only is connected to a private cluster interconnect and no direct connection is allowed from outside to each node.

1. The user starts up the master component on the client node with a configuration file specified.

2. The master component invokes submasters on the login nodes of the clusters specified in the configuration file, with Globus GRAM [3], or SSH. For GRAM, the fork job-manager is recommended, since the submasters have to run on the head node that has connectivity to the global network as well as local.

3. The Submaster starts to broadcast UDP packets periodically. It advertises the number of currently connected workers for load balancing among submasters.

4. The user submits workers to the cluster node via a workers to the cluster node via queuing system. This can be done from the client node via Globus with appropriate job managers, such as PBS or Condor, or from the login node by the user who logged in there via SSH.

5. The workers start with waiting for the UDP packets from the submasters. They will find a submaster by receiving a UDP packet and make connection to it.

   To avoid excessive concentration to one submaster, workers do not connect to the first found submaster immediately. Instead, it will wait for a while for UDP packets from other submasters and connects to the least loaded submaster.

4.2. On-the-fly Component Addition

Submasters keep broadcasting UDP packets periodically so that it can be found by the workers there. To add nodes, users just have to submit the workers without any configuration, via its local queuing systems. The invoked workers will wait for the broadcast after they start up and then connect to the submaster.

A submaster also can be added on-the-fly. The master is always listening on a specified port. Invoking submasters with the IP address and port of the master will add new submasters and make them participate with computation. This means that whole new clusters can be added to the computation on-the-fly. To add a new cluster, one merely invokes submasters on the cluster specifying the master address and port, and invoke workers there.

4.3. Adaptation to Component removal

Here, we describe the steps will be taken when each component goes down, either via a failure (including

```
abstract class Code {
  ParentNode parent; /* Upstream node */
  Descendants descendants; /* Downstream */

  /* initialization  */
  void init(Map prop);

  /* the body  */
  void start();

  /*
   * handlers to handle received messages
   */
  void   handleReceiveParent     (Message msg);
  Object handleReceiveDescendant (Message msg);

  /*
   * callback methods for
   *  join/leave of down stream nodes
   */
  void handleAddDescendantNode    (int nodeID);
  void handleDeleteDescendantNode (int nodeID);
}
```

Figure 3. The *Code* Class

job kills via backfill), with explicit user or administrator intervention, or when the allocated compute time by the queuing system expires. When a worker is removed, the corresponding submaster will notice the event and invokes a handler method to notify the program, is described in section 4.4 in detail. The program has to handle the event and perform bookkeeping if needed. The handler will run in a separate thread so that the main thread can continue the execution.

When a submaster dies, the upstream component, i.e., the master will notice and invoke the handler, just like above. The master can be configured so that it automatically tries to invoke a new submaster on the same node. The downstream components, i.e., workers, will lose connection with the submaster, and go back to the waiting status for UDP packets from other submasters. If there are other submasters in the subnet that keep broadcasting UDP packets periodically, the worker will find it and connects to the submaster.

4.4. Class Framework for Programming API

Programming in Jojo2 is to extend the *Code* abstract class for each node. In simple 2-tier master-worker model, a programmer has to implement master and worker extending the *Code*. In 3-tier master-worker, the programmer also has to implement the intermediate submaster code.

Code Figure 3 shows the skeleton of the *Code* abstract class. Member *parent* and *descendants* denote upstream and downstream nodes, respectively. Calling methods on them will send messages to the corresponding nodes.

The *init* method will be called when the object is created. The argument *Map* includes contents of the properties file that is passed to the system at startup. It is guaranteed that other methods including *start* and handlers will never be invoked before the *init* invocation finishes. The *start* method is the “body” of the code, corresponds to the *main* in the usual Java application.

The *handleReceiveParent* and the *handleReceiveDescendant* are the handlers for messages from upstream and downstream, respectively. They are invoked when a message arrives with the message. The *handleAddDescendant* and the *handleDeleteDescendant* are “hooks” to inform programmers of the change in the running environment. They are called on node joining/leaving to the system. Note that these methods are invoked in a separate thread, so that one event handling will not cause delay in the handling of following events.

ParentNode *ParentNode* class stands for the upstream node. The *ParentNode* provides four varieties of methods to send messages, just like in our previous work Jojo[5].

Descendants Class *Descendants* stands for downstream nodes. Note that all the downstream nodes are represented as a single object, such that they are not treated separately. *Descendants* class just has a single method for sending messages; *broadcast(Message msg)*, that sends same messages to all the downstream nodes.

Message *Message* stands for the messages exchanged between nodes. *Message* contains integer tag to help dispatching the message as well as serialized object as the content.

4.5. A program example in Jojo2

Figure 4 and figure 5 show a fragment of simple program pair written in Jojo2, which computes PI in master-worker fashion with the Monte-Carlo method. Figure 4 shows the master code while figure 5 shows the worker code. This program will dynamically balance load distribution with self-scheduling.

Master is waiting for requests from the workers and provides jobs. Workers contacts to the master and obtain the number for the trial, perform the random trial, and report the results to the master. Note that we combine the job request and result report into a single message to simplify the program.

The *handleDeleteDescendantNode* method in the master code is handling node leaving from the running environment. The master memorizes jobs currently in charge of each node. When the master detects a node is leaving, it will return the job to be done by the node into the job queue, so that the job will be re-assigned to other nodes in the future.

```
public class PiMaster extends Code {
  boolean done = false;
  long times, doneTrial = 0, doneResult = 0;
  LinkedList<Long> jobQueue = new LinkedList<Long>();
  HashMap<Integer, Long> jobMap =
                      new HashMap<Integer, Long>();
  public void init(Map args) {  ...  }
  public synchronzied void start() {  ...  }

  public synchronized Object
  handleReceiveDescendant(Message msg) {
    if(msg.tag != PiWorker.MSG_TRIAL_REQUEST)
      return null;
    if(jobMap.containsKey(msg.nodeID)) {
      doneTrial  += jobMap.remove(msg.nodeID);
      doneResult += (Long)(msg.contents);
    }
    while(doneTrial < times) {
      if(! jobQueue.isEmpty()) {
        long perNode = jobQueue.remove();
        jobMap.put(msg.nodeID, perNode);
        return perNode;
      }
      while(jobQueue.isEmpty())
        try {wait();}catch(InterruptedException e) {}
    }
    done = true;
    notifyAll();
    return 0L;
  }

  public synchronized void
  handleDeleteDescendantNode(int nodeID) {
    long perNode = jobMap.remove(nodeID);
    jobQueue.add(perNode);
    notifyAll();
  }
}
```

Figure 4. The Master Program

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

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

  /** give a trial */
  private long trial(long trialTimes) { ... }
}
```

Figure 5. The Worker Program

5. Evaluation

Here we evaluate Jojo2 focusing on its scalability and adaptability with an application that infers genetic networks in a real lifescience application with the GP (genetic programming) technique. [7]

5.1. Evaluation Setup

Genetic network is the control relationships among (true, physical) genes and can be expressed in the form of nonlinear simultaneous differential equations. The application automatically coefficients of the equations

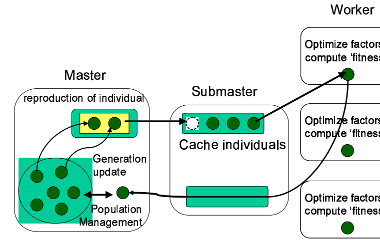

Figure 6. Overview of the Application

based on given data series by using Genetic Programming.

GP is essentially a variant of GA (Genetic Algorithm); where the individuals "genes" (not to be confused with the real genes we are dealing with in the problem domain) sets of equations. Figure 6 shows the overview of the configuration of the application. The master is in charge of the generation of individual "genes", and updating the population. Submasters cache individual genes, and provide them to the workers. They prefetch jobs from master so that it always keeps twice as many as the number of workers that have. Workers retrieve individuals from submasters, optimize factors in the equations, and compute its fitness.

Each worker optimize cofficients in the equation with Runge-Kutta Method to compute its fitness, and the average run time for one job mainly depends on the Runge-Kutta step size. In other words, we can control the granularity of the worker by changing the step size. The smaller the step size we use the larger (or coarser) the granularity we obtain.

We used TSUBAME Grid Cluster as the evaluation platform. Each node in TSUBAME has 8 AMD Opteron chips with 2 CPU cores, and has 32 GBytes memory. Below, we denote each CPU core just as 'CPU' for simplicity. We submitted the workers so that each worker will be assigned one CPU, i.e., each node will be hosting 16 workers.

5.2. Evaluation of the Scalability

We performed experiments to investigate how worker granularity and number of submaster affect on the scalability. We controlled granularity by setting Runge-Kutta step size as $2 \cdot 10^{-2}$ and $1 \cdot 10^{-2}$. The average worker running time for the step sizes are 1086 ms and 2157 ms, respectively. We performed experiments with 128, 256, 512, and 1024 CPUs, varying the number of submaster 2, 4, and 8. We did not perform the experiments with one submaster, because it is not possible to handle 1024 CPUs with one submas-

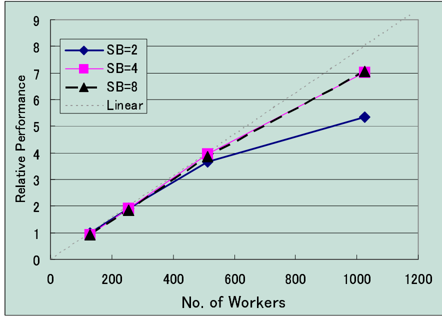

Figure 7. Scalability with Step: $2 \cdot 10^{-2}$

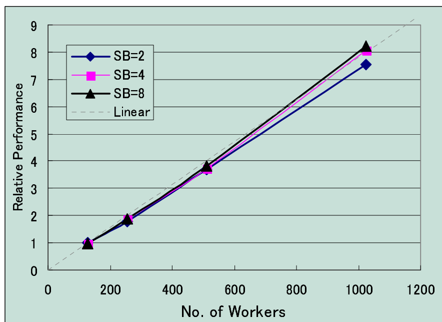

Figure 8. Scalability with Step $1 \cdot 10^{-2}$

ter due to the number of file descriptor limitation per process.

The results are shown in figure 7 and figure 8. *SB* in legend denotes the number of submasters. Note that the performances shown are relative to the case with 128 CPUs with 2 submasters. Both of them show good scalability. Even in the worst case (SB = 2, step = $2 \cdot 10^{-2}$), it shows 5.2 times faster for 1024 CPUs, compared with 128 CPUs.

We can observe the number of submaster matters especially for large number of CPUs and fine granularity. With 1024 CPUs and $2 \cdot 10^{-2}$, 2 submaster shows just 5.2 times speed up while 4 and 8 submaster show more than 7. This is because submasters could become bottlenecks when they have to handle large number of workers with short turn around. Increasing the number of submasters distributes the load and results in better scalability.

Comparing figure 7 and figure 8, we can see that to have proper granularity for worker is very important to obtain good scalability.

	(a)	(b)	(c)
Time spent [sec]	5937	6768	11165
Total CPU · sec.	1519872	1540608	1544320
Relative Efficiency	1.0	0.98	0.98

Table 1. Time spend with disturbance

5.3. Evaluation of Adaptability

To confirm adaptability and adaptation overhead in Jojo2, we performed a set of experiments by "injecting" the following artificial disturbances; (a) No disturbance, i.e., all the worker run from start to end. (b) One half of the workers run from start to end. Another half of them are periodically stopped for 5 min., after they run 15 min. During the run, there were five stops. (c) One half of the workers run from start to end. Another half of them go away after they run 15 min. We used 256 CPUs and the same application as above. We employed $5 \cdot 10^{-3}$ as the Runge-Kutta step size and four submasters.

The results are shown in Table 1. The first row shows time spent for the whole execution, the second shows total CPU·sec. used for the execution, and the third shows relative efficiency compared with (a).

As we can see from the table, total CPU·sec. spent for executions are roughly the same. This means that, re-joined workers are effectively utilized immediately in the computation, Relative efficiency for (b) is slightly smaller than 1.0. It means that there are some overhead due to removing and re-joining workers, but it is acceptably small.

Removing worker from a pool will impose overhead derived from abandoned computation. When a worker is removed, running job on the worker will be discarded and the computation time spent till then for the job is abandoned.

Joining worker also imposes some overhead. When a worker joins a pool, it has to wait for UDP packet arrives from then submaster, and download code into the virtual machine to execute (plus possiblly JIT compilation), before it actually starts computation.

5.4. Experiment in a Wide Area Grid

We also performed an evaluation on a Grid testbed spanning wide area network in Japan, which was setup for Grid Challenge 2006. The testbed was composed of 7 clusters, 862 CPUs in total. While we cannot get numbers from the experiment due to the fact that the testbed was shared by many participants, we confirmed that Jojo is capable of utilizing widely distributed Grid environment.

6. Related work

Our previous work **Jojo**[5], was an attempt to map multi-layered master-worker on to the multi-layered grid-of-clusters. Although it showed good scalability, it cannot adapt to the underlying resource changes.

The work in [1] proposed a system that uses two GridRPC[8] middleware in cascading manner; **Ninf-G**[9] and Ninf-1, to map multi-layered master-worker on to the Grid environment. Their system shares most shortcomings with the previous version of Jojo. It cannot adapt to the changing resources and requires detailed knowledge of participating nodes.

Phoenix[10] is a programming environment that can adapt node number increase and decrease. In Phoenix, they use logical node name for message passing, instead of physical one. The mapping between logical nodes and physical node can be dynamically changed, without affecting the program itself. It cannot adapt sudden-death of nodes, however. Before removing nodes, Phoenix requires remapping nodes and date evacuation. Although Jojo2 is specialized for Master-Worker and is not as versatile as Phoenix, it is more adaptable to changing environment.

7. Conclusion

We proposed a new grid middleware *Jojo2* that allows efficient programming of fine-grained, heirarchical master-worker applicatons on the grid-of-clusters environment. It allows not only nodes faults but also intentional addition and removal of nodes to enable efficient utilization of the grid, assuming that future grid infrastructures will hiearchically consist of a federation of clusters with private addresses for intra-node communication.

We evaluated Jojo2 on a large-scaled Grid environment with a real application and confirmed its scalability and adaptability.

For future work, we will address the following issues:

- Higher level API set is required. The programming API shown here is relatively primitive and the threashold is still high for the end users. For example, handling node joining and leaving could be too much burden for average application programmers, not familiar with the parallel programming. We are implementing a higher level API set that hides these details from the programmer and make them concentrate on their particular domain problem.
- Harnessing with libraries written in other languages is important for serious scientific computation. Jojo2 is written in Java and currently limited to the Java written computation libraries while serious scientific libraries tend to be written in C, C++ or Fortran. We are investigating a way to execute C-written libraries from the leaf node of the Jojo2 so that we can get the benefit of computation speed from C-written library while keeping the flexibility of Java language as a whole system.

References

[1] K. Aida and T. Osumi. A case study in running a parallel branch and bound application on the grid. In *Proceedings of the 2005 International Symposium on Applications and the Internet (SAINT 2005 Workshops)*, pages 164–173, 2005.

[2] D. P. Anderson, C. Christensen, and B. Allen. Designing a runtime system for volunteer computing. In *Proc. of SC06 (the International Conference for High Performance Computing, Networking, Storage and Analysis)*, November 2006.

[3] K. Czajkowski, I. Foster, C. Kesselman, N. Karonis, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[4] M. Matsuda, T. Kudoh, Y. Kodama, R. Takano, and Y. Ishikawa. Efficient mpi collective operations for clusters in long-and-fast networks. In *Proc. of Cluster2006*, 2006.

[5] H. Nakada, S. Matsuoka, and S. Sekiguchi. A java-based programming environment for hierarchical grid: Jojo. In *CCGrid 2004*, 2004.

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

[7] E. Sakamoto and H. Iba. "inferring a system of differential equations for a gene regulatory network by using genetic programming". In *Proc. of the Congress on Evolutionary Computation (CEC2001)*, pages 720–726, 2001.

[8] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. GridRPC: A Remote Procedure Call API for Grid Computing. submitted to Grid2002.

[9] 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.

[10] K. Taura, T. Endo, K. Kaneda, and A. Yonezawa. Phoenix : a parallel programming model for accommodating dynamically joining/leaving resources. In Proc. of PPoPP 2003, pages 216–229, 2003.

[11] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In *Grid Computing: Making the Global Infrastructure a Reality.* John Wiley & Sons Inc., December 2002.