The Design and implementation of a Fault-Tolerant RPC system: Ninf-C

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

Yoshio Tanaka
National Institute of Advanced Industrial Science and Technology (AIST)
1-1-1 Umezono, Tsukuba, 305-8568, Japan
yoshio.tanaka@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

We describe the design and implementation of a fault tolerant GridRPC system, Ninf-C, designed for easy programming of large-scale master-worker programs that take from few days to few months for its execution in a Grid environment. Ninf-C employs Condor, developed at University of Wisconsin, as the underlying middleware supporting remote file transmission and checkpointing for system-wide robustness for application users on the Grid. Ninf-C layers all the GridRPC communication and task parallel programming features on top of Condor in a non-trivial fashion, assuming that the entire program is structured in a master-worker style—in fact, older Ninf master-worker programs can be run directly or trivially ported to Ninf-C. In contrast to the original Ninf, Ninf-C exploits and extends Condor features extensively for robustness and transparency, such as 1) checkpointing and stateful recovery of the master process, 2) the master and workers mutually communicating using (remote) files, not IP sockets, and 3) automated throttling of parallel GridRPC calls; and in contrast to using Condor directly, programmers can set up complex dynamic workflow as well as master-worker parallel structure with almost no learning curve involved. To prove the robustness of the system, we performed an experiment on a heterogeneous cluster that consists of x86 and SPARC CPUs, and ran a simple but long-running master-worker program with staged rebooting of multiple nodes to simulate some serious fault situations. The program execution finished normally avoiding all the fault scenarios, demonstrating the robustness of Ninf-C.

1. Introduction

To perform effective computation with unstable and heterogeneous resources, such as resources in the Grid, the following characteristics are required for the subject programs:

1. Adapting to computational performance heterogeneity among the computational resources.
2. Robustness against increase/decrease of computational resources.
3. Tolerance against and recovery from general faults that occur, beyond what can be coped with by characteristics 1) and 2).
4. Transparency from 1) - 3) at the programming level

The Master-worker model well exhibits characteristics 1) and 2), as well as 4) (transparency) when coupled with appropriate Grid programming middleware. The master manages a task queue, and the workers retrieve tasks from the master and process them and return the results to the master. The master-worker model allows graceful increase and decreases of computation nodes, and adapts well to performance difference among the computation nodes with simple scheduling strategies. It also is somewhat robust to faults in general since the essential state of the system can be centralized to the master program.

In order to program such master-worker applications on the Grid, we believe that RPC (Remote procedure call)

is quite promising, since existing programs amenable to master-worker parallelism can easily be adapted to be parallelized, such as search problems, as well as fine level of control requiring dynamic adaptation required on the Grid can be programmed easily within each application, such as flexible resource selection in matching subprogram granularity with computational power, exploiting network efficiency overlapping of master-worker processing and data transfer, as well as implementing complex worker control strategies such as workstealing, bounds propagation and search termination in branch-and-bound algorithms, etc. Several past work including ours have demonstrated the effectiveness of master-worker programming on the Grid using GridRPC middleware such as Ninf [10] and Ninf-G [11, 14], in applications such as complex numerical optimizations[3, 7], weather predictions, as well as genomic sequencing.

However, most of the GridRPC-based middleware to date is insufficient with respect to item 3) above; they have relied on task retries when faults are detected, being insufficient for worker tasks of long duration. In fact, in a non-dedicated Grid environment, repeated failures have actually known to occur for a variety of reasons, where worker tasks make little or no progress, almost perpetually repeating the dispatch-failure cycle. Moreover, with the exception of Nimrod[5], coping with such situations are not transparent to the programmer, as explicit task state lists must be maintained and properly checked. Even Nimrod does not entirely automate this process as the parameter space where the master will enumerate over to dispatch worker tasks must be explicitly specified (as well as this causing the loss of flexibility mentioned above for GridRPC-based system as mentioned above.)

Ideally, we will want a GridRPC system where the above 4 characteristics can be implemented in a master-worker program in a highly dynamic and faulty Grid environment, with workers that may execute for hours, days, or even weeks per each task execution. To cope with such requirements, we have designed and implemented a new version of our GridRPC middleware Ninf-C, specially tailored for programming such long-running master-worker applications on the Grid. Ninf-C employs Condor[9, 1], a high-throughput distributed job scheduling system developed at University of Wisconsin, as the underlying middleware supporting remote file transmission and checkpointing for system-wide robustness for application users on the Grid. Ninf-C layers all the GridRPC communication and task parallel programming features on top of Condor in a non-trivial fashion, assuming that the entire program is structured in a master-workder style—in fact, older Ninf master-worker programs can be run directly or trivially ported to Ninf-C. In contrast to the original Ninf, Ninf-C exploits and extends Condor features extensively for robustness and transparency, such as 1) checkpointing and stateful recovery of the master process, 2) the master and workers mutually communicating using (remote) files, not IP sockets, and 3) automated throttling of parallel GridRPC calls; and in contrast to using Condor directly, programmers can set up complex dynamic workflow as well as master-worker parallel control structure that appear in modern master-worker programs of very large scale such as complex hierarchical branch-and-bound algorithms [3], replica exchange methods, etc., with almost no learning curve involved. This is made possible by the easy-to-use RPC-based programming interface for master programs, as well as a scientific IDL (Interface description language) to declare the function interface of the worker functions.

To prove the robustness of the system, we performed an experiment on a heterogeneous cluster that consists of x86 and SPARC CPUs, and tested a simple but long-running master-worker program with staged rebooting of multiple nodes to simulate some serious fault situations. Despite that the program as well as IDL was essentially unmodified from the original Ninf version, and thus the new features of Ninf-C was transparent from the programmer, program execution finished normally avoiding all the fault scenarios where it would have definitely failed for the original Ninf, demonstrating the robustness of Ninf-C.

The rest of paper is composed as follows: Section 2 gives an overview of the Condor system. In section 3 we describe the design and implementation of Ninf-C. Section 4 demonstrates a working example program and performs an evaluation of the system using the program. Section **??** describes related work, and we conclude in Section 6 will conclude with future directions for further robustness and scalability in the future versions of Ninf-C.

2. Overview of Condor

Condor is a job queuing system, developed by the University of Wisconsin, aiming to achieve high-throughput computing utilizing unused computers within the campus. Recently, it is also used as a meta-scheduler for Globus managed resources. Condor manages numerous computer resources and assigns them to submitted jobs form users.

Condor uses a flexible mechanism called 'Match Making'[12] to allocate the servers for the jobs. Condor also supports user program check-pointing and preemption to enable flexible priority based job scheduling.

2.1. Condor Architecture

Figure 1 shows the overview of the Condor System. Computers participating in a Condor pool can have 3 roles,

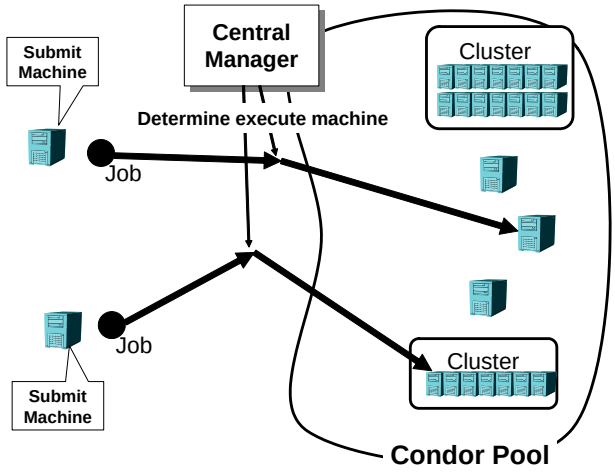

Figure 1. Overview of Condor.

i.e. central manager, submission machine, and execution machines. A computer can play multiple roles. A Condor pool has to have one central manager and one or more submission machines and one or more execution machine. Users submit jobs from the submission machine, and the central manager takes care of allocating an execution machine for each and every job.

Submission machines embody job queues, and send the job information periodically to the central manager. An execution machine monitors its own resource information, such as load average and amount of free memory area, and report it to the central manager periodically. The central manager takes these information and decides a suitable execution machine to be matched with a queued job.

2.2. Condor Universes

Condor supports multiple execution environments and distinguishes them as 'universes'. The most common universes are *standard* and *vanilla*. The vanilla universe is the most primitive universe. In this universe, advanced features such as checkpointing and preemption will not work. In the standard universe these feature can be used, but Condor does not support this universe for several architectures. There are some more universes, including 'globus' universe, for Condor-G.

2.3. Checkpoint and Remote System Call

Condor provides checkpoint and remote system call facility, by replacing the standard C library with a custom made library. In the Condor standard universe, the file I/O system calls performed by executables linked with the custom library will be hooked and executed on the submit machine, not the execute machine. As a result, the executable accesses the submit machine file system, not the execute machine file system.

In addition, the executable will be automatically checkpointed periodically. The checkpoint file will be stored on the submit machine or a checkpoint server if specified. Condor also supports process preemption and migration using the checkpoint file.

2.4. Condor-G

Condor is able to invoke jobs on computers managed by the GRAM gatekeeper, the resource management service of the Globus toolkit. This function is called Condor-G and the universe for Condor-G is called the 'globus universe'. In the globus universe, some of Condor's advanced capabilities, such as check-pointing and remote system-call, do not work, though basic capabilities, such as automatic server selection and automatic restart, are supported. [15]

Condor also can invoke Condor execute machine daemons on Globus managed servers, using Condor-G. This type of invocation, called Condor-Glidein, enables user jobs in the standard universe allowing check-pointing and remote system call.

3. Design of the Ninf-C

3.1. Requirements for robustness

To enable effective and robust execution of long-running master-worker programs, the followings are required:

1. The system can take checkpoint of the master program and restart automatically from the checkpoint file after the machine crash.
2. The system can take checkpoint of the worker programs and migrate them if needed.
3. The system can utilize computing resources that become available during the execution.

Although the Condor almost fulfills these requirements for standalone applications, except for the first item, it is not sufficient for communicating processes, like RPC systems, as mentioned below. We designed Ninf-C to fully utilize the Condor functions and avoid disturbing or replicating them.

3.2. Communication with files

As described above, checkpointing is essential for long-running applications. The problem is that, communicating

processes using sockets are difficult to checkpoint. Most checkpoint libraries, includes the Condor's one, does not support socket communication.

In Ninf-C, the master and the workers communicates via files, not sockets. The contents to be passed are written to files once, and transferred to the target by separated program (the Condor), and read into the target program. This enables checkpointing of the master and the workers.

As a side-effect, all the communication will be 'logged' on the master file system. Using this communication log, the master program is able to rebuild its internal state when it is restarted from checkpoint file. The detail is described in 3.6.

In addition, the file based communication extends the applicability of Ninf-C. For example, Condor-G can utilize privately addressed clusters as backend resources. In this case the executing machine and the submit machine cannot communicate directly with sockets. The file base communication works well for this situation.

3.3. Overview of the Ninf-C

The Ninf-C RPC is implemented using file staging facility provided by Condor. The worker program has to be prepared as a *remote executable* linked with the Ninf-C communication library. The master program sits on the submit machine, and execute the workers by submitting the remote executable as Condor jobs.

Figure 2 shows the details of the Ninf-C RPC implementation. When the master program invokes a worker job as a RPC call with some argument, the Ninf-C runtime library automatically marshals and writes out the argument as a file, and generates a submit file that specifies the worker remote executable as the executable, and the argument file as the stage-in file, a stage-out file to store the result of the RPC invocation. Then, the library submits the submission file by issuing a command named `condor_submit`, which is provided by the Condor.

The library monitors a log file to determine where the Condor system writes information about the submitted jobs, and obtains a Condor job id for the submitted worker job. The worker job will be 'match-made' with a computer resource in the Condor pool. The remote-executable will be staged to the resource, along with the input argument file, and invoked there. The remote-executable re-generates input arguments by reading and un-marshaling of the input argument file, and gives them to the worker function as arguments. When the worker function completes, the executable marshals the result of the function and writes out as the stage-out file, and finishes (terminates). The Condor system automatically detects its termination, stage-back the output file to the submit machine, and logs it to the log file.

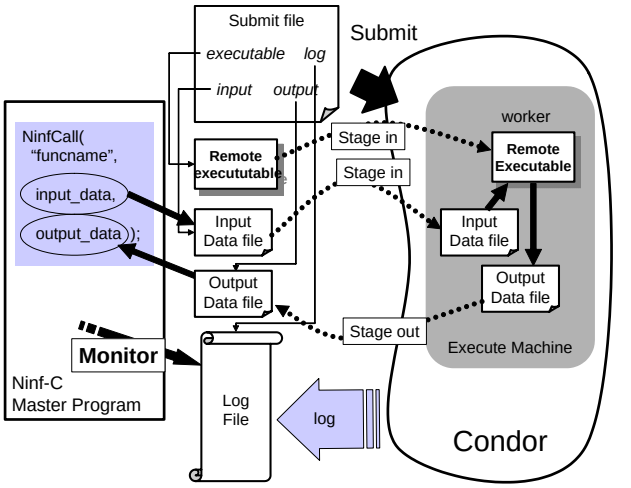

Figure 2. Overview of Ninf-C.

The master runtime library waits for the worker to finish monitoring the log file. When it finds that the worker is finished, it reads in and un-marshals the output file and stores the result of the worker job into a variable specified as the RPC arguments.

3.4. Programming API of Ninf-C

The programmer interface functions of Ninf-C are almost same as the those of our preceding projects: Ninf and Ninf-G, but slightly differ in terms of naming convention. Table 1 shows the important functions of Ninf-C API.

The primary interfaces are NinfCall and NinfCallAsync. These functions invoke remote function specified by the first argument. The former function blocks till the remote function finishes and returns the result. The latter function returns immediately without waiting for the result, and stores the session ID in the second argument. The session ID is used to identify each function invocation.

To wait for a session to be done, use NinfWait with the session ID specifying the session. We also provide several functions that wait for a set of sessions, such as NinfWaitAnd and NinfWaitOr.

3.5. Ninf IDL and its compiler

To use Ninf-C, programmers have to declare a worker function interface using an IDL (Interface Description Language) called Ninf IDL. Here, we omit description of the IDL specification here, since it is completely same as Ninf[10]. Figure 3 shows an example of interface description using the Ninf IDL.

Type	Description
`NinfErrorCode`	Error code
`NinfSessionId`	Session Identifier
Function	**Description**
`int NinfParseArg(int argc, char ** argv);`	Initializes the Ninf-C compornent.
`NinfErrorCode NinfFinalize();`	Finalizes the Ninf-C compornent
`NinfErrorCode NinfCall(char * entry, ...);`	Invokes a remote function specified by *entry*, in a blocking manner.
`NinfErrorCode NinfCallAsync ( char * entry, NinfSessionId * pSessionId, ...);`	Invokes a remote function specified by *entry*, in a non-blocking manner. Returns the id of the session in ** pSessionId*.
`NinfErrorCode NinfWait(NinfSessionId id);`	Waits for the finish of the session specified by *id*.
`NinfErrorCode NinfWaitAll();`	Waits for all the sessions invoked in advance, to finish.
`NinfErrorCode NinfWaitAny(NinfSessionId * id);`	Waits for any sessions invoked in advance, to finish, and stores it to **id*.
`NinfErrorCode NinfWaitAnd(NinfSessionId * idList, int length);`	Waits for all sessions in the *idList* done. The length of the list is specified as *length*.
`NinfErrorCode NinfWaitOr(NinfSessionId * idList, int length, NinfSessionId * id);`	Waits for any one of the sessions in the *idList* done. The length of the list is specified as *length*. Stores the done session into **id*.
`NinfErrorCode NinfGetState( NinfSessionId id, NinfSessionStatus * status);`	Gets status of the specified session and store it to *status*.
`NinfErrorCode NinfCancel(NinfSessionId id);`	Cancels the execution of the specified session.

Table 1. The Ninf-C API functions.

```
Module pi;

Define pi_trial(
   IN  int      seed,
   IN  int      n,
   OUT double * ratio
)
"tries n random points"
Required "pi_trial.o"
{
  extern double
         pi_trial(int seed, int n);
  * ratio = pi_trial(seed, n);
}
```

Figure 3. An example of inteface description.

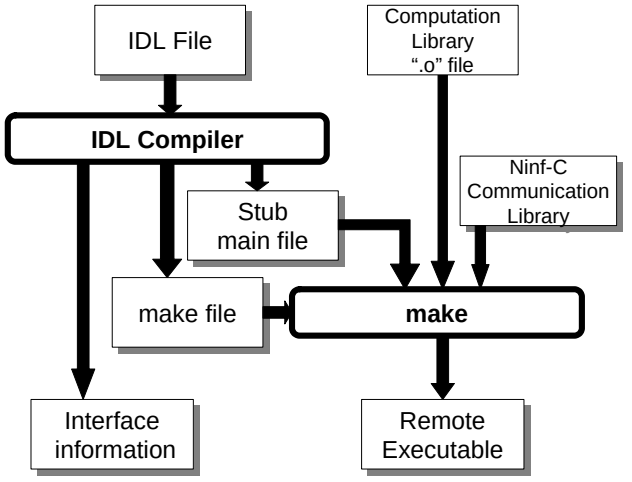

Figure 4. The IDL compiler.

Ninf-C provides an IDL compiler, called `ncgen` to process the IDL description. It reads the IDL description and generates interface information files, which contains binary encoded interface information of a function, stub main source code file, which provides wrapper code for the worker function, and a make file to help compilation and linkage of the remote executable. Figure 4 shows the diagram of IDL file processing.

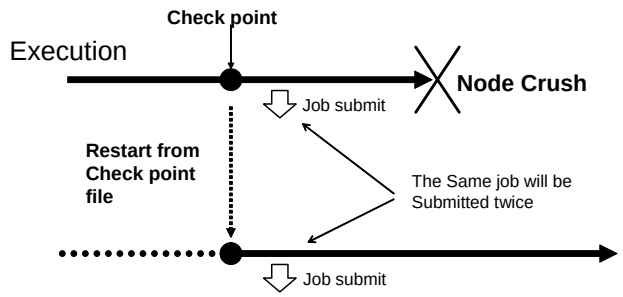

Figure 5. Checkpointing of the Master program.

3.6. Checkpointing of the master program

Condor provides checkpoint capability for remotely-executed program, that is the workers are automatically checkpointed, and may be migrated if needed. To make the whole system robust, not only the workers but also the master has to be checkpointed. Condor does provide the *scheduler universe*, that executes an executable on the submit machine under control of the Condor. The universe guarantees re-execution after the submit machine crash, but it does not provide checkpoint.

In addition, for the master process, simple checkpointing is not enough. Suppose a master process is checkpointed, and then submits a job, and the submit machine crashes. If we restart the master process from the checkpoint file, it will submit the same job once again(Figure 5). This will not affect the result of the computation, but consumes computational resource needlessly.

To solve this problem, Ninf-C implements two mechanisms: periodical checkpointing with the Condor checkpoint library and filename based double submit avoidance.

For the periodical checkpoint, master program is linked with the Condor checkpoint library and submitted to the scheduler universe, wrapped in a script. The script invokes the master program and periodically sends a signal that causes checkpoint. When the submit machine recovers from a crash, the script will be re-invoked by the Condor system. The script checks the existence of a checkpoint file, and if it exists, restarts the master program for the checkpoint file.

To implement this mechanism, we provide a command to submit master programs, called *ncrun*. The command takes the master program name and arguments for the program, and generates the wrapper script for the master program and submit file for the script, and submit it to the Condor scheduler universe.

Double submission avoidance mechanism is implemented as follows. The master program manages a sequential unique numbers for each RPC call and names the submission file according to the unique number. When the master invokes a new RPC, it tries to create a new submission file for the RPC. Before it actually creates the submission file, it checks the existence of the same named file. If it exists, this means that the submission is already done by a previous incarnation of the master program, before the crash. In this case, the master program stops the submission and just waits for the result.

```
Executable   = exec.$$(OpSys).$$(Arch)

Requirements = \
((Arch=="INTEL" && OpSys=="LINUX") || \
 (Arch=="SUN4c" && OpSys=="SOLARIS29"))
```

Figure 6. A submit file to utilize several architectures.

3.7. Utilize heterogeneous PC clusters

Ninf-C encodes files used for communication between the master and the workers, using Sun's XDR. This implies that the machine architecture for the master and the workers can be different.

Condor provides a mechanism to submit a job on a Condor-pool that consists of heterogeneous computers. Figure 6 shows the fragment of a submit file to submit a job on a pool with x86 Linux machines and SPARC Solaris machines. The condor system automatically selects the most suitable machines for the job and submits the job with appropriate executable. In this case, the executables have to be named as `exec.LINUX.INTEL` and `exec.SOLARIS29.SUN4c`.

Ninf-C can utilize heterogeneous Condor pool using this mechanism. The makefile generated by the Ninf-C IDL compiler generates a remote executable with a name that includes compiling system architecture and operating system, taken with the `condor_status` command. The remote executable is named as `_stub_(ModuleName)_(EntryName).(OSName).(ArchitectureName)`.

When an invocation occurs, the master program automatically scans its current working directory and gathers executable files that have names start with `_stub_(ModuleName)_(EntryName)`, and gathers possible executable architectures and operating systems from the filenames. Then it generates a submit file with a `Requirements` field for the architectures and operating systems.

3.8. Throttling

The master program submits one job, to the Condor, for each RPC invocation. The Condor system copies and stores the executable files to be submitted, for each job submission. This means that, if the master program invokes thousands of RPC calls, (a typical usage we are assuming), thousands of Condor jobs will be submitted, and thousands of copies of the executable file will be stored on the submit machine and this stressing the storage capacity. This is a undesirable because the executable linked with the Condor checkpoint library is always statically linked and, as a result, tends to be large.

To avoid this situation, Ninf-C has a throttling feature. In the configuration file, users can specify the maximum number of jobs that can be submitted simultaneously. During the execution, when the master wants to submit more than the specified value, it will block and wait for some of the submitted jobs to be done, and then resumes.

3.9. Steps to execute a Master-worker Program

Steps to execute a master-worker program using Ninf-C are as follows:

1. Write a master program using the API functions, compile and link with the Ninf-C communication library and the Condor checkpoint library. Ninf-C provides a compile driver called `nccc` to ease this step.

2. Generate a remote executable for the workers. Firstly, define the interface of the worker function with Ninf-IDL, and compile it with the IDL compiler called `ncgen`. The compiler will generate a makefile, interface files and stub main source code. Then, compile and link the stub main code by executing make command with the generated makefile.

3. Write a configuration file for the master, which specifies interface information file path and the maximum number of submitting jobs for throttling.

4. Launch the master program with the `ncrun` command, with arguments that specifies the configuration file. The `ncrun` will create a temporary sub directory and generate a wrapping script file and a submit file for the master program in it, and submit the submit file into the Condor pool. The condor system invokes the wrapping script, and the wrapping script invokes the master program. The master program submits jobs for the workers.

OS	CPU	#PE	#Nodes
LINUX	Pentium III 1.4GHz	2	5
Solaris 2.9	SPARC 450MHz	1	2
Solaris 2.9	SPARC 450MHz	2	2
Solaris 2.9	SPARC 450MHz	4	3

Table 2. The Condor pool used for the Experiment.

4. Evaluation

For evaluation, we conducted two expereiments; one is to confirm the robustness of the system, and another is to confirm its scalability.

4.1. Robustness test

To confirm robustness, we conducted an experiment on a Condor pool with heterogeneous resources. We executed a long-running master-worker application on the pool and injected artifitial faults by rebooting some of the machines in the pool, including the submit machine and the central manager.

Table 2 shows the Condor pool, which consists of X86 LINUX PCs and SPARC Solaris workstations. The central manager is running on one of the LINUX PCs and we used a SPARC machine with 2PEs as the submission machine.

Note that these computers are not dedicated for the experiment. These computers are publicly shared via the Globus GRAM, and users outside actually submitted jobs on the computers.

4.1.1. Sample Program We used a master-worker program that calculates PI using the Monte-Carlo method. This program randomly generates a large number of points in a square and counts the number of points that are located inside the inscribed circle of the square. PI can be derived from the ratio of the number of the interim points versus the number of all the points.

Figure 7 shows the core portion of the master program. Note that error-handling code is intentionaly omitted here for brevity The program calls worker n times in a non-blocking fashion, using `NinfCallAsync`, and waits for all jobs to finish by calling `NinfWaitAll`.

In the experiment setting we performed, the worker generates and tests 2×10^{10} points per one job, and the master submits 200 jobs for the worker. In total 4×10^{12} points was generated and tested. To finish the job, a worker takes approximately 50 min on a LINUX PC, and approximately 3 hours on a Solaris WS in the pool.

```
double ratios[n];
int    ids[n];

/* invoke rpc one by one */
for (int i = 0; i < n; i++){
  NinfCallAsync("pi/pi_trial",
                &(ids[i]),
                i, m,
                &(ratios[i]));
}

/* wait for all the rpc done */
NinfWaitAll();

/* sum up */
double ratioSum = 0.0;
for (int i = 0; i < n; i++);
  ratioSum += ratios[i];

/* print the result */
printf("pi = %f\n",
       (ratioSum / n) * 4.0);
```

Figure 7. A fragment of the Master Program.

4.1.2. Artificial Faults During the execution of the program shown above, we injected artifitial faults and observed the behavior of the system.

1. 210 min. later, rebooted one of the 4PE Solaris.
2. 350 min. later, rebooted one of the 4PE Solaris.
3. 540 min. later, killed Condor processes on the central manager
4. 600 min. later, rebooted one of the 4PE Solaris.
5. 660 min. later, rebooted the submission machine.
6. 740 min. later, killed Condor processes on a Linux node

4.1.3. Experiment Result The application successfully finished its execution. Figure 8 shows the number of the running jobs during the execution. The X-axis shows the elapsed time in minutes. The Y-axis shows the number of running jobs. We can say that, the number of running jobs drastically decreases after the injected faults, and then rapidly recovers. Note that, rebooting the submit machine caused a major damage on the system, but it was successfully overcome by the method proposed in this paper.

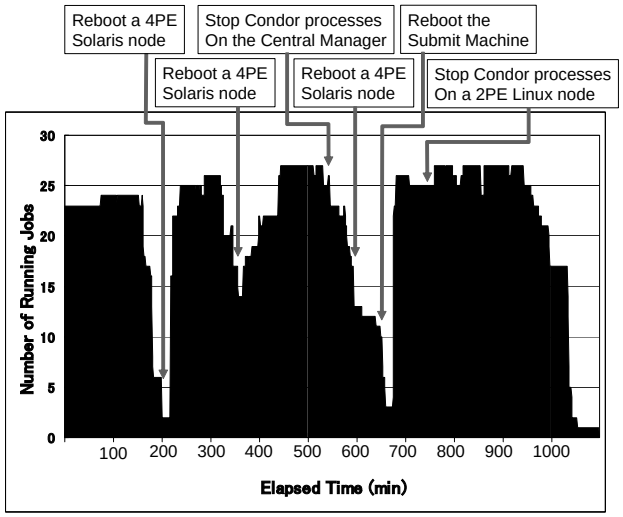

Figure 8. No. of working Job.

4.2. Scalability test

To confirm the scalability of the system, we conducted an experiment using a large-scaled Condor pool and a long runnning job.

4.2.1. Experimental platform As the platform for the experiment, we employed a large-scaled Condor pool deployed at *Titech Grid*, a production large cluster of clusters installed across campuses of the Tokyo Institute of Technology. The Titech Grid consists of 12 small to medium sized clusters with dual Pentium III 1.4 GHz, and has more than 800 processors in total. The condor pool on the Titech Grid has 734 processors at the time the experiment was conducted. To avoid invasion for jobs from other users, we restricted the maximum number of processors to 200.

4.2.2. Experimental setting As an application program, we employed a Molecular Dynamics simulation program; *Sander* from *Amber6*. Originally the Sander obtains configuration information from a file. We changed it slightly so that it takes configuration information from function arguments. Using Ninf-C, users can control configuration from master programs.

We conduted a thousand molecular dynamics simulations on a molecule with different initial velocity for each atom. The initial velocity is automatically generated randomly in the Sander, using a seed specified by the master program. As the target molecule, we employed small protein called Trp-Cage, that have 20 residue. The simulation time is 360 micro seconds. As a potential parameter, we employed parm99[16].

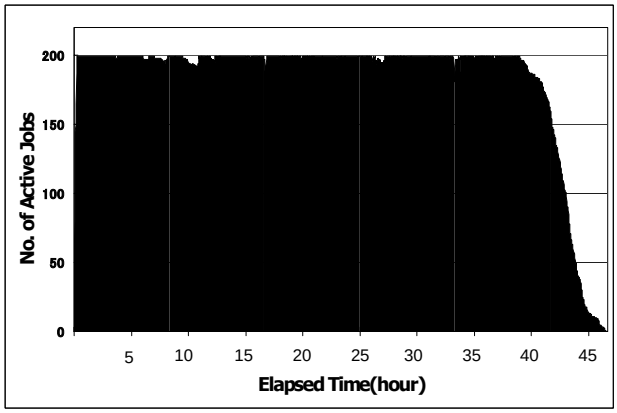

Figure 9. No. of working Job for MD simulation.

Each simulation takes around 7 hours on a Pentium III 1.4GHz node. Please note that the execution environment is shared by several users who may throw in jobs without using Condor.

4.2.3. Evaluation Result Figure 9 shows the number of working Condor jobs during the execution. We can observe that, we could almost always utilize the 200 processors; the upper limit of the setting. The total execution takes 2 days long, showing the stable operation of Ninf-C.

5. Related Work

5.1. Condor DAGMan

The Condor DAGMan (Directed Acyclic Graph manager)[2] is a job-flow scheduler that uses Condor as the job execution system. It takes a job-flow file, called the DAG file, which describes dependency graph among jobs, i.e. Condor submit files.

DAGMan itself runs as a scheduler universe job on the submit machine, as is with the Ninf-C master program. It reads a DAG file and determines jobs to be submitted next, and submits it, and monitors the job status using the log file. DAGMan does not have any state except for job-execution status, and job-execution status can be easily re-constructed from the log file. It means that it does not need checkpointing for robustness.

DAGMan and Ninf-C have some commonality in the basic architecture, in that they run as Condor scheduler universe jobs, monitor jobs with the log file, and control jobs with commands Condor supplied. The difference is that, DAGMan cannot handle dynamically changing workflows. Ninf-C gives programmers freedom to dynamically change the behavior of the programs depending on the preceding calculation result, etc.

Another difference is that, DAGMan requires deep knowledge of Condor, since the programmers have to writes their Condor submit files and setup input files. Ninf-C eases the burden by automatically generating these files.

5.2. Condor MW

MW(Master Worker)[8] is a generic framework for master-worker style applications, based on the Condor system. It provides C++ base classes for the driver (master), the worker and the task. Programmers have to extend these classes and fill up the specified methods for each class. The driver process runs on a submit machine as a scheduler universe job. MW supports three communication methods: PVM, sockets, and files. The file-communication is implemented using remote system call capability and provides robustness for the system.

The data transfer API that MW provides is similar to PVM. The programmer has to explicitly marshal and unmarshal the data to be transfered. In Ninf-C, a programmer does not take care of these issues.

Both MW and Ninf-C target master-worker style-applications, though the target worker grain size is different. MW maps multiple worker tasks on a single Condor job, to minimize communication cost, focusing on fine-grained master communication. On the other hand, Ninf-C maps one worker task on one Condor job, targeting coarse grained applications.

5.3. Ninf-G

Ninf-G[14] is a GridRPC system, implemented on top of the Globus Toolkit.[6] It provides easy to use RPC interface for users, just like the Ninf-C. It supports GridRPC API[13] that is under standardization effort in the GridRPC WG of the Global Grid Forum.

Since Ninf-G aims to execute relatively fine-grained master-worker type applications on large-scale Grids, the focus is on reducing each RPC invocation cost and optimize the API and implementation for that purpose. On the other hand, fault tolerance is not regarded as first priority. Ninf-G can provide primitive fault-tolerance for the workers using re-invocation by the upper-layer middleware, although it cannot do anything for master program.

Ninf-G directly accesses resources on Grids managed by the Globus Toolkit, while Ninf-C itself does not have any facility to directly access Grid, though it can indirectly access via Condor-G.

6. Conclusion

We described the design and implementation of a fault tolerant RPC system, Ninf-C, which is designed for large-scale master-worker programs. It is implemented on Condor and provides users with a robust RPC framework for long-running applications. To prove robustness of Ninf-C, we performed an experiment on a cluster. We ran a long-running master-worker program on the cluster and rebooted several machines of the cluster to simulate some serious fault situations. Despite this, the program execution still finished normally, proving the robustness of Ninf-C. We also tested a realistic program that ran stablely for two days on a Titech Grid.

For the future work, we will address following issue:

- Evaluation using more real-world applications

  The programs we used here have the simplest master-worker structure. To evaluate the system properly, we have to employ more large real-world applications. We are now planning to employ molecular dynamic method with replica-exchange.

- Evaluation on Grids with Condor-G

  As mentioned in 5.3, Ninf-C can utilize Grid resources managed by Globus GRAM, via Condor-G. We will perform experiments using large-scale applications on wide-area testbeds such as ApGrid[4], a Grid testbed in the Asia Pasific region, to confirm the scalability and the affinity with the Grid of Ninf-C.

Acknowledgement

We thank Jaime Frey from Condor Team, University Wisconsin, for his experties and many advices. We also thank Motonori Ohta, GSIC, Tokyo Institute of Technology, who helped us to use Amber for evaluation.

References

[1] Condor. http://www.cs.wisc.edu/condor/.

[2] Dagman. http://www.cs.wisc.edu/condor/dagman/.

[3] 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.

[4] ApGrid. http://www.apgrid.org/.

[5] R. Buyya, D. Abramson, and J. Giddy. Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid. In *Proceedings of HPC Asia 2000*, 2000.

[6] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. In *Proc. of Workshop on Environments and Tools, SIAM.*, 1996.

[7] K. Fujisawa, A. Takeda, M. Kojima, and K. Nakata. The sdpa (semidefinite programming algorithm) on the ninf (a network based information library for the global computing) (in japanese). In *The Institute of Statistical Mathematics Cooporative Research Report*, volume 135, pages 215–222, 2000.

[8] J.-P. Goux, S. Kulkarni, J. Linderoth, and M. Yorde. An enabling framework for master-worker applications on the computational grid. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9)*, pages 43 – 50, Pittsuburgh, Pennsylvania, August 2000.

[9] M. Livny, J. Basney, R. Raman, and T. Tannenbaum. Mechanisms for high throughput computing. *SPEEDUP Journal*, 11(1), June 1997.

[10] H. Nakada, M. Sato, and S. Sekiguchi. Design and implementations of ninf: towards a global computing infrastructure. In *Future Generation Computing Systems, Metacomputing Issue*, volume 15, pages 649–658, 1999.

[11] H. Nakada, Y. Tanaka, S. Matsuoka, and S. Sekiguchi. *Grid Computing: Making the Global Infrastructure a Reality*, chapter Ninf-G: a GridRPC system on the Globus toolkit, pages 625–638. John Wiley & Sons Ltd, March 2003.

[12] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proc. of HPDC-7*, 1998.

[13] 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.

[14] 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.

[15] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.

[16] J. Wang, P. Cieplak, and P. A. Kollman. How well does a restrained electrostatic potential (resp) model perform in calculating conformational energies of organic and biological molecules? *Journal of Computational Chemistry*, 21(12):1049–1074, 1999.