Utilizing the Metaserver Architecture in the Ninf Global Computing System

Hidemoto Nakada[1], Hiromitsu Takagi[2], Satoshi Matsuoka[3], Umpei Nagashima[4], Mitsuhisa Sato[5] and Satoshi Sekiguchi[1]

[1] Electrotechnical Laboratory, Tsukuba, Ibaraki, 3058568, Japan
[2] Nagoya Institute of Technology, Nagoya, Aichi, 4668555, Japan
[3] Tokyo Institute of Technology, Meguro-ku, Tokyo, 1528551, Japan
[4] Ochanomizu University, Bunkyo-ku, Tokyo, 1128610, Japan
[5] Real World Computing Partnership, Tsukuba, Ibaraki, 3050032, Japan

Abstract. Rapid increase in speed and availability of global-network is opening up the possibilities of globally-distributed supercomputing, including our *Ninf* system. Ninf is designed to utilize existing numerical libraries and resources on the Web. It also have global scheduling server called the Ninf *Metaserver*, which is responsible for scheduling and load-distribution of parallel tasks. The metaserver keep track of computational servers, and gathers informations needed for scheduling. Using the information, it balances loads over computational servers. We performed preliminary benchmarks using the metaserver. There, we observed that scheduling cost can be ignored, and dynamic scheduling by the server gained better score than static-cyclic distribution. We also report on our collaborative efforts in bridging Ninf with *NetSolve*, a similar system being developed at Univ. of Tennessee/ORNL.

1 Introduction

Rapid increase in speed and availability of supercomputers and mass storage servers interconnected with Gigabit-bandwidth networks is now making high-performance 'global computing' possible, in which computational and data resources in the network are collectively employed to solve large-scale problems. There have been several such recent proposals, including our **Ninf** (Network Infrastructure for global computing) [6][1] system, which makes available multiple remote computing and database servers on the network, allowing clients to semi-transparently call remote numerical applications and libraries from languages such as Fortran, C, and Java. By coupling with remote database servers and existing WWW frameworks, Ninf clients could flexibly access various data over the network, employ the data seamlessly within one's computation, and use the results either in a conventional way or with Web-based interfaces.

Ninf provides globally distributed parallel computing environment, on which several numerical computations are properly scheduled. Although wide area network bandwidth puts restrictions on communications among the parallel tasks, in

[6] http://ninf.etl.go.jp/

the scientific computation fields, there are numerous number of problems which require no communications among their tasks, such as parameter sensitive computations or Monte Calro simulations. There are some works concentrate only on these kind of calcuration, such as Nimrod[6], while Ninf gives more general framework. Our wide-area computation resource directory and scheduling server, called the *Metaserver*, keep track of computational servers, and gathers informations needed for scheduling, and blances loads over computational servers using the information.

One of the continuing technological challenges in Ninf is applicability of the Ninf system in a wide-area network setting. Applicability can be considered from two aspects, applicability of one Ninf computational server, and job scheduling over several servers. From former aspect, in [2], we have conducted extensive benchmarks in situations under which the computational performance overwhelms the network performance, thereby verifying usefulness of the Ninf system under certain situations, and furthermore discussed the design principles of remote network libraries as well as robustness of the Ninf system itself.

Here, from the latter aspect, we report on a preliminary experiment of job scheduling in the Ninf system. We experimented with automatic and dynamic load/task distribution using the metaserver. In the experiment, we verified the advantage of automatic load distribution over static, cyclic load balancing.

In addition, there are now collaborative efforts in bridging Ninf with *NetSolve*[3], a similar system being developed at Univ. of Tennessee/ORNL, using our adapter. Here, we report on a preliminary benchmarks of remote executions using the Ninf client calling the NetSolve server through the adapter.

2 Overview of the Ninf System

The Ninf system consists of *Ninf computational and database servers*, *Ninf client API* which defines the client bindings, and *Ninf metaservers* which perform wide-area detection, allocation, and scheduling of resources over the network. The clients and various servers communicate via Ninf RPC (Remote Procedure Call)[1], which is tailored for the needs of high-performance numerical computing.

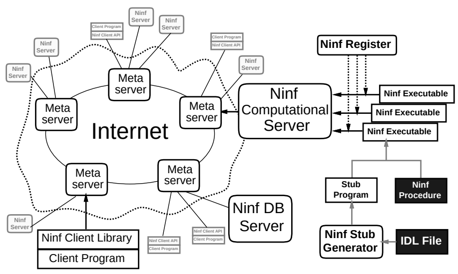

Fig. 1. Ninf Computational Clients and Servers

2.1 Ninf Client API

Ninf Client API is defined for Fortran, C/C++, and Java, and is also being defined for other languages such as Lisp. Ninf_call is a representative API used for invoking a named remote library on the server via Ninf RPC—e.g., to call a matrix multiply library remotely, one writes the following as if it were a local library:

```
double A[n][n], B[n][n], C[n][n];
Ninf_call("dmmul", n, A, B, C);
// or, Ninf_call("ninf://.../dmmul",...
```

Argument numbers, types, access modes (input/output), array size, as well as matrix size, region of usage, stride, are described by the Ninf IDL for each Ninf executable, compiled into a interpretive stub code, and registered with the Ninf server. (Note that the client programmer himself never sees or manipulates the IDL information itself.)

There are also APIs for asynchronous RPCs (Ninf_call_async), specification of transactions for parallel execution (Ninf_transaction_begin, Ninf_transaction_end), and queries of numerical database (Ninf_query, etc.).

Sub API for API Implementation: CIM (Common Interface Module) It is not trivial job to implement Ninf Client API. In order to make implementation of Ninf API easy, we provide sub-API called Ninf CIM(Common Interface Module) in C. It requires API implementers to provide some basic routines, called *traversers*, only. The routines traverse on the language native data structure and set/get data at that position. Ninf_cim_main(), CIM version of Ninf_call(), marshals / unmarshals data calling these traversers. Implementers have to know only about the data representation in the target language. Currently, an API for Lisp is implemented using CIM. We also implemented APIs for Excel and Mathematica.

2.2 Ninf Computational Server

The Ninf computational server is a process which services remote computing requests of remote clients by managing the communication and activation of the services requested via Ninf RPC. Binaries of computing libraries and applications are registered on the server process as *Ninf executables*, which can be semi-automatically generated with IDL descriptions using the Ninf stub generator.

Direct access to the Web On the Web, various useful data are published. For example, Matrix Market and our Matrix Workshop [7] provide matrices for testing various mathematical routines. Direct use of these published data allows easy evaluation of numerical routines.

[7] http://math.nist.gov/MatrixMarket/, http://ninf.etl.go.jp/MatrixWorkshop/

Ninf computational server provides direct Web access facility, by which users can specify data on Web as arguments for mathematical libraries registered with Ninf. Stub library which is linked with numerical routine and form the executable file on the server, has http client facility, enables application user to use any data on the Web directly.

The API is almost identical to the ordinal Ninf_call. Just using URL string as the data, users can use the data that can be located with the given URL. The following code fragment shows the usage of our direct web access facility. This code uses data on the Web as the second argument matrix.

```
Ninf_call("mmul",N, A, "http://.../..", C)
```

The Ninf client library automatically detect that the argument specifies an URL, and send the URL to the server. On the server side, the stub library receives the URL and connects to the Web server, and get the specified data by issuing the HTTP GET command, and store it in memory. Then, the stub library calls target numerical routine.

In addition to the GET command, Ninf client library can also use the HTTP PUT command to upload computed data to the Web server. [8] Ninf executable can upload data directly to Web servers which implement the GET command. This facility can be used, for example, to snapshot the intermediate results of the server computation on a Web server near Ninf computational server. We have a prototype GUI based matrices calculator which utilize this facility, called NinfCalc+.

2.3 Ninf Metaserver

The Ninf metaserver monitors multiple Ninf computing servers on the network, and performs scheduling and load balancing of client requests. The client need not be aware (but could specify) the physical location of computing servers. Additionally, metaserver controls the parallel, fault-tolerant execution of multiple sequence of Ninf_calls. In the next section, we describe metaserver facility in detail.

3 Global parallel computing in Ninf

The Ninf infrastructure consist of sets of servers and metaservers distributed in a network. Metaservers exchange information with each others periodically. For example, if a new Ninf server is added into a metaserver, the location of the new server is propagated among the metaserver network. Also, when a metaserver does not have a particular information available (e.g., availability of a certain library), a query is propagated through the metaserver network and the result is gathered by the issuing metaserver, who in turn delegates the information or makes automatic decisions for the client.

[8] The PUT command is optional and public Web servers often do not implement it as a defaults, because of security.

The Ninf metaserver keeps track of the following information for each Ninf server: (1) the location (address and port number), (2) the list of functions registered in the server, (3) the distance to the server with respect to the bandwidth of communication, (4) computational ability of the machine (clock speed, Flop/s etc ...) (5) the status of the server including the load average.

The current metaserver is implemented in Java, in order to exploit its portability, ease of network programming, and multi-threading, and also given that speed is not an absolute factor.

3.1 Parallel programming in Ninf

We have two API for parallel program, asynchronous call and transaction. Asynchronous call, Ninf_call_async(), simply splits invocation of a task and receiving the results. It invoke a task and returns immediately without waiting for the results. The client may continue its local computation, and then, wait for result to come back with Ninf_call_wait function. The client can issues mulitple request using this API.

We have another programming interface for parallel programming, *Ninf parallel transaction*, which supports macro dataflow parallel execution. The block of code surrounded by Ninf_transaction_begin and Ninf_transaction_end are not executed immediately; rather, data-dependency graph of the Ninf_call arguments are dynamically created, and at the end of the code block, the metaserver schedules the computation to multiple computational servers accordingly. When all the computations are complete, the metaserver returns the result to the client.

As an example, consider the addition of four matrices A, B, C, and D, with the result in G:

```
Ninf_transaction_begin();
Ninf_call("madd", n, A, B, E);
Ninf_call("madd", n, C, D, F);
Ninf_call("madd", n, E, F, G);
Ninf_transaction_end();
```

As the addition of A, B and C, D could be executed in parallel, they are executed in parallel, while the addition of the resulting E, F are executed after their termination.

3.2 Dynamic load balancing by Metaserver

Load Balancing is extremely important in parallel processing. Especially, when processor load cannot be determined statically, dynamic load balancing techniques have to take place to accomplish effective speedups.

We have implemented a technique on our metaserver. If all servers already have enough workloads, the metaserver stops task invocation. Instead, it reads and stores all the data needed for the invocation, and waits for a server available. When a server become available, the metaserver invoke the task on the server. These process is completely hidden from client by the metaserver, therefore clients can invoke several tasks even when no server available.

4 Evaluation of Metaserver Load Distribution

Here, we show the results of load balancing using the Ninf metaserver. As a benchmarking platform for our load balancing experiments, we are employing the 33-processor DEC Alpha WS Cluster (333Mhz) interconnected by switched 100Base-T network. One node executes the metaserver, while other nodes act as computation servers.

4.1 MetaServer baseline cost measurement

Firstly, we'll show the baseline cost of the metaserver load balancing, using EP from NAS Parallel Benchmarks, which is a typical Monte Calro simulation. This program has good characteristic for load partitioning. The workload can be divided into identical sized parts, therefore load imbalance does not occur. We performed experiment by comparing the two cases: user specified direct load balancing and automatic load balancing by the metaserver. This allows us to observe the baseline cost of load balancing by the metaserver, comparing these results. We performed experiment on three levels of different computation size : sample, class A and classB. Each level generates random numbers 2^{24}, 2^{28} and 2^{30} times, respectively. We divide the generation workload into identical sizes for each processor. For 32 processors, the generation workload for one processor is 2^{19}, 2^{23} and 2^{25}, respectively.

Results Table 1 shows the results. Metaserver cost increases in proportionally to the number of processors, however, is independent from the level of the program. Therefore, we can see that the cost can be ignored in the case of ClassB, the largest size.

Table 1. Load Distribution of EP(sec).

	number of servers	1	2	4	8	16
Sample	Via metaserver	35.7	18.1	9.4	6.2	6.9
	Direct specification	35.3	17.7	8.8	4.4	3.3
ClassA	Via metaserver	565.6	283.4	141.6	71.1	39.2
	Direct specification	565.4	282.4	141.2	70.6	37.2
ClassB	Via metaserver	2261.0	1132.1	566.3	283.3	141.7
	Direct specification	2259.0	1129.5	564.8	282.4	141.6

4.2 Dynamic Load Balancing by Metaserver

Target problem In this experiment, we employed the density of states (dos) calculation of a large molecule[4]. Density states of a large molecules are computed without matrix diagonalization, but instead, the entire system is modeled as a spring-connected kinetic model, upon which periodical external forces are applied, and the degree of resonance of the system is regarded as the density of the state at that frequency. This problem is well-suited for parallelization, as trial of each frequency could be executed independently; however, it requires dynamic load balancing as the load of computation at each frequency varies considerably. This problem is classified as a parameter sensitive computation.

Figure 2 shows the distribution of load according to the frequency. As one can observe. there is great variance in load, and as a result, static decompositions such as block or cyclic distribution will not work. Furthermore, it is extremely difficult to statically determine the load. Thus, it is a very favorable candidate for testing whether dynamic load balancing by the metaserver will work effectively.

Results The test was performed with 256 frequencies. Table 2 shows the results. Here, decomposition indicates the number of client-side decomposition of the 256 frequencies, e.g., if the decomposition were 32, then 32 Ninf_calls are made, each computing the dos for 8 frequencies. We also compare the results with simple cyclic distribution, and dynamic load balancing on the client-side using callbacks. Load balancing by callback requires some special programming techniques. The client side routine which is callbacked from the server side, takes responsibility to distribute loads.

Table 2. Execution Time of Dos for Different Load Balancing Schemes (sec.)

# Computing Servers		1	2	4	8	16	32
	32	1402.6	709.5	365.0	197.5	116.6	102.5
Metaserver	64	1419.0	728.2	377.9	201.4	109.3	79.1
[Decompositions]	128	1449.2	762.6	402.2	215.0	123.7	68.6
	256	1503.7	852.0	451.7	255.2	156.1	92.9
Callback		1385.9	697.6	351.8	179.8	91.3	52.9
Cyclic Dist.		1380.4	731.7	387.1	233.8	165.3	94.4

Observing the result, we see that execution time when we have only one computation server node increases as we increase the number of decompositions. This is due to increased overhead of combining multiple Ninf_calls into one.

As we increase the number of server, execution times monotonically decrease. When we alter the number of decomposition, the optimal number depends on the number of servers—for 8 nodes, 32 decompositions, and for 16 nodes, 64 decompositions, and so on. Increased number of decompositions results in better load balancing, but may not be optimal due to increased overhead. For this particular experiment, 4 decompositions per processor proved to be the optimal point. Figure 3 shows execution speed of each case. The speeds are measured comparing with linear speedup.

In addition, the benefit of dynamic metaserver load balancing proved to be advantageous over static cyclic distribution. However, compared to explicit client-side distribution using callbacks, the overhead of load distribution is greater. We speculate that this occurs for the following reasons:

1. The number of Ninf_calls are equal to the number of nodes for callbacks, whereas the number is equal to the number of decompositions for metaserver.
2. For metaserver, all the arguments of the Ninf_call are buffered at the metaserver, and metaserver does not invoke Ninf_call to the computation server until all

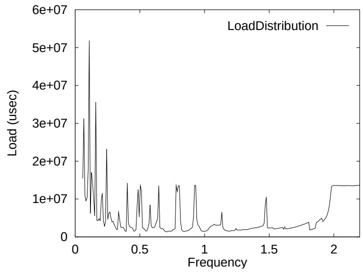

Fig. 2. Frequency Distribution of Load for DOS

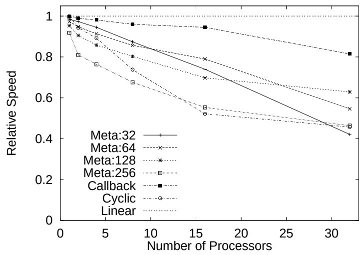

Fig. 3. Execution speed compared with the linear speedup.

the arguments arrive. This incurs not only buffering overhead, but also, we lose the opportunity of overlapping computation and communication.

3. Likewise, metaserver does not return to the client until all the Ninf_calls terminate.

Whereas the first overhead is fundamental to the approach, we expect that the second and the third overhead could be avoided with better implementation. Furthermore, despite the overhead, automatic load balancing with metaserver is much more ideal as it requires little special programming on the client side by the use of transactions.

5 Ninf-NetSolve Bridging

Several metacomputing frameworks are proposed, including our Ninf. Our metaserver have protocol forwarding and conversion facility, therefore, it can bridge different systems. There are now collaborative efforts in bridging Ninf with NetSolve as the first example of bridging such systems.

NetSolve is a global computing system being developed at Univ. of Tennessee. The functionality and the architecture of NetSolve is similar to Ninf: however, there is a great deal of difference in the protocols and some of the system APIs. By interfacing Ninf and NetSolve, we expect that not only it will benefit the users of both systems due to mutual interoperability, but also we expect that by comparing the two systems and their protocols, we could distill the advantageous characteristics of each, and design new systems and standard protocols. In fact, we are already in the process of proposing a new protocol based on the experiences gained from the adapters. Here, we describe how the adapters work, and their performance.

Figure 4 shows the architecture of NetSolve. NetSolve has a subsystem called an *Agent*, which manages the various information of its computational servers. Although the roles of Agent are similar to Ninf metaservers, one of the major differences is that metaservers act as proxy servers of client requests, whereas agents literally act as 'agents' whose role is to introduce the list of available computing servers according to the client requests.

Implementation of the Ninf → NetSolve Adapter For brevity, we describe the implementation of the Ninf → NetSolve adapter: the inverse adapter is similar in principle. As Figure 5 shows, the adapter acts as a computational server on the Ninf side, and acts as a client on the NetSolve side. The adapter is written in Java, actually, it is just a variant of our metaserver and still have full capability as a Ninf metaserver. We designed our metaserver based on delegation architecture, therefore, it can be implemented a protocol conversion facilities easily. Its roles are:

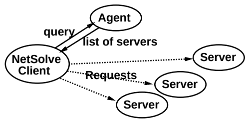

Fig. 4. Overview of NetSolve

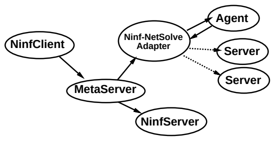

Fig. 5. Ninf → NetSolve Adapter

- *conversion of NetSolve interface information to Ninf interface information* — both systems make a two-phase call to the server, where the interface information of the callee library is sent to the client. We have discovered that Ninf and NetSolve have very different design philosophy in this respect, and mutual conversion is not trivial.
- *network data transfer format conversion* — The network data format differs considerably between Ninf and NetSolve. The size of data is implicitly obtained from the IDL in Ninf and is not part of the data, but in NetSolve there is a header indicating the data size. Thus, currently we must cache the network data entirely in the adapter for conversion, and this sacrificing performance.

5.1 Preliminary Evaluation

We performed some preliminary measurements to investigate the performance of the adapter. We employed the public NetSolve Agent located at comet.cs.utk.edu, and compared the calling timings of using the native NetSolve client and the Ninf client through the adapter. Although the exact computing server cannot be specified in NetSolve, it turned out that all the executions were being performed on dancer.cs.utk.edu, allowing us to perform a fair comparison. For the server-side library, we used degsv which is a part of LAPACK, and almost equivalent to the Linpack benchmark. The size of the array were set to be 50,100,150, and 200.

Table 3. Ninf-NetSolve Performance(sec).

Execution Time (sec.)	50	100	150	200
NetSolve Native	8.2	18.4	33.3	72.1
Ninf → NetSolve	8.5	21.7	38.8	82.4
Overhead	0.3	3.3	5.5	10.3

As we observe in Table 3, the overhead associated with the adapter increases as array size grows. This is largely due to buffering the argument arrays. We are now under negotiation with NetSolve people to standardize a data transfer protocol which can be used by not only our systems but also other metacomputing systems.

6 Conclusion and Future Work

We introduced some of the new feature of Ninf and also performed preliminary benchmarks for job scheduling in global computing systems. The features allows extensive use of data resource on the Web and easy implementation of the Ninf API. The key observation of the latter is that, with automatic and dynamic load/task distribution via Metaserver, using the dos calculation as the experimental target, and have shown that automatic load-balancing by the metaserver could be effective over static load balancing.

We have also covered our collaborative efforts of bridging Ninf with NetSolve, and have found that, despite apparent similarity in the functionalities, underneath Ninf and NetSolve have very different designs. We have also shown that adapters could be effectively designed to bridge the gap between the differences, with a small overhead.

As a future work, we are planning to improve the Metaserver scheduling, utilizing not only the load informations of servers but also network status, which will be taken from independent observing system such as NWS[5]. Another important work would be to standardize on a protocol for global computing. Currently, there are no standardized protocols for transferring numerical data of various formats over the Internet, nor there are standards for global computing. By agreeing upon a standard, and also by incorporating current standards such as CORBA and HTTP, we could effectively see widespread use of global computing systems with competing implementations and features, just as it is for other network services such as WWW, ftp, etc.

References

1. Sato, M., Nakada, H., Sekiguchi, S., Matsuoka, S., Nagashima, U. and Takagi, H.: Ninf: A Network based Information Library for a Global World-Wide Computing Infrastracture, *Proc. of HPCN'97 (LNCS-1225)*, pp. 491–502 (1997).
2. Takefusa, A., Matsuoka, S., Ogawa, H., Nakada, H., Takagi, H., Sato, M., Sekiguchi, S. and Nagashima., U.: Multi-client LAN/WAN Performance Analysis of Ninf: a High-Performance Global Computing System, *Proc. of Super Computing '97* (1997).
3. Casanova, H. and Dongarra, J.: NetSolve: A Network Server for Solving Computational Science Problems, *Proc. of Super Computing '96* (1996).
4. Nagashima, U., Hyugaji, S., Sekiguchi, S., Sato, M. and Hosoya, H.: An experience with super-linear speedup achieved by parallel computing on a workstation cluster: Parallel calculation of density of states of large scale cyclic polyacenes, *Parallel Computing*, Vol. 21, pp. 1491–1504 (1995).
5. R. Wolski, N. Spring and C. Peterson: Implementing a Performance Forecasting System for Metacomputing: The Network Weather Service, *Proc. of Super Computing '97* (1997).
6. D. Abramson, I. Foster, J. Giddy, A. Lewis, R. Sosic, R. Sutherst and N. White: The Nimrod Computational Workbench: A Case Study in Desktop Metacomputing, *Proc. of Australian Computer Science Conference* (1997).