Preliminary Evaluation of Scheduling in Ninf: a Global Computing System

Satoshi Matsuoka
Tokyo Institute of Technology

Hirotaka Ogawa
University of Tokyo

Atsuko Takefusa
Ochanomizu University

Hidemoto Nakada
Electrotechnical Laboratory

Kento Aida
Tokyo Institute of Technology

Abstract

Rapid increase in speed and availability of global-network is opening up the possibilities of globally-distributed supercomputing, including our *Ninf* system. However, performance characteristics of these systems have been little investigated, especially under multi-clients, multi-sites situations. In order to establish methodology to schedule multiple job requests to multiple computational servers effectively and guarantee performance per each client, we conducted benchmarks under various WAN environments. There, we observed that communication bandwidth dominated performance for communication-intensive applications such as Linpack, and aggregate bandwidth could be sustained for multi-clients located at different internet sites. Based on these observations, we propose the need for a simulation model based on queuing theory. And we also performed preliminary benchmarks using our scheduling server, called the Ninf *Metaserver*.

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) **Ninf**[1][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.

One of the continuing technological challenges in Ninf is applicability of the Ninf computational server in a wide-area network setting. 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. On the other hand, we still lack a clear model of computational resource distribution under environments where multiple computational resources are being shared by multiple resources; in other words, how we schedule the resources so as to satisfy the performance criteria of each client, is not yet clear.

We report several preliminary experiments of job scheduling in the Ninf system:

- Under WAN, we performed benchmarks where clients from a single-site and multiple-sites requested computation with different communication patterns. The observed performance from the client depends on server performance for computation-dominant tasks (e.g., EP in NAS Parallel Benchmarks[3]); on the other hand, for communication dominant benchmarks such as Linpack, different topologies of server and clients within the network could have adverse, often degrading effects on the observed server performance.
- Based on the above observation, we are currently constructing a simulation model and a simulator based on the queuing network model.
- We also experimented with automatic and dynamic load/task distribution using our prototype, wide-area computation resource directory and scheduling server, called 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*[4], a similar system being developed at Univ. of Tennessee/ORNL, using Ninf–NetSolve *adaptor* servers. Here, we report on a preliminary benchmarks of remote executions using the Ninf client calling the NetSolve server through the adaptor.

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,

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

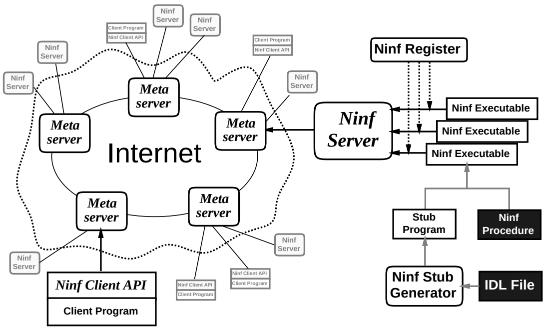

Figure 1: Ninf Computational Clients and Servers

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. There are also various supportive tools such as server registry tools, stub generators, query proxies, GUI builders, etc[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.

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("http://.../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.) Ninf_call is then made in two stages: when the client calls the server, the server returns the compiled IDL stub code to the client. Ninf_call then interprets the IDL code and marshalls the arguments, and makes the actual code and obtains the result from the server. 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.).

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. 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.

[2] Henceforth, we will concentrate on describing the computational aspects of Ninf.

3 Computational Server Evaluation under Multiple WAN Clients

When Ninf computational servers are deployed, each will simultaneously service Ninf_call requests from multiple clients distributed in the WAN. However, we still lack a clear model as to how to schedule the computational servers so as to satisfy the performance criteria of each client.

In order to establish such a model, we have conducted a performance benchmark where multiple clients from a single site as well as from multiple sites will make requests with different communication patterns to the computational server. The benchmarks employed were Linpack and EP from NAS Parallel Benchmarks. Linpack tends to manifest communication overheads as it is relatively communication intensive, whereas communication overhead is small and constant whereas computation is dominant.

3.1 Evaluation Environment

As a model client program, we employ a client which calls either the Linpack (sgetrf and sgetrs) or the EP remote library routines repeatedly. We assume that each client performs a Ninf_call on the interval of s seconds with probability p. We also assume that the number of clients and matrix size are fixed at c and n during the benchmark, respectively. We set the other parameters to be $s = 3$, $p = 1/2$ and $c = 1, 2, 4, 8, 16$. In the benchmark, we measured the server utilization, load average, and communication throughput.

3.2 Evaluation with the Linpack Benchmark

Double precision Linpack benchmark performs Gaussian elimination to solve a system of linear equations. Overall, it has the computational complexity of $O(\frac{2}{3}n^3 + 2n^2)$ and communication complexity of $8n^2 + 20n + O(1)$ bytes for a n by n matrix. For Ninf, we execute the LU-decomposition(dgefa) and backward substitution (dgesl) remotely (Figure 2). Thus, given the overall execution time of Ninf_call T_{Ninf_call} consists of communication time T_{comm} plus computing time T_{comp}, each of which has the above complexity, and Ninf_call performance P_{Ninf_call} can be given as $P_{Ninf\_call} = \frac{2/3n^3+2n^2}{T_{Ninf\_call}}$. The problem (matrix) sizes were set to $n = 600, 1000, 1400$.

3.2.1 Single-Site WAN Benchmark Results

For single-site, multi-client benchmarks, we employed the Cray J90 owned by the Electro-Technical Laboratory (ETL) of Tsukuba Japan (200Mflops×4, 512MB), and for clients, we used a cluster of 8 SuperSPARC(50MHz×2, 96MB) workstations at Ochanomizu University of Tokyo, Japan. The ftp throughput between the two sites was 0.17MB/s.

Table 1 shows the result. We observe that, Ninf_call performance improves even under WAN as n increases, due to comparatively smaller computation size. On the other hand,

n	c	Performance[Mflops] max/min/mean	Throughput[MB/s] max/min/mean	Server Utilization [%]	Server Load Av.
600	1	8.00/7.24/7.68	0.166/0.153/0.161	6.85	0.40
	2	6.05/1.51/3.77	0.124/0.030/0.077	6.90	0.42
	4	3.57/0.92/2.46	0.073/0.019/0.051	8.94	0.50
	8	1.52/0.49/1.08	0.031/0.011/0.022	7.99	0.49
	16	0.90/0.29/0.54	0.018/0.006/0.011	8.17	0.49
1000	1	12.75/7.26/10.50	0.160/0.091/0.131	6.43	0.39
	2	8.22/5.89/7.15	0.101/0.072/0.088	9.23	0.52
	4	4.80/1.46/3.97	0.061/0.018/0.049	9.39	0.53
	8	2.42/0.97/1.82	0.029/0.012/0.022	8.71	0.53
	16	1.36/0.47/0.88	0.016/0.006/0.011	8.75	0.49
1400	1	18.46/11.87/16.42	0.166/0.104/0.147	8.15	0.47
	2	10.44/8.03/9.34	0.092/0.070/0.082	13.58	0.73
	4	7.00/2.00/5.50	0.061/0.017/0.048	15.61	0.81
	8	2.91/1.42/2.46	0.025/0.012/0.021	9.20	0.56
	16	1.59/0.79/1.25	0.014/0.007/0.011	9.35	0.54

Table 1: Single-Site WAN Benchmark Results

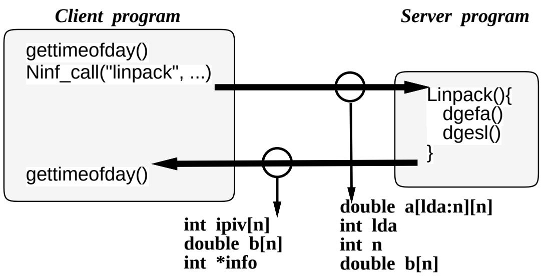

Figure 2: Ninf_call to Linpack routine

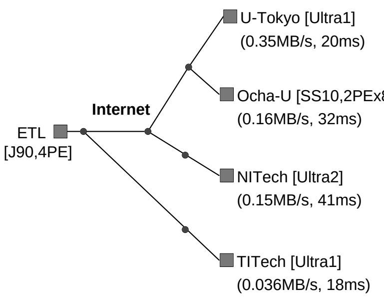

Figure 3: WAN Benchmarking Environment

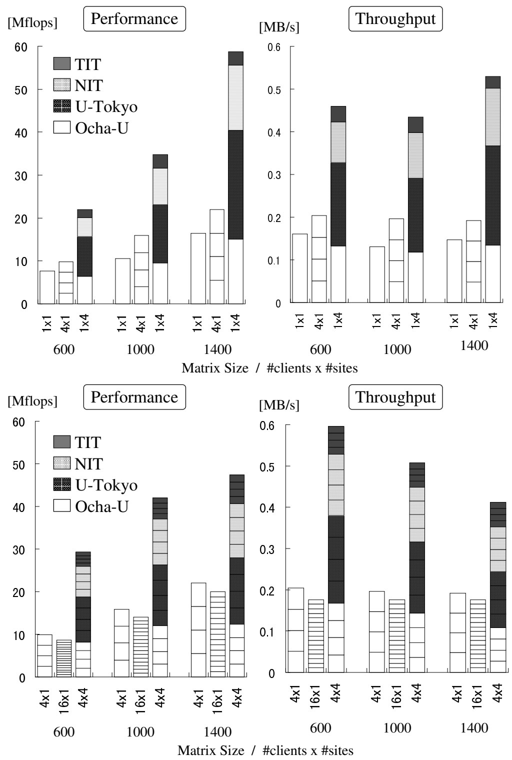

Figure 4: Multi-Site, Multi-Client WAN Benchmark Results

when c increases, deterioration of client-server communication throughput is significant. This is because the client-server bandwidth saturates with only 1 or 2 clients, and the number of Ninf_calls that arrive per time unit is suppressed; for example, on $c = 16$, both server utilization and load average have not saturated.

3.2.2 Multi-Site WAN Benchmark Results

For multi-site benchmarks, we employed clients from four geographically separate sites, namely Ochanomizu University (Ocha-U), University of Tokyo (U-Tokyo), Nagoya Institute of Technology (NITech), and Tokyo Institute of Technology (TITech), while the server is the ETL J90. Figure 3 shows the actual client machine employed at each site, ping latency and ftp throughput between the clients and the server, and the WAN topology. The number of clients at each site were set to $c = 1$ and 4. Figure 4 shows the results. As a comparison, we also show the benchmark results of the single-site benchmark between ETL and Ocha-U. The X-axis show the problem size and the number of clients ($c \times$ #sites, while the Y-axis shows the aggregate Ninf_call performance and communication throughput.

We can observe that, aggregate communication throughput from multiple sites are substantially higher than that from a single site. In fact, between Ocha-U and ETL, multi-site communication performance deteriorated only by 9% ∼ 18% when $c = 1$ (total 4 clients in WAN), and 18% ∼ 44% when $c = 4$ (total 16 clients in WAN), maintaining substantially higher bandwidth than when the same number of clients are located at Ocha-U. As a result, CPU utilization and load average is substantially greater for multi-site compared to single-site. Overall, although we observe some fluctuation in communication throughput, difference between client performances is small, and bound by point-to-point WAN throughput performance.

Also, although one might conjecture that the greater number of clients with larger problem size would easily saturate the computational power of J90, this is not necessarily the case as we observe that CPU utilization levels off even for $c = 4$ (total 16 procs, approx. 27% ∼34%), and similarly for load average. Because network load basically determines server performance for communication-dominant tasks, we can claim that server utilization and load average alone are not good criteria for Metaserver task distribution in WAN.

3.3 EP WAN/LAN Benchmark Results

EP(An embarrassingly parallel benchmark) is one of the kernel programs in the NAS Parallel Benchmark, performing (random-number) Monte-Carlo simulations[3]

For Ninf benchmarking, we execute the random number generation on the Ninf server. Communication complexity is O(1), namely, problem-size independent. Computational complexity is proportional to the number of random numbers generated, and becomes 2^{n+1} for 2^n trials. Therefore, The execution performance P_{Ninf_call} becomes $P_{Ninf\_call} = \frac{2^{n+1}}{T_{Ninf\_call}}$, where T_{Ninf_call} is the time taken for the entire Ninf_call . In each benchmark run, we execute 2^{24} trial samples for each PE, and execute them in a task-parallel manner on the 4-processor J90.

3.3.1 EP WAN/LAN Benchmark Results

Table 2 shows the EP benchmark performance for LAN/WAN multi-client execution. Because each Ninf_call is executed by 1PE of J90 Ninf_call performance is sustained up to $c = 4$ clients, where the server utilization achieves approximately 100%. On the other hand, for $c = 8, 16$ Ninf_call performance has degraded to 1/2 and 1/4, respectively. This is due to very low communication requirements of EP, and the Ninf_call performance mainly reflected server computing performance. Thus, for this class of applications such as parallel rendering/imaging and parameter sensitivity analysis, global computing is currently quite feasible; furthermore, we expect this result to hold for multi-server settings as well.

4 Modeling a Global Computing System as a Queuing System

The benchmarks of the previous section were performed under some given condition. However, since network bandwidth and quality will surely change both in short and long terms, benchmarks alone are not sufficient. in order to have generally automated job scheduling of computational resources in a wide-area network,

We are currently investigating how we could model global computing systems, thereby allowing simulation of overall system behavior. Our initial attempt is to attempt to model the global system with network queuing theory. Here, we describe a simple queuing model whereby Ninf clients and servers could be modeled effectively.

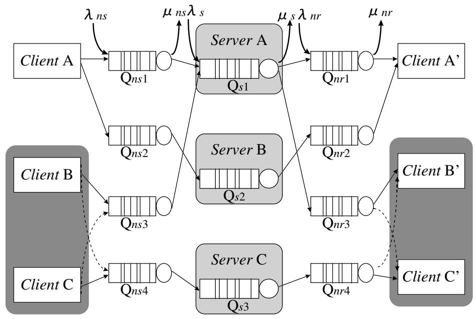

Figure 5: A Simulation Model of Global Computing System

4.1 Overview of the Simulation Model

Figure 5 gives the overview of our proposed simulation model of a typical RPC-based global computing system. In the figure, the clients A–A', B–B' and C–C' are the same clients acting as senders and receivers.

In the model, invocation and argument communication from the client to the server, processing at the server, and returning of the reply from the server to the client, are modeled as queues Q_{ns_i}, Q_{nr_i}, and Q_{s_i}, respectively.

λ_{ns}, λ_{nr} are arrival rates of requests in the network other than Ninf_call ; in other words, they model congestion and delays encountered in a wide-area network. Here, we assume they are both Poisson arrival, but other arrival rates could be employed. μ_{ns}, μ_{nr} are service rate of the queues, i.e., they effectively represent the network bandwidth, and they follow exponential distribution. λ_s, μ_s similarly represent degree of server congestion, and server processing power.

Also, as Figure 5 shows, we assume that Clients B and C reside on the same node. As the results from the previous section show, because the nodes residing in the same site share incoming/outgoing networks, each client can employ bandwidth that is inverse proportion to the number of clients in the node. Thus, a communication request from client B to server A is placed in the same queue as that of Client C to Server C, causing communication delays. The inverse holds true as well.

Currently, we are building a simulator based on the model in Java.

5 Evaluation of Metaserver Load Distribution

Load Balancing is extremely important in parallel processing. When processor load cannot be determined statically, dynamic load balancing techniques can be often used for effective speedups. We are experimenting with several load balancing techniques with Ninf metaserver; here, we report on some preliminary results.

5.1 Overview of Ninf Metaserver Load Balancing

Currently, clients specify parallel execution in Ninf in a coarse-grained manner using *transactions*. A user specifies a sequence of Ninf_calls surrounded by a *transaction block*. During execution, these sequences are checked for data dependency between the arguments, and independent

	c	Performance[Mops] max/min/mean	Comm. Time[sec] max/min/mean	Server Utilization [%]	Server Load Av.
LAN	1	0.17/0.17/0.17	0.01/0.01/0.01	30.51	1.44
	2	0.17/0.17/0.17	0.01/0.01/0.01	53.86	2.39
	4	0.17/0.16/0.17	0.01/0.01/0.01	98.18	4.18
	8	0.09/0.08/0.08	0.01/0.01/0.01	100.00	8.90
	16	0.04/0.04/0.04	0.02/0.01/0.01	100.00	16.88
WAN	1	0.17/0.17/0.17	0.06/0.04/0.04	25.02	1.21
	2	0.17/0.16/0.17	0.06/0.04/0.04	49.16	2.19
	4	0.17/0.16/0.17	0.24/0.04/0.05	98.14	4.16
	8	0.09/0.08/0.08	0.05/0.04/0.04	100.00	8.91
	16	0.04/0.04/0.04	0.09/0.04/0.04	99.94	16.88

Table 2: EP LAN/WAN Multi-Client Benchmark Results

Ninf_calls are executed in parallel on different computation severs.

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.

To a client, a transaction could be regarded as an atomic execution of aggregated Ninf_calls. At runtime, each client invocation of a Ninf_call within a transaction block will not result in a call, but will construct a portion of a data flow graph instead. When Ninf_transaction_end is reached, the constructed data flow graph is shipped to the metaserver. The metaserver in turn will schedule the Ninf_calls in the graph to appropriate computation servers. When all the computations are complete, the metaserver returns the result to the client.

5.2 Implementation of Dynamic Load Balancing

A Ninf metaserver manages multiple Ninf servers, including the number of currently executing tasks. The metaserver load balancer uses this information to suppress the number of tasks sent to the server. The tasks which are not immediately scheduled are queued within the metaserver; tasks are then subsequently scheduled as active tasks complete, depending on available libraries in the server, server load, network bandwidth, etc. The metaserver itself is implemented in Java.

5.3 Benchmarking Environment for Load Balancing

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

In the experiment, we employed the density of states (dos) calculation of a large molecule[5]. A large molecules density of states 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.

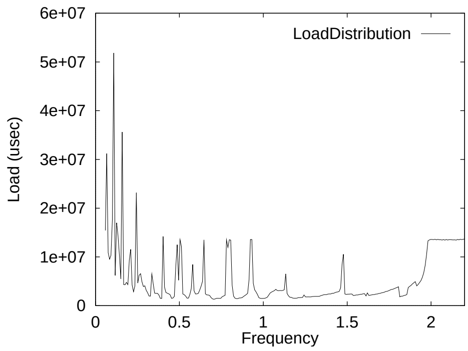

Figure 6: Frequency Distribution of Load for Dos

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.

Figure 6 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.

5.4 Results

The test was performed with 256 frequencies. Table 3 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[6].

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 ex-

# Computing Servers		1	8	16	32
Decompositions	32	1394.8	195.5	111.9	109.7
	64	1408.5	196.1	107.9	78.2
	128	1430.0	206.1	113.1	69.2
	256	1489.4	234.6	131.9	85.9
Single PE Execution		1339.0	—	—	—
Cyclic Dist.		1380.4	233.8	165.3	94.4
Callback		1385.9	179.8	91.3	52.9

Table 3: Execution Time of Dos for Different Load Balancing Schemes (sec.)

periment, 4 decompositions per processor proved to be the optimal point.

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 the arguments arrive. This incurs not only buffering overhead, but also, we lose the opportunity of pipeline parallelism.

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.

6 Ninf-NetSolve Adapter

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 adopters. Here, we describe how the adapters work, and their performance.

6.1 Overview of NetSolve

Figure 7 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

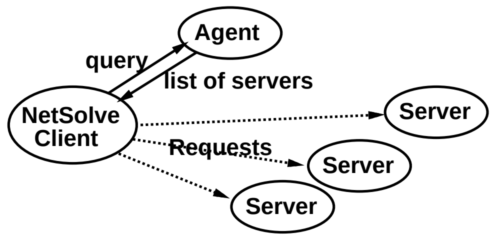

Figure 7: Overview of NetSolve

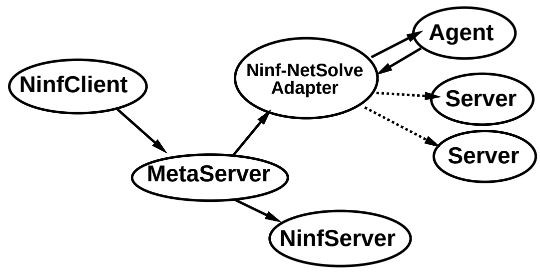

Figure 8: Overview of Ninf → NetSolve Adapter

to introduce the list of available computing servers according to the client requests. The client itself selects the server from the list; thus, there are no automated scheduling of computation as is with Ninf.

6.2 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 8 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, and has the following roles:

- *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. For instance, array size is determined to be dependent on some scalar argument of the same call, and whereas in Ninf this information is translated from IDL specification such that only the dependency relationship from the scalar argument to the matrix argument is returned to the client, in NetSolve a string which encodes the mutual relationship between the argument positions are returned.

- *network data transfer format conversion* — The network data format differs considerably between Ninf and NetSolve. For example, Ninf adopts the c-style array layout, whereas NetSolve assumes Fortran-style format. Also, 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.

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

Table 4: Ninf-NetSolve Performance

Thus, currently cache the network data entirely in the adapter for conversion. This is not favorable for performance.

6.3 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 times 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 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.

As we observe in Table 4, the overhead associated with the adapter increases as array size grows. This is largely due to buffering and re-ordering of the elements of the argument arrays. The re-ordering is fundamental to the difference between Ninf and NetSolve, but could be avoided by better implementation, or better yet, a standardized protocol which both systems could be optimized for.

7 Conclusion and Future Work

We have performed preliminary benchmarks for job scheduling in global computing systems. The key observation is that, for communication dominant benchmarks such as Linpack, different topologies of server and clients within the network could have adverse, often degrading effects on the observed server performance. Based on the observation, we have sketched the a simulation model based on the queuing network model. Furthermore, we also experimented 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 currently constructing a simulator based on the queuing model described here, using Java. We hope to be able to match the actual measurements of our benchmarks with the simulation results; if they match, then we could effectively use the simulator as a part of performance predictor in the metaserver, thereby allowing the metaserver to effectively schedule client tasks. On the other hand, if they do not match, then we will have to identify the factors which are not accounted for in the current model, and improve the model accordingly.

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.

Acknowledgment

We would like to thank other members of the Ninf group, especially Hiromitsu Takagi (NIT), Satoshi Sekiguchi (ETL), Mitsuhisa Sato (RWC), Umpei Nagashima (Ochanomizu Univ.) for their discussions and encouragements. We would also like to thank Osamu Tatebe (ETL) for his advice on parallel numerical computing.

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, *Proceedings 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, *To appear in Supercomputing '97*.

[3] NASA Ames Research Center: The NAS Parallel Benchmarks, http://science.nas.nasa.gov/Software/NPB/.

[4] Casanova, H. and Dongarra, J.: NetSolve: A Network Server for Solving Computational Science Problems, *Proceedings of Supercomputing '96* (1996).

[5] 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).

[6] Nakada, H., Takagi, H., Matsuoka, S., Nagashima, U., Sato, M. and Sekiguchi, S.: Global Parallel Computation using Ninf, *Proceedings of Joint Symposium on Parallel Processing '97*, pp. 281–288 (in Japanese) (1997).