OmniRPC: A Grid RPC Facility for Cluster and Global Computing in OpenMP
(Extended Abstract)

Mitsuhisa Sato[1], Motonari Hirano[2], Yoshio Tanaka[2], and Satoshi Sekiguchi[2]

[1] Real World Computing Partnership, Tsukuba, Japan
[2] Software Research Associates, Inc
[3] Electrotechnical Laboratory

Abstract. *Omni remote procedure call facility*, OmniRPC, is a thread-safe grid RPC facility for cluster and global computing environments. The remote libraries are implemented as executable programs in each remote computer, and OmniRPC automatically allocates remote library calls dynamically on appropriate remote computers to facilitate location transparency. We propose to use OpenMP as an easy-to-use and simple programming environment for the multi-threaded client of OmniRPC. We use the POSIX thread implementation of the Omni OpenMP compiler which allows multi-threaded execution of OpenMP programs by POSIX threads even in a single processor. Multiple outstanding requests of OmniRPC calls in OpenMP work-sharing construct are dispatched to different remote computers to exploit network-wide parallelism.

1 Introduction

In this paper, we propose a parallel programming model for cluster and global computing using OpenMP and a thread-safe remote procedure call facility, OmniRPC.

In recent years, two important computing platforms, a cluster of workstation/PC and a computational grid, have been gathering many interests in high performance network computing. Recent progress in microprocessors and interconnection networks motivates high performance computing using clusters out of commodity hardware. Advances in wide-area networking technology and infrastructure make it possible to construct large scale high-performance distributed computing environments, or computational grids that provide dependable, consistent and pervasive access to enormous computational resources.

Omni remote procedure call facility, OmniRPC, is a thread-safe implementation of Ninf RPC which is a grid RPC facility in a wide-area network. The remote libraries for OmniRPC are implemented as executable programs, and are registered in each remote computer. The OmniRPC programming interface is designed to be easy-to-use and familiar-looking for programmers of existing languages such as FORTRAN, C and C++, and is tailored for scientific computation. The user can call the remote libraries without any knowledge of the

R. Eigenmann and M.J. Voss (Eds.): WOMPAT 2001, LNCS 2104, pp. 130–136, 2001.

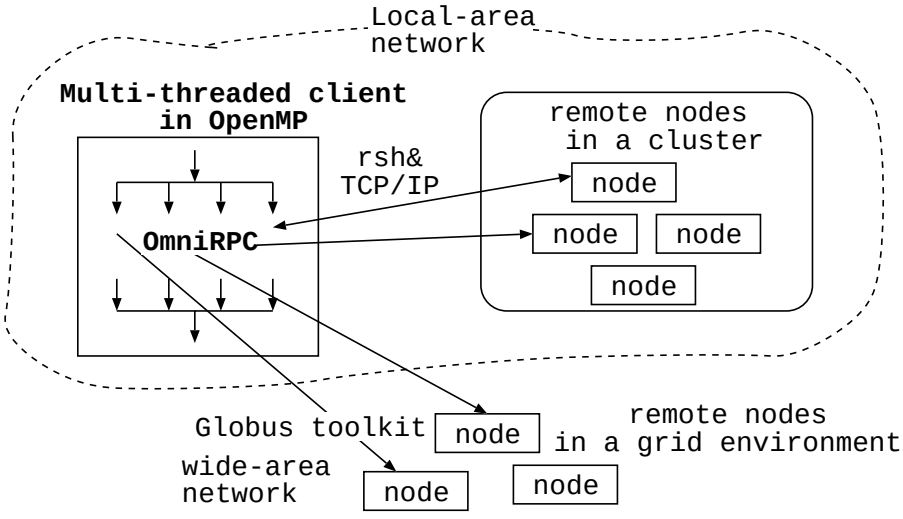

Fig. 1. OpenMP multi-threaded client and OmniRPCs

network programming, and easily convert his existing applications that already use popular numerical libraries such as LAPACK. A client can execute the time-consuming part of his program in multiple and heterogeneous remote computers, such as clusters and supercomputers, without any requirement for special hardware or operating systems. OmniRPC provides uniform access to a variety of remote computing resources.

At the beginning of execution, the initialization of OmniRPC collects the information about remote libraries registered in available remote computers. OmniRPC automatically allocates remote library calls dynamically on appropriate remote computers to facilitate location transparency. In order to support parallel programming, the multi-threaded client can issue multiple requests by OmniRPC simultaneously. Each outstanding request is dispatched to a different remote computer to exploit network-wide parallelism. Although the POSIX thread library can be used for programming multi-threaded clients, multi-threaded programming using thread library directly makes the client program complicated.

While the OpenMP Application Programming Interface (API) is proposed for parallel programming on shared-memory multiprocessors, OpenMP provides a multi-threaded programming model without the complexity of multi-threaded programming. We have developed Omni OpenMP compiler[4], which is a free and open-source, portable implementation of OpenMP. In a shared memory multiprocessor, threads in OpenMP are eventually bound to physical processors for efficient parallel execution. The POSIX thread implementation of the Omni OpenMP compiler allows multi-threaded execution of OpenMP programs by POSIX threads even in a single processor.

OpenMP provides an easy-to-use and simple programming environment for the multi-threaded client of OmniRPC. Figure 1 illustrates the OpenMP multi-threaded client and OmniRPCs. A typical application in cluster and grid environments is parametric execution which executes the same code with different input parameters. For this type of applications, we can use OpenMP parallel loop directives to execute OmniRPC calls in parallel for different remote computers.

For a computational grid environment, OmniRPC uses Globus toolkit[1] as a grid software infrastructure. Although a Globus implementation of MPICH, MPICH-G, can be used for parallel programming in Globus, message passing programming requires programmers to explicitly code the communication and makes writing parallel programs cumbersome. While our proposed model is limited to a master-slave model, it provides very simple parallel programming environment for a computational grid.

The parallel programming model with the OpenMP client of OmniRPC can be applied to other RPC facilities such as CORBA if these APIs are thread-safe. NetSolve[3] is a similar RPC facility to our OmniRPC and Ninf. It also provides a programming interface similar to ours and automatic load balancing mechanism by a agent. To our best knowledge, no experience of parallel programming with OpenMP is reported.

2 OmniRPC: A Thread-Safe Remote Procedure Call Facility

A client and the remote computational nodes which execute remote procedures may be connected via a local area network or over a wide-area network. A client and nodes may be heterogeneous: data in communication is translated into the common network data format.

The remote libraries are implemented as executable programs which contain network *stub* routine as its main routine, and registered in the *registry file* in each remote nodes. We call such executable programs *Ninf executables (programs)*. These stubs are generated from the interface descriptions by the Ninf IDL compiler.

In a client node, a user prepares his own *machine file* which contains the host names of available computation nodes. The OmniRPC initialization function, `OmniRPC_init`, reads registry files in the remote nodes to make the database which associates the entry names of remote functions with Ninf executables.

OmniRPC inherits its API and basic architecture from Ninf. `OmniRPC_Call()` is the sole client interface to call the remote library. In order to illustrate the programming interface with an example, let us consider a simple matrix multiply routine in C programs with the following interface:

```
double A[N][N],B[N][N],C[N][N];   /* declaration */
....
dmmul(A,B,C,N);  /* calls matrix multiply, C = A * B */
```

When the `dmmul` routine is available in a remote node, the client program can call the remote library using `OmniRPC_Call`, in the following manner:

```
OmniRPC_Call("dmmul",A,B,C,N); /* call remote library */
```

Here, `dmmul` is the entry name of library registered as a Ninf executable on a remote node, and `A`,`B`,`C`,`N` are the same arguments. As we see here, the client user only needs to specify the name of the function as if he were making a local function call; `OmniRPC_Call()` automatically determines the function arity and the type of each argument, appropriately marshals the arguments, makes the remote call to the remote node, obtains the results, places the results in the appropriate argument, and returns to the client. In this way, the OmniRPC is designed to give the users an illusion that arguments are shared between the client and the remote nodes.

To realize such simplicity in the client programming interface, a client remote function call obtains all the interface information regarding the called library function at runtime from the server. The interface information includes the number of parameters, these types and sizes and access mode of arguments (read/write). Using these informations, the RPC automatically performs argument marshaling, and generates the sequence of sending and receiving data from/to the nodes. This design is in contrast to traditional RPCs, where stub generation is done on the client side at compile time.

The interface to a remote function is described in Ninf IDL. For example, the interface description for the matrix multiply given above is:

```
Define dmmul(long mode_in int n, mode_in double A[n][n],
        mode_in double B[n][n], mode_out double C[n][n])
"... description ..."
Required "libxxx.o" /* specify library including this
                                                routine. */
Calls "C" dmmul(n,A,B,C);  /* Use C calling convention. */
```

where the *access specifiers* , `mode_in` and `mode_out`, specify whether the argument is read or written. To specify the size of each argument, the other `in_mode` arguments can be used to form a size expression. In this example, the value of `n` is referenced to calculate the size of the array arguments `A`, `B`, `C`. Since it is designed for numerical applications, the supported data type in Ninf IDL is tailored for such a purpose; for example, the data types are limited to scalars and their multi-dimensional arrays. The interface description is compiled by the *Ninf interface generator* to generate a stub program for each library function. The interface generator also automatically outputs a makefile with which the Ninf executables can be created by linking the stub programs and library functions.

To invoke a Ninf executable in a remote node, OmniRPC use the remote shell command “rsh” in a local area network and GRAM(Globus Resource Allocation Manager) API of Globus toolkit in a grid environment. The Ninf executable is invoked with the arguments of the client host name and port number for waiting the connection. For a grid environment, we designed OmniRPC on top of the Globus toolkit. The Globus I/O module is used in a grid environment

instead of TCP/IP in local area network. The Globus also provides security and authentication by GSI (Globus Security Infrastructure).

To handle multiple outstanding RPC requests from a multi-threaded client, OmniRPC maintains the queue for outstanding remote procedure calls. `OmniRPC_Call()` enqueues the request for the remote call, and blocked for waiting the return from the remote call. The scheduler thread is created to manage the queue. For the requested call in the queue, it searches the database of the remote function entries to schedule the requests to the remote nodes. When the results are sent back, the scheduler thread receives the results and stores it into the output argument of the call. Then, it resumes the waiting thread. The current implementation uses a simple round-robin scheduling. The machine file contains the maximum number of jobs as well as the list of host names. When all remote nodes are busy and the number of jobs reaches to the limit, the thread executing `OmniRPC_Call()` is blocked. As soon as the jobs for requested remote call is over, the next request is scheduled if any waiting requests exist in the queue.

3 OpenMP Client Using OmniRPC

Since OmniRPC is thread-safe, multiple remote procedure calls can be outstanding simultaneously from multi-threaded programs written in OpenMP.

A typical application of OmniRPC in OpenMP is to execute the same procedure over different input arguments as follows:

```
OmniRPC_init(); /* initialize RPC */
....
#pragma omp parallel for
for(i = 0; i < N; i++)
  OmniRPC_Call("work",i,...);
```

In this loop, the remote function `work` are executed in parallel with different arguments `i` in the remote nodes.

The procedure-level task-parallelism is also described as in the following code:

```
#pragma omp parallel sections
{
#pragma omp section
  OmniRPC_Call("subA");
#pragma omp section
  OmniRPC_Call("subB");
#pragma omp section
  OmniRPC_Call("subC");
}
```

The OpenMP clients of OmniRPC can be executed in a single processor. In our Omni OpenMP compiler, we need to execute OpenMP program containing OmniRPC calls in the following environment:

- Set the environment variable `OMP_SCHEDULE` to `"static,1"`, meaning cyclic scheduling with chunk size 1. In the compiler, the default loop scheduling is block-scheduling which may cause load imbalance when the execution time of each remote call changes.
- Set the environment variable `OMP_NUM_THREADS` to the number greater than the total number of jobs in available remote nodes. The large numbers of threads are needed to issue the remote procedure call simultaneously for many remote nodes. Furthermore, multiple requests to the remote nodes may hide the latency of communication and the overhead of executable invocation by the local scheduler in remote nodes.
- Compile with "mutex-lock" configuration. As default, the Omni OpenMP compiler uses the spin-lock for fast synchronization in a multiprocessor. It, however, sometimes delays the context-switch between threads in a single processor. The mutex-lock configuration uses the mutex lock of the POSIX thread library for better operating system scheduling.

In the recent release OpenMP 2.0, there is a new clause, `NUM_THREADS` to parallel region directives. This clause requests that a specific number of threads are used in the regions. This also works for nested regions with task-parallelism and loop-parallelism as in the following code:

```
#pragma omp parallel sections num_threads(3)
{
#pragma section
#pragma omp parallel for num_threads(10)
  for(i = 0; i < N; i++) OmniRPC_Call("workA",i);
#pragma section
#pragma omp parallel for num_threads(20)
  for(j = 0; j < N; j++) OmniRPC_Call("workB",j);
#pragma section
  workC();
}
```

4 Current Status and Future Work

In this paper, we proposed a parallel programming model using the thread-safe OmniRPC in an OpenMP client program for a cluster and global computing. The programmer can build a global computing system by using the remote libraries as its components, without being aware of complexities and hassles of network programming. OpenMP provides an ease-to-use and simple programming environment for a multi-threaded client of OmniRPC. Currently, we have finished a preliminary implementation of OmniRPC. We are doing several experiments and evaluations on some parametric search applications.

The current implementation employs a simple round-robin scheduling over available remote nodes. In the grid environment, the computation time of RPCs may be greatly influenced by many factors including computational ability of

the nodes, the distance to the nodes with respect to the bandwidth of communication, and the status of the nodes. More sophisticated scheduling using such dynamic information reported by the local job scheduler in the remote node will be required for efficient remote execution.

We are also developing a remote executable management tool for OmniRPC, which sends the IDL of remote functions to generate stubs in remote nodes automatically. It allows the user to install remote libraries without complex and time-consuming install procedure of remote executables for many remote nodes.

References

1. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, vol.11, No.2, pages 115–128, 1997. `http://www.globus.org/`.
2. M. Sato, H. Nakada S. Sekiguchi, , S. Matsuoka, U. Nagashima, and H. Takagi. Ninf: A Network based Information Library for Global World-Wide Computing Infrastructure. *Proc. of HPCN'97 (LNCS 1225)*, pages 491–502, 1997. `http://ninf.etl.go.jp/`.
3. H. Casanova and J. Dongarra. Netsolve: A network server for solving computational science problems. Technical report, University of Tennessee, 1996.
4. `http://pdplab.trc.rwcp.or.jp/Omni/`