Ninf-G Manual

This document is intended to provide the reader with a discussion of how to use the different components of the Ninf-G system and to serve as a reference manual for the commands and functions made available by Ninf-G. Although we offer a brief discussion of the Ninf-G System, this document is not necessarily intended to provide details about the Ninf-G components. The reader should refer to the *Documentation* section of the Ninf-G homepage (http://ninf.apgrid.org/) for more appropriate discussion of the Ninf-G system. The reader is expected to have some level of familiarity with programming and at least one programming languages, preferably the C language. Rudimentary knowledge of the UNIX operating system environment, the **make** utility, and the Globus toolkit will prove handy if installing and configuring Ninf-G for the UNIX environment.

Organization of this document

This document is divided into four parts. These parts are aimed at the needs of different types of users. Therefore, it is not necessary for a user to read all parts of this users' guide.

- **Introduction**
  This section provides a general overview of GridRPC and the Ninf-G system.
- **Installation Manual**
  This section provide installation instructions for the Ninf-G software. Currently
- **The Administrator's Manual**
  This section is aimed at the user who will be installing, managing the Ninf-G system, and providing their remote numerical libraries. It gives instructions of how to register numerical libraries with the Ninf-G system.
- **The User's Manual**
  This section is aimed at the Ninf-G usesr who is only interested in utilizing the client interfaces. It provides a discussion of the available client interfaces.
- **Appendices**
  These appendices provides reference manuals for the C client interfaces, and an overview of the Ninf-G IDL.

Request for Comments

Please help us improve future editions of this document by reporting any errors, inaccuracies, bugs, misleading or confusing statements, and typographical errors that you find. Email your bug reports and comments to us at ninf@apgrid.org. Your help is greatly appreciated.

Next

Table of Contents

Ninf-G Manual

This document is intended to provide the reader with a discussion of how to use the different components of the Ninf-G system and to serve as a reference manual for the commands and functions made available by Ninf-G. Although we offer a brief discussion of the Ninf-G System, this document is not necessarily intended to provide details about the Ninf-G components. The reader should refer to the *Documentation* section of the Ninf-G homepage (http://ninf.apgrid.org/) for more appropriate discussion of the Ninf-G system. The reader is expected to have some level of familiarity with programming and at least one programming languages, preferably the C language. Rudimentary knowledge of the UNIX operating system environment, the **make** utility, and the Globus toolkit will prove handy if installing and configuring Ninf-G for the UNIX environment.

Organization of this document

This document is divided into four parts. These parts are aimed at the needs of different types of users. Therefore, it is not necessary for a user to read all parts of this users' guide.

- **Introduction**
  This section provides a general overview of GridRPC and the Ninf-G system.
- **Installation Manual**
  This section provide installation instructions for the Ninf-G software. Currently
- **The Administrator's Manual**
  This section is aimed at the user who will be installing, managing the Ninf-G system, and providing their remote numerical libraries. It gives instructions of how to register numerical libraries with the Ninf-G system.
- **The User's Manual**
  This section is aimed at the Ninf-G usesr who is only interested in utilizing the client interfaces. It provides a discussion of the available client interfaces.
- **Appendices**
  These appendices provides reference manuals for the C client interfaces, and an overview of the Ninf-G IDL.

Request for Comments

Please help us improve future editions of this document by reporting any errors, inaccuracies, bugs, misleading or confusing statements, and typographical errors that you find. Email your bug reports and comments to us at ninf@apgrid.org. Your help is greatly appreciated.

Next

Introduction

Ninf-G is a Grid RPC system built on top of Globus Tookit. GridRPC is a middleware that provides remote library access and task-parallel programming model on the Grid. Representative systems include Ninf, Netsolve, etc. GridRPC can be effectively used as a programming abstraction in situations including the followings:

Utilizing resources that are specific to a specific computer on the Grid.

Commercial programs and libraries are often only provided in binary form, and cannot be executed on a machine with different CPU or networking architecture. Software License, source code incompatibility, could also be problematic. Special peripherals such as video cameras, electron microscopes, telescopes, and other various sensors may only be available as a resource on a particular machine, and their software libraries must

Executing Compute/Data Intensive Routines on Large Servers on the Grid.

For most programs, time is dominated by only a small portion of the entire program. By offloading such parts to large compute servers, we significantly reduce the time required for overall program execution. Another example is where the client machine is too constrained in terms of memory or disk space to perform large-scale computations. In such a case, transparent offloading without considering argument marshalling is desirable.

Parameter Sweep Studies using multiple servers on the Grid

Parameter sweep involves taking subsets of the overall parameter space and farming them off in a parallel, systematic manner to multiple servers. Each server performs identical computation with different sets of parameters, independent or semi-independent of other servers. There are a surprising number of real-world applications that could be categorized as parameter-sweep; Monte-Carlo is one such example. Although parameter sweep can be implemented using message passing libraries MPI, GridRPC allows considerably easier programming, and moreover allows the program to scale automatically to the Grid, such as remotely using multiple clusters, with considerations for properties such as resource allocation and security.

General and Scalable Task-Parallel Programs on the Grid.

GridRPC can be effectively used to implement task-parallel programs in a easy and transparent manner on the Grid. APIs are available to support various types of task-parallel synchronization, where multiple client-server interactions can become quite complex. Not only that GridRPC provides numerics-friendly, easy-to-use interface for task-parallelism, but also allows the computation to scale to the Grid in the same manner as the parameter sweep case. Section 2 will cover the basic •gGridifying•h of a given numerical library using GridRPC. Section3 will cover how binary executables with command-line interfaces and file I/O can be wrapped as a GridRPC component.

Section 4 will exemplify parallel programming for the parameter sweep case. (Section 5, task parallel programming, is in the works.).

Next

Installing Ninf-G

The Ninf-G software is available for UNIX/UNIX-like operating systems and bundled into one tar-gzipped file except for the Globus toolkit. This section attemppts to explain the steps to build and install Ninf-G at your site. It is recommended that you read completely through these instructions before beginning the installation.

Getting Started

To get started installing the Ninf-G system, you first should make sure the Globus toolkit is installed properly and available on your host. When install the Globus Toolkit Ver.2, **all bundles must be installed from source bundles** . Required libraries are not built from binary bundles. If you do not have Globus installed, you must perform the installation and deployment of the Globus toolkit prior to installing Ninf-G. The Globus Toolkit as well as the installation instruction is available from the Globus Project Web site . Both Globus Ver.2 (2.0 and 2.2) and Globus Ver.1 (1.1.3 and 1.1.4) could be used as a base system of the Ninf-G. Be sure that the installation instructions differs according to the version of the Globus Toolkit.

System Requirements

The Ninf-G depends on the Globus toolkit and its related software such as SSLeay and OpenLDAP. The platform you intend to install must be "Globus-enabled" at least. The Ninf Team installs the Ninf-G system on the following platforms prior to each software release and runs tests to ensure basic functionality. Bug reports are accepted from a wide range of hardware and software platforms, but problems with the platforms listed here are the easiest for the team to reproduce the test.

- Red Hat Linux 6.1, 6.2, 7.1, 7.2, 7.3
- Solaris 7, 8
- IRIX 6.5

Download

The latest release of the Ninf-G software is available from the Ninf Project's Download page . Ninf-G is provided as a single compressed tar file. The tar file contains:

- A "configure" script
- all source code
- C-based XML parsing library
- header files
- makefiles required to build and install Ninf-G
- sample numerical libraries and client applications
- manual documents

create user *ninf*

We recommend to create user *ninf* and install the Ninf-G by the user *ninf*.

Installation

After downloading the Ninf-G software and ensuring the above requirements, perform the following steps. The Ninf-G software is available for UNIX/UNIX-like operating systems. All of the client and server software except the Globus toolkit is bundled into one tar-gzipped file.
We assume that all of the Globus components work properly beforehand. The Globus toolkit is available from the Globus project. The Ninf-G distribution tar file is available from the Ninf homepage. Instruction steps are as below:

- **Environment variable *GLOBUS_LOCATION* (for GT2) or *GLOBUS_INSTALL_PATH* (for Globus Ver.1) must be defined to specify the installation directory of the Globus Toolkit.**
- **Notes for GT2.2 or later: --without-gpt option is required if *GPT_LOCATION* is not defined.**
- Place the tar file (e.g. *ng-latest.tgz*) on the system where you would like to maintain the source.
- Uncompress and untar the file

  ```
  % gunzip -c ng-latest.tgz | tar xvf -
  ```

  This creates a directory called ***ng*** , which makes up the source tree along with its subdirectories.
- Change directories into the ***ng*** directory, and type in the top diretory,

  ```
  % cd ng/
  %./configure
  ```

  This aids in the build process by providing host-specific software information to identify compilers and other tools to be used during compilation, and allows you to deploy the system in an easy way. If you would rather specify the installation directory explicitely or other site-specific parameters, you could add some arguments after the command like the following,

  ```
  % ./configure --prefix=/usr/local/ng
                --with-globus=/usr/local/globus
                --with-globusVersion=1
  ```

  This example defines the following required arguments:
  - Path to the Ninf-G installation directory
  - Path to the Globus Toolkit directory
  - Use Globus Ver.1 as a base toolkit

  All options to configure are documented in the below section or can be seen by running

  ```
  % ./configure --help
  ```

  When using Globus Ver.1, *--with-globusVersion=1* option must be specified. Version 2 is used as a default value of Globus Version.S

- Compile all components of the Ninf-G software by typing

```
% make
```

  Ninf-G uses a configure script generated by GNU autoconf to produce makefiles. If you have POSIX make program then the makefiles generated by configure will try to take advantage of POSIX make features. If your make is unable to process the makefiles while building you may have a broken make. Should make file during the build, try using GNU make.
- Install compiled files and the LDIF files onto the proper directory by executing as root or appropriate privileges.

```
% make install
```

- Register the host information by running the following command as a owner user of *GLOBUS_LOCATION*. It is usually a user *globus*.

```
% cd ng/bin
% ./server_install
```

- Add the information provider to the GRIS.
  - For Globus Ver.1:
    Add the following line to ${GLOBUS_DEPLOY_PATH}/etc/grid-info-resource.conf. This file maintains local resource information services provided by the GRIS which generate data based on information within the file.

```
0 cat ${GLOBUS_DEPLOY_PATH}/var/gridrpc/*.ldif
```

  - For GT2:
    - Add the following line to ${GLOBUS_LOCATION}/etc/grid-info-slapd.conf.

```
include  ${GLOBUS_LOCATION}/etc/grpc.schema
```

      The line should be put just below the following line.

```
include  ${GLOBUS_LOCATION}/etc/grid-info-resource.schema
```

    - Type the following command for restarting GRIS.

```
% ${GLOBUS_LOCATION}/sbin/SXXGris stop
```

```
% ${GLOBUS_LOCATION}/sbin/SXXGris start
```

- ## NOTES for dynamic linkage of the Globus shared libraries:

The Globus dynamic linking libraries (shared libraries) must be linked with the Ninf-G stub executables. In order to enable this, set the shared library linkage path appropriately

according to the system on which the Ninf-G stubs are installed. For example, this is done by adding the location of Globus libraries (${GLOBUS_LOCATION}/lib) to the environment variable ${LD_LIBRARY_PATH} of the Globus Gatekeeper spawning via inetd, or on Linux, add (${GLOBUS_LOCATION}/lib) in /etc/ld.so.conf.

Configure Options

Available options to be used for configuration can be listed by typing

```
% ./configure --help
```

The following is detailed information on the options to the configure script.

Generic Configure Options

The following are generic configure options that do not affect the functionality of Ninf-G.

Option	Description
--cache-file=FILE	cache test results in FILE
--help	print this message
--no-create	do not create output files
--quiet, --silent	do not print `checking...' messages
--version	print the version of autoconf that created configure

Directory and file names

These options specify where Ninf-G objects will be placed.

Option	Description
--prefix=PREFIX	install architecture-independent files in PREFIX [/usr/local]
--exec-prefix=EPREFIX	install architecture-dependent files in EPREFIX [same as prefix]
--bindir=DIR	user executables in DIR [EPREFIX/bin]
--sbindir=DIR	system admin executables in DIR [EPREFIX/sbin]
--libexecdir=DIR	program executables in DIR [EPREFIX/libexec]
--datadir=DIR	read-only architecture-independent data in DIR [PREFIX/share]
--sysconfdir=DIR	read-only single-machine data in DIR [PREFIX/etc]
--sharedstatedir=DIR	modifiable architecture-independent data in DIR [PREFIX/com]
--localstatedir=DIR	modifiable single-machine data in DIR [PREFIX/var]
--libdir=DIR	object code libraries in DIR [EPREFIX/lib]

--includedir=DIR	C header files in DIR [PREFIX/include]
--oldincludedir=DIR	C header files for non-gcc in DIR [/usr/include]
--infodir=DIR	info documentation in DIR [PREFIX/info]
--mandir=DIR	man documentation in DIR [PREFIX/man]
--srcdir=DIR	find the sources in DIR [configure dir or ..]
--program-prefix=PREFIX	prepend PREFIX to installed program names
--program-suffix=SUFFIX	append SUFFIX to installed program names
--program-transform-name=PROGRAM	run sed PROGRAM on installed program names

Host type:

Option	Description
--build=BUILD	configure for building on BUILD [BUILD=HOST]
--host=HOST	configure for HOST [guessed]
--target=TARGET	configure for TARGET [TARGET=HOST]

Features and packages:

In general, these options take the following forms.

Option	Description
--disable-FEATURE	do not include FEATURE (same as --enable-FEATURE=no)
--enable-FEATURE[=ARG]	include FEATURE [ARG=yes]
--with-PACKAGE[=ARG]	use PACKAGE [ARG=yes]
--without-PACKAGE	do not use PACKAGE (same as --with-PACKAGE=no)
--x-includes=DIR	X include files are in DIR
--x-libraries=DIR	X library files are in DIR
--enable and --with	options recognized:
--with-globusDir=DIR	specify where Globus is installed (*GLOBUS_LOCATION* (for GT2) or *GLOBUS_INSTALL_PATH* (for Globus Ver.1) environmental variable default)
--with-globusFlavor=FLAVOR	specify Globus runtime library flavor.
--with-globusVersion=VERSION(1 or 2)	specify the version of the Globus Toolkit (2 is default).
--with-cc=CC	specify C compiler to use (cc default)
--with-opt=OPT	specify C compiler options for optimization
--with-debug=OPT	specify C compiler options for debuggable executaable file creation
--with-cppflag=OPT	specify C preprocessor options

--enable-gcc	allow use of gcc if available
--enable-debug	enable generate executable with debug symbol (false default).

•@

Next

The Administrator's Manual

This section describes the way to create and manage Ninf-G libraries.

Managing the Ninf-G remote libraries

Before starting, be sure that the environment variables *GLOBUS_LOCATION* (for GT2) or *GLOBUS_INSTALL_PATH* (for Globus Ver.1), and *NS_DIR* are defined appropriately. The Ninf-G system has no server component in charge of dispatching an appropriate numerical library to the client unlike the original Ninf system. As described in the previous section, the Globus JobManager on the host which the Ninf-G remote libraries are installed spawns off a numerical library. Consequently, it is not required to start or stop the Ninf-G server component explicity as long as Globus is available. Be sure that the Globus gatekeeper and GRIS/GIIS are appropriately running and available on the host.

Next

The User's Manual

Build Ninf-G remote libraries

The Ninf-G remote libraries are implemented as executable programs which contain network stub routine as its main routine, and spawned off by the Gram component on the Ninf-G system. We call such executable programs *Ninf-G executable* associated with its name, and executes the found executable, sets up an appropriate communication with the client. The stub routine handles the communication to the Ninf-G system and its client, including argument marshalling. The underlying executable can be written in any existing scientific languages such as Fortran, C, etc, as long as it can be executed in the host.
To register and publish your numerical libraries and computational resource, you first should provide interface information about those numerical libraries and prepares the Ninf-G executable by performing the following task.

- Write a interface information of the numerical libraries you wish to register using interface description language called Ninf-G IDL. The complete syntax of Ninf-G IDL is described in the Appendix. The following code is a sample Ninf-G IDL for matrix multiplication.

  ```
  Module mmul;

  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. */
  ```

  As such, you should provide the following information in the Ninf-G IDL file.
    - Module name.
    - Interface Information(marshaling information) of each library routine.
    - Information to make whole Module.
- Compile a Ninf-G IDL with Ninf-G generator and generates a stub main routine which bridges each numerical librarary and our Ninf-G runtime routines, and Makefile to compile the whole program.

  ```
  % ns_gen <IDL File>
  ```

  This will make a Makefile named "module".mak, some "_stub_XXX.c" files, XML files, and LDIF files.
- Make Ninf-G executables by typing

  ```
  % make -f <module_name>.mak
  ```

  This will make some "_stub_XXX" files
- Copy the LDIF file within the directory ${GLOBUS_DEPLOY_PATH}/etc/gridrpc by typing,

  ```
  make -f <module_name>.mak install
  ```

  The LDIF file looks like the following,

```
dn: rpcFuncname=mmul/dmmul, sw=GridRPC
objectclass: GridRPCEntry
hn:
rpcFuncname: mmul/dmmul
module: mmul
entry: dmmul
path: /usr/local/ninf/ninf-g/tests/mmul/_stub_dmmul
stub:: PGZ1bmN0aW9uICB2ZXJzaW9uPSIyMjEuMDAwMDAwIiA+PGZ1
 PSJwaSIgZW50cnk9InBpX3RyaWFsIiAvPiA8YXJnIGRhdGFfdHlwZT
 cGU9ImluIiA+CjwvYXJnPgo8YXJnIGRhdGFfdHlwZT0ibG9uZyIgbW
 Pgo8L2FyZz4KPGFyZyBkYXRhX3R5cGU9Imxvbmci IG1vZGVfdHlwZT
```

NOTES for dynamic linkage of the Globus shared libraries:

The Globus dynamic linking libraries (shared libraries) must be linked with the Ninf-G stub executables. In order to enable this, set the shared library linkage path appropriately according to the system on which the Ninf-G stubs are installed. For example, this is done by adding the location of Globus libraries (${GLOBUS_LOCATION}/lib) to the environment variable ${LD_LIBRARY_PATH} of the Globus Gatekeeper spawning via inetd, or on Linux, add (${GLOBUS_LOCATION}/lib) in /etc/ld.so.conf.

Programming Interface

Before starting, be sure that the environment variables *GLOBUS_LOCATION* (for GT2) or *GLOBUS_INSTALL_PATH* (for Globus Ver.1), and *NS_DIR* are defined appropriately. Since the original Ninf client programming interface was designed to be as language independent as possible, we have developed a variety of programming interfaces such as C, Fortran, Java, and Lisp. The concept of the Ninf-G system is the same. Currently the Ninf-G system supports the C interface, and we are in the process of building the Java interface and other language bindings. Ninf-G provides both Grid RPC native API (GRPC API) and Ninf compatible API (Ninf API). Ninf API is a full-compatible API with Ninf version 1 which provides simple and easy programming interface to programmers. GRPC API provides lower level and flexible programming interface to programmers. According to the requirements, application programmers can build Grid-enabled applications using either GRPC API or Ninf API. In this section, we introduce you several representative functions using C interface for both GRPC API and Ninf API. The complete API reference is described in the Appendix.

Grid RPC API

Initialize

This function read configuration file for your client program and set system

parameters for Ninf-G.

```
main (int argc, char **argv) {
   char *conf_file = argv[1];

   grpc_initialize(conf_file);
   ...
```

Synchronous Call

The synchronous call to Ninf-G from C is the easiest to implement, just call the function, *grpc_call()* . This function returns an error code. It takes as arguments the name of a problem and the list of input data and output data. These inputs are listed according to the calling sequence defined in the corresponding Ninf IDL. The more detailed information on the Ninf-G IDL are described in the section called Ninf-G IDL The C prototype of the function is

```
grpc_call(grpc_function_handle_t *handle, <argument list>)
```

where *handle is a pointer to the handle to the numerical library on the Ninf-G system. Let us resume our example of the call to a simple matrix multiplication. In C, the local function to this caliculation looks like

```
 mmul(double *A, double *B, double *C),
```

Meanwhile, the equivalent synchronous call to Ninf-G in C is

```
grpc_function_handle_t *handle;
grpc_function_handle_init(&handle, "ninf.apgrid.org",
                          3030, "lib/mmul");
status = grpc_call(&handle, A, B, C);
```

As such, you should initialize a function handle to the Ninf-G remote library using *grpc_function_handle_init* which takes as arguments a server name, port number, and a module name of the library.

Asynchronous Call

The standard grpc_call() RPC is synchronous in that the client waits until the completion of the computation on the server side. For task-parallel execution, Ninf-G facilitates several asynchronous call APIs. For example, the most basic asynchronous call

```
int grpc_async_call(grpc_function_handle_t *handle,
                    <argument list>)
```

It is almost identical to *grpc_call* except that it returns immediately after all the

arguments have been sent. The return value is the session ID of the asynchronous call; the ID is used for various synchronizations such as waiting for the return value of the call.
There are several calls for synchronization. The most basic is the

```
grpc_wait(int sessionID);
grpc_wait_all();
```

, *grpc_wait()* is the function with which we wait for the result of the asynchronous call with the supplied session ID. *grpc_wait_all()* waits for all preceding asynchronous invocations made.

Running the Ninf-G Client

After developing the Ninf Client Application by the use of the programming interface, what you need to do for running the application are listed here.

- **Compile your Ninf-G client application using *ns_client_gen***

  ```
  ${NG_DIR}/bin/ns_client_gen -g -o test_client test_client.c
  ```

- **Write a configuration file as the following.**

  ```
  #
  #       Ninf-G Client Config File
  #

  port    = 24001                         # specify the port number
  serverhost      = brain.a02.aist.go.jp # Globus node
  ldaphost        = brain.a02.aist.go.jp # LDAP Host
  ldapport        = 2135                  # LDAP Port number
  ```

- **Obtain a Globus proxy**

  Before submitting any jobs to the Ninf-G system, you need to run the following command and get a proxy certificate, which allows you to utilize any Ninf-G numerical libraries on the Globus resource without reentering your pass phrase while your proxy is active. We assume you have already obtained a Globus Certificate from the Globus Certificate Authority and you are in the resources' grid mapfiles.

  ```
  % grid-proxy-init
  ```

  You can set the time for the proxy to be in effect; use

```
% grid-proxy-init -hours (nhours)
```

If you wish to determine your proxy status, use *grid-proxy-info -all* to display the contents or test the status of your proyx.

- Run the Ninf-G client

```
% ./test_client config.cl
```

Managing Jobs

Since the Ninf-G system is built on top of the Globus, you can directly take advantage of the Globus commands for managing the submitted jobs such as checking, killing, and cleaning. We give you very quick overview of the Globus commands, and more detailed explanation can be seen in the Globus manual.

Checking my job

```
% globus-job-run <hostname> /bin/ps -u <username>
```

When you submit a job to the batch scheduler, enter *globus-job-status* followed by the job contact string URL.

```
% globus-job-submit <hostname> <command>
https://<hostname>:3031/342265377/
% globus-job-status https://ninf.apgrid.org:3031/942265377/
```

The different status are PENDING(waiting in the queue), ACTIVE(running), SUSPENDED, DONE, and FAILED.

Canceling/Cleaning a Job

If you interrupt the *globusrun* or *globus-job-run* process(by entering CTRL-C or sending it a SIGINT), your jobs should automatically be canceld.

Ninf API

Initialize

This function parse argument list for your client program and retrieve arguments for Ninf-G system. This function returns new argc value. Here is an example showing typical usage.

```
main (int argc, char ** argv){
        argc = Ninf_parse_arg(argc, argv);
        ...
```

Synchronous call

This function calls Ninf function on a remote server. The first argument specifies server and function. Here is an example that calls "mmul" function.

```
Ninf_call("mmul", n, A, B, C);
```

You can omit the server name and port. In such case, a default server will be used. Default server can be specified by environment variable "NINF_SERVER" and "NINF_SERVER_PORT", or Ninf_parse_arg. This function returns 0 if succeed, returns -1 if failed.

Asynchronous Call

This function is almost same as Ninf_call, but it returns immediately. Using this function, you can invoke several sessions simultaneously. It returns non-negative ID if succeed, returns -1 if failed.

```
id = Ninf_call_async("mmul", n, A, B, C);
```

There are several calls for synchronization. The most basic is the

```
Ninf_wait(int sessionID);
Ninf_wait_all();
```

, *Ninf_wait()* is the function with which we wait for the result of the asynchronous call with the supplied session ID. *grpc_wait_all()* waits for all preceding asynchronous invocations made.

As mentioned above, `Ninf_call()` is the client interface to the Ninf compute and database servers. 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(N,A,B,C);                  /* calls matrix multiply, C = A
* B */
```

When the `dmmul` routine is available on a Ninf server, the client program can call the remote library using `Ninf_call` , in the following manner:

```
Ninf_call("dmmul",N,A,B,C); /* call remote Ninf library on server
*/
```

Here, `dmmul` is the name of library registered as a Ninf executable on a server, and `N,A,B,C` 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; `Ninf_call()` automatically determines the function arity and the type of each

argument, appropriately marshals the arguments, makes the remote call to the server, obtains the results, places the results in the appropriate argument, and returns to the client. In this way, the Ninf RPC is designed to give the users an illusion that arguments are shared between the client and the server.

The reader may also notice that the physical location of the Ninf server is not specified in the example. Ninf provides a variety of ways for the client to specify the server and its registered library:

- **Specify server with an environment variable.**
- **Directly specify server and library with an URL of the Ninf executable.**

To realize such simplicity in the client programming interface, we designed Ninf RPC so that client function call obtains all the interface information regarding the called library function at runtime from the server. Although this will cost an extra network round trip time, we judged that typical scientific applications are both compute and data intensive such that the overhead should be small enough. The interface information includes:

- **the number of parameters.**
- **argument types.**
- **access mode of arguments (read or write).**
- **size of arguments.**
- **argument marshaling information.**
- **how to send and receive data from/to the Ninf server.**

The client function call requests the interface information of the calling function to the Ninf server, which in turn returns the registered Ninf executable interface information to the client. The client library then interprets and marshals the arguments on the stack according to the supplied information. For variable-sized array arguments, the IDL must specify an expression that includes the input scalar arguments whereby the size of the arrays can be computed. This design is in contrast to traditional RPCs, where stub generation is done on the client side at compile time. As a result of dynamic interface acquisition, Ninf RPC does not require such compile-time activities at all, relieving the users from any code maintenance. [2] describes the structure of the interface information and the protocol in detail.

In the above example, the client function call sends the input arrays, `A` and `B`, whose size is computed by the parameter `N` . The Ninf server invokes the Ninf executable of `dmmul` library, and forwards the input data to it. When the computation is done, the result is sent back to the client through the server. The client function call stores the

returned data at the location pointed by the argument, C.

The Ninf RPC may also be invoked asynchronously to exploit network-wide parallelism. It is possible to issue the request to a Ninf server, continue with the other computation, and poll for the request later. Multiple RPC requests to different servers are also possible. For this reason, the asynchronous Ninf PRC is an important feature for parallel programming.

Next

Examples

This section give you a tutorial of how to use the Ninf-G system for programming on the Gird. Simplicity of programming is the most beneficial aspect of the Ninf-G system, and we hope that users will be able to gridify his programs easily after reading this document. We hope to extend this example further to cover more advanced Ninf-G features. Examples are provided for both GRPC API and Ninf API.

Grid RPC API

Gridfying a Numerical Library with Grid RPC API

We first cover the simple case where the library to be Gridifyied is defined as a linkable library function. Below is a sample code of a simple matrix multiply. The first scalar argument specifies the size of the matrix (n by n), parameters a and b are references to matrices to be multiplied, and c is the reference to the result matrix. Notice that, 1) the matrix (defined as arrays) do not itself embody size as type information, and 2) as a result there is a dependency between n and a, b, c. In fact, since array arguments are passed as a reference, one must assume the contents of the array are implicitly shared by the caller and the callee, with arbitrary choices as to using them as input, output, or temporary data structures.

```
void  mmul(int n, double * a, double * b, double * c){
  double t;
  int i, j, k;
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      t = 0;
      for (k = 0; k < N; k++){
        t += a[i * n + k] * b[k * n + j];
      }
      c[i*N+j] = t;
    }
  }
}
```

The main routine which calls mmul() might be as follows:

```
main(){
  double A[N*N], B[N*N], C[N*N];

  initMatA(N, A);  /* initialize */
  initMatB(N, B);  /* initialize */

  mmul(N, A, B, C);
}
```

In order to •gGridify•h, or more precisely, allow mmul to be called remotely via GridRPC, we must describe the interface of the function so that information not embodied in the language type system becomes sufficiently available to the

GridRPC•@system to make the remote call. Although future standardization is definitely conceivable, currently each GridRPC•@system has its own IDL (Interface Description Language); for example, Ninf has its own NinfIDL definition. Below we give the interface of mmul() defined by the NinfIDL syntax:

```
1: Module mmul;
2:
3: Define mmul(IN int    N,      IN  double A[N*N],
4:             IN double B[N*N], OUT double C[N*N])
5: "matmul"
6: Required "mmul_lib.o"
7: Calls "C" mmul(N, A, B, C);
```

Line 1 declares the module name to be defined. There is a one-to-one correspondence between a module and an IDL file, and each module can have multiple entries to gridify multiple functions. Lines 3-7 are the definition for a particular entry mmul/mmul. Here, lines 3 and 4 declare the interface of the entry. The difference between a NinfIDL entry definition and the C prototype definition is that there are no return values (the return value of the Ninf call is used to return status info), argument input/output modes are specified, and array sizes are described in terms of the scalar arguments.

We note here that NinfIDL has special features to efficiently support gridifying of a library (similar features are found in Netsolve IDL). In contrast to standard procedure calls within a shared memory space, GridRPC needs to transfer data over the network. Transferring the entire contents of the array will be naturally very costly, especially for huge matrices appearing in real applications. Here, one will quickly observe that surprising number of numerical libraries take for granted the fact that address space of data structures, in particular arrays are shared, and (a) only use subarrays of the passed arrays, (b) write back results in the passed arrays, and (c) pass arrays as scratchpad temporaries. The portion of the arrays to be operated, etc., are determined by the semantics of the operation according to the input parameters passed to the function. For example in mmul, the whole arrays need to be passed, and their sizes are all N by N, where N is the first scalar parameter; A and B only need to be passed as input parameters and their contents do not change, while C is used as a return argument and thus need not be shipped to the server, but the result needs to be shipped back to the client. In general, in order to determine and minimize the size of transfer, NinfIDL allows flexible description of the array shape used by the remote library. One can specify leading dimensions, subarrays, and strides. In fact arbitrary arithmetic expressions involving constants and scalar arguments can be used in the array size expressions.

Line 5 is the comment describing the entry, while line 6 specifies the necessary object file when the executable for the particular file is to be linked. Line 7 gives the actual library function to be called, and the calling sequence; here •gC•h denotes C-style (row-major) array layout.

The user compiles this IDL file using the Ninf IDL compiler, and generates the stub code and its makefile. By executing this makefile a Ninf executable is generated. The user will subsequently register the executable to the server using the registry tool.

Now the client us ready to make the call of the network. In order to make a GridRPC call, the user modifies his original main program in the following manner. We notice that only the function call is modified---No need to change the program to adjust to the skeleton that the IDL generator generates as is with typical RPC systems such as CORBA. Moreover, we note that the IDL, the stub files and the executables are only resident on the server side, and the client only needs to link his program with a generic Ninf client library.

```
main(){
  double A[N*N], B[N*N], C[N*N];
  grpc_function_handle_t handle;

  grpc_initialize(argv[1]);

  initMatA(N, A);  /* initialize */
  initMatB(N, B);  /* initialize */
```

```
  grpc_function_handle_default(&handle, "mmul/mmul");

  if (grpc_call(&handle, N, A, B, C) != GRPC_ERROR) {
    fprintf(stderr, "Error in grpc_call\n");
    exit(1);
  }

  grpc_function_handle_destruct(&handle);

  ...

  grpc_finalize();
}
```

Gridifying Programs that use Files

The above example assumes that the numerical routine is supplied as a library with well-defined function API, or at least its source is available in a way such that it could easily converted into a library. In practice, many numerical routines are only available in a non-library executable and/or binary form, with input/output interfaces using files. In order to gridify such •gcanned•h applications, GridRPC systems typically support remote files and their automatic management/transfer.

We take gnuplot as an example. Gnuplot in non-interactive mode inputs script from a specified file, and outputs the resulting graph to the standard output. Below is an example gnuplot script.

```
set terminal postscript
set xlabel "x"
set ylabel "y"
plot f(x) = sin(x*a), a = .2, f(x), a = .4, f(x)
```

If this script is saved under a filename •ggplot•h:

```
> gnuplot gplot > graph.ps
```

will store the postscript representation of the graph to the file graph.ps. In order to execute gnuplot remotely, we must package it appropriately, and moreover must automatically transfer the input (gplot) and output (graph.ps) files between the client and the server.

Ninf-G IDL•@provides a type *filename* to specify that the particular argument is a file. Below is an example of using gnuplot via GridRPC.

```
Module plot;
Define plot(IN filename plotfile, OUT filename psfile )
"invoke gnuplot"
{
    char buffer[1000];
    sprintf(buffer, "gnuplot %s > %s", plotfile, psfile);
    system(buffer);
}
```

The IDL writes the string command sequence to invoke gnuplot into a variable buffer[], and invokes gnuplot as a system library. The file specified as an input is automatically transferred to the temporary directory of the server, and its temporary file name is passed to the stub function. As for the output file, only the temporary file name is created and passed to the stub function. After the stub program is executed, the files in output mode as specified in the IDL are automatically transferred to the client, and saved there under the name given in the argument.

Below is an example of how this function might be called via GridRPC.

```
#include <stdio.h>
#include "grpc.h"

main(int argc, char ** argv){
  grpc_function_handle_t handle;

  grpc_initialize(argv[1]);

  grpc_function_handle_default(&handle, "plot/plot");

  if (grpc_call(&handle, argv[2], argv[3]) == GRPC_ERROR) {
    fprintf(stderr, "Error in grpc_call\n");
    exit(1);
  }

  grpc_function_handle_destruct(&handle);

  ...

  grpc_finalize();

}
```

We also note that, by combining this feature with the technique of using multiple servers simultaneously described in the next secion, we can process large amount of data at once.

Using Multiple Servers for Parallel Programming on the Grid --- The Parameter Sweep Survey Example.

GridRPC can serve as a task-parallel programming abstraction, whose programs can scale from local workstations to the Grid. Here, we take an example of simple parameter sweep survey, and investigate how it can be easily programmed using GridRPC.

Calculating PI using a simple Monte Carlo Method

As an example, we compute the value of PI using a simple Monte Carlo Method. We generate a large number of random points within the square region that exactly encloses a unit circle (actually, 1/4 of a circle). We calculate the value of PI by inverse computing the area of the circle according to the probability that the points will fall within the circle. The program below shows the original sequential version.

```
long pi_trial(int seed, long times){
  long l, long counter = 0;
  srandom(seed);
  for (l = 0; l < times; l++){
        double x = (double)random() /   RAND_MAX;
        double y = (double)random() /   RAND_MAX;
        if (x * x + y * y < 1.0)
          counter++;
  }
  return counter;
}

main(int argc, char ** argv){
```

```
  double pi;
  long times = atol(argv[1]);
  count = pi_trial(10, times);
  pi = 4.0 * (count / (double) times);
  printf("PI = %f\n", pi);
}
```

Gridifying the PI program.

First, we rewrite the program so that it does the appropriate GridRPC calls. The following steps are needed:

1. Separate out the pi_trail() function into a separate file (say, trial_pi.c), and create its object file trial_pi.o using a standard C compiler.
2. Describe the interface of pi_trial in an IDL file.

```
Module pi;

Define pi_trial(IN int seed, IN long times, OUT long * count)
"monte carlo pi computation"
Required "pi_trial.o"
{
  long counter;
  counter = pi_trial(seed, times);
  *count = counter;
}
```

3. Rewrite the main program so that it makes a GridRPC call.

```
main(int argc, char ** argv){
  double pi;
  long times, count;
  grpc_function_handle_t handle;

  grpc_initialize(argv[1]);

  times = atol(argv[2]);

  grpc_function_handle_default(&handle, "pi/pi_trial");

  if (grpc_call(&handle, 10, times, &count) == GRPC_ERROR){
    fprintf(stderr, "Failed in grpc_call\n");
    exit(2);
  }
  pi = 4.0 * ( count / (double) times);
  printf("PI = %f\n", pi);

  grpc_function_handle_destruct(&handle);

  grpc_finalize();
}
```

We now have made the body of the computation remote. The next phase is to parallelise it.

Employing Multiple Servers for Task Parallel Computation.

We next rewrite the main program so that parallel tasks are distributed to multiple servers. Although distribution of tasks

are possible using metaserver scheduling with Ninf (and Agents with Netsolve), it is sometimes better to specify a host explicitly for performance reasons, for low overhead and explicit load balancing. Ninf-G allows explicit specification of servers by specifying the hostname in the initialization of the function handle.

The standard grpc_call() RPC is synchronous in that the client waits until the completion of the computation on the server side. For task-parallel execution, Ninf-G facilitates several asynchronous call APIs. For example, the most basic asynchronous call grpc_call_async is almost identical to grpc_call except that it returns immediately after all the arguments have been sent. The return value is the session ID of the asynchronous call; the ID is used for various synchronizations such as waiting for the return value of the call.

There are several calls for synchronization. The most basic is the `grpc_wait(int ID)`, where we wait for the result of the asynchronous call with the supplied session ID. grpc_wait_all() waits for all preceding asynchronous invocations made. Here, we employ grpc_wait_all() to parallelize the above PI client so that it uses multiple simultaneous remote server calls:

```
 1  #include "grpc.h"
 2  #define NUM_HOSTS 8
 3  char * hosts[] = {"node0", "node1", "node2", "node3",
 4                    "node4", "node5", "node6", "node7"};
 5
 6  grpc_function_handle_t handles[NUM_HOSTS];
 7  int port = 4000;
 8
 9  main(int argc, char ** argv){
10    double pi;
11    long times, count[NUM_HOSTS], sum;
12    char * config_file;
13    int i;
14    if (argc < 3){
15      fprintf(stderr, "USAGE: %s CONFIG_FILE TIMES \n", argv[0]);
16      exit(2);
17    }
18    config_file = argv[1];
19    times = atol(argv[2]) / NUM_HOSTS;
20
21    /* Initialize GRPC runtimes. */
22    if (grpc_initialize(config_file) != GRPC_OK){
23      exit(2);
24    }
25    /* Initialize handles. */
26    for (i = 0; i < NUM_HOSTS; i++)
27      grpc_function_handle_init(&handles[i], hosts[i], port, "pi/pi_trial");
28
29    for (i = 0; i < NUM_HOSTS; i++){
30      /* Parallel non-blocking remote function invocation. */
31      if (grpc_call_async(&handles[i], i, times, &count[i]) == GRPC_ERROR){
32        grpc_perror("pi_trial");
33        exit(2);
34      }
35    }
36    /* Sync. */
37    if (grpc_wait_all() == GRPC_ERROR){
38      grpc_perror("wait_all");
39      exit(2);
40    }
```

```
41
42    for (i = 0; i < NUM_HOSTS; i++)
43      grpc_function_handle_destruct(&handles[i]);
44
45    /* Compute and display pi. */
46    for (i = 0, sum = 0; i < NUM_HOSTS; i++)
47      sum += count[i];
48    pi = 4.0 * ( sum / ((double) times * NUM_HOSTS));
49    printf("PI = %f\n", pi);
50
51    /* Finalize GRPC runtimes. */
52    grpc_finalize();
53  }
```

We specify the number of server hosts and their names in lines 2 and 3-4, respectively. Line 6 is the port number used, and line 19 divides the number of Monte Carlo trials with the number of servers, determining the number of trials per server. The for loop in lines 29-35 invokes the servers asynchronously. Line 47 aggregates the results returned from all the servers.

In this manner, we can easily write a parallel parameter sweep survey program using the task parallel primitives of GridRPC. We next modify the program to perform dynamic load balancing.

Ninf API

Gridfying a Numerical Library with GridRPC

We first cover the simple case where the library to be Gridifyied is defined as a linkable library function. Below is a sample code of a simple matrix multiply. The first scalar argument specifies the size of the matrix (n by n), parameters a and b are references to matrices to be multiplied, and c is the reference to the result matrix. Notice that, 1) the matrix (defined as arrays) do not itself embody size as type information, and 2) as a result there is a dependency between n and a, b, c. In fact, since array arguments are passed as a reference, one must assume the contents of the array are implicitly shared by the caller and the callee, with arbitrary choices as to using them as input, output, or temporary data structures.

```
void  mmul(int n, double * a, double * b, double * c){
  double t;
  int i, j, k;
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      t = 0;
      for (k = 0; k < N; k++){
        t += a[i * n + k] * b[k * n + j];
      }
      c[i*N+j] = t;
    }
  }
```

```
}
```

The main routine which calls mmul() might be as follows:

```
main(){
  double A[N*N], B[N*N], C[N*N];
  initMatA(N, A);  /* initialize */
  initMatB(N, B);  /* initialize */

  mmul(N, A, B, C);
}
```

In order to •gGridify•h, or more precisely, allow mmul to be called remotely via GridRPC, we must describe the interface of the function so that information not embodied in the language type system becomes sufficiently available to the GridRPC•@system to make the remote call. Although future standardization is definitely conceivable, currently each GridRPC•@system has its own IDL (Interface Description Language); for example, Ninf has its own NinfIDL definition. Below we give the interface of mmul() defined by the NinfIDL syntax:

```
1: Module mmul;
2:
3: Define mmul(IN int    N,       IN  double A[N*N],
4:             IN double B[N*N], OUT double C[N*N])
5: "matmul"
6: Required "mmul_lib.o"
7: Calls "C" mmul(N, A, B, C);
```

Line 1 declares the module name to be defined. There is a one-to-one correspondence between a module and an IDL file, and each module can have multiple entries to gridify multiple functions. Lines 3-7 are the definition for a particular entry mmul/mmul. Here, lines 3 and 4 declare the interface of the entry. The difference between a NinfIDL entry definition and the C prototype definition is that there are no return values (the return value of the Ninf call is used to return status info), argument input/output modes are specified, and array sizes are described in terms of the scalar arguments.

We note here that NinfIDL has special features to efficiently support gridifying of a library (similar features are found in Netsolve IDL). In contrast to standard procedure calls within a shared memory space, GridRPC needs to transfer data over the network. Transferring the entire contents of the array will be naturally very costly, especially for huge matrices appearing in real applications. Here, one will quickly observe that surprising number of numerical libraries take for granted the fact that address space of data structures, in particular arrays are shared, and (a) only use subarrays of the passed arrays, (b) write back results in the passed arrays, and (c) pass arrays as scratchpad temporaries. The portion of the arrays to be operated, etc., are determined by the semantics of the operation according to the input parameters passed to the function. For example in mmul, the whole arrays need to be passed, and their sizes are all N by N, where N is the first scalar parameter; A and B only need to be passed as input parameters and their contents do not change, while C is used as a return argument and thus need not be shipped to the server, but the result needs to be shipped back to the client. In general, in order to determine and minimize the size of transfer, NinfIDL allows flexible description of the array shape used by the remote library. One can specify leading dimensions, subarrays, and strides. In fact arbitrary arithmetic expressions involving constants and scalar arguments can be used in the array size expressions.

Line 5 is the comment describing the entry, while line 6 specifies the necessary object file when the executable for the particular file is to be linked. Line 7 gives the actual library function to be called, and the calling sequence; here •gC•h denotes C-style (row-major) array layout.

The user compiles this IDL file using the Ninf IDL compiler, and generates the stub code and its makefile. By executing this makefile a Ninf executable is generated. The user will subsequently register the executable to the server using the registry tool.

Now the client us ready to make the call of the network. In order to make a GridRPC call, the user modifies his original main program in the following manner. We notice that only the function call is modified---No need to change the program to adjust to the skeleton that the IDL generator generates as is with typical RPC systems such as CORBA. Moreover, we note that the IDL, the stub files and the executables are only resident on the server side, and the client only needs to link his program with a generic Ninf client library.

```
main(){
  double A[N*N], B[N*N], C[N*N];

  initMatA(N, A);  /* initialize */
  initMatB(N, B);  /* initialize */

  if (Ninf_call("mmul/mmul", N, A, B, C) != NINF_ERROR)
    Ninf_perror("mmul");
}
```

Gridifying Programs that use Files

The above example assumes that the numerical routine is supplied as a library with well-defined function API, or at least its source is available in a way such that it could easily converted into a library. In practice, many numerical routines are only available in a non-library executable and/or binary form, with input/output interfaces using files. In order to gridify such •gcanned•h applications, GridRPC systems typically support remote files and their automatic management/transfer.

We take gnuplot as an example. Gnuplot in non-interactive mode inputs script from a specified file, and outputs the resulting graph to the standard output. Below is an example gnuplot script.

```
set terminal postscript
set xlabel "x"
set ylabel "y"
plot f(x) = sin(x*a), a = .2, f(x), a = .4, f(x)
```

If this script is saved under a filename •ggplot•h:

```
> gnuplot gplot > graph.ps
```

will store the postscript representation of the graph to the file graph.ps. In order to execute gnuplot remotely, we must package it appropriately, and moreover must automatically transfer the input (gplot) and output (graph.ps) files between the client and the server.

NinfIDL•@provides a type *filename* to specify that the particular argument is a file. Below is an example of using gnuplot via GridRPC.

```
Module plot;
Define plot(IN filename plotfile, OUT filename psfile )
"invoke gnuplot"
{
    char buffer[1000];
    sprintf(buffer, "gnuplot %s > %s", plotfile, psfile);
    system(buffer);
}
```

The IDL writes the string command sequence to invoke gnuplot into a variable buffer[], and invokes gnuplot as a system library. The file specified as an input is automatically transferred to the temporary directory of the server, and its temporary file name is passed to the stub function. As for the output file, only the temporary file name is created and

passed to the stub function. After the stub program is executed, the files in output mode as specified in the IDL are automatically transferred to the client, and saved there under the name given in the argument.

Below is an example of how this function might be called via GridRPC.

```
#include <stdio.h>
#include "ninf.h"

main(int argc, char ** argv){
  argc = Ninf_parse_arg(argc, argv);

  if (argc < 3) {
    fprintf(stderr, "USAGE: plot_main INPUT PSFILE\n");
    exit(2);
  }
  if (Ninf_call("plot/plot", argv[1], argv[2]) == NINF_ERROR)
    Ninf_perror("Ninf_call plot:");
}
```

We also note that, by combining this feature with the technique of using multiple servers simultaneously described in the next secion, we can process large amount of data at once.

Using Multiple Servers for Parallel Programming on the Grid --- The Parameter Sweep Survey Example.

GridRPC can serve as a task-parallel programming abstraction, whose programs can scale from local workstations to the Grid. Here, we take an example of simple parameter sweep survey, and investigate how it can be easily programmed using GridRPC.

Calculating PI using a simple Monte Carlo Method

As an example, we compute the value of PI using a simple Monte Carlo Method. We generate a large number of random points within the square region that exactly encloses a unit circle (actually, 1/4 of a circle). We calculate the value of PI by inverse computing the area of the circle according to the probability that the points will fall within the circle. The program below shows the original sequential version.

```
long pi_trial(int seed, long times){
  long l, long counter = 0;
  srandom(seed);
  for (l = 0; l < times; l++){
        double x = (double)random() /   RAND_MAX;
        double y = (double)random() /   RAND_MAX;
        if (x * x + y * y < 1.0)
          counter++;
  }
  return counter;
}

main(int argc, char ** argv){
  double pi;
  long times = atol(argv[1]);
  count = pi_trial(10, times);
  pi = 4.0 * (count / (double) times);
  printf("PI = %f\n", pi);
}
```

Gridifying the PI program.

First, we rewrite the program so that it does the appropriate GridRPC calls. The following steps are needed:

1. Separate out the pi_trail() function into a separate file (say, trial_pi.c), and create its object file trial_pi.o using a standard C compiler.
2. Describe the interface of pi_trial in an IDL file.

```
Module pi;

Define pi_trial(IN int seed, IN long times, OUT long * count)
"monte carlo pi computation"
Required "pi_trial.o"
{
  long counter;
  counter = pi_trial(seed, times);
  *count = counter;
}
```

3. Rewrite the main program so that it makes a GridRPC call.

```
main(int argc, char ** argv){
  double pi;
  long times, count;
  argc = Ninf_parse_arg(argc, argv);
  times = atol(argv[1]);

  if (Ninf_call("pi/pi_trial", 10, times, &count) == NINF_ERROR){
    Ninf_perror("pi_trial");
    exit(2);
  }
  pi = 4.0 * ( count / (double) times);
  printf("PI = %f\n", pi);
}
```

We now have made the body of the computation remote. The next phase is to parallelise it.

Employing Multiple Servers for Task Parallel Computation.

We next rewrite the main program so that parallel tasks are distributed to multiple servers. Although distribution of tasks are possible using metaserver scheduling with Ninf (and Agents with Netsolve), it is sometimes better to specify a host explicitly for performance reasons, for low overhead and explicit load balancing. Ninf allows explicit specification of servers by the use of the URI format.

The standard Ninf_call() RPC is synchronous in that the client waits until the completion of the computation on the server side. For task-parallel execution, Ninf facilitates several asynchronous call APIs. For example, the most basic asynchronous call Ninf_call_async is almost identical to Ninf_call except that it returns immediately after all the arguments have been sent. The return value is the session ID of the asynchronous call; the ID is used for various synchronizations such as waiting for the return value of the call.

There are several calls for synchronization. The most basic is the `Ninf_wait(int ID)`, where we wait for the result of the asynchronous call with the supplied session ID. Ninf_wait_all() waits for all preceding asynchronous invocations made. Here, we employ Ninf_wait_all() to parallelize the above PI client so that it uses multiple simultaneous remote

server calls:

```
 1: #define NUM_HOSTS 16
 2: char * hosts[] =
 3: {"wiz00", "wiz01", "wiz02", "wiz03", "wiz04", "wiz05", "wiz06", "wiz07",
 4:  "wiz08", "wiz09", "wiz10", "wiz11", "wiz12", "wiz13", "wiz14", "wiz15"
 5: };
 6: int port = 4000;
 7:
 8: main(int argc, char ** argv){
 9:   double pi;
10:   long times, count[NUM_HOSTS], sum;
11:   int i;
12:   times = atol(argv[1]) / NUM_HOSTS;
13:
14:   for (i = 0; i < NUM_HOSTS; i++){
15:     char entry[100];
16:     sprintf(entry, "ninf://%s:%d/pi/pi_trial", hosts[i], port);
17:     if (Ninf_call_async(entry, i, times, &count[i]) == NINF_ERROR){
18:       Ninf_perror("pi_trial");
19:       exit(2);
20:     }
21:   }
22:   Ninf_wait_all();
23:   for (i = 0, sum = 0; i < NUM_HOSTS; i++)
24:     sum += count[i];
25:   pi = 4.0 * ( sum / ((double) times * NUM_HOSTS));
26:   printf("PI = %f\n", pi);
27: }
```

We specify the number of server hosts and their names in lines 1 and 2-5, respectively. Line 6 is the port number used, and line 12 divides the number of Monte Carlo trials with the number of servers, determining the number of trials per server. The for loop in lines 14-21 invokes the servers asynchronously: Line 16 generates the URI of the server, line 17 calls the server asynchronously, and line 22 waits for all the servers to finish. Line 23 aggregates the results returned from all the servers.

In this manner, we can easily write a parallel parameter sweep survey program using the task parallel primitives of GridRPC. We next modify the program to perform dynamic load balancing.

Dynamically Load Balancing Multiple Servers

The previous program assumed that the loads of the servers are more or less balanced. In other words, we partitioned the work evenly assuming that each server has equivalent compute power. Such may not be always the case: in a Grid environment, servers are typically heterogeneous; also, for some programs load cannot be evenly distributed in a predictable manner. For those cases, a dynamic load balancing scheme is required.

Although GridRPC systems (both Ninf and Netsolve) offer system level scheduling infrastructure to select lightly loaded servers for load balancing, for dynamically load balancing parallel programs the mechanism may not be entirely appropriate, as the load information may not propagate fast enough to serve the needs of fast, repetitive GridRPC invocations.

Instead, here we employ the asynchronous GridRPC APIs to explicitly balance load. For the above PI example, we further subdivide the load (# of trials) and invoke each server multiple times. Servers that have rapid task execution turnaround automatically gets assigned more computation, achieving overall balance of the load. Below is the modified code that balance the load:

```
 1: #define NUM_HOSTS 16
 2: char * hosts[] =
 3: {"wiz00", "wiz01", "wiz02", "wiz03", "wiz04", "wiz05", "wiz06", "wiz07",
 4:  "wiz08", "wiz09", "wiz10", "wiz11", "wiz12", "wiz13", "wiz14", "wiz15"
 5: };
 6: int port = 4000;
 7: #define DIV 5
 8:
 9: main(int argc, char ** argv){
10:    double pi;
11:    long times, whole_times, count[NUM_HOSTS], sum = 0;
12:    int i, done = 0;
13:    char entry[NUM_HOSTS][100];
14:    int ids[NUM_HOSTS];
15:
16:    whole_times = atol(argv[1]);
17:    times = (whole_times / NUM_HOSTS) / DIV ;
18:    for (i = 0; i < NUM_HOSTS; i++){
19:      sprintf(entry[i], "ninf://%s:%d/pi/pi_trial", hosts[i], port);
20:      if ((ids[i] =
21:           Ninf_call_async(entry[i], rand(), times, &count[i])) == NINF_ERROR){
22:        Ninf_perror("pi_trial");
23:        exit(2);
24:      }
25:    }
26:    while (1) {
27:      int id = Ninf_wait_any();        /* WAIT FOR ANY HOST */
28:      if (id == NINF_OK)
29:        break;
30:      for (i = 0; i < NUM_HOSTS; i++)  /* FIND HOST */
31:        if (ids[i] == id) break;
32:
33:      sum += count[i];
34:      done += times;
35:      if (done >= whole_times)
36:        continue;
37:      if ((ids[i] =
38:           Ninf_call_async(entry[i], rand(), times, &count[i])) == NINF_ERROR){
39:        Ninf_perror("pi_trial");
40:        exit(2);
41:      }
42:    }
43:    pi = 4.0 * ( sum / (double)done);
44:    printf("PI = %f\n", pi);
45:  }
```

In line 7, DIV determines how many times on the average the Ninf call should be made. We also define the array ids[] which holds the current GridRPC session ID for each hosts, and array entry which embodies the URI of each server.

In Line 17, we divide the parameter specified with arg[v] with the # of hosts and DIV. Lines 8-24 perform the initial asynchronous Ninf_calls to each server. Lines 26-42 make a call to a host whose call has just finished. Ninf_wait_any in line 27 returns the session ID of the call that has finished. When there are no precedings asynchronous Ninf_calls that has finished, Ninf_wait_any merely returns NINF_OK. The loop in line 30 computes the server that corresponds to the session ID. Lines 33 and 34 aggregate the result from that server, and line 38 makes the next asynchronous GridRPC call to the

server.

More complex task parallel interactions between the calls are possible, and will be subject of a future tutorial document.

Next

Ninf-G Client API Reference

Ninf-G provides both Grid RPC native API (GRPC API) and Ninf compatible API (Ninf API). Ninf API is a full-compatible API with Ninf version 1 which provides simple and easy programming interface to programmers. GRPC API provides lower level and flexible programming interface to programmers. According to the requirements, application programmers can build Grid-enabled applications using either GRPC API or Ninf API. This section shows the function reference of both GRPC API and Ninf API.

Grid RPC API Reference

Ninf-G client initialization and finalization

Ninf-G uses standard module initializationand finalization. Before any Ninf-G client functions are called, the following function must be called:

grpc_initialize (char *config_file_name);

This function returns GRPC_OK if Ninf-G was successfully initialized, and you are therefore allowed to subsequently call Ninf-G client functions. Otherwise, GRPC_ERROR is returned, and Ninf-G client functions should not be subsequently called. This function may be called multiple times.

To deactivate Ninf-G client, the following function must be called:

grpc_finialize ();

This function should be called once for each time the Ninf-G client was initialized

Ninf-G Handle Functions

int **grpc_function_handle_init** (
grpc_function_handle_t *handle,
char *host_name,
int port,
char *func_name)

This function initializes a handle to Ninf-G function along with hostname, port number and function name. A function handle contains information used in referencing a Ninf-G executable offered by the specified host.

- **handle -** a handle to Ninf-G function
- **host_name -** host name or hostname along with the desired job manager such as ninf.apgrid.org/jobmanager-pbs
- **port** - a port number to be used Currently this function ignores this argument and always set to 0.
- **func_name** - function name to be used

returns GRPC_OK if successfully initialized or GRPC_ERROR if failed

---

int
grpc_function_handle_default(
grpc_function_handle_t *handle,
char *func_name)

This function initializes Request access to interactive resources at the current time. A job request is atomic: either all of the requested processes are created, or none are created.

- **handle** - the contact information about the resource manager to which the request is submitted.
- **func_name** - a RSL description of the requested job

---

int *
grpc_function_handle_destruct (grpc_function_handle_t* handle)
This function destruct the handle and returns the result

- **handle** - the handle to be destructed.

---

grpc_function_handle_t *
grpc_get_handle (int sessionID)
This function returns a handle to the Ninf-G

- **sessionID** - the job_contact of the job in question.

Ninf-G Invocation Functions

int
grpc_call(grpc_function_handle_t *handle, <argument list>)

This function sends a blocking request to Ninf-G. **grpc_call** takes as argument the handle to Ninf-G function and the list of arguments in the calling sequence. If the call is successful, it returns GRPC_OK and the result of the computation is stored in the output arguments. If the call fails, it returns GRPC_ERROR. The output arguments are specified in the Ninf IDL. This function is synchronous in that the client waits until the completion of the computation on the server side.

---

int
grpc_call_async(grpc_function_handle_t *handle, <agument list>)

This function is almost identical to grpc_call except that it returns immediately after all the arguments have been sent. The return value is the session ID of the asynchronous call. The ID is used for various synchronizations such as waiting for the return value of the call.

This function sends a nonblocking request to Ninf-G and returns the job to the caller immediately. This allows the users to invoke multiple sessions simultaneously. **grpc_call_async** takes as argument the handle to Ninf-G function and the list of arguments in the calling sequence. If the call is successful, it returns the session ID and the result of the computation is stored in the output arguments. If the call fails, it returns GRPC_ERROR. The output arguments are specified in the Ninf IDL.

Ninf-G Invocation Functions using an argument stack and related Functions

grpc_arg_stack *
grpc_arg_stack_new(
int size
)
Allocate an argument stack. The integer argument to grpc_new_arg_stack() represents the max size of the stack. Returns a pointer to the allocated stack object if succeeds and NULL if it fails.

---

int
grpc_arg_stack_push_arg(
grpc_arg_stack *argStack,
void *arg
)
Pushes an argument to the stack. Returns GRPC_OK if succeeds and GRPC_ERROR if not.

---

void *
grpc_arg_stack_pop_arg(
grpc_arg_stack *argstack
)

Pops an argument from the stack. Returns pointer to the element if succeeds NULL if some error occured.

---

int
grpc_arg_stack_destruct(
grpc_arg_stack *argStack
)
Free the stack. Returns GRPC_OK if succeeds and GRPC_ERROR if not.

---

int
grpc_call_arg_stack(
grpc_function_handle_t *handle
grpc_arg_stack *argStack
)
Performs blocking call. This function corresponds to the grpc_call and the return values are the same.

---

int
grpc_call_arg_stack_async(
grpc_function_handle_t *handle
grpc_arg_stack *argStack
)
Performs nonblocking call. This function corresponds to the grpc_call_async and the return values are the same.

Ninf-G Session Control

int
grpc_probe(int sessionID)
This function probes the job whose session ID is specified in the argument.
It returns 1 if the job is done and 0 if not.

---

int
grpc_cancel(int sessionID)
This function cancels the job whose session ID is specified in the argument.
It returns GRPC_OK if success and GRPC_ERROR if failed.

Ninf-G Wait Functions

int
grpc_wait(int sessionID)
This function is the most basic call for synchronization, where you wait for the result of the asynchronous call with the supplied session ID. It returns GRPC_OK if the session succeeds and GRPC_ERROR if not.

---

int
grpc_wait_and(int * idArray, int length)
This function waits for a set of sessions specified in *idArray to be terminated. It returns GRPC_OK if all the specified sessions succeed and GRPC_ERROR if not.

---

int
grpc_wait_or (
int * idArray,
int length,
int * idPtr
)
This function waits for any single session in the sessions specified in *idArray to be terminated. It returns GRPC_OK if the finished session succeeded and GRPC_ERROR if not. The third argument idPtr returns the finished session's ID. If the argument length equals to zero, it returns GRPC_OK and set 0 to the idPtr.

---

int
grpc_wait_all()
This function waits for all preceding asynchronous invocations made. It returns GRPC_OK if all the sessions succeed and GRPC_ERROR if not.

---

int
grpc_wait_any(
int *idPtr
)
This function waits for any one of the preceding asynchronous invocations made. It returns GRPC_OK if the finished session succeeded and GRPC_ERROR if not. The argument idPtr returns the finished session's ID. If no previous invocation is left, it returns GRPC_OK and set 0 to the idPtr.

Ninf API Reference

Ninf-G client initialization and finalization

Ninf-G uses standard module initializationand finalization. Before any Ninf-G client functions are called, the following function must be called:

Ninf_parse_args (int argc, char *argv[]);

Ninf_parse_args analyzes the command line arguments and retrieves arguments for Ninf-G system. Arguments for application itself are kept as command line arguments. This function returns new ARGC if Ninf-G was successfully initialized, and you are therefore allowed to subsequently call Ninf-G client functions. ARGV is also renewed to point to the arguments list for the application. Otherwise, and error code (NINF_ERROR) is returned, and Ninf-G client functions should not be subsequently called.

Ninf-G Invacation Functions

int
Ninf_call(char *stub_name, <argument list>)

This function sends a blocking request to Ninf-G. **Ninf_call** takes as argument the name of Ninf-G function and the list of arguments in the calling sequence. If the call is successful, it returns NINF_OK and the result of the computation is stored in the output arguments. If the call fails, it returns NINF__ERROR. The output arguments are specified in the Ninf IDL. This function is synchronous in that the client waits until the completion of the computation on the server side.

---

int
Ninf_call_async(char *stub_name, <agument list>)

This function is almost identical to Ninf_call except that it returns immediately after all the arguments have been sent. The return value is the session ID of the asynchronous call. The ID is used for various synchronizations such as waiting for the return value of the call.

This function sends a nonblocking request to Ninf-G and returns the job to the caller immediately. This allows the users to invoke multiple sessions simultaneously. **Ninf_call_async** takes as argument the handle to Ninf-G function and the list of arguments in the calling sequence. If the call is successful, it returns NINF_OK and the result of the computation is stored in the output arguments. If the call fails, it returns NINF_ERROR. The output arguments are specified in the Ninf IDL.

Ninf-G Session Control

int
Ninf_session_probe(int sessionID)
This function probes the job whose session ID is specified in the argument.
It returns 1 if the job is done and 0 if not.

---

int
Ninf_session_cancel(int sessionID)
This function cancels the job whose session ID is specified in the argument.
It returns NINF_OK if success and NINF_ERROR if failed.

Ninf-G Wait Functions

int
Ninf_wait(int sessionID)
This function is the most basic call for synchronization, where you wait for the result of the asynchronous call with the supplied session ID. It returns NINF_OK if the session succeeds and NINF_ERROR if not.

---

int
Ninf_wait_and(int * idArray, int length)
This function waits for a set of sessions specified in *idArray to be terminated. It returns NINF_OK if all the specified sessions succeed and NINF_ERROR if not.

---

int
Ninf_wait_or (
int * idArray,
int length,
int * idPtr
)
This function waits for any single session in the sessions specified in *idArray to be terminated. It returns NINF_OK if the finished session succeeded and NINF_ERROR if not. The third argument idPtr returns the finished session's ID. If the argument length equals to zero, it returns NINF_OK and set 0 to the idPtr.

---

int
Ninf_wait_all()

This function waits for all preceding asynchronous invocations made. It returns NINF_OK if all the sessions succeed and NINF_ERROR if not.

---

int
Ninf_wait_any(int * idPtr)
This function waits for any one of the preceding asynchronous invocations made. It returns NINF_OK if the finished session succeeded and NINF_ERROR if not. The argument idPtr returns the finished session's ID. If no previous invocation is left, it returns NINF_OK and set 0 to the idPtr.

Ninf-G Error Handling Functions

int
Ninf_perror (char *str)
This function writes error messages pointed to by *str* to standard error.

---

char *
Ninf_error_string (
int error_code
)
This function returns a pointer to error message which corresponds to the specified error code.

---

int
Ninf_get_error (
int sessionID
)
This function returns an error code of the session along with the specified *sessionID* .

---

int
Ninf_get_last_error (
)
This function returns the error code of the last session.

Next

Ninf-G Interface Description Language

The following statements can be used in the NINF-G IDL files.

- `Module` module_name `;`
  This statement specifies the module name. ns_gen will generate a Makefile named .mak.

- `CompileOptions "` .... `";`
  This statement specifies some definition whish should be used in the resulting makefile.

- `FortranFormat "....";`
  Some fortran compiler have strange convention for generating labels from function name. For example a compiler compiles function named 'FFT' into '_FFT_'. Our stub routine uses C, not FORTRAN, so we have to handle these conventions in someway.

  This statement provides translation format. It is similar to the printf format string. You can use following two special characters in the string.

  - `%s` : original function name
  - `%l` : capitalized original function name

  We can cope with almost all strange conventions using these two.

  For example, consider to put under score before and after the function name.

  ```
  FortranFormat "_%s_";
  ```

  With this statement,

  ```
  Calls "Fortran" FFT(n,x,y);
  ```

  will generate function call in C, `_FFT_(n, x, y)`.

- `Library "....";`
  This statement specifies object files and libraries for each functions.

- `Globals {` ... C descriptions ... `}`
  This statement declares global variables shared by all routines.

- `Define` ninf-name `(` paramter1, paramter2, ... `)`
  ["description"]
  [`Required` "files-or-libs"]
  { `{` C descriptions `}` | `Calls` lang-spec function-name `(` arg1,arg2, ... `);`}
  This statement declares function interface and libraries needed by the function.

  For 'lang-spec', you specify the language you implement your numerical library. Currently, you can use 'fortran' of 'C'. 'C' is default. Using this information, the stub generator changes function call sequence. For fortran libraries, the format string specified in FortranFormat statement will be used.

  Each parameter must follow the following syntax.

  [mode-spec] [type-spec] param-name [`[` dimension [`:` range] `]`]+

For 'type-spec', you specify data type for the parameter. You can use, float, double, signed/unsigned char, short, int, long, longlong. NOTE: sizes of these data-types do not depend on platforms. The sizes is specified by XDR, char 1 byte, short 2 byte, int 4 byte, long 4byte, longlong 8byte, float 4byte, double 8byte .

For 'mode-spec', you specify input/output mode: IN, OUT, INOUT, WORK. 'IN' means that the parameter will be transferred from client to server. 'OUT' means the opposite. 'INOUT' means that the parameter will be transferred from client to server firstly, and after the calculation, it will go back to the client. Parameter marked as 'WORK' will not be transferred. Specified memory area will be just allocated on the server side. You can not use 'OUT' for scalar types.

For arrays, you can specify, size, upper-limit, lower-limit, stride. These can be omitted, except for size. For these values, you can use some expressions.

Expression can include constants, other IN-moded scalar parameter in the function definition, and some operators. We provide just 5 operators: +,-,*,/,%. Priority among these operators is same as ANSI C. You can also use parentheses in expressions.

Ninf-G IDL syntax

```
/* program toplevel */
program:  /* empty */
      | declaration_list
    ;

declaration_list:
      declaration
    | declaration_list declaration
    ;

declaration:
      ``Module'' IDENTIFIER ';'
    | ``FortranFormat'' STRING ';'
    | ``CompileOptions'' STRING ';'
    | ``Globals'' globals_body
    | ``Define'' interface_definition opt_string required interface_body
    ;

interface_definition:
        IDENTIFIER '(' parameter_list ')'
    ;

parameter_list:
       parameter
    | parameter_list ',' parameter
    ;

parameter:
       decl_specifier declarator
    ;

decl_specifier:
    type_specifier
    | MODE
    | MODE type_specifier
```

```
    | type_specifier MODE
    | type_specifier MODE type_specifier
    ;

type_specifier:
      TYPE
    | TYPE TYPE
    | TYPE TYPE TYPE    /* ex. unsigned long int */
    ;

declarator:
      IDENTIFIER
    | '(' declarator ')'
    | declarator '['expr_or_null ']'
    | declarator '['expr ':' range_exprs ']'
    | '*' declarator
    ;

range_exprs:
       expr     /* upper limit */
     | expr ',' expr    /* lower limit and upper limit */
     | expr ',' expr ',' expr /* lower, upper and step */

opt_string:
    /* empty */
    | STRING
    ;

language_spec:
    /* empty */
    | ``fortran''
    | ``C''
    ;

required:
    /* empty */
    | ``Required'' STRING
    ;

interface_body:
    '{' /* C statements */ '}'
    | ``Calls'' opt_string IDENTIFIER '(' id_list ')' ';'
    ;

globals_body:
    '{'/* C statements */ '}'
    ;

id_list: IDENTIFIER
    | id_list ',' IDENTIFIER
    ;

/* index description */
expr_or_null:
       expr
    | /* null */
```

```
    ;

expr:
      unary_expr
    | expr '/' expr
    | expr '%' expr
    | expr '+' expr
    | expr '-' expr
    | expr '*' expr
    ;

unary_expr:
      primary_expr
    | '*' expr
    | '-' expr
    ;

primary_expr:
     primary_expr '[' expr ']'
    | IDENTIFIER
    | CONSTANT
    | '('  expr  ')'
    ;

/* TYPE = int, unsigned, char, short, long, long float, double */
/* MODE = mode_in, mode_out, mode_inout, mode_work, IN, OUT, INOUT, WORK*/
/* IDENTIFIER = name */
/* CONSTANT = integer literals •Afloating point literals */
/* STRING = "..." */
```

IDL Sample

sample.idl:

```
Module sample;
Library "-lm";

Globals { int x,y,z; }

Define sin(IN double d, OUT double result[])
" This is test ..."
{
    double sin();
    *result = sin(d);
}

Define mmul(long mode_in int n, mode_in double A[n][n],
    mode_in double B[n][n],
    mode_out double C[n][n])
Required "sample.o"
Calls "C" mmul(n,A,B,C);

Define mmul2(long mode_in int n, mode_in double A[n*n+1-1],
        mode_in double B[n*n+2-3+1],
```

```
        mode_out double C[n*n])
Required "sample.o"
Calls "C" mmul(n,A,B,C);


Define FFT(IN int n,IN int m, OUT float x[n][m], float INOUT y[m][n])
Required "sample.o"
Calls "Fortran" FFT(n,x,y);
```

stub generator output

```
# This file 'pi.mak' was created by ns_gen. Don't edit

CC = cc
include $(NS_DIR)/lib/template

# CompileOptions:


#  Just a hack for compatibility
#  Define NS_COMPILER & NS_LINKER as $(CC) if it is not defined.

# Sorry, it may be necessary to edit this part by hand
NS_COMPILER = $(CC)
NS_LINKER = $(CC)

# stub sources

NS_STUB_SRC = _stub_pi_trial.c

# stub programs

NS_STUB_PROGRAM = _stub_pi_trial

# stub inf files
NS_INF_FILES = _stub_pi_trial.inf

# LDAP dif file
LDAP_DIF = root.ldif

all: $(NS_STUB_PROGRAM) $(NS_INF_FILES) $(LDAP_DIF)

_stub_pi_trial.o: _stub_pi_trial.c
        $(NS_COMPILER) $(CFLAGS) $(NS_CFLAGS) -c _stub_pi_trial.c
_stub_pi_trial: _stub_pi_trial.o pi_trial.o
        $(NS_LINKER) $(CFLAGS) -o _stub_pi_trial _stub_pi_trial.o $(LDFLAGS)
$(NS_STUB_LDFLAGS) pi_trial.o $(LIBS)
_inf_pi_trial: _inf_pi_trial.c
        $(NS_COMPILER) $(CFLAGS) $(NS_CFLAGS)  -o _inf_pi_trial  _inf_pi_trial.c
$(NS_STUB_LDFLAGS)
_stub_pi_trial.inf: _inf_pi_trial
         ./_inf_pi_trial _stub_pi_trial.inf



$(LDAP_DIF): $(NS_INF_FILES)
```

```
        $(NS_DIR)/bin/ns_gen_dif $(NS_STUB_PROGRAM)

install: $(LDAP_DIF)
        $(INSTALL) *.ldif $(LDIF_INSTALL_DIR)

clean:
        rm -f _stub_pi_trial.o _stub_pi_trial.c
        rm -f _inf_pi_trial.c _inf_pi_trial

veryclean: clean
        rm -f $(NS_STUB_PROGRAM) $(NS_INF_FILES) $(LDAP_DIF)
        rm -f pi_trial.o


# END OF Makefile
```

_stub_pi_trial.c

```
/*===============================================================================
Module          :ns_gen
Filename        :_stub_pi_trial.c
RCS             :
        $Source:
        $Revision:
        $Author:
        $Data:Wed Feb 27 16:09:22 2002
        $Locker:
        $state:
==============================================================================*/
#include "nsstb_stub_lib.h"

/* DESCRIPTION:
monte carlo pi computation
*/

static char stub_description[] = "monte carlo pi computation";

struct nslib_expression size20[] = {
          {
                {
                  {VALUE_CONST, 1} ,{VALUE_END_OF_OP, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                }
          }
};

struct nslib_expression start20[] = {
          {
```

```
                {
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                }
          }
};

struct nslib_expression end20[] = {
          {
                {
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                }
          }
};

struct nslib_expression step20[] = {
          {
                {
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                }
          }
};



struct nslib_param_subscript dims2[] = {
        { size20, start20, end20, step20 },
};
```

```
struct nslib_func_param params[] = {
        { DT_INT, MODE_IN, 0, NULL, NULL },
        { DT_LONG, MODE_IN, 0, NULL, NULL },
        { DT_LONG, MODE_OUT, 1, dims2, NULL },
};


nslib_func_info_t nslib_func_info = {
        221,
        {
          "pi",
          "pi_trial"
        },
        3,
        params,
        {
          {
                {
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                  {VALUE_NONE, 0} ,{VALUE_NONE, 0} ,{VALUE_NONE, 0} ,
                }
          },
          (nslib_backend_t)0, /* backend */
          (ns_bool)0, /* ns_bool: specify server side shrink */
          stub_description,
          0  /* information type :fortran or c */
        }
};

/* Globals */

/* Callback proxy */

/* Stub Main program */
main(int argc, char ** argv){
        int seed;
        long times;
        long *count;

        int tmp;

        nsstb_INIT(argc,argv);
        while(1){
                tmp = nsstb_REQ();
                if (!tmp){
                    break;
                }
```

```
                nsstb_SET_ARG(&seed,0);
                nsstb_SET_ARG(×,1);
                nsstb_SET_ARG(&count,2);
                nsstb_BEGIN();
{
  long counter;
  counter = pi_trial(seed, times);
  *count = counter;
}
                tmp = nsstb_END();
                if (!tmp) break;
        }
        nsstb_EXIT();
}
/*                      */
```