Introduction to MPI

Message-passing Programming

Overview

The purpose of this tutorial is to introduce the MPI communication library for message-passing computing. It presents a brief description of the steps necessary to compile and execute an MPI program on the eos network of workstations. In the process, it introduces the basic functionality found in every MPI program.

Configuring the MPI Environment (using CIS workstations as a "virtual cluster")

Most message-passing libraries today use `ssh` in order to provide secure communication between systems. Typically, a tunnel is established between daemons on each machine; the daemons are then responsible for delivering messages reliably to the destination process(es). Since this is meant to be a transparent activity, at least to users, it is necessary to ensure that establishing the `ssh` connection is automatic (i.e. non-interactive).

Perform the following steps:

0. Log into an Architecture lab system (e.g. `arch01`)

1. Run the program `ssh-keygen` to create your RSA SSH Protocol 2 public and private keys.

    - simply accept the defaults. In particular, do *not* enter a passphrase as you want to be able to open an ssh session on a remote machine without having to enter your password each time.
    - this will create a directory called '**.ssh**'. Within that directory, there will be a file called '**id_rsa**', which holds your private key, and another file called '**id_rsa.pub**' that holds your public key.

2. Make a copy of the public key file (**id_rsa.pub**) in a new file called '**authorized_keys**' in the same directory.

3. Now **ssh** into a remote system.

    - this will create a file called '**known_hosts**'. The file holds the keys for each remote machine you log into.
    - if you get a message about "permanently adding this machine" respond '**yes**'. You should only need to do this the first time you log into a machine.
    - Note: if you are denied login to a particular machine, your probably have an out-of-date ssh key. Edit your '**known_hosts**' file to remove the entry associated with the machine you are trying to log into, then repeat the ssh login as if it is the first time.

4. Test your setup by issuing a command such as '`ssh arch02 "echo hello"`' when logged into a different machine. You should receive a message back without having to enter your password.

    - repeat this process (steps 3 and 4) for all machines you want to add to your "virtual cluster".

5. Like many systems, EOS has several versions of MPI installed. A final preparation step is needed to ensure that your program uses the "correct" shared libraries.

Insert the following line into the .bashrc file in your home directory:

```
export LD_LIBRARY_PATH=/usr/lib64/openmpi/lib
```

MPI Programming - Getting Started

The sample program below is the Open MPI version of "Hello, world". If everything works right, each worker process (i.e. rank != 0) should communicate its rank and host machine back to the Master, which prints each message to the screen.

Sample Program - hello.c

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <mpi.h>

#define MASTER  0
#define TAG     0
#define MSGSIZE 100
#define MAX 25

int main(int argc, char* argv[])
{
    int my_rank, source, num_nodes;
    char my_host[MAX];
    char message[MSGSIZE];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_nodes);

    if (my_rank != MASTER) {
        gethostname (my_host, MAX);
        sprintf(message, "Hello from process %d on host %s!", my_rank,
my_host);
        MPI_Send(message, strlen(message) + 1, MPI_CHAR, MASTER, TAG,
MPI_COMM_WORLD);
    }
    else {
        gethostname (my_host, MAX);
        printf ("Num_nodes: %d\n", num_nodes);
        printf ("Hello from Master (process %d) on host %s!\n", my_rank,
my_host);
        for (source = 1; source < num_nodes; source++) {
            MPI_Recv(message, MSGSIZE, MPI_CHAR, source, TAG,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", message);
        }
    }

    MPI_Finalize();
```

```
    return 0;
}
```

Perform the following:

- the Open MPI package must be installed on your system (it is already installed on eos systems), and your execution path and library path must be set correctly.
- compile a C program using '**mpicc**', or a C++ program using '**mpiCC**'. These are MPI wrappers, respectively, for **gcc** and **g++**.

```
mpicc hello.c -o hello
```

- execute the program using the '**mpirun**' command. This will automatically initiate the communication daemons and start execution of the parallel program. The line:

```
mpirun [ -np X ] [ --hostfile <filename> ] <program>
```

  will initiate **X** processes, each running an instance of the executable **<program>**. Note that:

  - the '**-np**' option specifies the number of processes ('X') to execute
  - the '**--hostfile**' option will initiate processes on the hosts listed in the specified file **<filename>**. The hostfile simply lists a machine name on each line (e.g. **arch01**). The OpenMPI run-time system will attempt to start the specified number of processes, using the machines in the hostfile as a "virtual cluster" to run the executable named <program>. You may choose to alter the appropriate line in the hostfile to specify the number of processes to start on each machine (e.g. **arch01 slots=4**). This is typically used to inform the MPI run-time system that arch01 is a quad-core machine; therefore, 4 processes should be started on arch01 before moving on to the next machine. Note that MPI will not (automatically) start more processes than are specified as the total machine slots in the hostfile. To do so, you must use the flag "**--oversubscribe**".
  - there is a utility program (**archup.sh**) posted to the course info page that will query the systems in the Architecture lab and generate a hostfile for you called "my_hosts". Note: you may need to make the utility file executable (i.e. chmod 700 archup.sh). It is recommended that you use the Arch lab machines as your virtual cluster. It is also a courtesy that you do not completely occupy those machines. If you invoke the **archup** utility with the 'phy' option, it will designate 8 slots (processes) per machine, which is theoretically 50% occupied. Note: you can edit/modify the my_hosts file as you see fit, for experimental purposes.
  - alternatively, the '**--host**' option can be used to specify a short list of machines to run processes on. So, for example, the command:

    '**mpirun -np 3 --host arch01,arch02,arch03 hello**'

    will execute the above "hello" program, using three processes, started on three different machines in the CIS labs
  - to configure a specific number of slots per host, you can use the 'colon' notation, as in:

    '**mpirun -np 6 --host arch01:2,arch02:2,arch03:2 hello**'

- the MPI run-time system will automatically shutdown the communication daemons when your parallel program terminates.

Six Fundamental Functions

Although the MPI communication library is very rich (hundreds of functions), many message-passing programs can be written using just a few simple calls. The "Hello, world." program above is an example of a program that implements the common SPMD structure and uses basic communication functionality.

Startup / Termination

These two functions are used to initialize and shutdown an MPI program. They are essentially bookend calls that should go at the start and end of every MPI code. The prototype of the Initialize function is:

```
int MPI_Init(int *argc, char ***argv)
```

The Init routine must be called before any other MPI routine, and must only be called once. It's complementary function has the prototype:

```
int MPI_Finalize(void)
```

Finalize is used to clean up MPI state. It is the user's responsibility to ensure that all pending communications have completed before this call.

Obtaining Info - Assigning tasks to processes

MPI uses a process abstraction. Processes are started automatically, and are location-transparent. Information about the size of a set of communicating processes, and a process's rank within that virtual set, is often used to specify what work a process is to perform. The call to determine the cardinality of the set of communicating processes is:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

The Size function returns the number of processes involved in a communicator (i.e. a set of communicating processes). For **`MPI_COMM_WORLD`** (the set of *all* communicating processes), it indicates the total number of processes available. It is typical to follow the Size call with a Rank call, whose prototype is:

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

The Rank call indicates the rank of the process that calls it, in the range from ***`0... size-1`***, where **`size`** is the return value of the **`MPI_Comm_Size()`** function. Many MPI programs are written using the master-worker model (SPMD); where one process (typically the rank-zero process) assumes a supervisory role, and the other processes serve as compute nodes. In this framework, the two utility calls described here are useful for determining the roles of the various processes of a communicator.

Basic Point-to-Point Communication

The final two important functions have to do with the entire purpose of a message-passing library -- communication. Fundamentally, this means Send and Receive. The prototype for the Send function is:

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm comm)
```

This version of the Send function assumes standard communication mode. It is up to the MPI implementation to

decide whether outgoing messages will be buffered. MPI may buffer outgoing messages. If messages are buffered, the send call may complete before a matching receive is invoked. If the message is not buffered, the send call will not complete until a matching receive has been posted, and the data has been moved to the receiver. The complementary Receive function has prototype:

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm,
MPI_Status *status)
```

These two functions have similar arguments. The Send / Receive buffer consists of the storage containing `count` consecutive elements of the type specified by `datatype`, starting at address `buf`. The endpoints of a point-to-point communication are specified by the `source` and `destination` values. The `tag` is a user-defined identifier that can be used to distinguish between message types. As always, the communicator `comm` defines the set of communicating processes. The Receive function includes an extra `status` parameter, which contains {`source`, `tag`, `error`} information about a received message.

For a Receive, the length of the received message must be less than or equal to the length of the buffer. (An overflow error occurs if all incoming data does not fit into the receive buffer.) If a message that is shorter than the receive buffer arrives, then only those locations corresponding to the (shorter) message are modified. MPI makes some guarantees regarding message delivery:

- If a sender sends two messages in succession to the same destination, and both match the same Receive, then the second message will not be received while the first one is still pending. In other words, message order between processes A and B is preserved.
- If a pair of matching Send / Receive have been initiated on two processes, then at least one of these two operations will complete. In other words, simple messaging will not deadlock.

Basic Collective Communication

OK, so maybe there are 8 fundamental operations. There are two additional and useful routines that take advantage of the fact that there are usually many processes running on many different machines in an MPI environment. They represent the collective versions of the simple Send and Receive.

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm )
```

The Broadcast function is a one-to-many operation. It sends a message from the process with rank **`root`** to all processes of the communicating group, including itself. It should be called by all members of a group using the same argument for root. When this call completes, the contents of root's Send `buffer` will have been copied to all processes. There is also a common form of many-to-one communication, with prototype:

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root,
MPI_Comm comm)
```

The Reduce call, called by all participating members of a group, combines the elements provided in the `sendbuf` buffer of each process in the group, using the operation **`op`**, and returns the combined value in the `recvbuf` buffer of the process with rank root.

There are many other functions available in this rich API. See the online documentation for additional details on these other functions.