6-2018

INTER PROCESS COMMUNICATION BETWEEN TWO SERVERS USING MPICH

Nagabhavana Narla
005386663@coyote.csusb.edu

Follow this and additional works at: http://scholarworks.lib.csusb.edu/etd

Part of the Digital Communications and Networking Commons

Recommended Citation
http://scholarworks.lib.csusb.edu/etd/718

This Project is brought to you for free and open access by the Office of Graduate Studies at CSUSB ScholarWorks. It has been accepted for inclusion in Electronic Theses, Projects, and Dissertations by an authorized administrator of CSUSB ScholarWorks. For more information, please contact scholarworks@csusb.edu.
INTER PROCESS COMMUNICATION BETWEEN
TWO SERVERS USING MPICH

A Project
Presented to the
Faculty of
California State University,
San Bernardino

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Computer Science

by
Nagabhavana Narla
June 2018
INTER PROCESS COMMUNICATION BETWEEN
TWO SERVERS USING MPICH

A Project
Presented to the
Faculty of
California State University,
San Bernardino

by
Nagabhavana Narla
June 2018
Approved by:

Dr. Ernesto Gomez, Advisor, Computer Science and Engineering

Dr. Tong Lai Yu, Committee Member

Dr. Yunfei Hou, Committee Member
ABSTRACT

The main aim of the project is to launch multiple processes and have those processes communicate with each other using peer to peer communication to eliminate the problems of multiple processes running on a single server, and multiple processes running on inhomogeneous servers as well as the problems of scalability. This entire process is done using MPICH which is a high performance and portable implementation of Message Passing Interface standard.

The project involves setting up the password less authentication between two local servers with the help of SSH connection. By establishing a peer to peer communication and by using a unique shell script which is written using MPICH and its derivatives, I am going to demonstrate the process of inter-process communication between the servers.
ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my mentor, supporter, and advisor Dr. Ernesto Gomez for the guidance and encouraging me for completing this project. I would also like to thank my committee members Dr. Tong Lai Yu and Dr. Yunfei Hou for their valuable suggestions and unlimited support.

I would like to thank my Dad Mr. Rambabu Narla, Mom Mrs. Sheela Rani Narla for their unconditional love and support. Special thanks to my sister Miss. Nagabhargavi Narla for always being with me in every situation.
TABLE OF CONTENTS

ABSTRACT .............................................................................................................................. iii
ACKNOWLEDGEMENTS....................................................................................................... iv
LIST OF FIGURES ................................................................................................................ vii

CHAPTER ONE: INTRODUCTION

Background .............................................................................................................................. 1
Purpose ................................................................................................................................. 1

CHAPTER TWO: SYSTEM REQUIREMENT SPECIFICATION

Hardware Requirements ......................................................................................................... 3
Software Requirements ......................................................................................................... 3
Software Used ....................................................................................................................... 3

CHAPTER THREE: HISTORY OF MPICH

MPICH Installation ............................................................................................................... 9
Program Execution Using MPICH ........................................................................................ 15
MPICH Commands ............................................................................................................... 19
MPICH Routines .................................................................................................................. 21

CHAPTER FOUR: SSH CONNECTION BETWEEN THE SYSTEMS .................................. 23

PasswordLess SSH Authentication ....................................................................................... 24
Setting up SSH Keys ............................................................................................................. 24
Launch "n" Processes by Running the Same Program ........................................................... 27

CHAPTER FIVE: PEER TO PEER COMMUNICATION

Communication Between the Processes .............................................................................. 29
Unique Shell Script ............................................................................................................... 30
LIST OF FIGURES

Figure 1. OpenSUSE Control Center Interface......................................................... 4
Figure 2. YaST Control Center................................................................................. 5
Figure 3. Zypper Command Syntax in Terminal....................................................... 5
Figure 4. Zypper Help Command in Terminal............................................................ 6
Figure 5. Working of GCC Compiler ........................................................................ 7
Figure 6. Extracting Files from a Zip Folder.............................................................. 9
Figure 7. Configuring MPICH.................................................................................... 10
Figure 8. Configuring MPICH.................................................................................... 11
Figure 9. MPICH Configuration Completed ............................................................... 12
Figure 10. Building MPICH ...................................................................................... 13
Figure 11. Installing MPICH....................................................................................... 13
Figure 12. Setting Up PATH in Bin Sub Directory ..................................................... 14
Figure 13. Checking MPICH Installation ................................................................. 15
Figure 14. Example Code of CPI.C .......................................................................... 16
Figure 15. Example Code of CPI.C .......................................................................... 17
Figure 16. Example Code of CPI.C .......................................................................... 18
Figure 17. Program Execution Using MPICH ............................................................ 19
Figure 18. Modifying HOSTS.Allow File ................................................................. 23
Figure 19. Setting Up SSH Keys .............................................................................. 25
Figure 20. Password Less Authentication................................................................. 25
Figure 21. Ping Statistics to Check SSH on Ubuntu .................................................. 26
Figure 22. Ping Statistics to Check SSH on OpenSUSE .......................................... 26
Figure 23. Running MPICH Program Ten Times on OpenSUSE .................. 27
Figure 24. Running MPICH Program on OpenSUSE & Ubuntu .................. 28
Figure 25. P2PCOMM.CPP Program Code ........................................... 30
Figure 26. P2PCOMM.CPP Program Code ........................................... 31
Figure 27. P2PCOMM.CPP Program Code ........................................... 32
Figure 28. P2PCOMM.CPP Program Code ........................................... 33
Figure 29. P2PCOMM.CPP Program Code ........................................... 34
Figure 30. P2PCOMM.CPP Program Code ........................................... 35
Figure 31. Running Peer to Peer Communication Program Using MPICH ...... 38
Figure 32. HOSTS.TXT File ............................................................... 54
Figure 33. HOSTS.ALLOW File ............................................................ 56
Figure 34. HOSTS.ALLOW File ............................................................ 57
CHAPTER ONE

INTRODUCTION

Background

The motivation behind the project is when I started learning about the parallelism [3] and the scalability issues in the parallel processing. Working on multiple processes running on a single server is very hard because of the many issues like difficulty of writing code, debugging, managing concurrency, and testing. In this process, the server will be overwhelmed. To avoid this problem, we tend to add more servers, but the problem arises when there is a confusion of which process should be using which server. To avoid this, the processes should start communicating.

Once there is a communication between the processes in the presence of multiple servers, the problem of scalability arises. All these problems like synchronizing overhead, shared process memory space and debugging leads to new research on how to avoid these problems. This project is one small attempt to demonstrate a way to overcome these problems.

Purpose

The purpose of the project is to demonstrate a way of overcoming the problems that occur during multiple processes and single server execution and multiple process & multiple servers executions. By the end of the project, it will show how
using a different approach can overcome the problems of synchronization overhead, debugging and scalability. This project acts as a first step in updating SOS [4] library which is built on MPI which in turn uses network communication between concurrent processes.
CHAPTER TWO

SYSTEM REQUIREMENT SPECIFICATION

Hardware Requirements

• Laptop or PC running LINUX.
• AKEK Cluster at CSUSB and a PC for Development.

Software Requirements

• OpenSUSE Leap 42.2
• MPICH-3.1.4
• GNU C-Compiler.

Software Used

**OpenSUSE**

OpenSUSE [5] is a Linux distribution developed by community supported OpenSUSE project and number of other companies. It is an open source OS. It was previously known as SUSE Linux & SUSE Linux professional. OpenSUSE is very user-friendly and it helps in developing open source software tools for developers and system administrators.

The main features of OpenSUSE - YaST Control Center, Zypper Package Manager & Desktop Innovation.
YaST is the abbreviation for “Yet another Setup Tool”, it handles firewall configuration, network configuration, online updates, package manager & system setup. YaST is classified into two different modules AutoYaST & WebYaST.

Figure 1. OpenSUSE Control Center Interface.
Zypper Package Manager is a powerful package manager engine which resolves the dependency and it is a convenient package management API.

Figure 2. YaST Control Center.

Figure 3. Zypper Command Syntax in Terminal.
OpenSUSE released three different versions of the OS. They are desktop XGL and Compiz, KDE & GNOME.

**MPICH**

MPICH [1] is an elite and generally compact implementation of Message Passing Interface [2] (MPI) Standard. These are used on 9 out of 10 supercomputers in the entire world. The fastest supercomputer Taihu Light also uses MPICH and its derivatives.
GNU C – Compiler

GNU C – Compiler is a compiler system developed for supporting various programming languages. Originally, it’s called GNU C – Compiler as it is used to handle only C applications. Later its name is to GCC as now it supports C, C++, Objective – C, FORTRAN, Ada & Go.

The GCC Compilation system includes different phases. They are:

- Pre-processor
- Compiler
- Optimizer
- Assembler
- Linker

The compiler driver co-ordinates these phases.

The pre-processor stage is used to expand the macros and include the headers files. It can be done by using this command.
Ex: $ cpp hello.c > hello.i

In the compiler phase, the source code is changed to the assembly level language. The command can be as follows:

Ex: $ gcc -Wall -S hello.i

Here – S indicates the gcc to convert the given source code into the assembly level language.

In the optimization phase, the compiler tries to increase or decrease the components of the executable computer program.

The assembler converts the assembly level code to machine understandable code. And then generates an object file. The command can be as follows:

$ as hello.s -o hello.o

The output file is indicated with the option -o.
CHAPTER THREE
HISTORY OF MPICH

MPICH [1] is a small implementation of Message Passing Standard [2] which is used in parallel computing for the applications which use distributed computing. MPICH is an open source software. It was previously known as MPICH2. It is also available in UNIX like operating system.

The part “CH” in MPICH is taken from “Chameleon”. It is a small programming library developed and tested by Willian Group.

MPICH Installation

The MPICH can be installed differently in different operating systems and environments.

Step One

To install MPICH on the local machine it needs tar file of mpich and a C compiler. The tar file will be in the name of mpich-3.1.4.tar.gz. Unpack the tar.gz folder and go to the directory.

```
bhavana@opensuse:~$ tar xzf mpich-3.1.4.tar.gz
bhavana@opensuse:~$ cd mpich-3.1.4
bhavana@opensuse:/mpich-3.1.4$ 
```

Figure 6. Extracting Files from a Zip Folder.
Choose the installation directory. The installation directory is supposed to be empty. It is a good practice, if the directory is shared with all the machines that we run the program on. If not, then it should be copied to all the machines.

**Step Two**

```bash
./configure --prefix=/home/bhavana/mpich-install 2>&1 | tee c.txt
```

Figure 7. Configuring MPICH.

In this step, configure the folder in which mpich exists. In the command below, I enabled socket in ch3 and enabled threads. And disabled Fortran as I am not going to use Fortran in my project. If in case there is a use of Fortran, it can be enabled by simply removing the `--disable-fortran` command.

```bash
./configure --prefix=/home/bhavana/mpich-install --device=ch3:sock --enable-threads=multiple --with-thread-package=pthreads --disable-fortran 2>&1 | tee c1.txt
```

Bourne shells like bash and sh accepts 2>&1, that is the reason I used 2>&1 in the command. If there are any errors in the configuration, the errors are shown clearly in the terminal. By analyzing the errors, we can proceed to the next step. The entire process of configuring the file takes up to two minutes. The below screenshot describes the end of configuring MPICH.
Figure 8. Configuring MPICH.
Figure 9. MPICH Configuration Completed.
Step Three

In this step, we build MPICH, the screenshot below shows the command.

![Screenshot of making MPICH](image)

Figure 10. Building MPICH.

The command in the above screenshot is:

```
Make 2>&1 | tee m.txt
```

Step Four

This step installs the MPICH commands. Executing this step, collects all required executables and scripts in the sub directory called bin. Prefix argument specifies it to configure.

![Screenshot of installing MPICH](image)

Figure 11. Installing MPICH.

The command in the above screenshot is:

```
Make install 2>&1 | tee mi.txt
```
Step Five

This step adds the bin sub directory of the installation to the path in your start up script.

![Figure 12. Setting Up PATH in Bin Sub Directory.](image)

The command in the above screenshot is:

```
PATH=/home/bhavana/mpich-install/bin : $PATH ; export PATH
```

By the end of the step five, the installation is completed. And MPICH is ready to be used.

Step Six

To know if the installation was successful, the below two commands are used.

1. which mpicc
2. which mpiexec

By executing the above commands, we can know that the installation of MPICH is successful or not.
When the above commands are executed in the terminal, it will give the installation PATH of the directory where MPICH was installed. If you get the correct result, it means that the installation was successful.

Program Execution Using MPICH

The running of MPI program can be tested by going into the examples folder of the sub directory. Here I am demonstrating by taking an example cpi.c. It is in /examples/cpi path.

Here is the code for cpi.c.
Example cpi.c

```c
/* -*- Mode: C; c-basic-offset:4 ; indent-tabs-mode:nil; */
/*
 * (C) 2001 by Argonne National Laboratory.
 * See COPYRIGHT in top-level directory.
 */

#include "mpi.h"
#include <stdio.h>
#include <math.h>

double f(double);

double f(double a)
{
    return (4.0 / (1.0 + a*a));
}

int main(int argc, char *argv[])
{
    int n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;

    // Example code here...
}
```

Figure 14. Example Program Code of CPI.C
```c
double startwtime = 0.0, endwtime;
int namelen;
char processor_name[MPI_MAX_PROCESSOR_NAME];

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
MPI_Get_processor_name(processor_name,&namelen);

fprintf(stdout,"Process %d of %d is on %s\n",
    myid, numprocs, processor_name);
fflush(stdout);

n = 10000; /* default # of rectangles */
if (myid == 0)
    startwtime = MPI_Wtime();

MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

h = 1.0 / (double) n;
sum = 0.0;
```

Figure 15. Example Program Code of CPI.C
```c
#include "mpi.h"

double f(double x)
{
    return 4 / (1 + (x * x));
}

int main(int argc, char **argv)
{
    double h = 0.1;
    int n = 10;
    double mypi = 0;

    for (int i = 0; i < n; i++)
    {
        double x = h * (i + 0.5);
        mypi += f(x);
    }

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (myid == 0)
    {
        double endtime = MPI_Wtime();
        printf("pi is approximately %.16f, Error is %.16f\n",
               pi, fabs(pi - PI25DT));
        printf("wall clock time = %.16f\n", endtime - starttime);
        fflush(stdout);
    }

    MPI_Finalize();
    return 0;
}
```

Figure 16. Example Code of CPI.C
MPI program can be executed as follows

```
bhavana@linux-u22l:~/mpich-3.1.4> mpirun -n 5 ./examples/cpi
Process 1 of 5 is on linux-u22l
Process 2 of 5 is on linux-u22l
Process 3 of 5 is on linux-u22l
Process 4 of 5 is on linux-u22l
Process 0 of 5 is on linux-u22l
```

Figure 17. Program Execution Using MPICH.

**MPICH Commands**

**MPICC**

The command mpicc compiles and links the MPI Program. This command provides libraries to compile the program. When linking the program, it provides the necessary libraries from mpicc.

**Command Line Arguments**

- `-show`: Displays the commands that can be used without running them.
- `-help`: Provides help when ever needed.
- `-cc=name`: Displays the compiler name instead of the default name.
- `-config=name`: Returns configuration file for the specific compiler.
- `-echo`: Shows what the program is doing.
Examples

Consider an example file hello.c

- Compiling a single file.
  
  mpicc -c hello.c

- To create an executable and to link the output.
  
  mpicc -o hello hello.o

- Compiling and linking in a single command.
  
  mpicc -o hello hello.c

MPIEXEC

This command runs an MPI program.

Syntax.

mpiexec args executable pgmargs

Here, args is a command line argument, the executable is the name of the program and pgmargs are arguments for executables.

Command Line Arguments

- -n: Specify the number of processes to use.
- -host: Gives hostname of the running program.
- -path: Gives the file of the executable.
- -file: Gives the name of the program.
- -configfile: Displays all the arguments in the program.
MPICH Routines

Routines are used in MPI programs to create communication between two different nodes.

- **MPI_Init**: This routine starts the execution environment. It has two input parameters argc which is a point to the number of arguments and argv which pointer to argument vector. This routine should only be used when there is one thread only.

- **MPI_Comm_size**: It gives the size of the group associated with the communicator. It has two parameters comm which is a communicator and size determines the number of processes in the communication.

- **MPI_Comm_rank**: Gives the rank of calling processes in the communication. It has two input parameters comm which is a communicator and rank which gives the rank of calling processes.

- **MPI_Get_processor_name**: Gives the name of the processor. It has two parameters name which gives the name of the processor and resulteln gives the length of the name of the processor.

- **MPI_Bcast**: This routine broadcast messages from the root processor of the communicator. It has four parameters buffer which starts the address of the buffer, count which gives the number of entries, datatype, root which is a root of the processor, comm which gives a communicator.
- **MPI_Reduce**: Decreases the value of the processor to a single value. It has seven parameters `sendbuf`, `count`, `datatype`, `op`, `root`, `comm` and `recvbuf`.

- **MPI_Wtime**: It gives the completed time on calling the processor.

- **MPI_Finalize**: It ends the MPI execution. It determines all processes should call this routine before exiting.

- **MPI_Comm_world**: MPI_Init defines the routine `MPI_Comm_world` to all processes.

- **MPI_Abort**: Stops the executing program in the middle of the execution. It has two parameters `comm` and `errorcode` which is used to return to invoke environment.

- **MPI_Recv**: It begins non-blocking receive. It has different parameters `buf`, `count`, `datatype`, `source`, `tag`, `comm` and `request`.

- **MPI_Test**: The routine is used when you want to test the request. There are three parameters `request`, `flag`, and `status`.

- **MPI_Send**: It performs a blocking send operation. It has six parameters `buf`, `count`, `datatype`, `dest`, `tag` & `comm`. 
CHAPTER FOUR

SSH CONNECTION BETWEEN THE SYSTEMS

SSH [6] (Secure Shell) is an open source network protocol. It is used to login to the servers remotely. Here I am setting up the SSH connection between two servers so that there is a communication between two servers.

Setting up SSH connection starts off with editing the host files. We need to add the ipaddress of one server host file to another and save it. Also, edit the hosts.allow file and add SSH for the local host and SSH and SSHd for the ipaddress of another server.

```bash
# Example 1: Fire up a mail to the admin if a connection to the printer daemon # has been made from host foo.bar.com, but simply deny all others:
# lpd : foo.bar.com : spawn /bin/echo "%h printer access" | \ # mail -s "tcp_wrappers on %H" root 

# Example 2: grant access from local net, reject with message from elsewhere.
# in.telnetsd : ALL EXCEPT LOCAL : ALLOW
# in.telnetsd : ALL : |
# twist /bin/echo -e "\n\n\n\raccess from %h declined.\n\rGo away.";sleep 2 

# Example 3: run a different instance of rsyncd if the connection comes # from network 172.20.0.0/24, but regular for others:
# rsyncd : 172.20.0.0/255.255.255.0 : twist /usr/local/sbin/my_rsyncd-script
# rsyncd : ALL : ALLOW

SSH : localhost : ALLOW
SSH : 192.168.1.200 : ALLOW
SSHd : 192.168.1.200 : ALLOW
```

Figure 18. Modifying HOSTS ALLOW File.
Passwordless SSH Authentication

In the first step, turn off the fire wall on both machines to make the two servers communicate with each other. To turn off fire wall on OpenSUSE, login to root user and use the following command. This passwordless authentication is internal to the local machine.

```bash
root@opensuse:~# systemctl stop SuSEfirewall2.service
```

Setting Up SSH Keys

After turning off fire wall on both servers, I set up password less login via SSH. Login in to root user and use the following commands.

After executing the command, it asks to set the passphrase for the key. Here I just pressed enter because I want no password to authenticate between the two servers.
After the key is generated, use the following command to finish setting up the passwordless authentication.

```
bhavana@linux-u22l:~> ssh-keygen -t rsa -b 4096
Generating public/private rsa key pair.
Creating directory `/home/bhavana/.ssh`
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in `/home/bhavana/.ssh/id_rsa`.
Your public key has been saved in `/home/bhavana/.ssh/id_rsa.pub`.
The key fingerprint is:
SHA256:IC8hBMG152hTr1Kvppv20pU2aAMeq0VXhw4jvR1g bhavana@linux-u22l
The key's randomart image is:

[Image: A randomart image is shown, with SHA256 fingerprint and key content]
```

Figure 19. Setting Up SSH Keys.

Now I am going to test the connection by logging from one server to another server.

```
bhavana@linux-u22l:~> scp .ssh/id_rsa.pub root@localhost:/root/.ssh/authorized_keys
The authenticity of host 'localhost' can't be established.
ECDSA key fingerprint is SHA256:lgH5YACR8BslbkOhg12TvjuXWYyYD88LjY00e4jdmIY1.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (ECDSA) to the list of known hosts.
Password: 
```

Figure 20. Passwordless Authentication.
Figure 21. Ping Statistics to Check SSH on Ubuntu.

Figure 22. Ping Statistics to Check SSH on OpenSUSE.
In the above screenshot, I tested the SSH connection between OpenSUSE and Ubuntu & OpenSUSE and OpenSUSE.

Launch “n” Processes by Running the Same Program

In the below screenshot, I am running the hello world program script which when executed gives two different parameters. The shell script is written in a way which when executed gives, program name, IP address of the processes and the rank of the processes.

```
bhavana@opensuse:~> mpiexec -n 10 -host opensuse /home/bhavana/mpihello
Hello from IP address: 192.168.17.133, rank 0 / 10 processes
Hello from IP address: 192.168.17.133, rank 1 / 10 processes
Hello from IP address: 192.168.17.133, rank 2 / 10 processes
Hello from IP address: 192.168.17.133, rank 8 / 10 processes
Hello from IP address: 192.168.17.133, rank 3 / 10 processes
Hello from IP address: 192.168.17.133, rank 6 / 10 processes
Hello from IP address: 192.168.17.133, rank 5 / 10 processes
Hello from IP address: 192.168.17.133, rank 7 / 10 processes
Hello from IP address: 192.168.17.133, rank 9 / 10 processes
Hello from IP address: 192.168.17.133, rank 4 / 10 processes
```

Figure 23. Running MPICH Program Ten Times on OpenSUSE.
In the screenshot below, when the same script is executed on two machines using SSH passwordless authentication. It shows IP address of the two computers, program name, rank, and the number of processes.

```
mpiuser@localhost:--> mpiexec -n 10 -host opensuse,ubuntu ./mpiHello
Hello from IP address: 192.168.200.171, rank 1 / 10 processes
Hello from IP address: 192.168.200.171, rank 5 / 10 processes
Hello from IP address: 192.168.200.171, rank 7 / 10 processes
Hello from IP address: 192.168.200.241, rank 4 / 10 processes
Hello from IP address: 192.168.200.241, rank 8 / 10 processes
Hello from IP address: 192.168.200.241, rank 2 / 10 processes
Hello from IP address: 192.168.200.241, rank 0 / 10 processes
Hello from IP address: 192.168.200.171, rank 9 / 10 processes
Hello from IP address: 192.168.200.171, rank 3 / 10 processes
Hello from IP address: 192.168.200.241, rank 6 / 10 processes
mpiuser@localhost:--> 
```

Figure 24. Running MPICH Program on OpenSUSE and Ubuntu.
Communication between the Processes


In this chapter, we will use 2 machines to demo the MPICH program

Machine A

- Hostname: OpenSUSE.
- Create user: mpiuser, home dir: /home/mpiuser/
- IP address: 192.168.200.241 / 24
- Software installed: openSUSE, mpich, mpich-devel, gcc.

Machine B

- Hostname: linux-au3f.
- Create user: mpiuser, home dir: /home/mpiuser/
- IP address: 192.168.200.231 / 24
- Software installed: OpenSUSE, mpich, mpich-devel, gcc.

Please note that we will use both openSUSE machines to simplify the process of testing the program. If you use different OS(s) or different versions of the same OS, we may encounter compatibility problems, which are difficult to solve.
The source code will be “p2pcomm.cpp”, compiled and linked to produce the binary file “p2pcomm”. We will use command mpiexec to spawn 3 processes of the p2pcomm program.

Unique Shell Script

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <mpi.h>
#include <unistd.h>

using namespace std;

int countLineFromFile(string filePath) {
    ifstream dataFile;
    string lineBuf;
    dataFile.open(filePath, ios_base::in);
    if (!dataFile) {
        cerr << "Unable to open file datafile.txt";
        exit(1);  // call system to stop
    }

    int result = 0;
    while (getline(dataFile, lineBuf)) {
        cout << lineBuf << endl;
        result += 1;
    }
}
```

Figure 25. P2PCOMM.CPP Program Code.
```cpp
int main (int argc, char *argv[]) {
    if(argc <= 1) {
        cerr << "Usage: p2pcomm <filePath>" << endl;
        exit(1);
    }

    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Find out rank, size
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // We are assuming at least 2 processes for this task
    if(world_size < 2) {
        cerr << "World size must be greater than 1 for " << argv[0] << endl;
    }
```

Figure 26. P2PCOMM.CPP Program Code.
Figure 27. P2PCOMM.CPP Program Code.
Figure 28. P2PCOMM.CPP Program Code.

```cpp
if(flag != 0) {
    MPI_Irecv(&number, 1, MPI_INT, MPI_ANY_SOURCE, 0,
        MPI_COMM_WORLD, &request);
    // Set the flag = 0 so we know we have a socket listening
    flag = 0;
}

// Test if the connection has completed. If so, flag will be set to 1
MPI_Test(&request, &flag, &status);

// If the connection has been completed
if (flag == 1) {
    // Add it up
    result += number;

    // Increase the counter of returned node
    recvCount += 1;

    // Reset the retry counter
    retryCount = 0;
    cout << "Received: " << number << " from node: " << status.
```
MPI_SOURCE << endl;

// If all the node has returned value, quit and print the sum of values
// This is the normal case
if (recvCount >= world_size - 1) {
  break;
}

// Or if we have retried enough, quit and still print the sum of values, and the number of returned nodes
// This is an abnormal case, when a process is dead prematurely
else if (retryCount > RETRYMAX) {
  break;
}

// If the connection is not completed yet, we wait, and increase the retry counter
retryCount += 1;
usleep(WAITTIME);
```cpp
retryCount += 1;
usleep(WAITTIME);
}

cout << "Received from " << recvCount << " / " << world_size - 1 << " nodes. Total = " << result << endl;
}
//These are the working slaves, do all the work and send the result to master
else {
    int lineCount = 0;
    lineCount = countLineFromFile(argv[1]);
    //cout << "Number of lines: " << lineCount << endl;
    MPI_Send(&lineCount, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}

MPI_Finalize();
```
Explanation

The reason I am trying to launch the multiple copies of the program with a with TCP/IP [8] communication using peer to peer servers is that if we have multiple processes and a single server that is providing information to all other processes the server will be overwhelmed. To eliminate the problem, we need to add more servers.

If we add more servers, we get an issue that which process should be using which server because there is no communication. So, the servers should start communicating. Once the servers start to communicate the problem of scalability arises. This is the reason processes should start communicating with each other.

For this reason, I wrote a shell script which allows the processes to start communicating. And I tested the script with the help of a small data transfer file on both machines.

The source code will be “p2pcomm.cpp”, compiled and linked to produce the binary file “p2pcomm”. We will use command mpiexec to spawn 3 processes of the p2pcomm program. This command will be executed in machine A:

```
mpiuser@opensuse:~> mpiexec -n 3 -hosts opensuse,opensuse2/home/mpiuser/p2pcomm /home/mpiuser/test
```

The mpihello program will print this message:

- Received: xxx from node: 1
- Received: xxx from node: 2
Before executing the program, we need to compile and execute the C++ program to demonstrate the processes.

The source code is attached: p2pcomm.cpp. Compile and link the source code:

- Copy the source code to opensuse machine, `/home/mpiuser`, then execute this command:

```
mpiuser@opensuse:~>mpic++ -o p2pcomm p2pcomm.cpp
```

- Copy the source code to opensuse machine, `/home/mpiuser`, then execute this command:

```
mpiuser@opensuse2:~>mpic++ -o p2pcomm p2pcomm.cpp
```

**NOTE:** You must compile the source code in each machine to make it run properly. If you just copy the binary from this machine to another, the binary may not run, because of dynamic library linking problems.

After that, there will be a file named “p2pcomm” in `/home/mpiuser` of both machines. Set the file to execution mode:

```
mpiuser@opensuse:~>chmod u+x mpihello
mpiuser@opensuse2:~>chmod u+x mpihello
```

Now everything is set, we run this command in machine A (opensuse), using mpiuser:

```
mpiuser@opensuse:~>mpiexec -n 3 -hosts opensuse,opensuse2
/home/mpiuser/p2pcomm /home/mpiuser/test.txt
```

The output will be as follows
As you have seen the command line, we spawn 3 processes, in which process 0 will be the master, process 1 and 2 will be the slaves. Each slave will read the /home/mpiuser/test.txt file (different in each machine), count the number of lines, then send the result back to the master. The master then sums all the results to print out the final result.

In the following example

- The first test.txt file contains 4 lines, so the node 2 will send 4 to master:

  Received: 4 from node: 2

- The second file contains 2 lines, so the node 1 will send 2 to master:

  Received: 16 from node: 1

- Lastly, the master sum them up and report the total result:

  Received from 2/2 nodes. Total = 6.
CHAPTER SIX

CONCLUSION

I conclude that this project addresses the problems faced when working with multiple processes running on a single server, and multiple processes running on multiple inhomogeneous servers as well as the problems of scalability. This project provides a way to deal with all the above problems by setting up SSH connection between two passwordless authenticated servers by establishing a peer to peer communication and by using a unique shell script which is written using MPICH and its derivatives.

Note

AKEK was obtained under US Army Research Lab Agency program AHPCRC. This is a Federal type grant. It was purchased on 5/1/2011.
APPENDIX A

MPIHELLO.C
#define _GNU_SOURCE     /* To get defns of NI_MAXSERV and NI_MAXHOST */

#include <sys/types.h>

#include <sys/socket.h>

#include <ifaddrs.h>

#include <arpa/inet.h>

#include <netdb.h>

#include <stdio.h>

#include <string.h>

#include <mpi.h>

#include <stdlib.h>

void getlocalhostipaddr(char *ipaddr)
{

    struct ifaddrs *ifaddr, *ifa;

    int family, s, n;

    char host[NI_MAXHOST];
char loopback_inf[3];

strcpy(loopback_inf, "lo");

if (getifaddrs(&ifaddr) == -1) {
    perror("getifaddrs");
    exit(EXIT_FAILURE);
}

/* To maintain the first pointer, we need to go through the linked list, so that the list will be free in the later time*/

for (ifa = ifaddr, n = 0; ifa != NULL; ifa = ifa->ifa_next, n++) {
    if (ifa->ifa_addr == NULL || strcmp(ifa->ifa_name, "lo") == 0)
        continue;

    family = ifa->ifa_addr->sa_family;

    /* display the names of the family and interfaces */

    //printf("%-8s\n", ifa->ifa_name);

    /* display the address */

    if (family == AF_INET) {

        

    }
s = getnameinfo(ifa->ifa_addr, sizeof(struct sockaddr_in),

host, NI_MAXHOST, NULL, 0,
NI_NUMERICHOST);

if (s != 0) {
    printf("getnameinfo() failed: %s\n", gai_strerror(s));
    exit(EXIT_FAILURE);
}

strcpy(ipaddr, host);
break;
}
}
freeifaddrs(ifaddr);
}

int main(int argc, char** argv) {
    char ipaddr[NI_MAXHOST];

    // Initialize the MPI environment

MPI_Init(NULL, NULL);

// Get the number of processes

int world_size;

MPI_Comm_size(MPI_COMM_WORLD, &world_size);

// Get the rank of the process

int world_rank;

MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

// Get the name of the processor

getlocalhostipaddr(ipaddr);

// Print off a hello world message

printf("Hello from IP address: %s, rank %d / %d processes\n",
        ipaddr, world_rank, world_size);

// Finalize the MPI environment.

MPI_Finalize();
APPENDIX B

P2PCOMM.CPP
```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <mpi.h>
#include <unistd.h>

using namespace std;

int countLineFromFile(string filePath) {
    ifstream dataFile;
    string lineBuf;
    dataFile.open(filePath, ios_base::in);
    if (!dataFile) {
        cerr << "Unable to open file Filepath";
        exit(1);  // call system to stop
    }
    int result = 0;
    while (getline(dataFile, lineBuf)) {
        result++;
    }
    return result;
}
```
cout << lineBuf << endl;

result += 1;

}

dataFile.close();

return result;

}

int main (int argc, char *argv[]) {

if(argc <= 1) {

cerr << "Usage: p2pcomm <filePath>" << endl;

exit(1);

}

// Initialize the MPI environment

MPI_Init(NULL, NULL);

// Find out rank, size

int world_rank;

MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
int world_size;

MPI_Comm_size(MPI_COMM_WORLD, &world_size);

// We are assuming at least 2 processes for this task

if (world_size < 2) {

cerr << "World size must be greater than 1 for " << argv[0] << endl;

MPI_Abort(MPI_COMM_WORLD, 1);
}

cout << "Program name: " << argv[0] << endl;

cout << "Process MPI id: " << world_rank << endl;

// This is the master

if (world_rank == 0) {

    int recvCount = 0;

    int retryCount = 0;

    int flag = -1;

    const int RETRYMAX = 3;

    const int WAITTIME = 1000000; // Microseconds
MPI_Status status;

MPI_Request request;

int result = 0;

int number;

while (true) {

    // If we has no waiting receiving, flag will be != 0. And we create a
    // listening socket

    if(flag != 0) {

        MPI_Irecv(&number, 1, MPI_INT, MPI_ANY_SOURCE, 0,
        MPI_COMM_WORLD, &request);

        // Set the flag = 0 so we know we have a socket listening

        flag = 0;
    }

    // Test if the connection has completed. If so, flag will be set to 1

    MPI_Test(&request, &flag, &status);
}
//If the connection has been completed
if (flag == 1) {
    //Add it up
    result += number;
    //Increase the counter of returned node
    recvCount += 1;
    //Reset the retry counter
    retryCount = 0;
    cout << "Received: " << number << " from node: " << status.MPI_SOURCE << endl;
}

//If all the node has returned value, quit and print the sum of values
//This is the normal case
if(recvCount >= world_size - 1){
    break;
}
//Or if we have retried enough, quit and still print the sum of values,
and the number of returned nodes

//This is an abnormal case, when a process is dead prematurely

else if (retryCount > RETRYMAX) {

    break;

}

//If the connection is not completed yet, we wait, and increase the retry counter

retryCount += 1;

usleep(WAITTIME);

}

cout << "Received from " << recvCount << " / " << world_size - 1 << " nodes. Total = " << result << endl;

}

//These are the working slaves, do all the work and send the result to master

else {


int lineCount = 0;

lineCount = countLineFromFile(argv[1]);

//cout << "Number of lines: " << lineCount << endl;

MPI_Send(&lineCount, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);

}

MPI_Finalize();
At the end of the hosts.txt file, add the ipaddress of the local servers that you want to give permission to access and save hosts.txt file.

Host file editing should be done on all the servers on which you want to perform communication.

192.168.17.135 linux-au3f

192.168.17.133 opensuse

![hosts.txt file content]

Figure 32. HOSTS.TXT File.
APPENDIX D

HOSTS.ALLOW
In every Linux system, there is a host.allow file in /home/bin/etc folder. Editing the hosts.allow file and adding the below four lines of code, gives the servers the ability to perform SSH communication locally and also through the network.

SSH : localhost : ALLOW

SSHd : localhost : ALLOW

SSH : 192.168.17.133 : ALLOW

SSHd : 192.168.17.133 : ALLOW

Figure 33. HOSTS.ALLOW File.
Figure 34. HOSTS.ALLOW File.
APPENDIX E

TEST.TXT
#This file is purely for the testing purpose.

#This file contains six lines of text

One

Two

Three

Four

Five

Six
APPENDIX F

MPICH
This appendix determines the installation steps of the MPICH on local PC.

Step 1:

Configuring MPICH

./configure –prefix=/home/bhavana/mpich-install –with-device=ch3:sock –
enable-threads=multiple –with-thread-package=pthreads –disable-fortran
2>1 | tee c1.txt

Step 2:

Building MPICH

Make 2>&1 | tee m.txt

Step 3:

Installing MPICH

Make install 2>&1 | tee mi.txt

Step 4:

Setting up PATH in Bin Sub Directory

PATH=/home/bhavana/mpich-install/bin:$PATH; export PATH
REFERENCES


[4] Introducing Non-Determinism to the Parallel C Compiler. Retrieved from-
http://scholarworks.lib.csusb.edu/etd/22/.

https://www.opensuse.org/.

https://en.opensuse.org/SDB:Configure_openSSH.
