

Configuración de clústers con MPI

Yahaira Valeria Gomez Sucasaca

Computación Paralela y Distribuida
Departamento de Ciencia de la Computación
Universidad Católica San Pablo
Email: yahaira.gomez@ucsp.edu.pe

15 de abril de 2022

1. Introduction

MPI es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de paso de mensajes diseñada para ser usada en programas que exploten la existencia de múltiples procesadores. En este informe mostraré cómo se realiza la configuración de un clúster con MPICH2 y nfs a través de una red LAN usando una varias máquinas virtuales (1 master y dos clientes) con sistema Linux Ubuntu.

2. Requisitos previos

1. Instalar MPI en cada una de las máquinas
 - Instalamos con la intrucción: **sudo apt-get install openmpi-bin**
 - Verificamos la instalación con **mpirun --version**
2. Funcion de SSH: Esta herramienta permite la comunicación remota entre nuestro clúster. Los mensajes serán encriptados y, por ende, necesitaremos compartir la clave con nuestros clientes para así ya no hacer la verificación de la contraseña de cada usuario.
 - Instalamos con **sudo apt-get install openssh-server**

3. Instalar NFS NFS (Network File System) es un protocolo que permite acceso remoto a un sistema de archivos a través de la red. Permite a un usuario de equipo ver, almacenar y actualizar archivos en un equipo remoto como si estuvieran en el propio equipo del usuario. Como mencionamos antes, para esta prueba tendremos 1 máquina maestra (master) y 1 máquina cliente1. Entonces instalamos NFS en las máquinas de la siguiente manera:
- En el master lo instalamos con la intrucción: **sudo apt-get install nfs-kernel-server**
 - En las máquinas cliente con **sudo apt-get install nfs-common**

3. Configuración del clúster

Las computadoras usarán SSH para comunicarse entre sí por medio del usuario mpiuser. Cada máquina tendrá una clave que verifique que es una máquina confiable (clase SSH), esto para que no necesitemos de contraseñas para el inicio de sesión por SSH.

3.1. Configuración en el host

Para el protocolo SSH se usará openssh: Cabe aclarar que, se modificó el archivo hosts en todas las máquinas para asociarles un nombre a cada IP de las máquinas del clúster.

Ejecutamos la siguiente instrucción **sudo nano /etc/hosts** y modificamos el archivo como se ve en la Figura 1 para facilitar las configuraciones, tanto en la máquina máster y cliente1.

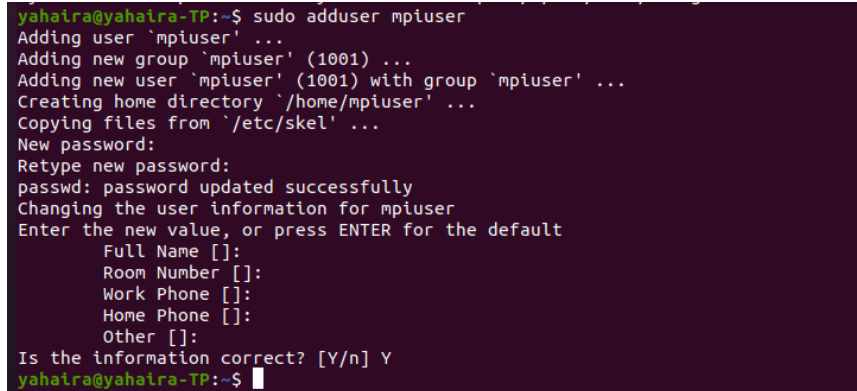
GNU nano 4.8		/etc/hosts
127.0.0.1	localhost	
127.0.1.1	yahaira-TP	
192.168.0.26	master	
192.168.0.25	cliente1	

Figura 1: Archivo /etc/hosts de máster y cliente1

3.2. Creando un nuevo usuario

Aunque puede operar su clúster con su cuenta de usuario existente, se recomienda crear nuevos usuarios para mantener las configuraciones simples.

Creamos un nuevo usuario llamado `mpiuser` tanto en el máster como cliente1, tal y como se ve en la Figura 2.



```
yahaira@yahaira-TP:~$ sudo adduser mpiuser
Adding user 'mpiuser' ...
Adding new group 'mpiuser' (1001) ...
Adding new user 'mpiuser' (1001) with group 'mpiuser' ...
Creating home directory '/home/mpiuser' ...
Copying files from '/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for mpiuser
Enter the new value, or press ENTER for the default
  Full Name []:
  Room Number []:
  Work Phone []:
  Home Phone []:
  Other []:
Is the information correct? [Y/n] Y
yahaira@yahaira-TP:~$
```

Figura 2: Creando un nuevo usuario `mpiuser`

3.3. Configuración SSH

En la sección de Requisitos previos ya habíamos descargado la librería `openssh-server`. Ahora solo tenemos que entrar desde el usuario `mpiuser` (haciendo uso de la instrucción `su - mpiuser`). Luego, dado que el `sshservidor` ya está instalado, podemos iniciar sesión en otras máquinas mediante `ssh mpiuser@192.168.0.26` en máster y `mpiuser@192.168.0.25` en cliente1

Una vez que entremos generamos las claves para la comunicación entre máquinas. Ejecute `ssh-keygen -t rsa` para la generación de claves, tal y como muestra la Figura 3.

Luego copiamos las claves de las máquinas con la instrucción `ssh-copy-id master`

3.4. Configuración de NFS

- MASTER: Ya descargamos el paquete en la sección 1, así que ahora solo queda crear una carpeta con el nombre `cloud` que compartiremos en la red (`mkdir cloud`)

```

mpiuser@yahaira-TP: ~
Generating public/private rsa key pair.
Enter file in which to save the key (/home/mpiuser/.ssh/id_rsa): passw_clus.txt

Enter passphrase (empty for no passphrase):

[1]+  Stopped                  ssh-keygen -t rsa
mpiuser@yahaira-TP:~$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/mpiuser/.ssh/id_rsa):
Created directory '/home/mpiuser/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/mpiuser/.ssh/id_rsa
Your public key has been saved in /home/mpiuser/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:EWmYLG07W/GrfjEVd+aMDBrjWRmD5ouiacAm1nBqHNw mpiuser@yahaira-TP
The key's randomart image is:
+---[RSA 3072]-----+
|      . 0.000      |
|    + + B.=.. 0   |
| . . . + =.* = *   |
| + E. . =.. 0 0    |
| o * . +So         |
| oB .o o =         |
|=. o + +          |
| + . o            |
| . .oo            |
+---[SHA256]-----+

```

Figura 3: Generación de claves

Ahora exportaremos el directorio cloud, así que editaremos el archivo `/etc/exports` y colocaremos `/home/mpiuser/cloud *(rw,sync,no_root_squash,no_subtreecheck)`

Ejecutamos `exportfs -a` y reiniciamos el servidor con `sudo service nfs-kernel-server restart`

- CLIENT1: Creamos el mismo directorio, montamos el directorio compartido con `sudo mount -t nfs master:/home/mpiuser/cloud /cloud`

Y listo, ya tenemos la configuración hecha.

3.5. Código de ejemplo

En este caso ejecutaré uno de los ejemplos de la carpeta de MPICH que descargamos, el conocido `Hola Mundo`; para esto creamos un nuevo archivo con `mpicc -o hello hello.c`

Copio el ejecutable en mi directorio cloud y ejecuto `mpirun -np 15 -hosts client1,192.168.0.26 ./hello`. El resultado se ve en la Figura 4.

```

mpiuser@yahaira-TP:~/cloud$ mpirun -np 15 -hosts client1,192.168.0.26 ./hello
Hello world from process 5 of 15
Hello world from process 1 of 15
Hello world from process 9 of 15
Hello world from process 11 of 15
Hello world from process 3 of 15
Hello world from process 13 of 15
Welcome to Ubuntu process 7 of 15
Hello world from process 4 of 15
Hello world from process 8 of 15
Hello world from process 10 of 15
Hello world from process 6 of 15
Hello world from process 12 of 15
Hello world from process 14 of 15
Hello world from process 0 of 15
Hello world from process 2 of 15

```

Figura 4: Ejecución de programa hello.c

4. Odd-even sort

Es una variación de bubble-sort. Este algoritmo se divide en dos fases: fase impar y par. El algoritmo se ejecuta hasta que se ordenan los elementos de la matriz y en cada iteración ocurren dos fases: Fases pares e impares. En la fase impar, realizamos una ordenación de burbuja en elementos indexados impares y en la fase par, realizamos una ordenación de burbuja en elementos indexados pares.

4.1. Código del programa

```

#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

//FUNCIÓN PARA DIVIDIR LA LISTA DE ENTEROS
int merge(double *ina, int lena, double *inb, int lenb, double *out) {
    int i,j;
    int outcount=0;

    for (i=0,j=0; i<lena; i++) {
        while ((inb[j] < ina[i]) && j < lenb) {
            out[outcount++] = inb[j++];
        }
        out[outcount++] = ina[i];
    }
    while (j<lenb)

```

```

        out[outcount++] = inb[j++];

    return 0;
}

//REALIZANDO LA DIVISIÓN
int domerge_sort(double *a, int start, int end, double *b) {
    if ((end - start) <= 1) return 0;

    int mid = (end+start)/2;
    domerge_sort(a, start, mid, b);
    domerge_sort(a, mid, end, b);
    merge(&(a[start]), mid-start, &(a[mid]), end-mid, &(b[start]));
    for (int i=start; i<end; i++)
        a[i] = b[i];

    return 0;
}

int merge_sort(int n, double *a) {
    double b[n];
    domerge_sort(a, 0, n, b);
    return 0;
}

//IMPRIMO LA EL ESTADO DE ORDENACIÓN DE LA LISTA
void printstat(int rank, int iter, char *txt, double *la, int n) {
    printf("[%d] %s iter %d: <", rank, txt, iter);
    for (int j=0; j<n-1; j++)
        printf("%6.3lf,", la[j]);
    printf("%6.3lf>\n", la[n-1]);
}

// FUNCIÓN EN LA QUE DIVIDO EL TRABAJO, ENVÍO Y RECIBO DATOS CON MPI
void MPI_Pairwise_Exchange(int localn, double *locala,
int sendrank, int recvrank, MPI_Comm comm) {

    // TENDREMOS DOS RANGOS PARA EL ENVÍO Y RECEPCIÓN DE RESULTADOS
    // RANGO DE ENVÍO: SOLO ENVÍA LA LISTA DE ENTEROS
    // RANGO DE RECEPCIÓN: RECIBE LA LISTA, ORDENA Y DEVUELVE LA MITAD DE LA LISTA

```

```

    int rank;
    double remote[localn];
    double all[2*localn];
    const int mergetag = 1;
    const int sortedtag = 2;

    MPI_Comm_rank(comm, &rank);
    if (rank == sendrank) {
        MPI_Send(locala, localn, MPI_DOUBLE, recvrank, mergetag,
                 MPI_COMM_WORLD);
        MPI_Recv(locala, localn, MPI_DOUBLE, recvrank, sortedtag,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    } else {
        MPI_Recv(remote, localn, MPI_DOUBLE, sendrank, mergetag,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        merge(locala, localn, remote, localn, all);

        int theirstart = 0, mystart = localn;
        if (sendrank > rank) {
            theirstart = localn;
            mystart = 0;
        }
        MPI_Send(&(all[theirstart]), localn, MPI_DOUBLE, sendrank,
                 sortedtag, MPI_COMM_WORLD);
        for (int i=mystart; i<mystart+localn; i++)
            locala[i-mystart] = all[i];
    }
}

// ORDENAMIENTO DE LISTA DE IMPARES
int MPI_OddEven_Sort(int n, double *a, int root, MPI_Comm comm)
{
    int rank, size, i;
    double *local_a;

    // OBTENGO EL RANGO Y TAMAÑO DE MPI_Comm
    MPI_Comm_rank(comm, &rank); //&rank = address of rank
    MPI_Comm_size(comm, &size);

```

```

    local_a = (double *) calloc(n / size, sizeof(double));

// DISPERSO LA LISTA local_a EN CADA MÁQUINA
    MPI_Scatter(a, n / size, MPI_DOUBLE, local_a, n / size, MPI_DOUBLE,
        root, comm);
// ORDENO LA LISTA
    merge_sort(n / size, local_a);

// EMPIEZO A ORDENAR (HAGO USO DE LA FUNCIÓN MPI_Pairwise_Exchange)
// OPERO EN CADA MÁQUINA
    for (i = 1; i <= size; i++) {

        printstat(rank, i, "antes de", local_a, n/size);

        if ((i + rank) % 2 == 0) { // means i and rank have same nature
            if (rank < size - 1) {
                MPI_Pairwise_Exchange(n / size, local_a, rank, rank + 1, comm);
            }
        } else if (rank > 0) {
            MPI_Pairwise_Exchange(n / size, local_a, rank - 1, rank, comm);
        }

    }

    printstat(rank, i-1, "despues de", local_a, n/size);

// ENVÍO DATOS
    MPI_Gather(local_a, n / size, MPI_DOUBLE, a, n / size, MPI_DOUBLE,
        root, comm);

    if (rank == root)
        printstat(rank, i, " all done ", a, n);

    return MPI_SUCCESS;
}

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

    MPI_Init(&argc, &argv);

```



```

    int n = argc-1;
    double a[n];
    for (int i=0; i<n; i++)
        a[i] = atof(argv[i+1]);

    MPI_OddEven_Sort(n, a, 0, MPI_COMM_WORLD);

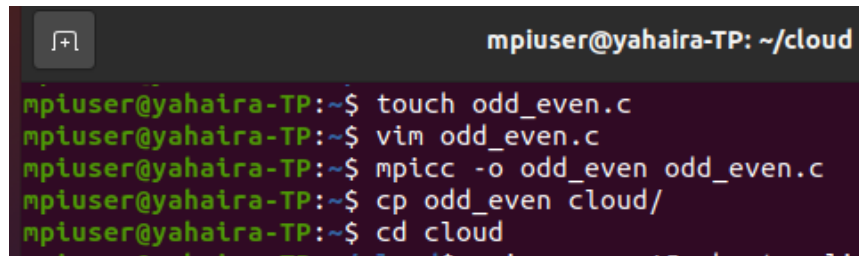
    MPI_Finalize();

    return 0;
}

```

4.2. Resultados de la prueba

Ejecutamos los comandos que se observan en la Figura 5 para correr el programa



```

mpluser@yahaira-TP: ~/cloud
mpluser@yahaira-TP:~$ touch odd_even.c
mpluser@yahaira-TP:~$ vim odd_even.c
mpluser@yahaira-TP:~$ mpicc -o odd_even odd_even.c
mpluser@yahaira-TP:~$ cp odd_even cloud/
mpluser@yahaira-TP:~$ cd cloud

```

Figura 5: Comando para ejecución de programa odd_even.c

- Con 3 máquinas en paralelo
- Con 6 máquinas en paralelo

Como se ve en las Figuras 6 y 7 el programa muestra como cada máquina va ordenando la lista de enteros que ingresamos, es claro que a medida que aumentemos la cantidad de máquinas en paralelo el resultado será más rápido, ya que habrá mejor distribución de tareas.

```

mpluser@yahaira-TP:~/cloud$ mpirun -np 3 -hosts client1,192.168.0.26 ./odd_even 43 54 63 28 79 81 32 47 84 17 25 49
[1] antes de iter 1: <[0] antes de iter 1: <32.000,47.000,79.000,81.000>
[2] antes de iter 1: <28.000,43.000,54.000,63.000>
[0] antes de iter 2: <28.000,43.000,54.000,63.000>
17.000,25.000,49.000,84.000>
[1] antes de iter 2: <[2] antes de iter 2: <17.000,25.000,32.000,47.000>
[1] antes de iter 3: <43.000,47.000,54.000,63.000>
[0] antes de iter 3: <17.000,25.000,28.000,32.000>
[0] despues de iter 3: <17.000,25.000,28.000,32.000>
49.000,79.000,81.000,84.000>
[1] despues de iter 3: <43.000,47.000,49.000,54.000>
[2] antes de iter 3: <49.000,79.000,81.000,84.000>
[2] despues de iter 3: <63.000,79.000,81.000,84.000>
[0] all done iter 4: <17.000,25.000,28.000,32.000,43.000,47.000,49.000,54.000,63.000,79.000,81.000,84.000>
mpluser@yahaira-TP:~/cloud$

```

Figura 6: Ejecución de programa odd_even.c con 3 máquinas en paralelo

```

mpluser@yahaira-TP:~/cloud$ mpirun -np 6 -hosts client1,192.168.0.26 ./odd_even 43 54 63 28 79 81 32 47 84 17 25 49
[3] antes de iter 1: <[2] antes de iter 1: <32.000,47.000>
79.000,81.000>
[4] antes de iter 1: <17.000,84.000>
[4] antes de iter 2: <[0] antes de iter 1: <43.000,54.000>
[0] antes de iter 2: <43.000,54.000>
[5] antes de iter 1: <25.000,49.000>
[5] antes de iter 2: <25.000,49.000>
47.000,84.000>
[5] antes de iter 3: <[4] antes de iter 3: <[3] antes de iter 2: <17.000,32.000>
25.000,47.000>
49.000,84.000>
[5] antes de iter 4: <49.000,84.000>
[1] antes de iter 1: <28.000,63.000>
[2] antes de iter 2: <79.000,81.000>
[1] antes de iter 2: <28.000,63.000>
[1] antes de iter 3: <54.000,63.000>
[3] antes de iter 3: <79.000,81.000>
[3] antes de iter 4: <[4] antes de iter 4: <79.000,81.000>
25.000,47.000>
[0] antes de iter 3: <28.000,43.000>
[0] antes de iter 4: <28.000,43.000>
[5] antes de iter 5: <81.000,84.000>
[5] antes de iter 6: <81.000,84.000>
[2] antes de iter 3: <17.000,32.000>
[2] antes de iter 4: <54.000,63.000>
[3] antes de iter 5: <54.000,63.000>
[1] antes de iter 4: <17.000,32.000>
[1] antes de iter 5: <32.000,43.000>
[4] antes de iter 5: <49.000,79.000>
[4] antes de iter 6: <63.000,79.000>
[2] antes de iter 5: <25.000,47.000>
[2] antes de iter 6: <43.000,47.000>
[5] despues de iter 6: <81.000,84.000>
[4] despues de iter 6: <63.000,79.000>
[3] antes de iter 6: <49.000,54.000>
[3] despues de iter 6: <49.000,54.000>
[0] antes de iter 5: <17.000,28.000>
[0] antes de iter 6: <17.000,28.000>
[1] antes de iter 6: <[2] despues de iter 6: <43.000,47.000>
25.000,32.000>
[1] despues de iter 6: <28.000,32.000>
[0] despues de iter 6: <17.000,25.000>
[0] all done iter 7: <17.000,25.000,28.000,32.000,43.000,47.000,49.000,54.000,63.000,79.000,81.000,84.000>
mpluser@yahaira-TP:~/cloud$

```

Figura 7: Ejecución de programa odd_even.c con 6 máquinas en paralelo