Universidad Tecnológica Metropolitana
Facultad de Ingeniería
Departamento de Computación y Informática
Comunicación de Datos

LABORATORIO N°2
"IMPLEMENTACIÓN DE UN CLUSTER"

Nombres: Claudio Acuña A.
Guillermo Rojas L
Cristian Garrido A
José Acuña A

Fecha: 03/01/2014

Índice

Objetivos

Objetivo General

1. Conocer, analizar, implementar y operar un programa de aplicación que funcione sobre un cluster basado en red Fast Ethernet.

Objetivos Específicos

1. Realizar un cluster mpi en sistemas operativo linux
2. Realizar un algoritmo de operaciones básicas de una matriz de 900x900
3. Implementar el algoritmo, distribuyendo las cargas de procesamiento en las diferentes computadoras que componen el cluster.
4. Analizar los resultados obtenidos.

Metodología

Para realizar esta tarea, se montará un "clúster" tipo "Beowulf" sobre tres equipos corriendo en Ubuntu 13.10. Entre estos equipos existe un nodo "maestro" y dos nodos "esclavos", los cuales están conectados mediante ethernet a un switch.

Una vez ya instalados y configurados todos los equipos, se procede a la interconexión de estos. Para otorgar una mayor seguridad se utiliza SSH y también NFS para un acceso de los nodos esclavos hacia el maestro y viceversa.

Finalmente se comunican los nodos mediante OpenMPI utilizando un programa realizado en c++.

Marco Teórico

¿Que es un Cluster?

Un Cluster es una red de computadoras que se comportan como si fuese una (trabajan en conjunto y se comportan como un recurso informático unificado), para trabajar en cierto problema que pueda ser resuelto paralelamente.

Las principales características de un Cluster son:

- Consta de 2 o más nodos (computadoras).
- Los nodos del Cluster están conectados entre sí por al menos un canal de comunicación.
- Los Clusters necesitan software de control especializado.

Cluster de alto rendimiento: Está diseñado para dar altas prestaciones en cuanto a capacidad de cálculo. Los motivos para utilizar este tipo de Cluster son la necesidad de resolver problema de gran complejidad y apalear los costos de implementación de una máquina más poderosa.

A través de un Cluster de alto rendimiento se consiguen capacidades de cálculo superiores a las de un computador más costoso.

Cluster de alta disponibilidad: Se caracteriza por compartir los discos de almacenamiento de datos y por estar constantemente monitorizándose entre sí.

Cluster de balanceo de carga: Está formado por uno o más computadores que actúan como front-end del Cluster, la tarea de éstos es la de repartir las peticiones de servicio que reciba el Cluster a los demás computadores (que conforman el back-end del Cluster).

¿Que es un Cluster tipo Beowulf?

Es un Cluster formado por computadores normalmente idénticos, conectado a una red de área local con librerías y programas instalados que permiten la repartición de los proceso. El resultado es un computador paralelo de alto rendimiento con hardware de computadores personales y software Open Source (Linux).

El primero fue creado en 1994 usando 16 PC con procesadores de 200 MHz que estaban conectados a un switch. El rendimiento teórico era de 3,2 GFlops.

Ventajas

En un sistema distribuido se deben tener en consideración las siguientes características:

- Economía: La relación precio-rendimiento es mayor que en sistemas centralizados, sobre todo cuando se busca altas prestaciones.
- Velocidad: Llega un momento donde no se puede encontrar un sistema centralizado suficientemente potente, siempre existirá un sistema más potente uniendo más nodos.
- Distribución de máquinas: Se pueden tener máquinas inherentemente distribuidas por el tipo de trabajo que realizan.
- Alta disponibilidad: Si una máquina falla, no se cae todo el sistema, sino que este se recupera de las caídas y sigue funcionando con quizás algo menos de velocidad.
- Escalabilidad: Puede empezarse un cluster con pocas máquinas, y cuando la carga aumenta, se añaden más nodos. No es necesario deshacerse de máquinas antiguas, ni tener inversiones elevadas para tener máquinas suficientemente potentes.
- Comunicación: Los computadores necesariamente estarán comunicados, para un funcionamiento correcto y eficaz se crean nuevas funciones avanzadas de comunicación. Estas nuevas primitivas (usadas por los programas y los usuarios) mejoran las comunicaciones con otras máquinas.
- Sistema de ficheros con raíz única: Este sistema de ficheros hace que la administración sea más sencilla y deja a cargo del sistema varias de las tareas.
- Capacidad de comunicación de procesos y de intercambio de datos universal: Permite enviar señales a cualquier proceso del cluster, asimismo permite trabajar conjuntamente con cualquier proceso e intercambiar datos. Por lo tanto será posible tener todos los procesos trabajando en un mismo trabajo.

Desventajas

- La principal desventaja de los clusters es la complejidad que implica su creación. Ahora veremos los problemas que ocurren al intentar implantar las ventajas antes mencionadas:

- La economía, la velocidad y la distribución de máquinas no tienen problemas de implantación porque son inherentes a los sistemas distribuidos.

- Alta disponibilidad: Para lograr una alta disponibilidad tenemos que implantar los mecanismos necesarios para que cuando se caiga una máquina, se continúen entregando todos los servicios. También se tiene que disponer de los mecanismos adecuados para que el nodo que ve el fallo del servidor busque los servidores alternativos en busca de la información que necesita. Además también se debe disponer de los mecanismos necesarios para los nodos que han caído, cuando vuelvan a conectarse al cluster puedan continuar con su trabajo normalmente.

- Escalabilidad: El problema es que más nodos implica que haya también más comunicación, por lo que tenemos que diseñas un sistema lo más escalable posible.

- Comunicación: Un cluster necesita más comunicación que los sistemas normales, por lo tanto se deben crear nuevos métodos de comunicación lo más eficientes posible.

- Sistema de ficheros con raíz única: Hay que independizar los sistemas de ficheros distintos de cada uno de los nodos para crear un sistema de ficheros general.

- Capacidad de comunicación de procesos y de intercambio de datos universal: Para conseguir este objetivo se necesita distinguir unívocamente cada proceso del cluster. Una vez podemos direccionar con que proceso queremos comunicarnos, para enviar señales necesitaremos un sencillo mecanismo de comunicación y seguramente el mismo sistema operativo en el otro extremo que entienda las señales. Para compartir datos, se pueden enviar por la red o crear memoria compartida a lo largo del cluster.

¿Que es MPI?

MPI (Message Passing Interface)

La Interfaz de Paso de Mensajes (MPI), es un protocolo de comunicación entre computadoras. Es una especificación estándar para una librería de funciones de paso de mensajes, independiente de la plataforma y de dominio público.
Proporciona funciones para ser utilizadas con lenguajes de programación como C,C++, Fortran, etc.

MPI debe ser implementado sobre un entorno que se preocupe del manejo de los procesos y la E/S por ejemplo, puesto que MPI sólo se ocupa de la capa de comunicación, necesita un ambiente de programación paralelo nativo.

Los programas y bibliotecas MPI son portables y rápidas, debido a que se han implementado en casi todas las arquitecturas de memoria distribuida y han sido optimizados para el hardware en el que se ejecuta. Pero, como sólo especifica el método de paso de mensajes, el resto del entorno puede ser totalmente diferente en cada implementación, por lo que se impide esa portabilidad que teóricamente tiene.

¿Que es OpenMPI?

El proyecto Open MPI es una implementación de código abierto de MPI que es desarrollada y mantenida por un consorcio de académicos, investigadores y participantes de la industria. Open MPI es por lo tanto capaz de combinar la experticia, tecnología y recursos de la comunidad de alto cómputo para poder construir la mejor biblioteca MPI disponible. Toda la documentación e información acerca de Open MPI se encuentra en: www.open-mpi.org.

¿Qué es SSH?

SSH (Secure Shell)
El intérprete de órdenes segura (SSH), es el nombre de un protocolo y programa que permite acceder a equipos remotos de forma segura, además de permitir el control de estos a través de comandos por consola. SSH permite además copiar datos de forma segura con el uso de claves.

¿Qué es NFS?

NFS (Network File System)

El sistema de archivos de red (NFS) es un protocolo del modelo OSI a nivel de aplicación. Posibilita que varias computadoras en red puedan acceder a archivos y ficheros dentro de esta, de la misma manera que se haría dentro de un fichero local.

- El protocolo NFS está incluido por defecto en los Sistemas Operativos UNIX y la mayoría de las distribuciones de Linux.
- NFS tiene dos partes principales: un servidor y uno o más clientes, que acceden a los archivos en forma remota.
- Los datos se almacenan en el servidor, no es necesario replicarlos en los clientes.
- Es posible compartir dispositivos de almacenamiento masivo tales como: CD-ROM, USB, DVD, etc.

Especificaciones de las Máquinas

En las máquinas utilizadas tanto maestro como esclavos, tiene las mismas especificaciones que se muestran a continuación:

Descripción: Equipo de escritorio
Producto: ()
Anchura: 64 bits
Capacidades: smbios-2.6 dmi-2.6 vsyscall32
Configuración: boot=normal chassis=desktop uuid=DA704149-0E7A-11E2-8BE7-505054503030

- core
    - descripción: Placa base
    - producto: DH61WW
    - fabricante: Intel Corporation
    - id físico: 0
    - versión: AAG23116-302
    - serie: BTWW24000111
    - ranura: To be filled by O.E.M.

- firmware
    - Descripción: BIOS
    - Fabricante: Intel Corp.
    - Id físico: 0
    - Versión: BEH6110H.86A.0044.2012.0531.1710
    - Date: 05/31/2012
    - Tamaño: 64KiB
    - Capacidad: 960KiB
    - Capacidades: pci upgrade shadowing cdboot bootselect socketedrom edd int13floppy1200 int13floppy720 int13floppy2880 int5printscreen int9keyboard int14serial int17printer acpi usb biosbootspecification

- cpu
    - Descripción: CPU

- Producto: Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
- Fabricante: Intel Corp.
- Id físico: 4
- Información del bus: cpu@0
- Versión: Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
- Serie: To Be Filled By O.E.M.
- Ranura: LGA1155 CPU 1
- Tamaño: 1600MHz
- Capacidad: 4GHz
- Anchura: 64 bits
- Reloj: 25MHz

Desarrollo

Configuración Clúster MPI en Ubuntu.

Primero ocupamos 3 máquinas, con sistema operativo GNU/Linux. Para este caso usaremos Ubuntu 13.10. A su vez, el nombre de usuario de cada nodo, lo llamaremos exactamente igual. Usamos el nombre "cluster" (sin comillas). Al mismo tiempo definimos cada nodo con un nombre de máquina propio:

- Nombre equipo maestro: n11 Ip maestro: 10.1.10.113
- Nombre equipo esclavo: n12 Ip esclavo: 10.1.10.114
- Nombre equipo esclavo: n13 Ip esclavo: 10.1.10.115

Luego de instalar el sistema operativo en cada una de las máquinas empezamos con la instalación de de los software necesarios a utilizar en los nodos esclavos:

- sudo apt-get update
- sudo apt-get upgrade
- sudo apt-get install gcc g++
- sudo apt-get install openmpi-bin openmpi-common libopenmpi1.3 libopenmpi-dev

- sudo apt-get install ssh
- sudo apt-get install nfs-kernel-server nfs-common portmap
- sudo apt-get install build-essential

Después creamos la carpeta en donde se compartirán los archivos entre las tres máquinas, para eso nos posicionamos en carpeta cluster y realizaremos los siguientes comandos:

- cd /home/ cluster
- mkdir .ssh
- sudo chmod 777 .ssh
- cd ../../
- sudo mkdir /mpi
- sudo chmod 777 /mpi
- cd /mpi
- sudo chown user /mpi

Luego de que la carpeta compartida esté montada en el nodo maestro ejecutamos en cada nodo esclavo los siguientes comandos:

- sudo mount 10.1.10.113:/mpi /mpi
- sudo gedit /etc/fstab
- 10.1.10.113:/mpi /mpi nfs
- sudo mount –a

Para el nodo maestro realizaremos los mismos pasos de instalación de sistema operativo, de software necesario para funcionar y la creación de la carpeta compartida también :

- sudo apt-get update
- sudo apt-get upgrade
- sudo apt-get install gcc g++
- sudo apt-get install openmpi-bin openmpi-common libopenmpi1.3 libopenmpi-dev
- sudo apt-get install ssh
- sudo apt-get install nfs-kernel-server nfs-common portmap

- sudo apt-get install build-essential

Creación de carpeta a compartir

- cd /home/ cluster
- mkdir .ssh
- sudo chmod 777 .ssh
- cd ../../
- sudo mkdir /mpi
- sudo chmod 777 /mpi
- cd /mpi
- sudo chown user /mpi

Luego de tener la carpeta compartida lista, editamos los ficheros host y exports, para eso ejecutamos el siguientes comando para poder abrir el archivo:

- sudo gedit /etc/hosts

Una vez abierto el archivo host lo editamos con lo que se muestra a continuación:

127.0.0.1 localhost

“ip” “usuario” “equipo”
10.1.10.113 cluster n13 # nodo maestro
10.1.10.014 cluster n14 # nodo esclavo 1
10.1.10.015 cluster n15 # nodo esclavo 2

The following lines are desirable for IPv6 capable hosts
::1 ip6-localhost ip6-loopback

Ahora abrimos el archivo exports asi:

- sudo gedit /etc/exports

Después de abrir el archivo lo editamos con lo siguiente:

/etc/exports: the access control list for filesystems which may
be exported
to NFS clients. See exports(5).
gss/krb5i(rw,sync,fsid=0,crossmnt,no_subtree_check)
/srv/nfs4/homes gss/krb5i(rw,sync,no_subtree_check)
/mpi *(rw,sync)

Luego de editar los archivos, compartimos el directorio con los nodos esclavos de con el siguiente comando:

- sudo service nfs-kernel-server restart

En este paso esperamos a que los esclavos tengan monada la carpeta, una vez ya montada ejecutamos los siguientes comandos:

- ssh-keygen -t rsa
- cd /home/cluster/.ssh
- cat id_rsa.pub >> authorized_keys
- ssh-copy-id 10.1.10.114
- ssh-copy-id 10.1.10.115
- ssh 10.1.10.114

Finalmente para ejecutar nuestro codigo nos posicionamos en la carpeta compartida

- cd /mpi

Clonamos el repositorio e ingresamos a este

- git clone https://github.com/cris-gar/cluster_Lad_Cobra

- cd /mpi/cluster_Lad_Cobra

Luego para Compilar y ejecutar los codigos lo hacemos de la siguiente manera:

- mpicc <nomArchivo>.c -o <nomAliasArchivo>
- mpirun -np <numero de hilos> --hostfile /home/hostfile ./<nomAliasArchivo>

Para revisar el estado de distribuición de carga, recomendamos utilizar htop. Se puede instalar y ejecutar con los siguientes comandos:

- sudo apt-get install htop
- htop

Códigos

Código Resta

```
#include<stdio.h>
#include<mpi.h>
#define NUM_ROWS_A 900 //Filas de la matriz [A]
#define NUM_COLUMNS_A 900 //Columnas de la matriz [A]
#define NUM_ROWS_B 900 //Filas de la matriz [B]
#define NUM_COLUMNS_B 900 //Columnas de la matriz [B]
#define MASTER_TO_SLAVE_TAG 1 //Etiqueta para mensajes enviados del maestro a esclavos
#define SLAVE_TO_MASTER_TAG 4 //Etiqueta para mensajes enviados de esclavos a maestro

void makeAB(); // Fabrica las [A] y [B] matrices
void printArray(); //Imprime la salida en una matriz [C];
int rank; //Rango de procesos
int size; //Número de procesos
int i, j, k; //Variables auxiliares
double mat_a[NUM_ROWS_A][NUM_COLUMNS_A]; // Declara entrada en [A]
double mat_b[NUM_ROWS_B][NUM_COLUMNS_B]; // Declara entrada en [B]
double mat_result[NUM_ROWS_A][NUM_COLUMNS_B]; //Declara una salida [C]
double start_time; //mantiene tiempo de inicio
double end_time; // mantiene tiempo de fin
int low_bound; // Límite inferior del número de filas de A asociadas a un esclavo
int upper_bound; // Límite superior del número de filas de A asociadas a un esclavo
int portion; // porción del número de filas de [A] asociadas a un esclavo

MPI_Status status; //  Almacenaje de estado de un  MPI_Recv
MPI_Request request; // Captura de solicitud de un MPI_Isend

int main(int argc, char *argv[])
{
   MPI_Init(&argc, &argv); // Inicializa operaciones MPI
   MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Obtén el rango
   MPI_Comm_size(MPI_COMM_WORLD, &size); // Obtén el número de procesos

   /* maestro inicializa la carga*/
```

```
   if (rank == 0) {
        makeAB();
        start_time = MPI_Wtime();
        for (i = 1; i < size; i++) {//"Para cada esclavo que no sea el maestro
             portion = (NUM_ROWS_A / (size - 1)); // clacula porción sin el maestro
             low_bound = (i - 1) * portion;
             if (((i + 1) == size) && ((NUM_ROWS_A % (size - 1)) != 0)) {// Si las filas en [A] no pueden
serigualmente divididas entre esclavos
                  upper_bound = NUM_ROWS_A; // Último esclavo, se queda con las filas sobrantes
             } else {
                  upper_bound = low_bound + portion; //las filas de [A] son equitativamente divisibles
entre esclavos
             }
             // Envía el límite inferior primero sin bloquear, al esclavo  deseado
             MPI_Isend(&low_bound, 1, MPI_INT, i, MASTER_TO_SLAVE_TAG, MPI_COMM_WORLD,
&request);
             // Luego, en vía el l{imite superior sin bloquear al eslavo deseado
             MPI_Isend(&upper_bound, 1, MPI_INT, i, MASTER_TO_SLAVE_TAG + 1,
MPI_COMM_WORLD, &request);
             // Finalmente envía el la porción asignada de filas de [A] sin bloquear al eslavo deseado
             MPI_Isend(&mat_a[low_bound][0], (upper_bound - low_bound) * NUM_COLUMNS_A,
MPI_DOUBLE, i, MASTER_TO_SLAVE_TAG + 2, MPI_COMM_WORLD, &request);
        }
   }

   // Transmite a [B] a todos los esclavos
   MPI_Bcast(&mat_b, NUM_ROWS_B*NUM_COLUMNS_B, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   /* Carga hecha por los esclavos*/
   if (rank > 0) {
        // Recibe límite inferior del maestro.
        MPI_Recv(&low_bound, 1, MPI_INT, 0, MASTER_TO_SLAVE_TAG, MPI_COMM_WORLD,
&status);
        // Luego, recibe límite superior desde el maestro
        MPI_Recv(&upper_bound, 1, MPI_INT, 0, MASTER_TO_SLAVE_TAG + 1, MPI_COMM_WORLD,
&status);
         // Finalmente recibe la porción de filas de [A] a ser procesadas por el maestro
```

```
        MPI_Recv(&mat_a[low_bound][0], (upper_bound - low_bound) * NUM_COLUMNS_A, MPI_DOUBLE,
0, MASTER_TO_SLAVE_TAG + 2, MPI_COMM_WORLD, &status);
        for (i = low_bound; i < upper_bound; i++) {// Itera a através de un conjunto de filas dadas de [A]
                for (j = 0; j < NUM_COLUMNS_B; j++) {// Itera a através de un conjunto de culomnas dadas de
[B]
                                mat_result[i][j] = (mat_a[i][j] - mat_b[i][j]);
                }
        }

        //Envía de vuelta el límite inferior primero, sin bloquear, al maestro
        MPI_Isend(&low_bound, 1, MPI_INT, 0, SLAVE_TO_MASTER_TAG, MPI_COMM_WORLD,
&request);
        //Envía de vuelta el límite superior primero, sin bloquear, al maestro
        MPI_Isend(&upper_bound, 1, MPI_INT, 0, SLAVE_TO_MASTER_TAG + 1, MPI_COMM_WORLD,
&request);
        //Finalmente envía la porcion procesada de datos sin bloquear al maestro
        MPI_Isend(&mat_result[low_bound][0], (upper_bound - low_bound) * NUM_COLUMNS_B,
MPI_DOUBLE, 0, SLAVE_TO_MASTER_TAG + 2, MPI_COMM_WORLD, &request);
    }

    /* El maestro recibe la carga*/
    if (rank == 0) {
        for (i = 1; i < size; i++) {// Hasta que todos los esclavos hayan devuelto los datos procesados
                //Recibe límite inferior de un esclavo
                MPI_Recv(&low_bound, 1, MPI_INT, i, SLAVE_TO_MASTER_TAG, MPI_COMM_WORLD,
&status);
                //Recibe límite superior de un esclavo
                MPI_Recv(&upper_bound, 1, MPI_INT, i, SLAVE_TO_MASTER_TAG + 1,
MPI_COMM_WORLD, &status);
                //Recibe datos procesados de un esclavo
                MPI_Recv(&mat_result[low_bound][0], (upper_bound - low_bound) * NUM_COLUMNS_B,
MPI_DOUBLE, i, SLAVE_TO_MASTER_TAG + 2, MPI_COMM_WORLD, &status);
        }
        end_time = MPI_Wtime();
        printf("\nRunning Time = %f\n\n", end_time - start_time);
        printArray();
    }
```

```
    MPI_Finalize(); // Finaliza operaciones MPI
    return 0;
}

void makeAB()
{
    for (i = 0; i < NUM_ROWS_A; i++) {
            for (j = 0; j < NUM_COLUMNS_A; j++) {
                    mat_a[i][j] =1;
            }
    }
    for (i = 0; i < NUM_ROWS_B; i++) {
            for (j = 0; j < NUM_COLUMNS_B; j++) {
                    mat_b[i][j] = 2;
            }
    }
}

void printArray()
{
    for (i = 0; i < NUM_ROWS_A; i++) {
            printf("\n");
            for (j = 0; j < NUM_COLUMNS_B; j++)
                    printf("%8.2f ", mat_result[i][j]);
    }
    printf("\n\n");
}
```

Código Suma

```
#include<stdio.h>
#include<mpi.h>
#define NUM_ROWS_A 900 //rows of input [A]
#define NUM_COLUMNS_A 900 //columns of input [A]
#define NUM_ROWS_B 900 //rows of input [B]
#define NUM_COLUMNS_B 900 //columns of input [B]
#define MASTER_TO_SLAVE_TAG 1 //tag for messages sent from master to slaves
#define SLAVE_TO_MASTER_TAG 4 //tag for messages sent from slaves to master

void makeAB(); //makes the [A] and [B] matrixes
void printArray(); //print the content of output matrix [C];
void A();
void B();
int rank; //process rank
int size; //number of processes
int i, j, k; //helper variables
double mat_a[NUM_ROWS_A][NUM_COLUMNS_A]; //declare input [A]
double mat_b[NUM_ROWS_B][NUM_COLUMNS_B]; //declare input [B]
double mat_result[NUM_ROWS_A][NUM_COLUMNS_B]; //declare output [C]
double start_time; //hold start time
double end_time; // hold end time
int low_bound; //low bound of the number of rows of [A] allocated to a slave
int upper_bound; //upper bound of the number of rows of [A] allocated to a slave
int portion; //portion of the number of rows of [A] allocated to a slave

MPI_Status status; // store status of a MPI_Recv
MPI_Request request; //capture request of a MPI_Isend

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

   MPI_Init(&argc, &argv); //initialize MPI operations
   MPI_Comm_rank(MPI_COMM_WORLD, &rank); //get the rank
   MPI_Comm_size(MPI_COMM_WORLD, &size); //get number of processes
```

```
   /* master initializes work*/
   if (rank == 0) {
        int editarA = 1;
        int editarB = 1;
        makeAB();
        printf("La matriz A contiene solo valores 1\n");
        printf("La matriz B contiene solo valores 2\n");

        printf("Desea editar alguna celda de la Matriz A S=1 N=0\n");
        scanf("%d",&editarA);
        if (editarA == 1)
        {
             A();
        }

        printf("Desea editar alguna celda de la Matriz B S=1 N=0\n");
        scanf("%d",&editarB);
        if (editarB == 1)
        {
             B();
        }


        start_time = MPI_Wtime();
        for (i = 1; i < size; i++) {//for each slave other than the master
             portion = (NUM_ROWS_A / (size - 1)); // calculate portion without master
             low_bound = (i - 1) * portion;
             if (((i + 1) == size) && ((NUM_ROWS_A % (size - 1)) != 0)) {//if rows of [A] cannot be equally
divided among slaves
                  upper_bound = NUM_ROWS_A; //last slave gets all the remaining rows
             } else {
                  upper_bound = low_bound + portion; //rows of [A] are equally divisable among slaves
             }
             //send the low bound first without blocking, to the intended slave
             MPI_Isend(&low_bound, 1, MPI_INT, i, MASTER_TO_SLAVE_TAG, MPI_COMM_WORLD,
&request);
```

```
                //next send the upper bound without blocking, to the intended slave
                MPI_Isend(&upper_bound, 1, MPI_INT, i, MASTER_TO_SLAVE_TAG + 1,
MPI_COMM_WORLD, &request);
                //finally send the allocated row portion of [A] without blocking, to the intended slave
                MPI_Isend(&mat_a[low_bound][0], (upper_bound - low_bound) * NUM_COLUMNS_A,
MPI_DOUBLE, i, MASTER_TO_SLAVE_TAG + 2, MPI_COMM_WORLD, &request);
        }
    }

    //broadcast [B] to all the slaves
    MPI_Bcast(&mat_b, NUM_ROWS_B*NUM_COLUMNS_B, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    /* work done by slaves*/
    if (rank > 0) {
        //receive low bound from the master
        MPI_Recv(&low_bound, 1, MPI_INT, 0, MASTER_TO_SLAVE_TAG, MPI_COMM_WORLD,
&status);
        //next receive upper bound from the master
        MPI_Recv(&upper_bound, 1, MPI_INT, 0, MASTER_TO_SLAVE_TAG + 1, MPI_COMM_WORLD,
&status);
         //finally receive row portion of [A] to be processed from the master
        MPI_Recv(&mat_a[low_bound][0], (upper_bound - low_bound) * NUM_COLUMNS_A, MPI_DOUBLE,
0, MASTER_TO_SLAVE_TAG + 2, MPI_COMM_WORLD, &status);
        for (i = low_bound; i < upper_bound; i++) {//iterate through a given set of rows of [A]
                for (j = 0; j < NUM_COLUMNS_B; j++) {//iterate through columns of [B]
                                mat_result[i][j] = (mat_a[i][j] + mat_b[i][j]);
                }
        }

        //send back the low bound first without blocking, to the master
        MPI_Isend(&low_bound, 1, MPI_INT, 0, SLAVE_TO_MASTER_TAG, MPI_COMM_WORLD,
&request);
        //send the upper bound next without blocking, to the master
        MPI_Isend(&upper_bound, 1, MPI_INT, 0, SLAVE_TO_MASTER_TAG + 1, MPI_COMM_WORLD,
&request);
        //finally send the processed portion of data without blocking, to the master
        MPI_Isend(&mat_result[low_bound][0], (upper_bound - low_bound) * NUM_COLUMNS_B,
MPI_DOUBLE, 0, SLAVE_TO_MASTER_TAG + 2, MPI_COMM_WORLD, &request);
```

```
    }

    /* master gathers processed work*/
    if (rank == 0) {
        for (i = 1; i < size; i++) {// untill all slaves have handed back the processed data
            //receive low bound from a slave
            MPI_Recv(&low_bound, 1, MPI_INT, i, SLAVE_TO_MASTER_TAG, MPI_COMM_WORLD,
&status);
            //receive upper bound from a slave
            MPI_Recv(&upper_bound, 1, MPI_INT, i, SLAVE_TO_MASTER_TAG + 1,
MPI_COMM_WORLD, &status);
            //receive processed data from a slave
            MPI_Recv(&mat_result[low_bound][0], (upper_bound - low_bound) * NUM_COLUMNS_B,
MPI_DOUBLE, i, SLAVE_TO_MASTER_TAG + 2, MPI_COMM_WORLD, &status);
        }
        end_time = MPI_Wtime();
        printf("\nRunning Time = %f\n\n", end_time - start_time);
        printArray();
    }
    MPI_Finalize(); //finalize MPI operations
    return 0;
}

void makeAB()
{
    for (i = 0; i < NUM_ROWS_A; i++) {
        for (j = 0; j < NUM_COLUMNS_A; j++) {
            mat_a[i][j] = rand () % 10;
        }
    }
    for (i = 0; i < NUM_ROWS_B; i++) {
        for (j = 0; j < NUM_COLUMNS_B; j++) {
            mat_b[i][j] = rand () % 10;
        }
    }
}
```

```
void A()
{
	int fila, columna;
	double aux = 0;
	int editar = 1;

	while(editar == 1)
	{
		printf ("Ingrese fila y columna a editar: \n");
	scanf("%d %d",&fila,&columna);
	printf("Ingrese el nuevo valor para la celda: \n");
	scanf("%lf",&aux);
	mat_a[fila][columna]=aux;
	printf ("¿Desea editar otra celda? S=1 o N=0: \n");
	scanf ("%d",&editar);
	}

}

void B()
{
	int fila, columna;
	double aux = 0;
	int editar = 1;

	while(editar == 1)
	{
		printf ("Ingrese fila y columna a editar: \n");
	scanf("%d %d",&fila,&columna);
	printf("Ingrese el nuevo valor para la celda: \n");
	scanf("%lf",&aux);
	mat_b[fila][columna]=aux;
	printf ("¿Desea editar otra celda? S=1 o N=0: \n");
	scanf ("%d",&editar);
	}

}
```

```
void printArray()
{
   for (i = 0; i < NUM_ROWS_A; i++) {
           printf("\n");
           for (j = 0; j < NUM_COLUMNS_B; j++)
                   printf("%8.2f ", mat_result[i][j]);
   }
   printf("\n\n");
}
```

Código Multiplicación

```
#include<stdio.h>
#include<mpi.h>
#define NUM_ROWS_A 900 //rows of input [A]
#define NUM_COLUMNS_A 900 //columns of input [A]
#define NUM_ROWS_B 900 //rows of input [B]
#define NUM_COLUMNS_B 900 //columns of input [B]
#define MASTER_TO_SLAVE_TAG 1 //tag for messages sent from master to slaves
#define SLAVE_TO_MASTER_TAG 4 //tag for messages sent from slaves to master

void makeAB(); //makes the [A] and [B] matrixes
void printArray(); //print the content of output matrix [C];
void A();
void B();
int rank; //process rank
int size; //number of processes
int i, j, k; //helper variables
double mat_a[NUM_ROWS_A][NUM_COLUMNS_A]; //declare input [A]
double mat_b[NUM_ROWS_B][NUM_COLUMNS_B]; //declare input [B]
double mat_result[NUM_ROWS_A][NUM_COLUMNS_B]; //declare output [C]
double start_time; //hold start time
double end_time; // hold end time
int low_bound; //low bound of the number of rows of [A] allocated to a slave
int upper_bound; //upper bound of the number of rows of [A] allocated to a slave
int portion; //portion of the number of rows of [A] allocated to a slave
```

```
MPI_Status status; // store status of a MPI_Recv
MPI_Request request; //capture request of a MPI_Isend

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

   MPI_Init(&argc, &argv); //initialize MPI operations
   MPI_Comm_rank(MPI_COMM_WORLD, &rank); //get the rank
   MPI_Comm_size(MPI_COMM_WORLD, &size); //get number of processes

   /* master initializes work*/
   if (rank == 0) {
         int editarA = 1;
         int editarB = 1;
         makeAB();
         printf("La matriz A contiene solo valores 1\n");
         printf("La matriz B contiene solo valores 2\n");

         printf("Desea editar alguna celda de la Matriz A S=1 N=0\n");
         scanf("%d",&editarA);
         if (editarA == 1)
         {
               A();
         }

         printf("Desea editar alguna celda de la Matriz B S=1 N=0\n");
         scanf("%d",&editarB);
         if (editarB == 1)
         {
               B();
         }


         start_time = MPI_Wtime();
         for (i = 1; i < size; i++) {//for each slave other than the master
```

```
                portion = (NUM_ROWS_A / (size - 1)); // calculate portion without master
                low_bound = (i - 1) * portion;
                if (((i + 1) == size) && ((NUM_ROWS_A % (size - 1)) != 0)) {//if rows of [A] cannot be equally
divided among slaves
                        upper_bound = NUM_ROWS_A; //last slave gets all the remaining rows
                } else {
                        upper_bound = low_bound + portion; //rows of [A] are equally divisable among slaves
                }
                //send the low bound first without blocking, to the intended slave
                MPI_Isend(&low_bound, 1, MPI_INT, i, MASTER_TO_SLAVE_TAG, MPI_COMM_WORLD,
&request);
                //next send the upper bound without blocking, to the intended slave
                MPI_Isend(&upper_bound, 1, MPI_INT, i, MASTER_TO_SLAVE_TAG + 1,
MPI_COMM_WORLD, &request);
                //finally send the allocated row portion of [A] without blocking, to the intended slave
                MPI_Isend(&mat_a[low_bound][0], (upper_bound - low_bound) * NUM_COLUMNS_A,
MPI_DOUBLE, i, MASTER_TO_SLAVE_TAG + 2, MPI_COMM_WORLD, &request);
        }
    }

    //broadcast [B] to all the slaves
    MPI_Bcast(&mat_b, NUM_ROWS_B*NUM_COLUMNS_B, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    /* work done by slaves*/
    if (rank > 0) {
        //receive low bound from the master
        MPI_Recv(&low_bound, 1, MPI_INT, 0, MASTER_TO_SLAVE_TAG, MPI_COMM_WORLD,
&status);
        //next receive upper bound from the master
        MPI_Recv(&upper_bound, 1, MPI_INT, 0, MASTER_TO_SLAVE_TAG + 1, MPI_COMM_WORLD,
&status);
         //finally receive row portion of [A] to be processed from the master
        MPI_Recv(&mat_a[low_bound][0], (upper_bound - low_bound) * NUM_COLUMNS_A, MPI_DOUBLE,
0, MASTER_TO_SLAVE_TAG + 2, MPI_COMM_WORLD, &status);

        for (i = low_bound; i < upper_bound; i++) {//iterate through a given set of rows of [A]
                for (j = 0; j < NUM_COLUMNS_B; j++) {//iterate through columns of [B]
                        mat_result[i][j] = 0;
```

```
                        for (k=0; k < NUM_COLUMNS_A; k++){
                                mat_result[i][j] = (mat_result[i][j] + (mat_a[i][k] * mat_b[k][j]));
                        }
                }
        }

        //send back the low bound first without blocking, to the master
        MPI_Isend(&low_bound, 1, MPI_INT, 0, SLAVE_TO_MASTER_TAG, MPI_COMM_WORLD,
&request);
        //send the upper bound next without blocking, to the master
        MPI_Isend(&upper_bound, 1, MPI_INT, 0, SLAVE_TO_MASTER_TAG + 1, MPI_COMM_WORLD,
&request);
        //finally send the processed portion of data without blocking, to the master
        MPI_Isend(&mat_result[low_bound][0], (upper_bound - low_bound) * NUM_COLUMNS_B,
MPI_DOUBLE, 0, SLAVE_TO_MASTER_TAG + 2, MPI_COMM_WORLD, &request);
   }

   /* master gathers processed work*/
   if (rank == 0) {
        for (i = 1; i < size; i++) {// untill all slaves have handed back the processed data
                //receive low bound from a slave
                MPI_Recv(&low_bound, 1, MPI_INT, i, SLAVE_TO_MASTER_TAG, MPI_COMM_WORLD,
&status);
                //receive upper bound from a slave
                MPI_Recv(&upper_bound, 1, MPI_INT, i, SLAVE_TO_MASTER_TAG + 1,
MPI_COMM_WORLD, &status);
                //receive processed data from a slave
                MPI_Recv(&mat_result[low_bound][0], (upper_bound - low_bound) * NUM_COLUMNS_B,
MPI_DOUBLE, i, SLAVE_TO_MASTER_TAG + 2, MPI_COMM_WORLD, &status);
        }
        end_time = MPI_Wtime();
        printf("\nRunning Time = %f\n\n", end_time - start_time);
        printArray();
   }
   MPI_Finalize(); //finalize MPI operations
   return 0;
}
```

```
void makeAB()
{
   for (i = 0; i < NUM_ROWS_A; i++) {
          for (j = 0; j < NUM_COLUMNS_A; j++) {
                 mat_a[i][j] = rand () % 10;
          }
   }
   for (i = 0; i < NUM_ROWS_B; i++) {
          for (j = 0; j < NUM_COLUMNS_B; j++) {
                 mat_b[i][j] = rand () % 10;
          }
   }
}

void A()
{
          int fila, columna;
          double aux = 0;
          int editar = 1;

          while(editar == 1)
          {
                 printf ("Ingrese fila y columna a editar: \n");
          scanf("%d %d",&fila,&columna);
          printf("Ingrese el nuevo valor para la celda: \n");
          scanf("%lf",&aux);
          mat_a[fila][columna]=aux;
          printf ("¿Desea editar otra celda? S=1 o N=0: \n");
          scanf ("%d",&editar);
          }

}

void B()
{
          int fila, columna;
```

```
        double aux = 0;
        int editar = 1;

        while(editar == 1)
        {
                printf ("Ingrese fila y columna a editar: \n");
        scanf("%d %d",&fila,&columna);
        printf("Ingrese el nuevo valor para la celda: \n");
        scanf("%lf",&aux);
        mat_b[fila][columna]=aux;
        printf ("¿Desea editar otra celda? S=1 o N=0: \n");
        scanf ("%d",&editar);
        }

}

void printArray()
{
   for (i = 0; i < NUM_ROWS_A; i++) {
        printf("\n");
        for (j = 0; j < NUM_COLUMNS_B; j++)
                printf("%8.2f ", mat_result[i][j]);
   }
   printf("\n\n");
}
```

Conclusión

La computación paralela es de gran importancia hoy en día para el cálculo de operaciones extremadamente grandes por la escalabilidad que se le puede dar a esas operaciones. En esa característica entran en juego los cluster como la vía válida para desarrollar cálculos paralelos.

En el caso del laboratorio no se pueden ver muchas diferencias entre cálculos seriales (en 1 pc) y paralelos (más de 1 pc) por el tamaño de cálculos que estamos haciendo (muy bajos), pero a medida que se aumenta el número de datos a procesar por las máquinas se va a ir viendo una diferencia de tiempo más notoria entre los cálculos secuenciales y paralelos, y ese tiempo puede seguir siendo optimizado agregándole escalabilidad (mas pc's al cluster) o aumentando/disminuyendo la granularidad de los problemas entre otras variables.

En conclusión, para investigaciones y avances tecnológicos actuales en donde los cálculos son tremendamente grande y complejos se debe ocupar computación paralela y una forma de hacerlo es mediante un cluster por lo que es una herramienta fundamental para optimizar el trabajo computacional.