MPI

Alvarado Morán Óscar
Bermúdez Marbán Dante
García Avendaño Martín

12 de noviembre del 2019

1. Introducción

1.1. Modelo 'Message-Passing'

El paso de mensajes es básicamente un protocolo de comunicación con cierto tipo de reglas, las que incluyen que aquel nodo que envía el mensaje decide qué datos enviar, los datos están completamente conectados después de que el destinatario acepta recibir el mensaje, puede esperar el recibidor por siempre para recibir el mensaje, es decir, una vez que acepta recibir el mensaje, éste esperará por siempre hasta que el mensaje entre.

1.2. Conceptos básicos.

Un programa con MPI no es más que una colección de procesos que intercambian mensajes.

MPI no es, como comúnmente se le conoce, una revolucionaria forma de programación para cómputo paralelo, sino que es un intento de la recolección de las mejores características de distintos sistemas de paso de mensajes que han sido desarrollados a través de los años, mejorándolos donde es necesario y estandarizando todo esto. Es una librería, no un lenguaje, es un estándar de paso de mensajes. Un correcto programa con MPI debe funcionar en cualquier implementación de MPI sin ningún cambio.

Las implementaciones principales son OpenMPI y MPICH. Hay tres tipos de versiones de MPI:

- MPI-1: Define las operaciones de comunicación estándar y se basa en el modelo de procesos estáticos.
- MPI-2: Provee soporte para manejo de procesos dinámicos y I/O paralelo.
- MPI-3: Provee operaciones colectivas sin bloqueo, entre otras nuevas características.

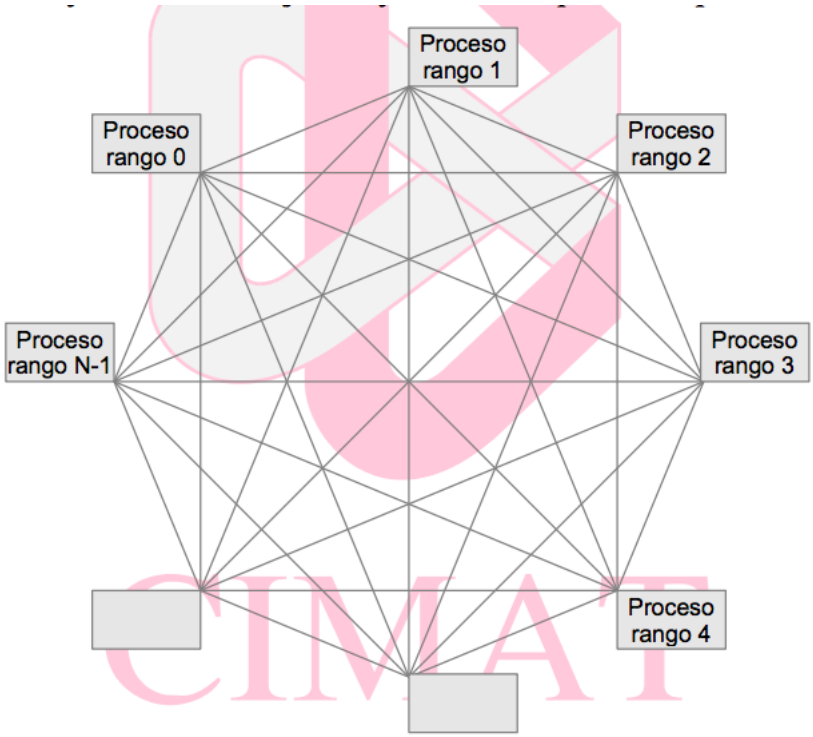

Figura 1: Esquema maestro-esclavo

2. Modelo maestro-esclavo

MPI es adecuado para tareas en que se divide en subtareas. La mayoría de los procesos (esclavos) hacen los cálculos y hay uno (maestro) que çontrola.ªl resto.

Este modelo consiste en elegir un proceso, al que le llamaremos maestro, controle al resto de procesos, llamados esclavos.

3. Instrucciones

Todas las instrucciones tienen la forma `MPI_...` y para usarse en C se debe incluir `mpi.h`.

Todo programa que usa MPI debe usar `MPI_Init` al principio y terminar con `MPI_Finalize`

```
1 #include<stdio.h>
2 #include<mpi.h>
```

```
3
4 int main(int argc, char *argv[]){
5     MPI_Init(&argc, &argv);
6     ...
7     MPI_Finalize();
8 }
```

3.1. MPI_Comm_size

Retorna el número de procesos

3.2. MPI_Comm_rank

Si hay n procesos, a cada proceso le corresponde un identificador llamado rango que va desde 0 hasta n-1. Esta función retorna dicho identificador

3.3. MPI_Send

```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator)
```

- **data** es el buffer de donde se leerá
- **count** es la cantidad de elementos que se leerán
- **datatype** es el tipo de dato que se está leyendo
- **destination** es el rango del proceso al que se le está enviando el mensaje.
- **tag** Es una etiqueta para identificar el mensaje
- **communicator** es el comunicador que se utiliza. MPI provee uno que comunica a todos llamado `MPI_COMM_WORLD`

3.4. MPI_Recv

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

- **data** es el buffer de donde se leerá
- **count** es la cantidad de elementos que se leerán
- **datatype** es el tipo de dato que se está leyendo
- **destination** es el rango del proceso al que se le está enviando el mensaje.
- **tag** Es una etiqueta para identificar el mensaje
- **communicator** es el comunicador que se utiliza. MPI provee uno que comunica a todos llamado `MPI_COMM_WORLD`
- **status** provee información sobre el mensaje.

4. Comunicación colectiva MPI

La comunicación colectiva nos permite intercambiar información entre múltiples procesos lo que facilita el paralelismo, existen diferentes tipos de comunicación cada una adecuada para un determinado objetivo

- **Difusión (Brodcast)**: Un proceso envía mensaje a todos los demás procesos
- **Reducción(reduction)**: Un proceso obtiene datos de los demás procesos y realiza alguna operación con esos datos
- **Dispersión(scatter**: Un solo proceso divide la información que posee en múltiples fragmentos.
- **Recopilar(gather)**: Es la operación opuesta del scatter la cual consiste en que el proceso padre junta toda la información de los procesos.

4.1. Brodcast

Es apropiado usar el Brodcast cuando deseas que un proceso padre envíe una copia de un determinado dato o conjunto de datos a todos los procesos entonces los procesos hijos recibirán una copia de esa información es decir todos tendrán la misma información y la guardaran en su memoria intermedia.
La función se define como `MPI_Bcast`

4.2. Reduction

Las reducciones son operaciones que se aplicaran a todos los buffers de los procesos dependientes de un proceso padre es decir si se tienen dos procesos hijos y se desea realizar una suma el proceso padre recibirá un dato de cada uno de los procesos hijos el realizara la suma y enviara el resultado a los buffers de los procesos hijos.
La función se define como `MPI_Redu`

4.3. Scattering and Gathering

Scattering and Gathering son o se pueden interpretar como funciones opuestas es decir el Scattering distribuye y el Gathering recolecta. Para dejar esto mas claro si deseásemos enviar un componente de un vector de 5 datos (5,4,3,6,7) a cada uno de nuestro procesos hijos usaremos la función scattering la cual dividirá el vector y enviara un 5 a el proceso hijo 1 el 4 y el proceso dos mientras que su función contraria gathering se podría usar para obtener un valor de cada uno de los procesos y obtener un vector
Sus funciones respectivas se definen como `MPI_Scatter` y `MPI_Gather`

Referencias

[1] `https://computing.llnl.gov/tutorials/openMP/`

[2] `https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-directives?view=vs-2019#atomic`