
Computación Distribuida

Nosotros no vamos a estudiar los sistemas distribuidos; sino cómo dividir y ejecutar tareas para computarlas de forma distribuida.

Sistema Distribuido

Un **sistema distribuido** consiste en una **colección de computadoras autónomas, conectadas a través de una red y un middleware de distribución**, que permite que las computadoras coordinen sus actividades y compartan los recursos del sistema, de modo que los usuarios perciban el sistema como una única instalación informática integrada.

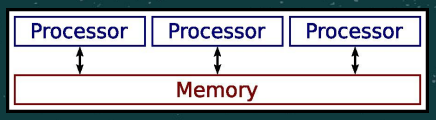
Ejemplo de un cluster de computadoras

Digamos sobre el servidor web de Google, desde la perspectiva de los usuarios, mientras envían la consulta buscada, asumen el servidor web de Google como un sistema único. Sin embargo, detrás de la cortina, Google ha construido una gran cantidad de servidores que se distribuyen (geográfica y computacionalmente) para darnos el resultado en unos pocos segundos.

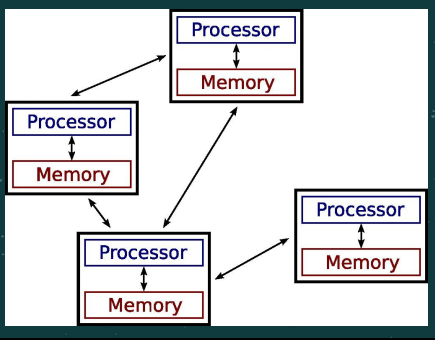
Computación Paralela vs Distribuida

- En la computación paralela, **todos los procesadores pueden tener acceso a una memoria compartida** para intercambiar información entre procesadores.
- En la computación distribuida, **cada procesador tiene su propia memoria privada** (memoria distribuida). La información se intercambia pasando **mensajes** entre los procesadores.

Computación Paralela



Computación Distribuida

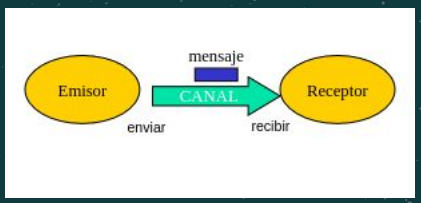


Modelo de paso de mensajes

- Los procesos no comparten memoria (variables, objetos, etc.)
- La comunicación se hace mediante operaciones explícitas de envío y recepción
- Dos primitivas:

  Send: Envía un mensaje

  Receive: Recibe un mensaje



Válido para cualquier arquitectura de computadores : 1) sistemas distribuidos, 2) arquitecturas paralelas sin memoria compartida y también en sistemas de memoria compartida

Sincronización entre emisor y receptor

Sincrónica: se requiere que ambos procesos estén sincronizados (rendevouz). El emisor es bloqueado hasta que se sabe que su petición es aceptada. Y viceversa, el receptor se bloquea hasta que el remitente envía el mensaje.

Asincrónica: el emisor continúa inmediatamente después de que ha pasado su mensaje para la transmisión. La comunicación asíncrona requiere un buffer para almacenar los mensajes.

Identificación y simetría

Comunicación directa:
Cada proceso que desea comunicarse debe nombrar explícitamente el destinatario o el remitente de la comunicación.

- enviar(P,mensaje) ->Enviar un mensaje al proceso P
- recibir(Q,mensaje) ->Recibir un mensaje del proceso Q

Comunicación indirecta:
Los mensajes se envían a, y se reciben de, buzones (también llamados puertos).

Comunicación simétrica: Los procesos tanto receptor como emisor necesitan nombrar al otro para comunicarse.
Comunicación asimétrica: Sólo el emisor nombra al destinatario□. Resuelve el problema en aplicaciones cliente/servidor.

Llamadas bloqueantes / no bloqueantes

- Las operaciones de envío y recepción pueden estar definidas como bloqueantes o no bloqueantes.
- □ Un envío/recepción con bloqueo es un ejemplo de **comunicación síncrona.**
- Un envió/recepción sin bloqueo es un ejemplo de **comunicación asíncrona.**

Message Passing Interface (MPI)

- MPI es una **librería de paso de mensajes** estandarizada y portátil desarrollada para computación distribuida y paralela.
  - Permite a los usuarios crear aplicaciones paralelas mediante la creación de procesos paralelos y el intercambio de información entre estos procesos. **Los procesos no comparten memoria.**
  - Proporciona la capacidad de paralelizar el código en un sistema distribuido no-homogéneo. Por ejemplo, es posible paralelizar un programa completo en una red de computadoras, o nodos, que se comunican en la misma red.
- Las ventajas de MPI sobre las librerías de paso de mensajes más antiguas es la portabilidad y la velocidad
  - MPI se ha implementado para casi todas las arquitecturas de memoria distribuida
  - cada implementación está, en principio, optimizada para el hardware en el que se ejecuta

Compilar y ejecutar programas MPI

Instalar MPI:

```
sudo apt install libopenmpi-dev openmpi-doc
```

Comilar:

```
mpicc program_name.c -o binary_file
```

Ejecutar:

```
mpirun -np [number of processes] ./binary_file
```

Hello world!

```
#include <stdio.h>
#include <mpi.h>
void main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
  printf("Hello world!\n");
  MPI_Finalize();
}
```

Cuando se inicia el programa, consta de un solo proceso (denominado proceso parent, root o master). Cuando la rutina `MPI_Init` se ejecuta dentro del proceso root, provoca la creación de `n` procesos adicionales (para alcanzar el número de procesos (`np`) especificado en la línea de comandos de `mpirun`), llamados procesos *child*. Luego, cada uno de los procesos continúa ejecutando versiones separadas del programa `mpi_hello_world`.

`MPI_Finalize` corta la comunicación entre los procesos y limpia los estados de MPI.

Comunicador: `MPI_COMM_WORLD`

Cuando se ejecuta un programa con MPI todos los procesos se agrupan en lo que llamamos un **comunicador**. El comunicador predeterminado se llama `MPI_COMM_WORLD`. Básicamente agrupa todos los procesos cuando se inició el programa y hace que los procesos estén conectados y puedan comunicarse entre sí.

Hello world! (II)

MPI asigna un número entero a cada proceso, comienza con 0 para el proceso *parent* y aumenta cada vez que se crea un nuevo proceso. Un ID de proceso también se denomina *rank*. `MPI_Comm_rank` devuelve el rank de un proceso en un comunicador. MPI_Comm_size devuelve la cantidad de procesos.

```
int num_procs, my_id;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

printf("Hello world! I'm process %i out of %i processes\n", my_id,
num_procs);

MPI_Finalize();
```

Procesos con tareas diferentes

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

if( my_id == 0 ) { /* do some work as process 0 */ }
else if( my_id == 1 ) { /* do some work as process 1 */ }
else if( my_id == 2 ) {   /* do some work as process 2 */ }
else {  /* do this work in any remaining processes */ }

MPI_Finalize();
```

MPI Datatypes

MPI datatype	C datatype
MPI_INT	int
MPI_SHORT	short
MPI_LONG	long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_CHAR	char
MPI_BYTE	unsigned char

Rutinas de comunicación básicas

`MPI_Send`, para enviar un mensaje a otro proceso. Una vez que un programa llama a MPI_Send, se bloquea hasta que se haya realizado la transferencia de datos y la variable `buf` se pueda reutilizar de forma segura.

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
int tag,MPI_Comm comm);
```

`MPI_Recv`, para recibir un mensaje de otro proceso. `MPI_Recv` se bloquea hasta que se completa la transferencia de datos y la variable `buf` este disponible para su uso.

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

En MPR_Recv, los valores de count, source y tag se pueden especificar para permitir mensajes de longitud desconocida, de varias fuentes (MPI_ANY_SOURCE) o con varios valores de etiqueta (MPI_ANY_TAG).

Ejercicio MPI_send y MPI_recv

Implementar un programa donde el usuario de un valor de entrada y el proceso root envíe el valor al siguiente proceso. Luego, cada proceso envía el valor al siguiente proceso (por ID de proceso) hasta haber pasado por todos los procesos.

Operaciones colectivas

Las operaciones colectivas son realizadas por rutinas MPI que son llamadas por cada miembro de un grupo de procesos que desean que se realice alguna operación para ellos como grupo. Una función colectiva puede especificar la transmisión de *mensajes one-to-many*, *many-to-one*, or *many-to-many*.

MPI admite tres clases de operaciones colectivas:

- Sincronización
- Movimiento colectivo de datos
- Computación colectiva

Sincronización

La función `MPI_Barrier` se puede utilizar para sincronizar un grupo de procesos. Una vez que un proceso haya llamado a `MPI_Barrier`, se bloqueará hasta que todos los procesos del grupo también hayan llamado a `MPI_Barrier`.

```
int MPI_Barrier(MPI_Comm comm)
```

Ejemplo MPI_Barrier

```
MPI_Init(&argc, &argv);
int my_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

printf("[MPI process %d] I start waiting on the barrier.\n", my_rank);
MPI_Barrier(MPI_COMM_WORLD);

printf("[MPI process %d] I know all MPI processes have waited on the
barrier.\n", my_rank);

MPI_Finalize();
```

Movimiento colectivo de datos

Hay varias rutinas para realizar tareas de distribución de datos colectivos:

- `MPI_Bcast`: Broadcast de datos desde el proceso pasado como argumento (root) a todos los procesos del comunicador
- `MPI_Gather`: Recopilar datos de los procesos participantes en una sola estructura
- `MPI_Scatter`: Dividir una estructura en porciones y distribuir esas porciones a otros procesos
- `MPI_Allgather`: Recopilar datos de diferentes procesos en una sola estructura que luego se envía a todos los participantes (Gather-to-all)
- `MPI_Alltoall`: Recopilar datos y luego distribuir entre todos los participantes (all-to-all scatter/gather)

Ejemplo MPI_Bcast

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

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
  printf("Enter a number to broadcast:\n");
  scanf("%d", &value);
} else {
  printf("process %d: Before MPI_Bcast, value is %d\n", rank, value);
}
MPI_Bcast(&value, 1, MPI_INT, 0, MPI_COMM_WORLD);
printf("process %d: After MPI_Bcast, value is %d\n", rank, value);
MPI_Finalize();
```

Cuando los procesos están listos para compartir información con otros procesos como parte de un broadcast, TODOS ellos deben ejecutar una llamada a MPI_Bcast. No hay una llamada MPI separada para recibir un broadcast.

Ejemplo MPI_Scatter

```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype
sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm)
```

```
int process_Rank, size_Of_Comm;
int distro_Array[4] = {39, 72, 129, 42};
int scattered_Data;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Comm);
MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);

MPI_Scatter(&distro_Array, 1, MPI_INT, &scattered_Data, 1, MPI_INT, 0,
MPI_COMM_WORLD);

printf("Process has received: %d \n", scattered_Data);
MPI_Finalize();
```

Ejemplo MPI_Gather

```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

```
int root_rank = 0;
int my_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
int my_value = my_rank * 100;
printf("Process %d, my value = %d.\n", my_rank, my_value);
if(my_rank == root_rank) {
  int buffer[4];
  MPI_Gather(&my_value, 1, MPI_INT, buffer, 1, MPI_INT, root_rank,
MPI_COMM_WORLD);
  printf("Values collected on process %d: %d, %d, %d, %d.\n", my_rank,
buffer[0], buffer[1], buffer[2], buffer[3]);
}
else {
  MPI_Gather(&my_value, 1, MPI_INT, NULL, 0, MPI_INT, root_rank,
MPI_COMM_WORLD);
}
```

Rutinas de computación colectiva

El cómputo colectivo es similar al movimiento de datos colectivos con la característica adicional de que los datos pueden modificarse a medida que se mueven. Las siguientes rutinas se pueden utilizar para el cálculo colectivo.

- `MPI_Reduce`: Realizar una operación de reducción. Es decir, aplica alguna operación a algún operando en cada proceso participante. Por ejemplo, agregar un número entero que resida en cada proceso y coloque el resultado en un proceso especificado en la lista de argumentos MPI_Reduce.
- `MPI_Allreduce`: Realizar una reducción dejando el resultado en todos los procesos participantes.
- `MPI_Reduce_scatter`: Realizar una reducción y luego distribuir el resultado.
- `MPI_Scan`: Realizar una reducción dejando resultados parciales (calculados hasta el punto de participación de un proceso en el recorrido del árbol de reducción) en cada proceso participante.

Operaciones de computación colectiva integradas en MPI

Operation handle	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_LAND	Logical AND
MPI_LOR	Logical OR
MPI_LXOR	Logical Exclusive OR
MPI_BAND	Bitwise AND
MPI_BOR	Bitwise OR
MPI_BXOR	Bitwise Exclusive OR
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and

MPI_Reduce

La subrutina `MPI_Reduce` combina datos de todos los procesos en un comunicador usando una de varias operaciones de reducción para producir un único resultado que aparece en un proceso objetivo específico.

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

Cuando los procesos están listos para compartir información con otros procesos como parte de una reducción de datos, todos los procesos participantes ejecutan una llamada a `MPI_Reduce`, que usa datos locales para calcular la parte de la operación de reducción de cada proceso y comunica el resultado local a otros procesos. Solo target_process_ID recibe el resultado final.

Ejemplo MPI_Reduce

Cada proceso MPI envía su *rank* a reducción, el proceso root recopila el resultado

```
// Get my rank
int my_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

int reduction_result = 0;
MPI_Reduce(&my_rank, &reduction_result, 1, MPI_INT, MPI_SUM,
root_rank, MPI_COMM_WORLD);

if(my_rank == root_rank)
{
  printf("The sum of all ranks is %d.\n", reduction_result);
}
```

Bibliografía

- Links:
  - https://www.mpi-forum.org/
  - https://www.open-mpi.org/
- Libros:
  - Using MPI: Portable Parallel Programming with the Message-Passing Interface, by Gropp, Lusk, and Skjellum, MIT Press, 1994.
  - MPI: The Complete Reference, by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1996.
  - Designing and Building Parallel Programs, by Ian Foster, Addison-Wesley, 1995.
  - Parallel Programming with MPI, by Peter Pacheco, Morgan-Kaufmann, 1997.
  - MPI: The Complete Reference Vol 1 and 2,MIT Press, 1998 (Fall).