1. **OpenMP y MPI de cual elemento taxonómico anterior es**.

**OpenMP**

**Thomas Sterling (NASA)**

Ayuda a programar para arquitecturas paralelas de memoria compartida.  
Paraleliza nuestro software con menor esfuerzo.  
Para computadoras multiprocesadas utilizamos OpenMP porque la memoria es compartida.  
Gestiona los hilos.  
Vamos a definir directivas de pre-procesador con las cuales el pre-procesador va a agregar todo el código en C necesario.

**Cómo usarlo:**  
Hay que incluir la biblioteca omp.h, la que nos va a habilitar funciones como omp\_get\_num\_threads() para saber la cantidad de hilos que tenemos.  
Tenemos una variable de entorno que la configuramos en el shell que es $OMP\_NUM\_THREADS1, donde podemos definir la cantidad de hilos que queremos que se usen. Si no se indica esto, OpenMP va a usar la cantidad de hilos que pueda (el máximo de nuestro procesador).  
Toda la gestión de pasar de secuencial a paralelo lo hace automáticamente OpenMP. Para esto, lo gestiona con #pragma omp parallel.  
1Por ejemplo, export OMP\_NUM\_THREADS=2.

**Cómo compilar:**  
gcc -fopenmp programa.c  
Agregar -lm si se necesita la biblioteca matemática.

**Los tipos de variables en OpenMP son tres:**

* **Shared:** Todos los hilos acceden a la misma variable.
* **Private:** Cada hilo tiene su propia variable que, particularmente, se llaman igual en todos los hilos.
* **Reduction:** La variable, durante toda la ejecución de la parte paralela, se comporta como privada (cada hilo tiene su copia), pero cuando terminan, todos devuelven el valor final sobre la misma variable. Ejemplo de uso: buscar el máximo (todos los hilos buscan el máximo y después el hilo principal recolecta todos estos máximos y elige el máximo total).  
  A partir de muchos datos se quiere obtener un resultado.  
  Las operaciones que se pueden realizar son: +, \*, -, ^, |, ||, & o &&.  
  Ejemplo de *reduction*:

sum = 0.0

//sum es la variable de reducción y "+" es la operación matemática a realizar:

#pragma omp parallel for reduction(+:sum)

for (int i=0; i<20; i++)

sum = sum + (a[i] \* b[i]);

**Ejemplo de for:**

#pragma omp parallel

{

p=5;

#pragma omp for

{

for (i=0; i<24; i++)

...

}

}

Se encarga solo de asignar partes iguales del for a los distintos hilos.

#pragma omp for schedule(guided, chunk) nowait

chunk es la división que queremos que tenga. nowait sirve para que el hilo que ya terminó su tarea, se cierre. Si usamos wait entonces se espera hasta el último hilo.

Otra forma de paralelización en OpenMP son las *sections*: No se paraleliza lo mismo sino que cada hilo hace cosas distintas.

También tenemos el uso de sección crítica con *critical*: Hace que la tarea marcada como "crítica" sea realizada por un hilo por vez. Garantiza **exclusión mutua**.

#pragma omp critical

x = 2\*x +1;

**MPI**

Nos resuelve la comunicación independientemente de los núcleos de procesamiento.  
Es una API que nos ayuda a establecer comunicaciones punto a punto.  
Hay tipos de datos estandarizados

**Cómo usarlo:**  
Hay que incluir la biblioteca mpi.h.  
Cuando arrancamos nuestro programa, este empieza en monohilo. Cuando queremos empezar la parte paralela, hay que inicializarlo con MPI\_Init(&argc, &argv), que es la que hace que se empiecen a comunicar todos los procesos. Inicializa la comunicación entre todas las computadoras e indicamos que en ese punto comienza la comunicación entre los diferentes procesos.  
Una vez que termina la ejecución paralela, hay que cerrar la comunicación con MPI\_Finalize(), que va a ser el encargado de desconectar a todos los procesos.

**Cómo compilar:**  
Utilizando el wrapper:  
mpicc -o executable mpicode.c

**Cómo ejecutar:**  
La ejecución también se realiza a través de un wrapper:  
mpirun -np 2 ./a.out  
Con el flag -np indicamos cuantos procesos queremos que se ejecuten.  
Es importante tener cargado mpd, que es el demonio de MPI, que va a estar escuchando y cuando se ejecute mpirun va a ser el encargado de levantar el proceso y canalizar todas las ejecuciones del cluster.

**Cómo instalar el demonio:**

usuario@lac300:MPI$ su -

Contraseña:

root@lac300:~# apt search mpi | grep ^mpi

root@lac300:~# apt install mpich

root@lac300:~# apt clean

**Ejemplo Hola Mundo:**

#include <stdio.h>

#include <mpi.h>

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

int ierror, rank, size;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank); //Obtiene el número de proceso.

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size); //Obtiene la cantidad de procesos en ejecución.

printf("Hello world! I am %d out of %d.\n", rank, size);

MPI\_Finalize();

return 0;

}

**Comunicadores (communicators)**  
Todos los procesos forman parte al menos de un comunicador por defecto (MPI\_COMM\_WORLD), que es un canal de difusión.  
Se pueden crear nuevos comunicadores y se puede elegir que procesos pertenecen o no, por ejemplo, creadores de datos por un lado y consumidores de datos por otro.  
Podemos saber cuantos nodos (procesos) son parte del communicator con MPI\_Comm\_size().  
Si agregamos un mismo proceso en dos comunicadores distintos, este puede tener un número de proceso distinto en cada comunicador, y se averigua con MPI\_COMM\_RANK.

**Tipos de comunicaciones:**

* Punto a punto: de un proceso a otro. Las primitivas más simples son MPI\_Send y MPI\_Recv, a las cuales les tenemos que pasar como argumento:
  + rango del receptor
  + rango del que envía
  + tag para etiquetar la comunicación
  + el medio por el que nos vamos a comunicar

MPI\_Send es bloqueante, o sea que hasta que no se haya entregado el mensaje, el proceso queda bloqueado hasta que lo lean. Ejemplo:

int MPI\_Send(

void \*message, //Puntero al mensaje (desde un buffer).

int count, //Tamaño del mensaje. Cantidad de datos.

MPI\_Datatype datatype, //Tipo de datos a enviar.

int dest, //Número del proceso destino.

int tag, //Etiqueta.

MPI\_Comm comm) //Comunicador a utilizar.

Los tipos de datos son:

| **MPI datatype** | **C datatype** |
| --- | --- |
| MPI\_CHAR | signed char |
| MPI\_SHORT | signed short int |
| MPI\_INT | signed int |
| MPI\_LONG | signed long int |
| MPI\_UNSIGNED\_CHAR | unsigned char |
| MPI\_UNSIGNED\_SHORT | unsigned short int |
| MPI\_UNSIGNED | unsigned int |
| MPI\_UNSIGNED\_LONG | unsigned long int |
| MPI\_FLOAT | float |
| MPI\_DOUBLE | double |
| MPI\_LONG\_DOUBLE | long double |
| MPI\_BYTE |  |
| MPI\_PACKED |  |

Deberíamos usar estos tipos de datos ya que la idea es poder tener distintos terminales con distintos sistemas operativos y arquitecturas y que todas puedan trabajar como un único cluster.

MPI\_Recv() es parecido, pero con algunas diferencias:

int MPI\_Recv(

void \*message,

int count,

MPI\_Datatype dadadtype,

int source, //De qué proceso se quiere escuchar.

int tag, //Con qué tag escuchar.

MPI\_Comm comm,

MPI\_Status \*status) //Estructura de datos donde se ve cómo termió la comunicación.

**MPI tiene llamadas colectivas**:

* De sincronización:
  + Barrier: Indica un punto en el software en el que queremos que todos se pongan de acuerdo. Entonces los procesos más rápidos esperan a los más lentos. Se usa con MPI\_Barrier(MPI\_Comm comm) indicándole el comunicador de los procesos que deben usar la barrera. Todos los procesos del comunicador deben tener en su código la barrera implementada, si hay uno que no lo tiene, se bloquean (van a estar esperándolo).
* De comunicación:
  + Broadcast: De uno a muchos (los que están en el comunicador).  
    Su uso es:
* int MPI\_Bcast(
* void \*message,
* int count,
* MPI\_Datatype datatype,
* int root, //Se indica quien es el proceso que envía el mensaje.
* MPI\_Comm comm

)

* + Gather & Scatter: Con Scatter un proceso tiene toda la información y la distribuye entre el resto, incluido el; y con Gather, los procesos tienen la información y un proceso la recolecta, quedándose también con su información.  
    Prototipo de Scatter:

int MPI\_Scatter(

void \*sendbuf,

int send\_count,

MPI\_Datatype send\_type,

void \*recvbuf,

int recv\_count,

MPI\_Datatype recv\_type,

int root,

MPI\_Comm comm

)

El prototipo de Gather es igual al de Scatter.

* + All Gather: Todos recolectan la información que cada proceso tiene. Como todos los procesos tienen que hacer lo mismo, no hay proceso *root*.
* De reducción:
  + Reduce: Cuando solo un proceso hace la reducción. El proceso *root* es el que se queda con la información.  
    Prototipo de Reduce:
* int MPI\_Reduce(
* void \*operand,
* void \*result,
* int count,
* MPI\_Datatype datatype,
* MPI\_Op operator,
* int root,
* MPI\_Comm comm

)

* + All Reduce: Todos los procesos hacen la reducción. Todos los procesos tienen un dato y luego todos obtienen el resultado de la reducción. No hay *root*, pero el resto del prototipo de función es igual al de Reduce.

Los operandos que se pueden utilizar son:

| **Nombre de la operación** | **Significado** |
| --- | --- |
| MPI\_MAX | Maximum |
| MPI\_MIN | Minimum |
| MPI\_SUM | Sum |
| MPI\_PROD | Product |
| MPI\_LAND | Logical And |
| MPI\_BAND | Bitwise And |
| MPI\_LOR | Logical Or |
| MPI\_BOR | Bitwise Or |
| MPI\_LXOR | Logical XOR |
| MPI\_BXOR | Bitwise XOR |
| MPI\_MAXLOC | Maximum and location of max. |
| MPI\_MINLOC | Maximum and location of min. |

Instalar para ejecutar:

root@lac300:~# apt install mpi-default-bin

**Direfencia entre OpenMP y MPI**

OpenMP es un modelo de memoria compartida basada en hilos.  
MPI (Message Passing Interface) es un modelo de memoria distribuida, inclusive en distintas máquinas.  
Hilos = OpenMP  
Procesos = MPI