Computación paralela y distribuida: OpenMP y MPI

Alex Di Genova

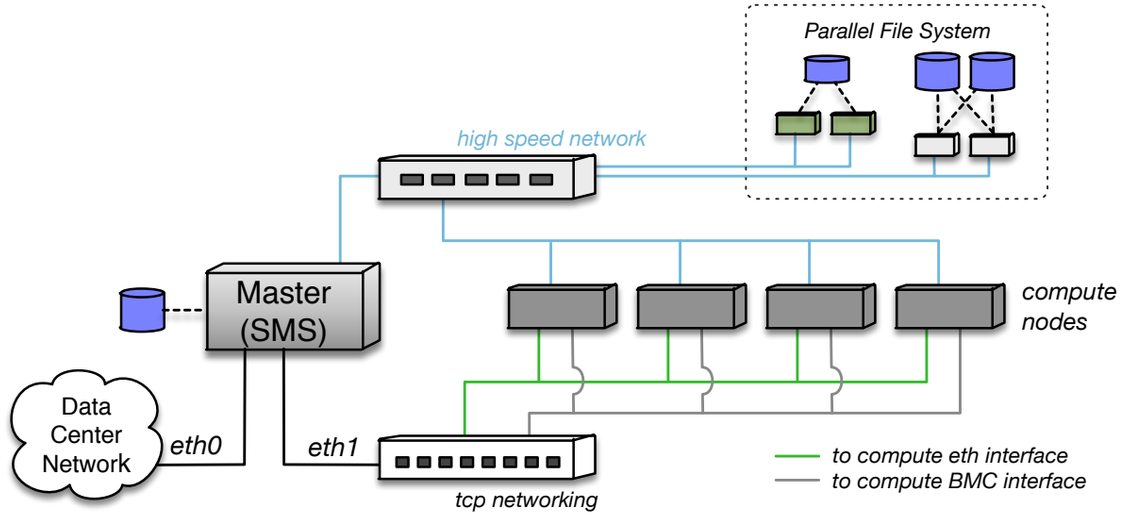
30/04/2024

Sistemas distribuidos

Vista Global

1 Maquina
X cores

- **Hilos (pthread)**
- **OpenMP**



1 Cluster
X maquinas
Y Cores
?

- **PTHREAD (hilos)**
- **OpenMP**
- **MPI**
- **Hadoop/Nextflow**

NVIDIA QUADRO P6000
- GPUs 3840
- RAM 24Gb

1 Tarjeta
Y Cores
?

- **CUDA**

Complete example

Matrix multiplication

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{bmatrix}$$

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in} + b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

$$\begin{pmatrix} 2 & 7 & 3 \\ 1 & 5 & 8 \\ 0 & 4 & 1 \end{pmatrix} \times \begin{pmatrix} 3 & 0 & 1 \\ 2 & 1 & 0 \\ 1 & 2 & 4 \end{pmatrix} = \begin{pmatrix} 23 & 13 & 14 \\ 21 & 21 & 33 \\ 9 & 6 & 4 \end{pmatrix}$$

https://github.com/adigenova/DCBI1302/tree/main/code

OpenMP

Introducción y funciones

OpenMP

Introduction

- **Definición : OpenMP**
  - OpenMP es una interfaz de programación de aplicaciones (API) para la paralelización de sistemas de memoria compartida, utilizando C, C++ o Fortran.
  - La API consta de directivas de compilador para especificar y controlar la paralelización, aumentada con funciones de tiempo de ejecución y variables de entorno.
  - Corresponde al usuario identificar el paralelismo e insertar las estructuras de control apropiadas en el programa (directivas).
  - En C/C++, la directiva se basa en la construcción **#pragma omp.**
- **Syntax**
  - **#pragma omp parallel** [clause[clause], …] new-line

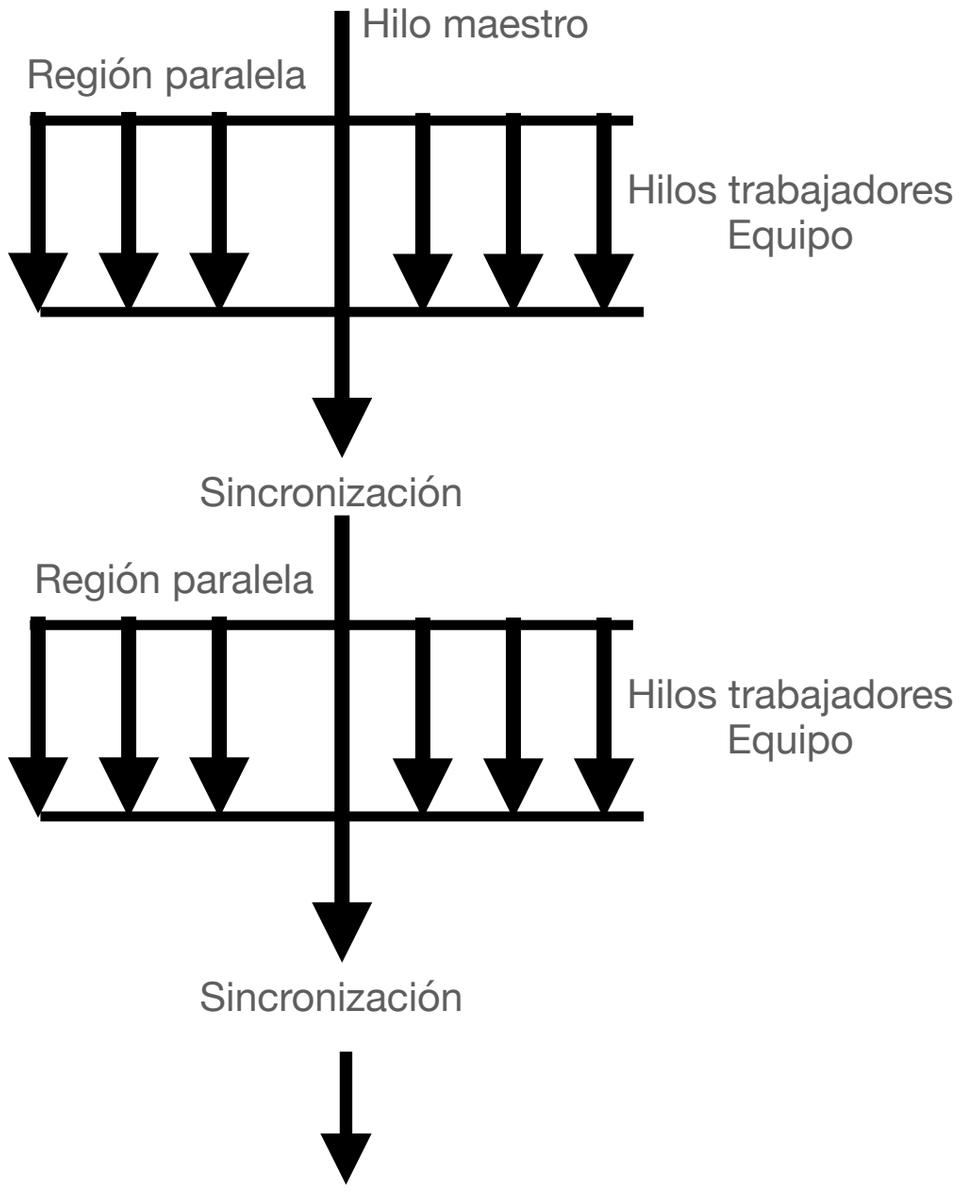
    Structured block
  - Es responsabilidad del programador identificar qué parte(s) del código se selecciona(n) para ejecutar en paralelo y usar las diversas construcciones para garantizar resultados correctos.
  - También se debe especificar la naturaleza (privada o compartida) o el "alcance" de las variables.

OpenMP

La región paralela

- **Región paralela**
  - Un programa paralelo en OpenMP comienza y termina con la región paralela. Es la piedra angular de OpenMP.
  - No hay límite en la cantidad de regiones paralelas, pero por razones de rendimiento es mejor mantener la cantidad de regiones al mínimo y hacerlas lo más grandes posible.
  - El hilo que encuentra la región paralela se llama hilo **maestro**. Crea los hilos adicionales y está a cargo de la ejecución general.
- Los hilos que están activos dentro de una región paralela se denominan **equipos**. Varios equipos pueden estar activos simultáneamente.
  - **OMP_NUM_THREADS** (variable ambiente)
  - **omp_set_num_threads()**
    - Función para modificar el numero de hilos
  - Clausula **num_threads(<nt>)**
- Las instrucciones dentro de la región paralela son ejecutadas por todos los hilos.
- Fuera de las regiones paralelas, el hilo maestro ejecuta las partes del codigo en forma serial.
- La sincronización de hilos se produce en la barrera implícita al final de cada región paralela.

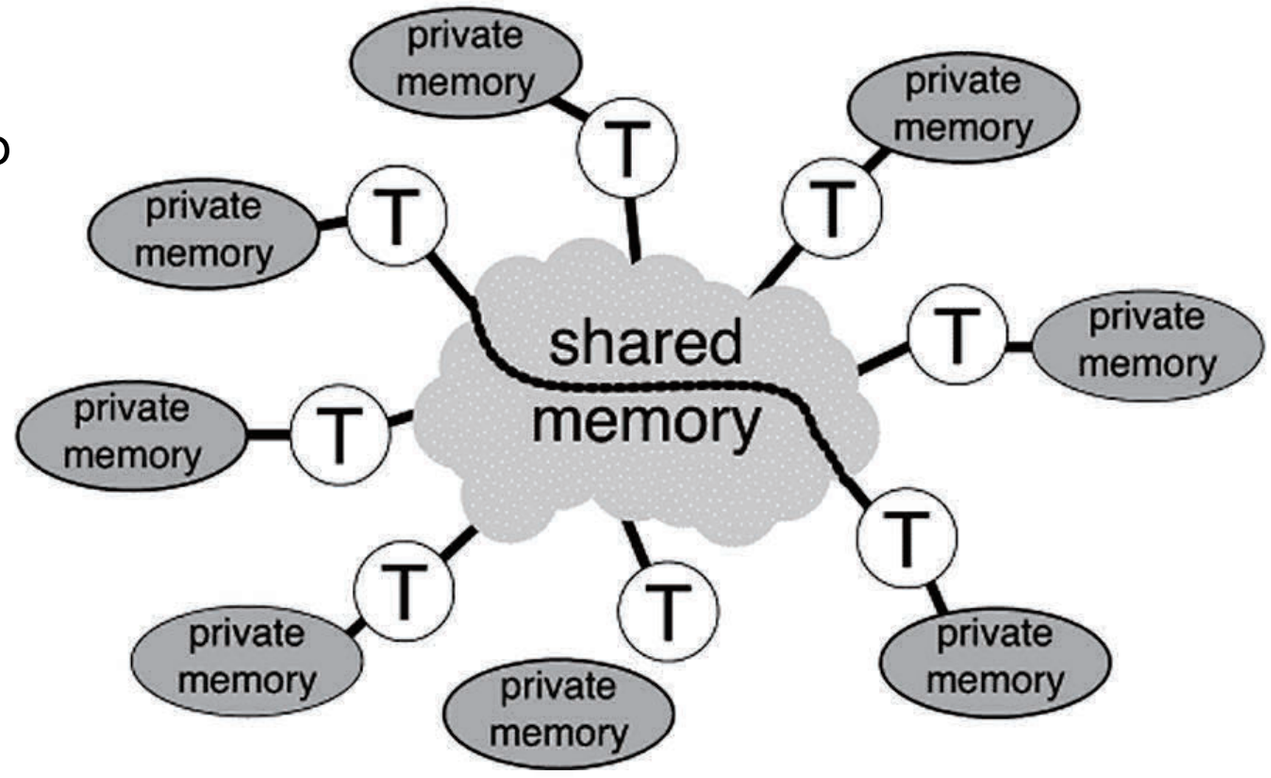
Modelo de Ejecución OpenMP (fork-join)



OpenMP

Modelo de Memoria

- **Modelo de Memoria**
  - Un programa OpenMP tiene dos tipos elementales diferentes de memoria: privada y compartida.
  - **Variables privadas.** Cada hilo tiene acceso único a su memoria privada y ningún otro hilo puede interferir. Nunca existe el riesgo de un conflicto de acceso con otro hilo. Aunque varios hilos pueden usar el mismo nombre para una variable privada, estas variables se almacenan en diferentes ubicaciones de memoria.
  - **Variables compartidas.** Cada hilo puede leer, así como escribir, cualquier variable compartida. Es responsabilidad del programador manejar correctamente esta situación.
    - variables globales se comparten por defecto.



OpenMP

Construcciones de trabajo compartido

- Una construcción de trabajo compartido debe colocarse dentro de una región paralela. Al encontrar una construcción de trabajo compartido, el OpenMP distribuye el trabajo a realizar entre los hilos activos en la región paralela.
- **Construcción de ciclos**, proporciona una forma sencilla de asignar el trabajo asociado con las iteraciones de ciclos a los hilos.
  - **#pragma omp for [clase,[], …clause]**
  - Las iteraciones del ciclo se distribuyen en los hilos y se ejecutan en paralelo.
- **Construcción de secciones,** son ideales para llamar a diferentes funciones en paralelo.

  #pragma omp sections [clause[[,]],clause]…]

  {

  #pragma omp section

  bloque de código

  #pragma omp section

  bloque de código

  }
- **Construcciones unicas,** especifica que el bloque dado es ejecutado por un solo hilo. No se especifica qué hilo. Otros hilos omiten el bloque y esperan (barrera) a que finalice la contrucción.

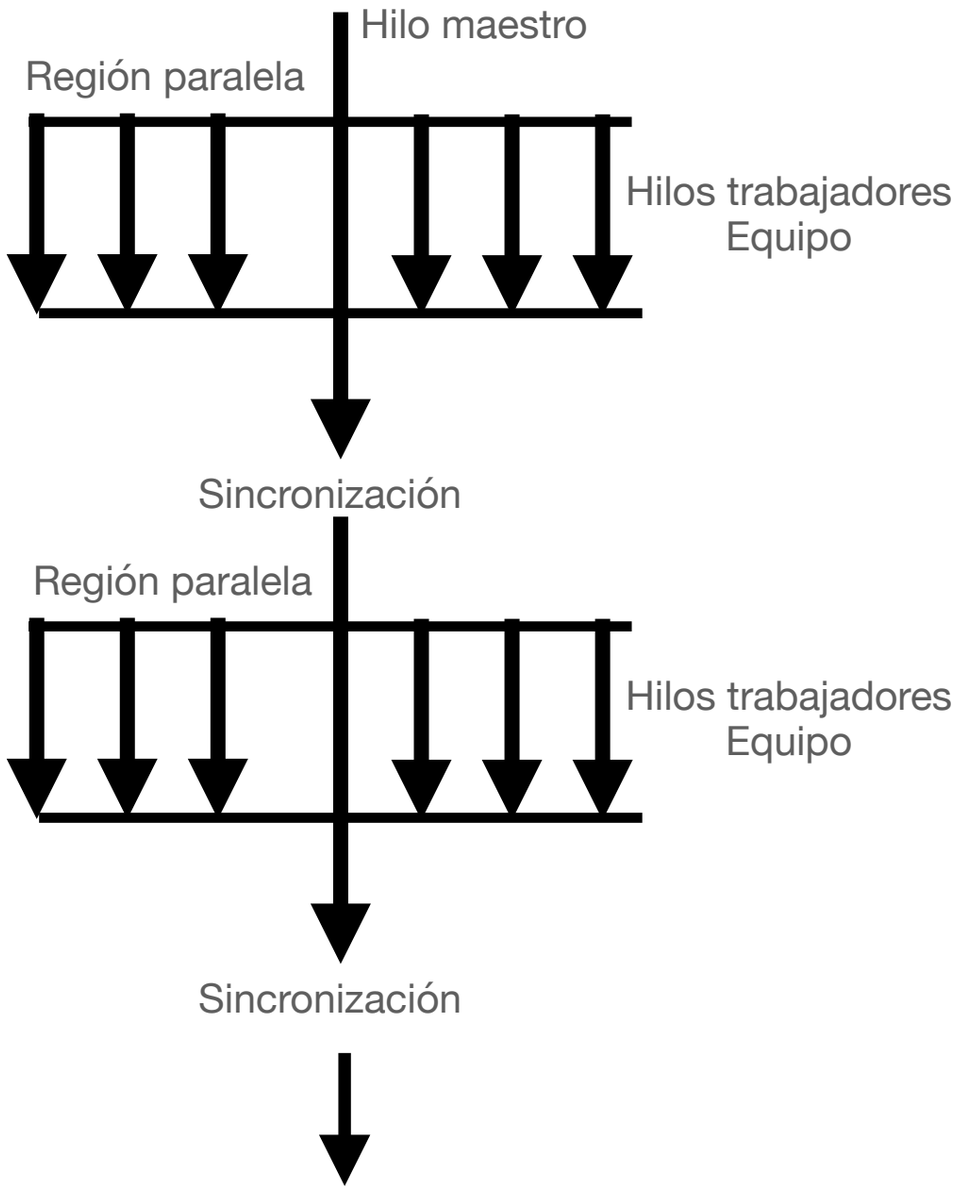
  #pragma omp single

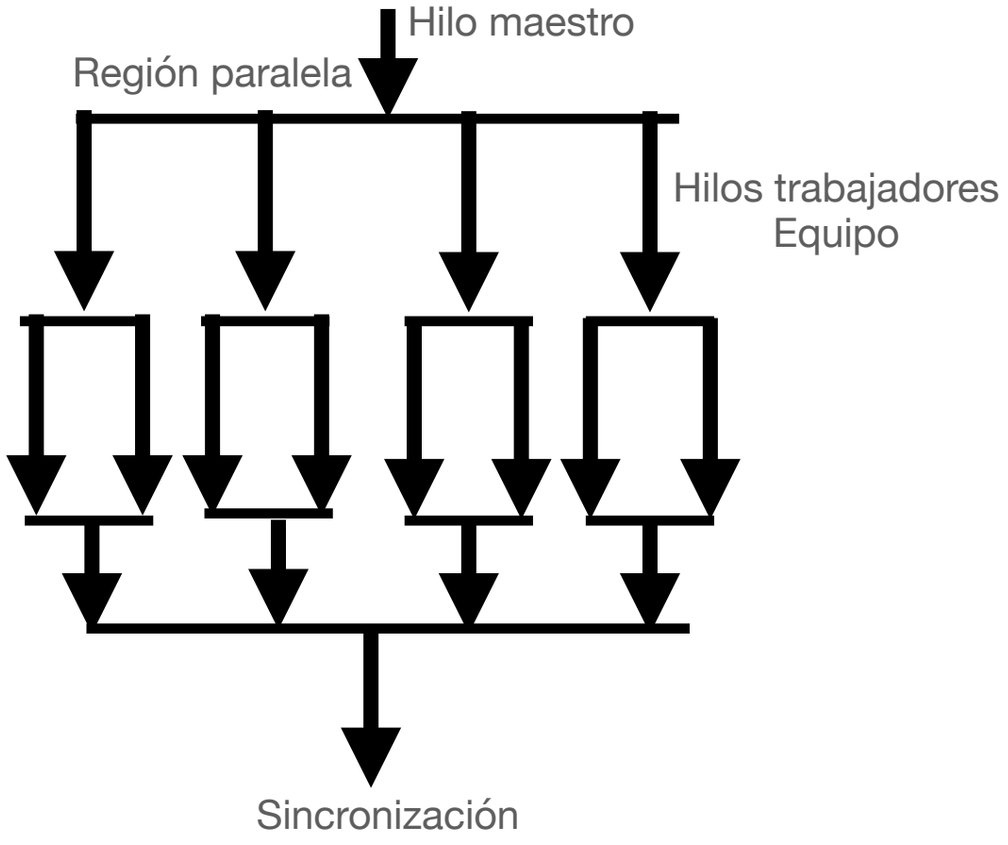
  Bloque de código

OpenMP

Paralelismo anidado

Modelo de Ejecución OpenMP (fork-join)






```
#pragma omp parallel num_threads(1)
{
  Work1();

  #pragma omp parallel num_threads(5)
  {  //1 x 5 = 5 threads
     Work2();
  }
}
```

OpenMP

Sincronización

- **Contrucción barrera,** obliga a todos los hilos a esperar hasta que todos hayan alcanzado la región de barrera en el programa (**default**).

- **Construccion critica**, Una región crítica es un bloque de código ejecutado por todos los hilos, pero se garantiza que solo un hilo a la vez puede estar activo en la región.

- **La construcción atómica,** se utiliza para garantizar el acceso mutuamente excluyente a una ubicación de memoria específica, representada a través de una variable. Se garantiza que el acceso a esta ubicación será atómico.

OpenMP

Funciones de tiempo de ejecución

- Estas funciones se pueden usar para consultar la configuración y también sobreescribir los valores iniciales, ya sea establecidos de forma predeterminada o especificados a través de variables de ambiente definidas antes del inicio del programa.

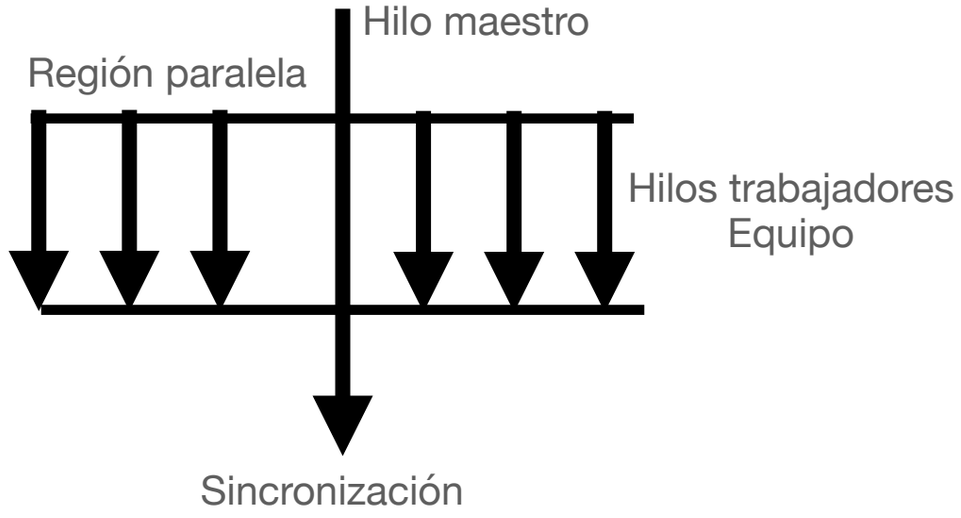
    - Un ejemplo es el número de hilos utilizados para ejecutar una región paralela. El valor inicial depende de la implementación, pero a través de la variable de entorno OMP_NUM_THREADS, este número puede establecerse explícitamente antes de que se inicie el programa. Durante la ejecución del programa, la función **omp_set_num_threads()** se puede usar para aumentar o disminuir el número de hilos que se usarán en las siguientes regiones paralelas.

Función	Descripción
omp_set_num_threads	Establece el número de hilos.
omp_get_num_threads	Número de hilos en el equipo actual.
omp_get_max_threads	número de hilos en la siguiente región paralela.
omp_get_num_procs	Número de procesadores disponibles para el programa.
omp_get_thread_num	Número de hilo dentro de la región paralela.
omp_in_parallel	Comprueba si está dentro de una región paralela.
omp_get_dynamic	Comprueba si el ajuste del hilo está habilitado.
omp_set_dynamic	Habilita o deshabilita el ajuste del hilos.
omp_get_nested	Comprueba si el paralelismo anidado está habilitado.
omp_set_nested	Habilita o deshabilita el paralelismo anidado.

Ejemplos OpenMP

Hello world

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

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

    // Comienzo de la region paralela
    #pragma omp parallel
    {
        printf("Hola Mundo... desde hilo = %d \n",
omp_get_thread_num());
    }
    // Fin de la region paralela
}
```




```
#definimos los hilos a utilizar
%env OMP_NUM_THREADS=3
```

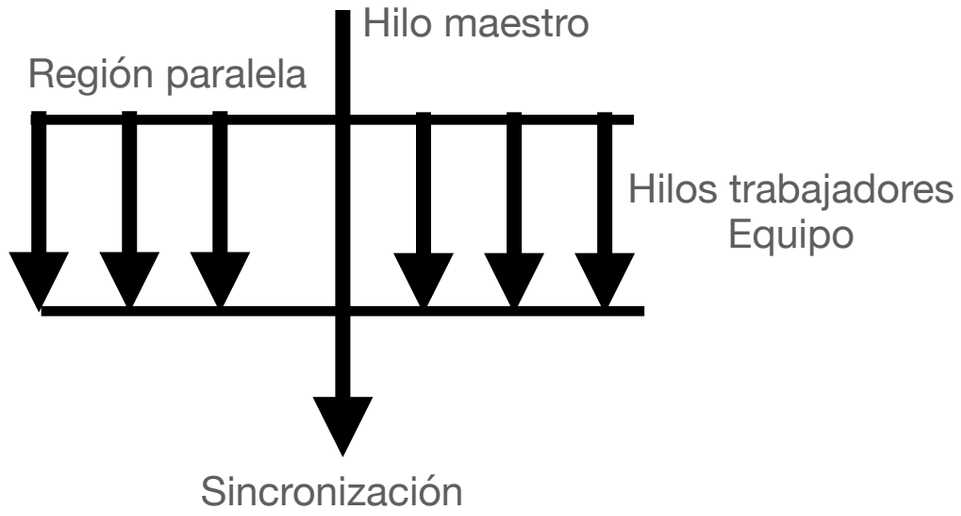
```
!gcc -o holamundo_openmp -fopenmp holamundo_openmp.c
```

```
!./holamundo_openmp
```

Hola Mundo... desde hilo = 1
Hola Mundo... desde hilo = 0
Hola Mundo... desde hilo = 2

Ejemplos OpenMP

For




```
#include <omp.h>
#include <stdio.h>

int main() {
  int k;

#pragma omp parallel
{
    for (k = 0; k < 10; k++)
      printf("Itr: %d tid=%d\n", k, omp_get_thread_num());
}
  return 0;
}
```

```
!gcc -o for_openmp1 -fopenmp for_openmp1.c
```

```
%env OMP_NUM_THREADS=3
!./for_openmp1
```

```
env: OMP_NUM_THREADS=3
Itr: 0 tid=0
Itr: 1 tid=0
Itr: 2 tid=0
Itr: 3 tid=0
Itr: 4 tid=0
Itr: 0 tid=1
Itr: 1 tid=1
Itr: 2 tid=1
Itr: 3 tid=1
Itr: 4 tid=1
```

```
#include <omp.h>
#include <stdio.h>

int main() {
  int k;

#pragma omp parallel
{
#pragma omp for
    for (k = 0; k < 10; k++)
      printf("Itr: %d tid=%d\n", k,
omp_get_thread_num());
}
  return 0;
}
```

```
env: OMP_NUM_THREADS=2
Itr: 5 tid=1
Itr: 6 tid=1
Itr: 7 tid=1
Itr: 8 tid=1
Itr: 9 tid=1
Itr: 0 tid=0
Itr: 1 tid=0
Itr: 2 tid=0
Itr: 3 tid=0
Itr: 4 tid=0
```

Ejemplos OpenMP

Secciones paralela

```
#include <omp.h>
#include <stdio.h>
#include <unistd.h>

void Work1(){
    printf("executing work 1 hilo:%d\n",  omp_get_thread_num());
    sleep(1);
}
void Work2(){
    printf("executing work 2 hilo:%d\n",  omp_get_thread_num());
    sleep(1);
}

void Work3(){
    printf("executing work 3 hilo:%d\n",  omp_get_thread_num());

}
void Work4(){
    printf("executing work 4 hilo:%d\n",  omp_get_thread_num());
    sleep(1);
}

int main() {

 #pragma omp parallel sections
 {
   { Work1(); }
   #pragma omp section
   { Work2();
     Work3();}
   #pragma omp section
   { Work4(); }
 }

 return 0;
}
```

```
%env OMP_NUM_THREADS=5
!./omp_sections
```

```
env: OMP_NUM_THREADS=5
executing work 1 hilo:1
executing work 2 hilo:2
executing work 4 hilo:3
executing work 3 hilo:2
```

```
%env OMP_NUM_THREADS=10
!./omp_single
```

```
int main() {

 #pragma omp parallel
 {
   Work1();
   #pragma omp single
   { Work2();
     Work3();
   }
    Work4();
 }

 return 0;
}
```

```
env: OMP_NUM_THREADS=10
executing work 1 hilo:0
executing work 1 hilo:1
executing work 1 hilo:3
executing work 1 hilo:2
executing work 1 hilo:4
executing work 1 hilo:5
executing work 1 hilo:6
executing work 1 hilo:7
executing work 1 hilo:8
executing work 1 hilo:9
executing work 2 hilo:3
executing work 3 hilo:3
executing work 4 hilo:3
executing work 4 hilo:4
executing work 4 hilo:1
executing work 4 hilo:8
executing work 4 hilo:0
executing work 4 hilo:7
executing work 4 hilo:2
executing work 4 hilo:9
executing work 4 hilo:5
executing work 4 hilo:6
```

Ejemplos OpenMP

For anidados y sección critica

```
int main() {

 #pragma omp parallel num_threads(1)
 {
   Work1();

   #pragma omp parallel num_threads(5)
   {  //1 x 5 = 5 threads
      Work2();
   }
 }

 return 0;
}
```

```
#%env OMP_NUM_THREADS=3
!./omp_nested
```

```
executing work 1 hilo:0
executing work 2 hilo:2
executing work 2 hilo:4
executing work 2 hilo:0
executing work 2 hilo:1
executing work 2 hilo:3
```

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
  int k;
 int sum=0;

#pragma omp parallel for shared(sum)
  for (k = 0; k < 10; k++) {
    int c=rand()%50;
    printf("Itr: %d tid=%d, my_contri=%d\n", k,
omp_get_thread_num(),c);
  #pragma omp critical
  sum+=c;
}

printf("Sum=%d",sum);
  return 0;
}
```

```
%env OMP_NUM_THREADS=5
!./sync_openmp
```

```
env: OMP_NUM_THREADS=5
Itr: 2 tid=1, my_contri=33
Itr: 3 tid=1, my_contri=35
Itr: 6 tid=3, my_contri=27
Itr: 7 tid=3, my_contri=36
Itr: 8 tid=4, my_contri=15
Itr: 9 tid=4, my_contri=42
Itr: 4 tid=2, my_contri=36
Itr: 5 tid=2, my_contri=49
Itr: 0 tid=0, my_contri=43
Itr: 1 tid=0, my_contri=21
Sum=337
```

OpenMP

Ejemplos

```
#include <stdio.h>
#include <omp.h>

int main() {
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int n = 10;
    int sum = 0;
    //
    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < n; i++) {
        sum += arr[i];
        printf("Thread %d: arr[%d] = %d\n",
omp_get_thread_num(), i, arr[i]);
    }

    printf("Sum = %d\n", sum);
    return 0;
}
```

```
#include <stdio.h>
#include <omp.h>

#define N 10000
double data[N];

// Función para calcular el promedio de un subconjunto del
arreglo
double calcularPromedio(int start, int end) {
    double sum = 0.0;
    for (int i = start; i < end; i++) {
        sum += data[i];
    }
    return sum / (end - start);
}

int main() {
    // Inicializar el arreglo de datos
    for (int i = 0; i < N; i++) {
        data[i] = i;
    }
    int num_threads = 4; // Número de hilos a utilizar
    double total_average = 0.0;

    #pragma omp parallel num_threads(num_threads)
    {
        int thread_id = omp_get_thread_num();
        int chunk_size = N / num_threads;
        int start = thread_id * chunk_size;
        int end = (thread_id == num_threads - 1) ? N : start +
chunk_size;

        double local_average = calcularPromedio(start, end);

        #pragma omp critical
        total_average += local_average;

        printf("Thread %d: Promedio Local = %f\n", thread_id,
local_average);
    }

    total_average /= num_threads;
    printf(“Promedio Total = %f\n", total_average);

    return 0;
}
```

OpenMP

Ejemplos

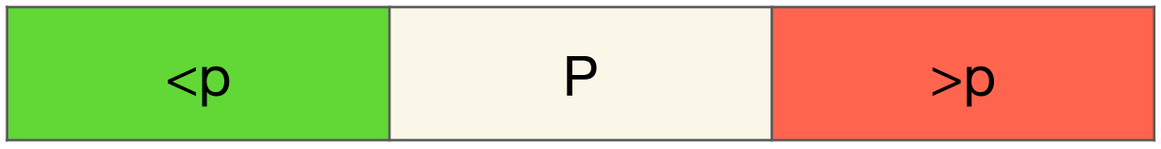
- Dado un texto, contar la frecuecia de los caracteres utilizando dos hilos de OpenMP.
- Considerar que el total de caracteres ascii es 256.

```
#include <stdio.h>
#include <string.h>
#include <omp.h>

#define MAX_TEXT_SIZE 10000

// Función para contar la frecuencia de caracteres en un texto
void countCharacterFrequency(const char* text, int* charCount) {
    int chartmp[256] = {0};
    for (int i = 0; text[i] != '\0'; i++) {
        char c = text[i];
        chartmp[c]++;
    }
    //sumamos el resultado en charCount
    #pragma omp critical
    for(int j = 0; j<256; j++){
        charCount[j]+=chartmp[j];
    }

}

int main() {
    char text[MAX_TEXT_SIZE];
    int charCount[256] = {0}; // 256  caracteres asccii
    // Texto
    const char* inputText = "Los recursos en línea ofrecen una variedad
de textos en español para mejorar
    la comprensión y el aprendizaje del idioma. Estos textos pueden ser
útiles para estudiantes de diferentes niveles de habilidad.";

    // Divide the text into two segments
    char* segment1 = strncpy(text, inputText, strlen(inputText) / 2);
    char* segment2 = strncpy(text + (strlen(inputText) / 2), inputText +
( strlen(inputText) / 2),  strlen(inputText) / 2);

    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                countCharacterFrequency(segment1, charCount);
            }
            #pragma omp section
            {
                countCharacterFrequency(segment2, charCount);
            }
        }
    }

    printf("Frecuencia de caracteres:\n");
    for (int i = 0; i < 256; i++) {
        if (charCount[i] > 0) {
            printf("Caracter '%c' aparece %d veces\n", (char)i,
charCount[i]);
        }
    }

    return 0;
}
```

OpenMP

Ejemplos

- **quicksort**
- Ejemplo clásico de la aplicación del principio de dividir para conquistar.
- algoritmo:
    - Primero se elige un elemento al azar, que se denomina el pivote.
    - El arreglo a ordenar se reordena dejando a la izquierda a los elementos menores que el pivote, el pivote al medio, y a la derecha los elementos mayores que el pivote:



- Luego cada sub-arreglo se ordena recursivamente.
- La recursividad se detiene en principio hay 1 elemento.

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define ARRAY_SIZE 1000000

void quickSort(int arr[], int left, int right) {
    if (left < right) {
        int pivot = arr[right];
        int i = left - 1;

        for (int j = left; j < right; j++) {
            if (arr[j] < pivot) {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        int temp = arr[i + 1];
        arr[i + 1] = arr[right];
        arr[right] = temp;

        int partition = i + 1;

        #pragma omp parallel sections
        {
            #pragma omp section
            quickSort(arr, left, partition - 1);
            #pragma omp section
            quickSort(arr, partition + 1, right);
        }
    }
}

int main() {
    int arr[ARRAY_SIZE];

    // Inicializar el arreglo con números aleatorios
    for (int i = 0; i < ARRAY_SIZE; i++) {
        arr[i] = rand() % 10000;
    }

    printf("Arreglo no ordenado:\n");
    for (int i = 0; i < 100; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Ordenar el arreglo usando Quick Sort paralelizado
    #pragma omp parallel
    {
        #pragma omp single
        quickSort(arr, 0, ARRAY_SIZE - 1);
    }

    printf("Arreglo ordenado:\n");
    for (int i = 0; i < 100; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

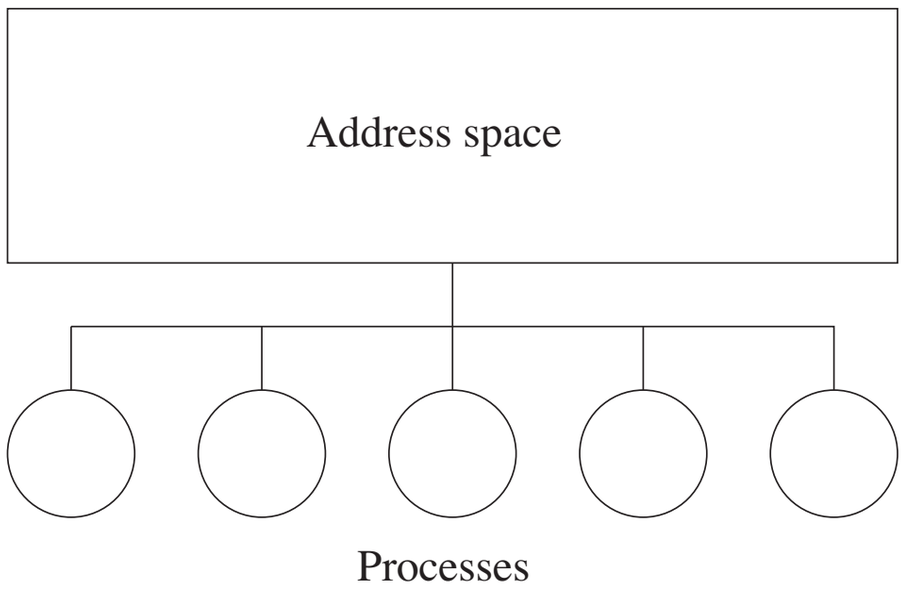
MPI

Message Passage Interface

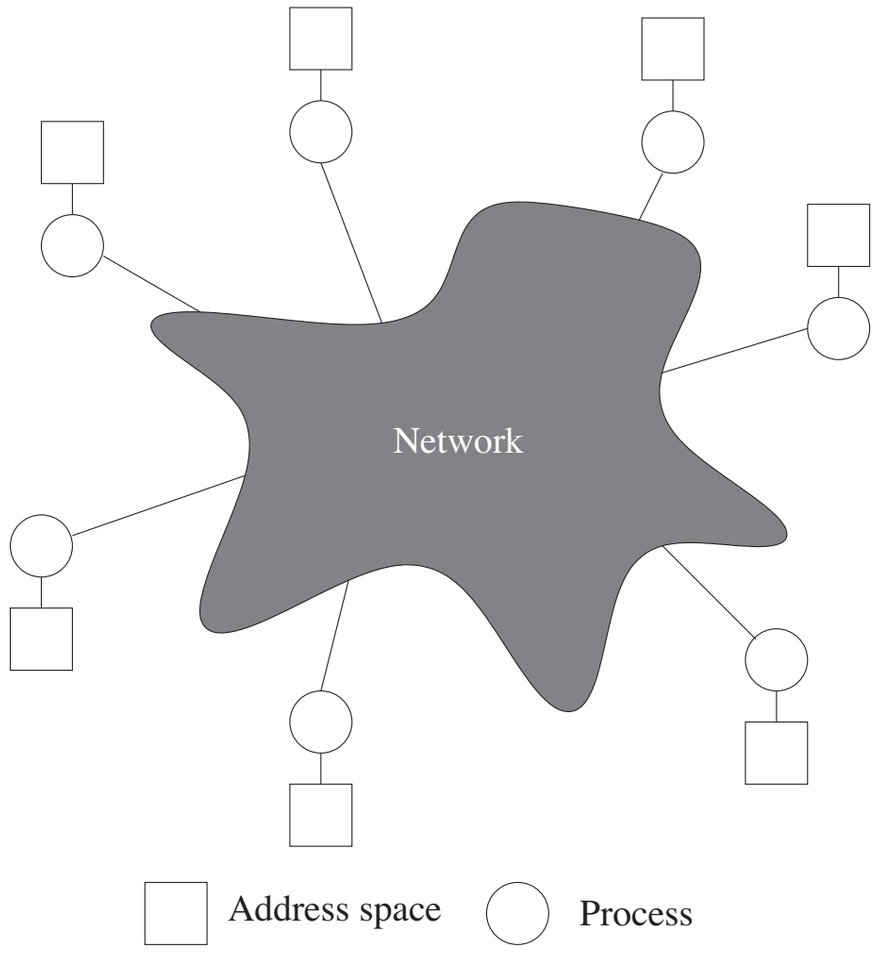
MPI

Modelo de paralelismo

Modelo de memoria compartida

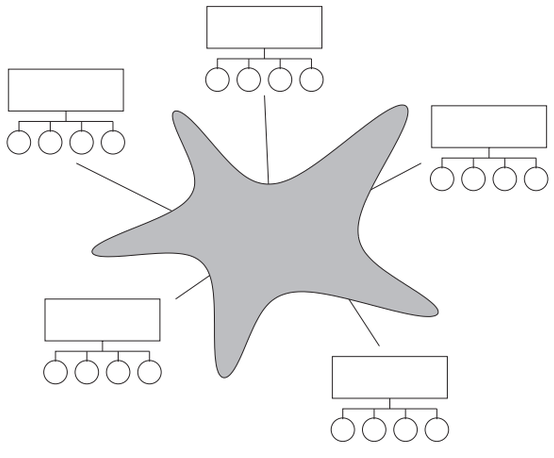


Modelo de paso de mensajes



Modelo Hibrido

PThread/ OpenMP / Fork



- El modelo de paso de mensajes postula un conjunto de procesos que solo tienen memoria local pero que pueden comunicarse con otros procesos enviando y recibiendo mensajes.
- Es una característica definitoria del modelo de paso de mensajes que la transferencia de datos desde la memoria local de un proceso a la memoria local de otro requiere que ambos procesos realicen operaciones.
- MPI es una implementación específica del modelo de paso de mensajes

MPI

Ventajas del modelo de paso de mensajes

- **Universalidad**. El modelo de paso de mensajes encaja bien en procesadores separados conectados por una red de comunicación (rápida o lenta). Por lo tanto, coincide con el nivel más alto del hardware de la mayoría de las supercomputadoras paralelas actuales.

- **Expresividad**. Se ha encontrado que el paso de mensajes es un modelo útil y completo para expresar algoritmos paralelos.

- **Facilidad de depuración.** La depuración de programas paralelos sigue siendo un área de investigación desafiante. La razón es que una de las causas más comunes de error es la sobrescritura inesperada de la memoria. El modelo de paso de mensajes, al controlar las referencias a la memoria de manera más explícita que cualquiera de los otros modelos, facilita la localización de lecturas y escrituras de memoria erróneas.

- **Rendimiento**. La razón más convincente por la que el paso de mensajes seguirá siendo una parte permanente del entorno informático paralelo es el rendimiento. A medida que las CPU modernas se han vuelto más rápidas, la gestión de sus cachés y la jerarquía de la memoria en general se ha convertido en la clave para aprovechar al máximo estas máquinas.

MPI

Que es?

- **MPI es una libreria, no un lenguaje.**
  - Especifica los nombres, las secuencias de llamada y los resultados de las funciones que se llamarán desde los programas C.
  - Los programas que los usuarios escriben en C se compilan con compiladores ordinarios y se vinculan con la libreria MPI.
- **MPI es una especificación, no una implementación particular.**
  - Un programa MPI correcto debería poder ejecutarse en todas las implementaciones de MP sin cambios.
- **MPI aborda el modelo de paso de mensajes.**
  - Enfoque en el movimiento de datos entre espacios de direcciones separados.

MPI

Conceptos basicos

- La comunicación se produce cuando una parte del espacio de direcciones de un proceso se copia en el espacio de direcciones de otro proceso.

- Esta operación es cooperativa y ocurre solo cuando el primer proceso ejecuta una operación de **envío (send)** y el segundo proceso ejecuta una operación de recepción (**receive**).

  - **MPI_Send**(address, count, datatype, destination, tag, comm)

  - **MPI_Recv**(address, maxcount, datatype, source, tag, comm, status)

    - **tag** es un número entero que se utiliza para la coincidencia de mensajes.

    - **comm** identifica un grupo de procesos y un contexto de comunicación.

    - **destination/source** es el ranking del destino/fuente en el grupo asociado con el comunicador **comm**.

MPI

comunicaciones colectivas

- Operaciones realizada por todos los procesos en un cómputo.
- Las operaciones colectivas son de dos tipos:
  - Las operaciones de **movimiento de datos** se utilizan para reorganizar los datos entre los procesos. La más simple de ellas es **broadcast**, pero se pueden definir muchas operaciones elaboradas de dispersión (**scattering**) y recopilación (**gathering**).
  - Operaciones de **cálculo colectivo (**mínimo, máximo, suma, OR lógico, etc., así como operaciones definidas por el usuario).

MPI

Funciones basicas

Función	Descripción
MPI_Init	Inicializar MPI
MPI_Comm_size	Determina cuántos procesos hay
MPI_Comm_rank	qué proceso soy
MPI_Send	Envio un mensaje
MPI_Recv	Recibo un mensaje
MPI_Finalize	Termina MPI

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

int main(int argc, char** argv) {
 //Iniciamos el embiente MPI
  MPI_Init(NULL, NULL);

  // obtenemos el numero de procesadores
  int world_size;
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);

  // obtenemos el ranking para cada proceso
  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

  // obtenemos el nombre del procesador
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  int name_len;
  MPI_Get_processor_name(processor_name, &name_len);
  if (world_rank !=3){
  printf("Hola mundo desde procesador %s, ranking %d de  %d procesadores\n",
         processor_name, world_rank, world_size);
  }else{
    printf("castigado\n");
  }
  // terminamos el ambien MPI
  MPI_Finalize();
}
```

```
! mpicc -o mpi_holamundo mpi_holamundo.c
! mpirun --allow-run-as-root -np 4 ./mpi_holamundo
```

```
Hola mundo desde procesador d4092f21c366, ranking 1 de  4 procesadores
Hola mundo desde procesador d4092f21c366, ranking 2 de  4 procesadores
castigado
Hola mundo desde procesador d4092f21c366, ranking 0 de  4 procesadores
```

MPI

Funciones colectivas basicas

- MPI broadcast (Envia un mensaje desde el proceso con ranking “root” a todos los demás procesos del comunicador.)

```
int MPI Bcast(void *buf, //variable a comunicar
              int count, // numero de elementos
              MPI Datatype datatype, // tipo de dato
              int root, //quien lo envia
              MPI Comm comm) // mundo comunicación
```

- MPI reduce (Reduce valores en todos los procesos a un único valor.)

```
int MPI Reduce(const void *sendbuf, //variable de local
               void *recvbuf, // resultado
               int count, // numero de elementos
               MPI Datatype datatype,  // tipo de dato
               MPI Op op, // operacion
               int root, // proceso que colecta el resultado
               MPI Comm comm) // mundo de comunicación
```

MPI

Funciones colectivas basicas

- MPI Gather (Colecta valores de un grupo de processos)

```
int MPI_Gather(const void *sendbuf, //direcion de lo que quiero enviar
               int sendcount, //cuantos items
               MPI Datatype sendtype, //tipo de dato
               void *recvbuf, //direccion de donde lo almaceno
               int recvcount,//cuanto voy a almacenar
               MPI Datatype recvtype, // que tipo
               int root,//que proceso lo recibe
               MPI Comm comm) // via de comunicación
```

- MPI Scatter (Envía datos desde un proceso a todos los demás procesos en un comunicador)

```
int MPI_Scatter(const void *sendbuf, //direcion de lo que quiero enviar
                int sendcount, //cuantos items
                MPI_Datatype sendtype, //tipo de dato
                void *recvbuf,  //direccion de donde lo almaceno
                int recvcount, //cuanto voy a almacenar
                MPI_Datatype recvtype, / que tipo
                int root, //que proceso lo envia
                MPI_Comm comm) // via de comunicación
```

MPI

Funciones colectivas

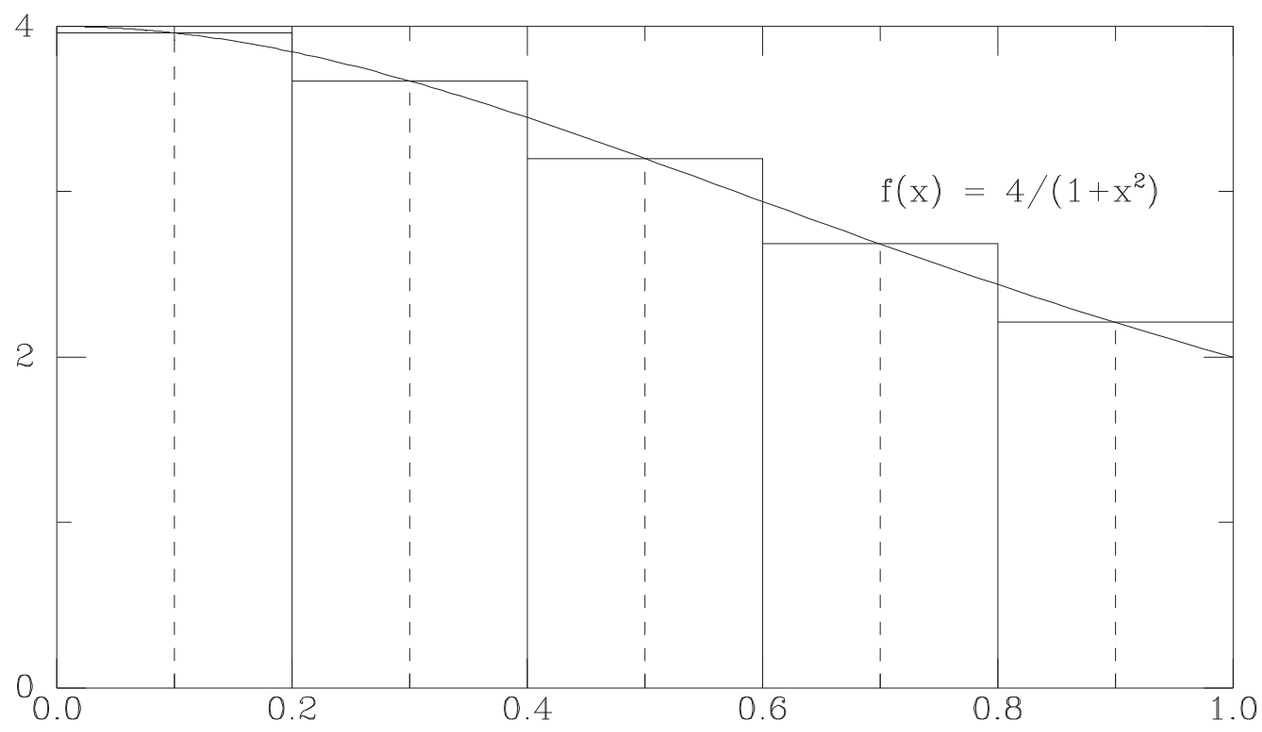
```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>


int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int N = 1000; // Tamaño del arreglo
    int local_N = N / size; // Tamaño del subarreglo para cada 
proceso
    int local_sum = 0;
    int global_sum = 0;

    int* data = NULL;
    int* local_data = (int*)malloc(local_N * sizeof(int));

    // Inicializar el arreglo de datos en el proceso 0
    if (rank == 0) {
        data = (int*)malloc(N * sizeof(int));
        for (int i = 0; i < N; i++) {
            data[i] = i;
        }
    }

    // Distribuir los datos entre los procesos
    MPI_Scatter(data, local_N, MPI_INT, local_data, local_N, 
MPI_INT, 0, MPI_COMM_WORLD);

    // Calcular la suma local
    for (int i = 0; i < local_N; i++) {
        local_sum += local_data[i];
    }
    printf("La suma local es: %d proceso=%d\n", local_sum,rank);

    // Sumar las sumas locales para obtener la suma global
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, 
MPI_COMM_WORLD);

    // El proceso 0 imprime el resultado
    if (rank == 0) {
        printf("La suma global es: %d\n", global_sum);
    }

    MPI_Finalize();

    return 0;
}
```

MPI

Funciones colectivas basicas (ejemplo)

Calculando el valor de pi

$$\int_0^1 \frac{1}{1+x^2} dx = \arctan(x)|_0^1 = \arctan(1) - \arctan(0) = \arctan(1) = \frac{\pi}{4},$$



MPI

Aproximando el valor de pi

```c
#include <stdio.h>
#include "mpi.h"
#include <math.h>

int main(int argc, char *argv[])
{
  int n=5000, myid, numprocs, i;
  double PI25DT = 3.14159265358979323846264643;
  double mypi, pi=0, h, sum, x;
  MPI_Init(NULL,NULL);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  //comunicamos el numero de intervalos a cada proceso del mundo
  //calculamos los intervalos y repetimos si es necesario
     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
      h   = 1.0 / (double) n;
      sum = 0.0;
      for (i = myid + 1; i <= n; i += numprocs) {
            x = h * ((double)i - 0.5);
            sum += (4.0 / (1.0 + x*x));
      }
      mypi = h * sum;
      printf("my id : %d  mypi : %.16f\n", myid,mypi);

      //colectamos los calculos de cada trabajador usando MPI_Reduce

      MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                 MPI_COMM_WORLD);
      //imprimimos el valor luego de sumar
      if (myid == 0)
          printf("pi es aproximadamente %.16f, El error es %.16f\n",
                 pi, fabs(pi - PI25DT));

  MPI_Finalize();
  return 0;
}
```

MPI

Multiplicación de matrices

```
int main(void)
{
  int      size, row, column;

  size = ARRAY_SIZE;

//puntero a la matriz resultante
int *final_matrix;
int num_worker, rank;
MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &num_worker);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if(rank == 0){
  // inicializamos los valores de la MA
  inicializamos_matriz(size, MA);
  //inicializamos los valores de la MB
  inicializamos_matriz(size, MB);
  //imprimimos
  printf("La matriz A es;\n");
  imprimir_matriz(size,MA);
  printf("La matriz B es;\n");
  imprimir_matriz(size,MB);
  //reservamos la memoria para la matriz final
    final_matrix = (int *) malloc(sizeof(int*) * size*size);
}

MPI_Bcast(MA, size*size , MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(MB, size*size , MPI_INT, 0, MPI_COMM_WORLD);

//chequeamos si proceso 1 recibio la información
if(rank == 1){
  printf("id:%d La matriz A es;\n",rank);
  imprimir_matriz(size,MA);
  printf("id:%d La matriz B es;\n",rank);
  imprimir_matriz(size,MB);
}
```

```
// determinamos la fila de inicio y fin para la proceso
trabajador
int startrow = rank * ( size / num_worker);
int endrow = ((rank + 1) * ( size / num_worker)) -1;
//calculamos las sub-matrices
int number_of_rows = size / num_worker;
int *result_matrix = (int *) malloc(sizeof(int*) *
number_of_rows * size);
    //multiplicamos
    int rowl=0;
    for(row = startrow; row <= endrow; row++) {
     for (column = 0; column < size; column++) {
      mult(size, row, column,rowl, MA, MB, result_matrix);
        rowl++;
        }
     }
    // log information
    if(rank==1){
       printf("id:%d  startrow=%d, endrow=%d,work=%d;
\n",rank,startrow,endrow,number_of_rows*size);
       imprimir_matriz2(number_of_rows,size,result_matrix);
    }

//recolectamos los resutlados de la matriz
    MPI_Gather(result_matrix, number_of_rows*size, MPI_INT,
           final_matrix, number_of_rows*size,  MPI_INT, 0,
MPI_COMM_WORLD);

  //imprimimos la matriz luego de recolectar los resultados
  if(rank == 0){
  printf("La matriz resultante C es (MPI);\n");
  imprimir_matriz2(size,size,final_matrix);
  }

 MPI_Finalize();

  return 0;
}
```

MPI

Tipos de datos

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t
MPI_SHORT	short
MPI_INT	int
MPI_LONG	long
MPI_LONG_LONG_INT	long long
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_LONG_LONG	unsigned long long
MPI_C_COMPLEX	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_PACKED	
MPI_BYTE	

MPI

Tipos de datos

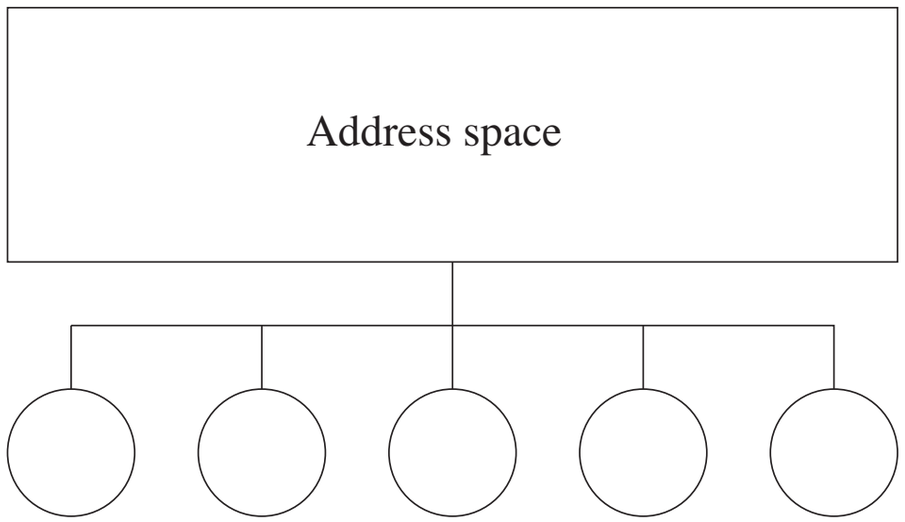
```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <time.h>

int main(int argc, char** argv) {
    int ProcRank, ProcNum, mTag = 0;
    struct { int x;
        int y;
        int z;
        } point;
    MPI_Datatype ptype;
    MPI_Status status;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
    MPI_Type_contiguous(3, MPI_INT, &ptype);
    MPI_Type_commit(&ptype);
    if (ProcRank == 1) {
        point.x = 45; point.y = 36; point.z = 0;
        MPI_Send(&point, 1, ptype, 0, mTag, MPI_COMM_WORLD);
    }
    if (ProcRank == 0) {
        MPI_Recv(&point, 1, ptype, 1, mTag, MPI_COMM_WORLD, &status);
        printf("Proceso: %d recibio punto con  coordenadas (%d; %d; %d)\n", ProcRank, point.x, point.y, point.z);
    }
    MPI_Finalize();
}
```

Modelos paralelos hibridos

Modelo Hibrido

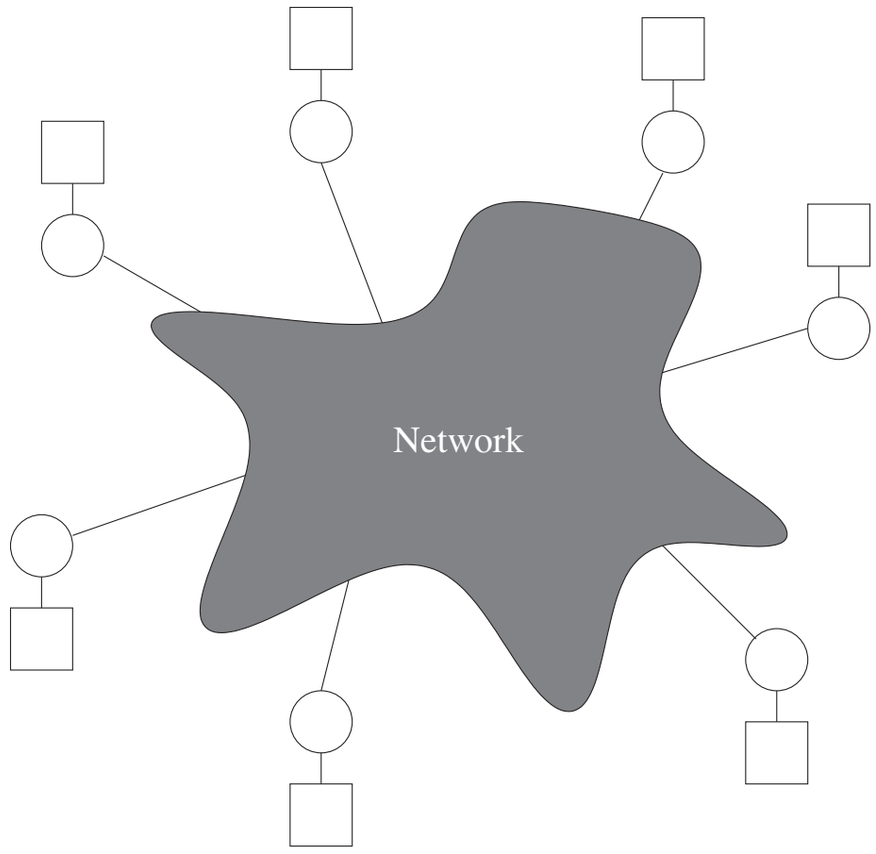
Modelo de paralelismo

Modelo de memoria compartida

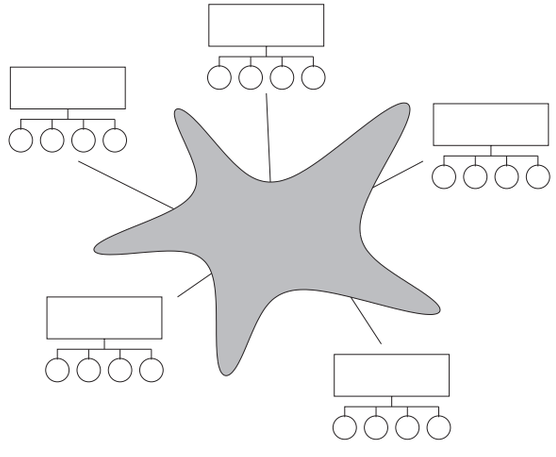


PThread/ OpenMP / Fork

Modelo de paso de mensajes



Modelo Hibrido



Procesos y hilos

MPI y OpenMP/Pthreads

- MPI = Process, OpenMP = Thread
- El programa comienza con un solo proceso.
- Los procesos tienen su propio espacio de memoria (privado)
- Un proceso puede crear uno o más hilos
- Los hilos creados por un proceso comparten su espacio de memoria
  - Leer y escribir en las mismas direcciones de memoria
  - Comparten los mismos identificadores de proceso y descriptores de archivo
- Cada hilo tiene un contador de instrucciones único y un puntero de pila
- Un hilo puede tener almacenamiento privado en la pila

Modelo hibrido

MPI & OpenMP/Phtreads

- MPI = Process, OpenMP = Thread
- Paralelización de dos niveles
- Ideal para el hardware de un clúster.
- MPI entre nodos
- OpenMP dentro de los nodos de memoria compartida

Modelo Hibrido

Ventajas

- **Escalabilidad**: MPI permite escalar una aplicación en múltiples servidores. OpenMP y Pthreads se utilizan para paralelizar el cómputo local en un servidor (nodo). Escalabilidad tanto a nivel de nodos como a nivel de núcleos de CPU.

- **Mejor Rendimiento/Flexibilidad**: MPI se centra en la comunicación entre procesos, mientras que OpenMP y Pthreads se centran en el paralelismo a nivel de hilos.
    - Control fino sobre cómo se divide y gestiona el trabajo.

- **Mejor uso de Recursos**: Al usar hilos en el nivel local de cada nodo, se pueden aprovechar al máximo los recursos de la CPU, especialmente en sistemas multiprocesador o con múltiples núcleos.

- **Reducción de la Latencia**: Al paralelizar tareas a nivel de nodo mediante hilos, se puede reducir la latencia en las comunicaciones entre los procesos MPI.

- **Facilita la Programación Concurrente**: MPI y OpenMP/Pthreads se centran en aspectos diferentes de la programación paralela, lo que facilita la programación concurrente y puede reducir la complejidad de la implementación.

Modelo hibrido

MPI & OpenMP

- En la programación híbrida, cada proceso puede tener varios hilos ejecutando simultáneamente
    - Todos los hilos dentro de un proceso comparten todos los objetos MPI (Comunicadores, mensajes, etc).
- MPI define 4 niveles de seguridad de hilos
    1. MPI_THREAD_SINGLE (solo 1 hilo)
    2. MPI_THREAD_FUNNELED (>1, pero solo el hilo maestro puede utilizar funciones MPI)
    3. MPI_THREAD_SERIALIZED (> 1, pero solo un hilo puede realizar llamadas MPI a la vez)
    4. MPI_THREAD_MULTIPLE (> 1, cualquier hilo puede realizar llamadas MPI en cualquier momento)
- MPI_Init_thread (no MPI_Init) si más de un hilo es necesario.
    - MPI_Init_thread(int required, int *provided)

Ejemplos

MPI/OpenMP

```
#include <stdio.h>
#include <stdlib.h>
//we load omp/mpi
#include <omp.h>
#include <mpi.h>
// defines the MPI_THREADS_MODE
#define MPI_THREAD_STRING(level)  \
        ( level==MPI_THREAD_SERIALIZED ? "THREAD_SERIALIZED" : \
                ( level==MPI_THREAD_MULTIPLE ? "THREAD_MULTIPLE" : \
                        ( level==MPI_THREAD_FUNNELED ? "THREAD_FUNNELED" :
\
                                ( level==MPI_THREAD_SINGLE ?
"THREAD_SINGLE" : "THIS_IS_IMPOSSIBLE" ) ) ) )

int main(int argc, char ** argv)
{
    /* Estos son los soportes de hilos deseados y disponibles.
        Se puede utilizar un código híbrido en el que todas las llamadas
MPI se realizan desde el hilo principal (FUNNELED).
        Si los hilos realizan llamadas MPI, MULTIPLE es el apropiado. */
    int requested=MPI_THREAD_FUNNELED, provided;

    /* Intentamos activar los hilos MPI usando el modo requerido:
MPI_THREAD_FUNNELED*/
    MPI_Init_thread(&argc, &argv, requested, &provided);
    if (provided<requested)
    {
        printf("MPI_Init_thread provee %s cuando %s fue solicitado.
Terminando el programa. \n",
                MPI_THREAD_STRING(provided), MPI_THREAD_STRING(requested) );
        exit(1);
    }

    int world_size, world_rank;

    MPI_Comm_size(MPI_COMM_WORLD,&world_size);
    MPI_Comm_rank(MPI_COMM_WORLD,&world_rank);

    printf("Hola desde %d de total :%d  procesos\n", world_rank,
world_size);

    //ocupamos openMP para crear una seccion paralela
    #pragma omp parallel
    {
        int omp_id  =omp_get_thread_num();
        int omp_num =omp_get_num_threads();
        printf("MPI rank # %2d OpenMP thread # %2d of %2d \n", world_rank,
omp_id, omp_num);
        fflush(stdout);
    }

    MPI_Finalize();
    return 0;
}
```

```
! mpicc -o mpi_openmp_e1 mpi_openmp_e1.c  -fopenmp
```

```
%env OMP_NUM_THREADS=3
! mpirun --oversubscribe --allow-run-as-root -np 4
./mpi_openmp_e1
```

```
env: OMP_NUM_THREADS=3
Hola desde 0 de total :4  procesos
Hola desde 1 de total :4  procesos
Hola desde 3 de total :4  procesos
Hola desde 2 de total :4  procesos
MPI rank #  1 OpenMP thread #  1 of  3
MPI rank #  1 OpenMP thread #  2 of  3
MPI rank #  2 OpenMP thread #  1 of  3
MPI rank #  1 OpenMP thread #  0 of  3
MPI rank #  2 OpenMP thread #  2 of  3
MPI rank #  2 OpenMP thread #  0 of  3
MPI rank #  0 OpenMP thread #  0 of  3
MPI rank #  0 OpenMP thread #  1 of  3
MPI rank #  0 OpenMP thread #  2 of  3
MPI rank #  3 OpenMP thread #  0 of  3
MPI rank #  3 OpenMP thread #  1 of  3
MPI rank #  3 OpenMP thread #  2 of  3
```

Ejemplos

MPI/OpenMP

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <omp.h>

#define VECTOR_SIZE 1000

int main(int argc, char* argv[]) {
    int rank, size;
    int local_size, local_start, local_end;
    double* vectorA, * vectorB;
    double local_sum = 0.0, global_sum = 0.0;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Calcular el tamaño local de los vectores
    local_size = VECTOR_SIZE / size;
    local_start = rank * local_size;
    local_end = local_start + local_size;

    // Asignar memoria para los vectores locales
    vectorA = (double*)malloc(local_size * sizeof(double));
    vectorB = (double*)malloc(local_size * sizeof(double));

    // Inicializar vectores locales
    #pragma omp parallel for
    for (int i = local_start; i < local_end; i++) {
        vectorA[i - local_start] = 1.0;  // Inicializar vectorA con 1.0
        vectorB[i - local_start] = 2.0;  // Inicializar vectorB con 2.0
    }

    // Calcular el producto escalar local
    #pragma omp parallel for reduction(+:local_sum)
    for (int i = 0; i < local_size; i++) {
        local_sum += vectorA[i] * vectorB[i];
    }

    printf("El producto escalar local es: %lf %d\n", local_sum, rank);
    // Reducir los resultados locales en un resultado global
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);

    // El proceso 0 imprime el resultado
    if (rank == 0) {
        printf("El producto escalar global es: %lf\n", global_sum);
    }

    // Liberar memoria y finalizar MPI
    free(vectorA);
    free(vectorB);
    MPI_Finalize();

    return 0;
}
```

```
! mpicc -o ppunto ppunto.c  -fopenmp
```

```
%env OMP_NUM_THREADS=4
! mpirun --oversubscribe
--allow-run-as-root -np 4
./ppunto
```

```
env: OMP_NUM_THREADS=4
El producto escalar local es: 500.000000
El producto escalar local es: 500.000000
El producto escalar local es: 500.000000
El producto escalar local es: 500.000000
El producto escalar global es: 2000.00000
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <pthread.h>

#define ARRAY_SIZE 1000000

// Estructura que contiene los datos que se pasan a cada hilo
struct ThreadData {
    int* array;
    int local_size;
    int local_sum;
};

// Función que se ejecuta en cada hilo para calcular la suma local
void* calculateSum(void* arg) {
    struct ThreadData* data = (struct ThreadData*)arg;
    for (int i = 0; i < data->local_size; i++) {
        data->local_sum += data->array[i];
    }
    return NULL;
}
```

```
! mpicc -o ppunto_pt ppunto_pt.c -lpthread
```

```
! mpirun --oversubscribe --allow-run-as-root -np 4 ./ppunto_pt
```

```c
int main(int argc, char* argv[]) {
    int rank, size;
    int* data;
    int local_size, local_start, local_end;
    int local_sum = 0;
    pthread_t* threads;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Calcular el tamaño local del array
    local_size = ARRAY_SIZE / size;
    local_start = rank * local_size;
    local_end = local_start + local_size;

    // Asignar memoria para el array local
    data = (int*)malloc(local_size * sizeof(int));
    threads = (pthread_t*)malloc(size * sizeof(pthread_t));

    // Inicializar el array local con valores aleatorios
    for (int i = 0; i < local_size; i++) {
        data[i] = rand() % 10; // Valores aleatorios entre 0 y 9
    }

    // Crear hilos para calcular la suma local en paralelo
    struct ThreadData threadData;
    threadData.array = data;
    threadData.local_size = local_size;
    threadData.local_sum = 0;

    pthread_create(&threads[rank], NULL, calculateSum, &threadData);

    // Esperar a que todos los hilos terminen
    pthread_join(threads[rank], NULL);

    // Realizar una operación de reducción para obtener la suma global
    MPI_Reduce(&threadData.local_sum, &local_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // El proceso 0 imprime la suma global
    if (rank == 0) {
        printf("Suma global: %d\n", local_sum);
    }

    // Liberar memoria y finalizar MPI
    free(data);
    free(threads);
    MPI_Finalize();

    return 0;
}
```

MPI/OpenMP time

- OpenMP time:

```
double t1, t2;
t1=omp_get_wtime();
//do something expensive...
t2=omp_get_wtime();
printf("Total Runtime = %g\n", t2-t1);
```

- MPI time:

```
double t1 = MPI_Wtime();
//do something expensive...
double t2 = MPI_Wtime();
if(my_rank == final_rank) {
printf("Total runtime = %g s\n", (t2-t1));
}
```

Consultas?

Consultas o comentarios?

Muchas gracias