

Introducción a la programación paralela. Y al cómputo de alto rendimiento.

Clemente González Julio César ¹

¹CADAC - Instituto de Astronomía
UNAM

Mexican Numerical Simulations School - IF - UNAM



Contenido

- 1 **Introducción al cómputo paralelo.**
 - Algunos conceptos importantes.
 - Entendiendo el rendimiento.
 - Paralelismo.
- 2 **Programando con OpenMP.**
 - Introducción
 - Usando OpenMP
- 3 **Programando con MPI**
 - Introducción
 - MPI. Características.
 - Usando MPI



Contenido

- 1 **Introducción al cómputo paralelo.**
 - Algunos conceptos importantes.
 - Entendiendo el rendimiento.
 - Paralelismo.
- 2 Programando con OpenMP.
 - Introducción
 - Usando OpenMP
- 3 Programando con MPI
 - Introducción
 - MPI. Características.
 - Usando MPI



Conceptos básicos.

Hardware

- Unidades de procesamiento: Dispositivos capaces de realizar operaciones.
- Procesadores: Dispositivos capaces de ejecutar un programa.
- Núcleos: Procesadores empaquetados en un mismo circuito electrónico.



Conceptos básicos

Software

- Proceso: Entidad lógica de un s. o. que representa la ejecución de un programa.
- Thread (hebra): Divisiones de un proceso que puede ejecutarse de forma simultánea e independiente.



Conceptos b'asicos.

Otros

- Rendimiento: número de operaciones aritméticas por segundo.
- n : Parámetro de un algoritmo del que depende el número de operaciones a realizar:
 - Tamaño de una matriz.
 - Puntos en una malla.
 - Número de partículas.
- Carga computacional: Operaciones de un algoritmo asociadas a n .
- Operaciones atómicas: Aquellas que se realizan “instantáneamente”.



Contenido

- 1 **Introducción al cómputo paralelo.**
 - Algunos conceptos importantes.
 - Entendiendo el rendimiento.
 - Paralelismo.
- 2 Programando con OpenMP.
 - Introducción
 - Usando OpenMP
- 3 Programando con MPI
 - Introducción
 - MPI. Características.
 - Usando MPI

Rendimiento de una computadora.

- Existe el rendimiento teórico o *peak performance*.
- Y el rendimiento máximo o *max performance*
 - Depende de la aplicación.
 - El proyecto TOP500 utiliza *Linpack*



La evolución del rendimiento.

La evolución del rendimiento está regida por:

- La “Ley de Moore”.
- Mejoras en las arquitecturas.



Para qué mayor rendimiento.



Cómo obtener mayor rendimiento.

- Esperar a que las tecnologías de cómputo cambien.
- Intentar usar de una mejor manera las tecnologías actuales.

Tecnologías paralelas disponibles.

- Procesadores multi-núcleos.
- Unidades de co-procesamiento como GPUs y Xeon-Phi.
- Computadoras paralelas.



Aceleración en programas paralelos.

La Aceleración o **SpeedUp** es una medida de la mejora en el tiempo de ejecución de un programa.

$$S_p = \frac{T_s}{T_p}$$

Donde:

- S_p Es la aceleración.
- T_s Es el tiempo de ejecución en secuencial.
- T_p Es el tiempo de ejecución en modo paralelo usando p procesadores.



Aceleración en programas paralelos.

El SpeedUp ideal se comporta:

$$T_p = \frac{T_s}{p}$$



SpeedUp y otros factores.

El SpeedUp ideal no es posible alcanzarse (excepto bajo ciertas condiciones).

Ley de Amdahl

Todo programa tiene una **fracción secuencial**, que es independiente del número de procesadores sobre los que corre.

$$T_p = f_s * T_s + \frac{(1-f_s)*T_s}{p}$$

$$\lim_{p \rightarrow \infty} S_p = \frac{1}{f_s}$$



Minimizando el efecto de la fracción secuencial.

- La fracción secuencial tiende a disminuir al aumentar n .
- Es más fácil crear programas paralelos para aumentar n .
- Que modificar el algoritmo para reducir la fracción secuencial.



Recapitulando.

- El rendimiento ha aumentado en forma “constante”.
- Se está complicando mucho aumentar la capacidad por procesador.
- Emplear el paralelismo sirve para incrementar el rendimiento.
- Es importante que el algoritmo tenga una fracción secuencial muy pequeña.
- Es mejor paralelizar al algoritmo que algunas de sus partes.



Recapitulando.

Es necesario definir metas.

- Para qué rango de n se desea ejecutar en el programa paralelo.
- Cuáles son los tiempos aceptables de ejecución.
- Cuántos procesadores serán necesarios.



Contenido

- 1 **Introducción al cómputo paralelo.**
 - Algunos conceptos importantes.
 - Entendiendo el rendimiento.
 - **Paralelismo.**
- 2 Programando con OpenMP.
 - Introducción
 - Usando OpenMP
- 3 Programando con MPI
 - Introducción
 - MPI. Características.
 - Usando MPI



Programas paralelos.

Un programa paralelo:

- Cuando se ejecuta, crea varias instancias que trabajan de forma coordinada.
- Cada instancia se reparte una carga computacional generada por n .
- Las instancias se coordinan intercambiando información.



Modelos de programación paralela.

- Memoria compartida (OpenMP).
- Intercambio de mensajes (MPI).
- Espacio global particionado (UPC).
- *Single Program Multiple Data* - SPMD (CUDA).



Arquitecturas paralelas.

Una computadora paralela debe tener:

- Varias unidades de procesamiento.
- Hardware de control y comunicación.
- Software de control y comunicación.



Arquitecturas paralelas.

Clasificación de Flynn

- SISD *Single Instruction Single Data.*
- SIMD *Single Instruction Multiple Data.*
- MISD *Multiple Instruction Single Data.*
- MIMD *Multiple Instruction Multiple Data.*



Diseño de programas paralelos.

Características deseadas.

- Deben ser correctos.
- Deben ser eficientes.
- Deben ser escalables.
- Deben ser portátiles.
- Deben ser flexibles.



Contenido

- 1 Introducción al cómputo paralelo.
 - Algunos conceptos importantes.
 - Entendiendo el rendimiento.
 - Paralelismo.
- 2 Programando con OpenMP.
 - Introducción
 - Usando OpenMP
- 3 Programando con MPI
 - Introducción
 - MPI. Características.
 - Usando MPI



¿Qué es OpenMP?

OpenMP es un API para escribir aplicaciones multi-thread.

- Un conjunto de directivas de compilación y funciones de biblioteca.
- Una forma simplificada de escribir programas multi-thread.
- Estándar conformado por más de 20 años de programas SMP.



¿Cómo se usa OpenMP?

- Muchos de los constructores de OpenMP son directivas del compilador.
 - `#pragma omp constructor [clausula[clausula]...]`
 - Ejemplo:
`#pragma omp parallel num_threads(4)`
- Para C, las declaraciones de las funciones y los tipos de datos están en:
 - `#include <omp.h>`
- Muchos de los constructores de OpenMP aplican un “bloque estructurado”.
 - Puede existir una sentencia `exit()` dentro de un bloque estructurado



Hola Mundo en OpenMP (I).

Verificando el ambiente de trabajo.

- Escribir un programa que muestre el típico “Hola Mundo!”.

```
void main(void){  
int mi_id = 0;  
printf(“Hola(%d) ”, mi_id);  
printf(“ Mundo”);  
}
```



Hola Mundo en OpenMP (II).

Verificando el ambiente de trabajo.

- Escribir un programa que muestre el típico “Hola Mundo!”.

```
#include "omp.h"
void main(void){
#pragma omp parallel
{
int mi_id = 0;
printf("Hola(%d) ", mi_id);
printf(" Mundo");
}
}
```



Hola Mundo en OpenMP (III).

Verificando el ambiente de trabajo.

- Escribir un programa que muestre el típico “Hola Mundo!”.

```
#include "omp.h"
void main(void){
#pragma omp parallel
{
int mi_id = omp_get_thread_num();
printf("Hola(%d) ", mi_id);
printf(" Mundo");
}
}
```



Hola Mundo en OpenMP (IV).

Suponiendo 4 hilos / hebras / (*threads*), una salida posible es:

```
Hola(3) Mundo  
Hola(0) Mundo  
Hola(2) Mundo  
Hola(1) Mundo
```



Revisión de OpenMP

- *OpenMP* es *multi-threading*, y con un modelo de memoria compartida.
 - Los *threads* se comunican a través de variables compartidas.
- Un mal manejo de datos compartidos puede causar condiciones de carrera.
 - Cuando dos o más *threads* “buscan” usar el mismo recurso a la vez.
- Y para controlarlos
 - Se usa la sincronización para proteger conflictos de datos.
- Sin embargo la sincronización es costosa.
 - Cambiar el orden en el acceso a los datos siempre que sea posible.



Contenido

- 1 Introducción al cómputo paralelo.
 - Algunos conceptos importantes.
 - Entendiendo el rendimiento.
 - Paralelismo.
- 2 Programando con OpenMP.
 - Introducción
 - Usando OpenMP
- 3 Programando con MPI
 - Introducción
 - MPI. Características.
 - Usando MPI



Modelo de programación de *OpenMP*

- Usa el paralelismo *Fork - Join*:
- Un *thread* principal se divide en un conjunto de *threads*.
- En el código fuente, el paralelismo se agrega de forma incremental.
 - Un programa secuencial va evolucionando hasta que se convierte en un programa paralelo.



Creación de threads.

Regiones paralelas (I).

Se crean threads en OpenMP con el constructor **parallel**.

- Ejemplo: para crear una región paralela con 4 threads y con llamada a función.

```
omp_set_num_threads(4);  
#pragma omp parallel  
{  
int mi_id = omp_get_thread_num();  
una_funcion(mi_id, A);  
}
```



Creación de threads.

Regiones paralelas (II).

- Ejemplo: para crear una región paralela con 4 threads y empleando cláusulas.

```
#pragma omp parallel num_threads(4)
{
  int mi_id = omp_get_thread_num();
  una_funcion(mi_id, A);
}
```



Otro ejemplo. Integración numérica.

Sabemos que:

$$\int_0^1 \frac{4.0}{1+x^2} dx = \pi$$

Y que se puede aproximar mediante la suma de rectángulos:

$$\sum_{i=0}^N F(x_i) \approx \pi$$

Donde cada rectángulo tiene un ancho Δx y una altura $F(x_i)$



Integración numérica.

Programa secuencial.

```
static long num_p = 10000;
double paso;
void main(void) {
    int i; double x, pi, sum = 0.0;
    paso = 1.0/(double) num_p;
    for(i=0; i < num_p; i++){
        x=(i+0.5)*paso;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = paso * sum;
}
```



Consideraciones para la versión paralela.

- Es necesario poner atención en cuales serán variables compartidas y cuales privadas.
- Es necesario el uso de las funciones:
 - `int omp_get_num_threads();` // Cuantos threads
 - `int omp_get_num_threads();` // Cuantos threads
- Función que puede resultar útil.
 - `double omp_get_wtime();` // Saber el tiempo.



Integración numérica.

Programa OpenMP versión 1 (I)

Declaraciones y definiciones.

```
#define NTHRDS 4
#include <omp.h>
static long num_p = 10000; // Numero total de segmentos
double paso; // Delta x
int main(void){
    int num_t;
    double pi, suma[4], sumaf;
    paso = 1.0 / (double)num_p;
```



Integración numérica.

Programa OpenMP versión 1 (II)

Parte paralela.

```
#pragma omp parallel num_threads(NTHRDS)
{
    int i;
    int mi_id = omp_get_thread_num();
    double x;
    num_t = omp_get_num_threads();
    suma[mi_id]=0.0;
    for(i = 0 + mi_id; i < num_p; i=i+num_t){
        x = (i + 0.5) * paso;
        suma[mi_id] = suma[mi_id] + 4.0 / (1.0 + x * x);
    }
}
```



Integración numérica.

Programa OpenMP versión 1 (II)

Integrando los resultados parciales.

```
// Se estima pi de forma secuencial  
int i;  
for (i=0; i<NTHRDS; i++)  
    sumaf=sumaf+suma[i];  
pi = paso * sumaf;  
printf("El_valor_aprox._de_pi_es:_%3.10lf_\n", pi);  
return EXIT_SUCCESS;  
}
```



Sincronización.

Se usa para imponer restricciones de acceso y proteger información en memoria compartida.

- **critical**
- **atomic**
- **barrier**
- **ordered**



Sincronización.

Cláusula *critical*.

Exclusión mutua: Solo un thread a la vez puede entrar en una región **critical**.

```
float res;  
#pragma omp parallel  
{ float B; int i, mi_id, nthrds;  
  mi_id = omp_get_thread_num();  
  nthrds = omp_get_num_threads();  
  for(i=mi_id; i<ITRS; i+nthrds){  
    B = funcion(i);  
    #pragma omp critical  
    otra_funcion(B, res);  
  }  
}
```



Sincronización.

Cláusula *atomic*.

La cláusula **atomic** provee exclusión mutua, pero solo aplica a la actualización de una locación de memoria.

```
#pragma omp parallel
{
    double tmp, B;
    B = funcion();
    tmp = funcion_grande(B);
    #pragma omp atomic
        X = X + tmp;
}
```



Integración numérica.

Programa OpenMP versión 2.

Consideraciones.

- Del ejercicio anterior, se usa un arreglo para guardar la suma parcial de cada thread.
- Si sucede que los elementos del arreglo comparten una línea de cache, lleva a una falsa compartición.
 - Se invalida el contenido del arreglo por copias de caché.
- Modificar el programa para que eso no pase.
- Es necesario usar la cláusula **critical**.



SPMD vs Worksharing

- Usar únicamente el constructor parallel crea un programa SPMD (single program multiple data).
 - Cada thread ejecuta el mismo código sobre diferentes datos.
- ¿Cómo separar que cada hilo de un mismo equipo realice tareas diferentes?
- A esto se le llama worksharing.
- Existe el constructor de ciclos.
- Existen otros constructores.



El constructor de ciclos para *worksharing*.

El constructor de ciclos para *worksharing* divide las iteraciones de un ciclo entre todos los threads en un equipo.

```
#pragma omp parallel
{
  #pragma omp for
  for(i = 0; i < N; i++)
    funcion_con_datos(i);
}
```



El constructor de ciclos para *worksharing*.

Ejemplo

El código en secuencial.

```
for(i = 0; i < N; i++){  
    a[i] = a[i] + b[i];  
}
```



El constructor de ciclos para *worksharing*. Ejemplo

El código en OpenMP programado como **SPMD**.

```
#pragma omp parallel
{
    int mi_id, nthrds, i_ini, i_fin;
    mi_id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    i_ini = mi_id * N / nthrds;
    i_fin = (mi_id + 1) * N / nthrds;
    if(mi_id == nthrds - 1) i_fin = N;
    for(i=0; i<N; i++)
        a[i] = a[i] + b[i];
}
```

El constructor de ciclos para *worksharing*. Ejemplo

- El código en OpenMP con una región **parallel** y un constructor de worksharing.
 - En este caso la variable *i* es privada.

```
#pragma omp parallel
#pragma omp for
  for(i = 0; i < N; i++){
    a[i] = a[i] + b[i];
  }
```



Combinando los constructores *parallel* y *worksharing*.

Hay una forma breve de ambos, en donde se coloca el constructor *parallel* y de *worksharing* en la misma línea.

- Separados.

```
double res[MAX]; int i;  
#pragma omp parallel  
{  
    #pragma omp for  
    for (i=0; i<MAX; i++)  
        res[i] = funcion();  
}
```



Combinando los constructores *parallel* y *worksharing*.

Hay una forma breve de ambos, en donde se coloca el constructor *parallel* y de *worksharing* en la misma línea.

- Juntos.

```
double res[MAX]; int i;  
#pragma omp parallel for  
    for (i=0; i<MAX; i++)  
        res[i] = funcion();  
}
```



¿Cómo trabajar con ciclos?

Acercamiento básico.

- Encontrar los ciclos computacionalmente intensivos.
- Hacer que las iteraciones del ciclo sean independientes.
 - Es decir, el ciclo se puede ejecutar en cualquier orden.
- Colocar las directivas OpenMP apropiadas y probar.



Reducciones.

Dado el siguiente código

```
double prom=0.0, A[MAX];  
int i;  
for (i=0;i< MAX; i++) {  
    prom = prom + A[i];  
}  
prom = prom/MAX
```



Reducciones.

- Se combinan datos en una sola variable de acumulación.
- Dependencia entre iteraciones difícil de remover.
- Es una situación común.
- El soporte a operaciones de reducción se incluyen en muchos ambientes de programación paralela.



Reducciones en OpenMP.

- En OpenMP existe la cláusula **reduction**.
 - reduction (operador: lista_variables)
- Dentro de un constructor *parallel* o worksharing:
 - Se hace una copia local de cada variable de la lista y se inicializa dependiendo de la operación.
 - Los compiladores buscan expresiones estándar de reducción que contengan al operador y la usan para actualizar su copia local.
 - Las copias locales son reducidas a un único valor y se combinan con el valor global original.



Reducciones en OpenMP

Las variables del “argumento” lista_variables deben ser del tipo compartidas dentro de la región *parallel*.

```
double prom=0.0, A[MAX];  
int i;  
#pragma omp parallel for reduction (+:prom)  
for(i=0;i< MAX; i++){  
    prom = prom + A[i];  
}  
  
prom = prom/MAX;
```

Reducciones en OpenMP.

Table: Operadores aritméticos.

Operador	Valor inicial
+	0
*	1
-	0

Reducciones en OpenMP.

Table: Operadores lógicos en C.

Operador	Valor inicial
&	~ 0
	0
^	0
&&	1
	0

Reducciones en OpenMP.

Table: Operadores de Fortran.

Operador	Valor inicial
.AND.	.true.
.OR.	.false.
.NEQV.	.false.
.IEOR.	0
.IOR.	0
.EQV.	.true.
MIN	Positivo más grande
MAX	Negativo más pequeño.

Integración numérica

Programa OpenMP versión 3.

Consideraciones.

- Del programa que calcula pi, paralelizarlo usando un constructor de ciclo.
- La idea es usar un número mínimo de cambios, hechos sobre la versión secuencial.



Sincronización.

Cláusula *barrier*.

La cláusula **barrier** obliga a que cada thread espere hasta que todos lleguen al mismo punto.

```
#pragma omp parallel shared (A, B, C) private(id)
{
    mi_id=omp_get_thread_num();
    A[id] = funcion1(mi_id);
    #pragma omp barrier
    #pragma omp for
        for(i=0; i<N; i++) { C[i] = funcion3(i,A); }
    #pragma omp for nowait
        for(i=0;i<N;i++){ B[i] = funcion2(C, i); }
    A[id] = funcion_4(id);
}
```



El constructor **master**

Permite hacer un bloque que solo ejecuta el thread principal.

```
#pragma omp parallel
{
    funcion1 ();
    #pragma omp master
    {
        funcion_importante ();
    }
    #pragma omp barrier
    otra_funcion ();
}
```



El constructor **single**

- Permite hacer un bloque que solo ejecuta un único thread cualquiera.
- Existe una barrera implícita al final del bloque.

```
#pragma omp parallel
{
    funcion1 ();
    #pragma omp single
    {
        funcion_importante ();
    }
}
```


Sincronización.

Cláusula *ordered*.

Una región *ordered* se ejecuta en orden secuencial.

```
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:res)
    for(i = 0; i < N; i++){
        tmp = FUNCION_X(i);
#pragma ordered
        res = res + consume(tmp);
    }
```



OpenMP y las variables de ambiente.

- Existen varias variables de ambiente que permiten modificar el comportamiento de un programa OpenMP.
- Definir el número de threads a usar.
 - OMP_NUM_THREADS
- Controlar como trabajará el calendarizador (scheduler) en los ciclos.
 - OMP_SCHEDULE
- Además de otras.

https://www.atmos.washington.edu/ovens/junk/ifc7docs/f_ug/par_var.htm



Ahora π calculado con Monte Carlo.

Hacer un programa en OpenMP que calcule π usando el método de Monte Carlo.



Contenido

- 1 Introducción al cómputo paralelo.
 - Algunos conceptos importantes.
 - Entendiendo el rendimiento.
 - Paralelismo.
- 2 Programando con OpenMP.
 - Introducción
 - Usando OpenMP
- 3 Programando con MPI
 - **Introducción**
 - MPI. Características.
 - Usando MPI



Una concesión.

Comenzar con MPI es relativamente fácil, con 6 funciones, todo trabaja.



Introducción.

El modelo de paso de mensajes.

- Cada unidad de cómputo (CPU) cuenta con sus propios recursos.
- La única forma de comunicarse con otra unidad es mediante un sistema de comunicación “externo”.
 - Usualmente es una red, que puede o no ser de alta velocidad.
- Para poder trabajar en conjunto, es necesario intercambiar información entre las múltiples unidades de cómputo.
- Ese intercambio se realiza mediante mensajes que emplean el medio de comunicación “externo”.
- Es necesario ocasionalmente sincronizarse entre las diferentes unidades de cómputo.



Origen MPI

- El intercambio de mensajes es un modelo de programación paralela utilizado en máquinas de memoria distribuida.
- Al principio cada empresa definía sus propias bibliotecas, provocando que el software no fuera portable.
- Algunas bibliotecas demostraron que es posible tener un ambiente de intercambio de mensajes portátil de manera eficiente.
- El principal objetivo del desarrollo de MPI fue la creación de un ambiente con una sintaxis y semántica de rutinas estándar.



Contenido

- 1 Introducción al cómputo paralelo.
 - Algunos conceptos importantes.
 - Entendiendo el rendimiento.
 - Paralelismo.
- 2 Programando con OpenMP.
 - Introducción
 - Usando OpenMP
- 3 Programando con MPI
 - Introducción
 - **MPI. Características.**
 - Usando MPI



Características de MPI

- Existen implementaciones gratuitas.
- Es posible hacer comunicaciones asíncronas.
- Grupos de procesos sólidos y eficientes.
- Maneja eficientemente los *buffers*.



Características de MPI

- Permite la programación eficiente.
- Es portable
 - Por qué es un estándar.
 - Y está formalmente especificado.



Aspectos importantes sobre MPI

- Dispone de rutinas para comunicaciones punto a punto.
- Dispone de rutinas para comunicaciones colectivas.
- Se pueden crear grupos de procesos.
- Se pueden definir contextos de comunicación.
- Se pueden definir topologías.
- Funciona con Fortran 77/90/95 y con C/C++.



Lo que no hace MPI.

- Operaciones de memoria compartida.
- Herramientas para la construcción de programas paralelos.
- Esquemas para depuración.
 - Es necesario recurrir a herramientas externas.
- Soporte explícito de *threads*.
- Manejo de tareas.



¿Como usar MPI? (I).

- MPI es un estándar.
- Que puede ser implementado por una empresa o un grupo.
- En forma de una biblioteca.
- Que se puede usar en el código fuente.
- De lenguaje C/C++ o Fortran 77/90/95.



¿Como usar MPI? (II).

- Preguntar si ya está instalado, o ...
- Instalarlo usando el gestor de paquetes de su distribución Linux, o ...
- Descargar el código fuente de openmp:
<https://www.open-mpi.org/software/ompi/v1.10/>
 - Compilarlo e instalarlo.
- O hacer lo anterior con MPICH:
<https://www.mpich.org/downloads/>



¿Como usar MPI? (III).

- O usar una versión de algún distribuidor.
- Compilar el código fuente de los programas con mpicc, mpicpp, mpif77 o mpif90
- Ejecutar el programa dentro del ambiente mpi.
 - mpirun -np programa_mpi parametros



¿Como usar MPI? (IV).

Revisar el estándar para ver tipos de argumentos para las rutinas de biblioteca.

- IN: Argumento(s) de entrada para la rutina que solo se consume y no se altera.
- OUT: Argumento(s) que son utilizados para colocar los valores resultantes de las operaciones dentro de la rutina.
- INOUT: Argumento(s) necesarios de entrada que además se utilizarán para la salida.



Elementos básicos de MPI.

- Rango: El modo en que MPI identifica a los diversos procesos. (Un entero positivo comenzando en cero).
- Etiquetas: La forma en que diferentes mensajes se pueden diferenciar entre el mismo emisor-receptor.
- Grupos: Conjunto de procesos que trabajan concurrentemente.
- Comunicadores: canales de comunicación entre procesos pertenecientes a un grupo y dentro de un contexto.



Tipos de rutinas en MPI.

- Bloqueantes y no bloqueantes.
- Locales y no-locales.
- Colectivas y punto a punto.



Tipos de datos MPI

MPI define sus propios tipos de datos base, por ejemplo:

- MPI_CHAR
- MPI_SHORT
- MPI_INT
- MPI_DOUBLE
- MPI_LONG_DOUBLE



Constantes dentro de MPI

Se definen un conjunto de constantes utilizados por algunas de las rutinas, que tienen un significado especial.

Algunos ejemplos:

- `MPI_COMM_WORLD`
- `MPI_ANY_TAG`
- `MPI_ANY_SOURCE`



Contenido

- 1 Introducción al cómputo paralelo.
 - Algunos conceptos importantes.
 - Entendiendo el rendimiento.
 - Paralelismo.
- 2 Programando con OpenMP.
 - Introducción
 - Usando OpenMP
- 3 Programando con MPI
 - Introducción
 - MPI. Características.
 - Usando MPI



Funciones básicas.

Las que no deben faltar:

Inicialización

- `MPI_Init(int *argc, char ***argv);` // En lenguaje C
- `MPI_INIT(error) !!` En lenguaje Fortran

Y terminación del ambiente.

- `MPI_Finalize();` // En C
- `MPI_FINALIZE(error) !!` En Fortran.

Funciones básicas.

Para conocer la cantidad de procesos:

- `MPI_Comm_size(MPI_comm comunicador, int *tam);`
- `MPI_COMM_SIZE(com, tam, error)`

Y el identificador de cada proceso.

- `MPI_Comm_rank(MPI_comm comunicador, int *ident); // O rango (rank)`
- `MPI_COMM_RANK(com, identificador, error)`

Hola mundo con MPI

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

int main( int argc, char* argv[] ) {
    int mi_id, procs;
    // Inicializacion de la parte paralela.
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &procs );
    MPI_Comm_rank( MPI_COMM_WORLD, &mi_id );

    printf( "Hola_mundo!_soy_el_proceso_n._%d_de_un_total_
            de_%d_procesos.\n",
            mi_id, procs );

    MPI_Finalize();
}
```



Compilando y ejecutando

- Para compilar es necesario usar un compilador que entienda MPI
 - mpicc
 - mpif77
 - mpif90
 - mpicxx
- Y un entorno de ejecución
 - mpirun -np num_procs /ruta/del/ejecutable argumentos



Comunicaciones punto-a-punto. Bloqueantes (I).

Envío

- `MPI_Send(void *buffer, int cant, MPI_Datatype tipo, int destino, int tag, MPI_Comm comunicador);`
- `MPI_SEND(buffer, cantidad, tipo, destino, tag, comunicador, error)`

Recepción.

- `MPI_Recv(void *buffer, int cant, MPI_Datatype tipo, int origen, int tag, MPI_Comm comunicador, MPI_Status status);`
- `MPI_RECV(buffer, cantidad, tipo, destino, tag, comunicador, status, error)`



Comunicaciones punto-a-punto. No bloqueantes (II).

Envío

- `MPI_Isend(void *buffer, int cant, MPI_Datatype tipo, int destino, int tag, MPI_Comm comunicador, MPI_Request *request);`
- `MPI_ISEND(buffer, cantidad, tipo, destino, tag, comunicador, request, error)`

Recepción.

- `MPI_Irecv(void *buffer, int cant, MPI_Datatype tipo, int origen, int tag, MPI_Comm comunicador, MPI_Request *request);`
- `MPI_IRECV(buffer, cantidad, tipo, destino, tag, comunicador, request, error)`



Comunicaciones punto-a-punto. No bloqueantes (III).

Para finalizar una operación no bloqueante.

Se puede hacer una sincronización.

- `MPI_Wait(MPI_Request *request, MPI_Status *status)`
- `MPI_WAIT(request, status, error)`

O se puede hacer una evaluación.

- `MPI_Test(MPI_Request *request, int bandera, MPI_Status *status)`
- `MPI_TEST(request, bandera, status, error)`



Un poco de luz.

Un ejemplo.

```
CALL MPI_COMM_RANK(com, mi_id, i_error)
IF (mi_id == 0) THEN
    CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, com,
        request, i_error)
    !! Calculos mientras se hace la comunicacion.
    CALL MPI_WAIT(request, status, i_error)
ELSE
    CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, com,
        request, i_error)
    CALL MPI_WAIT(request, status, i_error)
END IF
```



Operaciones colectivas.

Se dividen en:

- Sincronización.
- Distribución de datos
- Cálculos colectivos.



Operaciones de sincronización.

- Existe una que es la barrera.
- Ningún proceso continúa hasta que todos hayan llegado, al mismo punto del programa.
 - `MPI_Barrier(MPI_Comm comunicador);`
 - `MPI_BARRIER(comunicador, error)`



Distribución de datos (I).

Difusión

- `MPI_Bcast(void *buffer, int cantidad, MPI_Datatype tipo, int raíz, MPI_Comm comunicador)`
- `MPI_BCAST(buffer, cantidad, tipo, raíz, comunicador, error)`



Distribución de datos (II).

Recolección

- `MPI_Gather(const void *buf_envio, int cuenta_envio, MPI_Datatype tipo_env, void *buf_rec, int cuenta_rec, MPI_Datatype tipo_rec, int raíz, MPI_Comm comunicador)`
- `MPI_GATHER(buf_envio, cuenta_envio, tipo_env, buf_rec, cuenta_rec, tipo_rec, raíz, comunicador, error)`



Distribución de datos (III).

Dispersión.

- `int MPI_Scatter(const void *buf_env, int cuenta_env, MPI_Datatype tipo_envio, void *buf_rec, int cuenta_rec, MPI_Datatype tipo_rec, int raíz, MPI_Comm comm)`
- `MPI_SCATTER(buf_env, cuenta_env, tipo_env, buf_rec, cuenta_rec, tipo_rec, raíz, comunicador, error).`



Cálculos colectivos

- `MPI_Reduce(const void *buf_env, void *buf_rec, int cuenta, MPI_Datatype tipo, MPI_Op operacion, int raíz, MPI_Comm comunicador)`
- `MPI_REDUCE(buf_env, buf_rec, cuenta, tipo, operación, raíz, comunicador, error)`
- `MPI_Allreduce(const void *buf_env, void *buf_rec, int cuenta, MPI_Datatype tipo, MPI_Op operacion, MPI_Comm comunicador)`
- `MPI_ALLREDUCE(buf_env, buf_rec, cuenta, tipo, operación, raíz, comunicador, error)`



Continuará,...

Esta es aún una versión sin terminar...

