# UNIDAD III. CÓMPUTO PARALELO

Francisco J. Hernández López

fcoj23@cimat.mx

# ¿QUÉ ES EL CÓMPUTO PARALELO (CP)?

- Ejecución de más de un cómputo (cálculo) al mismo tiempo o "en paralelo", utilizando más de un procesador.

Sistema de Cómputo Paralelo

Hardware

Software

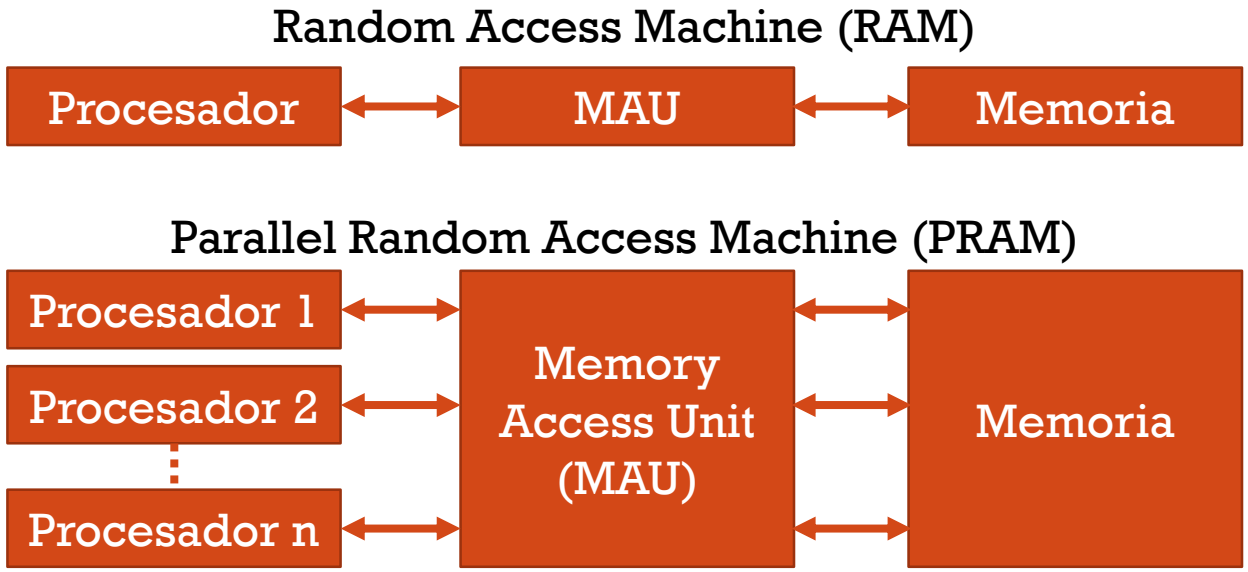
*Parallel programming: For multicore and cluster systems.* Rauber, Thomas, and Gudula Rünger. Springer Science & Business Media, 2013.

# MODELOS DE PROGRAMACIÓN EN PARALELO

- Modelo máquina
  - Es el nivel más bajo de abstracción.
  - Consiste en una descripción de HW y el SO. Ejemplo: Los registros y la entrada/salida de los buffers.
  - El lenguaje ensamblador esta basado en este nivel de modelos.
- Modelo de arquitectura
  - Interconexión de red de plataformas paralelas.
  - Organización de la memoria
  - Procesamiento síncrono o asíncrono.
  - Modelo de ejecución SIMD (Simple Intruction Multiple Data) o MIMD (Multiple Instruction Multiple Data)

# MODELOS DE PROGRAMACIÓN EN PARALELO (C1)

- Modelo computacional
  - Provee un método analítico para diseñar y evaluar algoritmos
  - La complejidad de un algoritmo se debe ver reflejado en el rendimiento de la computadora



# MODELOS DE PROGRAMACIÓN EN PARALELO (C2)

- Modelo de programación
  - Describe un sistema de cómputo paralelo en términos de la semántica del lenguaje de programación o el ambiente de programación.
  - Especifica al programador como codificar un algoritmo paralelo.
  - Influenciado por: La arquitectura, el compilador, librerías, etc.
  - Criterios por los cuales pueden ser diferentes:
    - El nivel de paralelismo.
    - Especificaciones explicitas definidas por el usuario.
    - El modo de ejecución: SIMD, MIMD, Síncrono o Asíncrono.
    - Modos y patrones de comunicación para el intercambio de información.
    - Mecanismos de sincronización.

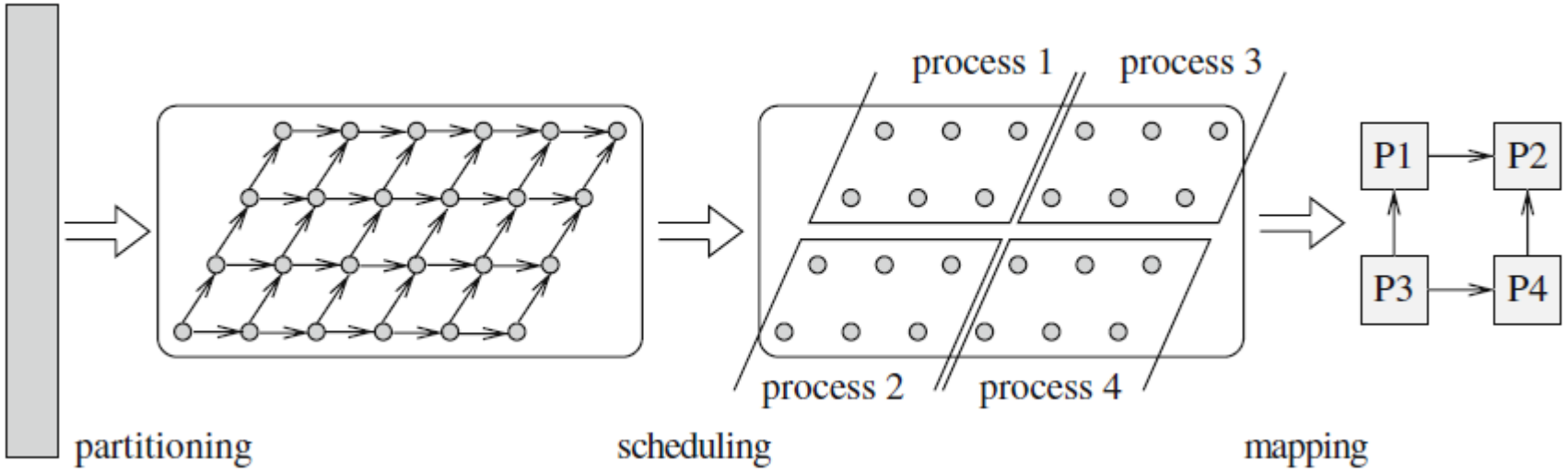
# PARALELIZACIÓN DE PROGRAMAS

- Asumiendo que el cómputo que se pretende paralelizar esta dado a partir de un algoritmo o programa secuencial.
- Se debe considerar el control y dependencia de los datos.
- Asegurar que el programa paralelo produce los mismos resultados que el programa secuencial.

Objetivo: Reducir el tiempo de ejecución de un programa tanto como sea posible, usando múltiples procesadores o cores.

# POSIBLES PASOS PARA LA PARALELIZACIÓN

1. Descomposición de los cálculos computacionales
2. Asignación de tareas a procesar o hilos
3. Mapeo de procesos o hilos a procesadores o cores físicos




Tomado de: *Parallel programming: For multicore and cluster systems.* Rauber, Thomas, and Gudula Rünger. Springer Science & Business Media, 2013.

# DESCOMPOSICIÓN DE LOS CÁLCULOS COMPUTACIONALES

- Aquí, los cálculos del algoritmo secuencial se descomponen en tareas y se determinan las dependencias entre la tareas.
- Dependiendo del modelo de la memoria, una tarea puede involucrar accesos a espacios de *direcciones compartidas* o puede ejecutar operaciones de *paso de mensajes*.
- Dependiendo de la aplicación, la descomposición de tareas se puede llevar a cabo:
  - En la fase de Inicialización → Descomposición Estática
  - Durante la ejecución del Programa → Descomposición Dinámica

Objetivo: Generar suficientes tareas para mantener a todos los cores ocupados todo el tiempo.

# ASIGNACIÓN DE TAREAS A PROCESAR O HILOS

- Un procesador o hilo:
  - Representa un flujo de control ejecutado por un procesador físico o core.
  - Puede ejecutar diferentes tareas, una por una.
- El número de hilos no necesariamente va ser el mismo que el número de cores físicos en un problema, aunque se suele utilizar de esta manera.
- La asignación de tareas a hilos también se le conoce como "Scheduling", puede haber Scheduling estático o dinámico.

Objetivo: Asignar las tareas tal que se obtenga un buen "Balance de Carga".

# MAPEO DE HILOS A CORES FÍSICOS

- El caso más simple es cuando cada proceso o hilo es mapeado a un solo core.
- Si ∃ menos cores que hilos, entonces múltiples hilos deben ser mapeados a un solo core. Este mapeo puede ser hecho por:
  - El Sistema Operativo (SO).
  - Soportado por sentencias del programa.

Objetivo: Tener a todos los cores trabajando, mientras que la comunicación entre ellos sea mínima.

# NIVELES DEL PARALELISMO

- **A nivel de bits**
- **A nivel de instrucción**
- **A nivel de datos**
- **A nivel de tareas**

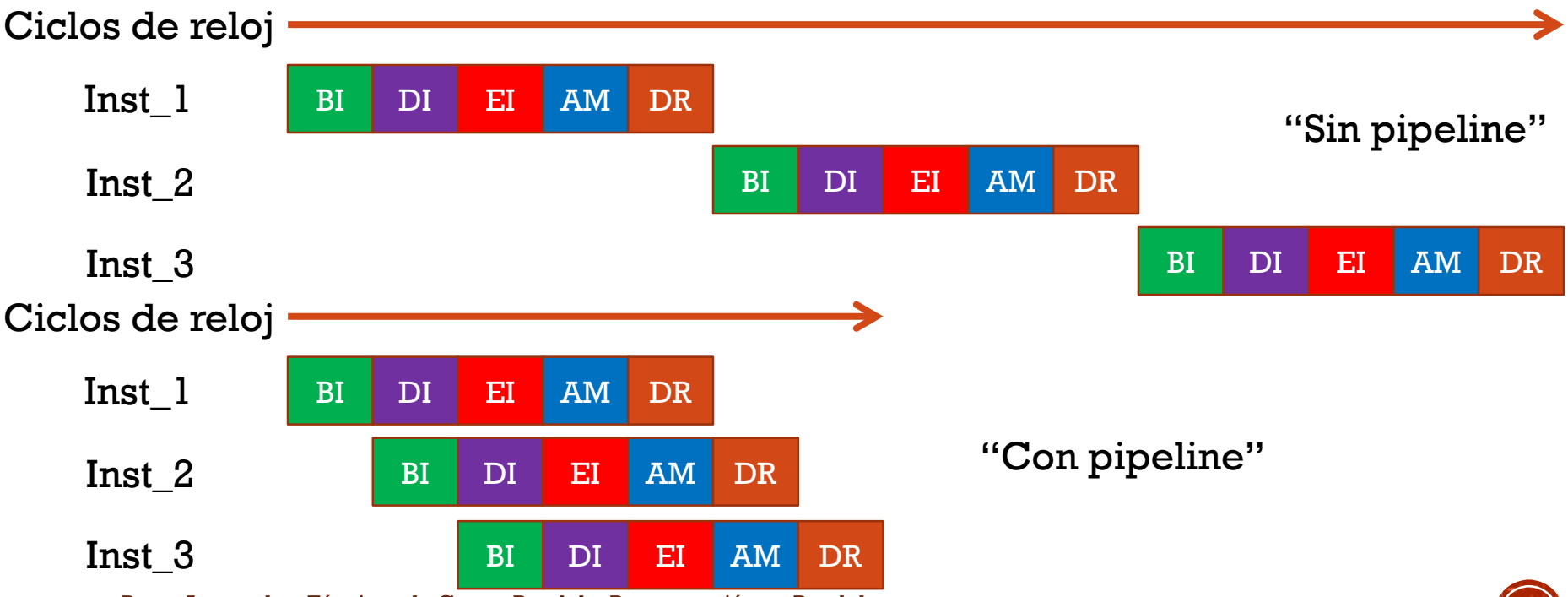
# A NIVEL DE BITS


- Basado en incrementar el numero de bits de una palabra (*word length, word size* o *word width*)
- Importante característica de los procesadores (8,16, 32 o 64 bits)
- Ejemplo:
  - Tenemos un procesador de 16 bits y queremos sumar dos enteros (int) de 32 bits
  - Primero el procesador realiza la suma de los primeros 16 bits y después de los últimos 16 bits, completando la suma de los int en dos instrucciones
  - Un procesador mayor o igual a 32 bits realizaría la operación en una sola instrucción.

# A NIVEL DE INSTRUCCIÓN

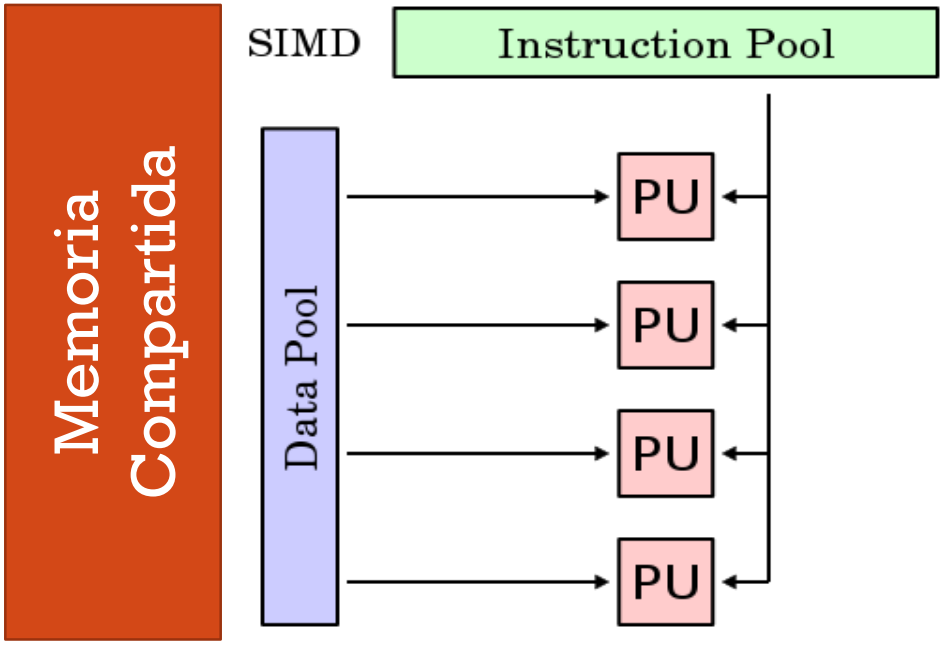
- Capacidad de traslapar instrucciones
- Depende del pipeline del procesador

BI → Buscar Instrucción
DI → Descifrar Instrucción
EI → Ejecutar Instrucción
AM → Acceso a Memoria
DR → Dar Respuesta



# A NIVEL DE DATOS

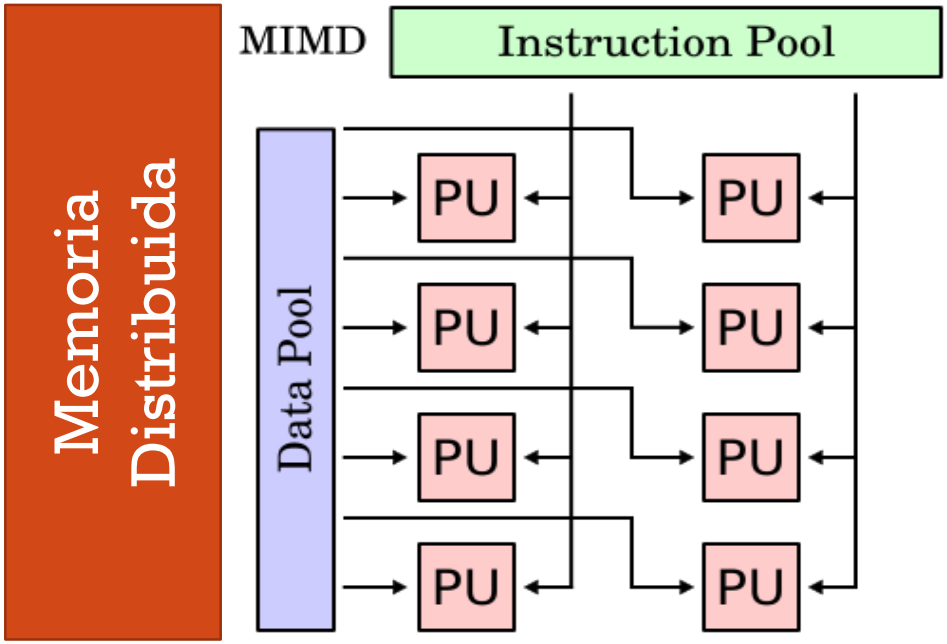
- Cada procesador realiza la misma tarea en diferentes partes de los datos (SIMD → Single Instruction Multiple Data)




SIMD [wikipedia]

# A NIVEL DE TAREAS

- Diferentes procesadores pueden estar ejecutando diferentes instrucciones en diferentes partes de los datos (MIMD)




MIMD [wikipedia]

# ARQUITECTURAS PARA EL CP


Computadora multi-core

Cluster

GPUs


FPGAs

Procesadores vectoriales


Xeon Phi Coprocessor

# LENGUAJES DE PROGRAMACIÓN PARA EL CP

**OpenMP** (Memoria Compartida)

**OpenMPI** (Memoria Distribuida)

**CUDA** (***C****ompute* ***U****nified* ***D****evice* ***A****rchitecture*) para GPUs.

**OpenCL (Open C**omputing **L**anguage**)** para GPUs y CPUs.

## Mezcla de Arquitecturas:

Computadora (o múltiples cores) y un GPU (o un arreglo de GPUs) .

Cluster con computadoras que contienen múltiples cores y a su vez cada maquina tiene un GPU (o un arreglo de GPUs).

Así como mezclamos las arquitecturas también mezclamos los lenguajes:

- CUDA con (OpenMP, OpenMPI).
- OpenCL con (OpenMP, OpenMPI).

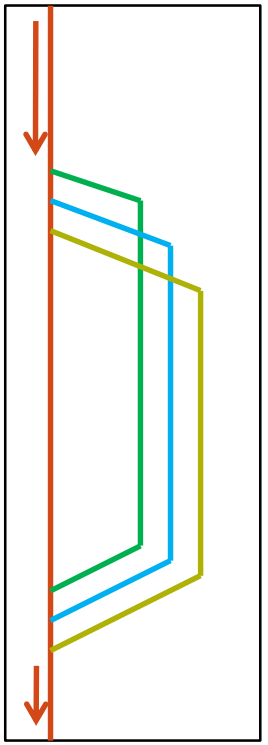
# OPENMP

- Estándar para programación en paralelo con memoria compartida
- Corre en sistemas multicore
- Existen también versiones de OpenMP para GPUs y Clusters
- Podemos programar usando OpenMP en Fortran, C/C++, Java, Phyton…
- WebPage: http://openmp.org/wp/

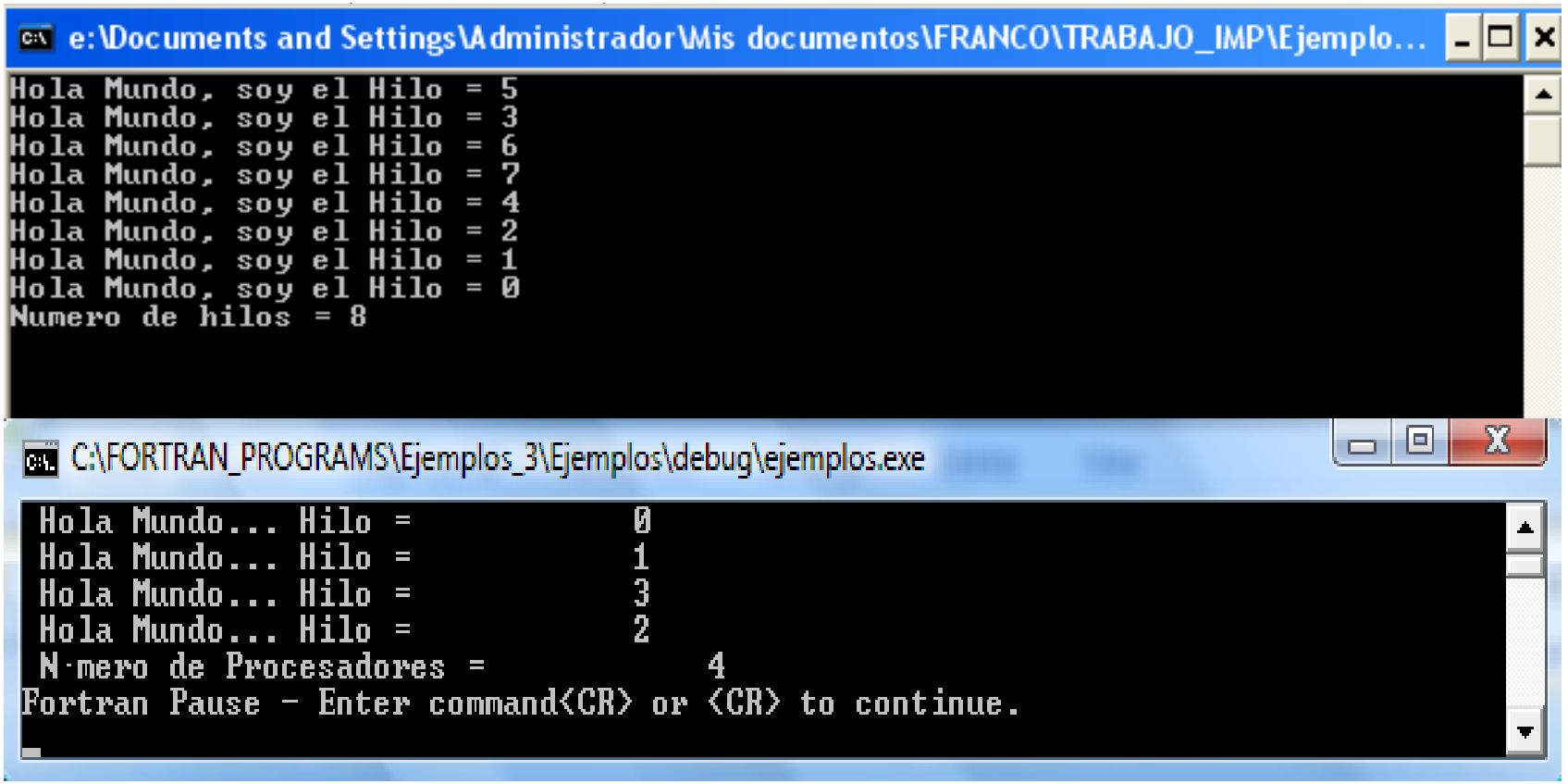
# "HELLO WORLD" EN OPENMP

HolaMundo.f

```
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c     Programa: HolaMundo utilizando OpenMP
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccc

      PROGRAM HolaMundo
          IMPLICIT NONE
          INTEGER NTHREADS,TID,OMP_GET_NUM_THREADS,OMP_GET_THREAD_NUM

!$OMP PARALLEL PRIVATE(TID)
C         Obtenemos e imprimimos el ID del THREAD.
          TID = OMP_GET_THREAD_NUM()
          PRINT *, 'Hola Mundo... Hilo = ', TID

C         Este bloque lo hace solo el THREAD 0
          IF (TID .EQ. 0) THEN
            NTHREADS = OMP_GET_NUM_THREADS()
            PRINT *, 'Número de Procesadores = ', NTHREADS
          END IF
!$OMP END PARALLEL

          pause
      END PROGRAM HolaMundo
```



# RESULTADO DEL "HELLO WORLD" EN OPENMP



# DIRECTIVAS EN OPENMP (C1)

- **Región Paralela:**
  - **!$**OMP PARALLEL, **!$**OMP END PARALLEL. (Fortran)
  - **#**pragma omp parallel **{ }**. (C/C++)

- **División del Trabajo:**
  - !$OMP **DO**, !$OMP END DO (Fortran)
  - #pragma omp **for** (C/C++)

  } Especifica la ejecución en paralelo de un ciclo de iteraciones.

  - !$OMP **SECTIONS**, !$OMP END SECTIONS
  - #pragma omp **sections** {}

  } Especifica la ejecución en paralelo de algún bloque de código secuencial.

  - !$OMP **SINGLE**, !$OMP END SINGLE
  - #pragma omp **single**

  } Define una sección de código donde exactamente un Hilo tiene permitido ejecutar el código.

# DIRECTIVAS EN OPENMP (C2)

- **Combinaciones de directivas:**
  - !$OMP **PARALLEL DO**, !$OMP END PARALLEL DO. (Fortran)
  - #pragma omp **parallel for** {} (C/C++)
  - !$OMP **PARALLEL SECTIONS**, !$OMP END PARALLEL SECTIONS (Fortran)
  - #pragma omp **parallel sections** {} (C/C++)

# DIRECTIVAS EN OPENMP (C3)

- **Sincronización:**
  - **CRITICAL**: Solo un Hilo a la vez tiene permitido ejecutar el código.
  - **ORDERED**: Asegura que el código se ejecuta en el orden en que las iteraciones se ejecutan de forma secuencial.
  - **ATOMIC**: Asegura que una posición de memoria se modifique sin que múltiples Hilos intenten escribir en ella de forma simultánea.
  - **FLUSH**: El valor de las variables se actualiza en todos los hilos (ejemplo: banderas).
  - **BARRIER**: Sincroniza todos los hilos.
  - **MASTER**: El código lo ejecuta sólo el hilo maestro (No implica un Barrier).

# DIRECTIVAS PARA MANIPULAR LOS DATOS

- ***private*(lista):** Las variables de la lista son privadas a los hilos, lo que quiere decir que cada hilo tiene una variable privada con ese nombre

- ***firstprivate*(lista):** Las variables son privadas a los hilos, y se inicializan al entrar con el valor que tuviera la variable correspondiente

- ***lastprivate*(lista):** Las variables son privadas a los hilos, y al salir quedan con el valor de la última iteración (si estamos en un bucle do paralelo) o sección

# DIRECTIVAS PARA MANIPULAR LOS DATOS (C1)

- ***shared*(lista):** Indica las variables compartidas por todos los hilos

- ***default*(shared|none):** Indica cómo serán las variables por defecto. Si se pone *none* las que se quiera que sean compartidas habrá que indicarlo con la cláusula *shared*

- ***reduction*(operador:lista):** Las variables de la lista se obtienen por la aplicación del operador, que debe ser asociativo

# CLAUSULA SCHEDULE DE OPENMP

- Utilizada con la directiva **DO** o **for**
- Especifica un algoritmo de planificación, que determina de qué forma se van a dividir las iteraciones del ciclo entre los hilos disponibles
- Se debe especificar un tipo y, opcionalmente, un tamaño (*chunk*)
- Tipos:
  - **STATIC**.
  - **DYNAMIC**.
  - **GUIDED**.
  - **RUNTIME**.

# EJEMPLO: SUMA DE VECTORES EN OPENMP

- Suma de vectores (c_h = a_h + b_h)
  - Creamos los vectores "a_h", "b_h" y "c_h" en el host
  - Inicializamos con cualquier valor los vectores "a_h" y "b_h" (Paralelizar con OpenMP esta parte)
  - Sumamos a_h y b_h; el resultado lo guardamos en c_h (Paralelizar con OpenMP esta parte)
  - Desplegamos la suma de los vectores.

# EJEMPLO: SUMA DE VECTORES EN OPENMP

```
// Código principal que se ejecuta en el Host
int main(void){
    float *a_h, *b_h, *c_h; //Punteros a arreglos en el Host
    long int N = 100000000;  //Número de elementos en los arreglos
    long int i;
    struct timeb start, end;
    int diff;
    size_t size_float = N * sizeof(float);

    a_h = (float *)malloc(size_float); // Pedimos memoria en el Host
    b_h = (float *)malloc(size_float);
    c_h = (float *)malloc(size_float);//También se puede con cudaMallocHost
    for (i = 0; i<N; i++){
        a_h[i] = (float)i;
        b_h[i] = (float)(i + 1);
    }
    printf("\n\nComenzando el procesamiento...");
    ftime(&start);
    Suma_vectores(c_h, a_h, b_h, N);
    ftime(&end);
    diff = (int)(1000.0 * (end.time - start.time)
        + (end.millitm - start.millitm));
    printf("\nProcesamiento Finalizado...");
    printf("\nTiempo: %u milisegundos...\n", diff);

    // Liberamos la memoria del Host
    free(a_h);
    free(b_h);
    free(c_h);
    return(0);
}
```

```
void Suma_vectores(float *c, float *a, float *b, long int N)
{
    long int i;
#pragma omp parallel shared(a,b,c,N) private(i)
    {
#pragma omp for nowait
        for (i = 0; i<N; i++){
            //c[i] = a[i] + b[i];
            c[i] = cos(a[i]) + (int)(b[i])%3;
        }
    }
}
```