



Dr. Cristian Mateos Diaz  
Computación Paralela y Distribuida - 2022

# ¿Por qué OpenMP?

- Usuario destacado: UCAR (Corporación Universitaria de Investigación Atmosférica)
  - Consorcio sin ánimo de lucro que conecta +100 universidades
  - Formación y fomento de la investigación en cambio climático
  - Gestiona el NCAR (Centro Nacional de Análisis Atmosférico Investigar)
  - Patrocinado por NSF (investigación no médica)
  - NCAR/UCAR es para las ciencias atmosféricas y afines lo que La NASA es para la astronomía.

# Supercomputadora NCAR Cheyenne

- 145,152 núcleos de procesador Intel Xeon
- 4032 nodos de dos sockets (36 núcleos/nodo)
- (36 núcleos/nodo)
- 313 TB de memoria total
- 6 nodos de inicio de sesión
- latencia de ping-pong  $< 1 \mu\text{s}$
- <https://www2.cisl.ucar.edu/resources/computational-systems/cheyenne/running-jobs/hyper-threading-cheyenne>

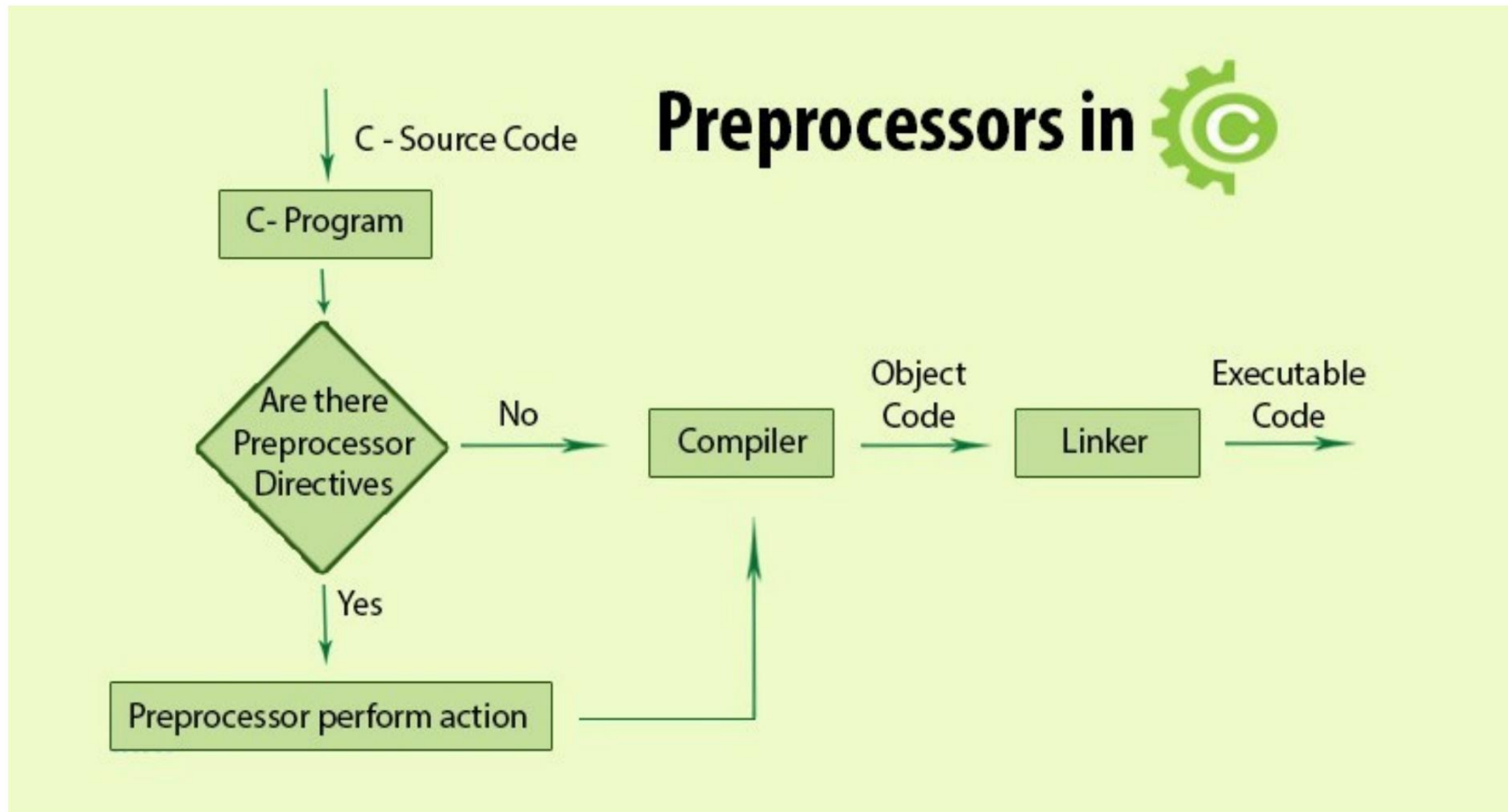


# Descripción general de OpenMP

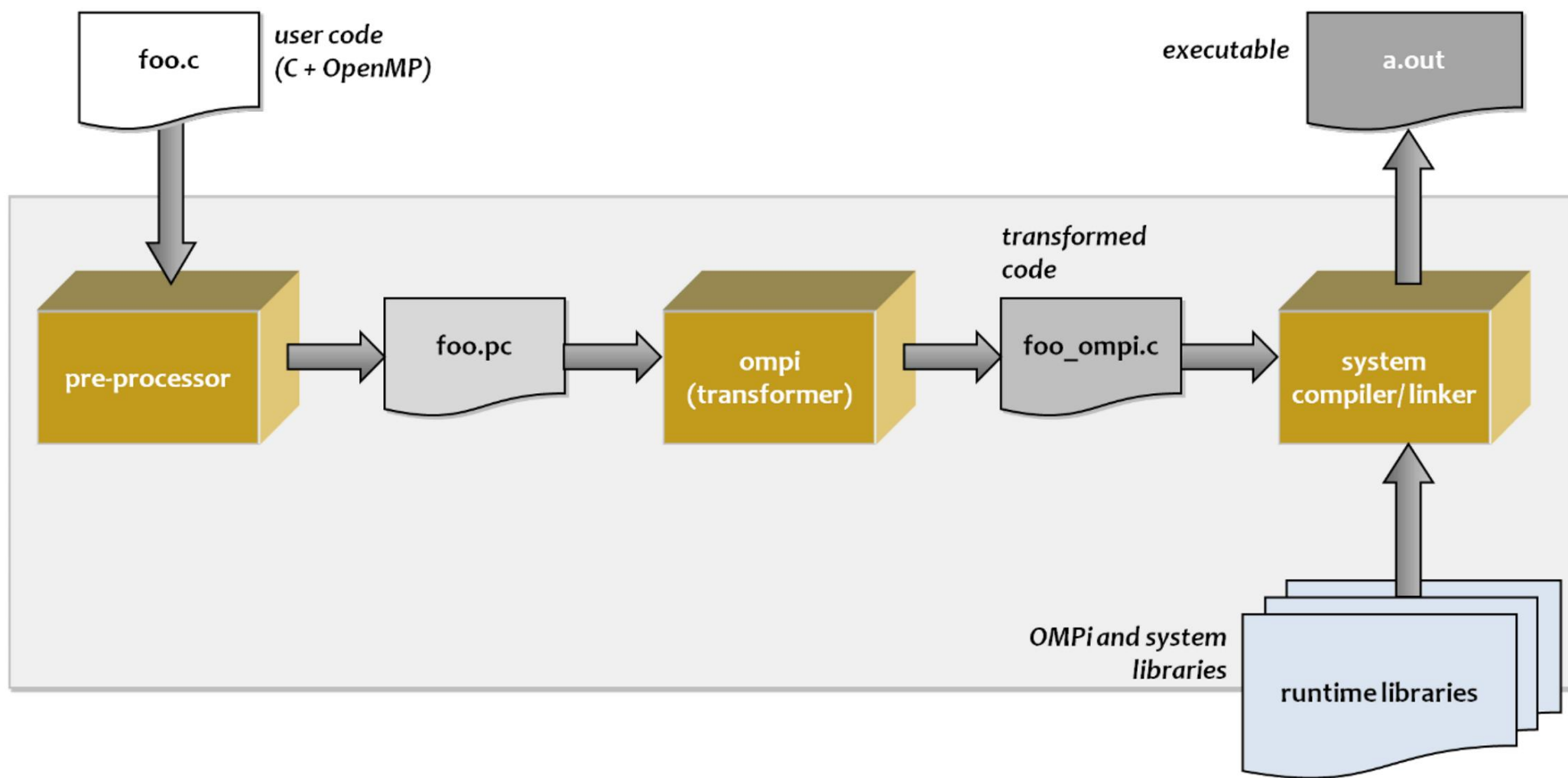
- OpenMP: una API para escribir aplicaciones de subprocesos múltiples
- Un conjunto de directivas de compilación y rutinas de biblioteca para programadores de aplicaciones paralelas
- Simplifica la escritura de programas de subprocesos múltiples (MT) en varios idiomas.
- Estandariza las últimas 20 prácticas de SMP

# Directivas del compilador: resumen

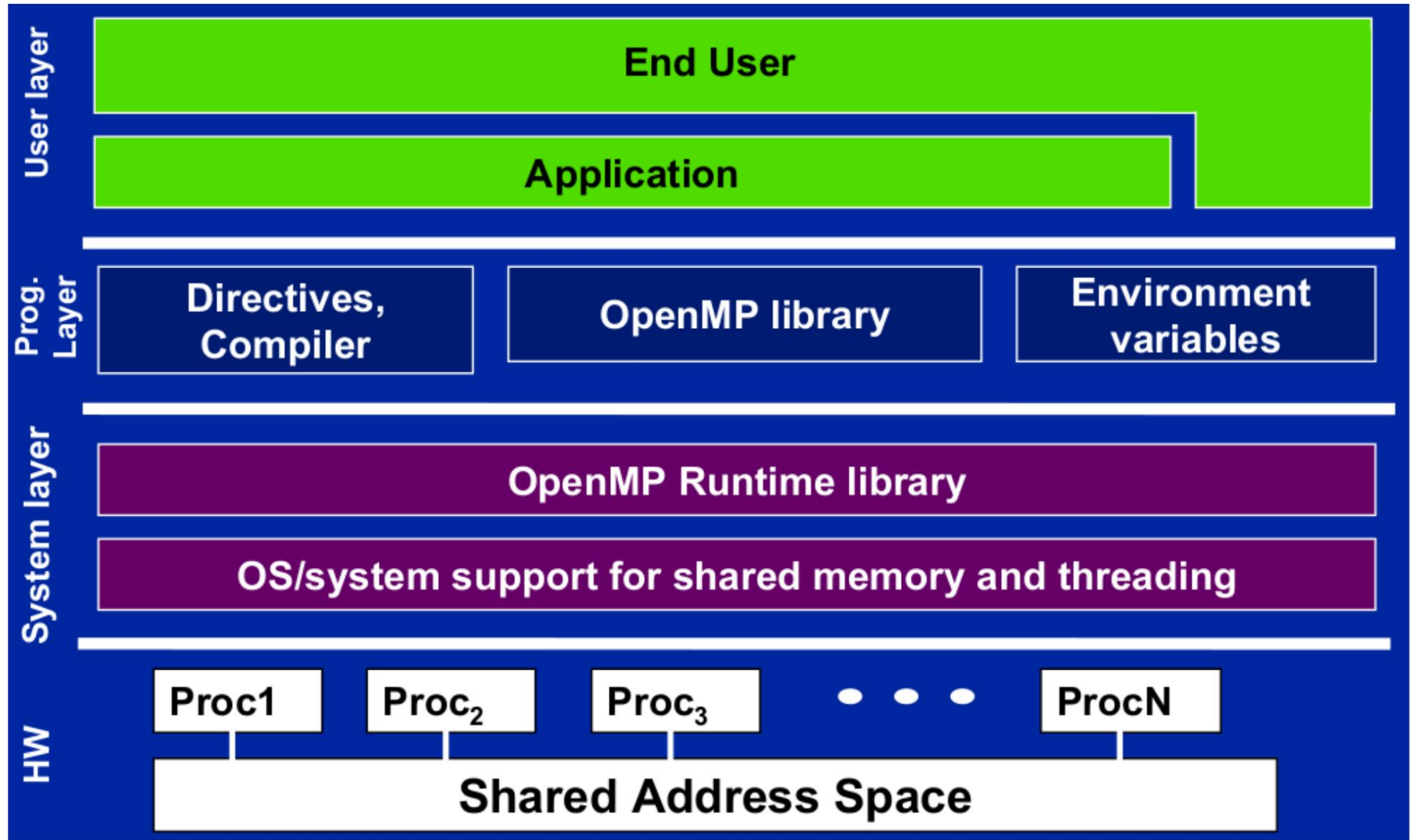
- Ejemplo 1: `#include <stdio.h>`
- Ejemplo 2: `#define PI 3.1415926`



# Generando código intermedio



# Pila de software OpenMP



Alternativa: Comunicación entre procesos a través de mensajes (MPI)

# Sintaxis de OpenMP

- La mayoría de las construcciones en OpenMP son directivas de compilación:

- **#pragma omp construcción [cláusula [cláusula]...]**

- Ejemplo: **#pragma omp paralelo num\_threads(4)**

- Prototipos y tipos de funciones en el archivo:

**#incluir <omp.h>**

- La mayoría de las construcciones de OpenMP se aplican a un "bloque estructurado".

- Bloque estructurado: un bloque de una o más sentencias con una  
punto de entrada en la parte superior y un punto de salida en la parte inferior
  - Está bien tener una salida () dentro del bloque estructurado





# Instalación (ver videotutorial)

- Instalar código::Blocks (<http://www.codeblocks.org/>)
  - Los usuarios de Mac preferirían otro IDE...
- Instalar el compilador GCC
  - No hay nada que hacer para los usuarios de Linux/Mac (compruebe que tiene 'gcc' y 'g++')
  - Usuarios de Windows: Instale MinGW (<http://www.mingw.org/>)
- Dígle a su IDE que use OpenMP
  - Instalar biblioteca (libomp, libomp-dev)
  - bandera “-fopenmp” (compilador)
  - Enlace a la biblioteca (carpeta “gomp” o MinGW)
- Puede usar <http://www.omp4j.org> más, por ejemplo, Eclipse

# Calentando A: Hola mundo

- Primero verifique que su entorno funcione escribiendo un hola mundo programa:

```
void main()
{

    int ID = 0;

    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);

}
```

# Calentando B: Hola mundo

- Luego verifique que su entorno OpenMP funcione escribiendo un programa hola mundo:

```
#include "omp.h"
void main()
{
    #pragma omp parallel
    {

        int ID = 0;

        printf(" hello(%d) ", ID);
        printf(" world(%d) \n", ID);
    }
}
```

- Compílelo usando: gcc -fopenmp -o exec\_name source\_name.c
- Cada hilo puede imprimir su propia ID usando **omp\_get\_thread\_num()**; dentro de la construcción paralela

# ¿Cómo interactúan los hilos?

- OpenMP usa un modelo de dirección compartida de subprocesos múltiples
  - Los hilos se crean implícitamente
  - Los hilos se comunican compartiendo variables
- El intercambio no intencional de datos genera condiciones de carrera:
  - *condición de carrera*: cuando el resultado del programa cambia a medida que los hilos están programados de manera diferente
- Para controlar las condiciones de carrera:
  - Utilice la sincronización para proteger los conflictos de datos
- La sincronización es costosa, por lo que:
  - Cambie la forma en que se accede a los datos para minimizar la necesidad de sincronización

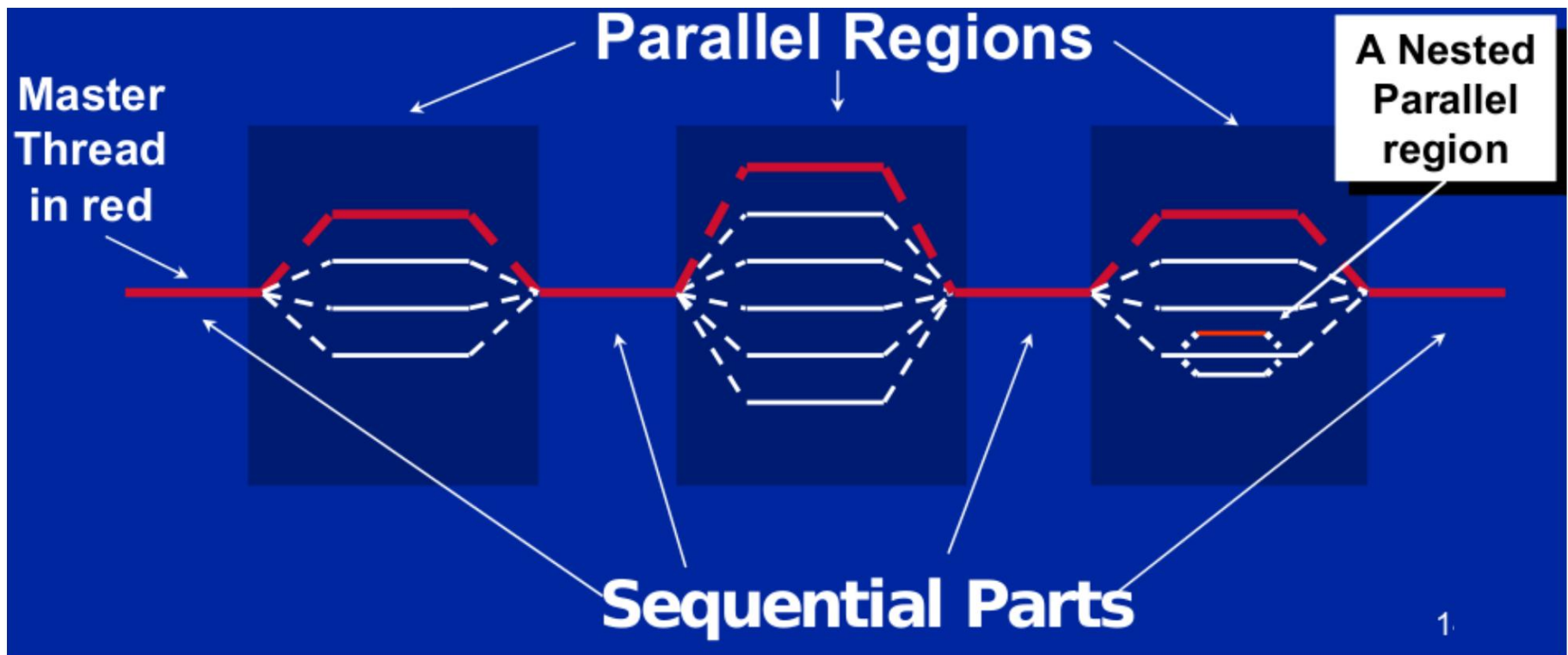
# Describir

- Introducción a OpenMP
- **Creación de hilos (¡usted está aquí!)**
- Sincronización
- Bucles paralelos
- Sincronizar maestros individuales y otras cosas
- Programe su para y secciones



# Modelo de programación OpenMP: Paralelismo bifurcación-uni3n

- *El hilo principal* genera un equipo de hilos seg3n sea necesario.
- El paralelismo se agrega gradualmente hasta que se alcanzan los objetivos de rendimiento: es decir, el programa secuencial evoluciona a uno paralelo



# Creación de subprocesos: paralelo

## Regiones

- Usted "crea" subprocesos en OpenMP con la construcción **paralela**
- Ejemplo: Creación de una región paralela de 4 subprocesos:

Each thread executes a copy of the code within the structured block

```
double A[1000];  
omp_set_num_threads(4);  
#pragma omp parallel  
{  
    int ID = omp_get_thread_num();  
    pooh(ID,A);  
}
```

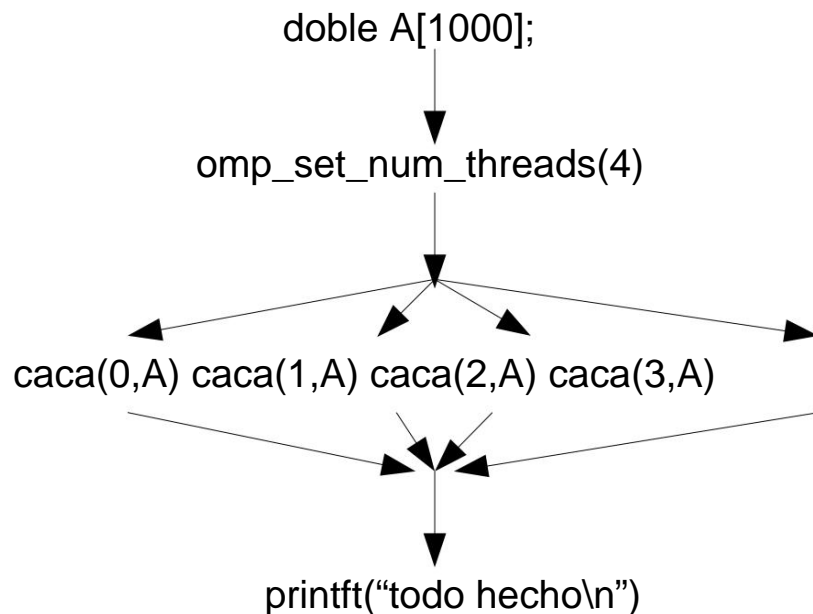
Runtime function to request a certain number of threads

Runtime function returning a thread ID

# Creación de subprocesos: paralelo

## Ejemplo de regiones

- Se comparte una sola copia de "A".
- Se establece una *barrera* al final de la región.

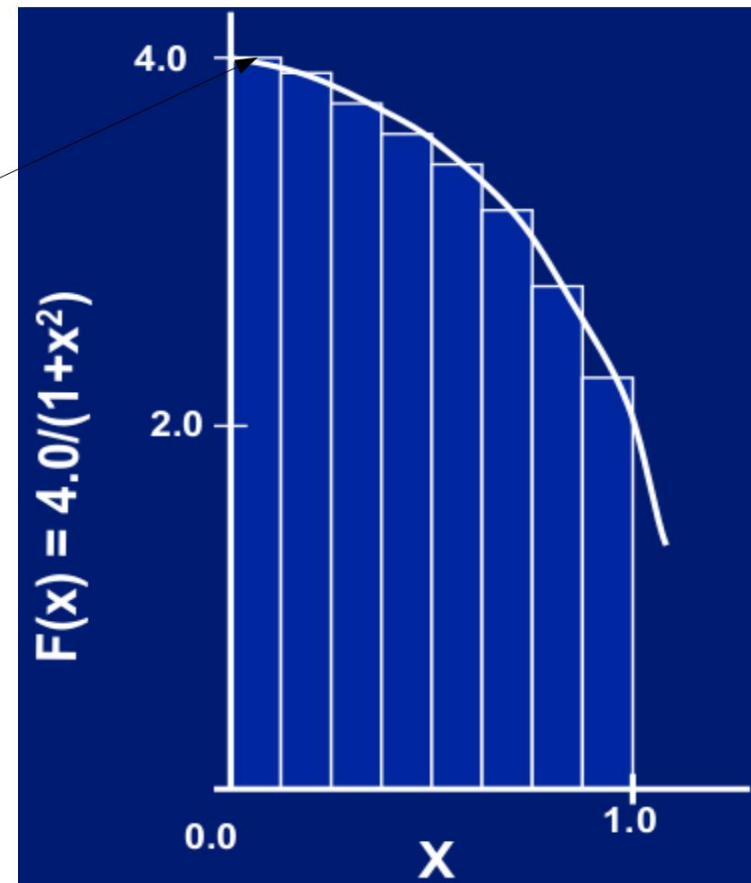


```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```



# Tarea: Integración numérica

- Matemáticamente, integrar  $4.0/(1+X^2)$  en  $[0,1]$  es igual a  $\pi$
- Podemos aproximar la integral como una suma de rectángulos:
  - $\sum F(X_i) \Delta X \approx \pi, i=\{0..N\}$
  - Cada rectángulo tiene ancho  $\Delta X$  y alto  $F(X_i)$  en el medio del intervalo  $i$



# Tarea: Integración numérica

```
static long num_steps = 100000;
double step;
void main ()
{
    int i;  double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;

    for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

# Tarea: Integración numérica

- Cree una versión paralela del programa pi usando un paralelo construir
- Tiempo de ejecución de impresión por subproceso
- Preste mucha atención a las variables compartidas frente a las privadas.
- Además de una construcción paralela, necesitará las rutinas de la biblioteca en tiempo de ejecución:
  - *int omp\_get\_num\_threads();*
  - *int omp\_get\_thread\_num();*
  - *doble omp\_get\_wtime();*




# Describir

- Introducción a OpenMP
- Creación de hilos
- **Sincronización (¡usted está aquí!)**
- Bucles paralelos
- Sincronizar maestros individuales y otras cosas
- Programe su para y secciones



# Sincronización

- Se utiliza para imponer restricciones de orden y para proteger el acceso a datos compartidos
- Sincronización de alto nivel
  - Crítico
  - Atómico
  - Barrera
  - Ordenado



**Discutido  
después**

# Sincronización: crítica

- Exclusión mutua como en un semáforo binario
- `#pragma omp crítica(algúnNombre)` y sección nombrada
- ¡Cuidado! Todas las secciones críticas sin nombre son mutuamente excluyentes

```
float res;  
#pragma omp parallel  
{  float B;  int i, id, nthrds;  
    id = omp_get_thread_num();  
    nthrds = omp_get_num_threads();  
    for(i=id;i<niters;i+nthrds){  
        B = big_job(i);  
#pragma omp critical  
        consume (B, res);  
    }  
}
```

# Sincronización: atómica

- Exclusión mutua, pero solo para proteger las actualizaciones de las variables.
- Mucho más rápido en comparación con los críticos; podría aprovechar la baja instrucciones de nivel para, por ejemplo, variable atómica + 1

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

# Describir

- Introducción a OpenMP
- Creación de hilos
- Sincronización
- **Bucles paralelos (¡usted está aquí!)**
- Sincronizar maestros individuales y otras cosas
- Programe su para y secciones





# SPMD versus trabajo compartido

- Una construcción paralela por sí sola crea un SPMD (Single Program Datos Múltiples) programa de uno secuencial
- Cada subproceso ejecuta de forma redundante el mismo código.
- ¿Cómo se dividen las rutas a través del código entre subprocesos dentro de un equipo? y **trabajo compartido**
  - Construcción de bucle
  - Secciones/construcciones de sección (discutidas más adelante)
  - Construcción única (discutido más adelante)

# Una nota sobre las variables

- `shared(list)` y limita lo que se comparte
- `private(var)` y copia de `var` inicializada aleatoriamente en la región
- `firstprivate(var)` y `var` se inicializa con el valor que tiene antes de la región paralela
- `lastprivate(var)` y el valor del último subproceso después de almacenar la región

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

int main (void)
{
    int i = 10;

    #pragma omp parallel private(i)
    {
        printf("thread %d: i = %d\n", omp_get_thread_num(), i);
        i = 1000 + omp_get_thread_num();
    }

    printf("i = %d\n", i);

    return 0;
}
```

# La construcción del bucle

- Divide las iteraciones de bucle entre los subprocesos de un equipo.
- En el siguiente ejemplo, la variable "I" se convierte en "privada" para cada subproceso de forma predeterminada (podría hacerlo explícitamente con una cláusula "privada(I)" en otras construcciones, como se explicó)

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

# La construcción del bucle: ejemplo motivador

- Secuencial para

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

- Construcción paralela

```
#pragma omp parallel
{
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    if (id == Nthrds-1) iend = N;
    for(i=istart;i<iend;i++) { a[i] = a[i] + b[i];}
}
```

- Construcción de bucle

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

# Combinación de bucles y construcciones paralelas

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

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

- Estos códigos son funcionalmente equivalentes
- Pero, las barreras se colocan implícitamente al final de cada construcción.

# Explotando la construcción for

- Encuentre bucles intensivos en computación
- Haga que las iteraciones de bucle sean independientes para que puedan ejecutarse de forma segura en cualquier orden sin dependencias transportadas por bucles.
- Coloque la directiva apropiada, ejecute y pruebe

```
int i, j, A[MAX];  
j = 5;  
for (i=0;i< MAX; i++) {  
    j +=2;  
    A[i] = big(j);  
}
```

Quitar  
dependencia →

```
int i, A[MAX];  
#pragma omp parallel for  
for (i=0;i< MAX; i++) {  
    int j = 5 + 2*i;  
    A[i] = big(j);  
}
```

# Reducción

- Vea este ejemplo:

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

- Estamos combinando valores en una sola variable de acumulación; esta verdadera dependencia no puede eliminarse trivialmente
- Esta es una situación común y se llama **reducción**.
- Se admiten reducciones aritméticas y lógicas

# Reducción

- Para especificar una reducción, usamos **reducción (op : lista)**
  - Se realiza una copia local de cada variable de la lista y se inicializa en función de la "op" (por ejemplo, 0 para "+")
  - El compilador encuentra expresiones de reducción estándar que contienen "op" y las usa para actualizar la copia local
  - Las copias locales se reducen a un valor único y se combinan con el original valor global
  - Las variables en "lista" deben ser compartidas en la región

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



# Tarea: construcción de bucle

- Le pediré que vuelva al programa serial pi y lo paralelice con una construcción de bucle y reducciones.
- Le pediré que realice la menor cantidad posible de cambios en el programa en serie.



# Describir

- Introducción a OpenMP
- Creación de hilos
- Sincronización
- Bucles paralelos
- **Sincronice cosas maestras individuales (¡usted está aquí!)**
- Programe su para y secciones



# Sincronización: Barrera

- Cada subproceso que encuentre este pragma debe esperar hasta que todos hilos en el equipo han llegado
- El siguiente código tiene barreras **explícitas** e **implícitas** .

```
#pragma omp parallel shared (A, B, C) private(id)
{
    id=omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier
    #pragma omp for
        for(i=0;i<N;i++){C[i]=big_calc3(i,A);}
    #pragma omp for nowait
        for(i=0;i<N;i++){ B[i]=big_calc2(C, i); }
    A[id] = big_calc4(id);
}
```

# Sincronización: Maestro

- La construcción maestra denota un bloque estructurado que solo ejecuta el subproceso maestro (¡cada región tiene uno!)
- Los otros subprocesos simplemente lo omiten (no implica sincronización).
- La construcción maestra se puede usar dentro de una construcción de bucle.

```
#pragma omp parallel
{
    do_many_things();
    #pragma omp master
        { exchange_boundaries(); }
    #pragma omp barrier
        do_many_other_things();
}
```

# Sincronización: Individual

- La construcción única denota un bloque de código que es ejecutado por cualquiera pero solo un hilo
- Una barrera está implícita al final del bloque único.
- Puede eliminar la barrera con una cláusula de no *espera*

```
#pragma omp parallel
{
    do_many_things();
    #pragma omp single
    {    exchange_boundaries();    }
    do_many_other_things();
}
```

# Sincronización: Ordenada

- La región ordenada se ejecuta en el orden secuencial
- Afecta solo al siguiente estado de cuenta
- Por ejemplo, imprimir resultados para cada "I"

```
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:res)

    for (I=0;I<N;I++){
        tmp = NEAT_STUFF(I);
#pragma ordered
        res += consum(tmp);
    }
```

# Rutinas de biblioteca en tiempo de ejecución

- Modificar/comprobar el número de subprocesos
  - `omp_set_num_threads()`, `omp_get_num_threads()`,  
`omp_get_thread_num()`, `omp_get_max_threads()`
- ¿Estamos ejecutando código en una región paralela activa?
  - `omp_en_paralelo()`
- ¿Desea que el sistema varíe dinámicamente la cantidad de subprocesos de una construcción paralela a otra?
  - `omp_set_dynamic()`, `omp_get_dynamic()`
- ¿Cuántos procesadores hay en el sistema?
  - `omp_num_procs()`
- ... además de algunas rutinas menos utilizadas

# Rutinas de biblioteca en tiempo de ejecución

- Para usar una cantidad fija y conocida de subprocesos en un programa, (1) dígame a OpenMP que no desea un ajuste dinámico de subprocesos, (2) establezca la cantidad de subprocesos, luego (3) guarde el número que obtuvo (usted en realidad podría obtener menos de lo esperado)
- Tenga en cuenta que estamos protegiendo la variable "num\_threads"

```
#include <omp.h>
void main()
{  int num_threads;
    omp_set_dynamic( 0 );
    omp_set_num_threads( omp_num_procs() );
#pragma omp parallel
    {  int id=omp_get_thread_num();
#pragma omp single
        num_threads = omp_get_num_threads();
        do_lots_of_stuff(id);
    }
}
```



# Variables de entorno

- `OMP_NUM_THREADS` almacena la cantidad predeterminada global de subprocesos para usar en construcciones paralelas
- Salida del programa:
  - `OMP_NUM_THREADS` (por ejemplo, "4")
  - 8
  - 2
  - 8
- `OMP_SCHEDULE` "programa[, tamaño\_de\_pieza]"
  - calendario y p. ej., dinámico, estático
  - `chunk_size` y predeterminado "1"

```
int main()
{ #pragma omp paralelo
{
    #pragma omp sencillo
    {
        printf("%d\n", omp_get_num_threads());
    }

} omp_set_num_threads(8);
#pragma omp paralelo {

    #pragma omp sencillo
    {
        printf("%d\n", omp_get_num_threads());
    }

} #pragma omp paralelo num_threads(2) {

    #pragma omp sencillo
    {
        printf("%d\n", omp_get_num_threads());
    }

} #pragma omp paralelo
{
    #pragma omp sencillo
    {
        printf("%d\n", omp_get_num_threads());
    }
}
}
```

# Describir

- Introducción a OpenMP
- Creación de hilos
- Sincronización
- Bucles paralelos
- Sincronizar maestros individuales y otras cosas
- ¡Programe sus para y secciones aquí!)



# Agenda tu para y secciones

- la construcción de secciones le da un bloque diferente a cada subproceso en el equipo
- Es posible que algunos hilos del equipo no tengan nada que ver.
- De manera predeterminada, hay una barrera al final de las "secciones omp" y use "nowait" cláusula para apagar la barrera

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
            X_calculation();
        #pragma omp section
            y_calculation();
        #pragma omp section
            z_calculation();
    }
}
```

# La cláusula del horario

- La cláusula de programación afecta la forma en que se asignan las iteraciones de bucle hilos
  - *Schedule(static [,chunk])* y asignar bloques de iteraciones de tamaño “chunk” a cada subproceso
  - *Schedule(dynamic [,chunk])* y cada subproceso agarra las iteraciones de "fragmento" una cola hasta que se hayan manejado todas las iteraciones
  - *Schedule(guided [,chunk])* y subprocesos toman dinámicamente bloques de iteraciones El tamaño del bloque comienza grande y se reduce a un tamaño de "trozo" a medida que avanza el cálculo
  - *horario (tiempo de ejecución)* y horario y tamaño de fragmento tomados del Variable de entorno OMP\_SCHEDULE
  - Omp4j admite programación estática y dinámica

# Programación estática vs dinámica

- Ejemplo: suponga que a cada subproceso se le asignan dos (bloques de) iteraciones y estos bloques toman gradualmente menos y menos tiempo, con 8 bloques en total
  - La programación dinámica podría proporcionar un mejor **equilibrio de carga**

