Introducción a OpenMP

Exposición: Federico Flaviani
Alejandro Amaro

Basado en el Taller Dictado por el Prof. Robinson Rivas (Universidad Central de Venezuela) en Concisa-Evi 2015

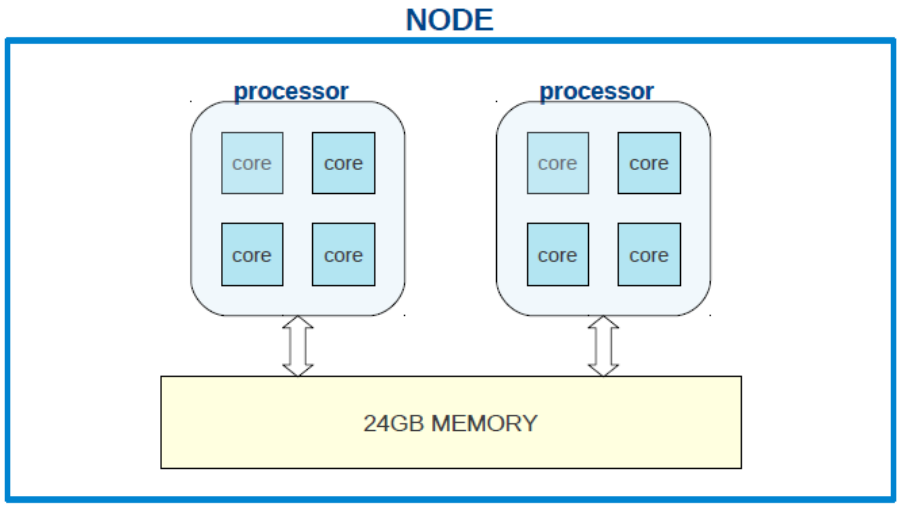
Modificado por:
Federico Flaviani
Alejandro Amaro

- Se ha modificado código a los ejemplos para que se puedan ejecutar en el ambiente del laboratorio Mys 128 y se convirtieron en ejercicios de la práctica.
- Se probaron todos los códigos luego de ser modificados ligeramente para garantizar su compilación y ejecución
- Se proporcionan todos los códigos para facilitar la realización de la práctica
- Se resolvieron los ejercicios no resueltos y se envían todos los códigos a la prof. Yudith Cardinale
- Se agregó el comando para el shell (bash) para hacer el cambio del número de hilos y se agregó un ejemplo para hacer lo mismo desde el código del programa en lenguaje C.
- Se comprobó la ejecución del código ejemplo MPI con OpenMP en el cluster de memoria distribuida del laboratorio CAR
- Se agregó un ejercicio que muestra el código serial de una multiplicación de matrices para paralelizarlo con OpenMP

- Las láminas proporcionan explicaciones exhaustivas para la práctica. No obstante lo que desea es que el participante realice los ejercicios del 1 al 9, en dos sesiones.

Entorno de uso

- Cuando se tienen arquitecturas con memoria compartida
- Cuando hay más de un core de procesamiento por núcleo



Ventajas de OpenMP

- El código es portable, ya que OpenMP encapsula las llamadas a rutinas de threading específicas de cada plataforma
- No requiere el control del programador sobre la sincronización (creación y eliminación de hilos)
- No usa el pase de mensajes, lo que minimiza la aparición de errores asociados a las transmisiones

Ventajas de OpenMP

- Es altamente escalable, el rendimiento suele mejorar cuando se ejecuta sobre un número mayor de procesadores
- Por la forma como se diseñó, el código paralelo se mantiene igual al secuencial, lo que permite ejecuciones tanto en uno como en muchos procesadores sin necesidad de recompilar

Desventajas de OpenMP

- No tiene un manejo de errores eficiente, lo que dificulta el proceso de ***debugging***
- No posee a la fecha un mecanismo de control sobre los detalles de implementación de los hilos. Por un lado esto simplifica la semántica pero impide hacer ajustes específicos de desempeño

Desventajas de OpenMP

- No tiene soporte para operaciones de bajo nivel para el manejo de memoria compartida
- El tiempo de arranque de los hilos es alto comparado con la ejecución de un hilo secuencial, por lo que en ocasiones (si el tiempo de ejecución del código es muy bajo) la implementación OpenMP resulta más ineficiente.

Lo que NO es OpenMP

OpenMP está definido como una serie de directivas de compilador ***más*** una pequeña suite de funciones de librería. OpenMP entonces:

- NO es una librería de pase de mensajes ni de código pre-elaborado
- NO está diseñado para trabajar con múltiples máquinas en un ambiente de memoria distribuida

Lo que NO es OpenMP

- NO es un lenguaje de programación, lo que facilita su aprendizaje y uso
- NO resuelve los problemas de generación de código paralelo en ambientes complejos. Por ejemplo, no resuelve la partición de datos cuando las condiciones dependen de variables aleatorias

Ejercicio 1. Ejecutar el siguiente programa en la computadora local

```
/* Programa e1.c */
#include “stdio.h”
#include “stdlib.h”
#include “omp.h”
int main()
{
#pragma omp parallel
     printf(“Hola mundo\n”);
exit (0);
}
```

Pregunta:
¿Cuáles de las instrucciones del tipo
include
se pueden eliminar en este programa sin impedir que se ejecute correctamente y por qué pueden ser eliminadas?

Para saber cómo se compila un programa, ver la lámina siguiente --->

Para compilar el programa

```
$ gcc -fopenmp e1.c -o e1
```

En general: gcc –fopenmp {source.c}
opciones
-o ejecutable

Número de hilos

- Se debe configurar la variable de entorno OMP_NUM_THREADS de acuerdo al S.O. donde se esté ejecutando. Esto debe hacerse *antes* de ejecutar el programa. No es necesario recompilar para ejecutar.

- Ejemplo, éste es el comando en el shell bash:
- **`$ export OMP_NUM_THREADS=8`**
- **`$ ./nombre-del-ejecutable`**

Ejercicio 2. Número de hilos

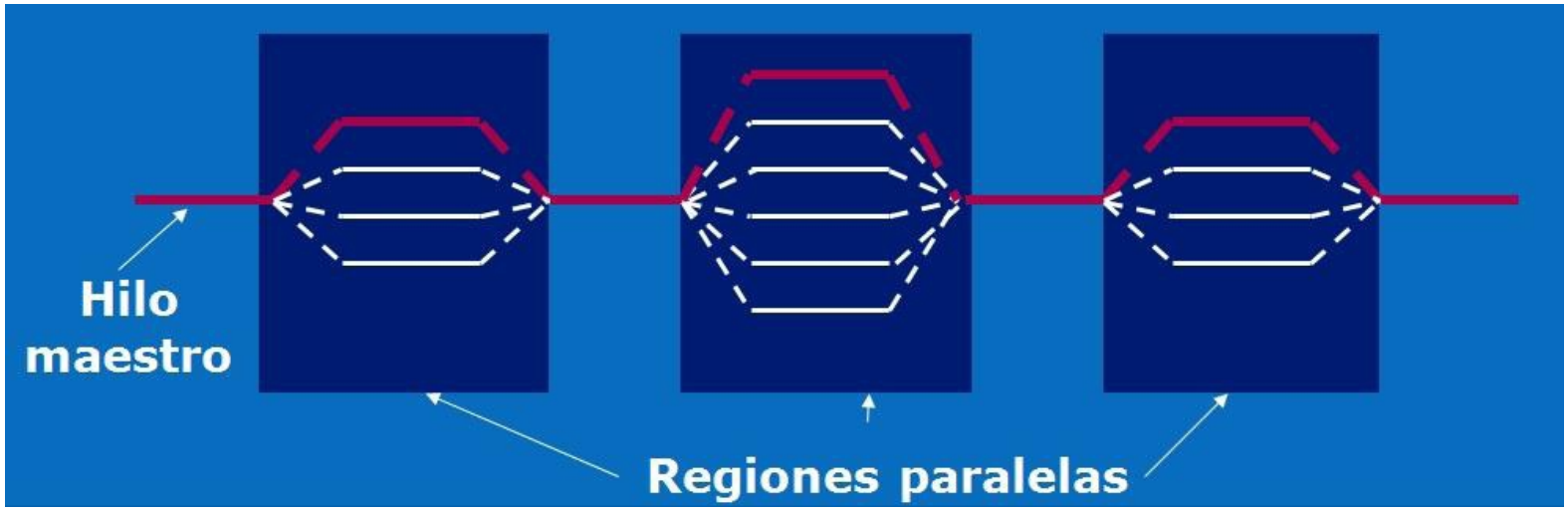
Modifique el programa del Ejercicio 1 como se muestra en el código que está más abajo (para establecer el número de hilos dentro del código del programa), compílelo y ejecútelo.
a) Compare la salida con el programa original y
b) Ejecute el programa del Ejercicio 1 con la opción de la lámina anterior.

```
/* programa e2.c */

#include "stdio.h"
#include "stdlib.h"

int main()
{
int cantidad_hilos=6;
omp_set_num_threads(cantidad_hilos);
#pragma omp parallel
     printf("Hola mundo\n");
exit (0);
}
```

Modelo fork-join

- El modelo se llama ***fork-join*** porque a partir del momento que comienza una sección paralela, a partir de un único hilo de ejecución se crean N hilos paralelos (fork o división). Estos hilos terminan su ejecución conjuntamente en un punto posterior llamado punto de reencuentro o join. En ese momento, la ejecución del programa continúa de nuevo con un solo hilo.

Modelo fork-join



Ejercicio 3. Ejecutar el siguiente programa en lacomputadora local

Las variables que hayan sido declaradas *antes* del inicio de la sección paralela, serán *compartidas* entre todos los hilos de ejecución.

```
/* Programa e3.c */
#include "stdio.h"
#include "stdlib.h"

int main() {
   int comp=0;
   # pragma omp parallel
   {
      int priv=0;
      priv++;
      comp++;
      printf("Hola mundo priv %d comp %d\n",priv,comp);
   }
   exit (0);
}
```

Ejercicio 3. (Comentarios)

```
/* Programa e3.c */
#include "stdio.h"
#include "stdlib.h"

int main() {
   int comp=0;
   # pragma omp parallel
   {
      int priv=0;
      priv++;
      comp++;
      printf("Hola mundo priv %d comp %d\n",priv,comp);
   }
   exit (0);
}
```

- **La variable comp es compartida entre los hilos durante la sección paralela, mientras que la variable priv es privada para cada hilo.**
- **Así, invariablemente la impresión de los hilos dará como resultado 1 para todas las variables priv, mientras que el resultado de las variables comp podría ser imprevisible (¿por qué?).**

Paralelismo de datos

- OpenMP implementa el paradigma de paralelismo de datos. Es adecuado principalmente en aquellos problemas donde se desea aplicar *la misma función o conjunto de instrucciones* a *diferentes partes (disjuntas) de un universo de datos.*
- Este tipo de problema es el más común cuando se trata de ejecutar instrucciones sobre elementos de arreglos, o sobre listas numeradas del tipo

```
for (i=inicio;i<fin;i++)
    x[i] = función(a[i])
```

Paralelismo de Datos

- Dado que el número de elementos a operar es conocido y normalmente estas operaciones se implementan con la instrucción for, los diseñadores de OpenMP implementaron una operación especial para las construcciones for de los lenguajes de programación (DO en Fortran).
- De esta forma el programador no tiene que cambiar la semántica de sus programas, ya que OpenMP se encargará de dividir los índices y asignarlos a los distintos hilos.

Ejemplo. Suma sobre una variable compartida (sum)

```
float producto(float* a, float* b, int N)
   { float sum = 0.0;
#pragma omp parallel for shared(sum)
      for  (int i=0; i<N; i++)
               { sum += a[i] * b[i]; }
      return sum;
    }
```

Ejercicio 4. Ejecutar el siguiente código en la computadora local:

```
/* Programa e4.c */

#include "stdio.h"
#include "stdlib.h"
#define N 4

float producto(float* a, float* b, int n) {
   float sum = 0.0;
   #pragma omp parallel for shared(sum)
      for  (int i=0; i<n; i++){
         sum += a[i] * b[i];
      }
      return sum;
}
```

Nota: una vez compilado, debe correr el mismo ejecutable varias veces y comparar los resultados de una ejecución a otra.

Continúa -->

Ejercicio 4. (Continuación)

```
int main()
{
   float total=0;
   float a[N],b[N];
   int k=0;
   for (k=0;k<N;++k) {
     a[k]=k;
     b[k]=k;
   }
   total = producto(a,b,N);
   printf("%.6f \n",total);
   int s,r,t;
   for (int i=0;i<N;i++){
      s=a[i];
      r=b[i];
      t=a[i] * b[i];
      printf("%d ",r);
      printf(" * %d ",s);
      printf(" = %d \n",t);
   }
   exit (0);
}
```

¿ Qué desventaja presenta este código?

Fin del Programa e4.c

Ejecución paralela

- Este código funciona correctamente en el caso secuencial, sin embargo al ejecutarse de forma paralela, es posible que frecuentemente los valores resultantes sean erróneos. ¿De dónde surge el error?

- De donde?
  - De donde?
    - De donde?
      - De donde por Dios?????

Concurrencia

- Al analizar el código de forma exhaustiva, se observa que en la línea

sum += a[i] * b[i];

no hay garantía que mientras uno de los hilos actualice la variable sum, otro no esté alterándola. De hecho, si N es grande y hay muchos hilos, la probabilidad de que tales interferencias ocurran es muy alta. Para evitar este tipo de cosas, una idea correcta es hacer que en el momento de la actualización, *un solo hilo pueda acceder a la variable sum.*

Sección Crítica

- OpenMP provee una cláusula que permite el acceso a bloques de código a uno solo de los hilos. Este bloque de código se denomina ***sección crítica*** y puede ser accedida solamente por uno de los hilos en cualquier momento, aunque otras partes del código pueden seguirse ejecutando de forma concurrente.

Ejemplo 4. Uso de la sección crítica

```
float producto(float* a, float* b, int N)
  { float sum = 0.0;
#pragma omp parallel for shared(sum)
    for (int i=0; i<N; i++)
        {
           #pragma omp critical
           sum += a[i] * b[i];
        }
    return sum;
  }
```

Ejercicio 5. Ejecutar el siguiente código en la computadora local:

```
/* Programa e5.c”

#include “stdio.h”
#include “stdlib.h”
#define N 4

float producto(float* a, float* b, int n) {
   float sum = 0.0;
   #pragma omp parallel for shared(sum)
      for  (int i=0; i<n; i++){
         #pragma omp critical
         sum += a[i] * b[i];
      }
      return sum;
}
```

Continúa -->

Ejercicio 5. (Continuación)

```
int main()
{
   float total=0;
   float a[N],b[N];
   int k=0;
   for (k=0;k<N;++k) {
     a[k]=k;
     b[k]=k;
   }
   total = producto(a,b,N);
   printf("%.6f \n",total);
   int s,r,t;
   for (int i=0;i<N;i++){
      s=a[i];
      r=b[i];
      t=a[i] * b[i];
      printf("%d ",r);
      printf(" * %d ",s);
      printf(" = %d \n",t);
   }
   exit (0);
}
```

¿ Qué desventaja presenta este código?

Fin del Programa e5.c

Sección crítica

¿Cuál es el problema de esta solución?

Reducciones

- Una forma de evitar esto sería tener variables privadas que hicieran acumulaciones parciales y, al final de la sección paralela, sumar todos los valores parciales en una simple variable totalizadora. OpenMP puede realizar este tipo de operaciones mediante la cláusula

reduction(operador:lista-de-variables).

reduction

- Con esta cláusula, se indica al compilador que una variable (o un conjunto de ellas) se usará de forma privada en cada uno de los hilos y será solamente al final de los cómputos que los resultados parciales obtenidos en los hilos se agruparán en un solo resultado global.
- Esta agrupación puede ser una suma (como en el caso del producto interno), o también otros operadores aritméticos y lógicos.

reduction

- Dentro de la cláusula parallel o el bloque de construcción de trabajo en paralelo:
  - Se crea una copia privada de cada variable de la lista y se inicializa de acuerdo al elemento neutro de cada operación.
  - Estas copias son actualizadas localmente por los hilos de acuerdo al cómputo que se realice.
  - Al final del bloque de construcción, las copias locales se combinan de acuerdo al operador y se actualiza la variable compartida original

reduction

Operador	Valor Inicial
+	0
*	1
-	0
^	0

Operador	Valor Inicial
&	~0
|	0
&&	1
||	0

Ejemplo 5

```
float producto(float* a, float* b, int N)
   { float sum = 0.0;
#pragma omp parallel for reduction(+:sum)
    for(int i=0; i<N; i++)
        { sum += a[i] * b[i];
        }
 return sum;
}
```

Ejercicio 6. Ejecutar el siguiente código en la computadora local:

```
/* Programa e6.c */

#include "stdio.h"
#include "stdlib.h"
#define N 4

float producto(float* a, float* b, int n) {
  float sum = 0.0;
  #pragma omp parallel for reduction(+:sum)
    for  (int i=0; i<n; i++){
      sum += a[i] * b[i];
    }
    return sum;
}
```

Nota: una vez compilado, debe correr el mismo ejecutable varias veces y comparar los resultados de una ejecución a otra.

¿Qué ventaja representa este cambio con respecto los códigos e4.c y e5.c ?

Continúa -->

Ejercicio 6. (Continuación)

```
int main()
{
   float total=0;
   float a[N],b[N];
   int k=0;
   for (k=0;k<N;++k) {
     a[k]=k;
     b[k]=k;
   }
   total = producto(a,b,N);
   printf("%.6f \n",total);
   int s,r,t;
   for (int i=0;i<N;i++){
      s=a[i];
      r=b[i];
      t=a[i] * b[i];
      printf("%d ",r);
      printf(" * %d ",s);
      printf(" = %d \n",t);
   }
   exit (0);
}
```

Fin del Programa e6.c

Balanceo de carga

- supóngase que se tienen 12 tareas cuyos tiempos de ejecución (en minutos) sean los siguientes:

Tiempo: {1, 3, 40, 30, 5, 6, 200, 100, 30, 1, 2, 150}

Si esas tareas se dividen entre dos hilos, éstos tendrían los siguientes vectores de tiempos:

Tiempo0: {1, 3, 40, 30, 5, 6} = 85 min

Tiempo1: {200, 100, 30, 1, 2, 150} = 483 min

Ejecución paralela

- Este código funciona correctamente en el caso secuencial, sin embargo al ejecutarse de forma paralela, es posible que frecuentemente los valores resultantes sean erróneos. ¿De dónde surge el error?

Balanceo de carga

- Sin embargo, dividiendo las tareas entre 3 hilos, los tiempos serían los siguientes

Tiempo0: {1, 3, 40, 30} = 74 minutos
Tiempo1: {5, 6, 200, 100} = 311 minutos
Tiempo2: {30, 1, 2, 150} = 183 minutos

Balanceo de carga

- Este problema surge cuando se desconoce el tiempo que tomará la ejecución de las tareas, y la distribución de dichos tiempos genera que unos hilos tomen cargas más pesadas (lentas) que otros. Una forma de contrarrestar este problema es haciendo agendas de trabajo (scheduling) diferentes de acuerdo a los tiempos de ejecución.

schedule

- Schedule (<tipo>[, <tamaño>])
  - schedule(static), n/t iteraciones contiguas por hebra.
  - schedule(static, K), asignación de k iteraciones contiguas.
  - schedule(dynamic), una iteración cada vez.
  - schedule(dynamic, K), k iteraciones cada vez.
  - schedule(guided, K), descenso exponencial con chunks de tamaño máximo K.
  - schedule(guided), descenso exponencial.
  - schedule(runtime), depende de la variable OMP_SHEDULE.

¿Cuándo usarla?

- **STATIC**: cuando todas las tareas a ejecutar son de complejidad similar. En este caso, no se pierde tiempo en la planificación y la ejecución será eficiente
- **DYNAMIC**: cuando las tareas tienen tiempos impredecibles, cuando la complejidad en tiempo es desconocida y además es altamente variable
- **GUIDED**: es útil cuando las cargas de tiempo son variables pero no demasiado, y se quiere minimizar el costo de planificación. Particularmente, cuando N es demasiado grande (>106)

Ejemplo

```
#pragma omp parallel for schedule
(static, 8)
    for( int i = start; i <= end; i += 2 )
      { if ( TestForPrime(i) )
          gPrimesFound++;
      }
```

En este ejemplo, cada hilo probará 8 números, y tomará siempre muestras de tamaño 8 hasta que se termine la ejecución. Sin embargo, todos los hilos tomarán la misma cantidad de muestras

Ejemplo parte 2

- Si se cambia la cláusula por dynamic:

```
 #pragma omp parallel for schedule (dynamic, 8)
for ( int i = start; i <= end; i += 2 )
      { if ( TestForPrime(i) )
          gPrimesFound++;
     }
```

Los hilos irán tomando grupos de 8. En la medida que cada hilo vaya terminando, va tomando otros 8 valores para probar. Es posible que al final de la ejecución, un hilo haya ejecutado muchas más iteraciones que otros, sin embargo los tiempos serán relativamente uniformes entre todos

Cláusula if

```
#pragma omp parallel for if ( n > 5000 )
for ( i= 0 ; i< n ; i++ )
    { x = (i+0.5)/n; area += 4.0/(1.0 +
x*x);
    } pi = area / n;
```

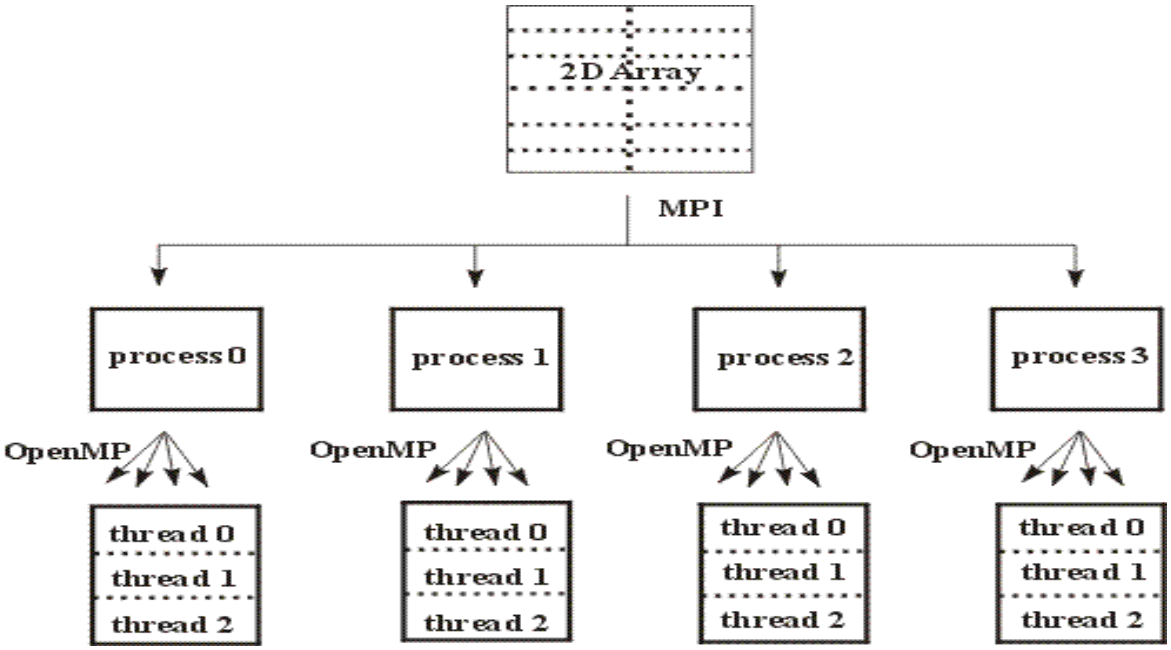
En el ejemplo, el cálculo paralelo de pi se hace únicamente si el número de iteraciones es mayor a 5000, ya que para valores menores se considera que el trabajo de crear y gestionar los hilos es mayor que la ganancia de tiempo.

Librería OpenMP

- omp_set_num_threads: Fija el número de hilos.
- omp_get_num_threads: Devuelve el número de hilos en ejecución en un momento determinado.
- omp_get_max_threads: Devuelve el número máximo de hilos que lanzará el programa en las zonas paralelas.
- omp_get_thread_num: Devuelve el número del *thread dentro del equipo (valor entre 0 y omp_get_num_threads() – 1).* Este valor puede ser utilizado para diferenciar tareas entre los procesadores.

MPI junto con OpenMP

- Las arquitecturas modernas permiten interactuar los modelos de cómputo de memoria compartida y distribuida



Ejercicio 07. MPI junto con OpenMP

Conectarse al cluster y correr el siguiente programa (e7.c). Detalles en la lámina siguiente.

```
/* Programa e07.c */

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

int main(int argc,char **argv){
   int i; int nodo,numnodos;
   int tam=32;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&nodo);
   MPI_Comm_size(MPI_COMM_WORLD,&numnodos);
   MPI_Bcast(&tam, 1, MPI_INT, 0, MPI_COMM_WORLD);
   #pragma omp parallel
      printf("Soy el hilo %d de %d hilos dentro del procesador %d de %d
procesadores\n",\
              omp_get_thread_num(),omp_get_num_threads(),nodo,numnodos);
   MPI_Finalize();
}
```

Ejercicio 07.

Para probar el programa e7.c en el cluster:

Paso 1: Compilar el programa **e7.c**

```
$ mpicc -fopenmp e7.c -o e7
```

Paso 2: Ejecutar el programa

```
$ mpirun --disable-hostname-propagation --machinefile ./maquinas.txt -np 8 ./e7
```

Ejercicio 07.

Paso 2: Ejecutar el programa

```
$ mpirun --disable-hostname-propagation --machinefile ./maquinas.txt -np 8 ./e7

Soy el hilo 1 de 2 hilos dentro del procesador 6 de 8 procesadores
Soy el hilo 1 de 2 hilos dentro del procesador 7 de 8 procesadores
Soy el hilo 0 de 2 hilos dentro del procesador 4 de 8 procesadores
Soy el hilo 0 de 2 hilos dentro del procesador 5 de 8 procesadores
Soy el hilo 1 de 2 hilos dentro del procesador 4 de 8 procesadores
Soy el hilo 0 de 2 hilos dentro del procesador 0 de 8 procesadores
Soy el hilo 1 de 2 hilos dentro del procesador 0 de 8 procesadores
Soy el hilo 0 de 2 hilos dentro del procesador 7 de 8 procesadores
Soy el hilo 0 de 2 hilos dentro del procesador 1 de 8 procesadores
Soy el hilo 1 de 2 hilos dentro del procesador 1 de 8 procesadores
Soy el hilo 0 de 2 hilos dentro del procesador 6 de 8 procesadores
Soy el hilo 0 de 2 hilos dentro del procesador 2 de 8 procesadores
Soy el hilo 0 de 2 hilos dentro del procesador 3 de 8 procesadores
Soy el hilo 1 de 2 hilos dentro del procesador 2 de 8 procesadores
Soy el hilo 1 de 2 hilos dentro del procesador 3 de 8 procesadores
Soy el hilo 1 de 2 hilos dentro del procesador 5 de 8 procesadores

$
```

Ejercicio 8. Calculo de Pi . Código serial.

Observe el siguiente código que calcula Pi:

```
/* Programa e8.c */
static long num_steps=100000;
double step, pi;
void main()
     { int i; double x, 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;
      printf("Pi = %f\n",pi);
}
```

Paralelice el código usando OpenMP y responda las siguientes preguntas:

1.¿Cuáles variables deben ser compartidas y cuáles privadas?

2.¿Debe haber secciones críticas?

3. De no haberlas, ¿puede resolverse usando otra técnica?

Ejercicio 9. El programa siguiente multiplica dos matrices de NxN

```
#include <stdio.h>
#include <stdlib.h>
#define N 5
int main ()
{
   /* DECLARACION DE VARIABLES */
   float A[N][N], B[N][N], C[N][N]; // declaracion de matrices de tamaño NxN
   int i, j, m; // indices para la multiplicacion de matrices
   /* LLENAR LAS MATRICES CON NUMEROS ALEATORIOS */
   srand ( time(NULL) );
   for(i=0;i<N;i++) {
      for(j=0;j<N;j++) {
         A[i][j]= (rand()%10);
         B[i][j]= (rand()%10);
      }
   }
   /* MULTIPLICACION DE LAS MATRICES */
   for(i=0;i<N;i++) {
      for(j=0;j<N;j++) {
         C[i][j]=0.; // colocar el valor inicial para el componente C[i][j] = 0
         for(m=0;m<N;m++) {
            C[i][j]=A[i][m]*B[m][j]+C[i][j];
         }
         printf("C: %f ",C[i][j]);
      }
      printf("  \n");
    }
   /* FINALIZAR EL PROGRAMA */
   return 0;
}
```

Utilice OpenMP para paralelizar este código