Universidad Nacional Mayor de San Marcos

Facultad de Ingeniería de Sistemas e Informática

Escuela Profesional de Ingeniería de Software

Proyecto de Programación Paralela

Docente:

Paucar Curasma, Herminio

Alumno:

Gomez Olivas, Deyvi Pedro

Lima - 2025

1. Identifique los siguientes bucles

versión secuencial

```
double res[10000];

for (i=0; i< 10000; i++)
    calculo_pesado(&res[i]);
```

versión paralela

```
double res[10000];
#pragma omp parallel for
for(i=0; i<10000; i++)
    calculo_pesado(&res[i]);
```

(a) Explique brevemente la diferencia de ejecución existente entre estos dos ejemplos de bucles.
En el caso de la versión secuencial, el for recorrerá todo el vector de manera ordenada hasta el final. Mientras que la versión paralela, similar a lo realizado en clase con MPI, cuando se empleaban límites locales para delimitar lo que debía recorrer cada proceso; la directiva #pragma omp parallel for distribuirá el for entre los hilos creados.

(b) Si un usuario está usando un procesador de 4 núcleos, ¿Qué se puede afirmar respecto al rendimiento con la adición de una sola línea de código #pragma omp parallel for en la versión paralela?
Se crearían 4 hilos, y cada uno de ellos operará con una parte del vector de manera paralela. El vector se distribuiría. Como es de 10000 elementos cada hilo debería operar con 2500 elementos.

2. Shared vs private
Explique las diferencias entre las cláusulas (atributos) shared y private con respecto al comportamiento de datos entre hilos, cuando se usan en una directiva omp parallel.
- private: Se crea una copia de la variable indicada por cada hilo, pero no se inicializa.
- shared: Todos los hilos comparten la variable. El valor inicial se mantiene y el valor puede ser modificado por cualquier. Aquí se puede dar una condición de carrera.

3. Explique cómo funcionan las funciones:
(a) ¿Cómo funcionan omp_set_num_threads() y omp_get_thread_num() en una región paralela omp?
- omp_set_num_threads: Establece el número de hilos a usar en la región paralela.
- omp_get_thread_num: Devuelve el id del hilo actual

(b) Si no se usa omp_set_num_threads(), ¿cómo especificar el número hilos en Linux?

- Como prefijo temporal, solo para esa ejecución

```
OMP_NUM_THREADS=n ./programa
```

- Estableciendo la variable de entorno para la sesión de la terminal

```
export OMP_NUM_THREADS=n
./programa
```

* Cuando no se usa ninguno de los métodos mencionados, el número de hilos se establecerá automáticamente en el número de núcleos del sistema. Se puede consultar con "nproc".

4. Región paralela - Verdadero / Falso

```
double A[10000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int th_id = omp_get_thread_num();
    calculo_pesado(th_id, A);
}
printf("Terminado");
```

(a) ¿Se señala el inicio de la ejecución de los hilos?

V La directiva #pragma omp parallel

(b) ¿Los hilos son sincronizados?

F No. Luego del fork de hilos, cada uno hace sus tareas independientemente.

(c) ¿El vector A es compartido?

V Si no se especifica, A por defecto es shared.

(d) ¿Se usaron funciones de OpenMP además de directivas?

V omp_set_num_threads y omp_get_thread_num.

5. Sobre el código siguiente se afirma Verdadero / Falso

```
#define N 10000;
int i;
#pragma omp parallel
  #pragma omp for
  for (i=0; i<10000; i++) {
     calculo();
  }
  printf("Terminado")
```

(a) ¿Las iteraciones son distribuidos entre los hilos?

V Sí, directiva #pragma omp for

(b) ¿Existe una barrera implícita de sincronización ante hilos al final del bucle?

V Sí, se espera a que todos terminen el for para continuar con el print.

(c) ¿omp for puede complementarse con schedule para especificar cómo hacer la distribución de carga del for?

V Sí, atributo schedule(static). static por defecto. static, dynamic, guided runtime.

6. Cláusula reduction

```
#include <omp.h>
#define NUM_THREADS 4
...
int i, tmp, res = 0;
#pragma omp parallel for reduction(+: res) private(tmp)
{
  for(i=0; i<15; i++) {
     tmp = Calculo();
     res += tmp;
  }
  printf("O resultado vale %d", res)
}
```

(a) ¿La variable res debe ser shared o private? Explique

res debe ser private, porque se está haciendo una operación de acumulación (reducción). La directiva reduction(+:res) ya trata a la variable res como private. La inicializa con un valor neutro de la operación a reducir y hace una acumulación local. Al final estas sumas locales son reducidas en la variable global res.

(b) Vea el código de arriba y explique qué sucede con respecto a la variable res indicada en reduction(+:res). Dé un ejemplo de cómo lo entiende para 4 hilos e i variando de 0 a 15.

La variable res es la que acumulará el valor de las operaciones, en este caso es una suma con tmp.

- #hilo 0: [0, 1, 2, 3] * el for se distribuye
  - i==0: Se calcula tmp y se suma a res
  - i==1: Se calcula tmp y se suma a res
  - i==2: Se calcula tmp y se suma a res
  - i==3: Se calcula tmp y se suma a res
- #hilo 1: [4, 5, 6, 7]
- #hilo 2: [8, 9, 10, 11]
- #hilo 3: [12, 13, 14, 15]

Debido a que res fue tratado como private cada hilo tiene un valor local de res = 4* calculo()

Y en la variable global res (resultado de la reducción)

res = 16* calculo()

7. Distribución de trabajo con secciones paralelas. Se pueden usar omp sections, cuando no se usen loops.

```
#pragma omp paralles
{
  #pragma omp sections
  {
    Calalo();
    #pragma omp section
      Calculo2();
    #pragma omp section
      Calculo3();
  }
}
```

alpha

(a) ¿Se distribuyen las secciones entre los hilos o los hilos entre las secciones?
Se distribuyen las secciones entre los hilos. Cada sección se pasa a un hilo distinto.

(b) Explique cómo funcionan la región paralela y las secciones paralelas definidas.
- La región paralela inicia en #pragma omp parallel donde se hará el fork de los hilos.
- Luego las secciones se distribuirán entre los hilos. Es decir, cada hilo ejecutará una sección.
- Hay 3 secciones
  - Sección 1: Calculo() sección implícita luego de #pragma omp sections
  - Sección 2: Calculo2()
  - Sección 3: Calculo3()

8. Sincronización
(a) Sección crítica #pragma omp critical

```
int main(){
    int total, res = 0;
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("total de hilos: %d\n", omp_get_num_threads());
        }
        #pragma omp for
        for(int i = 0; i < omp_get_num_threads(); i++){
            int id_t = omp_get_thread_num();
            #pragma omp critical
            {
                res = calc(id_t, i);
                res *= 2;
                total += res;
            }
        }
    }
    printf("suma total = %d\n", total)
}
```

(b) Atomicidad #pragma omp atomic

```
int main(){
    int total =0;
    #pragma omp parallel
    {
        #pragma omp master
        {
            printf("total de hilos: %d\n", omp_get_num_threads());
        }
        #pragma omp for
        for (int i=0; i< omp_get_num_threads(); i++){
            int id_t = omp_get_thread_num();
            #pragma omp atomic
            total += 2* calc(id_t; i);
        }
    }
    printf("suma total = %d\n", total);
}
```

(c) Barrera #pragma omp barrier (Simular comportamiento de #pragma omp single)

```
int main(){
    int total =0;
    #pragma omp parallel
    {
        #pragma omp master
        {
            printf("total de hilos: %d\n", omp_get_num_threads());
        }
        #pragma omp barrier
        printf("hilo %d // después de master\n", omp_get_thread_num());
        #pragma omp for
        for (int i=0; i< omp_get_num_threads(); i++){
            ...
        }
    } ...
}
```

alpha

10. Funciones de biblioteca para run-time para locks

¿Son directivas? No, son funciones de la API de OpenMP.

- omp_lock_t tipo de dato para definir un lock
- omp_init_lock inicializa el lock
- omp_set_lock adquiere el lock (& bloquea si ya está ocupado)
- omp_unset_lock libera el lock
- omp_test_lock intenta adquirir el lock sin bloquear
- omp_destroy_lock destruir el lock

Explique qué ocurre en este código

```
1   # include <stdio.h>
2   #include<omp.h>
3   omp_lock_t my_lock;
4   int main(){
5       omp_init_lock(&my_lock);
6
7       #pragma omp parallel num_threads(4)
8       {
9           int tid = omp_get_thread_num();
10          in i;
11
12          for(i=0; i<5; i++){
13              omp_set_lock(&my_lock);
14
15              printf("Thread %d - starting locked region\n", tid);
16              printf("Thread %d - ending locked region\n", tid);
17
18              omp_unset_lock(&my_lock);
19          }
20      }
21
22      omp_destroy_lock(&my_lock);
23      return 0;
24  }
```

- Línea 3: Se crea la variable my_lock con el tipo omp_lock_t
- Línea 5: Se inicializa el lock
- Línea 7: Se inicia la sección paralela y se establecen 4 hilos.
- Línea 9: Se obtiene el id del hilo actual
- Línea 10: Crea la variable i para el for
- Línea 12: Bloque for secuencial para cada hilo
- Línea 13: Se establece/adquiere el lock para que un solo hilo pueda acceder a la región dentro del lock a la vez.
- Línea 15 y 16: Se imprime mensaje de inicio y fin de la región bloqueada.
- Línea 18: Se libera el lock
- Línea 22: Se destruye el lock

Lo que hace el código es crear una region bloqueada. Cada hilo intenta acceder a la región bloqueada que está controlada por un lock. El lock solo permite que un hilo acceda a la vez. Pero cada hilo intentará acceder 5 veces a esa región. Por lo tanto cada hilo hará 5 print de inicio y fin de la región bloqueada.