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 en la versión paralela, similar a lo realizado en clase con MPI, cuando se usaban límites para delimitar lo que debía recorrer proceso, la directiva `#pragma omp parallel for` realizará el recorrido con el mismo comportamiento.

(b) Si un usuario está usando un procesador de cuatro 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 crearán 4 hilos, y cada uno de ellos operará con una parte del vector de manera paralela. El vector tendría segmentos. Como es de 10000 elementos:

- 10000/4 = 2500 elementos por hilo
- [0, 2500[
- [2500, 5000[
- [5000, 7500[
- [7500, 10000[

2. shared vs private

Explique la diferencia entre las cláusulas (atributos) shared y private con respecto al compartimiento de datos entre hilos, cuando se usan en una directiva `#pragma omp parallel for`.

- private:

Se crea una copia de la variable indicada por cada hilo, pero no se inicializan.

- firstprivate:

Similar a private, pero en cada iteración del hilo se inicializa la variable con el valor proporcionado en firstprivate.

- lastprivate:

Pasa el valor de la variable en la última interación a la variable global.

- shared:

Todos los hilos comparten la variable. El valor inicial se mantiene y el valor final puede ser modificado por cualquier hilo. Esta variable puede dar una condición de carrera.

3. Explique cómo funcionan la funciones

(a) ¿Cómo funcionan `omp_set_num_threads()` y `omp_get_thread_num()` en una región paralela `omp parallel`?

- omp_set_num_threads: Establece el número de hilos a usar en la región paralela.
- omp_get_thread_num: Devuelve el número de hilo actual.

* No confundir con omp_get_num_threads que devuelve el número de hilos total.

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

- Como prefijo en la ejecución del programa

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

- Estableciendo la variable de entorno OMP_NUM_THREADS

```
export OMP_NUM_THREADS=n
// luego ejecutar el programa
./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 el comando:

```
nproc
```

4. Región paralela verdadero/falso

```
OMP PARALLEL 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?

La directiva `#pragma omp parallel` indica la región paralela. Es donde se hará el fork de los hilos.

(b) ¿Los hilos son sincronizados?

No, los hilos no se sincronizan. Cuando se hace el fork de los hilos, cada hilo hace sus tareas independientemente de los demás. Se podría precisar que sí terminan sincronizados cuando se hace el join, pero durante la ejecución del código no hay directivas que garantizen que los hilos se sincronicen.

(c) ¿El vector A es compartido?

Sí, si no se especifica nada, por defecto A sería shared.

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

Sí, `omp_set_num_threads()` y `omp_get_thread_num()` son funciones de OpenMP.

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 distribuidas entre los hilos?

Sí, la directiva `#pragma omp for` distribuye las iteraciones entre los hilos creados por la directiva `#pragma omp parallel`.

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

Sí, cuando termina la directiva `#pragma omp for` se espera que todos los hilos terminen para continuar con la ejecución del código dentro de la región paralela, si existiera.

(c) ¿omp for puede complementarse con schedule para especificar cómo hacer la distribución de carga del `for (i=0 ; i < 10000 ; i++)`?

Sí, se puede usasr el atributo `schedule(static)` para que el código dentro de la directiva `#pragma omp for` se distribuya de manera específica. Existen otras opciones, como:

- static:
- dynamic:
- guided:
- runtime:

6. Claúsula reduction

```c
#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 en este caso 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 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 una suma con tmp, es la acumulación de tmp sobre res.

- Hilo 0: [0, 1, 2, 3]
  - 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.

  Al final res termina sumando 4 veces tmp.
- Hilo 1: [4, 5, 6, 7]
- Hilo 2: [8, 9, 10, 11]
- Hilo 3: [12, 13, 14, 15]

Lo mismo para los otros hilos.

Entonces:

$$\text{res} = 4 * \text{tmp} + 4 * \text{tmp} + 4 * \text{tmp} + 4 * \text{tmp} = 16 * \text{tmp}$$

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

```
#pragma omp parallel
{
  #pragma omp sections
  {
    Calculo();
    #pragma omp section
      Calcul2();
    #pragma omp section
      Calcul3();
  }
}
```

(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 funciona la región paralela y las secciones paralelas definidas.

- La región paralela `#pragma omp parallel` hará un fork de los hilos.
- Luego las secciones de distribuirán entre los hilos. Es decir cada hilo ejecutará una sección.
- Hay 3 secciones
  - Sección 1: Calculo() sección implicíta luego de `#pragma omp sections`
  - Sección 2: Calcul2()
  - Sección 3: Calcul3()

```
exercises ❯ gcc -fopenmp 7-sections.c -o ./build/out && ./build/out
Hilo: 2 || Sección 1
Hilo: 3 || Sección 2
Hilo: 7 || Sección 3
```

8. Sincronización Existen algunas instrucciones para sincronizar los accesos a la memoria compartida. Proponga un ejemplo de cada caso (a), (b), y (c).

(a) Sección crítica

`#pragma omp critical { ... }`

Apenas una thread puede ejecutar una sección crítica en un momento dado.

Código

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

int calc(int, int);

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);

  return 0;
}

int calc(int id_t, int i) {
  return id_t + i;
}
```

(b) Atomicidad

```
#pragma omp atomic
<instrucción atómica>;
```

Versión “light” de la sección crítica.

Funciona apenas para una próxima instrucción de acceso a la memoria..

Código

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

int calc(int, int);

int main() {
  int total, res = 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);

  return 0;
}

int calc(int id_t, int i) {
  return id_t + i;
}
```

(c) Barrera

```
#pragma omp barrier
```

barreras implícitas al final de las secciones paralelas! master. Se pueden provocar barreras.

Código

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

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++) {
      int id_t = omp_get_thread_num();
      #pragma omp atomic
      total += 2 * id_t;
    }
  }

  printf("suma total = %d\n", total);
}
```

10. Funciones de biblioteca para *run-time* para locks

¿Son directivas?

`omp_lock_t`, `omp_init_lock`, `omp_destroy_lock`, `omp_set_lock`, `omp_unset_lock`, `omp_test_lock`

No, son funciones de la API de OpenMP que se llaman desde el código en tiempo de ejecución.

- `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`: destruye el lock.

Explique qué ocurre en este código.

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  omp_lock_t my_lock;
5
6  int main() {
7    omp_init_lock(&my_lock);
8
9    #pragma omp parallel num_threads(4)
10    {
11      int tid = omp_get_thread_num();
12      int i;
13
14      for (i = 0; i < 5; ++i) {
```

```
15        omp_set_lock(&my_lock);
16
17        printf("Thread %d - starting locked region\n", tid);
18        printf("Thread %d - ending locked region\n", tid);
19
20        omp_unset_lock(&my_lock);
21      }
22    }
23
24    omp_destroy_lock(&my_lock);
25    return 0;
26  }
```

- Línea 4: Se crea la variable lock con el tipo `omp_lock_t`.
- Línea 7: Se inicializa el lock.
- Línea 9: Se inicia la sección paralela y se asignan 4 hilos.
- Línea 11: Se obtiene el id del hilo actual.
- Línea 12: Crea la variable i para el for.
- Línea 14: Bloque for secuencial para cada hilo.
- Línea 15: Se establece el lock para que solo un hilo pueda acceder a la región dentro del lock.
- Línea 17 y 18: Se imprime mensaje de inicio y fin de la region bloqueada.
- Línea 20: Se libera el lock.
- Línea 24: Se destruye el lock.

Lo que hace el código es crear una región 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 respectivamente.

```
exercises ❯ gcc -fopenmp 10.c -o ./build/out && ./build/out
Thread 3 - starting locked region
Thread 3 - ending locked region
Thread 3 - starting locked region
Thread 3 - ending locked region
Thread 3 - starting locked region
Thread 3 - ending locked region
Thread 3 - starting locked region
Thread 3 - ending locked region
Thread 3 - starting locked region
Thread 3 - ending locked region
Thread 1 - starting locked region
Thread 1 - ending locked region
Thread 1 - starting locked region
Thread 1 - ending locked region
Thread 1 - starting locked region
Thread 1 - ending locked region
Thread 1 - starting locked region
Thread 1 - ending locked region
Thread 2 - starting locked region
Thread 2 - ending locked region
Thread 2 - starting locked region
Thread 2 - ending locked region
Thread 0 - starting locked region
Thread 0 - ending locked region
Thread 0 - starting locked region
Thread 0 - ending locked region
Thread 0 - starting locked region
Thread 0 - ending locked region
```