Universidad Nacional de Ingeniería **2020-2**

Alumno: Walter J. Felipe Tolentino

Programación Paralela - CC332

Tarea N° 6 (Lab4)

---

1. **Source file :** ***1.cpp***

```cpp
#include <iostream>
#include <cassert> // or assert.h
#include <omp.h>
using namespace std;

int main(int argc, char **argv){
    int i=5,j=6;
    #pragma omp master
    printf("master : (i,j) = (%d,%d)\n", i, j);
    #pragma omp parallel private(i,j) num_threads(4)
    {
        i = 1;
        j = 2;
        int *ptr_i = &i, *ptr_j = &j;
        assert(*ptr_i==1 && *ptr_j==2);
        printf("thread %d : (i,j) = (%d,%d)\n",omp_get_thread_num() ,i,j);
        //#pragma omp barrier
    }
    #pragma omp master
    {
        assert(i==1 && j==2);
        printf("master : (i,j) = (%d,%d)\n", i,j);
    }
    return 0;
}
```

Out:

```
master : (i,j) = (5,6)
thread 2 : (i,j) = (1,2)
thread 0 : (i,j) = (1,2)
thread 1 : (i,j) = (1,2)
thread 3 : (i,j) = (1,2)
a.out: 1.cpp:21: int main(int, char**): Assertion `i==1 && j==2' failed.
Aborted (core dumped)
```

Como en cada hilo no aborta la ejecución por el ***assert(*ptr_i==1 && *ptr_j==2)****(line 14)*, entonces cada uno modifica sus variables cuya visibilidad es ***private(i,j)*** *(line 10)*.

Luego haciendo ***assert(i==1 && j==2)*** *(line 21)* fuera de la región en paralelo, se aborta la ejecución comprobando que modificando las variables privadas no cambia las globales

2. **Source file :** ***2.cpp***

```
1  #include <iostream>
2  #include <stdlib.h>
3  #include <omp.h>
4  #define CHUNKSIZE 25
5  #define N 100
6  using namespace std;
7
8  int main(int argc, char **argv){
9      int n[N], i, chunk=CHUNKSIZE;
10     for(i=0;i<N;i++) n[i]= 5*i ;
11
12     #pragma omp parallel shared(n, chunk) private(i) num_threads(4)
13     {
14 //        #pragma omp for ordered schedule(static, chunk) nowait
15         #pragma omp for ordered schedule(dynamic) nowait
16             for(i=0;i<N;i++){
17                 #pragma omp ordered
18                 {
19                     printf("thread %d \t n[%d] : %d \n", omp_get_thread_num() ,i,n[i]);
20                 }
21             }
22         #pragma omp barrier
23     }
24     return 0;
25 }
```

La impresión se realiza de forma ordenada usando la directiva ***#pragma omp for ordered*** *(line 15)* en el bucle for y la directiva ***#pragma omp ordered*** *(line 17)* antes de imprimir los 100 valores.

Out : using schedule(dynamic)

```
thread 2         n[0] : 0
thread 1         n[1] : 5
thread 0         n[2] : 10
thread 2         n[3] : 15
thread 3         n[4] : 20
thread 1         n[5] : 25
...
thread 2         n[95] : 475
thread 3         n[96] : 480
thread 1         n[97] : 485
thread 0         n[98] : 490
thread 2         n[99] : 495
```

Out : using schedule(static, chunk)

Descomentando la línea 14 y comentando la 15, hacemos el **schedule estático cíclico** por trozos de tamaño CHUNKSIZE=25 de forma cíclica [100/(25*4) = 1 ciclo] con periodo constante, repartiendo cada trozo a cada hilo hasta que se completen todos los trozos.

```
thread 0         n[0] : 0
thread 0         n[1] : 5
thread 0         n[2] : 10
thread 0         n[3] : 15
thread 0         n[4] : 20
thread 0         n[5] : 25
...
thread 1         n[25] : 125
thread 1         n[26] : 130
thread 1         n[27] : 135
thread 1         n[28] : 140
thread 1         n[29] : 145
...
thread 2         n[50] : 250
thread 2         n[51] : 255
thread 2         n[52] : 260
thread 2         n[53] : 265
thread 2         n[54] : 270
...
thread 3         n[75] : 375
thread 3         n[76] : 380
thread 3         n[77] : 385
thread 3         n[78] : 390
thread 3         n[79] : 395
```

3. **Source file :** ***3.cpp***

```
1  #include <iostream>
2  #include <omp.h>
3  using namespace std;
4
5  int main(int argc, char **argv){
6      int a, i;
7      #pragma omp parallel num_threads(4)
8      {
9          a = 0;
10         #pragma omp for reduction(+:a) nowait
11         for (i = 0; i < 10; i++) {
12             a += i;
13             //printf("%d, %d, %d\n",omp_get_thread_num(),i,a);
14         }
15         //#pragma omp barrier
16     }
17
18     #pragma omp master
19     cout<<"La suma es: "<<a<<endl;
20
21     return 0;
22 }
```

Out (multiple executions**)**

La suma es : 3
La suma es : 17
La suma es : 3
La suma es : 12
La suma es : 3

descomentando la línea 13 obtenemos (*se hace una tarea más por hilo, es por eso que se unen más hilos a la suma (reduction), ya que el tiempo de ejecución por hilo es mayor, esto permite una <<sincronización natural>> pero no perfecta****)*
La suma es : 33
La suma es : 32
La suma es : 32
La suma es : 33
La suma es : 32

Como se observa en realidad **NO** resulta lo que esperamos (la suma acumulada), puesto que en cada hilo ***a = 0*** *(line 9),* el iterador ***i*** nos es privado y ***a*** no es una variable compartida. Esto hace que en cada hilo se corre el riesgo que manipule data redundante y algunos hilos no se llegue a sincronizar antes de la impresión, es decir en cada ejecución la cantidad de hilos que se ejecuta antes de la impresión es distinta.

4. **Source file :** ***4.cpp***

```
1  #include <iostream>
2  #include <omp.h>
3  using namespace std;
4
5  int main(int argc, char **argv){
6      int a, i;
7      #pragma omp master
8      a = 0;
9      #pragma omp parallel shared(a) private(i) num_threads(4)
10     {
11         #pragma omp for schedule(dynamic) reduction(+:a) nowait
12         for (i = 0; i < 10; i++) {
13             a += i;
14             //printf("%d, %d, %d\n",omp_get_thread_num(),i,a);
15         }
16         #pragma omp barrier
17         #pragma omp single
18         cout<<"La suma es: "<<a<<endl;
19     }
20     return 0;
21 }
```

Out *(sin barrier - comentando line 16)*

La suma es: 28
La suma es: 15
La suma es: 28
La suma es: 28
La suma es: 13

Cuando no hay barrier, no hay sincronización y se podría imprimir un resultado erróneo antes que todos los hilos terminen su ejecución, especialmente cuando en cada hilo hay diferencias en tiempos de ejecución por ejemplo con el ***schedule(dynamic)***, pero esto se podría solucionar parcialmente sacando las líneas 17 y 18 ***(omp single)*** fuera de la región en paralelo, es decir encontrar algún hilo <<libre>> e imprimir con ese hilo, pero esto hace otro fork join, luego que termine el fork join anterior.

Pero si agregamos un barrier como es lo que hago en la línea 16 antes de la impresión del resultado, permite una sincronización en el mismo fork.

Out *(con barrier)*
La suma es : 45
La suma es : 45
La suma es : 45

Escribir un programa similar para hallar el mínimo y el máximo de un array de 100 elementos generados aleatoriamente entre 1 y 99

```cpp
#include <iostream>
#include <omp.h>
#include <time.h>
#define CHUNKSIZE 25
#define N 100
using namespace std;

int main(int argc, char **argv){
    int n[N], i, chunk=CHUNKSIZE, n_max, n_min;
    #pragma omp master
    {
        srand(time(NULL));
        for (i=0; i<N; i++) n[i] = 1 + rand()%(99 + 1 - 1);
        n_max = n[0];n_min = n[0];
    }
    #pragma omp parallel shared(n,n_max,n_min,chunk) private(i) num_threads(4)
    {
        #pragma omp for schedule(static, chunk) reduction(max:n_max) reduction(min:n_min) nowait
        for (i = 0; i < N; i++) {
            if(n[i] > n_max) n_max = n[i];
            if(n[i] < n_min) n_min = n[i];
        }
        #pragma omp barrier
        #pragma omp single
        {
            cout<<"El mín es: "<<n_min<<endl;
            cout<<"El max es: "<<n_max<<endl;
        }
    }
    return 0;
}
```

Se aplica todo lo aprendido

1. El master se crea el vector con valores aleatorios
2. Se paraleliza en 4 hilos con :
    a. visibilidad privada del iterador
    b. compartiendo n, n_max, n_min, chunk
3. Se genera un bucle for en paralelo con las cláusulas para reducir con min y max y el schedule estático cíclico de periodo constante (en este caso solo un ciclo 100/(25*4) = 1).
4. Se crea la barrera para que todos los hilos terminen su ejecución
5. Se imprime en un solo hilo usando single