



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor:

M.I Edgar Tista García

Asignatura:

Estructura de Datos y Algoritmos II

Grupo:

9

No de Práctica(s):

12 - 13

Integrante(s):

Díaz Hernández Marcos Bryan

*No. de Equipo de
cómputo empleado:*

Equipo Personal

No. de Lista o Brigada:

9

Semestre:

2021-1

Fecha de entrega:

24 de enero del 2021

Observaciones:

CALIFICACIÓN: _____

Objetivo de la practica

El estudiante conocerá y aprenderá a utilizar algunas de las directivas de OpenMP para implementar algún algoritmo paralelo.

Introducción

En este reporte se analizan los ejemplos de la practica 12 y 13, sin entrar en redundancia con lo que la misma practica documenta sobre los ejemplos, de tal forma que se puedan comprender los ejemplos y actividades de manera sencilla.

Ejercicios de la practica:

Sección 1: Ejemplos del manual - Practica 12:

Ejemplo 1 – Actividad 1:

- Análisis:

El primer ejemplo consiste en la implementación de la instrucción: `#pragma omp critical`, la cual permite que un solo hilo pueda realizar las instrucciones contenidas dentro del scope que marca la directiva, este tipo de instrucciones permiten que los datos no sean modificados por todos los hilos a la vez, y que se pueda conservar la integridad de la información.

En este caso se implementa para poder saber cuál es el número mayor de un arreglo, tal que se deba verificar si el valor con el cual está trabajando cada uno de los hilos pueda ser mayor o no, por eso mismo se reitera la condicional.

```
if(a[i]>max){
    #pragma omp critical
    {
        if(a[i]>max){
            max=a[i];
        }
    }
}
```

Doble comprobación

Con respecto a identificar cuando se debe utilizar una sentencia u otra, es necesario el saber que datos se van a verificar, para este caso, se utiliza un `pragma for`, para poder separar este ciclo en los `n` hilos y esta misma condición de separación nos da la pauta para decidir si se utiliza un `critical`, ya que si el valor requiere de una comprobación y debe existir un único resultado, lo más eficiente es utilizar la sentencia `critical`.

```
max=a[0];
#pragma omp parallel for
for(i=0;i<n;i++){
    if(a[i]>max){
        #pragma omp critical
        max=a[i];
    }
}
```

Condiciones para utilizar `critical`

- Evidencia de implementación:

```

7  int buscaMaximo(int *a);
8  int buscaMaximoA(int *a);
9  void llenarArreglo(int *a);
10
11
12 int main(){
13     printf("Paralela\n");
14     int retorno, *a;
15     a=(int *)malloc(sizeof(int)*n);
16     llenarArreglo(a);
17     retorno = buscaMaximo(a);
18     printf("Maximo: %d",retorno);
19
20     printf("\n\nSerial\n");
21     retorno = buscaMaximoA(a);
22     printf("Maximo: %d",retorno);
23     return 0;
24 }
25
26 int buscaMaximo(int *a){
27     int max,i;
28     max=a[0];
29     #pragma omp parallel for
30
31     for(i=0;i<n;i++){
32         if(a[i]>max){

```

```

Paralela
1  7  4  0  9  4  8  8  2  4
Maximo: 9

Serial
Maximo: 9
-----
Process exited after 0.2246 seconds with return value 0
Presione una tecla para continuar . . .

```

```

- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\Brain\Documents\EDA II\P12 - P13\Ejemplo A.exe
- Output Size: 131.0791015625 KiB
- Compilation Time: 4.78s

```

Ejemplo 2:

- Análisis:

En el ejemplo se introduce la instrucción `#pragma omp parallel for reduction ()`, esta cláusula permite que los valores que se le asignan a una variable común puedan ser guardados y a la vez pueda realizar una operación con los valores.

La forma en que se implementa en este caso es por medio del producto punto de matrices, de tal forma que los arreglos son repartidos en n hilos por la sentencia `pragma for`, esta sentencia lo que permite es que cada uno de los hilos modifiquen los valores de la suma. Debido a lo anterior es necesario el conservar el valor que regresa cada uno de los hilos, y por medio de la cláusula `reduction` es posible guardar en la variable de resultado cada uno de los datos que devuelven los hilos, y al final de la ejecución de los hilos, el modificar los datos de acuerdo con la operación asignada.

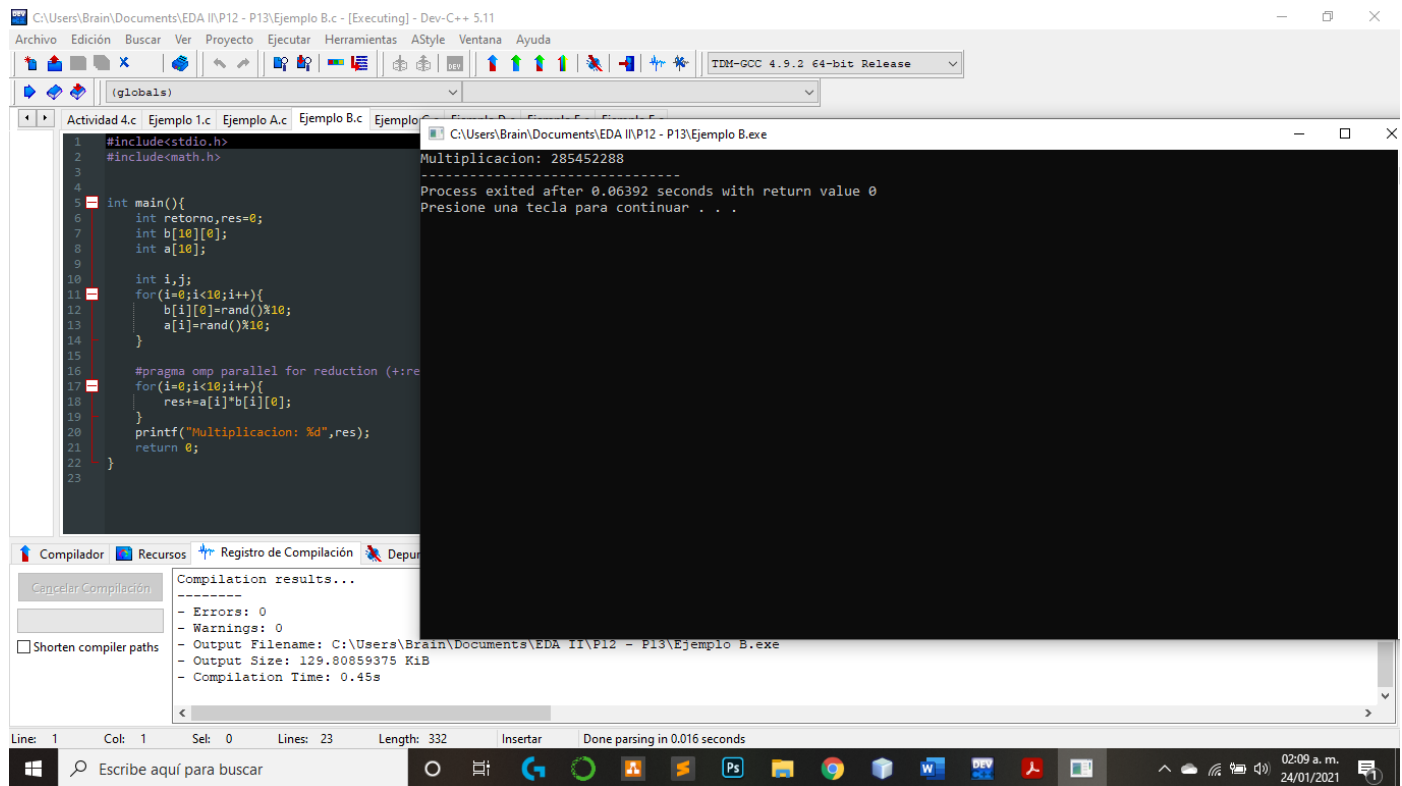
```

#pragma omp parallel for reduction (+:res)
for(i=0;i<10;i++){
    res+=a[i]*b[i][0];
}
printf("Multiplicacion: %d",res);
return 0;

```

Implementación de reduction

- Evidencia de implementación:



Ejemplo 3:

- Análisis:

En este ejemplo se introduce el constructor: `#pragma omp parallel sections`, por el cual es posible el fragmentar el código por medio de instrucciones, lo que a diferencia de la sentencia `pragma for`, este permite el segmentar secciones del código y de esta forma designar que solo un hilo modifique o ejecute esa sección del código, además de contar con un sincronizado implícito, ya que para ejecutar una segunda sección o un elemento externo al scope del constructor primero cada uno de los hilos deberá terminar su respectiva ejecución.

Una forma de utilizarlo es cuando hay secciones que son dependientes, porque permite que se ejecuten secciones específicas primero y posteriormente otras, incluso es posible el hacer secciones anidada para poder resolver dependencias, de esta forma se implementa la concurrencia.

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        v=alfa();
        #pragma omp section
        w=beta();
    }
    #pragma omp sections
    {
        #pragma omp section
        x=gama(v,w);
        #pragma omp section
        y=delta();
    }
}
```

Anidamiento de las secciones.

- Evidencia de implementación:

The screenshot shows the Dev-C++ IDE with a C program that uses OpenMP for parallel execution. The program defines three functions: `alfa()`, `beta()`, and `delta()`. The `main` function prints the initial state, then executes these three functions in parallel sections. After the parallel sections, it prints the state again. The compiler output window shows that the program compiled successfully with no errors or warnings.

```

1 #include<stdio.h>
2
3
4 int alfa();
5 int beta();
6 int delta();
7 int gama(int a, int b);
8 int epsilon(int a, int b);
9
10 int main(){
11     printf("Primer forma:\n");
12     int v,w,y,x,r;
13     #pragma omp parallel sections
14     {
15         #pragma omp section
16         v=alfa();
17         #pragma omp section
18         w=beta();
19         #pragma omp section
20         y=delta();
21     }
22     x=gama(v,w);
23     printf("\nAAA:%d\n",r=epsilon(x,y));
24
25     printf("\nSegunda forma:\n");
26 }

```

Primer forma:
Valor:1
Valor:5
AAA:6

Segunda forma:
Valor:1
Valor:5
AAA:6

Process exited after 0.06682 seconds with return value 0
Presione una tecla para continuar . . .

Compilation results...
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\Brain\Documents\EDA II\P12 - P13\Ejemplo C.exe
- Output Size: 130.9443359375 KiB
- Compilation Time: 0.47s

Ejemplo 4:

- Análisis:

El ejemplo es corto, pero de igual forma implementa una nueva forma de sincronización entre los hilos, que es la siguiente: `#pragma omp barrier`, el concepto detrás de este constructor es poner un límite de ejecución a los hilos, de tal manera que todos los hilos se deben de detener en el constructor, ya que pueden existir dependencias en la siguiente sección del bloque y es necesario que los hilos que obtienen valores que se serán utilizados posteriormente hayan obtenido los valores que son necesarios.

En este caso le añadí el número de identificador del hilo para comprobar que todos los hilos hayan ejecutado las instrucciones que les correspondían y posteriormente verificar que se ejecutan todos los hilos.

```

1 tid = omp_get_thread_num();
2 printf("Hilo:%d\n",tid);
3 alfa(A,B);
4 printf("Trabajando en A y B\n");
5
6 #pragma omp barrier
7
8 alfa(B,C);
9 printf("Trabajando en B y C\n");
10
11 return 0;

```

Implementación de barrier.

- Evidencia de implementación:

```

1 #include<stdio.h>
2
3 void alfa(int a, int b);
4
5 int main(){
6     int A,B,C,D,tid;
7     #pragma omp parallel shared(A,B,C)
8     {
9         tid = omp_get_thread_num();
10        printf("Hilo:%d\n",tid);
11        alfa(A,B);
12        printf("Trabajando en A y B\n");
13
14        #pragma omp barrier
15
16        alfa(B,C);
17        printf("Trabajando en B y C\n");
18    }
19    return 0;
20 }
21
22 void alfa(int a, int b){
23     printf("Hola: \n");
24 }
25

```

Output:

```

Hilo:0
Hola:)
Trabajando en A y B
Hilo:3
Hola:)
Trabajando en A y B
Hilo:1
Hola:)
Trabajando en A y B
Hilo:2
Hola:)
Trabajando en B y C
Hilo:)
Trabajando en B y C
Hilo:)
Trabajando en B y C
Hilo:)
Trabajando en B y C
Hilo:)
Trabajando en B y C
-----
Process exited after 0.1075 seconds with return value 0
Presione una tecla para continuar . . .

```

Compilation results...

```

- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\Brain\Documents\EDA II\P12 - P13\Ejemplo D.exe
- Output Size: 129.0125 KiB
- Compilation Time: 0.39s

```

Ejemplo 5:

- Análisis:

El siguiente constructor que se implementa es: `#pragma omp single`, el cual permite que la sección que se encuentra dentro del scope del constructor deba ser ejecutada por un único hilo, funciona de forma similar al `section`, y `critical`, debido a que solo un hilo ejecuta las instrucciones, pero la diferencia es que antes de esta puede existir una condición de carrera entre los hilos por lo que, el que logra entrar primero ejecuta las instrucciones y los demás deben de esperar que este termine para que se pueda ejecutar la siguiente sección del código.

```

actSingle();
#pragma omp single
{
    tid = omp_get_thread_num();
    printf("HiloSINGLE:%d\n",tid);
    actSingle();
}

```

Implementación del constructor single.

De igual forma coloque la impresión del hilo que ejecuta las instrucciones que se encuentran dentro del `single`, más que nada para poder verificar que se lleva a cabo la condición de carrera, y cualquier hilo puede acceder al constructor.

C:\Users\Brain\Doc	C:\Users\Brain\
Single:	Single:
Hilo:2	Hilo:1
Hola:)	Hola:)
HiloSINGLE:2	HiloSINGLE:1
Adios:(Adios:(

Condición de carrera.

- Evidencia de implementación:

```

1 #include<stdio.h>
2
3 void actTodos();
4 void actSingle();
5
6 int main(){
7     int tid;
8
9     printf("Single:\n");
10    #pragma omp parallel
11    {
12        tid = omp_get_thread_num();
13        printf("Hilo:%d\n",tid);
14        actTodos();
15        #pragma omp single
16        {
17            tid = omp_get_thread_num();
18            printf("HilosINGLE:%d\n",tid);
19            actSingle();
20        }
21        tid = omp_get_thread_num();
22        printf("Hilo:%d\n",tid);
23        actTodos();
24    }
25    printf("\nMaster:\n");
26

```

```

Single:
Hilo:0
Holo:)
HilosINGLE:0
Hilo:1
Holo:)
Hilo:3
Holo:)
Hilo:2
Holo:)
Holo:)
Hilo:0
Holo:)
Holo:)
Hilo:3
Holo:)
Hilo:1
Holo:)
Holo:)
Hilo:0
Holo:)
Holo:)
Hilo:1
Holo:)
Holo:)
Hilo:1
Holo:)
Hilo:0
Holo:)
Holo:)
Hilo:MASTER:0
Adios:(
Hilo:0
Holo:)
Holo:)
Hilo:2
Holo:)
Holo:)
Hilo:2
Holo:)
Holo:)
-----
Process exited after 0.1007 seconds with return value 0
Presione una tecla para continuar . . .

```

```

-----
Compilation results...
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\Brain\Documents\EDA II\P13\Ejemplo E.c
- Output Size: 131.0361328125 KiB
- Compilation Time: 0.41s

```

Ejemplo 6:

- Análisis:

El constructor que se introduce es: `#pragma omp master`, el cual permite la condición del constructor single, lo cual significa que solo un hilo puede modificar - implementar las instrucciones que encierra el constructor, pero en este caso los demás hilos no se detienen y continúan con las siguientes secciones del código, lo que puede hacer obsoleto el uso de este constructor si la sección que el corresponde es un dato clave para la ejecución del programa. Para poder resolver lo anterior se puede implementar de igual manera el constructor barrier para poder detener a todos los hilos en un punto.

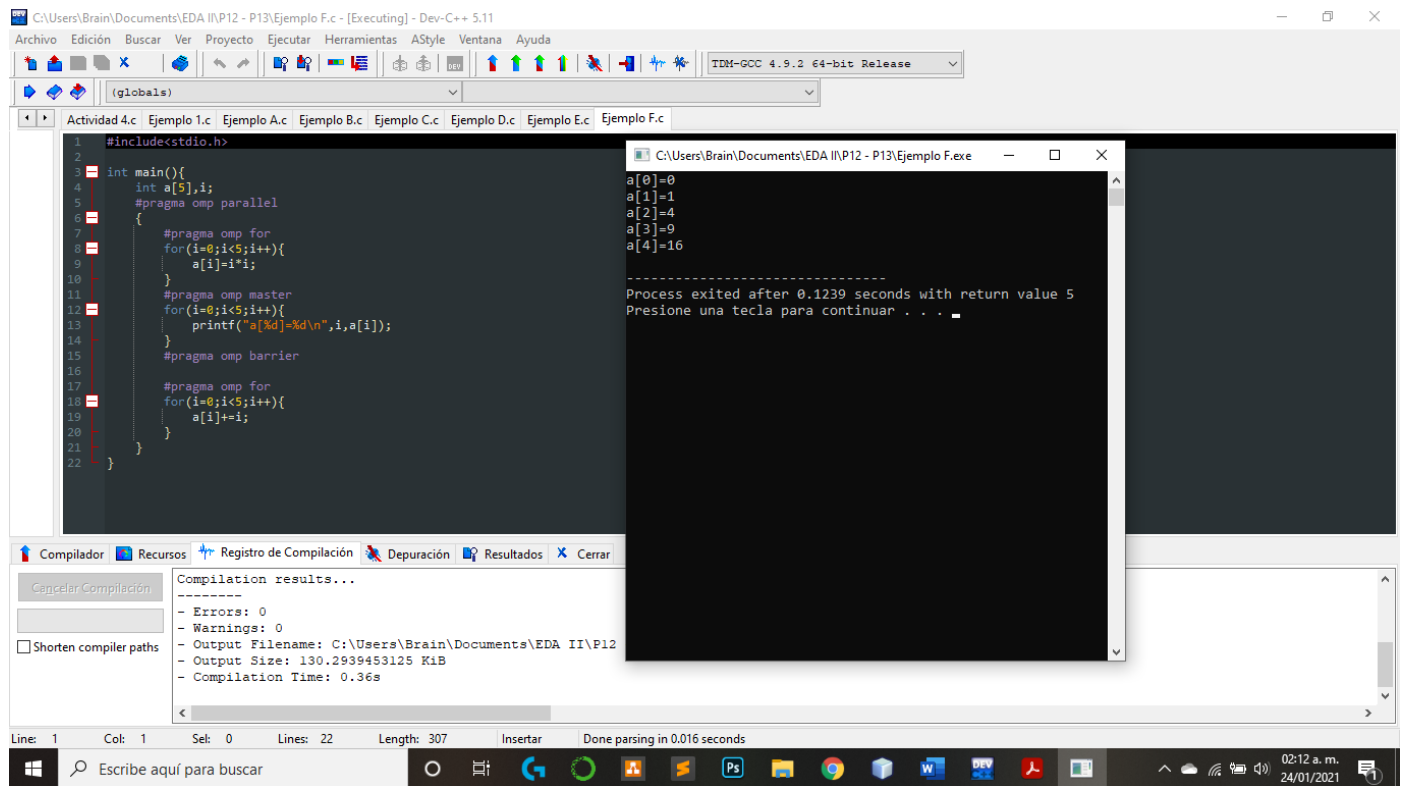
```

int main(){
    int a[5],i;
    #pragma omp parallel
    {
        #pragma omp for
        for(i=0;i<5;i++){
            a[i]=i*i;
        }
        #pragma omp master
        for(i=0;i<5;i++){
            printf("a[%d]=%d\n",i,a[i]);
        }
        #pragma omp barrier
    }
}

```

Implementación de constructor master.

- Evidencia de implementación:



- Actividad 4:
- Análisis:

En el ejercicio se pide que se implementen los dos constructores: sections y for, para poder comparar los tiempos de ejecución, para el caso de la implementación de este ejercicio, lo mejor fue el crear un menú para poder ver los tiempos de ejecución de los distintos constructores.

Para el caso de sections, se implementa por medio de: `#pragma omp parallel sections`, y con el indicador de cada sección: `#pragma omp section`, de esta forma en el programa lo que hago es colocar en cada uno de los ciclos una sección que se ejecute.

```

case 1:
{
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            empezar=omp_get_wtime();
            printf("Tiempo:%f\n",empezar);

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

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

            #pragma omp section
            {
                for(j=0;j<N;j++){
                    d[j]=e[j]+f[j];
                }
            }

            terminar=omp_get_wtime();
            printf("Tiempo:%f\n",terminar);
        }
    }
}
```

Implementación de sections

Para el caso del constructor for, realice lo mismo para poder tener las secciones en un mismo estado de la fragmentación del código, aunque las instrucciones se ejecutan 4 veces por la cantidad de hilos, que se asignan en la instrucción: `#pragma omp parallel`.

```
case 2:
#pragma omp parallel //La seccion del codigo
{
    empezar=omp_get_wtime();
    printf("TiempoEmpezo:%f\n",empezar);

    #pragma omp for
    for(i=0;i<N;i++){
        a[i]=b[i]-1;
    }

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

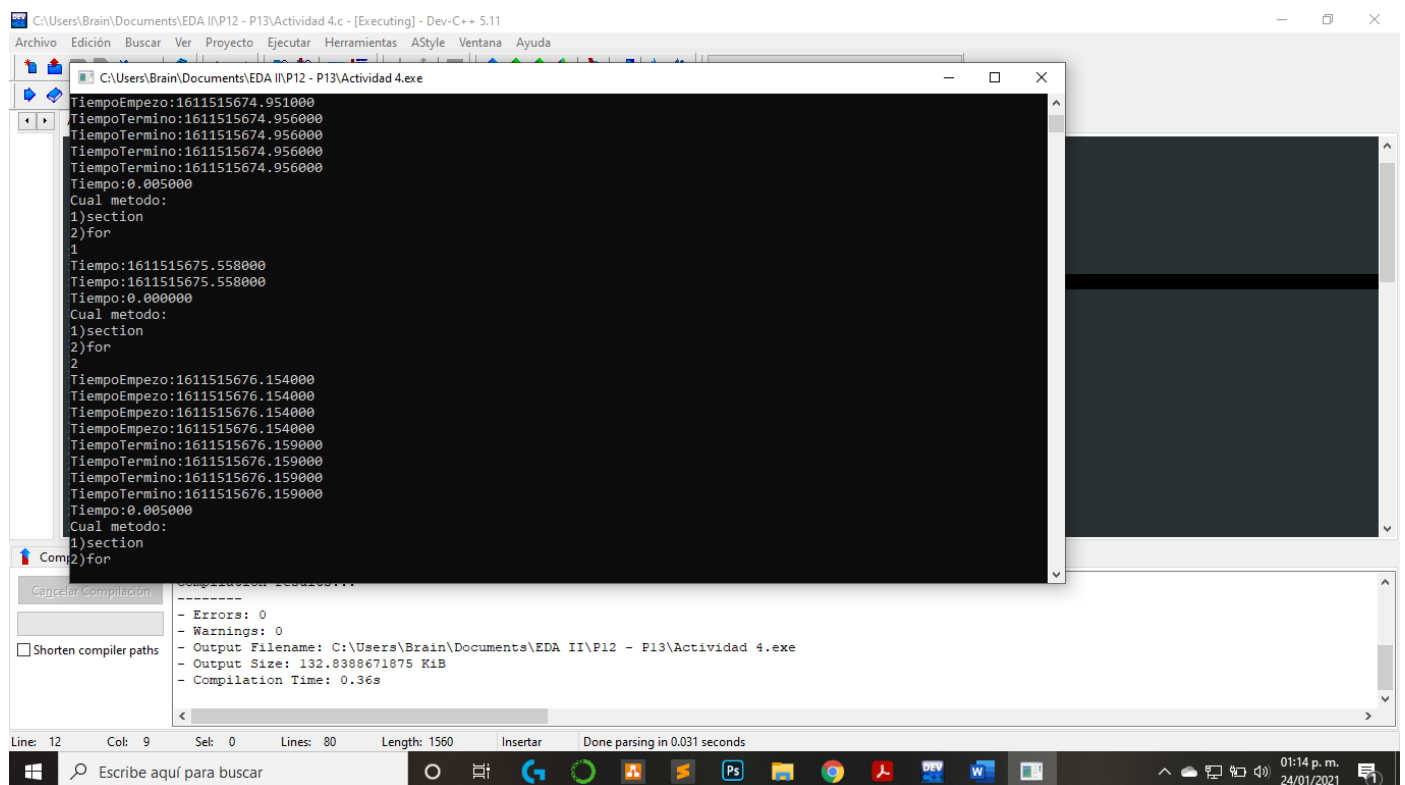
    #pragma omp for
    for(j=0;j<N;j++){
        d[j]=e[j]+f[j];
    }

    terminar=omp_get_wtime();
    printf("TiempoTermino:%f\n",terminar);
}
printf("Tiempo:%f\n",terminar-empezar);
break;
```

Implementación de for.

Resolviendo la pregunta que se plantea, el resultado muestra que el constructor for tarda más en ejecutarse, posiblemente porque las operaciones y el tamaño de los arreglos es tal que al separar la ejecución en 4 hilos, conlleva que unos se completen primero que otros, por lo que no hay una ventaja en utilizar el constructor for en contra del constructor sections.

- Evidencia de implementación:



```
Line: 12 Col: 9 Sel: 0 Lines: 80 Length: 1560 Insertar Done parsing in 0.031 seconds
```

Constructor Section y For.

- Actividad 5

En este caso se solicita el responder algunas preguntas con el ejemplo que corresponde al constructor master, por lo que no hay un código en los ejemplos que corresponda a esta actividad. Ahora resolviendo las preguntas que se plantean:

¿Qué sucede si se quita la barrera?: Los demás hilos no se detienen al ejecutarse el constructor master, por lo que ejecutan el tercer for, mientras que un solo hilo ejecuta las instrucciones del master.

```

C:\Users\Brain\Documents\EDA II\P12 - P13\Ejemplo F.exe
Despues de barrera: a[2]=6
Despues de barrera: a[3]=12
a[0]=0
a[1]=1
a[2]=6
a[3]=12
a[4]=20
Despues de barrera: a[0]=0
Despues de barrera: a[1]=2
Despues de barrera: a[4]=20

-----
Process exited after 0.0562 seconds with return value 5
Presione una tecla para continuar . . .
  
```

Sin barrera.

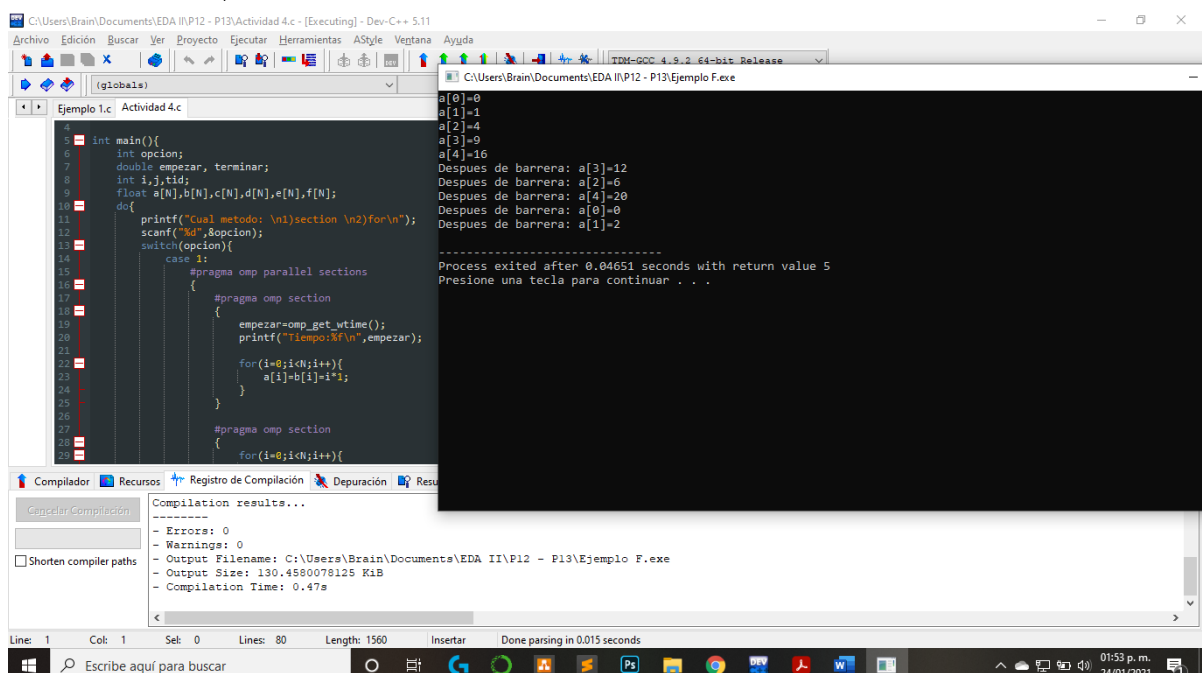
Si en lugar se utilizar el constructor master se utilizara single, ¿Qué otros cambios se tienen que hacer en el código?: Se debería colocar el constructor single el lugar del master, y eliminar el constructor barrier, debido a que el constructor single tiene su propia forma de sincronización de los hilos, por lo que tener dos barrier una explícita y otra implícita no es eficiente.

```

#pragma omp single
for(i=0;i<5;i++){
    printf("a[%d]=%d\n",i,a[i]);
}
//#pragma omp barrier
  
```

Con single.

- Evidencia de implementación:



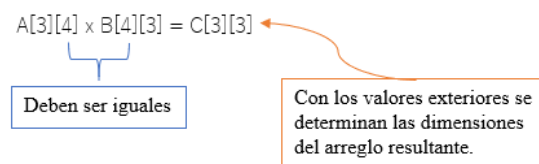
Practica 13:

Ejemplo 1:

- Análisis:

En esta sección se ven ejemplos más prácticos con respecto al uso de los constructores, para este ejemplo se muestra la implementación de la multiplicación de matrices, para esto es necesario el saber mas que nada como se multiplican las matrices ya que a partir de esta instancia o hecho, es posible saber como se deben llevar a cabo los ciclos para la multiplicación.

Normalmente se multiplica la fila por la columna, por lo que estas deben de ser iguales para poder realizarse la multiplicación:



Estando conscientes de esto, es necesario el saber los valores de A y B, ya que para obtener la matriz resultante se deberán utilizar el número de filas de A y el número de columnas de B, tal que se deba completar el arreglo resultante y por la condición de que las parte internas sean iguales este numero que es igual se deberá de tomar como el ciclo mas interno para poder realizar las operaciones para cada una de las localidades del arreglo C.

Para poder paralelizarlo se utiliza el constructor for, pero el problema con las variables al ser compartidas dentro de los ciclos más internos, se crearía un desastre entre el manejo de estas, por lo que se coloca que tanto como j y k, deben ser privadas, o también se podría colocar el numero de ciclos anidados y de esta forma repartir la carga.

```
#pragma omp parallel for private (j,k)
for(i=0;i<400;i++){
    for(j=0;j<400;j++){
        for(k=0;k<400;k++){
            c[i][j]+=a[i][k]*b[k][j];
            tid=omp_get_thread_num();
            printf("\nHilo:%d, valor C[%d][%d]=%d",tid,i,j,c[i][j]);
        }
    }
}
```

Implementación de la multiplicación de arreglos.

En este caso muestro el ejemplo con los valores de la primera actividad, pero la esencia es la misma en cualquiera de los casos que se quieran multiplicar matrices.

- Evidencia de implementación:

```

27 }
28
29
30 for(i=0;i<400;i++){
31     for(j=0;j<400;j++){
32         printf("\n A[%d]:%d",i,j,a[i][j]);
33     }
34 }
35
36 for(i=0;i<400;i++){
37     for(j=0;j<400;j++){
38         printf("\n B[%d]:%d",i,j,b[i][j]);
39     }
40 }
41
42 #pragma omp parallel for private (j,k)
43 for(i=0;i<500;i++){
44     for(j=0;j<500;j++){
45         for(k=0;k<500;k++){
46             c[i][j]+=a[i][k]*b[k][j];
47             tid=omp_get_thread_num();
48             printf("\nHilo:%d, valor C[%d][%d]=%d,tid,i,j,c[i][j]);
49         }
50     }
51 }
52

```

Compilation results...

```

- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\Brain\Documents\EDA II\P12 - P13\Ejemplo 1.exe
- Output Size: 131.80859375 KiB
- Compilation Time: 0.38s

```

Hilo:1, valor C[168][239]--397643246
Hilo:1, valor C[168][239]--397643182
Hilo:1, valor C[168][239]--397643172
Hilo:1, valor C[168][239]--397643172
Hilo:1, valor C[168][239]--397643156
Hilo:1, valor C[168][239]--397643102
Hilo:1, valor C[168][239]--397643066
Hilo:1, valor C[168][239]--397643050
Hilo:1, valor C[168][239]--397643005
Hilo:1, valor C[168][239]--397642984
Hilo:1, valor C[168][239]--397642969
Hilo:1, valor C[168][239]--397642967
Hilo:1, valor C[168][239]--397642955
Hilo:1, valor C[168][239]--397642937
Hilo:1, valor C[168][239]--397642916
Hilo:1, valor C[168][239]--397642892
Hilo:1, valor C[168][239]--397642868
Hilo:1, valor C[168][239]--397642841
Hilo:1, valor C[168][239]--397642817
Hilo:1, valor C[168][239]--397642809
Hilo:0, valor C[43][138]--438343988
Hilo:0, valor C[43][138]--438343956
Hilo:0, valor C[43][138]--438343956
Hilo:0, valor C[43][138]--438343955
Hilo:0, valor C[43][138]--438343950
Hilo:0, valor C[43][138]--438343950
Hilo:0, valor C[43][138]--438343924
Hilo:0, valor C[43][138]--438343924
Hilo:0, valor C[43][138]--438343909
Hilo:0, valor C[43][138]--438343909
Hilo:0, valor C[43][138]--438343900
Hilo:0, valor C[43][138]--438343896
Hilo:0, valor C[43][138]--438343881
Hilo:0, valor C[43][138]--438343866
Hilo:0, valor C[43][138]--438343821
Hilo:0, valor C[43][138]--438343821
Hilo:0, valor C[43][138]--438343813
Hilo:0, valor C[43][138]--438343813
Hilo:0, valor C[43][138]--438343801
Hilo:0, valor C[43][138]--438343801
Hilo:0, valor C[43][138]--438343795
Hilo:0, valor C[43][138]--438343739

Actividad 1:

- Análisis:

En este caso únicamente es la implementación secuencial y la versión paralela con arreglos de filas y columnas, pero no pude hacerlo de esa forma porque mi computadora no soportaba esa cantidad de procesos, e incluso con arreglos de 400x400, la memoria se me estaba acabando por lo que no pude ejecutar por completo cada una de las versiones, pero pude capturar como se realizaba la división entre los hilos:

Nombre	Estado	38% CPU	78% Memoria	91% Disco	0% Red
Administrador de ventanas de e...		6.7%	2,840.3 MB	0 MB/s	0 Mbps
Antimalware Service Executable		0%	117.6 MB	0 MB/s	0 Mbps
Host de servicio: SysMain		1.2%	76.1 MB	9.3 MB/s	0 Mbps
Word		0%	57.2 MB	0 MB/s	0 Mbps
IAStorDataSvc		0%	54.3 MB	0 MB/s	0 Mbps
Explorador de Windows		0.5%	34.8 MB	0 MB/s	0 Mbps
WMI Provider Host		0%	30.9 MB	0 MB/s	0 Mbps
Service		0%	29.2 MB	0 MB/s	0 Mbps
Administrador de tareas		2.3%	27.2 MB	0 MB/s	0 Mbps
Adobe Acrobat Reader DC (32 b...		0%	24.7 MB	0 MB/s	0 Mbps
Host de servicio: Servicio de dir...		0%	23.5 MB	0 MB/s	0 Mbps
Inicio		0%	21.7 MB	0 MB/s	0 Mbps
Adobe RdrCEF (32 bits)		0%	18.0 MB	0 MB/s	0 Mbps
Adobe RdrCEF (32 bits)		0%	16.1 MB	0 MB/s	0 Mbps

Mi computadora muriendo.

- Evidencia de implementación:

En la implementación por hilos, se puede ver como cada uno de los hilos tomaba una cantidad n de renglones y calculaba para cada uno los valores de las columnas correspondientes, lo que tardo demasiado, aunque estuviera en paralelo, llevaba mas de una hora y no terminaba, además que la memoria ya se me estaba acabando:

The screenshot shows the Dev-C++ IDE with a C++ program that implements matrix multiplication using OpenMP. The code is as follows:

```

27 }
28
29
30 for(i=0;i<400;i++){
31     for(j=0;j<400;j++){
32         printf("\n A[%d][%d]:%d",i,j,a[i][j]);
33     }
34 }
35
36 for(i=0;i<400;i++){
37     for(j=0;j<400;j++){
38         printf("\n B[%d][%d]:%d",i,j,b[i][j]);
39     }
40 }
41
42 #pragma omp parallel for private (j,k)
43 for(i=0;i<500;i++){
44     for(j=0;j<500;j++){
45         for(k=0;k<500;k++){
46             c[i][j]+=a[i][k]*b[k][j];
47             tid=omp_get_thread_num();
48             printf("\nHilo:%d, valor c[%d][%d]:%d",tid,i,j,c[i][j]);
49         }
50     }
51 }
52

```

The output window shows the following output:

```

Hilo:1, valor C[168][239]--397643246
Hilo:1, valor C[168][239]--397643182
Hilo:1, valor C[168][239]--397643172
Hilo:1, valor C[168][239]--397643172
Hilo:1, valor C[168][239]--397643156
Hilo:1, valor C[168][239]--397643102
Hilo:1, valor C[168][239]--397643066
Hilo:1, valor C[168][239]--397643050
Hilo:1, valor C[168][239]--397643005
Hilo:1, valor C[168][239]--397642984
Hilo:1, valor C[168][239]--397642969
Hilo:1, valor C[168][239]--397642967
Hilo:1, valor C[168][239]--397642955
Hilo:1, valor C[168][239]--397642937
Hilo:1, valor C[168][239]--397642916
Hilo:1, valor C[168][239]--397642892
Hilo:1, valor C[168][239]--397642868
Hilo:1, valor C[168][239]--397642841
Hilo:1, valor C[168][239]--397642817
Hilo:1, valor C[168][239]--397642809
Hilo:0, valor C[43][138]--438343988
Hilo:0, valor C[43][138]--438343956
Hilo:0, valor C[43][138]--438343956
Hilo:0, valor C[43][138]--438343955
Hilo:0, valor C[43][138]--438343950
Hilo:0, valor C[43][138]--438343950
Hilo:0, valor C[43][138]--438343948
Hilo:0, valor C[43][138]--438343924
Hilo:0, valor C[43][138]--438343909
Hilo:0, valor C[43][138]--438343909
Hilo:0, valor C[43][138]--438343900
Hilo:0, valor C[43][138]--438343896
Hilo:0, valor C[43][138]--438343881
Hilo:0, valor C[43][138]--438343866
Hilo:0, valor C[43][138]--438343821
Hilo:0, valor C[43][138]--438343821
Hilo:0, valor C[43][138]--438343813
Hilo:0, valor C[43][138]--438343813
Hilo:0, valor C[43][138]--438343801
Hilo:0, valor C[43][138]--438343801
Hilo:0, valor C[43][138]--438343795
Hilo:0, valor C[43][138]--438343739

```

En la version secuencial el hilo principal calculaba los valores para todas la columnas de una fila:

The screenshot shows the Dev-C++ IDE with a C++ program that implements matrix multiplication sequentially. The code is as follows:

```

37     for(j=0;j<400;j++){
38         printf("\n B[%d][%d]:%d",i,j,b[i][j]);
39     }
40 }
41
42 // #pragma omp parallel for private (j,k)
43
44
45 for(i=0;i<500;i++){
46     for(j=0;j<500;j++){
47         for(k=0;k<500;k++){
48             c[i][j]+=a[i][k]*b[k][j];
49             tid=omp_get_thread_num();
50             printf("\nHilo:%d, valor c[%d][%d]:%d",tid,i,j,c[i][j]);
51         }
52     }
53 }
54
55 for(i=0;i<500;i++){
56     for(j=0;j<500;j++){
57         printf("\n c[%d][%d]:%d",i,j,c[i][j]);
58     }
59 }
60 return 0;
61
62

```

The output window shows the following output:

```

Hilo:0, valor C[9][14]=18113
Hilo:0, valor C[9][14]=18167
Hilo:0, valor C[9][14]=18171
Hilo:0, valor C[9][14]=18171
Hilo:0, valor C[9][14]=18171
Hilo:0, valor C[9][14]=18176
Hilo:0, valor C[9][14]=18180
Hilo:0, valor C[9][14]=18180
Hilo:0, valor C[9][14]=18185
Hilo:0, valor C[9][14]=18227
Hilo:0, valor C[9][14]=18233
Hilo:0, valor C[9][14]=18233
Hilo:0, valor C[9][14]=18245
Hilo:0, valor C[9][14]=18317
Hilo:0, valor C[9][14]=18325
Hilo:0, valor C[9][14]=18325
Hilo:0, valor C[9][14]=18325
Hilo:0, valor C[9][14]=18343
Hilo:0, valor C[9][14]=18397
Hilo:0, valor C[9][14]=18421
Hilo:0, valor C[9][14]=18453
Hilo:0, valor C[9][14]=18453
Hilo:0, valor C[9][14]=18474
Hilo:0, valor C[9][14]=18482
Hilo:0, valor C[9][14]=18488
Hilo:0, valor C[9][14]=18506
Hilo:0, valor C[9][14]=18538
Hilo:0, valor C[9][14]=18538
Hilo:0, valor C[9][14]=18558
Hilo:0, valor C[9][14]=18560
Hilo:0, valor C[9][14]=18560
Hilo:0, valor C[9][14]=18560
Hilo:0, valor C[9][14]=18560
Hilo:0, valor C[9][14]=18570
Hilo:0, valor C[9][14]=18570
Hilo:0, valor C[9][14]=18570
Hilo:0, valor C[9][14]=18579
Hilo:0, valor C[9][14]=18579
Hilo:0, valor C[9][14]=18591
Hilo:0, valor C[9][14]=18612
Hilo:0, valor C[9][14]=18627
Hilo:0, valor C[9][14]=18627
Hilo:0, valor C[9][14]=18683
Hilo:0, valor C[9][14]=18704
Hilo:0, valor C[9][14]=18704
Hilo:0, valor C[9][14]=18719
Hilo:0, valor C[9][14]=18743

```

Sección 2: Programación concurrente y paralela en JAVA

- Sincronización en Java:

Ejemplo 1:

- Análisis:

En el ejemplo se crean cuatro clases, la primer clase Table, se encarga de la sincronizacion por medio del metodo printTable, ya que este permite que se ejecuten los hilos en orden de llamado del metodo, ya que si no se cuenta con la instrucción: synchronized, se genera la condicion de carrera entre los hilos y las impresiones son al azar.

Las siguientes dos clases son: Thread1-2, que son las clases que heredan de thread, y hacer la sobre escritura del metodo run, para poder ejecutar el metodo sincronizado de la clase Table. En este caso las clases tienen como atributo a un objeto de la clase table.

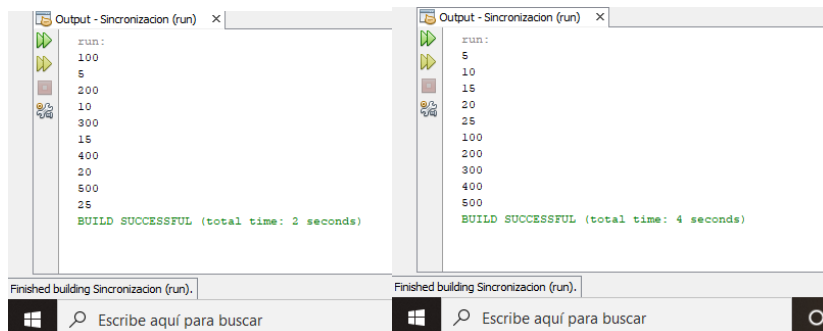
```
class MyThread2 extends Thread{
    Table t;

    MyThread2(Table t){
        this.t=t;
    }

    public void run(){
        t.printTable(100);
    }
}
```

Atributo Table.

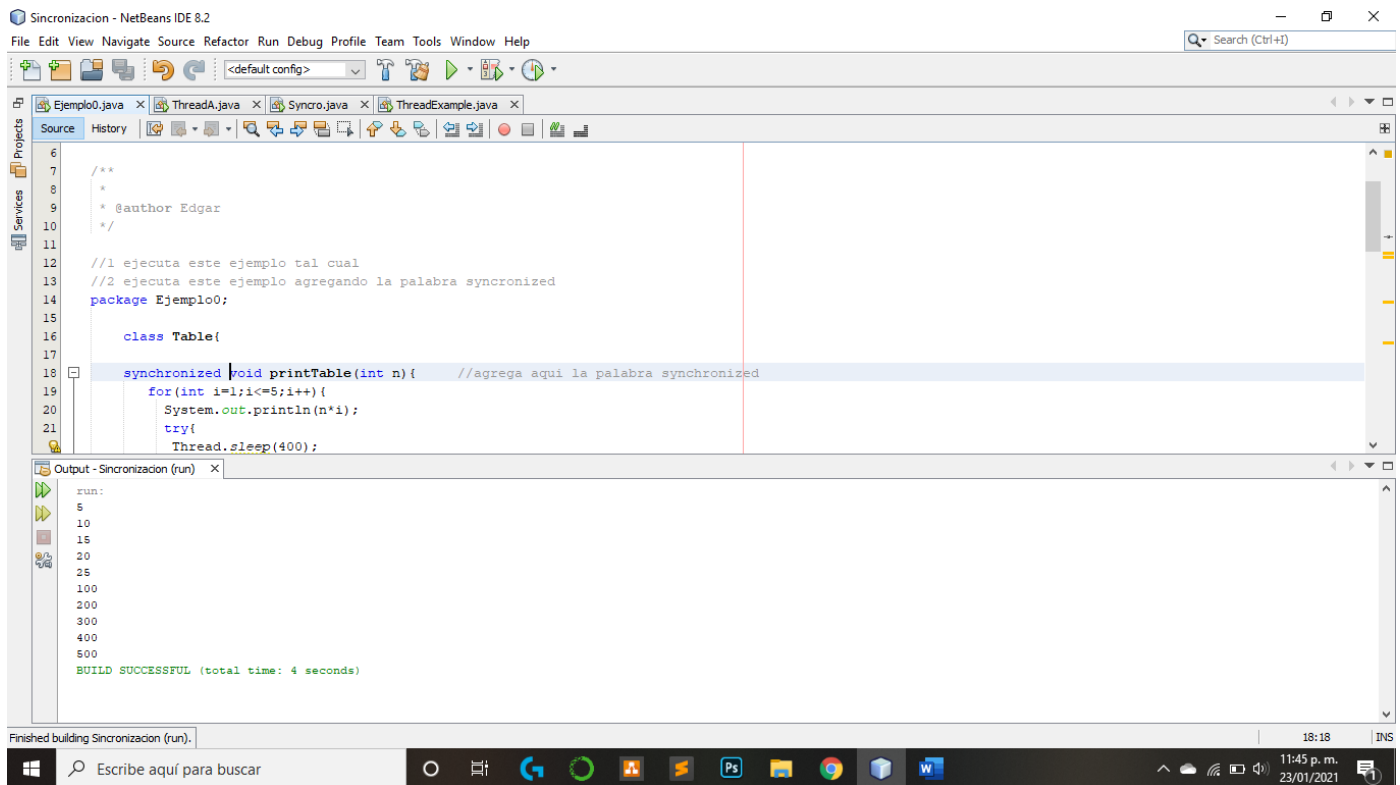
Ya en la clase principal se crean los objetos de cada una de las clases, y posteriormente se ejecuta cada uno de los hilos, lo que ejecuta el metodo run de cada una de las clases. Aquí entra la importancia de la sincronización ya que por medio de esta se hacen las impresiones de los valores de acuerdo con el orden en que se invocan los metodos, y en caso de que no estuviera sincronizados se lleva a cabo la condicion de carrera.



Sin sincronización y Con sincronización.

De igual forma si se vuelve a ejecutar uno de los hilos, el programa marca error por la sincronización debido a que ya se ejecutó una vez.

- Evidencia de implementación



Ejemplo 2:

- Análisis:

En este caso de tienen dos clases: ThreadA-B donde B hace la sincronización dentro del método run(), el cual sincroniza el objeto actual de la clase, y realiza un ciclo for con el atributo total, después del ciclo se encuentra un método importante: notify(), que manda una notificación al método wait() que espera esa confirmación que el método ha terminado su ejecución y eso permite que continúe con la impresión del valor de total.

```

110 void run() {
    synchronized(this) {
        for(int i=0; i<100 ; i++){
            total += i;
        }
        notify(); //Una vez que tern

```

Notifica el término del método run()

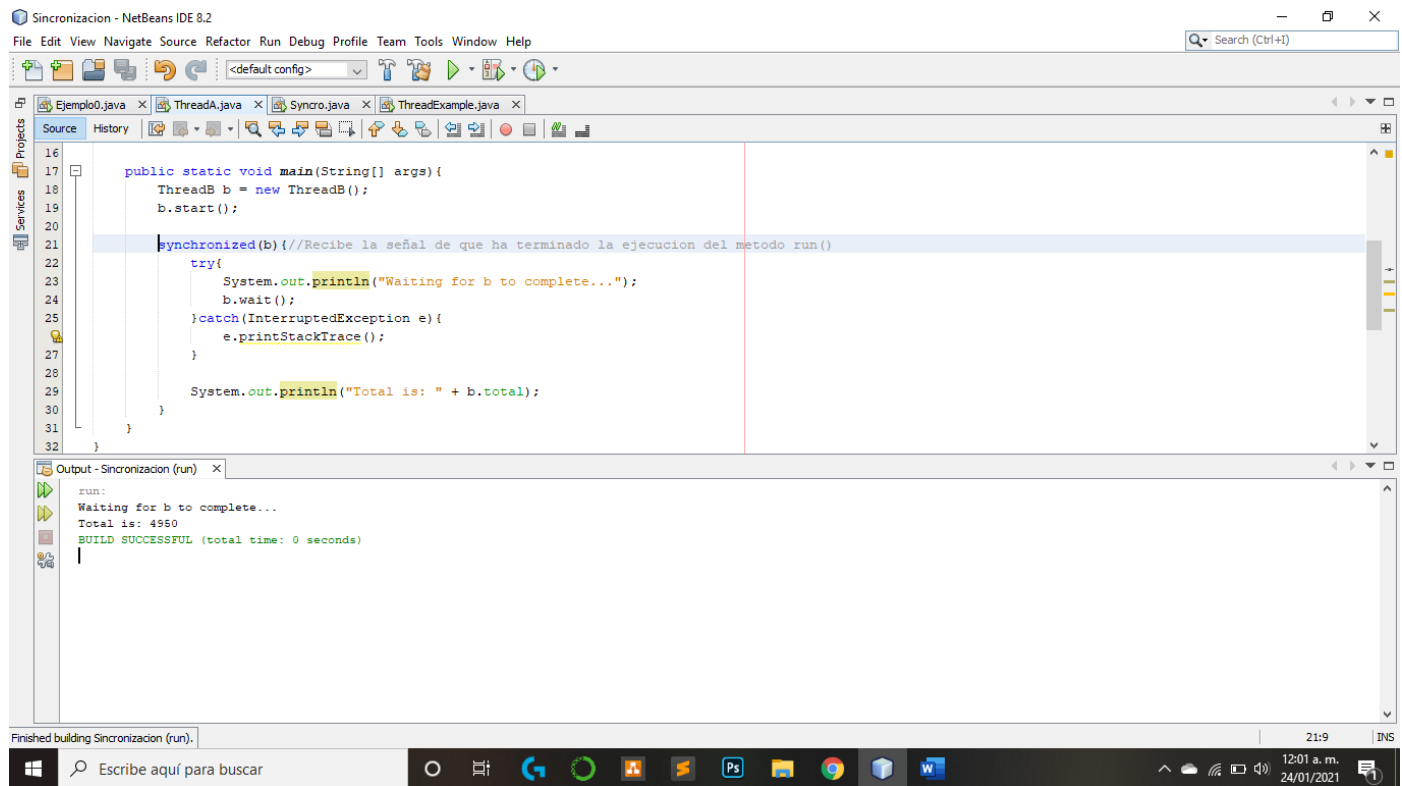
Dentro de la clase ThreadA se crea el objeto de B y se ejecuta el método run, después se sincroniza el objeto actual de forma que este método recibe la señal en el momento en que se termina la ejecución del método run(), en ese instante se ejecuta la impresión.

```

synchronized(b) {
    try{
        Sincronizado.

```

- Evidencia de implementación



Ejemplo 3:

- Análisis:

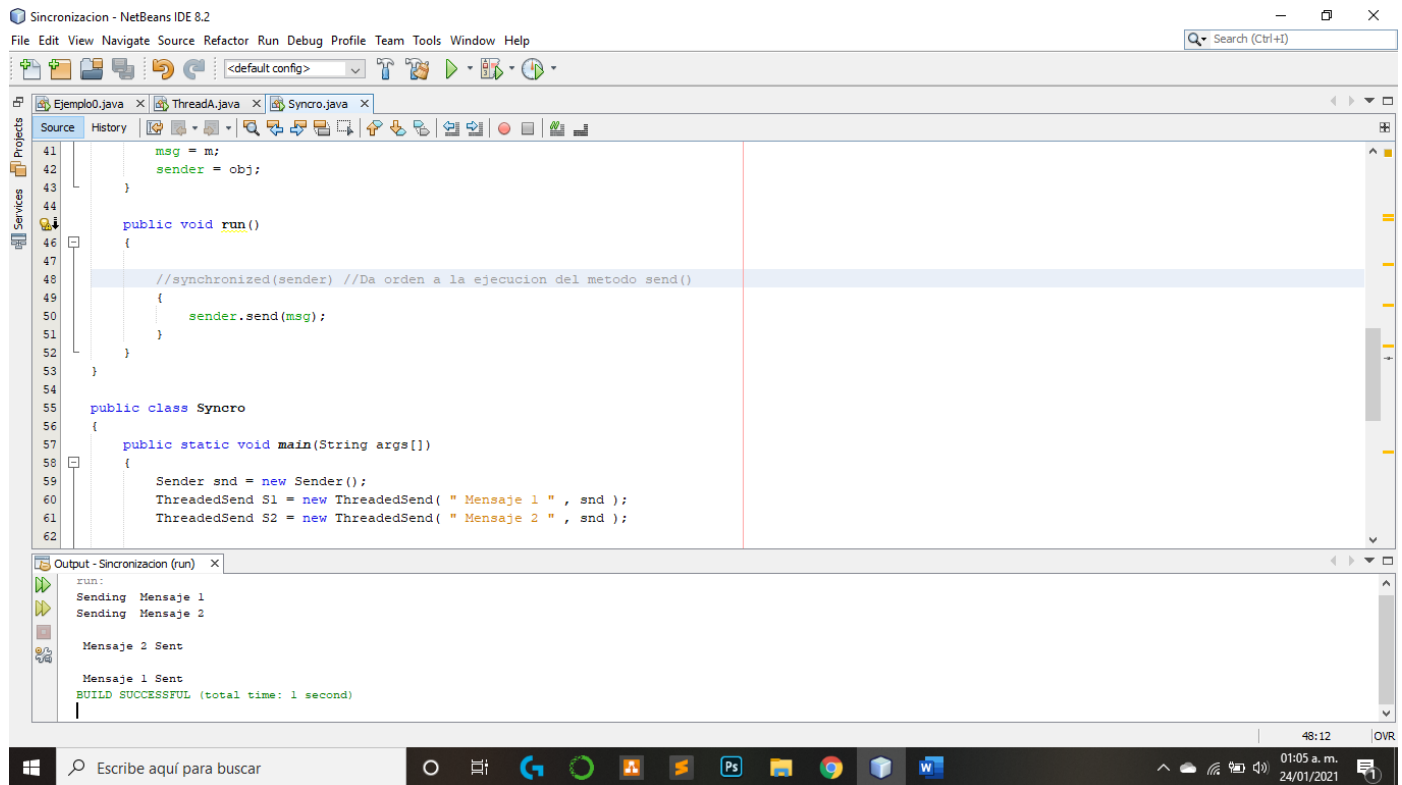
El ultimo ejemplo esta compuesto por 3 clases, la primer clase ejecuta el método send(), el cual se encarga de realizar la impresión de un mensaje, y posteriormente dormir al hilo para poder visualizar la ejecución de cada uno. La siguiente clase hereda de Thread, para poder hacer sobre escritura del método run(), y contener como atributo un objeto de la primer clase para ejecutar el método send().

El método run() lleva a cabo la sincronización de la ejecución de los hilos, tal que los hilos al ejecutar el método dentro de la clase Syncro puedan ejecutarse de acuerdo con el orden en que se ejecutaron. Además se ejecuta el método join() para cada uno de los hilos, para que mueran y el programa termine cuando estos finalicen por completo su ejecución.

run:	
Sending Mensaje 1	Sending Mensaje 1
Sending Mensaje 2	
	Mensaje 1 Sent
Mensaje 2 Sent	Sending Mensaje 2
	Mensaje 2 Sent
Mensaje 1 Sent	
BUILD SUCCESSFUL (tot	BUILD SUCCESSFUL (t

Sin sincronización y Con sincronización.

- Evidencia de implementación



Productor Consumidor:

- Análisis:

Este programa lleva a cabo una sincronización muy curiosa porque si se ejecuta en las condiciones establecidas jamás termina, y esto es debido a que no tiene un limite dentro del while(), pero fuera de ese aspecto este diseño modela el funcionamiento de un servidor, ya que a cada entrada que se recibe es necesario que se atienda, y así hasta que no haya más entradas.

Esta compuesto por dos clases, la primera en el método principal crea los objetos y realiza la implementación de los métodos producir y consumir. Además de crear los hilos sin heredad ni implementar una interfaz, lo que me parece algo nuevo y que no entiendo del todo, pero que con el método Runnable() indica que un hilo debe de ejecutar esa sección del código.

```
Thread t1 = new Thread(new Runnable()
{
    @Override
    public void run()
    {
        try
        {
            pc.produce();
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
});
```

Crear un hilo sin implementar una interfaz o heredar.

En la segunda clase se hace la sincronización de los métodos y es la que crea la condición de una ejecución infinita. Esta clase tiene como atributos una lista ligada, y un limite de 2 elementos, lo mas importante es como utiliza estos atributos en cada uno de los métodos.

En el primer método “Producir” se lleva a cabo la sincronización del objeto o del hilo que ejecuta la producción, esto significa que un hilo se ejecutara por completo y posteriormente el segundo hilo comenzara su ejecución. Dentro del método se llena la lista, con un limite de dos elementos, lo anterior no es tan explicito ya que con un solo elemento en la lista, notifica al consumidor que puede comenzar a consumir ya que existe un elemento en la lista.

Cuando el hilo del consumidor comienza a ejecutarse, vacía la lista, tal que si solo consumió un elemento, este mismo hilo notifica al consumidor para que llene la lista, esto se vuelve un ciclo infinito ya que en ningún momento las notificaciones entre los hilos se detienen. Incluso en el caso de que se llene la lista por el productor, este espera que el consumidor la vacíe y cuando esta vacía el productor la vuelve a llenar, esto por los método wait() y los notify().

```
while (list.size()==0)
    wait();

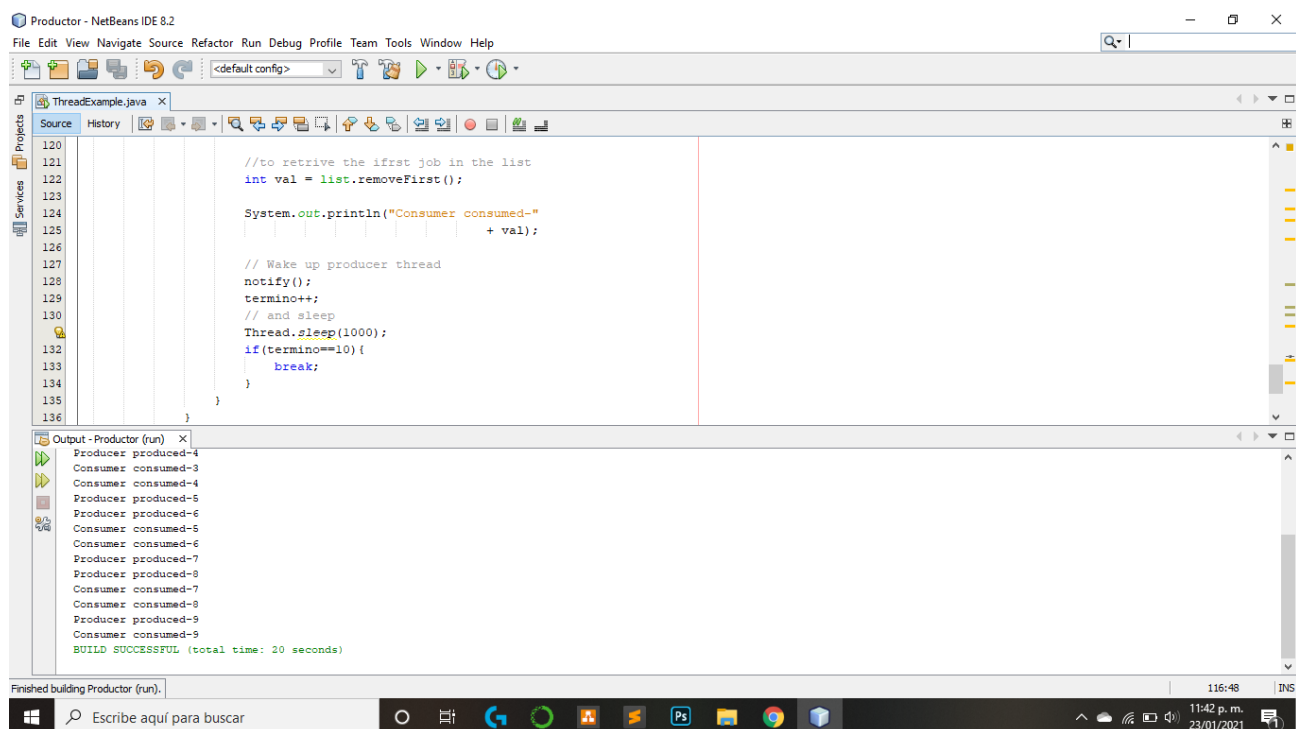
//to retrieve the ifrst job in the list
int val = list.removeFirst();

System.out.println("Consumer consumed-"
                  + val);

// Wake up producer thread
notify();
termino++;
```

Dependencia entre los hilos.

- Evidencia de implementación



Conclusiones

Al realizar cada uno de los ejemplo, y comprobar los elementos del OpenMP, me di cuenta de las diferencias que se tiene con respecto a Java, ya que no existen directivas en Java como en C, porque no encontré que los hilos se repartieran una tarea sino que entre todos resolvía una en el caso de Java, pero en C, con los constructores si es posible llevar a cabo una fragmentación de división de las tareas entre los hilos, y eso lo vuelve mas complejo ya que con varios de los constructores es posible simplificar demasiado las operaciones, como el ejemplo de `reduction()`.

Además en los ejemplos más prácticos, como el primero de la practica 13, se puede ver como realmente implementar los constructores, y en la parte de Java la sincronización me parece un método de doble filo, porque si no se realiza como debe de ser, puedes tener interacciones entre los objetos que no son deseadas, pero si se utiliza adecuadamente se pueden realizar procesos que son dependientes y eso muchas veces podrá ayudarnos a optimizar el procesamiento de la información.

Por ultimo decidí agregar instrucciones en algunos ejemplos simplemente por curiosidad de saber como se distribuyen las tareas entre los hilos, y que resultados obtiene cada uno, por todo lo anterior mencionado considero que se cumplió el objetivo de las prácticas.