

Trabajo OpenMP

Rodrigo Andre Cayro Cuadros

1. Analisis Cap 5: Productor y consumidor

-En este capítulo se analizara un problema paralelo que no es susceptible a la paralización haciendo uso de un for paralelo o de una directiva for.

a. Colas (Queue):

- La cola es una lista abstracta de un tipo de datos en el que cada elemento se inserta en la parte final de esta y aquellos elementos que se tengan que remover se hacen del frente de la cola.



b. El pase de mensajes (Message-Passing):

Otra aplicación natural seria el implementar el pase de mensajes por medio de la memoria compartida, donde cada thread tenga su propio mensaje compartido o su cola de mensajes compartidos, y cuando un thread intente enviar un mensaje a otro thread, este tendrá que ser “enqueue” en la cola de su thread destino. El thread podrá revisar el mensaje haciendo el dequeuing el mensaje al inicio de su cola.

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {  
    Send_msg();  
    Try_receive();  
}  
  
while (!Done())  
    Try_receive();
```

Este es un pequeño ejemplo del concepto del paso de mensajes en este caso se hace una impresión de números enteros, primero se envía el número de forma aleatoria a uno de los thread este lo pone en su cola y imprime el primero número de su cola y de igual manera lo dequeue de

su cola, de esta manera hasta que todos los threads hayan procesado y enviados sus mensajes respectivos, cada thread tiene un límite de mensajes específicos para mandar.

c. Envío de mensajes (Sending Messages):

Hay que notar que para poder acceder a un mensaje de la cola y colocar un nuevo mensaje en esta se genera una sección crítica con la que tenemos que lidiar, para este caso consideremos igual que en las listas enlazadas se tiene un puntero a la cola de esta para agilizar el proceso de inserción de igual manera cuando trabajamos con queue se tiene un puntero al último dato de la cola de esta manera solo cambiamos el puntero al nuevo mensaje que se ingrese, la sección crítica recae en que si tenemos que dos threads tratando de hacer estas operaciones es muy probable que se pierda último mensaje de la cola, lo que genera pérdida y problemas en esta sección de conflicto dentro del mismo programa. Este es un ejemplo de pseudocódigo de la función:

```
mesg = random();
dest = random() % thread_count;
# pragma omp critical
Enqueue(queue, dest, my_rank, mesg);
```

d. Recibiendo mensajes (Receiving Messages):

El thread propietario de la cola va a poder dequeue un mensaje de un mensaje dado por otra cola, es decir si al mismo tiempo se quiere tanto sacar un mensaje como insertar un mensaje se genera una área de conflicto esto causan directivas críticas. En este caso se hace uno de `queue_size` el cual es la diferencia entre los mensajes que se están agregando y los que se están retirando, esto permite estimar el valor de las operaciones que se necesitan y están representados por:

```
queue_size = enqueued - dequeued;
if (queue_size == 0) return;

else if (queue_size == 1)
#   pragma omp critical
    Dequeue(queue, &src, &mesg);
else
    Dequeue(queue, &src, &mesg);
Print_message(src, mesg);
```

e. Detección de final (terminal detection):

Se necesita de un criterio de parada para poder reconocer que se terminaron los mensajes dentro de una cola y este código sigue la siguiente premisa:

```

queue_size = enqueued - dequeued;
if (queue_size == 0 && done_sending == thread_count)
    return TRUE;
else
    return FALSE;

```

Donde el `done_sending` se incrementa cuando cada thread completa su propio ciclo for.

f. Función de inicio:

Para comenzar la ejecución se crea un arreglo de colas para cada thread de tal manera que sea compartida, es decir que pueda ser accedida por cualquier otro thread hacer acciones de dequeue y enqueue. Cada cola almacena los mensajes, puntero al inicio y al final, cuantos mensajes se agregan y cuantos se sacan.

Cuando se inicialicen las colas es necesario que todas terminen al mismo tiempo es por eso que se usa la directiva `#pragma omp barrier`, para que todos los threads terminen de efectuarse.

g. La directiva atómica:

Después de completar su envío de sus mensajes, cada thread incrementa el contador de `done_sending`. Claramente al incrementar `done_sending` es una sección crítica y puede ser protegida por medio de la directiva crítica. Y esto podemos hacerlo con:

```

#       pragma omp atomic
x += y++;

```

Algo que recalcar es que C solo permite directivas de asignación como lo muestra el ejemplo.

h. Análisis general de las siguientes secciones:

Como se expuso en el caso anterior también hay conflicto en tres secciones principales, dentro de estas encontramos:

```

done_sending++;
Enqueue(q-p, my_rank, msg);
Dequeue(q-p, &src, &msg);

```

Lo interesante que podemos recuperar es que nos permite darle nombre a las secciones críticas o directivas, de esta manera protegemos dos bloques críticos con diferentes nombre y que son ejecutados de forma simultanea:

```

# pragma omp critical(name)

```

Pero a pesar que se tiene un nombre para las regiones críticas no es suficiente. Otra alternativa es el uso de locks, esto nos facilita la exclusión mutua en secciones críticas de nuestros sistemas y puede ser usado como:

```

/* Executed by one thread */
Initialize the lock data structure;
. . .
/* Executed by multiple threads */
Attempt to lock or set the lock data structure;
Critical section;
Unlock or unset the lock data structure;
. . .
/* Executed by one thread */
Destroy the lock data structure;

```

También hay que tener en cuenta que es muy parecido a un mutex, en este caso si bloqueamos una sección crítica esta no puede ser accedida por los demás threads hasta que desaparezca este lock, de igual manera si hay muchos threads queriendo acceder entonces tienen que esperar a que se desbloquee la sección, después el thread que siga de igual manera bloqueara hasta que termine y así sucesivamente.

```

void omp_init_lock(omp_lock_t* lock_p /* out */);
void omp_set_lock(omp_lock_t* lock_p /* in/out */);
void omp_unset_lock(omp_lock_t* lock_p /* in/out */);
void omp_destroy_lock(omp_lock_t* lock_p /* in/out */);

```

Si vemos bien estas funciones la primera inicializa el lock, la segunda trata poner el lock si es efectiva se crea lock del thread, caso contrario el thread se bloquea, la tercera función hace alusión a liberar el lock y por último es destruir el lock.

f. Sugerencias:

No se debería de combinar diferentes tipos de exclusiones en una sola sección crítica y es peligroso de inicializar dos estructuras de exclusión mutuas.

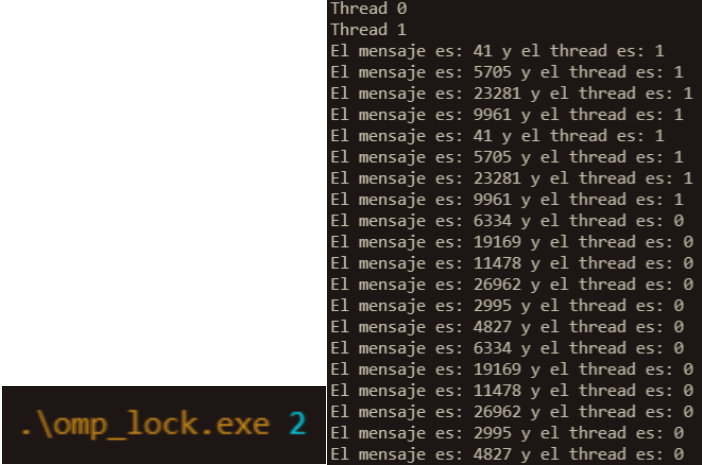
2. Implementación:

Link del repositorio: <https://github.com/rodRigocaU/ParalelTareas/tree/main/OpenMP-%23tarea3>

Resultado con 2 threads (el número de threads de igual manera se puede aumentar):

Como compilar: `gcc -g -Wall -fopenmp -o omp_lock lock.c`

Como ejecutar: `./omp_lock.exe <número de threads>`



```

Thread 0
Thread 1
El mensaje es: 41 y el thread es: 1
El mensaje es: 5705 y el thread es: 1
El mensaje es: 23281 y el thread es: 1
El mensaje es: 9961 y el thread es: 1
El mensaje es: 41 y el thread es: 1
El mensaje es: 5705 y el thread es: 1
El mensaje es: 23281 y el thread es: 1
El mensaje es: 9961 y el thread es: 1
El mensaje es: 6334 y el thread es: 0
El mensaje es: 19169 y el thread es: 0
El mensaje es: 11478 y el thread es: 0
El mensaje es: 26962 y el thread es: 0
El mensaje es: 2995 y el thread es: 0
El mensaje es: 4827 y el thread es: 0
El mensaje es: 6334 y el thread es: 0
El mensaje es: 19169 y el thread es: 0
El mensaje es: 11478 y el thread es: 0
El mensaje es: 26962 y el thread es: 0
El mensaje es: 2995 y el thread es: 0
El mensaje es: 4827 y el thread es: 0

```

.\omp_lock.exe 2