“AÑO DE LA UNIVERSALIZACIÓN DE LA SALUD”.

UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

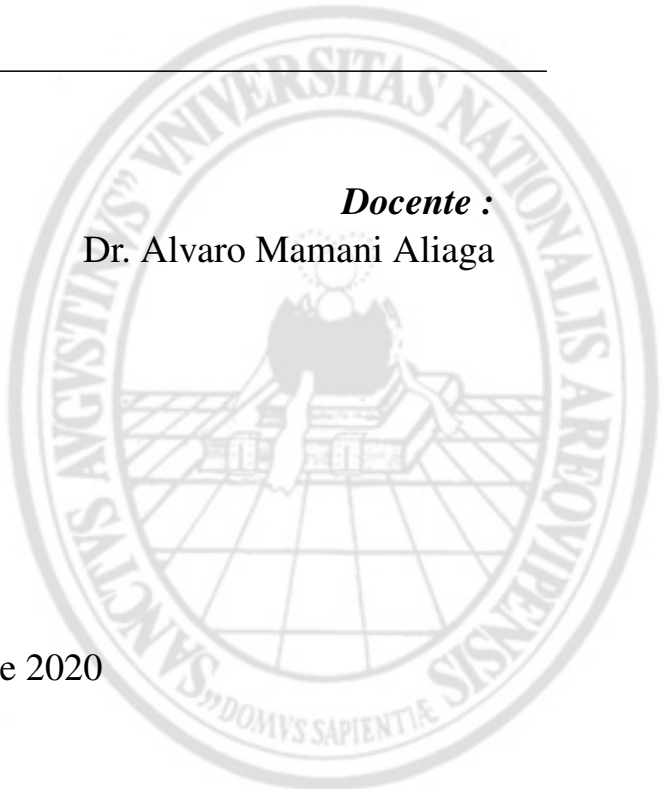
ESCUELA PROFESIONAL DE CIENCIA DE LA COMPUTACIÓN

COMPUTACIÓN PARALELA Y DISTRIBUIDA

Laboratorio 03

Alumnos:
Miguel Alexander, Herrera Cooper

Docente :
Dr. Alvaro Mamani Aliaga



7 de octubre de 2020

Índice

1. Arquitectura de la Maquina de Experimentación

1.1. Procesadores

La máquina posee 12 procesadores

```
cooper@cooper-legion-y545:~$ grep "processor" /proc/cpuinfo
processor       : 0
processor       : 1
processor       : 2
processor       : 3
processor       : 4
processor       : 5
processor       : 6
processor       : 7
processor       : 8
processor       : 9
processor       : 10
processor       : 11
```

Figura 1: Procesadores

1.2. Cores

```
cooper@cooper-legion-y545:~$ grep "core" /proc/cpuinfo
core id         : 0
cpu cores       : 6
core id         : 1
cpu cores       : 6
core id         : 2
cpu cores       : 6
core id         : 3
cpu cores       : 6
core id         : 4
cpu cores       : 6
core id         : 5
cpu cores       : 6
core id         : 0
cpu cores       : 6
core id         : 1
cpu cores       : 6
core id         : 2
cpu cores       : 6
core id         : 3
cpu cores       : 6
core id         : 4
cpu cores       : 6
core id         : 5
cpu cores       : 6
```

Figura 2: Cores

1.3. Generalidades

```
cooper@cooper-legion-y545:~$ lscpu
Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
CPU(s):                  12
On-line CPU(s) list:     0-11
Thread(s) per core:      2
Core(s) per socket:      6
Socket(s):               1
NUMA node(s):            1
Vendor ID:               GenuineIntel
CPU family:              6
Model:                   158
Model name:              Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
Stepping:                10
CPU MHz:                 2600.012
CPU max MHz:             2600.0000
CPU min MHz:             800.0000
BogoMIPS:                5199.98
Virtualisation:          VT-x
L1d cache:               32K
L1i cache:               32K
L2 cache:                256K
L3 cache:                12288K
NUMA node0 CPU(s):       0-11
```

Figura 3: Informacion General de la Máquina de Pruebas

2. Ejercicio 1

Implementar y comparar las tecnicas de sincronización Busy-Waiting y Mutex. Se debe obtener una Tabla similar a la Tabla 4.1

Resolución

Para la experimentación de las técnicas de sincronización solicitadas, hacemos uso del progrma para calcular Pi.

El experimento se ejecutó con un $n = 10^8$.

Los resultados se pueden apreciar en las siguientes tablas.

```
        // Busy - Waiting

while(flag != my_rank);
sum += my_sum;
flag = (flag+1) % numThreads;



        // Mutex
/*
pthread_mutex_lock(&mutex);
sum += my_sum;
pthread_mutex_unlock(&mutex);
*/
```

Número de Procesos : 1		
Threads	**Busy - Wait**	**Mutex**
1	0.977994	0.975927
2	0.516592	0.495327
4	0.265459	0.254917
8	0.184493	0.183762
16	0.187526	0.141605
32	0.262872	0.132002
64	0.470101	0.128593

Figura 4: Primera prueba

Número de Procesos : 2		
Threads	**Busy - Wait**	**Mutex**
1	0.980310	0.968577
2	0.509021	0.495594
4	0.280984	0.329828
8	0.270385	0.247923
16	0.372027	0.248034
32	0.457778	0.249012
64	1.099579	0.232425

Figura 5: Segunda prueba

Realizando un análisis a los resultados obtenidos se puede connotar lo siguiente :

- Si experimentamos con pocos hilos no hay mucha diferencia en el tiempo de ejecución, pero si se incrementa el número de hilos, *Busy-Waiting* demora mucho más.
- Esto se debe a que *Mutex*, cuando se desbloquea, deja que cualquier hilo que este esperando ingrese a operar, sin embargo, *Busy-Waiting* sólo permite ingresar al hilo con el id siguiente, básicamente en orden ascendente.

3. Ejercicio 2

Basado en la sección 4.7, implementar un ejemplo de productor - consumidor. Explicar porque no se debe utilizar.

Resolución

En ocasiones, bloquear o desbloquear un mutex depende de una condición ocurre en ejecución. Sin variables condicionales los programas tendrían que permanecer en espera-ocupada (hacer "polling" sobre datos) continuamente.

3.1. Consumidor :

- Bloquea (cierra) el mutex que protege a la variable global ítem.
- Espera por (item>0) que es una signal que envía el productor (así el mutex se desbloqueará automáticamente).
- Se despierta cuando el productor envíe signal (el mutex se bloquea de nuevo automáticamente), desbloquear el mutex y consumir ítem

3.2. Productor :

- Produce algo
- Bloquea (cierra) variable global mutex que protege a ítem, actualiza el ítem.
- Despierta (envía una "signal") hilos que están esperando.
- Desbloquea (abre) la variable mutex.

```
cooper@cooper-legion-y545:~/Desktop/4to/Computacion_Paralela/La
b_03$ mpirun -np 1 ./prodCon 5
Thread 0 > No hay mensaje de 4
Thread 1 > Hola del thread 1 al thread 0
Thread 2 > Hola del thread 2 al thread 1
Thread 4 > No hay mensaje de 3
Thread 3 > Hola del thread 3 al thread 2
```

Figura 6: Productor - Consumidor (5 hilos)

3.3. Uso de Mútex:

Si un programa utiliza más de un mutex, y los mutex pueden adquirirse en diferentes órdenes, el programa puede llegar a un punto muerto. Es decir, los hilos pueden bloquearse para siempre esperando adquirir uno de los mutex. Como ejemplo, supongamos que un programa tiene dos estructuras de datos compartidas -por ejemplo, dos matrices o dos listas enlazadas- cada una de las cuales tiene un mutex asociado. Supongamos además que se puede acceder a cada estructura de datos (leer o modificar) después de adquirir el mutex asociado a la estructura de datos.

4. Ejercicio 3

Implementar y explicar las diferentes formas de barreras Pthreads del libro

Resolución

pthread_barrier_t : Tipo de barrera.

pthread_barrier_init (barrier, attr, n) : Inicializa la barrera *barrier* con los atributos *attr* para que funciones con *n* hebras.

pthread_barrier_destroy(barrier) : Destruye la barrera *barrier*.

pthread_barrier_wait(barrier) : Bloquea a la hebra llamadora hasta que se ejecuten las *n* hebras.

El rendimiento usando una barrera fue mejor que la estrategia de simular una barrera con **Mutex**, ya que la implementación usa espera ocupada y el plthread_barrier ya es una función optimizada de pthread, aumenta el rendimiento. En la siguientes tablas, el número de barreras ha crecido junto con el número de subprocesos.

Número de Procesos : 1					
Threads	**150 B**	**300 B**	**450 B**	**600 B**	**750 B**
1	0.0001809	0.002791	0.0036287	0.00074707	0.000579118
2	0.002517	0.001514	0.0021750	0.00234603	0.002901077
4	0.0010669	0.002001	0.0027890	0.00350594	0.005269051
8	0.0018179	0.005044	0.0048530	0.00698685	0.005542994
16	0.0040059	0.007529	0.0113592	0.01307702	0.01688814

Figura 7: Barreras con Pthreads (Tiempo en ms)

Número de Procesos : 1					
Threads	**150 B**	**300 B**	**450 B**	**600 B**	**750 B**
1	0.000339	0.00008511	0.0001060	0.00012493	0.000116109
2	0.000180	0.00039291	0.0005087	0.00053596	0.000628248
4	0.000448	0.00117898	0.0022199	0.00109696	0.002032042
8	0.001379	0.00246000	0.0031671	0.00572681	0.005535841
16	24.14457	50.37656	74.17191	94.36107	122.8355

Figura 8: Barreras con Mutex (Tiempo en ms)

5. Repositorio

En el siguiente enlace se puede ver el código fuente del trabajo.