

**“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**

---

## **Trabajo OpenMP**

---

***Alumno:***

Miguel Alexander, Herrera Cooper

***Docente:***

Dr. Alvaro Mamani Aliaga

11 de noviembre de 2020



# Índice

<b>1. Actividad 1 - Odd - Even Sort</b>	<b>2</b>
1.1. Tablas Comparativas . . . . .	4
1.1.1. Tabla con 10000 datos . . . . .	4
1.1.2. Tabla con 100000 datos . . . . .	4
<b>2. Actividad 2 - Productor - Consumidor</b>	<b>5</b>
2.1. Queues . . . . .	5
2.2. Message-Passing . . . . .	5
2.3. Sending Messages . . . . .	6
2.4. Receiving Messages . . . . .	6
2.5. Termination Detection. . . . .	7
2.6. Ejecución . . . . .	7
<b>3. Actividad 3 - Matriz - Vector</b>	<b>8</b>
3.1. Ejecución del Problema Matriz - Vector . . . . .	8
3.2. Caché Coherence . . . . .	8
3.3. False Sharing . . . . .	9
3.4. Conclusiones . . . . .	10
<b>4. Repositorio</b>	<b>11</b>
<b>5. Referencias</b>	<b>11</b>

# 1. Actividad 1 - Odd - Even Sort

Implementar el método de ordenamiento ODD-EVEN SORT usando las dos formas presentadas en el capítulo 4, OPENMP. Deben explicar cual es la diferencia de cada forma implementada y replicar el cuadro de resultados, en su caso deben hacer los experimentos con mayor cantidad de hilos, 8, 16, 32 y mayor cantidad de datos.

## Resolución

Cuando intentamos paralelizar el ordenamiento de transposición odd-even usando la primera versión tenemos unos cuantos problemas potenciales. Primero a pesar de que cualquier iteración de una fase par o impar no depende de ninguna otra iteración en esa fase (par o impar segun corresponda), esto no se sucede para iteraciones en la fase  $p$  y fase  $p + 1$ . Necesitamos estar seguros que todos los threads han terminado la fase  $p$  antes de algún thread inicie la fase  $p + 1$ . Sin embargo parecido a la directiva parallel, la directiva parlllel or tiene un barrier implicito al final del loop, así que ninguno de los threads va a proceder a la siguiente fase, fase  $p + 1$ , hasta que todos los threads han terminado la fase actual, fase  $p$ .

```

1 void Odd_even(int a[], int n) {
2     int phase, i, tmp;
3     # ifdef DEBUG
4     char title[100];
5     # endif
6
7     for (phase = 0; phase < n; phase++) {
8         if (phase % 2 == 0)
9             # pragma omp parallel for num_threads(thread_count) \
10                default(none) shared(a, n) private(i, tmp)
11                for (i = 1; i < n; i += 2) {
12                 if (a[i-1] > a[i]) {
13                     tmp = a[i-1];
14                     a[i-1] = a[i];
15                     a[i] = tmp;
16                 }
17             }
18         else
19             # pragma omp parallel for num_threads(thread_count) \
20                default(none) shared(a, n) private(i, tmp)
21                for (i = 1; i < n-1; i += 2) {
22                 if (a[i] > a[i+1]) {
23                     tmp = a[i+1];
24                     a[i+1] = a[i];
25                     a[i] = tmp;
26                 }
27             }
28         # ifdef DEBUG
29         sprintf(title, "After phase %d", phase);
30         Print_list(a, n, title);
31         # endif
32     }
33 } /* Odd_even */

```

Un segundo problema potencial es la sobrecarga asociada con el forking y joining de los threads.

La implementación de OpenMP puede bifurcar y unir thread count threads en cada paso a travez del cuerpo del bucle exterior.

La primera fila de la siguiente tabla muestra tiempos de ejecución para 1,2,3 y 4 threads cuando la lista entrada contiene 20,000 elementos.

Table 5.2 Odd-Even Sort with Two parallel for Directives and Two for Directives (times are in seconds)				
thread_count	1	2	3	4
Two parallel for directives	0.770	0.453	0.358	0.305
Two for directives	0.732	0.376	0.294	0.239

Figura 1: Odd-Even Sort con dos directivas parallel y dos directivas for (Tiempos en segundos), resultados del libro IPP - Cap 5 - Peter Pacheco

Los tiempos obtenidos se pueden mejorar. Cada vez que ejecutamos uno de los bucles internos, usamos el mismo número de subprocesos (threads), por lo que parece ser superior para unir los hilos una vez y reutilizar el mismo equipo de subprocesos para cada ejecución de los bucles internos. OpenMP otorga directivas que nos permite realizarlo.

Podemos unir nuestro equipo de *thread count* de threads antes del bucle externo con una directiva *parallel*. Luego, en lugar de forjar un nuevo equipo de subprocesos con cada ejecución de uno de los bucles internos, usamos una directiva *for*, que le dice a OpenMP que paralice el bucle for con el equipo de subprocesos existente.

La directiva *for*, a diferencia de la directiva *parallel for*, no cede ningún hilo. Utiliza cualquier hilo que ya haya sido bifurcado en el bloque parallel que lo contiene.

Hay una barrera implícita al final del bucle. Los resultados del código, la lista final, serán los mismos que los resultados obtenidos del código original paralelizado.

```

1 void Odd_even(int a[], int n) {
2     int phase, i, tmp;
3
4     # pragma omp parallel num_threads(thread_count) \
5         default(none) shared(a, n) private(i, tmp, phase)
6     for (phase = 0; phase < n; phase++) {
7         if (phase % 2 == 0)
8             # pragma omp for
9             for (i = 1; i < n; i += 2) {
10                 if (a[i-1] > a[i]) {
11                     tmp = a[i-1];
12                     a[i-1] = a[i];
13                     a[i] = tmp;
14                 }
15             }
16         else
17             # pragma omp for
18             for (i = 1; i < n-1; i += 2) {
19                 if (a[i] > a[i+1]) {

```

```

20         tmp = a[i+1];
21         a[i+1] = a[i];
22         a[i] = tmp;
23     }
24 }
25 }
26 } /* Odd_even */

```

Los tiempos de ejecución para esta segunda versión de **Odd-Even Sort** se encuentran en la segunda fila de la Tabla de la Figura 1 Cuando utilizamos dos o más subprocesos, la versión que usa dos directivas **for** es al menos un 17 % más rápida que la versión que usa dos directivas **parallel for**, así que para este sistema el leve esfuerzo involucrado en hacer el cambio es muy valorado.

## 1.1. Tablas Comparativas

El tiempo está en segundos

### 1.1.1. Tabla con 10000 datos

n Threads	4	8	16	32
Directiva paralela para cada bucle interno	0.09285310	0.07411369	0.7379095	1.570934e+00
Directiva <b>"for"</b> en bucle interno	0.07442918	0.05813055	0.3989776	0.8376027

Figura 2: Tabla1. Comparación de tiempos(s) con 10000 datos

### 1.1.2. Tabla con 100000 datos

n Threads	4	8	16	32
Directiva paralela para cada bucle interno	9.127694e+00	6.335751e+00	1.305501e+01	2.163739e+01
Directiva <b>"for"</b> en bucle interno	8.776485e+00	5.980315e+00	1.031274e+01	1.655159e+01

Figura 3: Tabla1. Comparación de tiempos(s) con 100000 datos.

## 2. Actividad 2 - Productor - Consumidor

Implementar el problema productor.consumidor usando las formas de sincronización presentadas en la sección 5.8. Explicar el comportamiento de cada una de las formas.

### Resolución

### 2.1. Queues

Se debe tener en cuenta que una cola es un tipo de datos abstracto de lista en el que se insertan nuevos elementos en la "parte posterior" (rear) de la cola y los elementos se eliminan del "frente" (front) de la cola.

### 2.2. Message-Passing

Cada thread podría tener una cola de mensajes compartida, y cuando un thread quisiera "enviar un mensaje" a otro thread, podría poner el mensaje en cola en la cola del thread de destino.

Un thread podría recibir un mensaje retirando el mensaje al principio de su cola de mensajes.

Siguiendo el proceso que se muestra a continuación.

1. Después de crear el mensaje, el thread coloca el mensaje en la cola de mensajes correspondiente.
2. Después de enviar un mensaje, un thread verifica su cola para ver si ha recibido un mensaje.
3. Si es así, retira de la cola el primer mensaje de su cola y lo imprime.
4. Cada hilo alterna entre enviar e intentar recibir mensajes.
5. Dejamos que el usuario especifique la cantidad de mensajes que debe enviar cada thread.
6. Cuando un thread termina de enviar mensajes, recibe mensajes hasta que todos los threads están terminados, momento en el que todos los threads abandonan.

```

1 # pragma omp parallel num_threads(thread_count) \
2   default(none) shared(thread_count, send_max, msg_queues, done_sending
3   )
4   {
5       int my_rank = omp_get_thread_num();
6       int msg_number;
7       srandom(my_rank);
8       msg_queues[my_rank] = Allocate_queue();
9
10      # pragma omp barrier /* Don't let any threads send messages */
11                          /* until all queues are constructed */
12
13      for (msg_number = 0; msg_number < send_max; msg_number++) {
14          Send_msg(msg_queues, my_rank, thread_count, msg_number);
15          Try_receive(msg_queues[my_rank], my_rank);

```

```

15     }
16 #     pragma omp atomic
17     done_sending++;
18 #     ifdef DEBUG
19     printf("Thread %d > done sending\n", my_rank);
20 #     endif
21
22     while (!Done(msg_queues[my_rank], done_sending, thread_count))
23         Try_receive(msg_queues[my_rank], my_rank);
24
25     /* My queue is empty, and everyone is done sending */
26     /* So my queue won't be accessed again, and it's OK to free it */
27     Free_queue(msg_queues[my_rank]);
28     free(msg_queues[my_rank]);
29 } /* omp parallel */

```

### 2.3. Sending Messages

Una probabilidad sección crítica seria tener que acceder a una cola de mensajes para encolar un mensaje.

Cuando adicionamos un mensaje nuevo en la cola, se debe comprobar y actualizar el puntero trasero. Si dos subprocesos intentan hacer esto simultáneamente, podemos perder un mensaje que ha sido puesto en cola por uno de los subprocesos. Los resultados de las dos operaciones entrarán en conflicto y, por lo tanto, poner un mensaje en cola formará una sección crítica.

```

1 void Send_msg(struct queue_s* msg_queues[], int my_rank,
2     int thread_count, int msg_number) {
3     // int msg = random() % MAX_MSG;
4     int msg = -msg_number;
5     int dest = random() % thread_count;
6 #     pragma omp critical
7     Enqueue(msg_queues[dest], my_rank, msg);
8 #     ifdef DEBUG
9     printf("Thread %d > sent %d to %d\n", my_rank, msg, dest);
10 #     endif
11 } /* Send_msg */

```

### 2.4. Receiving Messages

Solo el propietario de la cola (es decir, el hilo de destino) saldrá de la cola de una cola de mensajes determinada. Siempre que eliminemos un mensaje a la vez, si hay al menos dos mensajes en la cola, una llamada **Dequeue** no puede entrar en conflicto con ninguna llamada a **Enqueue**, por lo que si hacemos un seguimiento del tamaño de la cola, puede evitar cualquier sincronización (por ejemplo, directivas críticas), siempre que haya al menos dos mensajes.

```

1     int msg = -msg_number;
2     int dest = random() % thread_count;
3 #     pragma omp critical
4     Enqueue(msg_queues[dest], my_rank, msg);

```

Si almacenamos dos variables, en cola y en cola, entonces el número de mensajes en la cola es y el único hilo que se actualizará dequeued es el propietario de la cola.

```

1 void Try_receive(struct queue_s* q_p, int my_rank) {
2     int src, msg;
3     int queue_size = q_p->enqueued - q_p->dequeued;
4
5     if (queue_size == 0) return;
6     else if (queue_size == 1)
7 #       pragma omp critical
8         Dequeue(q_p, &src, &msg);
9     else
10        Dequeue(q_p, &src, &msg);
11    printf("Thread %d > received %d from %d\n", my_rank, msg, src);
12 } /* Try_receive */

```

## 2.5. Termination Detection.

Si el *subproceso 'u'* o thread ejecuta este código, es muy posible que algún *subproceso 'v'*, envíe un mensaje al subproceso *'u'* después de que *'u'* haya calculado `queue_size = 0`. Por supuesto, después de que el subproceso *'u'* calcula **queue\_size= 0**, terminará y el el mensaje enviado por el *thread 'v'* nunca será recibido.

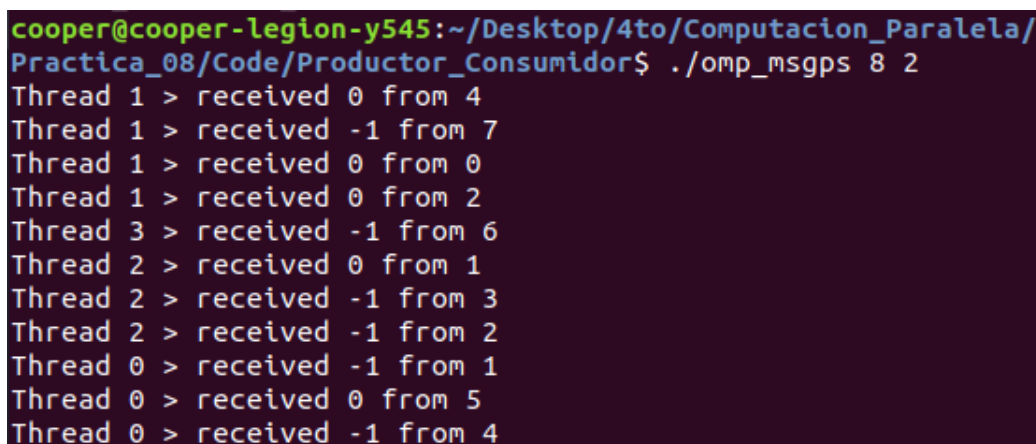
Leugo de que cada thread haya completado el bucle *for*, no enviará ningún mensaje nuevo. Por lo tanto, si agregamos un contador **done\_sending**, y cada hilo lo incrementa después de completar su ciclo *for*, entonces se puede implementar **Done** de la siguiente manera:

```

1 int Done(struct queue_s* q_p, int done_sending, int thread_count) {
2     int queue_size = q_p->enqueued - q_p->dequeued;
3     if (queue_size == 0 && done_sending == thread_count)
4         return 1;
5     else
6         return 0;
7 } /* Done */

```

## 2.6. Ejecución



```

cooper@cooper-legion-y545:~/Desktop/4to/Computacion_Paralela/
Practica_08/Code/Productor_Consumidor$ ./omp_msgps 8 2
Thread 1 > received 0 from 4
Thread 1 > received -1 from 7
Thread 1 > received 0 from 0
Thread 1 > received 0 from 2
Thread 3 > received -1 from 6
Thread 2 > received 0 from 1
Thread 2 > received -1 from 3
Thread 2 > received -1 from 2
Thread 0 > received -1 from 1
Thread 0 > received 0 from 5
Thread 0 > received -1 from 4

```

Figura 4: Ejecución del Programa Productor-Consumidor en OpenMP



### 3. Actividad 3 - Matriz - Vector

De la sección 5.9, ejecutar el problema de producto matriz-vector, analizar y explicar los conceptos de cache coherence , false sharing

#### Resolución

#### 3.1. Ejecución del Problema Matriz - Vector

Los resultados del Tiempo de Ejecución esta en segundos

	Dimensiones de la Matriz					
	8000000 x 8		8000 x 8000		8 x 8000000	
Threads	Tiempo	Eficiencia	Tiempo	Eficiencia	Tiempo	Eficiencia
1	1.351746519	1.0012	1.311885966	1.000	1.200276206	1.000
2	0.6795689765	0.930064907	0.594909248	0.836709789	1.630199251	0.47611873
4	1.311885966	0.635473814	1.145829844	0.265138926	1.832270645	0.0570097

Figura 5: Tiempo de Ejecucion y Eficiencias

- La eficiencia de dos hilos del programa con la entrada  $8 \times 8,000,000$  es más del 20 % menor que la eficiencia del programa con las entradas  $8,000,000 \times 8$  y  $8000 \times 8000$ .
- La eficiencia de cuatro hilos del programa con la entrada  $8 \times 8,000,000$  es más del 50 % menor que la eficiencia del programa con las entradas  $8,000,000 \times 8$  y  $8000 \times 8000$ .
- El rendimiento multiproceso del programa es mucho peor con la entrada  $8 \times 8,000,000$ .

#### 3.2. Caché Coherence

El uso de la coherencia de la caché puede tener un efecto dramático en el rendimiento de los sistemas de memoria compartida. Para ilustrar esto, echemos un vistazo a la multiplicación matriz-vector.

Recuerde que si  $A = (a_{ij})$  es una matriz  $m \times n$  y  $x$  es un vector con  $n$  componentes, entonces su producto  $y = Ax$  es un vector con  $m$  componentes, y su  $i$ -ésimo componente  $y_i$  se encuentra formando el producto escalar del  $i$ -ésimo fila de  $A$  con  $x$  :  $y_i = a_{i0}x_0 + a_{i1}x_1 + \dots + a_{i,n-1}x_{n-1}$

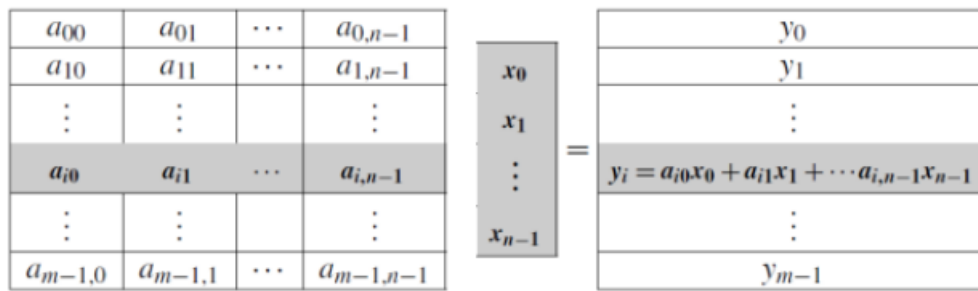


Figura 6: Caché Coherence

La coherencia de la caché se aplica en el "nivel de la línea de caché". Es decir, cada vez que se escribe cualquier valor en una línea de caché, si la línea también se almacena en la caché de otro núcleo, se invalidará toda la línea, no solo el valor que se escribió.

Entonces, cada escritura en algún elemento de  $y$  invalidará la línea en la caché del otro procesador. Por ejemplo, cada vez que el hilo 0 actualiza  $y[0]$  en la declaración C++  $y[i] += A[i][j]*x[j]$ ; si el hilo 2 o 3 está ejecutando este código, tendrá que recargar  $y$ . Cada hilo actualizará cada uno de sus componentes 8.000.000 veces. Vemos que con esta asignación de sub- procesos a procesadores y componentes de  $y$  a líneas de caché, todos los subprocesos tendrán que recargarse y muchas veces. Esto va a suceder a pesar del hecho de que solo un subproceso accede a cualquier componente de  $y$ ; por ejemplo, solo el subproceso 0 accede a  $y[0]$ .

### 3.3. False Sharing

**El uso compartido falso** es un problema común en el procesamiento paralelo de memoria compartida. Ocurre cuando dos o más núcleos contienen una copia de la misma línea de caché de memoria.

Si un núcleo escribe, la línea de caché que contiene la línea de memoria se invalida en otros núcleos. Aunque es posible que otro núcleo no esté usando esos datos (lectura o escritura), puede estar usando otro elemento de datos en la misma línea de caché. El segundo núcleo deberá recargar la línea antes de que pueda acceder a sus propios datos nuevamente.

El hardware de la caché garantiza la coherencia de los datos, pero a un costo de rendimiento potencialmente alto si el intercambio falso es frecuente. Una buena técnica para identificar problemas de uso compartido falsos es detectar aumentos bruscos inesperados en las pérdidas de caché de último nivel utilizando contadores de hardware u otras herramientas de rendimiento.

Como ejemplo simple, se puede considerar una función generada con un bucle **for** que incrementa los valores de la matriz. La matriz es volátil para obligar al compilador a generar instrucciones de almacenamiento en lugar de mantener valores en registros u optimizar el ciclo.

Cada hilo actualizará sus componentes asignados de  $y$  un total de 16.000.000 veces. Parece que muchas, si no la mayoría, de estas actualizaciones están obligando a los subprocesos a acceder a la memoria principal.

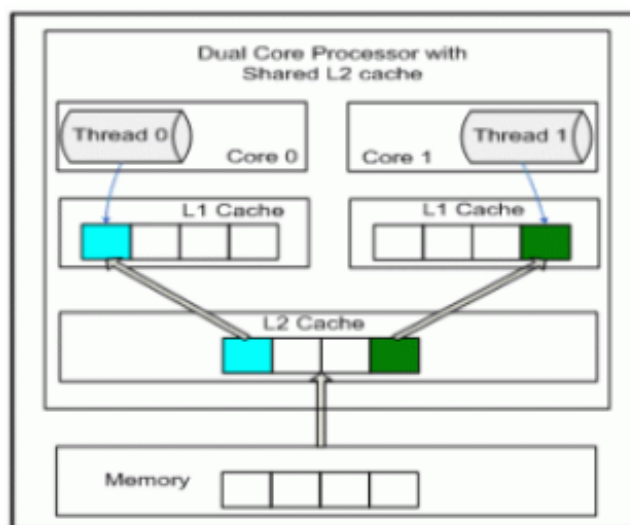


Figura 7: Estructura de False Shearing

### 3.4. Conclusiones

1. OpenMP es un estándar para la programación de sistemas de memoria compartida. Utiliza tanto funciones especiales como directivas de preprocesador llamadas pragmas, por lo que, a diferencia de Pthreads y MPI, OpenMP necesita establecer compatibilidad con el compilador.
2. Una de las propiedades más importantes de OpenMP radica en su diseño que fue realizado para que los desarrolladores puedan paralelizar incrementalmente los programas seriales existentes, en lugar de tener que escribir programas paralelos desde cero.
3. Los programas OpenMP inician múltiples hilos en lugar de múltiples procesos. Los hilos pueden ser mucho más ligeros que los procesos; pueden compartir casi todos los recursos de un proceso, excepto que cada subproceso debe tener su propia pila y contador de programas.
4. Para obtener la función de los prototipos y macros de OpenMP, incluimos el encabezado `omp.h` en los programas de OpenMP.
5. Hay varias directivas OpenMP que inician múltiples hilos; La más general es la directiva `parallel`:

```
#pragma omp parallel
structured block
```

6. La colección de hilos que ejecutan el bloque de código se denomina **equipo**. Uno de los threads en el equipo es el thread que ejecutaba el código antes de la directiva `parallel`. Este hilo o thread se llama **master**. Los threads adicionales iniciados por la directiva `parallel` se llaman **slaves**. Cuando todos los hilos están terminados, los hilos esclavos se terminan o se unen y el hilo maestro continúa ejecutando el código más allá del bloque estructurado.

7. Una directiva de ***barrier*** hará que los subprocesos de un **team** se bloqueen hasta que todos los subprocesos hayan alcanzado la directiva. Entonces las directivas ***parallel***, ***parallel for*** y ***for*** tienen barreras implícitas al final del bloque estructurado.

## 4. Repositorio

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

## 5. Referencias

Introduction to Parallel Programming - Capítulo 5 - Peter Pacheco

<https://github.com/yangyang14641/ParallelProgrammingCourse>