



Universidad Católica
San Pablo

CIENCIA DE LA COMPUTACIÓN

Computación Paralela y Distribuida

OpenPM Aplicaciones

Rodrigo Alonso Torres Sotomayor

CCOMP 8-1

.^{El} alumno declara haber realizado el presente trabajo de acuerdo a las normas de la Universidad Católica San Pablo”

OpenPM Aplicaciones

1. Introducción

En este trabajo se analizará como es que se aplica OpenPm a la programación de bucles y como se maneja el concepto de productor - consumidor. Donde se verá que tan eficiente puede llegar a ser la aplicación de este en el algoritmo Odd-Even Sort.

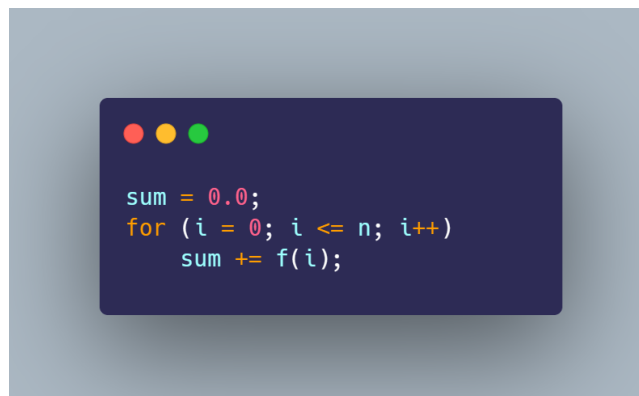
2. Requisitos de la PC

Para las pruebas de las barreras en este trabajo se utilizó una máquina virtual y las características de dicha máquina son:

- Sistema Operativo: Linux Ubuntu 20.04.
- Memoria base (RAM): 6.1 GiB
- Procesador: Intel Corei5-10400F CPU 2.90GHzx4
- Disk Capacity: 85.9 GB

3. Programación de bucles

A diferencia de la implementación de paralelización de bucles, se vio que la asignación exacta de las iteraciones del bucle a los hilos depende del sistema. Sin embargo, la mayoría de las implementaciones de OpenMP utilizan una partición aproximada por bloques: si hay n iteraciones en el bucle serial, entonces en el bucle paralelo las primeras $n / \text{thread count}$ se asignan al thread 0, las siguientes $n/\text{thread count}$ se asignan al thread 1 y así sucesivamente. No es difícil pensar en situaciones en las que esta asignación de iteraciones a threads podría ser menos óptima. Por ejemplo, supongamos que queremos paralelizar el bucle:

A code editor window with a dark blue background and light blue text. It contains the following code:

```
sum = 0.0;
for (i = 0; i <= n; i++)
    sum += f(i);
```

Figura 1: Bucle a paralelizar.

Además hay que pensar que el tiempo requerido por la llamada a ' $f()$ ' es proporcional al tamaño del argumento ' i '. Entonces, una partición por bloques de las iteraciones asignará mucho más trabajo al thread '*contador de thread - 1*' de lo que asignará al thread 0. Una asignación de trabajo más adecuada para los hilos podría lograrse mediante una partición cíclica de las iteraciones entre los threads. En una partición cíclica, las iteraciones se asignan una por una, de manera *round-robin* (de forma rotativa) a los threads. Supongamos '**t = contador de thread**'. Entonces, una partición cíclica asignará las iteraciones de la siguiente manera:

La declaración de la función $f()$ se puede ver en la Figura ??.

La llamada a $f(i)$ llama a la función seno i veces, y, por ejemplo, el tiempo para ejecutar $f(2i)$ requiere aproximadamente el doble de tiempo que el tiempo para ejecutar $f(i)$.

Thread	Iterations
0	0, n/t , $2n/t$, ...
1	1, $n/t+1$, $2n/t+1$, ...
\vdots	\vdots
$t-1$	$t-1$, $n/t+t-1$, $2n/t+t-1$, ...

Figura 2: Partición cíclica de la iteración.

```
double f(int i) {
    int j, start = i*(i+1)/2, finish = start + i;
    double return_val = 0.0;
    for (j = start; j <= finish; j++) {
        return_val += sin(j);
    }
    return return_val;
}
```

Figura 3: Declaración $f()$.

3.1. Resultados

Cuando se ejecuta el programa con $n = 10,000$ y un solo thread, el tiempo de ejecución fue de 3.67 segundos. Cuando ejecutamos el programa con dos threads y la asignación predeterminada, es decir, las iteraciones de 0 a 5000 en el thread 0 y las iteraciones de 5001 a 10,000 en el thread 1, el tiempo de ejecución fue de 2.76 segundos. Esto representa un aumento de velocidad de solo 1.33. Sin embargo, cuando se ejecuta el programa con dos threads y una asignación cíclica, el tiempo de ejecución se redujo a 1.84 segundos. Esto supone un aumento de velocidad de 1.99 en comparación con la ejecución de un solo thread y un aumento de velocidad de 1.5 en comparación con la partición por bloques de dos threads.

3.2. Análisis

Podemos ver que una buena asignación de iteraciones a threads puede tener un efecto muy significativo en el rendimiento. En OpenMP, la asignación de iteraciones a thread se llama *scheduling*, y la cláusula de *schedule* se puede utilizar para asignar iteraciones en una directiva '*parallel for*' o '*for*'.

4. Productor Consumidor

El problema del productor y el consumidor es un problema clásico de sincronización en la programación concurrente y paralela. Se refiere a la colaboración entre dos tipos de procesos, los "productores" que generan datos y los consumidores que consumen esos datos. Este problema se vuelve más interesante y desafiante en el contexto de la programación paralela con herramientas como OpenMP, donde múltiples hilos trabajan en conjunto para resolverlo.

El problema se plantea de la siguiente manera:

1. Hay un "almacén" (buffer o cola) compartido entre los productores y consumidores.
2. Los productores generan datos y los colocan en el almacén (buffer) si hay espacio disponible. Si el almacén está lleno, deben esperar hasta que haya espacio.
3. Los consumidores retiran datos del almacén y los procesan. Si el almacén está vacío, deben esperar hasta que haya datos disponibles.

El objetivo es garantizar que los productores y consumidores trabajen de manera sincronizada y segura, evitando problemas como la condición de carrera (race condition) o el bloqueo mutuo (deadlock).

En el contexto de OpenMP, se puede abordar el problema del productor y el consumidor utilizando directivas de OpenMP para controlar la ejecución paralela de hilos y asegurarte de que los productores y consumidores interactúen adecuadamente. Aquí hay una descripción general de cómo se podría implementar el problema utilizando OpenMP:

1. Declarar el almacén compartido como un arreglo o una estructura de datos compartida entre los hilos. Puedes usar variables compartidas y asegurarte de que se manejen de manera segura utilizando las cláusulas adecuadas de OpenMP, como `shared` y `private`.
2. Utiliza directivas de OpenMP para crear hilos que actúen como productores y consumidores. Puedes usar `#pragma omp parallel` para crear un grupo de hilos.
3. Utiliza las cláusulas de OpenMP como `#pragma omp critical` o `#pragma omp atomic` para garantizar que los hilos productores y consumidores realicen las operaciones de colocación y extracción de datos de manera segura y sincronizada. Por ejemplo, dentro de una región crítica, puedes verificar si el almacén está lleno o vacío antes de realizar operaciones.

5. Odd even sort

El Odd-Even Sort es un algoritmo de ordenamiento paralelo que se utiliza para ordenar elementos en una lista. La principal diferencia entre el Odd-Even Sort paralelo y su implementación secuencial radica en cómo se distribuye el trabajo entre múltiples hilos o procesos en paralelo.

El Odd-Even Sort paralelo se beneficia de la capacidad de ejecución en paralelo de múltiples tareas para acelerar el proceso de ordenamiento. Para implementar el Odd-Even Sort paralelo usando OpenMP, puedes dividir la lista en partes más pequeñas y paralelizar el proceso de ordenamiento de estas partes. Aquí hay un resumen de cómo funcionaría:

1. Divide la lista en partes iguales para distribuir entre los hilos disponibles. Por ejemplo, si tienes una lista de N elementos y M hilos, cada hilo ordenará N/M elementos.
2. Cada hilo ejecutará el algoritmo Odd-Even Sort en su parte de la lista de manera independiente. El algoritmo Odd-Even Sort compara y cambia de posición los elementos de manera iterativa hasta que toda la lista esté ordenada.
3. Después de cada iteración, debes sincronizar los hilos para asegurarte de que todos los elementos estén en la posición correcta antes de continuar con la siguiente iteración.
4. Repite el proceso hasta que la lista esté completamente ordenada.

La principal ventaja de esta implementación paralela es que varios hilos pueden trabajar simultáneamente en diferentes partes de la lista, lo que puede acelerar significativamente el proceso de ordenamiento en comparación con la versión secuencial.

5.1. Resultados

Para las pruebas se utilizó dos versiones del odd-even sort, donde la primera versión bifurca (crea) y une (finaliza) thread count hilos en cada iteración, mientras que la otra versión bifurca (crea) y une (finaliza) los hilos solo una vez. Estos versiones se pueden encontrar en

Los resultados de tiempo (en segundos) de ejecución de la primera versión fueron:

	1 Thread	2 Threads	3 Threads	4 Threads	6 Threads	8 Threads
100	8.304700e-05	1.452280e-04	1.852080e-04	1.9593e-04	5.771663e-03	6.8678317e-03
1000	2.599501e-03	2.11118e-03	1.418862e-03	2.091943e-02	8.442898e-02	6.678228e-02
10000	1.759169e-01	1.306921e-01	8.947834e-02	7.174362e-02	4.727225e-01	4.960042e-01
100000	1.854363e+01	9.558188e+00	7.201494e+00	5.977450e+00	1.168758e+01	1.128887e+01

Cuadro 1: Tiempo de ejecución en segundos del omp-odd-even1.c

Los resultados de tiempo (en segundos) de la segunda versión fueron:

	1 Thread	2 Threads	3 Threads	4 Threads	6 Threads	8 Threads
100	4.614700e-05	1.373930e-04	1.140580e-04	1.370742e-03	1.041148e-02	4.564642e-03
1000	1.726716e-03	1.0370e-03	1.589636e-03	1.373949e-03	2.137264e-02	3.066635e-02
10000	1.608011e-01	7.702742e-02	5.386036e-02	4.569191e-02	2.633806e-01	3.336477e-01
100000	1.857014e+01	9.215401e+00	6.891375e+00	5.358213e+00	8.982011e+00	8.819563e+00

Cuadro 2: Tiempo de ejecución en segundos del omp-odd-even2.c

A través de estos resultados es que se logra observar como es que a medida que

6. Código de la implementación

Los códigos utilizados para estas pruebas estan en: <https://github.com/RodATS/ComputacionParalela.git>