Universidad del Valle de Guatemala
Computación Paralela y Distribuida
Docente: Sebastián Galindo
Semestre 2, 2023

**Laboratorio #1**

**Fecha de Entrega**: 20 de agosto, 2023.

**Descripción**: En equipos de 3 personas, realice la siguiente implementación de un algoritmo secuencial y su transformación en versión paralela usando OpenMP.

**En todos los incisos debe dejar copia (en el PDF a entregar) de lo siguiente**:

- Función(es) principal(es) del código. La parte que contiene el cálculo o tarea más importante. No incluya encabezados ni operaciones de interacción con el usuario.
  - Bloque de código modificado
- Captura de pantalla de la salida (x1)
- Mediciones de tiempo (min 5, usar N o tamaño adecuado)
- Métrica de desempeño: speedup, efficiency

**Entregables**:

- Sus respuestas a los incisos requeridos en formato PDF.
- Código fuente (.c /.cpp) de programas finales (secuencial, paralelo, alternativo)
- Instrucciones de compilación (o bien un makefile)

**Materiales**: necesitará una máquina virtual con Linux.

**Contenido**:

**Ejercicio 1 (70 puntos)**

Existen muchas aproximaciones numéricas para estimar el valor de PI. Estas aproximaciones generalmente son series numéricas infinitas que nos permiten calcular PI con mayor precisión según el número de términos de la serie. Para nuestro laboratorio usaremos la siguiente serie:

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}.$$

Para implementar la serie anterior, podemos usar el siguiente algoritmo secuencial;

```
1. double factor = 1.0;
2. double sum = 0.0;
3. for (k = 0; k < n; k++) {
4.     sum  += factor/(2*k+1);
5.     factor = -factor;
6. }
7. pi_approx = 4.0*sum;
```

Como primer intento, vamos a agregar una directiva ***#pragma omp for*** y una cláusula ***reduction(+:sum)*** para acumular las sumas parciales de la serie numérica.

```
1. double factor = 1.0;
2. double sum = 0.0;
3. #pragma omp parallel for num_threads(thread_count) \
        reduction(+:sum)
4. for (k = 0; k < n; k++) {
5.     sum  += factor/(2*k+1);
6.     factor = -factor;
7. }
8. pi_approx = 4.0*sum;
```

a. (**5 pts**) Implemente la version secuencial y pruebe que esté correcta (piSeriesSeq.c). Implemente la versión paralela mencionada (**piSeriesNaive.c)** , compílelo y ejecútelo. Realice al menos 5 mediciones del valor con *threads >= 2 y n >= 1000* (pruebe ir incrementando, *registre todos los números resultantes*). Describa lo que sucede con el resultado respecto al valor preciso de PI (*3.1415926535 8979323846*).

b. (**5 pts**) Analíce el código fuente de **piSeriesNaive.c**. Identifique el tipo de dependencia que se da con la variable ***factor***.

c. (**5 pts**) Observe el algoritmo y la serie numérica. Describa en sus propias palabras la razón por la cual ***factor = − factor***.

d. (**5 pts**) Para eliminar la dependencia de loop, debemos modificar la forma como calculamos el valor ***factor***. Guarde una copia del programa anterior y reemplace el siguiente segmento de código. Realice al menos 5 mediciones del valor con t*hreads >= 2 y n >= 10e6* (*registre todos los números resultantes*). Describa lo que sucede con el resultado respecto al valor preciso de PI (*3.1415926535 8979323846*):

| | |
|---|---|
| **sum  += factor/(2*k+1);**<br>**factor = -factor;** | if (k % 2 == 0)<br>    factor = 1.0;<br>else<br>    factor = -1.0; |

| | |
|---|---|
| | `sum += factor/(2*k+1);` |
| *(versión con operador ternario)* | `factor = (k % 2 == 0) ? 1.0 : -1.0;`<br>`sum += factor/(2*k+1);` |

e. (**10 pts**) Ejecute el mismo código pero threads = 1 y realice al menos 5 mediciones (*registre todos los números resultantes*). Describa en sus propias palabras la razón por la cual el resultado es diferente.

f. (**10 pts**) Debemos cambiar el ámbito (scope) de una variable para resolver el problema que pueda darse respecto a los resultados en la versión paralela con threads > 1. Modifique el programa usando la cláusula de cambio de scope private(). Realice al menos 5 mediciones del valor con t*hreads >= 2 y n >= 10e6* (*registre todos los números resultantes*). Describa lo que sucede con el resultado respecto al valor preciso de PI (*3.1415926535 8979323846*). *Incluya una captura de pantalla del resultado final*.

g. (**15 pts**) Use la última versión paralela con n = 10e6 (o más si e6 es poco para la computadora de cada uno) y el número de hilos según la cantidad de cores de su sistema (i.e: nproc). Realice el cálculo de **speedup**, **eficiencia**, **escalabilidad fuerte** y **escalabilidad débil** para las siguientes condiciones (solamente modifique un parámetro a la vez). Tome por lo menos 5 medidas para sus datos:

| tiempo secuencial (threads = 1) | tiempo paralelo (threads = cores) | tiempo paralelo (threads = 2*cores) | tiempo paralelo (n = n*10 && threads = cores) |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

| | | | |
|---|---|---|---|
| | | | |
| | | | |

h. (**15pts)** Usando la versión final de su programa paralelo, modificarlo y pruebe las diferentes políticas de planificación y block_size. Registre sus datos y calcule las diferencias en speedup para cada uno de los mecanismos de scheduling (**static**, **dynamic**, **guided**, **auto**) usando los siguientes parámetros: **n = 10e6** (o más si aplica), **threads = cores**, probar **block_size** de **16, 64, 128** (en todos menos auto). Tome por lo menos 5 medidas de cada una. (tip: recuerde la opción *runtime* que permite pasarle el tipo para agilizar el proceso) Con cuál política de planificación obtuvo mejores resultados?

**Ejercicio 2 (10 puntos)**

Podemos notar que la serie mencionada sigue un patrón específico: suma los elementos con índices pares y resta los que tienen índices impares. Podemos agrupar la sumatoria de la siguiente manera:

$$\pi = 4\left[\sum_{i \in Even}^{\infty} \frac{1}{2i+1} - \sum_{j \in Odd}^{\infty} \frac{1}{2j+1}\right] \quad (2)$$

Siguiendo este planteamiento alternativo (dos sumatorias versus una) podemos evitar el tener que calcular el factor 1/-1 en cada paso del ciclo for, ahorrando 1 instrucción (de 2) por ciclo (la mitad).

a. (**15 pts**.) Implemente el programa descrito por la ecuación anterior (**piSeriesAlt.c)** , compílelo y ejecútelo. Realice al menos 5 mediciones del valor con *threads >= 2 y n >= 10e6 o adecuado* (*registre todos los números resultantes*). Describa lo que sucede con el resultado respecto al valor preciso de PI (*3.1415926535 8979323846*). Haga una comparación con los mismos parámetros (threads, n) de esta versión y su mejor versión del inciso h.

b. (**15 pts**.) Como se mencionó en clase, los compiladores (como gcc/g++) tienen muchas opciones de las cuales incluyen optimizaciones de codigo. En términos de speedup, lo que nos interesa son las opciones que priorizan tiempo de ejecución sobre tamaño del programa/código o facilidad para debugeo (sugerencia de lectura: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html ). Pruebe compilar su mejor versión al momento pero esta vez agregando la opción de optimización "-O2". Mida varias veces el tiempo de ejecución y compare con la versión sin la bandera de optimización. ¿Qué pudieron observar? Comenten entre el grupo e incluyan un resumen de su discusión.