# Homework 2: OpenMP Usage and Code Optimization

Nombre: Ing. Luis Carlos Alvarez Mata | Carne: 200902144
Nombre: Ing. Verny Moreles Soto | Carne: 201116212

OpenMP es un modelo de programación en paralelo para memoria compartida y multiprocesadores distribuidos de memoria compartida. En su nivel más elemental, OpenMP es un conjunto de directivas de compilador y rutinas de biblioteca en tiempo de ejecución invocables que amplían Fortran, C y C ++ para expresar el paralelismo de memoria compartida. OpenMP ha tenido mucho éxito en la explotación del paralelismo estructurado en aplicaciones. Con la creciente complejidad de las aplicaciones, existe una necesidad creciente de abordar el paralelismo irregular en presencia de estructuras de control complicadas. [1, 2, 3]

## 1. Implementación [4]

El programa PI serial proporcionado se aproxima a una integral definida cuyo resultado debería ser igual a pi, Figura 1. Para desarrollar esta aproximación se utiliza la instrucción *for* la cual realizar el calculo de cada *step*, para donde se utilizan 1000000000 *steps* para obtener tiempos no mayor a 10 segundos. La ejecución de los diferentes escenarios se realiza en una computadora con las siguientes características:

- Procesador: 2nd Generation Intel Core i5-2450M (2.6GHz)
- Gráficos: Gráficos HD Intel 3000.
- Memoria: 4GB DDR3.

```
for (i = 1; i <= num_steps; i++) {
   x = (i - 0.5) * step;
   sum = sum + 4.0 / (1.0 + x * x);
}
```

Donde es importante detallar en detalle las especificaciones del procesador y es necesario resaltar la cantidad de núcleos y de subprocesos para el analisis. A continuación, las especificaciones [5]:

- Cantidad de núcleos: 2
- Cantidad de subprocesos: 4
- Frecuencia básica del procesador: 2,50 GHz
- Frecuencia turbo máxima: 3,10 GHz
- Caché: 3 MB Intel® Smart Cache
- Velocidad del bus: 5 GT/s
- TDP: 35 W

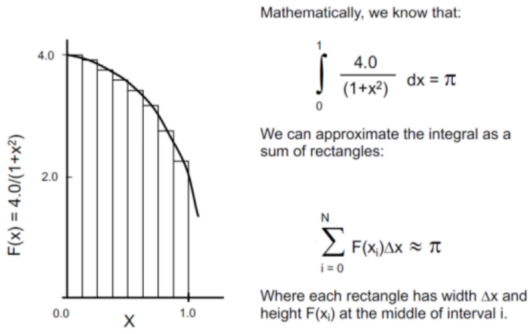


Figura 1: Descripción matemática de la calculo de Pi

## 1.1. Private usage

Se crea un equipo de subprocesos OpenMP que ejecutan la región. Ademas se utilizan las clausulas:

- reduction, la cual especifica un identificador de reducción, donde se le proporciona la operacion de suma (+) y la variable de reducción (sum).
- private, declara que el elemento es privados para una tarea.

```
#pragma omp parallel
{
   #pragma omp for reduction(+:sum) private(x)
   for (i=0; i< num_steps; i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
   }
}
```

## 1.2. Teams usage

Se utiliza la directiva *target* para especificar una construcción de destino con una construcción de ciclo de trabajo compartido. A continuacio se utiliza la directiva *teams distribute parallel for* para especificar una construcción de equipos que contiene una construcción de bucle de distribución de trabajo en paralelo y ninguna otra declaración. Ademas se utilizan las clausulas:

- reduction, la cual especifica un identificador de reducción, donde se le proporciona la operacion de suma (+) y la variable de reducción (sum).
- private, declara que el elemento es privados para una tarea.

- $\text{num}_t eams, estableceelnúmerodeequiposenlaregióndeequiposactualmap, asignaunelementodelistaoriginaldelentornodedatosdel$

```
#pragma omp target teams distribute parallel for num_teams(num_teams) reduction(+:sum) private(x)
    map(tofrom:sum)
for (i=0; i< num_steps; i++){
   x = (i+0.5)*step;
   sum = sum + 4.0/(1.0+x*x);
}
```

### 1.3. Thread usage

Crea un equipo de subprocesos OpenMP que ejecutan la región, donde se delimita el numero de subprocesos a usar para las regiones con la clusula $num_threads. Ademas se utilizan las clausulas$ :

- *reduction, la cual especifica un identificador de reducción, donde se le proporciona la operacion de suma (+) y la variable de reducción (sum).*

  *private, declara que el elemento es privados para una tarea.*

```
#pragma omp parallel num_threads(a)
{
    #pragma omp for reduction(+:sum) private(x)
    for (i=0; i< num_steps; i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
    }
}
```

## 2. Resultados

Para el análisis de resultados y llegar a entender las diferencias entre los modos de uso de OpenMP se plantearon los siguientes escenarios:

1. Caso 1, ejecución serial - pi.
2. Caso 2, utilizando la reducción y construcciones privadas - pi_omp_private.
3. Caso 3, utilizando los equipos y distribuir constructos - pi_omp_team.
4. Caso 4, utilizando un hilo - pi_omp_threads_1.
5. Caso 5, utilizando dos hilos - pi_omp_threads_2.
6. Caso 6, utilizando tres hilos - pi_omp_threads_3.
7. Caso 7, utilizando cuatro hilos - pi_omp_threads_4.

En la Figura 2 se muestran los resultados del tiempo de ejecución para cada escenario. Donde se visualiza que los tiempos más altos son los obtenidos por el caso 1 y 4, los cuales son estadísticamente iguales debido a que ambos realizan una ejecución serial.

Como siguiente caso se tiene el caso 6, donde a pesar de tener mayor cantidad de hilos que el caso 5 el tiempo de ejecución es mayor. Y esto se debe a que el procesador esta optimizado a trabajar con un numero de subprocesos o hilos pares, por lo tanto, el ejecutarse en 3 hilos y existir el cambio de contexto es posible que algún subproceso se esté realizando entre los dos procesadores y esto hace que el tiempo de ejecución sea mayor en comparación al caso 5.

Posteriormente se tiene un tercer bloque de casos los cuales estadísticamente, pero se tienen pequeñas diferencias. Los casos 2, 3, 5 y 7 tienen el mismo tiempo de ejecución. El caso 7 en la teoría nos debería presentar el tiempo menor al tener más hilos y poder paralelizar en más subprocesos la ejecución, pero vemos que los casos 2, 3 y 5 presenta tiempos muy similares.

Para el caso 2, OpenMP asigna la mayor cantidad de hilos posible del procesador, en este caso 4 hilos, por la cual se obtiene el mismo tiempo de ejecución que el caso 7. Para el caso 5, pese a tener 2 hilos menos en comparación con el caso 2 y 7, obtenemos el mismo tiempo de ejecución. Al profundizar en los procesadores Intel se entiende que el soporte de multithreading tiene un costo al manejar dos hilos por core, el cual al simplificarse el soporte de un hilo por core y tener la misma carga computacional por core el costo de ejecutar de un hilo por core y dos hilos por core es similar.

En este tercer bloque el caso 3 está un escalón más arriba al tener el menor tiempo de ejecución. La similitud del tiempo es debido a que se utiliza la directiva *teams distribute parallel for*, la cual asigna automáticamente la mayor cantidad de hilos que soporta el procesador donde a pesar de tener 3 equipos esto no tiene impacto. No haber utilizado la directiva *parallel for* realiza una ejecución serial

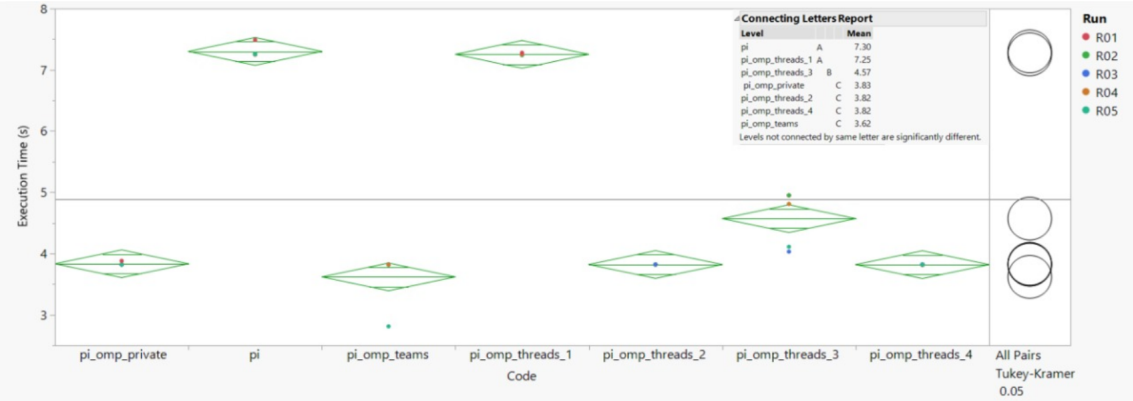


Figura 2: Ejecucion de los casos

## Referencias

[1] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," vol. 5, no. 1, pp. 46–55, conference Name: IEEE Computational Science and Engineering.

[2] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel Programming in OpenMP*. Morgan Kaufmann, google-Books-ID: vuAY5C5C1W0C.

[3] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of OpenMP tasks," vol. 20, no. 3, pp. 404–418, conference Name: IEEE Transactions on Parallel and Distributed Systems.

[4] OpenMP, "OpenMP application programming interface." [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[5] Intel Corporation. Procesador intel® core™ i5-2450m (caché de 3 m, hasta 3,10 GHz) especificaciones de productos. [Online]. Available: https://ark.intel.com/content/www/es/es/ark/products/53452/intel-core-i5-2450m-processor-3m-cache-up-to-3-10-ghz.html