# Ejercicios cortos para OpenMP

## Objetivos

Que el estudiante sea capaz de:

1. Diseñar un algoritmo paralalelo identificando y combinando los patrones de programación paralela estructurada adecuados tales como: fork, join, map y reduction.
2. Implementar un algoritmo paralelo identificando y combinando directivas pragma, funciones y tipos de OpenMP en lenguaje C++.
3. Depurar sistemáticamente un programa hasta lograr el funcionamiento esperado utilizando algún depurador simbólico (asociado al IDE de su preferencia) o haciéndolo desde la consola mediante la interfaz de "gdb" por "comandos".
4. Evaluar el desempeño de un programa utilizando conceptos simples como el tiempo de ejecución mediante funciones de OpenMP para tal efecto.

## Procedimiento

Para cada ejercicio, el procedimiento general es:

1. Diseñar un algoritmo paralelo mediante el método de Fosler: partición, comunicación, agregación y asignación de tareas.
2. Representar el algoritmo diseñado adecuadamente (con alguna herramienta de escritorio o en línea: https://www.draw.io) para discutirlo entre sí y con el docente o asistente.
3. Elaborar el código preliminar.
4. Depurarlo hasta asegurar su funcionamiento correcto.
5. Evaluar y comparar (cuando se requiera) su desempeño.
6. PRESENTAR al docente o asistente cada una de las versiones (cuando se especifiquen en el ejercicio) para su evaluación inmediata.
7. Dependiendo de la evaluación, continuar en el punto #2 o #3 o entregarlo y recibir la evaluación.

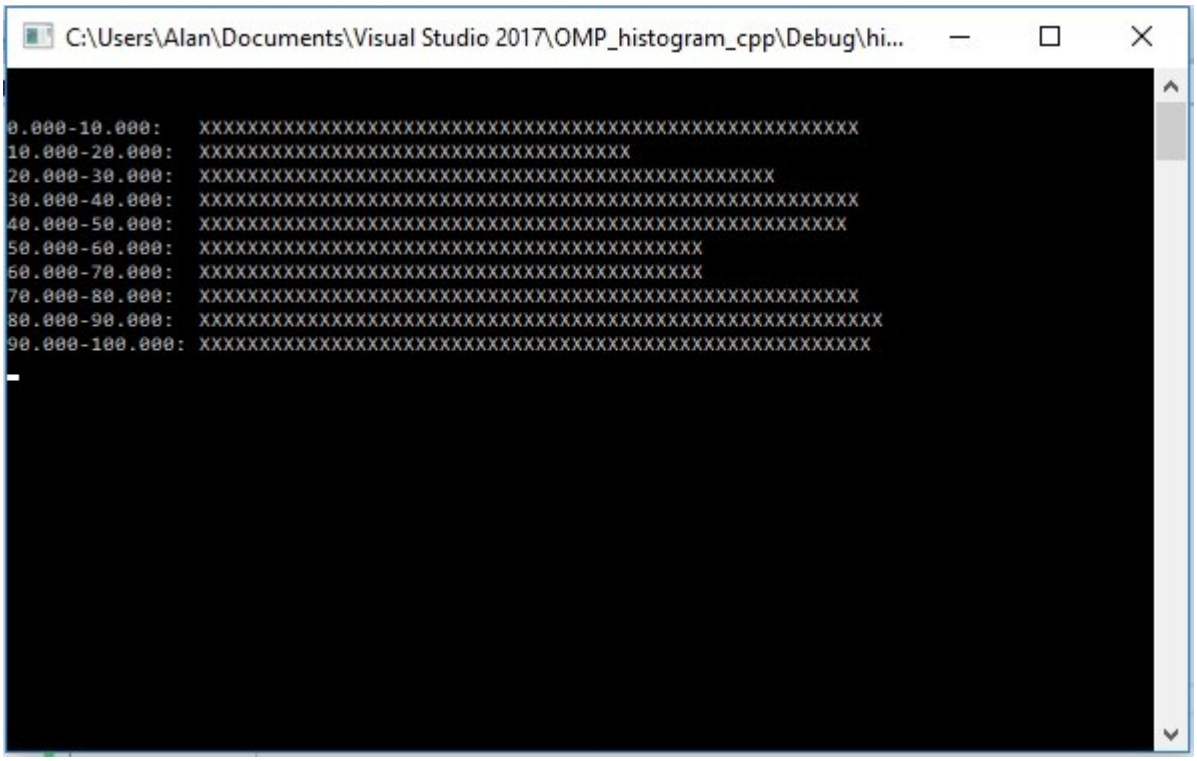
## Evaluación

Dado que son cinco ejercicios, y el peso total de esta actividad en la evaluación del curso es de 10%, cada ejercicio pesa 2%.

**Ejercicio #1** (basado en la asignación de programación OpenMP #1 del texto de Peter Pacheco)

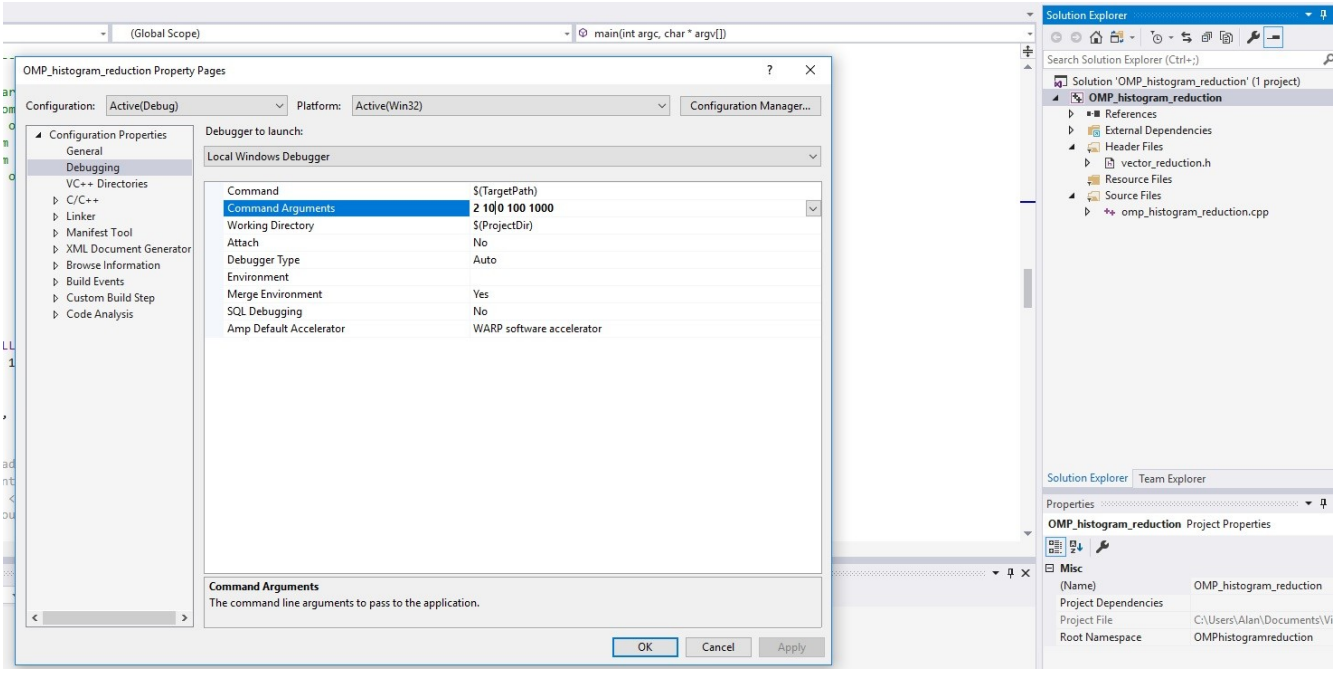
Elabore un programa paralelo mediante OpenMP en C++ para el ejemplo que se usó para ilustrar el método de Fosler que genera un histograma con una muestra al azar de números enteros. El programa recibe parámetros de consola para:
1. la cantidad de hilos,
2. la cantidad de rangos (y por ende de barras del histograma),
3. el valor mínimo del intervalo al que pertenecerán los números generados al azar,
4. el valor máximo del intervalo al que pertenecerán los números generados al azar,
5. la cantidad total de números al azar que se deberán generar.

La salida del programa es algo como:




**NOTAS**:

1. La elaboración y representación del algoritmo paralelo **debe** partir del código provisto.
2. En el código provisto, se entiende por "bin" una categoría de números, es decir un rango de números para el cual se representa finalmente una barra en el histograma.
3. Por tanto, el vector bin_count, representa, en cada posición i el conteo de un rango o "bin".
4. Para asignar valores a los parámetros de consola cuando se ejecute el programa desde Visual Studio, los mismos deben ser indicados en el "Solution Explorer" clickar-derecho el nombre del proyecto, luego escoger la opción "Debugging" y asignar los valores en el campo de la derecha denominado "Command Arguments".




**Procedimiento específico:**

1. Elabore una primera versión modificando el código provisto que NO use "reduction".
2. Elabore una segunda versión modificando la primera que use "reduction".
3. Compare el desempeño de ambas versiones usando "`omp_get_wtime()`" o <chrono> (ver trozo de código en el sitio del curso).
4. No olvide pedir la evaluación inmediata cuando considere que el trabajo está completo.

**Ejercicio #2** (basado en la asignación de programación OpenMP #2 del texto de Peter Pacheco)

Elabore un programa paralelo mediante OpenMP en C++ para aproximar el valor del número PI mediante el método de MonteCarlo. El programa deberá recibir como parámetros la cantidad de hilos y la cantidad de términos que se sumarán para aproximar el valor de PI. Ver página 267 del texto de Peter Pacheco.

La salida del programa es la aproximación de PI que se deberá contrastar en su precisión con el que se obtenga mediante la expresión: 4.0*atan(1.0).

$$\frac{\text{number in circle}}{\text{total number of tosses}} = \frac{\pi}{4},$$

since the ratio of the area of the circle to the area of the square is $\pi/4$.

We can use this formula to estimate the value of $\pi$ with a random number generator:

```
number_in_circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
   x = random double between -1 and 1;
   y = random double between -1 and 1;
   distance_squared = x*x + y*y;
   if (distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4*number_in_circle/((double) number_of_tosses);
```

**NOTAS**:

1. No olvide pedir la evaluación inmediata cuando considere que el trabajo está completo.

**Ejercicio #3**

Elabore un programa paralelo mediante OpenMP en C++ para ordenar mediante el algoritmo "merge-sort" un vector de números enteros generados al azar. El programa deberá recibir como parámetros del usuario la cantidad de hilos y la cantidad de números que se deberán generar al azar.

La salida del programa es la secuencia ordenada de números generados al azar por consola.

**NOTAS**:

1. El programa deberá basarse en el algoritmo clásico de "merge-sort" que aparece en: https://en.wikipedia.org/wiki/Merge_sort.
2. El programa deberá usar los siguientes componentes de la STL:
2.1 <vector>
2.2 <algorithm>: "sort" y "merge"
3. Obviamente deberán combinarse adecuadamente para que sea un algoritmo paralelo, porque ninguno de los dos lo es.
4. No olvide pedir la evaluación inmediata cuando considere que el trabajo está completo.
5. Puede suponer que la cantidad de hilos es potencia de dos y la cantidad de elementos par.

**Procedimiento específico**:

1. Elabore una primera versión en la que la intercalación o "merge" se cargue al hilo maestro.
2. Elabore una segunda versión en la que la intercalación se haga por niveles y se distribuya a partes iguales entre hilos en forma de árbol binario. Para esta versión puede suponer que el total de números generados al azar es potencia de dos. En el siguiente diagrama, el primer nivel (0,...7) representa el ordenamiento mediante "sort" por parte de 8 (2 a la tres) hilos. En los siguientes niveles el símbolo de "+" representa la intercalación mediante "merge" por cuatro, dos y un hilo.

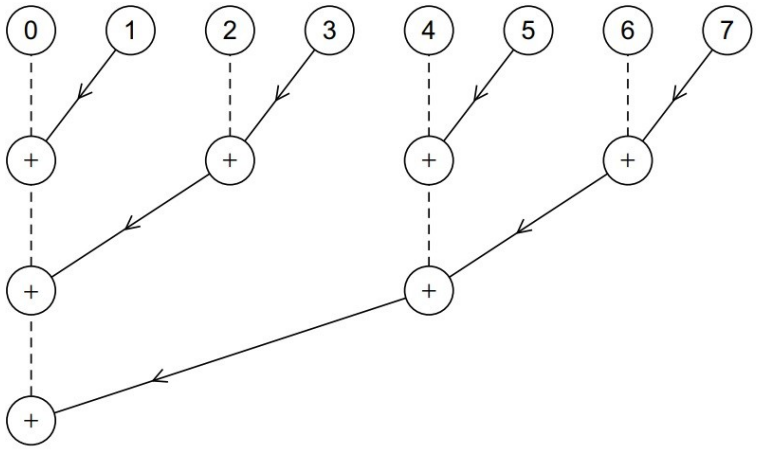



**FIGURE 2.23**

Adding the local arrays

**Ejercicio #4** (basado en la asignación de programación OpenMP #6 del texto de Peter Pacheco)

Elabore un programa paralelo mediante OpenMP en C++ demuestre el concepto de "productores" y "consumidores" como una estrategia de división de tareas útil al paralelizar una tarea más compleja. Cada hilo "productor" leerá un archivo "por líneas", las colocará en una cola compartida entre todos los hilos "productores" (reutilice la cola genérica de mensajes analizada en clase "queue_lk.h") como un mensaje. Los hilos "consumidores" por su parte, tomarán una de las líneas (o mensajes en la cola) y la "tokenizará" lo que debe entenderse como separar las palabras de la línea. Se puede suponer que las palabras en cada mensaje ya vienen separadas por blancos. Los "consumidores" desplegarán (ordenadamente, es decir, evitando los conflictos), por medio de "cout", las palabras de las líneas "tokenizadas".

Los parámetros de entrada son:
1. cantidad de hilos productores,
2. cantidad de hilos consumidores,
3. un archivo de texto por cada hilo consumidor, el nombre de cada archivo puede ser "arch_*.txt" sustituyendo el "*" por un número de hilo, por ejemplo: "arch_1.txt", "arch_0.txt" y así sucesivamente.

La salida por consola es la "tokenización" de cada línea generada por cada uno de los hilos consumidores. Se debe asegurar que no se entremezclan las "tokenizaciones".

**NOTAS**:

1. No olvide pedir la evaluación inmediata cuando considere que el trabajo está completo.

**Ejercicio #5**

Elabore un programa paralelo mediante OpenMP en C++ que paralelice el algoritmo de Floyd-Warshall para encontrar los costos de los caminos más cortos en un grafo con adyacencias ponderadas y positivas, ver: https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm. Utilice el código base provisto (omp_nested_floyd_warshall.cpp) y así sólo deberá elaborar el código de la función "algoritmoFloydWarshall(...)".

El programa indicado genera grafos sin pesos, pero usted puede suponer que cada adyacencia tiene el peso de 1.

La entrada a su programa es un archivo con la matriz de adyacencia generada, el cual incluye la cantidad de vértices en la primera fila. La salida será una matriz de costos que se deberá desplegar por consola.

**Procedimiento específico**:

1. Elabore una primera versión que sólo paralelice el "for" externo con dos hilos.
2. Elabore una segunda versión que paralelice el segundo "for", utilizando la función "`omp_set_nested(2);`" con 4 hilos .
3. Elabore una tercera versión que paralelice todos los "for" , utilizando la función "`omp_set_nested(3);`" con 8 hilos.
4. Compare el rendimiento de las tres versiones haciendo por lo menos cinco experimentos con cada uno, siempre usando la misma matriz de adyacencia. Use el "gmediano.txt" para la comparación de rendimientos.
5. Se obtiene un mejor rendimiento con la tercera versión? Si no es así, investigue por qué.

**NOTA**:
1. La función omp_set_nested() se invoca en todos los casos UNA SOLA VEZ, antes del "for" más externo.
2. Utilice los archivos "gpequenyo.txt", "gmediano.txt" y "ggrande.txt" para hacer pruebas.