Presentación Práctica 8 – ASD OpenMP / MPI

- Pedro Escobar Rubio
- Alejandro Fernández Trigo

Índice

1 Introducción

2 Descripción de los sistemas

3 Algoritmia

4 Procesamiento paralelo (OpenMP)

5 Procesamiento paralelo (MPI)

6 Conclusiones, bibliografía…

Descripción del hardware

<< Hemos usado tres máquinas con prestaciones bastante diferenciables >>

Máquina 1

Intel Core i7-10750H Cornet Lake

Frecuencia: 4489.02 MHz.

Caché: L1 Data: 6 x 32KB.

L1 Inst.: 6 x 32KB.

Level 2: 6 x 256KB.

Level 3: 12MB.

Memoria: DDR4 16GB.

Máquina 2

Intel Core i7-6500U Skylake-U/Y

Frecuencia: 2990.40 MHz.

Caché: L1 Data: 2 x 32KB.

L1 Inst.: 2 x 32KB.

Level 2: 2 x 256KB.

Level 3: 4MB.

Memoria: DDR4 4GB.

Máquina 3

AMD Ryzen 3 2200G Raven Ridge

Frecuencia: 3500.20 MHz.

Caché: L1 Data: 4 x 32KB.

L1 Inst.: 4 x 64KB.

Level 2: 4 x 512KB.

Level 3: 4MB.

Memoria: DDR4 8GB.

PD: Se usa el compilador provisto por las herramientas para C++ de Visual Studio 2022 (17.1.1 v143) sobre Windows 10 64Bits.

Descripción del algoritmo

El algoritmo devuelve un histograma; en cada columna de este se encuentra el número de apariciones de un número (generado de forma aleatoria).

De esta forma, la matriz tendrá un número de filas igual al número de *tests* (cada test es distinto a los demás), y un número de columnas igual al número de *slices* que tiene el histograma.

Cada número se calcula de forma aleatoria, haciendo:

```
slice = (rand() * current_n_slices) / (RAND_MAX + 1);
```

De esta forma, se determina en que *slice* se encuentra el número aleatorio y se suma uno en dicha posición de la matriz.

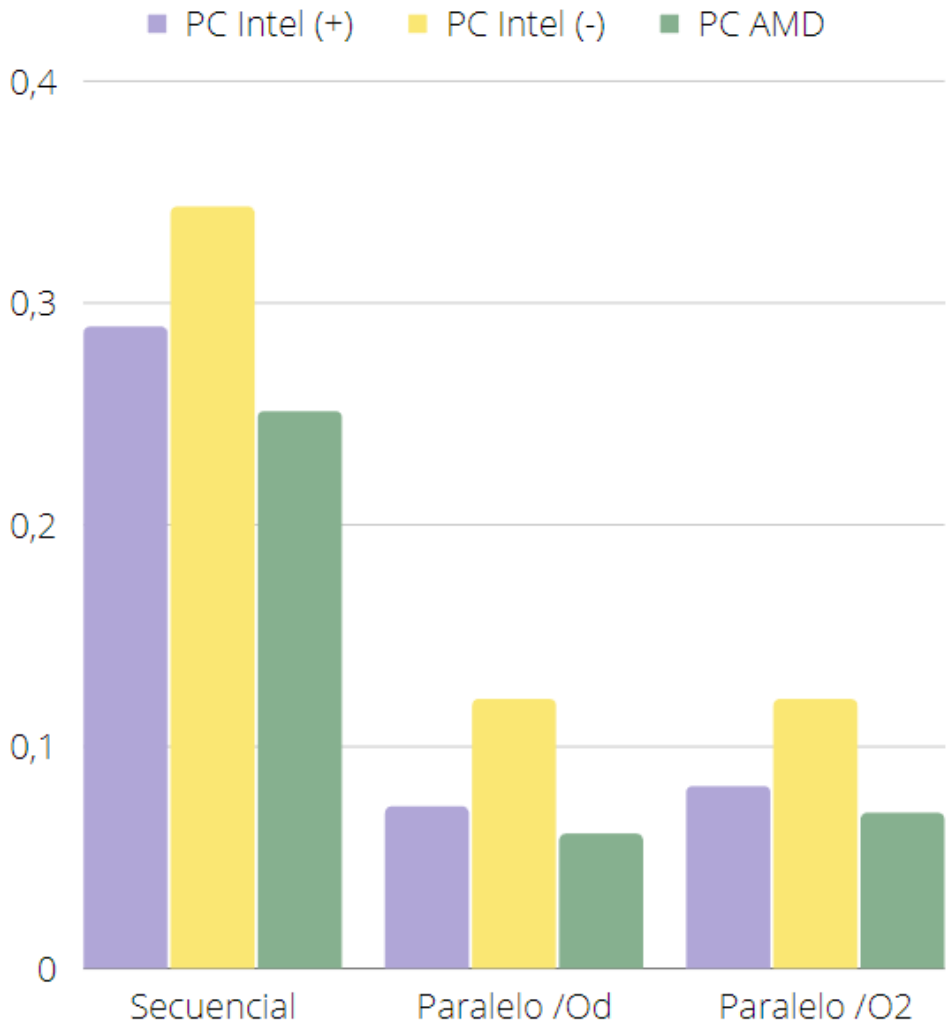
Procesamiento paralelo: OpenMP

```
void  par_histogram(int current_n_slices, int n_iter) {
    int iter, slice, test;

    // int n_threads = omp_get_num_threads();
    // int thread_num = omp_get_thread_num();
    // int n_tests_per_threads = N_TESTS / n_threads;

    #pragma omp parallel for default(none) shared(n_iter, current_n_slices, hist) private(test, iter, slice)
    // #pragma omp parallel for private(test, iter, slice)
    for (test = 0; test < N_TESTS; test++)
    {
        /*
            Cada hilo usara nº aleatorios generados por una semilla distinta.
        */
        srand(seeds[test]);
        for (iter = 0; iter < n_iter; iter++) {
            slice = (rand() * current_n_slices) / (RAND_MAX + 1);
            hist[test][slice] ++;

        }
    }
    // #pragma omp barrier

}
```

Se paraleliza la ejecución de los *tests*.

Cada hilo consulta solo sus propios índices, sin “machacar” los de los demás.

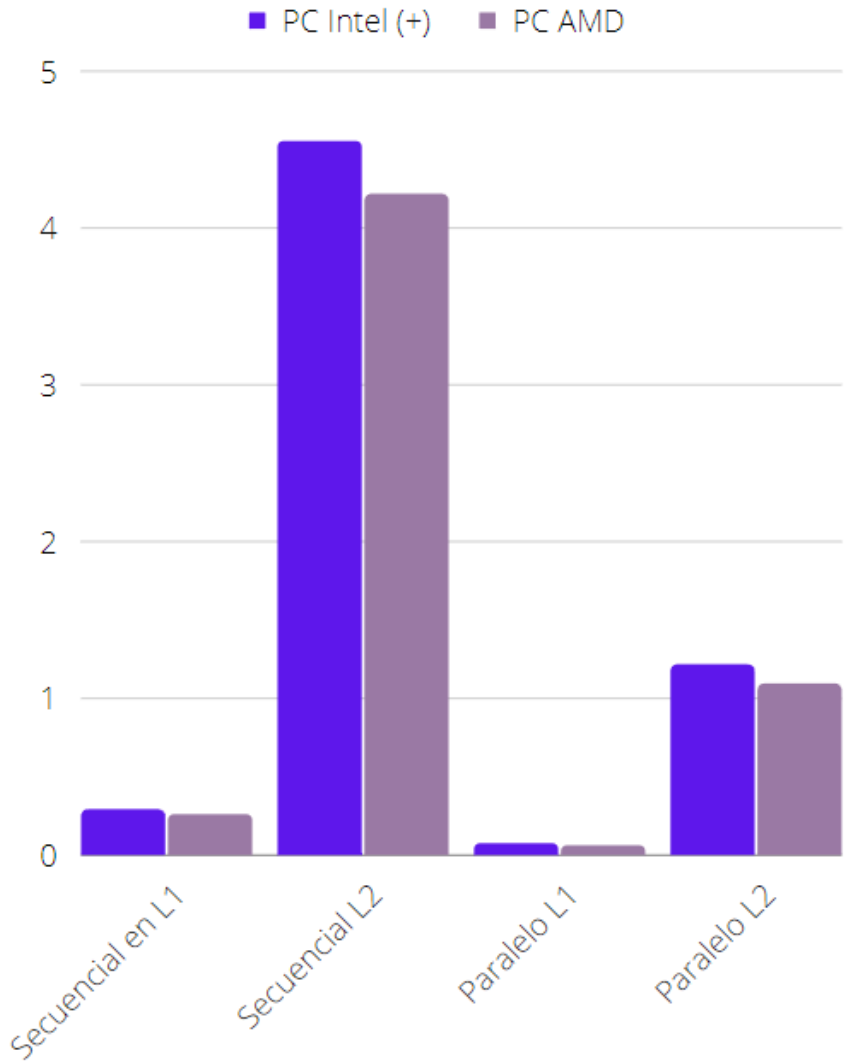
Resultados: OpenMP (I)



<<<<< Comparativa de tiempos medidos para los tres equipos; ¡claras diferencias entre la versión secuencial y las versiones paralelas!

```
# PAR and SEQ standard deviation for the N_SLICES:       64
     0 : 19166.562500: 19166.562500 . Diff: 0.000000
     1 : 21285.906250: 21285.906250 . Diff: 0.000000
     2 : 17729.593750: 17729.593750 . Diff: 0.000000
     3 : 12263.312500: 12263.312500 . Diff: 0.000000
- Size: 64 . SEQ Minimum seconds for all the tests: 0.257221
- Size: 64 . PAR Minimum seconds for all the tests: 0.059207
# PAR and SEQ standard deviation for the N_SLICES:      128
     0 : 9332.406250: 9332.406250 . Diff: 0.000000
     1 : 8480.750000: 8480.750000 . Diff: 0.000000
     2 : 7692.703125: 7692.703125 . Diff: 0.000000
     3 : 6188.781250: 6188.781250 . Diff: 0.000000
- Size: 128 . SEQ Minimum seconds for all the tests: 0.264055
- Size: 128 . PAR Minimum seconds for all the tests: 0.060519
# PAR and SEQ standard deviation for the N_SLICES:      256
     0 : 4468.867188: 4468.867188 . Diff: 0.000000
     1 : 4166.500000: 4166.500000 . Diff: 0.000000
     2 : 3721.632812: 3721.632812 . Diff: 0.000000
     3 : 3379.835938: 3379.835938 . Diff: 0.000000
- Size: 256 . SEQ Minimum seconds for all the tests: 0.254285
- Size: 256 . PAR Minimum seconds for all the tests: 0.073724
# PAR and SEQ standard deviation for the N_SLICES:      512
     0 : 2009.523438: 2009.523438 . Diff: 0.000000
     1 : 1963.546875: 1963.546875 . Diff: 0.000000
     2 : 1941.347656: 1941.347656 . Diff: 0.000000
     3 : 1981.687500: 1981.687500 . Diff: 0.000000
- Size: 512 . SEQ Minimum seconds for all the tests: 0.266824
- Size: 512 . PAR Minimum seconds for all the tests: 0.071670
# PAR and SEQ standard deviation for the N_SLICES:     1024
     0 : 1003.699219: 1003.699219 . Diff: 0.000000
     1 : 953.548828: 953.548828 . Diff: 0.000000
     2 : 968.593750: 968.593750 . Diff: 0.000000
     3 : 974.720703: 974.720703 . Diff: 0.000000
- Size: 1024 . SEQ Minimum seconds for all the tests: 0.250915
- Size: 1024 . PAR Minimum seconds for all the tests: 0.060699
```

Resultados: OpenMP (II)



<<<<< Diferencias de tiempo cuando saturamos los niveles de caché. Medimos los tiempos sólo sobre un equipo ya que los resultados son similares.

- L1 --> 4*32*1024 = 131.072B

  ¿Y si aumentamos el tamaño de la matriz?

  ```
  #define   MAX_N_SLICES ((RAND_MAX+1)/4)
  ```

  16384*4*4 = **262.144B >> 131.072B**

- L2 --> 4*512*1024 = 2.097.152B

  ```
  #define   MAX_N_SLICES ((RAND_MAX+1)*5)
  ```

  Intentar desbordar niveles de caché más bajos provoca un aumento del tiempo de ejecución demasiado grande.

Procesamiento paralelo: MPI

```
ElementType(*h_part)[RANGE];
h_part = (ElementType(*)[RANGE]) mat;
int iter, test;
unsigned long  slice;
int p, my_rank;

// Total de procesos (p) y proceso actual(my_rank):
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
int n_test_per_process = N_TESTS / p;

//@  STUDENTS MUST WRITE HERE THE PARALLEL VERSION
for (test = 0; test < n_test_per_process; test++)
{
    /*
        Cada proceso usara n° aleatorios generados por una semilla distinta.
    */
    srand(seeds[test + n_test_per_process * my_rank]);
    //srand(seeds[test]);
    for (iter = 0; iter < n_iter; iter++) {
        slice = (rand() * RANGE) / ((unsigned long)RAND_MAX + 1);
        h_part[test][slice] ++;
    }
}
//@  STUDENTS MUST WRITE HERE THE PARALLEL VERSION
```

Para que se usen semillas distintas!

0 + 4/4 * 0
0 + 4/4 * 1
….

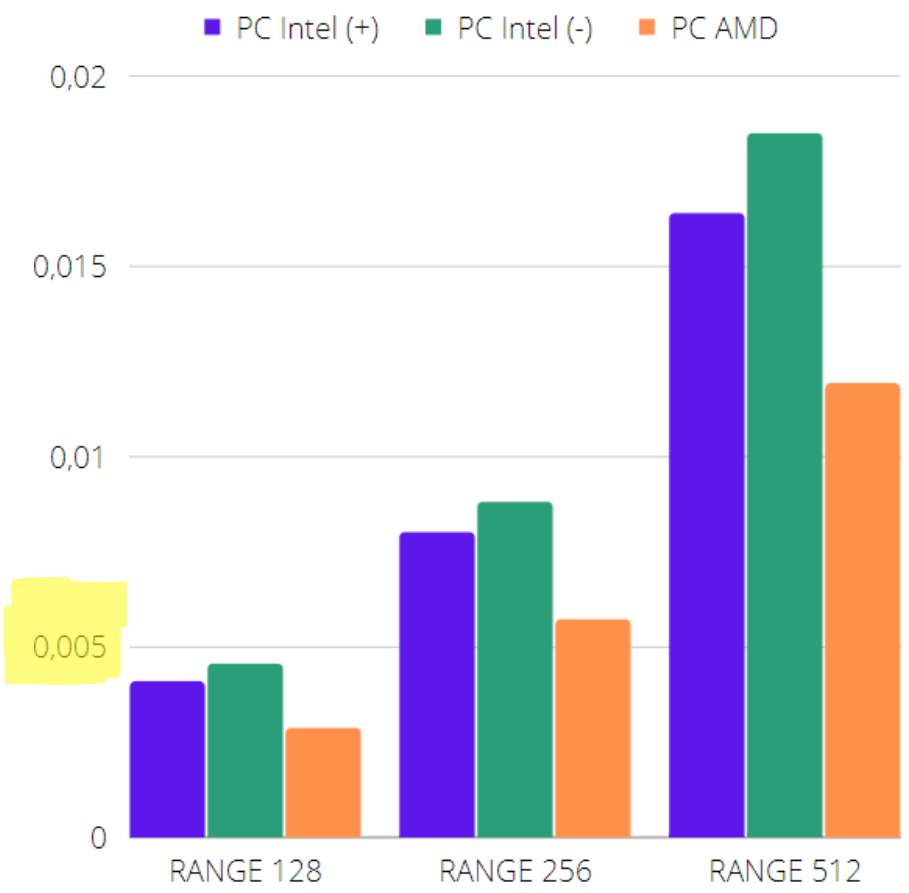
(Para n=4 y N_TESTS=4, n_test_per_process = 1)

Procesamiento paralelo: MPI

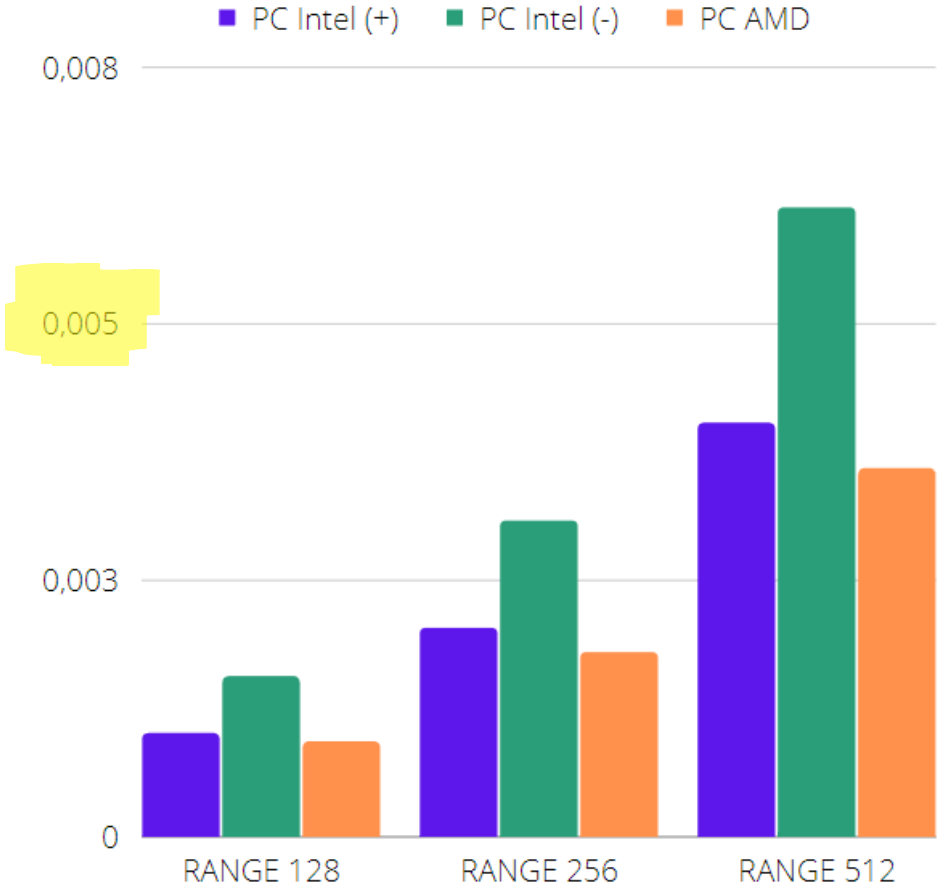
```
//@  STUDENTS MUST WRITE HERE THE MESSAGES TO COLLECT DATA
// Maestro:
if (my_rank == 0) {

    /*
        El maestro (0) guarda todos los datos recibidos de cada proceso esclavo.
    */
    for (int proceso = 1; proceso < p; proceso++)
    {
        MPI_Recv(&hist[n_test_per_process * proceso][0], (n_test_per_process * RANGE), MPI_UNSIGNED_LONG, proceso, 0, MPI_COMM_WORLD, &status);
    }

// Esclavos:
} else {

    /*
        Cada proceso envía su hist_partial al maestro. Todo se envía al 0, esto es, al maestro.
    */
    MPI_Send(&hist_partial[0], (n_test_per_process * RANGE), MPI_UNSIGNED_LONG, 0, 0, MPI_COMM_WORLD);

}
//@  STUDENTS MUST WRITE HERE THE MESSAGES TO COLLECT DATA
```

mpiexec –n 4 ASD-P8-MPI.exe

Resultados: MPI (I)

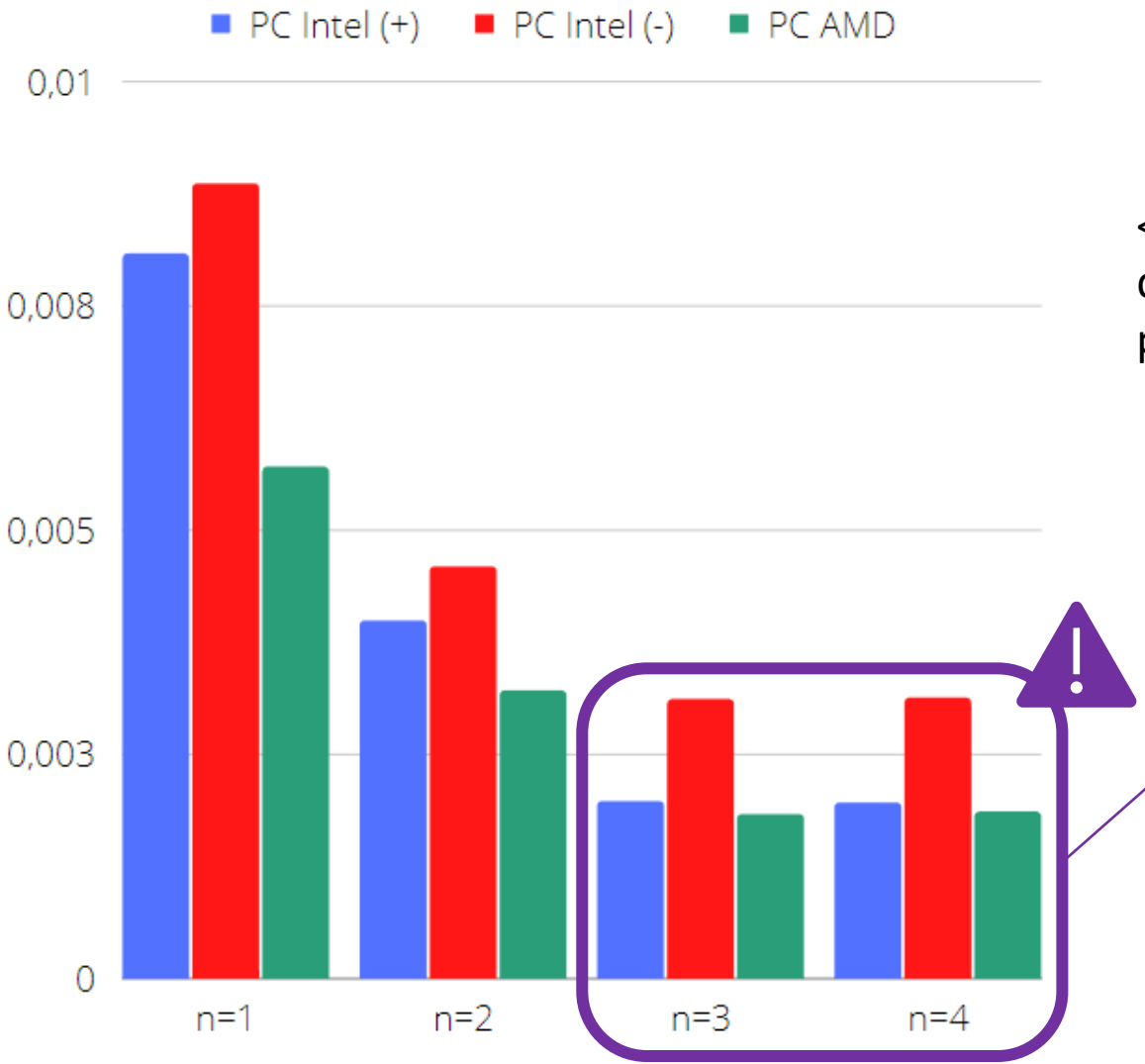


SECUENCIAL



PARALELO n=4

Resultados: MPI (II)

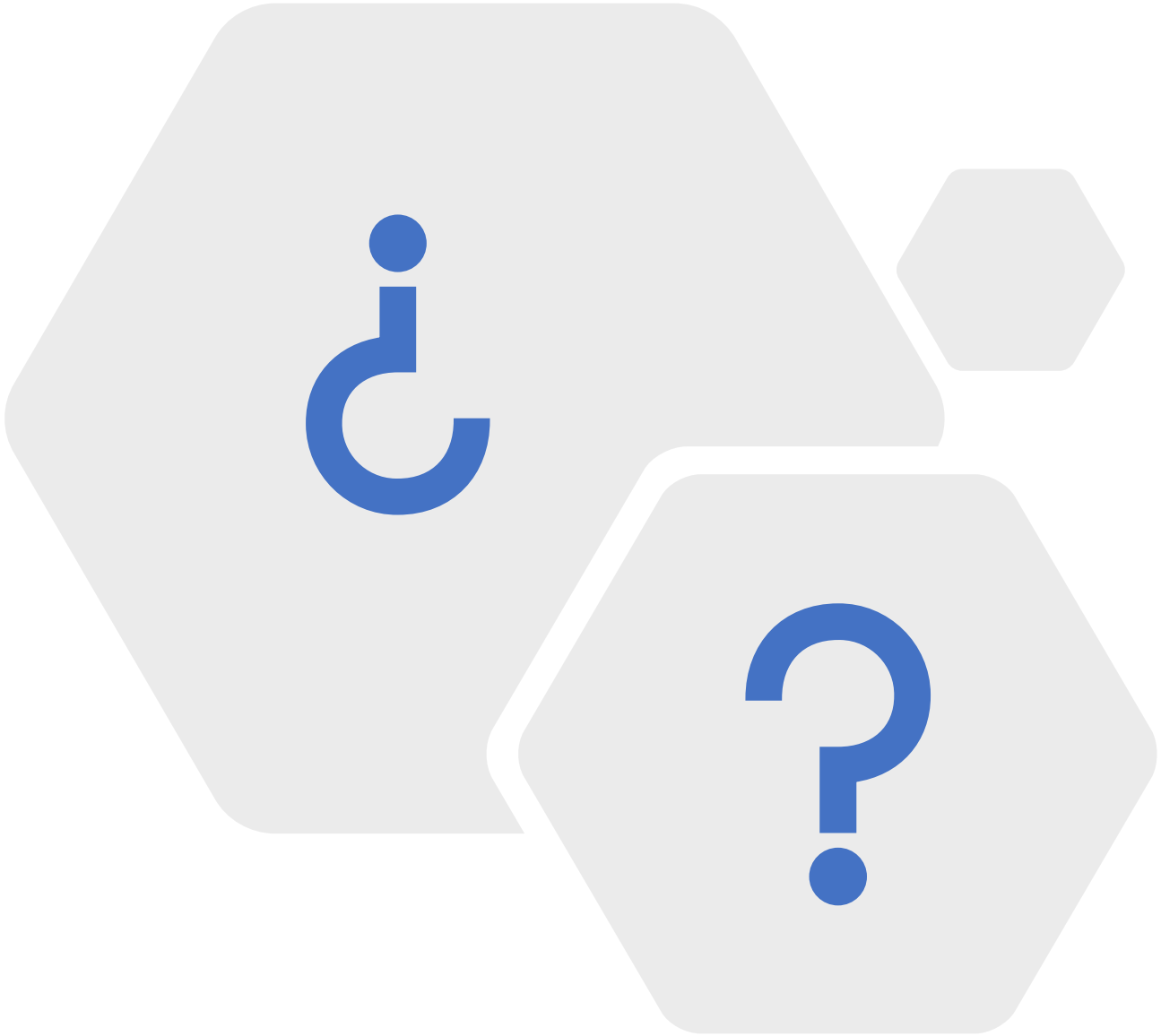


<<<<< Para el tamaño prefijado de RANGE = 256, vemos las diferencias en el rendimiento paralelo para distintos nº de procesos (n).

La secuencia de ejecución para N_TESTS = 4 sería:

4/1 = 4
4/2 = 2
4/3 = 1,333… = 1
4/4 = 1

¡Es igual!



¿Alguna pregunta?

* Copia completa del proyecto/código AQUÍ.