Universidad Nacional San Agustín de Arequipa
Facultad de Ingeniería de Producción y Servicios
Escuela Profesional de Ciencia de la Computación

UNSA
UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

Laboratorio 04

Odd Even Sort

Autor: Juan Manuel Soto Begazo
CUI: 20192267
Docente: Dr. Alvaro Henry Mamani Aliaga
Computación Paralela y Distribuida
Grupo A
Arequipa, Peru

1. Actividades

- Analizar las implementaciones sobre el método de ordenamiento ODD-EVEN SORT usando las dos formas presentadas en el capítulo 5, OPENMP.

La implementación se encuentra en el siguiente repositorio de Git Hub.

2. Desarrollo

2.1. Algortimo de ordenamiento Odd-Even Sort

El algoritmo de ordenación por trasposición Odd-Even Sort (Ordenación por Trasposición Par-Impar) es una variante del algoritmo de burbuja que ofrece mayores posibilidades de paralelización. En cada fase, los elementos en posiciones impares y pares se comparan y, si es necesario, se intercambian. Este algoritmo garantiza que después de un número específico de fases (n), la lista estará ordenada.

En el presente laboratorio, exploraremos dos implementaciones paralelas del algoritmo Odd-Even Sort.

2.2. First OpenMP implementation of odd-even sort

2.2.1. Implementación

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

int thread_count;
const int N = 1e5 + 10;

void Odd_even(int a[], int n) {
   int phase, i, tmp;

   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
         for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
               tmp = a[i-1];
               a[i-1] = a[i];
               a[i] = tmp;
            }
         }
      else
```

```
24  #        pragma omp parallel for num_threads(thread_count) \
25              default(none) shared(a, n) private(i, tmp)
26           for (i = 1; i < n-1; i += 2) {
27              if (a[i] > a[i+1]) {
28                 tmp = a[i+1];
29                 a[i+1] = a[i];
30                 a[i] = tmp;
31              }
32           }
33     }
34  }
35
36  void work(int n_thread, int sz){
37     int  n;
38     int* a;
39     double start, finish;
40
41     thread_count = n_thread;
42     n = sz;
43     a = (int *)malloc(n * sizeof(int));
44     srand(1);
45     for (int i = 0; i < n; i++) a[i] = rand() % 100;
46
47     start = omp_get_wtime();
48     Odd_even(a, n);
49     finish = omp_get_wtime();
50
51     printf("%d %e\n", n, finish - start);
52  }
53
54  int main() {
55     for(int i = 1000; i < N; i += 5000) work(2, i);
56     return 0;
57  }
```

2.2.2. Análisis

Para el análisis de esta implementación, nos enfocaremos exclusivamente en la función `Odd-even`, ya que el resto del código se ocupa únicamente de generar aleatoriamente una lista de tamaño n en la función `work`, mientras que en la función `main`, se realiza la prueba con diferentes tamaños de la lista a ordenar.

Ambas implementaciones de las funciones de ordenamiento aprovechan el hecho de que después de n iteraciones, la lista estará completamente ordenada. Es crucial tener en cuenta que si deseamos paralelizar este proceso, debemos considerar que la iteración $phase+1$ depende de la iteración $phase$. Por lo tanto, la paralelización no puede llevarse a cabo a nivel de fases; en cambio, debe realizarse dentro de cada fase. Esto se debe a que el resultado del trabajo de cada hilo es independiente, aprovechando el hecho de que en una fase solo se ordenan las posiciones pares o impares. Con esto

en mente, procederemos a paralelizar el trabajo dentro de una fase distribuyendo la tarea entre los hilos.

En la primera implementación, se crean hilos para realizar el trabajo dependiendo de si la variable *phase* es par o impar. Después de determinar la paridad de la fase, se crean los hilos con la directiva `#pragma omp parallel for num_threads(thread_count) default(none) shared(a, n) private(i, tmp)`. Esta directiva se encarga de distribuir la carga de trabajo del bucle `for` que se encuentra en la línea 16 en caso de que *phase* sea par, o en el bucle `for` de la línea 26 en caso de que *phase* sea impar.

Es relevante señalar que en cada iteración de la variable *phase*, se crean y destruyen hilos para realizar ese trabajo específico. Esto genera una carga adicional debido al proceso de creación y eliminación. En la segunda implementación, exploraremos cómo mejorar este aspecto.

2.3. Second OpenMP implementation of odd-even sort

2.3.1. Implementación

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <omp.h>
5
6  const int N = 1e5 + 10;
7  int thread_count;
8
9  void Odd_even(int a[], int n) {
10    int phase, i, tmp;
11 #  pragma omp parallel num_threads(thread_count) \
12        default(none) shared(a, n) private(i, tmp, phase)
13    for (phase = 0; phase < n; phase++) {
14       if (phase % 2 == 0)
15 #        pragma omp for
16          for (i = 1; i < n; i += 2) {
17             if (a[i-1] > a[i]) {
18                tmp = a[i-1];
19                a[i-1] = a[i];
20                a[i] = tmp;
21             }
22          }
23       else
24 #        pragma omp for
25          for (i = 1; i < n-1; i += 2) {
26             if (a[i] > a[i+1]) {
27                tmp = a[i+1];
28                a[i+1] = a[i];
29                a[i] = tmp;
30             }
31          }
```

```
32      }
33  }
34
35  void work(int n_thread, int sz){
36      int  n;
37      int* a;
38      double start, finish;
39
40      thread_count = n_thread;
41      n = sz;
42      a = (int *)malloc(n * sizeof(int));
43      srand(1);
44      for (int i = 0; i < n; i++) a[i] = rand() % 100;
45
46      start = omp_get_wtime();
47      Odd_even(a, n);
48      finish = omp_get_wtime();
49
50      printf("%d %e\n", n, finish - start);
51  }
52
53  int main() {
54      for(int i = 1000; i < N; i += 5000) work(2, i);
55      return 0;
56  }
```

2.3.2. Análisis

En esta segunda implementación, los hilos de trabajo se crean al inicio del programa, es decir, se generan una única vez y están disponibles para llevar a cabo tareas hasta el final del programa, momento en el cual son eliminados. En este enfoque, al determinar la paridad de la variable *phase*, se utiliza la directiva `#pragma omp for` para instruir a los hilos que el trabajo en el bucle `for` en la línea 16, en caso de que *phase* sea par, o en la línea 25, en caso de que *phase* sea impar, debe distribuirse entre los hilos disponibles en ese momento.

Esta estrategia nos permite reutilizar los hilos de manera continua, evitando así la pérdida de tiempo asociada a la creación o eliminación constante de hilos. Esta mejora se refleja en la comparación de tiempos, como se ilustra en la Figura 2.1, donde se presenta una comparativa de los tiempos de ejecución para ambos métodos utilizando 2 hilos, y en la Figura 2.2 para 4 hilos.

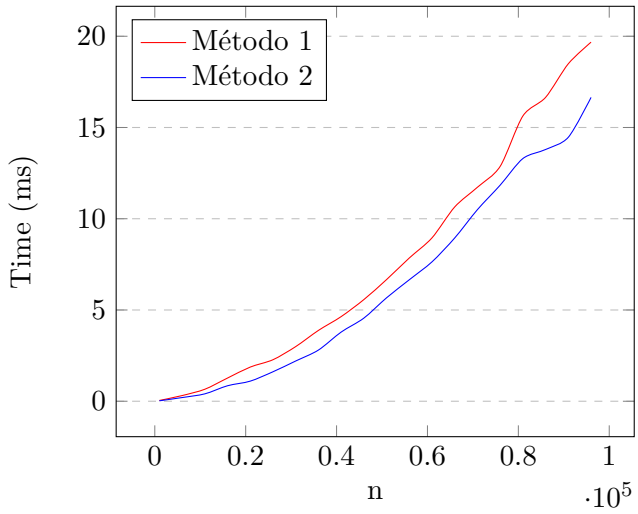

Figura 2.1: Tiempo de ejecución comparando ambos métodos con 2 hilos

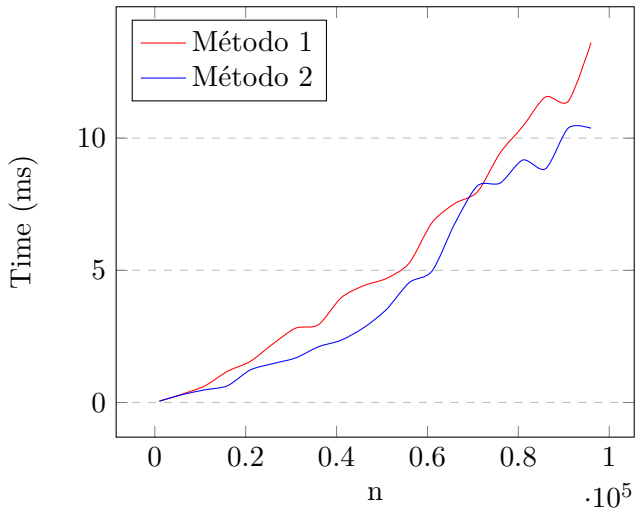

Figura 2.2: Tiempo de ejecución comparando ambos métodos con 4 hilos