

Laboratorio #2

Nombre:	<u>Andrea De Lourdes Lam Pelaez</u>	Carné:	<u>20102</u>
Nombre:	<u>Gabriel Alejandro Vicente Lorenzo</u>	Carné:	<u>20498</u>

Fecha de Entrega: 27 de agosto de 2023.

Descripción: Leer datos de números desde un archivo csv y clasificarlos. Escribir la lista ordenada de números a un segundo archivo. Paralelizar mediante OpenMP usando parallel for, sections, tasks, modificación de scope de variables, políticas de planificación y medición de tiempos. En parejas, realice la creación de un programa secuencial que lee de un archivo, clasifica los datos leídos de menor a mayor y escribe los resultados a un segundo archivo.

Código de referencia en canvas:

- qsort.c
- fileHandler.cpp

Entregables:

- Código fuente (.c /.cpp) de programas finales (secuencial y paralelo)
- Instrucciones de compilación (o bien un makefile)
- Informe escrito donde detalla y lista las modificaciones realizadas al programa secuencial y el porqué de la decisión.

Materiales: necesitará una máquina virtual con Linux.

Contenido:

El programa qsort.c muestra una implementación secuencial de quicksort. Este algoritmo es eficiente y apropiado para manejar una amplia variedad de tipos de datos. Quicksort es del tipo divide and conquer, en el que aplicamos la misma idea a porciones cada vez más reducidas de la data. La idea general es:

1. Elegir cómo dividimos los datos. Usamos un valor fijo p para partir la data y la separamos en valores menores y mayores que el valor $p = a[j]$.
2. La mitad inferior de la data inicia en $a[lo]$ hasta $a[j - 1]$. Toda la data en LO es menor que p . (La diferencia de un valor $LO - p$ es negativa).
3. La mitad superior de la data inicia en $a[j + 1]$ hasta $a[hi]$. Toda la data en HI es mayor que p . (La diferencia de un valor $HI - p$ es positiva).

Una de las partes importantes es la elección del pivote. Podemos elegir el primer número de la lista, el último o el valor central. Una vez elegido el pivote comparamos los valores de la lista de forma secuencial. Vamos comparando los valores de los extremos LO y HI de la data y modificando la posición de búsqueda a la siguiente (LO++ o HI--). Revise el código secuencial en qsort.c y asegúrese de entender el funcionamiento del mismo. Puede usar como referencia visual el link al siguiente:

<https://www.hackerearth.com/practice/algorithms/sorting/quick-sort/visualize/>

Compile el código y corralo con cualquier tamaño de elementos y una tarea: qsort N 1

Manejo de archivos:

En C/C++ podemos realizar la lectura y escritura usando archivos mediante la biblioteca `<fstream>`. Para acceder al archivo, debemos crear un objeto del tipo adecuado: escritura o lectura. Esto crea un flujo o “stream” que nos permite manejar el archivo como una salida más. En C++ podemos redireccionar el I/O mediante la sintaxis de chevrone:

- Output – `cout<<”Texto a desplegar a pantalla”`
- Input – `cin>>variableQueAlmacenaIngresoDesdeTeclado`

Algo similar podemos hacer mediante `fstream`. Una vez creado el objeto correspondiente con sus características propias, podemos manejar el objeto con la sintaxis estándar de C++:

- Output – `archivoSalida<<dataParaEscribir`
- Input – `archivoEntrada>>variableQueAlmacenaDatos`

Use el programa `fileHandler.cpp` como ejemplo de manejo de archivos. Este programa lee desde un archivo separado por comas (CSV), extrae los números primos y los escribe a un archivo.

Ejercicio 1 (50 puntos)

Usando los dos archivos ejemplo, diseñe un programa que escriba N números aleatorios a un archivo. Cada número está separado por coma. Luego lea los números desde el archivo y guárdalos en memoria local. Luego realice la clasificación de los números y almacene en un segundo archivo los números ya clasificados. Como el arreglo puede tener un tamaño variable dependiendo de la cantidad de números aleatorios, debemos usar memoria de Heap en lugar de la memoria de Stack. Para esto utilice la función `new type[N]`: `int * Array = new int[n]`; Esto le permitirá crear un arreglo de cualquier tamaño usando el valor de n cualquiera que se defina antes de la llamada a `new`. En ocasiones cuando `new` no esté disponible, puede hacer lo mismo con `malloc`: `int * Array = (int*) malloc(n*sizeof(int))`; Debe cuidar de limpiar la memoria del Heap al terminar de usarla. Para esto use la función `delete`: `delete Array []`

https://github.com/andreadelou/Lab2_paralela

Ejercicio 2 (50 puntos)

Luego de crear el programa que haga la clasificación de los números desde un archivo, escriba la versión paralela. Recuerde que OpenMP es un programa que nos permite hacer paralelismo de forma incremental. Identifique primero las partes del programa que se beneficiarán con una ejecución paralela y modifique únicamente esa parte. Registre el tiempo de ejecución para compararlo con el de la versión secuencial.

Continúe realizando ajustes al programa y paralelice otras tareas dentro del mismo. Observe si esto logra una mejora significativa respecto a la primera versión. Compare sus programas en medidas de speedup. Realice esto de forma iterativa varias veces buscando cada vez un mejor programa. Anote el speedup que logre encontrar en cada una de sus iteraciones. Finalmente, como parte de los entregables

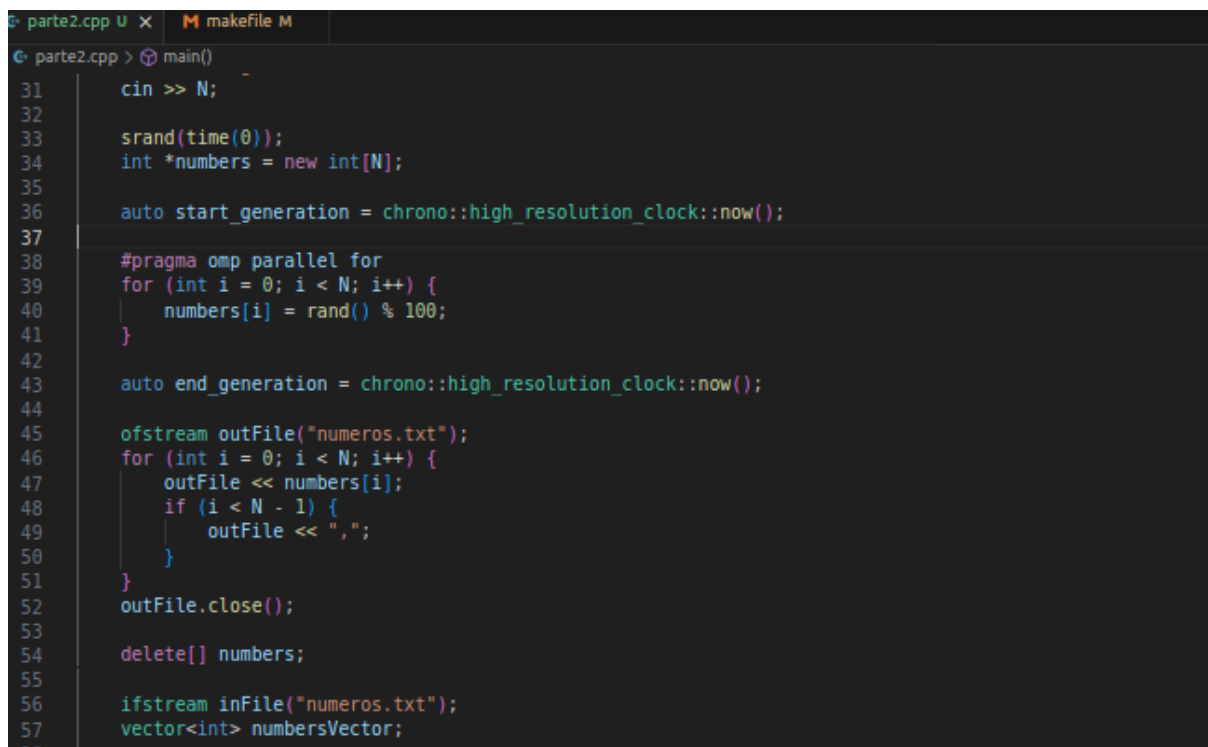
se le solicita un informe en donde liste y detalle cuales son las modificaciones que realizó para la versión final de su programa paralelo. En este informe debe detallar la modificación que hizo y cuál es el razonamiento detrás de dicha modificación.

Informe ejercicio #2

En la segunda fase de nuestro programa, se emprendieron una serie de modificaciones orientadas a la optimización mediante técnicas de paralelización. A través del uso estratégico de directivas pragma, se exploraron diversas prácticas para distribuir la carga de trabajo entre hilos y lograr un rendimiento superior. Tras exhaustivas pruebas y la comparación de tiempos de ejecución se pudo observar que la paralelización de la generación de números aleatorios y el subsiguiente proceso de ordenamiento es una manera óptima de aprovechar la paralelización. Esta estrategia demostró los resultados más prometedores en términos de eficiencia y reducción de tiempos, esto debido a que se obtienen tiempos ridículamente bajos (a pesar de haber ingresado una cantidad considerable de números a ordenar) tendiendo a cero y provocando que la salida en terminal que indica el fin del programa tuviera una respuesta casi inmediata.

A continuación se muestra una captura de pantalla reflejando el cambio en el cual se realiza el cambio paralelizando el for que anteriormente era secuencial:

Captura de pantalla paralelización del for



```

31  cin >> N;
32
33  srand(time(0));
34  int *numbers = new int[N];
35
36  auto start_generation = chrono::high_resolution_clock::now();
37
38  #pragma omp parallel for
39  for (int i = 0; i < N; i++) {
40      numbers[i] = rand() % 100;
41  }
42
43  auto end_generation = chrono::high_resolution_clock::now();
44
45  ofstream outFile("numeros.txt");
46  for (int i = 0; i < N; i++) {
47      outFile << numbers[i];
48      if (i < N - 1) {
49          outFile << ",";
50      }
51  }
52  outFile.close();
53
54  delete[] numbers;
55
56  ifstream inFile("numeros.txt");
57  vector<int> numbersVector;
58

```

Además, se agrega la captura de pantalla en la cual se puede apreciar la paralelización del ordenamiento (para esta mejora se tuvo una discusión para la mejor forma de implementar esta y se trataron de explorar más las opciones que ofrece la librería <omp.h> para estas situaciones).

Captura de pantalla paralelización del ordenamiento

```

1  parte2.cpp x  makefile M
2  parte2.cpp > main()
3      vector<int> numbersVector;
4
5      if (inFile.is open()) {
6          string line;
7          while (getline(inFile, line, ',')) {
8              int number;
9              stringstream(line) >> number;
10             numbersVector.push_back(number);
11         }
12         inFile.close();
13     } else {
14         cout << "No se pudo abrir el archivo." << endl;
15         return 1;
16     }
17
18     auto start_sorting = chrono::high_resolution_clock::now();
19
20     #pragma omp parallel
21     {
22         #pragma omp single nowait
23         {
24             sort(numbersVector.begin(), numbersVector.end());
25         }
26     }
27
28     auto end_sorting = chrono::high_resolution_clock::now();
29
30     ofstream outFileSorted("numeros_ordenados.txt");
31     for (size_t i = 0; i < numbersVector.size(); i++) {
32         outFileSorted << numbersVector[i];
33         if (i < numbersVector.size() - 1) {
34             outFileSorted << ",";
35         }
36     }
37     outFileSorted.close();
38
39     auto duration_generation = chrono::duration_cast<chrono::milliseconds>(end_generation - start_generation);
40     auto duration_sorting = chrono::duration_cast<chrono::milliseconds>(end_sorting - start_sorting);
41
42     cout << "Tiempo de generación: " << duration_generation.count() << " ms" << endl;

```

Cómo se puede observar, estas pequeñas modificaciones, que son viables gracias a librería de omp, dan resultados sumamente relevantes, esto se puede ver en la captura de pantalla final a continuación.

Captura de pantalla de resultados de tiempo

```

1  parte2.cpp x  makefile M
2  parte2.cpp > main()
3      using namespace std;
4
5      bool isPrime(int number) {
6          if (number == 0 || number == 1) {
7              return false;
8          }
9          int divisor;
10         for (divisor = number / 2; number % divisor != 0; --divisor) {
11             ;
12         }
13         if (divisor != 1) {
14             return false;
15         } else {
16             ;
17         }
18     }
19
20     PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
21
22     * gabc@Ubuntu:~/Desktop/Paralela/Laboratorio2/Lab2_paralela$ ./parte2
23     Ingrese la cantidad de números aleatorios: 1912931
24     Tiempo de generación: 137 ms
25     Tiempo de ordenamiento: 331 ms
26
27     * gabc@Ubuntu:~/Desktop/Paralela/Laboratorio2/Lab2_paralela$ ./parte2
28     Ingrese la cantidad de números aleatorios: 13932
29     Tiempo de generación: 1 ms
30     Tiempo de ordenamiento: 2 ms

```

Comparándola con la versión del ejercicio anterior, estos tiempos son sumamente bajos, indicando que se realizaron las modificaciones correctas para las instrucciones de este caso.

La razón por la cual se realizaron estos cambios en específico se puede discutir brevemente pero de una manera justificada. En lo que respecta al for que genera los números aleatorios, se decidió que fue una buena ocasión para utilizarlo ya que la generación de números aleatorios no implica una race condition en ninguna parte del programa. Mientras que utilizarlo para la escritura del archivo en orden sí, ya que implicaría que varios hilos se están peleando por tener acceso al archivo de salida. En lo que respecta al ordenamiento se decidió utilizarlo en esta ocasión ya que el cálculo de sorting respecto al número correspondiente que está analizando es independiente del programa en sí, es decir, su ejecución puede ser llevada a cabo por otro hilo mientras su hilo principal realiza las tareas de preparar el ambiente para los archivos de salida y esperar el resultado.

En conclusión, estas modificaciones al ejercicio de la parte 1, además de incluir una forma de mirar los tiempos de ejecución, son las más óptimas actualizaciones al código secuencial que se pueden haber llevado a cabo debido a los resultados obtenidos y la cantidad de números que se estaban analizando, puede ser que en otro contexto cambie pero para esta implementación de programación paralela los resultados respaldan que es una implementación óptima.