Donaldo Sebastian Garcia Jimenez 19683
Marco Ramírez 19588

UVG	Universidad del Valle de Guatemala Facultad de Ingeniería
	CC3069 Computación Paralela y Distribuida Ciclo 1 de 2023

Laboratorio 2 − OpenMP

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.

Instrucciones

En parejas, realicen 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:

- qsort.c
- fileHandler.cpp

Quicksort

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 sequential 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

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 chevrones:

- 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.

Clasificación secuencial

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 guárdelos 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:

Donaldo Sebastian Garcia Jimenez 19683
Marco Ramírez 19588

```
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 [ ]
```

Clasificación paralela

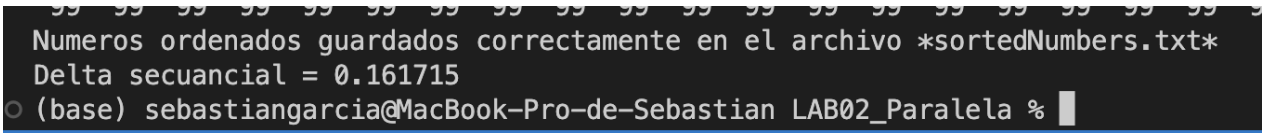
Luego de crear el programa que haga la clasificación de los números desde un archivo, escriba la versión paralela. Recuerde que OpenMP es un programa que nos permite hacer paralelismo de forma incremental. Identifique primero las partes del programa que se beneficiarán con una ejecución paralela y modifique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

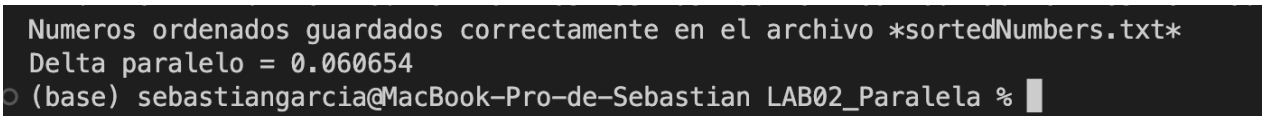
Entrega

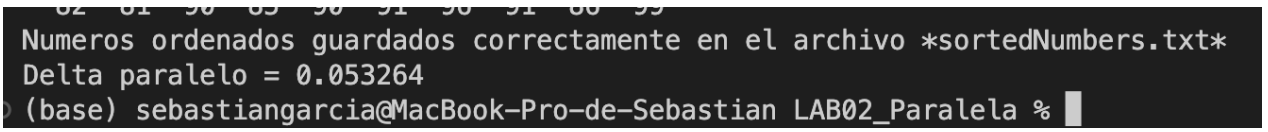
Los códigos secuencial y paralelo, así como un resumen (y capturas) de los resultados de tiempos y speedups en un documento PDF.

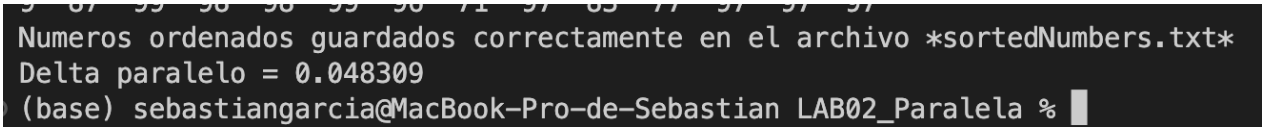
Secuencial



Parallel







Donaldo Sebastian Garcia Jimenez 19683
Marco Ramírez 19588

```
Numeros ordenados guardados correctamente en el archivo *sortedNumbers.txt*
Delta paralelo = 0.042043
(base) sebastiangarcia@MacBook-Pro-de-Sebastian LAB02_Paralela %
```

```
Numeros ordenados guardados correctamente en el archivo *sortedNumbers.txt*
Delta paralelo = 0.039205
(base) sebastiangarcia@MacBook-Pro-de-Sebastian LAB02_Paralela %
```

Código paralelo 1:

```
void quickSort(int array[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(array, low, high);
        #pragma omp task shared(array)
        quickSort(array, low, pi - 1);
        #pragma omp task shared(array)
        quickSort(array, pi + 1, high);
        #pragma omp taskwait
    }
}
```

Código paralelo 2:

Donaldo Sebastian Garcia Jimenez 19683
Marco Ramírez 19588

```
void quickSort(int array[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(array, low, high);
        #pragma omp parallel
        {
            #pragma omp sections
            {
                #pragma omp section
                quickSort(array, low, pi - 1);
                #pragma omp section
                quickSort(array, pi + 1, high);
            }
        }
    }
}
```

Comparación

Secuencial	Paralelo	Speed Up	Efficiency	NumThreads
0,161715	0,060654	2,666188545	0,2666188545	10
0,161715	0,053264	3,036103184	0,3036103184	10
0,161715	0,048309	3,347512886	0,3347512886	10
0,161715	0,042043	3,846419142	0,3846419142	10
0,161715	0,039205	4,124856523	0,4124856523	10

Al momento de hacer el primer bloque de código el paralelo era un poco más lento en su delta, esto es más visible en datos más grandes. Por ejemplo en 1 millón de datos el delta secuencial es 15 y el delta paralelo 5. Sin embargo con el segundo bloque de código paralelo se pudo reducir a 0.4. En la tabla se hizo la prueba con 100,000 datos ya que el secuencial con 1,000,000 tardaba mucho como mencioné anteriormente. Asimismo, se usaron 10 hilos y se logró reducir el tiempo por mucho usando las directivas de section

para lograr paralelismo anidado mejorando potencialmente el rendimiento. En base a lo anterior se logró demostrar la diferencia entre el código paralelo y secuencial, evidenciando las mejoras a nivel de procesamiento de un código secuencial a un código paralelo.