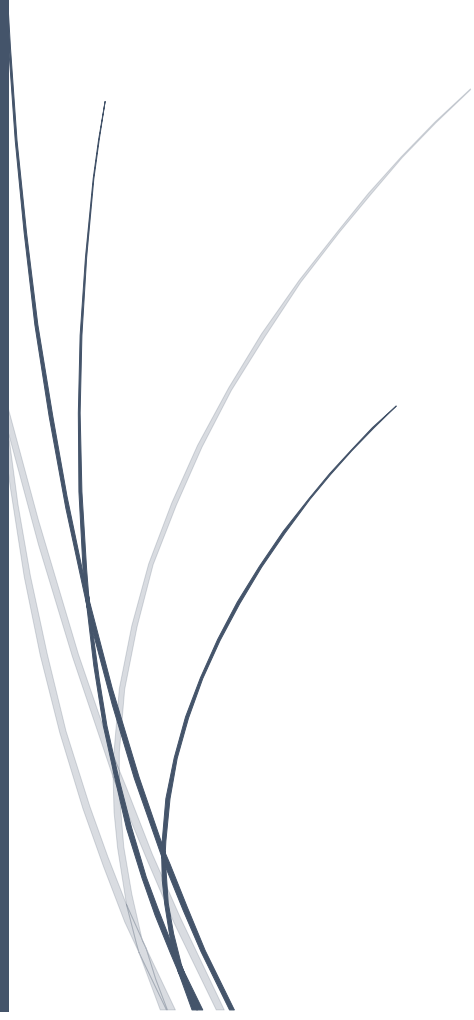
14-12-2017

Trabajo Multihilo

Fase 2



José Antonio García García - UO251317
Cristóbal Solar Fernández - UO220253
Hugo Leite Pérez - UO257671

Contenido

Introducción.

La finalidad de esta fase del proyecto es implementar las versiones del trabajo que se hicieron en un solo hilo, pero esta vez con el numero de hilos que permita la maquina sobre la que se ejecuten. Esto aumentará la eficiencia de los programas y utilizará toda la potencia de la computadora.

Tanto la versión normal como la que contiene instrucciones SIMD funcionan de manera muy similar por lo que se explicarán ambas a la vez.

Includes, variables y signatura de métodos.

```
#include <thread>
```

El único include que es necesario añadir en ambas versiones frente a sus versiones de un solo hilo es este, que permite llamar a los métodos necesarios para el correcto funcionamiento de los hilos.

```
12    // Estructura
13    typedef struct {
14        unsigned int size;  // Numero de posiciones con las que tendra que calcular
15        float* vector1;
16        float* vector2;
17        float* vector3;
18        int entero;         // En funcion sub2: 0=normal, 1=ultimo. En funcion mult: valor del countpos
19        int* resultados;    // Array para guardar los resultados de cada hilo en la funcion countpos
20        int posResultados;  // Posicion que le corresponde del array resultados
21    } datosStruct;
```

Esta estructura será la que se pasará en la creación de hilos como contenedor de parámetros, y tiene los siguientes apartados:

- Size: guardará el tamaño de los vectores con los que tratará cada hilo.
- Vector1, Vector2 y Vector3: serán los punteros a los vectores tanto de entrada como de salida, según se necesiten, que utilizará cada hilo.
- Entero: esta variable tomará valores 0 o 1 según se necesite en la función sub2 para determinar si un hilo va a tratar o no con el final del vector original. Por otro lado, en la función de multiplicación de un vector por un entero, almacenará el valor del entero.
- Resultados: es de uso exclusivo de la función de contar números positivos. Guardará el puntero al array donde cada hilo guardará el resultado que haya calculado.
- PosResultados: también es de uso exclusivo de la función de contar números positivos. Guarda la posición que le corresponde a cada hilo en el array “resultados” para guardar el número de positivos que ha contado.

```
27      double medirTiempos(int);
28      void hacerCalculos(int, datosStruct*, HANDLE*);
29
30      // Metodos para hacer hilos
31      int hilosCountPos(int, datosStruct*, HANDLE*);
32      void hilosSub2(int, datosStruct*, HANDLE*);
33      void hilosMultiplicacion(int, datosStruct*, HANDLE*, int);
34      void hilosAnd(int, datosStruct*, HANDLE*);
35
36      // Metodos para hacer las funciones
37      DWORD WINAPI countPos(LPVOID datos);
38      DWORD WINAPI sub2(LPVOID datos);
39      DWORD WINAPI multiplicacion(LPVOID datos);
40      DWORD WINAPI and (LPVOID datos);
```

La función medirTiempos ahora tiene un parámetro de tipo entero, este parámetro es el numero de hilos que se ha calculado previamente y con el que se trabajará.

La función hacerCalculos pasa a tener como parámetros un array de datosStructs que guardará los parámetros que tendrá cada hilo y un array de HANDLEs que guardará los hilos que se hayan ido creando. Aunque estos dos parámetros se le pasen al método hacerCalculos, los que los utilizarán son los métodos de las operaciones, por ello también recibirán dichos parámetros.

Aparecen 4 métodos nuevos, uno por cada operación, que tienen las palabras reservadas DWORD WINAPI en su signatura. Esto es un requisito para el correcto funcionamiento de los métodos de la librería thread que se incluyo al principio del proyecto. Estos tipos de métodos solo pueden tener un parámetro. Como para nuestro programa necesitamos pasarles un struct, le ponemos de tipo LPVOID, que es lo mismo que un puntero a un tipo void. Usamos LPVOID por comodidad ya que viene de la API de Windows.

Funcionamiento del programa.

```
58  int main() {
59      printf("Aplicacion multithread.\r\n");
60      // Calcular numero de hilos
61      int numThreads = calcularNumeroThreads();
62      printf("Numero de hilos %i\r\n", numThreads);
63      // Reserva el espacio de memoria
64      crearVectores();
65      for (int i = 0; i < REPETICIONES; i++) {
66          printf("El tiempo en la iteracion %i es: %f segundos\r\n", i + 1, medirTiempos(numThreads));
67      }
68      // Liberar el espacio de memoria
69      liberarVectores();
70      printf("Enter para finalizar.\r\n");
71      getchar();
72  }
73
74  int calcularNumeroThreads() {
75      SYSTEM_INFO sysinfo;
76      GetSystemInfo(&sysinfo);
77      return sysinfo.dwNumberOfProcessors;
78  }
```

El único cambio que se realiza en el main es la llamada a la función calcularNumeroThreads que busca el número máximo de hilos que soporta de forma concurrente la maquina sobre la que se va a ejecutar el programa. Este numero se mostrará por pantalla y se pasará a la hora de medir tiempos.

```
80  double medirTiempos(int numThreads) {
81      // Se reserva memoria para guardar un array de datosStruct
82      datosStruct* arrayStructs = (datosStruct*)malloc(sizeof(datosStruct)*numThreads);
83      // Se reserva memoria para guardar un array de HANDLE
84      HANDLE* arrayHilos = (HANDLE*)malloc(sizeof(HANDLE)*numThreads);
85      // Get clock frequency in Hz
86      QueryPerformanceFrequency(&frequency);
87      // Get initial clock count
88      QueryPerformanceCounter(&tStart);
89      // Calculos
90      for (int i = 0; i < NTIMES; i++) {
91          hacerCalculos(numThreads, arrayStructs, arrayHilos);
92      }
93      // Get final clock count
94      QueryPerformanceCounter(&tEnd);
95      // Compute the elapsed time in seconds
96      dElapsedTimeS = (tEnd.QuadPart - tStart.QuadPart) / (double)frequency.QuadPart;
97      // Print the elapsed time
98      /*printf("Elapsed time in seconds: %f\n", dElapsedTimeS);*/
99      // Se libera la memoria reservada
100     free(arrayStructs);
101     free(arrayHilos);
102     // Return the elapsed time if it'll util
103     return dElapsedTimeS;
104 }
105
```

Ahora la función medirTiempos reserva memoria para el array de structs y el de hilos según el numero de threads y se los pasa al método hacerCalculos como parámetros además del numero de hilos. Tras la medición de tiempos se liberará la memoria. Esto mejora considerablemente el rendimiento que si hiciéramos esta reserva de memoria dentro de cada método ya que no es necesario reserva y liberar memoria NTIMES por ejecución, con una vez es suficiente. Se podría mejorar un poco más el tiempo total del programa si se reservara la

memoria en el main, antes incluso de entrar en el bucle REPETICIONES, pero como no afecta a los tiempos medidos se decidió dejarlo como aparece en la imagen.

```
131  void hacerCalculos(int numThreads, datosStruct* arrayStructs, HANDLE* arrayHilos) {
132      // Como las operaciones se realizaran una detras de otra podemos reutilizar el array de structs y el de hilos
133      int countPos = hilosCountPos(numThreads, arrayStructs, arrayHilos);
134      hilosSub2(numThreads, arrayStructs, arrayHilos);
135      hilosMultiplicacion(numThreads, arrayStructs, arrayHilos, countPos);
136      hilosAnd(numThreads, arrayStructs, arrayHilos);
137  }
```

El método hacer cálculos sigue llamando a cada una de las operaciones que llamaba hasta ahora, pero pasándoles el numero de hilos, y los 2 arrays.

Funcionamiento de las operaciones

Todas las operaciones son pequeñas variantes de sus versiones de un solo hilo por lo que se explicará de forma detallada una de ellas y se darán anotaciones sobre las demás.

```
141  int hilosCountPos(int numThreads, datosStruct* arrayStructs, HANDLE* arrayHilos) {
142      // Se reserva memoria para un array donde se alojaran los resultados del countpos
143      int* arrayResultadosCountPos = (int*)malloc(sizeof(int)*numThreads);
144      // Aqui se crean los structs y se llama a la operacion countPos con cada hilo
145      for (int i = 0; i < numThreads; i++) {
146          arrayStructs[i].size = SIZE / numThreads;
147          arrayStructs[i].vector1 = &vectorW[(SIZE / numThreads)*i];
148          arrayStructs[i].resultados = arrayResultadosCountPos;
149          arrayStructs[i].posResultados = i;
150          // Se crea el hilo
151          arrayHilos[i] = CreateThread(NULL, 0, countPos, &arrayStructs[i], 0, NULL);
152      }
153
154      // Aqui se espera a que todos los hilos acaben
155      for (int i = 0; i < numThreads; i++) {
156          WaitForSingleObject(arrayHilos[i], INFINITE);
157      }
158
159      // Aqui se suman todos los resultados recogidos por cada hilo.
160      int resultado = 0;
161      for (int i = 0; i < numThreads; i++) {
162          resultado += arrayResultadosCountPos[i];
163      }
164      free(arrayResultadosCountPos);
165      return resultado;
166  }
```

Todas las funciones de hilos funcionan de forma muy similar por lo que se tomara esta de imagen sacada del código de la versión multithread sin instrucciones SIMD de la operación de contar números positivos.

La principal variación del countPos frente a las demás operaciones es que, como es necesario guardar el resultado final de los números positivos contados por cada hilo para luego utilizar dicho valor, es necesario almacenarlos en algún lugar, es por esta razón que se reserva memoria para un array de enteros del mismo tamaño que hilos se va a necesitar. Es aquí donde se guardarán los resultados.

Para crear los hilos es necesario almacenar el número de elementos con los que deben trabajar, esto se guarda en la variable size de cada uno de los structs dentro del array de structs que se le paso como parámetro al método. Luego se le pasa la dirección del array W que es el que almacena los números aleatorios de los que se contarán los positivos. Pero, como no queremos que todos los hilos trabajen con la misma sección del array, aumentados la dirección que se guarda según el número de hilos que ya hayan sido creados, permitiendo así recorrer el array W de forma individual por cada hilo sin leer dos veces la misma dirección. En la variable resultados se guarda un puntero al array que previamente se había creado para almacenar los resultados de los números positivos y en posResultados la posición que le corresponde a cada hilo dentro de dicho array para escribir la cantidad de números positivos que ha leído. Por último, se crea un hilo con la función de la librería CreateThread a la que se le pasan como parámetros los que se ven en la imagen que son:

- Primero: especifica una serie de atributos de seguridad para los hilos, como no los necesitamos lo ponemos a NULL.

- Segundo: especifica el tamaño en bytes de la cola de hilo, la ponemos a 0 para que tome el valor por defecto.
- Tercero: especifica un puntero a la función que se quiere que ejecute el hilo.
- Cuarto: especifica un puntero al único parámetro que se le pasará a la función que ejecutará el hilo. Es por este cuarto parámetro de la función createThreads que creamos un struct para cada hilo.
- Quinto: especifica los flags de creación del hilo, se pone a 0 para que el hilo se ejecute justo después de su creación.
- Sexto: especifica un puntero a al identificador del hilo. Nosotros solo necesitamos que los hilos se ejecuten, y nos da igual el orden en el que acaben que es para lo que sirve este parámetro, por lo que lo ponemos a NULL para que no nos devuelva ese identificador.

Tras esto se crea el hilo y se guarda en el array de hilos. En este momento este hilo ya se estaría ejecutando.

Se repite este proceso tantas veces como numero de hilos soporte el sistema que es el parámetro numThreads.

En el siguiente for se espera a que todos los hilos acaben para asegurarnos de que se hayan hecho todas las operaciones y los datos con los que trabajemos sean reales y completos.

Ahora que todos los hilos han contado los números positivos que les correspondía, se suman todos los valores y se devuelve un entero con la suma total.

```
167  DWORD WINAPI countPos(LPVOID datos) {
168      // Countpos
169      int resOp1 = 0;
170      datosStruct *structDatos = (datosStruct*)datos;
171      int sizeDatos = structDatos->size;
172      float* vectorLocal = structDatos->vector1;
173      for (int i = 0; i < sizeDatos; i++) {
174          if (vectorLocal[i] > 0) {
175              resOp1++;
176          }
177      }
178      structDatos->resultados[structDatos->posResultados] = resOp1;
179      return 0;
180  }
181
```

Esta es la función que ejecutará cada hilo, lo primero que se hace es un cast del parámetro que se le pasa a la función (datos), al struct que definimos al principio para que sepa el programa que tipos de datos contiene. Luego se mete en variables locales dichos datos para mejorar la legibilidad del programa y se ejecuta de igual forma que se hacía con la versión singlethread. Esto es posible porque hemos guardado la dirección en la que tiene que empezar cada hilo a leer el vector, sino tendríamos que modificar el valor de “i” para cada hilo.

Al final se mete el numero de positivos que ha encontrado este hilo en la posición del array que le corresponda y se devuelve un 0 para demostrar que el hilo ha terminado de forma correcta.

```
167  DWORD WINAPI countPos(LPVOID datos) {
168      // Countpos
169      datosStruct *structDatos = (datosStruct*)datos;
170      int sizeDatos = structDatos->size;
171      float* vectorWLocal = structDatos->vector1;
172      float* vectorSLocal = structDatos->vector2;
173
174      float mask = 0.0;
175      __m256 mask2 = _mm256_set1_ps(mask);
176
177      int *tmp = (int*)vectorSLocal;
178      __m256 *vectorW256 = (__m256*)vectorWLocal;
179      __m256 *vectorS256 = (__m256*)vectorSLocal;
180      for (int i = 0; i < sizeDatos / 8; i++) {
181          vectorS256[i] = _mm256_cmp_ps(vectorW256[i], mask2, _CMP_GT_OQ);
182      }
183      int resOp1 = 0;
184      for (int i = 0; i < sizeDatos; i++) {
185          if (tmp[i] != 0)
186              resOp1++;
187      }
188      structDatos->resultados[structDatos->posResultados] = resOp1;
189      return 0;
190  }
```

Esta es la versión multihilo con instrucciones SIMD. Lo único que cambia frente a la versión singlethread es lo explicado previamente: se hace una cast al struct y se cambian los valores a variables locales para mejorar la facilidad de lectura del programa. Las operaciones son idénticas a las del singlethread con SIMD.

```
205  DWORD WINAPI sub2(LPVOID datos) {
206      // Sub2
207      datosStruct *structDatos = (datosStruct*)datos;
208      int sizeDatos = structDatos->size;
209      int ultimo = structDatos->entero;
210      float* vectorULocal = structDatos->vector1;
211      float* vectorAuxLocal = structDatos->vector2;
212      for (int i = 0; i < sizeDatos; i++) {
213          if (ultimo == 1 && i == SIZE - 1) {
214              vectorAuxLocal[i] = -vectorULocal[i] / 2;
215          }
216          else {
217              vectorAuxLocal[i] = (vectorULocal[i + 1] - vectorULocal[i]) / 2;
218          }
219      }
220      return 0;
221  }
```

La única anotación relevante ocurre en las funciones sub2 de la version normal y la SIMD. Para diferenciar el ultimo hilo de los demás se guarda en la variable entero del struct un 0 o un 1. En caso de ser el ultimo hilo y llegar a la última posición se hace una operación diferente.

```
215  DWORD WINAPI sub2(LPVOID datos) {
216      // Sub2
217      datosStruct *structDatos = (datosStruct*)datos;
218      int sizeDatos = structDatos->size;
219      int ultimo = structDatos->entero;
220      float* vectorTLocal = structDatos->vector1;
221      float* vectorSLocal = structDatos->vector2;
222
223      __m256 *vectorT256 = (__m256*)(vectorTLocal);
224      __m256 *vectorTAux256 = (__m256*)(vectorTLocal + 1);
225      __m256 *vectorS256 = (__m256*)vectorSLocal;
226      __m256 k256 = _mm256_set1_ps(0.5);
227
228      for (int i = 0; i < sizeDatos / 8 - 1; i++) {
229          if (ultimo == 1 && i == sizeDatos - 1) {
230              vectorSLocal[sizeDatos - 1] = 0.0;
231              vectorSLocal[sizeDatos - 2] = (float)((vectorSLocal[sizeDatos - 1] - vectorSLocal[sizeDatos - 2])*0.5);
232              vectorSLocal[sizeDatos - 3] = (float)((vectorSLocal[sizeDatos - 2] - vectorSLocal[sizeDatos - 3])*0.5);
233              vectorSLocal[sizeDatos - 4] = (float)((vectorSLocal[sizeDatos - 3] - vectorSLocal[sizeDatos - 4])*0.5);
234              vectorSLocal[sizeDatos - 5] = (float)((vectorSLocal[sizeDatos - 4] - vectorSLocal[sizeDatos - 5])*0.5);
235              vectorSLocal[sizeDatos - 6] = (float)((vectorSLocal[sizeDatos - 5] - vectorSLocal[sizeDatos - 6])*0.5);
236              vectorSLocal[sizeDatos - 7] = (float)((vectorSLocal[sizeDatos - 6] - vectorSLocal[sizeDatos - 7])*0.5);
237              vectorSLocal[sizeDatos - 8] = (float)((vectorSLocal[sizeDatos - 7] - vectorSLocal[sizeDatos - 8])*0.5);
238          }
239          else {
240              vectorS256[i] = _mm256_sub_ps(vectorT256[i], vectorTAux256[i]);
241              vectorS256[i] = _mm256_mul_ps(vectorS256[i], k256);
242          }
243      }
244
245      return 0;
246  }
```

Pasa lo mismo con la versión SIMD, aunque en este caso la operación es algo más laboriosa.

Tiempos.

En esta imagen se pueden ver los tiempos de todas las versiones del programa; que de izquierda a derecha serian: singlethread, singlethread SIMD, multithread y multithread SIMD. Se aprecia la mejoría en los tiempos de las versiones singlethread frente a las multithread que es de casi la mitad, así como el incremento en el uso de la CPU.

Los tiempos que se midieron están también en el Excel adjunto con las medias y las desviaciones.

Con estos datos podemos apreciar que, para un programa normal y corriente, si podemos aplicar instrucciones de acceso a paquetes de datos y multihilo, la mejoría que obtenemos es muy importante, aunque se complica bastante el código del programa.

Aportación individual

Hugo Leite Pérez: Nada.

Cristóbal Solar Fernández: Revisión de la memoria.

José Antonio García García: Creación e implementación del programa multithread y multithread SIMD, redacción y revisión de la memoria. Creación del Excel y rellenados los datos.