# Visual C++ programación de memoria compartida (OpenMP).

José María Cámara Nebreda, César Represa Pérez, Pedro Luis Sánchez Ortega

***Visual C++programación de memoria compartida (OpenMP)***. 2018

Área de Tecnología Electrónica

Departamento de Ingeniería Electromecánica

Universidad de Burgos

# Índice

# Introducción

La programación paralela se utiliza para resolver problemas computacionales complejos maximizando la utilización de los limitados recursos hardware disponibles. Se han propuesto diferentes paradigmas para ello:

- Memoria compartida (OpenMP)
- Memoria distribuida (MPI)
- Computación heterogénea (CUDA)
- Híbrida: mezcla de al menos dos de los anteriores

Estas alternativas pueden ser consideradas como aproximaciones de "nivel medio". No demasiado alto, de manera que se mantenga el control sobre lo que ocurre "por debajo"; tampoco demasiado bajo de forma que el programador no se vea envuelto en una tarea abrumadora.

La mayoría de las opciones planteadas se presentan como extensiones a lenguajes de alto nivel, habitualmente lenguajes que presentan un buen rendimiento en términos de eficiencia computacional. Uno de ellos es C/C++.

Las aplicaciones que hacen uso de estas técnicas de paralelización quedan habitualmente ocultas para los usuarios habituales e incluso son desconocidas para algunos programadores. A pesar de ello, en los últimos años, el hardware paralelo ha ido quedando disponible para todos los usuarios de manera que cobra sentido hacer aprovechamiento de él dentro de una misma aplicación. Generar código paralelo a un nivel más bajo resulta, además de tedioso, sujeto a la comisión de errores en forma de condiciones de Carrera, abrazos mortales, etc.

En este pequeño manual se trata de proporcionar un ejemplo de cómo los paradigmas de programación paralela pueden ser incluidos en la mayoría de las aplicaciones de una forma sencilla pero que al mismo tiempo produzca un considerable incremento en el rendimiento global del sistema. En este caso concreto vamos a tratar de explicar cómo introducir secciones de código de memoria compartida (OpenMP) dentro de código C++. La herramienta de desarrollo que emplearemos será MS Visual Studio 2017 Community, juntamente con plantillas Windows Forms para minimizar el esfuerzo. El objetivo es ayudar, tanto a quienes, acostumbrados a crear aplicaciones de escritorio, quieran introducir código paralelo en ellas, como a quienes estén habituados a la programación paralela pero no tengan claro cómo integrarla en una aplicación de escritorio con una interfaz de usuario amigable.

# Capítulo1: Ajustes de la solución

Antes de abordar la creación de aplicaciones multihilo, va a ser necesario realizar algunos ajustes:

1. Crear un proyecto Visual C++ utilizando MS Visual Studio 2017 (File->New->Project). Si la opción de utilización de plantilla de Windows Forms no aparece, probablemente será necesario instalar el paquete resaltado en la Figura 1.

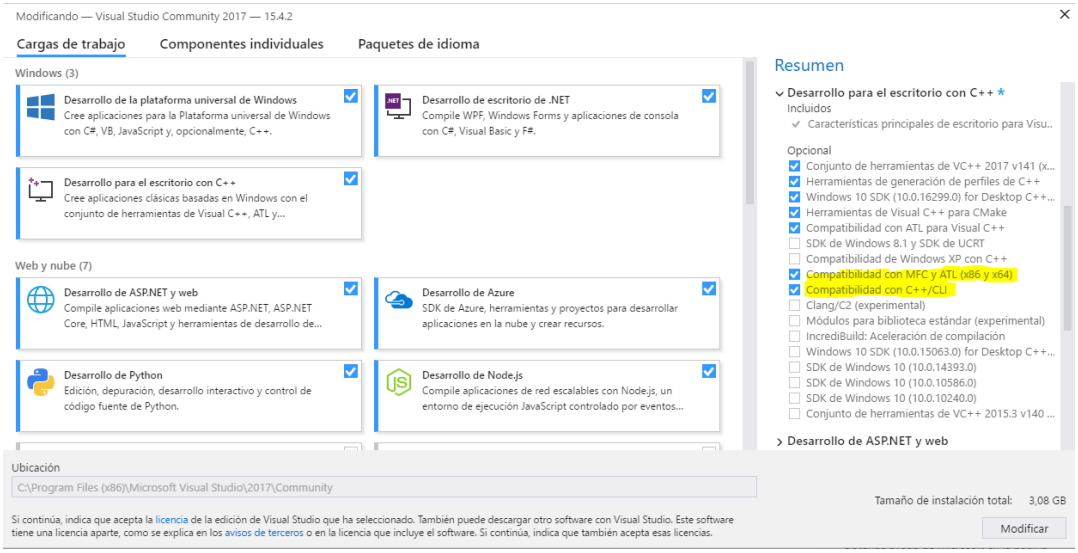



Figura 1

2. Encontrar la plantilla de Windows Forms como se muestra en la Figura 2.

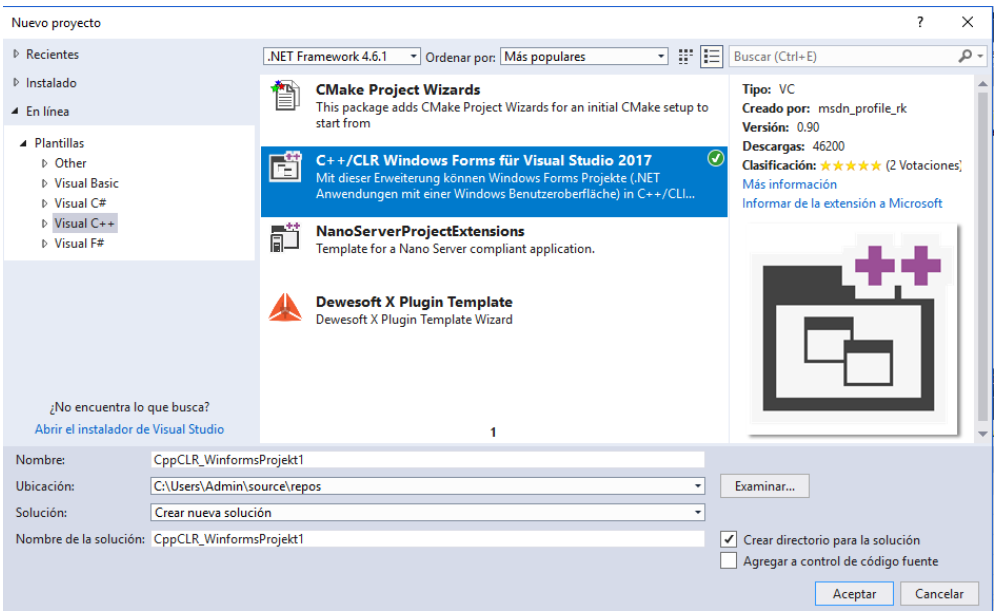



Figura 2

3. En Project->Settings habilitar la compatibilidad con OpenMP como se muestra en la Figura 3.

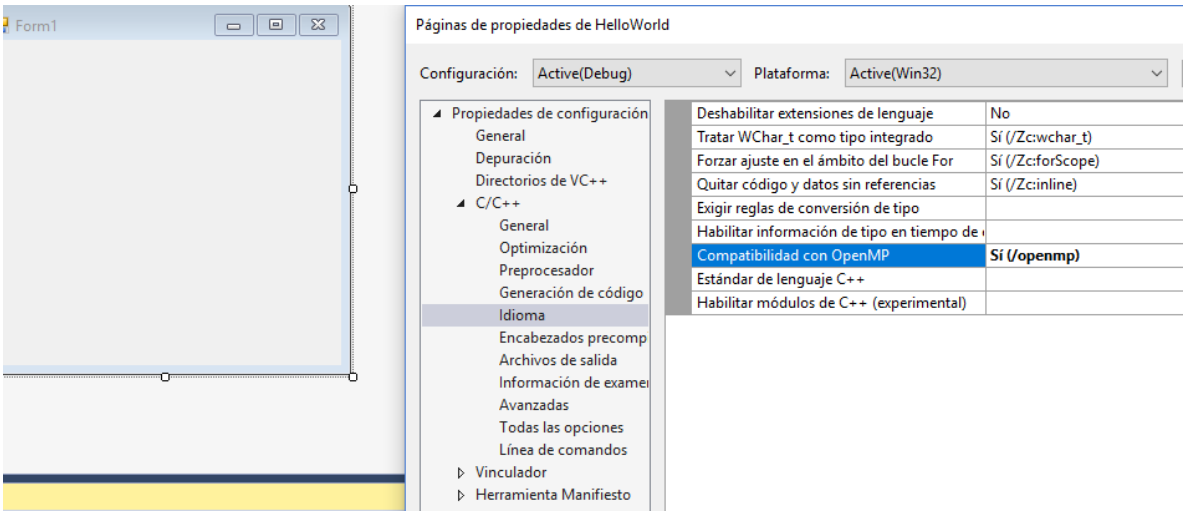



Figura 3

4. Como vamos a utilizar algunos de los controles disponibles en la caja de herramientas, pulsar Ctl+Alt+X para acceder a ella. Los controles serán visibles solo cuando se selecciona la Ventana de diseño.

Ahora ya estamos preparados para crear nuestro primer programa. Vamos a abordarlo en el siguiente capítulo.

# Capítulo2: Un primer programa multihilo

En este capítulo veremos cómo construir el programa multihilo más simple. Lo único que hará será contar el número de hilos lanzados por el Usuario.

En primer lugar localizamos en la caja de herramientas los controles que vamos a necesitar:

| | |
|---|---|
| Label | Label |
| Text box | TextBox |
| Combo box | ComboBox |
| Button | Button |

Los arrastramos al lienzo de Form 1 como se ve en la Figura 4.

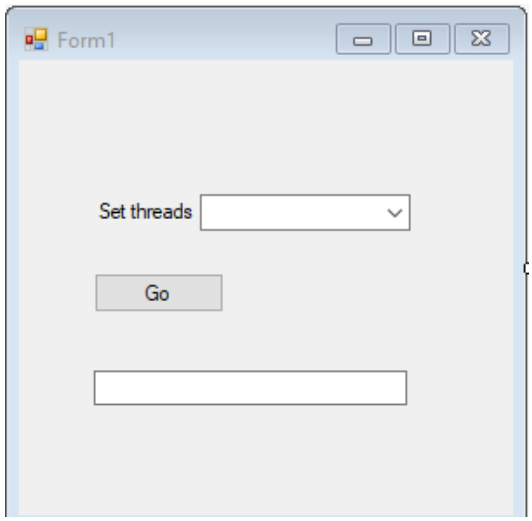



Figura 4

Los controles adquieren un nombre por defecto y, aquellos que incluyen texto pueden presentar también un mensaje por defecto. Por ejemplo, el nuevo botón insertado adquiere el nombre "button 1" y este mismo texto. En la vista de diseño se pueden editar sus propiedades simplemente haciendo click sobre el control y acudiendo a la ventana que se abre en la esquina inferior derecha. Se pueden cambiar los valores por defecto si se desea, por ejemplo, cambiando la leyenda del botón a "Go" como se muestra en la Figura 5. Quedará tal y como aparece en la Figura 4.

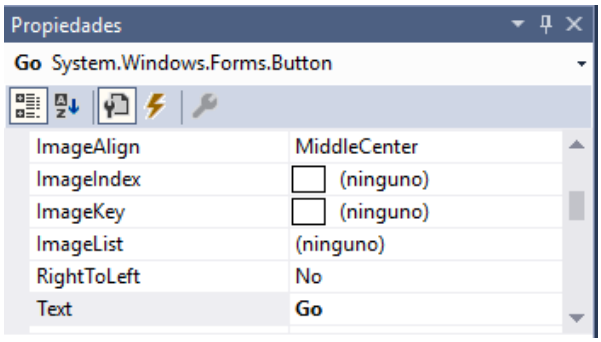



Figura 5

Lo mismo se puede hacer con la etiqueta 1.

El control "combo box" funciona de manera distinta. Se supone que contiene una lista de opciones para que el usuario pueda seleccionar la que desea. La Figura 6 muestra la lista de propiedades para este control y la denominada "Items" entre ellas. Seleccionando la colección se pueden introducir tantas como se desee; una por línea.

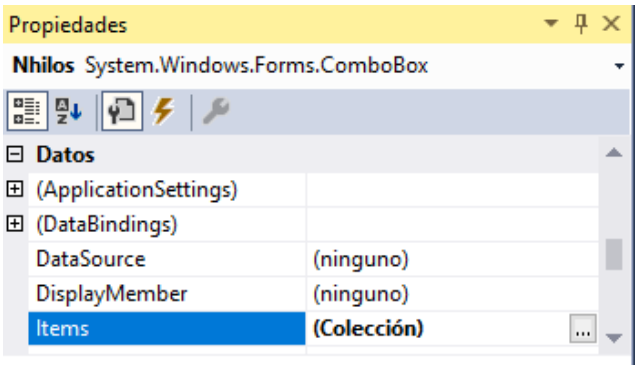



Figura 6

Este control lo vamos a utilizar para seleccionar el número de hilos. Lo que suele tener sentido en este caso es lanzar como máximo tantos hilos como núcleos tenga el procesador por lo que será razonable que la lista incluya ese número de posibilidades.

Ahora tenemos la interfaz de usuario, pero por el momento, no hace nada; vamos a introducir el código que deberá ejecutar. Hemos visto cómo modificar propiedades desde la Ventana de Diseño; esto se puede realizar también por código o incluso en tiempo de ejecución. Para pasar a la Ventana de se puede hacer click derecho sobre Form1.h en el explorador de soluciones o pulsar "F7".

Además de propiedades, los controles también tienen eventos asociados. En la misma ventana de propiedades se puede pulsar el icono para acceder a la lista de eventos disponibles para ese control.

Pero antes de programar acciones, es necesarios realizar algunos ajustes previos.

1. Incluir, al principio de form1.h, el fichero de cabecera de OpenMP:

```
#pragma once
#include "omp.h"
```

2. Declarar las variables globales. En este ejemplo necesitamos solamente dos. Un entero que representa el número de hilos que se van a lanzar, y un puntero a string para el mensaje que se va a construir:

```
private:
		int nThreads;
		String^ message;
		/// <summary>
		/// Erforderliche Designervariable.
```

3. Inicializar las variables globales. Lanzaremos un solo hilo por defecto:

```
public:
		Form1(void)
		{
			InitializeComponent();
			nThreads = 1;
		}
```

Vamos a abordar ahora la cuestión del código. En este ejemplo simple solamente se van a programa dos métodos. El más sencillo es el que permite al Usuario seleccionar, mediante una combo box, el número de hilos a lanzar. Seleccionamos la pestaña de eventos de la combo box obteniendo lo que vemos en la Figura 7 .

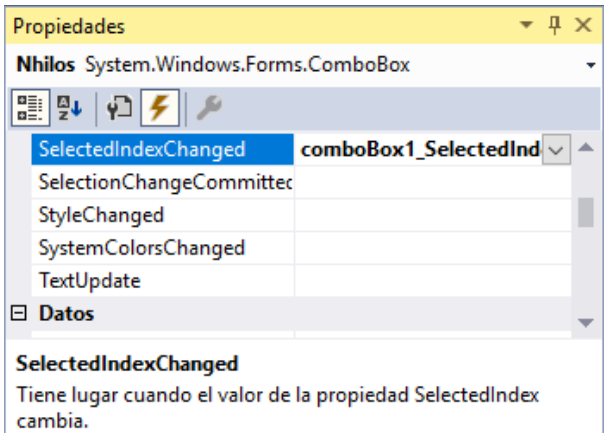



Figura 7

Si acudimos a la Ventana de Código, encontraremos el método "comboBox1_SelecdIndexChanged" ya preprogramado. Nada más hay que introducir, entre las llaves, el Código correspondiente a la acción a emprender. La acción consistirá en actualizar el número de hilos:

```
private: System::Void comboBox1_SelectedIndexChanged(System::Object^  sender,
System::EventArgs^  e) {
            nThreads = int::Parse(Nhilos->Text);
      }
```

La segunda acción es n poco más compleja y constituye el corazón del programa, ya que incluye el Código paralelo. Seleccionar el botón "Go" para acceder a su pestaña de eventos. Entre ellos, seleccionar el click del ratón. En el menú desplegable se puede seleccionar el método "Go_click" (Figura 8).

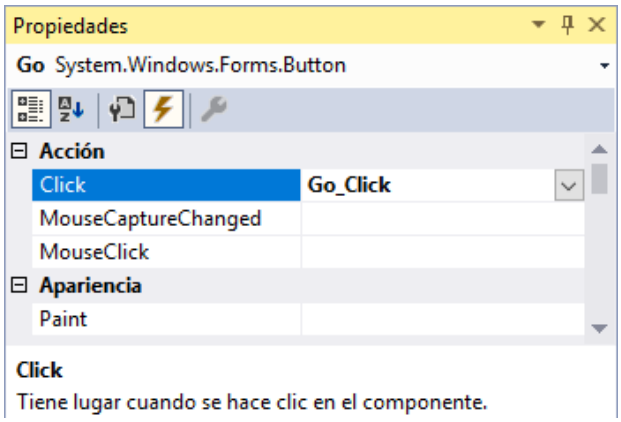



Figura 8

Si acudimos a la Ventana de código, encontraremos el método "Go_click" preprogramado. De nuevo, introduciremos el Código entre las llaves:

```
private: System::Void Go_Click(System::Object^  sender, System::EventArgs^  e) {
      int sum=0;
#pragma omp parallel num_threads(nThreads)
      {
      #pragma omp parallel reduction(+:sum)
            sum = 1;
      }
message = String::Concat("Hello World from nthreads ", Convert::ToString(sum));
textBox1->Text = message;
}
```

¿Qué significa?

- La variable "sum" almacena el número de hilos. Ya tenemos este número en nThreads; ahora los vamos a calculary de nuevo, pero en paralelo.
- `#pragma omp parallel` declara que lo que se encuentra entre las llaves se va a ejecutar en paralelo. Tantos hilos como indique "nThreads" se van a lanzar en paralelo. El Código es el mismo para todos ellos y las variables declaradas fuera de esta "Región paralela" son compartidas por defecto.
- `#pragma omp parallel reduction(+:sum)` inicia una operación de reducción, lo que significa que se va a realizar una determinada operación sobre el conjunto de valores que los diferentes hilos adjudican a una variable. La operación en este caso es una suma y la

variable tiene el mismo nombre casualmente. Como todos los hilos adjudican el valor “1” a “sum”, y todos los valores se suman, el resultado debe ser igual al número de hilos.

- Para comprobar que es correcto se imprime un mensaje en la caja de texto.

En favor de la claridad del ejemplo, lo hemos resuelto haciendo un uso horrible de los hilos y las variables. Si lo ejecutamos, veremos un mensaje de saludo de parte de todos los hilos que el usuario ha decidido lanzar.

No se debe pretender que los hilos impriman los mensajes por sí mismo. Esto, además de complicado, resulta innecesario. Es mejor pensar en las regiones paralelas como secciones de código en las que se realizan tareas complejas y mantener la interfaz de usuario en un solo hilo.

# Capítulo 3: Background workers

En el capítulo anterior veíamos cómo construir una aplicación paralela. La estructura es válida para la mayor parte de aplicaciones que necesitemos construir, pero tiene un problema. El procesamiento en paralelo tiene sentido cuando hay que realizar multitud de operaciones. En tal situación la solución anterior funciona, pero mientras el programa está realizando los cálculos, la interfaz de Usuario va a quedar "congelada".

Así que funciona, pero no tiene sentido, ya que hemos decidido construir una aplicación de escritorio por algún motive. De lo contrario podríamos recurrir a la típica aplicación de consola para problemas computacionales complejos.

¿Cómo mantener la interfaz de Usuario active mientras se lleva a cabo cálculo intensivo en todos los núcleos disponibles? Aquí es donde aparecen los "background workers". Se trata de hilos independientes que se ejecutan en background, de manera que no interfieren con el hilo principal: la interfaz de usuario.

El background worker se encuentra en la caja de herramientas: BackgroundWorker

Cuando se arrastra a la ventana de diseño él se mueve inmediatamente hacia abajo, ya que no es una parte visible de la interfaz (Figura 9).

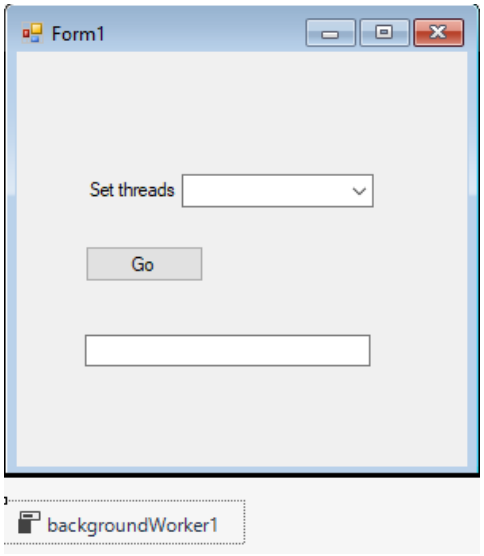



Figura 9

Haciendo doble click en el icono Podemos acceder al Código asociado. Aparece un método por defecto llamado backgroundWorker1_DoWork que permite programar lo que el hilo tiene que hacer. El método Go_Click se vuelve extremadamente simple; simplemente arranca el hilo en backgroud:

```
private: System::Void Go_Click(System::Object^  sender, System::EventArgs^  e) {
             backgroundWorker1->RunWorkerAsync();
         }
```

Todo el trabajo lo realiza el método DoWork:

```
private: System::Void backgroundWorker1_DoWork(System::Object^  sender,
System::ComponentModel::DoWorkEventArgs^  e) {
	int sum = 0;
#pragma omp parallel num_threads(nThreads)
	{
#pragma omp parallel reduction(+:sum)
		sum = 1;
	}
message = String::Concat("Hello World from nthreads ", Convert::ToString(sum));
	textBox1->Text = message;
}
```

⚠ ¡Hay un problema! Si se intenta depurar el nuevo código, se va a producir una excepción. El motivo: el control textBox1 es llamado por un hilo distinto al que lo creó. Esto es inseguro y puede generar muchos problemas. Tendremos que implementar un acceso seguro a este control.

Vamos a crear un nuevo método que sirva de interfaz segura con textBox1. Debajo vemos ambos juntos, de manera que es sencillo entender su funcionamiento conjunto:

```
private: System::Void backgroundWorker1_DoWork(System::Object^  sender,
System::ComponentModel::DoWorkEventArgs^  e) {
	int sum = 0;
#pragma omp parallel num_threads(nThreads)
	{
#pragma omp parallel reduction(+:sum)
		sum = 1;
	}
	message = String::Concat("Hello World from nthreads ",
Convert::ToString(sum));
	SetText(message);
}

private: void SetText(String^ texto){
	if (this->textBox1->InvokeRequired) {
		SetTextDelegate^ d = gcnew  SetTextDelegate(this, &Form1::SetText);
		this->Invoke(d, gcnew array<Object^> {texto});
	}
	else{
		this->textBox1->Text = texto;
	}
}
```

El método “SetText” se debe utilizar para acceder a textBox1 de forma segura, tanto desde el hilo en background como desde el hilo principal. Él decidirá cuándo invocar al delegado.

## Capítulo 4: multiplicación de matrices

Hasta ahora hemos aprendido cómo preparar el entorno para una aplicación paralela pero no hemos incluido aún ninguna paralelización. Necesitamos un problema de cálculo intensivo, y uno de los más habituales es la multiplicación de matrices.

La multiplicación de matrices es una operación matemática por todos conocida (vamos a multiplicar matrices cuadradas por sencillez). Esto está bien para empezar, pero es que además, la multiplicación de matrices presenta una gran escalabilidad, es fácil de programar y puede ser utilizada en muchas aplicaciones más complejas.

¿Qué necesitamos para multiplicar matrices? En primer lugar, las matrices. Se trata de arrays bidimensionales de números en coma flotante (pueden ser de otro tipo, pero los flotantes están bien). Vamos a declarar las nuevas variables a usar en el programa:

```
private:
	int nThreads;
	String^ message;
	int rows;
	float** matrixA;
	float** matrixB;
	float** matrixR;
```

Junto con las tres matrices R = (A x B), hemos declarado también una variable entera para representar el número de lilas (también columnas) de cada una. Tenemos que asignarle un valor por defecto:

```
public:
		Form1(void)
		{
			InitializeComponent();
			nThreads = 1;
			rows = 4;
		}
```

Matrices de este tamaño (4x4) son muy pequeñas para nuestras necesidades de cálculo intensivo, pero le daremos al Usuario la posibilidad de modificar su tamaño. Para ello vamos a necesitar una segunda etiqueta, así como otra combo box. A la etiqueta le daremos el valor “Size” y proporcionaremos una colección de valores a la combo box entre 5 y 8000 con los valores intermedios que se desee.

```
private: System::Void comboBox1_SelectedIndexChanged_1(System::Object^  sender,
System::EventArgs^  e) {
	rows = int::Parse(comboBox1->Text);
}
```

Si queremos visualizar el contenido de las matrices, necesitaremos añadir unas cajas de texto para que el programa lo escriba. Les pondremos las correspondientes etiquetas Las cajas de texto son

de una sola línea por defecto, así que tendremos que ajustar su propiedad “multiline” a “true” e incrementar apreciablemente su tamaño (Figura 10).

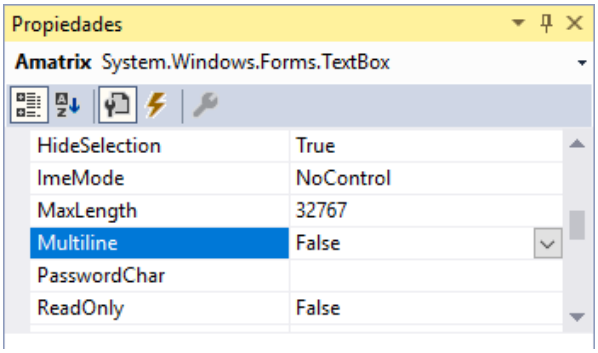



Figura 10

Una vez que tenemos una caja de texto con su etiqueta Podemos copiarla y pegarla para crear las otras dos más fácilmente.

Vamos a añadir otro botón llamado “Initialize”. Un click en él arrancará dos acciones:

1. Asignar espacio para las matrices.
2. Rellenarlo con valores numéricos.

Debería ser algo como lo siguiente:

```
private: System::Void Initialize_Click(System::Object^  sender, System::EventArgs^
e) {
	matrixA = new float*[rows];
	matrixB = new float*[rows];
	matrixR = new float*[rows];
	for (int i = 0; i < rows; i++)
		matrixA[i] = new float[rows];

	for (int i = 0; i < rows; i++)
		matrixB[i] = new float[rows];

	for (int i = 0; i < rows; i++)
		matrixR[i] = new float[rows];

	for (int i = 0; i < rows; i++)
		for (int j = 0; j < rows; j++) {
			matrixA[i][j] = matrixB[i][j] = i + j;
			matrixR[i][j] = 0;
		}
	Amatrix->ResetText();
	Bmatrix->ResetText();
	Rmatrix->ResetText();
	if (rows<20) {
		for (int i = 0; i < rows; i++) {
			for (int j = 0; j < rows; j++) {
	Amatrix->AppendText(String::Concat(Convert::ToString(matrixA[i][j]), " "));
	Bmatrix->AppendText(String::Concat(Convert::ToString(matrixB[i][j]), " "));
	Rmatrix->AppendText(String::Concat(Convert::ToString(matrixR[i][j]), " "));
```

```
                    }
                    Amatrix->AppendText("\n");
                    Bmatrix->AppendText("\n");
                    Rmatrix->AppendText("\n");
                }
            }
}
```

Con todo ello, la interfaz de Usuario debería parecerse a la Figura 11.

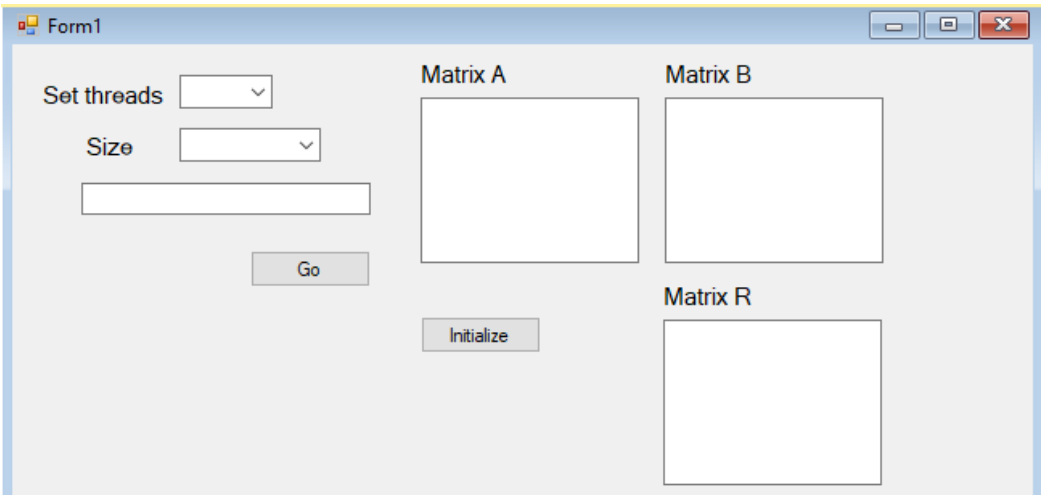



Figura 11

Hemos añadido la condición de que las matrices tengan un tamaño menor de 20 para que su contenido se muestre. De otro modo no veríamos nada claro en pantalla. Los valores numéricos se asignan según las coordenadas de cada elemento de la matriz. Es simplemente una forma sencilla de inicializarlos y luego comprobar que los cálculos son correctos.

Podemos probar con una matriz 5x5 para comprobar que todo está correcto. Tendríamos que ver lo que se muestra en la Figura 12.

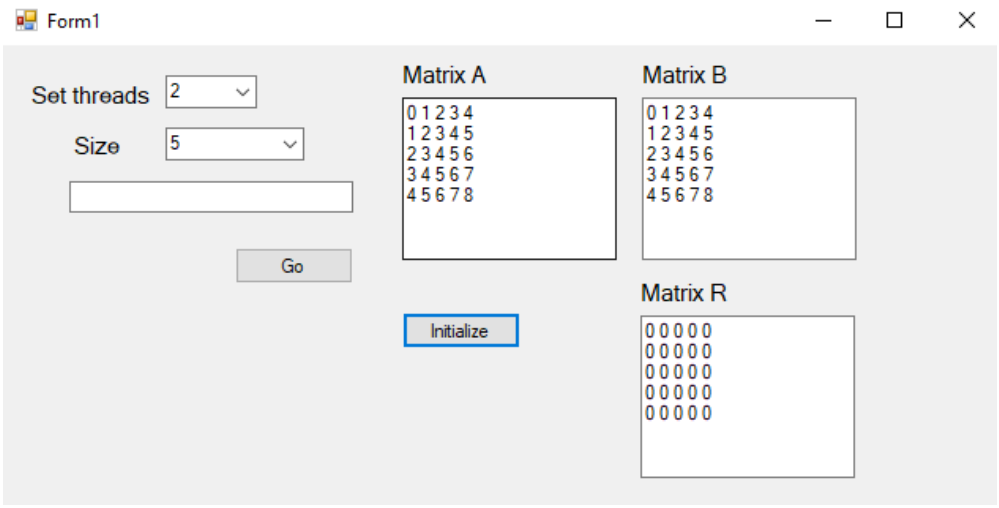



Figura 12

El siguiente paso es programar los cálculos. Deberán ser realizados por el background worker y comenzar cuando el botón “Go” sea accionado. El proceso de multiplicación de matrices consta de tres bucles anidados, pero la paralelización la aplicaremos solamente al bucle exterior:

```
#pragma omp parallel num_threads(nThreads)
	{
#pragma omp for
		for (int i = 0; i < rows; i++)
			for (int j = 0; j < rows; j++)
				for (int k = 0; k < rows; k++) {
					matrixR[i][j] += matrixA[i][k] * matrixB[k][j];
				}
	}
```

El número de iteraciones (rows) se repartirá entre “nThreads” hilos lanzados, de manera que cada uno de ellos ejecutará una fracción del total de iteraciones.

Necesitaremos ver los resultados. Para ello añadiremos un poco de código extra al método Go_Click:

```
if (rows < 20) {
	Rmatrix->ResetText();
	for (int i = 0; i < rows; i++) {
		for (int j = 0; j < rows; j++) {
	Rmatrix->AppendText(String::Concat(Convert::ToString(matrixR[i][j]), " "));
		}
		Rmatrix->AppendText("\n");
	}
}
```

Así, para matrices pequeñas Podemos comprobar que los resultados son correctos: Figura 13.

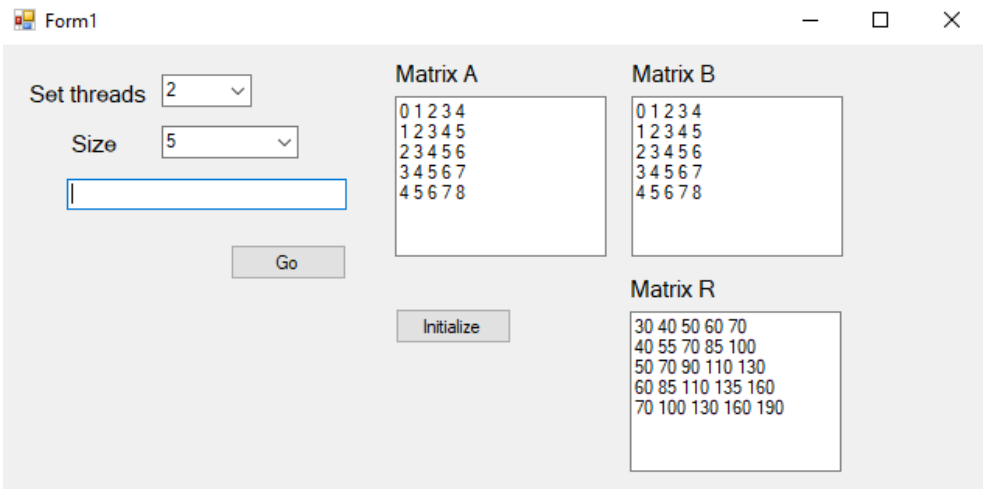



Figura 13

⚠ En algunos casos puede suceder que los primeros elementos de la matriz R aparezcan como “0”. No se trata de un error de cálculo. Se muestran como cero porque, mientras se

están realizando los cálculos, el hilo principal continúa actuando sobre la interfaz de usuario; es decir, que empieza a imprimir los resultados antes de que estén disponibles. Podríamos sincronizarlos, pero no es muy importante en este momento ya que los resultados solo nos sirven para comprobar los cálculos y no serán mostrados en casos reales.

# Capítulor 5: temporizadores y contadores

Ahora ya funciona, pero ¿cómo de bien? Necesitamos alguna información extra para determinar si estas técnicas de programación realmente mejoran el rendimiento o no. Lo primero que tenemos que hacer es medir el tiempo que se tarda en realizar los cálculos. Esto conlleva una pequeña modificación en el método backgroundWorker1_DoWork:

```
double stime = omp_get_wtime();
#pragma omp parallel num_threads(nThreads)
	{
#pragma omp for
		for (int i = 0; i < rows; i++)
			for (int j = 0; j < rows; j++)
				for (int k = 0; k < rows; k++) {
					matrixR[i][j] += matrixA[i][k] * matrixB[k][j];
				}
	}
	stime = omp_get_wtime() - stime;
	message = String::Concat("Elapsed time: ", Convert::ToString(stime), "
seconds");
	SetText(message);
```

Arrancamos un temporizador antes del inicio de los cálculos, tomamos el tiempo cuando finalizan y mostramos el tiempo transcurrido. Se debería obtener lo que se muestra en la Figura 14.

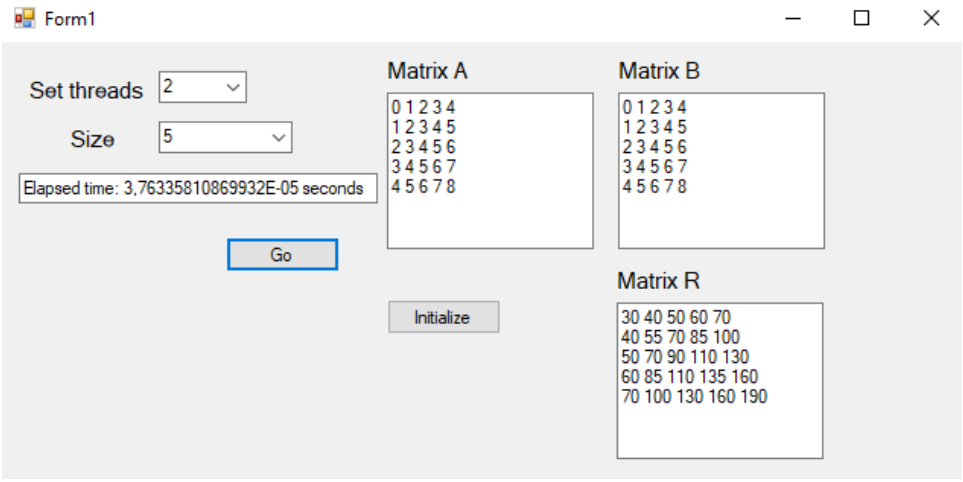



Figura 14

El tiempo transcurrido proporciona una Buena idea del funcionamiento de la paralelización. No obstante, puede ser interesante conocer algo más sobre el rendimiento de la aplicación. Ahí es cuando entran en juego los contadores de rendimiento. ¿Qué son entonces los contadores de rendimiento?

## Contadores de rendimiento.

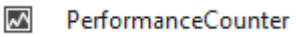
De acuerdo con la web de Microsoft:

*"Counters are used to provide information as to how well the operating system or an application, service, or driver is performing. The counter data can help determine system bottlenecks and fine-*

*tune system and application performance. The operating system, network, and devices provide counter data that an application can consume to provide users with a graphical view of how well the system is performing."*

El Framework .NET que estamos usando incluye el espacio de nombres System.Diagnostics que proporciona acceso a los contadores disponibles en el sistema. El Explorador de Servidores, habitualmente en el lado izquierdo de la pantalla, proporciona una lista de los contadores disponibles para nuestro sistema (Figura 16).

Para utilizar cualquiera de ellos, arrastrar el control desde la caja de herramientas:




La mayor parte de los contadores son dependientes de la plataforma así que conviene asegurarse de que los que se están usando vayan a estar disponibles en la plataforma de destino. En este caso vamos a monitorizar el porcentaje total de CPU utilizado:

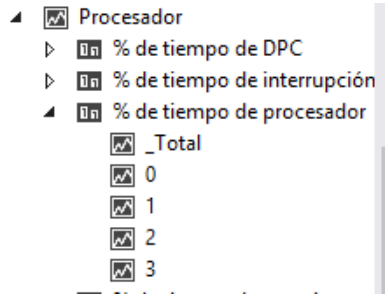



Figura 15

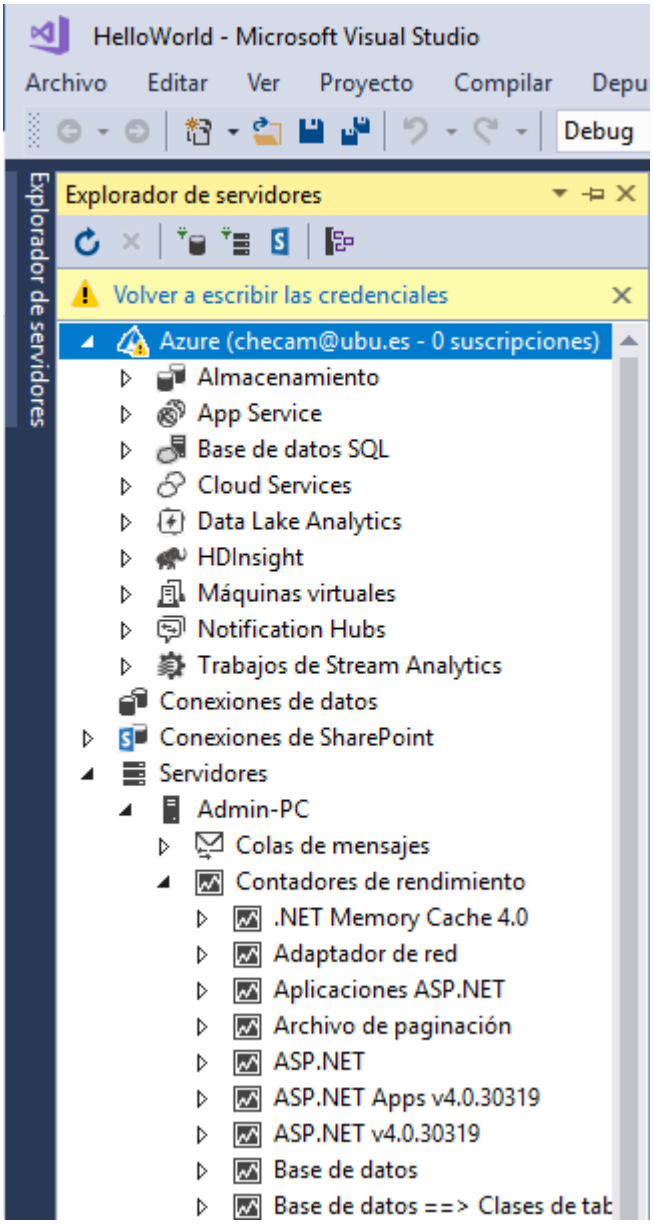



Figura 16

Lo vamos a mostrar en una caja de texto, así que tendremos que añadirla al Diseño junto con una etiqueta.

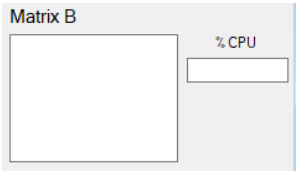



Figura 17

Los contadores de rendimiento proporcionan la información cuando se les solicita. Podemos hacerlo manualmente, mediante un botón al efecto, pero resultaría muy tedioso para el usuario, por lo que vamos a programar un temporizador.

## Temporizadores.

Los temporizadores son relojes que discurren a un ritmo determinado. Podemos usar uno para ordenar la actualización de los contadores. Lo Podemos encontrar en la caja de herramientas: Timer .

En la Ventana de propiedades Podemos ajustar su intervalo. Lo vamos a ajustar a 500 para que salte cada 0,5s. Un doble click sobre su icono nos llevará al método timer1_Tick:

```
private: System::Void timer1_Tick(System::Object^  sender, System::EventArgs^  e) {
	this->textBox5->Text = Convert::ToString(performanceCounter1->NextValue());
}
```

Basta con llevar el valor retornada a la caja de texto.

La interfaz sería finalmente la siguiente:

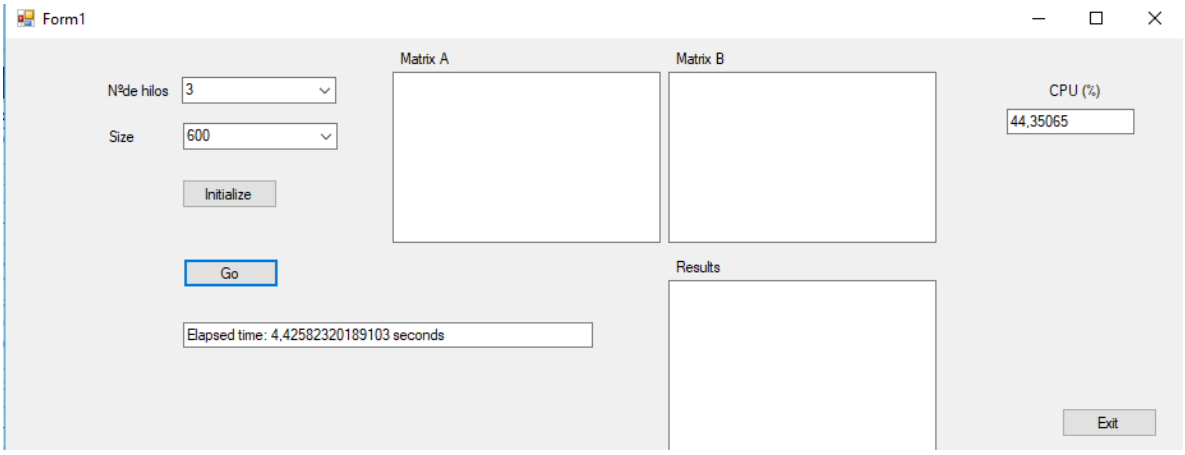



Figura 18

Hemos añadido sin explicación el botón de salida “Exit”. Es conveniente disponer del él, pero dejamos que el estudiante aprenda por su cuenta cómo incluirlo.

# Capítulo 6: trabajo del alumno

En este momento deberíamos tener nuestra multiplicación de matrices funcionando. ¿Qué viene ahora?

Se pretende que el alumno intente algunos cambios en la aplicación para optimizar su rendimiento.

En el capítulo 4 vimos cómo paralelizar el bucle "for" utilizando la distribución por defecto del número total de iteraciones entre los hilos disponibles. Este reparto lo realiza el sistema antes de la ejecución y no tenemos control sobre ello.

Podemos divider de forma explícita el número de iteraciones en "trozos" de un tamaño determinado y asignárselos a los hilos de forma estática o dinámica.

| Planificación estática (trozos de 10 iteraciones) | Planificación dinámica (trozos de 10 iteraciones) |
|---|---|
| `#pragma omp parallel num_threads (N)`<br>`{`<br>`  #pragma omp for schedule(static,10)`<br><br>`  for(i=0;i<n;i++){`<br>`  Operations to be performed on variable j`<br>`  }`<br>`}` | `#pragma omp parallel num_threads (N)`<br>`{`<br>`  #pragma omp for schedule(dynamic,10)`<br><br>`  for(i=0;i<n;i++){`<br>`  Operations to be performed on variable j`<br>`  }`<br>`}` |

La función `omp_get_wtime()` nos proporcionará información útil sobre el comportamiento de la aplicación en cada caso. El Contador de % CPU dará la explicación de los resultados.

El alumno deberá encontrar y documentar los ajustes mejores posibles junto con una explicación de los resultados obtenidos.