Visual C++ shared memory (OpenMP) programming.

José María Cámara Nebreda, César Represa Pérez, Pedro Luis Sánchez Ortega
Visual C++shared memory (OpenMP) programming. 2018
Área de Tecnología Electrónica
Departamento de Ingeniería Electromecánica
Universidad de Burgos

Table of contents

Introduction

Parallel programing is widely used to solve complex computational problems by making the most of a limited number of hardware resources. Several paradigms have been proposed:

- Shared memory (OpenMP)
- Distributed memory (MPI)
- Heterogeneous computation (CUDA)
- Hybrid: mixing at least two of the previous ones

These alternatives can be seen as “mid-level” programming approaches. Not too high so you still take control of most of what’s been done underneath, not too low so you don’t get involved in a daunting task.

Most of the above-mentioned options are usually presented as extensions to high level programming languages, and those languages are often those more adequate when efficient computation comes at stake. One of them is C/C++.

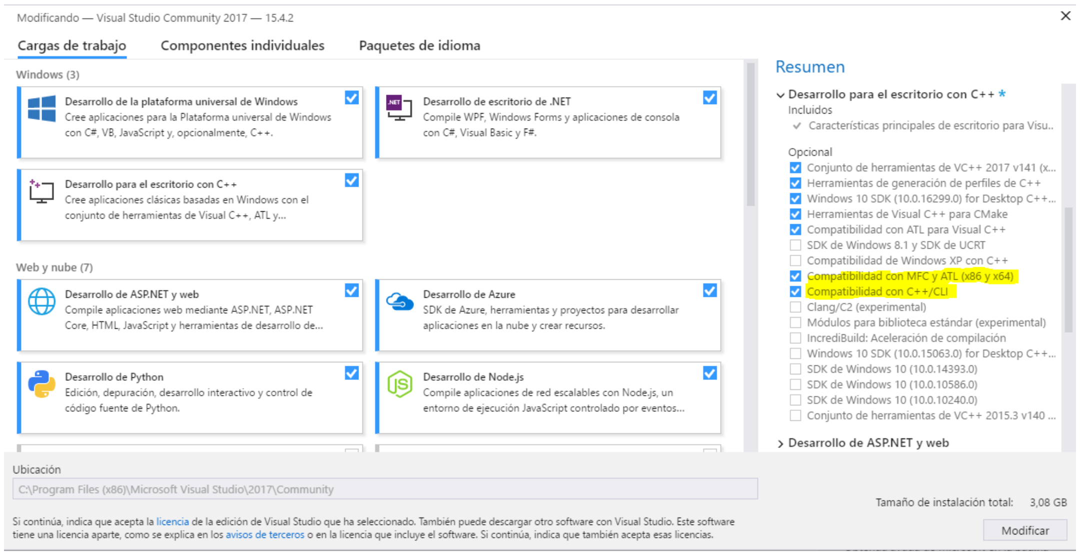
Applications making use of these parallelization techniques are usually hidden to the common users and, in many cases unknown to programmers. However, in the last years parallel hardware has become available to all users so it makes sense to take advantage of it within a single application. Writing multithreaded code at a lower level makes it tedious and prone to mistakes in the form of race conditions, deadlock and more.

In this manual we try to give an example of how parallel programming paradigms can be included in most application easily and with high improvement on overall system performance. In this particular case we aim to explain how to insert shared memory OpenMP sections within Visual C++ code. Our development tool will be MS Visual Studio 2017 Community version and we will use Windows Forms templates for simplicity. It aims to help either those who are used to writing desktop applications and want to embed parallel code within or those who are used to parallelization but don't know how to insert it into a desktop application.

Chapter 1: Solution settings

Before we can create multithreaded application, we have to make some settings:

1. Create a Visual C++ project using MS Visual Studio 2017 (File->New->Project). If the Windows Forms template does not show up, you'll probably need to install the software highlighted in Figure 1.



Modificando — Visual Studio Community 2017 — 15.4.2

Cargas de trabajo | Componentes individuales | Paquetes de idioma

Windows (3)

- ✓ **Desarrollo de la plataforma universal de Windows** – Cree aplicaciones para la Plataforma universal de Windows con C#, VB, JavaScript y, opcionalmente, C++.
- ✓ **Desarrollo de escritorio de .NET** – Compile WPF, Windows Forms y aplicaciones de consola con C#, Visual Basic y F#.
- ✓ **Desarrollo para el escritorio con C++** – Cree aplicaciones clásicas basadas en Windows con el conjunto de herramientas de Visual C++, ATL y...

Web y nube (7)

- ✓ **Desarrollo de ASP.NET y web** – Compile aplicaciones web mediante ASP.NET, ASP.NET Core, HTML, JavaScript y herramientas de desarrollo de...
- ✓ **Desarrollo de Azure** – SDK de Azure, herramientas y proyectos para desarrollar aplicaciones en la nube y crear recursos.
- ✓ **Desarrollo de Python** – Edición, depuración, desarrollo interactivo y control de código fuente de Python.
- ✓ **Desarrollo de Node.js** – Compile aplicaciones de red escalables con Node.js, un entorno de ejecución JavaScript controlado por eventos...

Ubicación

C:\Program Files (x86)\Microsoft Visual Studio\2017\Community

Si continúa, indica que acepta la licencia de la edición de Visual Studio que ha seleccionado. También puede descargar otro software con Visual Studio. Este software tiene una licencia aparte, como se explica en los avisos de terceros o en la licencia que incluye el software. Si continúa, indica que también acepta esas licencias.

Resumen

Desarrollo para el escritorio con C++ *

Incluidos

- ✓ Características principales de escritorio para Visu..

Opcional

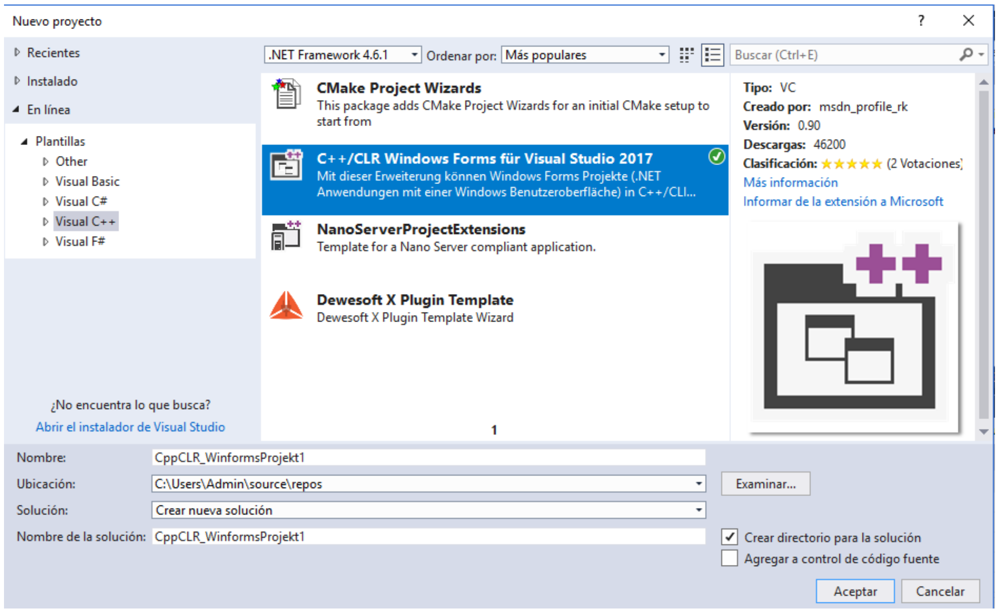
- ✓ Conjunto de herramientas de VC++ 2017 v141 (x...
- ✓ Herramientas de generación de perfiles de C++
- ✓ Windows 10 SDK (10.0.16299.0) for Desktop C++...
- ✓ Herramientas de Visual C++ para CMake
- ✓ Compatibilidad con ATL para Visual C++
- ☐ SDK de Windows 8.1 y SDK de UCRT
- ☐ Compatibilidad de Windows XP con C++
- ✓ Compatibilidad con MFC y ATL (x86 y x64)
- ✓ Compatibilidad con C++/CLI
- ☐ Clang/C2 (experimental)
- ☐ Módulos para biblioteca estándar (experimental)
- ☐ IncrediBuild: Aceleración de compilación
- ☐ Windows 10 SDK (10.0.15063.0) for Desktop C++...
- ☐ SDK de Windows 10 (10.0.14393.0)
- ☐ SDK de Windows 10 (10.0.10586.0)
- ☐ SDK de Windows 10 (10.0.10240.0)
- ☐ Conjunto de herramientas de VC++ 2015.3 v140 ...

Desarrollo de ASP.NET y web

Tamaño de instalación total: 3,08 GB

Figure 1

2. Find the Windows Forms template as shown in Figure 2.



Nuevo proyecto

.NET Framework 4.6.1 | Ordenar por: Más populares | Buscar (Ctrl+E)

CMake Project Wizards
This package adds CMake Project Wizards for an initial CMake setup to start from

C++/CLR Windows Forms für Visual Studio 2017
Mit dieser Erweiterung können Windows Forms Projekte (.NET Anwendungen mit einer Windows Benutzeroberfläche) in C++/CLI...

NanoServerProjectExtensions
Template for a Nano Server compliant application.

Dewesoft X Plugin Template
Dewesoft X Plugin Template Wizard

1

Tipo: VC
Creado por: msdn_profile_rk
Versión: 0.90
Descargas: 46200
Clasificación: ★★★★★ (2 Votaciones)
Más información
Informar de la extensión a Microsoft

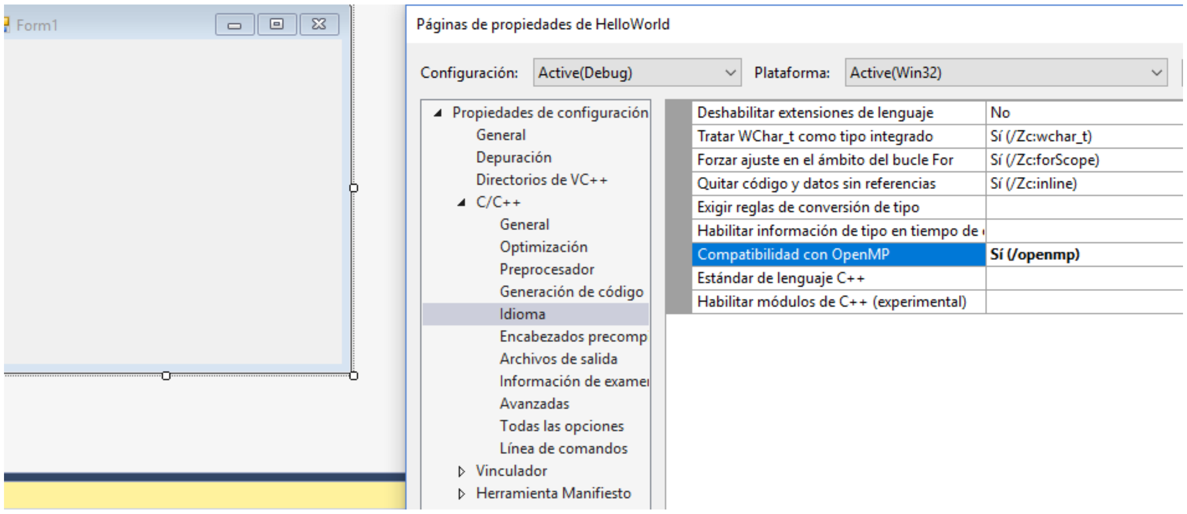
¿No encuentra lo que busca?
Abrir el instalador de Visual Studio

Nombre: CppCLR_WinformsProjekt1
Ubicación: C:\Users\Admin\source\repos
Solución: Crear nueva solución
Nombre de la solución: CppCLR_WinformsProjekt1

[✓] Crear directorio para la solución
[ ] Agregar a control de código fuente

Figure 2

3. In Project->Settings enable OpenMP compatibility as depicted in Figure 3.



Páginas de propiedades de HelloWorld

Configuración: Active(Debug) Plataforma: Active(Win32)

- Propiedades de configuración
  - General
  - Depuración
  - Directorios de VC++
  - C/C++
    - General
    - Optimización
    - Preprocesador
    - Generación de código
    - Idioma
    - Encabezados precomp
    - Archivos de salida
    - Información de exame
    - Avanzadas
    - Todas las opciones
    - Línea de comandos
  - Vinculador
  - Herramienta Manifiesto

Deshabilitar extensiones de lenguaje	No
Tratar WChar_t como tipo integrado	Sí (/Zc:wchar_t)
Forzar ajuste en el ámbito del bucle For	Sí (/Zc:forScope)
Quitar código y datos sin referencias	Sí (/Zc:inline)
Exigir reglas de conversión de tipo	
Habilitar información de tipo en tiempo de	
Compatibilidad con OpenMP	**Sí (/openmp)**
Estándar de lenguaje C++	
Habilitar módulos de C++ (experimental)	

Figure 3

4. Since we will use some of the controls available in the tool box, type Ctl+Alt+X to gain access to them. The controls will be visible when the designer window is selected only.

Now, we are ready to build our firs program. Let’s do it in the next chapter.

Chapter 2: A simple multithreaded program

In this chapter we will see how to construct the simplest multithreaded program. On that simply counts the number of threads issued by the user.

Let’s find in the tool box, the controls we are about to use in this chapter:

Label	Label
Text box	TextBox
Combo box	ComboBox
Button	Button

Drag them onto the Form 1 canvas so it looks similar to Figure 4.

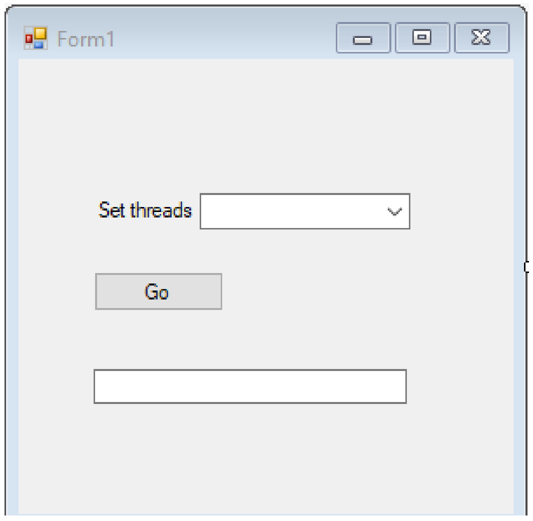
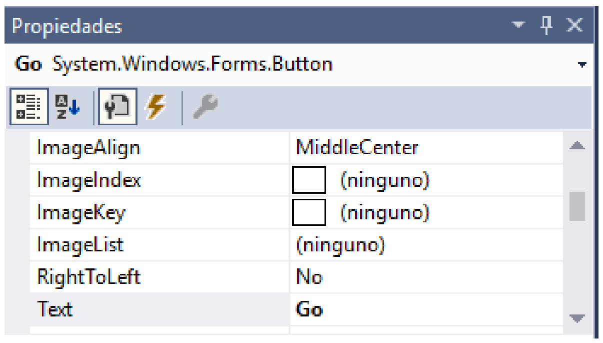


Figure 4

Controls take always a default name and those that include text, may have a default message as well. You will find out that the button you inserted takes default name “button 1” and the same default text. On the design view you just have to click on the control to get access to its properties on the left down window of your screen. Change these values if you will. You can change button 1 text property to “Go” for instance, as shown in Figure 5. It will then look exactly as in Figure 4.



Propiedades

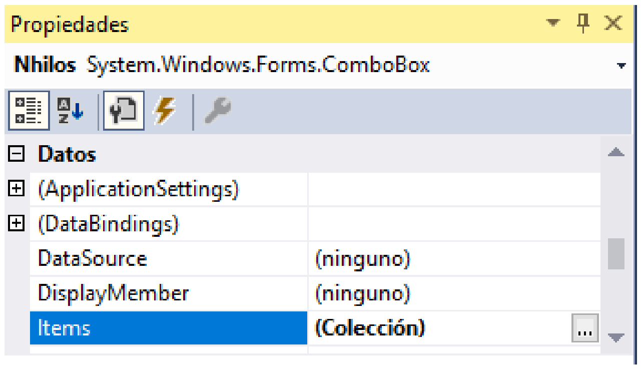
Go System.Windows.Forms.Button

ImageAlign	MiddleCenter
ImageIndex	(ninguno)
ImageKey	(ninguno)
ImageList	(ninguno)
RightToLeft	No
Text	**Go**

Figure 5

You can do the same with label 1 text property.

The combo box works a bit different. It is supposed to contain a list of items so the user can select the desired one. Figure 6 shows the properties window for this control and the "Items" property among them. Select the collection of items so you can include as many as you want; one per line.



(ApplicationSettings)	
(DataBindings)	
DataSource	(ninguno)
DisplayMember	(ninguno)
Items	(Colección)

Figure 6

We are using this control to set the number of threads to be issue. It normally makes sense to launch as many as the number of cores in your processor. So the list would span from 1 to that number.

In this particular case we have a few controls but, when the number of controls of the same kind may increase it is a good practice to change their default names as well. You can always do it later but you will probably need to change them in several places within your code.

So far, we have the user interface but, it does nothing at all. Let's write the code then. We have already seen how to modify certain properties from the design window. This can also be done from the code window or by your program at run time. To switch to the code window, you just have to right click on Form1.h in the solution explorer or type "F7".

Apart from properties, controls have associated events. In the same properties window, you can click on the ⚡ icon to see the events for the selected control.

But before you program the actions, some preliminary adjustments must be done.

1. Include the OpenMP header. This is done at the beginning of form1.h:
```
#pragma once
#include "omp.h"
```

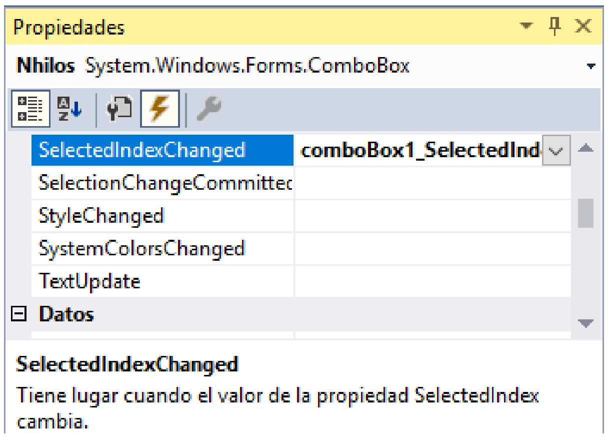
2. Declare global variables. In this example we need only two of them. An integer number that represents the number of threads to be issued, and a string pointer to a message to be built:
```
private:
              int nThreads;
              String^ message;
```

```
                    /// <summary>
                    /// Erforderliche Designervariable.
```

3. Initialize them. We will launch one thread by default:

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

Now let us tackle the code issue. In this simple example only two methods need to be programmed. The easiest of them is the response to the selection of a concrete number of threads in the combo box by the user. Select the events for the combo box and then you will display something like Figure 7 .


Propiedades

Nhilos System.Windows.Forms.ComboBox

SelectedIndexChanged	comboBox1_SelectedInd
SelectionChangeCommittec	
StyleChanged	
SystemColorsChanged	
TextUpdate	
⊟ Datos	

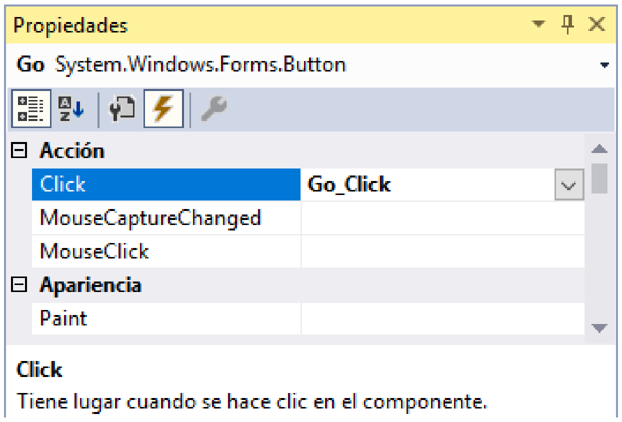
SelectedIndexChanged
Tiene lugar cuando el valor de la propiedad SelectedIndex cambia.

Figure 7

If you go to the code window, you will find the “comboBox1_SelecdIndexChanged” method preprogramed. You just have to add the code of the action to be taken, between the brackets. The action will be to update the number of threads:

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

The second action is a bit more complex and it is the heart of the program, since it includes the parallel part of it. Now select the “Go” button to display its events tab. Among them, the mouse Click event. In the drop down list you can select the “Go_click” event (Figure 8).


Propiedades

Go System.Windows.Forms.Button

Acción

Click	Go_Click
MouseCaptureChanged	
MouseClick	

Apariencia

Paint	

Click

Tiene lugar cuando se hace clic en el componente.

Figure 8

If you go to the code window, you will find the "Go_click" method preprogramed. Then again, you just have to add your code between the brackets. The code will be this:

```
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;
}
```

¿What does it mean?

- Variable "sum" holds the number of threads. We already know them on nThreads ; now we are going to calculate them again but in parallel.
- `#pragma omp parallel` declares that what is within the brackets will be executed in parallel. As many as "nThreads" threads will be launched in parallel. The code is the same for all of them and varaibles declared outside this "Parallel region" are shared by default.
- `#pragma omp parallel reduction(+:sum)` starts a reduction operation, this meaning that an operation will be performed on values all threads put in the variable. The operation is a sum and the variable has the same name. Since all the threads put a "1" in "sum", and all the values of sum are added, the result must be equal to the number of threads.
- To check that out a message is printed on the text box.

for the shake of clarity, we do a horrible and senseless use of the variables and threads in this example. If you run it you will see how a hello message is displayed on behalf of all the threads the user decided to launch.

Don’t expect to be able to write messages from the threads individually. This would be all but easy. Not impossible but mostly meaningless. Think of the parallel regions as sections of code where complex calculation is performed efficiently. Keep user interface single threaded.

Chapter 3: Background workers

In the previous chapter we saw how to build up a parallel application. That structure works for most applications we may need to design but it has a problem. Parallel processing makes sense when lots of operations must be performed. In such situation our previous solution can do the job but, for so long as the program is calculating, the user interface will be “frozen”.

So, it works but it doesn’t make sense since we have decided to design a desktop application for a reason, otherwise we could go back to the more common console application for heavy computational problems.

How do we keep our user interface active while intensively processing calculation on all available cores? This is when background workers come along. Background workers behave as independent threads that execute in background, so they don’t interfere with the main thread: the user interface.

The background worker is in the tool box: BackgroundWorker

When you drag it on the design window, it will immediately move downwards, since it is not a visible part of the user interface (Figure 9).

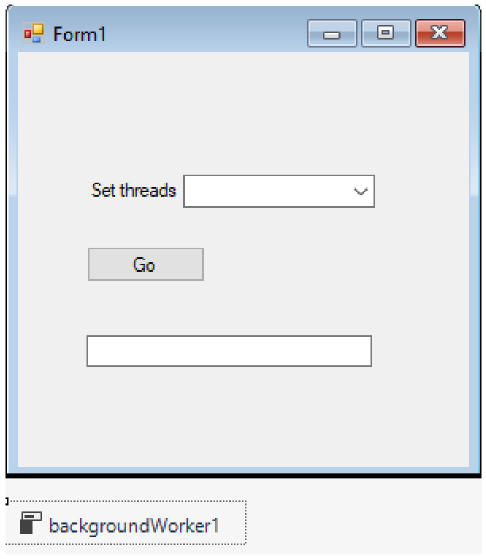


Figure 9

Now, double click on the backgroundWorker1 icon and see what the code looks like. There is a default method called backgroundWorker1_DoWork that allows you to state what the worker must do. The Go_Click method becomes extremely simple. It just starts the worker:

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

And all the work is done by the DoWork method:

```
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;
}
```

⚠ But there is a problem! If you try to debug the new code, an exception will pop up. The reason: the textBox1 control is being called from a thread other than its creator. This is unsafe and may lead to many issues. We must implement a safe access to this control.

Let us create a new method that serves as a safe interface to textBox1. Below, we can see both together so it is easy to see their combined work:

```
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;
       }
}
```

Set text must be used to access textBox1 in a safe manner, either from the background worker or by the main thread. It will decide when to invoke the delegate.

Chapter 4: matrix multiply

So far we have learned how to get everything ready for a parallel application but no real parallelization has been introduced. We need a calculation intensive problem to solve and one of the most typical is the matrix multiplication problem.

Everyone knows how matrices are multiplied (we will multiply square matrices for simplicity). This is a good start but, apart from that this is a problem that presents fantastic scalability, it is easy to program and can be applied in many more complex applications.

What do we need to multiply matrices? First, the matrices. These matrices are two-dimensional arrays of float numbers (could be any other type but float is a good choice). Let's declare the new variables to be used in this program:

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

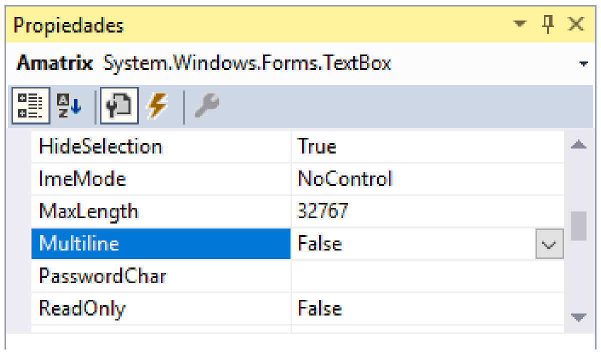
Along with the three matrices R = (A x B), we have also declared an integer variable representing the number of rows (also columns) for each one. We have to give it a default value:

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

Matrices of this size (4x4) are tiny for our purposes but we will give the user the opportunity to change that. To do so we need a second label and a second combo box. Set the label text to something like "Size" and provide a collection of values to the combo box ranging from 5 to 8000 with the intermediate values you wish.

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

If we want to visualize the contents of the matrices, we will need some text boxes where the program can show their contents. We will attach their correspondent labels too. Text boxes are single lined by default but we can switch their "multiline" property to "true" and make them as long and high as we want (Figure 10).


Propiedades

Amatrix System.Windows.Forms.TextBox

HideSelection	True
ImeMode	NoControl
MaxLength	32767
Multiline	False
PasswordChar	
ReadOnly	False

Figure 10

Once we have the text box and a label, we can copy paste them twice to make room for the two remaining matrices.

We are going to add a new button called “Initialize”. A click on it will trigger two actions:

1. Allocate memory space for the three matrices.
2. Fill the space with numbers.

This is what it should look like:

```
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");
                }
        }
}
```

The user interface so far should look similar to Figure 11.

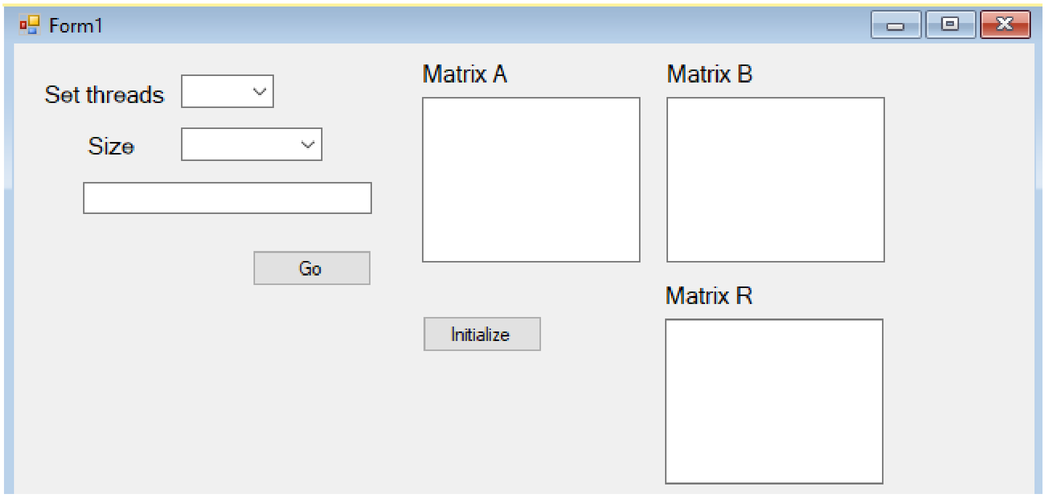
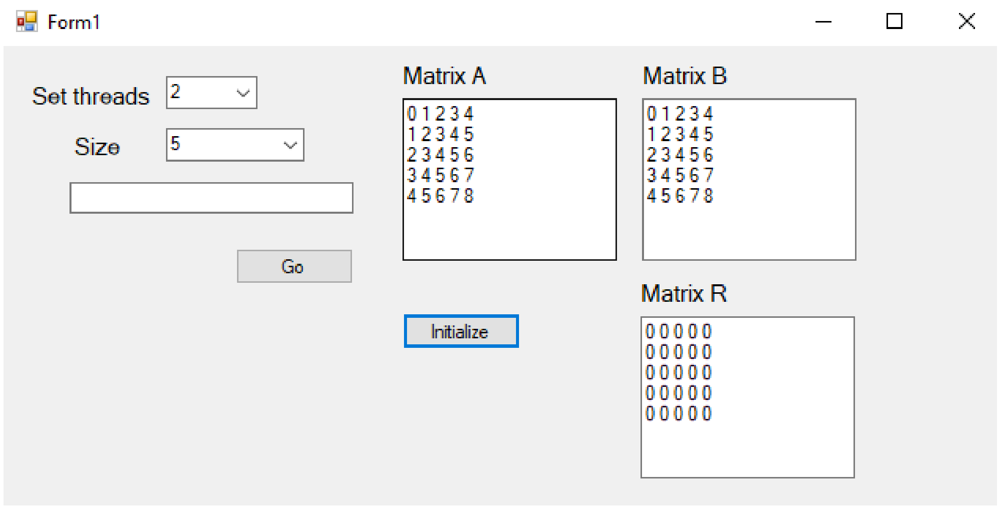


Figure 11

We have added the condition for the matrices to be smaller than 20 rows to print then for otherwise we will see nothing clear on the boxes. The numerical contents of the matrices A and B are determined according to the coordinates of each element within the matrix. It is just a simple way to fill the matrices and then check that the calculations are correct.

If we try a 5x5 matrix initialization, we can check whether it is correct. If so we should be watching Figure 12 on our computer.



Set threads: 2
Size: 5

Matrix A
```
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
```

Matrix B
```
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
```

Matrix R
```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

Figure 12

Now we only must program the calculations. They must be performed by the background worker and start when the “Go” button is clicked. A matrix multiplication process consists of three nested for loops. We will parallelize in this case the outermost of them:

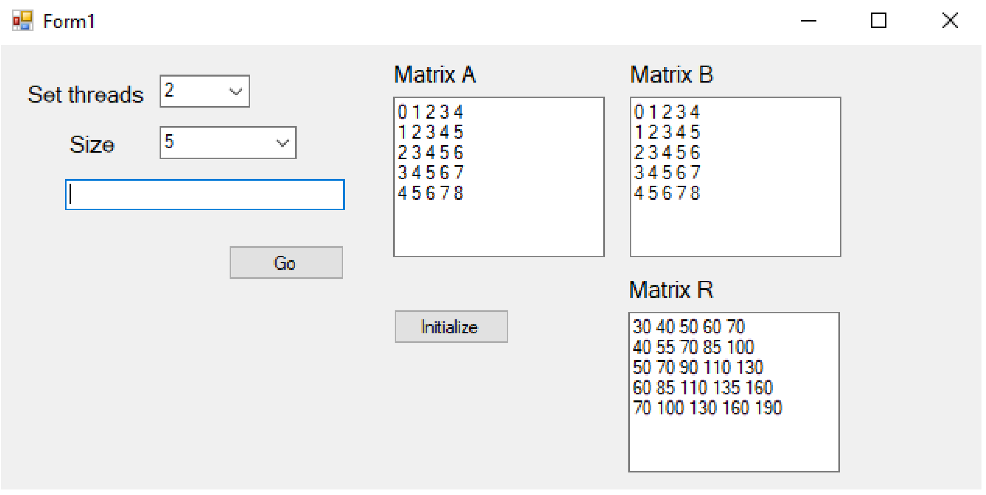
```
#pragma omp parallel num_threads(nThreads)
       {
#pragma omp for
              for (int i = 0; i < rows; i++)
                     for (int j = 0; j < rows; j++)
                            for (int k = 0; k < rows; k++) {
                                   matrixR[i][j] += matrixA[i][k] * matrixB[k][j];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

The number of iterations (rows) will split up among the “nThreads” launched so each one of them will execute only a fraction of the calculations.

It would be good to see the results. For this purpose, we can add some extra code to Go_Click method:

```
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");
       }
}
```

Then, for small matrices we can check the results as depicted in Figure 13.



```
Matrix A
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8

Matrix B
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8

Matrix R
30 40 50 60 70
40 55 70 85 100
50 70 90 110 130
60 85 110 135 160
70 100 130 160 190
```

Figure 13

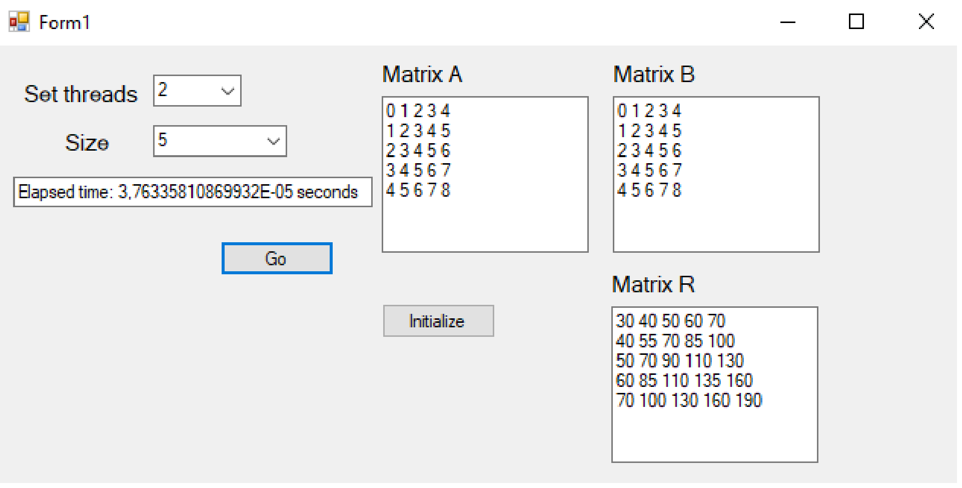
⚠ You may find that, in some cases, the first elements of Matrix R are set to “0”. This is not a calculation mistake. They are cero because the main thread keeps going while the calculations are still in progress. Therefore, the results are not available yet. We could try to synchronize the two but it is not particularly important at this moment since the results are only meant to check that calculations are correct and are not displayed in real operations.

Chapter 5: timers and counters

Now it works but, how fine? We need some extra information to determine whether these programming techniques really improve performance or not. First thing we need to do it measure time taken to resolve the calculations. To do so we just need a small modification in backgroundWorker1_DoWork method:

```
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);
```

We start a timer before the calculations begin, take time when they finish and display de elapsed time. What you should obtained is depicted in Figure 14.



```
Set threads 2
Size 5
Elapsed time: 3,76335810869932E-05 seconds

Matrix A
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8

Matrix B
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8

Matrix R
30 40 50 60 70
40 55 70 85 100
50 70 90 110 130
60 85 110 135 160
70 100 130 160 190
```

Figure 14

The elapsed time gives you a good idea on how your parallelization is doing. Nevertheless, you may be interested in some more data about system performance. This is when performance counters come into play. But, what are performance counters in the first place?

Performance counters.

According to Microsoft Web Site:

"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-

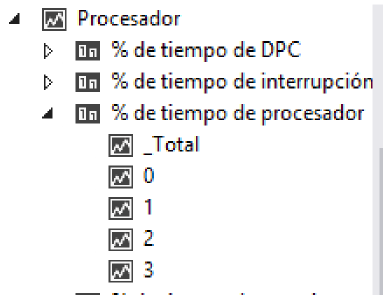
tune system and application performance. The operating system, network, and devices provide counter data that an application can consume to provide users with a graphical view of how well the system is performing.”

The .NET Framework we are using includes the System.Diagnostics namespace that provides access to the counters available in the system. The Server Explorer, usually on the left side of the screen, gives you a list of the counters available to your system (Figure 16).

To use any of them you will need to drag the Performance Counter control from the toolbox:

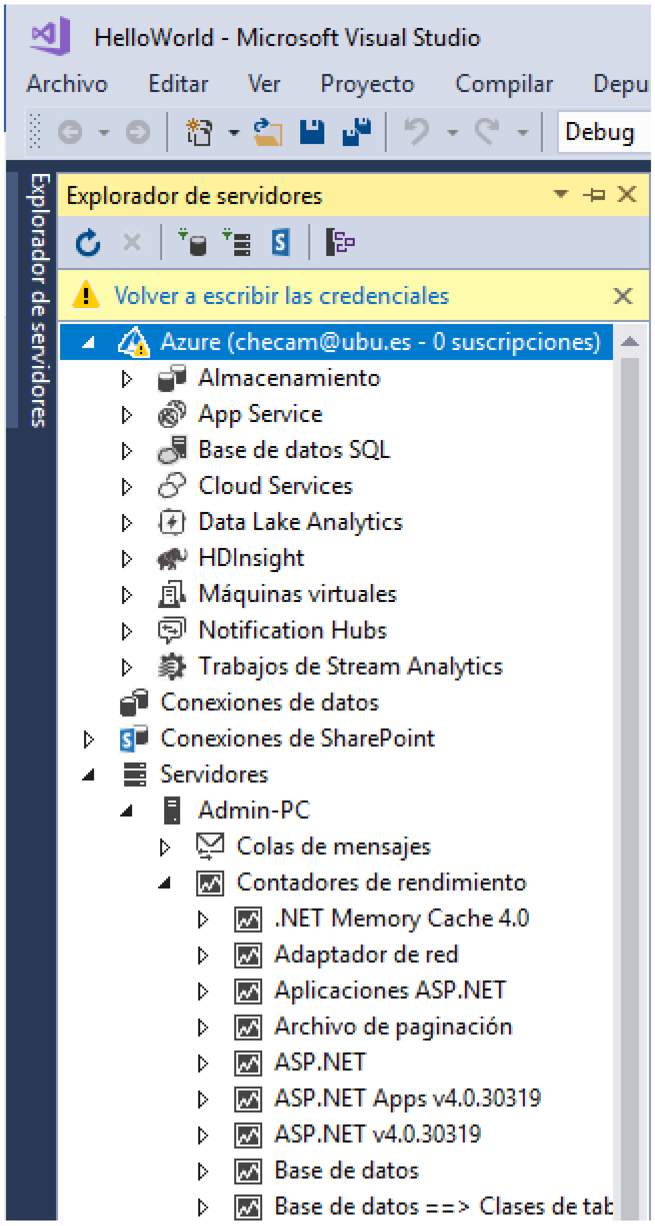
PerformanceCounter

Most of the counters are platform dependant so make sure that the counters to be used are available for your platform. In our case we will to monitor the overall percentage of CPU used:



Procesador
% de tiempo de DPC
% de tiempo de interrupción
% de tiempo de procesador
_Total
0
1
2
3

Figure 15



HelloWorld - Microsoft Visual Studio

Explorador de servidores

Volver a escribir las credenciales

Azure (checam@ubu.es - 0 suscripciones)
Almacenamiento
App Service
Base de datos SQL
Cloud Services
Data Lake Analytics
HDInsight
Máquinas virtuales
Notification Hubs
Trabajos de Stream Analytics
Conexiones de datos
Conexiones de SharePoint
Servidores
Admin-PC
Colas de mensajes
Contadores de rendimiento
.NET Memory Cache 4.0
Adaptador de red
Aplicaciones ASP.NET
Archivo de paginación
ASP.NET
ASP.NET Apps v4.0.30319
ASP.NET v4.0.30319
Base de datos
Base de datos ==> Clases de tab

Figure 16

We will display the CPU usage on a text box so we will add a label and a box to do so.

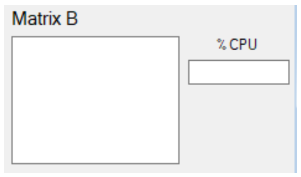


Figure 17

Performance counters provide information when they are told to do so. We could do it manually by clicking a button but this is too tedious for the user so we will do it on a timer tick.

The API is available to developers so application programs can make use of this information.

https://www.codeproject.com/Articles/8590/An-Introduction-To-Performance-Counters

https://www.developer.com/net/net/article.php/3356561/Reading-and-Publishing-Performance-Counters-in-NET.htm

Timers.

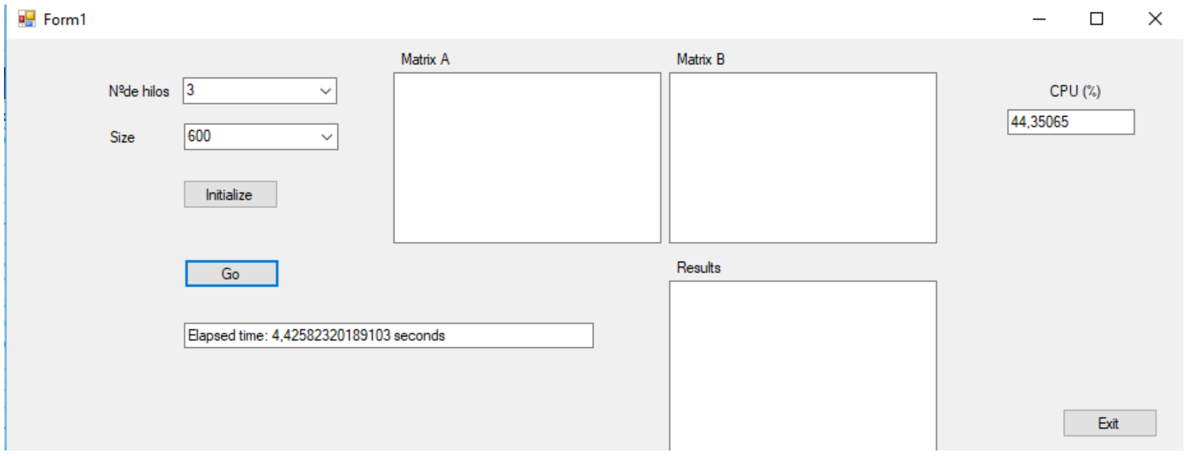
Timers are clocks that tick at a preconfigured pace. We can use one of them to trigger counter updates. We can find them on the tool box: ⏱ Timer .

On the properties window we can set the interval. Set this parameter to 500 for 0,5s. Then double click the timer icon to switch to the timer1_Tick method:

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

It is as simple as moving the value returned by the counter to the text box.

The final interface could be:



Nºde hilos: 3

Size: 600

CPU (%): 44,35065

Elapsed time: 4,42582320189103 seconds

We have just added but not explained the “exit” button. It is always nice to have one but we rely on the student’s abilities to program it.

Chapter 6: student’s work

At this this point we hopefully have our matrix multiply up and running. What to do next?

The student is meant to try some changes on the application in order to optimize its performance.

In chapter 4 we saw how to parallelize the for loop using the default distribution of the overall number of iterations among the available number of threads. It is done by the system before execution and we have no control over it.

We can explicitly split the number of iterations into chunks of a certain size and assign them to threads either statically or dynamically.

Static scheduling (chunks of 10 iterations)	Dynamic scheduling (chunks of 10 iterations)
`#pragma omp parallel num_threads (N) { #pragma omp for schedule(static,10) for(i=0;i<n;i++){ Operations to be performed on variable j } }`	`#pragma omp parallel num_threads (N) { #pragma omp for schedule(dynamic,10) for(i=0;i<n;i++){ Operations to be performed on variable j } }`

The `omp_get_wtime()` function will provide useful data to compare the performance of the different options. The % CPU counter may provide an explanation to the results.

The student will try to find and document the best possible settings along with a reasonable explanation for the results obtained.