Gaston Hillar
Dr. Dobb's Guru Bloggers

Gaston Hillar Bio | Contact | Archive

BLOGS

Scott W. Ambler
Walter Bright
Eric Bruno
Andrew Koenig
Mark Nelson
Ken North
Mike Riley
Al Williams

CHANNELS

Architecture & Design
C/C++
Database
Development Tools
Embedded Systems
High Performance Computing
Java
Jolt Awards
Mobility
Open Source
Security
Web Development
Windows/.NET
Visual Studio 2010

INFO-LINK

Measuring Speedup is Challenging with Intel Turbo Boost Technology

January 22, 2010

Sometimes, parallelized code can run slower than its sequential version because Intel Turbo Boost Technology can make the latter run at faster clock frequencies than the former. Therefore, if you work with a microprocessor with Intel Turbo Boost Technology enabled, you will have to pay attention to the changes introduced by this technology.The easiest way to understand the changes introduced by microprocessors with Intel Turbo Boost Technology is working with a very simple code snippet as an example of the situation that could happen in more complex applications. In this case, I'm going to work with C# and .NET 4 Beta 2.

I'm going to use a very simple C# Windows Forms application with two buttons to measure the speedup achieved by a parallelized version of a very simple algorithm. In order to keep the example simple, I'm going to use very simple code. It's main purpose is to explain the information shown by these new debugging windows. It doesn't represent a best practice. It is just code added to a Form class:

```
using System; using System.Collections.Generic; using System.ComponentModel; using System.Data; using System.Drawing; using System.Linq; using System.Text; using System.Windows.Forms; // Added for Parallel.For, and Concurrent Collections using System.Threading.Tasks; using System.Collections.Concurrent;

namespace WindowsFormsApplication1 { public partial class Form1 : Form { public Form1() { InitializeComponent(); } private ConcurrentQueue _results; private const int _maxIterations = 50000000;

private double CalculateProbability(int probabilityIteration) { // Do something that takes a long time return (Math.Sqrt((double) probabilityIteration) * Math.Sqrt((double)probabilityIteration)); } private void AddProbability(double probability) { _results.Enqueue (probability); }

// Parallelized version private void button1_Click(object sender, EventArgs e) { var sw = System.Diagnostics.Stopwatch.StartNew (); _results = new ConcurrentQueue(); Parallel.For(0, _maxIterations, i => { AddProbability(CalculateProbability(i)); }); button1.Text = sw.Elapsed.ToString(); }

// Sequential version private void button2_Click(object sender, EventArgs e) { var sw = System.Diagnostics.Stopwatch.StartNew (); _results = new ConcurrentQueue(); int i; for (i = 0; i < _maxIterations; i++) { AddProbability(CalculateProbability(i)); } button2.Text = sw.Elapsed.ToString(); } } }
```

There are two Button controls:

* *button1*: runs a parallelized version using Parallel.For, introduced in .NET 4 with the Task Parallel Library (TPL).

* *button2*: runs a sequential version with a classic for loop.

The code uses the new System.Threading.Collections.ConcurrentQueue to store double results. A ConcurrentQueue represents a variable size first-in-first-out (FIFO) collection. It is possible to add and remove items from

Recent Articles

- JVM Language Summit 2011 Videos
- Breaking Away From The Unit Test Group Think
- C++0x's Tools for Library Authors
- GData: Accessing Google-Application Data
- User-Defined Hash Functions for Unordered Maps in C++0x

Most Popular

Stories Blogs

- Breaking Away From The Unit Test Group Think
- Jolt Awards: The Best Books
- C++0x's Tools for Library Authors
- Enhancing Assertions
- CUDA, Supercomputing for the Masses: Part 1

Dr. Dobb's TV

View All Videos

this concurrent collection letting the Task Parallel Library manage the necessary low-level coordination stuff. Both the sequential and the parallelized versions use the concurrent collection. The former doesn't need the concurrent collection; however, the idea is to keep the example as simple as possible and to call the same CalculateProbability method in both cases.

The parallelized version (button1) runs a Parallel.For from 0 to _maxIterations (exclusive). The code just uses some processing power. However, it will try to take advantage of all the available hardware threads. Again, this is not a best practice for parallelized code. However, it allows you to compare the results with a sequential version with Intel Turbo Boost Technology. For example, this way, you will learn that measuring speedup can be more complicated than expected.

The sequential version (button2) runs the same algorithm but using a classic for loop from 0 to _maxIterations - 1. Therefore, it will use just one hardware thread.

An Intel Core i7 820QM mobile microprocessor offers 4 physical cores and 8 hardware threads. I'm going to explain the results offered with a specific configuration to illustrate the additional complexity introduced by Turbo Boost.

The parallelized version with Intel Turbo Boost Technology *enabled* takes 6.1 seconds to run. The sequential version with Intel Turbo Boost Technology *enabled* takes 6.5 seconds to run. The speedup achieved by the parallel execution is: Speedup = (Serial execution time) / (Parallel execution time) Speedup = 6.5 / 6.1 = 1.065x

The aforementioned situation means that using 8 hardware threads (4 physical cores with Hyper-Threading technology) the parallelized code runs just 1.065 times faster than the sequential version.

What's going on? Intel Turbo Boost Technology is improving the performance for the sequential version because it overclocks a single core or one core at a time. As the example is based on managed code, there is no possibility to work with thread affinity. Therefore, the scheduler usually moves the single thread from one core to the other. The microprocessor is cooler because the other cores are almost idle. There is just one core with heavy workload and Turbo Boost can overclock it to higher frequencies as the other cores are idle.

Turbo Boost also overclocks cores when running the parallelized version. However, as in this case, all the cores have heavy workloads, the microprocessor cannot keep all the cores overclocked a lot of time, because it consumes more power and its temperature increases faster. Therefore, the average clock frequency for all the cores running the parallelized version is lower than the one achieved for the sequential version.

Now, I'm going to explain the results offered with the same hardware configuration with Turbo Boost disabled.

The parallelized version with Intel Turbo Boost Technology *disabled* takes 7.2 seconds to run. The sequential version with Intel Turbo Boost Technology *disabled* takes 9.8 seconds to run. The speedup achieved by the parallel execution is: Speedup = (Serial execution time) / (Parallel execution time) Speedup = 9.8 / 7.2 = 1.361x

As you can see, the numbers are completely different. It is still a poor speedup. However, it is better than the first case. The speedup is better but it took more time to run each algorithm.

The problem with the previously shown code is that the CalculateProbability method requires little processing power. Therefore, the overhead introduced by the need to call a delegate in each loop's iteration reduces the potential speedup. Of course, there are many other techniques to achieve better speedups using Task Parallel Library features. However, the idea of this post is to present both the benefits and the distortion introduced by Intel Turbo Boost Technology.

It is very important to understand that Intel Turbo Boost Technology improved both the parallel and the serial execution times. However, it is necessary to understand this new technology to improve your parallelized code in order to take full advantage of the complex underlying multicore hardware.

There is a very easy way to enable and disable Intel Turbo Boost Technology in both 32-bits and 64-bits versions of Windows. You can use TMonitor, right-click on its window, select Turbo from the context menu that appears and click on Enable or Disable. Besides, as explained in my previous post, this tool will allow you to understand how the multiplier for each core changes while the code is being executed.

Care to Comment?

Subject (max length: 75):

Comments:

Captcha:

Type the characters you see in the picture above.

Log On To Add Your Comment

AROUND THE WEB

Custom Development Becoming Key Strategic Differentiator

At financial services firms, developers rule once again!

Quick Read

How Well Does the HP TouchPad Handle HTML5?

Testing HTML5 apps on the TouchPad reveals a fast software stack with a few important limitations.

Quick Read

Learn Git Fast

A series of approachable, to-the-point tutorials on how to use Git.

Quick Read

A Layman's Introduction to Formal Grammar

For an easy-to-understand explanation of formal grammar in one concise essay, start with this.

Quick Read

A Deep Look at Font Rasterization

How fonts are rasterized and optimized for display (with an emphasis on open source tools).

Quick Read

How the Three Major JVMs Do Garbage Collection

No two JVMs do garbage collection the same way. Here is how the HotSpot, Oracle JRockit, and IBM JVMs differ in their approaches to GC — all in great detail.

Quick Read