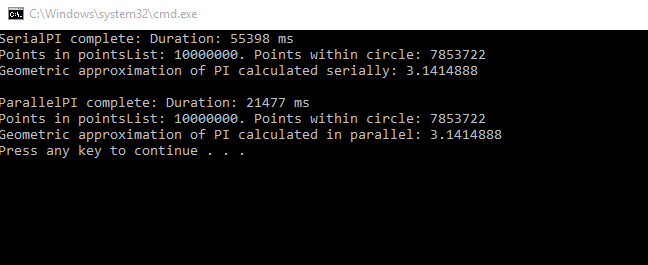
Student: Brian Johnston

Class:Cop2362

Professor Hamilton

Chapter 24

Print screen:



Code:

GraphDemo Code

using System;

using System.Diagnostics;

using System.IO;

using Windows.UI.ViewManagement;

using Windows.UI.Xaml;

using Windows.UI.Xaml.Controls;

using Windows.UI.Xaml.Media.Imaging;

using Windows.UI.Xaml.Navigation;

using System.Runtime.InteropServices.WindowsRuntime;

using System.Threading.Tasks;

using System.Threading;

// The Blank Page item template is documented at http://go.microsoft.com/fwlink/?LinkId=234238

namespace GraphDemo

{

/// <summary>

/// An empty page that can be used on its own or navigated to within a Frame.

/// </summary>

public sealed partial class GraphWindow : Page

{

// Reduce pixelWidth and pixelHeight if there is insufficient memory available

private int pixelWidth = 10500;

private int pixelHeight = 7000;

private WriteableBitmap graphBitmap = null;

private int bytesPerPixel = 4;

private byte[] data;

private byte redValue, greenValue, blueValue;

private CancellationTokenSource tokenSource = null;

public GraphWindow()

{

this.InitializeComponent();

Window.Current.SizeChanged += App.WindowSizeChanged;

int dataSize = bytesPerPixel \* pixelWidth \* pixelHeight;

data = new byte[dataSize];

graphBitmap = new WriteableBitmap(pixelWidth, pixelHeight);

}

private void cancelButton\_Click(object sender, RoutedEventArgs e)

{

if (tokenSource != null)

{

tokenSource.Cancel();

}

}

private async void plotButton\_Click(object sender, RoutedEventArgs e)

{

Random rand = new Random();

redValue = (byte)rand.Next(0xFF);

greenValue = (byte)rand.Next(0xFF);

blueValue = (byte)rand.Next(0xFF);

tokenSource = new CancellationTokenSource();

CancellationToken token = tokenSource.Token;

Stopwatch watch = Stopwatch.StartNew();

try

{

//call generateGraphDataAsync method asynchronously

await generateGraphDataAsync(data, 0, pixelWidth / 2, token);

generateGraphData(data, 0, pixelWidth / 2, token);

duration.Text = string.Format("Duration (ms): {0}", watch.ElapsedMilliseconds);

}

catch (OperationCanceledException oce)

{

duration.Text = oce.Message;

}

Stream pixelStream = graphBitmap.PixelBuffer.AsStream();

pixelStream.Seek(0, SeekOrigin.Begin);

pixelStream.Write(data, 0, data.Length);

graphBitmap.Invalidate();

graphImage.Source = graphBitmap;

}

private void generateGraphData(byte[] data, int partitionStart, int partitionEnd, CancellationToken token)

{

int a = pixelWidth / 2;

int b = a \* a;

int c = pixelHeight / 2;

for (int x = partitionStart; x < partitionEnd; x++)

{

int s = x \* x;

double p = Math.Sqrt(b - s);

for (double i = -p; i < p; i += 3)

{

token.ThrowIfCancellationRequested();

double r = Math.Sqrt(s + i \* i) / a;

double q = (r - 1) \* Math.Sin(24 \* r);

double y = i / 3 + (q \* c);

plotXY(data, (int)(-x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));

plotXY(data, (int)(x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));

}

}

}

private void plotXY(byte[] data, int x, int y)

{

int pixelIndex = (x + y \* pixelWidth) \* bytesPerPixel;

data[pixelIndex] = blueValue;

data[pixelIndex + 1] = greenValue;

data[pixelIndex + 2] = redValue;

data[pixelIndex + 3] = 0xBF;

}

private async Task generateGraphDataAsync(byte[] data, int partitionStart, int partitionEnd, CancellationToken token)

{

//Create Task okject that runs the generateGraphData method and uses the await operator to wait for the task to complete

Task task = Task.Run(() => generateGraphData(data, partitionStart, partitionEnd, token));

await task;

}

}

}

Plinq code

using System;

using System.Collections.Generic;

using System.Linq;

using System.Diagnostics;

using System.Threading;

using System.Globalization;

namespace PLINQ

{

class Program

{

public const int NUM = int.MaxValue / 1000;

static void Main(string[] args)

{

//Test1();

Test2();

}

public static void Test1()

{

Console.WriteLine("\nTest 1");

int[] numbers = new int[NUM];

Random random = new Random(999);

for (int i = 0; i < NUM; i++)

{

numbers[i] = random.Next(200);

}

// TO DO: Create a LINQ query that retrieves all numbers that are greater than 100

var over100 = from n in numbers.AsParallel() where TestIfTrue(n > 100) select n;

Stopwatch timer = new Stopwatch();

timer.Start();

// TO DO: Run the LINQ query and save the results in a List<int> object

List<int> numbersOver100 = new List<int>(over100);

long milliseconds = timer.ElapsedMilliseconds;

timer.Stop();

// TO DO: Display the results

Console.WriteLine("There are {0} numbers over 100.", numbersOver100.Count);

Console.WriteLine("Time taken was {0} ms", milliseconds);

}

public static bool TestIfTrue(bool expr)

{

Thread.SpinWait(1000);

return expr;

}

public static void Test2()

{

Console.WriteLine("\nTest 2");

try

{

// TO DO: Create a LINQ query that retrieves customers and orders from arrays

// Store each row returned in an OrderInfo object

var orderInfoQuery = from c in CustomersInMemory.Customers.AsParallel()

join o in OrdersInMemory.Orders.AsParallel()

on c.Split(',')[0] equals o.Split(',')[1]

select new OrderInfo

{

CustomerID = c.Split(',')[0],

CompanyName = c.Split(',')[1],

OrderID = Convert.ToInt32(o.Split(',')[0]),

OrderDate = Convert.ToDateTime(o.Split(',')[2],

new CultureInfo("en-US"))

};

Stopwatch timer = new Stopwatch();

timer.Start();

// TO DO: Run the LINQ query and save the results in a List<OrderInfo> object

List<OrderInfo> orderInfo = new List<OrderInfo>(orderInfoQuery);

long milliseconds = timer.ElapsedMilliseconds;

timer.Stop();

// TO DO: Display the results

Console.WriteLine("There are {0} orders", orderInfo.Count);

Console.WriteLine("Time taken for joining two arrays in memory: {0} ms", milliseconds);

}

catch (Exception ex)

{

Console.WriteLine("Error: {0}", ex.Message);

}

}

}

}

Calculate Pi

using System;

using System.Diagnostics;

using System.Collections.Generic;

using System.Threading;

using System.Threading.Tasks;

using System.Collections.Concurrent;

namespace CalculatePI

{

class Program

{

// Tuning constants:

// If you have lots of memory, increase NUMPOINTS to improve the accuracy

private const int NUMPOINTS = 10000000;

private const int RADIUS = 10000;

// Value to seed the random number generator for each calculation.

// Using the same seed value ensures that the same results should be generated each time

private const int SEED = 269222;

// If you have a very fast processor, increase SPINWAITS to show the effects of parallelization

private const int SPINWAITS = 1000;

private static double SerialPI()

{

List<double> pointsList = new List<double>();

Random random = new Random(SEED);

int numPointsInCircle = 0;

Stopwatch timer = new Stopwatch();

timer.Start();

try

{

for (int points = 0; points < NUMPOINTS; points++)

{

int xCoord = random.Next(RADIUS); //Generates a pair of coordinates that are in the range of the RADIUS

int yCoord = random.Next(RADIUS); //Generates a pair of coordinates that are in the range of the RADIUS

double distanceFromOrigin = Math.Sqrt(xCoord \* xCoord + yCoord \* yCoord); //Use Pythagorean theorem to calulate distance

pointsList.Add(distanceFromOrigin); //Add the result to the collection

doAdditionalProcessing(); //Call method

}

//Iterates through pointsList collection and examines if value is less than or equal to radius and increments numPointsInCircle

foreach (double datum in pointsList)

{

if (datum <= RADIUS)

{

numPointsInCircle++;

}

}

double pi = 4.0 \* numPointsInCircle / NUMPOINTS; //Calculates pi based on the ratio of the number of points that lie within the circle to the total

return pi;

}

finally

{

long milliseconds = timer.ElapsedMilliseconds;

Console.WriteLine("SerialPI complete: Duration: {0} ms", milliseconds);

Console.WriteLine("Points in pointsList: {0}. Points within circle: {1}", pointsList.Count, numPointsInCircle);

}

}

private static double ParallelPI()

{

ConcurrentBag<double> pointsList = new ConcurrentBag<double>();

Random random = new Random(SEED);

int numPointsInCircle = 0;

Stopwatch timer = new Stopwatch();

timer.Start();

try

{

//Lambda expression to run task in parallel

Parallel.For(0, NUMPOINTS, (x) =>

{

int xCoord;

int yCoord;

//Protect calls to random with lock

lock (pointsList)

{

xCoord = random.Next(RADIUS);

yCoord = random.Next(RADIUS);

}

double distanceFromOrigin = Math.Sqrt(xCoord \* xCoord + yCoord \* yCoord); //Use Pythagorean theorem to calulate distance

pointsList.Add(distanceFromOrigin); //Add the result to the collection

doAdditionalProcessing(); //Call method

});

//Iterates through pointsList collection and examines if value is less than or equal to radius and increments numPointsInCircle

foreach (double datum in pointsList)

{

if (datum <= RADIUS)

{

numPointsInCircle++;

}

}

double pi = 4.0 \* numPointsInCircle / NUMPOINTS; //Calculates pi based on the ratio of the number of points that lie within the circle to the total

return pi;

}

finally

{

long milliseconds = timer.ElapsedMilliseconds;

Console.WriteLine("ParallelPI complete: Duration: {0} ms", milliseconds);

Console.WriteLine("Points in pointsList: {0}. Points within circle: {1}", pointsList.Count, numPointsInCircle);

}

}

private static void doAdditionalProcessing()

{

Thread.SpinWait(SPINWAITS);

}

static void Main(string[] args)

{

double pi = SerialPI();

Console.WriteLine("Geometric approximation of PI calculated serially: {0}", pi);

Console.WriteLine();

pi = ParallelPI();

Console.WriteLine("Geometric approximation of PI calculated in parallel: {0}", pi);

}

}

}