Programming III

Centennial College

Week#11 - ONLINE 2020
Winter

Topic: Parallel Programming

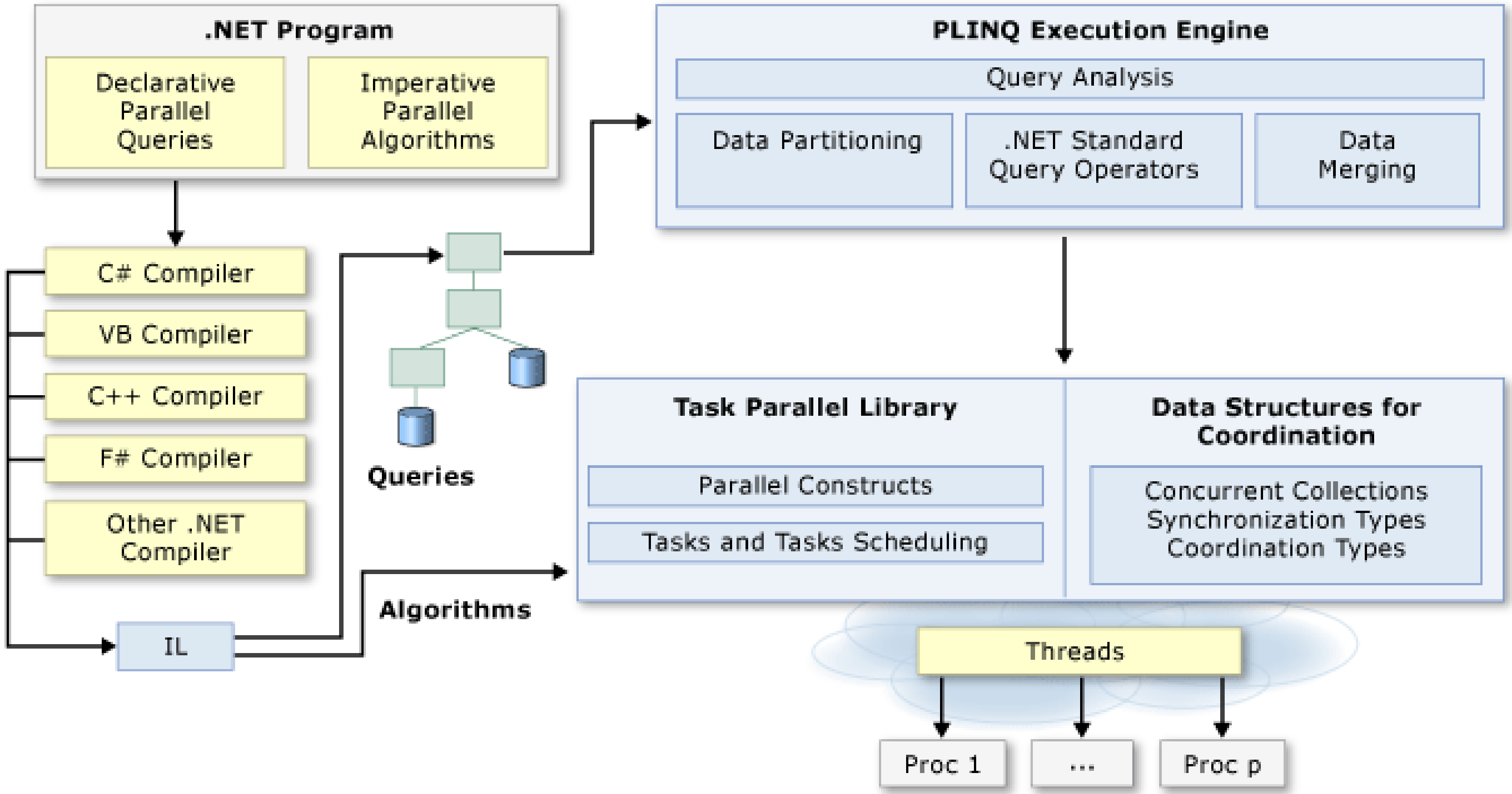
Parallel Programming

- Programming to leverage multicores or multiple processors is called parallel programming.
- Parallel programming is the general discipline of doing multiple computations in parallel, each of which is doing some sub computation independently of the larger single problem
- It is a subset of the broader concept of multithreading

Parallel Programming(Con't)

- Multithreading is the approach of using multiple threads of execution to process different operations, e.g., if you have two things to do, use one thread to do one and another thread to do the other
- An operating system is to handle thread execution in available cores
- .NET parallel API's take maximum advantage of available CPU resources

.NET Parallel Programming Architecture



https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/index

Processes and Threads

- A process is an executing program. OS uses processes to separate the applications that are being executed
- A thread is the basic unit to which OS allocate processor time.
- Multiple threads can run in the context of a process. All threads of a process share its virtual address space

Parallel LINQ(PLINQ)

- The implementation of the LINQ to Objects extension methods that parallelizes the operations

```
// create array with size of 1 million of random integers in the range 1-999
int[] values = Enumerable.Range(1, 10000000)
                         .Select(x => random.Next(1, 1000))
                         .ToArray();

// time the Min, Max and Average LINQ extension methods
Console.WriteLine("Min, Max and Average with LINQ to Objects using a single core");
var linqStart = DateTime.Now; // get time before method calls
var linqMin = values.Min();
var linqMax = values.Max();
var linqAverage = values.Average();
var linqEnd = DateTime.Now; // get time after method calls

// display results and total time in milliseconds
var linqTime = linqEnd.Subtract(linqStart).TotalMilliseconds;
DisplayResults(linqMin, linqMax, linqAverage, linqTime);

// time the Min, Max and Average PLINQ extension methods
Console.WriteLine("\nMin, Max and Average with PLINQ using multiple cores");
var plinqStart = DateTime.Now; // get time before method calls
var plinqMin = values.AsParallel().Min();
var plinqMax = values.AsParallel().Max();
var plinqAverage = values.AsParallel().Average();
var plinqEnd = DateTime.Now; // get time after method calls

// display results and total time in milliseconds
var plinqTime = plinqEnd.Subtract(plinqStart).TotalMilliseconds;
DisplayResults(plinqMin, plinqMax, plinqAverage, plinqTime);

// display time difference as a percentage
Console.WriteLine("\nPLINQ took " +
   $"{((linqTime - plinqTime) / linqTime):P0}" +
   " less time than LINQ");
```

Parallel LINQ(PLINQ)

- The implementation of the LINQ to Objects extension methods that parallelizes the operations

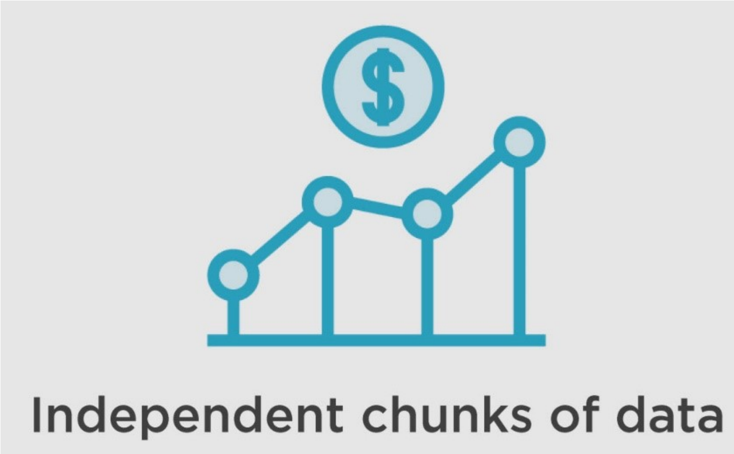
```
// create array with size of 1 million of random integers in the range 1-999
int[] values = Enumerable.Range(1, 10000000)
                         .Select(x => random.Next(1, 1000))
                         .ToArray();

// time the Min, Max and Average LINQ extension methods
Console.WriteLine("Min, Max and Average with LINQ to Objects using a single core");
var linqStart = DateTime.Now; // get time before method calls
var linqMin = values.Min();
var linqMax = values.Max();
var linqAverage = values.Average();
var linqEnd = DateTime.Now; // get time after method calls

// display results and total time in milliseconds
var linqTime = linqEnd.Subtract(linqStart).TotalMilliseconds;
DisplayResults(linqMin, linqMax, linqAverage, linqTime);

// time the Min, Max and Average PLINQ extension methods
Console.WriteLine("\nMin, Max and Average with PLINQ using multiple cores");
var plinqStart = DateTime.Now; // get time before method calls
var plinqMin = values.AsParallel().Min();
var plinqMax = values.AsParallel().Max();
var plinqAverage = values.AsParallel().Average();
var plinqEnd = DateTime.Now; // get time after method calls

// display results and total time in milliseconds
var plinqTime = plinqEnd.Subtract(plinqStart).TotalMilliseconds;
DisplayResults(plinqMin, plinqMax, plinqAverage, plinqTime);

// display time difference as a percentage
Console.WriteLine("\nPLINQ took " +
   $"{((linqTime - plinqTime) / linqTime):P0}" +
   " less time than LINQ");
```

Task Parallel Library (TPL)

- TPL is a set of software API to implement parallel processing, it is originally introduced with .NET Framework 4.0
- TPL is to make developers more productive by simplifying the process of adding parallelism and concurrency to applications
- TPL scales the degree of concurrency dynamically to most efficiently use all the processors that are available
- TPL handles the partition of the work, the scheduling of threads on the ***ThreadPool***, cancellation support and state management
- It is in the ***System.Threading.Tasks*** namespace

Data Parallelism

- The same operation is performed concurrently on elements in a collection
- The source collection is partitioned so that multiple threads can operate on different segments concurrently



Task Parallelism

- It refers to one or more independent tasks running concurrently
- A task represents an asynchronous operation, and in some ways it resembles the creation of a new thread or ***ThreadPool*** work item
- ***Parallel*** class and ***PLINQ*** are internally built on the task parallelism constructs. Task parallelism is the lowest-level approach to parallelism

```
// Fig. 21.9: ParallelizingWithPLINQ.cs
// Comparing performance of LINQ and PLINQ Min, Max and Average methods.
using System;
using System.Linq;

class ParallelizingWithPLINQ
{
   static void Main()
   {
      var random = new Random();

      // create array of random ints in the range 1-999
      int[] values = Enumerable.Range(1, 10000000)
                        .Select(x => random.Next(1, 1000))
                        .ToArray();

```

Fig. 21.9 | Comparing performance of LINQ and PLINQ `Min`, `Max` and `Average` methods. (Part 1 of 6.)

```
         // time the Min, Max and Average LINQ extension methods
         Console.WriteLine(
            "Min, Max and Average with LINQ to Objects using a single core");
         var linqStart = DateTime.Now; // get time before method calls
         var linqMin = values.Min();
         var linqMax = values.Max();
         var linqAverage = values.Average();
         var linqEnd = DateTime.Now; // get time after method calls

         // display results and total time in milliseconds
         var linqTime = linqEnd.Subtract(linqStart).TotalMilliseconds;
         DisplayResults(linqMin, linqMax, linqAverage, linqTime);

```

Fig. 21.9 | Comparing performance of LINQ and PLINQ `Min`, `Max` and `Average` methods. (Part 2 of 6.)

```
      // time the Min, Max and Average PLINQ extension methods
      Console.WriteLine(
         "\nMin, Max and Average with PLINQ using multiple cores");
      var plinqStart = DateTime.Now; // get time before method calls
      var plinqMin = values.AsParallel().Min();
      var plinqMax = values.AsParallel().Max();
      var plinqAverage = values.AsParallel().Average();
      var plinqEnd = DateTime.Now; // get time after method calls

      // display results and total time in milliseconds
      var plinqTime = plinqEnd.Subtract(plinqStart).TotalMilliseconds;
      DisplayResults(plinqMin, plinqMax, plinqAverage, plinqTime);

      // display time difference as a percentage
      Console.WriteLine("\nPLINQ took " +
         $"{((linqTime - plinqTime) / linqTime):P0}" +
         " less time than LINQ");
   }
```

Fig. 21.9 | Comparing performance of LINQ and PLINQ `Min`, `Max` and `Average` methods. (Part 3 of 6.)

```
    // displays results and total time in milliseconds
    static void DisplayResults(
       int min, int max, double average, double time)
    {
       Console.WriteLine($"Min: {min}\nMax: {max}\n" +
          $"Average: {average:F}\nTotal time in milliseconds: {time:F}");
    }
 }
```

Fig. 21.9 | Comparing performance of LINQ and PLINQ `Min`, `Max` and `Average` methods. (Part 4 of 6.)

```
Min, Max and Average with LINQ to Objects using a single core
Min: 1
Max: 999
Average: 499.96
Total time in milliseconds: 179.03

Min, Max and Average with PLINQ using multiple cores
Min: 1
Max: 999
Average: 499.96
Total time in milliseconds: 80.99

PLINQ took 55 % less time than LINQ
```

Fig. 21.9 | Comparing performance of LINQ and PLINQ `Min`, `Max` and `Average` methods. (Part 5 of 6.)

```
Min, Max and Average with LINQ to Objects using a single core
Min: 1
Max: 999
Average: 500.07
Total time in milliseconds: 152.13

Min, Max and Average with PLINQ using multiple cores
Min: 1
Max: 999
Average: 500.07
Total time in milliseconds: 89.05

PLINQ took 41 % less time than LINQ
```

Fig. 21.9 | Comparing performance of LINQ and PLINQ `Min`, `Max` and `Average` methods. (Part 6 of 6.)

Threading Issue

- Starting multiple threads that access the same data, you can get intermittent problems, data synchronization, that are hard to find
- These problems are the same whether you use Task, Parallel LINQ, or Parallel Class
  - Race conditions
  - deadlock

Good Practice for Lock

- Minimize the amount of code and computation inside a locked context
- Only lock exactly the amount of time you really need to
- Lock with caution, or else you might end up deadlocking

Data Structure for Parallel Programming

- .NET framework 4.0 introduces several new types that are useful in parallel programming, including a set of concurrent collection classes, lightweight synchronization primitives, and types for lazy initialization
  - ConcurrentBag<T> : use it rather than List<T> as List<T> is not thread safe
  - BlockingCollection<T>
  - ConcurrentDictionary<T>
  - ConcurrentStack<T>
  - ConcurrentQueue<T>
- Use these types with any multithreaded application code, including the TPL and PLINQ

Asynchronous vs Parallel

- Asynchronous programming is a bit more general in that it has to do with latency (something on which your app has to wait, for one reason or another); whereas multithreaded programming is a way to achieve parallelization (one or more things that your application has to do at the same time)
- These two topics are closely related with each. An application that performs work on multiple threads in parallel will often need to wait until such work is completed in order to take some action (e.g. update the user interface)
- Parallel.For, Parallel.ForEach and Invoke could be wrapped in Task.Run to relieve the UI thread of work

Reference

- https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/index
- https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread?view=netframework-4.7.2
- https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1?view=netframework-4.7.2
- https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/data-parallelism-task-parallel-library
- https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallelloopresult?view=netframework-4.7.2
- https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/data-structures-for-parallel-programming