Parallel Programming

Estimated time for completion: 60 minutes

Overview:

In this lab you will be using the Parallel extensions added to .NET 4 to parallelize a piece of sequential code.

Goals:

- Learn how to use Parallel loops
- Refactor algorithms to provide better parallel performance.

Lab Notes:

This lab will be using Visual Studio 2010, and requires a machine with at least two real cores.

Part 1: Running the sequential code.

In the first part of the lab you will run some code that attempts to solve the Travelling Salesman problem. The program attempts to find the shortest path between all the cities the traveler has to visit, commonly called brute force were by every route is evaluated, resulting in N! combinations to be considered . You will therefore see that as you increase the number of cities for the salesman to visit that the compute required increases, exponentially.

Steps :

1. Open the project TravellingSalesman.sln from the before directory
2. Familiarize yourself with the Program.cs inside the TravellingSalesmanApp project. Try modifying the number of wayPoints used by changing the constant **wayPoints** value in Main, up to a maximum of 15, and get some idea of how computationally intensive the application is.

Part 2: Taking advantage of multiple cores.

In this second part your task will be to produce a new method for calculating the shortest path but now taking advantage of multiple cores.

Steps:

1. Take a copy of the method `SequentialShortestPath` and paste it back into the code renaming it to `ParallelShortestRoute`. Add a method call in main to invoke

`CalculateShortestRoute` with `ParallelShortestPath` below `CalculateShortestRoute` call for the sequential algorithm.

2. Now you need to work on `ParallelShortestPath` method so it does indeed run in parallel. For this job you will use `Parallel.ForEach`, by replacing the sequential foreach with a `Parallel.ForEach`. You will need to use the variant of `Parallel.ForEach` that allows you to have some local state if you are to reduce the amount of synchronization necessary when evaluating a route to determine if it is the shortest so far. Also remember that at the point of merging a local result with a global result to include some form of synchronization.

```
public static ParallelLoopResult ForEach<TSource,
   TLocal>(IEnumerable<TSource> source, Func<TLocal> localInit, Func<TSource,
   ParallelLoopState, TLocal, TLocal> body, Action<TLocal> localFinally);
```

(Link to above method documentation)

3. Now run the code and confirm that the shortest distance is the same for both sequential and parallel versions, once happy increase the number of way points to about 10. You should see a speed improvement; the exact improvement will be a function of your machine architecture. However irrespective of your architecture you won't be seeing a speed improvement you would have hoped for. Why?

Part 3: A more optimal parallel version

In the previous part you successfully managed to parallelize some sequential code. The process was relatively trivial and produced some level of speed up. As is often the case turning a sequential foreach into a Parallel.ForEach doesn't always yield the fully expected gain, we often need to look at the algorithm closer to determine exactly what needs to be parallelized, in this example you have only really parallelized appx. 50% of the workload. In the above case whilst the evaluating of each trip is performed in parallel the computation to determine all the trip permutations is in fact happening sequentially. Each of the parallel tasks will have to queue up waiting to be given a set of trips to evaluate. So in this part you will attempt to parallelize more of the code to include the computation of the trip permutations.

Steps :

1. The **HeapPermute** method produces all permutations. For example, assuming you had three way points to visit that would be 3!, six combinations.
   1 ,2, 3
   1, 3, 2
   2,1,3
   2,3,1
   3,1,2
   3,2,1
   To parallelize this piece of code we could have three tasks each one always starting its permutation with a unique value, and thus the remaining permutations for a given start point could be performed in parallel. This is known as Geometric Decomposition, and works well when the amount of work per task is equal, and each task can run completly

independent of each other.

2. To achieve this take a copy of `TwoLoopsSequentialShortesPath` code and call the new method `OptimisedParallelShortestPath`. You will notice that the algorithm has been modified to exhibit the behavior described above. There is now two loops, an outer for loop, and an inner foreach. Each iteration of the outer loop represents a possible first way point, the inner loop then computes all possible trips for that given first way point. You will now parallelize the outer for loop using `Parallel.For` thus maximizing the amount of code that is running in parallel.
3. Add a another call to CalculateShortesRoute to invoke OptimisedParallelShortestPath, you should see a speed up close to the number of cores in your machine.

Part 4: Monte Carlo

Mote Carlo

If you have not done so already trying modifying the length of the trip by adjusting the ***nWayPoints*** *constant in main to a value greater than 10 but no more than 15.You should notice that even with the fully optimized parallel version its taking a while. In fact a 15 city trip took 18 days to calculate the shortest route on an 8 core machine, this type of problem is known as an NP Complete. Consider the travelling salesman problem, waiting 18 days for a route is probably not acceptable, getting the best answer you can in a given period of time would perhaps be more acceptable, the more guesses we can make in a given period of time the better our chance of getting a good route. This technique is known as Mote Carlo...and strange as it may sound it is commonly used in the financial world.*

Steps:

1. Set the **nWayPoints** constant in **Main** to 15.
2. Comment out any calls you have to `CalculateShortestRoute.`
3. The method `GetShortestTestTrip` returns the shortest route for the given test trip, this was achieved via brute force over 18 days. You will attempt to build a method that trys to get the same result but in a fraction of the time by simply guessing. Create a method call `GuessShortestRoute` that takes a trip and a cancellation token. The method then simply creates random variations keeping hold of the best guess it has made. When the cancellation token is signaled it returns the best guess it has so far. Below is some code that code be used to invoke the guessing method. It creates a task per core :-

```
CancellationTokenSource stopGuessing = new CancellationTokenSource();

 Task<Trip>[] tasks = new Task<Trip>[Cores.PhysicalCores];
 for (int nTask = 0; nTask < tasks.Length; nTask++)
 {
     tasks[nTask] = Task.Factory.StartNew<Trip>(() =>
     {
         return GuessShortestRoute(route, stopGuessing.Token);
     });
 }
 Thread.Sleep(2000);  // Wait two seconds before asking for best guess
 stopGuessing.Cancel();
 Trip shortestTrip = tasks
      .Select(t => t.Result)
      .OrderBy(t => t.TotalDistance).First();
```

4. A simple implementation of GuessShortestRoute may look like this :-

```
private static Trip GuessShortestRoute(Trip trip,
 CancellationToken cancellationToken)
{
   MapLocation[] points = trip.WayPoints.ToArray();

   Trip shortestTrip = trip;
   Random rnd = new Random();

   while (!cancellationToken.IsCancellationRequested)
   {
     for (int nSwap = 0; nSwap < points.Length; nSwap++)
     {
        Swap(ref points[nSwap], ref points[rnd.Next(points.Length)]);
     }

      Trip candidateTrip = new Trip(trip.Start, trip.End, points);
      if (candidateTrip.TotalDistance < shortestTrip.TotalDistance)
      {
         shortestTrip = candidateTrip;
      }
    }

   return shortestTrip;
}

private static void Swap<T>(ref T lhs, ref T rhs)
{
   T temp = lhs;
   lhs = rhs;
   rhs = temp;
}
```

5. By increasing the amount of time you are prepared to wait for a result the better the chance of getting a good result (but not guaranteed). The great thing about this approach

is that as you add more cores to the problem the better the result will be for a constant period of time X. Since each task is running independently in parallel, and thus the algorithm scales.

6. The GuessShortestRoute method above is very crude can you think of ways to improve it?

Solutions

after\TravellingSalesman.sln