Task Parallel Library in .NET

- This lecture provides a very quick introduction to the Task Parallel Library (TPL); it is similar to OpenMP in many ways
- The initial colorful slides are from Joe Hummel, High-Performance Computing with .NET and HPCS (SIGCSE 2009)
- The remaining examples are from an online tutorial **`http://msdn.microsoft.com/en-us/magazine/cc163340.aspx`**

Task Parallel Library (TPL)

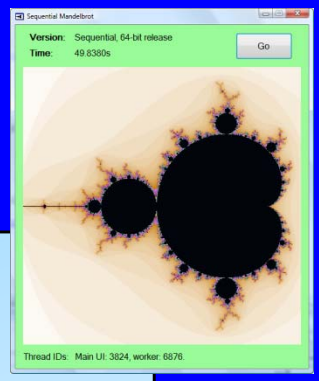
- Target:

  "Developers looking for a more productive framework in .NET for multi-threading"

- Status: beta
- Pros: higher programmer productivity
- Cons: frameworks offer limited set of features
- Limitations: shared-memory; .NET

Philosophy

- Think in terms of tasks, not threads…
  - Expose as much parallelism as you can for scalability
- Create many more tasks than cores
  - 10x more?




```
Task[,] tasks = new Task[pixels, pixels];

for (int yp = 0; yp < pixels; yp++)
  for (int xp = 0; xp < pixels; xp++)
    tasks[yp,xp] = Task.Create(MandelbrotColor, ...);

Task.WaitAll(tasks);
```

Parallel.For / Foreach / Invoke

- Turns a set of iterations / blocks into a set of tasks:

```
Parallel.For(0, N, loopbody);    // iterations in parallel:

Parallel.ForEach(datastructure, loopbody);    // elements in parallel:

Parallel.Invoke( {code1, code2, ...} );    // code blocks in parallel:
```

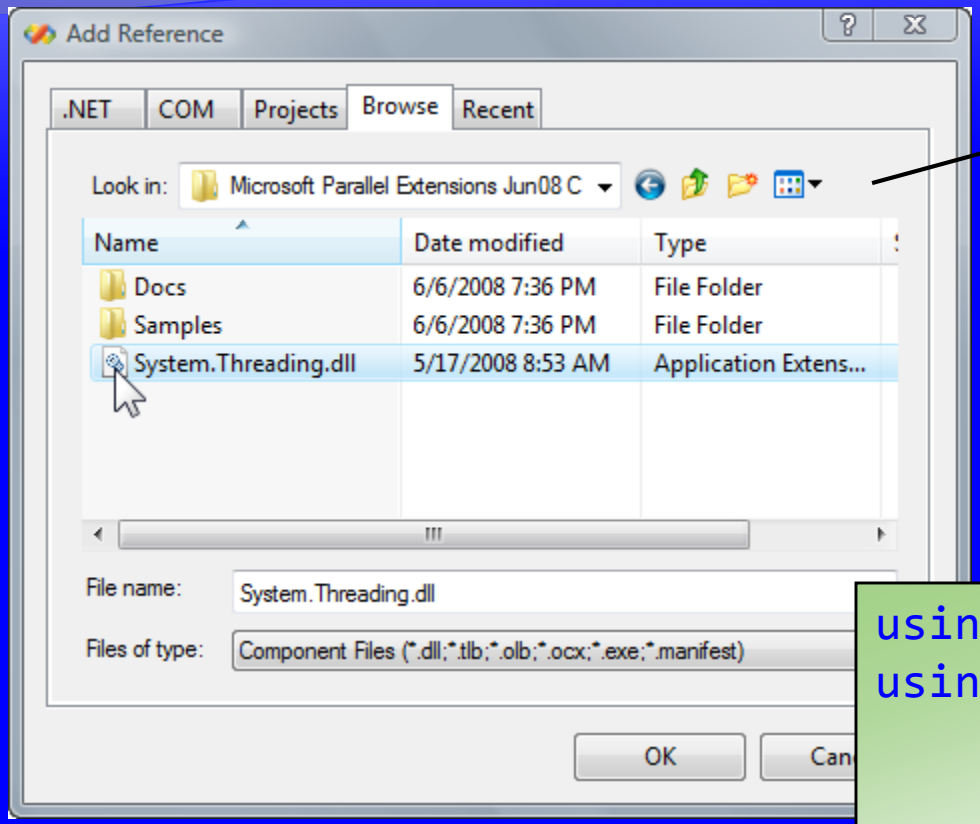
- Tasks are assigned to threads from a thread pool
- Each thread has local work queue (local => better caching)
- Threads "work-steal" from others to balance workload

Requires installation of…

- Parallel Extensions to the.NET Framework 3.5
  - Also known as *PFx*
  - Freely available
  - http://msdn.microsoft.com/en-us/concurrency/default.aspx

Visual Studio usage

- Reference Threading.dll, import namespaces:



Project menu, Add Reference…, Browse tab, navigate to C:\Program Files (x86)\Microsoft Parallel Extensions\, select .DLL

```
using System.Threading;
using System.Threading.Tasks;
```

Concurrent collections

- PFx provides thread-safe data structures:

```
using System.Threading.Collections;
  :
ConcurrentQueue<int>    q = new ConcurrentQueue<int>();
ConcurrentStack<object> s = new ConcurrentStack<object>();

Parallel.For(0, 1000, i =>
  {
     q.Enqueue(i);
     .
     .
     .
     int  j;
     bool b = q.TryDequeue(out j);
     if (b)
       DoComputation(j);
  }
);
```

A Simple Example

- Squaring the elements of an array

```
for (int i = 0; i < 100; i++) {
  a[i] = a[i]*a[i];
}
```

- A parallelized version

```
Parallel.For(0, 100, delegate(int i) {
 a[i] = a[i]*a[i];
});
```

- You must import

```
using System.Threading;
```

Parallelizing Matrix Multiplication - 1

- Here is the traditional algorithm using triply nested for loops that have not been parallelized

```
void SeqMatrixMult(int size, double[,] m1, double[,]
  m2, double[,] result) {
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      result[i, j] = 0;
      for (int k = 0; k < size; k++) {
        result[i, j] += m1[i, k] * m2[k, j];
      }
    }
  }
}
```

Parallelizing Matrix Multiplication - 2

- Here is the parallelized version

```
void ParMatrixMult(int size, double[,] m1,
  double[,] m2, double[,] result) {
   Parallel.For( 0, size, delegate(int i) {
     for (int j = 0; j < size; j++) {
       result[i, j] = 0;
       for (int k = 0; k < size; k++) {
         result[i, j] += m1[i, k] * m2[k, j];
       }
     }
   });
}
```

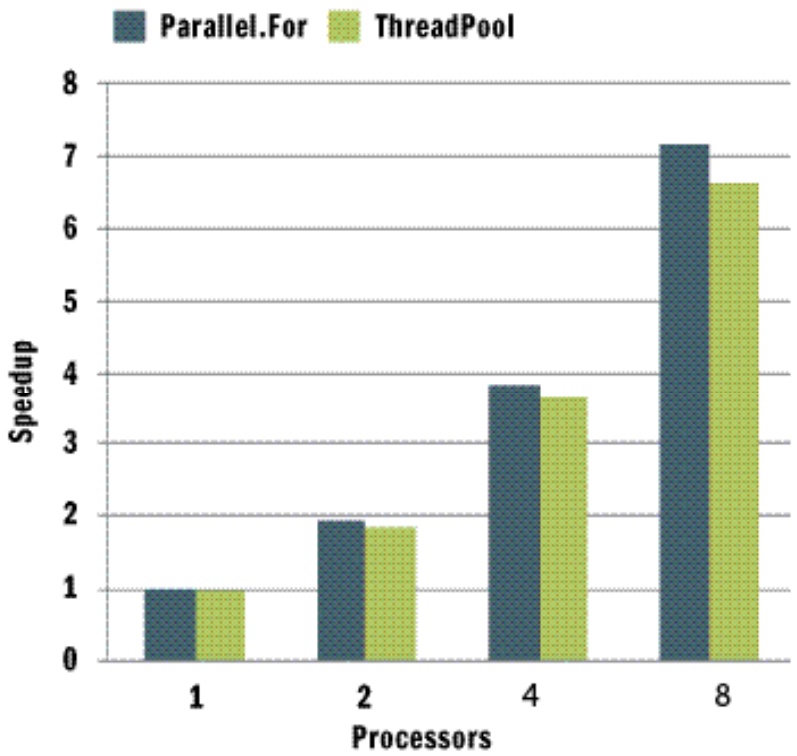
Parallelizing Matrix Multiplication - 3

- Getting even more parallelism; unlike OpenMP using TPL we can parallelize nested loops

```
Parallel.For( 0, size, delegate(int i) {
  Parallel.For( 0, size, delegate(int j) {
    result[i, j] = 0;
    for (int k = 0; k < size; k++) {
      result[i, j] += m1[i, k] * m2[k, j];
    }
  });
});
```

Some Test Data

- The figure below shows the results of some testing.
  - parallelizing the outer loop of a matrix multiplication with 750x750 elements
  - The tests were conducted on a four-socket dual-core machine with 3GB of memory



Using Tasks

- Structure of a typical task

```
class Task {
  Task( Action action );
  void Wait();
  void Cancel();
  bool IsCompleted { get; }
   ...
}
```

- A task is created by supplying an associated action that can potentially be executed in parallel
- You can cancel the task and all tasks created in its associated actions (child tasks) by calling Cancel
- Tasks are an improved thread pool where work items return a handle that can be canceled or waited upon, and where exceptions are propagated

Consider this Tree Application

```
class Node : Tree {
  int depth;  // The depth of the tree
  Tree left;  // The left sub tree
  Tree right; // The right sub tree ...
}

class Leaf : Tree {
    int value; // values are stored in the leafs ...
}

override int Sum() {
  int l = left.Sum();
  int r = right.Sum();
  return (r + l);
}
```

Parallelization using Futures

- ```
  override int Sum(){
    Task<int> l = new Task<int>( left.Sum );
    int r = right.Sum();
    return (r + l.Value);
  }
  ```
- This illustrates the use for “futures”
  - For each left subtree, we create a new future of type int, passing a delegate as the constructor argument.
  - In this sample, we pass the sum method of the left child, left.Sum, without calling it.
  - We continue by calculating the sum of the right subtree.
  - By creating the future, other processors could potentially start evaluating the sum of the left subtree in parallel.
  - In the end, we request the value of the future using the Value property.
  ```

Exercise: Parallelizing a Correlation Calculation

- You are given a C# program that calculates the correlation between two arrays of values
  - You are given the methods to calculate Sxx and Syy
  - You need to complete the method that calculates Sxy
  - Run your program sequentially to make sure it works
- Next you will parallelize this program using the Task Parallel Library
  - You will parallelize the calculations of Sxx, Syy, and Sxy by using the Parallel.Invoke method
  - Run your program and see if you get any speedup