PLINQ and Office Add-ins

Donny Amalo

Parallel Computing Platform Group

Microsoft Corporation

Contents

Introduction

Since the launch of Visual Studio Tools for Office (VSTO) on the .NET Framework 2.0, writing Office add-ins has become much simpler and easier, at least compared to using the COM model (yikes). Now, combined with the parallel computing features built into the .NET Framework 4, we can build parallelized Office add-ins.

Building Office add-ins that take advantage of parallelism is in many cases straightforward. In this document, we will demonstrate this through two Office add-in implementations. The first is an Excel add-in that performs a Monte Carlo simulation, and the second is a Word add-in that implements an alternative spell checker. For these examples, we will use Parallel LINQ to Objects (PLINQ) to enable the parallel computation.

Building Office add-ins, we need to keep in mind two key challenges:

- The UI must be responsive.
- UI elements can only be updated from the main thread.

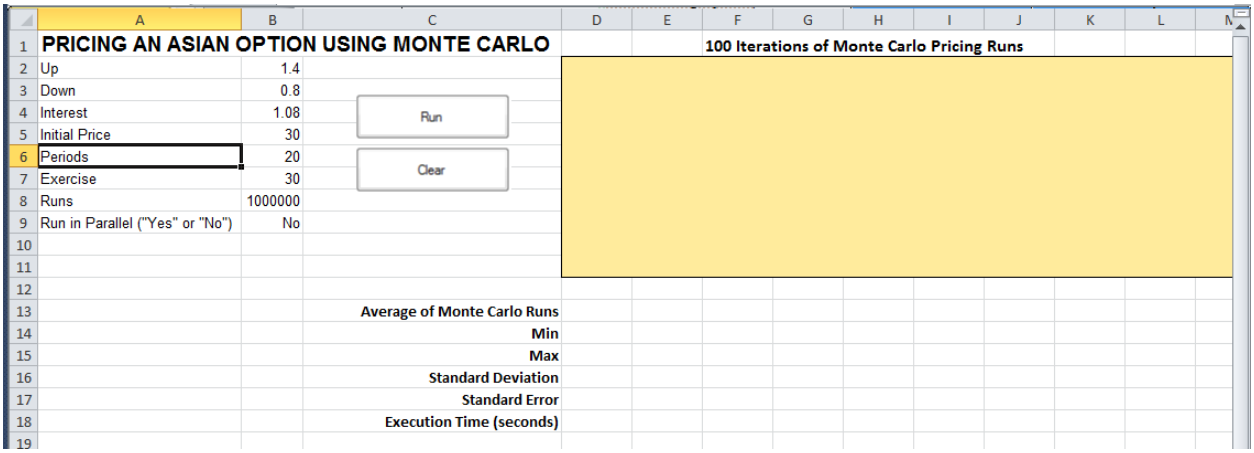
Monte-Carlo Simulation for Excel

Let's look at how to create a parallelized add-in for Excel. In this example, we use PLINQ to implement an Asian Option pricing add-in using a Monte Carlo algorithm. The algorithm computes the value of a financial instrument called an *option* by simulating the price paths over certain periods of time. We then calculate the payoff value of each price path. To run this simulation, we need to have the following as inputs:

- **Up/Down:** parameters to model the stock price over time. Every simulation period (e.g. every day), the stock price is multiplied by either the Up or the Down factor, to simulate movement of the price.
- **Interest:** the projected rate of return of a similar investment.
- **Initial Price:** the price of the stock at the beginning of the simulation.
- **Periods:** the number of time periods over which to calculate the payoff.
- **Exercise:** the strike price of the option.

We calculate the payoff over a specified number of runs, on each run following one randomly chosen price path. We do these calculations thousands of times, which then give us the average payoff. This is a computationally intensive process, one that's very parallelizable.

To implement this Monte Carlo simulation in Excel, we start by creating a new Workbook project. Then, we allocate cells to hold the input parameters of the simulation: Initial Price, Interest, Periods, etc. We also need a button to start the calculations. The resulting sheet will look something like the following:



When the user starts the calculation by clicking the Run button, we need to read the input parameters from the Excel sheet. We can do that with this code:

```
// Get data from the form
double up       = (double)this.Range["B2"].Value2;
double down     = (double)this.Range["B3"].Value2;
double interest = (double)this.Range["B4"].Value2;
double initial  = (double)this.Range["B5"].Value2;
int periods     = Convert.ToInt32(this.Range["B6"].Value2);
double exercise = (double)this.Range["B7"].Value2;
int runs        = Convert.ToInt32(this.Range["B8"].Value2);
```

Keep in mind Excel cell must be updated from the main thread. In order to parallelize the workload, we will compute the option prices in parallel on ThreadPool threads, and then write the results back to the sheet sequentially.

The core algorithm to compute the option pricing can be expressed as a LINQ query:

```
string[] columns = { "D", "E", "F", "G", "H", "I", "J", "K", "L", "M" };
int[] rows = { 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };

var cells = from column in columns
            from row in rows
            select new { column, row };

results = from cell in cells
          let price = PriceAsianOptions(
               initial,
               exercise,
               up,
               down,
               interest,
               periods,
               runs)
          select new PricingResult
          {
              Price = price,
              Column = cell.column,
              Row = cell.row
          };
```

To make this query run in parallel, we add AsParallel() to it:

```
results = from cell in
               cells.AsParallel().WithMergeOptions(ParallelMergeOptions.NotBuffered)
          let price = PriceAsianOptions(
               initial,
               exercise,
               up,
               down,
               interest,
               periods,
               runs)
          select new PricingResult
          {
              Price = price,
              Column = cell.column,
              Row = cell.row
          };
```

Notice the WithMergeOptions call in the query that sets the NotBuffered option. By default, PLINQ accumulates results in a buffer, and only yields them once a certain number of output elements has been produced. This reduces the overhead in the query, but increases the time to produce the first element. Since we want to update the Excel spreadsheet immediately as results are produced, we set the NotBuffered merge option.

We can then write the results into the sheet like so:

```
foreach (var result in results)
{
    this.Range[string.Format("{0}{1}", result.Column, result.Row)].Value2 = result.Price;
    min = Math.Min(min, result.Price);
    max = Math.Max(max, result.Price);
    sumPrice += result.Price;
    sumSquarePrice += result.Price * result.Price;
    count++;
    stdDev = Math.Sqrt(sumSquarePrice - sumPrice * sumPrice / count) /
                ((count == 1) ? 1 : count - 1);
    stdErr = stdDev / Math.Sqrt(count);
    Application.Calculate();
}
```

This is what the spreadsheet looks like after running the program:

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	**PRICING AN ASIAN OPTION USING MONTE CARLO**					**100 Iterations of Monte Carlo Pricing Runs**							
2	Up	1.4		9.559098	9.52876	9.545733	9.528731	9.501127	9.535135	9.510579	9.545241	9.535781	9.534127
3	Down	0.8		9.559098	9.52876	9.543535	9.546995	9.540657	9.550124	9.523286	9.499004	9.528134	9.515254
4	Interest	1.08	Run	9.559098	9.562327	9.528526	9.539052	9.504135	9.54878	9.5266	9.52777	9.522893	9.546075
5	Initial Price	30		9.559098	9.555992	9.54985	9.509489	9.537574	9.530241	9.536569	9.526825	9.550991	9.527727
6	Periods	20	Clear	9.559098	9.516583	9.534303	9.535614	9.529451	9.547809	9.503141	9.534727	9.527477	9.51812
7	Exercise	30		9.559098	9.545011	9.520618	9.553373	9.526505	9.553039	9.520681	9.533494	9.510697	9.55525
8	Runs	1000000		9.559098	9.518957	9.520618	9.515906	9.528104	9.501924	9.532655	9.491242	9.542221	9.547045
9	Run in Parallel ("Yes" or "No")	Yes		9.559098	9.512771	9.526834	9.526645	9.524338	9.503693	9.520681	9.538077	9.519245	9.524649
10				9.52876	9.512771	9.526834	9.531655	9.551496	9.542152	9.532655	9.505778	9.549054	9.547045
11				9.52876	9.545733	9.50981	9.513536	9.515631	9.536347	9.543851	9.562246	9.516796	9.524649
12													
13			**Average of Monte Carlo Runs**	9.532322									
14			**Min**	9.491242									
15			**Max**	9.562327									
16			**Standard Deviation**	0.001689									
17			**Standard Error**	0.000169									
18			**Execution Time (seconds)**	14.70936									

On an eight-core machine, our Excel add-in can get about 7.5x speedup.

A Spellchecker Add-In for Word

To demonstrate how to develop a parallel add-in for Word, we implemented a spellchecker based on Levenshtein distance. This example uses the same spell checking algorithm as the SpellCheck application posted in the "Samples for Parallel Programming with the .NET Framework 4" at http://code.msdn.microsoft.com/ParExtSamples. Word – of course – already has a high-quality spellchecker, one that is often significantly faster than the algorithm we present here. Our implementation is presented here merely as an illustration of what can be done as an add-in, or as an extension to existing ones.

To begin, we load an English word list, and then read the word to spell check. For this example, our word list is a simple flat text file with one word per line. We will use the File.ReadAllLines() method to load the entire word list into an IList<string>. And, the code below gives us the word selected in the current document:

```
string word = Application.Selection.Range.Words.First.Text;
```

It is not recommended to read or write to the document from multiple threads. In this particular example we only read a single word sequentially, and then perform the spell check of that word in parallel. After all of the suggested words are found, we then present them to the user from a single thread.

We will then spell check the word using three algorithms: a parallel Levenshtein distance algorithm, a sequential Levenshtein distance algorithm, and the Word spell checker. We implement the sequential Levenshtein distance algorithm with a LINQ query that calculates the distance from each word in the word list to the mispelled word, and then takes the *n* closest matches. The TakeTop() operator we use in the query is not a part of LINQ or PLINQ, but it as part of our samples.

```
suggestedWords = m_wordList
                    .Select(word => new WordScorePair() { word = word, distance =
LevenshteinDistance(word, originalWord) })
                    .TakeTop(p => p.distance, m_maxNumOfSuggestions)
                    .Select(p => p.word)
                    .ToList();
```

Similarly to the Excel add-in example, making this run in parallel is achieved by adding the AsParallel() operator.

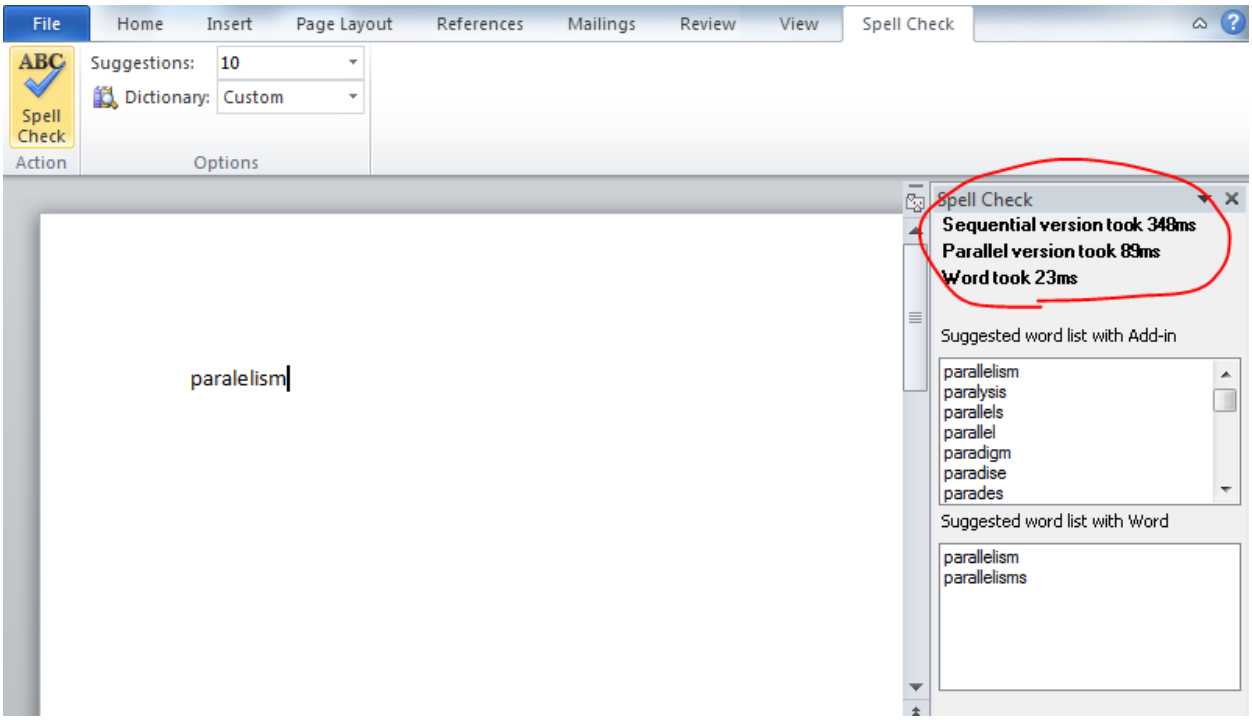
```
suggestedWords = m_wordList
                    .AsParallel()
                    .Select(word => new WordScorePair() { word = word, distance =
LevenshteinDistance(word, originalWord) })
                    .TakeTop(p => p.distance, m_maxNumOfSuggestions)
                    .Select(p => p.word)
                    .ToList();
```

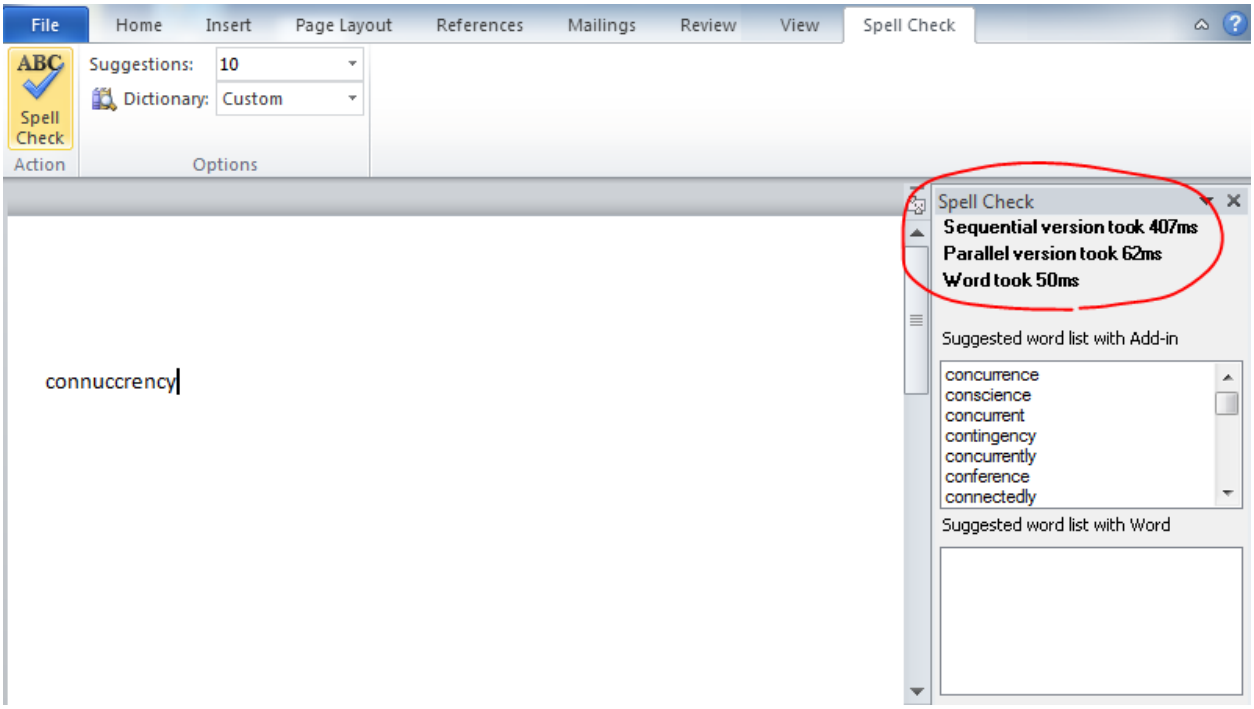
To get Word to perform spell check for us, we can use the following code:

```
suggestions = Globals.ThisAddIn.Application.GetSpellingSuggestions(originalWord);
```

To display the results, we created a new user control (SpellCheckTaskPane) with a ListBox to contain the suggested words. This new user control is hosted in the Office Task Pane as a custom Task Pane. For more information on how to create your own custom TaskPane, see the MSDN documentation.

```
taskPane = new SpellCheckTaskPane();
customTaskPane = this.CustomTaskPanes.Add(m_taskPane, "Spell Check");
```





As mentioned previously, Word's spell checker is quite fast. However, the comparison of our interest is between the sequential and the parallelized algorithm. As we can see from the output, the parallel approach performs significantly faster than the sequential approach (this screenshot was taken on an eight-core machine), especially if dealing with a large word list. Additionally, given that we're using a different algorithm, it's fun to note that there are times when our spell checking algorithm suggests words not suggested by Word's algorithm.

Conclusion

You can leverage the parallelism with the .NET 4 and VSTO in a straightforward way and in doing so make your Office add-ins perform faster. We hope these two examples can help you develop your Office apps to take advantage of multicore machines today.

Links

Parallel Computing Development Center

Office Development with Visual Studio (VSTO)

Office with VSTO Custom Task Panes overview

Optimizing using Concurrency Visualizer

Excel add-in sample