Optimizing Parallel Applications Using Concurrency Visualizer: A Case Study

Boby George, Pooja Nagpal

Parallel Computing Platform Group

Microsoft Corporation

Table of Contents

Introduction

Parallel Extensions to the .NET Framework, a set of libraries released as a part of .NET 4, is designed to ease the development of parallel applications. For example, adding AsParallel() to most LINQ-to-Objects queries enables the utilization of all cores in a multi-core system. However, to fully realize the benefits of parallelism, developers have to understand the execution profile of their application.

To gain insight into the developer experience of using parallel programming with .NET, the Parallel Extensions team routinely undertakes the development of nontrivial parallel applications. This allows us to gauge the effectiveness of the API and tooling support, and also provides perspective on the common pitfalls encountered when developing applications for multi-core systems.

One effective approach to experience problems developers are likely to encounter when trying to parallelize an existing application is as follows: implement a sequential version of the algorithm, parallelize it by applying Parallel Extensions constructs, track down the performance inhibiters and finally redesign the algorithm for optimal performance. In this article, we capture the triumphs and tribulations of implementing a console-based spell checker application, and delineate how the new Concurrency Visualizer was used to detect and fix performance issues.

Spell Checker Explained

Spell checking is a feature found in most word processing applications. It enhances the document writing experience by offering suggestions on how to correct a presumably misspelled word. For our purposes, we created a standalone console version of a spell checking application that accepts a misspelled word, and lists out the words that closely match the input word. The algorithm computes the similarity between the input word and all words in the reference dictionary, and outputs the N most similar words.

Multiple metrics exist for determining the similarity of two strings. The similarity metric we chose is based on the number of character insertions, removals and replacements required to transform one string into another string. This similarity metric, which measures the edit distance between two sequences, is commonly referred to as Levenshtein distance. As for reference dictionary, we used the "Official 12Dicts package" word list from Kevin's Word List Page, which contains about 200,000 common English words of different sizes and characteristics.

The first step in our process was to develop a sequential version of the spell checker algorithm. Since computation of multiple edit distances can be viewed as a data transformation (for each word in the dictionary, find the edit distance with respect to original word) operation, we decided to implement this using LINQ.

Sequential Version

The spell checker application needs to be able to read in the reference dictionary stored on the disk. We created a spell checker class that contains code as follows:

```
class SpellChecker
{
```

```
    List<string> wordList;
    public SpellChecker(string wordListFile)
    {
        wordList = File.ReadLines(wordListFile).Select(line => line.ToLower()).ToList();
    }
}
```

We use a struct named WordScorePair to keep track of a word's Levenshtein distance with respect to the input word.

```
struct WordScorePair
{
    public string word;
    public int distance;
}
```

For each word in the dictionary, we compute the edit distance from the misspelled word using the algorithm described in the Wikipedia page for Levenshtein distance. This algorithm computes the recurrence Edit(i, j) that represents the edit distance between the first i characters of the misspelled word and the first j characters of the word from dictionary. The recurrence is defined as follows:

```
Edit(i,j)
    = j     if i = 0
    = i     if j = 0
    = Edit(i-1,j-1)   if i > 0, j > 0, and string1[i-1] = string2[j-1]
    = 1 + Min(Edit(i-1,j), Edit(i,j-1), Edit(i-1,j-1))    otherwise
```

We calculate edit recurrence for all values of i from 0 to length of first string and for all values of j from 0 to length of second string. The intermediate results of edit recurrence were stored in a two-dimensional array (later this turned out to be highly inefficient). Once the computation is over, the bottom-right element of the matrix contains the edit distance between the misspelled word and the word from dictionary. (For details, refer to Wikipedia).

Once we have calculated the edit distance for all words in the dictionary, the top N words with the smallest edit distance are chosen as suggestions to the user. The LINQ query that would perform the computation for a user-specified word (originalWord) and the number of suggestions to find (numSuggestions) is as follows:

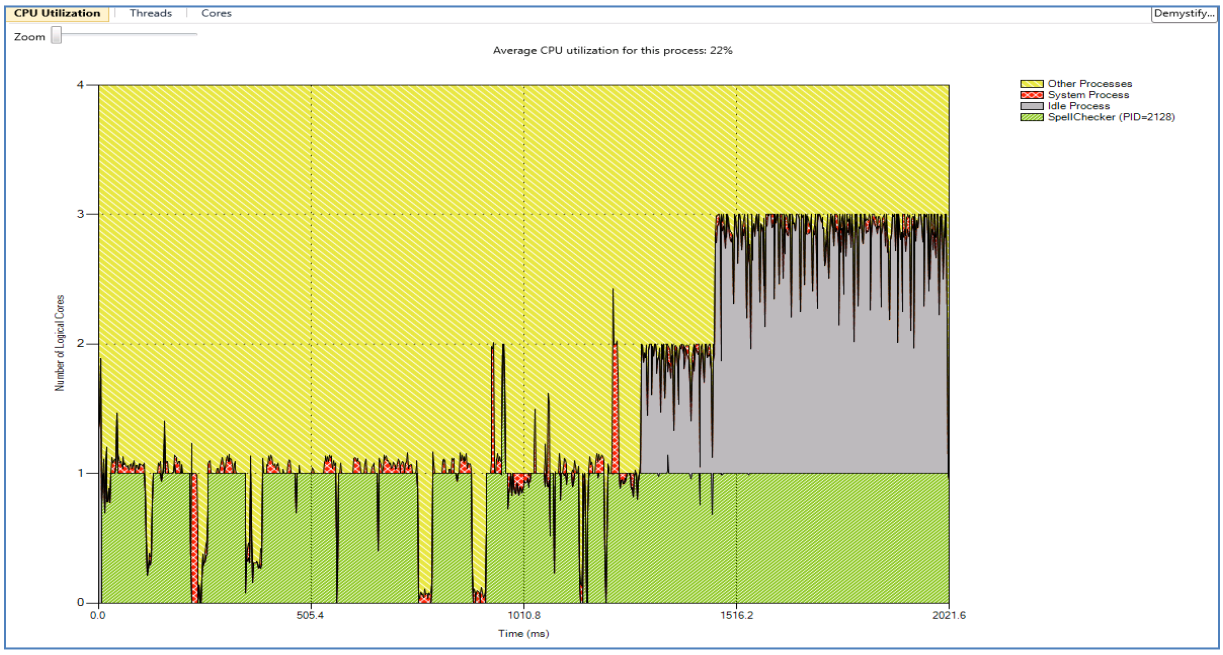
```
public List<string> Suggest(string originalWord, int numSuggestions)
{
    return wordList
        .Select(word => new WordScorePair()
           { word = word, distance = LevenshteinDistance(word, originalWord) })
        .OrderBy(p => p.distance)
        .Take(numSuggestions)
        .Select(p => p.word)
        .ToList();
}
```

Profiling using Concurrency Visualizer

After implementing the LINQ version of the application, we profiled it using Visual Studio 2010 Concurrency Visualizer. To use the profiler, ensure that the symbol path is correct (please refer to MSDN documentation for more details on how to configure symbols), click on Analyze -> Performance Wizard, and select the Concurrency Radio button. Ensure that the "Visualize the behavior of a multithreaded application" checkbox is checked. Click Next, choose the application to profile, and click Finish to launch the profiler. For more details on how to use the Concurrency Visualizer please refer to the Beginner's Guide to Profiling Parallel Apps blog post. The profiler will execute the application and after a few minutes will load a concurrency report. The concurrency visualization report consists of three views:

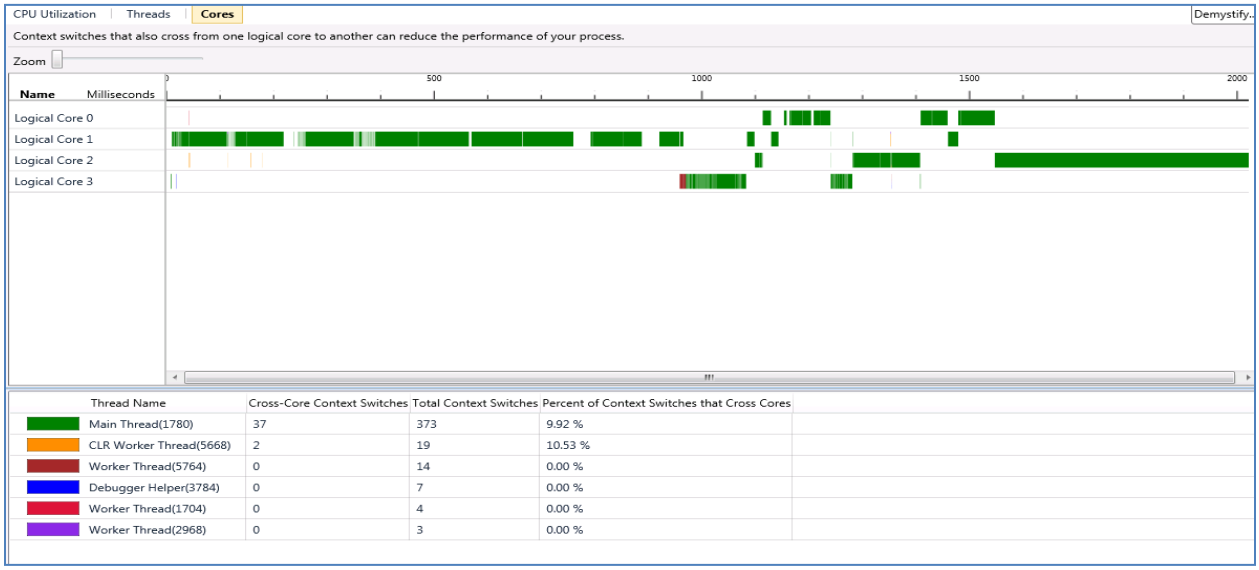
- CPU Utilization: provides a snapshot of logical CPU core utilization during execution of the profiled application
- Threads: tracks the activities of the threads in the profiled application
- Cores: tracks how the profiled application's threads were executed across logical CPU cores

Let us take a look at how the report looks for our spell checker application when executed on a four-core Intel 64-bit machine. The CPU Utilization graph clearly shows that only one core was getting utilized by our sample application. It should be noted that the CPU utilization view does not show which specific logical cores were used to execute the application, only the percentage of the total capacity utilized is shown in the graph.



The graph also shows that apart from our application, the CPU was being consumed by other processes including the Concurrency Visualizer itself. When you switch to the Threads view (which only displays the activities of the profiled application), you can see the application had a main thread executing user specified work and five CLR worker threads.

Finally, the Cores view shows which threads got executed on which cores at what time:



Once we studied the report and understood that the application is indeed single-threaded and functioning as we expected, let us try to parallelize it.

Parallelization with PLINQ

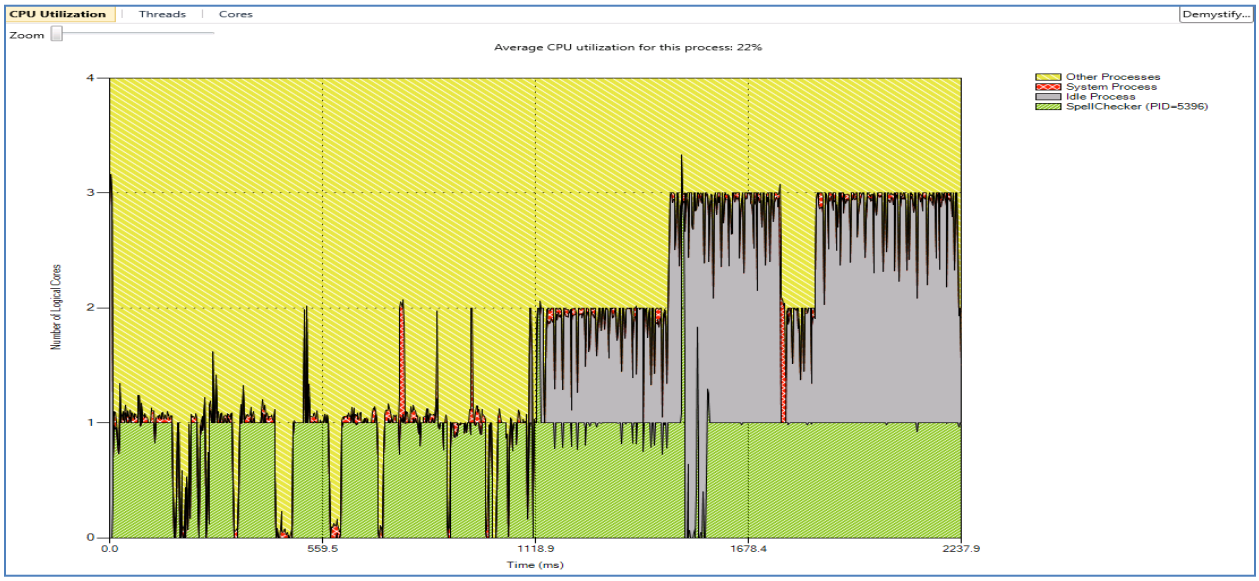
To attain speedup, the overhead involved in parallel execution should be small in comparison with the amount of work done in parallel. Therefore, we decided to parallelize the calls to LevenshteinDistance method and not the LevenshteinDistance method itself. We accomplish this by adding AsParallel() to the LINQ query, as shown below:

```
public List<string> Suggest(string originalWord, int numSuggestions)
{
    return wordlist
       .AsParallel()
       .Select(word => new WordScorePair()
          { word = word, distance = LevenshteinDistance(word, originalWord) })
```

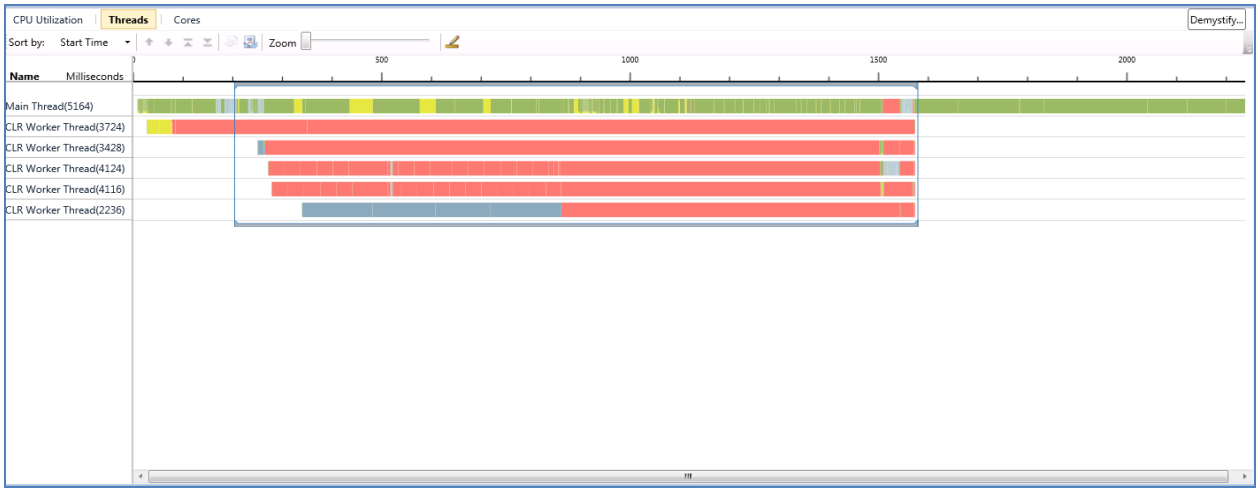
```
        .OrderBy(p => p.distance)
        .Take(numSuggestions)
        .Select(p => p.word)
        .ToList();
}
```

The LINQ query has no shared state in the delegates, and hence it is safe to parallelize without any changes. Resist the temptation to convert all for-loops in the LevenshteinDistance function to Parallel.For loops, as it is not safe to parallelize loops that have dependencies across iterations.

We compiled the application and profiled it using the ConcurrencyVisualizer. In order to compare the performance of the parallel application versus the sequential application, we decided to use the metric named speedup. Speedup is calculated by dividing the total execution time of sequential application over the total execution time of parallel application. Since the execution time listed in the article might not be repeatable (due to variations in machine configurations and execution environment), readers are encouraged to focus more on the approach employed and less on the performance numbers listed in this article..



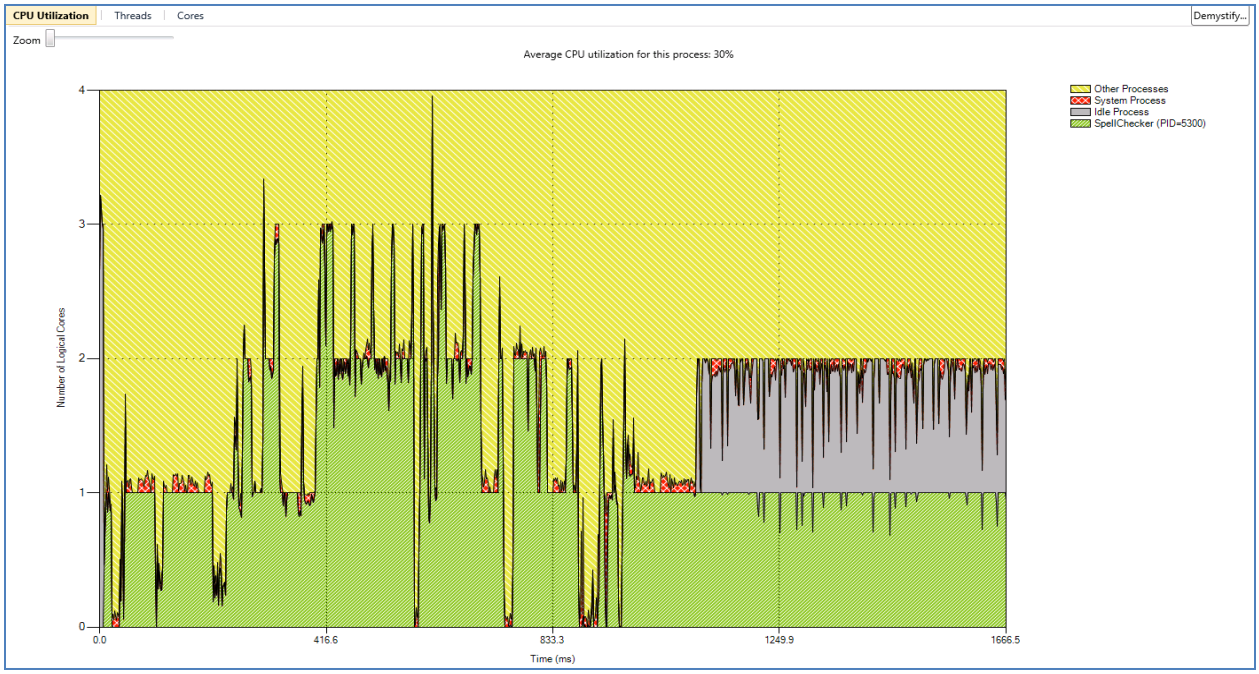
Looking at the CPU Utilization view, we were surprised to find that there is no parallelism at all. Let us switch to the Core view and see how the threads are getting executed:



The boxed section within the Cores view gives the detailed view of what is happening when the PLINQ query is getting executed. The main thread (represented by green color) is the only thread that is executing the PLINQ query, so indeed, the application in running in sequential mode. For certain query forms, if PLINQ determines that parallelization may lower performance, it automatically switches to sequential mode. Please refer to the blog post "PLINQ Queries That Run Sequentially" for more details on which scenarios PLINQ would switch to sequential execution. In such scenarios, if users determine that their workload is large enough to benefit from parallelism, the query can be forced to execute in parallel by specifying the ForceParallelism option. Let us see if that would help in our case. The modified query would look as follows:

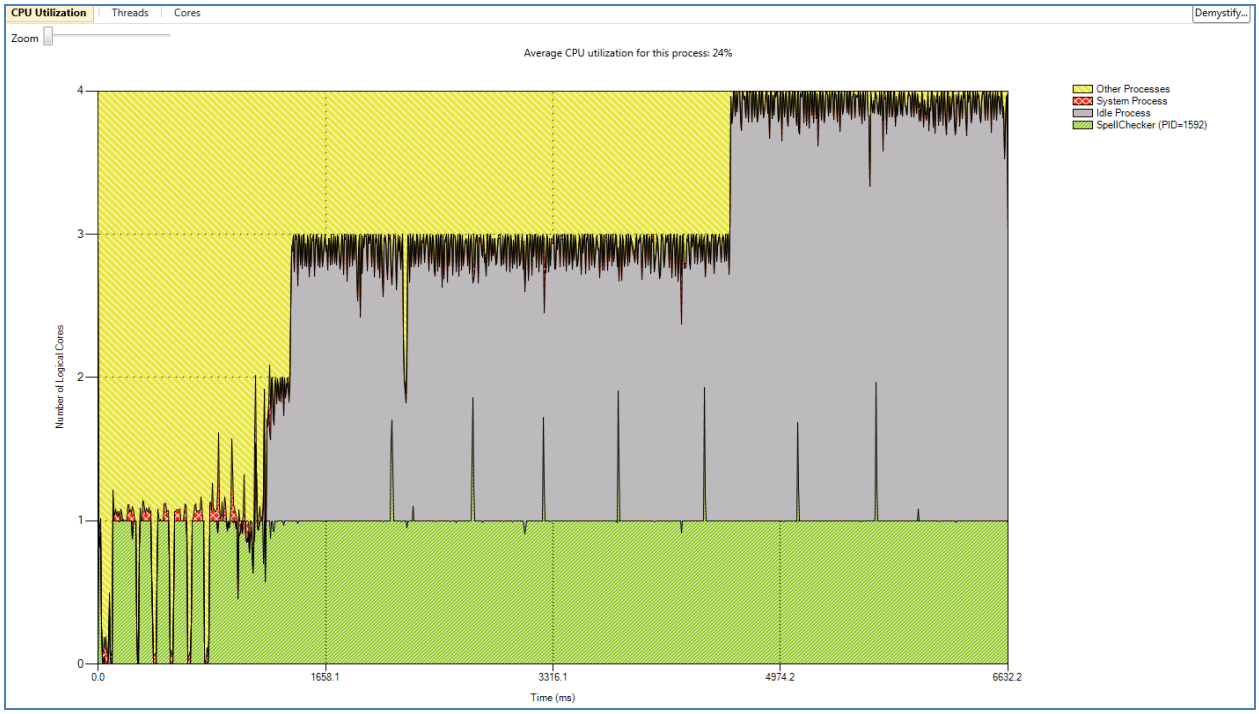
```
public List<string> Suggest(string originalWord, int numSuggestions)
{
   return wordlist
        .AsParallel().WithExecutionMode(ParallelExecutionMode.ForceParallelism)
        .Select(word => new WordScorePair() { word = word, distance = LevenshteinDistance(word,
originalWord) })
        .OrderBy(p => p.distance)
        .Take(numSuggestions)
        .Select(p => p.word)
        .ToList();
 }
```

After compiling and profiling the updated application, we can see that the CPU Utilization view shows activity in more than one core, so indeed we have a parallel application that utilizes multiple cores.

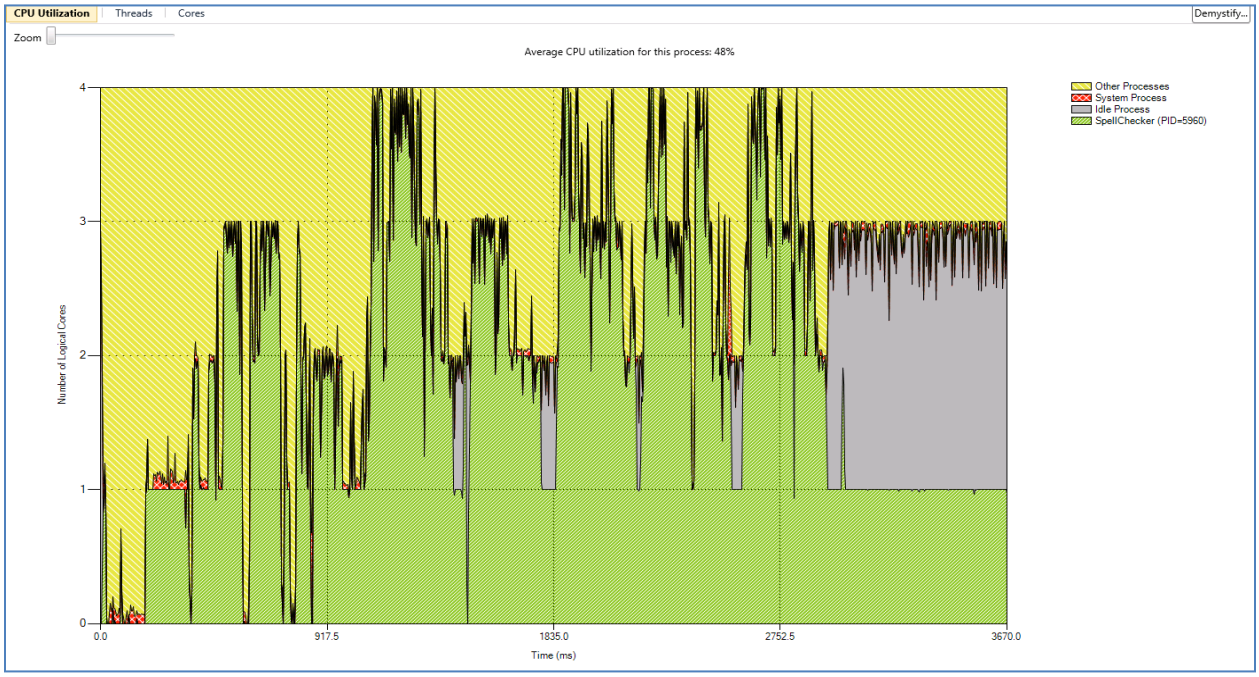


But did we get significant speedup? Not really. If you compare the total execution time, our original LINQ-based application took about 2021 milliseconds with average CPU utilization of about 22%, while the PLINQ-based application (with ForceParallelism option) took about 1666 milliseconds with an average CPU utilization of about 30%. Clearly, there is no real gain from parallelizing the application.

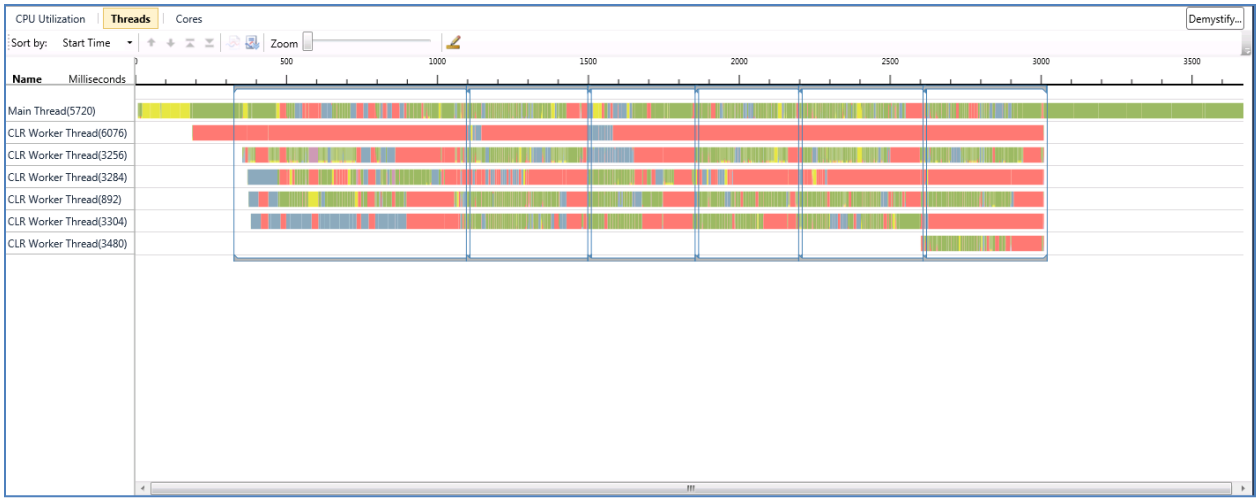
In such scenarios, one possible thing to try is to increase the workload size for the application and see if that results in a greater speedup. In our case, the simplest way to increase the parallel execution time is to search for more than one word. So, we modified the application to search for six misspelled words of varying length. The updated CPU Utilization views for both LINQ and PLINQ (with ForceParallelism option) are as follows:



Notice that the average CPU utilization (24%) remains the same (as expected), while the execution time increased to 6632 milliseconds in the case of the LINQ-based application. In the case of the PLINQ-based application, both the execution time (3670 ms) and average CPU utilization (48%) improved.



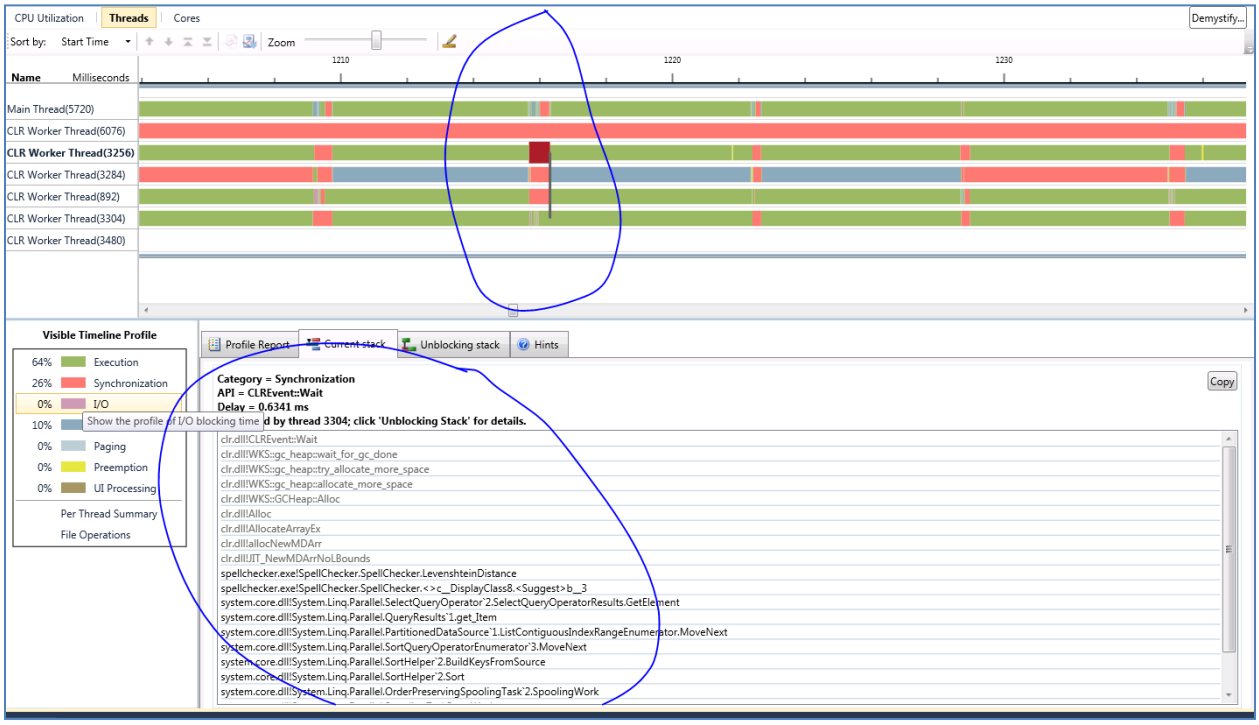
In the Threads view, we can clearly see how the PLINQ query was executed. The longer execution times give us a better picture of how the application used multiple threads. Note that the first query takes more time to execute due to startup costs than subsequent queries.



As far as the speedup goes, we get about 1.8 speedup and the average CPU utilization has jumped up to 2x for a 4 core machine.

Optimization – Reduce garbage collections

Although the CPU utilization has improved, looking at the Visible Timeline profile section in the Threads view, we realize that majority of the time is spent in synchronization and not in execution. To understand the reasons for synchronization, we zoomed into one of the query execution phases. The Threads view shows an interesting pattern:

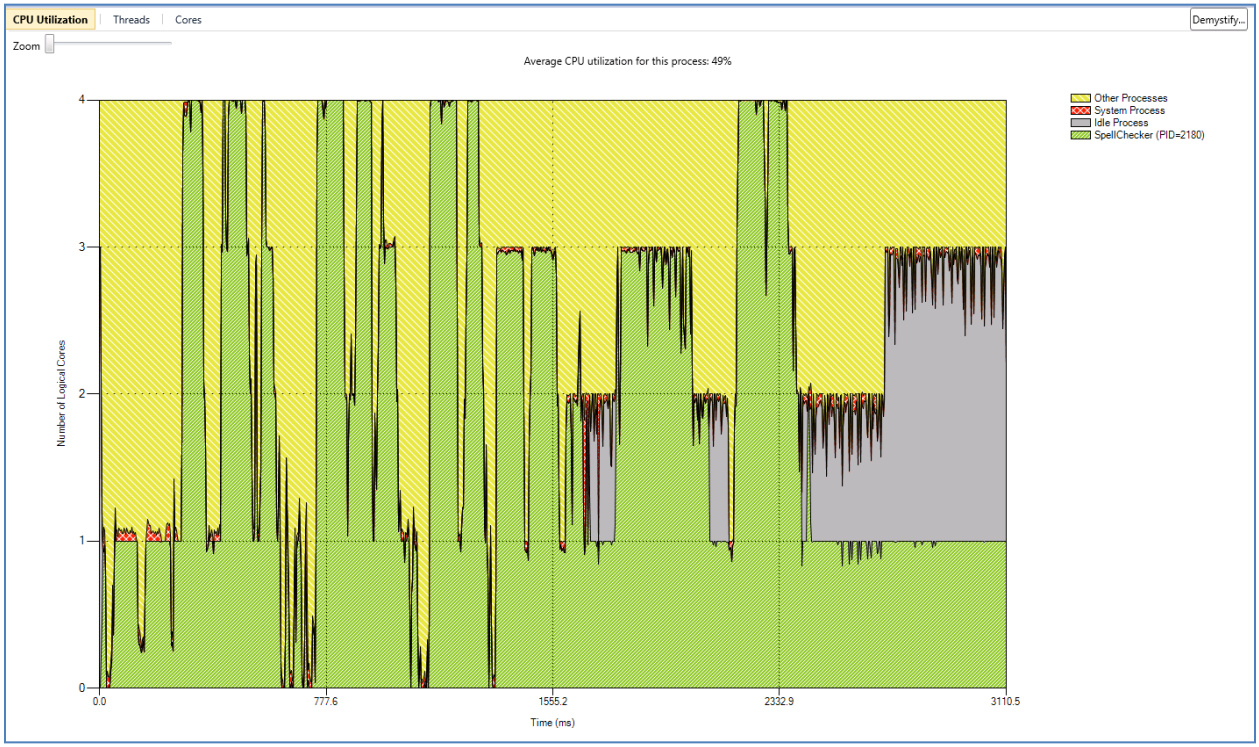


As circled in the figure, there are times when only one thread is executing and the rest are blocked. One reason for this pattern to appear repeatedly is when garbage collections (GCs) are frequent. This cause is particularly likely if the call stack includes a reference to clr.dll. In our case we studied the call stacks

and noticed that GC is frequently occurring inside the LevenshteinDistance function. Looking through the code for reasons for frequent GCs, we noticed that each call to the LevenshteinDistance method allocates a two-dimensional array. Clearly in our case, this is inefficient as we can optimize our code by reusing the same array for different invocations of LevenshteinDistance. When modifying the code to reuse array, note that for sequential algorithm, we only need one array. However, for parallel algorithm, we need one array for each thread that participates in the computation. To create a two-dimensional array on a per-thread basis, we used the new ThreadLocal type that provides per-thread storage for objects. The LevenshteinDistance function was updated to take in the two-dimensional array as an input parameter. The modified code looks as follows:

```
public List<string> Suggest(string originalWord, int numSuggestions)
{
   using (var distanceMatrix = new ThreadLocal<int[,]>(()=> new int[maxWordLength+1,
   maxWordLength+1)
   {
        return wordlist
          .AsParallel().WithExecutionMode(ParallelExecutionMode.ForceParallelism)
          .Select(word =>  new WordScorePair() { word = word, distance =
          LevenshteinDistance(word, originalWord, distanceMatrix.Value) })
          .OrderBy(p => p.distance)
          .Take(numSuggestions)
          .Select(p => p.word)
         .ToList();
    }
}
```

Let us profile one more time to see how we are performing.

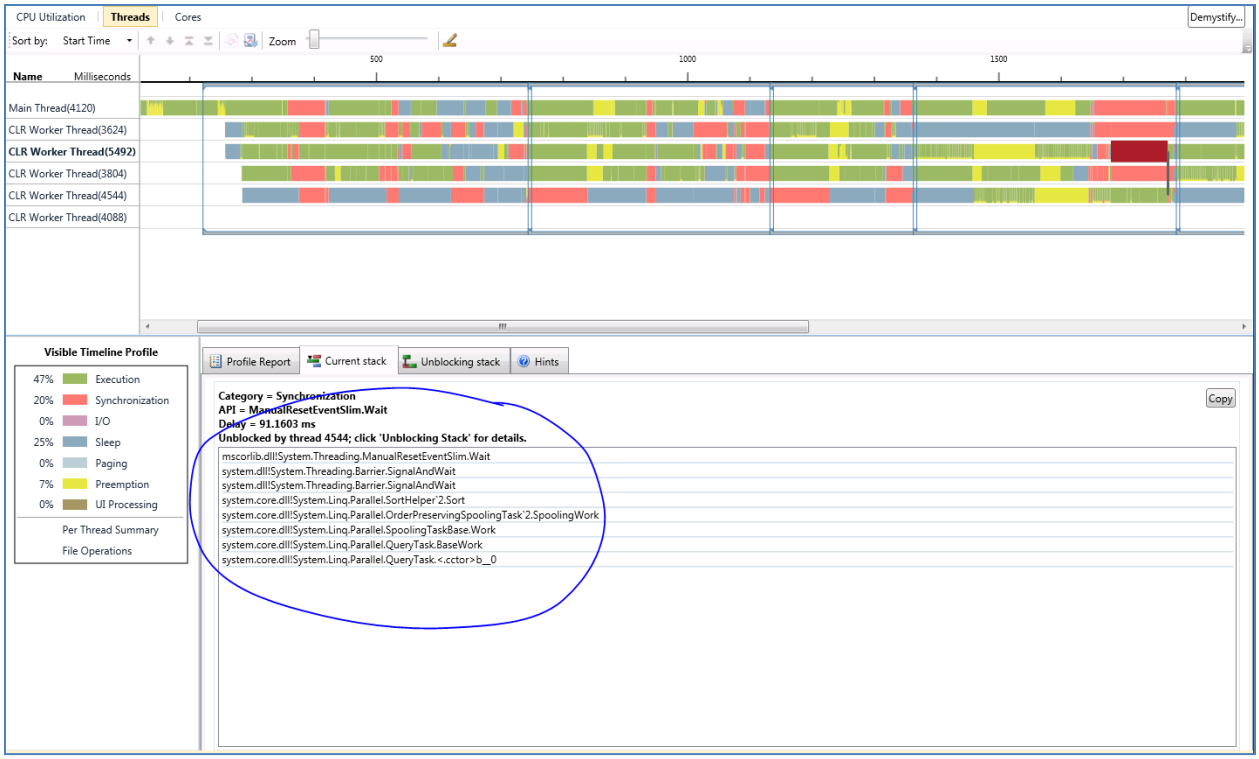


The CPU utilization graph shows that the CPU utilization across all cores has improved. Further the elapsed time for the application has shrunk to 3110 milliseconds. Our optimization effort has paid off.

Optimization – Reduce synchronization overheads in the algorithm

Let us spend some more time and see if we can find more avenues to improve the performance. Taking a look at the Threads view, we see that the application still spends a lot of time in synchronization.

Let us try to understand the synchronization aspects of the application by looking at the various synchronization sections in the graph, shown in red. After studying the call stacks at several portions where a thread was blocked, we noticed that towards the end of each query they have reference to PLINQ's sort function. For example, below is the call stack of a synchronization section for the fourth query:

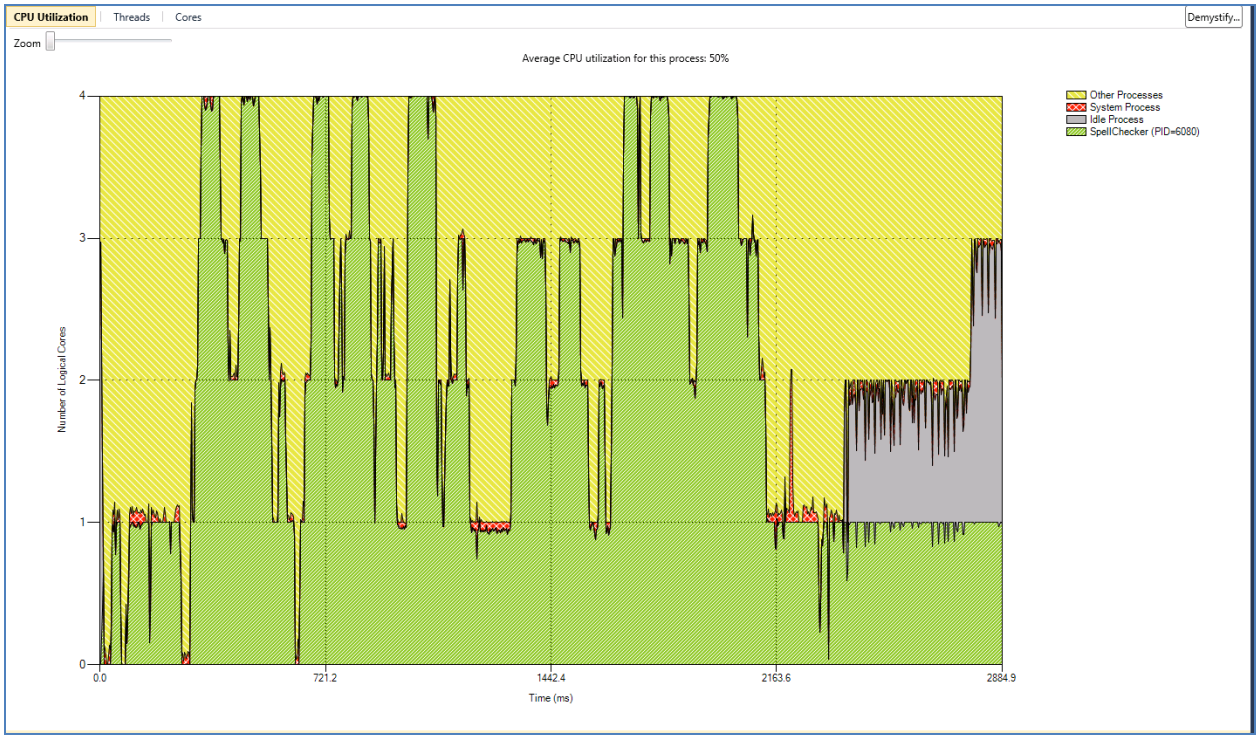


If we look at our PLINQ query, we have an OrderBy operation that sorts all the 200,000 words based on their edit distance. Sorting in parallel needs some communication between threads and will show benefits in performance only if the workload is large enough to justify the extra synchronization. From our profile, it looks like that the work may not be big enough when compared to the extra synchronization. Can we find an alternate algorithm that can avoid sorting or reduce the overhead of a full sort?

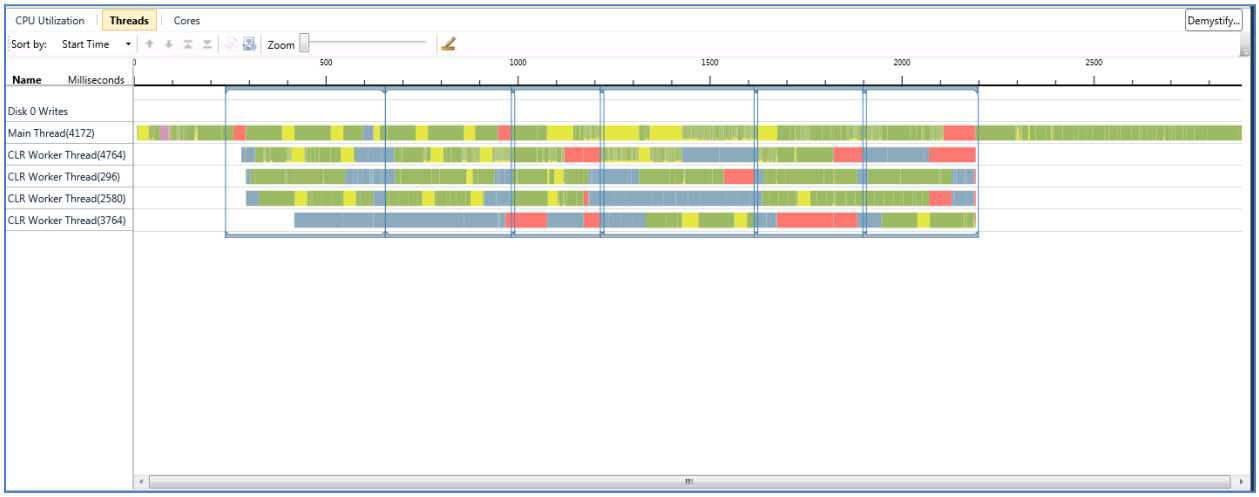
Looking closer, we realize that our application does not require that ***all*** the words in the dictionary are sorted according their distance. In fact, it only needs the first N words in the sorted sequence, since only the best N matches are shown to the user. The motivation for this approach was that the amount of synchronization between threads required to merge lists of size N (in our case N = 5) should be smaller than the amount of synchronization needed to sort 200,000 words.

Based on this insight into the working of the application, we revised both the sequential and parallel versions of the application to directly choose the top N suggestions. In the parallel case, each thread maintained a sorted list to keep track of the best N elements and merged them at the end to get the final suggestions. In case of sequential version, we need to maintain only a single list.

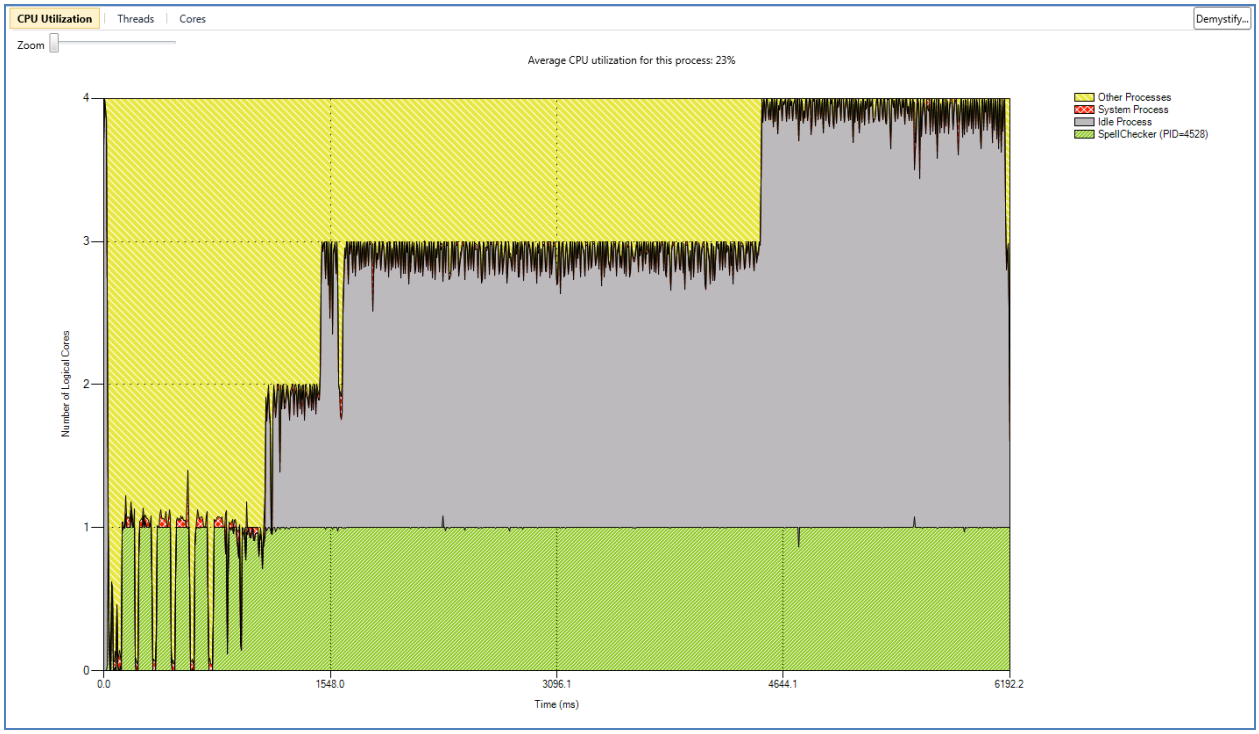
Let us see how the execution profile looks after implementing this new algorithm (refer to the code sample → ParallelExtensionsExtras → SortedTopN.cs for details):



The average CPU utilization now jumped to 50% and the application execution time has dropped to 2884 milliseconds. Let us look at the Threads view and see if synchronization still consumes majority of the application execution time:



Clearly, the cost of synchronization has gone down. In order to compute the speedup, let us profile our sequential version with all the optimizations. The new CPU Utilization view for the LINQ query is as shown below:



The execution time for LINQ also dropped to 6192 milliseconds. Hence, the overall parallel speedup is about 2.2, with the average CPU utilization increased from 23% to 50%. Note that the profiler does take some cycles. To get the true speedup, we executed the application without the profiler. Our LINQ execution time is 4881ms and PLINQ execution time is 1371ms. On a 4-core box, getting a speedup of 3.56x is clearly a significant performance gain.

Conclusion

New features in .NET 4 make the development of parallel programs significantly easier. We found PLINQ to be a convenient model for expressing parallelism, and our spell checker was easily parallelized as a single PLINQ query.

We had to tune our algorithm to achieve optimal performance and scaling, and we found that Visual Studio 2010 Concurrency Profiler was very helpful for this task. Another interesting takeaway from our investigation was that –frequent garbage collection can inhibit scalability of the application, so it is important to reduce memory allocations in the concurrent part of an application.

Overall, our investigation confirmed that the new libraries and tools in .NET 4 significantly simplify the process of developing programs that scale on multi-core machines.