Parallel Concordance in C#

SICSA MultiCore Challenge 2010

Hans-Wolfgang Loidl
<hwloidl@macs.hw.ac.uk>

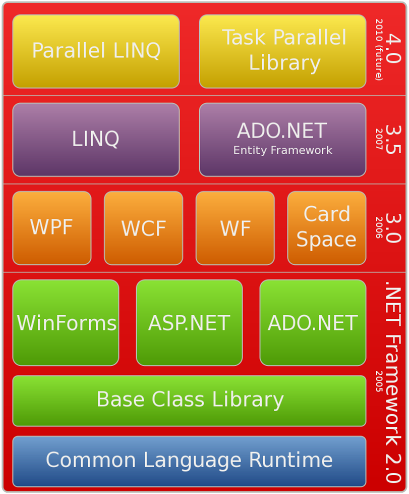
School of Mathematical and Computer Sciences,
Heriot-Watt University,
Edinburgh

December, 2010

Why C#

- The *Parallel Pattern* approach for C# advocates a high-level parallel programming model.
- In essence, these are *skeletons* in disguise.
- From .Net 4.0 onwards this is supported through the Task Parallel Library (TPL).
- This acknowledges that more user-friendly approaches to parallel programming are desirable in the age of desktop parallelism on multi-cores.
- Based on the recent book: "*Parallel Programming with Microsoft .NET — Design Patterns for Decomposition and Coordination on Multicore Architectures*", by C. Campbell, R. Johnson, A. Miller, S. Toub. Microsoft Press. August 2010. `http://msdn.microsoft.com/en-us/library/ff963553.aspx`

DotNet Structure



The .NET Framework Stack

Focus of the implementation

- Explore the claim of easy parallelism.
- Test the sequential efficiency of the Mono implementation of C# and .NET under Linux.
- Not: optimised sequential implementation.
- No serious parallel performance tuning is done.

Structure of the program

- Read from file
- Split into words (`Split`)
- Normalise words (all lower case, no punctuation)
- Add all possible subsequences to a hashtable mapping strings to lists of indices

Data Parallelism with C#'s Patterns

```
var options = new ParallelOptions() {
  MaxDegreeOfParallelism = k };
Parallel.For(m, n, options, i =>
{
  ...
});
```

Top-level Concordance Method

```
public static void concordanceParallel(string file,
                                       int n, int k) {
  words = Concordance.readFile(file);
  /* Parallel version, using only k tasks */
  var options = new ParallelOptions() {
    MaxDegreeOfParallelism = k };
  Parallel.For(0, len-1, options, i => {
    for (int j = i+1; j<Math.Min(i+n,len); j++) {
      if (words[i].Length>0) {
        Concordance.addSequence(file, words, i, j, i); }
    }
  });
}
```

Top-level Concordance Method

- **Easy to use data-parallelism over the outer for-loop.**
- **Implicit load-balancing based on the options passed to the parallel loop.**
- **To avoid bottlenecks, an array of hashtables is used in `addSequence`.**

Hardware and Software Setup

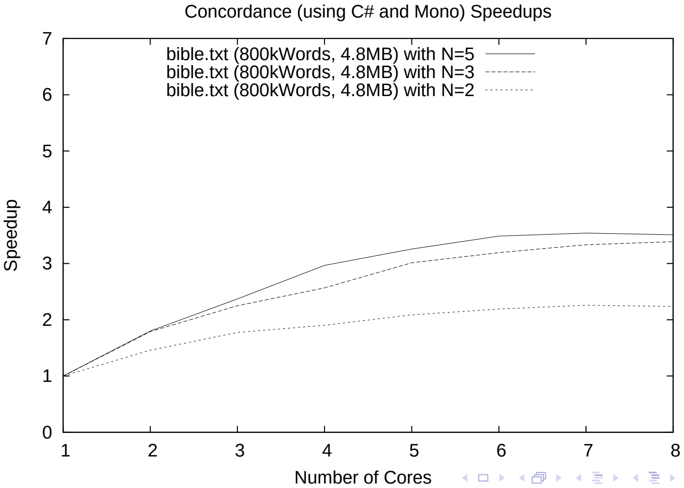
Hardware:

- Eight-core Intel Xeon E5410,
- 2.33GHz,
- 8GB RAM,
- 6MB L2 cache

Software:

- CentOS 5.5
- Mono C# & JIT compiler version 2.8.0.0
- Mono RTE & JIT compiler (to amd64) version 2.8.0.0

Performance Results



Conclusions

- Parallel patterns make heavy use of *delegates* in C# to realise skeletons, i.e. higher-order functions with parallel execution.
- Many more patterns exist: Pipeline, Divide-and-Conquer, Futures etc.
- A small set of control parameters can be used to tune parallel performance.
- Without serious tuning the relative speedups are humble: ca 3.5 on 8 cores

Further Work

- Use a customised `TaskScheduler` to tune the parallelism. By default it uses a workpools (both local and global) and thread stealing.
- Compare performance with an explicitly threaded version.
- Compare performance with Microsoft's C# implementation on Windows.
- Use optimised C front-end as tokenizer and call it from within C#.

An Example of Parallel Aggregates

```
var options = new ParallelOptions() {
                 MaxDegreeOfParallelism = k};
Parallel.ForEach(seq /* sequence */, options,
                 () => 0,  // The local initial partial result
                 // The loop body
                 (x, loopState, partialResult) =>  {
                    return Fib(x) + partialResult; },
                 // The final step of each local context
                 (localPartialSum) => {
                    // Protect access to shared result
                    lock (lockObject)
                      {
                        sum += localPartialSum;
                      }
                 });
```