S. Okur, D. Dig

How do Developers Use Parallel Libraries?

Presented by Paolo Antonucci

25/04/2013

Introduction

Computing is moving towards **parallelism.**

Parallelism can be made easier by the use of **parallel libraries.**

We present today an empirical study on how parallel libraries are used **in real world** by developers.

We will see the answers to **specific questions** about this.

Outline

We will go through:

- Motivation
- Background – Microsoft .NET parallel libraries
- Methodology
- Results
- Conclusion

Motivation

- Educating developers
  - by pointing them to the most used constructs straight away
  - by showing them some common mistakes
- Providing useful information to library developers
  - Learning how parallel libraries are used (and misused) in practice is crucial for effectively improving them
- Providing information to the research community
  - Research in other not strictly inherent fields (e.g. verification) can be affected by these data

Background

We here analyze the usage of some .NET parallel libraries, in particular:

- `System.Threading`
- Concurrent Collections (CC)
- Task Parallel Library (TPL)
- Parallel LINQ

  (LINQ = Language INtegrated Query)

Most of these can be compared to similar libraries in Java.

Background – Task Parallel Library

- Classes `Task` and `Task<T>` representing a task to be executed with no specific associated thread

- Static `Parallel` class providing functionality for parallel loops and invocation of methods

```
Parallel.For(0, 10, counter => { ProcessTask(counter, 17); });


static void ProcessTask(int taskNum, int somethingElse)
        {
            // Do something
        }
```

Example

Methodology – corpus of data

- Corpus: CodePlex (Microsoft) and Github
- Downloaded all active projects importing TPL, PLINQ, CC and `System.Threading` libraries
- Some filters applied:
  - Broken (not-compiling) and very small applications
  - Applications importing but never using parallel libraries
  - Application only using `System.Threading` for delays and timers
- Result: 17.6 million significant lines of code

Methodology – analysis infrastructure

- Microsoft Roslyn API for syntactical and semantic analysis of C# applications
  - More than just syntactical analysis, able to answer questions such as “What is the type of this variable?”

- Ad hoc ANALYZER tool developed
  - Specific analysis implemented for each question

Results:
Q1: Are developers embracing multithreading?

	(Significant lines of code)							
Type	**Small (1K – 10K)**		**Medium (10K – 100K)**		**Large (> 100K)**		**Total**	
Total .NET 4.0 compilable applications	6020	100%	1553	100%	205	100%	7778	100%
Multithreaded applications	1761	29%	916	59%	178	87%	2855	37%
Application adopting TPL, PLINQ	412	7%	203	13%	40	20%	655	8%

Multithreading is widely used in medium and large applications, however the adoption of advanced libraries is still limited.

Results:
Q2: Which parallel constructs are mostly used?

That is, in practice, what methods of which classes are called most often?

As an example we will show here the results of this analysis for the TPL library.

It is worth reminding that we are not speaking of heuristics: semantic analysis means that these results are 100% precise.

Results:
Q2: Which parallel constructs are mostly used?

Class name	% in library	Method name	# Call sites	% in class	# Apps
`TaskFactory`	30	`StartNew`	1256	72	286
		`FromAsync`	121	7	32
`Task`	23	`ContinueWith`	372	28	122
		`Wait`	273	20	110
		`Start`	243	18	92
		Constructor	225	17	82
		`WaitAll`	172	13	91
`Parallel` (static class)	14	`For`	450	53	102
		`ForEach`	365	43	133
		`Invoke`	37	4	23
`Task<TResult>`	11	`ContinueWith`	536	86	113
		Constructor	85	4	40

Results:
Q2: Which parallel constructs are mostly used?

The table only shows the first few most popular methods, out of 1651 total methods.

Out of these, 1114 (that is 67%) are never used in the corpus!

We draw two important conclusions:

Parallel library usage follows a power-law distribution: 10% of the API methods account for 90% of the total usage.

If you study just a very small bunch of important classes and methods, you are ready to go for most common situations!

Results:
Q3: Which advanced features do developers use?

Or even better, do they use them at all?

Just an example: `Parallel.Invoke`, `For` and `ForEach`.

They take an optional argument, `ParallelOptions`, which can be used for advanced features such as limiting the maximum concurrency and specifying a custom task scheduler.

Only **3%** of calls use `ParallelOptions`!

> *The advanced features and optional arguments*
> *are rarely used in practice.*

Results – Q4: Do developers make their parallel code unnecessarily complex?

TPL provides some high-level constructs that allow developers to implement parallel code more concisely.

```
static void Complicated()
{
    Task FirstTask = new Task(FirstMethod);
    Task SecondTask = new Task(SecondMethod);
    Task[] tasks = new Task[]
        {FirstTask, SecondTask};
    Array.ForEach(tasks, t => t.Start());
    Task.WaitAll(tasks);
}

static void Compact()
{
    Parallel.Invoke(FirstMethod, SecondMethod);
}
```

```
static void FirstMethod()
{
    // Do something
}

static void SecondMethod()
{
    // Do something else
}
```

Methods `Complicated()` and `Compact()` are equivalent.

Results – Q4: Do developers make their parallel code unnecessarily complex?

Not unfrequently, programmers launch tasks in for/foreach loops.

ANALYZER detected that in 29% (underapproximation) of these cases, the `Parallel.For`/`Foreach` construct could have been used, which is more compact and less error-prone.

Despite the fact that parallel programs are already complex, developers make them even more complex than they need to be.

Results – Q5: Are there constructs that developers commonly misuse?

Sometimes developers seem to misunderstand what the `Parallel.Invoke()` method does.

```
static void Misuse(string someString)
{
    // Some instructions
    Parallel.Invoke(() => MyMethod(someString));
    // Some more instructions
}
```

"Oh well, if it is called Parallel, it must run in parallel…"

This happens in as much as 11% of usages of `Parallel.Invoke()`!

Results – Q5: Are there constructs that developers commonly misuse?

Another commonly misunderstood method is PLINQ's `AsParallel()` method.

This method converts an `Enumerable` into a `ParallelEnumerable` collection. Any method called on such a parallel enumeration will execute in parallel.

```
foreach (var module in Modules.AsParallel())
    module.Refresh();
```

This code executes sequentially

27 occurrences of this in 19 applications, accounting for as much as 12% of all `AsParallel()` usages!

Results – Q5: Are there constructs that developers commonly misuse?

In conclusion:

Some parallel constructs are not always understood and employed properly, leading to code with parallel syntax but sequential execution

Compile-time warnings could help mitigate this problem in some cases.

Conclusion

- This paper is a greatly helpful reading to developers planning to embrace these parallel libraries
- The findings of this research have been shared with the developers of the analyzed libraries
- The results also provide some interesting starting points for research in related fields
  - For example refactoring of unnecessarily complex code

My personal conclusion

+ Interesting paper, very smooth and pleasant reading
+ Self-contained
+ May have impact on future library development

- Some answers not relevant, some assumptions are probably an oversimplifications
- Could have said a bit more about Microsoft Roslyn
- Overall probably not a milestone in research in computer science

References

- Website of this study: http://learnparallelism.net

- The Roslyn Project http://msdn.microsoft.com/en-us/vstudio/roslyn.aspx

- More references in the paper

Question time

Questions?