nuget install MakeParallel

Einfach mal parallel machen

Oliver Sturm • @olivers • oliver@oliversturm.com

Oliver Sturm

- Training Director at DevExpress
- Consultant, trainer, author, software architect and developer for over 25 years
- Microsoft C# MVP
- Contact: oliver@oliversturm.com

Agenda

- The technical side
- Steps to a parallelizable application
- Parallel calculations
- OliDTP
- STM

The technical side – Threads

- Old style: create Threads manually
- Alternative: ThreadPool
- Most “pedestrian” approach available in .NET today - potentially greatest control

The technical side – Fibers?

- “Sub-thread threads”
- Main advantage in the past: save context switches
- Win32 API available
- Unsupported in .NET

“In general, fibers do not provide advantages over a well-designed multithreaded application.” — Microsoft

The technical side – Tasks

- Current technology to schedule work for parallel execution
- Complex and complete API available, yet easy to use
- Automatic scheduling
- Language integration

The technical side – async/await

- Often associated with parallel computation techniques, but async/await are “only” a language feature
- Don’t create parallelization by themselves

The technical side – Locking & Co

- `lock` keyword - Monitor.Enter/Exit
- Monitor – TryEnter, Wait, Pulse, cross-AppDomain support by locking on MarshalByRefObject
- Mutex – cross AppDomains, global for OS-wide syncing
- SpinLock – spin instead of block, advantageous for very short wait times
- ReaderWriterLockSlim – parallel reads, synchronised writes
- …

The technical side – Locking & Co

- Semaphore, SemaphoreSlim – global and local (unless “slim”), resource access by multiple threads
- Wait Handles – Mutex and Semaphore are WaitHandles, (Auto/Manual)ResetEvent can be signalled explicitly
- Barrier – block multiple threads
- Interlocking – atomic basic operations Add, Increment, Decrement, Read, (Compare)Exchange
- …

The technical side – Locking & Co

- Microsoft calls all these techniques “synchronisation primitives”
- Synchronisation is hard!
  - … and prone to errors
  - … especially when maintaining it later
  - … and it reduces parallelization
- Recommendation: avoid the need for synchronisation

The technical side – Immutable Data

- No changes, no collisions
- Self-made structures or standard collection types (System.Collections.Immutable)
- API lends itself well to isolation tasks - make sure each executing task has a stable view of its data!

The technical side – STM

- Software Transactional Memory
- Idea: transaction control over in-memory operations
- Exists in many functional environments: Erlang, Haskell, Clojure, …
- Much easier to do in environments where data is by default immutable
- Open-source framework: Shielded

The technical side – Automation

- Libraries can simplify creation and handling of tasks or threads
- PLINQ
- Rx
- Actors
- … others

The technical side – Code Structure

- Complex nested responsibilities are hard to parallelize
- Clearly structured networks of functions are easier to parallelize
- Coordinating write access to shared data is important
- Avoiding shared data is better!
- Side-effect free function implementation is even better!

The technical side – Services

- Service patterns enforce separation of concerns
- The more services, the more separation
- Shared data is impossible
- Call patterns and scheduling are a responsibility of the executing infrastructure

Steps to a Parallelizable Application

- Start by identifying parts of application logic that can benefit from running in parallel
- Are there specific pain points reported by customers?
- Are you aware of specific inefficiencies?
- What’s the most complex algorithmic logic you execute?

Step: Identify the Code

- Identify the code that represents the logic in question
- Code may be spread out in different places
    - … or it may be tightly embedded in other structures like UI code

Step: Analyse the Logic

- Identify “steps”
- Look for how information is generated and how it “flows”
- Define or deduce your data pipeline
- If a “step” loops or repeats in some way, it may be parallelizable
- Keep in mind the “size” of tasks - not too small and not too large
- Consider restructuring steps to facilitate a data pipeline that’s easier to parallelize

Step: Consider Avoiding Data Sharing

- Steps that modify data can’t easily be parallelized
- Locking data is a bad option
- Assign exclusive working data to parallel tasks
- Functional way: pass data, receive results, accumulate
- In-place isolation can be an option, e.g. in RDBMS scenarios

Step: Consider Isolation

- Tasks that access data independently must expect it to change through outside modifications
- Modifications from other tasks in the same system point to planning mistakes
- Infrastructure might facilitate concurrent changes, e.g. in web applications

Demo

Parallel Calculations

Demo

OliDTP

Poor man’s vector graphics

Processing a Structural Change

- Create independent steps:
    - Create a layer bitmap for each layer
    - Accumulate the layer bitmaps into one
- Instead of passing all the document data to one opaque "step", chunks of data per layer will be handled independently
- Accumulation becomes necessary because we will receive partial results
- Big data, anyone? Mapreduce?

Proposing Isolation

- Clone the document structure before rendering commences, so that any parallel changes to it don’t influence the rendering results

Demo

STM

Shielded

Sources

Demo source code: https://github.com/oliversturm/make-parallel-demos

This presentation: https://oliversturm.github.io/make-parallel-demos

- Deprettified content in pdf format: https://oliversturm.github.io/cs-state-of-the-nation/slidecontent.pdf

Thank You

Please feel free to contact me about the content anytime.

oliver@oliversturm.com