



Everyday Parallel Programming in C# and .NET

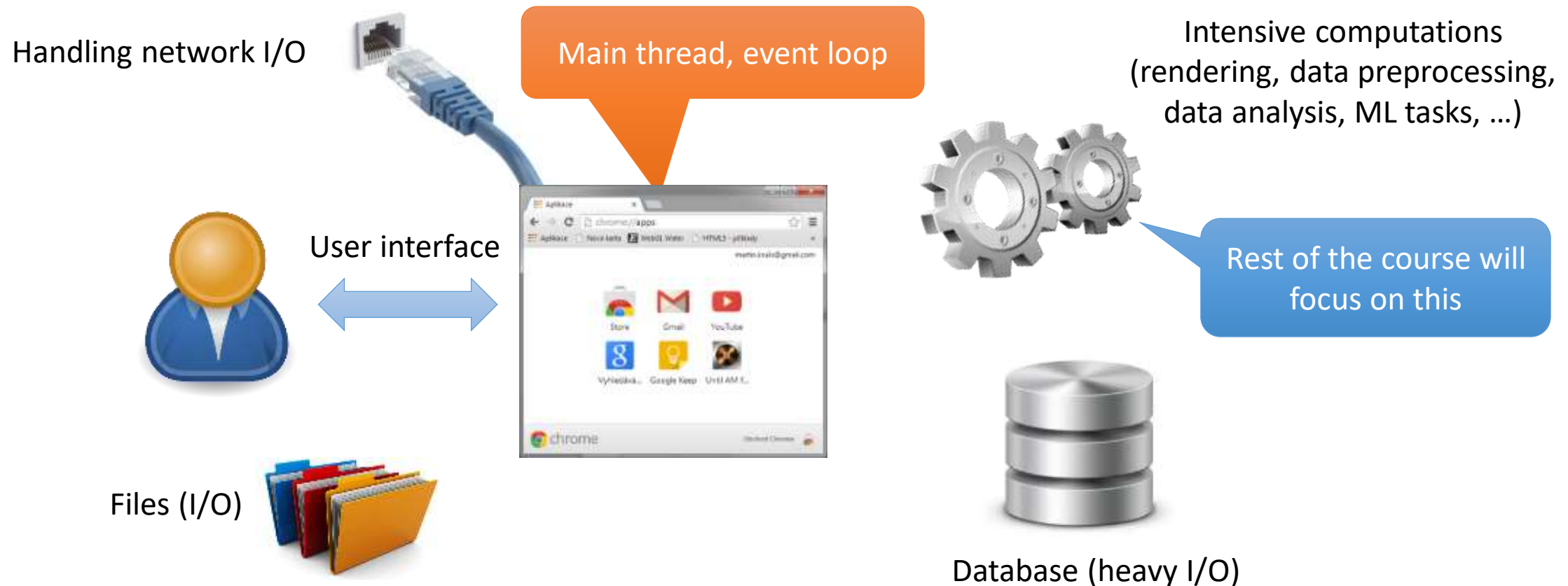
NPRG042: Programming in Parallel Environment

Martin Kruliš

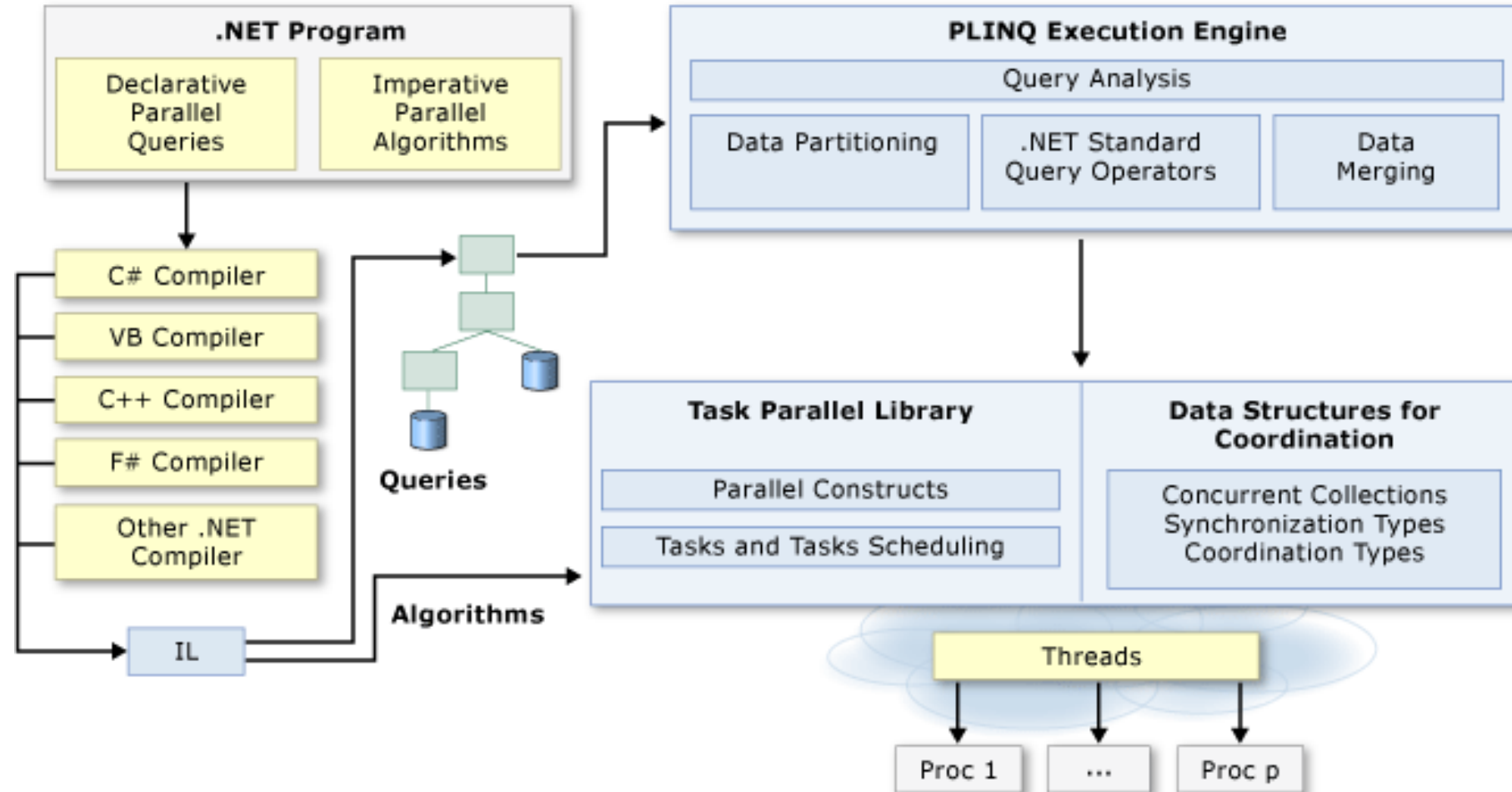
Everyday Parallel Programming



- Modern applications need to perform many tasks simultaneously



Parallel Programming in .NET





Common Threads

- **Thread** class
 - Constructor expects a delegate to execute
 - Has to be explicitly started (**Start()** method)
 - Foreground vs background threads
 - **IsBackground** property
 - Applications ends when all foreground threads end
 - Background threads are killed when app terminates
 - Waiting for termination
 - **Join()** – optionally with timeout
 - **Abort()** – throws aborting exception inside threads
 - **Thread.CurrentThread**

Deprecated! Use cancelation token...



Common Threads

- **Thread** class

- **Priority**
- **Suspend()**, **Resume()**
- **Sleep()**, **Interrupt()**
- **Yield()**
- **VolatileRead()**, **VolatileWrite()**, **volatile**

Deprecated! Not safe.

- Basic synchronization

```
lock (object) {  
    // critical section bound to object  
}
```



Synchronization Primitives

- **System.Threading** namespace
 - **Monitor, Mutex, ReaderWriterLock, Semaphore**
 - **ReaderWriterLockSlim, SemaphoreSlim**
 - **Barrier**
 - **AutoResetEvent, ManualResetEvent, EventWaitHandle, CountdownEvent**
 - Thread can wait on the event until it is signaled
 - **Interlocked** – provides atomic operations
 - **ThreadLocal<T>, AsyncLocal<T>**
 - **CancellationToken**

Slim versions are better



Thread Pool

- **ThreadPool** class
 - Often better alternative to individual threads
 - Especially for non-blocking tasks
 - Only uses background threads
 - Used automatically in many situations
 - **Task** execution (including async/await handling)
 - Asynchronous timers
 - Registered wait handlers' callbacks
 - Can be used manually
 - **ThreadPool.QueueUserWorkItem(proc)**
 - Passing a **WaitCallback** delegate

Quite crude, Tasks are usually better...



Parallel Constructs

- **Parallel** class
 - **Parallel.For(0, N, methodOrDelegate) ;**
 - **Parallel.For(0, N, i => { ... }) ;**
 - Parallel equivalent of for construct
 - **Parallel.ForEach(enum, x => { ... }) ;**
 - Parallel equivalent of foreach construct
 - **Parallel.Invoke(actions)**
 - Invokes an array of delegates concurrently

Tasks



- **Task, Task<Result>** classes
 - Independent encapsulated jobs executed by a thread pool in the background
 - Separate creation and execution

```
Task t = new Task(action, ...);  
t.Start();
```
 - Other ways how to create/start a task
 - **Task.Run(action)** – creates and executes a task
 - **Task.Factory.StartNew()** – also creates and executes
 - **t.RunSynchronously()** – run a task in the current thread immediately



Tasks

- Synchronization
 - **Wait()**
 - Optionally with timeout and/or cancelation token
 - **Task.WaitAny(tasks), Task.WaitAll(tasks)**
- Dependencies
 - **var t2 = t1.ContinueWith(action)**
 - Create new task which is a continuation of given task
 - **Task.WhenAny(tasks), Task.WhenAll(tasks)**
 - Create a task which is executed when any of/all given tasks terminate

Tasks



- Task Scheduling

- Global queue + Local Queues (similar to TBB)
- Employs work stealing for load balancing
- Task inlining (sync. execution in waiting thread)
- **TaskScheduler** class
 - Some actions may be given a scheduler object
 - `Start()`, `ContinueWith()`, ...
 - **TaskScheduler.FromCurrentSynchronizationContext**
 - Returns scheduler related to sync. context of the current thread
 - Very useful for UI (WinForms, WPF) – allows a specific task to be executed in the main thread (to operate UI controls)

Oops, lectures causality violation!



Concurrent Data Structures

- **System.Collections.Concurrent** namespace
 - **ConcurrentBag<T>**
 - Unordered collection
 - **ConcurrentDictionary<K,V>**
 - `AddOrUpdate()` , `GetOrAdd()` , `TryUpdate()` , `TryRemove()`
 - **ConcurrentQueue<T>**
 - **ConcurrentStack<T>**
 - **BlockingCollection<T>**
 - Collection for producer/consumer problems



Asynchronous Programming

- Asynchronous functions

- `public async Task<Res> foo() { ... }`
- Can be suspended in the middle (see `await`)
- Note that instead of result, it yields a task

Similar (or even the same) constructs can be found in other languages (JavaScript, Python, Rust, C++, Scala, Swift...)

- Waiting for tasks

```
var t = somethingThatReturnsTask();  
var res = await t;
```

- Suspends the code until a result of a task become available
 - Does not block the physical thread (core)
- Only available in **async** functions

Unlike `Wait()`

Dataflow



- **Dataflow** package

- Data-passing pipeline-like parallel model
- Data are passed through the flow in *messages*
- *Dataflow blocks* – buffer and process the data
 - `ISourceBlock<TOut>` - read-only
 - `ITargetBlock<TIn>` - write-only
 - `IPropagatorBlock<TIn, TOut>` - source & target
- Blocks can be linked into pipelines or even graphs
 - `ISourceBlock<TOut>.LinkTo(target)`
 - Arbitrary number of links (0:N)
 - Optionally may receive a filtering predicate

Installed as a separate
NuGet package

Dataflow



- Programming Model
 - Source block calls **OfferMessage ()** of target
 - Message can be accepted, rejected, or postponed
 - Postponed messages can be reserved for later
`ISourceBlock.ReserveMessage ()`
 - Reserved messages should be later processed
`ISourceBlock.ConsumeMessage (), ReleaseReservation ()`
 - Sending messages externally (outside the dataflow)
 - `Post ()`, `SendAsync ()`
 - `Receive ()`, `ReceiveAsync ()`, `TryReceive ()`
 - Support for the completion concept
 - Completion may propagate and can be waited for

Dataflow



- Predefined blocks
 - Buffering
 - **BufferBlock<T>** - generic buffer
 - **BroadcastBlock<T>** - holds only the most recent value
 - **WriteOnceBlock<T>** - holds the first written value
 - Execution
 - **ActionBlock<TIn>** - calls a delegate on receive
 - **TransformBlock<TIn, TOut>** - gets a delegate which acts as map() function
 - **TransformManyBlock<TIn, TOut>** - similar to **TransformBlock**, but produces zero or more outputs per each input value

Dataflow



- Predefined grouping blocks
 - **BatchBlock<T>**
 - Reads a given number of messages and returns them as an array
 - **JoinBlock<T1, T2>, JoinBlock<T1, T2, T3>**
 - Does not implement **ITargetInput**, but provides **Target1**, **Target2** (and **Target3**) properties with this interface
 - Zips given messages in tuples
 - **BatchedJoinBlock<T1, T2>** (or **<T1, T2, T3>**)
 - Combination of batch and join block
 - Produces an array of tuples

PLINQ



- Parallel Language-Integrated Queries
 - LINQ – SQL-like in-memory lazy-evaluated queries conducted on enumerable containers
 - Parallel LINQ can resolve the queries concurrently
 - `var res = from x in data.AsParallel()
 where x > 10 select x;`
 - PLINQ may choose seq. evaluation based on heuristics, but we can override by selecting the execution mode or even selecting the degree of parallelism
 - Enforcing ordering
 - `.AsParallel().AsOrdered()`
 - `query.ForAll()` instead of using `foreach`

Discussion

