contents

preface xix
acknowledgments xxiii
about this book xxv
about the author xxix
about the cover illustrator xxxi

PART 1 BENEFITS OF FUNCTIONAL PROGRAMMING APPLICABLE TO CONCURRENT PROGRAMS 1

1 *Functional concurrency foundations* 3

1.1 What you'll learn from this book 5

1.2 Let's start with terminology 6

Sequential programming performs one task at a time 6 ▪ *Concurrent programming runs multiple tasks at the same time* 7 ▪ *Parallel programming executes multiples tasks simultaneously* 8 ▪ *Multitasking performs multiple tasks concurrently over time* 10 ▪ *Multithreading for performance improvement* 11

1.3 Why the need for concurrency? 12

Present and future of concurrent programming 14

1.4 The pitfalls of concurrent programming 15

Concurrency hazards 16 ▪ The sharing of state evolution 18 ▪ A simple real-world example: parallel quicksort 18 ▪ Benchmarking in F# 22

1.5 Why choose functional programming for concurrency? 23

Benefits of functional programming 25

1.6 Embracing the functional paradigm 26

1.7 Why use F# and C# for functional concurrent programming? 27

2 *Functional programming techniques for concurrency 30*

2.1 Using function composition to solve complex problems 31

Function composition in C# 31 ▪ Function composition in F# 33

2.2 Closures to simplify functional thinking 34

Captured variables in closures with lambda expressions 36 ▪ Closures in a multithreading environment 37

2.3 Memoization-caching technique for program speedup 39

2.4 Memoize in action for a fast web crawler 43

2.5 Lazy memoization for better performance 46

Gotchas for function memoization 47

2.6 Effective concurrent speculation to amortize the cost of expensive computations 48

Precomputation with natural functional support 50 ▪ Let the best computation win 51

2.7 Being lazy is a good thing 52

Strict languages for understanding concurrent behaviors 53 Lazy caching technique and thread-safe Singleton pattern 54 ▪ Lazy support in F# 56 ▪ Lazy and Task, a powerful combination 56

3 *Functional data structures and immutability 59*

3.1 Real-world example: hunting the thread-unsafe object 60

.NET immutable collections: a safe solution 63 ▪ .NET concurrent collections: a faster solution 68 ▪ The agent message-passing pattern: a faster, better solution 70

3.2 Safely sharing functional data structures among threads 72

3.3 Immutability for a change 73

Functional data structure for data parallelism 76 ▪ Performance implications of using immutability 76 ▪ Immutability in C# 77 ▪ Immutability in F# 79 ▪ Functional lists: linking cells in a chain 80 ▪ Building a persistent data structure: an immutable binary tree 86

3.4 Recursive functions: a natural way to iterate 88

The tail of a correct recursive function: tail-call optimization 89
Continuation passing style to optimize recursive function 90

PART 2 HOW TO APPROACH THE DIFFERENT PARTS OF A CONCURRENT PROGRAM 95

4 *The basics of processing big data: data parallelism, part 1 97*

4.1 What is data parallelism? 98

Data and task parallelism 99 ▪ The "embarrassingly parallel" concept 100 ▪ Data parallelism support in .NET 101

4.2 The Fork/Join pattern: parallel Mandelbrot 102

When the GC is the bottleneck: structs vs. class objects 107
The downside of parallel loops 110

4.3 Measuring performance speed 110

Amdahl's Law defines the limit of performance improvement 111
Gustafson's Law: a step further to measure performance improvement 112 ▪ The limitations of parallel loops: the sum of prime numbers 112 ▪ What can possibly go wrong with a simple loop? 114 ▪ The declarative parallel programming model 115

5 *PLINQ and MapReduce: data parallelism, part 2 118*

5.1 A short introduction to PLINQ 119

How is PLINQ more functional? 120 ▪ PLINQ and pure functions: the parallel word counter 121 ▪ Avoiding side effects with pure functions 123 ▪ Isolate and control side effects: refactoring the parallel word counter 124

5.2 Aggregating and reducing data in parallel 125

Deforesting: one of many advantages to folding 127 ▪ Fold in PLINQ: Aggregate functions 129 ▪ Implementing a parallel Reduce function for PLINQ 135 ▪ Parallel list comprehension in F#: PSeq 137 ▪ Parallel arrays in F# 137

5.3 Parallel MapReduce pattern 139

The Map and Reduce functions 140 ▪ Using MapReduce with the NuGet package gallery 141

6 *Real-time event streams: functional reactive programming 148*

6.1 Reactive programming: big event processing 149

6.2 .NET tools for reactive programming 152

Event combinators—a better solution 153 ▪ .NET interoperability with F# combinators 154

6.3 Reactive programming in .NET: Reactive Extensions (Rx) 156

From LINQ/PLINQ to Rx 159 ▪ IObservable: the dual IEnumerable 160 ▪ Reactive Extensions in action 161 Real-time streaming with RX 162 ▪ From events to F# observables 163

6.4 Taming the event stream: Twitter emotion analysis using Rx programming 164

SelectMany: the monadic bind operator 171

6.5 An Rx publisher-subscriber 173

Using the Subject type for a powerful publisher-subscriber hub 173 ▪ Rx in relation to concurrency 174 ▪ Implementing a reusable Rx publisher-subscriber 175 ▪ Analyzing tweet emotions using an Rx Pub-Sub class 177 ▪ Observers in action 179 ▪ The convenient F# object expression 180

7 *Task-based functional parallelism 182*

7.1 A short introduction to task parallelism 183

Why task parallelism and functional programming? 184 ▪ Task parallelism support in .NET 185

7.2 The .NET Task Parallel Library 187

Running operations in parallel with TPL Parallel.Invoke 188

7.3 The problem of void in C# 191

The solution for void in C#: the unit type 191

7.4 Continuation-passing style: a functional control flow 193

Why exploit CPS? 194 ▪ Waiting for a task to complete: the continuation model 195

7.5 Strategies for composing task operations 200

Using mathematical patterns for better composition 201 ▪ Guidelines for using tasks 207

7.6 The parallel functional Pipeline pattern 207

8 *Task asynchronicity for the win 213*

8.1 The Asynchronous Programming Model (APM) 214

The value of asynchronous programming 215 ▪ Scalability and asynchronous programming 217 ▪ CPU-bound and I/O-bound operations 218

8.2 Unbounded parallelism with asynchronous programming 219

8.3 Asynchronous support in .NET 220

Asynchronous programming breaks the code structure 223
Event-based Asynchronous Programming 223

8.4 C# Task-based Asynchronous Programming 223

Anonymous asynchronous lambdas 226 ▪ Task<T> is a monadic container 227

8.5 Task-based Asynchronous Programming: a case study 230

Asynchronous cancellation 234 ▪ Task-based asynchronous composition with the monadic Bind operator 238 ▪ Deferring asynchronous computation enables composition 239 ▪ Retry if something goes wrong 240 ▪ Handling errors in asynchronous operations 241 ▪ Asynchronous parallel processing of the historical stock market 243 ▪ Asynchronous stock market parallel processing as tasks complete 245

9 *Asynchronous functional programming in F# 247*

9.1 Asynchronous functional aspects 248

9.2 What's the F# asynchronous workflow? 248

The continuation passing style in computation expressions 249 ▪ The asynchronous workflow in action: Azure Blob storage parallel operations 251

9.3 Asynchronous computation expressions 256

Difference between computation expressions and monads 257 ▪ AsyncRetry: building your own computation expression 259 ▪ Extending the asynchronous workflow 261 ▪ Mapping asynchronous operation: the Async .map functor 262 ▪ Parallelize asynchronous workflows: Async.Parallel 264 ▪ Asynchronous workflow cancellation support 268 ▪ Taming parallel asynchronous operations 271

10 Functional combinators for fluent concurrent programming 275

10.1 The execution flow isn't always on the happy path: error handling 276

The problem of error handling in imperative programming 277

10.2 Error combinators: Retry, Otherwise, and Task.Catch in C# 279

Error handling in FP: exceptions for flow control 282 ▪ Handling errors with Task<Option<T>> in C# 284 ▪ The F# AsyncOption type: combining Async and Option 284 ▪ Idiomatic F# functional asynchronous error handling 286 ▪ Preserving the exception semantic with the Result type 287

10.3 Taming exceptions in asynchronous operations 290

Modeling error handling in F# with Async and Result 295 ▪ Extending the F# AsyncResult type with monadic bind operators 296

10.4 Abstracting operations with functional combinators 300

10.5 Functional combinators in a nutshell 301

The TPL built-in asynchronous combinators 301 ▪ Exploiting the Task.WhenAny combinator for redundancy and interleaving 302 ▪ Exploiting the Task.WhenAll combinator for asynchronous for-each 304 ▪ Mathematical pattern review: what you've seen so far 305

10.6 The ultimate parallel composition applicative functor 308

Extending the F# async workflow with applicative functor operators 315 ▪ Applicative functor semantics in F# with infix operators 317 ▪ Exploiting heterogeneous parallel computation with applicative functors 318 ▪ Composing and executing heterogeneous parallel computations 319 ▪ Controlling flow with conditional asynchronous combinators 321 ▪ Asynchronous combinators in action 325

11 *Applying reactive programming everywhere with agents 328*

11.1 What's reactive programming, and how is it useful? 330

11.2 The asynchronous message-passing programming model 331

Relation with message passing and immutability 334
Natural isolation 334

11.3 What is an agent? 334

The components of an agent 335 ▪ What an agent can do 336 ▪ The share-nothing approach for lock-free concurrent programming 336 ▪ How is agent-based programming functional? 337 ▪ Agent is object-oriented 338

11.4 The F# agent: MailboxProcessor 338

The mailbox asynchronous recursive loop 340

11.5 Avoiding database bottlenecks with F# MailboxProcessor 341

The MailboxProcessor message type: discriminated unions 344 ▪ MailboxProcessor two-way communication 345 ▪ Consuming the AgentSQL from C# 346 ▪ Parallelizing the workflow with group coordination of agents 347 ▪ How to handle errors with F# MailboxProcessor 349 ▪ Stopping MailboxProcessor agents—CancellationToken 350 ▪ Distributing the work with MailboxProcessor 351 ▪ Caching operations with an agent 352 ▪ Reporting results from a MailboxProcessor 357 ▪ Using the thread pool to report events from MailboxProcessor 359

11.6 F# MailboxProcessor: 10,000 agents for a game of life 359

12 *Parallel workflow and agent programming with TPL Dataflow 365*

12.1 The power of TPL Dataflow 366

12.2 Designed to compose: TPL Dataflow blocks 367

Using BufferBlock<TInput> as a FIFO buffer 368 ▪ Transforming data with TransformBlock<TInput, TOutput> 369 ▪ Completing the work with ActionBlock<TInput > 370 ▪ Linking dataflow blocks 372

12.3 Implementing a sophisticated Producer/Consumer with TDF 372

A multiple Producer/single Consumer pattern: TPL Dataflow 372
A single Producer/multiple Consumer pattern 374

12.4 Enabling an agent model in C# using TPL Dataflow 374

Agent fold-over state and messages: Aggregate 377
Agent interaction: a parallel word counter 378

12.5 A parallel workflow to compress and encrypt a large stream 382

Context: the problem of processing a large stream of data 383 ▪ Ensuring the order integrity of a stream of messages 388 ▪ Linking, propagating, and completing 389 ▪ Rules for building a TDF workflow 390 ▪ Meshing Reactive Extensions (Rx) and TDF 391

PART 3 MODERN PATTERNS OF CONCURRENT PROGRAMMING APPLIED..... 395

13 Recipes and design patterns for successful concurrent programming 397

13.1 Recycling objects to reduce memory consumption 398

Solution: asynchronously recycling a pool of objects 399

13.2 Custom parallel Fork/Join operator 401

Solution: composing a pipeline of steps forming the Fork/Join pattern 402

13.3 Parallelizing tasks with dependencies: designing code to optimize performance 404

Solution: implementing a dependencies graph of tasks 405

13.4 Gate for coordinating concurrent I/O operations sharing resources: one write, multiple reads 409

Solution: applying multiple read/write operations to shared thread-safe resources 409

13.5 Thread-safe random number generator 414

Solution: using the ThreadLocal object 415

13.6 Polymorphic event aggregator 416

Solution: implementing a polymorphic publisher-subscriber pattern 416

13.7 Custom Rx scheduler to control the degree of parallelism 419

Solution: implementing a scheduler with multiple concurrent agents 419

13.8 Concurrent reactive scalable client/server 422

Solution: combining Rx and asynchronous programming 423

13.9 Reusable custom high-performing parallel filter-map operator 431

Solution: combining filter and map parallel operations 431

13.10 Non-blocking synchronous message-passing model 435

Solution: coordinating the payload between operations using the agent programming model 436

13.11 Coordinating concurrent jobs using the agent programming model 440

Solution: implementing an agent that runs jobs with a configured degree of parallelism 441

13.12 Composing monadic functions 444

Solution: combining asynchronous operations using the Kleisli composition operator 445

14 Building a scalable mobile app with concurrent functional programming 449

14.1 Functional programming on the server in the real world 450

14.2 How to design a successful performant application 451

The secret sauce: ACD 452 ▪ A different asynchronous pattern: queuing work for later execution 453

14.3 Choosing the right concurrent programming model 454

Real-time communication with SignalR 457

14.4 Real-time trading: stock market high-level architecture 457

14.5 Essential elements for the stock market application 461

14.6 Let's code the stock market trading application 462

Benchmark to measure the scalability of the stock ticker application 482

appendix A Functional programming 484

appendix B F# overview 498

appendix C Interoperability between an F# asynchronous workflow and .NET Task 513

index 516