

Einführung	XI
Teil I: Der Überblick.....	1
1 Concurrency mit modernem C++	3
1.1 C++11 und C++14: Die Grundlagen	3
1.1.1 Das Speichermodell.....	4
1.1.2 Multithreading.....	4
1.2 C++17: Die parallelen Algorithmen der Standard Template Library.....	6
1.2.1 Ausführungsstrategie	7
1.2.2 Neue Algorithmen.....	7
1.3 Fallstudien	7
1.3.1 Berechnung der Summe eines Vektors	7
1.3.2 Thread-sichere Initialisierung eines Singletons	7
1.3.3 Fortwährende Optimierung mit CppMem.....	7
1.4 C++20: Die concurrent Zukunft	8
1.4.1 Atomare Smart Pointer	8
1.4.2 Erweiterte Futures	8
1.4.3 Latches und Barriers.....	9
1.4.4 Coroutinen	9
1.4.5 Transaction Memory.....	9
1.4.6 Task-Blöcke	9
1.5 Herausforderungen.....	10
1.6 Best Practice	10
1.7 Zeitbibliothek	10
1.8 Glossar	10

Teil II: Die Details	11
2 Das Speichermodell	13
2.1 Der Vertrag.....	13
2.1.1 Die Grundlagen.....	14
2.1.2 Die Herausforderungen	15
2.2 Atomare Datentypen	16
2.2.1 Starkes versus schwaches Speichermodell	16
2.2.2 <code>std::atomic_flag</code>	18
2.2.3 Das Klassen-Template <code>std::atomic</code>	23
2.2.4 Benutzerdefinierte atomare Datentypen	30
2.2.5 Die atomaren Operationen	31
2.2.6 Freie atomare Funktionen	31
2.2.7 <code>std::shared_ptr</code>	31
2.3 Synchronisations- und Ordnungsbedingungen	34
2.3.1 Die sechs Variationen des C++-Speichermodells	34
2.3.2 Sequenzielle Konsistenz.....	36
2.3.3 Acquire-Release-Semantik.....	38
2.3.4 <code>std::memory_order_consume</code>	46
2.3.5 Relaxed-Semantik	51
2.4 Fences	53
2.4.1 <code>atomic_thread_fence</code> als Speicherbarriere	53
2.4.2 Die drei Fences	54
2.4.3 Acquire Release Fences	56
2.4.4 Synchronisation mit atomaren Variablen oder Fences	57
3 Multithreading	63
3.1 Threads.....	64
3.1.1 Erzeugung.....	64
3.1.2 Lebenszeit.....	65
3.1.3 Argumente	68
3.1.4 Methoden	71
3.2 Geteilte Daten	73
3.2.1 Mutexe	75
3.2.2 Locks	81
3.2.3 Thread-sichere Initialisierung	90
3.3 Thread-lokale Daten	96
3.4 Bedingungsvariablen.....	98
3.4.1 Der Arbeitsablauf.....	100

3.4.2	Lost Wakeup und Spurious Wakeup	102
3.5	Tasks	102
3.5.1	Tasks versus Threads.....	103
3.5.2	std::async	104
3.5.3	std::packaged_task	110
3.5.4	std::promise und std::future	112
3.5.5	Tasks als sicherer Ersatz für Bedingungsvariablen	119
4	Parallele Algorithmen der Standard Template Library.....	123
4.1	Ausführungsstrategie.....	124
4.1.1	Parallel und vektorisierte Ausführungsstrategie	124
4.2	Algorithmen	126
4.3	Die neuen Algorithmen	126
4.3.1	Das funktionale Erbe	130
5	Fallstudien	133
5.1	Berechnen der Summe eines Vektors	134
5.1.1	Single-threaded Summation.....	134
5.1.2	Multi-threaded Summation mit einer geteilten Variable	140
5.1.3	Thread-lokale Summation	146
5.1.4	Schlussfolgerung	155
5.2	Thread-sichere Initialisierung eines Singletons	156
5.2.1	Double-Checked Locking Pattern	156
5.2.2	Performanzmessung.....	158
5.2.3	Das Thread-sichere Meyers Singleton.....	160
5.2.4	std::lock_guard	162
5.2.5	std::call_once mit dem std::once_flag.....	163
5.2.6	Atomare Variablen	164
5.2.7	Performanzzahlen der verschiedenen Thread-sicheren Implementierungen	168
5.3	Fortwährende Optimierung mit CppMem	168
5.3.1	CppMem – Ein Überblick	170
5.3.2	CppMem: Nicht-atomare Variablen	173
5.3.3	CppMem: Locks	178
5.3.4	CppMem: Atomare Variablen mit sequenzieller Konsistenz	179
5.3.5	CppMem: Atomare Variablen mit Acquire-Release-Semantik	184
5.3.6	CppMem: Atomare Variablen mit nicht-atomaren Variablen.....	188
5.3.7	CppMem: Atomare Variablen mit Relaxed-Semantik	189
5.4	Schlussfolgerung.....	191

6	C++20	193
6.1	Atomare Smart Pointer	194
6.1.1	Eine Thread-sichere, einfach verkettete Liste	194
6.2	Erweiterte Futures	196
6.2.1	std::future	196
6.2.2	std::async, std::packaged_task und std::promise	197
6.2.3	Neue Futures erzeugen	198
6.3	Latches und Barriers	200
6.3.1	std::latch	201
6.3.2	std::barrier	202
6.3.3	std::flex_barrier	203
6.4	Coroutinen	204
6.4.1	Die Generatorfunktion	205
6.4.2	Die Details	207
6.5	Transactional Memory	209
6.5.1	ACI(D)	209
6.5.2	Synchronized- und atomic-Blöcke	210
6.5.3	Transaction safe versus Transaction unsafe Code	213
6.6	Task-Blöcke	214
6.6.1	Fork und join	214
6.6.2	define_task_block vs. define_task_block_restore_thread	215
6.6.3	Das Interface	216
6.6.4	Der Scheduler	217
Teil III:	Anhang	219
A	Herausforderungen	221
A.1	ABA	221
A.1.1	Eine Analogie	221
A.1.2	Ein unkritisches ABA-Szenario	222
A.1.3	Eine lock-freie Datenstruktur	222
A.1.4	Das ABA-Problem	223
A.1.5	Lösung des ABA-Problems	223
A.2	Blockieren	225
A.3	Verletzung von Programminvarianten	226
A.4	Data Race	228
A.5	False Sharing	229
A.6	Lebenszeitprobleme von Variablen	230

A.7	Race Conditions.....	231
A.8	Threads verschieben	231
A.9	Deadlock	233
B	Best Practice	235
B.1	Allgemein	235
B.1.1	Code Reviews	235
B.1.2	Minimiere Sie das Teilen von veränderlichen Variablen	236
B.1.3	Minimieren Sie das Warten	236
B.1.4	Verwenden Sie dynamische Codeanalyse-Werkzeuge	236
B.1.5	Verwenden Sie statische Codeanalyse-Werkzeuge	237
B.1.6	Wenden Sie die passende Abstraktion an	238
B.1.7	Ziehen Sie unveränderliche Daten vor	238
B.2	Multithreading	238
B.2.1	Threads	238
B.2.2	Teilen von Variablen	240
B.2.3	Bedingungsvariablen	243
B.2.4	Promises und Futures	246
B.3	Speichermodell	247
B.3.1	Programmieren Sie nicht Lock-frei	247
B.3.2	Setzen Sie bewährte Muster zum Lock-freien Programmieren ein	247
B.3.3	Verwenden Sie die Zusicherungen des Speichermodells	247
B.3.4	Verwenden Sie volatile nicht zur Synchronisation	248
C	Die Zeitbibliothek	249
C.1	Das Zusammenspiel des Zeitpunkts, der Zeitdauer und des Zeitgebers	249
C.2	Zeitpunkt.....	250
C.2.1	Vom Zeitpunkt zur Kalenderzeit	250
C.2.2	Bruch des gültigen Zeitbereichs.....	252
C.3	Zeitdauer	253
C.3.1	Berechnungen	255
C.4	Zeitgeber	258
C.4.1	Genauigkeit und Stetigkeit	258
C.4.2	Die Epoche	261
C.5	Schlafen und Warten	263
C.5.1	Konventionen.....	263
C.5.2	Verschiedene Wartestrategien	264
	Glossar	269
	Stichwortverzeichnis	271