Aufgabe 1

Den optimalen Speedup erreicht man, indem man in diesem Beispiel die äusserste Schlaufe der Matrix Multiplikation parallelisiert. Bei kleinen Werten von N lohnt es sich auch die zweite Schlaufe zu parallelisieren.

Folgende Laufzeiten wurden mit einem Intel Core i7-4712MQ (4 physikalische Cores, 8 logische Kerne), 2.3 GHz, 64-bit mit optimierter Kompilation ("Release" build configuration) gemessen. Durchschnitt von 3 aufeinanderfolgenden Testläufen, in Sekunden, gerundet auf 2 signifikante Stellen.

N = 200, M = 400, K = 600 von decimal (128-Bit) Zahlen

Version	Time [sec]	Speedup
Sequential	13.0	1
Outer parallel for-loop	**2.8**	**4.6**

Es wird ein Speedup von ca. 4.6 auf einer 4-Core-Maschine erreicht. Der Speedup ist leicht höher als die Anzahl echter Cores, weil der Gewinn logischer Prozessoren (Hyper-Threading) wegen Memory Stalls auf den Array-Zugriffen zugute kommt.

Aufgabe 2

Die nachfolgenden Messungen wurden mit der gleichen Test-Umgebung wie vorher durchgeführt. Die Resultate sind für beide Varianten weitgehend gleich. Die rot hervorgehobenen Resultate zeigen eine effiziente Konfiguration, die sowohl bei kleineren Arrays und einer höheren Anzahl Cores einen genügend hohen Parallelisierungsgrad bietet.

Array von 50 Millionen int Zahlen.

Performance-Problem:

- ⇨ In der Musterlösung ist der TPL-Aufruf in einer separaten Hilfsmethoden. Dies bringt einen entscheidenden Performance-Gewinn. Wenn der TPL-Aufruf direkt in der Haupt-Quicksort-Methode inlined ist, wird nicht mehr genügend optimiert (JITer).

Es ist am effizientesten, jeweils nur einen Subtask bei der Parallelisierung in einem Schritt zu starten und joinen (siehe Musterlösung). Die Messresultate sind für die Varianten von je zwei expliziten Sub-Tasks und Parallel.Invoke() sind aber bei Konfigurationen mit gutem Speedup (z.B. Threshold = 10,000) weitgehend gleich.

Es wird ein Speedup von 3.5 auf der 4 Core Maschine erreicht, was gut ist.

Threshold	Laufzeit [Sek]	Speedup
- (sequential, template)	5.6	1
1	4.3	1.3
10	2.2	2.5
100	1.7	3.3

1000	**1.6**	**3.5**
10,000	**1.6**	**3.5**
100,000	1.8	3.1
1,000,000	1.8	3.1
10,000,000	3.2	1.8
100,000,000 (sequential)	5.5	1

Aufgabe 3

Der Deadlock entsteht dadurch, dass die Tasks gegenseitige Warteabhängigkeiten haben und die Anzahl Worker Threads hier beschränkt ist (`ThreadPool.SetMaxThreads`). Ohne Angabe dieser Limite erzeugt die TPL dynamisch neue Threads (Thread Injection). Dies lässt sich am untenstehenden Screenshot der Task Debugging Ansicht gut erkennen (Breakpoint bei Console-Ausgabe in Task mit Bedingung k == 1 setzen): Jeder Task wird mit einem neuen Thread ausgeführt. Diese Lösung ist wegen der nicht-hierarchischen Warteabhängigkeiten grundsätzlich nicht für einen Thread Pool geeignet: Es führt zu Erzeugung vieler Threads durch die Thread Injection (was verzögert und ineffizient ist) oder bei einer limitierten Anzahl Worker Threads zu Deadlocks.

Parallel Tasks

ID	Status	Location	Task	Thread Assignment	AppDomain
1	Running	CyclicTa	Main.An	6084 (Worker Threa	1 (CyclicTask
2	Running	CyclicTa	Main.An	3240 (Worker Threa	1 (CyclicTask
3	Running	CyclicTa	Main.An	5348 (Worker Threa	1 (CyclicTask
4	Running	CyclicTa	Main.An	5024 (Worker Threa	1 (CyclicTask
5	Running	CyclicTa	Main.An	4864 (Worker Threa	1 (CyclicTask
6	Running	CyclicTa	Main.An	2752 (Worker Threa	1 (CyclicTask
7	Running	CyclicTa	Main.An	5732 (Worker Threa	1 (CyclicTask
8	Running	CyclicTa	Main.An	560 (Worker Thread	1 (CyclicTask
9	Running	CyclicTa	Main.An	4940 (Worker Threa	1 (CyclicTask
10	Running	CyclicTa	Main.An	2880 (Worker Threa	1 (CyclicTask