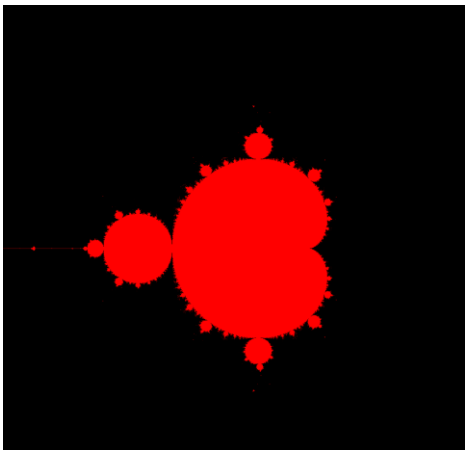
Florian Ramaj und John Ervin-Laylo

Dokumentation – Mandelbrot

Aufgabenstellung

Die Aufgabenstellung ist die Mandelbrot-Menge (siehe Screenshot von der fertigen Lösung):

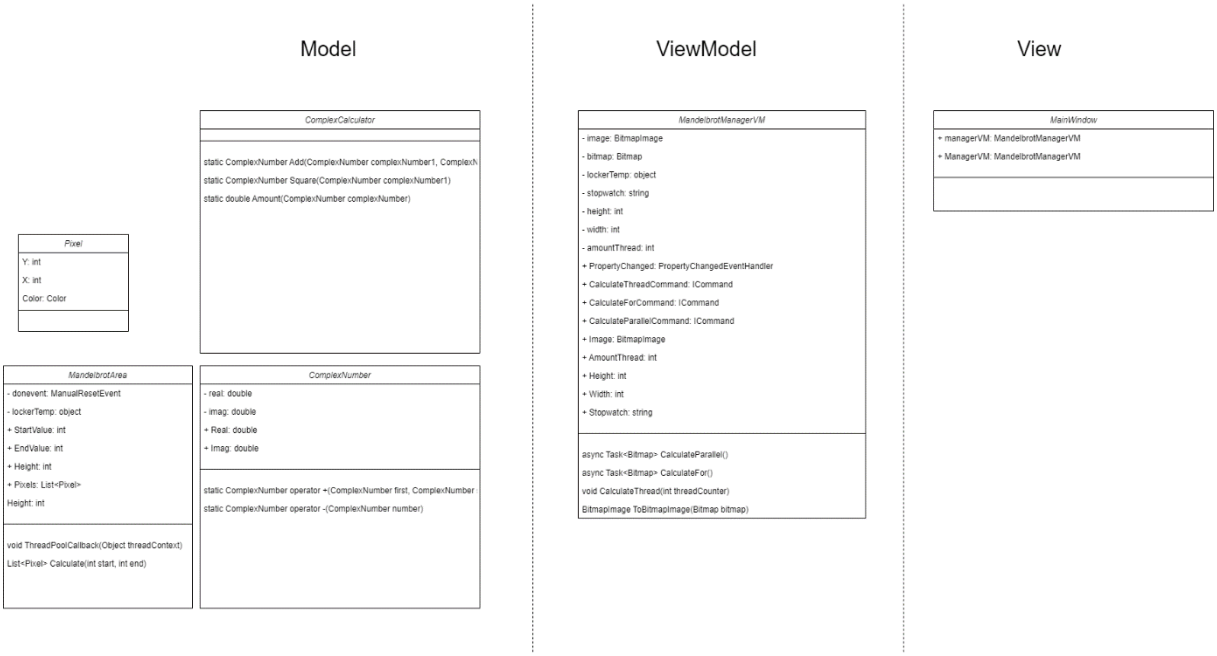


Bilder berechnet man, indem man jedem Pixel (x,y) eines Bildes eine komplexe Zahl zuordnet ($c = c_0 + a * x + b_i * y$) und beginnend mit $z_0=0$ untersucht, ob und wann die Iterationen anfangen, zu „explodieren“. Bleiben die Werte klein, wird das Pixel häufig schwarz gefärbt, kommt es zu einer „Explosion“ der Zahlenwerte, wird die Anzahl der dafür notwendigen Iterationen als Farbe kodiert.

Mandelbrot

Klassendesign

Grundsätzlich haben wir uns für das Model View ViewModel Muster entschieden.



Diesbezüglich gibt es im Model folgende Klassen:

- Pixel
  Beinhaltet folgende Werte, wie X, Y und Color. Diese Werte werden in der View benötigt um das Mandelbrot anzuzeigen.
- ComplexCalculator
  Diese Klasse ermöglicht das Rechnen mit komplexen Nummern.
- ComplexNumber
  Diese Klasse selbst repräsentiert selbst eine komplexe Zahl, die selbst einen realen und imaginären Teil besitzt.
- MandelbrotArea
  Auf diese Klasse selbst wird noch genauer eingegangen im Kapitel „3. For-Schleife mit N-Threads“.

Das Viewmodel selbst besitzt selbst nur eine Klasse die alles behandelt. Die Klasse selbst beinhaltet die drei Rechnungsmethoden:

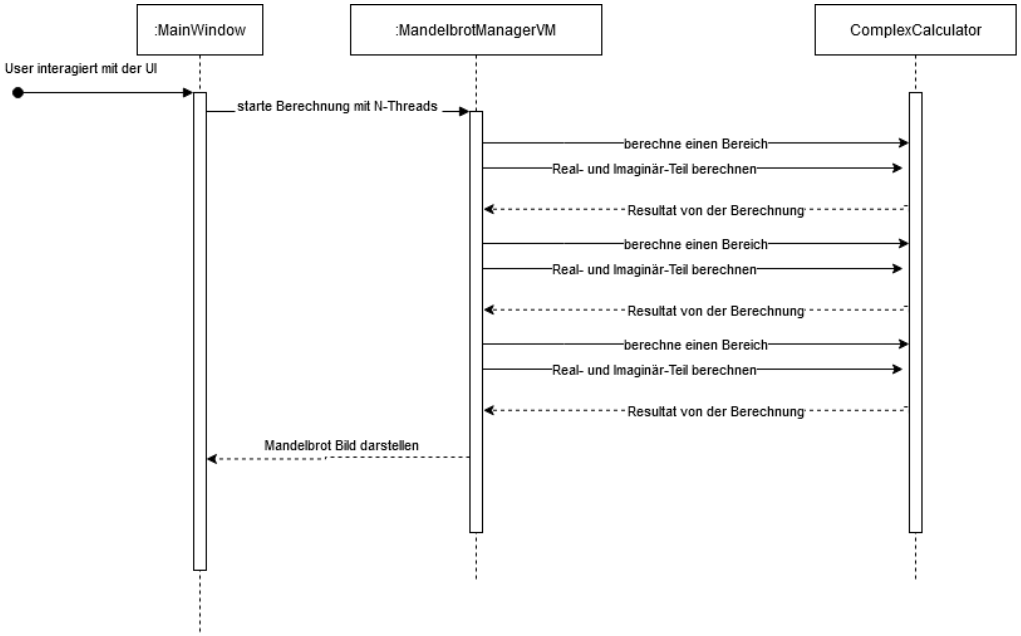
- Berechnung mit normalen for-Schleifen
- Berechnung mit parallel-for-Schleifen
- Berechnung mit N-Thread und normalen for-Schleifen

Die View selbst besteht aus einem XAML Dokument, welches Visuell die View darstellt und zusätzlich ein „Codebehind“, welches die View mit dem Viewmodel verbindet.

1. For-Schleife mit einem Thread

Hierzu wurde über zwei for-Schleifen iteriert und die Punkte berechnet.

Sequenzdiagramm

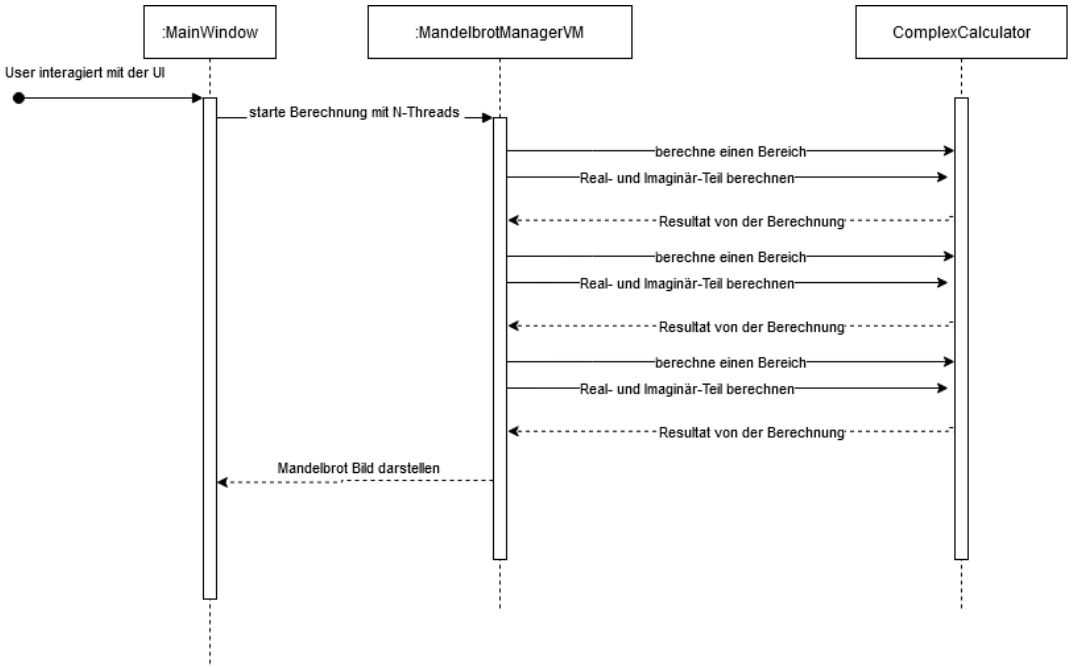


2.Parallel-Schleife mit einem Thread

Bei diesen Lösungsansatz wurden statt normalen for-Schleifen, die „Parallel.For“ Schleife verwendet.

Parallel.For versucht, die Anzahl der benötigten Threads zu minimieren, um einen optimalen Durchsatz zu erreichen. Für rechenintensive Arbeitslasten bedeutet dies in der Regel, dass die Anzahl der Threads ungefähr der Anzahl der Kerne in Ihrem Rechner entspricht, obwohl sie je nach Arbeitslast etwas höher sein kann. Parallel.For versucht definitiv nicht, einen Thread pro Iteration zu erstellen. Das würde erhebliche Kosten verursachen, und ein Teil des Zwecks von Parallel.For ist es, solche Kosten zu vermeiden. Was Deadlocks angeht, so ist das in den meisten realen Szenarien, die von einer parallelen Schleife profitieren würden, kein Problem. Wenn Sie jedoch alle Iterationen der parallelen Schleife als potenziell gleichzeitig ablaufend betrachten, könnten Sie in genau denselben Situationen in Deadlocks geraten, als wenn Sie für jede Iteration einen Thread gestartet hätten. Dies würde voraussetzen, dass die Iterationen eine zyklische Abhängigkeit aufweisen, z. B. dass Iteration 1 von etwas abhängt, das von Iteration 2 ausgeführt wird, und dass Iteration 2 von etwas abhängt, das von Iteration 1 ausgeführt wird.

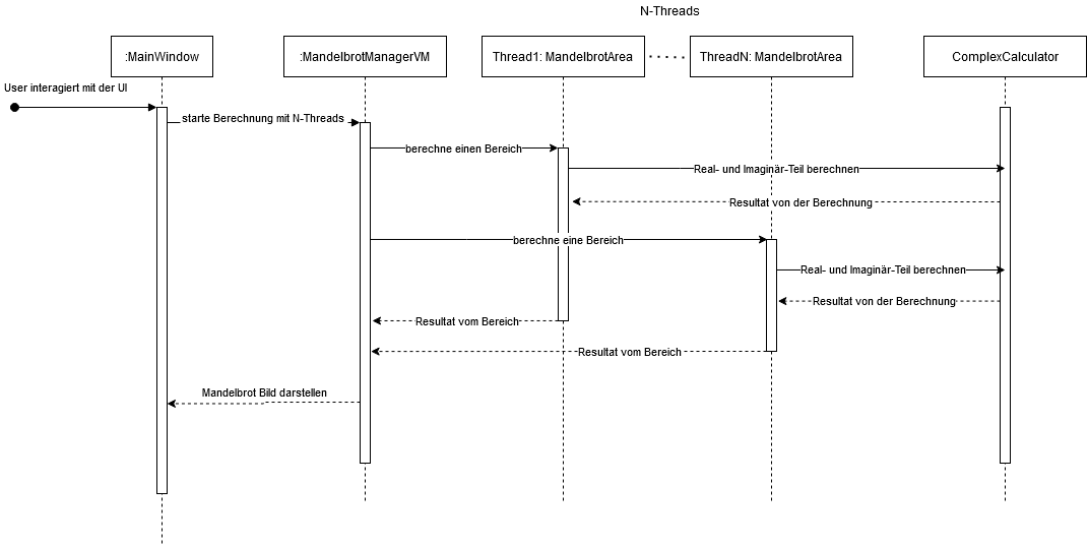
Sequenzdiagramm



3. For-Schleife mit N-Threads

Als Referenz wurde folgende Dokumentation genommen „https://docs.microsoft.com/en-us/dotnet/api/system.threading.threadpool.queueuserworkitem?view=net-6.0“. Es wird das Master Worker Pattern verwendet. Der Master teilt die Aufgaben unter den Worker, um genau zu sein wird in der Applikation die erwünschten Threads eingelesen und diese Anzahl wieder spiegeln die Worker. Zum Schluss wird auf alle Worker gewartet und deren Ergebnisse bearbeitet.

Sequenzdiagramm



Resultate

Im folgenden Kapitel finden Sie die Vergleiche der unterschiedlichen Lösungsansätze. Die Tests wurden unter folgenden Hardwaresettings durchgeführt:

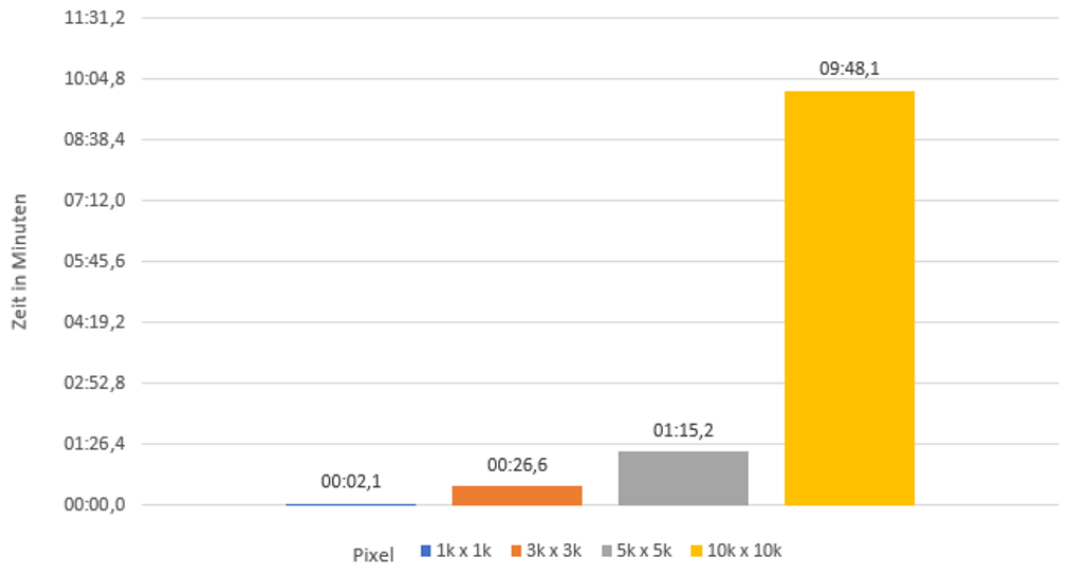
- Prozessor: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz, 2304 MHz, 4 Kerne, 8 logische Prozessoren
- Arbeitsspeicher: 16GB

1. Durchgang

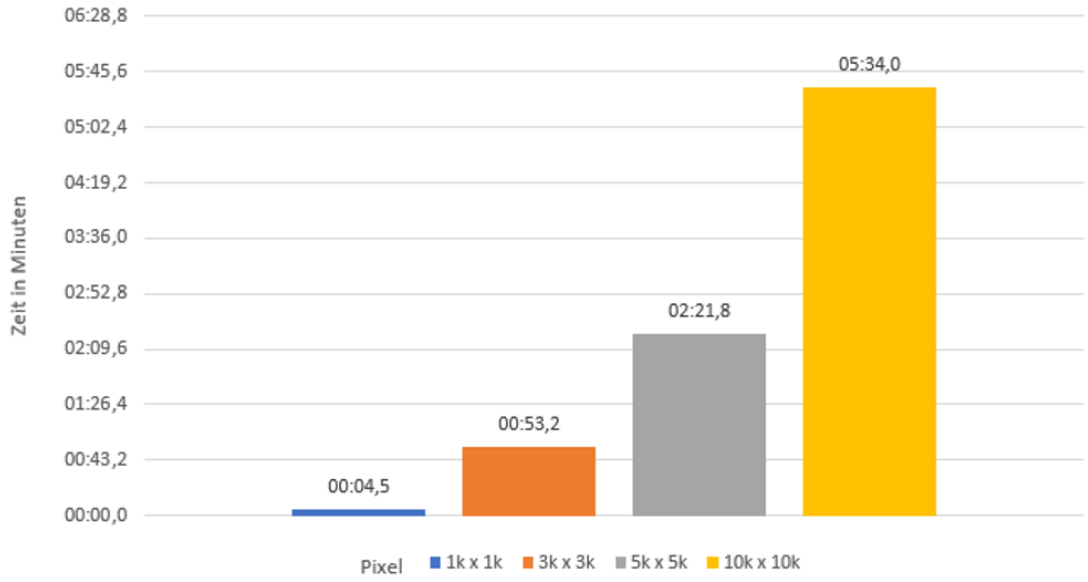
In der ersten Version ist uns ein Designfehler aufgefallen. Die Erstellung von Objekten verlangsamte die Berechnung drastisch, wie Sie in den folgenden Diagrammen sehen können.

Die For-Schleife und das Parallel-For liefen noch durch, doch bei 10+ Threads kam es zu einer Out-of-Memory-Exception.

For-Schleife



Parallel-For

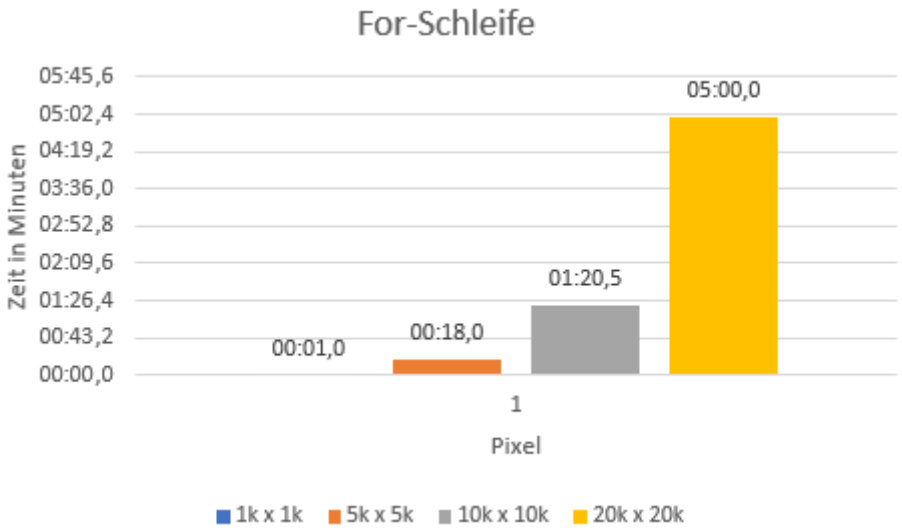


Zur Optimierung haben wir uns vorgenommen, die Klasse Pixel und die Klasse ComplexNumber durch Tuples zu ersetzen, damit wir bei der Berechnung keine Objekte erstellen.

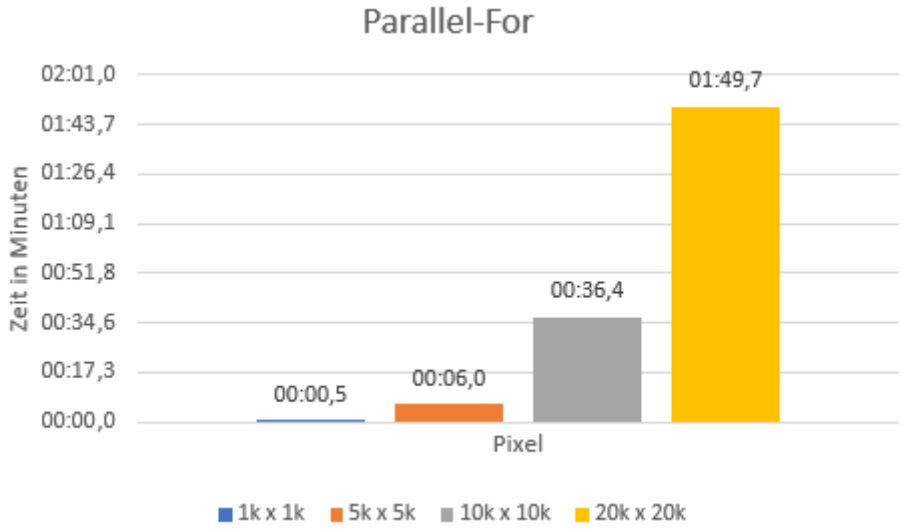
2. Durchgang

Beim zweiten Durchlauf kann man sehen, dass es sich stark verbessert hat. An letzter Stelle steht die For-Schleife, an zweiter Stelle die parallele For-Schleife und am schnellsten ist die Schleife mit den mehreren Threads.

For-Schleife



Parallel-For



Threads