Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Aufgabe 5

OpenMP+MPI, Gray+Blur

Mikhail Poberezhnyi

Sergey Wolf

01.01.2021

Fachbereich 4

Studiengang Angewandte Informatik

Betreuer: Herr Prof. Dr. Kovalenko

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Ausgansbild

Als Ausgangsbild wurde das Bild Lena.jpg verwendet, welches oft in dieser Art von Aufgaben eingesetzt wird. Das Original stammt aus `https://www.cosy.sbg.ac.at/~pmeerw/Watermarking/lena.html` [4]

1.2 Gaussian Blur

Der Gaußsche Weichzeichner, oder wie er oft auch im deutschsprachigen Raum genannt wird - Gaussian Blur, ist ein nichtlinearer Tiefpassfilter zur Rauschunterdrückung (LP-Filter). Der visuelle Effekt dieses Filters ist ein glattes, unscharfes Bild. Das bedeutet, dass er Intensitätsschwankungen zwischen benachbarten Pixeln reduziert. Der Gauß-Filter schneidet besser ab als andere gleichmäßige Tiefpassfilter wie der Mean-Filter [2]. Daher ist er ein bevorzugter Vorverarbeitungsschritt in Bildverarbeitungs- und Objekterkennungsalgorithmen. [2]

1.3 Arbeitsweise von Gaussian Blur

Der Gauß-Filter arbeitet so, dass das Eingangsbild mit einem Gauß-Kernel gefaltet wird. Dieser Prozess führt einen gewichteten Durchschnitt der Nachbarschaften des aktuellen Pixels durch, und zwar so, dass weit entfernte Pixel eine geringere Gewichtung erhalten als solche im Zentrum. Das Ergebnis ist ein unscharfes Bild mit besseren Kanten als bei anderen gleichmäßigen Glättungsalgorithmen.

1.4 Verwendeter Filter

Für die Lösung dieser Aufgabe haben wir einen 3x3 Filter verwendet, dieser kann bei Bedarf aber auch auf 5x5 umgestellt werden:

```
14 vector<vector<int>> ConvolutionEffects::
15 getFilterForEffectType(EffectType effectType) {
16     if (effectType == EffectType::GaussianBlur3x3) {
17         _filterHeight = 3;
18         _filterWidth = 3;
19         return {{1, 2, 1},
20                 {2, 4, 2},
21                 {1, 2, 1}};
22     } else if (effectType == EffectType::GaussianBlur5x5) {
23         _filterHeight = 5;
24         _filterWidth = 5;
25         return {
26                 {1, 4, 6, 4, 1},
27                 {4, 16, 24, 16, 4},
28                 {6, 24, 36, 24, 6},
29                 {4, 16, 24, 16, 4},
30                 {1, 4, 6, 4, 1},
31         };
32     }
33     ...
```

Listing 1.1: Verwender Filter, 3x3 bzw. 5x5

Kapitel 2

Implementierung

Der Prozess der Projektentwicklung fand auf zwei Betriebssystemen statt: Linux und Windows. Um die plattformübergreifende Ausführung des Programmcodes zu realisieren, wurde das Projekt mit Hilfe von Cmake organisiert. Zwei Dateien “main.cpp“ und “scatter_version.cpp“ können als Haupteinstiegspunkt bezeichnet werden. Die erste zeigt alle implementierten Algorithmen. Hier wird jeder Algorithmus in einem separaten Prozess ausgeführt.

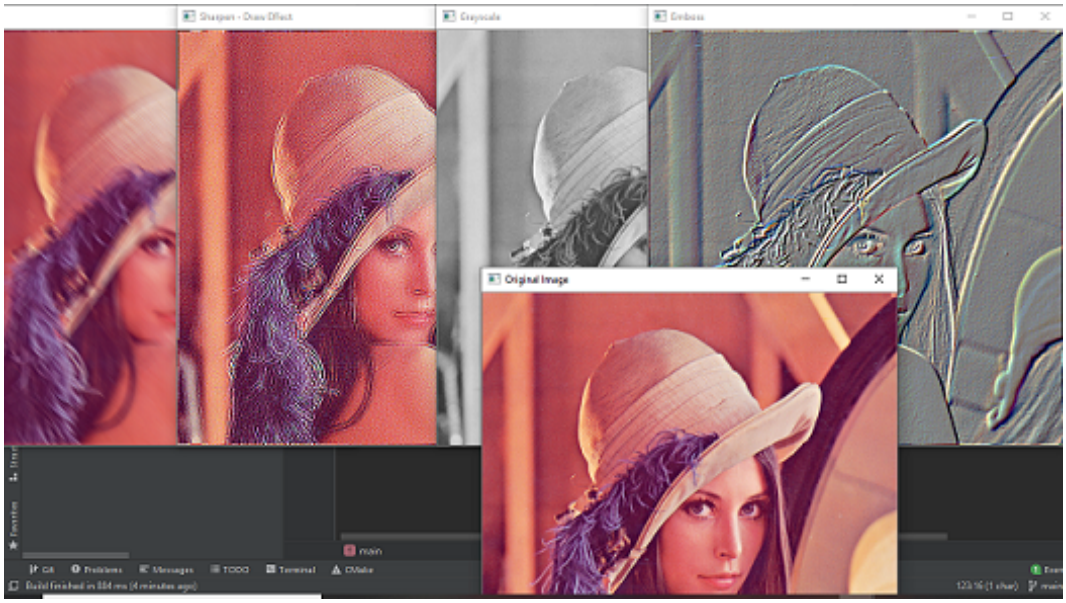

Abbildung 2.1: main.cpp

Die zweite konzentriert sich auf die Paralisierung eines separaten Algorithmus. Hier wird das Bild zwischen den Prozessen geteilt und parallel verarbeitet.
Zum besseren Überblick ist der Konvolutionsalgorithmus in einer eigenen Klasse “ConvolutionEffects“ untergebracht. Diese Klasse enthält alle notwendigen Daten für den Konvolutionsalgorithmus, nämlich zu den verschiedenen Matrizen des Konvolutionskerns [3].

```
17 enum EffectType {
18     GaussianBlur3x3,
19     GaussianBlur5x5,
20     MotionBlur,
21     Emboss,
22     Sharpen
23 };
24
25 class ConvolutionEffects {
26 private:
```

```
27      uint _filterWidth, _filterHeight, _width, _height;
28      Mat _sourceImage;
29      vector<vector<int>> getFilterForEffectType(EffectType effectType);
30  public:
31      ConvolutionEffects(Mat &sourceImage);
32      Mat makeConvolutionMagic(EffectType effectType, double factor, double bias);
33  };
```

Listing 2.1: Klasse ConvolutionEffects

2.1 MPI Parallelisierung

Das Bild wird im Nullprozess gelesen. Mit der Funktion MPI_Bcast werden dann Informationen über das Bild sowie ein Verweis auf die Bilddaten auf die Prozesse verteilt. Somit erhält jeder Prozess die gleiche Menge an Daten, die von der Instanz der Klasse ConvolutionEffects weiterverarbeitet wird. Die verarbeiteten Daten werden von der Funktion MPI_Scatter in den Nullthread gesammelt [1].

```
72      MPI_Bcast(image_properties, 4, MPI_INT, 0, MPI_COMM_WORLD);
73      ...
74      // from process #0 scatter to all others:
75      MPI_Scatter(full_image.data, send_size, MPI_UNSIGNED_CHAR,
76                  part_image.data, send_size, MPI_UNSIGNED_CHAR,
77                  0, MPI_COMM_WORLD); // from process #0
78
79      executionTime = MPI_Wtime();
80
81      ConvolutionEffects convolutionEffects(part_image);
82      Mat filteredImage = convolutionEffects
83      .makeConvolutionMagic(EffectType::GaussianBlur5x5,1.0 / 256.0, 0.0);
84      part_image = filteredImage;
85
86      MPI_Gather(part_image.data, send_size, MPI_UNSIGNED_CHAR,
87                 full_image.data, send_size, MPI_UNSIGNED_CHAR,
88                 0, MPI_COMM_WORLD);
89
90      executionTime = MPI_Wtime() - executionTime;
91
92      // compute max, min, and average timing statistics
93      MPI_Reduce(&executionTime, &maxTime, 1, MPI_DOUBLE,MPI_MAX, 0,
            MPI_COMM_WORLD);
94      MPI_Reduce(&executionTime, &minTime, 1, MPI_DOUBLE, MPI_MIN,
            0,MPI_COMM_WORLD);
95      MPI_Reduce(&executionTime, &avgTime, 1, MPI_DOUBLE, MPI_SUM,
            0,MPI_COMM_WORLD);
```

Listing 2.2: MPI Parallelisierung

Die in der Bibliothek enthaltene Funktion MPI_Wtime wurde verwendet, um Informationen über die Programmlaufzeit zu sammeln. Zur Berechnung der maximalen, minimalen und durchschnittlichen Werte der Programmlaufzeit wurde die Funktion MPI_Reduce verwendet.

2.2 OpenMP Convolution Parallelisierung

Zur Parallelisierung des Konvolutionsalgorithmus wurde die Bibliothek OpenMP verwendet. Zeile 9 des Listings zeigt den für die Parallelisierung verantwortlichen Code, der im Folgenden näher betrachtet werden wird.

```
14 Mat sourceImage = _sourceImage.clone();
15 Mat filteredImage = _sourceImage.clone();
16 uint width = _width;
17 uint height = _height;
18 vector<vector<int>> filter = getFilterForEffectType(effectType);
19
20 //apply the filter and parallelize among threads
21 #pragma omp parallel for collapse(2) default(none) shared(factor, bias, wid...
22 for (int x = 0; x < width; x++) {
23     for (int y = 0; y < height; y++) {
24         double red = 0.0, green = 0.0, blue = 0.0;
25
26         //multiply every value of the filter with corresponding image pixel
27         for (int filterY = 0; filterY < _filterHeight; filterY++) {
28             for (int filterX = 0; filterX < _filterWidth; filterX++) {
29             ...
30
31             }
32         }
33         ...
34
35     }
36 }
37
38 return filteredImage;
```

Listing 2.3: OpenMP Convolution Parallelisierung

Die **pragma omp parallel for** Directive ist das Hauptelement in der Bibliothek OpenMP und ist für die parallele Ausführung des annotierten Codes verantwortlich. In diesem Punkt wird die Schleife parallelisiert, indem sie in gleiche Teile aufgeteilt wird, die in parallelen Threads ausgeführt werden.

Mit der **collapse(2)** Directive wird angegeben, wie viele Schleifen parallelisiert werden. In diesem Fall wird jeder verschachtelte Zyklus parallel ausgeführt.

Die **default(none)** Directive erfordert, dass für jede Variable, auf die im Konstrukt verwiesen wird und die kein vorgegebenes Data-Sharing-Attribut hat, das Data-Sharing-Attribut explizit festgelegt werden muss, indem sie in einer **schared()** Directive aufgeführt wird.

Die **shared()** Directive deklariert mehrere Elemente, die von Tasks gemeinsam genutzt werden sollen, die von parallelen Generierungskonstrukt(**parallel for**) erzeugt wurden.

Eine Parallelisierung der For Schleife in der Zeile 27 (Siehe Abbildung 2.3) ist zwar grundsätzlich möglich, bei unseren Testversuchen haben wir aber festgestellt, dass diese zusätzliche Parallelisierung keine nennenswerten Vorteile mit sich bringt. Die Bearbeitungszeit dauerte sogar länger, als mit nur einer Parallelisierung.

Kapitel 3

Auswertung der Ergebnisse und Fazit

Für die Messung wurden jeweils verschiedene Anzahl von MPI Prozessen und OpenMP Threads verwendet. Im Programm wurde eine Zeitmesskomponente eingebaut, welche die jeweils vergangene Zeit für einen Durchlauf pro Thread misst und am Ende des Vorganges in der Konsole ausgibt.
Die somit ermittelten Daten sind in den Tabellen 3.1 und 3.2 jeweils für Linux und Windows zusammengefasst. Ebenfalls sind die Ergebnisse der Messung in der Abbildung 3.1 und 3.2 bildlich dargestellt.
Man sieht hier den direkten Unterschied zwischen einem einzigen Prozess und der späteren Aufteilung in parallelisierte, Nebenläufige Prozesse und den direkten Zusammenhang mit der gemessenen Zeitkomponente.

Mit zunehmender Anzahl von Prozessen nimmt die Laufzeit der einzelnen Prozesse proportional ab, da jeder Prozess weniger Daten zu verarbeiten hat. Außerdem wird mit zunehmender Anzahl von Prozessen der Unterschied zwischen der Ausführungsgeschwindigkeit und der Anzahl paralleler Threads deutlicher. Aus den gemessenen Daten kann abgeleitet werden, dass die optimale Ausführungszeit durch Parallelisierung in 4 Threads erreicht wird. Sowohl eine geringere aber auch eine höhere Anzahl von Threads führt zu einer großeren Ausführungszeit. Diese Tendenz wird beobachtet, wenn die Anzahl der Prozesse steigt.

Der hier auftretende Leistungsabfall ist dadurch zu erklären, dass mit zunehmender Parallelisierung die Threads beginnen, sich gegenseitig zu blockieren. Damit lässt sich schlussfolgern, dass eine höhere Anzahl von MPI Prozessen und OpneMP Threads nicht zwangsläufig stetig zu besserer Leistung und Ausführungsgeschwindigkeit beiträgt, stattdessen muss hier für den Einzelfall und pro Einsatzzweck individuell entschieden werden, um das Optimum an Geschwindigkeit zu gewährleisten.

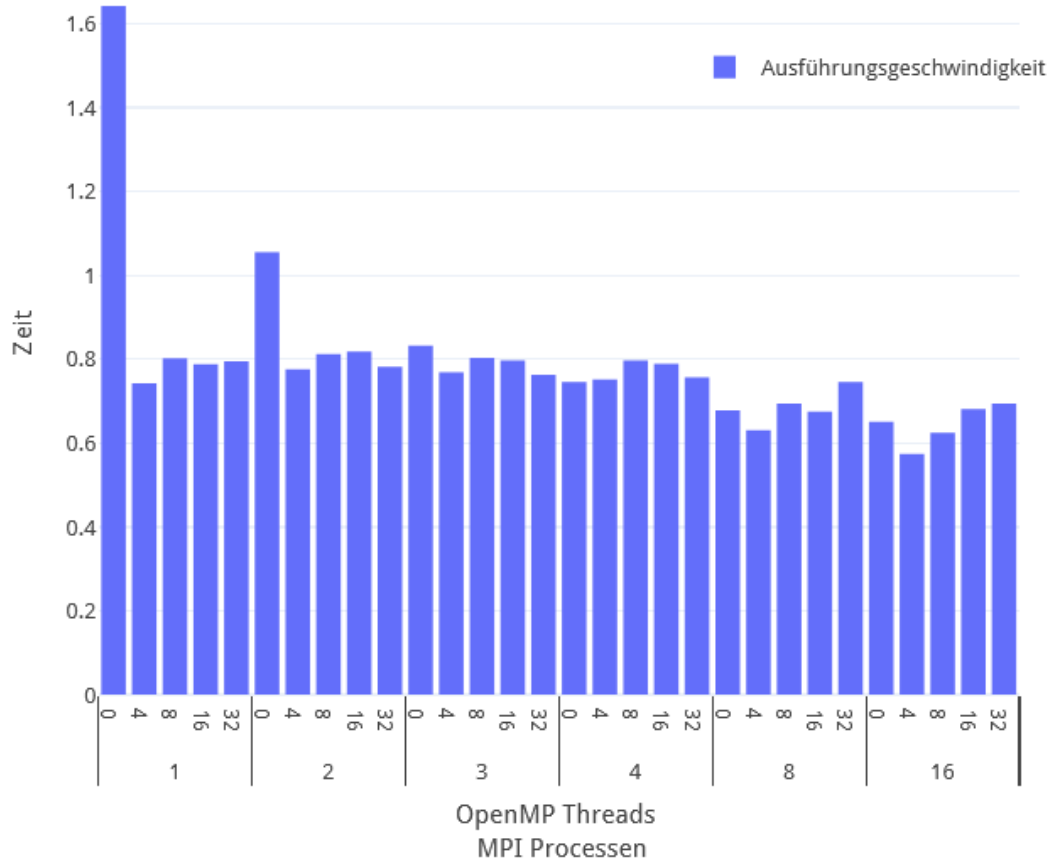

Abbildung 3.1: Zeitmessung, Linux

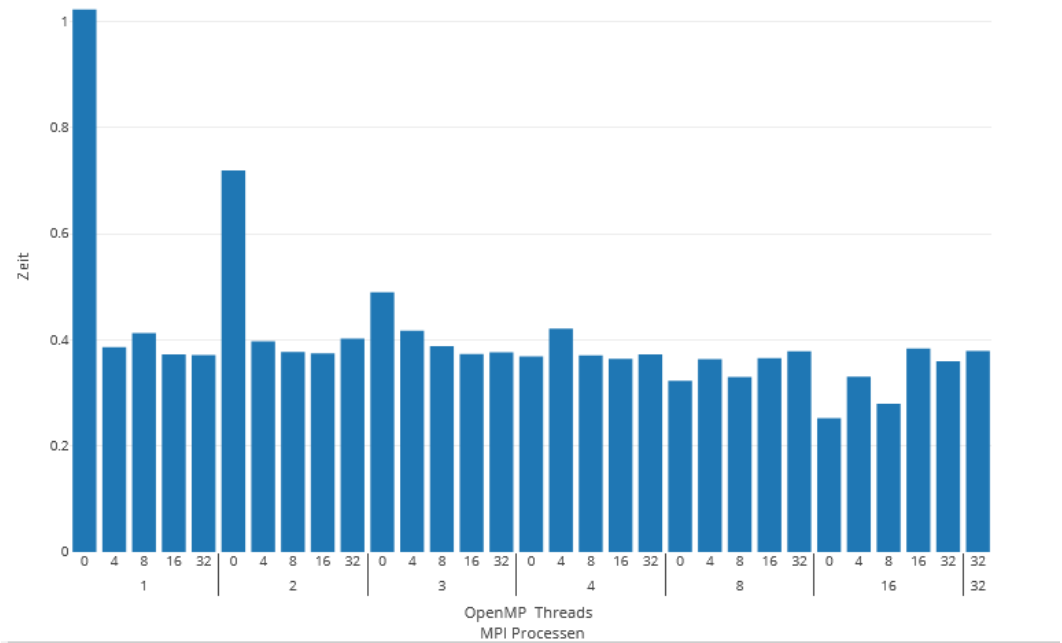

Abbildung 3.2: Zeitmessung, Windows

Tabelle 3.1: Linux: Zeitmessung der MPI / OpenMP Prozesse

MPI Process Anzahl	OpenMP Threads Anzahl	Min	Max	Durchschnitt	Total
1	0	1.642391	1.642391	1.642391	1.642391
2	0	1.055025	1.055043	1.055034	2.110068
3	0	0.832707	0.832795	0.832757	2.498270
4	0	0.742769	0.746843	0.745664	2.982654
8	0	0.604979	0.765349	0.677869	5.422953
16	0	0.418261	0.793985	0.650544	10.408699
1	4	0.743100	0.743100	0.743100	0.743100
2	4	0.776158	0.776161	0.776160	1.552320
3	4	0.768504	0.768998	0.768831	2.306492
4	4	0.746842	0.754334	0.752130	3.008521
8	4	0.408948	0.779478	0.631164	5.049314
16	4	0.373274	0.743497	0.574829	9.197269
1	8	0.801823	0.801823	0.801823	0.801823
2	8	0.812568	0.812572	0.812570	1.625140
3	8	0.803305	0.803400	0.803368	2.410103
4	8	0.790987	0.799810	0.797159	3.188636
8	8	0.552330	0.782953	0.694357	5.554853
16	8	0.538936	0.728036	0.624890	9.998243
1	16	00.787860	0.787860	0.787860	0.787860
2	16	0.818183	0.818194	0.818188	1.636377
3	16	0.796745	0.797124	0.796936	2.390809
4	16	0.769593	0.796064	0.789196	3.156784
8	16	0.549030	0.806943	0.675066	5.400527
16	16	0.476853	0.800577	0.681516	10.904250
1	32	0.794642	0.794642	0.794642	0.794642
2	32	0.781522	0.781528	0.781525	1.563049
3	32	0.763100	0.763238	0.763179	2.289537
4	32	0.649753	0.792699	0.756794	3.027177
8	32	0.610846	0.817771	0.745598	5.964780
16	32	0.389170	0.853335	0.694657	11.114507

Tabelle 3.2: Windows: Zeitmessung der MPI / OpenMP Prozesse

MPI Process Anzahl	OpenMP Threads Anzahl	Min	Max	Durchschnitt	Total
1	0	1.010920	1.010920	1.010920	1.010920
2	0	0.714817	0.714840	0.714828	1.429656
3	0	0.479880	0.484650	0.483059	1.449177
4	0	0.377326	0.379877	0.379074	1.516294
8	0	0.227588	0.424233	0.316352	2.530814
16	0	0.147624	0.364252	0.264370	4.229920
1	4	0.386240	0.386240	0.386240	0.386240
2	4	0.397363	0.397386	0.397375	0.794749
3	4	0.417084	0.417142	0.417121	1.251363
4	4	0.400246	0.429598	0.421444	1.685777
8	4	0.232094	0.443659	0.363483	2.907862
16	4	0.250586	0.436451	0.330682	5.290910
1	8	0.412871	0.412871	0.412871	0.412871
2	8	0.377320	0.377342	0.377331	0.754662
3	8	0.388192	0.388245	0.388225	1.164674
4	8	0.370139	0.370742	0.370531	1.482123
8	8	0.268889	0.382895	0.330062	2.640498
16	8	0.195612	0.370302	0.279473	4.471569
1	16	0.372808	0.372808	0.372808	0.372808
2	16	0.374498	0.374521	0.374510	0.749019
3	16	0.373176	0.373228	0.373209	1.119628
4	16	0.363574	0.364566	0.364202	1.456807
8	16	0.342112	0.378263	0.365723	2.925787
16	16	0.293087	0.410355	0.383612	6.137793
1	32	0.371297	0.371297	0.371297	0.371297
2	32	0.402355	0.402361	0.402358	0.804715
3	32	0.376774	0.376846	0.376815	1.130445
4	32	0.370423	0.373335	0.372530	1.490119
8	32	0.218352	0.435633	0.378338	3.026704
16	32	0.151750	0.442183	0.359782	5.756516
32	32	0.314925	0.434993	0.379126	12.132046

Literaturverzeichnis

[1] *Gaussian Blur Algorithm.* https://www.pixelstech.net/article/1353768112-Gaussian-Blur-Algorithm,Zugriff: 04.12.2020.

[2] *Gaussian Blur, Image Filtering.* https://lodev.org/cgtutor/filtering.html#Gaussian_Blur_,Zugriff: 04.12.2020.

[3] *OpenCV.* https://www.opencv-srf.com/p/introduction.html,Zugriff: 04.12.2020.

[4] *Original image: Lena.* https://www.cosy.sbg.ac.at/~pmeerw/Watermarking/lena.html,Zugriff: 04.12.2020.