Gruppe: Mold Florian, Aytac Karakaya

Ansatz

Wir haben eine parallele for-loop von OpenMp verwendet. Dabei haben die imageData Variable geshared, da dies ein Array ist, wo die Farbinformationen zu den Pixeln gespeichert werden. Die Datenstruktur kann geshared werden, da jeder Thread auf andere Indizes zugreift und somit keine Probleme enstehen. Zusätzlich sharen wir die Breite und die Höhe des Bildes, da diese Variablen nur gelesen werden. Per default sharen wir keine Variablen. Collapse haben wir auf 2 gesetzt, da es zwei for-loops ineinander sind. Diese for-loops können ohne Probleme collapsed werden, da die innere Iteration nichts von der Äußeren abhängt. Daher werden h*w Tasks erzeugt, die von den Threads bearbeitet werden können. Dadurch ist der Workload pro Thread besser aufgeteilt. Die Tests wurden mit 8 Kernen gemacht.

Das Schedule haben wir auf **static** gesetzt, da hier die Blöcke die bearbeitet werden müssen gleich auf alles Threads aufgeteilt werden.

```
#pragma omp parallel for shared(imageData, w, h) collapse(2) default(none) schedule(static)
    for (int px = 0; px < w; px++) {
        for (int py = 0; py < h; py++) {
            calcPix(px, py, &: imageData);
        }
    }
```

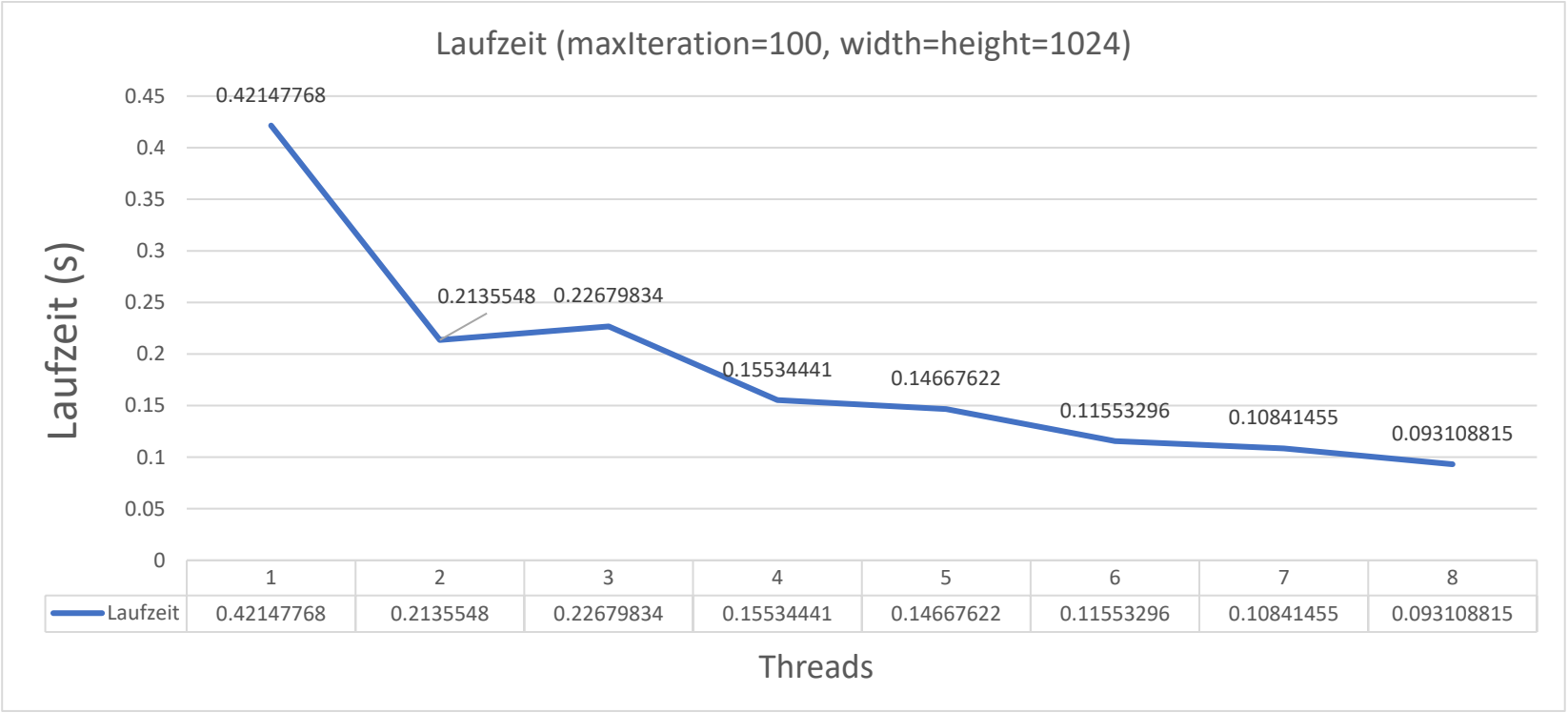
Tests

Schedule: static

maxIterations = 100

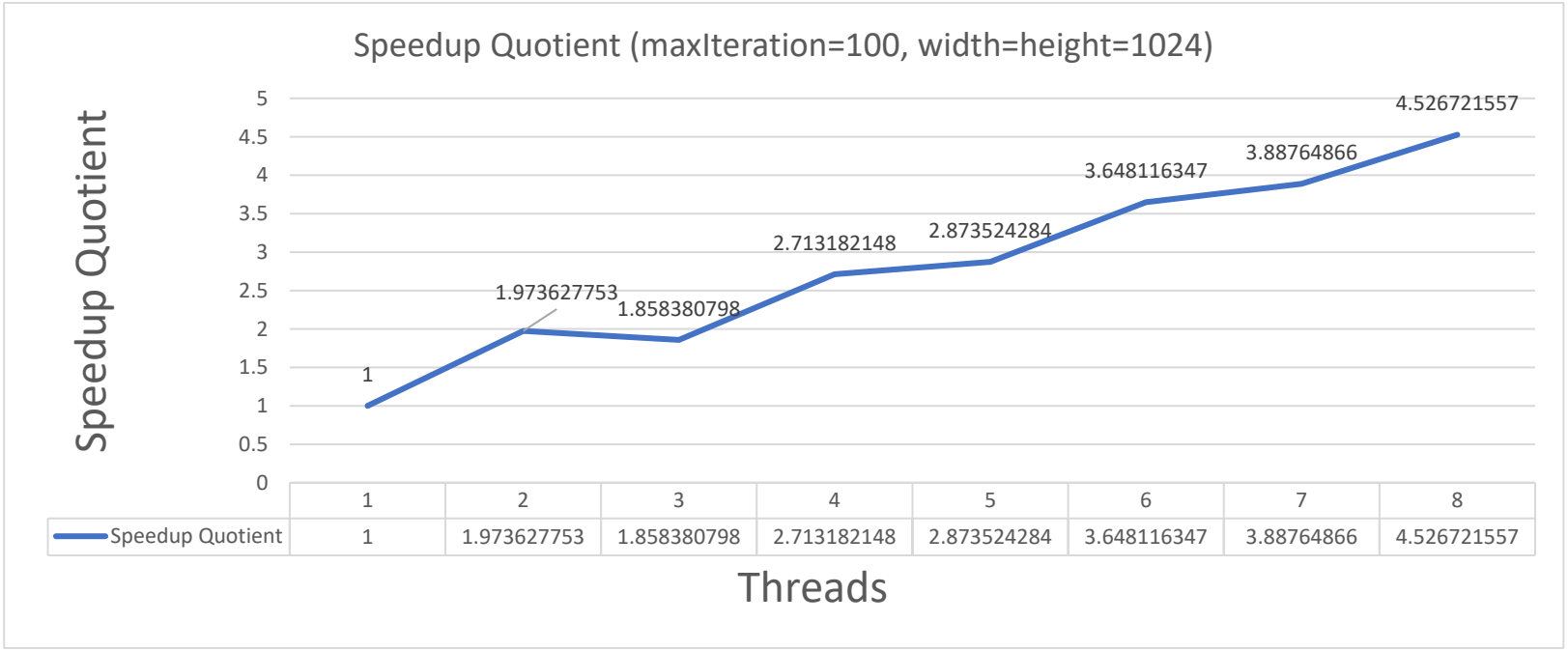
Dieser Versuch wurde mit einer maxIteration von 100 und einer Breite und Höhe von 1024px ausgeführt. Dabei wurde das Programm mit 1 bis 8 Threads ausgeführt. Jede Ausführung mit einer bestimmten Threadanzahl wurde 100x mal durchgeführt und daraus der Mittelwert der Ausführungsdauer berechnet. Wie man im Graph unten sieht, ist die Ausführungsdauer zwischen einem Thread und zwei Threads circa die Hälfte. Bei drei Threads lässt sich etwas interessantes beobachten, da die Laufzeit wieder steigt. Bei 4 Threads sinkt die Laufzeit wieder deutlich. Danach sinkt die Laufzeit leicht kontinuierlich ab.

Gruppe: Mold Florian, Aytac Karakaya



In der nächsten Grafik sieht man den Speedup des Programms. Hier wurde die sequenzielle Laufzeit durch parallele Laufzeit für jeden Thread dividiert. Der Speedup bei zwei Threads ist fast 2, daher wurde die Laufzeit beinahe halbiert. 3 Threads sind deutlich langsamer als 2 Threads. Bei 4 Threads liegt der Speedup bei 2.71, was schon nicht mehr den Optimalen 4 entspricht. Bei 6 Threads steigt der Speedup wieder deutlich auf 3,64 und bei 8 Threads liegt der Speedup bei 4.52, was nur etwas besser ist als eine 4-fach bessere Laufzeit. Der Speedup steigt nicht linear mit der Anzahl der Threads und sinkt immer weiter ab bei weiteren Threads.

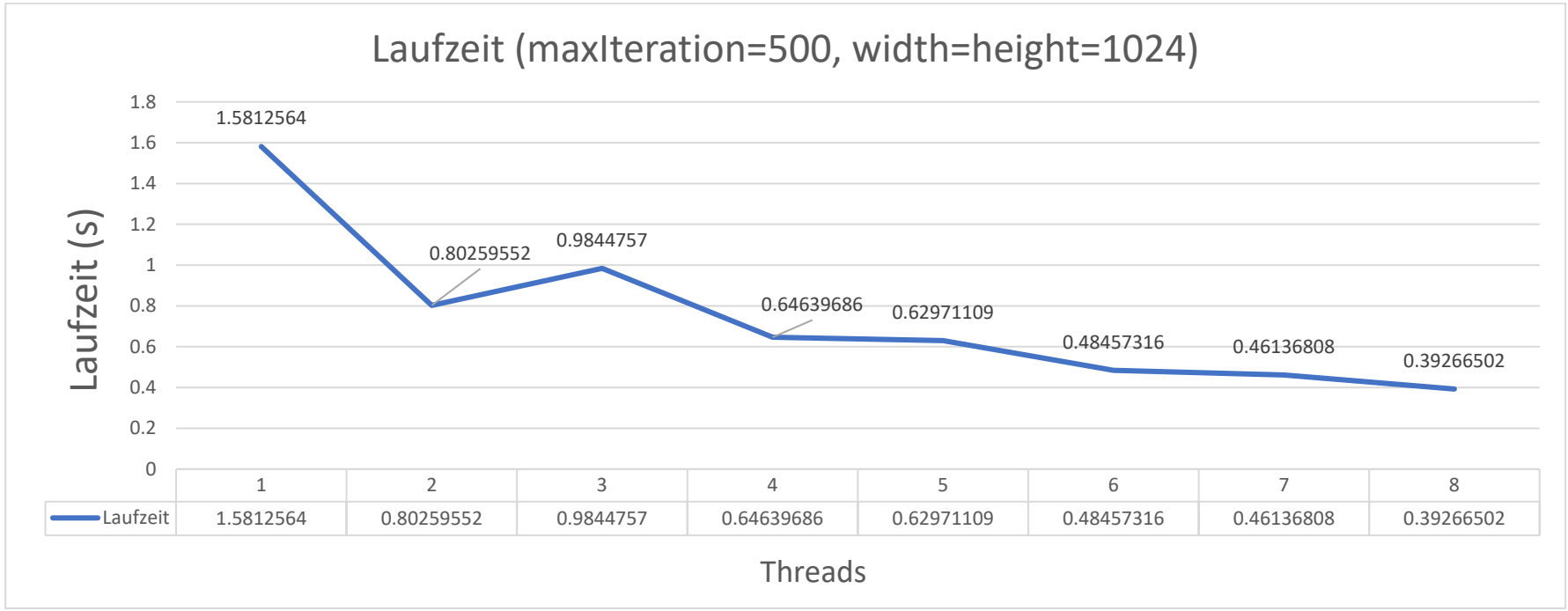
Gruppe: Mold Florian, Aytac Karakaya



Gruppe: Mold Florian, Aytac Karakaya

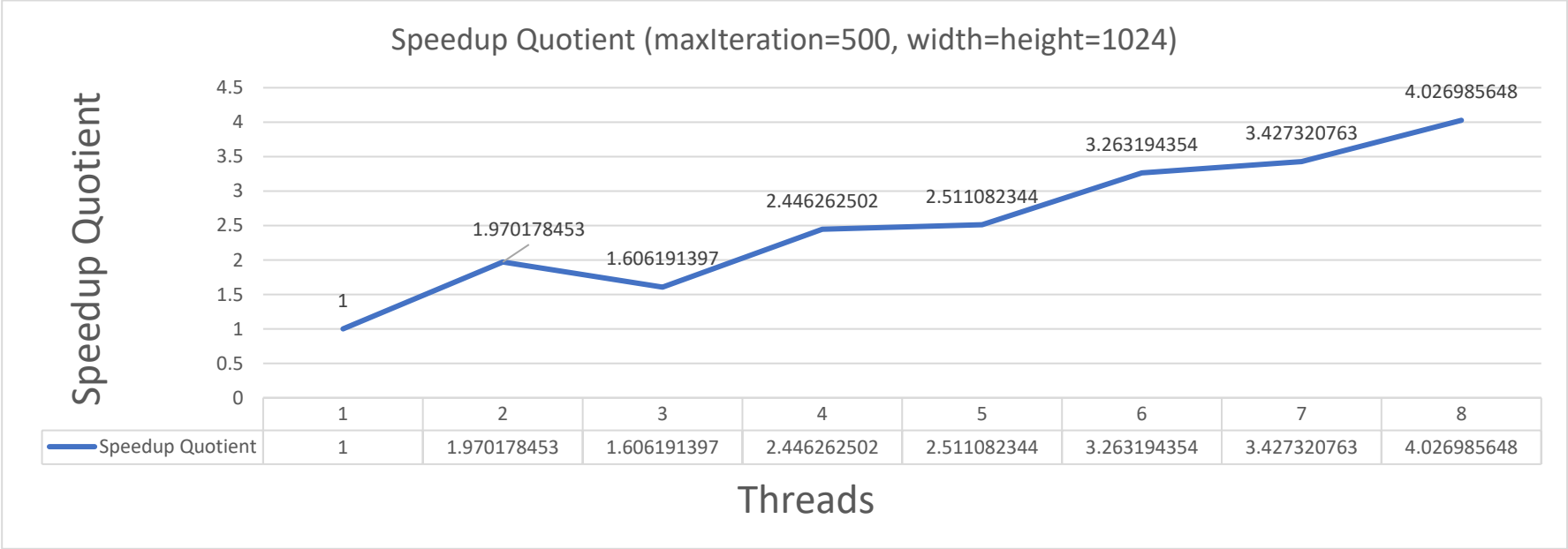
maxIterations = 500

Dieser Versuch wurde mit einer maxIteration von 500 und einer Breite und Höhe von 1024px ausgeführt. Dabei wurde das Programm mit 1 bis 8 Threads ausgeführt. Jede Ausführung mit einer bestimmten Threadanzahl wurde 100x mal durchgeführt und daraus der Mittelwert der Ausführungsdauer berechnet. Der Graph sieht fast genauso aus, wie bei 100 Iterationen, da die Laufzeit kontinuierlich sinkt, außer bei drei Threads. Wie man im Graph unten sieht, ist die Ausführungsdauer zwischen einem Thread und zwei Threads circa die Hälfte. Bei drei Threads lässt sich etwas interessantes beobachten, da die Laufzeit wieder steigt. Bei 4 Threads sinkt die Laufzeit wieder deutlich. Danach sinkt die Laufzeit leicht kontinuierlich ab.



Gruppe: Mold Florian, Aytac Karakaya

In der nächsten Grafik sieht man den Speedup des Programms. Hier wurde die sequenzielle Laufzeit durch parallele Laufzeit für jeden Thread dividiert. Der Speedup bei zwei Threads ist fast 2, daher wurde die Laufzeit beinahe halbiert. 3 Threads sind deutlich langsamer als 2 Threads. Bei 3 Threads liegt der Speedup bei 1.60, was deutlich weniger ist als bei 100 Iterationen und weit unter dem Optimum. Bei 6 Threads steigt der Speedup wieder deutlich auf 3,26 und bei 8 Threads liegt der Speedup bei 4. Die Speedup Werte liegen unter den Tests mit 100 maxIterations und auch weit unter dem Optimum. Der Speedup steigt nicht linear mit der Anzahl der Threads und sinkt immer weiter ab bei weiteren Threads.

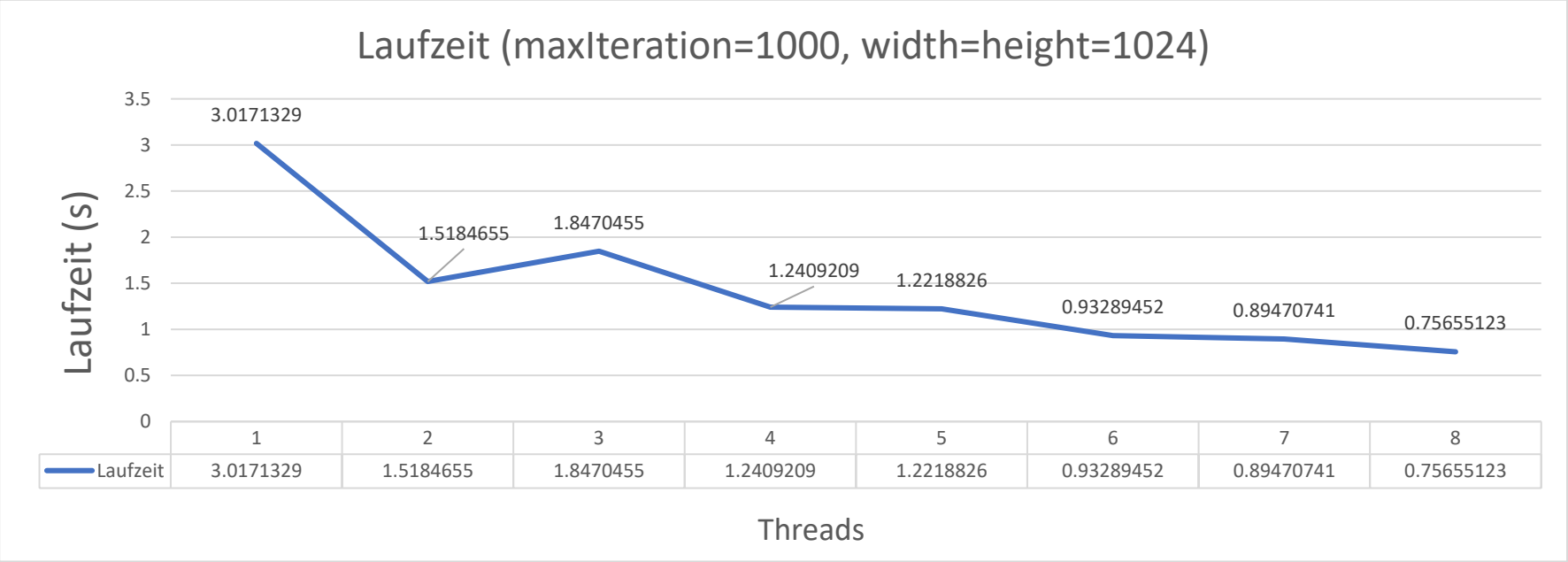


	1	2	3	4	5	6	7	8
Speedup Quotient	1	1.970178453	1.606191397	2.446262502	2.511082344	3.263194354	3.427320763	4.026985648

Gruppe: Mold Florian, Aytac Karakaya

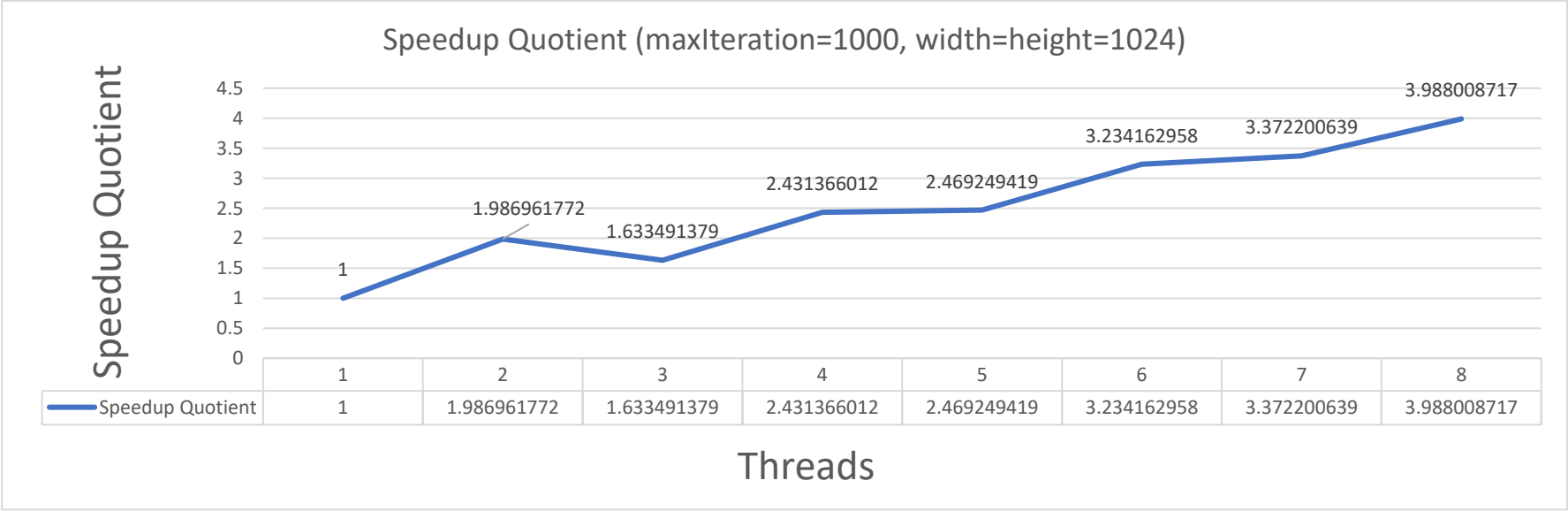
maxIterations=1000

Dieser Versuch wurde mit einer maxIteration von 1000 und einer Breite und Höhe von 1024px ausgeführt. Dabei wurde das Programm mit 1 bis 8 Threads ausgeführt. Jede Ausführung mit einer bestimmten Threadanzahl wurde 100x mal durchgeführt und daraus der Mittelwert der Ausführungsdauer berechnet. Der Graph sieht fast genauso aus, wie bei 500 Iterationen, da die Laufzeit kontinuierlich sinkt, außer bei drei Threads. Wie man im Graph unten sieht, ist die Ausführungsdauer zwischen einem Thread und zwei Threads circa die Hälfte. Bei drei Threads lässt sich etwas interessantes beobachten, da die Laufzeit wieder steigt. Bei 4 Threads sinkt die Laufzeit wieder deutlich. Danach sinkt die Laufzeit leicht kontinuierlich ab.



In der nächsten Grafik sieht man den Speedup des Programms. Hier wurde die sequenzielle Laufzeit durch parallele Laufzeit für jeden Thread dividiert. Der Speedup bei zwei Threads ist fast 2, daher wurde die Laufzeit beinahe halbiert, was optimal ist. 3 Threads sind deutlich langsamer als 2 Threads. Bei 3 Threads liegt der Speedup bei 1.63, was deutlich weniger ist als bei 100 Iterationen, aber ähnlich zu den 500 maxIterations und weit unter dem Optimum. Bei 6 Threads steigt der Speedup wieder deutlich auf 3,23 und bei 8 Threads liegt der Speedup bei 3.9. Die Werte sind ähnlich zu den Werten bei 500 maxIterations Die Speedup Werte liegen unter den Tests mit 100 maxIterations und auch weit unter dem Optimum. Der Speedup steigt nicht linear mit der Anzahl der Threads und sinkt immer weiter ab bei weiteren Threads.

Gruppe: Mold Florian, Aytac Karakaya

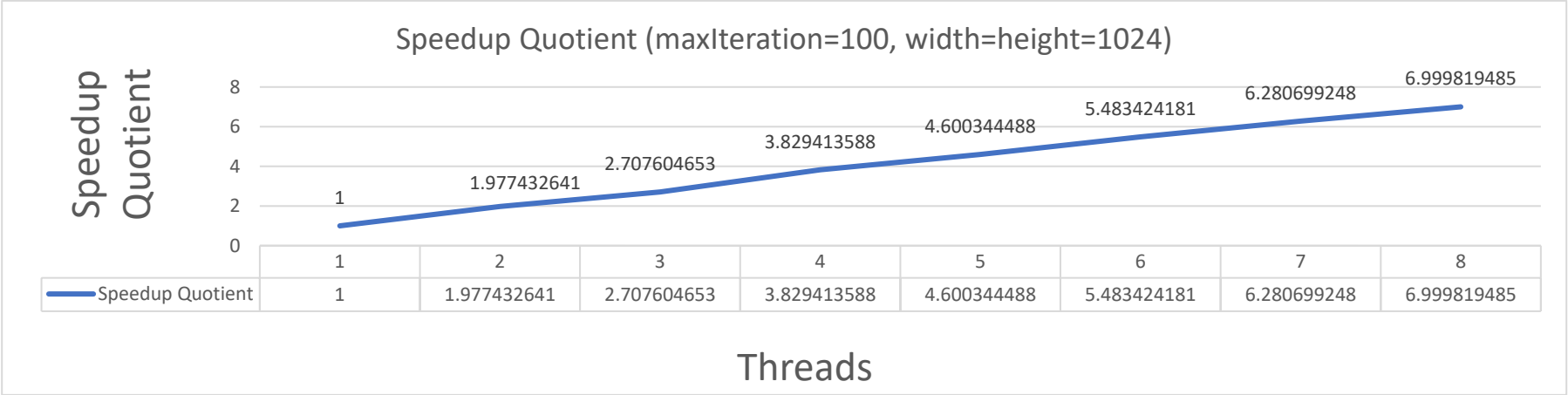


Genau die gleichen Tests wurden mit **dynamic** und **guided** durchgeführt. Die Ergebnisse haben teilweise einen besseren Speedup als bei **static**. Die genauen Ergebnisse können in den Dateien speedup-analyse.xlsx, speedup-analyse-dynamic.xlsx und speedup-analyse-guided.xlsx nachgelesen werden.

Gruppe: Mold Florian, Aytac Karakaya

Beispiel: schedule(guided)

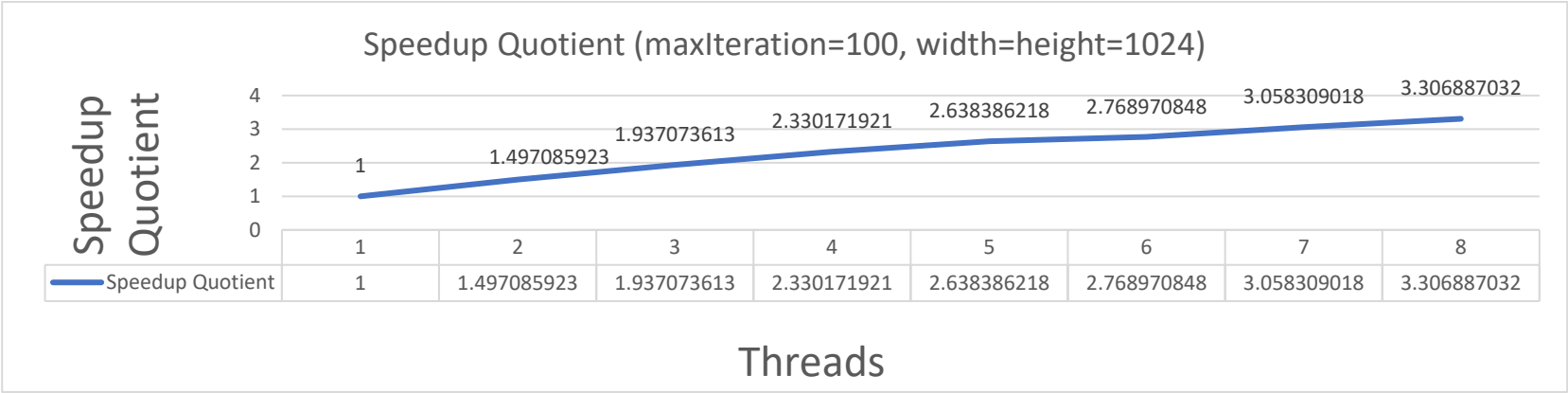
Ein 7x Speedup bei 8 Threads im Vergleich zu 4.5 bei **static**.



	1	2	3	4	5	6	7	8
Speedup Quotient	1	1.977432641	2.707604653	3.829413588	4.600344488	5.483424181	6.280699248	6.999819485

Beispiel schedule(dynamic)

Ein 3.5x Speedup bei 8 Threads im Vergleich zu 4.5 bei **static**.



	1	2	3	4	5	6	7	8
Speedup Quotient	1	1.497085923	1.937073613	2.330171921	2.638386218	2.768970848	3.058309018	3.306887032

Gruppe: Mold Florian, Aytac Karakaya