Um das Programm zu Parallelisieren haben wir innerhalb der calculate-Funktion diese beiden for-Schleifen angepasst:

```
/* over all rows */
for (i = 1; i < N; i++)
     {
          double fpisin_i = 0.0;

          if (options->inf_func == FUNC_FPISIN)
          {
               fpisin_i = fpisin * sin(pih * (double)i);
          }

          /* over all columns */
          for (j = 1; j < N; j++)
          {
               …
          }
     }
```

Wir haben drei verschiedene Arten der Parallelisierung ausprobiert. Die erste von denen Zeilenweise, wofür wir die erste for-Schleife mithilfe von openMP parallel for angepasst haben:

```
#pragma omp parallel for private(i,j,star, residuum) num_threads(options->number)
/* over all rows */
for (i = 1; i < N; i++)
{
     double fpisin_i = 0.0;

     if (options->inf_func == FUNC_FPISIN)
     {
          fpisin_i = fpisin * sin(pih * (double)i);
     }

     /* over all columns */
     for (j = 1; j < N; j++)
     {
          …
     }
}
```

Die Zweite Parallelisierung verlief Spaltenweise, weshalb wir nur die Indices vertauschen und die Sinusrechnung verschieben mussten.

```
#pragma omp parallel for private(j,i,star, residuum) num_threads(options->number)
/* over all columns */
for (j = 1; j < N; j++)
{
      /* over all rows */
      for (i = 1; i < N; i++)
      {
                  double fpisin_i = 0.0;

                  if (options->inf_func == FUNC_FPISIN)
                  {
                              fpisin_i = fpisin * sin(pih * (double)i);
                  }
                  …
      }
}
```

Die Dritte Parallelisierung verlief Elementweise dazu mussten wir die Aufteilung neu berechnen:

```
#pragma omp parallel for private(k,i,j,star, residuum) num_threads(options->number)
/* over all columns */
for (int k = 0; k < max_k; k++)
{
      i = k / (N-1) + 1;
      j = k % (N-1) + 1;

      double fpisin_i = 0.0;
      …
}
```

Desweiten wurden folgende flags benutzt:

CFLAGS = -std=c99 -pedantic -Wall -Wextra -Ofast -flto -march=native -fopenmp

Die jeweils verschiedenen Programme wurden gegeneinander gemessen, wofür folgende Parameter benutzt wurden:

Threads: 12
Methode Nr: 2
Interlines: 512
Funktion Nr: 2
Termination Nr: 2
Iterationen: 10000

Die Ergebnisse haben wir in folgender Tabelle angegeben, die Zahlen sind die vergangene Zeit in Sekunden.

Methode	Messung 1	Messung 2	Messung 3	Mittel
Sequenz	5474.64	5453.01	5485.38	5471.01
Element	1147.67	1158.77	1162.78	1156.41
Spalten	1123.75	1115.41	1127.19	1122.11
Zeilen	478.29	477.53	443.13	466.31

Alle Messungen aller Methoden lieferten das selbe Ergebnis zurück. Man kann deutlich erkennen, dass die zeilenweise Parallelisierung am schnellsten läuft. Bei diesen Parametern ist sie gut 11,7 mal schneller als die sequentielle Methode. Auch ist sie mehr als Doppelt so schnell wie die anderen parallelen Methoden, was sich dadurch erklären lässt, dass der cache zeilenweise besser genutzt wird, wie wir letzte Woche bereits festgestellt haben. Da es sich bei der zeilenweisen Methode um die Schnellste handelt haben wir diese für die folgende Leistungsanalyse benutzt

Messung 1

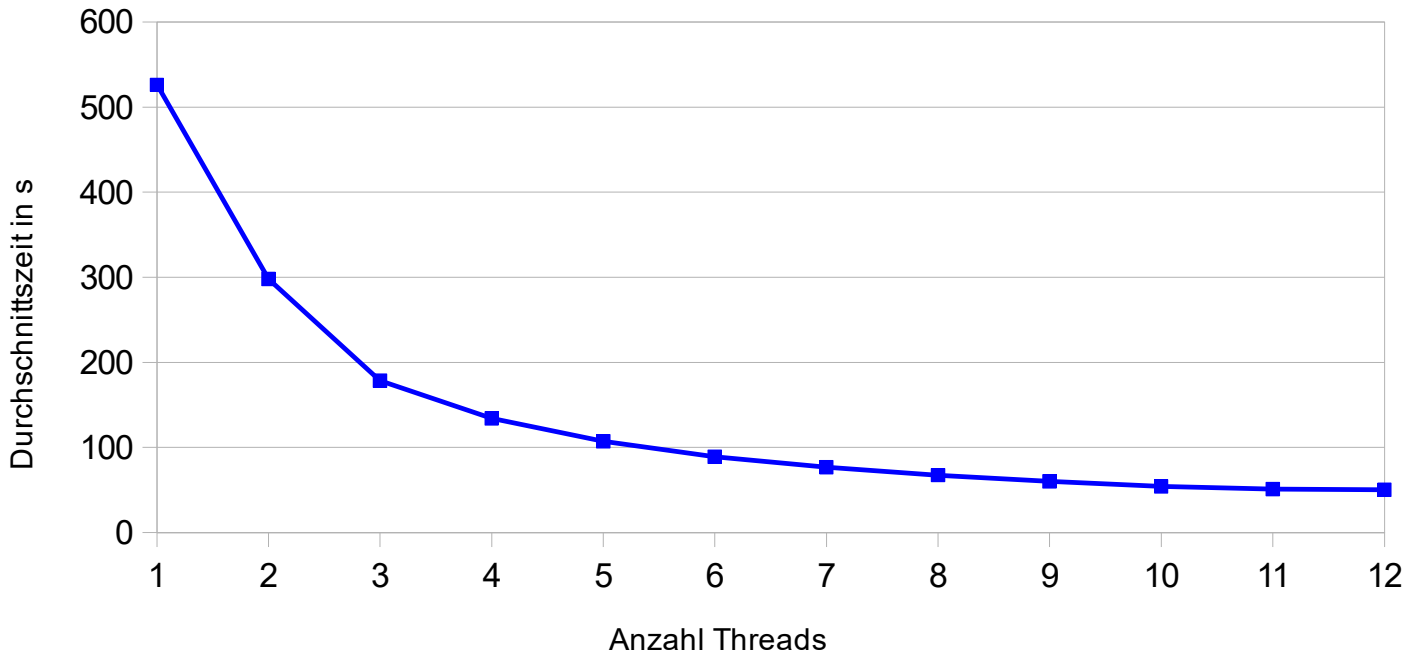
In der ersten Messung wurden verschiedene Anzahlen von Threads getestet. Folgende Parameter wurden benutzt:

Threads: 1-12
Methode Nr: 2
Interlines: 512
Funktion Nr: 2
Termination Nr: 2
Iterationen: 1000

Es ergaben sich folgende Messwerte:

Threads	Zeit 1	Zeit 2	Zeit 3	Mittelwert
1	526,25	526,36	525,95	526,18
2	264,54	264,66	364,58	297,93
3	178,45	178,66	178,01	178,38
4	133,44	135,17	133,51	134,04
5	107,28	107,11	107,35	107,25
6	89,11	88,96	89,38	89,15
7	76,95	76,77	76,61	76,78
8	67,40	67,37	67,47	67,41
9	60,24	60,25	60,02	60,17
10	54,43	54,03	54,01	54,16
11	52,28	49,95	51,34	51,19
12	52,48	48,09	50,83	50,47

Die Mittelwerte haben wir, der Anschaulichkeit halber in eine Grafik eingefügt:

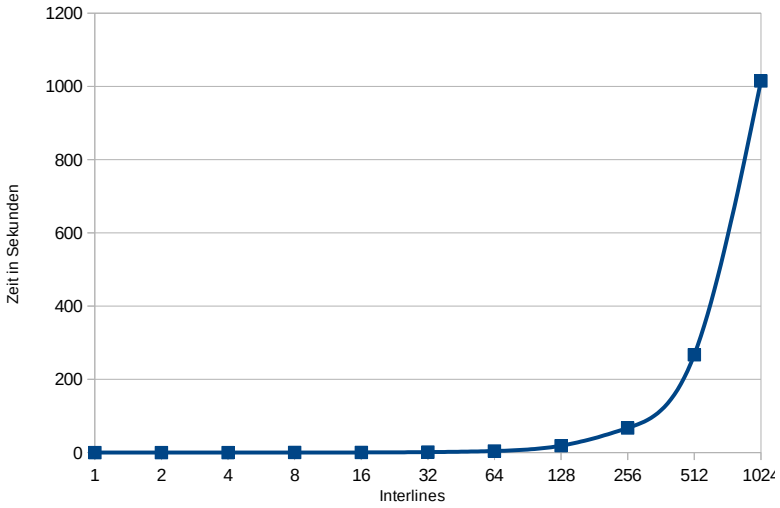


Messung 2

Die zweiten Messung wurde zwei mal durchgeführt, in ihr wurde die Anzahl der Interlines variiert. Folgende Parameter wurden benutzt:

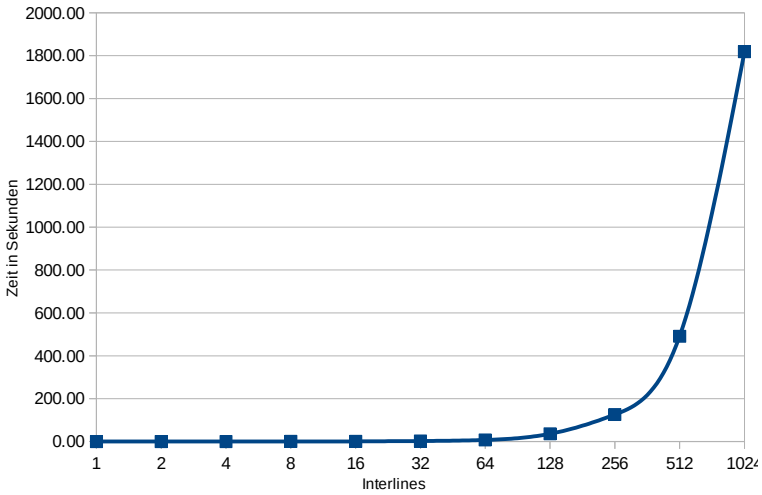
Threads: 12
Methode Nr: 2
Interlines: 1-1024
Funktion Nr: 2
Termination Nr: 2
Iterationen: 5700

Interlines	Messung1	Messung2	Messung3	Mittel
1	0.03	0.04	0.03	0.03
2	0.04	0.04	0.04	0.04
4	0.06	0.06	0.06	0.06
8	0.14	0.14	0.14	0.14
16	0.42	0.42	0.37	0.40
32	1.10	1.25	1.27	1.21
64	4.39	4.46	4.07	4.31
128	18.45	19.82	17.93	18.73
256	71.67	63.73	66.37	67.26
512	293.68	253.60	254.35	267.21
1024	1019.46	1007.20	1018.32	1014.99



Threads: 12
Methode Nr: 2
Interlines: 1-1024
Funktion Nr: 2
Termination Nr: 2
Iterationen: 10000

Interlines	Messung1	Messung2	Messung3	Mittel
1	0.08	0.08	0.08	0.08
2	0.09	0.11	0.09	0.09
4	0.17	0.13	0.11	0.14
8	0.26	0.29	0.23	0.26
16	0.82	0.72	0.74	0.76
32	2.20	2.00	2.84	2.35
64	7.36	7.33	7.31	7.34
128	37.98	39.23	30.36	35.86
256	125.66	126.11	125.93	125.90
512	479.33	506.76	487.18	491.09
1024	1822.65	1813.17	1820.67	1818.83



Trotz der Unterschiedlichen Parameter, sehen die beiden Graphen kongruent aus. Es lässt sich außerdem erkennen, dass das Wachstum der Rechenzeit nicht linear mit der Anzahl der Interlines zusammenhängt. Besonders in der linken Messung erkennt man an den letzten 5 Einträgen der gemittelten Werte die Quadratische abhängigkeit. Die Anzahl der Iterationen wurde hier bewusst so gewählt das für 1024 Interlines ungefähr 1024 Sekunden Rechenzeit benötigt werden, sodass man die Zeiten mit den bekannten 2er Potenzen vergleichen kann. An dem Sprung von 1024 zu 512 erkennt man nun einfach, dass die Rechenzeit sich nicht halbiert sondern ungefähr geviertelt hat, da die Zeit nicht auf etwa 512 sondern etwa 256 gesprungen ist. Auch der Sprung zu 256 ist ungefähr ein viertel, statt ca 256 auf ca 64. Diese Verhalten ist zu erwarten, da die Interlines die Größe einer Zweidimensionalen Matrix angeben. Es wäre erstaunlich wenn die Rechenzeit also nicht mindestens Quadratisch ansteigt.