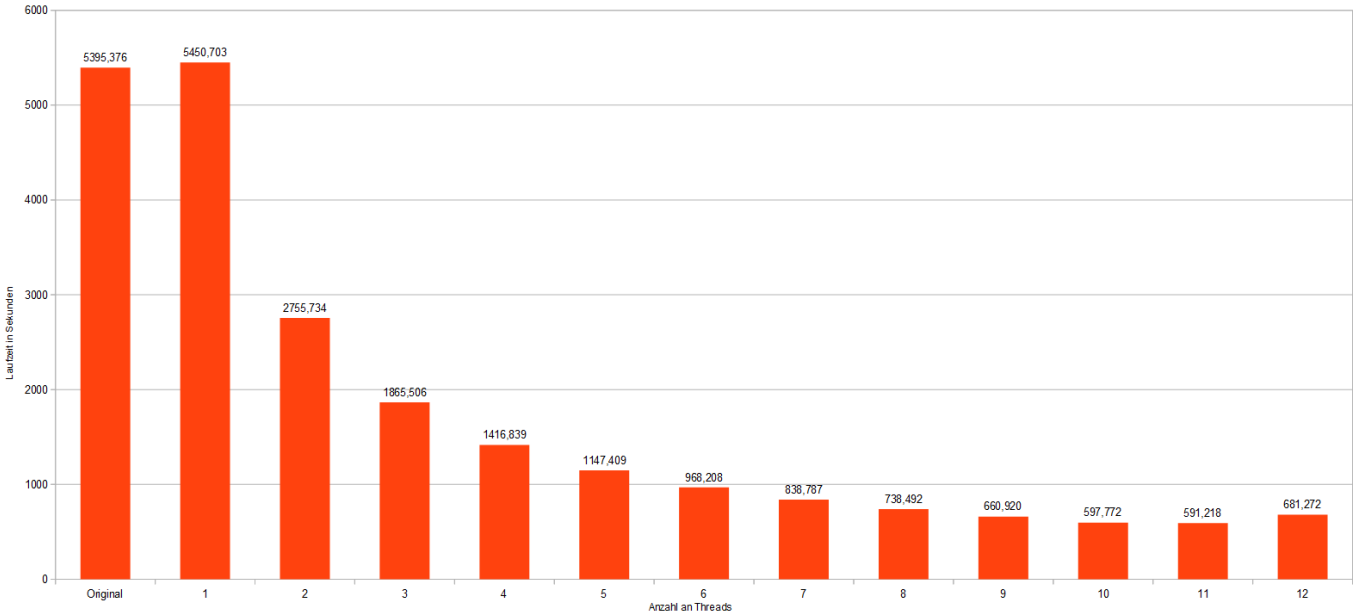
Vorwort

Für die Parallelisierung des Programms haben wir ca. 4h gebraucht, sind allerdings nur auf einen Speedup Faktor von ~ 7 gekommen. Die nächsten 10h haben wir damit verbracht eine bessere Lastverteilung zu erreichen, allerdings ohne Erfolg. Letzendlich haben wir eine Verbesserung gefunden, indem wir die Variablen, die wir dem Thread übergeben, nicht extra lokal abspeichern, sondern über einen Pointer dereferenzieren. Des Weiteren haben wir die Laufzeit optimiert, indem wir den Lock erst nach den verschachtelten for-loops anfordern und die boolschen Ausdrücke ausgewertet als Parameter übergeben.
Als Parameter für die Messung haben wir 12 Threads, Jacobi-Verfahren, 512 Interlines, die zweite Störfunktion und Abbruch nach 10240 Iterationen gewählt.
Alle Messungen haben auf dem Knoten west1 stattgefunden.

Messungen auf West1

Threads	Messung 1	Messung 2	Messung 3	Durchschnitt
Original	5392,137	5396,863	5397,127	5395,375
1	5460,855	5441,503	5449,752	5450,703
2	2760,255	2753,238	2753,71	2755,734
3	1863,192	1869,164	1864,163	1865,506
4	1415,983	1411,212	1423,323	1416,839
5	1147,374	1147,481	1147,373	1147,409
6	969,689	966,488	968,448	968,208
7	839,295	838,31	838,756	838,787
8	738,144	738,426	738,906	738,492
9	659,767	663,97	659,023	660,92
10	597,639	597,517	598,161	597,772
11	592	589,386	592,267	591,217
12	681,427	681,207	681,182	681,272



Interpretation

Wie auch bei der Parallelisierung mit openmp haben wir eine höhere Laufzeit für die Verarbeitung mit einem Thread als das nicht-parallelisierte Original. Dies liegt an dem höheren Verwaltungsaufwand, also dem Erstellen des Threads und der Zuweisung der Ressourcen, für jeden Durchlauf der while-Schleife. Ein weiterer möglicher Faktor wäre eine Wechselwirkung von der -O3 Flag und der -pthread Flag. Für die Leistungsanalyse der vorherigen Woche haben wir die Möglichkeit einer Wechselwirkung von -O3 und -fopenmp in Betracht gezogen (Quelle: https://stackoverflow.com/a/22012670), möglicherweise gibt diese Wechselwirkung auch hier.
Des weiteren verhält sich die Leistungssteigerung nahezu linear: Die Berechnung mit 2 Threads braucht halb so lange wie die mit einem Thread, 10 Threads brauchen ungefähr halb so lange wie 5 Threads und 11 Threads bringen eine Beschleunigung von ca $\frac{10}{11}$ gegenüber einer Ausführung mit 10 Threads.
Einzig die Laufzeit mit 12 Threads bildet eine Ausnahme, da sich die Laufzeit hier wieder erhöht. Dies liegt an der Verteilung der Threads auf die Prozessoren.
Beim Start des Programmes gibt es einen Master-Thread, der das Programm sequentiell ausführt. Dieser erzeugt dann <Anzahl an Threads> WEITERE Sklaven-Threads in der dafür vorgesehenen for-loop. Da wir 12 Kerne auf dem Knoten west1 haben, können die 12 Sklaven- plus 1 Master- Thread nicht mehr gleichmäßig aufgeteilt werden, sodass sich mindestens 2 Threads einen Kern teilen müssen.
Hierdurch erhalten wir auch die langsamere Ausführungszeit.