1 Umsetzung der Datenaufteilung

Messung 1)

Für die erste Messung der Datenaufteilung wurden folgende Optionen benutzt:
partdiff-openmp-{element, spalten, zeilen} 1 2 128 1 2 1024
benutzt, und einmal
partdiff-openmp-{element, spalten, zeilen} 12 2 128 1 2 1024

Element 1 Core
Time:
real: $0m39.38s$
user: $0m39.19s$
sys : $0m0.07s$

Spalten 1 Core
Time:
real: $0m29.13s$
user: $0m29.02s$
sys : $0m0.01s$

Zeilen 1 Core
Time:
real: $0m6.86s$
user: $0m6.82s$
sys : $0m0.02s$

Element 12 Core
Time:
real: $0m3.72s$
user: $0m44.30s$
sys : $0m0.02s$

Spalten 12 Core
Time:
real: $0m4.18s$
user: $0m49.83s$
sys : $0m0.01s$

Zeilen 12 Core
Time:
real: $0m0.84s$
user: $0m9.91s$
sys : $0m0.04s$

Hierbei lässt sich feststellen, dass der Zeilen basierte Algorithmus am schnellsten ist. Dies liegt wahrscheinlich an der Kontinuität der Daten, da diese besser in den Cache geladen werden können so bekommt jeder Thread seine eigenen Teile der Matrix, welche unabhängig zu den der anderen Threads sind. Des Weiteren wird die Anzahl der Kontextwechsel der Threads viel geringer sein, all die des Element basierten Algorithmus. Zusätzlich müssen bei Zeilen bzw. Spalten basierten Algorithmus nur die halbe Matrix berechnet werden, da dort die Mathematik dies zulässt.
Die Geschwindigkeit des Element basierenden Algorithmus lässt sich am wahrscheinlichsten damit erklären, dass bei den Berechnungen mit einem Core die ganze N $\times$ N Matrix berechnet werden muss, da hier nicht Mathematisch optimiert werden kann. Die schnellere Geschwindigkeit mit 12 Cores ergibt sich dann durch dass bruteforce artige berechnen der Elemente, die zusätzlich besser durch den Cache adressiert werden als die des Spalten basierten Algorithmus.
Trotzdem müssen bei dem Element basierten Algorithmus N $\cdot$ N Kontextwechsel durchgeführt werden.
Dies wird aber nicht durch die Laufzeit Analyse bestätigt, da die Zeit innerhalb des Betriebssystem nicht merklich von den der anderen Algorithmen abweicht, daher vermuten wir, dass

OpenMp intern die Threads recycelt.
Zu guter Letzt ist der Spalten basierte Algorithmus so extrem viel langsamer, da er nicht den Cache effektiv nutzen kann und ständig die werde aus dem Hauptspeicher in den Cache nachgeladen werden müssen.

Vergleich der OpenMp Scheduling Algorithmen

Alle Test wurden dieses mal mit 12 Cores gemacht, hierbei wurden die Programme wie folgt aufgerufen.

partdiff-openmp-{element, zeilen} 12 1 512 1 2 1024

Nach der Messung ergaben sich folgende Zeiten:

Schedule(static, 1)

Element 12 Core
Time:
real: $2m46.66s$
user: $18m16.66s$
sys : $0m2.45s$

Zeilen 12 Core
Time:
real: $1m43.66s$
user: $2m43.20s$
sys : $0m0.98s$

Schedule(static, 2)

Element 12 Core
Time:
real: $2m22.44s$
user: $24m45.58s$
sys : $0m3.54s$

Zeilen 12 Core
Time:
real: $0m11.01s$
user: $2m43.20s$
sys : $0m0.66s$

Schedule(static, 4)

Element 12 Core
Time:
real: $1m22.59s$
user: $15m20.04s$
sys : $0m3.54s$

Zeilen 12 Core
Time:
real: $0m9.94s$
user: $1m53.11s$
sys : $0m0.52s$

Schedule(static, 8)

Element 12 Core

Time:

real: $0m44.81s$
user: $8m47.95s$
sys : $0m5.63s$

Zeilen 12 Core

Time:

real: $0m10.23s$
user: $1m47.70s$
sys : $0m0.65s$

Schedule(dynamic, 1)

Element 12 Core

Time:

real: $18m40.31s$
user: $218m32.07s$
sys : $4m32.28s$

Zeilen 12 Core

Time:

real: $0m10.24s$
user: $1m58.43s$
sys : $0m1.47s$

Schedule(dynamic, 4)

Element 12 Core

Time:

real: $11m33.46s$
user: $136m44.00s$
sys : $2m1.22s$

Zeilen 12 Core

Time:

real: $0m9.36s$
user: $1m48.43s$
sys : $0m1.07s$

Schedule(guided)

Element 12 Core

Time:

real: $0m28.48s$
user: $4m37.25s$
sys : $2m1.21s$

Zeilen 12 Core

Time:

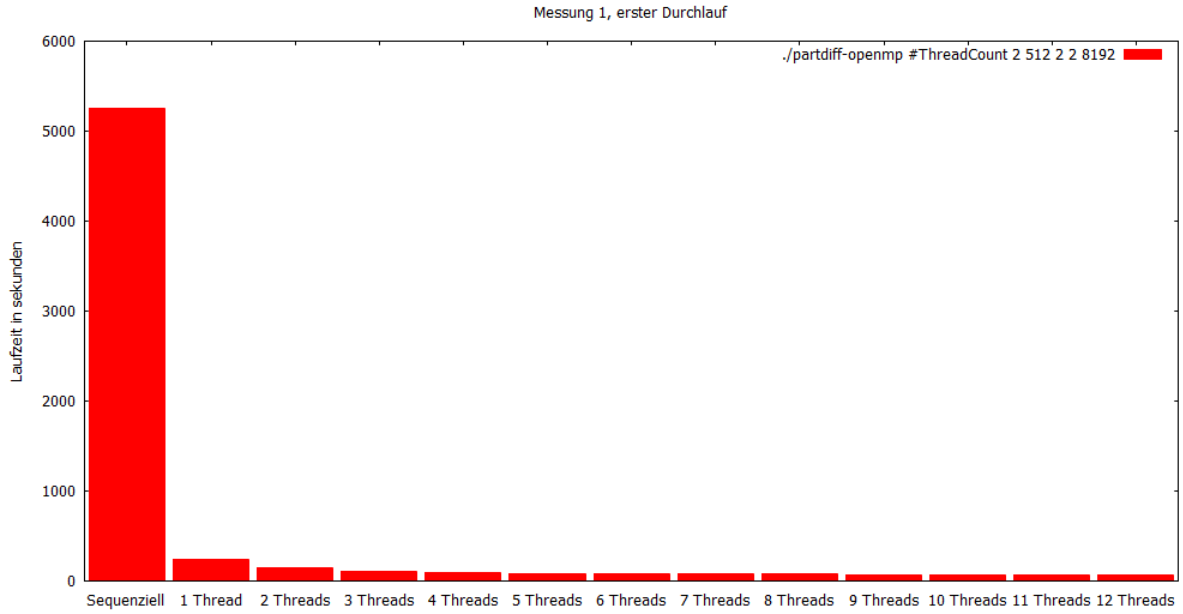
real: $0m12.05s$
user: $1m49.23s$
sys : $0m0.83s$

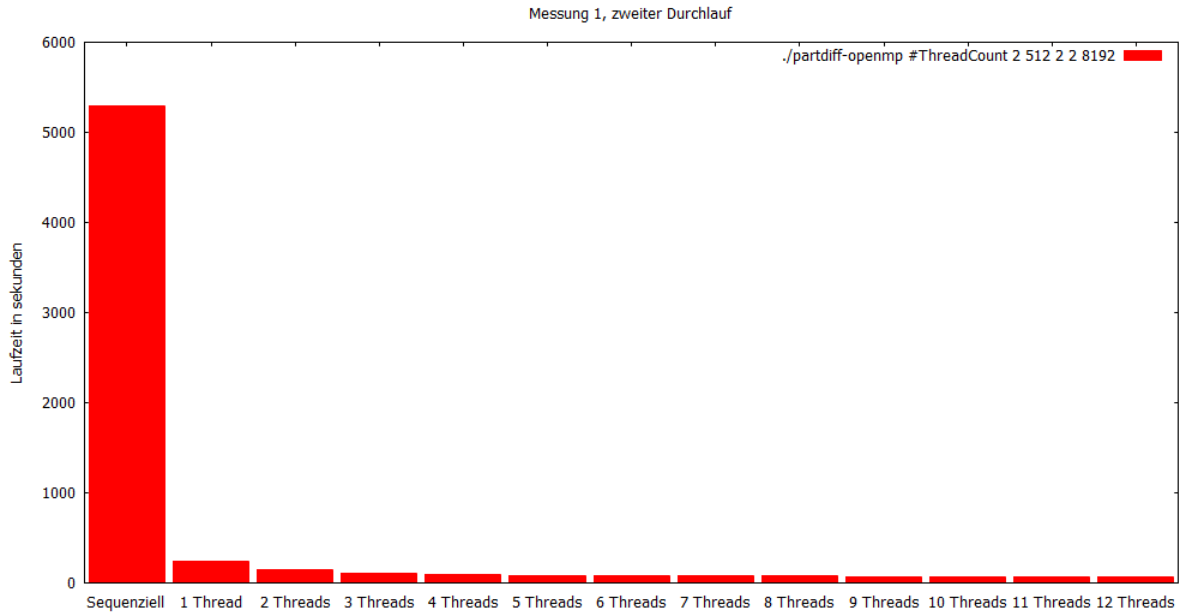
Zu sehen ist, dass die Laufzeiten des Element Programms deutlich länger sind, als die des Openmp Programms, welches hauptsächlich auf den Zeilen arbeitet. Dies könnte sich durch die häufigen Kontextwechsel erklären lassen, da der Lauf mit dem dynamic schedule einen extremen Unterschied aufweist. Auch könnten die Interna von Openmp dieses Phänomen verursachen, da eine dynamic schedule Berechnung auch länger als eine static schedule Berechnung braucht. Dafür erzeugt der guided schedule einen enormen Geschwindigkeitsboost auch wenn dieser immer noch nicht an die Basisimplementierung heranreicht. Als Hauptgrund könnte wie bereits beschrieben der häufige Kontextwechsel schuld sein, allerdings ist auch die Nutzung des Caches nicht sehr optimal, da die Elemente über die Threads verstreut auf den Cache zugreifen, daher ist auch ein Laufzeitunterschied der static schedules bemerkbar, welche immer besser wird je größer die Chunksize ist.

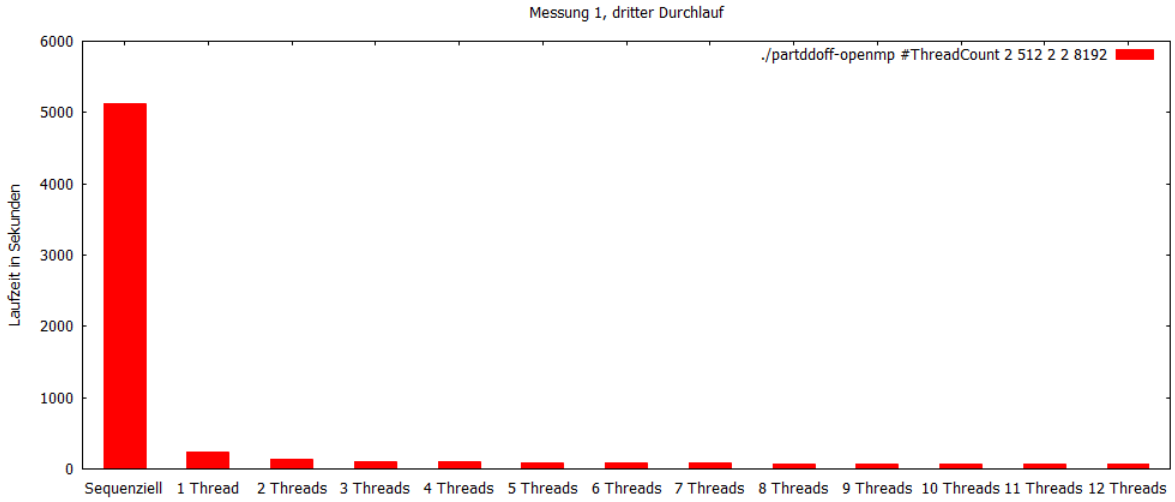
Leistungsanalyse

a)

Für die erste Messung wurde das Programm mit den Parametern:
./partdiff-openmp i 2 512 2 2 8192 gestartet, wobei $i \in \{1, \ldots, 12\}$



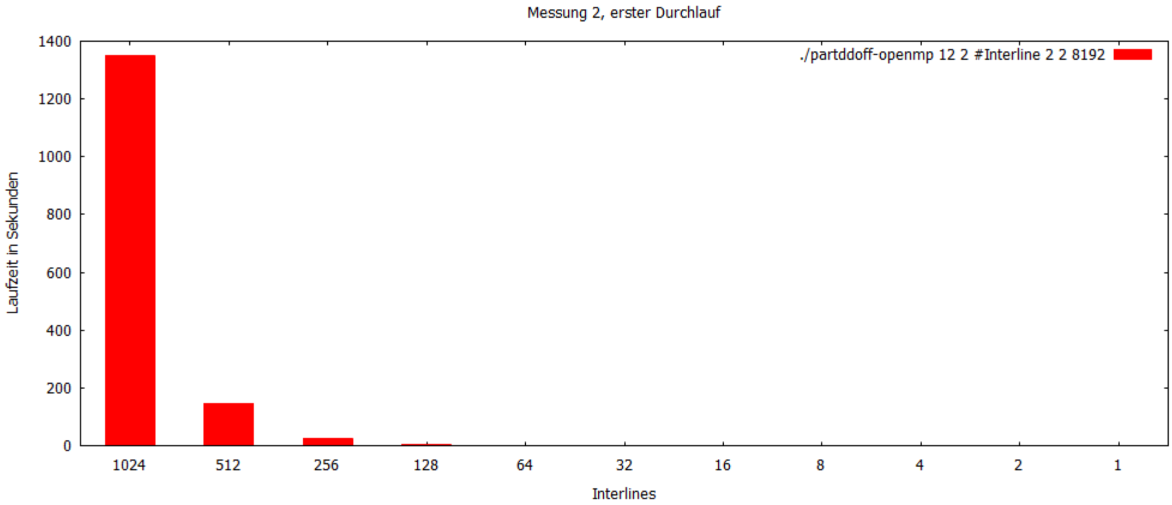


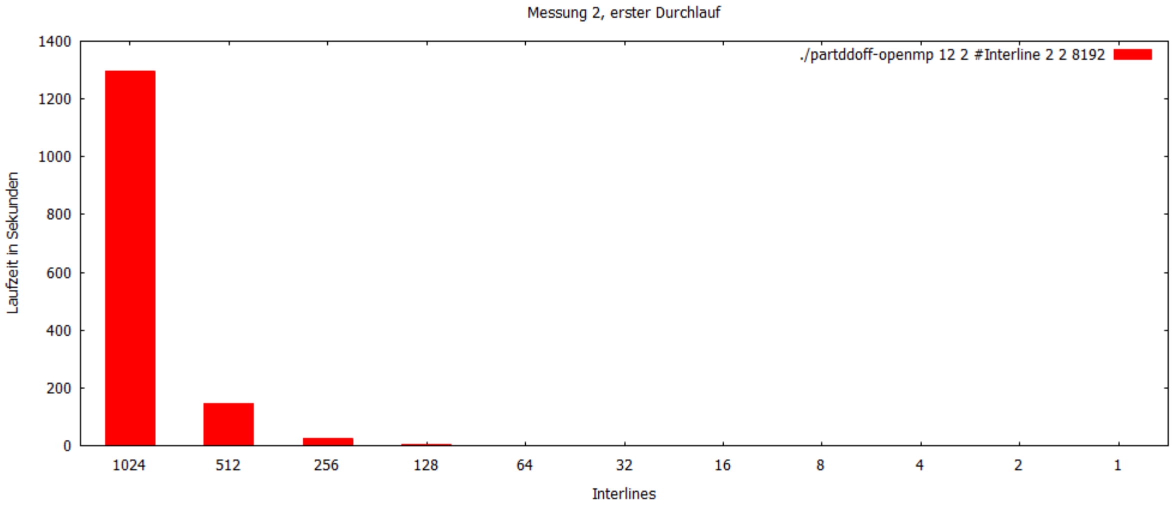


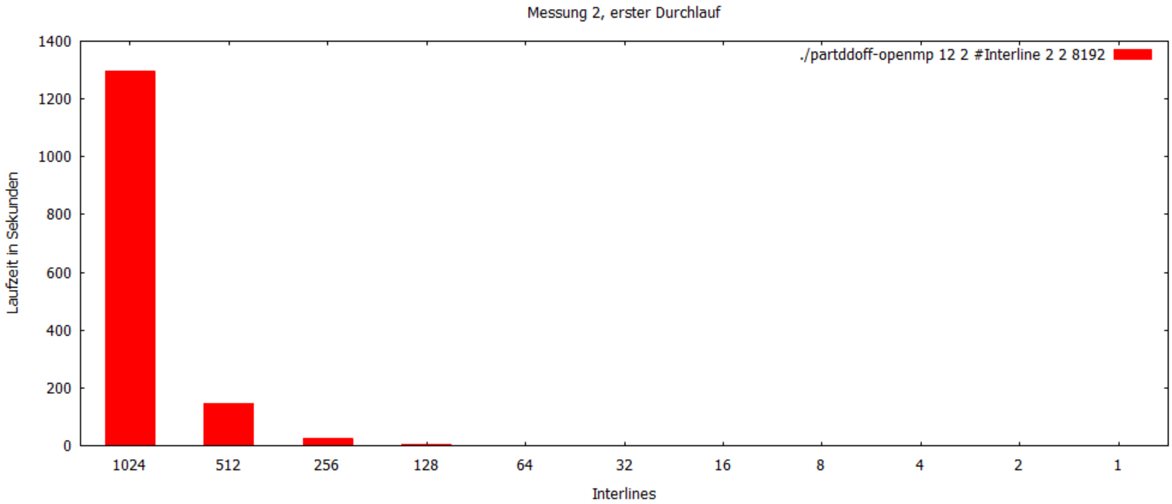
Die Ergebnisse sind ziemlich eindeutig zugunsten der Parallelisierung. Die Balken ähneln weiterhin der Funktion $\alpha \cdot \frac{1}{x^2}, \alpha \in \mathbb{R}$ und es ist außerdem deutlich zu erkennen, dass die Geschwindigkeit nicht linear mit der Anzahl der Cores steigt. Dies könnte sich durch schedulung Vorgänge innerhalb von OpenMp erklären, da die Kontextwechsel die Geschwindigkeit ausbremst. Alternativ kann es auch an dem reduction Vorgang bzw gleichzeitiger Zugriff auf einen Datensatz handeln, der durch ein Lock geschützt ist, und somit ein andere Thread warten muss.
Weiterhin kann es sein, dass das Betriebssystem die Ressourcen an andere Jobs verteilt, und so dem Prozess keine Rechenzeit zugeteilt wird.
Des Weiteren kann Hyperthreading auch auf die Leistung einwirken, da sich die Kerne einen Cache teilen müssen, und somit der Cache schneller voll wird. Damit steht nicht der volle Cache zur Verfügung, und somit müssen Werte häufiger aus dem Hauptspeicher in den Cache geladen werden.

b)

Für die Messung wurden der Aufgabe entsprechen $2^i, i \in \{1, \dots, 10\}$ Interlines verwendet. Für die Anzahl der Iterationen wurden 8192 festgelegt.







Die extrem hohe Laufzeit mit den Interlines von 1024 lässt sich damit erklären, dass die Zeilen der Matrix nicht mehr ganz in den Cache passen. Damit müssen während der Berechnung Daten aus dem RAM in den Cache geladen werden, was die Geschwindigkeit deutlich verlangsamt. Da mit 512 Interlines deutlich mehr Daten in den Cache passen und somit die schneller Adressiert werden können.
Auch ist anzumerken, dass die Laufzeit ab 64 Interlines und darunter sich nicht mehr unterscheidet, auch dies lässt sich durch den Cache erklären, da nun alle Daten die nötig sind in den Cache passen, und somit die Cores mit voller Leistung rechen können.