PARALLEL PROGRAMMING

Assignment 4

MPI – Matrix Multiplication

Tammo Johannes Herbert (319391), Rico Jasper (319396), Erik Rudisch (343930)

19.06.2013

Exercise 1 – Going for Speed

Zur Lösung der Aufgabe haben wir sieben Versionen mit verschiedenen Ansätzen zur Lösung entwickelt (siehe auch Abbildung 1: Versionsabhängigkeit). Hier ist eine Auflistung der Versionen und ihrer jeweiligen Bestwerte:

Version 0:

- Unveränderte Version
- Keine Compileroptimierung
- WpS: 67,7 (96x96), 14,3 (1024x1024)

Version 1: basiert auf Version 0

- Compileroptimierung -O3 -funroll-loops -march=native
- WpS: 581,5 (96x96), 56,5 (1024x1024)

Der Performancesprung ist für beide Matrixgrößen groß. Dennoch übertrifft die Performance für kleine Matrizen bei weiten die der großen. Grund dafür sind Cache Misses.

Basierend auf Version 1 haben wir drei verschiedene Verbesserungen ausprobiert, welche auf eine erhöhte Parallelität auf Instruktionsebene abzielen.

In Version 2A wurde die innere k-Schleife abgerollt.

Version 2A: basiert auf Version 1

- k-Schleife abgerollt
- WpS: 881,2 (96x96), 41,7 (1024x1024)

Es gibt einen leichten Anstieg für kleine Matrizen. Den Performanceverlust für große Matrizen können wir uns allerdings nicht ganz erklären. Das Problem der Cache Misses bleibt bei dieser Version bestehen.

In Version 2B wird das Problem der Cache Misses angegangen, indem statt der k- die j-Schleife abgerollt wird.

Version 2B: basiert auf Version 1

- j-Schleife abgerollt
- WpS: 1001,8 (96x96), 558,6 (1024x1024)

Dies hat einen großen Performancegewinn für große Matrizen zur Folge. Selbst kleine Matrizen scheinen davon zu profitieren.

Eine andere Variante zur Bewältigung der Cache Misses ist Matrix B zu transponieren und die k-Schleife abzurollen.

Version 2C: basiert auf Version 1

- Matrix B transponiert
- WpS: 919,2 (96x96), 579,8 (1024x1024)

Die Ergebnisse sind vergleichbar mit Version 2B. Der Overhead der Transponierung sorgt jedoch etwas schlechtere Werte.

Die folgendenen Versionen basieren jeweils auf die drei vorherigen, die nun parallelisiert wurden. Dabei wurde stets die äußerste Schleife (i) parallelisiert.

Version 3A: basiert auf Version 2A

- i-Schleife parallelisiert
- WpS: 2137,5 (96x96), 249,0 (1024x1024)

Version 3B: basiert auf Version 2B

- i-Schleife parallelisiert
- WpS: 2208,6 (96x96), 1315,2 (1024x1024)

Version 3C: basiert auf Version 2C

- i-Schleife parallelisiert
- WpS: 1826,6 (96x96), 1031,2 (1024x1024)

Die Parallelisierung führt in Anbetracht einer theoretisch möglichen Verbesserung um den Faktor 24 nur zu einer geringen Leistungssteigerung. Möglicherweise existiert hier eine Art Flaschenhals.

Hier noch mal eine Übersicht über alle Versionen:

Version	Quelltextversion	WpS (96x96)	WpS (1024x1024)
0	v0	67,7	14,3
1	v1	581,5	56,5
2A	v2	881,2	41,7
2B	v3	1001,8	558,6
2C	v6	919,2	579,8
3A	v5	2137,5	249,0
3B	v4	2208,6	1315,2
3C	v7	1826,6	1031,2

Tabelle 1: Leistungsmessung

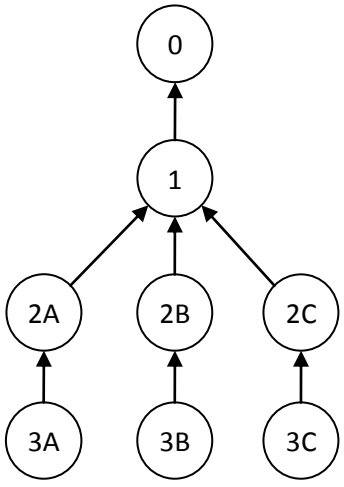

Abbildung 1: Versionsabhängigkeit

Exercise 2 – Sparse Matrices

(a) Sparse Matrix Vector Multiplication

Wir summieren für jede Zeile der Eingabematrix A bzw. des Ausgabevektor c die Summe von Produkten typisch für eine Matrix-Vektor-Multiplikation. Jedoch werden pro Zeile nur die nicht-null Elemente der Matrix A mit dem Eingabevektor b multipliziert. Daher läuft die innere Schleife nur von `row_ptr[i]` nach `row_ptr[i+1]-1`. Die Spalte wird entsprechend über `col_ind[j]` bestimmt.

```
1  for i in 0 ... n-1                              // for each row
2        c[i] = 0
3        for j in row_ptr[i] ... row_ptr[i+1]-1    // for each column
4              c[i] += val[j] * b[col_ind[j]]
```

(b) Parallelization

Für eine Parallelisierung ist es wünschenswert, dass jede Recheneinheit möglichst dieselbe Menge an Arbeit verrichtet. Die Arbeit pro Zeile ist aber abhängig von der Anzahl der nicht-null Elementen und kann daher variieren.

Die zeilenweise parallelisierte Variante erfordert außerdem das replizieren des Eingabevektors b. Im Falle der CRS-Matrix A kann dieser Vektor allerdings in Relation jedoch groß ausfallen, da die Dichte von nicht-null Elementen recht dünn sein kann.

Hinsichtlich des ersten Problems kann man die Lasten versuchen auszugleichen. Beispielsweise könnten mehrere Zeilen einer Recheneinheit zugeteilt werden, um die Gesamtverteilung auszugleichen. Denkbar ist jedoch auch das Aufbrechen von „großen“ Zeilen in kleinere Stücke.

(c) Embedding into a FEM simulation

Die Matrix kann in den gegebenen Varianten A und B auf Grund der Bandstruktur in einer sparsamen statischen Struktur dargestellt werden. Beispielsweise könnte man in einem Array lediglich das Band speichern, womit die Nullen jenseits des Bandes nicht mehr betrachtet werden müssten. In Variante A könnte eine Eingabematrix beispielsweise so aussehen:

$$A = \begin{pmatrix} 1 & 2 & 0 & 0 & 5 & 0 & 0 & 0 \\ 1 & 2 & 3 & 0 & 0 & 6 & 0 & 0 \\ 0 & 2 & 3 & 4 & 0 & 0 & 7 & 0 \\ 0 & 0 & 3 & 4 & 0 & 0 & 0 & 8 \\ 1 & 0 & 0 & 0 & 5 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 5 & 6 & 7 & 0 \\ 0 & 0 & 3 & 0 & 0 & 6 & 7 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 7 & 8 \end{pmatrix}$$

In einer kompakteren Darstellung könnte auf Grund der Bandstruktur und das Muster der Nullen innerhalb des Bandes nun die Elemente einer Zeile wie folgt angeordnet werden:

$$A^* = \begin{pmatrix} 0 & 0 & 1 & 2 & 5 \\ 0 & 1 & 2 & 3 & 6 \\ 0 & 2 & 3 & 4 & 7 \\ 0 & 3 & 4 & 5 & 8 \\ 1 & 0 & 5 & 6 & 0 \\ 2 & 5 & 6 & 7 & 0 \\ 3 & 6 & 7 & 8 & 0 \\ 4 & 7 & 8 & 0 & 0 \end{pmatrix}$$

Nach einem ähnlichen Vorgehen für Variante B könnte die kompakte Matrix A^* so aussehen:

$$A^* = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 4 \\ 1 & 3 & 4 & 5 \\ 2 & 3 & 4 & 6 \\ 3 & 5 & 6 & 7 \\ 4 & 5 & 6 & 8 \\ 5 & 7 & 8 & 0 \\ 6 & 7 & 8 & 0 \end{pmatrix}$$

Für eine parallele Matrix-Vektor-Multiplikation ist es auch hier möglich die Matrix A^* zeilenweise aufspalten. Dabei müsste man nun aber nicht mehr zur Berechnung einer Zeile Zugriff auf den vollständigen Vektor besitzen. Beispielsweise benötigt man in Variante B für die Berechnung der fünften Zeile nur die Elemente 3, 5, 6 und 7 des Vektors. Das Ergebnis müsste dementsprechend nur an die Rechenknoten übertragen werden, welche dieses auch benötigen. Auf Grund der symmetrischen Adjazenzmatrix sind das ebenfalls 3, 5, 6 und 7.

Ist die Vernetzung nicht symmetrisch und vorhersehbar, so könnte man A^* jedoch nicht so kompakt gestalten. In diesem Fall müsste man zumindest das Band in der vollen Breite abbilden. Wenn es dann dazu noch dünn ausgefüllt ist, so stößt man auf ähnliche Probleme, wie schon in (b). Je schmaler man in so einem Fall das Band ist, umso besser.