Höhere Programmierkonzepte
Praktikum III

Prof. Dr. Nikolaus Wulff

24. Nov – 3. Dez 2020

1 Robuster JUnit Matrix Test

Sie haben in Praktikum II bereits recht einfache Tests für die Matrizenmultiplikation entwickelt – allerdings meist nur mit 2x2 oder 3x3 Matrizen und ganzzahligen Einträgen. Sie benötigen robustere Tests, zu denen Sie das Ergebnis analytisch vorher wissen bei beliebiger Dimension n der Matrizen. Sie holen diesen fehlenden Test aus Praktikum II zunächst nach bevor es an die eigentliche Parallelisierung geht.

Hilbert Matrizen

Sehr einfach wird das Arbeiten mit JUnit, wenn Sie zum Problem die Lösung schon kennen und somit *Axela* und den *Iris* Solver gezielt testen können. Dies ist z.B. immer dann der Fall wenn Sie zu einer Matrix A deren Inverse A^{-1} kennen. Erweitern Sie ihre bisherigen Tests um die Testmethode `testHilbertMatrix` wie im Listing (1), in dem Sie in ihrem Test zwei Hilfsmethoden zum Erzeugen einer Hilbert[1] Matrix $\mathcal{H}_n$ und deren Inversen $\mathcal{H}_n^{-1}$ zu beliebig vorgegebener Dimension n einbauen.

Das Produkt $\mathcal{H}_n * \mathcal{H}_n^{-1} \equiv E_n$ muss gleich der n-dimensionalen Einheitsmatrix E_n sein, was sich sehr einfach als Test-Setup für die Genauigkeit Ihrer Matrizenmultiplikation bei exakt bekanntem Ergebnis[2] verwenden lässt. Sie werden bemerken wie schnell sich bereits bei kleinen Dimensionen $n < 10$ numerische Fehler signifikant aufschaukeln! So dass Sie noch einmal auf die besprochene `assertEqualsMatrix` Methode aus der Vorlesung mit einer vorgegebenen Toleranz Δ verwiesen werden. Wähle ich sonst meist $\Delta = 1.\text{E-}14$,

[1] Der Mathematiker David Hilbert definierte 1894 diese symmetrischen $n \times n$ Matrizen mit Einträgen $(\mathcal{H}_n)_{ij} = \frac{1}{i+j-1}$ für $1 \leq i, j \leq n$, d.h. Einträgen $1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots$. Auch die inverse Matrix lässt sich geschlossen mit Hilfe des Binominalkoeffizienten angeben. Eine einfache Übersicht und Formeln finden Sie bei Wikipedia.

[2] Ähnliche Tests lassen sich auch mit Vandermonde und anderen bekannten Matrizen erstellen, für die auch die Inversen analytisch berechenbar sind. Das Bestehen solcher Tests ist wichtige Voraussetzung bevor Sie es wagen sollten mit Zufallsmatrizen zu arbeiten.

so ist dies für die Hilbert $\mathcal{H}_n$ Tests selbst bei `double` Genauigkeit und moderaten Dimensionen $6 < n < 10$ nicht mehr möglich.

```
1  @Test
2  void testMultiplyHilbertMatrix() {
3    delta = 1.E-8;
4    for (n = 1; n <= 7; n++) {
5      double[][] a = createHilbertMatrix(n);
6      double[][] b = createInverseHilbertMatrix(n);
7      double[][] expected = createIdMatrix(n);
8      double[][] returned = axela.solve(TIMES, facts(a, b));
9      assertMatrixEquals(expected, returned, delta);
10   }
11 }
```

Listing 1: JUnit Test mit Hilbert Matrizen.

2 Parallele Matrix Multiplikation

Das Lösen linearer Gleichungen und die Matrizenmultiplikation sind sehr häufig vorkommende Aufgaben in naturwissenschaftlichen Anwendungen. Für quadratische $n \times n$ Matrizen hat das Matrixprodukt eine Laufzeit $\mathcal{O}(n^3)$, d.h eine Verdoppelung von n bedeutet eine achtfache Rechenzeit. Das Matrixprodukt $\mathbb{R}^{r\times t} \ni \mathcal{C} = \mathcal{A}\cdot\mathcal{B}$ zweier reeller Matrizen $\mathcal{A} \in \mathbb{R}^{r\times s}$ und $\mathcal{B} \in \mathbb{R}^{s\times t}$ ist elementweise definiert als

$$\mathcal{C}_{ij} = \sum_{k=1}^{s} \mathcal{A}_{ik} \cdot \mathcal{B}_{kj} \qquad 1 \le i \le r,\ 1 \le j \le t\,. \tag{1}$$

Da Matrizenmultiplikation eine häufige numerische Operation ist, lohnt es sich diese wichtige Operation mit geeigneten parallelen Algorithmen zu beschleunigen, die es im Rahmen dieses Praktikums zu entwickeln gilt.

2.1 Parallele Versionen

Die Algorithmen „matParallel$_{1\ldots4}$“ der Tabelle [1] beschreiben als Pseudocode verschiedene Möglichkeiten Matrizen in parallelen Threads zu multiplizieren. „matParallel$_1$“ parallelisiert z.B. die beiden äußeren Schleifen und führt die Berechnung der innersten Schleife als voneinander unabhängige, sequentielle Aufgaben aus, während „matParallel$_2$“ lediglich die mittlere Schleife parallelisiert. Algorithmen 3 und 4 verwenden die transponierte Matrix $\mathcal{B}^T$ bzw. den transponierten Spaltenvektor $\vec{b}_j^T$ der Matrix $\mathcal{B}$. Es handelt sich sicherlich um keine optimalen Algorithmen aber sie sind recht einfach zu implementieren und eine gute Übung, um parallele Konzepte in Java zu erlernen.

beg algorithm „matParallel$_1$“
variables a, b, c: type matrix
con for $i = 1$ **to** $a.rows$ **do**
con for $j = 1$ **to** $b.cols$ **do**
for $k = 1$ **to** $a.cols$ **do**
c_{ij} += $a_{ik} \cdot b_{kj}$
end
end /* end con inner */
end /* end con outer */
end algorithm /* „matParallel$_1$“ */
	beg algorithm „matParallel$_2$“
	variables a, b, c: type matrix
	for $i = 1$ **to** $a.rows$ **do**
	con for $j = 1$ **to** $b.cols$ **do**
	for $k = 1$ **to** $a.cols$ **do**
	c_{ij} += $a_{ik} \cdot b_{kj}$
	end
	end /* end con */
	end
	end algorithm /* „matParallel$_2$“ */

beg algorithm „matParallel$_3$“ **variables** a, b, c, d: type matrix $d = b^T$ **con for** $i = 1$ **to** $a.rows$ **do** **for** $j = 1$ **to** $b.cols$ **do** $c_{ij} = \vec{a}_i \cdot \vec{d}_j$ **end** **end** /* end con */ **end algorithm** /* „matParallel$_3$“ */	**beg algorithm „matParallel$_4$“** **variables** a, b, c: type matrix **con for** $i = 1$ **to**$a.rows$ **do** **for** $j = 1$ **to** $b.cols$ **do** $c_{ij} = \vec{a}_i \cdot \vec{b}_j^T$ **end** **end** /* end con */ **end algorithm** /* „matParallel$_4$“ */

Tabelle 1: Pseudocode einiger paralleler Algorithmen zur Matrizenmultiplikation $\mathcal{C} = \mathcal{A} \cdot \mathcal{B}$, beachten Sie die parallele Schleifenanweisung **con for**.

Anhand der Zeiten in Tabelle (2.2) ist gut zu erkennen, dass diese wirklich *sehr naiven*(!) Algorithmen – wenn überhaupt –, nur für große Matrizen einen Zeitgewinn bieten.

1. Erweitern Sie Ihren JUnit Test um Methoden Zufallsmatrizen[3] beliebiger Größe $A, B \in \mathbb{R}^{n \times m}$ als Funktion der Parameter n, m zu generieren und in Ihren Testfällen zu verwenden. Da Sie sich mit den Hilbert Matrizen davon überzeugt haben, dass ihre `mult` und `assertMatrixEqual` Methoden im JUnit Test richtig rechnen, können Sie dagegen nun auch ihre optimierten und/oder parallelisierten Axea/Iris Implementierungen mit beliebig großen Zufallsmatrizen testen.

2. Entwickeln Sie passend zu den Pseudo-Codes der Tabelle [1] einen seriellen Algorithmus `matSeriell_x` mit $x \in \{1..4\}$, d.h. statt **con for** werden überall nur normale for-Schleifen eingesetzt. Wählen Sie den Ihnen am geeignetsten erscheinenden Algorithmen aus, von dem Sie sich eine Optimierung[4] versprechen und begründen Sie Ihre Entscheidung.

[3] Zufallszahlen lassen sich mit `Math.random()` generieren.

[4] Ja Sie lesen richtig bereits diese einfachen unterschiedlichen seriellen for-Schleifen bergen ein Optimierungspotential, durch die Art der Speicherzugriffe!

3. Implementieren Sie anschließend denselben Algorithmus in einer parallelen Variante `matParallel_x` passend zu dem Pseudo-Codes der Tabelle [1].

   Spannend ist die Antwort auf die Frage: wie verhält sich die parallele zur optimierten Variante und diese Beiden wiederum zur naiven Implementierung der dreifach geschachtelten for-Schleife des JUnit Tests **die nicht optimiert werden soll!**. Diese nicht optimierte Variante des JUnit Tests dient als *baseline* für vergleichende Laufzeitmessungen und natürlich zur Überprüfung der Korrektheit der Rechenergebnisse.

4. Instrumentieren Sie den Test mit einer Zeitmessung[5] und berechnen Sie den Speed-Up Faktor als Funktion der Matrizengröße für große Matrizen mit $n \sim 128 - 2048$ analog zur Tabelle (2.2). **Vergewissern Sie sich, dass auch die parallelen Algorithmen die richtigen Ergebnisse liefern und nicht nur schneller aber falsch rechnen** - oder aber die Ergebniss der parallelen Berechnung noch gar nicht vorliegen (Stichwort *Future*)!

5. Versuchen Sie die unterschiedlichen Laufzeiten der drei implementierten Algorithmen zu interpretieren und die gewonnenen Erkenntnisse zu deren Optimierung auszunutzen. Gerade bei der Synchronisation der Threads gibt es eine Menge an Optimierungspotential, evt. bringt ein *ThreadPool* zum Recyceln der Threads einen deutlichen Performancegewinn. Das *java.util.concurrent*-Package hält hierzu eine Menge an Hilfsmitteln bereit.

Hinweis

Einen Test erweitern heißt zusätzliche Testmethoden hinzufügen oder evt sich von einem anderen Test abzuleiten aber auf keinen Fall schon laufende Testfälle zu löschen und dann beim Abtestat nicht mehr vorzeigen zu können, da sie ja schon zu Hause oder vor zwei Wochen im Praktikum II liefen und nun "entfernt wurden" da sie überflüssig¸sind. Nein einmal geschriebene Tests werden fast nie überflüssig und verhindern das sich bereits gefixte Bugs wieder einschleichen... Und Tests müssen nicht unbedingt hochperformant sein, da sie nicht in den Produktivcode einfließen. Sei müssen allerdings zuverlässig und exakt sein. Es gibt nichts schlimmeres als sich im Projekt auf falsche Test(Ergebnisse) zu verlassen.

[5] Die Anweisung `System.nanoTime()` liefert ausreichend genaue Zeitmessungen.

2.2 Vergleichende Zeitmessung

dim	baseline	optimized		parallelized		speedUp	Amdahl
n	b[ms]	s[ms]	b/s	p[ms]	b/p	s/p	pi[%]
128	2198	1567	1,4	605	3,6	2,6	70
256	25112	13125	1,9	2593	9,7	5,1	92
512	223358	109508	2,0	17722	12,6	6,2	96
1024	10874039	956144	11,4	134364	80,9	7,1	98
2048	115253961	8083989	14,3	1112870	103,6	7,3	99

Die angegebenen Zeitmessungen dienen nur als ungefähre Anhaltspunkte. Ihre Meßwerte können je nach Anzahl an CPUs oder laufenden Prozessen und nach der genauen Ausgestaltung der Algorithmen davon abweichen. Auf meinem Oktocore Desktop Rechner liefert die serielle optimierte Version meines Algorithmus durch Ausnützung der Cache-Lines bereits einen Speed-Up von ~ 10, wenn ich diesen dann auch noch parallelsiere kommt noch einmal ein Faktor $\sim 5-7$ hinzu. Der optimierte parallele Algorithmus ist also insgesamt ~ 100 Mal schneller als die naive Matrizenmultiplikation aus Praktikum II. Wenn ich mit diesen Messungen den Parallelisierungsgrad nach Amdahl's Gesetz ausrechne (letzte Spalte der Tabelle) komme ich auf eine Parallelisierung von $\pi \simeq 99\%$.

Ein **fehlendes, deutliches Speed-Up** ist in jedem Fall ein klares Indiz für eine fehlerhafte und/oder ungünstige Implementierung eines parallelen Algorithmus.