Meilenstein II - Abgabe

Marlene Böhmer, 2547718
Andreas Meyer, 2552569

17.08.2015

Überblick

Als Sprache haben wir uns für Java entschieden. Für die Implementierung orientieren wir uns am Fork- Join Modell. Vorab: Wir unterscheiden zwischen globaler und lokaler Iteration. Letzteres meint die Berechnung des Outflows und das setzten der Akkus einer Spalte, ersteres meint eine Anzahl lokaler Iterationen, und anschließendes Austauschen des horizontalen Flows und die Neuberechnung der Knotenwerte.

NPOsmose erzeugt ein Supervisor Objekt. Der Supervisor enthält ein Objekt des Typs Grid, eine von uns zu implementierende Klasse, über die erstmal nur zu sagen ist, dass sie das Gitter darstellt. Über den Supervisor wird die Anzahl der lokalen Iterationen gesteuert, dazu kann man maxLocal anpassen. Außerdem wird hier der ExecutorService erzeugt und beendet, der nachher die nebenläufige Arbeit auf die Threads verteilt.

Die Methode computeOsmose() stößt mit Grid.globlIteration() die Berechnung einer globalen Iterationen an und erwartet eine Rückgabe, die über die Konvergenz informiert. Wenn computeOsmose() terminiert wird das Grid, welches ImageConvertible implementiert, zurückgegeben.

Das Grid enthält alle Columns, die Callable implementieren, und startet mit der ExecutorService Methode invokeAll() die nebenläufige Berechnung einer globalen Iteration auf allen Spalten, erhält von jeder Spalte ein double zurück, um auf Konvergenz zu prüfen.

Die Column führt mit call() eine globale Iteration wie folgt aus:

1. Phase: Die lokale Iteration zur Berechnung des vertikalen Flows und des horizontalen Outflows aller Knoten einer Spalte. Falls die Änderung der Knotenwerte klein genug ist, werden die lokalen Iterationen abgebrochen um Ressourcen zu sparen.

2. Phase: Austausch des Outflows zwischen den Spalten durch einen Exchanger.

3. Phase: Berechnet den für den aktuellen globalen Iterationsschritt endgültigen Werte aller Knoten und berechnet für alle Knoten i: x=$\sum_i (outflow_i - inflow_i)^2$, x wird dann ans Grid zurückgegeben um die Konvergenz zu prüfen.

Nähere Informationen zur Implementierung können der Javadoc und den Kommentaren im Code entnommen werden.

Data Races und Terminierung

Um Data Races zu vermeiden sind alle Klassen als Monitore implementiert und alle Attribute mit privat bzw. protected gekapselt. Bei public Methoden werden nur Werte (keine Referenzen) übergeben, mit Ausnahme der Getter für die Outflows der Spalten der für den Austausch im sequentiellen Teil gedacht ist. Im nebenläufigen Kontext wird der Outflow mittels Exchanger zwischen den Spalten getauscht. Da nach dem Tausch aber sowohl die Werte des Inflows als auch des Outflows bekannt sein müssen, speichert sich jede Spalte vor dem Austausch eine Referenz auf den Outflow zwischen. Die Objekte müssen nicht durch Locks geschützt werden, da bis zum Austauschen alle Schreizugriffe abgeschlossen sind und duch das blockierende Warten beim Austauschen auch garantiert ist, dass nach dem Austauschen die Nachbarspalten fertig mit den Schreibzugriffen sind. Nach dem Austauschen finden dann nur noch Lesezugriffe auf die (eigenen und gemerkten) Outflows statt und mit dem return werden die gemerkten Referenzen wieder vergessen. Es gibt also keine Dataraces.

Deadlocks beim Exchange werden dadurch vermieden, dass die äußeren Spalten nur in die Richtung tauschen, in der auch ein Nachbar existiert. Um ein verharren in Zyklen, durch seltenes horizontales propagieren, zu vermeiden, verringern wir die Anzahl lokaler Iterationen in dem Fall, dass sich die Inflow-Outflow Differenz über mehrere globale Iterationen nicht mehr an das Konvergenzkriterium annähert. Sollte die Anzahl der lokalen Iterationen bei 1 sein, wird in die sequentielle Ausführung gewechselt, in der auch das globale Konvergenzkriterium geprüft wird.