Nebenlaeufige Programmierung
Meilenstein 1

Daniel Schäfer (2549458)
Christian Bohnenberger (2548364)

16.07.2015

1 Einfuehrung

Nach anschauen des Codes ist uns klar geworden, dass es auf jeden Fall wichtig sein wird die Rate fuer jeden "Node" zu speichern um das staendige Iterieren ueber guards in getRateForTarget zu vermeiden.
Wir werden eine Node Klasse anlegen, die einen einzelnen Punkt implementiert. Dieser kennt seine x und y Koordinate als int, als auch seine Rate in die Richtungen oben, rechts, unten und links. Ebenfalls wichtig wird der double-Wert value, der den momentanen Wert des Knotens enthaelt. Ausserdem einen Akku fuer den Knoten links, rechts, oben und unten. Hier speichern wir Uebertraege in die entsprechenden Richtungen.
Eine komplette Spalte betrachten wir als Thread und in der Berechnung eines Iterationsschrittes des Osmozeprozesses werden wir zuerst fuer jeden Thread (also jede Spalte), die Akkumulatoren nach oben, unten, rechts und links anhand der gespeicherten Rates berechnen. Dies wird ueber einen iterativen Funktionsaufruf, der nacheinander die calculate() Funktion auf den Knoten der Spalte aufruft, uebernommen. Anschliessend rufen wir auf jedem Knoten die Funktion flow() auf. Diese fasst die Akkumulatoren nach links und rechts NICHT an, uebertragt jedoch die Uebertraege nach oben und unten in die entsprechenden Knoten obendrueber bzw. untendrunter und addiert die Akkumulatoren auf deren momentanen Wert. Communicate wird nach barriercount Iterationsschritten (barriercount ist vom Benutzer setzbar) aufgerufen und sorgt dafuer dass die Uebertraege nach links und rechts ueber die Spalten hinaus an die korrespondierenden Nachbarn verteilt werden. Das Erstellen neuer Nodes in Nachbarspalten ist synchronized um Data Races zu vermeiden. Es wird also eine Barrier geben die nach barriercount Schritten die Iteration unserer calculate() und flow() Funktion fuer jeden Thread 'anhaelt' um einen korrekten Aufruf von communicate() zu ermoeglichen.
Zu den Datentypen: Die Spalten werden wir als Hashmap von Nodes implementieren um Zugriffe mit einem perfekten Key auf bestimmte 'Positionen' in Laufzeit O(1), sowie einfuegen neuer Nodes an eine bestimmte Position in O(1) als auch angenehmes Iterieren ueber das valueset in O(n) zu ermoeglichen.
Ebenfalls sehr relevant zum Sparen von Speicher und der Effektivitaet unserer Hashmap ist die Tatsache, dass wir zu Beginn nur die Knoten erzeugen 'muessen' die tatsaechlich eine Value ¿ 0 enthalten. Beim Erzeugen von Knoten werden alle ihre Werte (also auch die Rate nach oben, unten usw.) berechnet. Wir haben uns gegen das Loeschen von Knoten entschieden die ihre Value 'nullen', da wir vermeiden wollen Knoten staendig erneut erzeugen zu muessen, da dies doch relativ aufwendig ist aufgrund der Iteration ueber Guards um die Rates zu erhalten. Zum Ersparen zusaetzlicher Laufzeit ist es wichtig, dass wir bei der Funktion communicate() ZUERST ueberpruefen ob ueberhaupt ein Uebertrag in Knoten (x,y) existiert bevor wir auf dessen Existenz ueberpruefen, da wir ihn nur erzeugen moechten (angenommen er waere noch nicht erzeugt) falls ein Akkumulator auf seinen Wert addiert werden wuerde.
Unsere Barriers (Cyclic barrier) implementieren wir in der Klasse Picture. Sie werden in

der run() Funktion einer Column, eingesetzt um Data Races und andere ungewollte nebenlaeufige Effekte zu vermeiden! Ebenfalls in Column enthalten sind die Columns leftColumn und rightColumn als Referenzen auf diese Spalten um nachher den Datenaustausch von commmunicate() einfacher zu realisieren. Die Spalten rufen also communicate() auf allen Knoten ihrer Spalte auf. Die einzelnen Spalten sind in der Klasse Picture in einer ArrayListe enthalten, die das komplette Osmosebild repraesentiert. Das Bild wird unter anderem die Spalten in entsprechender (maximal)Groesse erstellen und anschliessend die Referenzen der Nachbarspalten fuer jede Spalte setzen.
Wir betrachten lokale Konvergenz (Spaltenkonvergenz) indem wir die Summe der Akkumulatoren die 'nach links gehen' in einer Spalte addieren und schauen wie viel in der Spalte links daneben 'nach rechts geht'. Wir ueberpruefen also ob Inflow einer Spalte $\approx$ Outflow der Spalte. Bei einer Differenz $< \epsilon$ erkennen wir lokale Spaltenkonvergenz. Tritt in allen Spalten lokale Spaltenkonvergenz auf sprechen wir von globaler Spaltenkonvergenz. Diese muessen wir erkennen, da wir ab dann mit precisetest auf globale Konvergenz testen. Wie in der Aufgabenstellung gefordert setzen wir barriercount = 1.
Alle Effekte, die auf 'allen Spalten gleichzeitig' arbeiten muessen implementieren wir in der run methode einer der barrier die wir verwenden. So ist gewaehrleistet dass keine dataraces auftreten. An solchen stellen testen wir zB auf globale und oder lokale Konvergenz, wir ermoeglichen das Ausgeben von testrestults nach einer gewissen Anzahl von Schritten. Diese Schrittanzahl ist natuerlich fuer den Benutzer nach belieben waehlbar, genau wie der Wert barriercount.
Globale Konvergenz erkennen wir exakt wie in der Aufgabenstellung gefordert unter verwendung der zweiten euklidischen Norm. Sollte dieser Wert $\leq \epsilon$ sein setzen wir eine Art selbstimplementierte 'flag' in jedem Thread, der den Thread in naher Zukunft zum terminieren bringen wird (aehnelt der Message Parsing Idee einen boolchan zu haben der Terminierung mitteilt).

Wichtig ist zu erwaehnen das jede Spalte fuer sich nebenlaeufig rechnet, innerhalb einer Spalte(=eines Threads) calculate() und flow() jedoch sequentiell berechnet werden.