Nebenläufige Programmierung
Sommersemester 2015

Dennis Nikolay - 2544912
Moritz Beck - 2544065

Projektkonzept

Unsere Idee zur Lösung des Projekts ist wie folgt:
Jede Spalte wird durch einen separaten Thread repräsentiert, der die folgenden Felder hat:

1. **ArrayList<double> vertexValue** - repräsentiert die y-Koordinaten des jeweiligen Knotens und den Wert. Dazu wird das Zehnfache der Koordinate auf den Wert addiert. Dies ermöglicht die spätere Rekonstruktion auf ersichtliche Weise (Werte <= 1). Die Knoten werden nur gespeichert, falls der Wert ungleich Null ist. Das Einfügen erfolgt sortiert.
2. **int columnIndex** - speichert die x Koordinate der jeweiligen Spalte, die der Thread bearbeitet.
3. **ArrayList<double> accValues** - repräsentiert die y-Koordinaten des jeweiligen Akkumulators und den Wert. Dazu wird das Zehnfache der Koordinate auf den Wert addiert. Dies ermöglicht die spätere Rekonstruktion auf ersichtliche Weise, da Werte <= 1. Um Letzteres sicherzustellen wird nach jeder Iteration der Durchschnitt des zu propagierenden Wertes berechnet. Die Knoten werden nur gespeichert, falls der Wert ungleich null ist. Das Einfügen erfolgt sortiert.
4. **sync double valueSum** - Summe aller Knotenwerte
5. **AtomicBoolean terminate** - Bool flag zur Einleitung der Terminierung
6. **Exchanger leftExchanger** - Exchanger zum Propagieren der Werte zum linken Nachbarn
7. **Exchanger rightExchanger** - Exchanger zum Propagieren der Werte zum rechten Nachbarn

Vertikales Propagieren

Jeder Thread iteriert über *vertexValues*. Für jeden Knoten innerhalb der Liste werden alle der möglichen Regeln entweder auf die Knoten direkt (up/down) oder auf die Akkumulatoren (left/right) angewandt. Dazu wird das nächste oder vorherige Element der Liste betrachtet. Falls die Koordinate mit der Erwarteten übereinstimmt, wird propagiert - unter Berücksichtigung, dass jeder Knoten erst propagiert und anschließend seinen Wert aktualisiert. Andernfalls wird ein neues Element mit der erwarteten Koordinate und dem zu propagierenden Wert an dieser Stelle eingefügt. Dadurch ist die Liste sortiert. Falls ein Wert Null wird, wird dieses Element aus der Liste gelöscht. Falls wir feststellen, dass der Wert sich nur geringfügig vom vorherigen unterscheidet, wird das vertikale Propagieren gestoppt und erst wieder aufgenommen, sobald einmal horizontal propagiert wurde. Dadurch wird nicht unnötig vertikal iteriert und dennoch das Perturbieren durch horizontale Transitionen beachtet.
Wir haben uns aufgrund von Speichereffizienz für dieses Design entschieden.

Horizontales Propagieren

Wir berechnen zunächst den Unterschied zwischen *inflow* und *outflow*. Der Zeitpunkt des Austauschs berechnet sich beim horizontalem „Flow“ immer gleich (deshalb $i = k$), da beide Threads gleiche Berechnungen mit identischen Werten ausführen. Dabei ist es möglich, dass der Austausch mit dem linken Nachbarn öfter als mit dem Rechten stattfindet und umgekehrt. Je kleiner der Abstand von *inflow* zu *outflow*, desto öfter wird ausgetauscht. Der Austausch erfolgt über einen Java-Exchanger, wodurch sichergestellt wird, dass die Übergabe der Masse korrekt gehandhabt wird. Nach Austausch der Akkumulatorliste wird das Feld mit einer neuen Liste initialisiert. Wenn *leftExchanger* bzw. *rightExchanger* noch nicht initialisiert worden sind, wird ein neuer Thread gestartet und dies dem Observer mitgeteilt. Letzterer kontrolliert den Programmverlauf mithilfe einer Liste aller Threads.

Konvergenzverhalten und Terminierung

Eine Instanz der Klasse Observer überwacht den Programmverlauf. Jeder Thread (Observable) benachrichtigt den Observer sobald er jede Iteration austauscht (evtl. lokale Konvergenz). Falls dies bei allen Threads der Fall ist, wird auf globale Konvergenz geprüft. Dazu wird das Feld der *valueSum* aller Threads vom Observer gelesen.
Am Ende ist nicht zwangsläufig in allen Spalten gleich oft iteriert worden, allerdings spielt dies keine Rolle, weil sich die Werte in den Spalten in nachfolgenden Iterationen kaum mehr ändern werden. Dadurch wird eine globale Instanz, die die Anzahl der lokalen Iterationen synchronisiert, vermieden. Sobald wir globale Konvergenz erkannt haben, setzen wir in allen Threads den Boolean *terminate* auf true, woraufhin diese zum nächstmöglichen Zeitpunkt terminieren.

Shared Memory

Data Races werden bei gemeinsamen Daten durch Verwendung von Monitormethoden ausgeschlossen (siehe Klassenbeschreibung oben).