Studenten: Christoph Rosenhauer (2549220)
Nebenläufige Programmierung SoSe 2015 Lukas Schaal (2539218)
Universität des Saarlandes Jens Heinen (2542182)
Milestone 1

1 Allgemeines & Überblick

Die Graphstruktur $n \times m$ soll durch eine variable Anzahl von Threads bearbeitet werden. Dabei kann das von 1 Thread (optional sequentielles Bearbeiten) bis zu m Threads reichen, so dass jede Spalte des Graphs in jeder Iteration von genau einem Thread bearbeitet werden. Zusätzlich werden sog. Synchronisationsgrenzen angegeben, nach wie vielen Iterationsschritten die Akkumulatoren an den Spaltenübergängen auf jeden synchronisiert werden sollen, unabhängig von lokaler Konvergenz. Dieser Parameter soll von außen steuerbar sein.

2 Nebenläufige Probleme

Der Zeitpunkt der Synchronisation soll anhand eines Parameters erfolgen. Falls diese Bedingung erfüllt ist, werden mitels Shared Memory die Daten der Akkumulatoren ausgetauscht. Im Detail wie folgt:

Wir starten mit x Iterationstasks für jede Spalte, die an den Threadpool übergeben werden. Danach rufen wir *shutdown* auf dem Pool auf (er nimmt danach keine neuen Tasks an, führt die übergebenen jedoch weiter aus. Hierbei ist die Reihenfolge der Ausführung prinzipiell egal.) Danach wartet der ThreadPool mit *awaitTermination* als Bedinung einer while-Schleife, bis alle noch laufenden Threads ihren Task beendet haben. Dann wird ein Synchronisationstask an den Threadpool übergeben. An dieser Stelle ist wichtig, dass zum Beginn der Synchronisation zweier (oder mehrerer Spalten) kein Thread mehr einen Iterationstask auf einer der Spalten ausführt. Ein Thread nimmt sich nun die Referenzen auf die Nachbarspalten und verrechnet Akkumulatordaten und aktuelle Werte der Knoten. Hierbei muss darauf geachtet werden, ob die propagierenden Spalten von einem oder mehreren Threads bearbeitet werden. In Abhängigkeit

3 Verwendete Datenstrukur

Der gegebene Graph ist ein $n \times m$ Graph. Dabei wird das Objekt intern auch in m Spalten unterteilt.

Die Anzahl der operierenden Threads wird über einen Parameter gegeben. Ein Thread bekommt dabei einen Task übergeben und bearbeitet nach diesem Task eine oder mehrere Spalten des Graphen. Die Threads werden mittels eines Threadpools / Scheduler den Spalten zugewiesen und bei Bedarf erstellt. Die Tasks beinhalten lokalen Austausch, Synchronisation zwischen mehreren Spalten und globale Konvergenz.

Jede Spalte besteht aus ihren Knoten und Vektoren der links- und rechtsbenachbarten Akkumulatoren. Durch Call-by-Reference kann ein Thread, der gerade Spalte j bearbeitet, auch Daten von Spalte $j-1$ & $j+1$ bekommen. An dieser Stelle sollte durch genaues Locking die Gefahr von Data Races verhindert werden.

Ein Knoten ist ein Objekt, das nur seinen aktuellen Wert kennt. Die Übergangsrate ist in der GraphInfo Klasse enthalten.

4 Konvergenz

4.1 Lokale Konvergenz

Die lokale Konvergenz innerhalb einer Spalte wird nach festgelegten Iterationsschritten (abhängig von Parameter) überprüft. Im Falle einer erkannten lokalen Konvergenz innerhalb einer Spalte wird mit den benachbarten Spalten synchronisiert. Dafür werden der Iterationszustand i von Spalte j und Zustand k von Spalte j+1 angepasst, so dass i = k gilt. Dazu wird die Spalte mit der lokalen Konvergenz so lange iteriert, bis sie auf dem gleichen Stand ist wie die benachbarte Spalte. Falls

nicht, wird nach x Iterationen definitiv synchronisiert, wobei x von einem Parameter abgeleitet wird. Inwiefern x davon abhängt, wird durch Tests ermittelt und im Laufe der Implementierung optimiert.

4.2 Globale Konvergenz

Globale Konvergenz erfolgt implizit durch lokale Konvergenz. Dazu existieren 2 Synchronisationsarten: Eine festgelegt durch den Parameter, eine bei lokaler Konvergenz einer Spalte. Wenn lokale Konvergenz vorliegt, wird das Verhältnis zwischen Inflow und Outflow der betreffenden Spalten überprüft. Wenn der Unterschied zwischen Inflow & Outflow zu groß wird , muss in kleineren Iterationsdistanzen überprüft werden.