MODELLING AND SIMULATION

Fachrichtung 6.2 — Informatik
Universität des Saarlandes
Prof. Dr. Verena Wolf
Thilo Krüger, Dipl. Chem.

Nebenläufige Programmierung (Sommersemester '15)
Blatt Praktisches Projekt

Beachten Sie Folgendes: Wenn Sie Fragen zur Aufgabenstellung haben, benutzen Sie das *mCMS-Forum* zur Diskussion. Außerdem ist unter den Materialien das Video zur 18. Vorlesung verlinkt. In diesem wird die Aufgabenstellung ausführlich erläutert. Zur Lösung bilden Sie bitte eine Zweier- oder Dreiergruppe mit Kommilitonen und/oder Kommilitoninnen und laden die Gruppenzusammensetzung im *mCMS* unter "Projektgruppen" hoch. Laden Sie bitte Ihre Lösung wie gewohnt im *mCMS* unter "Abgaben" hoch. Falls mehrere Mitglieder einer Gruppe eine Abgabe im *mCMS* machen, wird vom betreuenden Tutor nichtdeterministisch eine Abgabe ausgewählt und bewertet.

Regularien

Ihre Lösung der Aufgabe fassen Sie bitte mit allen erforderlichen Dokumenten in ein Archiv im `zip`- oder `tgz`-Format zusammen. Diese können Sie als Abgabe im *mCMS* spätestens am Montag, *17. August, 23:59 Uhr, Meilenstein 2* einreichen.

Meilenstein 2, 17. August 2015

Ihre Lösung muss die folgenden Dokumente umfassen:

- Ihre Implementierung, die in Form von kompilierendem *Java-8*- oder alternativ *Go*-Code verfasst ist. Achten Sie unbedingt darauf, dass Ihr Code angemessen kommentiert und insgesamt selbsterklärend ist.
- Ein Dokument (in pdf, nicht handschriftlich, maximal 2 Seiten – Diagramme und Zeichnungen ausgenommen), in welchem Sie Ihren Lösungsansatz, insbesondere in Hinblick auf die Nebenläufigkeit des eigentlichen Problems, darlegen.
- Falls die Dokumentation des Codes zusammen mit der Implementierung nicht für ein ausreichendes Verständnis der Lösung ausreicht, ist eine persönliche Abnahme des Projekts notwendig. Diese wird individuell mit den Tutoren vereinbart und wird mit allen Gruppenmitgliedern durchgeführt.

Ihre Abgabe wird von einem Tutor begutachtet und bewertet. Folgende Bewertungen Ihrer Abgabe sind möglich:

- Ihre Lösung ist bereits korrekt (sowohl funktional als auch in Sinne der Nebenläufigkeit), dann haben Sie das Projekt bestanden und diesen Teil der Klausurzulassung erhalten. Die Mitglieder der besten Gruppen, die das Projekt an dieser Stelle bestanden haben, erhalten einen Bonus von jeweils 0,3 auf die Endnote.
- Ihre Lösung hat noch funktionale und/oder nebenläufige Fehler, zeigt aber, dass Sie sich ausführlich mit der Aufgabenstellung beschäftigt haben. In diesem Fall wird Ihnen die Möglichkeit eingeräumt nachzuarbeiten und die Fehler zu beseitigen. Details werden nach dem 17. August bekanntgegeben.

- Ihre Lösung zeigt nicht, dass Sie sich ausführlich mit der Aufgabenstellung beschäftigt haben. In diesem Fall haben Sie das Projekt nicht bestanden und damit die Klausurzulassung nicht erhalten. Sie können den Kurs "Nebenläufige Programmierung" dann nicht mehr bestehen.

Meilenstein 1, 20. Juli 2015

Desweiteren ist es für alle verpflichtend, bis zum Montag, 20. Juli einen Entwurf des oben beschriebenen Dokuments einzureichen. Dieser Entwurf muss dokumentieren, dass Sie sich hinreichend mit der Aufgabenstellung beschäftigt haben. Tut sie das nicht, ist der *Meilenstein 1* nicht bestanden, damit dürfen Sie keine Lösung der Aufgabe (Meilenstein 2) mehr abgeben und haben das Projekt und damit den Kurs "Nebenläufige Programmierung" nicht bestanden. Das gilt natürlich auch, wenn Sie die Abgabe überhaupt nicht machen. Zeigt Ihr Entwurf, dass Sie sich hinreichend mit der Aufgabenstellung beschäftigt haben, werden Sie – sofern nötig – an einem anschließenden individuell zu vereinbarenden Besprechungstermin mit Ihrem Tutor teilnehmen. Auch hier gilt: Nehmen Sie (alle Gruppenmitglieder) diesen Termin nicht wahr, ist der *Meilenstein 1* (und damit NP) nicht bestanden.
Wir werden bis zum 21. Juli 2014 wie gewohnt Office-Hours abhalten um Ihre projektrelevanten Fragen zu klären. Falls es Fragen organisatorischer Art gibt, oder es andere nicht inhaltliche Probleme gibt, die sich nicht offensichtlich im Forum klären lassen, können Sie auch gerne eine E-Mail an Thilo schreiben.
Das schlussendliche Bestehen des praktischen Projekts (*Meilenstein 2*) ist notwendige Voraussetzung zum Bestehen der Vorlesung.

Ein Osmose-Prozess

Wir betrachten einen Graphen (V, E), der eine 2-dimensionale Gitterstruktur der Größe $n \times m$ beschreibt, siehe Abbildung 1, links. Jedem Knoten $v \in V$ wird ein Wert zwischen 0 und 1 zugeordnet. Dies kann als ein Schwarz-Weiß-Bild mit $n \times m$ Pixeln gesehen werden, dessen (normierte) Grauwerte bekannt sind.

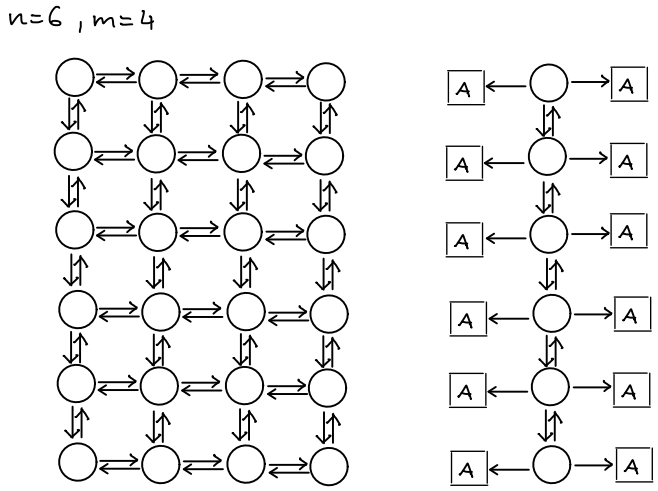

Abbildung 1: Eine 2-dimensionale Gitterstruktur (links) und die möglichen Flüsse in einer verteilten Lösung, wobei "A" für ein Akkumulatorfeld des rechten bzw. linken Nachbarn steht.

Nun starten wir einen dynamischen Osmoseprozess, der iterativ die (Grau-)Werte verändert. In jedem Iterationsschritt "propagiert"jeder Knoten einen gewissen Anteil seines Wertes an die (direkten) Nachbarknoten. Jeder Knoten hat 4 ausgehende Transitionen zu seinen 4 direkten Nachbarknoten (links, rechts, oben, unten), sofern er nicht am Rand liegt. Randknoten haben nur Transitionen zu Nachbarknoten.

Osmoseiteration

Wir nehmen an, dass jede Kante $(v, w) \in E$ des Graphen beschriftet ist mit einer Übergangsrate $q_{vw} \geq 0$, so dass $\sum_{w,w\neq v} q_{vw} \leq 1$ für alle Knoten v gilt. Wir schreiben q_v für die Übergangsrate von v nach v, wobei $q_v := 1 - \sum_{w,w\neq v} q_{vw}$. Sei nun $p_v(i)$ der Wert des Knotens v vor der $i+1$-ten Iteration (der initiale Wert ist $p_v(0)$). Dann ergibt sich $p_v(i+1)$ durch

$$p_v(i+1) = p_v(i) - \sum_{w,w\neq v} p_v(i) \cdot q_{vw} + \sum_{w,w\neq v} p_w(i) \cdot q_{wv},$$

wobei $i \in \{0, 1, \ldots\}$. Für $i \to \infty$ konvergieren die Werte $p_v(i)$ zu einem Limit, das dem finalen Schwarz-Weiß-Bild entspricht.

Beschreibung eines Osmoseprozesses

Um auch sehr große Osmoseprozesse beschreiben zu können, verwenden wir zur Spezifikation sogenannte *guarded commands*. Ein *command* beschreibt eine Menge von Kanten im Graphen und hat die Form

```
guard : rate -> update
```

Sei x die Zeile und y die Spalte eines Knotens in der Gitterstruktur, dann ist `guard` ein Boolscher Ausdruck über x und y (z.B. $x > 0 \wedge y > 0$). Falls dieser zu `true` evaluiert, dann gibt es eine Transition von $v = (x, y)$ zum Knoten `update` mit der Übergangsrate `rate`. Hierbei ist `update` eine Funktion, die mit (x, y) als Argument einen Nachfolgeknoten $w = (x', y')$ berechnet (z.B. $x' = x + 1, y' = y$). Die Übergangsrate `rate` ist ebenfalls eine Funktion die mit (x, y) als Argument eine reelle Zahl $q_{vw} \geq 0$ berechnet (z.B. $c \cdot x \cdot y$, für eine feste positive Zahl c). Achtung: Hier muss sichergestellt sein, dass $\sum_w q_{vw} \leq 1$ ist. Wir werden uns bei der initialen Bedingung $p(0)$ auf solche Vektoren beschränken, deren Einträge allen Knoten den Wert Null zuordnen bis auf einen Knoten, genannt Anfangsknoten, v_0, der den Wert 1 hat.

Dynamische Datenstruktur

Wir nehmen an, dass die Größe der Gitterstruktur es nicht zulässt, dass die Übergangsraten als Matrix im Speicher zur Verfügung steht. Weiterhin ist es möglich, dass manche Knoten niemals einen positiven Wert bekommen. Daher arbeiten wir mit einer dynamischen Datenstruktur für den Graphen, die anfangs nur den Knoten v_0 enthält und in jedem Schritt nur solche Knoten hinzufügt, deren Wert sich von 0 auf eine positive Zahl ändert. (Optional können auch Knoten, deren Wert auf 0 sinkt wieder entfernt werden, bzw. kann der Speicherplatz für neu explorierte Knoten verwendet werden.) Ein Beispiel für eine Datenstruktur, die für das Projekt verwendet werden kann ist in Java die ArrayList.

Sequentielle Lösung

Seien nun gegeben:

- guarded commands
- v_0
- ϵ

und gesucht:

- die Werte $p_v(i)$ für alle $v \in V$ und i.

Wir nehmen Konvergenz an, sobald $||p(i) - p(i+1)||_2 < \epsilon$. Für eine sequentielle Lösung definieren wir eine Datenstruktur `Knoten` mit den Einträgen `double Wert`, `double Akkumulator` und vier Referenzen auf die möglichen Nachfolgerknoten. Der folgende Algorithmus liefert eine sequentielle Lösung:

```
while (not_converged)
  for all v in V
     for all w with (v,w) in E
        w.Akku=w.Akku+v.Wert*q_{vw};
        v.Akku=v.Akku-v.Wert*q_{vw};
     endfor
  endfor
  for all v in V
     v.Wert = v.Wert + v.Akku;
     v.Akku = 0;
  endfor
endwhile
```

Aufgabenstellung

Unterteilen Sie die Gitterstruktur in m Spalten und lassen Sie einen Thread nur maximal den flow einer Spalte der Gitterstruktur zur gleichen Zeit bearbeiten. Entwerfen Sie zum Beispiel eine Datenstruktur, in der Sie Akkumulatorfelder für jeden horizontalen Nachbarn anlegen. Lassen Sie die Werte einer Spalte für eine bestimmte Anzahl von internen/lokalen Iterationsschritten propagieren - inklusive dem "Befüllen" der Akkumulatorfelder, die zu horizontalen Nachbarn (also anderen Spalten) gehören. Geben Sie erst nach dieser bestimmten Anzahl von lokalen Iterationsschritten den outflow zu den anderen Spalten weiter (z.B. nach 100 Schritten.) Dies resultiert in einer Approximation der Lösung, die je nach Dynamik des Prozesses und der gewählten lokalen Anzahl von Iterationen variiert.[1]
Betrachten Sie Abbildung 1, rechts: Angenommen in Spalte j wird während einer Reihe von lokalen Iterationen für einen Knoten v der Nachbarspalte $j+1$ der Anteil x an inflow akkumuliert. Jetzt muss synchronisiert werden: Sei i die Gesamtzahl von Iterationen bzgl. Spalte j und k die Gesamtzahl von Iterationen bzgl. Spalte $j+1$. Dann sollte der flow x nur dann zum Wert des Knotens v addiert werden, wenn $i = k$. Betrachten Sie die folgenden Probleme im Zusammenhang mit Ihrer Implementierung:

- Propagieren zwischen Spalten: Stellen Sie sicher, dass die Übergabe von Masse, die horizontal propagiert wird und daher von einem Thread an einen anderen übergeben werden muss, korrekt gehandhabt wird. Hierbei können Sie sich entscheiden, ob Sie message passing oder gemeinsame Variablen verwenden. Im letzteren Fall sind Data Races zu vermeiden.

- Synchronisieren beim Propagieren: Überlegen Sie sich, wie die Bedingung $i = k$ (siehe oben) sichergestellt werden kann. Ist es möglich ohne eine zentrale Instanz auszukommen, die die Anzahl der lokalen Iterationen synchronisiert.

- Lokale Konvergenz: Versuchen Sie, lokale Konvergenz zu erkennen, um nicht unnötig bzgl. der vertikalen Transitionen zu iterieren. Beachten Sie, dass lokale Equilibria durch ein Propagieren über die horizontalen Transitionen perturbiert werden.

- Terminierung und globale Konvergenz: Überlegen Sie sich eine Lösung, um globale Konvergenz zu erkennen und die Terminierung aller Threads sicherzustellen.

Achtung: Es kann zu divergentem/zyklischen Verhalten kommen, wenn nur selten horizontal propagiert wird. Um dies zu verhindern, ist es sinnvoll, die horizontal propagierten Werte zu überwachen.
Falls für den gesamten horizontalen "Flow" zwischen zwei Spalten gilt

$$inflow \approx outflow$$

dann sollte nach jedem (internen) Schritt horizontal propagiert werden und auf globale Konvergenz überprüft werden.

[1] Achtung: eine gute Approximation reicht aus (tradeoff zwischen Effizienz und Genauigkeit sollte mit Parametern steuerbar sein!)

Für die Lösung des Projekts stellen wir Ihnen einen Parser für die guarded commands (sowohl für Go, als auch für Java) bereit. Desweiteren wird Ihnen die Möglichkeit gegeben, Ihre Ergebnisse so auszugeben, dass sie als Schwarzweißbild geplottet werden können.

Bonus

Hier ein paar sehr kurze Stichpunkte darüber, was helfen könnte, einen Bonus zu bekommen:

- Angenommen, Ihre Implementierung hat die korrekte Semantik im Hinblick auf die berechneten Ergebnisse und auf mögliche Nebenläufigkeits-Probleme.
- Beeindrucken Sie uns damit, wie sie es gelöst haben:
- Elegant!!
- Effizient (schnell, geringer Speicherbedarf, skaliert)
- Genau (ohne langsam zu sein)