Densely Connected Biclusters

Stefan Dresselhaus, Thomas Pajenkamp

Parallele Algorithmen und Datenverarbeitung
Programmierprojekt im Wintersemester 2013/14

Universität Bielefeld – Technische Fakultät

18. April 2014

Inhaltsverzeichnis

Todo list

1 Zielsetzung des Projekts

Im Rahmen dieses Programmierprojekts wurde ein Programm entworfen und entwickelt, um Densely Connected Biclusters, im weiteren DCB, in einem biologischen Netzwerk zu ermitteln. Bei DCB handelt es sich um Teilgraphen eines Netzwerks, dessen Knoten untereinander hoch vernetzt sind und Objekte mit ähnlichen Eigenschaften repräsentieren.
Die Suche nach DCB ist ein NP-schweres Problem. Da mit einem geeigneten Algorithmus jedoch voneinander unabhängige Lösungspfade einzeln verfolgt werden können, ist das Problem gut für eine parallele Berechnung geeignet, wodurch die Gesamtlaufzeit stark reduziert werden kann. Zudem sind real verwendete biologische Netze üblicherweise nur schwach vernetzt. Daher können durch die Forderung nach einer hohen Konnektivität der DCB viele Lösungskandidaten schnell ausgeschlossen werden und die schlimmstenfalls nichtdeterministisch polynomielle Laufzeit findet kaum Anwendung.

1.1 Densely Connected Biclusters

Ausgangsbasis ist ein ungerichteter ungewichteter Graph $G = (V, E)$, dessen Knoten $n \in V$ mit jeweils p Attributen versehen sind. Jedem Knoten n ist zu jedem Attribut i ein numerischer Wert a_{ni} zugewiesen.
Ein DCB $D_k = (V_k, E_k)$ ist ein Teilgraph von G, der durch die Parameter $\alpha \in [0, 1]$, $\delta \in \mathbb{N}$ und $\omega \in \mathbb{R}^p$ beschränkt wird und die folgende drei Eigenschaften erfüllt.

- Der Teilgraph ist zusammenhängend.
- Die Dichte des Teilgraphen unterschreitet einen Schwellenwert α nicht, also $\frac{2 \cdot |E_k|}{|V_k|(|V_k|-1)} \geq \alpha$.
- Für mindestens δ Attribute liegen die Werte aller Knoten des Teilgraphen höchstens ω_i auseinander. Anders ausgedrückt

$$\delta \leq \left| \left\{ 1 \leq i \leq p \;\middle|\; \omega_k \geq \left(\max_{n \in V_k} a_{ni} - \min_{n \in V_k} a_{ni} \right) \right\} \right|.$$

2 Wahl der Programmiersprache und Konzept der Parallelisierung

Die Wahl der Programmiersprache zur Verwirklichung des Projekts beeinflusst stark die Methoden der Programmierung und die Art der Parallelisierung. Klassischerweise werden für sequentielle und parallele Programme gleichermaßen imperative Sprachen wie C(++), Fortran oder auch Java verwendet, wofür Erweiterungen zur parallelen Programmierung existieren oder einige Werkzeuge direkt in die Sprache eingebaut sind. Bekannte Ansätze hierfür sind MPI und openMP.
Unser Projekt geht in eine etwas andere Richtung. Bei imperativer Programmierung muss ein großes Augenmerk auf die Vermeidung unerwünschter wechselseitiger Beeinflussungen verschiedener Threads und Prozesse gelegt werden, die fehlerhafte Rechenergebnisse zur Folge haben. Außerdem muss bei der Thread-/Prozesskommunikation immer die Gefahr von Verklemmungen beachtet werden, die schlimmstenfalls zu einem kompletten Stillstand der Programmausführung führen. Beide Probleme sind schwierig zu detektieren und zu lokalisieren.
Die genannten klassischen Probleme des Parallelrechnens können mit pur funktionaler Programmierung gut vermieden werden. Nebenbedingungen treten in pur funktionalem Programmcode (einen korrekten Compiler/Interpreter vorausgesetzt) garantiert nicht auf. Da das DCB-Problem bis auf das Einlesen der Eingabedaten und die Ausgabe pur funktional realisierbar ist, ist

es optimal für eine derartige Implementierung geeignet Die konkrete Wahl der funktionalen Programmiersprache fiel auf *Haskell*.

Für Haskell wurden Bibliotheken entwickelt, die eine einfache und effiziente Programmierung paralleler Programme erlauben. Wir verwenden das Paket *parallel* in Verbindung mit *repa*-Arrays. Durch *parallel* können geeignete Algorithmen mit wenig Aufwand, aufgeteilt werden. Dabei werden Funktionsaufrufe unevaluiert in einem Array gespeichert und dort von freien Threads abgearbeitet. Diese Technik nennt man Work-Stealing und die noch nicht ausgewerteten Funktionen werden in Haskell *Sparks* genannt. Man kann sich dies als einen auf den Funktionsaufruf beschränkten light-weight Thread vorstellen – mit weniger Overhead. Die *repa*-Arrays bieten Funktionen, um die einzelnen Elemente eines Arrays parallel zu berechnen. Mit diesen Techniken lässt sich sequentieller Programmcode einfach parallelisieren, da hierfür nur wenige Änderungen im Programmcode erforderlich sind. Es müssen lediglich die Berechnungsfunktionen an die parallelisierende Funktion übergeben und die Funktion zur Auswertung der Arrayelemente ausgetauscht werden.

Zwei wichtige Punkte müssen dennoch beachtet werden. Zum einen verwendet Haskell das Konzept *Lazy Evaluation*. Befehle werden immer nur soweit berechnet, wie sie an anderer Stelle benötigt werden. Dadurch entstehen manchmal zur Laufzeit große Bäume nur teilweise ausgewerteter Befehle, welche die Ausführungszeit durch eine hohe Garbage-Collector-Auslastung stark negativ beeinflussen. Es muss demnach darauf geachtet werden, die Berechnung später ohnehin erforderlicher Funktionen frühzeitig zu erzwingen. Zum anderen ist die Anzahl der Sparks standardmäßig nicht begrenzt, sodass auch hier zu große Arrays entstehen können, deren Abarbeitung allerdings im Verlaufe des Programms durch o. g. Lazy Evaluation evtl. gar nicht erforderlich ist. Daher beschränken wir die Anzahl der möglichen Sparks (und somit der maximal möglichen Worker-Threads) auf 1000. Erwähnenswert ist noch, dass diese Technik *nicht* von Hyper-Threading profitiert,da kein Kontextwechsel der Threads nötig ist, und wir somit 1000 „echte“ Kerne für eine maximale Auslastung benötigen. Die obere Grenze wird eher durch Amdahls Gesetz, denn durch die verfügbaren Kerne beschränkt.

3 Der Algorithmus

Für die Darstellung des Eingabegraphen, in dem die DCB gesucht werden, verwenden wir eine Adjazenzmatrix. Die anschließende Laufzeitbetrachtung bezieht sich auf diese Datenstruktur.
Der DCB-Algorithmus besteht aus einer Vorverarbeitungsphase, in der Cluster-Seeds aus 2 jeweils verbundenen Knoten generiert werden, und einer anschließenden Expansion dieser Seeds unter Berücksichtigung der in Abschnitt 1.1 vorgestellten Nebenbedingungen (Constraints). Diese Cluster (im Folgenden Graphen genannt) bestehen zu Anfang aus genau 2 Knoten, die sämtliche Bedingungen erfüllen. Eine erste Optimierung findet nun statt, da es nach diesem Schritt auch verbundene Knoten geben kann, die nicht zur initialen Bildung der Graphen beigetragen haben. Diese Paare können im Folgenden komplett ausgeschlossen werden, da sie die Attributsbedingung nicht erfüllen können[1]. Folglich würde jeder Graph mit diesem Knotenpaar insgesamt auch gegen selbige Attributsbedingung verstoßen. Alle hiervon betroffenen Kanten können somit aus der Adjazenzmatrix gelöscht werden.
Wir exportieren 2 Funktionen nach außen, die in der Lage sind, den Graphen zu expandieren: `step` und `maxDCB`. `step` liefert alle möglichen expandierten Graphen aus einer Liste von bestehenden DCB – allerdings verliert man somit alle Graphen, die nicht expandiert werden konnten. Als Ergebnis hat man eine gewisse Mindestzahl an Knoten im Graphen. Da die Seeds mit 2 Knoten

[1] Ein Graph aus 2 verbundenen Knoten ist immer maximal dicht und zusammenhängend.

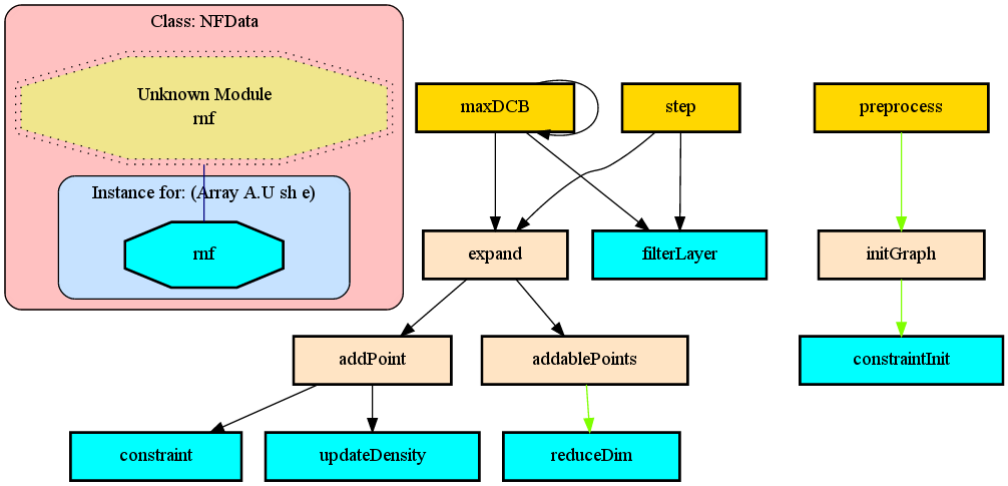

Abbildung 1: Übersicht über die Hierarchie des DCB-Moduls. Gelb hinterlegte Funktionen sind nach außen hin sichtbar.

beginnen, man aber z. B. alle DCB mit 4 Knoten oder mehr haben möchte, kann man `step` so häufig aufrufen, dass alle Graphen mit weniger Knoten gar nicht zurückgegeben werden. Wir verwenden dies in unserem Algorithmus einmalig, da wir nur Cluster mit 3 oder mehr Knoten zurückliefern.

Die Funktion `maxDCB` übernimmt die eigentliche Berechnung, sodass wir diese im Detail besprechen. Zunächst jedoch geben wir einen kleinen Überblick über die Hilfsfunktionen im Hintergrund:

filterLayer filtert eine Menge von s Graphen der Größe m, indem es Duplikate herausfiltert. Laufzeit: $\mathcal{O}(m \cdot s \log s)$.

constraint überprüft, ob der Graph noch die Constraints erfüllt und wenn ja, wie diese aussehen.
Laufzeit: $\mathcal{O}(k)$ bei k Attributen, da die Maximal- und Minimalwerte des Graphen tabelliert werden.

updateDensity errechnet die Änderung der Dichte des Graphen anhand des hinzuzufügenden Punktes.
Laufzeit: $\mathcal{O}(m)$ bei m Knoten im Ursprungsgraph, da die Graphendichten gespeichert werden.

reduceDim ist eine interne Hilfsfunktion, die eine Dimension einer Array-Shape verwirft.
Laufzeit: $\mathcal{O}(1)$.

addablePoints traversiert die Adjazenzmatrix und liefert alle mit einem Graphen verbundenen Knoten, die nicht selbst im Graphen enthalten sind.
Laufzeit: $\mathcal{O}(n \cdot m)$ bei m Knoten im Ursprungsgraphen und einer Adjazenzmatrix $n \times n$.

addPoint erweitert wenn möglich einen bestehenden Graphen um einen Knoten, indem es zunächst ein `updateDensity` durchführt und anschließend mittels `constraint`

alle Nebenbedingungen überprüft.
Laufzeit: $\mathcal{O}(k+m)$, falls die Density-Constraint erfüllt bleibt, sonst $\mathcal{O}(k)$.

expand wendet `addPoint` auf alle Ergebnisse von `addablePoints` an.
Laufzeit: $\mathcal{O}(n \cdot m \cdot (k+m))$ im worst-case einer voll besetzten Adjazenzmatrix. In der Praxis sind die Eingabegraphen kaum vernetzt und die Attributzahl k ist klein, sodass sich die average-case-Laufzeit unter Berücksichtigung der alternativen Laufzeit von `addPoint` an $\mathcal{O}(n \cdot m \cdot k) \approx \mathcal{O}(n \cdot m)$ annähert.

```
1  -- | Calculates all maximum DCB. A maximum DCB is a densely connected bicluster that
2  --   cannot be expanded by any additional node without breaking the constraints.
3  --
4  --   This function does return the seed graphs themselves if they cannot be expanded.
5  maxDCB :: [Graph] -> Adj -> Attr -> Density -> MaxDivergence -> Int -> [Graph]
6  maxDCB [] _ _ _ _ _ = []
7  maxDCB gs adj attr dens maxDiv minHit =
8      let next = L.map (expand adj attr dens maxDiv minHit) gs
9                       +|| (parBuffer 1000 rdeepseq)
10         (maximal, expandable) = part (\_ rm -> rm == []) (zip gs next)
11         expandable' = filterLayer $ concat expandable
12         -- Divide solutions into expandable solutions and maximum solutions.
13         -- Expandable solutions yield a result via the 'expand' function.
14     in maxDCB expandable' adj attr dens maxDiv minHit L.++ maximal
15     -- append maximum solutions of prospective function calls and maximum solutions
16     -- of this iteration
```

Listing 1: Die maxDCB-Funktion

Der rekursive Funktionsaufruf findet in Zeile 14 statt. Hier werden iterativ alle expandierbaren Möglichkeiten evaluiert bis sie maximal erweitert sind und an die nicht erweiterbaren Graphen angehängt. In den Zeilen 8/9 wird die Expansion auf den Eingabegraphen `gs` parallel in einem Puffer von höchstens 1000 parallelen Anweisungen ausgeführt. Die Strategie, welche wir für die parallele Evaluation verwenden, lautet `rdeepseq`. Dadurch werden diese Graphen direkt vollständig ausgewertet und müssen nicht (z. B. durch Lazy Evaluation) nachberechnet werden.

Anschließend partitionieren wir die expandierten Graphen in maximal erweiterte und in weiter expandierbare (Z. 10). Letztere filtern wir noch (Z. 11) nach Duplikaten, um redundante Weiterberechnung (und damit einen erhöhten Rechenaufwand) zu vermeiden. Zurückgeliefert werden somit alle Graphen, die maximal expandiert sind.

Die Funktion `expand` wird letztendlich für jeden Graphen genau einmal aufgerufen. Der Rechenaufwand der m-ten Expansionsstufe mit s Graphen ist zusammen mit der Filterung doppelter Graphen $\mathcal{O}(sm \cdot (n(k+m) + \log s))$, für schwach vernetzte Eingabegraphen eher $\mathcal{O}(sm \cdot (nk + \log s))$. k ist die Anzahl an Attributen und n die Größe der Adjazenzmatrix. Allerdings wächst die Anzahl der Graphen pro Iteration im ungünstigsten Fall exponentiell an, woraus sich die Schwierigkeit des Problem als NP-schwer ergibt. In schwach vernetzten Eingabegraphen ist jedoch zu erwarten, dass die anfänglichen Seed-Graphen kaum erweiterbar sind, wodurch sich der gesamte Rechenaufwand stark reduziert. Dennoch besteht viel Potential zur Parallelisierung der Berechnung zur Verkürzung der Rechenzeit.

4 Ausführung und Auswertung

Im folgenden Abschnitt gehen wir genauer auf die verwendeten Compileroptionen und den durchgeführten Benchmark ein, sowie eine Auswertung der dadurch angefallenen Daten.

Tabelle 1: Speedup nach Amdahl und tatsächliche Messung

Kerne	Speedup	Erreicht
1	1	1
2	1.959	1.965
3	2.878	2.839
4	3.761	3.629

4.1 Compileroptionen

Als Compileroptionen sind in der mitgelieferten .cabal-Datei folgende Angaben eingestellt:
`ghc-options: -Odph -rtsopts -threaded -fno-liberate-case -funfolding-use-threshold1000 -funfolding-keeness-factor1000 -optlo-O3 -fllvm` Hierbei stehen die einzelnen Flags für

`-Odph`	maximale GHC-Optimierung
`-rtsopts`	Runtime-Optionen (+RTS -Nx -l -s etc.)
`-threaded`	Multithreading
`-fno-liberate-case`	Code-duplizierung jenseits von -O2 vermeiden
`-funfolding-use-threshold1000`	Analogon zu `#pragma unroll 1000` in C++, wo möglich.
`-funfolding-keeness-factor1000`	empfohlen für besseres Unfolding
`-fllvm`	Auf llvm kompilieren statt direkt Maschienencode zu erzeugen
`-optlo-O3`	llvm-Compiler mit -O3 starten

Insbesondere das Unfolding der Funktionen und das Weiterreichen des Codes an LLVM bringt einen extremen Performance-Zugewinn. LLVM kann hier auf die jeweils benutzte Architektur weiter optimiren.

4.2 Garbage-Collector-Optimierung

TODO machen!

4.3 Laufzeit und Amdahls Gesetz

Wir haben den Test mit einer bereigestellten 4000x4000-Matrix (sparse, 80000 Einträge) insgesamt 10x für jede Konfiguration (1,2,3 oder 4 Kerne) durchrechnen lassen. Dies ist in Abbildung 2 zu sehen. Die Varianz war mit $< 0.003s$ zu gering um sinnvoll eingezeichnet zu werden. Wir haben das Programm in 2 Teile unterteilt. Zum einen das Einlesen, welches Single-Threaded jeweils $0.9447 \pm 2e-5s$ im Single-Threaded und $\approx 1.05s$ im Multithreading-Fall benötigt hat. Bei einer Single-Thread-Laufzeit von $44.6581 \pm 2.30e-2s$ für den Rest des Programms, ergibt sich nach Amdahl die in Tabelle 1 Minimallaufzeit für das gesamte Programm.
Man muss hierbei berücksichtigen, dass Amdahl Effekte, wie Supralinearität nicht berücksichtigt, welche in der Realität zwar auftreten können, aber nicht müssen. Dies fällt bei uns im Fall von 2 Kernen auf, wo wir leicht über der Schätzung nach Amdahl liegen.

Insgesamt lässt sich hierbei sehen, dass wir fast immer gleichauf mit Amdahls Gesetzt liegen und somit im ideal zu erwartenden Bereich der Parallelisierung.

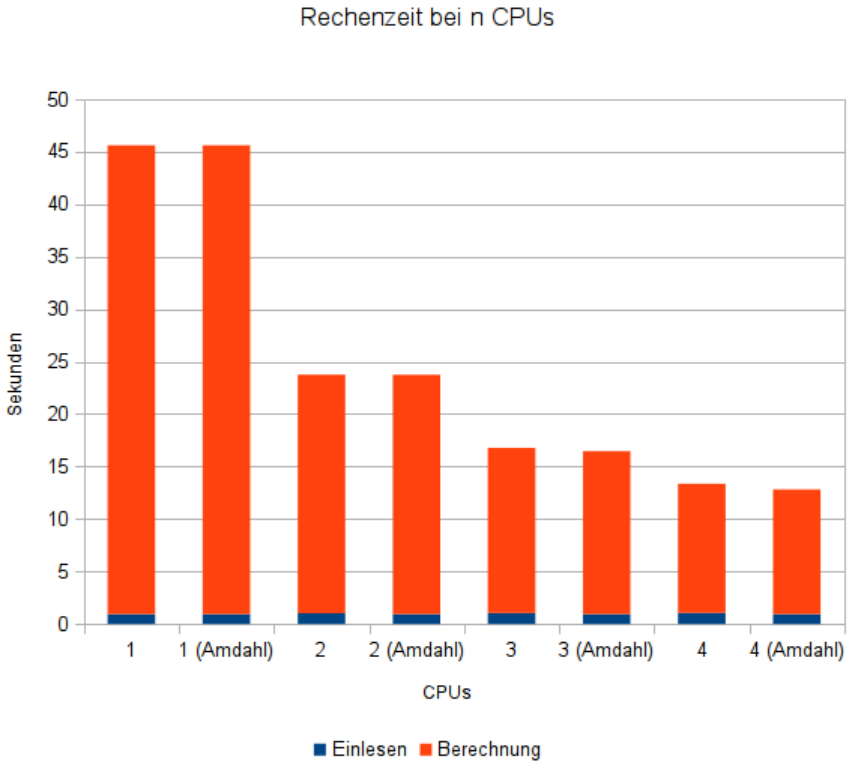

Abbildung 2: Graphische Darstellung der Benchmark-Auswertung. Dieser Benchmark wurde auf einer 4-Kern-Maschiene (i7-2600) gemacht, sodass die Laufzeit bei 4 Kernen nicht optimal ist, da durch Hintergrundaufgaben ca. 5% CPU-Last auf einem Kern lasteten.

5 Fazit

Noch optimierbar: GC-Nutzung (Threadscope hat einschnitte) Nicht optimierbar: Wechsel zwischen den Generationen