

# Densely Connected Biclusters

Stefan Dresselhaus, Thomas Pajenkamp

Parallele Algorithmen und Datenverarbeitung  
Programmierprojekt im Wintersemester 2013/14

Universität Bielefeld – Technische Fakultät

6. März 2014

# Inhaltsverzeichnis

<b>1 Zielsetzung des Projekts</b>	<b>2</b>
1.1 Densely Connected Biclusters . . . . .	2
<b>2 Wahl der Programmiersprache und Konzept der Parallelisierung</b>	<b>3</b>
<b>3 Der Algorithmus</b>	<b>3</b>
<b>4 Ausführung und Auswertung</b>	<b>3</b>
<b>5 Fazit</b>	<b>4</b>
<b>Literatur</b>	<b>5</b>

## 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  $\alpha$  nicht, also  $\frac{2 \cdot |E_k|}{|V_k|(|V_k| - 1)} \geq \alpha$ .
- Für mindestens  $\delta$  Attribute liegen die Werte aller Knoten des Teilgraphen höchstens  $\omega_i$  auseinander. Anders ausgedrückt

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

## 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. Nebenwirkungen 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 Datenstrukturen automatisch in kleine Teilprobleme, auch *Sparks* genannt, aufgeteilt und von freien Threads abgearbeitet werden. Die *repa*-Arrays bieten Funktionen, um die einzelnen Arrayelemente parallel zu berechnen. Damit lässt sich sequentieller Programmcode einfach parallelisieren, da hierfür nur wenige Änderungen 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 stark negativ beeinflussen. Es muss demnach darauf geachtet werden, die Berechnung später ohnehin erforderlicher Funktionen zu erzwingen. Zum anderen ist die Anzahl der Sparks standardmäßig nicht begrenzt. Dadurch kann es vorkommen, dass neue Sparks schneller generiert werden, als sie abgearbeitet werden. Deshalb ist es sinnvoll, die Anzahl zeitgleich existierender Sparks im Programm zu beschränken.

## 3 Der Algorithmus

An bereits erstelltem Pseudocode orientieren, eventuell anpassen an Details der Programmierung zur besseren Effizienz.

## 4 Ausführung und Auswertung

Amdahls Gesetz, Minskys Vermutung

Nach jedem Erweiterungsschritt: Sammeln und Aufgaben neu verteilen → Kommunikation

## **5 Fazit**

Wir sind toll.

## Literatur

[Dum29] Dummy. „Dummy. Dummy“. In: *Dummy* (26. Okt. 1929).