Densely Connected Biclusters

Stefan Dresselhaus, Thomas Pajenkamp

Parallele Algorithmen und Datenverarbeitung
Programmierprojekt im Wintersemester 2013/14

Universität Bielefeld – Technische Fakultät

11. März 2014

Inhaltsverzeichnis

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. Diese Funktionsaufrufe werden unevaluiert in ein Array gepackt 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 Arrayelemente parallel zu berechnen. Mit diesen Techniken 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 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 nichtmal mehr ein Kontextwechsel der Threads nötig ist) und wir somit 1000 „echte“ Kerne für eine maximale Auslastung benötigen. Dies sollte für den Moment ausreichen.

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.