Vorlesung

Cloud- und Web-Technologien

(CS3140)

Knotenzentrische Algorithmen mit Flink

Professor Dr. rer. nat. habil. Sven Groppe
https://www.ifis.uni-luebeck.de/index.php?id=groppe

Chronologische Übersicht über die Themen

Nr	Thema	
1	Einleitung	
2	Einführung in das Semantic Web, RDF und SPARQL	 Datenmodell
3	Die Semantic Web-Ontologiesprachen RDFS und OWL	
4	Multiplattform-Entwicklung mit Kotlin	 Multiplattform
5	Fortgeschrittene Themen mit Kotlin	
6	Einstieg in Cloud Computing, Hadoop	 Backend
7	Operatoren der relationalen Algebra in Hadoop	
8	Datenverarbeitung mit Pig	
9	Einführung in Spark und Flink	
10	Stromverarbeitung mit Flink	
11	**Knotenzentrische Algorithmen mit Flink**	
12	HTML und CSS	Web
13	Browserprogrammierung mit JS/JQuery und Serverprogrammierung mit PHP Hypertext Preprocessor	
14	Zusammenfassung und Ausblick	

Graphberechnungen

- in vielen Frameworks langwierig
  - geringe Lokalität von Speicherzugriffen
  - sehr wenige Arbeitsschritte pro Knoten
  - sich verändernder Parallelitätsgrad
  - **verteilte Berechnungen über mehrere Rechner verschlimmert das Problem!**

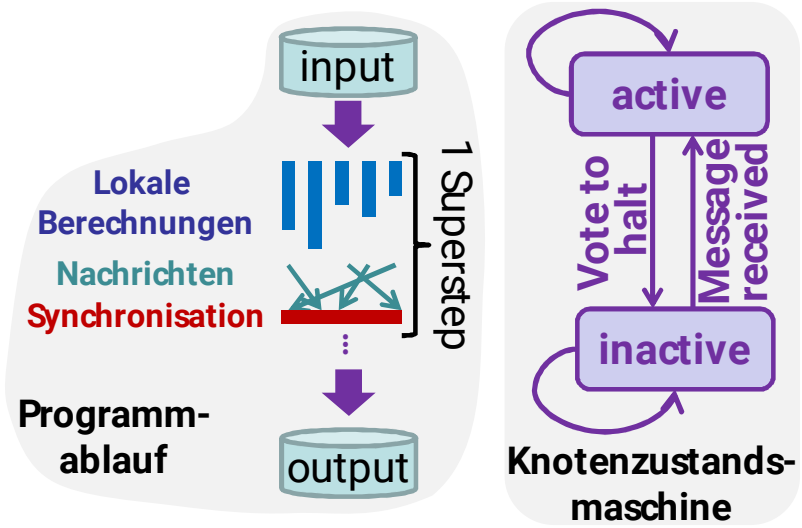
Alternative Möglichkeiten zur Graphberechnung

- Single-Computer-Graphbibliothek
  - nicht skalierbar
- MapReduce
  - ineffizient: Speicherung des Graphzustandes in jedem Verarbeitungsschritt

    $\rightarrow$ zu viel Kommunikation notwendig
- **besser:** speziell für Graphen entwickeltes verteiltes Programmiermodell und verteilte Graphverarbeitungsengine

„Think like a vertex!“ (Pregel[1]-ähnliche Systeme)

- Algorithmus
  „aus Sicht eines Knotens“
  - Eigener Knotenwert kann verändert werden
  - Nachrichten können an andere (beliebige) Knoten (mit bekannter ID) versendet und empfangen werden



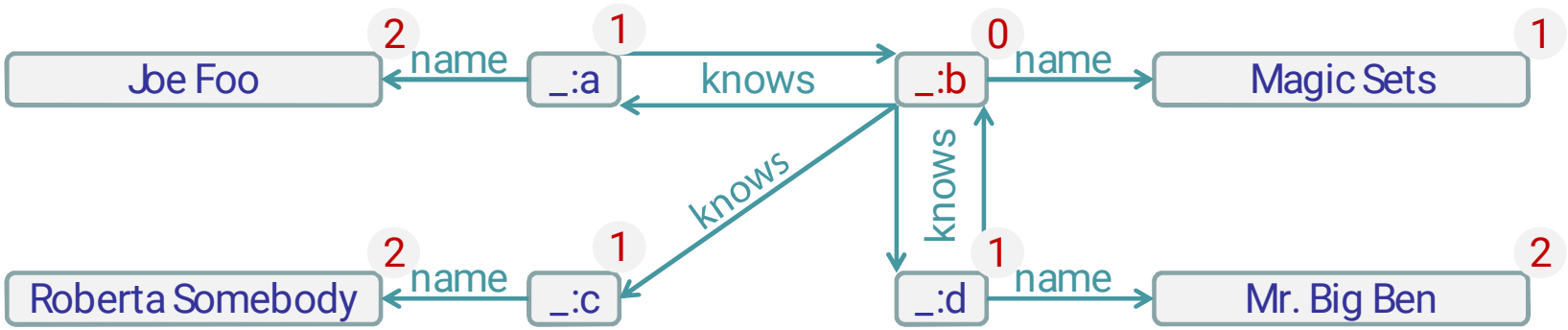
- Wiederholung bis keine Änderungen/Nachrichten oder maximale Iterationsanzahl erreicht
- Motivation
  - Alle Programme dieses Programmiermodells sind hoch parallelisierbar (⇝ Verteilung der Knotenberechnungen)

[1] G. Malewicz et al.: **Pregel: a system for large-scale graph processing**. In SIGMOD, 2010.

Apache Flink/Gelly

- Unterstützung mehrerer Programmiermodelle für Graphberechnungen
  - Am Allgemeinsten: **Knotenzentrisch (Pregel)**
  - aber Ausführung anderer (eingeschränkter) Programmiermodelle (Scatter-Gather/Gather-Sum-Apply) evtl. performanter
    - Einschränkungen z.B.
      - Nachrichten können nur zu Nachbarn versendet werden
      - Trennung der Knotenwertaktualisierung vom Nachrichtenversand

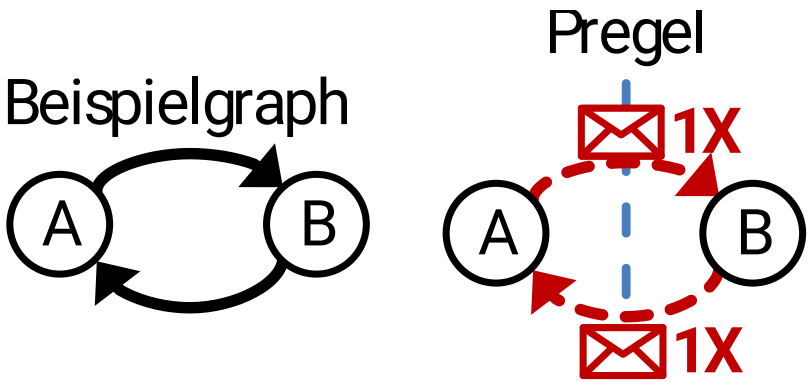
Single-Source-Shortest-Path
als Pregel-Programm



Verteilte Graphverarbeitungsframeworks

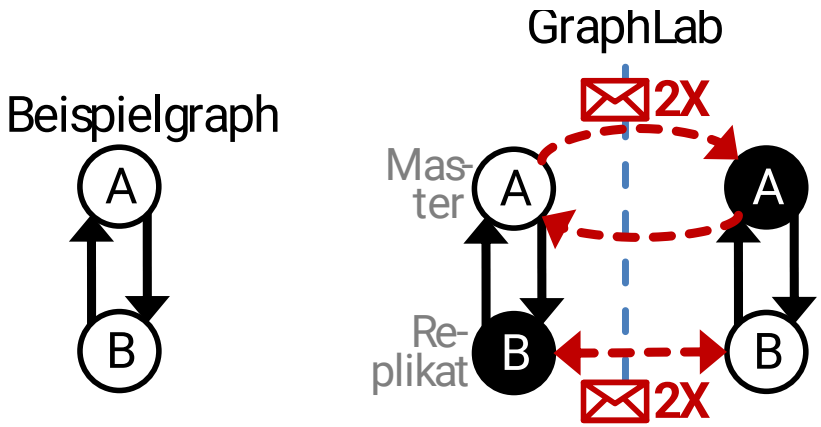
- Pregel (C++)
  - Ursprung der Idee der knotenzentrischen Algorithmen
- GraphLab (C++)
- PowerGraph (C++)
- PowerLyra (C++)
  - basiert auf PowerGraph (mit weiteren hybriden Partitionierungsstrategien)
- GraphX (Scala/Java JVM)
  - verwendet Spark zur verteilten Verarbeitung von Graphen
- Gelly (Java JVM)
  - verwendet Apache Flink zur verteilten Verarbeitung von Graphen

Kantenschnitt für das Partitionieren von Graphen - Pregel



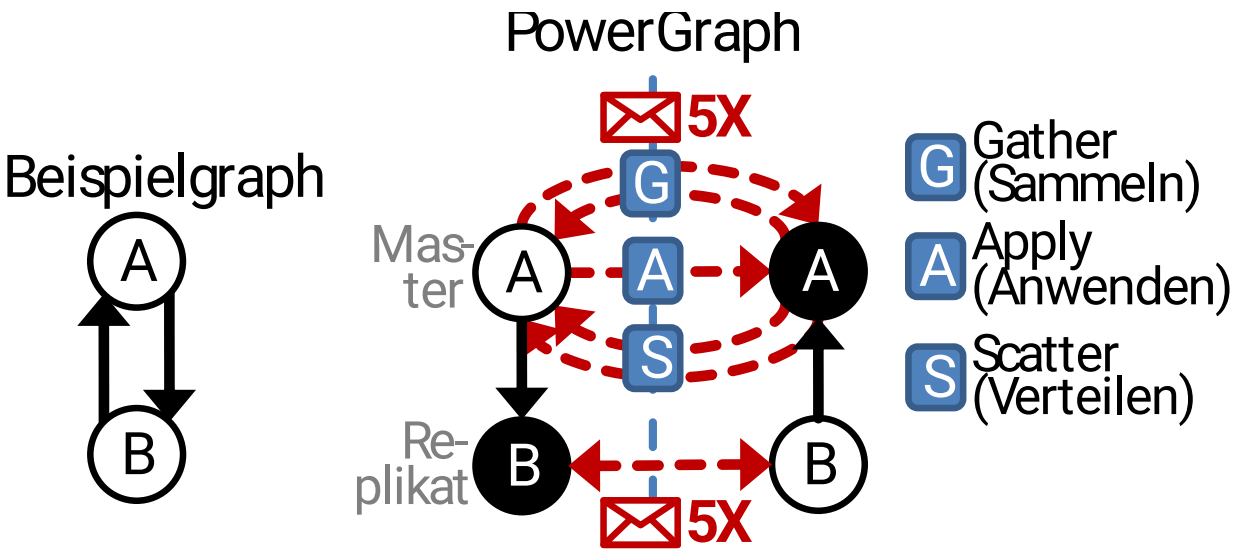
- Kantenschnitt mit Verteilung der Knoten auf die Rechner
- Kommunikation per Nachrichten entlang der Kanten

Kantenschnitt für das Partitionieren von Graphen - GraphLab



- Kantenschnitt mit Verteilung der Knoten auf die Rechner, aber Replikate der Knoten und Kanten in beiden Rechnern (Kantenstart-/ende)
- Kommunikation zwischen Replikate und Master für Updates

Knotenschnitt für das Partitionieren von Graphen - Powergraph/PowerLyra



- Knotenschnitt: Zusätzliche Knotenreplikate für die Kanten für parallele Knotenverarbeitung

Knotenschnitt für das Partitionieren von Graphen - GraphX/Gelly

- Erweiterung des allgemeinen Datenflussframeworks in Spark
  - Umwandlung von graph-spezifischen Operationen zu Operationen der Basis-Engine wie etwa Join, Map und Group-By
  - Knotenreplikation und Partitionierung basierend auf Knotenschnitt zum Balanzieren des Workloads
- Gelly basierend auf Flink
  - Umwandlung von graph-spezifischen Operationen zu Operationen der Basis-Engine wie etwa Join, Reduce, Map und coGroup

Partitionierungsstrategien

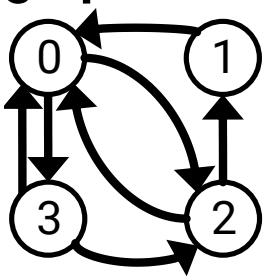
- Zuordnung der Knoten und/oder Kanten des zu verarbeitenden Graphen zu Rechner
- Ziele:
  - Gleichmäßige Verteilung
  - Effiziente Verarbeitung mit geringen Kommunikationskosten

Partitionierungsstrategie "Random"

- Hashwert der Kante bestimmt Rechner
  - 2 Varianten
    - Richtung der Kante spielt Rolle bei der Zuordnung
    - kanonisch: Richtung der Kante spielt keine Rolle, d.h. Zuordnung von (u, v) und (v, u) zum selben Rechner
- Vorteile
  - schnelle Berechnung des zugeordneten Rechners
  - gleichmäßige Verteilung der Kanten
  - hoch parallelisierbar
- Unterstützung von folgenden Graphdatenbanken
  - Powergraph (kanonische Variante)
  - GraphX (beide Varianten)

Part.-strategie "Random" - Beispiel

Beispiel-graph:



Kante (u, v)	Rechner kanonisch $h(u,v) = (u+v)\ mod\ 3$	Rechner nicht kanonisch $h(u,v) = (u+5\times v)\ mod\ 3$
$(1,0)$	1	1
$(0,2)$	2	1
$(2,0)$	2	2
$(2,1)$	0	1
$(3,0)$	0	0
$(0,3)$	0	0
$(3,2)$	2	1

Part.-strategie "Oblivious" (Powergraph) 1/3

- "Greedy" Heuristik, um Kanten zu platzieren
  - aber den Replikationsfaktor (d.h. Anzahl Rechner auf denen Kanten eines bestimmten Knotens gespeichert sind) zu minimieren
- Nächste Platzierung einer Kante basiert auf vorherigen Platzierungen (vorgenommen durch denselben Rechner)
  - keine Kommunikation zwischen Rechnern über vorherige Platzierungen notwendig (daher "oblivious"/nichtsahnend)
  - Verteilung nicht so schnell berechenbar wie bei Random

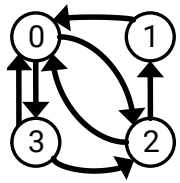
Part.-strategie "Oblivious" (Powergraph) 2/3

- Sei $A(u)$ die Menge an Rechnern, auf denen u repliziert ist
  - durch Platzierungen des betrachteten Rechners unabhängig von anderen verteilenden Rechnern
- Platzierung einer Kante (u, v) (nach i Schritten zufälliger Verteilung)
  - mit dem Ziel der Minimierung von $|A(u)| + |A(v)|$
    - Fall 1: $A(u) \cap A(v) \neq \phi$, d.h. auf einen Rechner sind bereits Replikate von u und v

      $\rightarrow$ Platzierung von (u, v) auf wenigsten ausgelasteten Rechner aus $A(u) \cap A(v)$
    - Fall 2: Nur ein Knoten wurde bereits platziert. O.B.d.A.: $A(u) = \phi \wedge A(v) \neq \phi$. Platzierung auf wenigsten ausgelasteten Rechner aus $A(v)$
    - Fall 3: $A(u) = A(v) = \phi$: Platzierung der Kante auf den am wenigsten ausgelasteten Rechner
    - Fall 4: $A(u) \neq \phi$, $A(v) \neq \phi$, aber $A(u) \cap A(v) = \phi$: Platzierung auf wenigsten ausgelasteten Rechner aus $A(u) \cup A(v)$

Part.-strategie "Oblivious" - Beispiel 3/3

- Zufällige Verteilung nach i Schritten



Knoten	0	1	2	3
Rechner	$\{0,1\}$	$\{1,2\}$	$\emptyset$	$\{1\}$

- Nachfolgende Partitionierung

Kante (u,v)	$A(u)$	$A(v)$	$A(u) \cap A(v)$	Rechner
$(1,0)$	$\{1,2\}$	$\{0,1\}$	$\{1\}$	1
$(0,2)$	$\{0,1\}$	$\emptyset$	$\emptyset$	0
$(2,0)$	$\{0\}$	$\{0,1\}$	$\{0\}$	0
$(2,1)$	$\{0\}$	$\{1,2\}$	$\emptyset$	2
$(3,0)$	$\{1\}$	$\{0,1\}$	$\{1\}$	1
$(0,3)$	$\{0,1\}$	$\{1\}$	$\{1\}$	1
$(3,2)$	$\{1\}$	$\{0,2\}$	$\emptyset$	2

Eingeschränkte Partitionierungsstrategien

- Klasse von Partitionierungsstrategien
- Hash-Wert der Kanten bestimmt Platzierung
  - Einschränkungen zur Reduzierung des Replikationsfaktors
    - Zuordnung des Knotens v zu einer eingeschränkten Menge $S(v)$
    - Kante (u, v) ist platziert in eine Partition aus $S(u) \cap S(v)$
- Beispielstrategien:
  - Grid/2D Edge (in dieser Vorlesung...)
  - Perfect Difference Set (PDS)

Grid (Powergraph)/2D Edge (GraphX)

- Rechner organisiert in quadratischer Matrix
- Eingeschränkte Menge $S(v)$ enthält alle Rechner der gesamten Zeile und Spalte der Position $h(v)$
  - Kante (u, v) wird auf einen aus der Schnittmenge dieser Rechner platziert
    - Bsp. unten: $h(u) = 1, h(v) = 9$
      $\Rightarrow$ Kante (u, v) kann auf Rechner 3 oder 7 platziert werden
    - Jede Kante kann auf mind. 2 Rechnern platziert werden
  - obere Schranke $2 \times \sqrt{N} - 1$ des Replikationsfaktors (N Anzahl Rechner)

Zuordung Kanten zu Rechner

$h(u) = 1$	2	3
4	5	6
7	8	$h(v) = 9$

Partitionierungsstrategie: High-Degree Replicated First (HDRF) (Powergraph) 1/3

- HDRF berücksichtigt bei Kantenzuordnung zu Rechner
  - bestehende Zuordnungen
    - Bevorzugung der Rechner, die bereits einen Knoten der Kante repliziert haben

      $\rightarrow$ geringerer Replikationsfaktor
  - Kantengrade
    - Zuordnung zu Rechner mit Replikaten des Knotens mit geringerem Kantengrad

      $\rightarrow$ geringerer Replikationsfaktor für den Knoten mit geringerem Grad & Bevorzugung von Knoten mit hohen Grad
  - Auslastung des Rechners
    - Bevorzugung der Rechner mit geringerer Auslastung

Partitionierungsstrategie: High-Degree Replicated First (HDRF) (Powergraph) 2/3

- Jede Maschine M erhält für eine Kante (u, v) einen Score
  $C(u, v, M) = C_{REP}(u, v, M) + \lambda \times C_{BAL}(M)$
  - Score $C_{BAL} \in$ [0,1) repräsentiert Ladefaktor
    (mehr Kanten bereits auf Rechner gespeichert $\rightarrow$ kleinerer Wert)
  - **Auswahl des Rechners mit höchsten C Score**
  - $C_{REP}(u, v, M) = g(u, M) + g(v, M)$
  - $g(v, M) = 1 + (1 - \theta(v))$ if $M \in A(v)$, else 0
  - Inkrementierung der partiellen Gradzähler $\delta(u)$ und $\delta(v)$ für Kante (u, v)
    - Normalisierter Gradzähler: $\theta(v) = \frac{\delta(v)}{\delta(u)+\delta(v)}$

HDRF (Powergraph) - Beispiel 3/3

- Verteilung der Kante $(1, 0)$

Rechner M oder Knoten n	**0**	**1**
$C_{BAL}(M)$ **(höhere Auslastung $\rightarrow$ kleinerer Wert)**	$\frac{1}{100}$	$\frac{3}{4}$
$\delta(n)$ **nach Update für Kante** $(1, 0)$	1	3
$\theta(n) = \frac{\delta(v)}{\delta(u)+\delta(v)}$	$\frac{1}{4}$	$\frac{3}{4}$
$A(n)$ **(aufgrund vorheriger Verteilung)**	$\emptyset$	$\{0, 1\}$
$g(1, M) = 1 + (1 - \theta(1))$ **if** $M \in A(1)$**, else** 0	$1\frac{3}{4}$	$1\frac{1}{4}$
$g(0, M) = 1 + (1 - \theta(0))$ **if** $M \in A(0)$**, else** 0	0	0
$C_{REP}(1, 0, M) = g(1, M) + g(0, M)$	$1\frac{3}{4}$	$1\frac{1}{4}$
$C(1, 0, M) = C_{REP}(1, 0, M) +$ $\lambda \times C_{BAL}(M)$ **mit** $\lambda = \frac{3}{4}$	$1\frac{303}{400}$ $= 1,7575$	$1\frac{13}{16}$ $= 1,8125$

Part.-strategie: Hybrid (PowerLyra)

- Hybride Strategie
  - Knotenschnitt für Knoten mit hohem Grad
    - Hashwert des Startknotens bestimmt Knoten
    - Hohe Replikationsfaktoren (& damit bessere Auslastung)
  - Ansonsten Kantenschnitt
    - Kantenschnitt für Kanten mit hohem Grad des Zielknotens
      - Hashwert des Startknotens bestimmt Knoten
    - Kantenschnitt für Kanten mit geringem Grad des Zielknotens
      - Hashwert des Zielknotens bestimmt Knoten
      - Dadurch Minimierung des Replikationsfaktors
- Berücksichtigung der tatsächlichen Knotengrade:
  Dadurch Part. in mehreren Phasen:
  1. Kantenschnitt aller Kanten und Aktualisierung der Gradzähler
  2. Wiederzuordnungsphase für Knotenschnitt der Knoten mit Grad über einen Threshold (Default: 100)

Zum Weiterlesen…

- Shiv Verma, Luke M. Leslie, Yosub Shin, Indranil Gupta:
  An Experimental Comparison of Partitioning Strategies in Distributed Graph Processing,
  VLDB 2017
  http://www.vldb.org/pvldb/vol10/p493-verma.pdf

Zusammenfassung

- Knotenzentrische Algorithmen als Programmierparadigma zum verteilten Berechnen von Graphalgorithmen
  - unterstützt den Entwickler gut zu parallelisierende Graphalgorithmen zu entwickeln
- Verteilung des Graphen
  - Kanten-/Knotenschnitt
  - Partitionierungsstrategien für Kanten/Knoten
    - Random
    - Oblivious
    - Grid/2D Edge
    - High-Degree Replicated First (HDRF)
    - Hybrid