Grundlagen paralleler Algorithmen

Stefan Lang

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg
INF 368, Raum 532
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

WS 11/12

Themen

- Grundlagen paralleler Algorithmen
- Lastverteilung

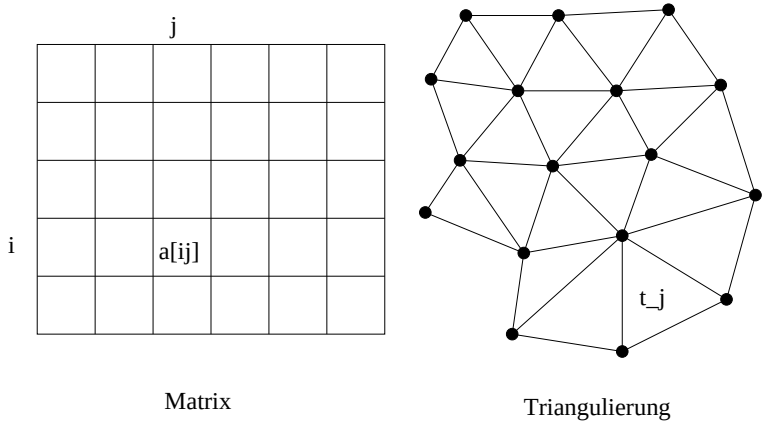
Grundlagen paralleler Algorithmen

Parallelisierungsansätze zum Entwurf paralleler Algorithmen:

1. Datenpartitionierung: *Zerlegen* eines Problems in unabhängige Teilaufgaben. Dies dient der Identifikation des maximal möglichen Parallelismus.
2. Aggolmeration: Kontrolle der *Granularität* um Rechenaufwand und Kommunikation auszubalancieren.
3. Mapping: Abbilden der Prozesse auf Prozessoren. Ziel ist eine optimale Abstimmung der logischen Kommunikationsstruktur mit der Maschinenstruktur.

Grundlagen paralleler Algorithmen: Datenzerlegung

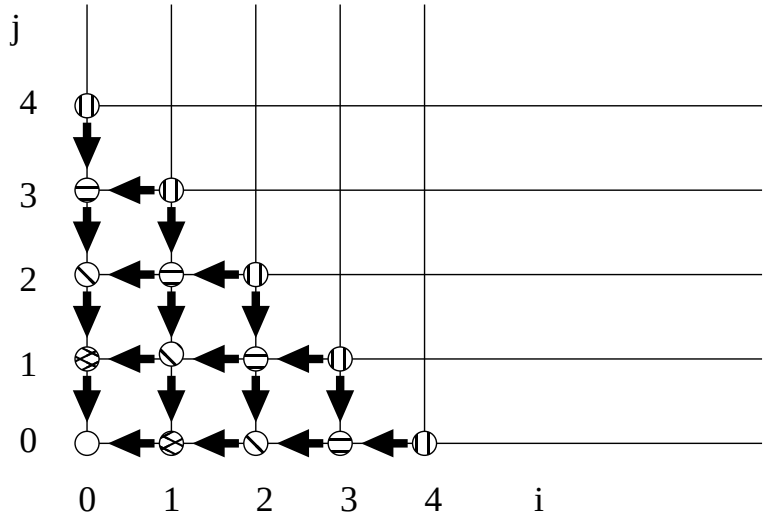
- Berechnungen sind direkt an eine bestimmte Datenstrukturen geknüpft.
- Für jedes Datenobjekt sind gewisse Operationen auszuführen, oftmals ist dieselbe Folge von Operationen auf unterschiedliche Daten pro Objekt anzuwenden. Somit kann jedem Prozess ein Teil der Daten(objekte) zugeordnet werden.



Matrix Triangulierung

- Matrixaddition $C = A + B$ können alle Elemente c_{ij} vollkommen parallel bearbeitet werden. In diesem Fall würde man jedem Prozess Π_{ij} die Matrixelemente a_{ij}, b_{ij} und c_{ij} zuordnen.
- Triangulierung bei numerischer Lösung partieller Differentialgleichungen. Hier treten Berechnungen pro Dreieck auf, die alle gleichzeitig durchgeführt werden können, jedem Prozess würde man somit eine Teilmenge der Dreiecke zuordnen.

Grundlagen paralleler Algorithmen: Datenabhängigkeit



Datenabhängigkeiten im Gauß–Seidel–Verfahren.

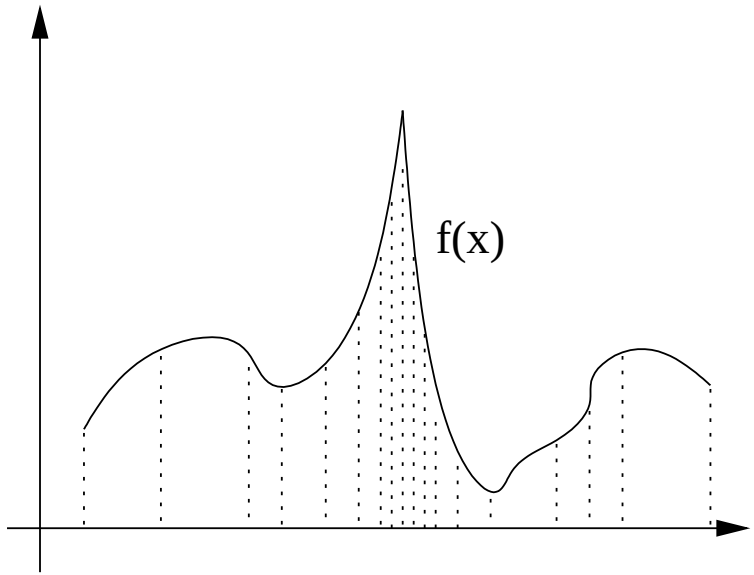
- Operationen können oft nicht für alle Datenobjekte gleichzeitig durchgeführt werden.
- Beispiel: Gauß-Seidel-Iteration mit lexikographischer Nummerierung. Berechnung am Gitterpunkt (i, j) hängt vom Ergebnis der Berechnungen an den Gitterpunkten $(i-1, j)$ und $(i, j-1)$ ab.
- Der Gitterpunkt $(0, 0)$ kann ohne jede Voraussetzung berechnet werden.
- Alle Gitterpunkte auf den Diagonalen $i + j = const$ können parallel bearbeitet werden.

Grundlagen paralleler Algorithmen: Funkt. Zerlegung

Funktionale Zerlegung

- bei unterschiedlichen Operationen auf gleichen Daten.
- Beispiel Compiler: Dieser vollführt die Schritte lexikalische Analyse, Parsing, Codegenerierung, Optimierung und Assemblierung. Jeder Schritt kann einem separaten Prozess zugeordnet werden („Makropipelining“).

Irreguläre Probleme



- Keine a-priori Zerlegung möglich
- Berechnung des Integrals einer Funktion $f(x)$ durch adaptive Quadratur
- Intervallwahl hängt von f ab und ergibt sich während der Berechnung

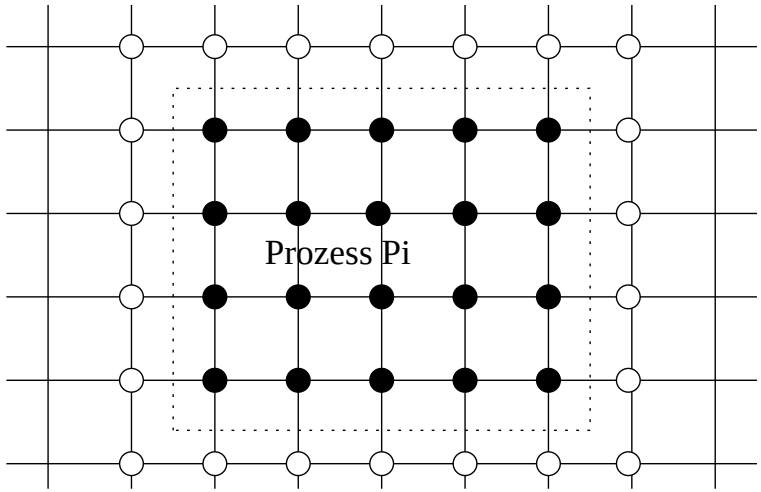
Grundlagen paralleler Algorithmen: Agglomeration I

- Zerlegungsschritt zeigt die maximale Parallelität auf.
- Nutzung (im Sinne von ein Datenobjekt pro Prozess) ist meist nicht sinnvoll (Kommunikationsaufwand)
- Agglomeration: Zuordnung von mehreren Teilaufgaben zu einem Prozess, damit wird Kommunikation für diese Teilaufgaben in möglichst wenigen Nachrichten zusammengefaßt.
- Reduktion der Anzahl der zu sendenden Nachrichten, weitere Einsparungen bei *Datenlokalität*
- Als *Granularität* eines parallelen Algorithmus bezeichnet man das Verhältnis:

$$\text{Granularität} = \frac{\text{Anzahl Nachrichten}}{\text{Rechenzeit}}.$$

- Agglomeration reduziert also die Granularität.

Grundlagen paralleler Algorithmen: Agglomeration II



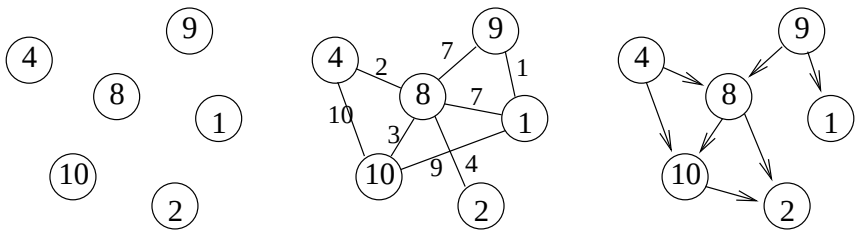
Gitterbasierte Berechnungen

- Berechnungen werden für alle Gitterpunkte parallel durchgeführt werden.
- Zuweisung einer Menge von Gitterpunkten zu einem Prozeß
- Alle Modifikationen auf Gitterpunkten können u. U. gleichzeitig durchgeführt, es bestehen also keine Datenabhängigkeiten.

Grundlagen paralleler Algorithmen: Agglomeration III

- Ein Prozess besitze N Gitterpunkte und muss somit $O(N)$ Rechenoperationen ausführen.
- Nur für Gitterpunkte am Rand der Partition ist eine Kommunikation notwendig.
- Es ist also somit nur für insgesamt $4\sqrt{N}$ Gitterpunkte Kommunikation notwendig.
- Verhältnis von Kommunikationsaufwand zu Rechenaufwand verhält sich somit wie $O(N^{-1/2})$
- Erhöhung der Anzahl der Gitterpunkte pro Prozessor macht den Aufwand für die Kommunikation relativ zur Rechnung beliebig klein *Oberfläche–zu–Volumen–Effekt*.

Wie ist die Agglomeration durchzuführen?



1. Ungekoppelte Berechnungen
2. Gekoppelte Berechnungen
3. Gekoppelte Berechnungen mit zeitl. Abhängigkeit

Grundlagen paralleler Algorithmen: Zerlegen (a)

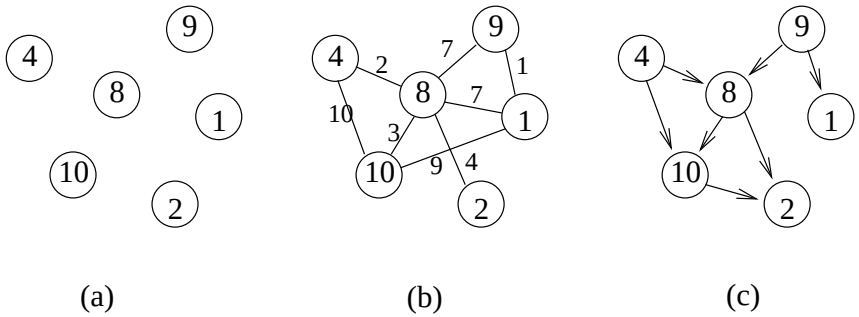
(a) Ungekoppelte Berechnungen

- Berechnung besteht aus Teilproblemen, die völlig unabhängig voneinander berechnet werden können.
- Jedes Teilproblem kann unterschiedliche Rechenzeit benötigen.
- Darstellbar als Menge von Knoten mit Gewichten. Gewichte sind ein Maß für die erforderliche Rechenzeit.
- Agglomeration ist trivial. Man weist die Knoten der Reihe nach (z.B. der Größe nach geordnet oder zufällig) jeweils dem Prozess zu der am wenigsten Arbeit hat (dies ist die Summe all seiner Knotengewichte).
- Agglomeration wird komplizierter wenn die Anzahl der Knoten sich erst während der Berechnung ergibt (wie bei der adaptiven Quadratur) und/oder die Knotengewichte a priori nicht bekannt sind (wie z.B. bei depth first search).

$\rightarrow$ Lösung durch dynamische Lastverteilung

- zentral: ein Prozess nimmt die Lastverteilung vor
- dezentral: ein Prozess holt sich Arbeit von anderen, die zuviel haben

Grundlagen paralleler Algorithmen: Zerlegen (b)



(b) Gekoppelte Berechnungen

- Standardmodell für statische, datenlokale Berechnungen.
- Berechnung wird durch einen ungerichteten Graphen beschrieben.
- Erst ist eine Berechnung pro Knoten erforderlich, deren Berechnungsdauer wird vom Knotengewicht modelliert. Danach tauscht jeder Knoten Daten mit seinen Nachbarknoten aus.
- Anzahl der zu sendenden Daten ist proportional zum jeweiligen Kantengewicht.
- Regelmäßiger Graph mit konstanten Gewichten: triviale Agglomeration.

Grundlagen paralleler Algorithmen: Zerlegen

- Allgemeiner Graph $G = (V, E)$ bei P Prozessoren: Knotenmenge V ist so zu partititonieren, dass

$$\bigcup_{i=1}^{P} V_i = V, \quad V_i \cap V_j = \emptyset, \quad \sum_{v \in V_i} g(v) = \sum_{v \in V} g(v)/|V|$$

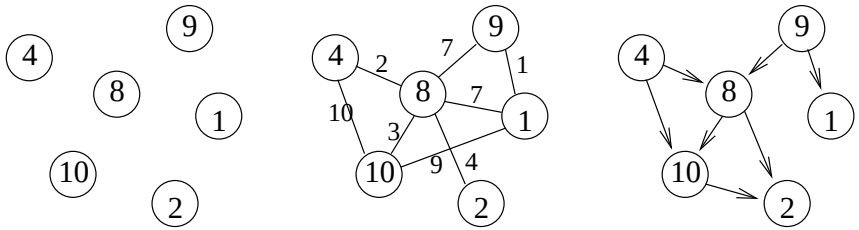
  und die Separatorkosten

$$\sum_{(v,w) \in S} g(v,w) \to \min, \; S = \{(v,w) \in E | v \in V_i, w \in V_j, i \neq j\}$$

  minimal sind.

- Dieses Problem wird als *Graphpartitionierungsproblem* bezeichnet.
- $\mathcal{NP}$–vollständig, bereits im Fall konstanter Gewichte und $P = 2$ (Graphbisektion) $\mathcal{NP}$–vollständig.
- Es existieren gute Heuristiken, welche in linearer Zeit (in der Anzahl der Knoten, $O(1)$ Nachbarn) eine (ausreichend) gute Partitionierung erzeugen.

Grundlagen paralleler Algorithmen: Zerlegen (c)



(a) (b) (c)

(c) Gekoppelte Berechnungen mit zeitlicher Abhängigkeit

- Modell ist ein gerichteter Graph.
- Ein Knoten kann erst berechnet werden kann, wenn alle Knoten von eingehenden Kanten berechnet sind.
- Ist ein Knoten berechnet wird das Ergebnis über die ausgehenden Kanten weitergegeben. Die Rechnzeit entspricht den Knotengewichten, die Kommunikationszeit entspricht den Kantengewichten.
- Im allgemeinen sehr schwer lösbares Problem.
- Theoretisch nicht „schwieriger als Graphpartitionierung“ (auch $\mathcal{NP}$–vollständig)
- Praktisch sind keine einfachen und guten Heuristiken bekannt.
- Für spezielle Probleme, z.B.adaptive Mehrgitterverfahren, kann man jedoch gute Heuristiken finden.

Grundlagen paralleler Algorithmen: Mapping

Mapping: Abbilden der Prozesse auf Prozessoren

- Menge der Prozesse Π bildet ungerichteten Graphen $G_\Pi = (\Pi, K)$: Zwei Prozesse sind miteinander verbunden, wenn sie miteinander kommunizieren (Kantengewichte konnten den Umfang der Kommunikation modellieren).
- Ebenso bildet die Menge der Prozessoren P mit dem Kommunikationsnetzwerk einen Graphen $G_P = (P, N)$: Hypercube, Feld.
- Sei $|\Pi| = |P|$ und es stellt sich folgende Frage:
  Welcher Prozess soll auf welchem Prozessor ausgeführt werden?
- Im allgemeinen wollen wir die Abbildung so durchführen, dass Prozesse, die miteinander kommunizieren möglichst auf benachbarte oder nahe Prozessoren abgebildet werden.
- Dieses Optimierungsproblem bezeichnet man als *Graphabbildungsproblem*
- Dieses ist wieder $\mathcal{NP}$–vollständig (leider).
- Heutige Multiprozessoren besitzen sehr leistungsfähige Kommunikationsnetzwerke: In cut-through-Netzwerken ist die Übertragungszeit einer Nachricht praktisch entfernungsunabhängig.
- Das Problem der optimalen Prozessplatzierung ist damit nicht mehr vorrangig.

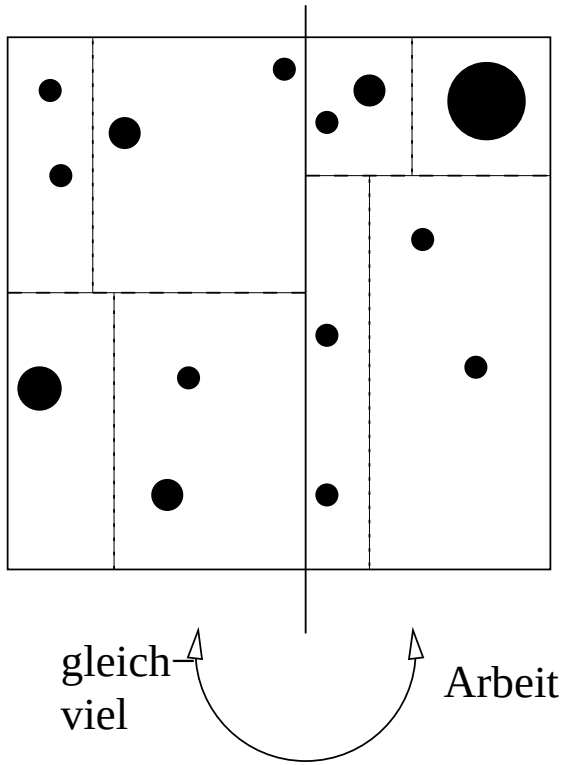
Grundlagen paralleler Algorithmen: Lastverteilung

Situation 1: Statische Verteilung ungekoppelter Probleme

- Aufgabe: Aufteilung der anfallenden Arbeit auf die verschiedenen Prozessoren.
- Dies entspricht dem Agglomerationsschritt bei welchem man die parallel abarbeitbaren Teilprobleme wieder zusammengefasst hat.
- Das Maß für die Arbeit sei dabei bekannt.
- **Bin Packing**
  - Anfangs sind alle Prozessoren leer.
  - Knoten, die in beliebiger Reihenfolge oder sortiert (z.B. der Grösse nach) vorliegen, werden nacheinander auf den Prozessor mit der aktuell wenigsten Arbeit gepackt.
  - Das funktioniert auch dynamisch, falls im Rechenvorgang neue Arbeit entsteht.

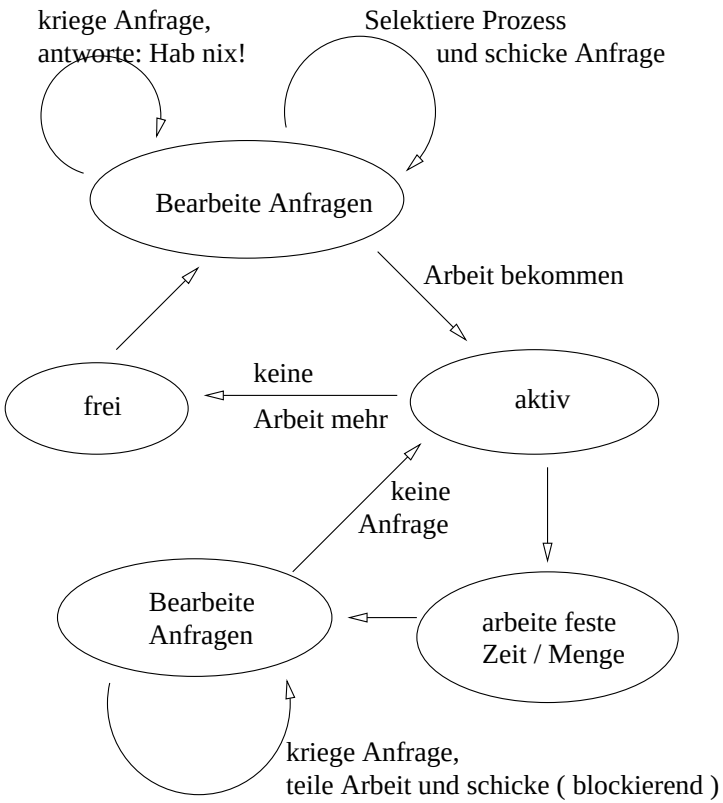
Grundlagen paralleler Algorithmen: Lastverteilung

- **Rekursive Bisektion**
  - Jedem Knoten sei eine Position im Raum zugeordnet.
  - Der Raum wird orthogonal zu den Koordinatensystem so geteilt, daß in den Teilen etwa gleich viel Arbeit besteht.
  - Dieses Vorgehen wird dann rekursiv auf die entstandenen Teilräume mit alternierenden Koordinatenrichtungen angewandt.



Grundlagen paralleler Algorithmen: Lastverteilung

Situation 2: Dynamische Verteilung ungekoppelter Probleme



Aktivitäts-/Zustandsdiagramm

- Das Maß für die Arbeit sei unbekannt.
- Prozess ist entweder aktiv (verrichtet Arbeit) oder frei (arbeitslos).
- Beim Teilen der Arbeit sind folgende Fragen zu berücksichtigen:
  - Was will ich abgeben? Beim Travelling-Salesman Problem z.B. möglichst Knoten aus dem Stack, die weit unten liegen.
  - Wieviel will ich abgeben? z.B. die Hälfte der Arbeit (half split).
- Neben der Arbeitsverteilung kann weitere Kommunikation stattfinden (Bildung des globalen Minimums bei branch-and-bound beim Travelling-Salesman).
- Desweiteren besteht das Problem der Terminierungserkennung hinzu. Wann sind alle Prozesse idle?

Grundlagen paralleler Algorithmen: Lastverteilung

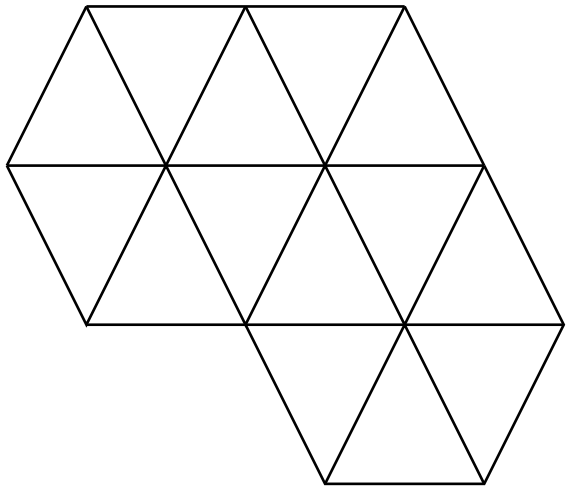
Welcher Idle-Prozess soll als nächster angesprochen werden?
Verschiedene Selektionsstrategien:

- **Master/Slave(Worker) Prinzip**
  Ein Prozess verteilt die Arbeit. Er weiß, wer aktiv bzw. frei ist und leitet die Anfrage weiter. Er regelt (da er weiß, wer frei ist) auch das Terminierungsproblem. Nachteil diese Methode skaliert nicht. Alternativ: hierarchische Struktur von Mastern.
- **Asynchrones Round Robin**
  Der Prozess Π_i hat eine Variable target_i. Er schickt seine Anfrage an Π_{target_i} und setzt dann $\text{target}_i = (\text{target}_i + 1)\%P$.
- **Globales Round Robin**
  Es gibt nur eine globale Variable target. Vorteil: keine gleichzeitigen Anfragen an denselben Prozess. Nachteil: Zugriff auf eine globale Variable (was z.B. ein Server Prozess machen kann).
- **Random Polling**
  Jeder wählt zufällig einen Prozess mit gleicher Wahrscheinlichkeit ($\rightarrow$ Paralleler Zufallsgenerator, achte zumindest auf das Austeilen von seeds o.Ä.). Dieses Vorgehen bietet eine gleichmäßige Verteilung der Anfragen und benötigt keine globale Resource.

Grundlagen paralleler Algorithmen: Lastverteilung

Graphpartitionierung

Betrachten wir ein Finite-Elemente-Netz:



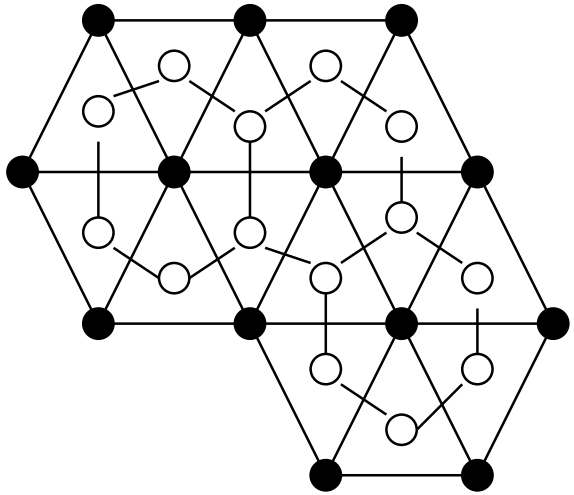
Es besteht aus Dreiecken $T = \{t_1, \ldots, t_N\}$ mit

$$\bar{t}_i \cap \bar{t}_j = \begin{cases} \emptyset \\ \text{ein Knoten} \\ \text{eine Kante} \end{cases}$$

- Arbeit findet beim Verfahren der Finiten Elemente in den Knoten statt.
- Alternativ kann man auch in jedes Dreieck einen Knoten legen, diese mit Kanten verbinden und den so entstehenden Dualgraphen betrachten.

Grundlagen paralleler Algorithmen: Lastverteilung

Graph und zugehöriger dualer Graph



Die Aufteilung des Graphen auf Prozessoren führt zum Graphpartitionierungsproblem. Dazu machen wir die folgenden Notationen:

$$G = (V, E) \qquad \text{(Graph oder Dualgraph)}$$
$$E \subseteq V \times V \qquad \text{symmetrisch (ungerichtet)}$$

Die Gewichtsfunktionen

$$w : V \longrightarrow \mathbb{N} \qquad \text{(Rechenaufwand)}$$
$$w : E \longrightarrow \mathbb{N} \qquad \text{(Kommunikation)}$$

Grundlagen paralleler Algorithmen: Lastverteilung

Die Gesamtarbeit

$$W = \sum_{v \in V} w(v)$$

Außerdem sei k die Anzahl der zu bildenden Partitionen, wobei $k \in \mathbb{N}$ und $k \geq 2$ sei. Gesucht ist nun eine Partitionsabbildung

$$\pi : V \longrightarrow \{0, \ldots, k-1\}$$

und der dazugehörige Kantenseparator

$$X_\pi := \{(v, v') \in E \mid \pi(v) \neq \pi(v')\} \subseteq E$$

Das Graphpartitionierungproblem besteht nun darin, die Abbildung π so zu finden, daß das Kostenfunktional (Kommunikationskosten)

$$\sum_{e \in X_\pi} w(e) \rightarrow \min$$

minimal wird unter der Nebenbedingung (Gleichverteilung der Arbeit)

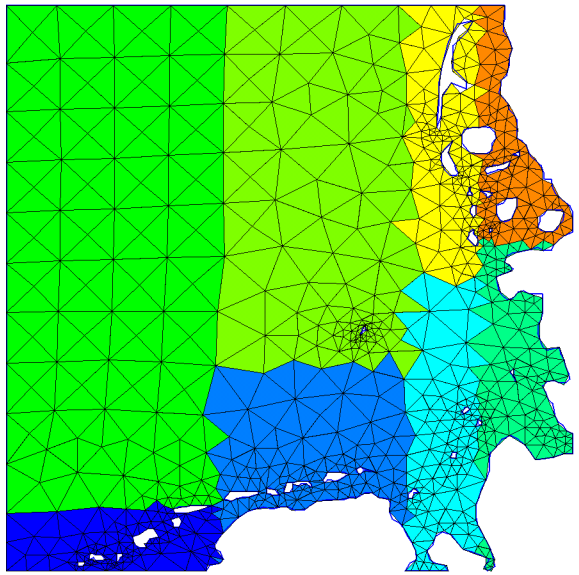
$$\sum_{v, \pi(v)=i} w(v) \leq \delta \frac{W}{k} \quad \text{für alle } i \in \{0, \ldots, k-1\}$$

wobei δ das erlaubte Ungleichgewicht bestimmt (δ = 1.1 10% Abweichung).

Grundlagen paralleler Algorithmen: Lastverteilung

- Berechnungkosten dominieren Kommunikationskosten. Ansonsten wäre wegen der hohen Kommunikationskosten evtl. die Partitionierung unnötig. Das ist allerdings nur ein Modell für die Laufzeit!
- Bei Zweiteilung spricht man vom Graphbisektionsproblem. Durch rekursive Bisektion lassen sich 2^d-Wege-Partitionierungen erzeugen.
- Problematischerweise ist die Graphpartitionierung für $k \geq 2$ NP-vollständig.
- Optimale Lösung würde also die eigentliche Rechnung dominieren. Paralleler Overhead und ist daher nicht annehmbar.

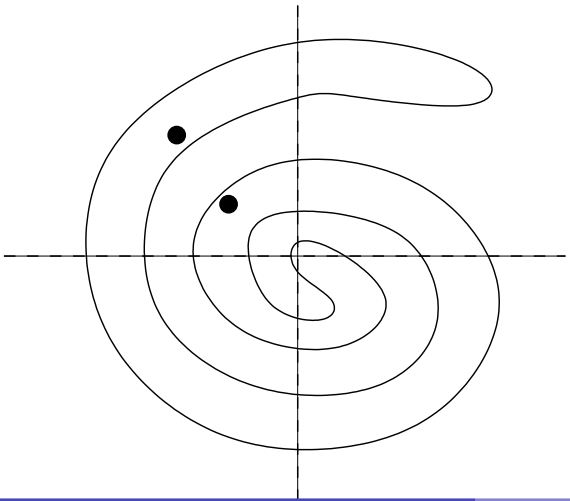
$\rightarrow$ Notwendigkeit schneller Heuristiken.



Grundlagen paralleler Algorithmen: Lastverteilung

Rekursive Koordinatenbisektion(RCB)

- Man benötigt die Positionen der Knoten im Raum (bei Finite-Elemente Anwendungen sind die vorhanden).
- Bisher haben wir das Verfahren unter dem Namen **Rekursive Bisektion** gesehen.
- Diesmal ist das Problem gekoppelt. Daher ist es wichtig, daß der Raum, in dessen Koordinaten die Bisektion durchgeführt wird, mit dem Raum, in dem die Knoten liegen, übereinstimmt.
- Im Bild ist das nicht der Fall. Zwei Knoten mögen zwar räumlich dicht beieinanderliegen, eine Koordinatenbisektion macht aber keinen Sinn, da die Punkte hier nicht gekoppelt sind, es also einem Prozessor gar nichts nützt, beide zu speichern.

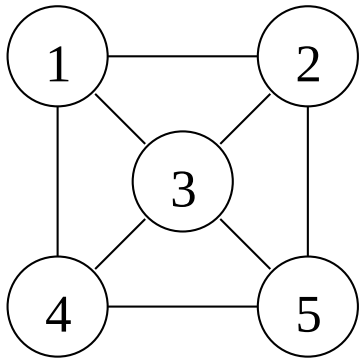


Grundlagen paralleler Algorithmen: Lastverteilung

Rekursive Spektralbisektion(RSB)

Hier werden die Positionen der Knoten im Raum nicht benötigt. Man stellt zunächt die Laplacematrix A(G) zum vorliegenden Graphen G auf. Diese ist folgendermaßen definiert:

$$A(G) = \{a_{ij}\}_{i,j=1}^{|V|} \quad \text{mit} \quad a_{ij} = \begin{cases} \text{grad}(v_i) & i = j \\ -1 & (v_i, v_j) \in E \\ 0 & \text{sonst} \end{cases}$$



	1	2	3	4	5
1	3	−1	−1	−1	0
2	−1	3	−1	0	−1
3	−1	−1	4	−1	−1
4	−1	0	−1	3	−1
5	0	−1	−1	−1	3

Graph und zugehörige Laplacematrix

Grundlagen paralleler Algorithmen: Lastverteilung

Dann löse das Eigenwertproblem

$$Ae = \lambda e$$

Der kleinste Eigenwert λ_1 ist gleich Null, denn mit $e_1 = (1, \ldots, 1)^T$ gilt $Ae_1 = 0 \cdot e_1$. Der zweitkleinste Eigenwert λ_2 allerdings ist ungleich Null, falls der Graph zusammenhängend ist.
Die Bisektion findet nun anhand der Komponenten des Eigenvektors e_2 statt, und zwar setzt man für $c \in \mathbb{R}$ die beiden Indexmengen

$$I_0 = \{i \in \{1, \ldots, |V|\} \mid (e_2)_i \leq c\}$$
$$I_1 = \{i \in \{1, \ldots, |V|\} \mid (e_2)_i > c\}$$

und die Partitionsabbildung

$$\pi(v) = \begin{cases} 0 & \text{falls } v = v_i \wedge i \in I_0 \\ 1 & \text{falls } v = v_i \wedge i \in I_1 \end{cases}$$

Dabei wählt man das c so, daß die Arbeit gleichverteilt ist.

Grundlagen paralleler Algorithmen: Lastverteilung

Kerninghan/Lin

- Iterationsverfahren, daß eine vorgegebene Partition unter Berücksichtigung des Kostenfunktionals verbessert.
- Wir beschränken uns auf Bisektion (k-Wege Erweiterung möglich) und nehmen die Knotengewichte als 1 an.
- Außerdem sei die Anzahl der Knoten gerade.
- Das Verfahren von Kerningham/Lin wird meist in Kombination mit anderen Verfahren benutzt.

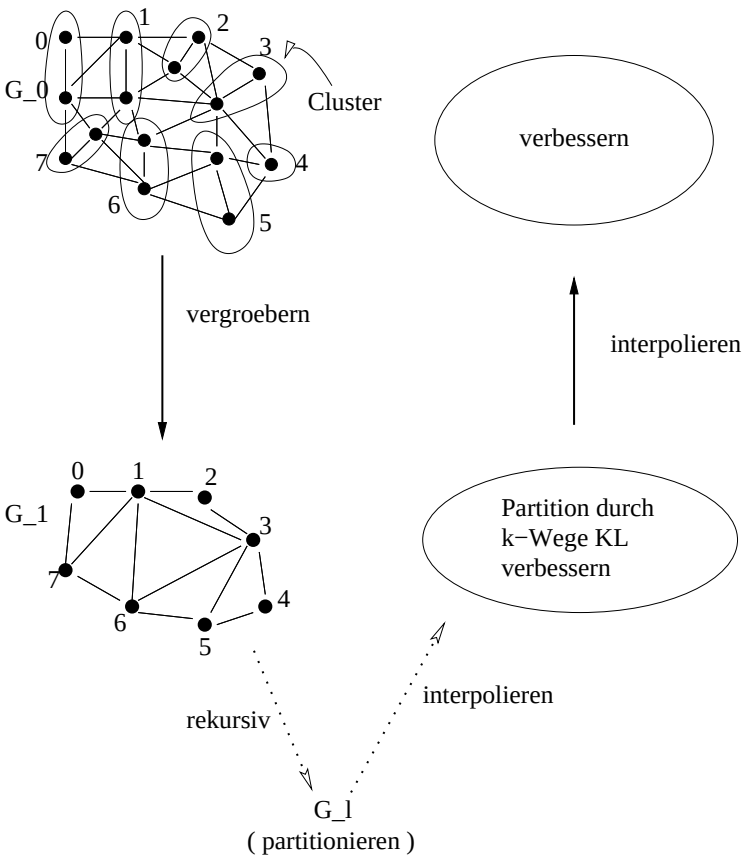
Grundlagen paralleler Algorithmen: Lastverteilung KL

i = 0; $| V |$ = 2n;
// Erzeuge Π_0 so, dass Gleichverteilung erfüllt ist.
while (1) { // Iterationsschritt
$V_0 = \{v \mid \pi(v) = 0\}$;
$V_1 = \{v \mid \pi(v) = 1\}$;
$V_0' = V_1' = \emptyset$;
$\bar{V_0} = V_0$;
$\bar{V_1} = V_1$;
for (i = 1 ; i $\leq$ n ; i++)
{
// Wähle $v_i \in V_0 \setminus V_0'$ und $w_i \in V_1 \setminus V_1'$ so, dass
$\sum_{e \in (\bar{V_0} \times \bar{V_1}) \cap E} w(e) - \sum_{e \in (V_0'' \times V_1'') \cap E} w(e) \rightarrow max$
// wobei
$V_0'' = \bar{V_0} \setminus \{v_i\} \cup \{w_i\}$
$V_1'' = \bar{V_1} \setminus \{w_i\} \cup \{v_i\}$
// setze
$\bar{V_0} = \bar{V_0} \setminus \{v_i\} \cup \{w_i\}$;
$\bar{V_1} = \bar{V_1} \setminus \{w_i\} \cup \{v_i\}$;
} // for
// Bem.: max kann negativ sein, d.h. Verschlechterung der Separatorkosten
// Ergebnis an dieser Stelle: Folge von Paaren $\{(v_1, w_1), \ldots, (v_n, w_n)\}$.
// V_0, V_1 wurden noch nicht verändert.
// Wähle nun eine Teilfolge bis m $\leq$ n, die eine maximale
// Verbesserung der Kosten bewirkt („hill climbing “)
$V_0 = V_0 \setminus \{v_1, \ldots, v_m\} \cup \{w_1, \ldots, w_m\}$;
$V_1 = V_1 \setminus \{w_1, \ldots, w_m\} \cup \{v_1, \ldots, v_m\}$;
if (m == 0) break; // Ende
} // while

Grundlagen paralleler Algorithmen: Lastverteilung

Multilevel k-Wege Partitionierung

1. Zusammenfassen von Knoten des Ausgangsgraphen G^0 (z.B. zufällig oder aufgrund schwerer Kantengewichte) in Clustern.
2. Diese Cluster definieren die Knoten in einem vergröberten Graphen G^1.
3. Rekursiv führt dieses Verfahren auf einen Graphen G^l.



1. G^l wird nun partitioniert (z.B. RSB/KL)
2. Anschließend wird die Partitionsfunktion auf dem feineren Graphen G^{l-1} interpoliert.
3. Diese interpolierte Partitionsfunktion kann jetzt wieder mittels KL rekursiv verbessert und anschließend auf dem nächst feineren Graphen interpoliert werden.
4. So verfährt man rekursiv bis zum Ausgangsgraphen.
5. Die Implementierung ist dabei in $\mathcal{O}(n)$ Schritten möglich. Das Verfahren liefert qualitativ hochwertige Part.

Grundlagen paralleler Algorithmen: Lastverteilung

Weitere Probleme

Weitere Probleme bei der Partitionierung sind

- **Dynamische Repartitionierung:** Der Graph soll mit möglichst wenig Umverteilung lokal abgeändert werden.
- **Constraint Partitionierung:** Aus anderen Algorithmenteilen sind zusätzliche Datenabhängigkeiten vorhanden.
- **Parallelisierung des Partitionsverfahrens:** Ist nötig bei großen Datenmengen (Dafür gibt es fertige Software)