Zusammenfassung der Vorlesung Rechnerstrukturen 3 WS 1999/2000

Anmerkungen

Diese Zusammenfassung ist insofern nicht umfassend, als das technische Aspekte auf CPU-Ebene überwiegend ausgeklammert wurden.

Bei Fehlern, Anmerkungen, Fragen oder Kritik bitte ich um Mail unter kontakt@carsten-buschmann.de. Weiterhin bin ich unter www.carsten-buschmann.de erreichbar.

Zur Notation: Indizes werden häufig durch Groß-/Kleinschreibung verdeutlicht, d.h. Emin = E_{min}.

1. Einführung in die Parallelverarbeitung

Parallelverarbeitung	Zielt auf gleichzeitige Berechnung von Ergebnissen
Parallelrechner	Computer, mit dem Parallelverarbeitung möglich ist: Menge von CPUs, die kommunizieren können
Grad der Parallelität	Anz. der an der Berechnung beteiligten CPUs
Moore's Gesetz	erreichbare Rechenleistung pro Chip verdoppelt sich alle 18 Monate
MIMD/Kontrollparallelität	multiple instruction stream, multipe datastream jede CPUs führt ein eigenes Programm aus (z.B. Parsitec)
SIMD/Datenparallelität	singe instruction stream, multipe datastream alle CPUs führen zu einem Zeitpunkt den gleichen Befehl aus (z.B. Systola)
Durchsatz	Anz. der Probleme, die pro Zeiteinheit von einem Rechner gelöst werden kann
Latenz	Ausführungszeit der Berechnung eines spez. Problems
Pipelining	Daten werden auf einem einfach gerichteten Datenpfad von einer CPU zur nächsten weitergereicht
Speedup	Latenz schnellster seq. Alg. --- Latenz schnellster par. Alg.
Skalierbar	Parallelitätsgrad wächst mind. linear mit Problemgröße
Sieb des Erathostenes (MIMD/SIMD)	MIMD: kann Effizienz verringern, Erg. verfälschen → mehr CPUs bringen keine Verbesserung mehr beide: geringe Parallelitätsgrad bei opt. Speedup, massive Parallelität wird nur mit geringerem Speedup belohnt
Satz Speedup	bei p CPU's ist der Speedup max. p
Amdahls Gesetz	$\text{Gesamtspeedup} <= \frac{1}{f + (1-f) / p}$
Komplexität, Sorten	Maß für den Aufwand einer Berechnung: Zeit, Platz, Parallelität (Periode, Leistung, Fläche)
Obere/Untere Schranken	$O(f) \Leftrightarrow$ ex. k, so daß die Ausführung max. k * f Zeiteinheiten dauert $\Omega(f) \Leftrightarrow$ ex. k, so daß die Ausführung min. k * f Zeiteinheiten dauert
Komplexität eines Problems Θ	$\Theta(f) \Leftrightarrow O(f)$ und $\Omega(f)$
Kosten einer parallelen Berechnung	=Parallelitätsgrad * Latenz
Satz Kosten parallel/seriell	Cpar >= Cseq
Optimaler paralleler Algorithmus	$\Leftrightarrow$ Cpar <= k * Cseq mit k unabh. von n und p d.h. Speedup linear zu p und par. Alg mit 1 CPU nicht schlechter als seq. Alg. bei n -> ∞

2. Parallele Rechnermodelle

Lemma Sortieren mit PRAM	EREW PRAM mit p CUPs kann p Zahlen in O(log p) sortieren, das ist optimal → Θ(log p)
Satz Simulation versch. PRAMs	Jede PRAM kann von einer EREW PRAM mit einem Zeitfaktor von O(log p) simuliert werden
UMA	Uniform memory access: Speicher ist gleichschnell von allen CPUs

	zugreifbar
NUMA	Speicher verteilt bei gemeinsamen Adressraum, Geschw. variiert abh. von zugreifendem Prozessor und Anz der zugreifenden CPUs
Probleme Speicherzugriff	- Phys. gleichzeitiger Zugriff nicht möglich - cache coherence
Zusammenfassung PRAM	einfaches Modell zur Untersuchung max. Parallelisierbarkeit in Hinsicht auf gem. Speicherzugriff unrealistisch

Distributed Memory Modelle

Durchmesser	eines Graphen ist das Max. der min. Pfade zw. je 2 Knoten
Schnittbreite	min. Anz. Kanten, die man entfernen muß, um Graph hälftig zu teilen
Leitungslänge	Länge der längsten Leitung
Grad	Anz. Kanten, die von einem Knoten ausgehen

	Knoten	Durchmesser	Schnittbreite	Grad	konstante Kantenlänge
Array	p	p-1	1	2	Ja
Ring	p	p/2	2	2	Ja
Mesh	$p=k^2$	2k-2	k	4	Ja
Torus	$p=k^2$	k-1	2k	4	Ja
3-D-Mesh	$p=k^3$	3k-3	k^2	6	Ja
Baum	p-1	2(log p - 1)	1	3	Nein
Hypercube	p	log p	p/2	log p	Nein
CCC	$p=2^{k*}k$	2k	2^{k-1}	3	Nein

3. ISA

ISA Eigenschaften	Mesh, global getaktet, verteilter Speicher, Kommunikationsregister feinkörnig, schnelle Akkumulationsops., CPU optimiert auf Fläche und Leistungsverbrauch, bitserielle Verarbeitung, typisch für Bildbearbeitung, Numerik, Video, Krypto
Senden entlang Zeilen	`<C:= Cwest; 1111; 0111>`
Zeilenakkumulation	`<C:= C + Cwest; 1111; 0111>`
Vorteile ISA	- kurze Leitungen - kleine CPUs (small ist fast, many) - Akkumulationssops. - Skalierbarkeit - Taktverschiebung kein Problem → hohe Taktraten möglich
ISA vs. SIMD	ISA leistungsfähiger: Broadcast: ISA 1 Befehl, SIMD-Mesh braucht n – 1
ISA vs. MIMD	kleinere Kontrollflussbreite
(links) Ringschieben	`<C:= Cwest; 1111; 0111>` `<C:= Ceast; 1111; 1110>`
Gossiping	1. Broadcast der ersten Spalte über die Reihen 2. Sichern des gesendeten Wertes 3. Schreiben des zu sendenden Wertes 4. Ringschieben der Zeilen

```
< C:=R[n-1]; 1...1; 1...1 >;
for i:= 1 to n-1 do begin
  < C:=Cwest; 1...1; 01..1 >;   senden entlang der Zeilen
  < R[i]:=C; 1...1; 1...1 >;    Wert sichern
  < C:=R[n-1]; 1...1; 1...1 >;  zusendende Zahl nach C
  < C:= Cwest; 1...1; 01..1>    Ringschieben
  < C:= Ceast; 1...1; 1..10>
  < C:=R[n-1]; 1...1; 1...1 >;  sichern der geschobenen
end                             zu sendenden Zahl
```

4. Parallele Algorithmen auf versch. Modellen

Komplexität opt. Merge seriell: O(n * log n) (in jedem Falle)

Algorithmus Merge PRAM

```
forall
  if i < n/2 then { low :=  n/2; high := n-1 }
             else { low := 0;    high := n/2-1 }
  x := A[i];
  repeat
    index := floor((low + high)/2);
    if x <= A[index] then high := index – 1;
                     else low := index + 1;
  until low > high;
  B[i + high – n/2 + 1] := x;
endforall;
```

Komplexität: O(log n) wg. log n Durchläufen der Schleife, Rest O(1)

Merge Array

OETS: Odd-Even-Transposition-Sort auf Array

0-1-Prinzip (Knuth): Sortiernetz sortiert bel. Folgen ⇔ Sortiert alle Folgen von 0 und 1
Beweis siehe Folien (2)

Vorteile 0-1-prinzip:
1. nur 2^n statt n! Kombinationen zu überprüfen
2. graphischer Beweis mit SW-Darstellung mgl.

Karo zum Merge im Array: Sortiernetz s. Folien (3)
Komplexität O(n), Ω (n)=n/2 (schlimmster Fall: halbes Array durchwandern)

Merge im Mesh
1. Zeilen abwechselnd auf- und absteigend sortieren
2. Spalten nach unten sortieren
3. Zeilen aufsteigend sortieren

Beweis (0-1-P.):
2 Fälle:
1. i, j beide gerade oder ungerade (ungerade Anz. Zeilen dazw.) → 2 Buckel
2. i, j einer gerade einer ungerade (gerade Anz. Zeilen dazw.) → 1 Buckel

Komplexität:
O(√n), Ω(√n) (Durchwandern halber Zeile oder Spalte)

Bitonische Folge heißt 0-1-Folge, wenn max. 2 Wechsel zw. 0 und 1 (6 Sorten s. Folien (4))

Bn: n gerade: [0:n/2][1, n/2+1]...[n/2-1:n-1]

Satz Bn: Bn auf eine bit. Folgen der Länge n angewandt → 2 bit. Folgen der Länge n/2 entstehen, untere Hälfte enthält größere Elemente

Bitonic Merge: Folge von Bn, 2 Bn/2,..., n/2 B2 Netze
Besonders geeignet: Hypercube, weil hier wie benötigt die CPUs verbunden sind, die Zweierpotenz Abstand haben
Komplexität: O(log n), jede Stufe n/2 Vergleicher
2. Liste muß gespeigelt werden: rek Hälften-tauschen (Hypercube: Tauschen mit Abst. d-1, d-2,...,1 → O(log n) zum Spiegeln von n Zahlen

Sortieren auf PRAM
Komplexität: Tsort(n)=Tsort(n/2) + Tmerge(n)= Tsort(n/2) + c*log(n)
Tsort(1)=0
Tsort(n)= c*log(n) + c*log(n/2) + c*log(n/4) + ... + c *1
=c*(log(n) + (log(n) – 1) + (log(n) – 2) + ... + 1)
=(log n /)2 * (log n + 1)
→O(log^2 n)

schneller sortieren auf PRAM: n^2 CPUs
CPUs in i-ter Zeile erhalten A[i], CPUs in j-ter Spalte erhalten A[j]
alleCPUs: wenn A[i] > A[j], dann g auf 1 sonst 0
Zeilensummation der g → s[i] = Anz. Elemente kleiner A[i]
CPU (i,0) schreibt A[i] nach Erg[s[i]]
Komplexität: O(log n) wg. Summation nach Turnieralgorithmus

Sortieren im Mesh zuerst Spalten Sortieren (O(√n) mit OETS)
dann (log(√n)) mal benachbarte „Spalten“ Mergen O(√n)
→ insges. O(log n *√n)

Shear Sort
```
for i := 0 to log√n –1 do
begin
  sort_rows_alternating; O(√n)
  sort_colums;           O(√n)
end
sort_rows;
```
In jedem Durchgang wird die Anz. der schmutigen Zeilen halbiert → log √n
→O(log n * √n)

schneller sortieren im Mesh Mesh in 4 Viertel aufteilen
Merge_neu
```
  sort_rows_alternating; O(√n)
  sort_colums;           O(√n)
  sort_rows_alternating; O(√n)
  sort_colums;           O(√n)
  sort_rows;             O(√n)
```
→ O(√n)
Tsort(√n)=Tsort(√n)/2) + Tmerge(√n) Tmerge(√n) =c*√n Tsort(1)=0
Tsort(√n)= Tmerge(√n) + Tmerge((√n)/2) + Tmerge((√n)/4) + ... + Tmerge(2) + 0
= c*√n + c*(√n)/2 + c*(√n)/4 + ... + c * 1
= c* (√n + (√n)/2 + (√n)/4 + ... + 1)
$=c^* \sum_{i=0}^{ld\sqrt{n}} 2^i = \frac{2^{ld\sqrt{n}+1}-1}{2-1} = c^*(2\sqrt{n}-1) = O(\sqrt{n})$

Bitonic Sort solange aufteilen, bis nur 1 Element pro Folge, dann bitonic Merge
Komplexität:
T(n) = log n + T(n/2)
T(n) = log n + log(n/2) + ... + 1
= log n + (log n –1) + ... + 1
=(log n)(log n +1)/2
=O(log^2 n) Vergleicherstufen à n/2 Vergleicher

5. Parsitec

Aufbau	24 Knoten durch 3 versch. Netze miteinander verbunden
Knoten	PowerPC 133 MHz, eigener Hauptspeicher, Festplatte - 2 Entry-Knoten: 64MB, 2GB, 2 Ethernetkarten sind Verbindung zur Außenwelt - 12 Rechenknoten: Schnittstellen zu internen Netzen, 32 MB - 10 I/O-Knoten: wie Rechenknoten, aber PS/2, VGA, par. und ser. Port
Control Net	1 Kontrollboard pro Rack, ser. Verb. über C-Net, VT100 Konsole mit Racks verb. C-Net für Kontrolle, Reset und Booten der Konten
Ethernet	2 Hubs, über Entry Verb. mit IDA-Netz, für Administrationsaufgaben des OS sowie Synchronisation und Kontrolle paralleler Anwendungen
High Speed Link	75MB/s, 6 Router mit je 2 * 4 Ausgängen mit Crossbarswitch für Kommunikation von EPX (embedded parix) zwischen Knoten, Weiterleitung von Benutzereingaben/Ausgaben von/zu Entryknoten zu/von Rechenkonten
Parix	- Knoten logisch zu Würfel angeordnet - auf den Rechenknoten können Threads gestartet werden - links zw. Threads können etabliert, über sie gelesen/geschrieben, gebrochen werden

6. Höchstleistungsarchitekturen

Vektorprozessor hat zusätzlich Verarbeitungseinheiten für Vektoren und arithmetische und logische Operationen sowie zum vektoriellen Zugriff

Vorteile
1. Ergebnisse können in extrem tiefen Pipelines erzeugt werden (Komponenten unabh. voneinander)

2. weniger Anweisungen → geringe Befehlsbandbreite, weniger Pipeline stalls
3. Vektoren liegen auf bekannte Weise im Speicher → schneller gepipelineter Zugriff möglich
4. Kontrollhazard am Ende einer Loop (alle Stufen der Pipeline müssen ausgewertet werden, bis Branch entschieden werden kann) (die bei seriellem Zugriff nötig wäre) fallen weg

Anlaufzeit: Anz. Taktzyklen bis zum Erscheinen des ersten Ergebnisses

Initiierungsrate: Anz. Taktzyklen zw. Eingabe zweier aufeinanderfolgender Operandenpaare

Pipelinetiefe: $\text{Pipelinetiefe} >= \text{Gesamtlatenz in Zyklen} = \frac{\text{Gesamtlatenz der Einheit}}{\text{Zykluszeit}}$

Einschränkung: der Vektorlänge, d.h. VLR (Vektor-Length-Register) auf neue Länge setzen

Partionierung: längeren Vektor auf mehrere Vektoren verteilen

Schrittweite: normalerweise: Matrix zeilenweise im Speicher
Zugriff auf Spaltenvektor: Load/Store with Stride

Chaining: Ausführen des 2. Vektorbefehls bereits dann, wenn die erste Komponente des Ergebnissen des ersten Vektorbefehls vorliegt
Vorr.: Überall gleiche Initiierungsraten

SIMD-Architekturen

Hillis Thesen: Neuronen schneller als Transistoren (falsch)
Hirn hat mehr Neuronen (falsch)
Speicher meist passiv, Großteile der Gesamttransistoren ungenutzt
von-Neumann-Flaschenhals

Hillis Vision: Rechner, der CPUs mit individuellem Speicher vernetzt
Massive Parallelität und Konnektivität

Architektur CM-1:
- 64000 CPUs mit je 4Kbit Speicher in 4 Quadranten (10-dim. Hypercube) à 1024 Clusters (Mesh) mit je 16 CPUs
- je ein Microsteuerwerk versorgt die Quadranten mit Befehlen, sie sind über crossbar-Netz mit Vorrechner verbunden
- CPUs bitseriell mit lookup-Tabelle
- im Hypercube Paketrouting so, daß bei jedem Schritt eine 1 aus rel. Adresse verschwindet (Hot-Potatoe-Routing), „trotzdem“ rel. wenig Staus
- Programmierung: Paris (z.T. sehr aufwendige Makros (Matrixmult))

Probleme:
- Bitserielle Verarbeitung → schlechte FP-Leistung
- schlecht unterteilbar
- Routing Flaschenhals, Hypercube mit Leitungsproblemen behaftet
- nur MIMD macht sowohl Daten- als auch Kontrollparallelität möglich
- Spezialkomponenten

MIMD-Architekturen
- beliebige Unterteilbarkeit

CM-5
- sowohl Daten- als auch Kontrollparallelität möglich
- 2 getrennte Kommunikationsnetze (Daten- und Steuernetz)
- Beschleunigung durch Vektoreinheiten
- jeder Knoten kann Rechenkonten oder Kontrollprozessor CP (evtl. Partitionsmanager) sein
- Verbindungstopologie Fat-Tree: kleiner Durchmesser, große Schnittbreite, aber Struktur unübersichtlich
- Unterstützung Datenparallelität: über Steuernetz Abwicklung globaler Operationen: Broadcast, Replikation, Reduction, Permutation
- 2 Programme für CP und für Rechenknoten

Intel Paragon XP/S
- Mesh aus Paaren von i860 CPUS, verteilter Speicher ohne gemeinsamen Adressraum (16-32 MB)
- Intel: Router Flaschenhals → Knoten sind Router (iMRC) mit eigener CPU, an der der eigentliche Konten hängt, 16 Bit breite HS-Links
- Serviceknoten (machen Frontend-Rechner überflüssig), I/O-Konten (Massespeicher, ext. Netze) und Rechenknoten
- Wormhole-Routing: erst horiz., dann vertikal (Deadlock durch einsammeln des Schwanzes bei Blockade unmöglich)

Cray T3E
- 3D-Torus, Knoten aus Router mit dec alpha EV5 CPU mit 64 – 512 MB
- jeder Link in beide Richtungen 500MB/s
- Netzinterface hat 512 von außen zugreifbare E-Register, die Zugriff auf entfernten Speicher möglich machen→ quasi shared memory

Zusammenfassung
1. Topologie: weg vom Hypercube, hin zu lokaleren Netzen

2. Knoten: Standard-CPUs (evtl. mit Spezialhardware) statt proprietären CPUs, Router mit separater CPU
3. Partition: Partitionsmanager, Rechen-, I/O-, OS-Knoten
4. Unix, PVM
5. SPMD = Single Program Multiple Data: Rechenaufwand pro Kommunikationsaufwand -> Max
6. neue Tendenz: Beowulf-Machines (COTS)

Vektorrechner
- gut für technisch-wissenschaftliche Anwendungen
- Vorteil: Vektorisierende Compiler
- Nachteile: Speedup begrenzt (kein Par.recher), Vektorlänge begrenzt

SIMD
- lassen massivste Parallelität zu
- gut für Probleme, wi gleichartige Operationen auf großen Datenmenge ausgeführt werden müssen (Bildverarbeitung)
- aber: Eingeschränkt in der Programmierung (if-Statements, load balancing, iterative Verfahren)

MIMD
- am leistungsfähigsten
- moderate Parallelität (8-4000 CPUs, je sehr leistungsfähig)
- Vorteile: universell einsetzbar, steigen in der Leistung mit den CPUs
- Nachteile: bei intensiver Kommunikation langsam, teuer

7. Parallele Algorithmen

Matrixmultiplikation

```
for (k=0;k<p;k++)
  for (i=0;i<n;i++)
     for (j=0;j<m;j++)
       C[i,j]= C[i,j]+A[i,k]*B[k,j]
```

innere Schleifen parallelisierbar → O(n)

1. Feinkörnig: 1 Datum pro Knoten (Systola) (siehe auch oben)
2. Mittelkörnig: Teilmatrix pro Knoten
3. Grobkörnig: eine Zeile pro Knoten

Matrixmultiplikation ISA

```
< R1:=0; 1^n;1^n>          c[i,j]=0
for k := 0 to p-1 do begin
  < C := Cwest; 1^n;1^n>   senden der 1. Spalte von A
  < R2 = C; 1^n;1^n>
  < C := Cnorth; 1^n;1^n>  senden der 1. Zeile von B
  < R3 = C; 1^n;1^n>
  < R4:= R3*R2; 1^n;1^n>    a[i,k] * b[k,j]
  < R1 := R1 + R4; 1^n;1^n> C[i,j] + a[i,k] * b[k,j]
end;
```

Transitive Hülle: von G=(V,E) ist G*=(V,E*), wobei eine Kante e*=(v,w) genau dann in E*, wenn es einen Pfad von v nach w in G gibt

Warshall's Algorithmus

```
for (k=0;k<n;k++)
  for (i=0;i<n;i++)
     for (j=0;j<n;j++)
       A[i,j]= A[i,j] or (A[i,k]and A[k,j])
```

Beweisidee:

1. Alle Kanten, die eingetragen werden, sind in G*
2. alle Kanten in G* werden durch Alg. gefunden

Details s. Folien (5)

Parallelisierung: A(k+1)[i,j]= A(k) [i,j] or (A(k) [i,k]and A(k) [k,j])
→ innere 2 Schleifen können parallelisiert werden.

Realisierung Systola

```
for (k=0;k<n;k++)
{
  broadcast A[i,k] to all processors of row i;
  broadcast A[k,j] to all processors of column j;
  for_all (i=0;i<n;i++; j=0;j<;j++)
     A[i,j]= A[i,j] or (A[i,k]and A[k,j])
  ringshift all rows;
  ringshift all rows;
}
```

LAISA

```
< set 0, R4, eins; eins >
for k = 0 to n-1 do
```

```
begin
  < set CW, CR3; eins; [2..n] >
  < set CN, CR2; [2..n]; eins >
  < and R3, R2, R1; eins; eins >
  < or  R4, R1, CR4, eins; eins >
  < set CW, c; eins; [2..n] >
  < set CE, C; eins; [1..n-1] >
  < set CN, c; [2..n]; eins >
  < set CS, C; [1..n-1]; eins >
end
```

Kürzeste Pfade

G=(V,E,w) kantengesichteter Graph. Kürzester Pfad von i nach j ist derjenige, dessen Summe der Kantengewichte minimal ist.

Alg. von Floyd

d[i,j] = Kantengewicht, ∞ sonst

```
for (k=0;k<n;k++)
  for (i=0;i<n;i++)
     for (j=0;j<n;j++)
       d[i,j]= min( d[i,j],  d[i,k] + d[k,j])
```

am Ende ist d die Matrix der min. Pfadlängen

Beweis:

1. wenn am Ende d[i,j] < ∞, dann ex. Pfad von i nach j
2. d[i,j] enthält am Ende höchstens die Länge des kürzesten Pfades von i nach j
3. d[i,j] enthält am Ende mindestens die Länge des kürzesten Pfades von i nach j

Details s. Folien (6)

Implementierung analog zu Matrixmult oder Warshall

Minimaler Spannbaum

G=(V,E,w) kantengewichteter, ungerichteter (d symmetrisch) Graph. Ein min. Spannbaum ist ein kreisfreier Teilgraph T=(V, E‘) von G, so daß die Summe der Kantengewichte von E‘ minimal ist.

Satz min. Spannbaum

G=(V,E,w), U Teilmenge von V, dann gehört die Kante mit min. Gewicht, die einen Knoten von U mit einem Knoten von V-U verbindet, zum min. Spannbaum

Beweis:

Sei e nicht im Min. Spannbaum T=(V. Emin). → ex. e‘ in Emin, die U mit V-U verbindet mit w(e‘) >w(e). Ersetzt man e‘ durch e, erhält man Spannbaum mit geringerem Gewicht. → Widerspruch zur Annahme.

Alg. 1 min. Spannbaum

1. Alle Knoten sind Superkonten
2. Suche für alle Superknoten die min ausgehende Kante, füge sie dem Spannbaum hinzu.
3. Bilde aus den Zusammenhangskomponenten neue Superknoten
4. Gibt es noch mehr als einen Superknoten, gehe zu 2

Satz von Mags und Plotkin

(i,j) gehört zum Spannbaum ⇔ ex. kein Pfad von i nach j, dessen max. Kante kleinere Gewicht hat als (i,j)

Beweis:

1. (i,j) im min. Spannbaum, ex. Pfad mit Max Kante kleiner (i,j) → Widerspruch
2. (i,j) nicht im min. Spannbaum, ex. kein Pfad mit max. Kante kleiner (i,j) → Widerspruch

Alg. 2 min. Spannbaum

zu Beginn d(i,j) = Adjazenzmatrix, ∞ wenn kein Pfad

```
for (k=0;k<n;k++)
  for (i=0;i<n;i++)
     for (j=0;j<n;j++)
       d[i,j]= min( d[i,j],max( d[i,k], d[k,j]))
for (i=0;i<n;i++)
  for (j=0;j<n;j++)
    if w(i,j)=d[i,j] then TakeToSpanningTree(i,j)
```

bipatiter Graph

⇔ man kann Graph so teilen, daß alle Kanten vom einen Teile in den anderen führen ⇔ alle Kreise im Graph haben gerade Länge

Alg. bipatiter Graph

d[i,j] = Adjazenzmatrix, ∞ wenn a[i,j]=0

```
isBipatit := 1;
for (k=0;k<n;k++)
  for (i=0;i<n;i++)
     for (j=0;j<n;j++)
       d[i,j]= min( d[i,j],  d[i,k] + d[k,j])
```

```
for (k=0;k<n;k++)
  for (i=0;i<n;i++)
     for (j=0;j<n;j++)
       if d[i,k]=d[k,j]and(i,j)∈E then IsBipatit:=0
```

Brücke — ist eine Kante e in einem Graph G, wenn G nach Entfernen von e nicht mehr zusammenhängend ist.

Satz Brücken — eine Kanten e ist Brücke ⇔ e liegt nicht auf einem Kreis in G

Abstand — von 2 Knoten i und j d(i,j) ist die Anz. der Kanten, die man auf Pfad von i nach j mindestens durchlaufen muß

Satz Kreis — Kante (i,j) liegt auf Kreis ⇔

1. ex. Knoten k mit d(i,k) =d (k,j) oder
2. ex. Knoten k mit d(i,k) = d(k,j) + 1 mit erste Kante des Pfades von i nach k ist nicht (i,j)

Alg. Brücken

```
for (k=0;k<n;k++)
  for (i=0;i<n;i++)
     for (j=0;j<n;j++)
       begin
         if d[i,j] > d[i,k] + d[k,j] then
         begin
           d[i,j] = d[i,k] + d[k,j];
           first[i,j] = first[i,k];
           m[i,j] = m[i,k];
         end else
         if d[i,j] = d[i,k] + d[k,j] then
           m[i,j] = m[i,j] or m[i,k] or
                              first[i,j] != first[i,k]
        end;
for all i,j Bridge[i,j]:=true
for (k=0;k<n;k++)
  for (i=0;i<n;i++)
     for (j=0;j<n;j++)
       if (d[i,k] = d[k,j]) or
      (d[i,k] = d[k,j]+1 and m[i,k]) then
          Bridge[i,j]=false;
```

Wenn Kante keine Brücke ist, liegt sie auf irgendeinem kleinsten Kreis gerader oder ungerader Länge, die der Alg. dann findet.

Fouriertransformation (FT) — Umwandlung von Stützstellendarstellung eines Polynoms in Koeffizientendarstellung

Primitive Einheitswurzel — ω heißt n-te Einheitswurzel, wenn ω^n=1

ω heißt primitive n-te Einheitswurzel, wenn ω^n=1 und ω^k != 1 für alle 1<= k<n

Eigenschaften:

3. wenn n gerade, ω primitive n-te Einheitswurzel → ω^n/2=-1
4. ω primitive n-te Einheitswurzel →ω^2 ist primitive n/2-te Einheitswurzel

Diskrete FT — ω primitive n-te Einheitswurzel, die nxn Matrix F mit $F(i,j) = \omega^{i \cdot j}$ heißt Fouriermatrix. Sei a ein n-dim. Zeilenvektor → die Abb. f(a) = a * F heißt DFT

inverse DFT — ω primitive n-te Einheitswurzel, die nxn Matrix F^-1 mit $F^{-1}(i,j) = \frac{\omega^{i \cdot j}}{n}$ heißt inverse Fouriermatrix. Sei a ein n-dim. Zeilenvektor →die Abb. $f^{-1}(a) = a \cdot F^{-1}$ heißt inverse DFT

FFT — Schnelle Fouriertransformation

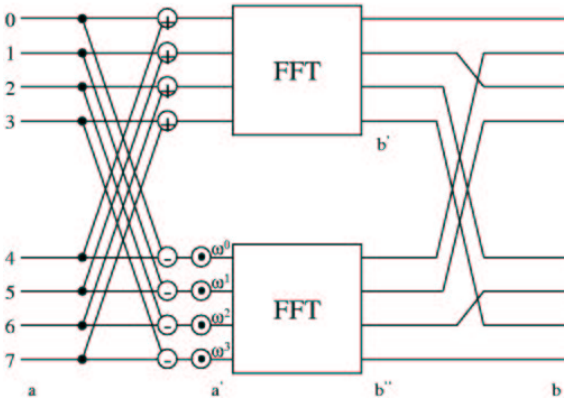
Idee: Einzelne Berechnungen der Matrix-Vektor-Multiplikation in spezieller Reihenfolge auszuführen, so daß auf bereits berechnete Ergebnisse zurückgegriffen werden kann. Dabei werden die Eigenschaften 3 und 4 von ω ausgenutzt. Dabei muß eine Zweierpotenz sein.

gerade Komponenten von b:

$$b_{k'} = b_{2k} = \sum_{i=0}^{n-1} a_i \omega^{i \cdot 2k} = \cdots = \sum_{i=0}^{n/2-1} (a_i + a_{i+n/2}) \omega^{i \cdot 2k} = \sum_{i=0}^{m-1} (a_i + a_{i+m}) v^{i \cdot k}$$

gerade Komponenten von b:

$$b_{k'} = b_{2k+1} = \sum_{i=0}^{n-1} a_i \omega^{i \cdot 2k+1} = \cdots = \sum_{i=0}^{n/2-1} (a_i - a_{i+n/2}) \omega^{i(2k+1)} = \sum_{i=0}^{m-1} \omega^i (a_i - a_{i+m}) v^{i \cdot k}$$



Komplexität seriell	m Additionen, m Subtraktionen und m Multiplikationen, sowie nochmals m Multiplikationen, um ω^\k aus ω^\k-1 * ω zu berechnen. Perfect Shuffle hinten auf Aufwand n →
	T(n)=3n + T(n/2), T(1)=0
	T(n) = 3n + 3n /2 + ... + 1=3n * log n → O(n log n)
Komplexität parallel	T(n)=T(n/2) + c, T(1)=0
	T(n) = c + c/2 + ... + 1=c* log n → O(log n)