Themengebiete „Parallele Algorithmen“

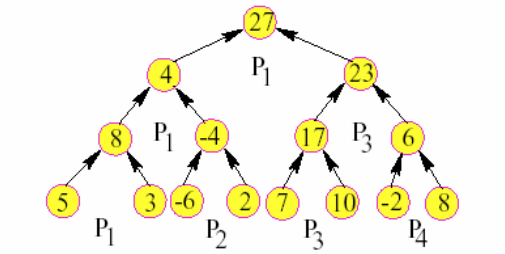
- **Grundlagen**
  - **PRAM Modelle**
    - Netzwerkmodelle
    - Speicherzugriffsarten
- **Präfixsummenalgorithmus**
  - Iterative Version
  - Rekursive Version
- **Pointer Jumping**
- **Sortieren mit Partitioning & Merging**
- **Berechnung des Maximums (alle auf Common CRCW PRAM)**
  - O(1) Zeit mit $O(n^2)$ Prozessoren ➔ $O(n^2)$ Workingkomplexität
  - O(log n) Zeit mit O(n/log n) Porzessoren ➔ O(n) Workingkomplexität
  - O(log log n) Zeit mit O(n) Prozessoren ➔ O(n log log n) Workingkomplexität
  - Accelerated Cascading
    - O(log logn) Zeit mit O(n) Workingkomplexität
- **List Ranking**
  - Basierend auf Präfixsummen ➔ O(n) Zeit mit log n Prozessoren ➔ O(n log n) Workingkomplexität
  - Basierend auf 2-Färbung ➔ O(log n log log n) Zeit in O(n) Workingkomplexität
  - O(log n) Zeit in O(n) Workingkomplexität
- **Euler Tour Technik**
- **Routing a Tree** (Rücktransformation der Euler Tour)
- **Auswerten eines Ausdruckes**
  - Tree Contraction
    - Rake-Operation
- **Parallele suche im sortierten Array**
- **Mergingverfahren**
  - Merging durch Ranking
  - Schnelles Merging
  - Workoptimales Merging
  - Sortieren mit Merging
- **Connected Components**
  - Pseudoforrest
  - Algorithmus für dichte Graphen [Adjazenzmatrix]
  - Algorithmus für dünne Graphen]Kantenliste]
- **Grafting**
  - Conditional Grafting
  - Unconditional Grafting
  - Mixed
- **2d-Konvexe Hülle Problem**
  - Insbesondere Berechnung der Common Tangente
- **Halbebenenschnittproblem**
  - Duality Transformation
  - 2 Variable Linear Programming Problem
- **Grundlagen**
  - Parallelisierbarkeit
  - P=NC Problem
  - P-Vollständigkeit
  - NC Reduzierbarkeit
  - Circuit Value Problem (P-Vollständiges Problem)

Fragenkatalog „Parallele Algorithmen“

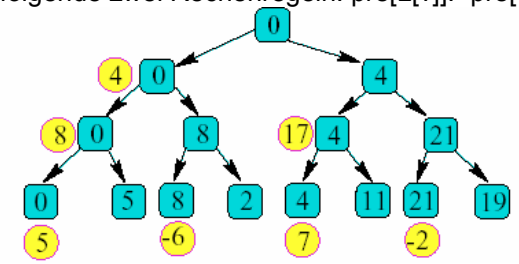
1. **Was ist die Working-Komplexität?** [Das Produkt aus Ausführungszeit (T(n)) und Anzahl der Prozessoren (P(n))]
2. **Wann ist ein paralleler Algorithmus Effizient?** [Wenn er in der Workingkomplexität eines anderen Algorithmus liegt. Sind zwei Algorithmen bzgl Workingkomplexität gleich, so ist der schnellere besser!]
3. **Wann ist ein Qualitätsmaß schwer zu bestimmten?** [wenn W1(n) > W2(n) aber T1(n)<T2(n)]
4. **Wann ist ein Algorithmus Work-Optimal bzw Work-Time-Optimal?** [Work-optimal: Wenn die Workcomplexity in der Zeitkomplexitaet des zugehörigen sequentiellen Allgorithmus liegt. $W(n)=O(T_{seq}(n))$ | Work-time-optimal: Wenn die Zeit des parallelen Alg. ($T_{par}(n)$) nicht verbessert werden kann.
5. **Was ist ein Parallelrechner?** [Ein Rechner mit mehreren Prozessoren der diese getrennt oder zusammen für bestimmte Aufgaben einsetzen kann]
6. **Welchen Sinn macht Parallelrechnen ? Wo liegen die Grenzen?** [Riesige Rechengebiete, die mit normalen Seq-Rechnern nur in hoher Zeit berechnet werden könnten (Protein folding, Wetterbericht). Wegen physikalische Beschränkungen im herkömmlichen VLSI Design
    - Minimisierung nur bis auf die größe eine Silizium Atoms theoretisch möglich. D.h. Grenze in ca. 15 Jahren erreicht, Mooresches Gesetz verliert dann seine Gültigkeit.
    - Lichtgeschwindigkeit: Elektronen brauchen Zeit um durch die Leiterbahnen zu laufen 3x10^8 m/s. Zeit um zwischen zwei Atomen „Daten“ auszutauschen ist 10^{-18} s bei vielen Tausen Transistoren steigt der Wert noch auf 10^{-15}s. => Grenze ist bei 1000 Teraflops erreicht. Aktuelle normale Rechner sind im Gigaflop bereich.]
7. **Was sind die Vorteile/Probleme von Parallelrechnern?** [Viele parallele Techniken bekannt, aber konventionelle von Neumann Architektur, die Soft- und Hardware ist heutzutage zu eng verwoben. Zu viele verschiedene Parallelrechnermodelle]
8. **Erkläre das PRAM Modell?** [Parallel Random Access Machine: N prozessoren greifen auf den gleichen Speicher zu und sollen zu jeder Zeit auf jede Zelel zugreifen können => evtl. Konflikte. Spezielle Lösung der Uni des Saarlandes => SB-PRAM. Lösung mittels zwischengeschalteter „Network“ Komponente. PRAM ist ein theoretische Modell, andere Modell zB Butterfly, hypercube und Vermaschung werden auch verwendet. Aber Algorithmen für **PRAM Modell ist einfach auf andere Systeme übertragbar]**
9. **Welche Parallelcomputer-Modelle gibt es?** [SIMD, MIMD, SISD, MISD]
    - ⇨ SIMD
        - Wichtigstes Modell zum Lösen von regulären Strukturen
    - ⇨ MIMD
        - Generelleres Modell für Probleme ohne reguläre Struktur
    - ⇨ SISD
        - Der normale sequentielle Rechner (KEIN Parallelrechner)
    - ⇨ MISD
        - Macht keinen Sinn
10. Was sind die wichtigsten Design-Aspekte für Network SIMD Modelle? Erkläre diese 3? [Communication Diameter, Bisection Width und Scalability]
    a. **Communication Diameter?**
        i. Größter Abstand zwischen zwei Knoten
        ii. Wenn Diameter=d => Ω (d)
    b. **Bisection Width?**
        i. Anzahl der Linkentfernungen zum Teilen des Graphen in zwei gleiche Teile
        ii. Je größer die Bisection Width, desto besser können die Knoten miteinander Kommunizieren, da es mehr Wege gibt
    c. **Scalability?**
        i. Einfachheit des Erweiterns des Graphen um einen weiteren Prozessor
        ii. Je größer die Bisection Width, desto mehr Links müssen beim Hinzufügen eines neuen Prozessor angelegt werden
        iii. Extremfälle: Vollständig vermaschter Graph, linearer Graph
11. **Welche Netzwerkmodelle gibt es?** [Mesh, Hypercube]
12. **Welche verschiedenen PRAM Modelle gibt es bzgl. des Speicherzugriffs?** [EREW,CREW,CRCW]

a. **Sind diese untereinander kompatibel?** [Schwächere laufen uneingeschränkt auf stärkeren Modell also eine Programm für EREW läuft immer auch auf einem CRCW Modell | Stärkere Modelle laufen nur mir zusätzlicher Zeit bzw. mehr Prozessoren simulativ auf einem schwächeren Modell]

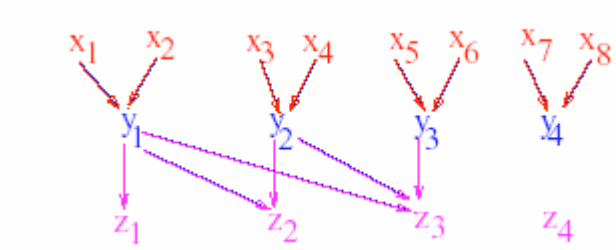
13. **Wann ist ein solches Modell schwächer als ein Anderes?** [Erst Zeit, dann Work]
14. **Was ist der Präfixsummenalgorithmus?** [Gegeben ist eine geordnete Liste von Elementen, auf welche ein beliebiger, binärer und assoziativer Operator angewandt werden soll. Z.B. „+".Bei + wäre das bei folgendem Set (5,3,-6,2,7,10,-2,8) das Ergebnis (5,8,2,4,11,21,19,27). Sequentiell ist dies einfach in O(n) Zeit zu berechnen.]
    a. **Wie geht das parallel?** [Schritt 1: Bottom-Up: Zunächst schreiben wir alle Zahlen in die Blätter eines Baumes. Jedem Knoten weisen wir einen Prozessor zu, der die



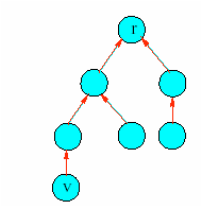
Summe der beiden Söhne berechnet:
Schritt 2: Nun rechnen wir weiter Top-Down: Die Präfixsumme des linken Sohnes ist immer die gleiche , wie die des Vaters, was zur Folge hat, dass der linke Pfad im Baum alles NULLEN enthält. Die Präfixsumme des rechten Sohnes eines Knotens ist die aus Schritt 1 berechnete Summe des linken Sohnes addiert mit der Präfixsumme seines Vaters. Es gibt also folgende zwei Rechenregeln: pre[L[v]]:=pre[v] und



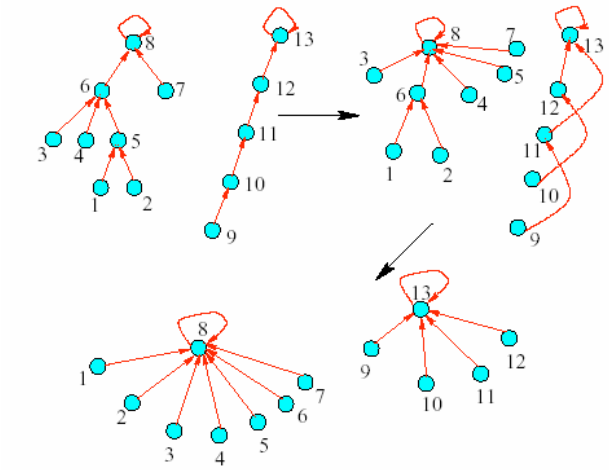
pre[R[v]:=sum[L[v]]+pre[v].
Das Ergebnis steht nun in den Blättern, allerdings um eins nach rechts geshiftet. Das ist aber nicht Schlimm, denn der letzte Wert steht in der Wurzel aus dem 1. Schritt.
    b. **Komplexität?** [Zeit: O(log n) | Prozessoren O(n) ➔ Prozessoren auf O(n/log n) reduzierbar wie? KEINE AHNUNG!]
15. **Algorithmus „Präfixsumme": Es gibt eine rekursive Variante, wie funktioniert diese?** [Rekursive Berechnung der Präfix Summen in jeder Hierarchiebene, also immer zwei Knoten addieren in jeder Ebene



Unterscheidung zwischen geraden und ungerade Indices.
Bei den geraden Indices müssen nur die Vorgänger aufaddiert werden. Bei den ungeraden Indices muss jeweil auf das ungerade x_i zugegriffen werden, welche nicht in den Vorgängerwerten enthalten ist.]
    a. **Zeitkompexlität?** [O(log n), da in jedem Schritt das Problem halbiert wird
    b. **Working Komplexität?** [O(n), da O(n/log n) Prozessoren]
16. **Wie geht der Algorithmus „Pointer Jumping"?** [Das Problem ist es, zu einem Knoten in einem Wald von Bäumen den zugehörigen Wurzelknoten zu finden. Zu diesem Zweck verwendet man sogenannte gerootete gerichtete Bäume (Rooted directed Tree). Bei denen von jedem Knoten ein Weg zu dessen Wurzel führt. Jeder Knoten hat also ausgangsgrad „1" in Richtung Wurzel. Die Wurzel selber hat keinen Ausgang sondern nur Eingänge



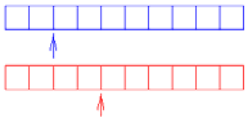
Der Pointer Jumping Algorithmus macht nun aus diesem Rooted directed Tree einene Stern, in welchem man den Wurzelknoten eines Knotens direkt ablesen kann, das geht so:



]

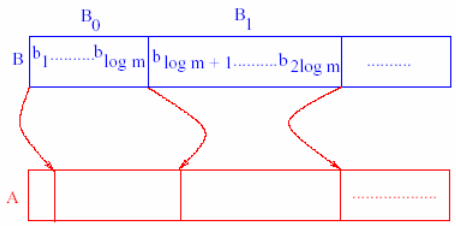
a. **Für was kann man den Algorithmus gut brauchen?** [z.B. für den Test ob zwei Knoten in der gleichen Zusammenhangskomponente sind]
b. **Welche Komplexität hat er?** [Zeit: O(log h) wobei h die Tiefe des Baumes ist | Prozessoren: O(n)]
c. **Welches PRAM Modell wird hier benötigt?** [CREW]

17. **Was versteht man unter der „Partitioning und Merging Strategie“?** [Das ist im Prinzip ein D&C Sortieralgorithmus nur für Parallelrechner, Der Divide Schritt nennt sich hier Partitioning Schritt, weil ja nicht rekursiv geteilt wird sondern die Aufgaben direkt an verschiedene Prozessoren verteilt wird. Das Merging ist ist sequentiell gesehen ein reines Mergesort,
    a. **Wie geht das Sequentiell?** [d.h. 2 sortierte Listen werden über zwei Zeiger miteinander verschmolzen:



Zwei Pointer durchlaufen die Listen, die Elemente werden paarweise verglichen und der betreffende Pointer inkrementiert. Die Ergebnisliste wir in ein neues Array geschrieben. Wenn List 1 länge n hat und Liste 2 länge m, dann hat die Ergebnisliste entsprechende Länge n+m => Die Komplexität ist O(n+m)]

    b. **Wie überträgt man das ins Parallele?** [Wir teilen das Gesamtproblem in viele kleine Teile auf, die man dann direkt sequentiell löst. Genauer gesagt wir teilen eine der beiden Listen in log m große Teile auf z.b. Liste B. Das letzte Element einer jeden Teilliste steht immer an Position i*log m. Hat man die Liste B aufgeteilt, weißt man jedem Endelement der Liste einen Prozessor zu, der die Liste A mit binärer Suche bzgl. dieses Endelemtentes aufteilt. Zwei korrespondierende Listen nennt man „Matching Blocks“

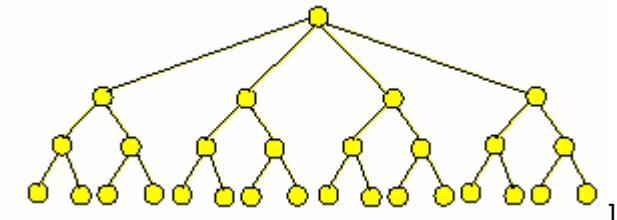


Sollte der Fall auftreten, dass es nun Teillisten in A gibt, die größer als log m sind, dann werden diese Teillisten wieder in log m große Teile aufgeteilt und diesmal die Listen aus B bzgl A geteilt. Nun haben wir sicher gestellt, dass die Teillisten höchsten

log m Größe haben. Dies geht in O(log log n) Zeit). Jetzt fehlt nur noch das sequentielle Merging, bei dem jeder Prozessor seine 2 Listen mit log m Elementen sequentiell merged.

c. **Mit welcher Komplexität geht das alles?** [O(log n) Zeit und O(n) Prozessoren]
d. **Welches PRAM Modell wird benötigt?** [CREW, da beim Teilen der Listen auf die gleichen Daten zugegriffen wird]

18. **Welche verschiedenen Lösungsansätze gibt es zur „Berechnung des Maximums", d.h. größtes Element in einer Liste?** [Einen O(1) Algorithmus, der leider O(n²) Prozessoren benötigt | Einen Workoptimalen Algorithmus mit O(n) Workingkompl. | Und einen recht schnell in O(log log n) Zeit]
    a. **Wie funktioniert der O(1) Algorithmus?** [Erezeuge ein Boolean Array, mit p Elementen, initialisiert mit „1". Weise jedem Element der Liste mit p Elementen p Prozessoren zu (also insgesamt p² Prozessoren). Jeder Prozessor berechnet genau einen Vergleich, so dass letztendlich jedes Element mit jedem in O(1) verglichen wird. Der Prozessor markiert sein Element jeweils mit „0" im Boolean Array, falls dieses kleiner ist, als dasjenige mit dem er es verglichen hat. Am Ende bleibt natürlich genau ein mit „1" markiertes Element zurück"]
        i. **Komplexität?** [p² Prozessoren, aber in O(1) Zeit]
        ii. **Welches PRAM Modell wird benötigt?** [Common CRCW ➔ Alle lesen und schreiben auf der gleichen Liste zur gleichen Zeit. Die Werte die geschrieben werden sind aber immer die Gleichen („0"), weshalb das Common Modell ausreicht]
    b. **Wie funktioniert der Work-Optimale Algorithmus?** [Alle Elemente der Liste werden in die Blätter eines Baumes geschrieben, an jedem Knoten sitzt ein Prozessor (also O(n) Stück). Gerechnet wird in log n Schritten, die untersten Prozessoren fangen an und vergleichen jeweils ihre beiden Elemente und schreiben das größere der Beiden in ihren Knoten. Die Prozessoren weiter oben verfahren gleich, bis letztendlich der letzte Prozessor die Wurzel berechnet, in welcher das größte Element gespeichert ist]
        i. **Komplexität?** [Zeit: O(log n) Workingkompl.: O(n log n) bzw. O(n) Prozessoren: O(n) bzw. O(n/log n)]
        ii. **Wie kann man die Anzahl der Prozessoren noch verringern?** [Die untersten Prozessoren haben ja nach dem ersten Schritt nichts mehr zu tun und können eigentlich die nächste Ebene berechnen, die Anzahl der benötigten Prozessoren halbiert sich ja in jeder Ebene]
    c. **Wie funktioniert der O(log log n) Zeit Algorithmus?** [Ähnlich wie der Zweite, aber mit einem komplexeren Baum. Die Wurzel hat nicht 2 sondern SQRT(n) Kinder, bei den Knoten weiter unten nimmt die Anzahl der Kinden jetzt immer weiter ab, bis man im k-ten Level schließlich nur noch 2 Kinder hat. Natürlich sind in diesem Fall jetzt die paarweisen vergleiche pro Knoten nicht mehr möglich, weshalb man hierzu jetzt an jedem Knoten den O(1) Algorithmus anwendet

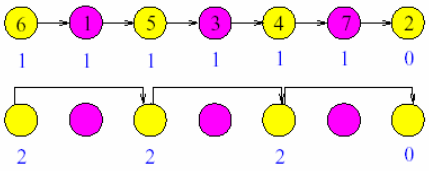


]
        i. **Komplexität?** [O(log log n) Zeit O(n) Prozessoren]
        ii. **Benötigtes PRAM Modell?** [Common CRCW]
    d. **Was versteht man unter Accelerated Cascading?** [Die kombination zweier Algorithmen, durch Ausnutzung der Vorteile beider]
    e. **Wie erhält man damit einen Algorithmus zur Berechnung des Maximums, der sowohl Zeit als auch Workoptimal ist?** [Wir kombinieren den Workoptimalen, aber langsamen zweiten Algorithmus mit dem nicht workoptimalen aber sehr schnellen dritten Algorithmus. Wir beginnen mit dem Workoptimalen Algorithmus und reduzieren das Problem bis auf bis Level (log log log n), dann kommt der schnelle Algorithmus zum Einsatz, welcher die restlichen Elemente bearbeitet.
        i. **Komplexität?** [O(log log n) Zeit und O(n) Work)
        ii. **PRAM Modell?** [Common CRCW]
19. **Was ist List Ranking?** [Gegeben ist ein lineare Liste (Successor Array), das Problem ist es, festzustellen, welchen Rang bzgl. des Endelementes die übrigen Elemente habe. Das

Endelement hat Rang „0“, das vorletzte Rang „1“ usw.. Sequentiell ist das Problem in O(n) Zeit lösbar]

a. **Wie geht der O(n) Zeit und O(n log n) Work Algorithmus?** [Einfach aber leider Suboptimal! Der Algorithmus ist im Prinzip der Präfixsummenalgorithmus. Alle Elemente bis auf das Endelement werden mit „1“ initialisiert und vom Endelement her die Präfixsummen berechnet]
   i. **Welche Komplexität hat er?** [O(n) Zeit und O(n log n) Work]
   ii. **Benötigtes PRAM-Modell?** [CREW]

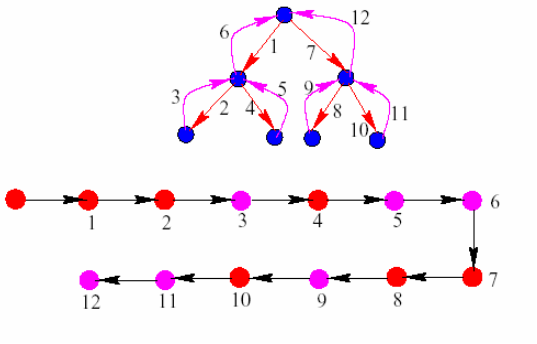
b. **Eine verbesserte Variante geht in O(log n log log n) Zeit und in O(n) Work. Wie geht das?** [3 Schritte:
Schritt 1: Bei der Initialen Liste wird auch wieder jedes Element bis auf das Endelement mit Rang „1“ initialisiert. Wir möchten eine Liste mit n/logn Knoten. Dazu verwenden wir Zweifärbung und teilen die Listen sukzessive mittel 2-Färbung immer weiter durch Zwei. Zweifärbung geht nach Theorem parallel in O(log n) Zeit) und O(n) Work. Wir müssen nun also log log n mal den Zweifärbungsalgorithmus ausführen. Wann immer eine Liste aufgeteilt wird, werden die verbleibenden Knoten um den Rang der entfernten Knoten erhöht:



Nun erzeugen wir zu unserem vorhandenen Successor Array noch ein „Predeccesor Array“ P und ein Array U, in welchem wir alle Informationen zu den entfernten Knoten speichern, also Knotenindex, Vorgänge, Nachfolger und Rang……??????????????????]

c. **Die beste Variante geht in O(log n) Zeit und in O(n) Work, was wurde dabei noch verbessert?** [???]

20. **Was ist die „Euler Tour Technik“?** [Der Hintergrund ist es, parallele Tiefensuche in einem Baum effizient auf einem Parallelrechner durchführen zu können. Eigentlich muss man dazu im Baum aber sukzessive durchlaufen. Die Lösung ist, den Baum in eine Liste umzuwandeln, denn es gibt sehr gute parallele Algorithmen für Listen. Die Eulertour beschreibt also eine Tour durch einen Baum, die sämtliche Knoten abläuft unter Zuhilfenahme zusätzlicher Kanten (für die Rückwege!)



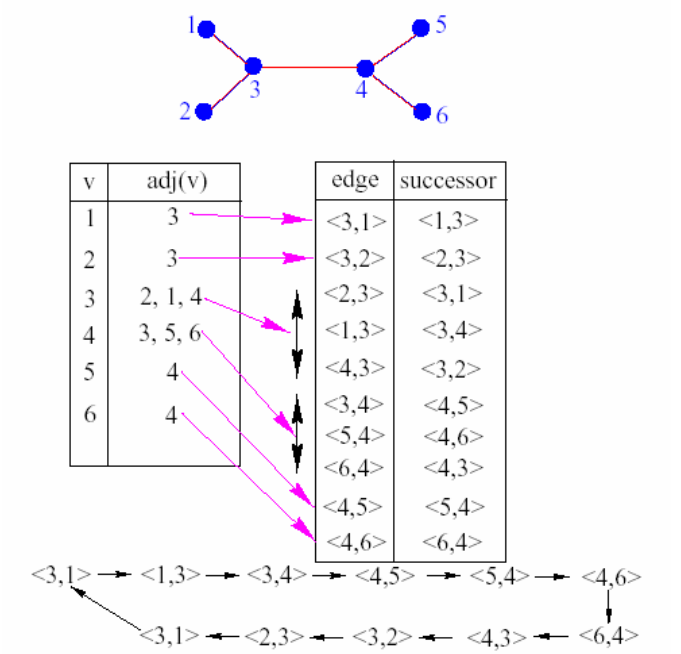
.

a. **Wie ist die genaue Definition eines Eulerkreises?** [Ein Eulerkreis ist eine Tour den Baum entlang, bei dem jede Kante des Graphen genau einmal besucht wird und man am Ende wieder am Ausgangspunkt angelangt ist. Jeder Knoten wurde dann passiert. Ein Eulerkreis ist dann möglich, wenn jeder Knoten gleichen Indegree wie Outdegree hat]

b. **Wie kann man die Eulertour sequentiell erzeugen?** [Mit Tiefensuche erzeugt. Die Liste ist zyklisch (➔ eben eine Euler Kreis). Bei n Knoten im Baum, besteht die Liste aus 2n-2 Knoten, wegen der zusätzlichen Kanten. Jede Kante hat genau eine Nachfolgerkante. Wir speichern zu jedem Knoten seine Nachbarn in einer Adjazenzliste ab. Den Nachfolger der Kante erhalten wir mit

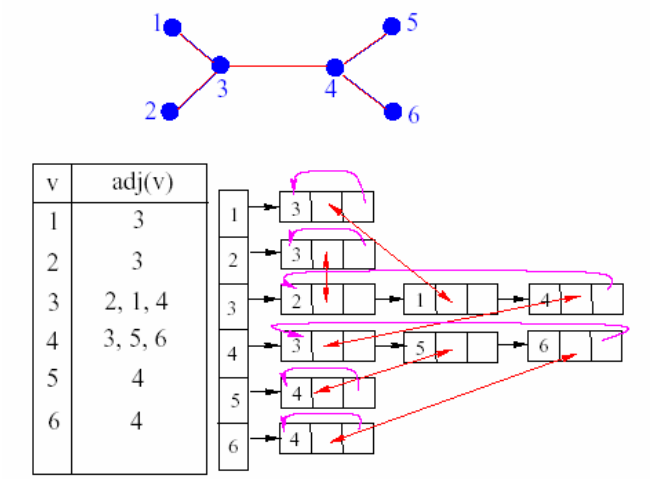
$$s(< u_i, v >) =< v, u_{(i+1) \bmod d} >, 0 \leq i \leq (d-1)$$

Praktisch geht das nun so:



]

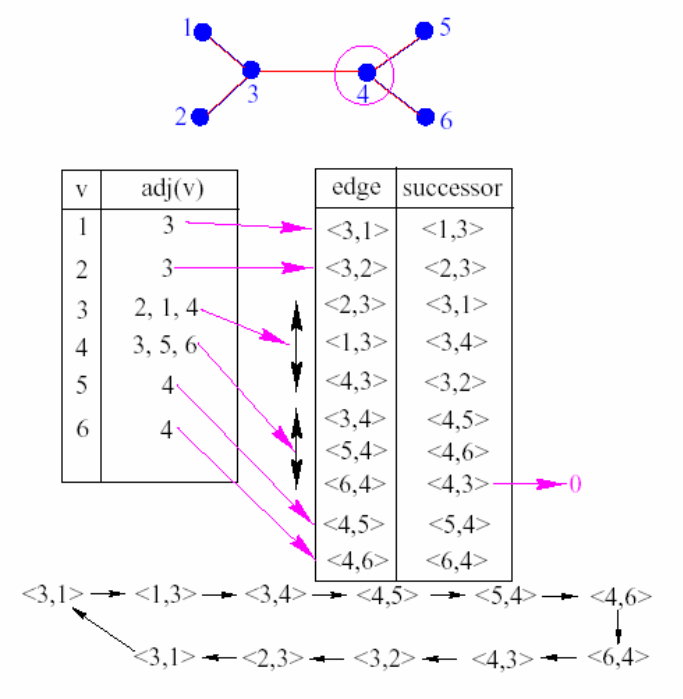
c. **Wie kann man das nun parallel erledigen?** [Wir gehen davon aus, dass wir den Baum als ein Set von Adjazenzlisten gegeben haben



In der Liste müssen zu jedem Knoten die Nachfolgerknoten gespeichert sein. Wir weisen dann jedem Knoten der Adjazenzliste einen Prozessor zu]

i. **Welche Komplexität haben wir hier?** [O(1) Zeit und O(n) Prozessoren]
ii. **Benötigtes PRAM-.Modell?** [EREW genügt, da kein concurrent reaf/write]

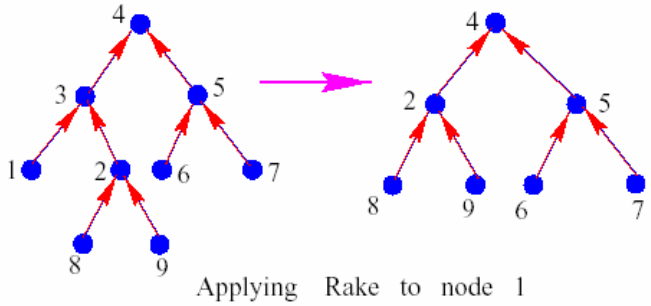
21. **Wir haben für die Euler-Tour-Technik auch eine Umkehrung kennengelernt, „Rooting a Tree“ – Wie funktioniert diese Technik?** [Wir möchten nun eine Eulertour wieder in einen Baum verwandeln und müssen aber sagen, welchen Knoten wir als Wurzel haben wollen



Wir nehmen z.B. Wurzel 4 und splitten die Kanten <6,4> und <4,3>. Nun haben wir wieder einen Baum.

a. **Wie geht der Algorithmus genau?** [Aufsplitten der Euler Tour, in dem wir die zu löschende Kante „0“ setzen. Alle übrigen Kanten werden auf „1“ gesetzt und parallel die Präfixsummen berechnet. Nun haben wir sehr schön die Reihenfolge der Vorgänger und können diese bequem auslesen]
b. **Was kann man sehr gut nun mit Hilfe beider Verfahren berechnen?** [Post-/Pre-/In-Order, Level eines Knotens und die Anzahl der Nachfolger]

22. **Was ist Tree Contraction?** [Es gibt Bäume, für die eine Eulertour völlig ungeeignet ist, z.B. Bäume mit arithmetischen Ausdrücken. Beim Bäume zusammenzuziehen kommt die Rake-Operation zr Anwendung!
    a. **Wie funktioniert die Rake-Operation?** [T=(V,E) sei ein Binärbaum mit Wurzel und für jede Kante v ist p(v) der Vorgänger. Sib(v) ist das Kind von p(v) und das Geschwisterkind von v. Für ein Blatt u mit p(u) ungleich r tue folgendes:
        i. Entferne u und p(u) von T
        ii. Verbinde sib(u) mit p(p(u))

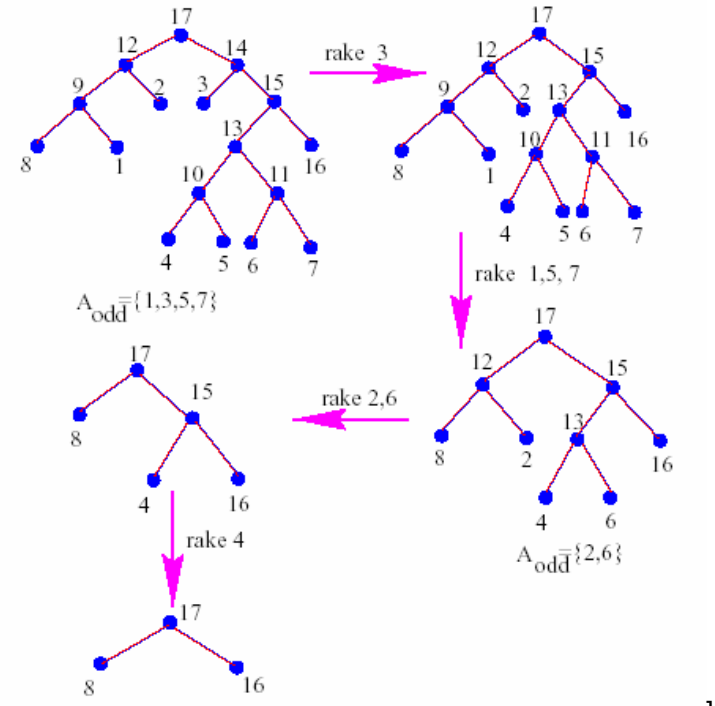


Applying Rake to node 1

In Worten: Rake angewandt auf ein Blatt u bewirkt, dass das Blatt und sein Vater entfernt werden, der Geschwisterknoten wird mit dem Großvater verbunden! Bei jeder Rakeoperation wird also ein Blatt und ein innerer Knoten entfernt]

    b. **Wie funktioniert Tree-Contraction nun genau?** [Alle Blätter bis auf das linkeste und rechteste werden von links nach rechts durchnummeriert und in zwei Arrays A_{odd} und A_{even} gespeichert. Das erzeugen dieser Arrays geht in O(1) Zeit bei n Prozessoren. Jetzt kommt die Rake-Operation: Parallel werden alle linken Kinder die im A_{odd} Array gespeichert sind „geraked“, dann alle restlichen im A_{odd} Array. Genau das gleiche machen wir nun im A_{even} Array, allerdings fängt man mit den rechten Kindern an und nimmt dann den Rest. Übrig bleibt ein Baum aus 3 Knoten bestehend aus der Wurzel

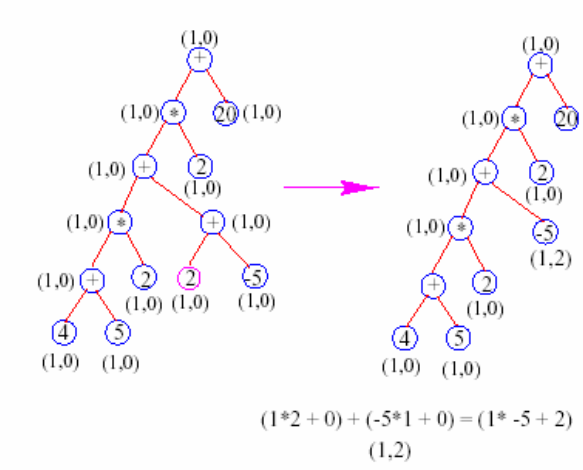
und den zuvor weggelassenen 2 Knoten, die nicht in die Arrays A_{even} und A_{odd} kamen.



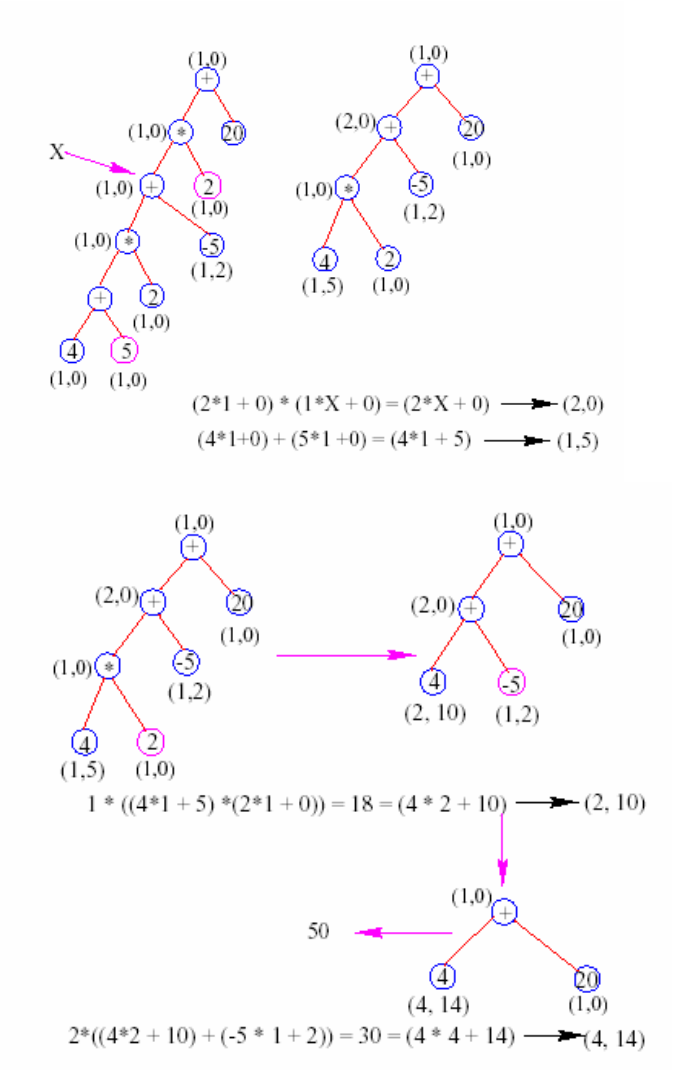
]

c. **Welche Komplexität hat die Tree Contraction?** [O(log n) Zeit und O(n) Work]

23. **Algorithmus zur „Auswertung eines Ausdruckes“, dargestellt als Baum: Wie funktioniert das?** [Jetzt kommen die eben besprochenen Techniken zur Anwendung. Man könnte den Baum auch Bottom-Up durchrechnen, würde dazu aber O(n) Zeit benötigen. Besser geht es wenn wir Tree-Contraction anwenden und zwar in O(log n) Zeit! Z.B.:

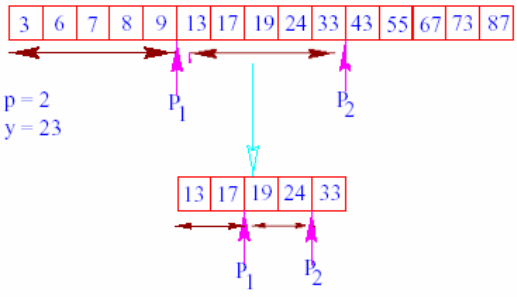


Jeder Knoten wird mit (1,0) initialisiert. Knoten „2“ markiert und der Level Hochgezogen. Die erste Zahl in der Klammer bezeichnet den Multiplikator und die zweite Zahl den Summanden der auf die Multiplikation aufaddiert wird s.o.. Gemäß der Rake Operation wird nun der ganze Baum zusammengezogen bis man das Endergebnis hat. (3er Baum aus dem man das Endergebnis direkt berechnen kann)



Das Ergebnis ist also 50.]

24. **Algorithmus „Parallele Suche in einem sortierten Array“: Wie geht das?** [Eingabe ist ein sortiertes Array von Elementen und ein Queryelement y. Wir haben p Prozessoren zur Verfügung. Wir unterteilen das Array in p+1 Teile und weisen den ersten p Teilen je einen Prozessor zu. Der Prozessor schaut nur, ob sein letztes Element größer oder kleiner dem gesuchten ist. Auf diese Weise finden wir leicht das nächste Intervall in dem y ist. Hat dieses Intervall noch mehr als p Elemente, wird es wieder in p+1 Teile unterteilt, usw. Besteht ein Intervall nur noch aus p Elementen, weisen wir jedem Element direkt einen Prozessor zu und finden das gesuchte y damit

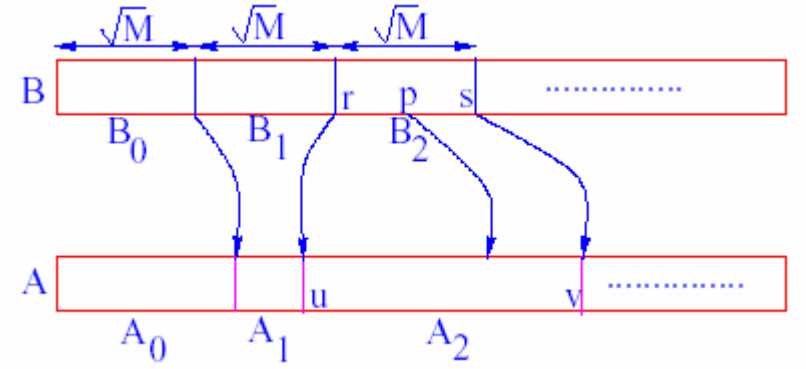


]

    a. **Welche Komplexität hat dieser Algorithmus?** [$O(\frac{\log(n+1)}{\log(p+1)})$ Zeit bei p Prozessoren]

25. **Welche Mergingverfahren haben wir dabei kennengelernt?** [Merging mit Ranking | Schnelles Merging | Workoptimales Merging]
    a. **Wie geht Merging mit Ranking?** [Berechne den Rang jedes Elementes der zwei sortierten Listen. Berechne daraus den Rang der Elemente in der Ergebnisliste]

b. **Wie geht das schnelle Merging?** [Eingabe sind zwei sortierte Listen A und B. Das geht so:
- Aufteilen von Array B in SQRT(m) Teile mit jeweils SQRT(m) Elementen
- Ranken des jeweils letzten Elementes der Teillisten von B in A ➔ Dadurch erhalten wir korrespondierende Listen von B zu A. Das geht mit m Prozessoren in O(1)
- Die korrespondierenden Blöcke werden nun paarweise und rekursiv gemerged ➔ D&C: Der jeweils größere Teilblock wird wieder in SQRT(m) Blöcke unterteilt und dann auch wieder die korrespondierenden Listen berechnet usw.



Ist das Subproblem klein genug, wird es sequentiell in O(1) gelöst. Jeder der p Prozessoren macht also Parallel im Prinzip das Gleiche nur auf anderen Teilblöcken. Am Ende wissen wir die Ranks von A bzgl b und umgekehrt und können daraus die Elemente in sortierter Reihenfolge aufschreiben]
    i. **Welche Komplexität hat „schnelles Merging“?** [O(log log m) Zeit bei O(m+n)log log m) Workingkomplexität]
c. **Wie funktioniert Work-Optimales Merging?** [Work-Optimal heisst erstmal, den Algorithmus auf eine Workinkomplexität von O(m+n) zu bringen.
- Wir haben zwei Listen mit n Elementen.
- Diese beiden Listen teilen wir in n/log log n Blöcke auf mit jeweils log log n Elementen
- Die jeweils letzten Elemente der Blöcke schreiben wir in zwei Liste A’ und B’ und berechnen die Ranks von A’ bzgl. B’ und umgekehrt
- Nun berechnen wir noch die Ranks von A’ bzgl B um zu sehen, in welchen Blöcken von B die Elemente aus A’ liegen. Für B’ bzgl. A berechnen wir das auch. Im Prinzip haben wir nun wieder korrespondierende Listen wie beim schnellen Merging. Die Listen nun können wieder rekursiv weiter aufgeteilt werden bis alles gemerged ist ➔ Der Algorithmus ist so kompliziert, dass man alleine 20 Minuten zu erklären brauchen würde, sodass hier nur die Kurzfassung steht]
d. **Welche Komplexität hat er nun?** [O(log log n Zeit bei O(n) Work und O(n/log log n) Prozessoren]

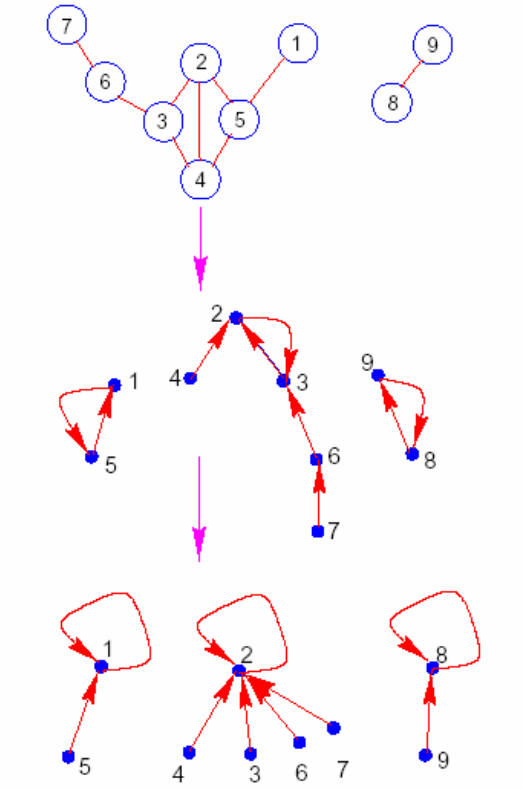
26. **Wie kann man effizient sortieren?** [mit dem vorherigen Mergingalgorithmus. In dem wir unser unsortierte Array in einen Binärbaum schreiben und auf Blattebene anfangen zu sortieren mit Workoptimalem Mergesort. Wir arbeiten und nun hoch bis zur Wurzel, in welcher dann das sortierte Array am Ende steht]
    a. **Welche Komplexität hat dieser Algorithmus?** [O(log n log log n) Zeit, weil unser Baum ja log n Tiefe hat und das Workoptimale Merging pro Schritt O(log log n) Zeit braucht. Die Workingkomplexität ist dementsprechend auch um den Faktor log n größer also beim Merging direkt]
27. **Was versteht man unter verbundenen Komponenten?** [Zwei Knoten sind in der gleichen Zusammenhangskomponenten, wenn sie gleich sind, oder wenn es einen Pfad vom einen zum anderen gibt. ➔ Diese Rekation ist eine Aquivalenzrelation]
28. **Wir haben zwei Algorithmen kennen gelernt, die diese Problem lösen, wie sind jeweils die Eingaben zur Berechnung?** [Adjazenzliste bei dichten Graphen ➔ Nachteil n² Einträge | Kantenliste ➔ Sehr gut bei dünnen graphen mit wenig Kanten]
    a. **Was ist ein Pseudo-Forrest?** [Ein gerooteter gerichteter Graph in dem jeder Knoten Outdeg=1 hat. Die Wurzel zeigt auch auf einen Knoten, das heisst es existiert in jedem Pseudo Forrest genau ein Zyklus]
    b. **Wie kann man beweisen, dass ein Pseudoforrest nur einen Zyklus hat?** [Einfach: Zwei Zyklen bedeuten mindestens einmal Outdeg=2, was nicht erlaubt ist. qed]
    c. **Wie konstruiert man einen Pseudoforrest?** [Wir suchen uns iterativ Adjazente Knoten und mergen sie zu immer größer werdenen Gruppen zusammenhängender

Knoten. Es entstehen nun lauter gerichtete Pseudo-Forrests, wobei die Wurzel eines jeden Baumes der Repräsentant der zugehörigen Zusammenhangsliste ist.]

29. **Wie funktioniert der Algorithmus für dichte Graphen zur Berechnung der Zusammenhangskomponenten?** [

	1	2	3	4	5	6	7	8	9
1	0	0	0	0	1	0	0	0	0
2	0	0	1	1	1	0	0	0	0
3	0	1	0	1	0	1	0	0	0
4	0	1	1	0	1	0	0	0	0
5	1	1	0	1	0	0	0	0	0
6	0	0	1	0	0	0	1	0	0
7	0	0	0	0	0	1	0	0	0
8	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	0	1	0

Gegeben ist eine Adjazenzmatrix, aus welcher wir mit Hilfe des Pseudoforrests, die einzelnen Sets von zusammenhängenden Knoten berechnen. Wenn wir nun diese Pseudoforrests haben, dann werden diese zu Sternen geschrumpft mit Hilfe von Pojnter Jumping (wurde etwas modifiziert, damit das mit dem Zyklus klappt). Danach ist jeder Knoten repräsentiert durch die Wurzel seines Sternes. Der Pseudoforrest liefert nun zwar zusammenhängende Sets, man kann aber nicht sagen, ob vielleicht zwei dieser Set auch wieder zusammenhängend sind, und genau das muss man nun noch überprüfen. Das geht in dem man die Kinder der Teil-Set alle miteinander vergleicht:



oben sind das die Vergleiche: (5,4),(5,3)…(5,7),(4,9)…(7,9),(5,9) => die Sterne 1 und 2 sind z.B. über die Kante zwischen den Knoten 5 und 4 miteinander verbunden (siehe ursprüngliche Adjazenzmatrix))]

a. **Welche Komplexität hat dieser Algorithmus?** [O(n²) Work (wegen der Adjazenzmatrix) und O(log n) Zeit (wegen Parallelberechnung mit?! n²/log n Prozessoren]
b. **Welches PRAM Modell kommt hier zum Einsatz?** [Common CRCW, wobei man das in ein CREW Modell umwandeln könnte]

c. **Geht das nicht schneller als mit O(n²) Work?** [Bei Dichten Graphen, wegen der Adjanzenmatrix nicht. Bei dünnen Graphen kommen Kantenlisten zum Einsatz, da ist es besser]

30. **Was ist Grafting?** [Auch eine Methode Bäume zu kombinieren die verbunden sind]
    a. **Wie kann man einen Stern erkennen?** [Ob eine zusammenhängende Komponente ein stern ist kann man herausfinden, indem man jedem Knotenv einen Prozessor zuweisst, der folgendes macht:

Set $Star(v) = true$

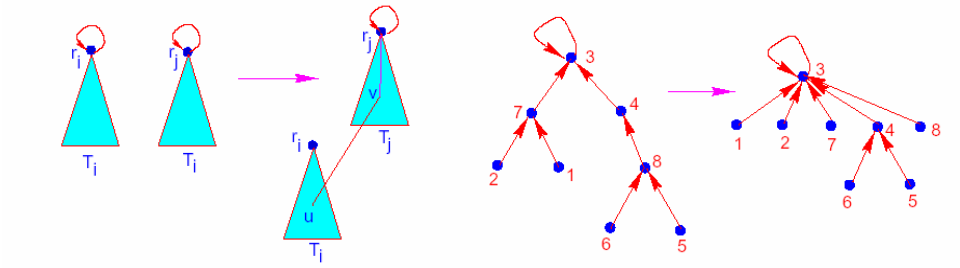
if $D(v) \neq D(D(v))$ then

Set $star(v), star(D(v)), star(D(D(v))) := false$

Set $star(v) := star(D(v))$

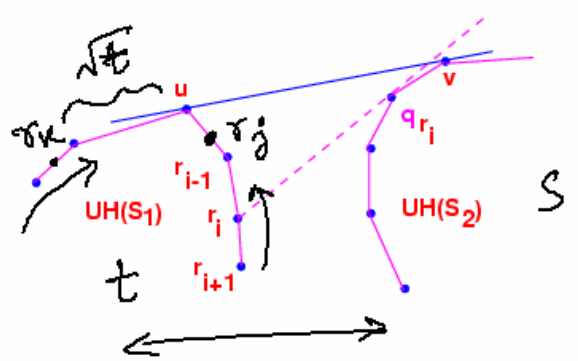
Liefert einer der Prozessoren Star(v)=true zurück, ist der zu v gehörende Graph ein Stern]

    b. **Was ist unconditonal Grafting?** [Will man zwei Bäume des Pseudoforrest miteinander graften macht man dies mittel D(ri)=v, man haengt also den einen Baum unter den anderen. Die Bäume sind jetzt recht hoch, diese reduzieren wir durch Pointer Jumping:

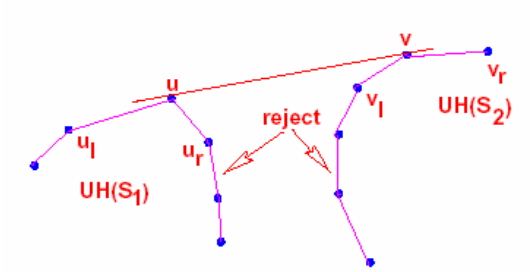


…]
    c. **Was ist conditional Grafting?** […]

31. **Was ist das 2D Konvexe-Hülle Problem?** [Die Konvexe Hülle einer Punkte menge sind diejenigen Punkte die verbunden mit einem einfachen Kantenzug alle anderen Punkte beinhalten. Anders ausgedrückt ist ein Punktepaar auf der konvexen Hülle eines Sets, gdw. alle anderen Punkte auf einer Seite der Verbindungsgeraden liegen. Die konvexe Hülle ist eindeutig. Die Punkte auf der konvexen Hülle nennt man auch Extrempunkte]
    a. **Kennen Sie einen optimalen parallelen Algorithmus zur Lösung?** [Da sich das komplexe Hülle Problem auf einfaches Sortieren zurückführen lässt und sortieren in O(n log n) Zeit gibt, ist ein paralleler Algorithmus dann optimal, wenn seine Workingkomplexität in O(n log n) liegt. Einen solchen Algorithmus gibt es!]
    b. **Wie konstruiert man damit die konvexe Hülle?** [Wir teilen zunächst die Punktemenge auf in eine obere Punktemenge und eine untere Punktemenge. Dazu suchen wir uns den linkesten und den rechtesten Punkt und verbinden diese. Diese gerade bildet die Grenze zwischen oberer und unterer Punktemenge. Wir müssen nun für die beiden Teilmengen die Obere bzw. die untere Hülle berechnen. Wir berechnen zunächste die obere Hülle. Die untere Hülle berechnet sich analog. Wir haben einen D&C Algorithmus, also haben wir 2 Phasen. Zuerst sortieren wir die Punkte nach ihren X-Koordinaten. In der Top-Down Phase unterteilen wir die Punktemenge nun rekursiv in zwei Teile. Im Basisfall berechnen wir die Konvexe Hülle und beginnen mit der Bottom-Up Phase und mergen die Hüllen sukzessive zusammen. Das Hauptproblem ist es nun, eine gemeinsame Tangente zu berechnen, die die beiden Hüllen verbindet. Nehmen wir an die Punktemenge von $UH(S_1)$ heisst t und die von $UH(S_2)$ heisst s. Wir teilen die Punkte nun in SQRT(s) bzw. SQRT(t) Intervalle auf. Zu jedem Punkt in t berechnen wir eine Tangente zu $UH(S_2)$, soweit dies möglich ist! Eine dieser Tangenten ist nun sicher die Common Tangente, unsere Suche also begrenzt. Die Common Tangente ist nun genau diejenige Tangente, für die beide Nachbarn beider Punkte der Common Tangente unterhalb der Tangente liegen



Als letztes müssen wir nun nur noch die inneren Punkte unterhalb der Common Tangente entfernen

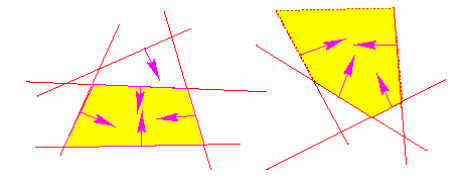


]

c. **Welche Komplexität hat dieser Algorithmus?** [Wir haben SQRT(t)*SQRT(s)=O(n) Prozessoren
   - Im Divide Schritt das Aufteilen geht in O(log n)
   - Berechnen der Tangenten geht in O(1)
   - Finden der Common Tangente in O(1)
   - Test auf Common Tangente in O(1)
   - Entfernen der Punkte in der Mitte in O(1)

   Insgesamt haben wir also O(log n) Zeit und O(n log n) Work]
d. **Welche PRAM Modell wird benötigt?** [CREW wegen paralleler Suche ➔ EREW möglich (!?)]

32. **Was ist Halbebenen-Schnitt?** [Halbebenen sind durch eine gerade begrenzte Flächen in der Ebene. Existieren mehrere solche Halbebenen, dann schneiden sich diese normalerweise und grenzen eine Fläche ein. Diese Fläche ist immer konvex, muss aber nicht zwingend geschlossen sein

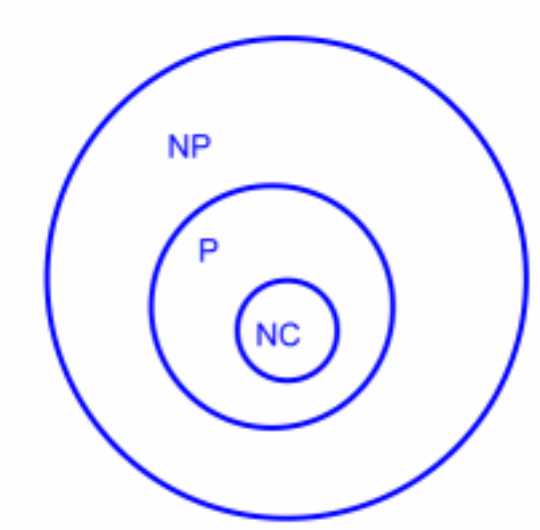


]

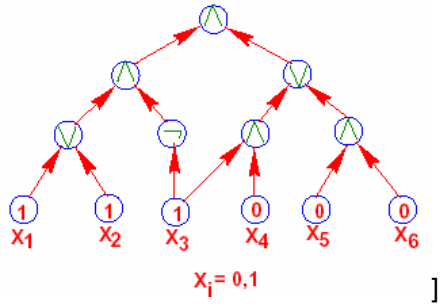
a. **Wie funktioniert der Algorithmus?** [Wir wenden Duality Transformation an, d.h. wir wandeln die Geraden aus dem Primalspace in Punkte im Dualspace um. Aus der geraden y=mx+b wird der Punkt P(x,y). Es gelten die üblichen Regeln der Duality Transformation (Geometrische Algorithmen). Aus den Punkten um Dualspace berechnen wir nun die konvexe Hülle. Die Kanten der Konvexen Hülle im Dualspace transformieren wir nun in Punkte. Diese Punkte sind nun genau die Schnittpunkte unseres Halbebenenschnitts]
b. **Welche Komplexität hat dieser Algorithmus?** [Das Transformieren der Gerade, Kanten bzw. Punkte geht bei n Prozessoren in O(1). Die Konstruktion der konvexen Hülle geht in O(log n) Zeit und O(n log n) Work auf einem CREW PRAM was auch der Gesamtkomplexität des Halbebenenschnittproblems entspricht]

33. **Was ist das „2 Variable Linear Programming Problem“?** [Gegeben ist eine Function (Objective Function) cx+dy, die es zu minimieren gilt. Diese Funktion ist zu minimieren mittels verschiedener Constraints: $a_ix+b_iy+c_i\leq 0$ und $1\leq i\leq n$. Diese Constraints begrenzen Halbebenen. Der gesuchte Punkt unserer zu minimierenden Funktion liegt innerhalb dieses Halbebenenschnitts an irgendeinem Extrempunkt. Die Extrempunkte, d.h. die Schnitte des Halbebenenschnittes zu finden haben wir eben besprochen. Die Komplexität entspricht natürlich der des Halbebenenschnitts!]

34. **Wann ist ein Problem parallelisierbar?** [Ein Problem ist parallelisierbar, wenn es sehr schnell (in Polylogaritmischer Zeit, also $O(\log^k n)$) auf einer akzeptablem Menge von Prozessoren ($O(n^c)$) gelöst werden kann. (c und k sind Konstanten und n die Eingabegröße)]
35. **Was ist die Klasse NC, wie ist sie definiert?** [Die Klasse NC ist also definiert als ein Set aller Sprachen L, die
    - Alle Eingaben von Größe n haben
    - L kann auf einem PRAM in $O(\log^k n)$ Zeit erkannt werden]
36. Parallelen zum P=NP Problem? [Es ist unbekannt ob NC=P gilt➔ Wenn die beiden Klassen nicht gleich sind, dann muss es Probleme aus P geben, die sich zwar auf einer RAM gut lösen lassen, aber nicht auf einem PRAM]



    i. **Was heisst NC Reduzierbarkeit?** [Seien L1 und L2 Sprachen. L1 ist NC reduzierbar auf L2, wenn Es einen NC Algorithmus gibt, der einen beliebigen Input u1 für L1 in eine Eingabe u2 für L2 umwandelt, sodass u1$\in$L1, dann und nur dann wenn u2 $\in$ L2]]
37. **Was ist P-Vollständigkeit?** [Eine Sprache L ist P-Vollständig, wenn L$\in$P und jede Sprache in P NC reduzierbar ist auf L]
38. **Was versteht man unter dem „Circuit Value Problem“ CVP?** [Das Ergebnis eines Booleschen Schaltkreises unter einer bestimmten Eingabe von Werten zu finden



]
    a. **Warum ist dieses P-Vollständig und wie weißt man das nach?** [Es liegt auf jeden Fall in P, denn sequentiell können wir dieses Problem in O(n) lösen. Mit Hilfe der Touring-Maschine können wir herausfinden, dass unser Schaltkreis von einem NC Algorithmus in polynomieller Zeit gelöst werden kann. L ist als NC reduzierbar. Weil L ein beliebige Sprache aus P war, ist das CVP P vollständig]
39. **Welche anderen P-Vollständigen Probleme wurden angesprochen?** [Ordered Depth-First Search|Maximum Network Flow|Linear Inequalities]