# *Ein schneller, paralleler Algorithmus zur Berechnung aller maximaler Cliquen eines Graphen*

*Alexander M. Gross*

Seminar „*Algorithmen auf geordneten Strukturen*“

Priv.-Doz. Dr. Elias Dahlhaus
Institut für Informatik
Abteilung V
Wintersemester 1995/1996

Rheinische Friedrich-Wilhelms-Universität Bonn

# Inhaltsverzeichnis

# Ein schneller, paralleler Algorithmus zur Berechnung aller maximaler Cliquen eines Graphen ([DK 3])

*Dieser schnelle, parallele Algorithmus zur Berechnung aller maximaler Cliquen bzw. maximaler unabhängiger Mengen in beliebigen Graphen benötigt $O(\log^3(nM))$ Parallelzeit und $O(M^6n^2)$ Prozessoren auf einer CREW-PRAM (concurrent read/exclusive write parallel random access machine), wobei n die Anzahl der Knoten und M die Anzahl der maximalen Cliquen sei.*

*Frühere Ansätze zur Berechnung aller maximaler Cliquen bzw. aller maximalen unabhängigen Mengen arbeiten sequentiell oder nur sehr eingeschränkt effizient parallel.*

*Gegeben sei ein beliebiger Graph G, eine natürliche Zahl K, bestimme K Cliquen von G oder bestimme, daß weniger als K Cliquen in G vorkommen.*

*Zur Anwendung gelangt dabei eine neue universelle Methode für Gitterstrukturen von vollständigen bipartiten Graphen und neue Erkenntnisse auf solchen Gittern.*

# 1 Einführung

Einige bedeutende Klassen von Graphen besitzen eine Anzahl von Cliquen, die polynomiell mit der Anzahl der Knoten im Graphen verknüpft ist. Ein Beispiel hierfür sind Kantengraphen. Für diese Klasse existieren Algorithmen, die in polynomieller Zeit die Menge aller Cliquen berechnen.

Ein früher allgemeiner Algorithmus zur Berechnung aller Cliquen eines Graphen, der in polynomieller Zeit bezogen auf die Anzahl der Knoten und die Anzahl der Cliquen durchführbar ist, ist ein Algorithmus von Bierstone [TIAS].
Schnellere Algorithmen zur Berechnung von Cliquen in anderen Graphen sind ebenfalls bekannt [NNS], [DK 1], diese werden wir aber hier nicht betrachten.

Der im folgenden präsentierte Algorithmus stellte eine Verallgemeinerung dieser Algorithmen dar und berechnet Cliquen für einen beliebigen gegebenen Graphen.

Zunächst erfolgen in Abschnitt 2 einige einfache Definitionen. Die Abschnitte 3 und 4 befassen sich mit benötigten Grundlagen. Abschnitt 5 beinhaltet den parallelen Algorithmus und wir beleuchten näher dessen Aufbau.

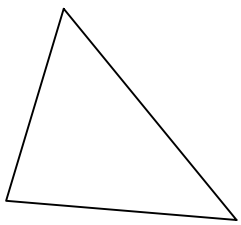
# 2 Grundlegende Definitionen

## *2.1 Definition : Clique*

Ein Graph $G=(V,E)$ besteht aus einer Menge $V$ von *Knoten* und einer Menge $E$ von *Kanten*. Eine (maximale) *Clique* von $G$ ist ein maximaler vollständiger Teilgraph von $G$. Daraus folgt, daß eine Clique durch die Menge ihrer Knoten identifiziert wird.

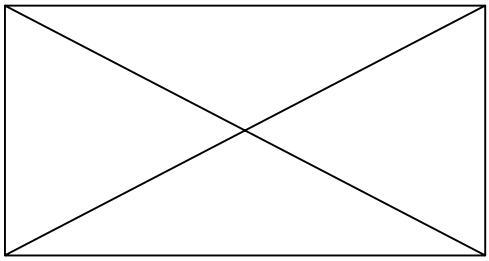
Zur Veranschaulichung des Cliquenbegriffs geben wir folgendes

## *2.2 Beispiel : Verschiedene Typen von Cliquen*

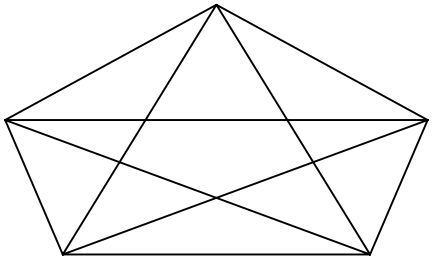


3-Clique

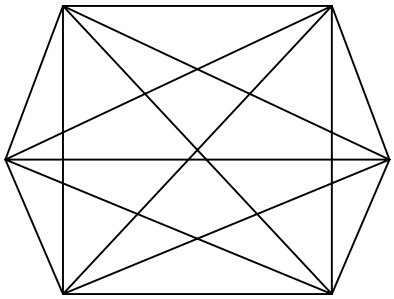
Abb. 1 : Cliquentypen



4-Clique

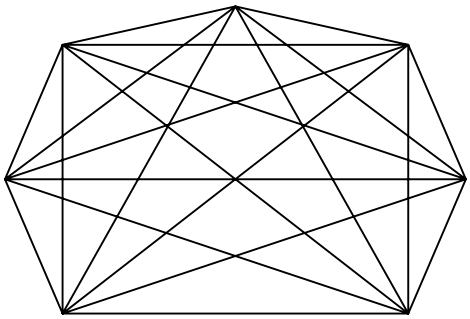


5-Clique

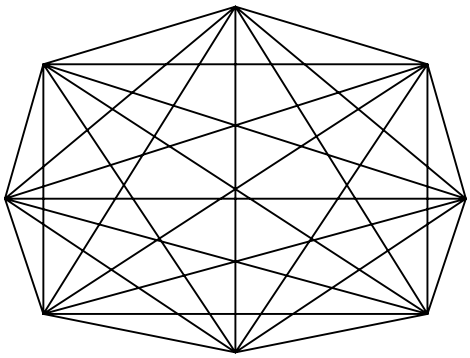


6-Clique

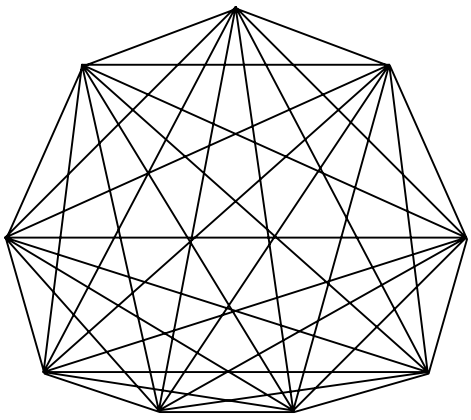
Abb. 2 : Cliquentypen



7-Clique



8-Clique



9-Clique

Abb. 3 : Cliquentypen

## *2.3 Bemerkung zur Problemklasse $NC^k$*

Die Klasse von Berechnungsproblemen, die berechenbar ist von uniformen Folgen boolescher Schaltkreise von $O(log^k\ n)$ Tiefe und polynomieller Größe wird bezeichnet mit $NC^k$.

$NC=\bigcup_k NC$ ist identisch mit der Klasse von Problemen, die in polylog Zeit und polynomiell verknüpfter Anzahl Prozessoren auf einer Parallelen Random Access Maschine (PRAM) lösbar ist.

Zunächst werden wir uns dem Berechnungsmodell der Random Access Maschine zuwenden.

# 3 RAM-Maschinen

Das grundlegende Modell der Random Access Maschine gehört zur Gruppe der moderneren Berechnungsmodelle. Es ist stark an das J.v.Neumann’sche Rechnermodell angelehnt.

Das Modell von v.Neumann besteht aus einer Rechen- und Steuereinheit, einem Hauptspeicher, der sowohl die Instruktionen als auch die Daten beinhaltet, zusätzlich Ein- und Ausgabeeinheiten und einem Hintergrundspeicher.

Die Steuereinheit kann wahlfrei auf die Speicherzellen im Hauptspeicher zugreifen, diese Daten manipulieren und wieder speichern. Dabei wird die Zeit für eine elementare Operation durch die Zeit, die für die Kommunikation zwischen Speicher und Steuereinheit notwendig ist, majorisiert.

Dieses Rechnermodell ist in fast allen gängigen sequentiell arbeitenden Computern realisiert.

## *3.1 Definition : Random Access Maschine*

Eine *Random Access Maschine*, kurz *RAM*, ist ein theoretisches Rechnermodell.

Eine RAM besteht aus einem Akkumulator $\alpha$ und einer beschränkten Anzahl von Indexregistern $\gamma_1, \gamma_2, \ldots, \gamma_g$, einer Menge von Speicherzellen $\pi_i$, $i \in \mathbb{N}$, auf die wahlfrei zugegriffen werden kann, und einem Programmzähler *LC*.
Der Inhalt der Speicherzellen, der Register und des Programmzählers sind beliebig große ganze Zahlen.

Somit kann der Speicher auch als eine Funktion $\pi: \mathbb{N} \rightarrow \mathbb{Z}$ aufgefaßt werden.

Die folgende Abbildung zeigt ein Schaubild der RAM.

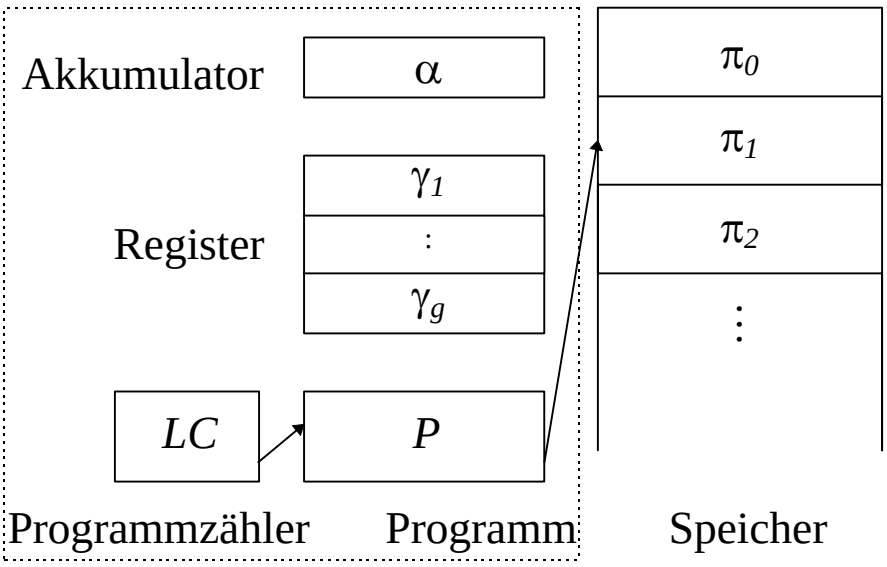



Abb. 4 : Random Access Maschine

Eine Random Access Maschine kann als eine Art „springende“ TM (Turing Maschine) angesehen werden, da der wahlweise Zugriff auf den Speicher einem Sprung des Lese-/Schreibkopfes an die entspechende Bandzelle entspricht (anstatt einer Folge von Kopfbewegungen).

Nun wollen wir eine genauere Beschreibung des Maschinenmodells RAM geben. Um die syntaktische Definition zu geben, benutzen wir die Backus-Naur-Form. Auf eine genaue semantische Definition verzichten wir ([Inf]).

Zunächst definieren wir die syntaktischen Klassen :

## *3.2 Definition : Syntaktische Kategorien der RAM*

- <REG> ::= $\alpha$ | $\gamma_j$ mit $1 \leq j \leq g$
- <LOP> ::= $\pi_i$ | <REG>
- <OP> ::= $i$ | $\pi_i$ | <REG>
- <MOP> ::= $\pi_{i+\gamma j}$ mit $i \in \mathbb{N}$ und $1 \leq j \leq g$
- <G> ::= $\gamma_j$ mit $1 \leq j \leq g$
- <NUM> ::= $\mathbb{N}$

Dabei repräsentiert $\pi_i$ den Inhalt der Speicherzelle mit der Nummer $i$. Wie oben gesagt kann $\pi$ auch als Funktion $\pi$: $\mathbb{N} \rightarrow \mathbb{Z} \supset \mathbb{N}$ interpretiert werden. Eine Adressenrechnung wird durch die Verwendung der indirekten Adressierung möglich. Sie erfolgt über die Benutzung modifizierter Operanden <MOP>.

Wir können die Befehle der RAM in vier verschiedene Kategorien einteilen :

- Transportbefehle,
- Sprungbefehle,
- arithmetische Befehle und
- Indexbefehle.

Die Transportbefehle dienen zum Laden und Speichern des Akkumulatorinhaltes und der Registerinhalte von bzw. in die Speicherzellen.

Die Sprungbefehle ermöglichen die Steuerung des Programmflußes.

Die arithmetischen Befehle ermöglichen der RAM das Ausführen elementarer Rechenoperationen.

Ein RAM-Programm ist nun eine numerierte Folge von RAM-Befehlen.

Als Konvention nehmen wir an, daß die Befehle eines RAM-Programms fortlaufend von 0 an numeriert sind.
Zum Start eines RAM-Programms nehmen wir zusätzlich an, daß alle Speicherzellen mit 0 initialisiert sind und nur die Speicherzelle $\pi_0$ die Eingabe enthält.
Nach Beendigung des RAM-Programms wird die Ausgabe in der Speicherzelle $\pi_1$ zur Verfügung gestellt.

Da eine RAM in jeder Speicherzelle beliebig große Zahlen speichern kann, und damit in einer Operation auch Daten beliebiger Größe

verarbeiten kann, benötigen wir ein geeignetes Komplexitätsmaß, welches die Größe der Operanden berücksichtigt.

Diese Notwendigkeit tritt bei den Turing Maschinen nicht auf, da in einer Zeiteinheit nur eine Bandzelle pro Band bearbeitet werden kann, deren Größe nur von der (festen) Kardinalität des Bandalphabetes abhängt.

Daher definieren wir nun für die RAM zwei Kostenmaße.

## *3.3 Definition : Kostenmaße für die RAM*

Für die RAM können wir in natürlicher Weise zwei Kostenmaße definieren :

- *Einheitskostenmaß*, d.h. jeder Speicherzugriff und jeder Befehl kostet eine Zeiteinheit
- *Logarithmisches Kostenmaß* (auch Bit-Kostenmaß genannt), d.h. die Kosten jedes Speicherzugriffs und jedes Befehls sind proportional zur Länge der Operanden (in Binärdarstellung).

## *3.4 Definition : Kosten für die Bereitstellung von Operanden*

Sei $L(n)$ die Länge der Dualdarstellung von $n$ :

$$L(n) := \begin{cases} 1, n = 0 \\ \lfloor \log n \rfloor + 1, sonst \end{cases}$$

Die folgenden Tabellen zeigen die Kosten für die Bereitstellung von Operanden :

| Operand | Einheitskostenmaß | Log.-Kostenmaß |
|---|---|---|
| $i$ | 0 | 0 |
| REG | 0 | 0 |
| $\pi_i$ | 1 | $L(i)$ |
| $\pi_{i + \gamma j}$ | 1 | $L(i) + L(\gamma_j)$ |

und die Kosten der Befehle :

| Befehlsart | Einheitskostenmaß | Log.-Kostenmaß |
|---|---|---|
| Transportbefehl | 1 | $1 + L(m)$ |
| Sprungbefehl | 1 | $1 + L(k)$ |
| Arithm. Befehl | 1 | $1 + L(m_1) + L(m_2)$ |
| Indexbefehl | 1 | $1 + L(i) + L(\gamma_j)$ |

wobei $m$, $m_1$ und $m_2$ Operanden und $k$ der Sprunglabel sind.

Mit Hilfe dieser verschiedenen Kostenmaße können wir einem RAM-Programm auch verschiedene Komplexitäten zuordnen.

## *3.5 Definition : Einheitskomplexität eines RAM-Programms*

Gegeben sei ein RAM-Programm *P*.

Dann ist die *Einheitszeitkomplexität* von *P*

$T'_P(n) := \max_{|x|=n} \{$Anzahl der ausgeführten Operationen bei Eingabe $x\}$

und die *Einheitsspeicherkomplexität* von *P*

$S'_P(n) := \max_{|x|=n} \{$maximale Anzahl der benutzten Register und Zellen während der Berechnung bei Eingabe $x\}$.

## *3.6 Definition : Log-Komplexität eines RAM-Programms*

Gegeben sei ein RAM-Programm *P*.

Dann ist die *log.-Zeitkomplexität* von *P*

$T_P(n) := \max_{|x|=n} \{$Summe der log.-Kosten der Operationen während der Berechnung bei Eingabe $x\}$

und die *log.-Speicherkomplexität* von *P*

$S_P(n) := \max_{|x|=n} \{$maximale Summe der Längen der Binärdarstellung der Zelleninhalte bei Eingabe $x\}$.

Zur Erläuterung dieser Definition sei ein kleines Beispiel gedacht.

## *3.7 Beispiel : Berechnung von $2^n$.*

Folgendes RAM-Programm *P* dient zur Berechnung von $2^n$ :

```
0   α ← 1
1   γ1 ← π0
2   IF γ1 = 0 THEN GOTO 6
3   α ← α * 2
4   γ1 ← γ1 - 1
5   GOTO 2
6   π1 ← α
```

Eine kleine Analyse zeigt, daß dieses Programm eine Einheitszeitkomplexität von $T'_P(n) = O(n)$ besitzt.

# 4 Parallele RAM-Maschinen

Zunächst werden wir unser Modell der RAM erweitern. Dies führt zum parallelen Berechnungsmodell der *PRAM*.

## *4.1 Definition : Parallele RAM-Maschine*

Eine Menge von Tupeln ($\alpha$, $\gamma_1,\ldots,\gamma_g$, $LC$) und eine Menge von Speicherzellen $\pi_i$, $i \in \mathbb{N}$, heißt *PRAM*, falls

- jedes Tupel ($\alpha$, $\gamma_1,\ldots,\gamma_g$, $LC$) mit der Menge der Speicherzellen eine RAM bildet; (es gelten die im vorigen Kapitel definierten Befehle.)
- die Menge der Speicherzellen für alle RAM's dieselbe ist, d.h. die RAM's keinen lokalen Speicher besitzen;
- jede RAM durch eine für sie spezifische Nummer ausgezeichnet ist;
- alle RAM's dasselbe RAM-Programm bearbeiten;
- die RAM's synchronisiert sind, d.h. alle RAM's beginnen ihre Anweisung zeitgleich. Die nächste Anweisung wird erst gemeinsam von allen RAM's begonnen, nachdem alle aktiven RAM's die vorherige Anweisung ausgeführt haben.

Eine RAM als Teil einer PRAM unterscheidet sich also grundsätzlich von einer normalen RAM dadurch, daß sie keinen unendlich großen privaten Speicher hat.
Daher können wir eine RAM, die Bestandteil einer PRAM ist, auch als *Prozessor*, also als reines Rechen- und Steuerwerk, ansehen.

Es gilt jedoch folgende

## *4.2 Bemerkung zum Speicher*

Es ist keine Einschränkung, daß die RAM's keinen lokalen Speicher haben. Der globale Speicher kann mittels einer geeigneten Adreßrechnung so aufgeteilt werden, daß er in unendlich große lokale Speicher für jede RAM und einen unendlich großen gemeinsamen Speicher zerfällt.
Diese Speicheraufteilung kann z.B. mittels Methoden ähnlich zum Cantor'schen Diagonalverfahren konstruiert werden.

## *4.3 Definition : Master-Prozessor*

Die Berechnung einer PRAM wird dadurch gestartet, daß der Prozessor mit der Prozessornummer 0 die Berechnung beginnt. Die Berechnung einer PRAM ist beendet, wenn der Prozessor mit der Prozessornummer 0 stoppt. Man kann also den Prozessor $P_0$ als *Master-Prozessor* ansehen.

## *4.4 Definition : Laufzeit und Prozessoren*

Die Laufzeit $TP_P$ eines PRAM Programmes $P$ ist die Laufzeit des Prozessors mit der Prozessornummer 0.
$SP_P$ bezeichne die maximale Anzahl von Prozessoren, die während der Berechnung aktiviert werden.

## *4.5 Bemerkung zu Kostenmaßen*

Damit hängt die Laufzeit eines PRAM Programms von den Komplexitätsmaßen ab, die wir auf die aktivierten Prozessoren anwenden. Es gibt hier also auch ein Einheits- und ein logarithmisches Kostenmaß.

Die Tatsache, daß viele Prozessoren gleichzeitig auf einen gemeinsamen Speicher zugreifen, kann zu Konflikten führen :

1. Zwei Prozessoren wollen gleichzeitig aus einer Zelle lesen.
2. Ein Prozessor will aus einer Zelle lesen, in die ein anderer Prozessor schreiben will.
3. Zwei Prozessoren wollen in die gleiche Zelle schreiben.

Der Fall 2 kann leicht gelöst werden mit Hilfe der folgenden

## *4.6 Definition : Berechnungsphasen*

Der Berechnungsvorgang läuft in drei Phasen ab :

- Lesephase : Jeder Prozessor liest die Daten, die für den nächsten elementaren Befehl nötig sind, aus dem gemeinsamen Speicher.
- Berechnungsphase : Jeder Prozessor führt eine Operation aus, die nur von den lokalen und gerade gelesenen Daten abhängig ist.
- Schreibphase : Jeder Prozessor schreibt das Ergebnis seiner Berechnung in den gemeinsamen Speicher.

Somit findet der Lese- und der Schreibvorgang zu verschiedenen Zeiten statt, dadurch sind Lese- und Schreibkonflikte ausgeschlossen. Die anderen beiden Fälle lassen sich nicht allgemein lösen. Daher definieren wir je nach Lösung dieses Problems geeignete Modelle.

## *4.7 Definition : PRAM-Modelle*

Wir unterscheiden die folgenden Modelle von PRAM’s :

- EREW-PRAM (Exclusive Read Exclusive Write) : gemeinsames Lesen und Schreiben sind verboten.

- CREW-PRAM (Concurrent Read Exclusive Write) : nur gemeinsames Lesen ist erlaubt. Schreibzugriffe auf dieselbe Speicherzelle sind verboten.
- CRCW-PRAM (Concurrent Read Concurrent Write) : sowohl gemeinsames Lesen als auch gemeinsames Schreiben ist erlaubt.

Im Falle der CRCW-PRAM müssen wir noch folgende Fälle betrachten:

- Common CRCW-PRAM : gemeinsames Schreiben ist nur erlaubt, falls alle Prozessoren die in eine Speicherzelle schreiben wollen, den gleichen Wert schreiben.
- Arbitrary CRCW-PRAM : beim gemeinsamen Schreiben setzt sich einer der Prozessoren willkürlich durch.
- Priority CRCW-PRAM : beim gemeinsamen Schreiben setzt sich der Prozessor mit der kleinsten Prozessornummer durch.

Das Modell der PRAM erweitert die Menge der berechenbaren Funktionen nicht, da PRAM’s leicht durch RAM’s simuliert werden können. Dabei simuliert die RAM in $SP(n)$ Schritten je einen Schritt der $SP(n)$ aktiven Prozessoren der PRAM. Dadurch erhalten wir ein RAM-Programm mit Laufzeit $SP(n) \bullet TP(n)$.

Ein PRAM-Programm ist somit optimal, wenn das Prozessor-Zeit-Produkt $SP(n) \bullet TP(n)$ in der gleichen Größenordnung wie die Zeit des besten sequentiellen RAM-Programms liegt.

# 5 Ein paralleler Algorithmus

Nun haben wir die Grundlagen für unser eigentliches Thema geschaffen und kommen nun zum Algorithmus zur Berechnung von Cliquen.

Doch zunächst noch kurz eine

## *5.1 Anmerkung zum verwendeten Modell*

Im weiteren Verlauf werden wir das Modell CREW-PRAM verwenden.

Wir bezeichnen die *Anzahl der Knoten* durch *n*, die *Anzahl der Kanten* durch *m* und die *Anzahl der Cliquen* durch *M*.

Das grundlegende Ergebnis auf der sequentiellen Komplexität der Berechnung aller Cliquen beschreibt folgendes

## *5.2 Theorem 1*

Es existiert ein Algorithmus, welcher die Menge aller Cliquen in einem beliebigen Graphen berechnet und der $O(n+m)$ Platz und $O((n \bullet m)M)$ Zeit benötigt.

Es folgt eine

## *5.3 Allgemeine Beschreibung des parallelen Algorithmus*

Wir nehmen im folgenden an, daß $G=(V,E)$ und $V=\{v_1,\ldots,v_n\}$. Wir beginnen mit der Beschreibung des Algorithmus.

```
ALGORITHM :

Input : G=(V,E), V={v1,...,vn}


// Menge von Cliquen von G=(V,E)
PROCEDURE CLIQUE(V,E)

IF |V|=1 THEN CLIQUE(V,E):={V}

ELSE

BEGIN

Sei G1 der Teilgraph von G induziert durch
{v1,...,v⌐n/2¬}
Sei G2 der Teilgraph von G induziert durch
{v⌐n/2¬+1,...,vn}


DO IN PARALLEL
U:=CLIQUE(G1)          // Menge von Cliquen von G1
W:=CLIQUE(G2)          // Menge von Cliquen von G2
```

```
FOR EACH u∈U, v∈W DO

BEGIN

PROCEDURE COMP_MAX(D_{u,v})
D_{u,v} := {c⊆u∪v : c ist vollständig und
                    maximal in G beschränkt auf u∪v}
E_{u,v} := {c∈D_{u,v} : c ist eine Clique in G}

END
```

$CLIQUE(C) := \bigcup_{\substack{u\in U \\ v\in V}} E_{u,v}$

```
END

END PROCEDURE CLIQUE

Output : CLIQUE(V,E)
```

## *5.4 Bemerkung zur Korrektheit des Algorithmus*

Seien $G_1$ und $G_2$ wie im Algorithmus definiert und $V_1$ und $V_2$ seien deren zugehörige Knotenmengen. Sei $c$ eine Clique von $G=(V,E)$. Dann sind $c \cap V_1$ und $c \cap V_2$ Untermengen von Cliquen von $V_1$ und $V_2$. Diese bezeichnen wir mit $u$ und $v$. Dann gilt $c \in D_{u,v}$ und deshalb auch $c \in E_{u,v}$.

## *5.5 Anmerkung zur Komplexität*

Cliquen können von einer CREW-PRAM in $O(\log n)$ Zeit und bei Verwendung von $O(n^2)$ Prozessoren überprüft werden. Die Rekursionstiefe der Prozedur beträgt $\lceil \log n \rceil$.

Es gilt nun die parallele Komplexität der Berechnung von $D_{u,v}$ zu betrachten, `PROCEDURE` ***`COMP_MAX`*** $(D_{u,v})$.

## *5.6 Bemerkung zur Berechnung von Teilgraphen von G*

Die vollständigen Mengen $u$ und $v$ sind disjunkt. Jeder maximale, vollständige Teilgraph von $G$ beschränkt auf $u \cup v$ korrespondiert zu einem maximalen, vollständigen, bipartiten Teilgraphen des bipartiten Graphen $(u \cup v, E')$, wobei $E'$ sei die Menge von Kanten von $E$, die beliebige Knoten aus $u$ auf dem Weg zu einem Knoten aus $v$ besuchen.

Wir erhalten folgendes Ergebnis ([Wi]) :

## *5.7 Theorem 2*

Der maximale, vollständige, bipartite Teilgraph eines bipartiten Graphen bildet eine Gitterstruktur im Sinne einer allgemeinen Algebra.

Die Gitterstruktur ist wie folgt definiert ([Bi 1], [Bi 2]) :

Zunächst definieren wir einen Hilfsschließungsoperator :

Sei $A$ eine beliebige Teilmenge von $U$. Dann

$A_2 := P_2(A) := \{x \in v : \forall y \in A\{y,x\} \in E'\}$

und

$A_1 := P_1(A_2) := A'_2 := \{y \in u : \forall x \in A_2\{y,x\} \in E'\}$

## *5.8 Beobachtung*

$A_1 \cup A_2$ bilden einen maximalen, vollständigen, bipartiten Teilgraphen und alle maximalen, vollständigen, bipartiten Teilgraphen sind von dieser Form.

Nun definieren wir die Gitteroperationen $\vee$, $\wedge$ :

$A_1 \cup A_2 \vee B_1 \cup B_2 := (A_1 \cup B_1)_1 \cup (A_2 \cap B_2) = P_1(A_2 \cap B_2) \cup (A_2 \cap B_2)$

$A_1 \cup A_2 \wedge B_1 \cup B_2 := (A_1 \cap B_1)_1 \cup (A_1 \cap B_1)_2 = (A_2 \cap B_2) \cup P_2(A_2 \cap B_2)$

Es ist folgendes zu beobachten :

## *5.9 Lemma 1*

Für jeden maximalen, vollständigen, bipartiten Teilgraphen $A_1$, $A_2$ von $(u \cup v, E')$ haben wir

$$A_1 \cup A_2 = \bigvee_{a \in A1} \left( \{a\}_1 \cup \{a\}_2 \right)$$

Wir können nun den folgenden Algorithmus angeben zur Berechnung aller maximaler, vollständiger, bipartiter Teilgraphen von $(u \cup v, E')$.

```
Procedure COMP_MAX(D_{u,v})
1)  i:=0
    U_0 := {∅_1∪∅_2} ∪ {{a}_1∪{a}_2 : a ∈ u}
2)  Repeat
        i:=i+1
        U_i := UNION(U_{i-1}) := {r ∨ s : r, s ∈ U_{i-1}}
    Until U_i = U_{i-1}
3)  Ouput D_{u,v} := U_i
```

## *5.10 Analyse des Algorithmus*

Es läßt sich zeigen, daß $U_i$ mindestens alle $A_1$, $A_2$ enthält und die Größe von $A_1$ höchstens $2^i$ ist. Daraus folgt, daß die Repeat-Schleife höchstens $O(\log n)$ mal durchlaufen wird.

Die Berechnung von $A_1$ und $A_2$ aus $A$ benötigt $O(log\ n)$ Zeit und $O(n^2)$ Prozessoren. Dies gilt auch für die Berechnung von $\vee$. Offensichtlich ist die Größe von $U_i$ verknüpft mit $M$.

Die Berechnung von $U_i$ aus $U_{i-1}$ wird aufgeteilt in die folgenden Teilprozeduren :

1) Für alle $s, t \in U_{i-1}$ berechne $s \vee t$
2) Lösche Duplikate in $U_i$
3) Füge $U_i$ durch sortieren in ein Feld der Länge von höchstens $M$ ein

wobei

1) kann durchgeführt werden in $O(log\ n)$ Zeit von $O(M^2n^2)$ Prozessoren
2) kann durchgeführt werden in $O(log\ n)$ Zeit von $O(M^4n)$ Prozessoren
3) kann durchgeführt werden in $O(log\ M)$ Zeit von $O(M^2)$ Prozessoren

## *5.11 Folgerung*

$U_i$ kann berechnet werden aus $U_{i-1}$ in $\max(O(log\ n), O(log\ M))$ Zeit von $\max(O(M^4n), O(M^2n^2))$ Prozessoren.

Wir schließen daraus folgendes

## *5.12 Theorem 3*

Die Menge aller Cliquen eines beliebigen Graphen kann von einer CREW-PRAM in $\max(O(log^3\ n), O(log^3\ M))$ Zeit von $\max(O(M^6 n), O(M^4 n^2))$ Prozessoren berechnet werden.

Eine erweiterte Analyse des Algorithmus ermöglicht es folgendes Problem in *NC* zu lösen :

Input : Ein Graph *G* und eine natürliche Zahl *K*
Output : *K* Cliquen von *G* , falls diese existieren
Aussage „es existieren weniger als *K* Cliquen in *G*“, sonst

## *5.13 Skizzierung eines Algorithmus*

Wir betrachten einen Algorithmus, der die Menge aller Cliquen eines Graphen berechnet. Wir starten die Berechnung und stoppen sobald wir einen Bereich mittels Divide-And-Conquer erhalten, der *K* oder mehr Cliquen besitzt. Dann dehnen wir die Cliquen dieses Bereichs auf den gesamten Graphen aus mit Hilfe eines der verfügbaren MIS-Algorithmen (siehe [Lu], [GS]).

## *5.14 Bemerkung zum MIS-Problem*

Rein sequentiell betrachtet berechnet der Algorithmus folgendes :

- Initialisiere *I* mit der leeren Menge
- Für *v*=1,…,*n* : falls Knoten v nicht adjazent ist zu einem beliebigen Knoten in *I*, dann füge Knoten *v* zur Menge *I* hinzu

Der Output des MIS-Algorithmus ist die sog. lexikographische erste maximale unabhängige Menge.

# 6 Denkbare Anwendungen und verwandte Themen

Die zentrale Aussage dieser Ergebnisse ist, daß die Berechnunsgprobleme von allen Cliquen und maximalen unabhängigen Mengen effizient parallelisierbar sind für mehrere bedeutende Klassen von Graphen.

Die Ergebnisse fordern außerdem die Existenz uniformer boolescher Schaltkreise von $O(log^3\ n)$ Tiefe und poly-Größe für die Berechnung aller Cliquen für beliebige Klassen von Graphen und daß die Anzahl der Cliquen polynomiell verknüpft ist.
Dies steht im Zusammenhang mit den letzten Ergebnissen von Grigoriev ([GK]) bzgl. paralleler Aufzählung aller perfect matchings in bipartiten Graphen mit polynomiell verknüpften Permanenten.

Ein verwandtes Problem ist die schnelle, parallele Dekomposition von Cliquen eines Graphen. Tarjan ([Ta]) bietet hierfür einen Algorithmus, welcher auf einer streng, sequentiellen Unterroutine zur Berechnung minimaler Ordnungen basiert.
Solange die Anzahl der Cliquenseparatoren eines beliebigen Graphen polynomiell zur Anzahl der Knoten verknüpft ist, existiert ein schneller, paralleler Aufzähler für alle Cliquenseparatoren.

Mit dem zuvor beschriebenen Ansatz ist es möglich das Problem der Cliquenseparatoren in *NC* zu berechnen und darüber hinaus auch das Problem der Dekomposition von Cliquen eines beliebigen Graphen.

Im Zusammenhang mit diesem Ergebnis für verschiedene Unterklassen von perfekten Graphen gewinnt die Frage von parallelen Berechnungen von maximalen Cliquen oder maximal, unabhängigen Mengen für perfekte Graphen ([GLS]) an Bedeutung.

# Literaturverzeichnis


[AB] Alon, N., and Boppana, R.B., *The Monotone Circuit Complexity of Boolean Functions*, Manuscript, MIT 1986

[AM] Auguston, J.M. and Minker, J., *An Analysis of Some Graph Theoretical Cluster Techniques*, J. ACM 17(1970), pp. 571-588

[Bi 1] Birkhoff, G., *Subdirect Unions in Universal Algebra*, Bull. Amer. Soc. 50(1944), pp. 764-768

[Bi 2] Birkhoff, G., *Lattice Theory*, 3rd ed. Amer. Soc., Providence 1967

[CN] Chiba, N., and Nishivuki, T., *Arboricity and Subgraph Listing Algorithms*, SIAM J. of Comput. 14(1985), pp. 210-223

[Cl] Cole, R., *Parallel Merge Sorting*, Proc. 27th IEEE FOCS (1986), pp. 511-516

[CV] Cole, R., and Vishkin, U., *Approximate and Exact Scheduling with Applications to List, Tree and Graph Problems*, Proc. 27th IEEE FOCS (1986), pp. 478-491

[Co] Cook, S.A., *A Taxonomy of Problems with Fast Parallel Algorithms*, Information and Control 64 (1986), pp. 2-22

[DK 1] Dahlhaus, E., and Karpinski, M., *The Matching Problem for Strongly Chordal Graphs is in NC*, Research Report No. 855-CS, Department of Computer Science, University of Bonn 1986

[DK 2] Dahlhaus, E., and Karpinski, M., *Fast Parallel Computation of Perfect and Strongly Perfect Elimination Schemes*, IBM Research Report # RJ 5901 (59206), IBM Almaden Research Center, San Jose 1987; submitted for publication

[DK 3] Dahlhaus, E., and Karpinski, M., *A Fast Parallel Algorithm for Computing all Maximal Cliques in a Graph and the Related Problems*, SWAT 88, LNCS 318, pp. 139-144.

[GHS] Gabor, C.P., Hsu, W.L., and Supowit, K,J., *Recognizing Circle Graphs in Polynomial Time*, Proc. 26th IEEE FOCS (1985), pp. 106-116

[GJ] Garey, M.R., and Johnson, D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman:San Francisco 1979

[Ga] Gavril, F., *Algorithms for Minimum Coloring, Maximum Clique, Minimum Coloring by Cliques, and Maximum Independent Sets of a Chordal Graph*, SIAM J. Comput. (1972), pp. 180-187

[GS] Goldberg, M., and Spencer, T., *A New Parallel Algorithm for the Maximal Independent Set Problem*, Proc. 28th IEEE FOCS (1987), pp. 161-165

[Go] Golumbic, M.C., *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York 1980

[GK] Grigoriev, D.Yu., and Karpinski, M., *The Matching Problem for Bipartite Graphs with Polynomially Bounded Permanents is in NC*, Proc. 28th IEEE FOCS (1987), pp. 166-172

[GLS] Grötschel, M., Lovász, L., and Schrijver, A., *The Ellipsoid Method and its Consequences in Combinational Optimization*, Combinatorica 1(1987), pp. 169-197

[Hi] Hirschberg, D., *Fast Parallel Sorting Algorithms*, Communications of the ACM 21(1978), No. 8, pp. 657-66l

[Inf] Skript *Informatik I,II*, Department of Computer Science, University of Bonn (WS 86/87, SS 87)

[Lu] Luby, M., *A Simple Parallel Algorithm for the Maximal Inpedendent Set Problem*, PROC. 17th ACM STOC (1985), pp. 1-9

[MC] Mulligan, G.D., and Corneil, D.G., *Corrections to Bierstone's Algorithm for Generating Cliques*, JACM 19(1972), pp. 244-247

[NNS] Naor, J., Naor, M., and Schäffer, A., *Fast Parallel Algorithms for Chordal Graphs*, Proc. 19thACM STOC (1987), pp. 355-364

[Ra] Razborov, A.A., *Bound on the Monotone Network Complexity of the Logical Permanent*, Matem. Zametk 37 (1985); in Russian

[Ta] Tarjan, R., *Decomposition by Clique Separations*, Discrete Mathematics 55(1985), pp. 221-232

[TIAS] Tsukiyama, S., Ide, M., Ariyoshi, H. and Shirakawa, I., *A New Algorithm for Generating All the Maximal Independent Sets*, SIAM J. Comput 6(1977), pp. 505-517

[Wi] Wille, R., *Subdirect Decomposition Of Concept Lattices*, Algebra Universalis 17(1983), pp. 275-287