Multilevel Repartinioning Algorithms

Ausarbeitung zur Veranstaltung „Parallel Algorithms“

Igor Jankovic, André Greb
FH Bonn-Rhein-Sieg
Sommersemester 2002

10. Juni 2002

Inhaltsverzeichnis

1 Zur generellen Problematik

Partitionierung von Graphen ist ein effizienter Weg um große Rechenaufgaben auf eine bestimmte Anzahl von Prozessoren zu verteilen.

Sie wird zum größten Teil in Wissenschaftlichen Berechnungen eingesetzt wie zum Beispiel im Bezug auf grosse Datenbanken oder im Bereich wie Data Mining, VLSI Schaltungs-Layouts Simulationen usw. Weitere praxisbezogene Beispiele gehen von Luftwiderstandsberechnungen, Auto-Crashberechnungen bis hin zu Wetterberichten. Man getrost sagen — ohne obige Liste ins Unendliche fortsetzen zu wollen — daß Graph-Partitionierung viele Anwendungen in der heutigen Zeit hat und dadurch ein durchaus wichtiges Thema in der heutigen Wissenschaft darstellt.

Im allgemeinen geht es bei der Graph-Partitionierung um die Aufteilung eines Graphen in disjunkte Untermengen. Um aber die Graph-Partitionierung im allgemeinen besser verstehen zu wollen, muß man sich vorher über die Weise im klaren sein, wie ein Graph verteilt ist. Ebenso ist es wichtig, die Bedeutung einiger Begriffe zu kennen.

Hinsichtlich der Knoten eines Graphen sind drei Begriffe wichtig:

1. Das *Gewicht* eines Knotens sagt etwas über den Berechnungsaufwand aus, der mit diesem knoten in Zusammenhang steht.

2. Neben dem Gewicht taucht ein weiterer wichtiger Begriff auf, die *Größe* eines Knotens. Sie ist eine Referenzwert für den Kommunikationsaufwand, der im Falle des Transfers eines Knotens (mehr dazu später) betrieben werden muß.

3. Die *Dichte* eines Knotens stellt das Verhältnis von Gewicht zu Größe dar.

Entsprechend ist das Gewicht einer Graphpartition definiert: Die Gewichte aller Knoten, welche in der Partition enthalten sind, werden aufsummiert und ergeben ihr Gewicht.

Desweiteren, um den folgenden Beitrag von meinen Kollegen besser verstehen zu können, müssen wir uns mit den folgenden Begriffen auseinandersetzen:

Load-Balance: Sind die Gewichte aller Partitionen eines Graphen gleich groß, dann bedeutet das implizit, daß die Berechnungsarbeit auf alle Prozessoren gleich gut verteilt ist. Man sagt auch, der Graph ist gut ausbalanciert oder: der Graph besitzt eine gute Load-Balance.

Edge-Cut: Anzahl aller Kanten, deren inzidente Knoten in unterschiedlichen Untermengen eines partitionierten Graphen liegen. Sind die Kanten gewichtet/footnote (Das Kanten-Gewicht sagt dann etwas über die Höhe des Kommunikationsaufwands zwischen den inzidierenden Knoten der Kante aus.), dann entspricht der Edge-Cut der Summe der betreffenden Kantengewichte.

Da verschiedene Partitionen von verschiedenen Prozessoren bearbeitet werden, bedeutet ein grosser Edge-Cut gleichzeitig einen hohen Kommunikationsaufwand zwischen den beteiligten Prozessoren. Zwei wesentliche Ziele der in diesem Kontext vorgestellten Partitionierungsverfahren sind die Minimierung des Edge-Cuts und die Erreichung einer guten Load-Balance.

Wie man an Abbildung 1 sehen kann, erzeugt die dortige Unterteilung einen Edge-Cut von 3. Die beiden Partitionen P_1 und P_2 haben das gleiche Gewicht, der Graph ist also gut ausbalanciert.

Es gibt Fälle, in denen sich die Struktur des Graphen und/oder seine Gewichtungen permanent ändern, beispielsweise während einer komplexen Simulation. Die Load-Balance gerät dann gewissermaßen aus dem Gleichgewicht. Dann müssen einige Knoten auf andere Prozessoren umverteilt werden, man spricht in diesem Zusammenhang auch von *Knoten-Migration*. Eine Umverteilung bedeutet einen zusätzlichen Kommunikationsaufwand, der noch durch die Tatsache verstärkt wird, daß Knoten i.a. mit einer Menge von Berechnungsdaten assoziiert werden.

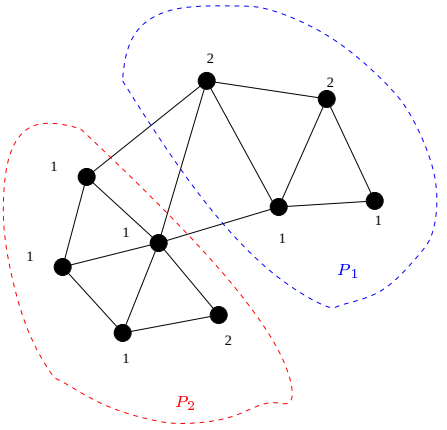


Abbildung 1: Ein einfacher Graph mit zwei Partitionen mit den Gewichten $w(P_1) = 6 = w(P_2)$ und einem Edge-Cut von 3.

Die Neustrukturierung eines solchen aus dem Gleichgewicht geratenen Graphen bezeichnet man als *Repartitionierung*. Ein Verfahren zur Repartitionierung muß also neben einer guten Load-Balance und einem kleinen Edge-Cut auch auf eine geringe Knoten-Migration achten.

Der Berechnungsaufwand zur Graph-Partitionierung ist sehr hoch, wenn man herkömmliche Verfahren anwendet. Daher haben sich heuristische Verfahren etabliert, die zwar nicht immer optimale Lösungen liefern, aber letzteres mit weniger Rechenaufwand. Zu den typischen Klassen der heuristischen Verfahren zählen kombinatorische Verfahren, geometrische Verfahren und Multilevel-Verfahren.

Geometrische Verfahren verwenden zur Partitionierung Koordinaten-Informationen der Graph-Elemente (sofern vorhanden, ggfs. werden Koordinaten trickreich künstlich generiert). Der Edge-Cut wird in diesen Verfahren nur eingeschränkt berücksichtigt, die Qualität der Aufteilungen ist meistens nicht besonders gut, wenngleich sie als besonders schnell gelten (vgl. **??**)Als Beispiel sei das CND-Verfahren genannt, dessen Beschreibung sich ebenfalls in **??** befindet.

Die Berücksichtigung von Adjazenzinformationen ist in den *kombinatorischen Verfahren* besser berücksichtigt, daher liefern diese Verfahren einen besseren Edge-Cut. Jedoch gelten letztere als langsam und schlecht skalierbar auf parallele Architekturen, ebenso werden nicht immer zusammenhängende Partitionen erzeugt, was sich wiederum negativ auf den Kommunikationsaufwand auswirkt . Als Beispiel sei hier das LND-Verfahren aus genannt (vgl. letzten beiden Aussagen mit **??**).

Die letzten beiden Klassen von Partitionierungsalgorithmen werden hier nicht weiter vorgestellt. Wir wollen uns im folgenden nur mit den sog. *Multilevel-Verfahren* weiter auseinandersetzen, insbesondere mit Multilevel-basierten Repartitionierungsverfahren. Multilevel-Verfahren stellen ein Paradigma dar, das Partitionierungsproblem zu lösen: Das Ausgangsproblem wird durch ein ähnliches, aber einfacher strukturiertes Problem ersetzt. Das einfacher strukturierte Problem wird gelöst, und die erhaltene Lösung wird wieder auf das Ausgangsproblem übertragen. Dabei werden weitere Verfeinerungsschritte durchgeführt, um das erhaltene Ergebnis auf grober Stufe weiter an eine Lösung für das Ausgangsproblem anzunähern.

Multilevel-Partitionierungsverfahren zeichnen sich durch drei Bearbeitungsphasen aus:

1. Coarsening Phase
2. Initial Partitioning Phase

3. Refinement Phase

Coarsening: Aus dem Graphen, den es zu bearbeiten gilt, wird in mehreren aufeinanderfolgenden Schritten eine Serie von verwandten, aber gröberen Graphen erzeugt. Am Ende dieser Phase erhält man einen Graphen mit nur wenigen Knoten und Kanten.

Partitioning: Beginnt, wenn der gröbste gewünschte Level im vorhergehenden Coarsening erreicht wurde. Es wird eine Zerlegung nach einem best. Schema vorgenommen. Da sie auf dem gröbsten Level stattfindet, werden nur wenige Knoten und Kanten in der Berechnung berücksichtigt. Daher läuft diese Partitionierung schneller ab, als auf dem voll detaillierten Graphen.

Refinement: Die im Partitioning erreichte Unterteilung wird nun weiter verfeinert, ebenfalls in mehreren (dieses mal jedoch aufsteigenden) Detailstufen, bis der Ausgangsgraph wieder erreicht ist. Die erreichten Zwischenergebnisse in den einzelnen Detailstufen werden jeweils auf die folgende Detailstufe übertragen.

Wir beschäftigen uns in unserer Ausarbeitung nur mit Multilevel-Repartitionierung, die sehr ähnlich abläuft. Der Unterschied zur Multilevel-Partitionierung liegt darin, daß bei einer Repartitionierung bereits vor Beginn ein vorpartitionierter Graph zur Verfügung steht. Die Phase der initialen Partitionierung wird durch einen Repartitionierungsschritt ersetzt, der in unserem Fall aus einem Diffusioning besteht (mehr zum Diffusioning später). Alle drei Phasen können als Multilevel-Verfahrensweisen umgesetzt werden, also auch die Phase der Diffusion.

Obige Schemata zur Multilevel-Diffusion existieren in seriellen und parallelen Varianten. Beide möchten wir in den folgenden Abschnitten geordnet nach ihren einzelnen Phasen vorstellen. Es geht uns dabei primär um die Vermittlung der zentralen Ideen, die wir umgangssprachlich aber auch in Form von Pseudocode vermitteln wollen. Für die Exaktheit und Übereinstimmung der gezeigten Pseudocode-Abschnitte mit den tatsächlichen Algorithmen geben wir keine Gewähr, sie dienen nur zur einigermaßen unmissverständlichen Erklärung der Sachverhalte.

Zum Abschluss unserer Ausarbeitung betrachten wir die Performance der vorgestellten Verfahren anhand der in **??** enthaltenen Benchmarks.

2 Repartitionierungsverfahren

Die Berechnungsstruktur, und mit ihr der Graph, ändern sich oft dynamisch. So werden beispielsweise Teile des Graphen vergröbert oder verfeinert. Es kann auch der Fall sein, da sich der Graph signifikant verändert. In jedem der genannten Fälle ist es notwendig, die Graph-Unterteilung zu ändern, um eine gute Load-Balance und einen möglichst geringen Edge-Cut beibehalten zu können.

Mit der Umverteilung der Knoten tritt ein weiteres Problem auf: Mit einem Knoten werden eine Menge von Daten assoziiert, die zur jeweiligen Berechnung benötigt werden (z.B: Koordinaten, mechanische Druckverhältnisse, etc.). Eine Verschiebung eines Knotens bedeutet eine Verschiebung der assoziierten Daten und somit einen erhöhten Kommunikationsaufwand, insbesondere bei der Verschiebung einer großen Anzahl von Knoten. Diese Migrationskosten stellen also eine dritte Randbedingung dar, die ein Partitionierungs-Verfahren in diesem Fall berücksichtigen sollte. Verfahren, welche zur Veränderung einer bestehenden Graph-Unterteilung eingesetzt werden, nennt man Repartitionierungsverfahren.

Da sich in solchen Fällen ein Graph in seiner Struktur und Gewichte-Verteilung oft ändert, ist es zusätzlich wünschenswert, wenn die Kosten eines derartigen Repartitionierungs-Algorithmus möglichst gering sind.

In diesem Zusammenhang ist es wichtig, zwei weitere Größen zu erwähnen, die in Zusammenhang mit dem oben genannten stehen:

TotalV: Die Summe der Knotengewichte all derer Knoten, welche nach einer kompletten Repartitionierung ihre ursprüngliche Partition verlassen haben. TotalV repräsentiert das gesamte Kommunikationsvolumen, um einen Graphen neu auszubalancieren.

MaxV: Das Kommunikationsvolumen der einzelnen Partitionen (sozusagen ein TotalV pro Partition) wird verglichen und das Maximum dieser Kommunikationsvolumina ausgewählt. Dieses Maximum bezeichnet man als MaxV. Dieser Wert liefert eine obere Schranke für die maximale Kommunikationsdauer pro Prozessor (der ja für eine Partition zuständig ist) in einer Repartitionierung.

Es gibt verschiedene Ansätze, einen Graphen zu repartitionieren: From-Scratch-Ansätze, Cut-And-Paste-Verfahren und Diffusion-basierte Vorgehensweisen.

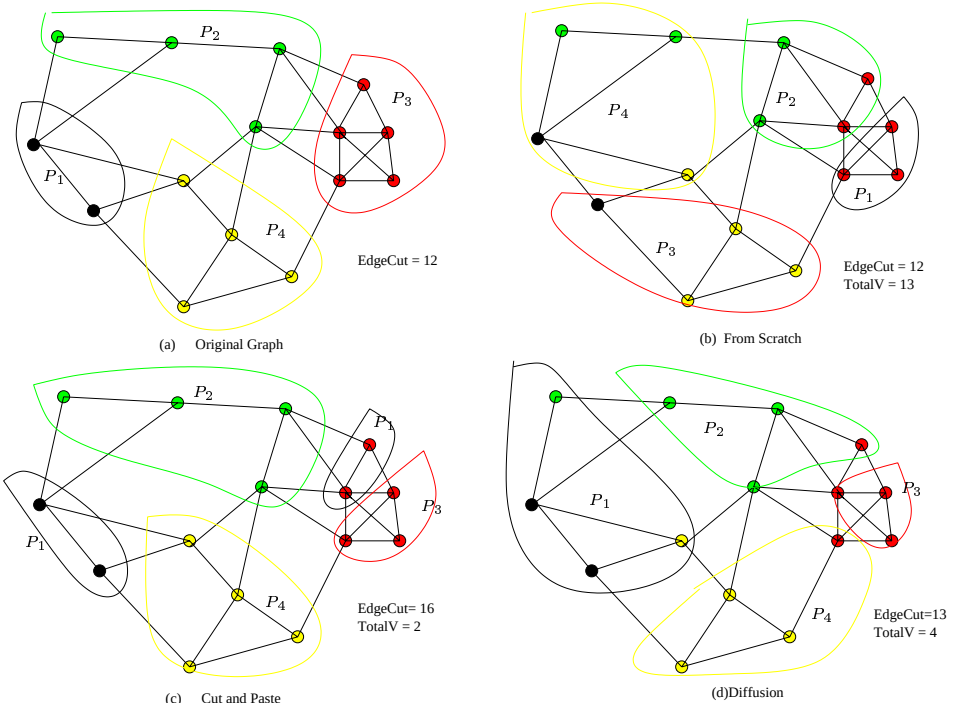


Abbildung 2: Vergleich eines originalen, vorpartitionierten Graphen (a) mit dem Repartitionierungs-Ergebnis nach From-Scratch-Verfahren (b), Cut-And-Paste-Verfahren (c) und Diffusion-basierten Verfahren (d).

2.1 Repartitioning from Scratch

Im einfachsten Fall kann man zur Repartitionierung einen bestehenden Partitionierungs-Algorithmus verwenden, und den veränderten Graphen von Grund auf neu partitionieren. Die verwendeten heuristischen Verfahren erzeugen i.a. aber nicht die gleiche Partitionierungsstruktur, wie sie einmal im Original existiert hatte (s. Abbildung 2). Das wirkt sich entsprechend schlecht auf eine darauffolgende Umverteilung der Knoten aus: Die Migrationskosten sind entsprechend hoch, es werden mehr Knoten als notwendig verschoben.

2.2 Cut-and-Paste-Verfahren

Die wesentliche Idee der Cut-and-Paste Verfahrensweise besteht im Herausnehmen von einem oder mehreren Knoten aus einer übergewichtigen[1] Partition und der anschließend neuen Zuordnung dieser Knoten zu einer untergewichtigen Partition, unabhängig davon, ob diese neue Partition benachbart ist oder nicht.

Das erzeugt eine gute Load-Balance und sorgt implizit dafür, daß nur die notwendigsten Knoten ausgetauscht werden. Sie hat aber Nachteile, die anhand Abbildung 2 ersichtlich werden: Da die Neu-Zuteilung der Knoten auf andere Partitionen ohne Berücksichtigung der Nachbarschaften passiert, kann man unzusammenhängende Partitionen erhalten, was dann für einen hohen Edge-Cut sorgt.

2.3 Multilevel Diffusion Verfahren

Diffusion-Verfahren sorgen unter Berücksichtigung der vorhergehenden Struktur eines Graphen für eine Verbesserung der Load-Balance im Hinblick auf die Erhaltung zusammenhängender Partitionen (vgl. wieder Abbildung 2).

Sie versuchen, die Unterschiede zwischen der Original-Partitionierung und der Neupartitionierung klein zu halten, indem sie während der Neupartitionierung die einzelnen Partitionen inkrementell ändern. Übergewichtige Partitionen geben Knoten an ihre adjazenten Partitionen ab. Diese wiederum können ebenfalls Knoten an benachbarte Partitionen weitergeben, usw.

Es werden bevorzugt schwere Knoten (= Berechnungsaufwand hoch) und große Knoten (= Kommunikationsaufwand hoch) bewegt, um pro Schritt nicht nur ein besseres Balancing sondern auch um eine bessere Kommunikationsstruktur zu erzeugen.

Diffusion-Verfahren kann man nach der Art und Weise kategorisieren, welche Sichtweise auf den Gesamt-Graphen verwendet wird, um Entscheidungen hinsichtlich der Knoten-Bewegungen zu treffen: Wenn ein Diffusion-Schema nur lokale Informationen verwendet, sprich: es berücksichtigt nur die Gewichtsverhältnisse unmittelbar benachbarter Partitionen, dann nennt man dies undirected Diffusion. Nutzt das Schema aber eine globale Sicht auf alle Partitionen, um Entscheidungen hinsichtlich der Knoten-Bewegungen zu treffen, dann spricht man von directed Diffusion.

Es gibt verschiedene Arten, eine globale Sicht auf die Partitionierungsdaten zu erhalten. Im einzelnen auf diese einzugehen sprengt den Rahmen dieser Ausarbeitung. Einführenden Informationen finden sich in (Ref auf das Simulation-Paper).

Diffusion kann auch in Form einer Multilevel-Verfahrensweise angewandt werden. Oft reicht ein initialer Diffusion-Schritt auf dem gröbsten Detaillierungsgrad (= gröbstem Level) nicht aus, um eine zufriedenstellende Load-Balance zu erreichen. Dann wird, wie bei allen Multilevel-Vorgehensweisen, das Ergebnis der Diffusion auf den Graphen mit der nächsthöheren Detaillierungsstufe projiziert und erneut eine Diffusion durchgeführt.

Eine Sicht auf die Repartitionierung unter Zuhilfenahme von Diffusion-Schemata ergibt sich im Gesamt-Kontext wie in Abbildung 3. Man sieht eine vorherige Erzeugung mehrerer Detailstufen des zu repartitionierenden Graphen mittels Coarsening, gefolgt von einem Partitionierungs- oder aber Repartitionierungsschritts wie oben beschrieben und ein abschließendes Refinement, um die erhaltene Neupartitionierung hinsichtlich weiterer Randbedingungen wie den Edge-Cut zu verbessern (verfeinern), bis der ursprüngliche Graph in seiner vollen Detailstufe wieder erreicht ist. Im gezeigten Fall würde die Phase der Repartitionierung nur einmal auf dem gröbsten Level durchlaufen werden.

[1] übergewichtig: das Gesamtgewicht der Partition (das entspricht der Summe der Knotengewichte) liegt über dem Durchschnitt.

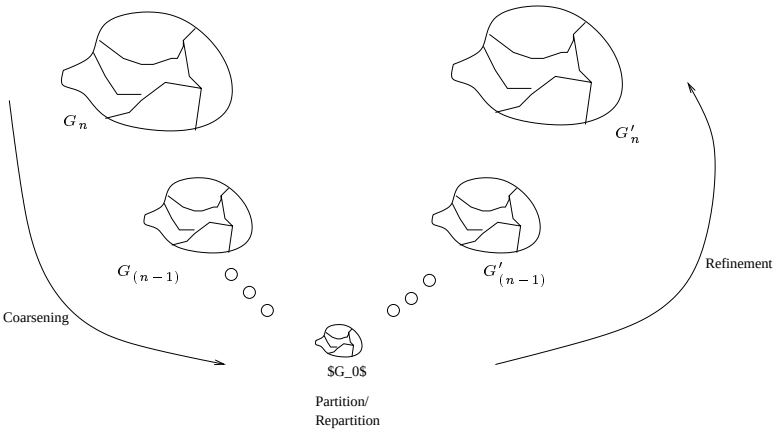


Abbildung 3: Berbeitungsphasen eines Multilevel - Partitionierungs / Repartitionierungsschemas.

Von den erwähnten Diffusion-basierten Multilevel-Repartitionierungsverfahren gibt es serielle und parallel ausführbare Varianten, die im folgenden näher vorgestellt werden sollen. Die Parallelen Varianten unterscheiden sich von den seriellen hinsichtlich der Gesamt-Vorgehensweise: Es werden die drei vorgestellten Phasen durchlaufen, jede einzelne von ihnen (bis auf eine Ausnahme) aber parallel.

¡

2.3.1 Serielles Coarsening

Vergröberung kann durch Erzeugung von Knoten-Matchings erfolgen. Für jede Detaillierungsstufe $G_(i+1)$ wird aus dem vorhergehenden Ausgangs-Graphen G_i durch ein maximales Kanten-Matching M_i ein verwandter, gröberer Graph durch Zusammenlegung der inzidierenden Knoten der betroffenen Kanten erzeugt (vgl. Abbildung 4) Übrigbleibende Knoten werden unverändert in den gröberen Graphen kopiert. Zusammenfallende Knotengewichte werden addiert. Der neu entstandene Knoten v_n aus ursprünglich zwei Knoten v_{a_1} und v_{a_2} inzidiert dann mit den inzidierenden Kanten von v_{a_1} und v_{a_2}, ausgenommen hiervon ist natürlich die Kante (v_{a_1}, v_{a_2}) selbst.

Das Matching kann auf verschiedene Weise erfolgen, beispielsweise per zufälliger Zuordnung oder es werden gezielt die schwersten Kanten zuerst zu einem Matching herangezogen, was in den weiteren Verarbeitungsschritten zu einem besseren Gesamtkantengewicht führt (vgl. Abbildung 4). Das hat in weiteren Verarbeitungsschritten zur Folge, daß die Partitionierung des Graphen an Kanten mit kleinen Gewichten orientiert ist, was sich entsprechend günstig auf den Kommunikationsaufwand (resp. Edge-Cut) auswirkt.

2.4 Serielle Undirected Diffusion

Bei der Undirected Diffusion handelt es sich um ein lokales Verfahren, das jeweils nur direkt aneinander angrenzende Partitionen berücksichtigt. Sie zielt grundlegend auf eine Verbesserung der Load-Balance ab, orientiert sich aber Fall-weise auch bereits vor der darauffolgenden Refinement-Phase am Edge-Cut, wie der folgende Pseudo-Code verdeutlichen soll.

```
PRE : Coarsening Phase bereits durchgeführt
POST : Initiale Repartitionierung auf grobem Level mit hinreichend guter
```

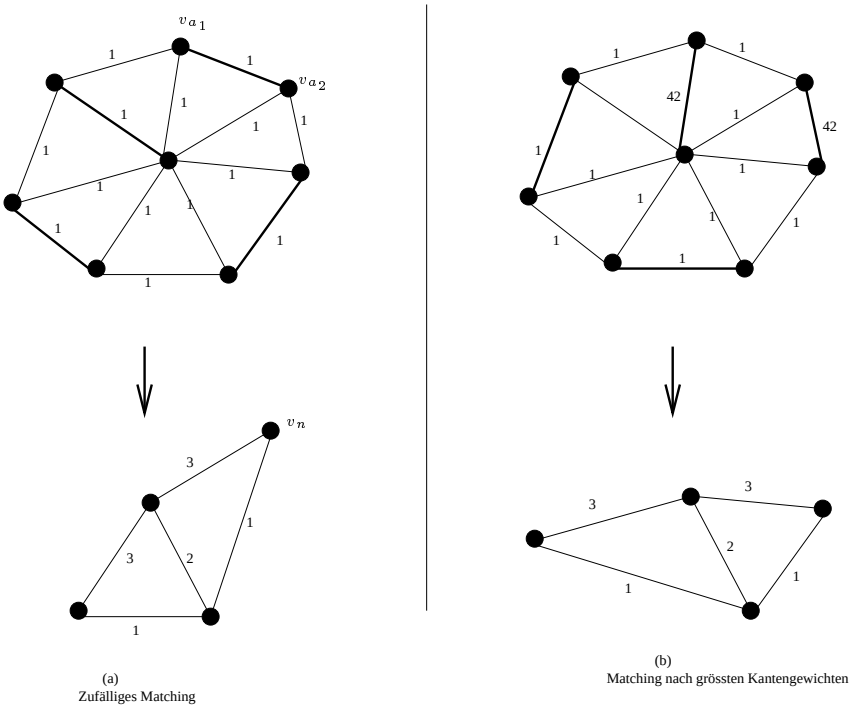


Abbildung 4: Ein Serieller Coarsening-Schritt durch Matching. (a) Zufalls-gesteuerte Variante, (b) Nach größtem Kanten-Gewicht orientiert.

```
        Load-Balance
Input : Gröbster Graph G[0] aus der Menge G[] der erstellten
        vergröberten Versionen des Graphen G
Output: Repartitionierter (noch nicht verfeinerter)
        Graph G[i] auf einem groben Detail-Level i

i := 0; // starte bei gröbstem Detail-Level
d := maximal erwünschter Detail-Level, d := _ klein_

W̄ := 1/k · (Σ_{j=1}^{k} weight(P_j)) // Durchschnittliches Gewicht
                // aller Partitionen P_j

while( (Toleranzschwelle für Load-Balance nicht erreicht)
        AND (i < d) )//solange Detail klein genug bleibt
{
    for( all P ∈ G[i] )
    {
        for(all k ∈ P mit k == Grenzknoten)
        {
            if( weight(P) > W̄ ){
                suche Nachbarpartition P_u zu P mit weight(P_u) < weight(P);
                if( mehrere gleich-untergewichtige P_u vorhanden ){
                    wähle diejenige P_e aus mit Edge_ cut(P_e,P) nach einer
                    eventuellen Ausführung = min.;
                    transferiere k nach P_e;
                }//if
                else{
                    transferiere k nach P_u;
```

```
            }//else
        }//if
        else{ // Gewicht der Partition also o.k. == im Durchschnitt W̄ liegend
            suche Nachbarpartition P_e zu P aus mit edge_ cut(P_e,P) nach einer
            eventuellen Ausführung = min.;
            transferiere k nach P_e;
        }//else
    }// for all k
}//for all P
projiziere Ergebnisgraphen G[i] auf G[i+1];
increment i;// Nächste Detail-Stufe
} //while
```

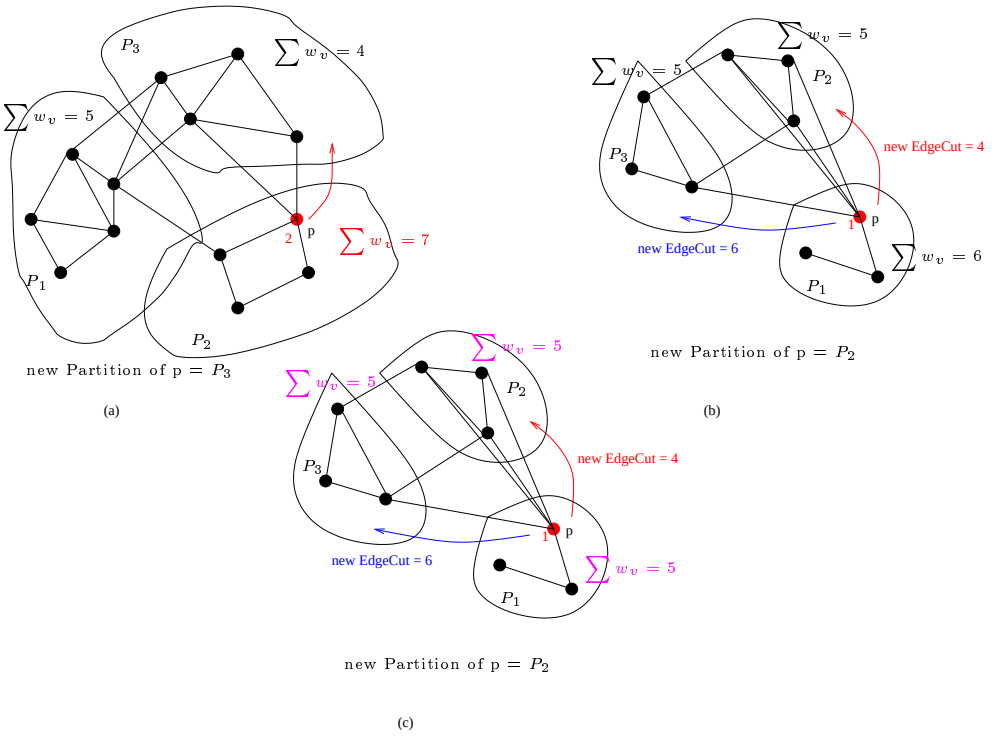


Abbildung 5: Die drei wichtigsten Entscheidungskriterien während eines elementaren Schrittes der Undirected Diffusion

Die wesentlichen Entscheidungen des Algorithmus, wann ein Knoten in eine andere Partition ausgelagert wird, sind zusätzlich in Abbildung 5 visualisiert. Im Fall (a) wird ein Knoten aufgrund des Übergewichts der Partition P_2 in eine benachbarte untergewichtige Partition P_3 verlagert. In Fall (b) entscheidet sich der Algorithmus bei der Auslagerung eines Knotens für Partition P_2, da sie gegenüber Partition P_3, die das gleiche Gewicht besitzt, einen besseren Edge-Cut erzeugt. Bei ausgeglichen schweren Partitionen, wie in Fall (c) gezeigt, wird sich wiederum anhand eines günstigeren Edge-Cuts entschieden.

2.5 Serielle Directed Diffusion

Hier wird im Gegensatz zum vorherigen Verfahren eine globale Sichtweise auf alle Partitionen des Graphen hinsichtlich der Knoten-Verschiebung berücksichtigt. Nach einem bestimmten Verfahren, auf das wir hier nicht weiter eingehen wollen, wird eine Knotentransfer-Matrix erzeugt, die im wesentlichen besagt, wieviel Gewicht zwischen benachbarten Partitionen ausgetauscht werden soll. Die wesentliche Vorgehensweise bleibt die gleiche, wie sie im vorhergehenden Abschnitt vorgestellt wurde. Der Unterschied besteht hauptsächlich in der Berücksichtigung der Transfer-Matrix, wenn es um die Entscheidung geht, ob ein Grenzknoten migriert werden soll oder nicht. Die prinzipielle Vorgehensweise sei wieder anhand von Pseudo-Code erläutert:

```
PRE : Coarsening Phase bereits durchgeführt
POST : Initiale Repartitionierung auf grobem Level mit hinreichend guter
       Load-Balance
Input : Gröbster Graph G[0] aus der Menge G[] der erstellten
       vergröberten Versionen des Graphen G
Output: Repartitionierter (noch nicht verfeinerter)
       Graph G[i] auf einem groben Detail-Level i

i := 0; // starte bei gröbstem Detail-Level
d := maximal erwünschter Detail-Level, d := _ klein_

Erzeuge Transfer-Matrix M aus Graphen G;

while( (Toleranzschwelle für Load-Balance nicht erreicht)
AND (i < d) )//solange Detail klein genug bleibt
{
    for( all P ∈ G[i] )
    {
        for(all k ∈ P mit k == Grenzknoten)
        {
            if( k mit einer Partition P_u benachbart, die anhand M für einen
                Transfer in Frage kommt ){
                if( M bietet mehrere gleichwertige Partitionen an ){
                    suche Partition P_e mit edge_cut(P,P_e) = min im Falle
                    eines Knoten-Transfers;
                    transferiere k nach P_e;
                }//if
                else{
                    transferiere k nach P_u;
                }//else
            Aktualisiere M;
            }//if
        }// for all k
    }// for all P
    projiziere Ergebnis von G[i] auf G[i+1];
    increment i;// nächste Detail-Stufe
}//while
```

2.5.1 Serielles Multilevel Refinement

Die abschließende Multilevel-Refinement-Phase ist einer Undirected Multilevel-Diffusion-Phase sehr ähnlich, nur geht es in dieser Phase primär um die Verbesserung des Edge-Cuts.

Das allgemeine Prinzip kann man wie folgt umschreiben:

```
PRE : Multilevel-Diffusion endete auf Graph G[i] mit Detaillierungsstufe i,
    und i < g; g entspricht voller Detaillierungsstufe
Input : Graph G[i]
Output: Graph G[g]

while( i < g )
{
    for( all Partitionen P ⊂ G[i] )
    {
        for( all Knoten k ∈ P mit k == Grenzknoten )
        {
            suche benachbarte Partition P';
            if( ( P' = initiale Partition von k//k befand sich
                    bereits in vorangegangenen Schritten in P'
            OR ( edge_cut(P',P) würde sich im nächsten Schritt verbessern
                AND load_balance(G[i]) bleibt hinreichend gut )
            OR ( load_balance(G[i]) würde sich im nächsten Schritt verbessern
                AND edge_cut(P,P') bleibt hinreichend gut)
            {
                transferiere k nach P';
            }//if
        }//for all k
    }//for all P
    projiziere Ergebnis von G[i] auf G[i+1];
    increment i;
}//while
```

Anmerkung: Der Grund, warum man einen Knoten wieder zurück in eine Partition versetzt, aus der er ursprünglich kam besteht in den resultierenden Migration-Costs: Bleibt der Knoten dort, wo er herkam, dann muß er nach der erfolgten Repartitionierung nicht in eine andere Partition verschoben werden. Entsprechend reduzieren sich die Werte für TotalV und MaxV.

2.5.2 Paralleles Coarsening

Wie in der seriellen Variante auch, wird in dieser Phase eine Sequenz von sukzessive gröber werdenden Graphen $G_i = (V_i, E_i)$ erzeugt, wobei jeweils $G_{(i+1)}$ aus G_i durch Berechnung eines Matchings der Knoten erzeugt wird. Ein Matching wird auf jedem Prozessor separat durchgeführt, und zwar auf der ihm zugeordneten Knotenmenge (vgl. Abbildung 6), was eine gute parallele Skalierbarkeit impliziert. Das setzt natürlich voraus, daß vorher bereits eine initiale Partitionierung durchgeführt wurde. Abbildung 6 und der darauffolgende Pseudo-Code stellen die wesentlichen Ideen hinter dem Verfahren vor.

```
Input : Ausgangs-Graph in vollem Detail G
```

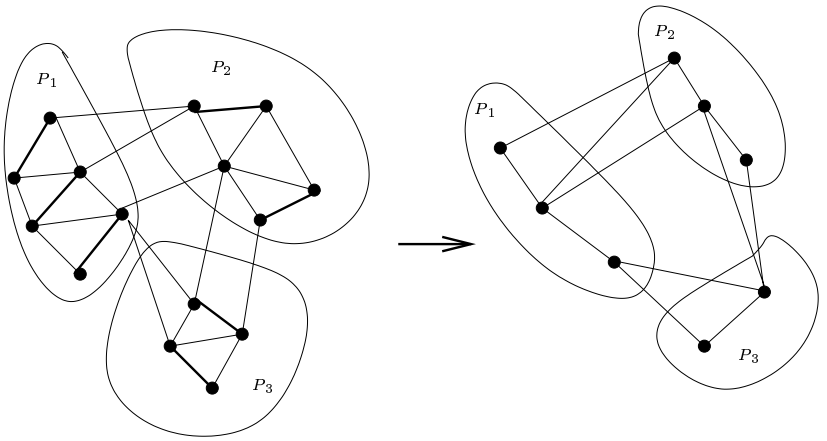


Abbildung 6: Paralleles Coarsening. Das Matching wird nur innerhalb der Partitionen durchgeführt.

```
Output: Menge G[] von Graphen in n verschiedenen Vergröberungsstufen,
    G[n] entspricht der gröbsten Stufe
PRE : Initiale Partitionierung auf G wurde bereits durchgeführt

n := Anzahl der Detaillierungsgrade
G[0] := Originalgraph G
k := Anzahl der Prozessoren

for( i = 0 to (n-1))
{
    G[i+1] := G[i];

    for( all Partitionen P ⊂ G[i+1] ) in parallel
    {
        compute_matching(P);
        update_partition_of_graph(G[i+1],P);
    }//for
}//for
```

2.5.3 Parallele Directed Diffusion

Bei der vorgestellten Directed-Variante der parallelen Diffusion wird auf der gröbsten Stufe des Graphen *seriell* ein Directed Diffusion-Schritt durchgeführt. Das ist nicht weiter kritisch für die Gesamt-Performance des Algorithmus, da es sich hierbei nur um einen einzelnen kleinen Schritt im Gesamt-Algorithmus handelt, und dieser aufgrund seiner Anwendung auf die gröbste Detail-Stufe nur auf eine entsprechend kleine Knoten- und Kantenmenge angewandt wird (s. auch **??**). Währenddessen werden alle anderen Prozessoren ebenfalls beschäftigt: Der besagte gröbste Graph wird auf alle Prozessoren mit einem Broadcast verteilt, danach wird separat auf jedem der Prozessoren nach dem bereits vorgestellten Verfahren der seriellen Directed Diffusion eine Umverteilung der Knoten vorgenommen. Jeder berechnet so

seinen individuellen Graphen[2]. Aus diesen Graphen wird nach bestimmten Kriterien der beste ausgewählt. In ?? wird nach geringstem Edge-Cut selektiert, weiterhin sind auch eine Auswahl von niedrigstem TotalV, niedrigstem MaxV oder der besten Load-Balance möglich. In etwas formalerer Form läßt sich obiges so ausdrücken:

```
Input : Graph G[n] mit gröbstem Detaillierungsgrad
Output: Neu partitionierter Graph G[n] in k verschiedenen Versionen,
    k entspricht der Anzahl der Prozessoren
PRE : Coarsening bereits durchgeführt
POST : Neu partitionierter Graph hat einen guten Edge-cum

G[n] := Graph mit gröbstem Detaillierungsgrad n
G_e := zukünftiger Ergebnisgraph
D[k] := Feld von Graphen mit Detaillierungsgrad n
    zum späteren Vergleich der parallel errechneten
    Ergebnisgraphen

for( i = 0 to (k-1)) in parallel
{
    D[i] = compute_directed_diffusion(G[n]);

}//for

G_e := select_minimal_edge_cut(D[]);
G[n] := G_e
```

Diese Art der Diffusion auf einem einzelnen Level ergibt nicht immer eine zufriedenstellende Load-Balance. Daher werden im Anschluss noch einige Schritte der im folgenden erklärten Undirected Diffusion durchgeführt, um die Balance zu verbessern (zumindest im vorgestellten Verfahren unter ??).

2.5.4 Parallele Undirected Diffusion

Die im Artikel vorgestellte Undirected-Variante arbeitet ähnlich wie ein Refinement-Algorithmus. Pro Iteration werden zwei Subphasen durchlaufen. Ausgehend von einer bestehenden Numerierung der Partitionen mit Indices, werden in der ersten Subphase parallel für jede Partition nur Knoten von Partitionen niedrigerer Numerierung hin zu Partitionen höherer Numerierung transferiert. In der darauffolgenden zweiten Subphase wird das umgekehrte getan: Knoten wandern nur von Partitionen höherer Numerierung hin zu Partitionen mit niedrigerer Numerierung.

Das hat folgenden Sinn: Da die Diffusion der Knoten zwischen Partitionen parallel abläuft, kann es passieren, daß Knoten zweier unausgeglichener Partitionen zeitgleich zwischen ihnen hin und her transferiert werden, sodaß sich die Load-Balance nicht verbessern kann.

Abbildung 7 zeigt diese Problematik im Teilbild (a): Der Prozessor, welcher eine Diffusion für Partition P_1 durchführen soll, entscheidet sich, Knoten v in Partition P_2 zu verschieben, um den Edge-Cut von 5 auf 3 zu verringern. Zeitgleich versucht ein anderer Prozessor, der für die Diffusion der Knoten aus Partition P_2 zuständig ist, das gleiche: Er will den Edge-Cut von 5 auf 2 reduzieren, indem er Knoten u in Partition P_1 verschiebt. Das Ergebnis sieht man im unteren Teil der Abbildung: Der

[2]Bisher haben wir verschwiegen, daß die Auswahl der Grenzknoten in allen Diffusion-Schritten nach dem Zufallsprinzip erfolgt. Daher entstehen im obigen Fall jeweils neue Partitionierungen

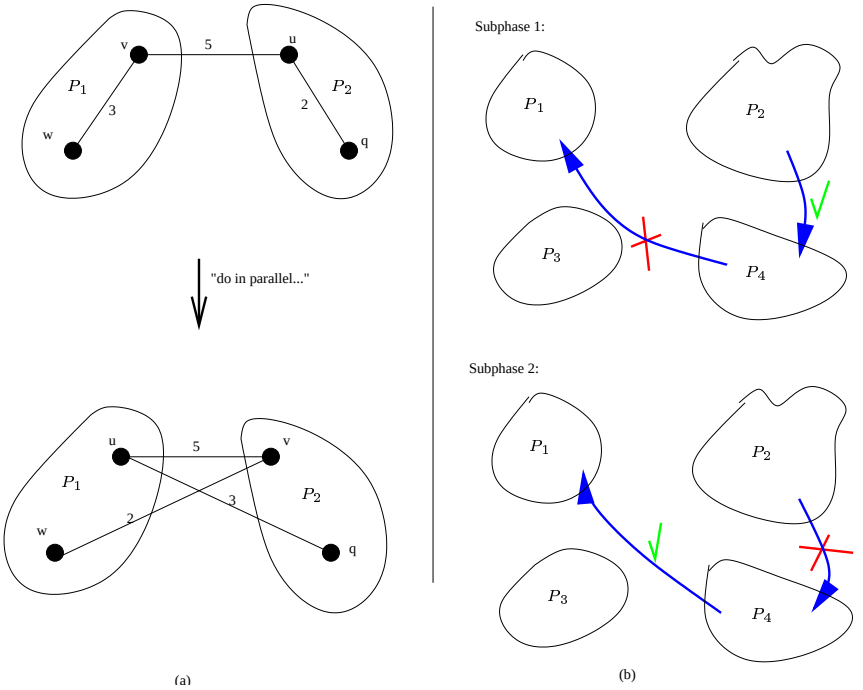


Abbildung 7: Parallel Undirected Diffusion: (a) Problematik der zeitgleichen Knoten-Verschiebung. (b) Prinzipielle Idee der Subphasen.

Edge-Cut hat sich durch die zeitgleiche Diffusion auf 10 verschlechtert. Teil (b) der Abbildung zeigt noch einmal ergänzend das Grundprinzip der beiden Subphasen, wie sie im folgenden wieder in Form von Pseudo-Code vorgestellt werden.

```
Input : Graph G[0] in gröbster Detail-Stufe mit k Partitionen
Output: Repartitionierter Graph auf einer mittleren Detail-Stufe
PRE : Paralleles Coarsening bereits durchgeführt
    und Partitionen sind mit einem Index gekennzeichnet
POST : Hinreichend gute Load-Balance auf zuletzt erreichter Detail-Stufe

i := 0; // starte bei gröbstem Detail-Level
d := maximal erwünschter Detail-Level, d: = _klein_

W̄ := 1/k · Σ_{j=1..k} weight(P_j) // Durchschnittliches Gewicht
                                  // aller Partitionen P_j

while( (Toleranzschwelle für Load-Balance noch nicht erreicht)
       AND (i < d) )//solange Detail klein genug bleibt
{
    //Phase 1:
    for( all P_q ∈ G[i] ) do in parallel
    {
        for(all k ∈ P_q mit k == Grenzknoten)
        {
            if( weight(P) > W̄ )
            {
```

```
                <label> suche Nachbarpartition P_r zu P_q mit
                (r < q) AND (weight(P_r) < weight(P_q));
                if( mehrere gleich-untergewichtige P_r vorhanden ){
                    wähle diejenige P_e mit edge_cut(P_e,P_q) = min.
                        nach einer eventuellen Ausführung;
                    transferiere k nach P_e;
                }//if
                else{
                    transferiere k nach P_r;
                }//else
            }//if
            else // Gewicht der Partition also o.k.
            // == im Durchschnitt W̄ liegend
            {
                suche Nachbarpartition P_e zu P_q aus mit edge_cut(P_e,P_q) =
                    minimal nach einer eventuellen Ausführung;
                transferiere k nach P_e;
            }//else
        }//for all k
    }//for all P_q

    //Phase 2
    // entspricht exakt Phase 1 mit Ausnahme von <label>:
    // --> suche Nachbarpartition P_r zu P_q mit
    // (r > q) AND (weight(P_r) < weight(P_q));

    projiziere Ergebnisgraphen G[i] auf G[i+1];
    increment i; // Nächste Detailstufe

}//while
```

Die Auswahlkriterien zur Migration der Knoten in je einer Subphase entsprechen exakt den Auswahlkriterien der seriellen Undirected Multilevel-Diffusion mit der Ausnahme, daß hier noch die Indizierung der Partitionen mitberücksichtigt wird.

2.5.5 Parallel Multilevel Refinement

Die abschließende parallel durchgeführte Multilevel-Verfeinerung funktioniert nach dem gleichen Schema wie oben in der parallelen Undirected-Variante des Multilevel Diffusionings gezeigt, mit der Ausnahme, daß in den Verfeinerungsschritten zur Entscheidung, ob ein Knoten migriert werden soll oder nicht, weitere Kriterien hinzugezogen werden, und zwar exakt die gleichen, wie sie schon im seriellen Multilevel-Refinement vorgestellt wurden. Erneuten Pseudo-Code möchten wir uns an dieser Stelle sparen.

3 Experimente

Test-Objekte: Paralleler Multilevel Directed Repartitionierungs-Algorithmus. Paralleler Multilevel undirected Repartitionierungs-Algorithmus.

Umgebung: Cray T3D mit jeweils 64, 128 und 256 150Mhz Prozessoren vom Typ Dec Alpha (EV4).Die Prozessoren sind untereinander mit einem dreidimensionalen Torus-Netzwerk verbunden, welches eine hohe Bandbreite besitzt. Als Kommunikations-Software wurde eine Cray-Software-Bibliothek verwendet.

3.1 Zu den Eingabedaten und Ausgabe-Daten

Als Eingabedaten dienten vier mittelgroße bis grosse Graphen, die synthetisch hergestellt wurden. Diese wurden zum Test der Repartitionierungs-Algorithmen künstlich per Zufalls-Generator verändert (Kantengewichte wurden verändert). Folgende Abbildung zeigt einige Eckdaten zu den Graphen:

Graph Name	Anzahl der Knoten	Anzahl der Kanten	Beschreibung
A	257000	505048	Dual of a 3D Finite element mesh
B	1017253	2015714	Dual of a 3D Finite element mesh
C	4039160	8016848	Dual of a 3D Finite element mesh
D	7833224	15291280	Dual of a 3D Finite element mesh

Abbildung 8: Eckdaten zu den Eingabe-Graphen.

Zur Erläuterung: Für eine 64-Prozessor-Maschine wurde eine 64-fache Partitionierung erzeugt. Jeder der 64 Prozessoren erhielt so seine eigene Partition. Im nächsten Schritt —um also künstlich eine Imbalance zu erzeugen— wählt jeder Prozessor per Zufall einen Knoten aus seiner Partition aus, und setzt dessen Gewicht von 1 auf 3. Dabei haben Schloegel et al. festgestellt, daß diese Vorgehensweise Partitionen erzeugt, deren Gewicht etwa 50% über dem Durchschnittsgewicht liegen. Die Graphen, welche auf diese Weise erzeugt wurden, dienten als Eingabedaten für unsere Diffusion-Algorithmen.

Es wurde dabei vorausgesetzt, daß ein Graph gut genug ausbalanciert ist, wenn er eine restliche Imbalance von 5% enthält. Dieser Wert wurde auch in allen Experimenten mit den Diffusion-Verfahren erreicht.

Zum Output: Die folgende Tabellen und Graphen illustrieren die Ergebnisse und Erkenntnisse der Tests. Bei den Graphen konzentriert man sich auf die Ergebnisse von:

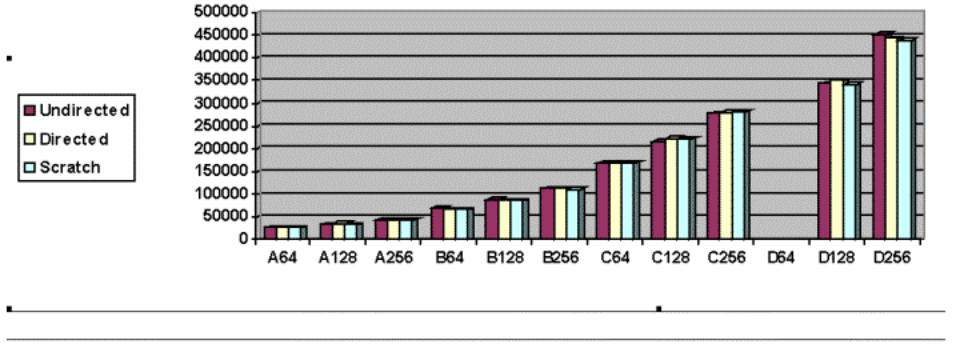
- Adaptive Multilevel Directed Diffusion Algorithmus
- Adaptive Multilevel undirected Diffusion Algorithmus
- Repartitioning from Scratch

Im folgenden werden Edge-Cut, TotalV, MaxV und Run Time näher untersucht. Vorbemerkung: Es wurde in allen Fällen eine Load-Imbalance erzeugt, die etwa

zwischen 40% und 50% lag. Alle Werte bezüglich der Laufzeit sind in Sekunden gemessen worden.

3.2 Ergebnisse

3.2.1 Edge-Cut



	A64	A128	A256	B64	B128	B256	C64	C128	C256	D64	D128	D256
Undirected	26457	32898	42536	69428	87687	112522	169336	217165	278607		345122	452386
Directed	25895	34635	42502	67032	86836	112199	169559	223031	279832		350334	445263
Scratch	25099	32855	43130	65395	85812	110459	167617	222368	281195		343354	438704

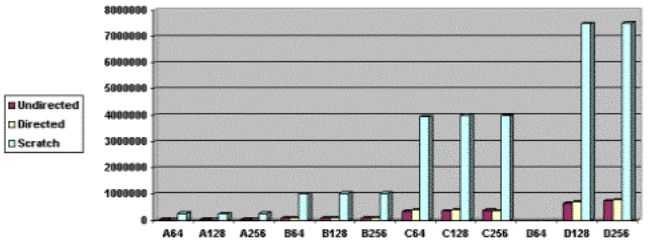
Abbildung 9: Vergleiche der Edge-Cuts.

Diese Übersicht zeigt die Ergebnisse des Multilevel Directed und undirected Diffusion Algorithmus für die Test-Beispiele mit 64, 128 und 256 Prozessoren hinsichtlich des Edge-Cuts und lässt den Betrachter auch einen Vergleich ziehen im Bezug auf die Repartitionierungs From-Scratch. Wie man anhand dieses Balken-Diagramms erkennen kann, ist das Testergebnis der Partitionierung mit den Multilevel Directed und Undirected Diffusion Algorithmen im Hinblick auf den Edge-Cut fast gleich gut. Auch ist anzumerken, daß der Edge-Cut sehr nahe an den Ergebnissen der From-Scratch-Verfahren liegt. Man sieht, daß beide Multilevel-Verfahren ähnlich gute Ergebnisse erzielen.

3.2.2 TotalV und MaxV

Dieses Diagramm zeigt, wie viele Knoten insgesamt bei der Benutzung des Adaptive Multilevel Directed und Undirected Diffusion Algorithmen bewegt werden müssen. Verglichen mit der From-Scratch-Partitionierung erkennt man, daß die From-Scratch-Partitionierung im Vergleich zu den adaptiven Multilevel Diffusion Algorithmen eine signifikant höhere Anzahl der zu bewegenden Knoten aufweist: Multilevel-Diffusion-Verfahren bewegen im Vergleich nur etwa 10% der Knoten.

—hier MaxVTable Dieser Graph zeigt uns die maximalen Knotenmengen, die aus einer Partition rein und raus bewegt werden mußten. Man sieht hier im Ver-



	A64	A128	A256	B64	B128	B256	C64	C128	C256	D64	D128	D256
Undirected	24819	24089	25423	92806	94704	94328	342078	358453	369216		640172	743892
Directed	25858	26670	25620	94397	90785	95074	386720	382812	351109		700727	768296
Scratch	253856	252054	256020	996825	1016832	1016166	3943621	3998118	3984525		7487361	7507463

Abbildung 10: Übersicht zu TotalV.

gleich, daß die Multilevel-Diffusion-Verfahren überall bessere Ergebnisse erzielen als die From-Scratch-Methode.

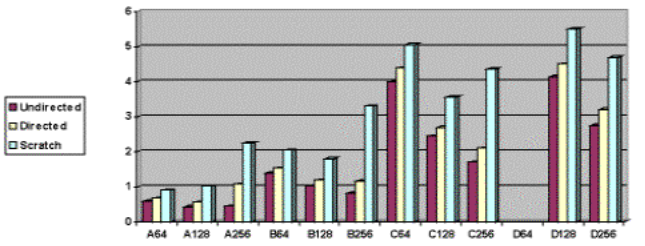
3.2.3 Zur Laufzeit

Obiges Diagramm enthält vergleichende Werte bezüglich der Laufzeiten der Algorithmen auf den vorgestellten Test-Architekturen. Auch hier schneiden die Diffusion-basierten Verfahren besser ab.

3.3 Abschließende Bemerkungen

Trotz der vorhergehenden Coarsening-Phase haben sich die vorgestellten Algorithmen als schnelle Algorithmen herausgestellt, die neben der Verbesserung von Load-Balance und dem Edge-Cut auch die Migrationskosten für Knoten gering halten. Wenn man sich die gezeigten Tabellen vergleichend betrachtet, dann erkennt man, daß der ganz klare Vorteil der Diffusion-Verfahren darin liegt, daß sie die Migrationskosten für Knoten gering halten Diffusion-basierte Verfahren haben jedoch bei sehr stark ändernden Graph-Strukturen Probleme, dort lohnt es sich wieder, From-Scratch-Verfahren einzusetzen.

In der Praxis haben sich die Software-Bibliotheken METIS und PARMETIS etabliert, welche die hier vorgestellten Algorithmen implementieren. Diese Bibliotheken bieten fertige Funktionalitäten zur Partitionierung von Graphen, Finite-Elemente-Strukturen und Sparse-Matrizen. Beide Bibliotheken sind frei verfügbar.



	A64	A128	A256	B64	B128	B256	C64	C128	C256	D64	D128	D256
Undirected	0.573	0.424	0.436	1.377	1.017	0.817	3.994	2.435	1.689		4.145	2.747
Directed	0.666	0.547	1.066	1.534	1.174	1.161	4.385	2.685	2.092		4.511	3.203
Scratch	0.902	1.019	2.247	2.039	1.799	3.294	5.056	3.558	4.352		5.511	4.692

Abbildung 11: Vergleich der Laufzeiten.

Literatur

[1] Schloegel et al.: *Parallel Multilevel Diffusion Algorithms for Repartitioning of Adaptive Meshes*, University of Minnesota, Technical Report

[2] Karypis, Kumar: *A coarse-grain parallel multilevel k-way partitioning algorithm.*, Proceedings of the 8th SIAM conference on Parallel Processing for Scientific Computing, 1997

[3] Schloegel et al: *Multilevel Diffusion Schemes for Repartitioning of adaptive Meshes*, Technical Report 97-013, University of Minnesota, 1997

[4] Webseite: *METIS und PARMETIS Graph Partitioning Librarys*, URL: http://www.cs.umn.edu/ metis

[5] Schloegel et al.: *Graph Partitioning for High Performance Scientific Simulations*, Manuskript eines Buch-Kapitels, University of Minnesota, 2000