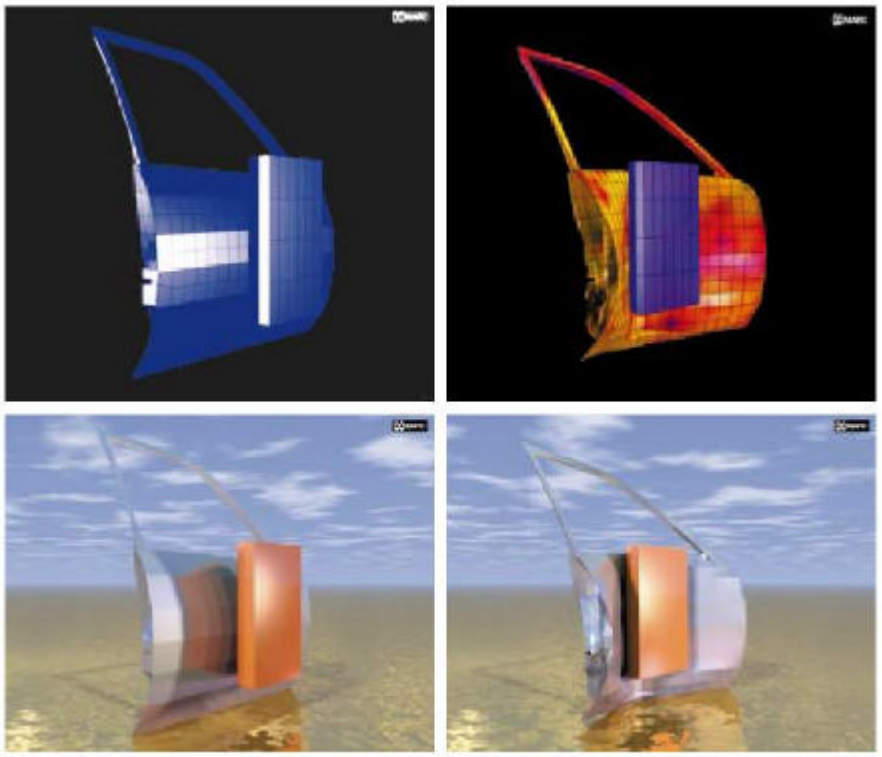
Ausarbeitung des Papers
Paralleler Multilevel Algorithmus für Multigewichtige Graphen Partition

Kirk Schloegel, George Karypis und Vipin Kumar
Army HPC Research Center
Department of Computer Science and Engineering
University of Minnesota,
Minneapolis, MN 55455

Technikal Report: TR 99-031

Juni 10, 2002



Ioannis Savvidis für das Fach Parallele Algorithmen

Inhaltverzeichnis

1. Einleitung

Die Partition von Graphen ist ein sehr praktisches Anwendungsgebiet der parallelen Programmierung. Graphen werden benutzt, um eine Simulation zu interpretieren. Früher war es der Fall, dass diese Graphen bei ihren Knoten nur ein Gewicht hatten. Deshalb wurden auch parallele eingewichtige Graphenpartitionen entwickelt. In den vergangenen Jahren aber, hat die Komplexität und die Genauigkeit von Modellen, die von wissenschaftlichen Simulationen benutzt werden, so zugenommen dass die traditionellen Graphenpartitionsalgorithmen ungenügend geworden sind. Zum Beispiel wird in multiphysischen Simulationen, eine Variation von Materialien und Prozessen zusammen simuliert. Das Resultat ist eine Klasse von Problemen, in denen die Berechnungs- und Speicherungsvoraussetzungen nicht gleichmäßig auf die Mengen verteilt sind. Existierende Partitionsalgorithmen können benutzt werden, um die Menge der Berechnung oder der Speicherung so zu dividieren, dass jeder einzelne Prozessor eine gleiche Menge hat. Aber sie sind nicht in der Lage beide Mengen gleichmäßig zu verteilen. Diese Schwäche kann dazu führen, dass die Berechnungsmengen sehr ungleichmäßig verteilt werden und so die generelle Effizienz begrenzt wird, oder dass die Speicherungsmengen ungleichmäßig verteilt werden, so dass die Größe des Problems die gelöst werden soll begrenzt wird. Die Abbildung 1 zeigt so ein Problem. Es zeigt einen Graphen an, dessen Knoten verschiedene Berechnungsmengen und Speichermengen haben, der drei verschiedene Partitionen geben kann. Die Partition in der Abbildung 1(b) verteilt gleichmäßig die Berechnungsmengen aber ungleichmäßig die Speichermengen. Die Partition in der Abbildung 1(c) verteilt gleichmäßig die Speichermengen, aber ungleichmäßig die Berechnungsmengen. Die Partition in der Abbildung 1(d), die beide Mengen gleichmäßig verteilt hat ist die gewünschte Partition. Generell benötigen multiphysikalische Simulationen Partitionen, um nicht nur eine eingewichtige Menge zu verteilen, sondern eine größere Anzahl von Gewichten. (In diesen Kapitel müssen zweigewichtige Knoten gleichmäßig verteilt werden, Berechnungsmengen und Speichermengen). Diese Voraussetzung ist ebenfalls in Multiphasen Berechnungen präsent, in der verschiedene (wahrscheinlich übereinander kommende) Elemente der Knoten, in verschiedene Phasen von Berechnungen teilnehmen.

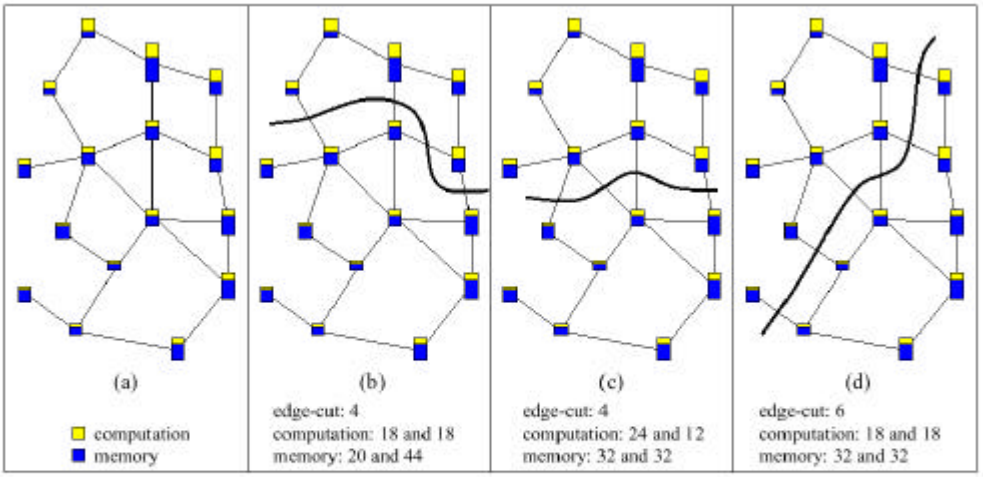


Figure 1: An example of a computation with non-uniform memory requirements. Each vertex in the graph is split into two colors. The size of the yellow portion represents the amount of computation associated with the vertex, while the size of the blue portion represents the amount of memory requirement work associated with the vertex. The partitioning in (b) balances the computation. The partitioning in (c) balances the memory, but only the partitioning in (d) balances both of these.

Die allgemeine Charakteristik von solchen Problemen ist, dass alle Berechnungen von Partitionen mehr als ein Gewicht erfüllen müssen. Traditionelle Graphen Partitionstechniken wurden kreiert, um nur ein Gewicht gleichmäßig zu verteilen. Eine Erweiterung von dem Graphenpartitionsproblem, dass es uns erlaubt ein Gleichgewicht mit vielen Gewichten zu schaffen, ist einen Gewichtsvektor für jeden Knoten zu bestimmen, der eine Größe von *m* hat. Das Problem, der Partition mit einen minimalen Kantenschneiden, wird dann abhängig von der Begrenzung, dass jedes von den *m* Gewichten, gleichmäßig in den Teilen verteilt werden muss. Solch ein multigewichtiger Graphenpartitionsalgorithmus sowie ein Sequentieller Algorithmus für multigewichtige Partition wird dargestellt in [2].

Es ist wünschenswert einen multigewichtigen Partitionsalgorithmus parallel zu berechnen, für eine Anzahl von Gründen. Berechnungsmengen in parallelen und wissenschaftlichen Simulationen sind oft zu groß, um den Speicher eines Prozessors zu besetzen. Außerdem sind Anpassungsberechnungen, welche die Menge braucht, um verteilt zu werden oft so groß wie der Simulationsprozess. In solchen Berechnungen kann das Runterladen der Mengen zu einen einzelnen Prozessor, für die Wiederverteilung, zu einen großen Verkehrsstau führen. So ist ein wirkungsvoller paralleler multigewichtiger Graphenpartitionsalgorithmus, zur leistungsfähigen Ausführung der großen Mehrphasen- und Multiphysikprobleme, der Schlüssel zur ganzen Sache.

Der ausarbeitete parallele multigewichtige Partitionsalgorithmus besteht aus drei Phasen. Es ist so zu sagen eine Kombination von dem sequentiellen multigewichtigen Partitionsalgorithmus von [2], der parallelisiert worden ist mit den Techniken, die benutzt worden sind, um einen parallelen eingewichtigen Partitionsalgorithmus zu entwickeln in [3]. Die drei Phasen sind die coarsening (verkleinerungs) Phase, die Ausgangspartition (initial Partitioning) und die Optimierungsphase. (Abbildung 2) In der coarsening Phase ist der ursprüngliche Graph mehrmals hintereinander nach unten verkleinert, bis es nur eine kleine Anzahl von Knoten hat. In der Ausgangspartitionsphase wird eine Partition des verkleinerten Graphen berechnet. In der Mehrebenenoptimierungsphase wird das Ausgangsverteilen mehrmals hintereinander mit einer heuristischen Kernighan-Lin (KL) Art verfeinert [5] wie es zurück zu dem ursprünglichen Graphen projiziert wird. Die Bearbeitung dieser Phasen kann uns direkt zu den parallelen Formulierungen des coarsening und des Ausgangsverteilens für multigewichtige Partitionen führen. Die Schlüsselherausforderung ist die parallele Formulierung der Optimierungsphase. Die Optimierungsphase mit einem eingewichtigen Partitionsalgorithmus wird parallelisiert, indem man die KL heuristisch Funktion expandiert, soweit dass die Optimierung parallel durchgeführt werden kann während sie wirkungsvoll bleibt. Diese Expansion kann die Partition veranlassen, während des Optimierungsprozesses unausgeglichen zu werden, aber die Ungleichheiten werden schnell in folgenden Wiederholungen behoben. Schließlich wird eine ausgeglichene Partition am feinsten waagerecht ausgerichteten Graphen erreicht. Tatsächlich ist die Herausforderung der ausgleichenden multigewichtigen Partitionen so schwierig, dass eine bessere Lösung Situationen vermeiden soll, in denen die Partition Unausgeglichen wird. Dieses kann korrigiert werden, indem man entweder den Optimierungsalgorithmus sequentiell ausführt, oder sonst, indem man die Menge der Optimierung einschränkt, die ein Prozessor in der Lage ist, durchzuführen. Das erste verringert die Komplexität des Algorithmus und aus das zweite ergibt eine schlechte Partitionsqualität. Kein von diesen beiden ist wünschenswert. Folglich liegt die Herausforderung in der Entwicklung eines parallelen multigewichtigen Graphenpartitionsalgorithmus, eine Expansion des Optimierungsalgorithmus zu entwickelt, die gleichzeitig wirkungsvoll ist, und die Lastabgleichung für jedes Gewicht beibehält.

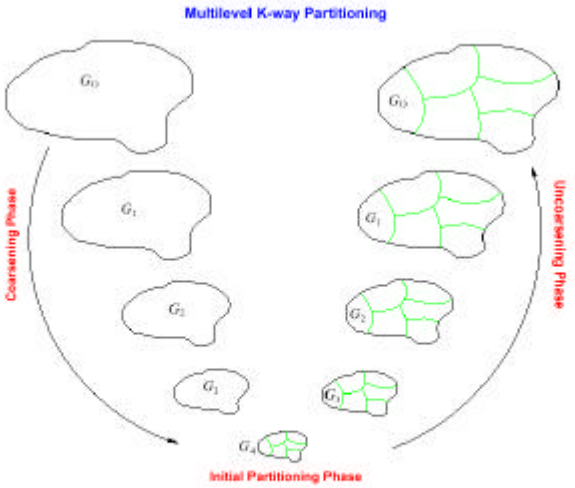


Figure 2: The three phases of multilevel k-way graph partitioning. During the coarsening phase, the size of the graph is successively decreased. During the initial partitioning phase, a k-way partitioning is computed. During the multilevel refinement phase, the partitioning is successively refined as it is projected to the larger graphs. G_0 is the input graph, which is the finest graph. G_{i+1} is the next level coarser graph of G_i. G_4 is the coarsest graph.

2. Hintergrund

Es folgen mathematische Interpretationen, die bei der Entwicklung des Algorithmus benutzt wurden.

2.1 Uniformierte Graphen Partition Kernighan – Lin

Definition UGP:
Geben einer symmetrischen Kostenmatrix [d_{ij}] die einen undirekten Graphen $G = (V, E)$ definiert mit $|V| = 2n$ Knoten, nennt man eine Partition $V = A \dot{\cup} B$, so dass $|A| = |B|$ ***uniformierte Partition***. Das uniformierte partitions Problem ist das finden einer Partition $V = A \dot{\cup} B$, so das die Kosten $C(A, B) = \sum_{i \in A, j \in B} d_{ij}$ minimal sind.

Definition des Swaps:
Gegeben einer uniformierten Partition A, B und den Elementen $á \in A$ und $b \in B$, nennt sich die folgende Operation

$$A' = (A - \{á\}) \dot{\cup} \{b\}$$
$$B' = (B - \{b\}) \dot{\cup} \{á\}$$

Swap.

Wenn á ∈ A, dann
Externe Kosten: $E(á) = \sum_{i \in B} d_{áé}$

Interne Kosten: $I(á) = Ó_{j \, î A} d_{áj}$

$D(v) = E(v) – I(v)$

Lemma 1: *Das Swap von á und b gibt eine Verminderung der Kosten (gains) von g(á, b) = D(á) + D(b) – 2d$_{áb}$.*

Algorithmus:

1. ∀ v werden die D(v) berechnet,
2. Wir nehmen Paare(á$_1$, b$_1$) (á$_2$, b$_2$) ...(á$_k$, b$_k$),so dass g$_1$ so groß wie möglich ist (nicht unbedingt positiv)
3. For i = 1,...,k
   Für jedes Swap kalkulieren wir die D wieder mit
   D'(x) = D(x) + 2d$_{xá'i}$ – 2d$_{xb'i}$, x ∈ A – {á$_i$}
   D'(y) = D(y) + 2d$_{yá'i}$ – 2d$_{yb'i}$, y ∈ B – {b$_i$}
4. Wir wählen k so das G(k) = $Ó_{i=k}^{k}$ g$_i$ Maximum ist. Wenn G(k) ≤ 0 stoppen wir. Wenn G(k) > 0 fangen wir wieder bei Schritt 1 an. Wenn k = n ist dann ist logischer Weise G(k)= 0 und das heißt, dass alle Knoten die Seiten gewechselt haben.

2.2 Multigewichtige Graphenpartition

Das multigewichtige Partitionsproblem kann wie folgt formuliert werden. Betrachten Sie einen Graphen. *G = (V, E)*, so dass jeder Knoten *õ Î V* hat einen Gewichtvektor w^{õ} von der Größe *m*, die mit ihr dazugehörig sind, und von jeder Kante *e Î E* hat ein Skalargewicht, das zu ihm angebracht wird. Es gibt keine Beschränkungen auf den Gewichten der Kanten, aber wir nehmen, ohne Verlust des Allgemeinen, dass die Gewichtvektoren der Knoten die Eigenschaft erfühlen, die $Ó_{"õ î V} w^{õ}_i = 1.0$ für *i = 1, 2,…,m* an. Falls die Knotengewichte nicht die oben genannte Eigenschaft erfühlen, können wir jedes $w^{õ}_i$ mit $Ó_{"õ î V} w^{õ}_i$ dividieren, um sicherzugehen, dass die Eigenschaft befriedigt wird. Beachten Sie, dass diese Normalisierung nicht in keiner Weise unsere Formenfähigkeit begrenzt. Wenn *P* der Partitionsvektor der Größe |*V*| ist, so, dass für jeden Knoten *õ, P[õ]* die Teilnummer speichert, zu der *õ* gehört. Die Lastsungleichheits l_i in bezug auf das *ith* Gewicht der k- Partition wird wie folgt definiert:

$$l_i = k \max_j \left(\sum_{\forall v : P[v]=j} w_i^v \right) \qquad (1)$$

Wenn die k- Partition tadellos ausgeglichen ist, dann ist $Ó_{"õ:P[õ]=j} w^{õ}_i$ für alle *j* gleich *1/k*, und l_i *= 1.* Eine Lastungleichheit von $l_i = x$ zeigt, dass eine Berechnung von Größe *W* an den *k* Prozessoren während der *ith* Phase *xW/k* Zeit in Anspruch nimmt, anstelle von der *W/p* Zeit, die im Fall vollkommener Lastausgleichung erforderlich ist. Eine Lastungleichheit von *1 + á* zeigt an, dass die Partition eine Ungleichmäßigkeit von *á%* hat.

Das Ziel ist, eine Kweise Partition *P* (Partitionsvektor) von *G* zu finden so, dass die Summe der Gewichte der Kanten, die durch die Partition durchtrennt werden, herabgesetzt wird abhängig von der Begrenzung, $\forall i, l_i \leq c_i$. Wo *c* ein Vektor von Größe *m*, so dass $\forall i, c_i \geq 1,0$ ist. Der Vektor *c* wird vom Benutzer spezifiziert und auf die Menge der Lastengleichheit reflektiert wird, die der Benutzer bereit ist, für jede Gewichtsposition anzunehmen.

2.3 Weitere Definitionen und Notationen

Gegeben ist eine Menge *A* von Objekten, so das $\forall x \in A$ einen Gewichtsvektor w^x der Größe *m* hat. Wir definieren $w^A_i = \sum_{\forall x \in A} w^x_i$.
P ist der Partitionsvektor vom Graphen *G =(V, E)*. Ein Knoten *v*, der benachbart ist mit einem Knoten der sich in einer anderen Partition befindet, nennt sich **boundary Knoten** und befindet sich in einer **boundary Partition**.

{**Neighbourhood** $N(v) \mid \forall v \in V \wedge v \in$ boundary Partition, $N(v) = \bigcup_{u \in Adj(v) \wedge P[u] \neq P[v]} P[u]$}

Für jeden Knoten v berechnen wir die Kosten (gains) der Bewegung in eine benachbarte Partition. So berechnen wir für jedes b ∈ N(v) $ED[v]_b$ als die Summe der Gewichte der Knoten (v, u), so dass P[u] = b ist. Anschließend berechnen wir ID[v], als die Summe der Knoten (v, u), so dass P[u]=P[v] ist. $ED[v]_b$ nennt sich **external degree** von v zu der Partition b, und ID[v] nennt sich **internal degree** von v. Gegeben der oben genannten Definitionen sind die Kosten (gains) der Bewegung des Knoten v in die benachbarte Partition b gleich $ED[v]_b - ID[v]$, oder für b ∈ N(v) ist $g[v]_b = ED[v]_b - ID[v]$.

Lemma 2: Betrachten wir eine Menge *S* von *n* Objekten. Lassen Sie (w^i_1, w^i_2) zwei Gewichte sein, die mit jedem Gegenstand i, so dass $w^S_1 = 1$ und $w^S_2 = 1$ dazugehörig sind. Wir können diese Gegenstände in zwei Teilmengen A und B verteilen, so dass $| w^A_1 - w^B_1 | \leq 2\grave{\imath}$ und $| w^A_2 - w^B_2| \leq 2\grave{\imath}$ wo $\grave{\imath} = max(w^i_j \mid i = 1, 2,...,n, und\ j = 1,2)$.

Dieses Lemma wird in der Ausgangsverteilung (initial Partition) benutzt. Dieses Lemma beweißt, dass ein Satz zweigewichtiger Objekte in zwei zerlegten Teilmengen verteilt werden kann so, dass der Unterschied zwischen irgendeinem der Gewichte, der zwei Sätze, durch zweimal das Höchstgewicht jedes möglichen Objekts begrenzt wird. Sie zeigen weiter, dass diese Grenze zu den m-Gewichten generalisiert werden kann. Der sequentielle multigewichtiger Partitionsalgorithmus dargestellt in [2], benutzt auch dieses Lemma. Dieser Algorithmus ist die Grundlage für die verteilende Ausgangsphase. In der Mehrebenenoptimierungsphase funktioniert ein gieriger Kweisenoptimierungsalgorithmus wie folgt. Sie werden bis auf ihrer angrenzenden Unterbereichen überprüft und verschoben, wenn diese Bewegung:

(a) Die Qualität der Partition verbessert, ohne die spezifizierten Abgleichungsanforderungen für jede Begrenzung zu verletzen.
(b) Die Abgleichung der mehrfachen Gewichte verbessert, ohne die Fachqualität zu verschlechtern.

Eine kleine Anzahl von solchen Wegen wird durch die Knoten durchgeführt an jedem mehrmals aufeinander optimierten Graphen.

3. Parallele multigewichtige Graphenpartition

3.1 Coarsening Phase

Die coarsening Phase generiert kleinere Graphen $G_i = (V_i, E_i)$ von den Anfangsgraphen $G_0 = (V_0, E_0)$, so dass $|V_i| > |V_{i+1}|$. Der Graph G_{i+1} wird aus G_i hergestellt, indem man ein maximales zusammenpassendes Matching $M_i \subseteq E_i$ von G_i findet und die Knoten einschmelzt, die sich mit diesen Kanten verbinden. Knoten die nicht mit diesen Kanten verbinden werden einfach in den G_{i+1} kopiert.

Wenn sich die Knoten $v, u \in V_i$ verschmelzen, um den Knoten $w \in V_{i+1}$ zu formen, bekommt w einen Gewichtsvektor, der die Summe der Gewichtsvektoren v und u beinhaltet, ebenfalls werden die Kanten die sich mit w verbinden die Union der Kanten sein, die mit v oder u verbindet sind minus die Kante (v, u). Für jedes Kantenpaar (x, v) und (x, u) wird eine Kante (x, w) kreiert, die als Kantengewicht die Summe der Kantengewichte von den Kantenpaaren hat. Deshalb werden in der Zwischenzeit der aufeinanderfolgenden Verkleinerungsebenen, die Gewichte der Knoten und der Kanten erhöht (Abbildung 3).

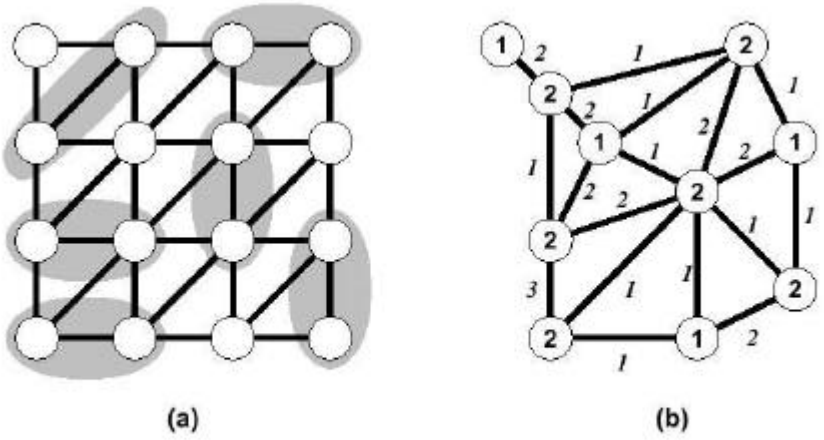


Figure 3: The process of finding a maximal matching and contracting the graph to obtain the next level coarser graph. Note that the vertices and edges of the coarse graph have weights to reflect the number of vertices and edges that are collapsed together.

Die Methode die für das Matching benutzt wird, beeinflusst sehr die Qualität der Partition und die Zeit die man für die Optimierungsphase braucht. Man nennt sie ***heavy-edge matching*** (HEM), und berechnet ein Matching M_i, so dass die Gewichte der Kanten in M_i groß sind. Das HEM wird durch einen Zufallsalgorithmus berechnet. Die Knoten werden durch Zufall besucht. Der zufällige ausgesuchte Knoten wird mit dem Knoten zusammengebracht, der mit den schwersten Kanten verbunden ist. Die coarsening Phase beendet, wenn der verkleinerte Graph G_m eine kleine Anzahl von Knoten hat.

In der dieser Phase, kollaborieren alle Prozessoren, um ein matching von Knoten parallel zu berechnen. In diesen Entwurf, berechnet jeder Prozessor ein Matching von Knoten die ihm gehören (Parallele Partition übernehmen die Tatsache, dass ein Graph auf die Prozessoren so verteilt ist, dass sie die gleiche Anzahl von Knoten und die dazugehörigen Nachbarschaftsinformationen haben). Die Herausforderung, wenn sie dieses leistungsfähig durchführt, behebt Absichten für Knoten, die unterschiedlichen Prozessoren gehören. Zum

Beispiel, will ein Knoten von einem Prozessor mit einen anderen Knoten zusammen gebracht werden, der einem zweiten Prozessor gehört. Aber, der Prozessor dem der zweite Knoten gehört, möchte diesen Knoten mit einem anderen Knoten zusammen bringen. Im allgemeinen Fall, benötigen die Lösungsberechnungen solcher Konflikte eine all-to-all Kommunikation zwischen den Prozessoren. Der existierende Entwürf unterteilt die Berechnungen, so dass solche Konflikte nicht zu Stande kommen [3]. In diesen Algorithmus wird eine Kolorierung von jedem Verkleinerten Graphen in jeder Ebene berechnet. Zur Kolorierung wird Luby's Algorithmus benutzt [4].

Die **heavy-edge** heuristische Funktion ist eine robuste Methode die Bevorzugungen von Kanten mit schweren Gewichten liefert, während ein maximales unabhängiges Set berechnet wird. Die **heavy-edge** heuristische Funktion neigt dazu, eine große Menge der dargestellten Kantengewichten in den aufeinanderfolgenden verkleinerten Graphen zu entfernen, und folglich eine hohe Qualitätsinitialenspaltung findet, die wenige Optimierungsbewegungen während der (uncoarsening) Optimierungsphase erfordert.

Im Kontext der multigewichtigen Partition, ist diese Eigenschaft der **heavy-edge** heuristischer Funktion, gleichmäßig anwendbar, und ist für das Konstruieren der aufeinanderfolgenden coarsed Graphen nützlich. Man kann den (coarsening) Verkleinerungsprozess auch verwenden, um zu versuchen, die Schwierigkeit des ausgleichenden Problems der Last in sich selbst zu verringern wegen des Vorhandenseins der mehrfachen Gewichte. Im allgemeinen ist es einfacher, eine ausgeglichene Partition zu berechnen, wenn die Werte der unterschiedlichen Elemente jedes Gewichtvektors nicht erheblich unterschiedlich sind. Wenn alle gewichte des Vektors gleich sind, dann ist die Partition gleich einer Partition eines Graphen mit einem Gewicht. Deshalb sollte man während des Coarsenings (wann immer möglich) Knotenpaare zu verschmelzen, die nicht große Unterschiede an Ihren Gewichten haben.

Ein solcher einfacher Algorithmus für das Berechnen eines maximalen unabhängigen Satzes besucht nach dem Zufall die Knoten. Wenn ein Knoten nicht noch zusammengebracht worden ist, bringt er ihn mit einem seiner angrenzenden nicht angepassten Knoten zusammen, der den Gewichtunterschied des entfernten Knotens herabsetzt. Es gibt viele mögliche Wege, die Gleichförmigkeit eines Gewichtvektors festzustellen. Eine Möglichkeit soll den Unterschied zwischen dem maximalen und dem minimalen Gewicht im Vektor verwenden, um den Mangel an Gleichförmigkeit in den Gewichten darzustellen. In diesem Fall werden Gewichtvektoren deren Unterschied (normalisiert in bezug auf die Summe der Gewichte des Vektors) kleiner sind bevorzugt. Eine wechselnde Approximation soll den Unterschied betrachten in bezug auf das durchschnittliche Gewicht des Vektors. Das heißt, nach Knoten v können wir die Quantität

$$Ó_{i=1}^{m} | w_i^v - 1/mÓ_{j=1}^{m} w_j^v |,$$

betrachten und Knoten bevorzugen für den die Gleichung kleiner ist. Dieser neuer Entwurf funktioniert ziemlich häufig besser als der ehemalige Entwurf durchführt, der versucht hat, den Unterschied zwischen dem kleinsten und größten Gewicht herabzusetzen.

Im wesentlichen haben wir zwei heuristische Funktionen, die wir verwenden können, um die maximalen unabhängigen Sätze zu berechnen. Die erste, (d.h., **heavy-edge**) zum Produzieren der aufeinanderfolgenden verkleinerten (coarsed) Graphen übersetzt, die das herausgestellte Kantengewicht herabsetzen; und folglich, führend zu besseren Qualität der Partition. Während die Zweite, (d.h., **balance-edge**) zum Produzieren der aufeinanderfolgenden coarsed Graphen übersetzt wird, die den Unterschied bezüglich der Gewichte innerhalb jedes Gewichtvektors

herabsetzen und um einfacher, ein Ausgangsverteilen zu berechnen, dass die ausgleichenden Begrenzungen erfühlen. Hauptsächlich kann man auch diese zwei Entwürfe kombinieren, indem man eins von beiden als primär Objektive verwendet und der andere als sekundär Objektive.

Anwendung nach der Kolorierung

Der Graph G_{i+1} wird von G_i kreiert, indem ein Matching von G_i gefunden wird und dann die Knoten kollabiert werden, die mit den Kanten von M_i verbunden sind. Da das Matching M_i ein unabhängiger Satz von Kanten ist, können wir einen parallelen Kolorierungsalgorithmus an einen linearen Graphen von G_i anwenden, um ein globales Matching parallel zu berechnen. Da ein Matching mit diesen Algorithmus sehr kostenspielig ist, wird das Matching basierend auf den Farbton des Graphen.

Verwenden wir einen Graphen der mit der Farbe *c* gefärbt wurde und der an jeden Konten eine Variable *Match* hat, die anfangs den Wert –1 hat. Am Ende der Phase wird *Match* den Knoten speichern mit dem er gematch wurde. Wenn ein Knoten *v* nicht gematch wurde hat *Match* den Wert *v* [3]. In der c^{th} Wiederholung, haben die Konten, die noch nicht gematch sind, die Farbe *c* und wählen die einen ungematchten Nachbarn mit HE gematcht, und anschließend aktualisieren sich die *Match* Variablen mit den dazugehörigen Knoten Nummer. Wenn *u* ein Knoten mit Farbe *c* ist und *(u, v)* die Kante die von *u* ausgewählt wurde. Da aber v nicht die Farbe *c* hat, wird dieser Knoten nicht bei diesem Zyklus benutzt. Aber es gibt eine Wahrscheinlichkeit, dass ein anderer Knoten *w* mit der Kante *(u, w)* und die Farbe *c* ausgewählt wird. Nachdem alle Knoten mit der Farbe *c* ungematchte Knoten ausgewählt haben, werden sie synchronisiert. Ihre *Match* Variable gelesen. Wenn *Match* gleich Ihrer Knotennummer ist, dann wird dieser gematcht (Match(nachbar) = Wähler).

Der oben genannte Algorithmus wird ziemlich leicht auf einem verteilten distributed memory parallel computer wie folgt eingeführt. Die *writes Match* Variablen werden zusammen gefügt und als ein message zu allen korespontiehrenden Prozessoren verschickt. Wenn ein Prozessor Verbindungsrequest empfängt für den gleichen Knoten, wird der ausgewählt, der mit der schwersten Kante verbunden ist. Während des Lesevorganges, stellen die Prozessoren, die die *Match* Variablen besitzen fest, ob sie diejenigen sind, die den verschmelzten Knoten in G_{i+1} speichern. Dieses wird getan, indem man eine gleichmäßige verteilte Zufallsvariable verwendet. Der Knoten wird mit der gleichen Wahrscheinlichkeit weggehalten oder gegeben. Die Experimente in [3] haben gezeigt, dass diese einfache heuristische zu einer sehr guten Partition führt.

Nachdem ein Matching M_i berechnet ist, weiß jeder Prozessor, wie viele Knoten (und die dazugehörigen Angrenzenlisten) er senden muß und wie viele er empfangen muß. Jeder Prozessor sendet und empfängt diese Subgraphen, und bildet den folgenden waagerecht ausgerichteten verkleinerten Graph, indem er die Angrenzenslisten der verschmolzenen Knoten vermischt. Der coarsening Prozess endet, wenn der Graph O(p) Knoten hat.

3.2 Paralleles Ausgangsverteilen (Parallel Initial Partitioning)

In der Ausgangsverteilungsphase, ist eine Partition von einem verkleinerten (coarsest) Graphen schnell berechnet. Dies kann durch einen Aufgabeaufspaltungsentwurf [3] durchgeführt werden. Hier wird der verkleinerte (coarsest) Graph von einem einzelnen Prozessor zusammengesetzt und anschließend zu den anderen Prozessoren verschickt. Danach

berechnet jeder Prozessor die selbe Spaltung von diesem Graphen gleichzeitig. Wenn k gleich 2 ist, dann hört die Ausgangsverteilung auf. Anderseits, wird die Spaltung benutzt, um zwei Untergraphen zu konstruieren, wo jeder Untergraph die Knoten von nur einem Gebiet der Spaltung beinhaltet. Auf dieser Art und Weise wird der Graphen jedes Mall retrospektiv in zwei Teile gespaltet. Zunächst berechnet die Hälfte der Prozessoren eine Bisection des ersten Subgraphen und die andere Hälfte berechnet eine Bisection des zweiten Subgraphen. Dieses rekursive Aufspalten fährt fort, bis ein Kweisenverteilen berechnet ist.

Die zweite Phase eines Partitionsalgorithmus hat als Ziel, ein Kweisenverteilen der verkleinerten Graphen $G_k=(V_k, E_k)$ zu berechnen so, dass jede Partition ungefähr mit $|V_0|/k$ Knoten des ursprünglichen Graphen enthält. Da während des Coarsenings, die Gewichte der Knoten und der Kanten des verkleinerten Graphen eingestellt wurden, um die Gewichte der Knoten und der Kanten des feineren Graphen zu reflektieren, enthält G_m genügende Informationen, um die ausgeglichene Partition intelligent zu erzwingen.

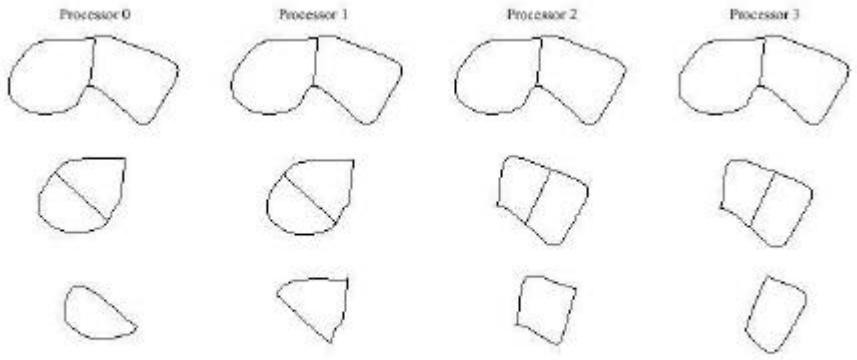


Figure 4: Performing the initial *k*-way partitioning in parallel. Each processor explores only a single path from the root to the leaves in the recursive bisection tree.

Während der Partitionsphase wird ein Pweisenverteilen des Graphen mit einem rekursiven Bisectionalgorithmus berechnet. Da der kleinste Graph nur O(p) Knoten hat, kann dieser Schritt sequentiell durchgeführt werden, ohne die Leistung des gesamten Algorithmus erheblich zu beeinflussen. Dennoch wird in diesem Algorithmus auch diese Phase parallelisiert, indem wir eine rekursive Spaltung verwenden. Dieses wird wie folgt getan: Die verschiedenen Stücke des kleinen Graphen werden zu allen Prozessoren mit einem all-zu-all gesendet. An diesem Punkt führen die Prozessoren rekursive Spaltungen durch. Jedoch, wie in Abbildung 4 veranschaulicht, erforscht jeder Prozessor nur einen einzelnen Weg des rekursiven Bisectionbaums. Am Ende speichert jeder Prozessor die Knoten, die seinem Gebiet des Pweisenverteilens entsprechen.

Lassen Sie G=(V, E) der Graph, den wir in zwei Subgraphen $G_A=(V_A, E_A)$ und $G_B=(V_B, E_B)$ halbieren möchten. In diesem Entwurf wählen wir zuerst einen Knoten $v \in V$ nach dem Zufall vor und stellen $V_A = \{v\}$ und $V_B = V/V_A$ und setzen dann alle Knoten $u \in V_B$ in eine Maximum-Prioritätswarteschlange entsprechend ihrer Gewinnfunktion ein (Abschnitt 2.3). Dann wählen wir wiederholend den oberen Knoten u von der Prioritätswarteschlange vor, verschieben ihn auf G_A und aktualisieren die Prioritätswarteschlange, um die neuen gains der Nachbarsknoten von u zu berechnen. Der Algorithmus beendet, sobald das Gewicht der Knoten in G_B mehr werden, als Hälfte des Gewichts der Knoten in G. Es kann gezeigt

werden, dass der maximale Unterschied bezüglich der Gewichte der zwei Teile durch zweimal das Gewicht des schwersten Knotens begrenzt wird. Außerdem ist die Qualität der Partition ziemlich gut (Lemma 2).

Anstatt, eine einzelne Prioritätswarteschlange zu benutzen, benutzen man m unterschiedliche Warteschlangen, in denen m die Zahl Gewichten ist. Ein Knoten gehört nur einer einzelnen Prioritätswarteschlange abhängig von dem relativen Reihe der Gewichte in seinem Gewichtvektor. Insbesondere ein Knoten v mit Gewichtvektor (w^v_1, w^v_2, ..., w^v_m) gehört der jth Warteschlange wenn $w^v_j=max_i(w^v_i)$. Das Bestehen von dieser mehrfachen Priorität steht auch Änderungen an, wie die Knoten von G_A auf G_B vorgewählt und verschoben werden. Zu jeder möglicher gegebenen Zeit abhängig von dem relativen Reihe der Gewichte des Graphen G_B, verschiebt der Algorithmus den Knoten von der Oberseite einer spezifischen Prioritätswarteschlange. Insbesondere wenn $w^{VB}_j=max_i(w^{VB}_i)$, dann die jth Warteschlange vorgewählt wird. Wenn diese Warteschlange leer ist, dann wird die nicht leere Warteschlange, die dem folgenden schwereren Gewicht entspricht, vorgewählt. Der Algorithmus beendet, sobald eins der Gewichte von G_A mehr als Hälfte des entsprechenden Gewichts von G werden.

3.3 Uncoarsening Phase oder Optimierungsphase

Die Optimierungsphase nimmt den partitionierten Graphen und in jeden rekursiven Schritt werden eingeschmelzte Knoten mit der KL heuristischen Funktion expandiert.

```
SelectMoveTarget(v, N(v))
{
   N' = {b | b ∈ N(v) and ED[v]_b − ID[v] ≥ 0}
   N'' = {b | b ∈ N' and IsBalanceOK(v, P[v], b)}

   if (|N''| = 0) then return -1

   b = N''[1];
   for (i = 2; i ≤ |N''|; i = i + 1) {
      j = N''[i]
      if (ED[v]_j > ED_b[v]) then b = j
      elif (ED[v]_j = ED_b[v] and IsBalanceBetterTT(v, P[v], b, j)) then b = j
   }

   if (ED[v]_b-ID[v] > 0 or (ED[v]_b-ID[v] = 0 and IsBalanceBetterFT(v, P[v], b))) then return b
   else return -1;
}
```

IsBalanceOK(u, a, b, c)

Diese Funktion Berechnet ob die Bewegung des Knotens u von a nach b die Balance c nicht beeinträchtigt.

IsBalanceFT(u, a, b, c)

Diese Funktion berechnet, ob die Bewegung des Knotens u von a nach b zu einer besseren Partition führt, anstatt den Knoten nicht zu bewegen. Es wird zuerst das l^a berechnet das noch den Knoten u beinhaltet und anschließend das l^a nach der Bewegung von u nach b. Aus diesen Vektoren werden dann zwei d^a und d^b berechnet.

$$d_i^a = \frac{l_i^a - 1}{c_i - 1} \quad and \quad d_i^b = \frac{l_i^b - 1}{c_i - 1}$$

Wenn $||d^a||_1 \leq ||d^b||_1$ dann führt die Bewegung des u nach b zu einer besseren Partition.

IsBalanceTT(u, a, b_1, b_2, c)

Diese Funktion berechnet, ob die Bewegung des Knotens u von a nach b_1 zu einer besseren Partition führt, anstatt den Knoten von a nach b_2 zu bewegen. Es wird zuerst das l^a berechnet nach der Bewegung von u nach b_1 und anschließend das l^b nach der Bewegung von u nach b_2. Aus diesen Vektoren werden dann zwei d^a und d^b berechnet.

$$d_i^a = \frac{l_i^a - 1}{c_i - 1} \quad and \quad d_i^b = \frac{l_i^b - 1}{c_i - 1}$$

Wenn $||d^a||_1 \leq ||d^b||_1$ dann führt die Bewegung des u nach b zu einer besseren Partition.

Parallele Multigewichtige (Multi-constaint) Multiebenen Optimierung.

Die hauptsächliche Herausforderung in der Entwicklung einer parallelen Multigewichtigen Graphenpartition ist es zu beweisen oder zu prüfen, ob ein paralleler Multiebenen Optimierungsalgorithmus entwickelt worden ist, der folgenden Kriterien voraußetzen.

1. Er muss die Abgleichung aller Gewichte (Multiparameter) garantieren.
2. Er muss die Optimierungsbewegungen maximieren.
3. Er muss skalierbar sein.

Um zu garantieren, dass Fachabgleichung während der parallelen Optimierung beibehalten wird, ist es notwendig globale Unterbereichgewichte nach jeder Knotenmigration zu aktualisieren. Solch ein Entwurf ist viel zu sequentiell um leistungsfähig durchgeführt zu werden. Für diesen Grund, lassen einzelngewichtige Partitionsalgorithmen eine Anzahl von Knotenbewegungen ausführen, bevor ein Aktualisierungsschritt durchgeführt wird. Eine der Implikationen der synchronen Optimierungsbewegungen ist, dass die Abgleichungsbegrenzung während der Optimierungswiederholungen verletzt werden kann. Dieses ist weil, wenn ein Unterbereich eine bestimmte Menge zusätzliches Knotengewicht halten kann, ohne die Abgleichungsbegrenzung zu verletzen, dann nehmen alle Prozessoren an, dass sie den ganzen Extraraum, für das Durchführen von Optimierungsbewegungen benutzen können. Selbstverständlich, wenn gerade zwei Prozessoren die Menge der zusätzlichen Knoten verschieben, die ein Unterbereich in sie halten kann, dann wird der Unterbereich Überladen. Dieses wird durch das Beispiel in Abbildung 5 veranschaulicht. Die Abbildung 5(a) zeigt einen Einzelnbegrenzungs- Zweiwegverteilen eines Graphen an, dass an zwei Prozessoren durchgeführt wird. Jeder der Knoten hat ein einzelnes Gewicht, das mit ihm dazugehörig ist, dass in der Abbildung gezeigt wird. Die Unterbereichgewichte sind 25 und 25. Folglich ist das Verteilen tadellos ausgeglichen. Lassen Sie die Ungleichheitstoleranz 20% sein. Folglich können Unterbereiche vom Gewicht 30 oder kleiner sein. Die Abbildung 5(b) zeigt den Zustand der Partition, nachdem die parallele Optimierung durchgeführt worden ist. Hier verschob der gelbe Prozessor einen Knoten (eingekreist) von Gewicht eins von links auf den rechten Unterbereich und einen Knoten (eingekreist) von Gewicht sechs von rechts auf den linken Unterbereich. Der gelbe Prozessor nimmt an, dass diese Bewegungen Unterbereichgewichte von 30 ergeben (nach links) und 20 (rechts), und also wird die

Abgleichungsbegrenzung beibehalten. Gleichzeitig die blauen Prozessorbewegungen ein Knoten (eingekreist) von Gewicht eins von links zum rechten Unterbereich und ein Knoten (eingekreist) von Gewicht sechs vom Recht zum linken Unterbereich. Der blaue Prozessor glaubt auch, dass die Abgleichungsbegrenzung beibehalten wird. Zusammen jedoch, ergeben diese Bewegungen das der linke Unterbereich, dass mit einem Gewicht von 35 Überladen wird und dass der rechte Unterbereich, das mit einem Gewicht von 15, untergewichtig wird.

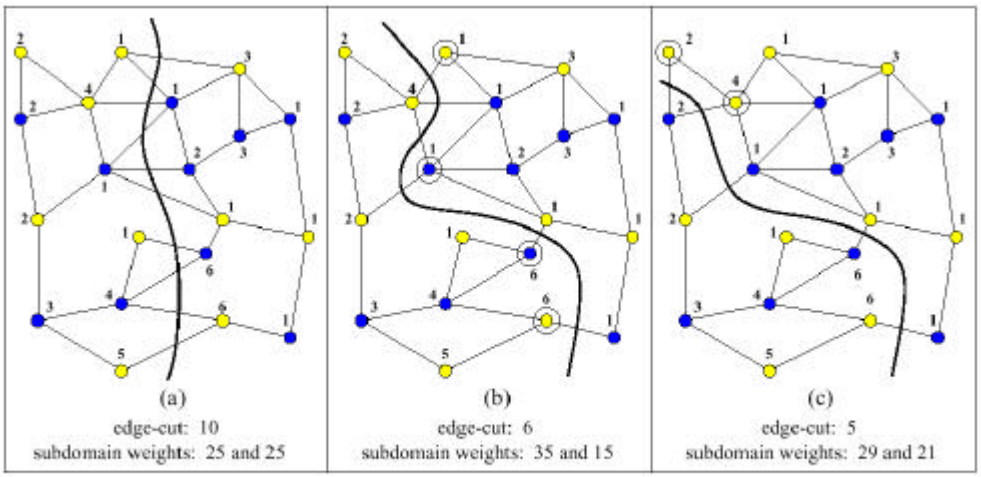


Figure 5: This figure shows a single constraint partitioning (a). During parallel refinement, the blue and yellow processors each move two vertices (b). This results in an imbalance in the partitioning. If we simply disallow vertices to move into the overweight domain, balance can be obtained easily (c).

Parallele Einzelngewichtige Graphenpartitioners adressieren diese Herausforderung durch Anregung folgender Optimierung, um die Abgleichung der Partition beim Optimieren seiner Qualität Widerherzuerstellen. Zum Beispiel, ist es häufig Ausreichend einfach weitere Knotenbewegungen in Überladene Unterbereiche abzulehnen und eine andere Wiederholung der Optimierung durchzuführen. Wenn man es in dieser Situation so macht, dann werden die Partition wieder ausgeglichen (wie in Abbildung 5(c) veranschaulicht). Im allgemeinen kann der Optimierungsprozess möglicherweise nicht immer das Verteilen beim Verbessern seiner Qualität auf diese Art auszugleichen (obgleich Erfahrung gezeigt hat, dass diese normalerweise ziemlich gut arbeitet). In diesem Fall können einige zunehmende Ränderbeschneidungen vorgenommen werden, um Knoten aus den überladenen Unterbereichen heraus zu verschieben.

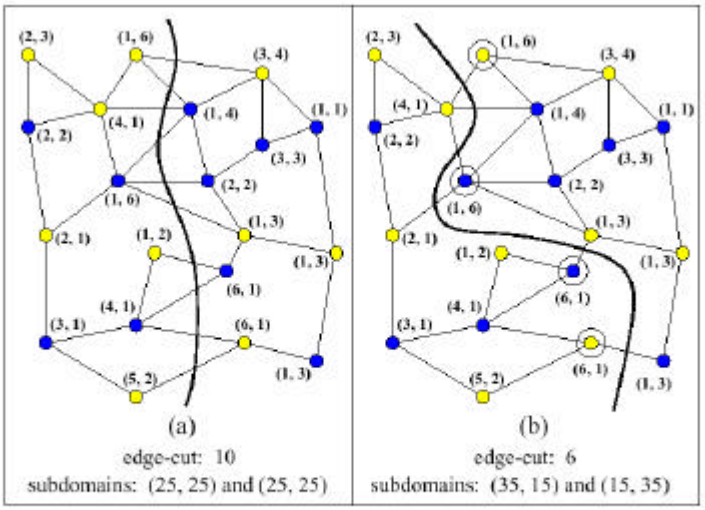


Figure 6: This figure shows a two-constraint partitioning (a). During parallel refinement, the blue and yellow processors each move two vertices (b). This results in an imbalance that is not easily corrected. Moving the same two vertices from Figure 5(c) will result in the second constraint becoming even more imbalanced.

Die reale Herausforderung ist, wenn dieses Phänomen im Kontext der mehrfachen Abgleichung Begrenzungen betrachten. Dieses ist, weil, sobald ein Unterbereich für eine gegebene Begrenzung Überladen wird, kann es sehr schwierig sein, die Ausgleichung der Partition wieder herzustellen. Die Abbildung 6 veranschaulicht diesen Punkt. Sie zeigt den Graphen der Abbildung 5, aber jetzt hat jeder Knoten zwei Gewichte. In Abbildung 6(a), sind die Unterbereichgewichte (25, 25) und (25, 25) und also ist die Partition tadellos ausgeglichen. Wieder wird angenommen, dass der Benutzer eine Ungleichheitstoleranz von 20% für beide Gewichte spezifiziert. Die Abbildung 6(b) zeigt den Zustand der Partition, nachdem die gleichen Optimierungsmaßnahmen getroffen sind, wie in der Abbildung 5(b). Dieses mal können die Knotenbewegung, in die Überladenen Unterbereiche, nicht verboten werden. Anderseits wird mit der Optimierung wie Normal fortgefahren. Dieses ist, weil beide Unterbereiche hier Überladen sind, das linke Unterbereich ist Überladen in bezug auf das erste Gewicht, und das rechte Unterbereich ist in bezug auf das zweite Gewicht Überladen. Hier, zum Beispiel, wird das Vertauschen der zwei Knoten, die in der Abbildung 5(c) vertauscht worden sind, nicht arbeiteten. Obwohl dieses die Ungleichheit in bezug auf die erste Begrenzung (Gewicht) verringert, erhöht es die Ungleichheit in bezug auf die zweite Begrenzung (Gewicht). Tatsächlich was angefordert wird, ist eine kompliziertere Approximation, in der die Knoten, die im ersten Gewicht größer sind als die vom zweiten, von links auf den rechten Unterbereich verschoben werden und die Knoten, die im zweiten Gewicht als das erste größer sind, werden von dem rechten auf den linken Unterbereich verschoben. Während, dass Ausgleichen einer zweigewichtigen Partition auf diese Art schwierig ist, wird das Problem, des Ausgleichens einer Multigewichtigen Partition schwieriger analog mit der Anzahl der Gewichte. Die Schwierigkeit der ausgleichenden Multigewichtigen Partition gegeben, soll eine bessere Lösung eine Situation vermeiden, in der das Partitionieren gleichgewichtgestört wird. Folglich möchte man einen Multigewichtigen Optimierungsalgorithmus entwickeln, der helfen kann, sicherzugehen, dass die Abgleichung während der parallelen Optimierung beibehält.

Ein Weg zum Sicherstellen, dass die Abgleichung während der parallelen Optimierung beibehalten wird, ist es die Menge des Extraknotengewichts, die ein Unterbereich anhalten kann, ohne Gleichgewichtsstörungen zu bekommen, durch die Zahl Prozessoren zu teilen.

Dieses bekommt dann das maximale Knotengewicht, dass es jedem einzelnen Prozessor erlaubt, in einem einzelnen Durchlauf, Knoten zwischen Unterbereichen zu tauschen. Betrachten Sie das Beispiel veranschaulicht in der Abbildung 7. Dieses zeigt die Unterbereichgewichte an, für eine 4-wege, 3-gewichtige Partition. Es wird angenommen, dass die Benutzertoleranz 5% ist. Die schattierten Barren repräsentieren die Unterbereichgewichte für jede der drei Gewichten (Länge des Gewichtsvektors). Die weißen Barren stellen die Menge des Gewichts dar, die, wenn sie dem Unterbereich hinzugefügt würden, das Gesamtgewicht zu 5% über den Durchschnitt wachsen würde. Das heißt mit anderen Worten, dass die weißen Barren die Menge des *Extraplatzes* für jeden Unterbereich anzeigen, dass eine Toleranztungleichheit von 5% hat. Die Abbildung 7 zeigt, wie der Extraplatz in Unterbereich A für die vier Prozessoren oben aufgespaltet werden kann. Wenn jeder Prozessor auf das Verschieben der angezeigten Mengen des Gewichts in Unterbereich A begrenzt wird, ist es nicht möglich für die Toleranz von 5% überschreitet wird.

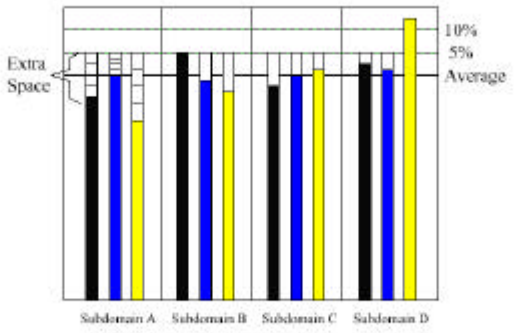


Figure 7: This figure shows the subdomain weights for a 4-way partitioning of a 3-constraint graph. The white bars represent the extra space in a subdomain for each weight given a 5% user specified load imbalance tolerance.

Während diese Methode garantiert, dass kein Unterbereich (dass es nicht von Start auf überladen ist) über der Ungleichheitstoleranz Überladen wird, ist es im großen Grade einschränkend. Dieses ist, weil im allgemeinen nicht alle Prozessoren herauf ihren zugeordneten Platz verwenden müssen, während andere mehr Knotengewichte in einem Unterbereich als durch ihre Hälfte lassen können. Im Kontext der einzelnen waagerecht ausgerichteten Optimierungsentwürfe (d.h., coarsened jene Entwürfe, in denen Verfeinerung nur am Inputgraphen und nicht an unterschiedlichem durchgeführt wird, Versionen des Diagramms), kann dieses erzielt werden, indem man eine zusätzliche Anzahl von Durchläufen durch die Knoten erlaubt. In diesem Fall wird ein lokales Minimum von Rand-Schnitten schließlich erreicht. Jedoch im Kontext der Mehrebenenoptimierung, kann eine Anzahl von Optimierungsbewegungen auf den coarsed Graphen nie ungefähr kommen (und also kann ein lokales Minimum möglicherweise nicht auf diesen Graphen erreicht werden). Dieses ist, weil die Granularität der Knoten hier eine Anzahl von möglichen Bewegungen verbieten kann (wenn die Gewichte von Knoten den Anteil übersteigen, die zu den Prozessoren zugeteilt werden). In diesem Fall sogar erlauben zusätzliche Optimierungsiterationen nicht, dass diese Maßnahmen getroffen werden. Da das Partitionieren zu den folgenden feineren Graphen projiziert wird, erlaubt die abnehmende Granularität der Knoten größere Bewegungsfreiheit und also wird dieser Effekt verringert. Jedoch ist das Resultat, dass die Zahl von dem Verringern der Optimierungsbewegungen den Ränderschneidens, die an den groben Graphen durchgeführt wurden, wird verringert verglichen mit der Begrenzung der sequentiellen Multimehrebenenoptimierungsalgorithmen (die nicht einen Entwurf benötigen, der herauf den Extraplatz eines Unterbereichs schneidet) und so, gibt dieser parallele Entwurf Resultate, die niedrigere Qualitätspartition als sequentielle Entwürfe liefern. Außerdem wie die Anzahlen

entweder der Prozessoren oder der Multigewichtigen (Vektorlänge der Gewichte) sich erhöht, erhöht sich auch dieser Effekt. Der Grund ist, dass, da die Anzahl der Prozessoren sich erhöht, die Stücke, die jedem Prozessor zugeordnet werden, kleiner werden. Da die Länge des Vektors (Anzahl der Gewichte pro Knoten) sich erhöht, wird jede zusätzliche Begrenzung auch geschnitten. Dies heißt, dass jeder Knoten, der für eine Bewegung vorgeschlagen wird, angefordert wird, um die Scheiben von allen Begrenzungen zu passen. Zum Beispiel betrachten Sie eine Dreibegrenzung 10-Wege Partition berechnet auf 10 Prozessoren. Wenn Unterbereich A 20 Maßeinheiten des ersten Gewichts, 30 Maßeinheiten des zweiten Gewichts und 10 Maßeinheiten des dritten Gewichts anhalten kann, dann müssen alle Prozessor sichergehen, dass die Summe der Gewichtsvektoren von allen Knoten, die sie in Unterbereich A verschiebt, kleiner als ist (2, 3, 1).

Es ist möglich, den Extraplatz der Unterbereiche intelligenter zuzuordnen als jedem Prozessor einen gleichen Anteil zu geben. Wir haben Entwürfe nachgeforscht, die die Belegungen bilden, die auf einer Anzahl von Faktoren basieren, wie den potentiellen Ränderschneiden Verbesserungen der Randknoten von einem spezifischen Prozessor zu einem spezifischen Unterbereich, die Gewichte von diesem Randknoten, und die Gesamtzahl Randknoten auf jedem Prozessor. Obwohl diese Entwürfe eine größere Anzahl der Optimierungsbewegungen, auf den coarsed Graphen gewähren, als der direkte Entwurf, werden immer noch die Bewegungen eingeschränkt als der sequentielle Algorithmus.

Der parallele Multigewichtige Optimierungsalgorithmus wurde so entwickelt, dass der nicht einschränkender als der sequentielle Algorithmus in bezug auf die Zahl der Optimierungsbewegungen ist, die er auf jedem waagerecht ausgerichteten Graphen erlaubt, während auch helfen kann, sicherzugehen, dass keine der Begrenzungen übermäßig unausgeglichen werden. Im Mehrebenenkontext ist dieser Algorithmus gerades so wirkungsvoll, wenn er die Qualität der Partition wie der sequentielle Algorithmus verbessert.

Dieser Algorithmus (im wesentlichen ein Reservierungsentwurf) führt einen zusätzlichen Durchlauf durch die Knoten auf jeder Optimierungswiederholung durch. Im ersten Durchlauf werden Optimierungsmaßnahmen gleichzeitig (als Normal) getroffen, jedoch aktualisieren wir nur temporäre Datenstrukturen. Zunächst führt man eine globale Verkleinerungsoperation durch, um festzustellen, ob oder nicht die Abgleichungsbegrenzungen verletzt werden. Wenn keine der Abgleichungsbegrenzungen verletzt werden, stuft sich diese Bewegungen als Normal ein. Andernfalls ist jeder Prozessor angefordert einen Teil seiner vorgeschlagenen Knotenbewegungen in jene Unterbereiche zu missbilligen, die Überladen sein würden, wenn alle Bewegungen erlaubt würden. Man stellt fest, wie viele Bewegungen in möglicherweise Überladene Unterbereiche wie folgt missbilligt werden sollten. Der Prozentsatz des Überschüssigen Knotengewichts, dass in diese Unterbereiche verschoben wird, wird berechnet, und jeder Prozessor wird fordern diesen Prozentsatz des Gewichts seiner vorgeschlagenen Bewegungen in das Unterbereich zu missbilligen. Die spezifischen verbotenen Bewegungen werden nach dem Zufall durch jeden Prozessor vorgewählt. Obwohl das Vorwählen von Bewegungen nach dem Zufall negativ Auswirkungen auf den Kantenschnitt haben kann, ist das nicht ein Problem, weil weitere Optimierungen die Effekte aller schlechten Auswallen, die geschehen um zu werden, leicht beheben kann. Außer diesen Änderungen ist unser Multigewichtige Optimierungsalgorithmus dem Grobkorn eingewichtigen Optimierungsalgorithmus ähnlich, der beschrieben wird in [1].

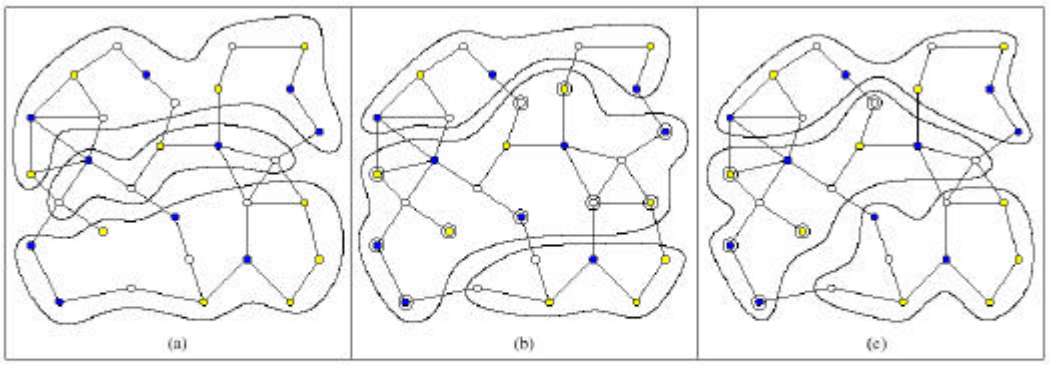


Figure 8: This figure gives an example of a our parallel refinement algorithm for a three-way single constraint partitioning. Here the imbalance tolerance is set to 10%. In figure (a) the top and bottom subdomains are overweight. In figure (b) each processor concurrently proposes a number of refinement moves. If all of these moves are committed, the middle subdomain will become overweight. In figure (c) 50% of the proposed moves on each processor are disallowed. The result is that the partitioning is balanced to within the tolerance.

Abbildung 8 veranschaulicht diesen Prozess im Kontext einer Eingewichtigen. Dieses ist ein Beispiel einer Dreiwegepartition auf drei Prozessoren. Jeder Knoten hat (einzelnes) Gewicht von Eins. Es gibt 30 Knoten in diesem Graphen, also sollte jeder Unterbereich ein Idealgewicht von 10 haben. Jedoch stellen wir eine Toleranztungleichheit von 10% fest, so dass Unterbereiche mit Gewicht von 11 annehmbar sind. Abbildung 8(a) gibt die Partition vor dem Anfang unseres Optimierungsalgorithmus. Hier sind die Unterbereichgewichte (von oben nach unten) 12, 6 und 12. Folglich ist der oberseitige und der unterseitige Unterbereich durch 20% Überladen, während der mittlere Unterbereich untergewichtig ist. Abbildung 8(b) zeigt die Partition, nach dem ersten Durchlauf unseres Optimierungsalgorithmus. Die eingekreisten Knoten stellen vorgeschlagene Bewegungen in den mittleren Unterbereich dar. Wenn alle diese Bewegungen festgelegt werden, sind die Unterbereichgewichte 8, 16 und 6. Da dieser Ungleichheit den mittleren Unterbereich über der Toleranz von 10% bringen wurde, müssen wir eine Anzahl von diesen Bewegungen missbilligen. Spezifisch wird der mittlere Unterbereich ursprünglich bei 6 belastet und kann bis 11 halten, während innerhalb der Ungleichheitstoleranz noch fallen kann. Folglich können wir fünf Knoten in dieses verschieben. Jedoch gibt es 10 vorgeschlagene Knotenbewegungen in diesen Unterbereich. Wir teilen diese zwei Zahlen und finden, dass jeder Prozessor die Hälfte seiner Bewegungen in den mittleren Unterbereich missbilligt. So müssen die blauen und gelben Prozessoren jeder zwei der vier vorgeschlagenen Bewegungen verbitten. Der weiße Prozessor missbilligt eine seiner zwei vorgeschlagenen Bewegungen. Abbildung 8(c) zeigt die Partition, nachdem dieses getan worden ist. Jetzt können die Bewegungen der eingekreisten Knoten sicher festgelegt werden.

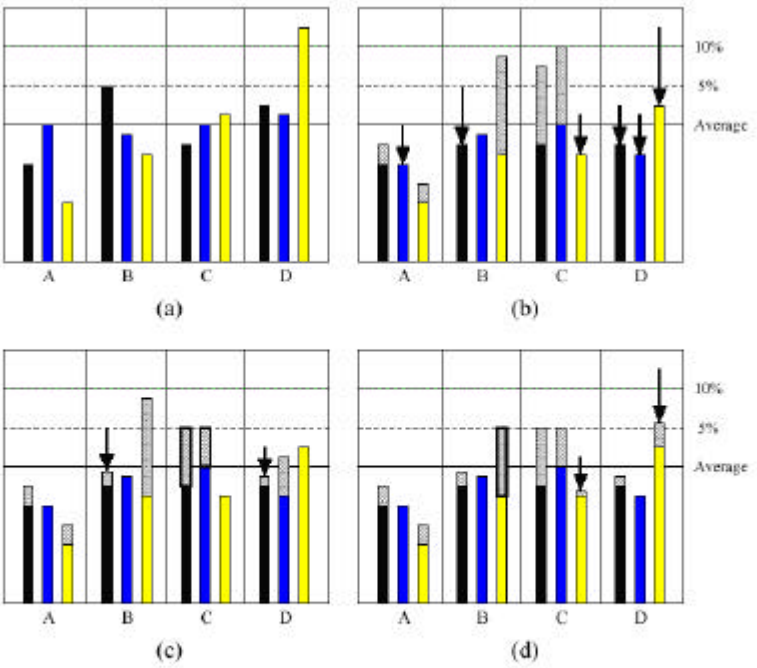


Figure 9: This figure shows the state of a partitioning prior to refinement (a), the proposed state of the partitioning after the first pass of our algorithm (b), the proposed state after moves are disallowed in order to maintain the balance of the first constraint (c), and the proposed state after moves are disallowed in order to maintain the balance of the third constraint (d).

Dieses Beispiel zeigt den parallelen Optimierungsalgorithmus für nur eine Eingewichtige. Zusätzliche Ausgaben ergeben den Kontext der Multigewichtigen. Dieses ist, weil (i) die Knoten nicht vom Maßeinheit Gewicht sind, und (ii) müssen wir die Knotenbewegungen festlegen oder missbilligen, die auf Multigewichten basieren. Wenn Knoten nicht vom Maßeinheit Gewicht sind, dürfen wir nicht einfach eine spezifische Anzahl von Knotenbewegungen missbilligen, aber müssen anstatt irgendeine Zahl von Bewegungen zu missbilligen so, dass die Summe der Gewicht von diesen die angeforderten Prozente (des Gesamtgewichts der vorgeschlagenen Bewegungen) spezifiziert durch den globalen Verkleinerungsbetrieb. Wir sprechen den zweiten Punkt an, indem wir der Reihe nach um jedes Gewicht des Knoten kümmern. Abbildung 9 veranschaulicht diesen Prozess. Abbildung 9(a) ist eine Erweiterung des Beispiels in Abbildung 7. Abbildung 9(b) gibt den Zustand nach dem ersten Durchlauf unseres Optimierungsalgorithmus. Die untenzeigende Zeiger zeigen die Menge des aus dem an Unterbereich heraus verschoben zu werden Knotengewichts, während die angewählten Checkboxen zeigen die Menge des Knotengewichts an, die in den Unterbereich verschoben wird. In diesem Beispiel, ist die Ungleichheitstoleranz 5%. Wenn die vorgeschlagenen Bewegungen festlegen, ist Unterbereich B in bezug auf das dritte Gewicht der Vektoren Überladen und der Unterbereich C ist in bezug auf das erste und zweite Gewicht der Vektoren Überladen. Folglich ist es notwendig einige dieser vorgeschlagenen Bewegungen zu missbilligen. Wir beginnen mit den ersten Gewicht. Wir müssen eine Anzahl von Bewegungen in Unterbereich C nicht erlauben so, dass das Gesamtgewicht von diesen die Ungleichheit (des ersten Gewichts) zu unten bis 5% bewegt. In Abbildung 9(c), ist dieses getan worden. Hier missbilligen die Unterbereiche B und D eine Anzahl von Bewegungen in den Unterbereich C. Beim Beheben der Ungleichheit des ersten Gewichts müssen genügende missbilligte Bewegungen von Unterbereich D bis Unterbereich C waren, damit die zweite Begrenzung auch behoben wird. Dieses war unbeabsichtigt. Jedoch ist es dadurch vorteilhaft, dass wir direkt auf das dritte Gewicht jetzt an umziehen können. (Wenn die Überschüssige

Ungleichheit des zweiten Gewichts nicht völlig beseitigt worden war, würde es notwendig gewesen die Bewegung von mehr Knoten in Unterbereich C zu verbitten.) Um die Abgleichung des dritten Gewichts sicherzustellen, ist es notwendig die Übertragung der Gewichte in Unterbereich B zu verbitten. In der Abbildung 9(d), wurde dieses getan. Die Bewegung von Knoten von den Unterbereichen C und D werden missbilligt. In diesem Fall ergibt das Missbilligen der Bewegung der Knoten von Unterbereich D zu Unterbereich B das Unterbereich D, das durch eine bescheidene Menge in bezug auf das dritte Gewicht Überladen ist. Nun da wir uns um alle Gewichte gekümmert haben, sind die restlichen Bewegungen festgelegt.

Es ist wichtig, zu merken, dass der oben genannte Entwurf noch nicht garantiert, dass die Abgleichungsgewichte beibehalten werden. Dieses ist, weil, wenn eine Anzahl von Knotenbewegungen verboten wird, die Gewichte der Unterbereiche, von denen diese Knoten stammen, werden größer als die Gewichte, die mit der globalen Verkleinerungsoperation berechnet worden waren. Es ist folglich möglich, dass einige dieser Unterbereiche überladen werden. Um diese Situation zu beheben, kann eine zweite globale Verkleinerungsoperation durchgeführt werden, gefolgt von einem anderen Umlauf in dem eine Anzahl von den (restlichen) vorgeschlagenen Knotenbewegungen verboten wird. Diese Korrekturen konnten zu andere Ungleichheiten führen, die diesen Prozess erfordern zu wiederholen, bis sie zusammenläuft (oder, bis alle vorgeschlagenen Bewegungen verboten worden sind). Wir haben in unserer Implementierung gewählt dieses Problem einfach ignorieren. Dieses ist, weil die Anzahl der missbilligten Bewegungen ein sehr kleiner Bruch der Gesamtzahl der Knotenbewegungen ist, und so jede mögliche Ungleichheit, die durch sie hervorgebracht wird, ziemlich bescheiden ist. Unsere experimentellen Resultate zeigen, dass die Menge der Ungleichheit eingeführt auf diese Art genug klein ist, dass weitere Optimierungen in der Lage sind, es zu beheben. Tatsächlich so lang, wie die Menge die eingeführte Ungleichheit Korrekturbar ist, kann solch ein Entwurf die hochwertigeren Partitionen möglicherweise ergeben, die mit den Entwürfen verglichen werden, die nur den durchführbaren Lösungsraum erforschen, wie in Abschnitt 4. besprochen.

3.4 Komplexität

Die Komplexitätsanalyse eines parallelen eingewichtigen Graphen Mehrebenenspartition wird innen dargestellt in [3]. Der beschriebene Algorithmus benutz 4 Unteralgorithmen (Kolorierung, Ausgangsverteilen , Matching und Optimierung). Diese haben ähnliche Anforderungen in Berechnung und Kommunikation. Die genaue Berechnung und die Kommunikationsanforderungen dieser Phasen hängen von dem Graphen ab. Insbesondere hängen die Berechnungsanforderungen vom durchschnittlichen Grad des Graphen, während die Kommunikationsanforderung vom durchschnittlichen Grad abhängt, sowie die chromatische Zahl des Graphen ab. Ein Graph mit einer höheren chromatischen Zahl erfordert mehr Wiederholungen und globale Synchronisierung während der coarsening und uncoarsening Phase, als der Zahl inneren Wiederholungen, die in jedem solchen Schritt durchgeführt werden, und zur Anzahl der Farben des Graphen proportional ist. Im Rest dieses Abschnitts, analysieren wir die parallele Komplexität und das scalability dieses Algorithmus unter den folgenden zwei Annahmen: (i) hat jeder Knoten im Graph einen kleinen begrenzten Grad; und (ii) wird diese Eigenschaft auch durch die aufeinanderfolgenden kleineren Graphen erfüllt.

Diese Annahmen erlauben uns, die folgenden Beobachtungen und die Vereinfachungen zu bilden:

(i) Die chromatische Zahl Graphen auf allen coarsening Ebenen ist eine kleine Konstante.
(ii) Die Größe der mehrmals hintereinander verkleinerten Graphen durch einen Faktor von 1+e, in dem $0< e \leq 1$ sich verringert.
(iii) Wir können die Zahl der Kanten von der Analyse ignorieren, da sie vom gleichen Auftrag wie die Zahl der Knoten sind.

Die Menge der Berechnung, durchgeführt durch jeden dieser vier Algorithmen ist zur Größe des Graphen proportional, das Lokal auf jedem Prozessor gespeichert wird. Da die Größe der mehrmals hintereinander verkleinerten Graphen durch einen Faktor von $1 + e$ für $0 < e \leq 1$ sich verringern, beherrscht die Berechnung, die für den ursprünglichen Graphen durchgeführt wird, die Berechnung, die für die folgende kleineren Graphen O(log n) durchgeführt wird. So ist die Menge der gesamten Berechnung durchgeführt $T_{calc} = O(n/p)$.

Die Menge der Kommunikation durchgeführt durch jeden dieser vier Algorithmen hängt von der Zahl Schnittstellen Knoten ab. Z.B. während des Farbtons, muss jeder Prozessor die gelegentlichen Zahlen kennen, die mit den Knoten neben den am Ort gespeicherten Knoten dazugehörig sind. Ähnlich während der Optimierung, jedes Mal wenn ein Knoten verschoben wird, müssen die angrenzenden Knoten mitgeteilt werden, um ihre externen Grad zu aktualisieren. Jeder Prozessor speichert n/p Knoten und nd/p-Kanten, in denen d der durchschnittliche Grad des Graphen ist. So ist die Zahl der Schnittstellen Knoten höchstens O(n/p). Da die Knoten zuerst nach dem Zufall verteilt werden, werden diese Schnittstellen Knoten gleichmäßig unter den p-Prozessoren verteilt. Folglich muss jeder Prozessor Daten mit $O(n/p^2)$ Knoten jedes Prozessors austauschen. Wechselweise muss jeder Prozessor Informationen für ungefähr $O(n/p^2)$ Lokal gespeicherten Prozessor der Knoten senden. Dieses kann vollendet werden, indem man den all-zu-all personifizierten Kommunikationsbetrieb verwendet, dessen Komplexität O(n/p)+O(p) ist. Die Kommunikationskomplexität über allen coarsening Ebenen O(log n), ist $T_{comm} = O(n/p) + O(plogn)$, da die Größe des Graphen mehrmals hintereinander halbiert wird. Damit die Gleichung gültig ist , müssen die Daten gleichmässig auf die Prozessoren verteilt sein.

So von T_{calc} und T_{comm} haben wir, dass die parallele Durchlaufzeit des Partitionsalgorithmus

$$T_{par} = O\left(\frac{n}{p}\right) + O(p \log n) \qquad (2)$$

und die isoefficiency Funktion ist $O(p^2 \log p)$. Die parallele Durchlaufzeit unseres multigewichtigen Graphenpartition ist ähnlich (die zwei Annahmen gegeben). Jedoch während der coarsening Phase und der Mehrebenenoptimierung, müssen alle *m* Gewichte betrachtet werden. Folglich beträgt die parallele Durchlaufzeit des multigewichtigen Algorithmus *m* Zeiten länger oder

$$T_{par} = O\left(\frac{nm}{p}\right) + O(pm \log n). \qquad (3)$$

Da die sequentielle Komplexität des sequentiellen multigewichtigen Algorithmus $O(nm)$ ist, ist die isoefficiency Funktion der multigewichtigen Partition auch $O(p^2 \log p)$.

4. Experimentelle Resultate

In diesem Kapitel werden experimentelle Resultate des Partitionsalgorithmus auf 32, 64 und 128 Prozessoren von einem Cray T3E dargestellt. Es wurden zwei Sets von Testproblemen konstruiert, um die Wirksamkeit des Partitionsalgorithmus im berechnen von schnellen Partitionen in hoher Qualität und Ausgeglichenheit zuzeigen. Beide Sets von Probleme wurden synthetisch von den vier Graphen festgelegt, die in Tabelle 1 beschrieben wurden.

Der Zweck des ersten Sets von Problemen ist es, die Fähigkeit des multigewichtigen Partitionsalgorithmus zu prüfen, für einige verhältnismäßig harte Probleme. Von jedem der vier eingegebenen Graphen beziehungsweise legte man Graphen mit zwei, drei, vier und fünf Gewichten fest.

Der Zweck des zweiten Sets von Problemen ist, die Leistung der multigewichtigen Partition im Kontext der Mehrphasenberechnungen zu prüfen, an denen unterschiedliche (vielleicht Überlappende) Teilmengen Knoten an den unterschiedlichen Phasen teilnehmen. Für jedes der vier Graphen, legten wir wieder Graphen mit zwei, drei, vier und fünf Gewichten fest beziehungsweise, die zwei -, drei -, vier - und Fünfphasen Berechnungen entsprechen. Im Fall vom Fünfphase Graphen, ist der Teil des Graphen, das aktiv ist (d.h. Berechnungsdurchführend), 100%, 75%, 50%, 50% und 25% der Unterbereiche. Im Vierphasenfall ist dieses 100%, 75%, 50% und 50%. In den drei und die Zweiphasen ist es 100%, 75% und 50% und 100% und 75%.

Graph	Num of Verts	Num of Edges
mrng1	257,000	1,010,096
mrng2	1,017,253	4,031,428
mrng3	4,039,160	16,033,696
mrng4	7,533,224	29,982,560

Table 1: Characteristics of the various graphs used in the experiments.

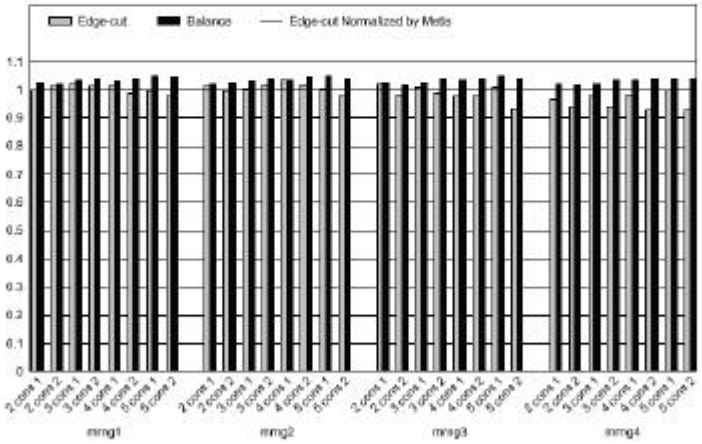


Figure 10: This figure shows the edge-cut and balance results from the parallel multi-constraint algorithm on 32 processors. The edge-cut results are normalized by the results obtained from the serial multi-constraint graph partitioning algorithm implemented in MeTiS.

Vergleich des sequentiellen- und des parallelen Multigewichtigen Algorithmen. Abbildungen 10, 11 und 12 vergleichen die Qualität der parallelen Partition, mit der Qualität

der sequentiellen Partition und geben die Maximallaastungleichheit der Partition an. Jede Abbildung zeigt vier Sets von Resultaten, eins für jedes der vier Graphen, die in Tabelle 1 beschrieben werden. Jedes Set besteht aus zwei -, drei -, vier - und Fünfgewichtige Typ 1 und 2 Probleme. Diese werden „*m cons t*“ beschriftet, wo *m* die Anzahl der Gewichte anzeigt und *t* die Art es Problems anzeigt (d.h., Typ 1 oder 2).

Abbildung 10, 11 und 12 zeigen, dass der parallele Partitionsalgorithmus in der Lage ist, Partitionen der ähnlichen oder besseren Qualität als das sequentiellen multigewichtigen Partitionsalgorithmus zu berechnen.

Der parallele Algorithmus kann manchmal, hochwertigere Partitionen als der sequentielle Algorithmus produzieren. Es gibt zwei Gründe für dieses. Zuerst ist der parallele Matching Entwurf nicht so wirkungsvoll wie der sequentielle. Das Resultat ist, dass der eben berechnete coarsened Graph dazu neigt, für den parallelen Algorithmus größer zu sein als für den sequentiellen Algorithmus. Dieses veranlasst den parallelen Algorithmus, mehr coarsening Stufen zu nehmen, um ein genug kleinen Graphen zu erhalten.

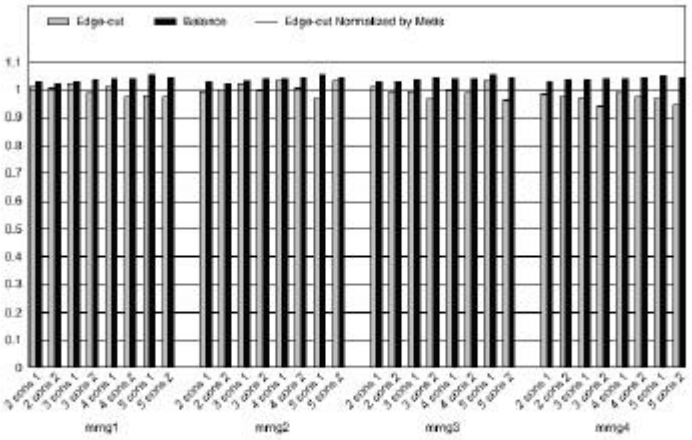


Figure 11: This figure shows the edge-cut and balance results from the parallel multi-constraint algorithm on 64 processors. The edge-cut results are normalized by the results obtained from the serial multi-constraint graph partitioning algorithm implemented in METIS.

Der zweite Grund liegt im sequentiellen Algorithmus. Sobald die Partition ausgeglichen wird, erforscht es nie den unpraktischen Lösungsraum, um die Fachqualität zu verbessern. Dieses ist eine schlechte Strategie, wenn eine Partition die berechnet wird so Unausgeglichen wird, dass die folgende Optimierungen nicht in der Lage sind, die Abgleichung wieder herzustellen. Jedoch ist der parallele Optimierungsalgorithmus in der Lage, die Fachabgleichung wieder herzustellen, wenn weiter, seine Qualität verbessernd.

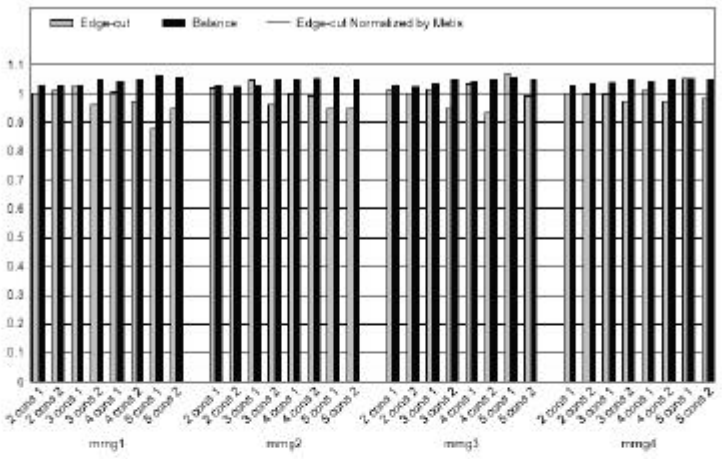


Figure 12: This figure shows the edge-cut and balance results from the parallel multi-constraint algorithm on 128 processors. The edge-cut results are normalized by the results obtained from the serial multi-constraint graph partitioning algorithm implemented in the MeTiS library.

k	serial time	parallel time
2	7.3	6.4
4	7.5	4.4
8	8.0	2.5
16	8.3	1.7

Table 2: Serial and parallel run times of the multi-constraint graph partitioner for a three-constraint problem on *mrng1*.

Zeit-Resultate. Tabelle 2 vergleicht die Laufzeiten des Partitionsalgorithmus des parallelen multigewichtigen Graphen mit dem sequentiellen multigewichtigen Algorithmus, der in der MeTiS Bibliothek eingeführt wird für *mrng1*. Diese Resultate zeigen nur bescheidene speedups für die parallele Partition. Der Grund ist, dass der Graph *mrng1* ziemlich klein ist, und so ist die Kommunikation und parallele overheads bedeutend. Jedoch verwenden wir *mrng1*, weil es das Einzige der Testgraphen ist, dass genug klein ist, auf einen einzelnen Prozessor des Cray T3E sequentiell laufen zu lassen.

Tabelle 3 gibt vorgewählte Zeitresultate und Leistungsfähigkeiten unserer parallelen multigewichtigen Graphen Partition des Algorithmus auf bis 128 Prozessoren graphisch dar. Tabelle 3 zeigt, dass der Algorithmus sehr schnell ist, da es in der Lage ist, eine Dreigewichtige 128-way Partition eines 7.5million Knoten Graphen in ungefähr 7 Sekunden auf 128-processors von einem Cray T3E zu berechnen. Sie zeigt auch, dass er ähnliche Laufzeiten erhält, während man, die Größe des Problems und die Zahl Prozessoren verdoppeln (oder vervierfachen). Z.B. ist die Zeit, die angefordert wird, um *mrng2* (mit 1 Million Knoten) auf acht Prozessoren zu verteilen ähnlich mit der Partition von (4 Million) Knoten *mrng3* auf 32 Prozessoren und *mrng4* (7,5 Million Knoten) auf 64 Prozessoren.

Tabelle 4 gibt die Abläufe des Kweise Partitionsalgorithmus des eingewichtigen Graphen, der in der ParMeTiS Bibliothek eingeführt wird auf den gleichen Graphen, die für unsere Experimente benutzt werden. Das Vergleichen von Tabellen 3 und Tabelle 4 zeigt an, dass das eine dreigewichtige Partitionsberechnung ungefähr doppelt solang braucht, wie die Partition eines Eingewichtigen. Z.B. dauert eine dreigewichtige Partition 9,3 Sekunden und 4,8 Sekunden eine eingewichtige Partition, die für *mrng3* auf 32 Prozessoren verteilt ist. Auch

das Vergleichen der speedups zeigt an, dass der multigewichtige Algorithmus etwas mehr Skalierbar ist als der eingewichtige Algorithmus. Z.B. ist das speedup von 16 bis 128 Prozessoren für *mrng3* 3,84 für den multigewichtigen Algorithmus und 3,26 für den eingewichtigen Algorithmus. Der Grund ist, dass der multigewichtige Algorithmus rechnerisch intensiver ist, als der eingewichtige Algorithmus ist, da mehrfache (nicht einzelne) Gewichte regelmäßig berechnet werden müssen.

Graph	8-processors		16-processors		32-processors		64-processors		128-processors	
	time	efficiency	time	efficiency	time	efficiency	time	efficiency	time	efficiency
mrng2	9.8	100%	5.3	92%	3.5	70%	2.5	49%	3.1	20%
mrng3	31.8	100%	16.9	94%	9.3	85%	5.7	70%	4.4	45%
mrng4	out of memory		30.7	100%	16.7	92%	9.2	83%	6.4	60%

Table 3: Parallel run times and efficiencies of our multi-constraint graph partitioner on three-constraint type 1 problems.

Graph	8-processors	16-processors	32-processors	64-processors	128-processors
mrng2	5.4	3.1	2.1	1.5	1.7
mrng3	15.8	8.8	4.8	3.0	2.7
mrng4	38.6	16.2	8.8	5.0	3.6

Table 4: Parallel run times of the single constraint graph partitioner implemented in PARMETIS.

Parallele Leistungsfähigkeit. Tabelle 3 gibt ausgewählte parallele Leistungsfähigkeiten des Partitionsalgorithmus bis auf 128 Prozessoren. Die Leistungsfähigkeiten werden in bezug auf die kleinste Zahl den gezeigten Prozessoren berechnet. Folglich für *mrng2* und *mrng3*, stellten wir die Leistungsfähigkeit von acht Prozessoren bis 100% ein, während wir die Leistungsfähigkeit von 16 Prozessoren bis 100% für *mrng4* einstellten. Die parallele multigewichtige Graphenpartition erhielt Leistungsfähigkeiten zwischen 20% und 94%. Die Leistungsfähigkeiten waren gut (zwischen 90% - 70%) als der Graph in bezug auf die Zahl der Prozessoren genug groß war. Jedoch ließen diese Operationen für die kleineren Graphen auf großer Zahl der Prozessoren fallen. Die isoefficiency der parallelen multigewichtigen Graphenpartition ist $O(p^2 \log p)$. Um eine konstante Leistungsfähigkeit beizubehalten, wenn wir die Zahl Prozessoren verdoppeln, muss die Größe der Daten mehr als viermal erhöhen. Da *mrng3* ungefähr viermal großer ist, als mrng2 können wir die isoefficiency Funktion experimentell prüfen. Die Leistungsfähigkeit der multigewichtigen Partition mit 32 Prozessoren für *mrng2* ist 70%. Die Verdoppelung der Zahl der Prozessoren bis 64 und die Erhöhung der Datengröße bis zum viermal (64-processors auf *mrng3*) erbringt eine ähnliche Leistungsfähigkeit. Diese ist besser als erwartet, da die isoefficiency Funktion voraussagt, dass wir in der Zunahme der Größe der Datei bis zu mehr als viermal, die gleiche Leistungsfähigkeit zu erhalten werden. Wenn wir die Resultate von 64 Prozessoren auf *mrng2* Überprüfen und die von 128 Prozessoren auf *mrng3* sehen wir eine abnehmende Leistungsfähigkeit von 49% zu 45%. Dieses ist, was wir gegründet auf der isoefficiency Funktion erwarten würden. Wenn die Resultate von 16 Prozessoren auf *mrng2* überprüft werden und die von 32 Prozessoren auf *mrng3* sieht man, dass der Fall der Leistungsfähigkeit größer ist (92% zu 85%). So hier erhalten wir eine etwas falschere Leistungsfähigkeit als erwartet. Die geringfügigen Abweichungen können der Tatsache zugeschrieben werden, dass die Zahl Optimierungsiterationen auf jedem Graphen von oben begrenzt ist. Jedoch wenn ein lokales Minimum vor diesem oberen Limit erreicht wird, dann keine weiteren Iterationen an diesem Graphen durchgeführt werden. Folglich während das obere Limit auf der Menge der Arbeit durch den Algorithmus erledigt ist, gilt das dasselbe für alle Experimente, kann die

tatsächliche Menge der Arbeit erledigt werden, abhängig von dem Optimierungsprozess etwas unterschiedlich sein

5. Literatur

[1] G. Karypis and V. Kumar. A coarse-grain parallel multilevel k-way partitioning algorithm. In Proceedings of the 8th SIAM conference on Parallel Processing for Scientific Computing, 1997.

[2] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. Technical Report TR 99-019, Dept. of Computer Science, Univ. of Minnesota 1998.

[3] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. Siam Review, 41(2):278-300, 1999.

[4] G. Karypis and V. Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 1998 (to appear). Also available on WWW at URL http://www.cs.umn.edu/~karypis. A short version appears in Intl. Conf. on Parallel Processing 1995.

[5] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. The Bell System Technical Journal 49(2):291-307, 1970.