Bachelorarbeit

Wolfgang Scholz

26. Januar 2007

Inhaltsverzeichnis

1 Einführung

In dem Fachbereich, mit dem sich diese Arbeit beschäftigt, geht es um automatische Parallelisierung von Programmen. Die Untersuchung stützt sich auf das Polyedermodell, einem mathematischen Rahmenwerk für Schleifenparallelisierung [Len93], und auf *LooPo* [Loo], dem Prototyp eines Compilers zur Parallelisierung des Lehrstuhls für Programmierung an der Universität Passau.

Durch die fortschreitende Entwicklung von Werkzeugen zur Parallelisierung hat sich deren Anwendungsbereich vergrößert. Eine größere Anzahl von Problemen kann mit Speedup parallel ausgeführt werden. Weitere Verbesserungen erhofft man sich von Parallelisierungstechniken, die mit einem heterogenen Rechnernetz auskommen, weil solche Netze wesentlich leichter zur Verfügung gestellt werden können.

Grids haben den Vorteil, dass die einzelnen Knoten nicht aufeinander abgestimmt sein müssen. Nahezu beliebige Rechenmaschinen können verwendet werden. Das auszuführende Programm trägt Sorge darüber, dass die Ausführung korrekt abgewickelt wird und das möglichst mit Speedup.

Ein herkömmliches *h-transformiertes* Programm bietet keine Funktionalität, um die beschriebenen Probleme zu umgehen. Die Idee ist nun, zusätzliche Strukturen einzuführen, die die Programmausführung an die wechselnden Umgebungsbedingungen sich anpassen machen. Ziel dieser Arbeit ist es, einen Teil dieser Stukturen zu spezifizieren und weitere vorzubereiten, um den Einbau in den Compiler LooPo [Loo] zu erleichtern.

1.1 Gliederung

Kapitel 2 stellt Grundlagen für die Parallelisierung von Programmen dar, die Ausgang für die später beschriebenen Erweiterungen bilden.

Kapitel 3 beschreibt einen Implementierungsvorschlag, der das Polyedermodell den Erfordernissen des Grids anpasst.

Kapitel 4 setzt den den Vorschlag mit anderen Ideen in Beziehung, welche zum Zeitpunkt dieser Arbeit im wissenschaftlichen Umfeld diskutiert werden.

Kapitel 5 evaluiert die Einsetzbarkeit anhand einer provisorischen Implementierung und einem einfachen Beispielprogramm.

2 Grundlagen

Dieses Kapitel erläutert einige Grundlagen des verwendeten Modells paralleler Programme. Die Arbeit stützt sich auf die Verfügbarkeit des parallelen Compilers [Loo], der benötigte Schritte der Programmtransformation durchführt, auf die hier nicht in aller Einzelheit eingegangen wird. Der geneigte Leser sei auf entsprechende Literatur verwiesen. Es werden nur solche Begrifflichkeiten aufgeführt, die im Lastausgleich Verwendung finden.

2.1 Programme und Transformationen

2.1.1 Kriterien für parallele Programme

Das Ziel einer parallelen Ausführung eines Programms ist es, die Programmlaufzeit zu verringern. Meist stellt sich das Problem so dar, dass eine bestimmte Hardwarekonfiguration zur Verfügung steht, die möglichst effizient ausgenutzt werden soll. Dazu müssen geeignete Teile des Problems parallelisiert und deren Ergebnisse zusammengeführt werden. Um dieser Anforderung nachzukommen, also ein sequentielles Programm in ein Paralleles umzuwandeln, ist Programmtransformation notwendig. Das Zielprogramm muss

- sowohl dem sequentiellen Programm gegenüber semantisch äquivalent sein,
- als auch auf der parallelen Maschine schneller ausgeführt werden können.

Zur semantischen Äquivalenz referieren wir auf Literatur, die die Semantik von sequentiellen und parallelen Programmen beschreibt.
TODO: Literaturreferenz: Programmsemantik (seq./par.)
Um dem Geschwindigkeitskriterium Rechnung zu tragen, ist es notwendig, Kostenmodelle für sequenzielle wie auch für parallele Programme einzuführen. In geeigneter Tiefe wird darauf in
TODO: Literaturreferenz: Kostenmodelle (seq./par.)
eingegangen.

Im Wesentlichen konzentriert sich die Programmtransformation auf verschachtelte Schleifen, da sich diese Programmteile am besten zur Parallelisierung eignen und in den meisten wissenschaftlichen Anwendungen einen Großteil der Rechenzeit ausmachen.

2.1.2 Notation Pseudo-Code

Um Programmteile darzustellen, verwenden wir eine von Pascal entliehene Notation für Pseudo-Code. Für den mit einschlägigen imperativen Programmiersprachen bewanderten Leser sollte diese Notation leicht eingängig sein.

```
f([integer], integer) -> [integer]
f(x, y) ->
begin
    for i:= 0 to length(x)-1 do
    begin
        x[i]:= x[i] * y;
    end;
    result:= x;
end;
```

Abbildung 1: Ein einfaches Programm

Sofern nicht anders angegeben, beschränken wir uns auf aufsteigende `for`-Schleifen mit einer Schrittweite von eins. Weiter werden auch keine Rekursionen verwendet. Funktionsaufrufe werden als atomar und seiteneffektfrei angenommen.

Um Typen von Funktionen zu deklarieren, verwenden wir eine Notation aus der funktionalen Umwelt. Typen in Mengenklammern bedeuten hierbei, dass es sich um eine Menge von Instanzen dieses Typs handelt. Analog stehen Typen in eckigen Klammern für Listen von Instanzen des Typs. Beispielhaft zeigt dies Abbildung 1.

2.1.3 Abhängigkeiten

Ein Programm kann aufgeteilt werden in die Menge der Operationen, die es enthält, und den Zustand. Operationen führen den Ausführungszustand in einen anderen über. Da die Menge der Operationen in einem sequentiellen Programm total geordnet ist, ist es aus einer naiven Sichtweise heraus beurteilt nicht möglich, Operationen umzuordnen, weil sich mit den Eingabezuständen auch die Programmausgabe ändern kann. Meist jedoch ändert eine Operation den Zustand nur so geringfügig, dass // TODO: Seidels Definition ist doch gut, warum soll ich die nicht einfach nehmen?

Operationen u und v
Sequenzielle Ausführungsreihenfolge $u\langle_{seq}v$
Abhängigkeitsrelation $\delta \subseteq \Omega \times \Omega$
$u\delta v$

2.1.4 Umordnung der Ausführungsreihenfolge

Sequenzielle Ausführungsreihenfolge Seidel S.9 $u <_{par} v$

2.2 Polyedermodell

Programm -> Modell -> Trafo. Modell -> Zielprogramm

2.3 Raum-Zeit-Mapping (space-time-mapping)

Das Raum-Zeit-Mapping transformiert den Indexraum (TODO: Indexraum einführen) des sequentiellen Quellprogramms in den Indexraum des Zielprogramms. In diesem Schritt wird in parallele Raum- und sequentielle Zeitdimensionen aufgeteilt. Vorhandene Datenabhängigkeiten werden dabei berücksichtigt.

Unter *virtueller* oder *logischer Zeit* versteht man die Projektion des transformierten Indexraums auf die Zeitdimensionen. Ein *virtueller* oder *logischer Prozessor* ist dementsprechend ein Punkt des transformierten Indexraums, projeziert auf die Raumdimensionen. (Seidel S.22)

TODO: Schedule? Allocation? Indextrafo? Zielcodegenerierung?

Zielcodegenerierung TODO: Überblicksartig erklären [GLW98]

2.4 Kommunikationen

(Abhängigkeitsgetrieben vs. Besitzergetrieben)
Warum abhängigkeitsgetriebene Kommunikation (Seidel S.29) hier besser ist

Erklären: global sequentiell und parallel und lokal sequentiell ausgeführte Schleifen

2.5 Tiling

Im Regelfall entsteht durch die Raum-Zeit-Transformation mehr Parallelität, als sinnvoll verwendet werden kann. Allerdings steigt im Allgemeinen mit der verwendeten Parallelität auch der Kommunikationsaufwand, wenn von verteiltem Speicher ausgegangen wird. Um diesem Problem zu entgehen, wird die Technik des *Tilings* angewendet, die zum Ziel hat, Kommunikation auf Kosten von Parallelität zu reduzieren. Die Namensgebung basiert auf dem Bestreben, der durch die Abhängigkeitsbeziehungen definierten Nachbarschaft der Punkte im transformierten Indexraum Rechnung zu tragen.

Dazu werden benachbarte Operationen zu Blöcken zusammengefasst und als eine Operation betrachtet. Abhängigkeiten innerhalb eines Blocks können somit vernachlässigt werden, da dort die sequentielle Ausführungsreihenfolge eingehalten wird. Im Programmcode bedeutet dies, dass äußere, eventuell parallele Schleifen aufgeteilt werden, was die parallelen Iterationen verringert und lokale

Iterationen erzeugt. So lassen sich bei geeigneter Wahl Kommunikationen verringern.

Die Bedienumgebung von LooPo [Loo] lässt zu, Tiling manuell einzustellen. Algorithmen, welche ein geeignetes Tiling automatisch berechnen, sind momentan Gegenstand der Forschung. Einige davon werden in [Sei04] vorgestellt.

Tiling kann nach außen hin transparent gestaltet werden. Deshalb wird im Weiteren von virtuellen Prozessoren gesprochen, obwohl es sich auch um Prozessortiles handeln kann. Dazu wird eine transparente Bibliothek zum Tiling vorgeschlagen. (TODO: Dieser Absatz ist doof)

3 Dynamischer Lastausgleich

In diesem Kapitel wird ein Protokollprototyp für den auf empirischen Informationen gestützten dynamischen Lastausgleich in parallelen Programmen diskutiert. Dieser adressiert speziell Erfordernisse, die im Bereich des *Grid-Computing* auftreten. Daher wird zuerst auf einige Eigenschaften von Grids eingegangen. Über grundlegende Lösungsideen einiger Probleme wird schließlich eine Implementierung eines solchen Protokolls vorgeschlagen.

Im Folgenden wird die dargestellte Herangehensweise *Lastausgleichsprotokoll* genannt, auch wenn dieser Ausdruck nicht charakterisierend ist.

3.1 Grid-Computing

3.1.1 Einführung

Das *Grid* ist eine Sammelbezeichnung für heterogene Rechensysteme in einem organisationsübergreifenden Netzwerk, wie etwa dem Internet. Das Verwenden vom Grid zum Lösen komplexer Probleme wird als *Grid Computing* bezeichnet. Das Problem wird dabei in Teilstücke zerlegt, die von den einzelnen *Knoten* parallel abgearbeitet werden können.

Bei den einzelnen Knoten im Grid handelt es sich entweder um einzelne handelsübliche Computer oder aber selbst wieder um komplexe parallele Architekturen.

Im Unterschied zu Hochleistungsrechnern oder Clustern ist im Grid der Zugriff auf einzelne Knoten nur limitiert möglich. In der Regel bestehen bis auf eine schlanke Abstraktionsschicht keine Gemeinsamkeiten unter den Knoten.

3.1.2 Chancen

Der Vorteil des Grid-Computing besteht darin, eine große Menge an Rechenzeit zur Verfügung zu haben, ohne die dafür notwendigen Hardware-Beschaffungskosten oder Systemadministration bereitstellen zu müssen. Die geringen Ansprüche des Grids an seine Knoten bringt eine hohe Verfügbarkeit mit sich. Einen großen Anteil der Knoten in Grids stellen Computer in privaten Haushalten, die per Pauschaltarif an das Internet angeschlossen und nur wenig ausgelastet sind.

3.1.3 Probleme

Anders als bei dedizierten Rechenanlagen sind die Knoten im Grid nicht aufeinander speziell angepasst. Leistung und Ausstattung sind in der Regel unterschiedlich, möglicherweise sogar im Vorfeld nicht ermittelbar, da Angaben nur einen Teil der Konfiguration spezifizieren können oder aber nicht den Tatsachen entsprechen. Die Verbindung der Knoten über das Internet verfügt über im Vergleich zu Hochleistungsnetzen geringer Bandbreite und einer langen Latenz. Sie

ist störanfällig und nicht als verlässlich einzustufen. Dies wirkt sich nachteilig auf die Transferzeit von Nachrichten aus.

Die Wahrscheinlichkeit, dass ein Knoten im Grid ausfällt, ist höher als bei professionell gewarteten Systemen. Falls es sich um Maschinen handelt, die auch zu anderen Zwecken eingesetzt werden, muss mit einer gewissen Fluktuation gerechnet werden. Ein Privatanwender mag seinen PC von der Rechenaufgabe abziehen wollen, da er ihn selbst benötigt. Dafür kann es wiederum vorkommen, dass im Laufe der Berechnung zusätzliche Knoten zur Verfügung stehen.

Ein weiterer Unterschied zu dedizierten Systemen besteht dahingehend, dass Knoten im Grid zum Teil für mehrere Aufgaben simultan verwendet werden. Die sich dadurch ändernde Last erschwert eine vorhergehende Einschätzung der zur Verfügung stehenden Rechenkapazität. Im Falle einer unerwarteten Überlast werden abhängige Berechnungen auf anderen Knoten verzögert, was im schlechtesten Fall zu einer Verzögerung der gesamten Berechnung führen kann (siehe Abbildung 2).

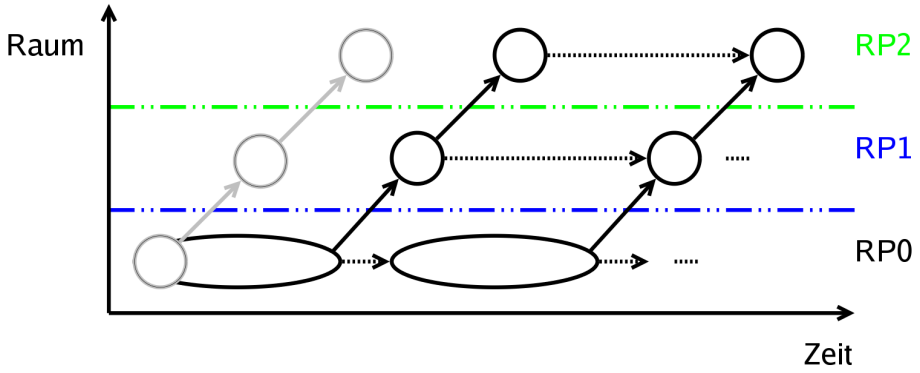

Abbildung 2: Durch die Überlast eines Prozessors werden davon abhängige Berechnungen verzögert.

Die Gründe, warum ein Knoten im Grid ausfallen kann, sind:

- Knoten werden unerwartet abgeschaltet.
- Knoten besitzen keine Sicherungsmaßnahmen gegen Stromausfall.
- Die Systeminstallation einiger Knoten ist nicht ausreichend stabil.
- Die Internetanbindung wird unterbrochen

Das Risiko, dass ein Knoten ausfällt, sollte daher im Grid nicht vernachlässigt werden.

3.2 Lösungsansatz: dynamische Lastumverteilung

3.2.1 Definitionen

Definition: Umgebungsbedingung Eine Eigenschaft des zum Programmlauf herangezogenen Rechensystems, die sich auf die Ausführung auswirkt.

Einige wichtige Umgebungsbedingungen sind:

- verfügbare Rechenzeit der Knoten
- Transferzeit für Nachrichten von und zu den Knoten
- Datenkapazität der Anbindung der Knoten (Grundlagen - muss noch definiert werden)
- Anzahl der Knoten
- ...

Um in einem parallelen Programm optimale Auslastung zu erzielen, werden Umgebungsbedingungen ermittelt und daraus die optimale Lastverteilung abgeleitet. Bei statischen Programmen bleibt diese Verteilung über die gesamte Programmlaufzeit gleich. Bei einem dedizierten Rechensystem ist diese Herangehensweise angemessen, weil die Umgebungsbedingungen sich kaum verändern. Im Grid gilt dies nicht. Versucht man, im Grid eine optimale Auslastung zu erreichen, muss die Lastverteilung den sich verändernden Umgebungsbedingungen angepasst werden.

Zwar gibt es weitere Umgebungsbedingungen, jedoch wird hier nur auf die verfügbare Rechenzeit der Knoten eingegangen. Verfahren zur Anpassung an andere Umgebungsbedingungen werden in Kapitel 6 erwähnt.

Ändert sich während der Ausführung die auf einem Knoten verfügbare Rechenzeit, so muss dynamisch die anteilige Last angepasst werden, um eine optimale Auslastung beizubehalten.

Definition: Optimale Lastverteilung in Abhängigkeit der jeweiligen verfügbaren Rechenzeit auf den Knoten Seien T ein Zeitraum und $1..n$ die zur Berechnung verwendeten Knoten in einem Grid. Seien außerdem p_i die verfügbare Rechenzeit des Knotens i und L_i seine Arbeitslast für T, dann gilt: Die Lastverteilung Φ ist optimal, gdw.

$$\frac{Berechnungszeit_1(L_1)}{p_1} = \ldots = \frac{Berechnungszeit_n(L_n)}{p_n}$$

Zu beachten ist hierbei, dass die Bedingung $\frac{Berechnungszeit_n(L_n)}{p_n} = 1$ nicht in der Definition auftaucht. Es spielt keine Rolle, ob die Knoten überlastet sind;

wichtig ist, dass sie nach Versteichen von T den gleichen Anteil ihrer Arbeitslast vollendet haben.

Es lässt sich leicht folgern, dass es für jede Menge von Umgebungsbedingungen eine solche optimale Lastverteilung gibt.

Das Besondere im Grid ist, dass die Umgebungsbedingungen

- weniger vorhersagbar sind,
- sich schneller verändern und
- (auf den einzelnen Knoten unterschiedlicher sind,)

als dies bei dedizierten Rechensystemen der Fall ist.

Daher ist es nicht möglich, bei Programmstart eine optimale Lastverteilung für den Rest des Programmlaufs zu bestimmen. Einen guten Anhaltspunkt für die momentanen Umgebungsbedingungen bieten empirische Messungen. Damit kann nicht ausgeschlossen werden, dass sie sich in nächster Zeit ändern. Wenn die Messungen allerdings in kürzeren Abständen wiederholt werden, als sich die Umgebungsbedingungen verändern, so kann die Zeit, während der eine die Lastverteilung nicht optimal ist, beschränkt werden.

Analytisch ist die optimale Verteilung nicht bestimmbar, weil die Umgebungsbedingungen nicht antizipiert werden können. Man kann jedoch annehmen, dass die einmal ermittelten Umgebungsbedingungen eine gewisse Zeit, wenn auch nicht während der gesamten Programmlaufzeit, gültig sind. Diese Annahme trifft zu, wenn konkurrierende Prozesse auf den Knoten gestartet werden, die eine Laufzeit ab mehreren Sekunden besitzen. Für viele Anwendungen des Grid-Computing trifft dies zu, sogar lassen sich im Benutzerverhalten bei Privatrechnern ähnliche Tendenzen erkennen.
TODO: Literatur: Statistik für die Benutzung von Privat-PCs

3.2.2 Regelung der Lastverteilung

Beispiel Sei Φ eine für die Umgebungsbedingungen optimale Lastverteilung.

$$\frac{Berechnungszeit_k(L_k)}{p_k} = t_0$$

Nun verändern sich die Umgebungsbedingungen derart, dass Knoten k für den nächsten Messzeitraum T nur noch die Hälfte der Rechenzeit zur Verfügung steht.

$$\frac{Berechnungszeit_k(L_k)}{0.5 * p_k} = 2 * t_0$$

Da die anderen Knoten mit unveränderter Geschwindigkeit weiterrechnen, ist das Optimalitätskriterium verletzt. Knoten k kann in der vorgegebenen Zeit nur 50%

seiner Last vollenden. Dies ist von Nachteil, weil sich dadurch die gesamte Berechnung verzögern kann. Wenn Berechnungen anderer Knoten von den Ergebnissen von k abhängen, so müssen diese auf deren Fertigstellung warten. Vergleichbar verhält es sich, wenn andere Umgebungsbedingungen sich verändern.

Da nun eine empirische Messung durchgeführt wird, kann eine *Lastumverteilung* vom Knoten k zu den anderen Knoten die Optimalität wiederherstellen. Die unoptimale Lastverteilung wirkte also nur für den Zeitraum einer Zeitmessung. Der Rest der Berechnung wird hinsichtlich der Verteilung der Last nicht beeinträchtigt. Lediglich die Berechnungskapazität des Gesamtsystems verringert sich um die halbe Kapazität von k, was allerdings bei einer größeren Anzahl von Knoten kaum ins Gewicht fällt.

Im Folgenden wird darauf eingegangen, wie eine solche Lastumverteilung realisiert werden kann.

Diese Problemsituation lässt sich mit einem dynamischen Regelsystem (siehe Abbildung 3) angehen. Die zeitlich veränderbare Größe der Lastverteilung soll hinsichtlich dem Optimalitätskriterium geregelt werden. Die wechselnden Umgebungsbedingungen stören diese Größe. Anhand von Messungen wird die momentane Ungleichverteilung (*Istwert*) gemessen und von einem Regelglied analysiert. Dieses ist auch dafür zuständig, Umverteilungen einzuleiten (*Stellwert*), welche dann auf den Knoten ausgeführt werden (*Regelstrecke*). Eine Rückkopplung besteht darin, dass sich eine erfolgte Umverteilung wieder auf die Lastverteilung auswirkt. TODO: Eigentlich ist für einen Regelkreis ein *Maß* für die Optimalität nötig.

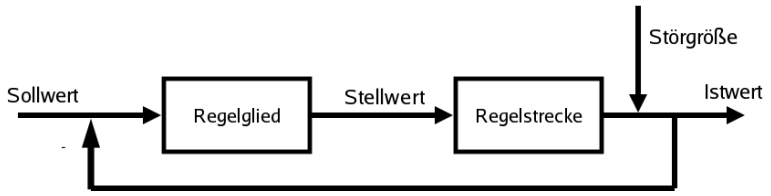

Abbildung 3: Ein Regelkreis

Mittlerweile sind genaue Uhren auf den meisten gängigen Rechensystemen verfügbar. Damit ist es möglich, jeden Knoten die Rechenzeit eines festgelegten Abschnitts seiner Berechnungen messen zu lassen. Im Folgenden wird dieser Zeitabschnitt *globaler Zeitschritt* genannt. Die ermittelte Zeit wird an das Regelglied weitergeleitet. Dieses vergleicht die empfangenen Werte und verteilt die Rechenlast neu, so dass das Optimierungskriterium der Gleichverteilung möglichst erfüllt wird. Die Knoten wiederum berechnen nun ihren neu verteilten jeweiligen Anteil, was die Grundlage für die nächste Zeitmessung darstellt.

Als weiteres wesentliches Performanzkriterium empfiehlt es sich, auch die

Kommunikationsleistung zu messen. Da der vorliegende Prototyp dies nicht berücksichtigt, wird in Kapitel 6 darauf eingegangen.

3.2.3 Umverteilungszyklus

Definition: Latenz des Lastausgleichs Die *Latenz* ist die Dauer, innerhalb der auf eine Änderung der Umgebungsbedingungen reagiert werden kann. Diese ist wegen vorhandenem Nichtdeterminismus meist nicht genau bestimmbar, deshalb wird sie eingegrenzt in ein Intervall zwischen *minimaler Latenz* und *maximaler Latenz.*

Für das Umverteilungsprotokoll lässt sich nun die Latenz bestimmen.
Es wird die Zeitdauer für die Berechnung der Last eines globalen Zeitschritts gemessen und abgeschickt. Aus dieser berechnet das Regelglied eine neue Lastverteilung. Es empfiehlt sich nicht, erst auf die Umverteilungsinformation vom Regelglied zu warten und dann weiterzurechnen, weil so alle Knoten des Netzwerks auf das Regelglied warten müssen, was eine unnötige Synchronisation bedeutet. Stattdessen berechnen die Knoten einen weiteren globalen Zeitschritt, obwohl die Lastverteilung eventuell nicht ausgeglichen ist. Erst danach erhalten sie die Umverteilungsinformation vom Regelglied. Dadurch hat dieses einen ganzen globalen Zeitschritt für den Algorithmus, der eine neue Lastverteilung anhand der Messwerte ermittelt, zur Verfügung.

So werden die Zeitschritte in die sich einander abwechselnden Phasen der Zeitmessung und dem Empfangen der Umverteilungsinformation unterschieden. Zu Beginn eines *Zeitmessungsschrittes* wird jedem Knoten die zu berechnende Last zugewiesen. Die gemessene Zeit ist also eine erste Bewertung der neuen Lastverteilung. Das Regelglied empfängt die Messdaten aller Knoten und kann so die Optimalität der neuen Verteilung bewerten oder zwischenzeitlich aufgetretene Änderungen der Umgebungsbedingungen erkennen. Ab dem Zeitpunkt der Messung bis zum Zeitpunkt, zu dem die daraus erstellte neue Lastverteilung bei den Knoten empfangen wird, wird keine erneute Zeitmessung durchgeführt. Eine solche Zeitmessung gibt nämlich die bisherige Lastverteilung wieder, das Regelglied hat aber schon mit der Berechnung einer neuen Lastverteilung begonnen und kann weitere Werte nicht mehr miteinbeziehen. Erst nachdem der *Umverteilungsschritt* abgeschlossen und die neue Lastinformation von den Knoten empfangen ist, macht es Sinn, erneut eine Zeitmessung durchzuführen.

Tritt eine Überlastung eines Knotens auf, so wird Rechenzeit vom Prozess abgezogen. So lange dem System diese veränderte Lastsituation nicht bekannt ist, wird dem Knoten weiterhin die gesamte Last zugewiesen, die nun allerdings nicht mehr innerhalb der vorgesehenen Zeit abgearbeitet werden kann. Im schlechtesten Fall tritt die Überlastsituation zu Beginn eines Umverteilungszeitschritts auf, in dem die Lastsituation nicht gemessen wird, so dass der Prozess den und den nächsten Zeitschritt mit geringer Rechenkapazität bewältigen muss, bevor eine

Zeitmessung durchgeführt und an das Regelglied weitergeleitet wird. Während das Regelglied die Messdaten empfängt und die Last entsprechend umverteilt, berechnen die Knoten einen weiteren Zeitschritt. Nach dessen Beendigung ist die Umverteilungsinformation bei den Knoten eingetroffen und wird ab dem nächsten Zeitschritt berücksichtigt. Die maximale Latenz liegt also bei drei globalen Zeitschritten: Ein Umverteilungszeitschritt, bei dem nicht gemessen wird, ein Zeitschritt bis zur Messung und einen bis zur Einarbeitung der Umverteilungsinformation.

Die minimale Latenz ist die Zeit, die für das Abschicken der Lastinformation, der Ermittelung der Umverteilung und der Übersendung der Umverteilungsinformation benötigt wird. Da das Regelglied zu Beginn eines Umverteilungsschritts Lastinformation empfängt und an dessen Ende die Umverteilung einleitet, beträgt die minimale Latenz einen globalen Zeitschritt.

Um den Bedarf für eine Umverteilung sinnvoll bewerten zu können, müssen dem Regelglied folgende Informationen zur Verfügung stehen:

- der Zeitbedarf jedes einzelnen Knoten für die Abarbeitung eines Messungsschritts und
- die Arbeitslast jedes Knoten, die im Messzeitraum zur Berechnung stand.

Der Zeitbedarf lässt sich als Momentanaufnahme nur vor Ort am Knoten bestimmen. Deshalb ist es nötig, diesen Wert vor einer Umverteilungsanalyse an das Regelglied weiterzuleiten.

Das Regelglied stellt den Zeitbedarf der Knoten in Beziehung. Die Umverteilung hat das Ziel, den Zeitbedarf für alle Knoten gleich zu machen. Dafür muss allerdings auch die Arbeitslast herangezogen werden.

Beispiel: Es soll eine Umverteilung zwischen zwei Knoten i und j eingeleitet werden. O.B.d.A. sei der Zeitbedarf t_i von Knoten i im Messzeitschritt größer als t_j, der von Knoten j. q_i sei die Anzahl der virtuellen Prozessoren des Knotens i und q_j die des Knotens j. Die Anzahl der von Knoten i nach Knoten j umzuverteilenden virtuellen Prozessoren berechnet sich nach folgendem Schema:

$$\frac{t_i - t_j}{\frac{t_i}{q_i} + \frac{t_j}{q_j}}$$

3.2.4 Anforderungen der dynamischen Lastzuordnung

Ist die Lastzuordnung statisch festgelegt, genügt die Problemgröße und die Anzahl der Knoten, um die Lastverteilung zu berechnen. Beides ist als Programmparameter gegeben und ändert sich zur Laufzeit nicht. Im Code genügen arithmetische Ausdrücke mit den beiden oder davon abgeleiteten Parametern, um die Lastverteilung zu berechnen. Eine Lastumverteilung ist dabei nicht möglich.

Will man nun eine Lastumverteilung möglich machen, ist der Zustand der Lastverteilung in Datenstrukturen mitzuführen. Es wird eine Bibliothek vorgeschlagen, die diese Datenstrukturen implementiert und ihre Funktionalität durch folgende Schnittstellen zur Verfügung stellt.

- **Zuordnung realer Prozessoren zu virtuellen Prozessoren** ($RP \rightarrow VP$)
  $RPtoVP(p \in R) := v|v \in V \wedge$ Diese Sicht ermöglicht eine Aufzählung aller zu berechnenden Iterationen eines Knotens. Außerdem können so die umzuverteilenden virtuellen Prozessoren ermittelt werden, wenn ein Knoten überlastet ist.

- **Zuordnung virtueller Prozessoren zu realen Prozessoren** ($VP \rightarrow RP$)
  Mit Hilfe dieser Sicht können die Berechnungsorte eines beliebigen virtuellen Prozessors für alle vergangenen Zeitschritte ausgelesen werden. So kann die Quelle einer Datenabhängigkeit ermittelt werden, auch, wenn der virtuelle Prozessor zwischenzeitlich umverteilt wurde und nun auf einem anderen Prozessor berechnet wird. Das Datum befindet sich noch auf dem Knoten, der es berechnet hat und muss von dort verschickt werden.

- **Umverteilung virtueller Prozessoren**
  Mit dieser Routine kann die momentane Lastverteilung verändert werden. Sie wird bei der Umverteilung verwendet.

3.3 Datenabhängigkeiten

Durch die Umverteilbarkeit von virtuellen Prozessoren werden Kommunikationen erschwert. Zum Zeitpunkt der Berechnung eines Datums kann im Allgemeinen nicht festgestellt werden, von welchem Knoten die Berechnungsergebnisse gebraucht werden, selbst, wenn Wissen über die Datenabhängigkeiten vorliegt. Es kann nun sein, dass sich das Ziel einer Datenabhängigkeit zum Zeitpunkt, wenn das Datum gebraucht wird, nicht mehr auf dem realen Prozessor befindet, als zum Zeitpunkt der Berechnung des Datums, weil zwischenzeitlich umverteilt wurde.

Eine gängige Technik für parallele Programme ist es, von anderen Prozessen benötigte Daten unmittelbar zu verschicken, sobald die Berechnungsergebnisse vorliegen. So wird gewährleistet, dass der Nachricht so viel Zeit wie möglich zur Verfügung steht, um das Ziel zu erreichen, so dass Berechnungen nicht verzögert werden.

Wird allerdings das Ziel der Datenabhängigkeit umverteilt, befindet es sich auf einem anderen realen Prozessor. Eine Möglichkeit wäre nun, den Empfänger die Nachricht weiterleiten zu lassen, bis die Daten denjenigen Prozessor erreichen, der sie benötigt. In Abbildung 4 wird dies verdeutlicht. Dies führt aber zu einem

beträchtlichem Mehraufwand an Kommunikation. Außerdem handelt es sich bei den Zwischenstationen, die die Nachricht passiert, um Knoten, von denen Last abgezogen wurde, möglicherweise, weil eine Überlastsituation vorlag. Die Weiterleitung der Daten abzuwickeln belastet diese Knoten zusätzlich. Daher ist diese Möglichkeit nicht sinnvoll.

Eine Lösung des Problems besteht darin, dass berechnete Daten so lange auf dem realen Prozessor verbleiben, bis sie gebraucht werden. Wie in Abbildung 5 gezeigt, muss auf diese Weise immer nur eine Kommunikation durchgeführt werden, da nach dem Versenden bis zum Zeitpunkt, an dem die Daten benötigt werden, keine Umverteilung mehr stattfinden kann. Wichtig dabei ist, dass Umverteilungen *vor* dem Versenden der Daten für diesen Zeitschritt durchgeführt werden.

3.4 Prototypische Implementierung

Um praktische Erfahrungen mit dem vorgestellten Verfahren des Lastausgleichs anhand empirischer Messungen zu sammeln, wurde ein Prototyp des Protokolls implementiert. Im Folgenden werden einige Eigenschaften davon hervorgehoben dargestellt.

3.4.1 Client-Server-Architektur

Der zuvor genannte Regelkreis ist mit einer *Client-Server-Architektur* realisiert, wobei ein herausgehobener Knoten als *Server* die Funktion des Regelglieds übernimmt. Alle übrigen Knoten schicken ihre Zeitmessungen zum Server, der die Berechnung des Lastausgleichs übernimmt und Umverteilungen anstößt. Er selbst nimmt an den Berechnungen nicht teil.

Die derzeitige Implementierung sieht nur einen Lastausgleich zwischen dem schnellsten und dem langsamsten Prozess vor. Dies erfüllt nicht das Optimalitätskriterium, da die anderen Knoten nicht berücksichtigt werden. Tritt eine Überlast auf einem Prozessor auf, so wird diese erst nach und nach, im Lauf mehrerer Umverteilungszyklen, auf das System verteilt. Sofern genügend globale Zeitschritte vorgesehen sind, wirkt sich dieses Problem jedoch kaum aus, wie aus den Untersuchungsergebnissen in Kapitel 5 hervorgeht.

3.4.2 Programmablauf

Abbildung 6 verdeutlicht den Ablauf des Protokolls.

Die optimale Zeitspanne zwischen zwei Umverteilungszyklen lässt sich anhand der folgenden Kriterien einschränken:

- Falls sich auf einer Maschine mehrere konkurrierende Prozesse die Rechenzeit teilen müssen, wird diese durch den Scheduler des Betriebssystems in

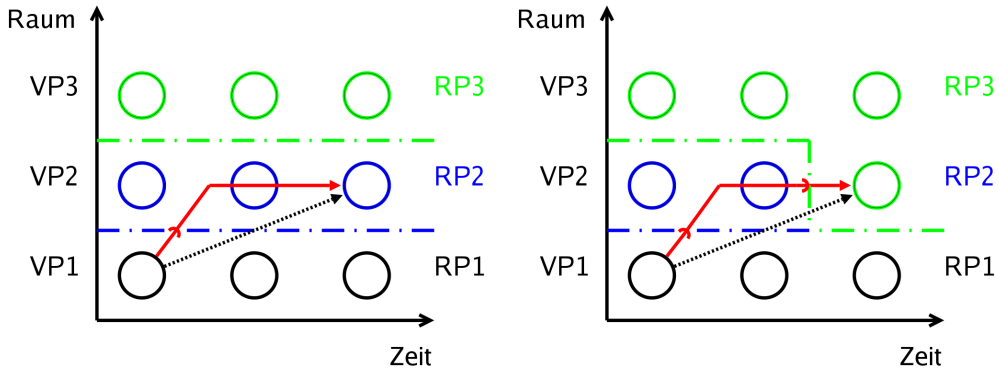

Abbildung 4: Kommunikation sofort nach Berechnung. Auf diese Weise steht am meisten Zeit zur Verfügung, um das Datum zu versenden. Allerdings führt eine Umverteilung des virtuellen Prozessors dazu, dass das Datum mehrmals verschickt werden muss.

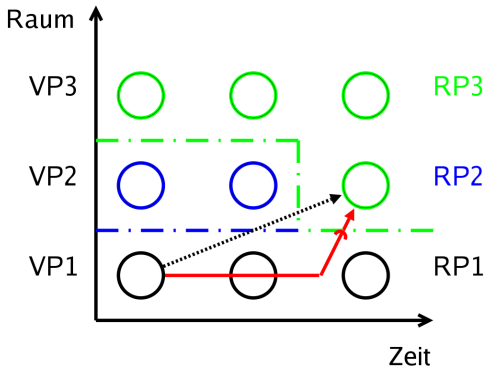

Abbildung 5: Wird hingegen erst bei Bedarf verschickt, so muss nur ein Mal gesendet werden

Zeitscheiben von betriebssystemabhängiger Größe aufgeteilt. Die entstehende zeitliche Granularität verfälscht die Zeitmessung. Daher sollte die Zeitspanne einer Messung zumindest so lange sein, dass jeder Prozess zwischenzeitlich gerechnet hat.

- Bei jedem Umverteilungs- und bei jedem Messungszeitschritt tritt protokollseitig Kommunikation auf. Wird die Dauer eines Zeitschritts kürzer gewählt, so erhöht sich der protokollseitige Overhead.
- Die Latenz des Lastausgleichs hängt direkt von der Dauer eines Umverteilungszyklus ab. Es sollten während der Programmlaufzeit ausreichend viele Zyklen durchlaufen werden, so dass eine während der Latenz vorherrschende suboptimale Lastverteilung keine große Relevanz hat.

3.4.3 Datenstrukturen

Bei den in Abschnitt 3.2.4 Datenstrukturen, die den Zustand der Lastverteilung beinhalten, ist Performanz ein wichtiges Kriterium, da bei jeder Kommunikation zwischen den Knoten in ihnen gesucht werden muss.

Reale und virtuelle Prozessoren Sowohl die Menge der realen als auch der Menge der virtuellen Prozessoren ist total geordnet. Daher werden dafür im Folgenden immer Listen mit Startindex 0 verwendet. Die Anzahl der realen und virtuellen Prozessoren wird für jeden Knoten als bekannt vorausgesetzt.

$RP \rightarrow VP$

```
RPtoVP(integer) -> [[integer]]
```

Jeder reale Prozessor iteriert jeden globalen Zeitschritt über seine aktuelle Arbeitslast. Da die Granularität der Parallelität fein sein kann, ist diese in Listen von Intervallen angelegt. Bei der Zuordnung der Arbeitslast sollte dabei möglichst eine Zerstückelung vermieden werden. Dies hat den Vorteil, dass ganze Intervalle von virtuellen Prozessoren in konstanter Zeit umverteilt werden können. Zum Iterieren sind pro Intervall zwei Werte aus der Datenstruktur zu übertragen.

$VP \rightarrow RP$

```
VPtoRP(integer, integer) -> integer
```

Sofern
TODO: HIER WEITERMACHEN
Eine Liste von Realprozessoren mit Zeitstempel. Der Zeitstempel gibt an, ab wann diese Residenz gueltig ist. Es muss also vom Ende der Liste weg nach einem t kleiner gleich dem angegebenen t gesucht werden. Der dazugehoerige reale

Prozessor ist der Aufenthaltsort zur angegebenen Zeit. Laufzeitueberlegungen: Fuer "kurzeÄbhaengigkeiten hat diese Datenstrukur selbst bei haeufigen Umverteilungen eine nahezu konstante Laufzeit. Fuer Abhaenigkeiten aber, bei denen zwischen Quellzeit und Zielzeit n Umverteilungen stattgefunden haben, ist auch die Laufzeit der Suche mit Faktor n zu versehen (Laufzeit bei langen Abhaengigkeiten linear zur Anzahl der Umverteilungen). Trotzdem: Umverteilungen sind generell recht teuer; es ist unwahrscheinlich, dass bei vielen Umverteilungen die Laufzeit der Suche stark zu Buche schlaegt.

Der Aktualisierungsaufwand pro Umverteilung ist auf allen Knoten linear, wobei hier eher die Synchronisation und die Kommunikation die Laufzeiten ausmachen.

Bei einer Implementierung mittels Suchbaum spielt die Laenge der Abhaengigkeit keine Rolle (Laufzeit logarithmisch zur Anzahl der Umverteilungen). Es ist aber im Speicher mit den doppelten Kosten wie in Ansatz 1) zu rechnen. Ausserdem ist die Laufzeit zur Aktualisierung pro Umverteilung auch logarithmisch.

Umverteilung virtueller Prozessoren / Prozessortiles Mit dieser Routine kann die momentane Lastverteilung verändert werden. Sie wird bei der Umverteilung verwendet.

TODO:

Da sich die Datenabhängigkeiten während des Programmlaufs nicht ändern, müssen sie nicht in Datenstrukturen mitgeführt werden. Es genügt, sie im Quelltext zu berücksichtigen. Der parallele Compiler allerdings muss für die Programmtransformation Abhängigkeiten aus dem Quelltext extrahieren. Im Gegensatz zu parallelen Programmen mit statischer Lastverteilung müssen nun Quelle und Ziel von Abhängigkeiten unter Berücksichtigung der Datenstrukturen für die Umverteilung bestimmt werden.

3.4.4 Kommunikation zwischen Knoten

TODO: Aufteilung des Schleifensatzes in global sequentiell (Zeitschleife), global parallel (virtuelle Prozessoren) und lokal sequentiell (Prozessortiles).

Um Datenabhängigkeiten Rechnung zu tragen, müssen die Knoten untereinander Berechnungsergebnisse austauschen. Da der parallele Compiler Schedule und Allokation wählt, können einige Modalitäten hinsichtlich des Kommunikationsverhaltens frei bestimmt werden:

- Synchrone oder asynchrone Parallelität (TODO: Wenn dieser Punkt zu opulent ist, weglassen; allerdings besteht ein gewisses Interesse an dem Punkt, weil die dynamische Lastverteilung ja auf Synchronität angewiesen ist: ohne globale Zeitschritte wird es schwierig)
- Kollektive Kommunikation oder Punkt-zu-Punkt-Kommunikation

- Häufigkeit der Kommunikation
- Bündelung von Kommunikationen: Da Kommunikations-Startups teuer sind, wird versucht, alle Daten an den selben Empfänger zu sammeln und schließlich innerhalb einer Operation zu versenden.

Hierzu existieren verschiedene Ansätze, wie die Kommunikation effektiv gestaltet werden kann. Siehe hierzu (TODO: Literaturreferenz: Kommunikationsmodalitäten). Durch den Zusatz der dynamischen Lastverteilung treten zusätzliche Anforderungen an das Kommunikationsverhalten auf, auf die nun eingegangen wird.

Kaskadierendes Warten bei kurzen Datenabhängigkeiten Bei synchroner Parallelität ist es möglich, Abhängigkeiten innerhalb eines globalen Zeitschritts zuzulassen. Während seiner Berechnung blockiert ein Knoten einfach so lange, bis benötigte Berechnungsergebnisse eingetroffen sind. Dieses Verfahren hat Vorteile, falls eine besonders feingranulare Parallelität nötig ist. Für den dynamischen Lastausgleich ist dieses Verfahren jedoch ungeeignet, da das eingesetzte empirische Verfahren zur Leistungsmessung bei den Knoten an seine Grenzen stößt, wenn Datenabhängigkeiten dazu führen, dass Knoten aufeinander warten müssen. Dies hat seinen Grund darin, dass der Zeitbedarf für die Kommunikation zwischen den Knoten aus messtechnischen Gründen auf die Berechnungszeit aufgeschlagen wird.
Wenn Quelle und Ziel von kurzen Abhängigkeiten nicht auf dem selben Knoten berechnet werden, sind die Zielprozessoren noch im gleichen globalen Zeitschritt auf eine Berechnung angewiesen. In diesem Fall steigt nicht nur die benötigte Rechenzeit des die Verlangsamung verursachenden Knoten an, sondern auch aller Knoten, deren Berechnungen von diesem Knoten abhängen. Durch diesem Umstand erscheint es für die *Umverteilungsbedarfsanalyse* so, als ob nicht nur einer, sondern eine ganze Reihe von Knoten ihre Aufgaben nicht bewältigen können. So kann es zu Umverteilungen kommen, die entgegen dem Optimalitätskriterium durchgeführt werden. Erst durch spätere Umverteilungsschritte, wenn die Ursache der Verzögerung beseitigt ist, werden die Nachteile dieser Umverteilung wieder neutralisiert.

Dieses Problem kann vermieden werden, wenn bei der Programmtransformation auf die Bedingung *forward communication only (FCO)* innerhalb der globalen Zeitschleife geachtet wird.

Definition: forward communication only (FCO) TODO: Definition

Dies hat den Vorteil, dass die innerhalb eines globalen Zeitschritts durchzuführenden Berechnungen nicht voneinander abhängen und die Messergebnisse nicht verfälscht werden.

Sinnvoll ist es, so viele globale Zeitschritte wie die maximale Latenz der Umverteilung zu gruppieren, und dann die Bedingung FCO anzuwenden. Angenommen, eine Umverteilung reicht aus, um die Lastverteilung auszugleichen, so kann

auf eine Überlastsituation reagiert werden, bevor Messergebnisse von anderen Knoten beeinflusst werden.

Punkt-zu-Punkt-Kommunikation Zwar ist kollektive Kommunikation theoretisch effizienter als Punkt-zu-Punkt-Kommunikation, allerdings spielt bei dieser Überlegung die Transferzeit von Nachrichten eine wichtige Rolle, da in kollektiven Operationen global synchronisiert wird. Treten im Programm nur wenige Datenabhängigkeiten auf, so ist die Punkt-zu-Punkt-Kommunikation vorzuziehen.

Implementierungsskizze:

```
client() -> ()
client() ->
begin
    $\forall$ globalen Zeitschritte
    begin
        $\forall$ eigenen VPs
        begin
            Berechnung_durchführen;
            $\forall$ nichtlokalen Ziel-VPs
            begin
                (Ziel-VP)->RP finden;
                Berechnete Daten nach Ziel-RP sortiert puffern;
            end;
        $\forall$ (gepufferten) Ziel-RPs
        begin
            Gesammelte Daten versenden;
        end;
        $\forall$ nichtlokalen Quell-VPs
        begin
            Benötigte Daten nach Quell-RP sortieren;
        end;
        $\forall$ Quell-RPs
        begin
            Daten empfangen und eigenen Datensatz damit aktualisieren;
        end;
    end;
end;
```

3.4.5 Codegenerierung

Damit beliebige parallele Programme mit dem Lastausgleichsprotokoll versehen werden können, wird der dafür nötige Code bei der Codetransformation in das Programm eincompiliert.

Zum Zeitpunkt der Arbeit ist der Einbau des Lastausgleichsprotokolls in die Codegenerierung von LooPo [Loo] noch nicht abgeschlossen. Einzelheiten dazu werden noch veröffentlicht (Siehe TODO: Literaturreferenz: noch nicht veröffentlichte Publikation vom Micky).

Datenstrukturen für Lastausgleichsprotokoll Die oben erwähnte Bibliothek wird dem transformierten Programm beigefügt und beides zusammen compiliert.

Initialisation Zum Programmbeginn wird ein Knoten als Server von der Berechnung ausgenommen und der Kommunikator *MPI_COMM_NODES* erstellt, der alle rechnenden Knoten enthält, nicht aber den Server. Dies hat den Sinn, dass die besondere Stellung des Servers in der Kommunikation zwischen den Knoten nicht berücksichtigt werden muss.

TODO: Kommunikatoren: MPI_COMM_WORLD <-> MPI_COMM_NODES mehr auswalzen?

Protokollkommunikation Servercode (Zeitmessung empfangen, Umverteilungsinformation abschicken), Umverteilungsalgorithmus, Code für Client (Zeitmessung, abschicken, empfangen),

Kommunikationsgenerierung Beim dynamischen Lastausgleich stehen die Kommunikationspartner erst mit dem globalen Zeitschritt fest, in dem die Kommunikation stattfindet. Es empfiehlt sich daher, die Kommunikationspartner anhand der Datenabhängigkeiten in einem Schleifensatz aufzuzählen. Berücksichtigt werden dabei:

- die aktuelle globale Zeit
- alle Datenabhängigkeiten
- die momentan eigenen virtuellen Prozessoren anhand der mitgeführten Datenstrukturen

So können alle Datenquellen und -Senken für den momentanen Zeitschritt aufgezählt werden.

Es erscheint nicht effizient, wenn für jeden virtuellen Prozessor ein Kommunikationsstartup erfolgt. Daher ist es sinnvoll, mit einer Kommunikation alle benoetigten Daten für die direkt abhängigen virtuellen Prozessoren eines realen Prozessors für einen Zeitschritt gebündelt zu übertragen. Ein Verfahren, wie Kommunikationen zurueckgehalten und schliesslich gebuendelt (maximal eine Nachricht pro realen Prozessor) versendet werden, stellt Michael Classen **??** vor. Dazu ist folgendes Wissen auf jedem Knoten nötig:

- reale Prozessoren
- eigene virtuelle Prozessoren
- Datenabhaengigkeiten
- für jeden eigenen virtuellen Prozessor: Ort der virtuellen Zielprozessoren
- für jeden eigenen virtuellen Prozessor: Ort der virtuellen Quellprozessoren
- Pufferung von Nachrichten

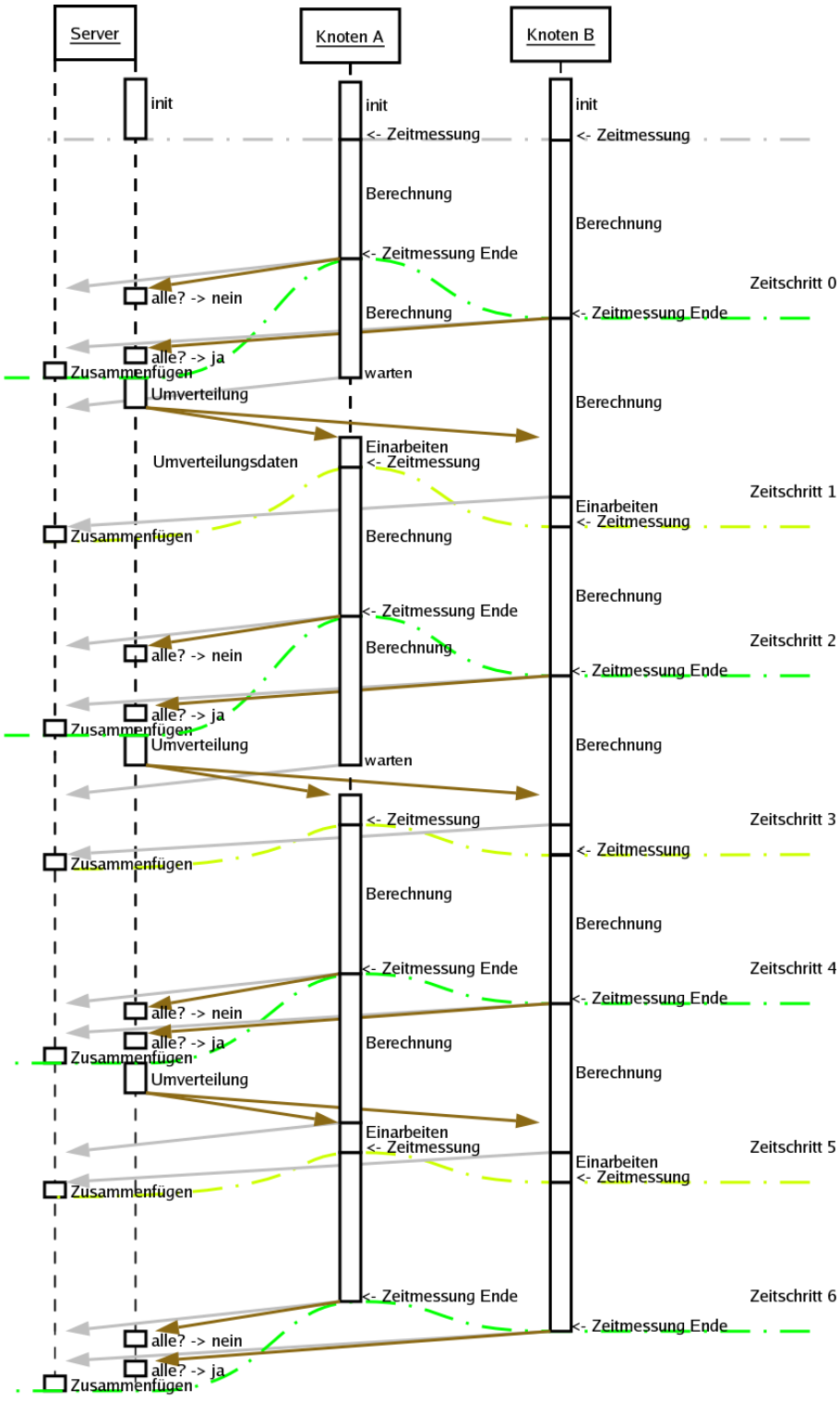

Abbildung 6: Sequenzdiagramm: Beispielablauf des Protokolls anhand zweier Knoten

4 Vergleich mit anderen Lösungsansätzen

Dieser Abschnitt zieht andere Ansätze zum dynamischen Lastausgleich zum Vergleich heran. Das *adaptive Tiling* verändert die Zuordnung von Prozessortiles zu Prozessoren so, dass eine ungleiche Auslastung, hervorgerufen durch die Form des Indexraumes, ausgeglichen werden. Dieser Ansatz ist aus der Diplomarbeit von Georg Seidel entnommen [**?**]. Ein *Taskfarm-Ansatz* vollzieht den dynamischen Lastausgleich auf einer höheren Abstraktionsebene auf der Basis einer Client-Server-Architektur [DGG+06].

4.1 Adaptives Tiling

Durch die Form des Index-Polyeders lässt sich nicht immer die anfallende Arbeitslast auf die Prozessoren gleich verteilen, wenn zusätzlich ein statisches Tiling angewandt wird. Um dieses Problem zu beheben, wird ein *adaptives Tiling* durchgeführt, das die Zuordnung der Tiles zu Prozessoren nach einer gewissen Anzahl von Zeitschritten neu verteilt. Anders als beim statischen Tiling werden die Tiles also mit der Zeit angepasst, um einen Lastausgleich zu erzielen. Dies geschieht anhand eine statischen Analyse des Indexraumes.

Nachdem die Parallelisierungsphase des Programms abgeschlossen ist, wird die entstandene Parallelität zu Tiles zusammengefasst. Als Eingabe für das adaptive Tiling stehen das Raum-Zeit-Mapping, der Indexzielraum, die Abhängigkeiten, die Tilegröße und eine Matrix zur Verfügung, die das Gitter der Tiles beschreibt. Optimiert werden Parameter wie die Auslastung der Prozessoren und die Anzahl der Kommunikations-Startups.

4.1.1 Gemeinsamkeiten

Die Zeit wird in gleich lange Intervalle aufgeteilt. Ein *Retiling* kann nur zu Beginn eines Intervalls stattfinden. So wird dem Overhead einer zu großen zeitlichen Granularität aus dem Weg gegangen. Die Unterscheidung zwischen lokaler und globaler Zeit ist also auch hier vorhanden.

4.1.2 Unterschiede

Ein wesentlicher Unterschied zum dynamischen Lastausgleichsprotokoll besteht darin, dass das sogenannte *Retiling* nicht zur Laufzeit entschieden, sondern statisch vorherberechnet wird. Als Eingabe für die Umverteilung stehen also keine empirischen Laufzeitmessungen zur Verfügung. Außerdem wird in der vorgelegten Form nicht der unterschiedlichen Ausstattung der Knoten Rechnung getragen. Es wird eine statisch vorhersagbare Ungleichverteilung von Arbeitslast anhand des Programmcodes berechnet. Ein Vorteil gegenüber der empirischen Herangehensweise ist, dass die entsprechende Anpassung ohne Latenz geschehen kann.

Allerdings kann den Unwägbarkeiten im Grid damit nicht entgegnet werden, da diese vor der Programmausführung nicht bekannt sind.

Technisch ist es denkbar, beide Ansätze zu kombinieren, um aus statisch extrahierbarem Wissen Lastausgleich ohne Latenz durchzuführen, während immer noch dynamisch auf ein nicht vorhersagbares Verhalten reagiert werden kann. Allerdings nimmt der dynamische Lastausgleich an, dass die empirisch gesammelte Information auf den folgenden Zeitschritt übertragen werden kann, was bei einer vollzogenen Umverteilung von anderer Seite her nicht mehr gilt. Die Umverteilungszeitpunkte können in im Vergleich zur Programmlaufzeit kurzen Abständen (im Bereich von einigen Sekunden) gewählt werden. Gegenüber dem Programmieraufwand und den Komplikationen mit der dynamsichen Lastausgleich steht also der Nutzen, den eine geringere Latenz bei der Reaktion auf statisch vorhersagbaren Lastspitzen darstellt.

4.1.3 (Fazit)

LBP behandelt ein anderes Ziel der Lastumverteilung: in einer wenig vorhersagbaren Struktur ein paralleles Programm auszuführen.

Retiling besitzt besondere Vorteile bei "herkömmlichen" Parallelrechnern, adressiert aber nicht die besonderen Erfordernisse im Grid.

Eventuell ließe sich adaptives Tiling zusätzlich zum dynamischen Lastausgleichsprotokoll in ein paralleles Programm einbringen. Das Lastausgleichsprotokoll behandelt ein sich dynamisch veränderndes Tile als virtueller Prozessor und ignoriert, dass der Berechnungsaufwand sich mit der Zeit verändert. Problematisch ist allerdings, dass sich durch dieses Retiling möglicherweise Kommunikationen verändern, was zu einer Verkomplizierung der Kommunikationsschleife der Knoten führt. Ob dieses Verfahren technisch machbar ist, ist noch zu erforschen.

TODO: Dieser Abschnitt ist zum vorhergehenden redundant -> zusammenfassen

4.2 Taskfarm-Ansatz

Bei der Parallelisierung wird die Arbeitslast in kleine Teile zerlegt, die parallel ausgeführt werden können. Diese *Tasks* können bei der Ausführung von einem Server auf verfügbare Clients frei verteilt werden. Mit jedem Task werden Quellcode und Eingabedaten an den Client geschickt. Ist die Berechnung beendet, sendet der Client das Ergebnis zurück. Der Server besitzt Wissen über Abhängigkeiten zwischen den einzelnen Tasks und verteilt nur solche Tasks, deren Datenquellen bereits abgearbeitet sind. Sobald ein neues Ergebnis vorliegt, schaltet der Server je nach Abhängigkeiten neue Tasks frei, die dann zum Abruf für die Clients zur Verfügung stehen. Eine Kommunikation der Clients untereinander findet nicht statt.

4.2.1 Gemeinsamkeiten und Unterschiede

Das Konzept der *Tasks* ist dem Konzept der virtuellen Prozessoren ähnlich, mit der Ausnahme, dass die Arbeitslast auch zeitlich fragmentiert wird.

In beiden Herangehensweisen bedient man sich eines Servers, der die Verteilung der Arbeitslast verwaltet. Der Server des Lastausgleichsprotokolls nimmt nur bei Bedarf Einfluss, um einer Ungleichverteilung entgegenzuwirken. Im Taskfarm-Ansatz besitzt der Server einen größeren Verantwortungsbereich, da die Clients untereinander nicht kommunizieren. Außerdem erfordert die zeitliche Fragmentierung eine stetige Betätigung des Servers, weil die Knoten nicht von selbst zum Programmende rechnen.

Die Anbindung der Knoten im Taskfarm-Ansatz ist flexibler gestaltet, weil über den Server alle Kommunikation läuft. So ist es für den Server möglich, abgestürzte Knoten zu ersetzen. Der Ausfall eines Knotens stellt also keine Gefahr für die Berechnung dar, es sei denn, es handelt sich dabei um den Server selbst. In der momentanen Fassung führt beim Lastausgleichsprotokoll ein Totalausfall eines Knotens zum Abbruch des Programms. In Kapitel 6 wird ein Verfahren vorgestellt, wie diesem Umstand beizukommen ist, um die Berechnung weiterführen zu können.

Da die gesamte Kommunikation über den Taskfarm-Server läuft, ist dessen Netzanbindung entscheidend für die Performanz des Systems. Bei Berechnungen mit erheblichem Kommunikationsaufwand und einer größeren Anzahl von Knoten kann dies zum begrenzenden Faktor für den Speedup führen. Die Belastung des Lastausgleichsprotokoll-Server steigt nur mit der Anzahl der Knoten, nicht aber mit dem programmspezifischen Verhältnis zwischen Kommunikation und Berechnung. Außerdem ist der protokollseitige Kommunikationsbedarf vergleichsweise gering, so dass auch bei einer größeren Anzahl von Knoten kein Flaschenhals zu erwarten ist.

Die Tasks stellen eine Vergröberung der zeitlichen Granularität dar. Eine zu große Granularität resultiert in Overhead beim Server, eine zu Geringe dagegen vergrößert die Latenz bei unvorhersehbaren Ereignissen, wie der Ausfall eines Knotens. Außerdem sind im Taskfarm-Ansatz Performanzeinbußen zu erwarten, da ein Task keine Kommunikation generieren kann. Es müssen also die Datenquellen aller Berechnungen im Task bereits bei seiner Vergabe berechnet worden sein. Die Vergrößerung eines Tasks stellt also auch eine Einschränkung der Parallelität dar.

4.2.2 (Fazit)

Die größere Abstraktion beim Taskfarm-Ansatz bietet die Möglichkeit, zusätzliche Funktionen leicht zu integrieren. Zum Beispiel ist beim Ausfall eines Knotens beim Lastausgleichsprotokoll ein Rollback zu implementieren, was zu einem viele Knoten betreffenden Overhead führen kann. In der Taskfarm ist dieser Overhead

vergleichsweise gering. Dies geht allerdings einher mit einer einer Verteuerung der Kommunikation, da diese den Umweg über den Server nehmen muss.

Gerade wegen der günstigeren Kommunikation ist das Lastausgleichsprotokoll mit guter Wahrscheinlichkeit performanter. Dies ist deshalb gerade im Grid-Computing ein wichtiger Grund, weil durch heterogene Architektur und tendenziell langsame Netzanbindung der erwartete Speedup nur gering ausfällt.

5 Empirische Untersuchung

Zum Zeitpunkt der Arbeit ist die Einbindung des vorgestellten Protokolls in den Compiler *LooPo* [Loo] noch nicht abgeschlossen. Um dennoch eine empirische Untersuchung durchführen zu können, wurde deshalb ein Programm zur Matrixmultiplikation per Hand um den für das Protokoll erforderlichen Code ergänzt.

5.0.3 Versuchsaufbau

Das Beispiel der Matrixmultiplikation wird gewählt, weil es einfach zu implementieren ist, vor allem deshalb, weil bei genügend großem Tiling keine Abhängigkeiten zwischen den Knoten auftreten. Zwar treten ohne Abhängigkeiten keine besonderen Herausforderungen auf, weshalb ein komplexes Beispiel mehr Aussagekraft hätte. Allerdings ist die Erstellung eines solchen Beispiels und die Fehlersuche von Hand wesentlich aufwändiger und würde den Rahmen der Arbeit sprengen.

Berechnet wird $A * B = C$, wobei $A \in \mathbb{R}^{n \times m}$, $B \in \mathbb{R}^{l \times n}$ und $C \in \mathbb{R}^{l \times m}$ mit $l, m, n \in \mathbb{N}$. $C_{i,j}$ ist definiert als $\sum_{k=0}^{n}(A_{k,j} * B_{i,k})$, wobei $i, j, k \in \mathbb{N}_0$.

Der Algorithmus besitzt eine kubische Laufzeit in Abhängigkeit der Eingabevariablen: $\Theta(l, m, n) = l * m * n$. Über jede der Eingabevariablen wird iteriert, und zwar in der Reihenfolge $l \rightarrow m \rightarrow n$. Eine *output*-Datenabhängigkeit besteht in der innersten Schleife bezüglich C, $(i, j, k) \rightarrow (i, j, k')$, wobei $k \leq k'$.

Die k-Schleife wird zum Tiling verwendet und beschreibt die lokale Zeit. Eine Raumschleife über j legt die virtuellen Prozessoren fest. Die äußerste i-Schleife bestimmt die globale Zeit und damit die Umverteilungszeitpunkte. Je nach Größe der Eingabematrizen kann also das Verhältnis zwischen globaler Zeit, Anzahl der virtuellen Prozessoren und Tilegröße variiert werden.

Im Beispiel werden A und B nicht zu Beginn kommuniziert, sondern nach einem einfachen Schema bei jedem Knoten berechnet. Die Berechnungsvorschrift lautet: $A_{i,j} = B_{i,j} = i + j$ (7)

Es ist also nicht notwendig, die Eingabematrices auf die Knoten zu übertragen, da sie dort anhand der Größe berechnet werden können. Nur das Sammeln der Berechnungsergebnisse auf einem der Knoten tritt als Kommunikation auf, die als solche bei der Zeitmessung zu Buche schlägt. Nach jedem globalen Zeitschritt werden die Berechnungsergebnisse (Zellen aus C) an den Server geschickt.

Zum Vergleich wurden drei verschiedene Versionen des Programms angefertigt. Eine sequentielle Version, eine parallele Version ohne Lastausgleich und den dazugehörigen Datenstrukturen und eine parallele Version mit Lastausgleich, bei der der erste Knoten im Netzwerk als Server deklariert wird.

Die Zeitmessung beginnt nach einer Barrierensynchronisation (siehe [MPI]) vor Berechnungsbeginn und endet, nachdem alle Ergebnisse auf einem Knoten zusammengelaufen sind.

```
matrixMult([[double]], [[double]]) -> [[double]]
matrixMult(A, B) ->
begin
    for i:= 0 to height(A)-1 do
        for j:= 0 to width(B)-1 do
        begin
            result[i,j]:= 0;
            for k:= 0 to width(A)-1 do
                result[i,j]:= result[i,j] + A[i,k] * B [k,j];
        end;
    result:= x;
end;
```

Abbildung 7: Matrixmultiplikation

5.0.4 Auswertung

Für die empirischen Tests wurden die Arbeitsplatzcomputer *bartok*, *chopin*, *ravel*, *gershwin*, *thomas* und *ett* des Lehrstuhls für Rechensysteme an der Universität Passau herangezogen.

TODO: Systemspezifikationen der Rechner darstellen.

Als Parameter wurden die Werte $l = m = n = 1024$ gewählt. Der Speicherbedarf für die drei Matrices beläuft sich dabei auf ca. 25 MB im Hauptspeicher. Bei $l = m = n = 2048$ liegt dieser bei ca. 100 MB. Eine Parameterwahl von $l = 64; m = 1024; n = 32768$ oder $l = 32768; m = 1024; n = 64$ führt zu ca. 280 MB Speicherbedarf.

Tabelle 5 zeigt die Ergebnisse der Versuchsanordnung. *ett* schneidet in der sequentiellen Berechnung wesentlich schlechter ab als die anderen Rechner. Da die parallele Programmversion davon ausgeht, dass alle Knoten gleich schnell rechnen, werden die anderen Knoten durch *ett* ausgebremst, obwohl ihr Berechnungsabschnitt schon bei etwa sechs Sekunden abgeschlossen ist.

Eine anhand der sequentiellen Berechnungszeiten bei den Parametern $l = m = n = 1024$ geschätzte optimale parallele Berechnungszeit ohne Kommunikation liegt bei 6,77 Sekunden. Wegen notwendigen Kommunikationen und dem Umstand, dass der Server nicht an der Berechnung teilnimmt, kann dieser Wert von der Lastausgleichsversion des Programms nicht erreicht werden.

Die besten Ergebnisse erzielt das Lastausgleichsprotokoll bei Werten von τ zwischen 0,0 und 0,2. Dies zeigt, dass der Zusatzaufwand der Umverteilung in den verwendeten Beispielen eher gering ist, obwohl bei $\tau = 0,0$ tatsächlich zu jedem Umverteilungszeitschritt ein Lastausgleich erfolgt.

TODO: Sagen, dass τ am besten ausgeschaltet ist. Die Umverteilung hat so gut wie keinen Overhead.

τ	Lastausgleich	ohne Lastausgleich	sequentiell
0,0	9,153s	53,481s	bartok: 34,873s
	9,069s	53,406s	chopin: 34,090s
	9,089s	53,924s	ett: 318,670s
0,1	9,283s		ravel: 36,203s
	9,129s		thomas: 35,662s
	9,282s		gershwin: 33,489s
0,2	9,361s		
	9,409s		
	9,252s		
	9,319s		
0,3	10,164s		
	9,887s		
0,4	11,388s		
0,5	12,857s		
0,8	14,629s		
1,0	64,284s		

Tabelle 1: Laufzeiten im heterogenen Rechnernetz für die Eingabewerte $l = m = n = 1024$

Liegt τ bei $1,0$, so wird kein Lastausgleich durchgeführt. Der Laufzeitunterschied gegenüber der parallelen Berechnung ohne Lastausgleichsprotokoll ergibt sich im Wesentlichen dadurch, dass der Knoten *bartok* als Server nicht an der Berechnung teilnimmt.

Da ein Lastausgleich in der aktuellen Implementierung immer nur vom langsamsten zum schnellsten Knoten erfolgt, stellt sich ein Ausgleich der Last erst nach einigen Umverteilungen ein. Durch einen besseren Umverteilungsalgorithmus beim Server ließe sich dieser Vorgang beschleunigen (siehe hierzu Kapitel 6).

Ein Teil der empirischen Tests wurde auf dem *hpcLine*-Parallelrechner des Lehrstuhls für Programmierung an der Universität Passau *Hermes* [Her] durchgeführt. Um ein dem Grid ähnliches Verhalten zu erzeugen, wurde auf ausgewählten Knoten parallel zum Programmlauf Überlast mit Hilfe von weiteren Prozessen generiert.

Für jeden dargestellten Eingabewert werden drei Programmläufe gemacht: Ein sequentieller Lauf auf einem der Knoten, ein paralleler Lauf mit gleicher Auslastung und ein paralleler Lauf, wobei kurz nach Programmstart auf einem Knoten eine Überlast erzeugt wird.

Weitere Variationen ergeben sich aus der Überlastsituation: Es wurde mit einfacher, doppelter, vierfacher und achtfacher Überlast auf einem Knoten experimentiert.

Die Implementierung des Lastausgleichsprotokolls initiiert einen Lastausgleich

τ	Lastausgleich	ohne Lastausgleich	sequentiell
0,0	10,083s	56,493s	bartok: 28,319s
	9,650s	56,331s	chopin: 29,869s
0,1	10,264s		ett: 327,047s
	9,850s		ravel: 29,291s
0,2	10,288s		thomas: 33,577s
	9,650s		gershwin: 28,268s
0,3	10,311s		
0,4	10,935s		
0,5	12,510s		
0,6	14,110s		
0,7	12,913s		
0,8	12,535s		
0,9	12,518s		
1,0	65,119s		

Tabelle 2: Laufzeiten im heterogenen Rechnernetz für die Eingabewerte $l = 64; m = 1024; n = 16384$

τ	Lastausgleich	ohne Lastausgleich
0,0	1,149s	6,074s
0,2	1,232s	
1,0	6,940s	

Tabelle 3: Laufzeiten im heterogenen Rechnernetz für die Eingabewerte $l = m = n = 512$

τ	Lastausgleich	ohne Lastausgleich
0,0	107,966s	437,984s
0,1	108,902s	
0,2	114,678s	
0,9	528,530s	

Tabelle 4: Laufzeiten im heterogenen Rechnernetz für die Eingabewerte $l = m = n = 2048$

Parameter	#Prozessoren	Überlast	ohne Lastausgleich	mit Lastausgleich
1024 1024 1024	32	0x	2,422s	$\tau = 0.5$ 2,717s
				$\tau = 0.5$ 2,738s
1024 1024 1024	32	2x	6,334s	$\tau = 0.5$ 18,254s
			6,497s	$\tau = 0.5$ 15,983s
				$\tau = 0.6$ 16,894s
				$\tau = 0.6$ 12,208s
				$\tau = 0.9$ 42,649s

Tabelle 5: TODO

nur dann, wenn der Geschwindigkeitsunterschied zwischen dem schnellsten Prozessor und dem Langsamsten einen Schwellwert übersteigt.

$$1 - \frac{t_{schnellster}}{t_{langsamster}} > \tau$$

Der Schwellwert $\tau \in [0; 1]$ bewirkt, dass bei kleinen Geschwindigkeitsunterschieden kein Lastausgleich durchgeführt wird. Dieser Wert ist als Programmparameter einstellbar und wirkt zum Teil wesentlich an der Laufzeit mit. Ist er zu groß gewählt, werden zu wenige Umverteilungen angestoßen und die Lastverteilung bleibt ineffizient. Zu viele Umverteilungen wegen einer zu kleinen Schwelle können wiederum Effizienzeinbußen bewirken, da die Datenstrukturen zur Lastinformation anwachsen. Besonders, wenn Abhängigkeiten über viele Zeitschritte hinweg vorhanden sind, ist dies von Nachteil.

TODO: Tabellen mit Ergebnissen

TODO: Ergebnisse auswerten

6 Ausblick

6.1 Zusammenfassung

Es wurde ein Protokoll mit prototypischer Implementierung zum dynamischen Lastausgleich in parallelen Programmen auf der Basis von empirischen Messungen gezeigt. Als Maß für den Lastausgleich wird dabei vorläufig nur die zur Verfügung stehende Rechenzeit herangezogen.

6.2 Denkbare Erweiterungen

Um den Anwendungsbereich im Umfeld vom Grid zu vergrößern, sind Erweiterungen des vorgestellten Protokolls denkbar. Einige davon werden nun in der Kürze vorgestellt.

6.2.1 Verbesserung des Umverteilungsalgorithmus

Die momentane Implementierung des Umverteilungsalgorithmus beim Server stellt erst nach mehreren Umverteilungen einen Ausgleich der Last her. Die hat seinen Grund darin, dass in einem Schritt nur Last vom langsamsten zum schnellsten Knoten transferiert wird. Bessere Ergebnisse erzielt hier eine Umverteilung von Last, bei der alle langsameren Knoten den Schnelleren in dem Maß Last abgeben, so dass die verfügbare Last bereits nach einem Schritt ausgeglichen ist. Die aktuelle Schnittstelle enthält bereits die für den Algorithmus notwendigen Daten, um die Umsetzung reibungsfrei zu gestalten.

6.2.2 Kommunikation zwischen Knoten

Die vorliegende Beispielimplementierung spart aus Komplexitätsgründen eine Kommunikation zwischen den Knoten aus. Nach dem Einbau in LooPo [Loo] generiert der Compiler die erforderlichen Kommunikationsschleifen automatisch.

6.2.3 Kommunikation als Umverteilungskriterium

Wie schon oben erwähnt, erfasst die Messung der Rechenzeit nicht alle Umverteilungskriterien. In einem kommunikationsrteichen Programm kann die Netzanbindung die Performanz entscheidend beeinflussen. Es ist also sinnvoll, auch dieses Kriterium in die Umverteilungsbedarfsanalyse miteinzubeziehen.

Mit der verwendeten MPI-Schnittstelle (**??**) nicht möglich, aktuelle Transferzeit und Durchsatz von oder zu einem Knoten zu bestimmen. Allerdings ist es denkbar, einen empfangenden Knoten die Zeit messen zu lassen, die er auf einen anderen Knoten warten muss. Übermittelt man diese Information dem Server, kann sie dieser in seinem Umverteilungalgorithmus berücksichtigen. Eventuell ist

es sogar möglich, Nachbarkeitsbeziehungen in der Netztopologie aufzufinden und in Umverteilungen auszunutzen.

6.2.4 Maßnahmen bei Ausfall eines Knotens

Die bisherige Implementierung sieht den Fall vor, dass Knoten unterschiedlich schnell ihre Berechnungen ausführen. Ist die Arbeitslast eines Paars globaler Zeitschritte bewältigt, kann die Arbeitslast so umverteilt werden, dass für die weitere Berechnung alle Knoten erwartungsgemäß gleich schnell fertig werden.

Der *worst case* für die Einschätzung der Performanzeinbußen kann berechnet werden. Die Annahme hierbei ist, dass alle Knoten auf den Überlasteten warten müssen und dass die Überlast zum schlechtest möglichen Zeitpunkt auftritt. Sei Nodes die Menge der Knoten, $T_{\text{global},i} \in \mathbb{R}$ die erwartete Berechnungsdauer des Knotens i bei aktueller Arbeitslast für einen globalen Zeitschritt, $\rho_i \in \mathbb{R}$ die Last des Knotens i und $l_{\max} \in \mathbb{N}_0$ die maximale Latenz des Protokolls, so gilt für die Zeit, bis das System eine Lastumverteilung vornehmen kann, t:

$$t = \max_{i \in \text{Nodes}} (\rho_i * T_{\text{global},i} * l_{\max})$$

Auf eine endliche Überlast kann das System also nach einer gewissen Zeit reagieren. Fällt jedoch ein Knoten während der Berechnung aus, so ist dies durch eine unendliche Überlast zu modellieren.

$$\lim_{\rho_i \to \infty} (t) = \infty$$

In diesem Fall kommt die Berechnung zum Erliegen, da der Server auf den überlasteten Knoten warten muss. Im Grid ist es allerdings nicht unwahrscheinlich, dass ein Knoten ausfällt, deshalb ist es notwendig, mit diesem Problem umgehen zu können.

Fällt also ein Knoten aus, muss ein anderer Ansatz gewählt werden.

Da die Knoten Berechnungsergebnisse speichern und untereinander kommunizieren müssen, kann ein Knoten nicht einfach ausgetauscht werden. Berechnungsergebnisse, die potenziell für spätere Berechnungen noch gebraucht werden, können zu einem späteren Zeitpunkt nicht mehr wiederhergestellt werden.

Um die Berechnungen zu wiederholen, müssten die Schleifensätze erneut aufgezählt werden. Deren Datenquellen können allerdings mittlerweile überschrieben worden sein, so dass schließlich das gesamte Programm neu gestartet werden muss. Diese Vorgehensweise bietet sich also als Lösung nicht an.

Lösungsvorschlag 1: Speichern versandter Ergebnisse auf den Knoten
Ein anderer Lösungsvorschlag wäre, versandte Ergebnisse zu speichern, um sie zu einem beliebigen späteren Zeitpunkt erneut verschicken zu können. Hier bietet sich eine *besitzergetriebene* Speicherform an, die das Datum als die Quelle einer

Abhängigkeit eines virtuellen Prozessors zu einem globalen Zeitschritt identifiziert.

Geht man davon aus, dass nur ein Knoten Um Speicherplatz zu sparen ist es sinnvoll, die versendeten Daten nur so lange vorzuhalten, bis das davon abhängige Berechnungsergebnis seinerseits versandt ist.

Ist das Berechnungsergebnis des Ziels der Abhängigkeit verschickt, sendet der Empfänger eine Nachricht zum Besitzer der Quelle. Dieser darf nun den Speicher freigeben.

Zur Verdeutlichung führen wir ein Beispiel an:
Seien auf Knoten A die Quelle einer Abhängigkeit, auf Knoten B das Ziel. Knoten C benötigt das Berechnungsergebnis von Knoten B.

$t = 0$
- A berechnet Ergebnis $e_{A,0}$, schickt es an B und speichert es.

$t = 1$
- B berechnet Ergebnis $e_{B,1}$, schickt es an C und speichert es.
- C empfängt $e_{B,1}$ und schickt eine Meldung an A.
- A löscht $e_{A,0}$.

$t = 2$
- C berechnet Ergebnis $e_{C,2}$.

In dem nun folgenden Szenario fällt Knoten B zur globalen Zeit 1 aus. Da eine genauere Ausführung den Rahmen der Arbeit sprengen würde, wird hier nur eine Skizze der einzelnen Aktionen angegeben:

$t = 0$
- A berechnet Ergebnis $e_{A,0}$, schickt es an B und speichert es.

$t = 1$
- B fällt aus und schickt keine Rückmeldung an den Server.

$t = 1$, später
- Am Server läuft der Timeout für B aus.
- B wird als ausgefallen markiert. Knoten D erhält eine besondere Auftragszuteilung für die Berechnung von
- $e_{B,1}$ vom Server.
- D fordert von A Daten zur Berechnung von $e_{B,1}$ an.
- D sendet Meldung an C, $e_{B,1}$ von D zu erwarten.
- A schickt gespeicherte Daten $e_{A,0}$ an D.
- D berechnet Ergebnis $e_{B,1}$, schickt es an C und speichert es als $e_{D,1}$
- Server führt Lastumverteilung durch: $e_{B,\geq 1}$ wird zu $e_{D,\geq 1}$

$t = 2$
- C berechnet Ergebnis $e_{C,2}$.

Der Server betrachtet einen Knoten als ausgefallen, wenn dieser eine gewisse Zeit lang keine Rückmeldung erbringt. Dieser Timeout ist so hoch anzusetzen, dass die geschätzte Wahrscheinlichkeit, dass die Verzögerung durch eine endliche Überlast zustandegekommen ist, gering ist. Andererseits bewirkt eine Erhöhung des Timeouts den zeitlichen Overhead, der durch den Ausfall eines Knotens produziert wird, da man damit rechnen muss, dass das System während dieser Zeit stillsteht.

Bei diesem Vorschlag verursacht eine zusätzliche Datenstruktur nennenswerten Speicheroverhead auf den Knoten. Außerdem ist zusätzliche Kommunikation zu erwarten. Im Gegenzug führt der Ausfall eines Knotens nicht mehr zu einem Abbruch der Berechnung, sondern lediglich zu einer Neuberechnung der verlorenen Daten.

Reagieren mehrere Knoten nicht mehr, kommt es auf den Fall an: Ist mit einem ausgefallenen Knoten auch der Speicherort seiner Quelle ausgefallen, führt dies immer noch zum Abbruch der Berechnung.
Dem kann abgeholfen werden, wenn man versandte Daten noch länger vorhält. Jedoch ist die absolute Sicherheit nur dann zu erreichen, wenn alles vorgehalten wird, bis alle davon abhängige Berechnungsergebnisse beim Server eingegangen sind. Hier ist Speicherverbrauch gegen Ausfallsicherheit abzuwägen.

Lösungsvorschlag 2: Speichern versandter Ergebnisse auf dem Server
Dieser Vorschlag nimmt an, dass der Server nicht ausfallen wird.

Jedes versandte Berechnungsergebnis wird nicht lokal auf dem Knoten vorgehalten, sondern an den Server geschickt und dort gespeichert. Kommt es zu einem Ausfall, gibt es zwei Möglichkeiten:

- Der Server schickt einen besonderen Berechnungsauftrag mitsamt aller benötigten Daten einem Knoten zu. Dieser führt die Berechnung durch und verbleibt der Besitzer der zugewiesenen Arbeitslast.
  Dies hat den Vorteil, dass ein definierter Knoten den Ausfall ersetzt. Die einzige Besonderheit für die anderen Knoten besteht darin, dass die *Prozessormap* (siehe Abschnitt 3.4.3) nicht zum aktuellen, sondern zu einem vergangenen Zeitpunkt verändert wird.

- Der Server berechnet die verlorenen Daten selbst und verteilt im nächsten Umverteilungsschritt die Last ganz regulär auf die verbleibenden Knoten. Allerdings ist das Berechnungsergebnis des "ausgefallenen" Zeitschritts auf keinem Knoten beheimatet. Dem muss gesondert Rechnung getragen werden.

Diese Methode hat den Vorteil, dass zwischen den Knoten kein aufwändiger Rollback implementiert und durchgeführt werden muss. Es kann eine beliebige Anzahl von Knoten ausfallen, ohne, dass dies zum Abbruch der Berechnung führt. Auch hier muss eine Datenstruktur eingeführt werden, die sich allerdings hier nur

beim Server Speicherplatz benötigt. Die zusätzliche Kommunikation tritt auch hier auf, mit dem besonderen Problem, dass der Server dadurch zum Falschenhals werden kann.

Praktikabilität Die meisten MPI-Implementierungen sehen Totalausfälle von Knoten nicht vor. Auch ist es nicht möglich, zur Laufzeit weitere Knoten hinzuzunehmen. Daher sind die momentanen Möglichkeiten eingeschränkt. Es gibt aber Bestrebungen, solche Funktionalität zur Verfügung zu stellen.

6.2.5 Peer-to-Peer-Implementierung

Jeden globalen Zeitschritt tritt eine Kommunikation zwischen dem Server und jedem Knoten auf. Dies kann bei einer großen Anzahl von Knoten zu einem Bottleneck beim Server führen.

Um dem Abzuhelfen, ist es denkbar, die Client-Server-Architektur zu Gunsten einer Peer-to-Peer-Struktur zu verlassen. In dieser kommunizieren die Knoten untereinander, tauschen aber zusätzlich auch die Lastinformation aus. Es gibt keine Instanz, die ein globales Wissen besitzt. Daher teilen die Knoten Last lokal unter sich auf. Die Lokalität ist hierbei entweder über den Abhängigkeitsgraph oder einem eigens erstellten Graph definiert. Ein Knoten besitzt die Lastinformation seiner Nachbarn und führt nach bestem Wissen mit diesen einen Lastausgleich durch. In einen solchen Lastausgleich sind alle direkten Vorgänger und Nachfolger im Abhängigkeitsgraph involviert.

6.2.6 Erweiterung zum Polyhedron-Modell

Eine Erweiterung auf das Polyhedron-Modell erfordert nur geringe Änderungen an den Datenstukturen. Einen wesentlicheren Einfluss hat diese Erweiterung auf die Vorverarbeitung (siehe Kapitel 2).

6.2.7 Maßnahmen bei Ausfall des Servers

Es ist denkbar, den Server zu duplizieren, um serverseitigen Ausfällen vorzubeugen. Dabei ist es nötig, die Server untereinander zu synchronisieren. Beim Ausfall eines Servers muss dies vom anderen Server bemerkt und die Knoten auf eine Kommunikation mit dem neuen Server umgestellt werden. Dieses Vorgehen erfordert, dass die Knoten in einem separaten Thread auf den Befehl einer solchen Umstellung lauschen. Außerdem müssen alle Kommunikationen mit dem Server mit Timeouts versehen werden, um die Kommunikation mit dem neuen Server reibungsfrei wiederaufnehmen zu können.

6.3 Persönliche Stellungnahme

Das vorgestellte Lastausgleichsprotokoll bietet eine performante Anpassung paralleler Programme an die Erfordernisse im Grid. Weil es durch die empirische Performanzmessung viele anderweitig unbekannte Faktoren erfasst, mag es im Grid Speedup bei akzeptabler Verlässlichkeit bringen, wo andere Verfahren keinen Erfolg haben.

Als Voraussetzung hierfür sehe ich die baldige Einbindung in LooPo, aber auch die Möglichkeit, mit unendlicher Überlast, also bei Totalausfall eines Knotens, ein Rollback durchführen zu können.

Die Einschränkung auf eine Client-Server-Architektur sehe ich für den Moment nicht als Problem an, da es in der momentanen Szene des Grid-Computing eher darauf ankommt, überhaupt Speedups zu erreichen. Dabei hat das globale Wissen des Servers eindeutig Vorteile, da bessere Algorithmen zur Lastumverteilung, die auch die Abhängigkeiten berücksichtigen, sehr viel einfacher zu entwickeln sind.

Literatur

[DGG+06] Jan Dünnweber, Sergei Gorlatch, Martin Griebl, Christian Lengauer, and Eduardo Argollo. Making a taskfarm component parallelize loops for the grid. In *Integrated Research in Grid Computing – CoreGRID Integration Workshop 2006*, pages 93–103, October 2006.

[GLW98] Martin Griebl, Christian Lengauer, and S. Wetzel. Code generation in the polytope model. In *IEEE PACT*, pages 106–111, 1998.

[Her] *hpcLine-Parallelrechner des Lehrstuhls für Programmierung an der Universität Passau.* http://www.infosun.fmi.uni-passau.de/infosun/programmierung/hpcline/.

[Len93] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[Loo] *LooPo - Loop Parallelization in the Polytope Model.* http://www.uni-passau.de/ loopo.

[MPI] *MPI Forum.* http://www.mpi-forum.org/.

[Sei04] Georg Seidel. Methods for adaptive tiling in the polyhedron model, 2004.