Adaptive Steuerung der Lastverteilung datenparalleler Anwendungen in Grid-Umgebungen

Dissertation

von
Sabina Rips

Schriftliche Arbeit zur Erlangung des Grades
eines Doktors der Naturwissenschaften

Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

Paderborn, Juni 2006

Meiner Tochter Saskia

Kurzfassung

Simulationstechniken haben einen immer höheren Stellenwert in Industrie und Wissenschaft. Ihre hohe Komplexität wird oft durch den Einsatz von Parallelrechnern bewältigt. Durch Simulation immer komplexerer Sachverhalte reichen selbst diese oftmals nicht aus, um effizient Berechnungen durchführen zu können. Umgebungen wie das Grid, in dem eine Vielzahl von leistungsstarken Ressourcen gekoppelt sind, stellen dafür eine geeignete Plattform bereit.

Die Simulationsprogramme, entwickelt für eine homogene Umgebung, wie sie ein Parallelrechner darstellt, weisen jedoch Defizite auf, wenn eine heterogene und dynamische Umgebung wie das Grid genutzt werden soll. Dieses betrifft insbesondere die Lastverteilung, die eine effiziente Ressourcenausnutzung und damit auch kurze Laufzeiten ermöglicht.

Das Ziel dieser Arbeit war daher, eine Lastverteilungssteuerung für Grid-Umgebungen zu entwickeln, die Anwenderprogramme und deren Lastverteilungswerkzeuge unterstützt, die Heterogenität und Dynamik des Grids zu nutzen.

Dafür wurden Methoden entwickelt und evaluiert, die durch Monitoring die Umgebung abtasten und analysieren und auf Basis dieser gewonnenen Informationen die Kommunikationsstruktur der Umgebung ebenso wie die Leistungsfähigkeit der Recheneinheiten erfassen und für Lastverteilung verwerten.

Dabei wird der Heterogenität der Recheneinheiten dadurch Rechnung getragen, dass ihre Leistungsfähigkeit zur Laufzeit gemessen und die optimale Last entsprechend dieser Ergebnisse ermittelt wird.

Die Kommunikationsverbindungen werden berücksichtigt, indem Last möglichst nur zwischen Recheneinheiten mit schnellen Verbindungen umverteilt wird, sofern dieses zu einem ausreichenden Lastgleichgewicht des kompletten Systems führt. Hierfür wird eine hierarchische Struktur benutzt, die zur Laufzeit ermittelt wird und den aktuellen Netzwerkzustand repräsentiert.

Sowohl der Aufbau dieser Hierarchie als auch Lastverteilungsentscheidungen werden verteilt getroffen, ohne Einsatz einer zentralen Instanz, was Ausfallsicherheit und Skalierbarkeit unterstützt.

Eine Umsetzung und Validierung des Konzepts wurde durch das Werkzeug *mLB* (meta Load Balancer) realisiert.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

In den letzten Jahren haben Simulationstechniken stark an Bedeutung gewonnen [LÖ1]. Die Minimierung der Entwicklungskosten eines Produktes ist nur einer von vielzähligen Gründen. Oftmals muss die Funktionsfähigkeit eines Produktes simulativ geprüft werden, bevor die Produktion eines Prototypen beginnen kann. Hierzu zählt z. B. die Motorentwicklung. Bei jeder geplanten Änderung der Brennraumgeometrie können Motorinnenströmungen und Verbrennungsprozesse simuliert werden, anstatt jedesmal einen kompletten Motor produzieren und testen zu müssen. Neben Kostenersparnis wird hier auch Zeitersparnis erreicht, wichtig, um heutzutage konkurrenzfähig zu bleiben.

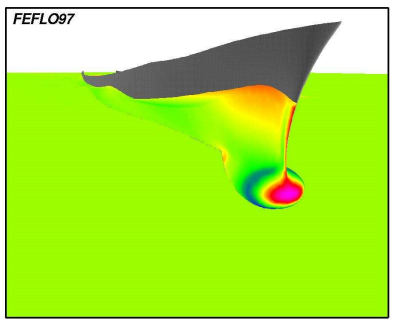
Die Entwicklung von Schiffen, Flugzeugen oder auch Raumfahrzeugen ist ohne vorherige Simulation undenkbar geworden.

In anderen Bereichen ist die Simulation die einzige Möglichkeit, Phänomene zu erforschen. Vorhersagen über das Wetter oder aber auch von Naturkatastrophen, wie z. B. die Auswirkungen von Tsunamis und Gletscherschmelzen, gehören hierzu. Das Einstein-Jahr 2005 brachte viele Simulationen aus dem Bereich der numerischen Relativität hervor. Auf dem Gebiet der Astrophysik sind schwarze Löcher ebenfalls nicht ohne Simulationen zu erkunden. Beispiele von Simulationen zeigt Abbildung 1.1 [1].

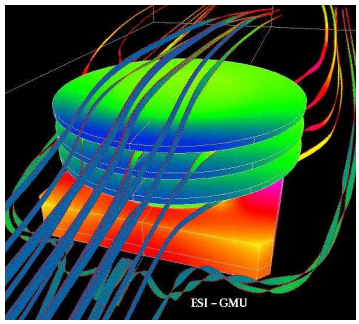
Was all diesen Anwendungsbeispielen gemeinsam ist, ist ihre hohe Komplexität, die eine enorme Leistungsfähigkeit der Rechenressource erfordert. Es müssen Gleichungen mit Millionen von Unbekannten gelöst werden, deren Berechnungen mehrere Tage beanspruchen können [Tur96].

Sequentielle Software stößt hier schnell an ihre Grenzen. Die Kapazität des Hauptspeichers reicht nicht aus und die Simulationszeiten werden unakzeptabel hoch. Daher gibt es zunehmend parallele Anwendungen, die besonders auch auf massiv parallelen Systemen arbeiten.

[1] mit freundlicher Genehmigung von Prof. Dr. R. Löhner



Strömungssimulation eines Schiffes



Strömungssimulation einer Kühlflüssigkeit um einen Chip

Abbildung 1.1: Beispiele von Strömungssimulationen [Loe]

Die immer komplexer werdenden Anwendungen stehen der Entwicklung von zugreifbaren Ressourcen gegenüber. Diese werden immer leistungsfähiger und verfügbarer. Betrachtet man z. B. die Rechenkapazitäten von Ein- und Mehrprozessormaschinen, so erkennt man ihren hohen Leistungszuwachs: Zirka alle fünf Jahre verdreifacht sich die Leistungsfähigkeit von Prozessoren und Speichern.

Aber nicht nur die Leistungsfähigkeit der einzelnen Maschinen ist angestiegen, sondern auch ihre Verfügbarkeit. Viele Institutionen besitzen heutzutage eigene leistungsstarke Rechenressourcen, die vor wenigen Jahren nur in großen Rechenzentren vorzufinden waren. Dieses hohe Potenzial an Rechenkapazität hat viele Bereiche des Hochleistungsrechnens vorangetrieben. Es ermöglicht Rechnungen auf lokalen Maschinen, die sonst nur durchzuführen waren, wenn Maschinen von Hochleistungsrechenzentren genutzt werden konnten.

Ein großer Bereich des Hochleistungsrechnens befasst sich mit Struktur- und Strömungssimulationen. Diese Anwendungen, die vor Jahren nur auf großen Parallelrechnern bearbeitet werden konnten, können heutzutage zum Teil schon auf einzelnen Maschinen berechnet werden. Um jedoch noch komplexere Simulationen durchführen zu können, reichen keineswegs einzelne Systeme aus. Selbst große Parallelrechner reichen oftmals nicht aus, um hochkomplexe Aufgaben zu lösen.

Um dieses zu ermöglichen, müssen Hochleistungsrechner gekoppelt werden. Eine Umgebung, die dieses leistet, ist das Grid. Das Grid ist definiert als eine Hochleistungsressource, die durch die Kopplung mehrerer Rechenressourcen zustande kommt [FK03].

Das Vorhandensein des Grids schließt jedoch keinesfalls eine effiziente Nutzung mit ein. Ohne Unterstützung können parallele Anwendungen diese Hochleistungsressource meist nicht angemessen nutzen. Entwickelt für ein homogenes System, wie es Parallelrechner darstellen, sind diese Anwendungen im Allgemei-

nen nicht in der Lage, die Heterogenität des Grids zu beachten und diese für eine effiziente Berechnung auszunutzen.

Die Motivation, das Grid zu nutzen, liegt für den FEM[2]-Anwender in zwei Hauptzielen begründet:

- Die Machbarkeit von Berechnungen: Durch Einsatz des Grids kann die Komplexität der Anwendung um Faktoren erhöht werden.
- Schnellere Ausführungszeiten: Durch Nutzung des Grids können Laufzeiten drastisch reduziert werden.

Für beide Aspekte bedarf es einer Unterstützung der Anwendung, insbesondere im Bereich Lastverteilung. Eine intelligente Lastverteilung gewährleistet die Machbarkeit von Ausführungen dadurch, dass Prozesse nicht vorzeitig aufgrund des Fehlens von Ressourcen abgebrochen werden müssen. Zudem werden durch eine optimale Verteilung der Rechenlast schnellere Ausführungszeiten erreicht.

1.2 Ziel der Arbeit

Das Ziel von Lastverteilungsmethoden für Struktur- und Strömungssimulationen ist im Allgemeinen die gleichmäßige Aufteilung des Simulationsgebietes auf die Prozessoren. Diese Gleichverteilung der Rechenlast ist angemessen für homogene Umgebungen, wie sie z. B. bei Parallelrechnern zu finden ist. Das Grid hingegen ist gekennzeichnet durch die unterschiedliche Leistungsfähigkeit der eingesetzten Ressourcen. Um dieser Heterogenität gerecht zu werden und eine effiziente Lastverteilung zu ermöglichen, bedarf es einer Unterstützung der Anwendung, die die Charakteristika des Grids berücksichtigen.

Ziel ist es aber nicht, neue Lastverteilungsmethoden zu entwickeln, sondern vorhandene, bewährte Strategien dabei zu unterstützen, ihre Berechnungen an die Umgebung anzupassen.

Dafür müssen drei Aspekte betrachtet werden:

- die Charakteristika der parallelen Anwendungen, bzw. die Merkmale der zu verteilenden Last und ihre Abhängigkeiten voneinander,
- die Lastverteilungsmethoden in Abhängigkeit von der Klasse der Anwendungen und
- die Charakteristika der Grid-Umgebung.

Abbildung 1.2 zeigt die Wechselwirkung dieser drei Aspekte. Die Umgebung beeinflusst das Laufzeitverhalten der Anwendung. In einer leistungsstarken Umgebung können kürzere Laufzeiten als in einer leistungsschwächeren erwartet werden. Die Anwendung setzt Lastverteilung ein, um optimierte Laufzeiten zu

[2]Finite Element Method, verbreitete Methode in der Strömungssimulation

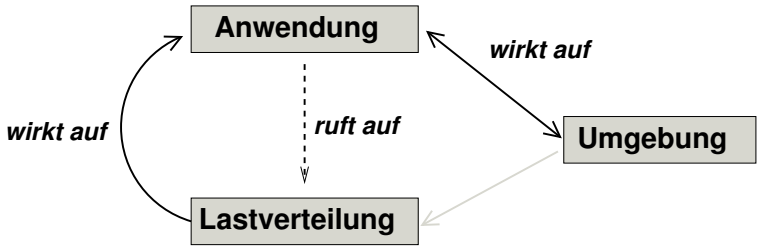


Abbildung 1.2: Wechselwirkung der Hauptkomponenten

erreichen.
Wurde Lastverteilung bisher unabhängig von der Umgebung betrachtet, so gilt es nun, eine Verbindung zwischen Lastverteilung und Umgebung herzustellen, um durch eine adäquate Lastverteilungssteuerung eine hohe Effizienz der Anwendung zu erreichen.

1.3 Aufbau der Arbeit

Die Grundlagen für die drei Hauptkomponenten Anwendung, Lastverteilung und Umgebung sind Thema von Kapitel 2. Ausgehend von deren Merkmalen, werden in Kapitel 3 Anforderungen für eine geeignete Lastverteilungssteuerung abgeleitet. Das Lösungskonzept ist Thema von Kapitel 4, das die entwickelten Strategien zur adaptiven Steuerung von Lastverteilung vorstellt. Die Umsetzung dieser Konzepte, die durch das Werkzeug *mLB* realisiert wurde, ist Inhalt von Kapitel 5. Kapitel 6 zeigt die Ergebnisse der Validierung von *mLB*. Ein Resümee bildet in Kapitel 7 den Abschluss dieser Arbeit.

Kapitel 2

Begriffsklärung/Grundlagen

2.1 Begriffsklärung

In diesem Abschnitt werden grundlegende Begriffe erklärt.

Bandbreite: Bandbreite ist die Differenz zwischen der höchsten und niedrigsten Frequenz einer Verbindung, d. h. die Breite ihres reservierten Frequenzbandes. Meist ist hiermit jedoch die Daten(transfer)rate gemeint, die angibt, wieviele Daten pro Sekunde über eine Netzwerkverbindung übertragen werden können.

Effizienz: Effizienz ist das Verhältnis eines in definierter Qualität vorgegebenen Nutzens zu dem Aufwand, der zur Erreichung des Nutzens nötig ist. Ein effizientes Verhalten führt daher wie auch ein effektives Verhalten zur Erzielung eines Nutzens, hält aber im Unterschied zu diesem den dafür notwendigen Aufwand möglichst gering.

Last: Last bezeichnet hier die Rechenlast der Anwendung, die durch die Problemgröße gegeben ist und auf die sich der Lastverteiler der Anwendung bezieht. Sie besteht aus mehreren Lasteinheiten. Bei gitterbasierten Anwendungen bilden die Gitterzellen die Lasteinheiten.

Latenz: Latenz gibt die Zeitspanne an, die ein Datenpaket vom Sender zum Empfänger benötigt.

PE: In dieser Arbeit wird vorausgesetzt, dass auf jeder Recheneinheit ein Anwendungsprozess bearbeitet wird. Damit bilden sie eine Einheit. Für die einfachere Lesbarkeit wird hier der Begriff „PE“ für „Anwendungsprozess, der auf einer Recheneinheit bearbeitet wird“ verwendet.

Recheneinheit: eine CPU mit dazugehörigen Komponenten.

Rechenknoten: System, bestehend aus einer oder mehreren Recheneinheiten, wie z. B. Multiprozessor-Architekturen.

Skalierbarkeit: Eine Anwendung gilt allgemein als skalierbar, wenn größere parallele Konfigurationen in der gleichen Zeit proportional größere Probleme lösen können als kleinere Probleme auf kleineren Konfigurationen [KDF+03]. Ein gut skalierbares paralleles Programm benötigt bei der doppelten Anzahl von Prozessoren die Hälfte der Rechenzeit.
In dieser Arbeit wird unter Skalierbarkeit von Zusatzkosten verstanden, dass die Gesamtzusatzkosten maximal proportional zu der Anzahl an Prozessoren wachsen.

2.2 Grundlagen

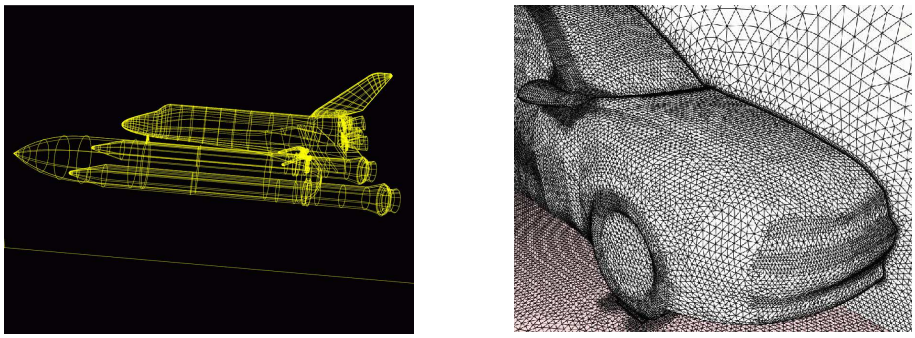
Im folgenden Abschnitt werden zunächst die Prinzipien der Zielanwendung beschrieben. Die Charakteristika des Grids zusammen mit den Zielsetzungen der Lastverteilung führen in Abschnitt 3.1 zu einem Anforderungskatalog, dessen Punkte erfüllt werden müssen, um adäquate Lastverteilung für Anwendungen, wie z. B. Strömungssimulationen, durchführen zu können.

2.3 Charakteristika der Anwendungen

Numerische Strömungssimulation (Computational Fluid Dynamics, CFD) ist die systematische Anwendung von Rechensystemen und numerischen Lösungstechniken auf mathematische Modelle, die entworfen wurden, um Strömungsphänomene zu beschreiben und zu simulieren.

CFD gehört zu einem Bereich von Simulationstechniken, die von Ingenieuren und Physikern genutzt werden, um das Verhalten eines ingenieurwissenschaftlichen Produkts oder einer physikalischen Situation vorherzusagen oder zu rekonstruieren.

In der numerischen Strömungssimulation sind **Finite Elemente Verfahren** (Finite Element Methods, FEM) eine weit verbreitete Technik. Dabei wird ein geometrischer Raum durch die Benutzung eines Gitters diskretisiert, indem das Grundgebiet in einfache Teilgebiete, die so genannten Gitterzellen, zerlegt wird. Für diese Teilgebiete werden dann jeweils eine oder mehrere numerische Gleichungen gelöst. Abbildung 2.1 [1] zeigt die Oberflächengitter eines Space-Shuttles und die eines Autos.



Oberflächengitter eine Shuttles **Oberflächengitter eines Autos**

Abbildung 2.1: Beispielgitter [Loe]

[1] mit freundlicher Genehmigung von Prof. Dr. R. Löhner

Die Simulation beginnt, indem für jede Zelle Berechnungen durchgeführt werden. Das geschieht solange, bis eine vorgegebene Restabweichung eingehalten oder unterschritten wird.

Handelt es sich um eine **adaptive Anwendung**, so wird das Gitter abhängig von dem zuvor erhaltenen Ergebnis an markanten Punkten (z. B. Stoßkanten) verfeinert und auf diesem veränderten Gitter weitergearbeitet (siehe Abbildung 2.2).

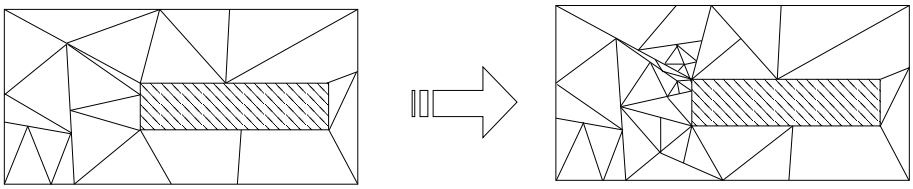


Abbildung 2.2: Beispiel einer Gitterverfeinerung

Gitteränderungen können aber auch durch sich verändernde Geometrien erfolgen. So ändert sich der Innenraum eines Zylinders durch die Bewegung des Kolbens. Im Ansaugtakt wird der Raum immer größer, da der Kolben sich nach unten bewegt. Damit erhöht sich die Anzahl an Zellen mit jedem Grad Kurbelwellenwinkel. Am unteren Totpunkt kann die Zellanzahl durchaus auf das Doppelte angestiegen sein [JLK+99]. Abbildung 2.3 zeigt das Gitter einer Motorgeometrie (Zylinder, Ein- und Auslasskanal) zur Simulation von Strömungs- und Verbrennungsprozessen, links am oberen Totpunkt, rechts am unteren.

Abbildung 2.3: Beispiel einer Gitteränderung durch Kolbenbewegung

Diese Veränderung der Zellanzahl führt auch zur Veränderung der Rechenlast. Es müssen mehr numerische Gleichungen gelöst werden. Das Datenaufkommen auf den betroffenen Recheneinheiten steigt erheblich. Aus anfänglich

Hunderttausenden können Millionen von Gleichungen werden [Tur96]. Der Speicherplatzbedarf solch einer Anwendung übersteigt dann oftmals die vorhandenen Ressourcen. Ist die Rechenplattform nicht leistungsfähig genug, muss ein Abbruch der Berechnung erfolgen.

Parallele Software hingegen bietet die Möglichkeit des Einsatzes von mehreren Rechenressourcen, um diese Rechenlast zu bewältigen. Insbesondere massiv parallele Systeme mit mehreren hundert Rechenknoten bilden eine geeignete Plattform für diese Anwendungen.

2.3.1 Parallele Anwendungen

Für die parallele Berechnung muss das Gitter auf die Prozessoren verteilt werden. Dafür wird es in Teile zerlegt (**Gebietszerlegung**). Auf jeder dieser Teile (Partitionen) wird dann die Berechnung durchgeführt, unterbrochen durch den Informationsaustausch zwischen den Prozessoren. Abbildung 2.4 zeigt den prinzipiellen Ablauf einer parallelen FEM-Anwendung.

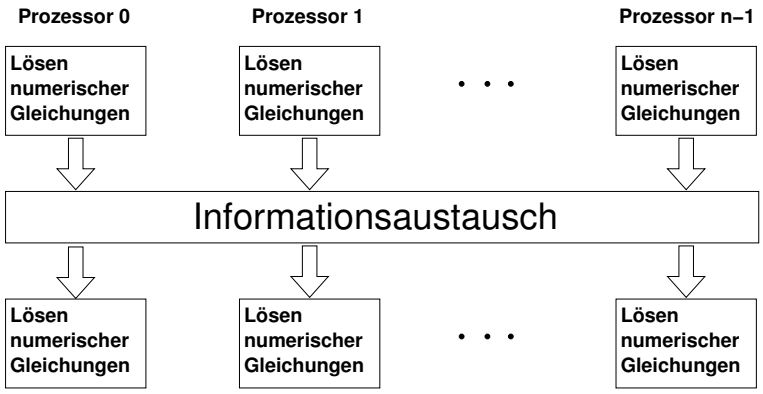


Abbildung 2.4: Ablauf einer parallelen FEM-Anwendung

Die Randzellen jeder Partition haben sogenannte Halozellen. Das sind Kopien der angrenzenden Nachbarzellen, die auf einer entfernten Recheneinheit liegen. Dadurch liegen die Informationen über diese Zellen bereits während der Berechnung vor, d. h. es muss bei Bedarf nur auf die Halozellen auf der lokalen Recheneinheit zugegriffen werden, anstatt eine Kommunikation mit der entsprechenden entfernten Recheneinheit durchzuführen. In jeder Iteration werden die Informationen aller Randzellen einer Partition, deren Kopien sich als Halos auf einer bestimmten Recheneinheit befinden, gemeinsam an die entsprechenden Recheneinheiten geschickt. So wird für jeden benachbarten Prozessor nur eine Nachricht benötigt. Dennoch bedeutet das Vorhandensein vieler Randzellen ein hohes Kommunikationsaufkommen, da die Nachrichtengröße abhängig von der Anzahl der zu verschickenden Zellen ist.

Ändert sich das Gitter, so erfolgt dies zum Ende der Berechnung hin oftmals nur in einigen wenigen Regionen (z. B. Stoßkanten oder kleinerer Raum im Kompressionstakt) und damit auch auf einigen wenigen Recheneinheiten.

Die Anzahl an Zellen und somit die Rechenlast auf jeder Recheneinheit entwickelt sich dadurch unterschiedlich. Dieses führt zu einem Ungleichgewicht, das die Effizienz der Berechnung stark beeinträchtigen kann. Um Effizienzverlusten vorzubeugen, wird daher die Rechenlast neu verteilt, also Lastverteilung durchgeführt.

2.4 Konzepte der Lastverteilung

Lastverteilung dient dazu, verteilte Anwendungen zu optimieren. Dabei steht im Allgemeinen die Minimierung der Laufzeit einer verteilten Anwendung im Vordergrund. Andere Aspekte, wie z. B. Speicherauslastung, können ebenfalls Ziele von Lastverteilung sein.

Um die Optimierung der Gesamtlaufzeit zu erreichen, ist das Hauptziel von Lastverteilung, gleiche Rechenzeiten aller Prozesse durch angemessene Verteilung der Rechenlast zu erzielen. Dieses Ziel basiert auf der Grundlage, dass der Prozess mit der höchsten Laufzeit die Gesamtlaufzeit bestimmt.

2.4.1 Lastverteilungsgrundlagen

Lastverteilung kann grundsätzlich zu zwei unterschiedlichen Zeitpunkten ausgeführt werden:

- zu Beginn einer Berechnung (initiale Partitionierung) und
- während einer Berechnung (Lastrepartitionierung)

Bei der **initialen Partitionierung** werden die zu berechnenden Daten in Teile zerlegt, um sie dann anschließend über die Recheneinheiten zu verteilen. Im Allgemeinen wird dieses zentral, d. h. von einer einzelnen Recheneinheit, durchgeführt. Eine gute initiale Partitionierung ermöglicht den Beginn einer effizienten Berechnung.

Wird während einer Berechnung ein Ungleichgewicht, z. B. durch Verfeinerung des Gitters, erzeugt, sind große Effizienzverluste zu erwarten. In diesem Fall muss die Last balanciert werden. Das kann entweder durch

- Neu-Partitionierung oder
- Lastumverteilung

erreicht werden.

Bei der **Neu-Partitionierung** werden die Daten auf einer Recheneinheit gesammelt, um anschließend wieder, wie bei der initialen Partitionierung, neu verteilt zu werden. Dieser Ansatz hat sich insbesondere dann bewährt, wenn z. B. durch programmiertechnische Einschränkungen kein Verschieben der Last von einem Prozessor zu einem anderen möglich ist. Dieses gilt z. B. für Programme, die in Fortran 77 geschrieben sind und keine dynamische Speicherverwaltung unterstützen.

Bei einer **Lastumverteilung** wird während des Simulationsprozesses die Last zu anderen Prozessoren verschoben (Lastmigration). Erfolgt diese Umverteilung dynamisch zur Laufzeit und ist sie abhängig von der Lastentwicklung während der Berechnung, so wird dies als **dynamische Lastverteilung** bezeichnet.

Entstehen Lastunterschiede durch Veränderungen, die zur Laufzeit auftreten, wie z. B. die Veränderung der Zellenanzahl bei Gitterverfeinerungen, dann muss dynamische Lastverteilung eingesetzt werden, um eine effiziente Berechnung zu erzielen. In dieser Arbeit werden nur Methoden der dynamischen Lastverteilung betrachtet, da nur diese für komplexe, dynamische Simulationsanwendungen geeignet sind.

Der Lastverteilungszyklus

Soll Lastverteilung stattfinden, so erfolgt dieses in mehreren Schritten. Diese bilden wiederum einen Lastverteilungszyklus. Ein Lastverteilungszyklus besteht aus einer Entscheidungsphase und einer Durchführungsphase.
In der **Entscheidungsphase** werden folgende Fragestellungen beantwortet:

- **Wann** wird Lastverteilung durchgeführt?
- **Wieviel** Last muss migriert werden?
- **Welche** Last muss migriert werden?
- **Wohin/Woher** muss Last migriert werden?

Hierbei wird oftmals nur der erste Teil, d. h., wann wird Lastverteilung durchgeführt, von der Endanwendung bearbeitet. Die anderen Bereiche werden dann von Lastverteilungswerkzeugen übernommen. Hier stehen dem Benutzer eine Vielzahl von mächtigen Werkzeugen zur Verfügung (siehe Kapitel 2.4.4).

In der **Durchführungsphase** erfolgt die Lastmigration. Daten werden verschoben und müssen auf einer anderen Recheneinheit integriert werden. Diese Migration erfordert einen erheblichen Eingriff in die Daten und Datenstrukturen der Anwendung. Daher kann die Migration nur von der Anwendung selbst geleistet werden.

2.4.2 Lastverteilung für gitterbasierte Anwendungen

In dieser Anwendungsklasse sind **Gebietszerlegungsmethoden** weit verbreitet: Das Gesamtgebiet wird in Teilgebiete zerlegt, auf denen dann parallel die Berechnung erfolgt. Abbildung 2.5 zeigt die Zerlegung eines Gebietes in vier Teilgebiete.

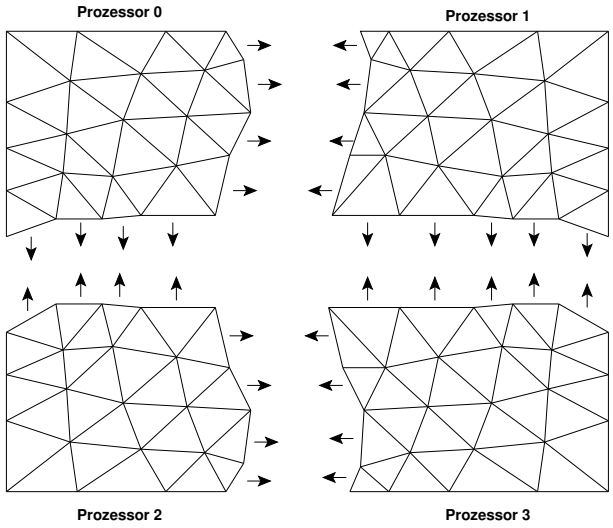


Abbildung 2.5: Teilpartitionen auf vier Prozessoren

Um adäquate Lastverteilungsmethoden entwickeln zu können bzw. vorhandene Werkzeuge auswählen zu können, müssen die Charakteristika der zu verteilenden Last und ihre Abhängigkeiten untereinander bekannt sein. Im Folgenden wird daher die Arbeitsweise von parallelen Finite-Elemente-Anwendungen beschrieben.

Die Gitterstruktur einer FEM-Anwendung wird auf einen Graphen, den sogenannten **dualen Graphen**, abgebildet (siehe Abbildung 2.6). Dabei bildet jede Gitterzelle einen Knoten, Knoten benachbarter Zellen werden durch Kanten verbunden. Dieser Graph spiegelt die Berechnungen, dargestellt durch die Knoten des Graphen, und die Kommunikation, dargestellt durch die Kanten des Graphen, wider.

Bei einer parallelen Anwendung existieren sogenannte Schnittkanten zwischen den einzelnen Teilgraphen. Diese sind Kanten, die über Prozessorgrenzen hinweg führen und notwendige Kommunikation zwischen den Prozessoren repräsentieren. Sei also $G = (V, E)$ dualer Graph, V die Menge der Knoten, E die Menge der Kanten, $e = (v, w) \in E$ eine Kante von Knoten v zu Knoten w, $v, w \in V$, PE_i, PE_j PEs, v wird von PE_i berechnet, w von PE_j, $i \neq j$.

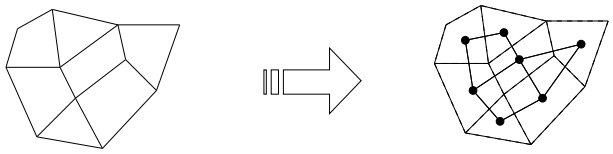


Abbildung 2.6: Abbildung eines Beispielgitters auf den dualen Graphen

Dann ist $e = (v, w)$ Schnittkante. Die Anzahl an Schnittkanten zwischen zwei PEs wird als Schnitt bezeichnet. Abbildung 2.7 zeigt dazu ein Beispiel.

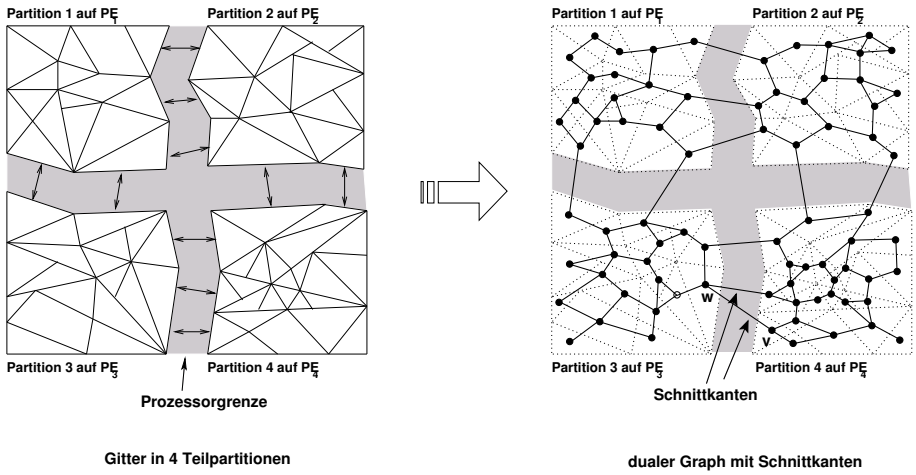


Abbildung 2.7: Beispiel von Schnittkanten

Ein wichtiges Merkmal der Arbeitsweise von FEM-Anwendungen ist der rege Datenaustausch zwischen benachbarten Elementen und, abhängig vom mathematischen Verfahren, dem geringen Austausch globaler Informationen unter den einzelnen Teilgebieten. Jedes Element erhält Daten von seinem benachbarten Element, zusätzlich werden in manchen Methoden wenige globale Daten benötigt. Bei iterativen Lösern zum Beispiel berechnet jeder Prozessor die Norm des Residuums über seinen Teil des Gitters. Alle Prozessoren benötigen den minimalen Wert von diesem, um zu entscheiden, ob die Lösung konvergiert (näheres siehe [FWM94]). Dieses bedingt globalen Datenaustausch.

Da eine hohe Anzahl an nötigem Datenaustausch zwischen den Elementen erforderlich ist, sollte dieser so wenig wie möglich über Prozessorgrenzen hinweg geschehen. Neben dem allgemeinen Ziel von Lastverteilung, die Last gleichmäßig zu verteilen, führt der letzte Aspekt zu dem weiteren Ziel, die Anzahl an Schnittkanten, d. h. den Schnitt, zu minimieren.

Um diesen Anforderungen gerecht zu werden, bedarf es komplexer graphentheoretischer Kenntnisse, und aufwändiger Umsetzung zu einer lauffähigen Anwendung. Diese Komplexität führte dazu, dass die Entwicklung von Lastverteilungsmethoden für gitterbasierte Anwendungen oftmals nicht vom CFD-Entwickler selbst geleistet wurde, sondern in einem eigenständigen Forschungsgebiet Fuß fasste. Durch die dualen Graphen wurde eine einheitliche Datenstruktur geschaffen, die die Schnittstelle zu den Lastverteilern bildet und auf denen diese arbeiten können.

2.4.3 Integration von Lastverteilung

Während der Laufzeit findet regelmäßig ein Datenaustausch zwischen den Zellen statt. Diese Zeitpunkte stellen Synchronisationspunkte der Anwendung dar, an denen Kommunikation zwischen den Prozessoren stattfindet. Da alle an Lastverteilung beteiligten Prozessoren zur gleichen Zeit diese initiieren müssen, werden dafür die Synchronisationspunkte genutzt (siehe Abbildung 2.8).

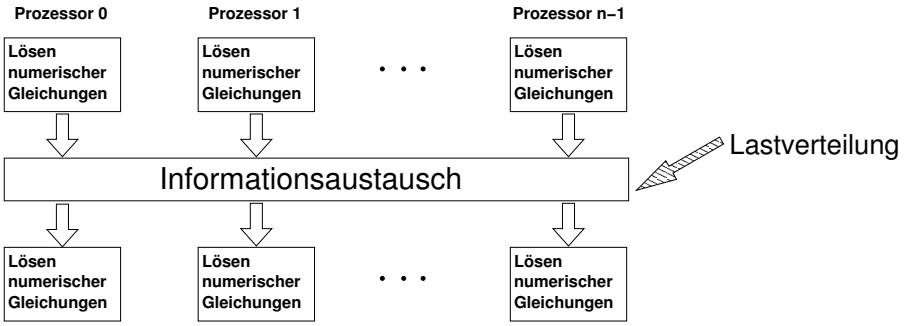


Abbildung 2.8: Ablauf einer parallelen FEM-Anwendung mit Lastverteilung

2.4.4 Lastverteilungswerkzeuge

Die Aufgabe der Lastverteilungswerkzeuge besteht darin, basierend auf dem dualen Graphen, zu entscheiden, welche Knoten zu welcher Partition verschoben werden müssen, um ein Lastgleichgewicht (wieder-)herzustellen. Die eigentliche Migration der Gitterzellen übernimmt die Anwendung selbst, da dieses ein Zugriff auf die Daten der Anwendung erfordert.

Bei der Auswahl von Lastverteilungsalgorithmen werden nur parallele Algorithmen betrachtet, da bei Einsatz von massiv-parallelen Systemen sequentielle Methoden nicht geeignet sind. Für eine sequentielle Lastverteilung ist es erforderlich, zunächst die Daten auf einem zentralen Prozessor einzusammeln, neu zu partitionieren, um sie dann wieder neu zu verteilen. Dieses Vorgehen ist nicht anwendbar in der Gridumgebung, da

- es einen beträchtlichen sequentiellen Bottleneck gibt (da der Geschwindigkeitszuwachs nach Amdahl [Amd67] vor allem vom sequentiellen Anteil des parallelen Problems abhängt),
- oftmals nicht genügend Hauptspeicher auf einer Hostmaschine verfügbar ist, um das gesamte verteilte Gitter zu speichern und
- hohe Kosten beim Kommunizieren mit dem gesamten Gitter entstehen
- und damit auch die notwendige Skalierbarkeit gefährdet ist.

Die Hauptziele der Lastverteilung für gitterbasierte Anwendungen sind:

1. Gleichmäßige Verteilung der Last auf alle Prozessoren, so dass alle Prozessoren zur gleichen Zeit an einem Synchronisationspunkt ankommen.
2. Minimierung der Interprozessor-Kommunikation, so dass Zusatzaufwand vermieden wird.

Bezugnehmend auf den dualen Graphen (siehe Abschnitt 2.4.2), bedeutet dies:

1. Teilung des Graphen in Partitionen mit der gleichen Anzahl von Knoten
2. Minimierung des Schnitts

Um dieses NP-vollständige Problem zu lösen, wurden viele Heuristiken und leistungsstarke Methoden entwickelt [BS94, KK97, WCE+95, SKK03, WLR93, OA02, FWM94, Gro91].

Eine Vielzahl von ihnen zielt darauf ab, das Gitter zu Beginn der Berechnung sequentiell in gleiche Teile zu zerlegen. Die so entstandenen Teilpartitionen werden dann auf die Prozessoren verteilt.

Dynamisches Verhalten einer Anwendung, z. B. durch Gitterverfeinerung oder aber Strukturänderungen, erfordert jedoch auch eine Umverteilung der Last zur Laufzeit. Dieses wird durch dynamische Lastverteilungsmethoden [DMP97, WCE97] erreicht. Anstatt die Teilgitter auf einem Prozessor zu sammeln und das daraus entstehende Gesamtgitter wiederum neu zu zerlegen, werden bei diesen Verfahren einzelne Gitterzellen von Prozessor zu Prozessor verschoben [SSLK00]. Neben der gleichen Lastmenge je Prozessor und der Schnittkantenminimierung wurden nun auch Lastumverteilungskosten betrachtet. Ein weiteres Ziel dieser Verfahren liegt daher in der Minimierung der notwendigen Migrationskosten. Dieses wird realisiert durch Minimierung der Anzahl zu verschiebender Gitterzellen zusammen mit dem Verschieben zwischen möglichst wenigen Prozessoren.

So ist das Ziel von Diffusionsverfahren [Cyb89, HB99], den Unterschied zwischen der Original-Partition und der neuen Partition so minimal wie möglich zu halten. Dieses wird dadurch erreicht, indem sie inkrementelle Veränderungen

an der Partitionierung vornehmen, um ein Gleichgewicht herzustellen.

Multilevel-Verfahren [Bar95, HB93, KK98, SKK97, WC01] sind heutzutage weitverbreitete Methoden hoher Güte. Sie bestehen aus mehreren Phasen. In der ersten Phase wird der duale Graph durch Zusammenfassen von Knoten vergröbert und das Problem damit verkleinert. Auf dieser reduzierten Problemgröße wird dann der entsprechende Algorithmus, um geeignete Partitionen zu berechnen, ausgeführt. In der letzten Phase wird der Graph bis zu seiner Originalgröße wieder verfeinert. Dabei werden bestimmte Optimierungsvorschriften beachtet.

Jostle [jos] und ParMETIS [par], die populärsten Werkzeuge in diesem Gebiet, bieten neben dem Einsatz von Multilevel-Verfahren auch die Möglichkeit, unterschiedlich große Teilpartitionen zu generieren. Sie ermöglichen es der FEM-Anwendung, Werte, die sich auf die Partitionsgrößen beziehen, bei ihrem Aufruf mit anzugeben. Diese Werte beinhalten den Anteil an Gesamtgitterzellen, den jeder Prozessor erhalten soll. Jostle bzw. ParMETIS erzeugen daraufhin die gewünschten Partitionsgrößen. Die Benutzung von Zoltan [BDH$^+$99] gewährleistet ebenso diese Funktionalität. Allerdings benutzt Zoltan als Lastverteilungsmethoden unter anderem Jostle und ParMETIS.

Jostle bietet noch eine weitere, interessante Funktionalität für heterogene Umgebungen. Es eröffnet die Möglichkeit, heterogene Netzwerke mit zu berücksichtigen. Dafür erwartet es einen gewichteten Prozessorgraphen, der das Kommunikationsnetzwerk repräsentiert. Ziel ist es nun, den dualen Graphen auf diesen Prozessorgraphen abzubilden. Dafür wird die in Jostle verwendete Multilevel-Methode in der letzten Phase erweitert. Jostle optimiert in dieser Phase den Schnitt durch Austausch von Knoten zwischen Partitionen (basierend auf dem Kernighan-Lin-Algorithmus [KL70]). Wenn es durch diesen Austausch zu einer Verringerung des Schnitts kommt, so werden diese Knoten vertauscht. In der modifizierten Version entscheidet das Gewicht der betroffenen Kante im Prozessorgraph, ob ein wirklicher Gewinn erreicht werden kann.
Soll diese Funktionalität von Jostle benutzt werden, so bleibt es allerdings der Anwendung überlassen, den Prozessorgraphen zu erstellen.

Hier wird deutlich, dass die Betrachtung der Umgebung eine Grundvoraussetzung für eine effiziente Lastverteilung darstellen kann. Daher werden im folgenden Abschnitt die Charakteristika der Zielumgebung aufgezeigt.

2.5 Charakteristika der Zielumgebung

2.5.1 Entwicklung des Gridcomputings

Schon lange besteht der Wunsch, Rechenressourcen zu koppeln. Das Verschalten von Workstations, PCs, SMPs, etc., um nicht genutzte Ressourcen einzusetzen und die Verfügbarkeit von hohen Rechenkapazitäten bereit zu stellen, ist heut-

zutage weit verbreitet. Unterstützt wurde dieses Bestreben von PVM (Parallel Virtual Machine) [pvm], einem Software Paket, das die Verschaltung von und den Nachrichtenaustausch zwischen Rechnern, auch unterschiedlicher Architektur, vereinfacht. PVM wurde durch MPICH [mpia], einer Implementierung des Message Passing Standards MPI (Message Passing Interface) [mpib], weitestgehend abgelöst.

In den 1990er Jahren begannen Hochleistungsrechenzentren, Parallelrechner zu verschalten, um die Komplexität aufwändiger Berechnungen noch steigern zu können. So entstand das Metacomputing [Gen99], wie es z. B. am Höchstleistungsrechenzentrum Stuttgart durchgeführt wurde [Res01]. Zusammen mit anderen Hochleistungsrechenzentren, wie z. B. das Leibniz Rechenzentrum München oder das Forschungszentrum Jülich, wurden in Deutschland viele Projekte durchgeführt [EHR+98], bei denen lokale Maschinen wie die Cray T3E, einer der schnellsten Parallelrechner ihrer Zeit, mit weiteren Hochleistungsrechnern über WANs gekoppelt wurden. In anderen Projekten [RRS99] wurden diese nationalen Rechner auch mit Maschinen aus den USA verbunden, um hochkomplexe parallele Rechnungen, wie z. B. die Simulation vom Wiedereintritt eines Shuttles in die Erdatmosphäre mit der Software URANUS [RBB+98], durchzuführen.

Die gemeinsame Nutzung verteilter Ressourcen wurde zu einem gängigen Prozedere, welches aber wieder neu zu lösende Probleme, wie z. B. Kompatibilität der Software, aufwarf. Die Kopplung von Hochleistungsrechnern, ebenso wie jene von anderen verteilten Ressourcen wiesen ähnliche Probleme und Möglichkeiten auf. Es entstand das neue Forschungsgebiet des Gridcomputings.

1998 gründeten Grid Entwickler und Anwender das Global Grid Forum (GGF [ggf]), um Grid Standards zu definieren und Ideen auszutauschen. Foster [FK03] definiert das Grid als eine Hardware und Software Infrastruktur, die einen zuverlässigen, konsistenten und kostengünstigen Zugriff auf Hochleistungsrechner ermöglicht.

Das Verschalten und Nutzen dieser Hochleistungsrechenressourcen war in den letzten Jahren Inhalt vieler Projekte [AJC00, EHR+98]. In der Initiative TeraGrid [ter] haben sich große Universitäten und Rechenzentren zusammengeschlossen, um Rechenleistung von 40 Teraflops und nahezu zwei Petabytes Speicher zu erreichen. Die Kommunikation des TeraGrids erreicht in einem dedizierten, nationalen Netzwerk bis zu 20 Gigabits pro Sekunde.

2.5.2 Leistungen von Grid-Umgebungen

Diese Arbeit fokussiert einen Teil des Gridcomputings besonders, das „Distributed Supercomputing“[Mes03], das die Verschaltung mehrerer Hochleistungsrechner umfasst.

Abbildung 2.9 zeigt eine Beispielkonfiguration, bei der Hochleistungsrechner

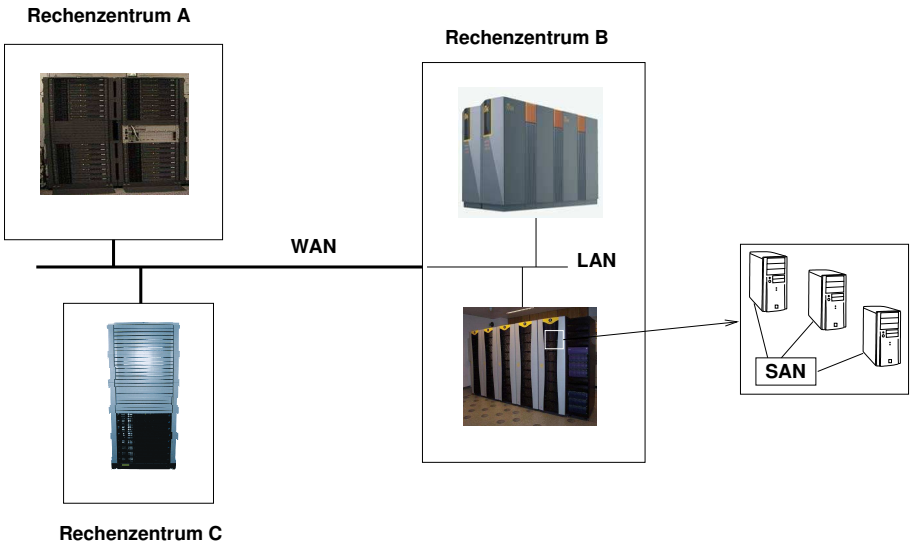


Abbildung 2.9: Beispielkonfiguration eines „Distributed Supercomputers“

verschiedener Rechenzentren gekoppelt sind. Dabei erfolgt die Vernetzung innerhalb von Rechenzentrum B über ein schnelles lokales Netzwerk. Oftmals wird hier ein HIPPI (High Performance Parallel Interface [hip]) eingesetzt. Die Rechenzentren kommunizieren untereinander über ein Wide Area Network (WAN). Die Kommunikation innerhalb jeder einzelnen Maschine erfolgt über ein System Area Network (SAN). Dieses kann ein internes Netzwerk des Maschinenherstellers sein, wie z. B. das der Cray XT3 [xt3], oder aber ein Hochleistungsnetzwerk für PC-Cluster, wie z. B. Infiniband [inf].

Diese Beispielkonfiguration kann als ein heterogener Cluster betrachtet werden, d. h. unterschiedliche Recheneinheiten sind durch unterschiedliche Kommunikationsverbindungen verbunden. Dieser Cluster zeichnet sich im Hinblick auf Gridcomputing dadurch aus, dass mehrere Parallelrechner und/oder PC-Cluster miteinander zu einer virtuellen Maschine verschaltet werden. Die Kommunikationsverbindungen zwischen den einzelnen Recheneinheiten variieren somit von sehr schnellen SANs innerhalb der einzelnen Maschinen über Highspeed-LANs innerhalb eines Rechenzentrums bis hin zu WANs, die Rechenzentren miteinander verbinden.

In anderen Bereichen des Gridcomputings werden eher Farmen von PCs und einzelne dedizierte PC-Cluster, verbunden durch LANs und WANs, miteinander verschaltet.

Was beiden Bereichen gemeinsam ist, ist ihre Heterogenität, sowohl im Bereich der Rechenleistung der einzelnen Recheneinheiten als auch im Bereich ihrer Kommunikationsverbindungen. LAN-Standard-Netzwerke, wie z. B. Fast

Ethernet, verfügen über bis zu 100 MegaBit pro Sekunde Bandbreite und 80 μs Latenz. Das wesentlich schnellere Gigabit Ethernet ermöglicht Bandbreiten bis zu einem GigaBit pro Sekunde. Innerhalb von Clustern verbreitete Hochgeschwindigkeitsnetzwerke hingegen, wie z. B. Myrinet [myr], Infiniband [inf] oder QSNet von Quadrics [qua] erreichen Bandbreiten bis zu 9 GBit/s und 2 μs Latenz. Tabelle 2.1 zeigt eine Übersicht über einige verbreitete Verbindungstechnologien [nws06].

Technologie	**Anbieter**	**MPI Latenz** μsec, short msg	**Bandbreite/Link** unidirektional, MBit/s
NUMAlink 4 (Altix)	SGI	1	3200
RapidArray (XD1)	Cray	1,8	20001
QsNet II	Quadrics	2	9002
Infiniband	Voltaire	3,5	8303
High Performance Switch	IBM	5	10004
Myrinet XP2	Myricom	5,7	4955
SP Switch 2	IBM	18	5006
Ethernet	verschiedene	30	100

Tabelle 2.1: Latenz und Bandbreiten einiger Verbindungstechnologien

Betrachtet man die Rechenleistung eines Standardprozessors, so beläuft sich diese auf maximal einen GFLOPS (Giga Floating Point Operations per Second). Hochleistungsprozessoren, wie z. B. der PowerPC 970, erreichen hingegen bis zu 9,2 GFLOPS.

Die zur Verfügung stehende Leistung kann von diesen Maximalwerten jedoch nicht abgeleitet werden. Faktoren wie Speicherbandbreite, Cachegrößen und verfügbarer Hauptspeicher haben einen relevanten Einfluss auf die Leistung einer Recheneinheit. Außerdem kann sie durch unterschiedliche Auslastungen oder auch Ausfälle während der Laufzeit variieren. Systemprozesse oder Prozesse anderer Benutzer bestimmen den Auslastungsgrad der CPU. Das Netzwerk ist nicht immer dediziert. Nachrichten müssen sich dann Verbindungen teilen. Weiterhin können komplette Netzwerkverbindungen ausfallen und im laufenden Betrieb durch andere, teils wesentlich langsamere, ersetzt werden. Vorausgesetzt die Software gestattet es, kann z. B. von einem Infiniband-Netzwerk mit einer Datenübertragungsrate von mehreren Gigabits pro Sekunde auf ein Fast Ethernet mit bis zu 100 MegaBit pro Sekunde innerhalb eines PC-Clusters umgeschaltet werden. Für die Anwendung ist dieses Geschehen vollständig transparent. Einzig die Geschwindigkeit bei der Übertragung von Nachrichten verändert sich.

Das Vorhandensein der unterschiedlichen Leistungsfähigkeit im Hinblick auf Kommunikation und Berechnung wurde im letzten Abschnitt aufgezeigt, ihre Feststellung und damit Bewertung der Umgebung ist Thema des folgenden Abschnitts.

2.5.3 Bewertungsverfahren

Die Bewertung der Rechenumgebung kann sowohl statisch als auch dynamisch erfolgen.

Statische Bewertung

Wird die Bewertung der Umgebung statisch bestimmt, so erfolgt dieses vor der Laufzeit der Anwendung. Entscheidungen, auch während der Laufzeit, basieren auf diesen ermittelten statischen Kapazitäten.

Statische Kapazitäten von **Recheneinheiten** ändern sich nicht und beinhalten Merkmale wie Prozessortyp, Taktfrequenzen, Größe des Hauptspeichers, Größe des Caches oder auch Busbreiten. Weit verbreitete Prozessoren von Arbeitsplatzrechnern sind z. B. der Intel Pentium 3, oftmals getaktet mit Werten von circa einem Gigahertz. Aktuelle Prozessortypen sind z. B. der Intel Pentium 4 oder AMD Athlon 64 mit einer Taktfrequenz von über drei Gigahertz.

Eine einfache Weise Systeme zu beurteilen, stellt die FLOPS-Rate dar. FLOPS (Floating Point Operations per Second) gibt die maximale Anzahl an Fließkomma-Operationen pro Sekunde an, die eine CPU erreichen kann. Damit bildet dieser Wert eine obere Schranke, die von einer realen Anwendung jedoch nicht erreicht werden kann.

Zwei der wichtigsten Parameter von **Netzwerkverbindungen** sind Latenz und Bandbreite. Niedrige Latenzen und hohe Bandbreiten kennzeichnen leistungsfähige Netzwerke.

Topologien von Netzwerken sind ein weiterer wichtiger Aspekt, um auf ihre Leistungsfähigkeit schließen zu können. So müssen Nachrichten in Torustopologien von Knoten zu Knoten weitergeleitet („geroutet“) werden und teilen sich damit Netzwerkverbindungen. Direkt verschaltete („switched“) Recheneinheiten hingegen haben über einen „Switch“(Umschalter) direkte Verbindung zum Kommunikationspartner.

Die oben beschriebenen Parameter können z. B. Datenblättern entnommen werden und verändern sich während der Laufzeit nicht.

Eine andere Möglichkeit ist, durch standardisierte Metriken die Leistungsfähigkeit eines Systems zu evaluieren (**Benchmarking**). Für anwendungsspezifisches Benchmarking werden vor dem Anwendungslauf Testmessungen mit der Anwendung selbst oder einer vergleichbaren Anwendung durchgeführt und das System bewertet. Standardisierte Benchmarks werden von der „Standard

Performance Evaluation Corporation“(SPEC [spe]) erstellt und gewartet. Diese werden auf die neueste Generation von Hochleistungsrechnern angewendet und die Ergebnisse publiziert. Anhand der Größe des Benchmark-Wertes kann die Leistungsfähigkeit einer Maschine der anderer Maschinen gegenübergestellt bzw. bewertet werden. So verfügt z. B. die IBM IntelliStation POWER 285 Workstation mit POWER5+ CPU (1900 MHz) über einen SPECfp2000-Wert von 2838 und zählt damit derzeit (Februar 2006) zu den schnellsten Maschinen. Maschinen wie IBM eServer pSeries 660 Model 6H1 mit RS64 IV CPU (600 MHz) erreichen hingegen einen SPECfp2000-Wert von nur 245 [spe06].
Diese Werte zeigen das große Spektrum an Leistungsfähigkeit vorhandener Ressourcen.

Sowohl Benchmarking als auch das Zugrundelegen von technischen Daten sind geeignet, die allgemeine Leistungsfähigkeit von Systemen zu bewerten und Systeme hinsichtlich ihrer maximal möglichen Leistung zu vergleichen (z. B. für Kaufentscheidungen). Sollen diese Informationen jedoch für eine Anwendung von Nutzen sein, so sind diese Verfahren eher geeignet für dedizierte Systeme. Es werden bei jeder Rechnung die gleichen Parameter zugrunde gelegt. Dadurch entsteht eine Reproduzierbarkeit und damit Vergleichbarkeit von Rechenläufen. Ein weiterer großer Vorteil ist, dass keine zusätzlichen Verwaltungskosten während der Laufzeit entstehen und damit die Laufzeit der Anwendung erhöhen. Da die meisten Systeme jedoch von mehreren Anwendern benutzt werden, stehen jedem Prozess geringere Kapazitäten, bedingt durch Ressourcen-Teilung, zur Verfügung. Daher sind diese Verfahren nur bedingt anwendbar für Anwendungen auf Mehrbenutzersystemen. Im Hintergrund laufende Betriebssystemprozesse erniedrigen zusätzlich die Verfügbarkeit der Ressourcen.

Dynamische Bewertung

Dynamische Kapazitäten werden zusätzlich durch die Auslastung der einzelnen Ressourcen bestimmt. In nicht-dedizierten Systemen stehen der Anwendung durch Betriebssystemprozesse oder Prozesse anderer Benutzer nicht die kompletten Ressourcen zur Verfügung. Die Last ändert sich in Abhängigkeit von dem Ressourcenverbrauch aller laufenden Prozesse. Um auf Systemzustandsänderungen unmittelbar reagieren zu können, erfolgt die Bewertung des Systems während des Anwendungslaufs.

Ein Nachteil dieses Verfahrens ist der Zusatzaufwand während der Laufzeit, der durch Beobachtung des Systems entsteht. Eine Reproduzierbarkeit der Messergebnisse ist ebenfalls durch die Unvorhersehbarkeit des Systemzustands nicht gegeben.

Um dem dynamischen Charakter des Grids gerecht zu werden, können für eine angemessene Lastverteilung nur dynamische Verfahren eingesetzt werden, die die Leistungsfähigkeit einer Ressource zur Laufzeit messen (Monitoring). Nur auf diese Weise kann gewährleistet werden, dass die Last entsprechend der

aktuellen Situation von Netzwerk und Recheneinheiten verteilt werden kann.

2.5.4 Monitoring-Systeme

Sowohl für die Analyse der Rechenleistung als auch für die des Netzwerks wurden verschiedene Monitoring-Systeme entwickelt. Eine grundlegende Unterscheidung betrifft dabei die Zeit der Messwert- bzw. Ergebnisverfügbarkeit. Sie kann zum einen nach der Laufzeit gegeben sein (Offline-Monitoring) bzw. während der Laufzeit (Online-Monitoring) [Sim00]. Offline-Monitoring erstellt Protokolle eines Anwendungslaufes. Mit Hilfe dieser können Performance-Probleme analysiert und die Anwendung entsprechend optimiert werden. Beim Online-Monitoring kann die Anwendung direkt auf die Daten zugreifen und damit auf Systemzustandsänderungen reagieren.

Zu den Offline-Monitoring Systemen gehören [GTC+00, PK03]. Sie benutzen Mechanismen, bei denen das Verhalten der Anwendung mitprotokolliert wird. NetLogger [GTC+00] arbeitet mit dieser Methode als Diagnosesystem für Performance-Probleme. Es liefert eine komplette Übersicht über das System, indem es Monitoring auf Netzwerk-, Maschinen- und Anwendungsebene kombiniert. Dadurch wird das Auffinden von Leistungseinbrüchen einer Anwendung unterstützt. Ein Nachteil dieser Verfahren ist, dass sie eine große Menge von Beobachtungsdaten erzeugen, die gegebenenfalls zu Analyse-Komponenten transferiert werden müssen.

Auch wenn diese Systeme einen guten Überblick über das Laufzeitverhalten einer Anwendung liefern, bieten sie nicht die Möglichkeit, das Verhalten der Anwendung in Abhängigkeit vom Systemzustand während der Laufzeit zu steuern.

Das Grundprinzip anderer Systeme ist, mittels Sensoren Systemparameter zu erfassen [LMK+03, WSH99] und diese an eine zentrale Instanz weiterzuleiten, die die Messwerte weiterverarbeitet.

Die Methode von Agarwala et al. [APK+03] nutzt das virtuelle Linux Dateisystem */proc* und erweitert es um Systeminformationen von entfernten Maschinen in einem */dproc* Dateisystem. Dabei werden die Informationen zwischen den Maschinen mittels Kernel-Kernel-Kommunikation ausgetauscht. Die CPU-Auslastung wird periodisch durch einen Thread, der die Task-Liste des Kernels untersucht, beobachtet. Das Netzwerkmonitoring wird durch ein Modul, das den Netzwerkverkehr auf allen offenen Verbindungen beobachtet, bewerkstelligt. */dproc* arbeitet auf Systemebene und erfordert eine Kernel-Erweiterung.

Der *Network-Analyser* [MP01] zielt auf die adäquate Verteilung von Tasks ab. Er sammelt Informationen von Workstation-Zuständen, wie z. B. CPU-Last, Prozess-Zustände, Ein-/Ausgabe der letzten 24 Stunden. Auf dieser Basis werden Auslastungsvorhersagen unter Zuhilfenahme von stochastischen Verfahren getroffen, die für die Verteilung der Tasks als Grundlage genommen werden. Die

erwartete Ausführungszeit von Tasks wird ebenfalls abgeschätzt.

Für das Grid-Umfeld wurden Monitoring-Systeme entwickelt, die die spezielle Netzwerkstruktur des Grids berücksichtigen, die im Allgemeinen aus mehreren Subnetzen besteht.

GridMon [LTT] beobachtet Netzwerke, um Fehler und Ineffizienz zu identifizieren und um Annahmen über die zu erwartende Leistungsfähigkeit machen zu können. Die Beobachtungen beziehen sich auf Konnektivität, Paketverluste, Durchsatz und Round Trip Time. GridMon arbeitet auf Systemebene. Für seinen Einsatz müssen dedizierte Maschinen bereitgestellt werden, die jeweils ein lokales Netzwerk beobachten. Diese Maschinen führen wiederum Messungen untereinander aus, um die Verbindungen zwischen ihnen zu analysieren.

Remos [LMK+03] (REsource MOnitoring System) erfasst Netzwerkdaten und sagt zukünftiges Verhalten, basierend auf historischen Daten, vorher. Dadurch, dass die Anwendung nur auf eine Anfrage hin Informationen erhält, wird der Overhead klein gehalten. Das Sammeln der Netzwerkdaten erfolgt mittels Kollektoren, die hierarchisch angeordnet sind. Auf unterster Ebene ist ein Kollektor verantwortlich für das Einbringen von Informationen über sein Netzwerk. So ist ein lokaler Kollektor z. B. zuständig für das Sammeln von Daten seines LANs. Lokale oder globale Kollektoren auf entfernten Sites können kontaktiert werden, um Informationen über diese entfernten Sites zu erhalten.

Der *Network Weather Service* (NWS) [Wol03] ist ein verteiltes System, das die Leistungsfähigkeit von Netzwerk und Rechenressourcen periodisch beobachtet und dynamisch vorhersagt. Das Performance-Monitoring erfolgt mittels Sensoren. Diese erzeugen Informationen über Netzwerk, CPU und Speicher. Basierend auf diesen Informationen, werden numerische Modelle benutzt, um Vorhersagen für zukünftiges Verhalten zu treffen. Zusätzlich gestartete Prozesse („probe“-Programme) werden sowohl für Performance-Messungen der CPU eingesetzt als auch für die des Netzwerks. Die Anordnung der Sensoren ist hierarchisch, was den Einsatz für große Systeme unterstützt.

Im Rahmen der Globus-Allianz [Glo], die sich zum Ziel setzt, grundlegende Technologien für das Grid zu entwickeln, wurden auch Monitoring-Systeme speziell für das Grid entwickelt. So bietet z. B. *Ganglia* [MCC04] die Möglichkeit, Monitoring-Daten von Clustern zu sammeln. Dabei nutzt es die hierarchische Struktur, die durch die Verschaltung mehrerer Cluster zustande kommt. Ganglia setzt „Monitoring-Daemons“ ein, die in gewissen Abständen Leistungsinformationen erfassen.

All diese Systeme arbeiten auf Betriebssystemebene. Sie beobachten die vorhandenen Leistungsfähigkeiten der Komponenten CPU und Netzwerk. Bei einigen Systemen [APK+03, MP01] erfolgt dieses durch passive Sensoren, d. h. es werden Informationen, die dem Betriebssystem bereits vorliegen, abgefragt. Mit

aktiven Sensoren arbeiten *GridMon*, *Remos* und *NWS*. Hier werden Prozesse explizit gestartet, um Messungen durchzuführen.

Auslastungsvorhersagen basieren auf historischen Daten. Das bedeutet, dass von vergangenen Auslastungen auf zukünftige geschlossen wird, also ein sich wiederholendes Auslastungsprofil vorausgesetzt wird.

2.6 Analyse der Eckpunkte

Betrachtet man die in Abschnitt 2.4.4 beschriebenen Lastverteilungsmethoden und ihre Ziele, so wird schnell deutlich, dass Lastverteilung in einer heterogenen Umgebung wie das Grid nur dann zur Effizienzsteigerung führen kann, wenn sie Umgebungsparameter bei ihren Entscheidungen miteinbezieht.

Die meisten der vorgestellten Lastverteilungsmethoden aus Abschnitt 2.4.4 gehen von einer Gleichverteilung der Last, und damit der Graphknoten, als Ziel aus. Dieses ist jedoch nur in homogenen Umgebungen angemessen, wie sie z. B. in Parallelrechnern zu finden sind. Das Grid weist jedoch eine heterogene und dynamische Struktur auf, in der Maschinen mit unterschiedlicher Leistungsfähigkeit an der Durchführung einer Anwendung beteiligt sind (siehe Abschnitt 2.5.1). Wie wichtig die Berücksichtigung dieser Heterogenität ist, wird von Entwicklern einiger Lastverteilungsmethoden hervorgehoben [SKK03, WLR93, EMP00, RN01, BL02]. So beschreiben Eickermann et al. [EHR+98], dass die Anpassung der Last an die Umgebung zu großen Effizienzsteigerungen führen kann. Das Werkzeug MinEX von Das et. al [DHB02] nutzt ebenfalls Informationen über die Umgebung, um Lastverteilung entsprechend der Umgebung durchzuführen, betrachtet jedoch keine dynamischen Werte, sondern bezieht sich auf vorangegangene Performance-Studien.

Soll Heterogenität berücksichtigt werden, so ist eine Voraussetzung, dass das Lastverteilungswerkzeug Teilpartitionen unterschiedlicher Größe generieren kann. Diese Möglichkeit wird aber nur von wenigen Werkzeugen, wie z. B. Jostle und ParMETIS, geboten.

Zur Unterstützung dieser Werkzeuge bedarf es einer Instanz, die die Umgebung analysiert und die aufbereiteten Informationen an die Lastverteilung weitergibt. Damit schließt sich die Lücke zwischen Umgebung und Lastverteilung (siehe Abbildung 2.10).

Die Umgebungsanalyse muss dabei durch geeignete Bewertungsmethoden erfolgen. Eine angemessene Methode stellt dabei das Monitoring dar, das zur Laufzeit Informationen über die Umgebung ermittelt und zur Verfügung stellt.

2.6.1 Monitoring

Das erforderliche Monitoring-System muss bei geringem Overhead dazu in der Lage sein, hinreichende Informationen über den Status der Recheneinheit wie auch den des Netzwerks zu liefern.

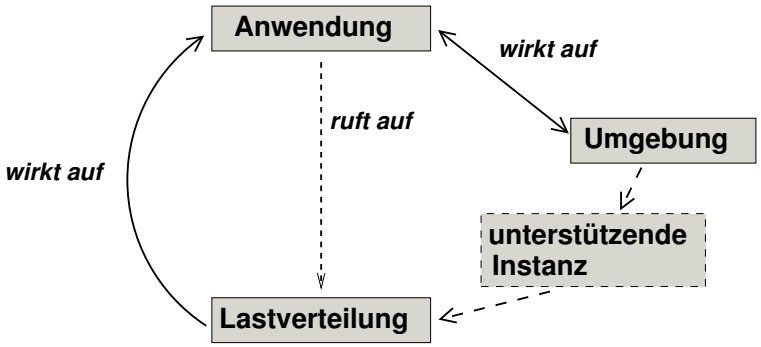


Abbildung 2.10: Erweiterung der Komponenten

Um dieses zu gewährleisten, muss es folgende Eigenschaften aufweisen:

- Skalierbarkeit
- Effizienz
- Verfügbarkeit auf Benutzerebene

Skalierbarkeit wird ermöglicht, indem das Sammeln von Monitoring-Daten und ihre Weiterverarbeitung verteilt erfolgt. Werden zentrale Instanzen eingesetzt, so würden sie, insbesondere in massiv-parallelen Systemen wie das Grid, zu einem Flaschenhals werden. Einen weiteren Aspekt stellt die Ausfallsicherheit dar. Fällt die zentrale Instanz aus, so ist das System nicht mehr arbeitsfähig und muss terminieren. Nur ein verteiltes System ermöglicht ein Weiterarbeiten bei Ausfall einer Recheneinheit.

Die Forderung nach **Effizienz** beinhaltet, dass aussagekräftige Informationen über das System gesammelt werden können, wobei der notwendige Zusatzaufwand gering bleibt. Der Spagat zwischen einem hohen Informationsgehalt der gesammelten Daten einerseits und einem möglichst geringen Overhead andererseits ist eine schwierige Aufgabe bei der Entwicklung von Monitoring-Systemen.

Daten über den Systemzustand müssen für die Anwendung **zugreifbar** sein. Ein Eingriff in das Betriebssystem bzw. zusätzliche Installationen sollten nicht notwendig sein. Dieses würde zum einen die Unterstützung der Systemadministration erfordern und zum anderen das Angebot nutzbarer Maschinen einschränken, da das Vorhandensein bestimmter Software vorausgesetzt werden müsste. Um den Anforderungen aller Benutzer gerecht zu werden, müssen solche Systeme über ein umfangreiches Informationsangebot verfügen. Dieses widerspricht jedoch der Forderung nach Effizienz, d. h. ausschließliche Sammlung notwendiger Informationen.
Ein wesentlicher Aspekt ist aber, dass ausschließlich die der Anwendung zur

Verfügung stehenden Ressourcen bewertet werden sollen. Daher muss das Monitoring auf Anwendungsebene stattfinden.

Alle in Abschnitt 2.5.4 vorgestellten Systeme arbeiten auf Betriebssystemebene. Um die erforderlichen Daten zu erhalten, muss Software installiert werden, oder es müssen sogar komplette Maschinen zur Verfügung gestellt werden (GridMon). Informationen darüber, welche Kapazitäten der eigenen Anwendung zur Verfügung gestellt werden, können nicht geliefert werden.

Ein wichtiger Punkt ist, dass alle Methoden voraussetzen, dass Informationen über den Aufbau des Netzwerks vorliegen, die Struktur also bekannt ist. Ein statischer, hierarchischer Aufbau bildet die Grundlage. Dieses impliziert wiederum das Vorhandensein von Software, die diese Information bereitstellt. Ein weiterer Aspekt ist, dass sich die Struktur des Netzwerks aus Anwendungssicht zur Laufzeit ändern kann. Dieses erfordert ein System, dass auf diese Strukturänderungen dynamisch reagiert.

Die vorgestellten Verfahren bieten ein großes Spektrum an Informationen, um den Bedürfnissen vieler Anwendungen gerecht zu werden. Diese Informationsvielfalt überschreitet jedoch den im Rahmen dieser Arbeit notwendigen Informationsbedarf, der sich rein auf die Rechen- und Netzwerkkapazitäten für die entsprechende Anwendung bezieht. Der zu hohe Aufwand des Monitorings führt zu einer geringeren Effizienz.

Somit können nicht alle Anforderungen an ein Monitoring-System erfüllt werden, da sich Skalierbarkeit auf vorhandene Informationen über das Netzwerk stützt und Effizienz durch zu große Informationsfülle eingeschränkt ist.

Daher wird auf keines der vorhandenen Systeme zurückgegriffen, sondern ein eigenes Monitoring-System entwickelt, um den Anforderungen gerecht zu werden.

Die unterstützende Instanz aus Abbildung 2.10 besteht also aus den zwei Bereichen Monitoring und Lastverteilungssteuerung (siehe Abbildung 2.11).

In den nächsten Kapiteln werden zunächst Anforderungen an diese zwei Bereiche herausgearbeitet und anschließend Lösungen vorgestellt.

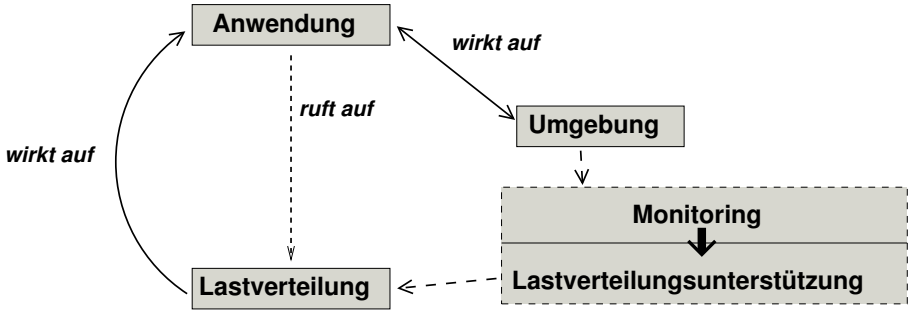


Abbildung 2.11: Teile der Unterstützungskomponente

Kapitel 3

Anforderungskatalog

Nach der Analyse der Eckpunkte im letzten Kapitel kristallisieren sich zwei Bereiche heraus, für die Methoden entwickelt werden müssen:

- Zum einen für die **Lastverteilungssteuerung**, das eigentliche Ziel dieser Arbeit, und
- für das **Monitoring-System**, das die Grundlage für die Lastverteilungssteuerung bildet.

Ein Anforderungskatalog für beide Bereiche ist Thema dieses Kapitels.

3.1 Anforderungen an die Lastverteilungssteuerung

Die Anforderungen ergeben sich aus den drei in den vorangegangenen Abschnitten beschriebenen Eckpunkten Anwendung, Lastverteilung und Umgebung (siehe Abbildung 3.1).

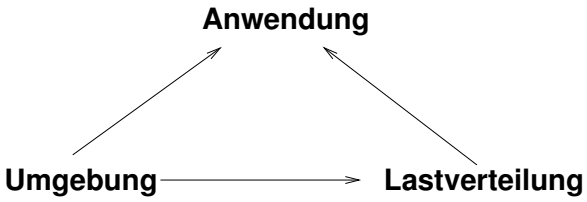


Abbildung 3.1: Abhängigkeiten

Dabei haben diese drei Punkte folgende Merkmale:

Umgebung:

1. Es liegt ein heterogenes System zugrunde, die Leistungsfähigkeit der Recheneinheiten kann stark differieren, sowohl in ihrer CPU-Leistung als auch in ihrem Speicherangebot.
   ⇒ **heterogene Recheneinheiten**
2. Das Grid besteht aus unterschiedlichen Kommunikationsnetzwerken, die sich durch ihre Übertragungsraten extrem unterscheiden können.
   ⇒ **heterogene Netzwerke**
3. Die Netzwerkstruktur des Grids ist **hierarchisch**. Subnetze werden gekoppelt. Diese bilden Netze, die wiederum gekoppelt werden.
4. Durch Ressourcenteilung ändert sich sowohl die Leistungsfähigkeit der Netzwerkverbindungen als auch die der Prozessoren während der Laufzeit.
   ⇒ **dynamisches System**
5. Es handelt sich um ein **massiv-paralleles System**. Eine hohe Anzahl an Recheneinheiten muss verwaltet werden.

Anwendung:

1. Die Berechnungen basieren auf Gitter.
   ⇒ **gitterbasierte Anwendung**
2. Die Gitterzellen tauschen oft Daten mit ihren Nachbarn aus. Es existiert ein hohes Kommunikationsaufkommen.
   ⇒ **eng kommunikativ gekoppelt**
3. Die Gitter können aus Millionen von Zellen bestehen. Für jede Zelle müssen Daten gespeichert werden.
   ⇒ **hoher Speicherbedarf**
4. Oftmals ist eine Schnittstelle zu frei verfügbaren Lastverteilungswerkzeugen vorhanden. Die Anwendung stellt dann den **dualen Graphen** des Gitters zur Verfügung.
5. Bei der Entwicklung von Simulationswerkzeugen stehen deren Leistungsfähigkeit und damit numerische Verfahren im Vordergrund. Der Bereich der Lastverteilung ist Teil eines anderen Arbeitsgebietes (Graphentheorie), das insbesondere von Experten wirkungsvoll umgesetzt werden kann.
   Anwendung und Lastverteilung stellen damit zwei getrennte und in sich abgeschlossene, aber voneinander abhängige Bereiche dar.

Lastverteilungsmethoden:

1. Neben der **gleichmäßigen Verteilung der Gitterzellen** auf die Prozessoren ist die **Minimierung der Schnittkanten** Ziel der Lastverteilung.

2. Die **Verfügbarkeit** parallel arbeitender Werkzeuge ist stark **begrenzt**.
3. Nur wenige Werkzeuge bieten die Möglichkeit, **Partitionsgrößen vorzugeben**. Ist diese Funktionalität allerdings vorhanden, so ist es Aufgabe des Anwenders, diese Größen zu bestimmen.
4. Die Verbindungsstruktur des Systems wird oftmals nicht betrachtet.
5. Lastverteilung wird zwischen allen Prozessen durchgeführt.

Das geplante Zusammenspiel dieser Aspekte zeigt Abbildung 3.2.

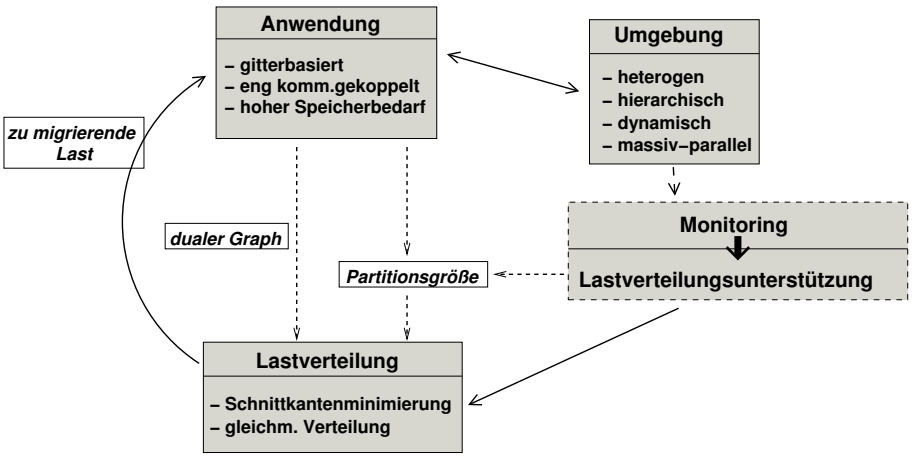


Abbildung 3.2: Zusammenspiel der Eckpunkte

Durch diese Charakteristika ergeben sich folgende **Anforderungen** an eine Lastverteilungssteuerung, die die Merkmale der Umgebung betrachtet und diese wirkungsvoll nutzt:

- Anpassung der Lastmenge an die Kapazitäten der einzelnen Prozessoren
- Berücksichtigung der Existenz von unterschiedlich schnellen Kommunikationsverbindungen durch Vermeidung langsamer Verbindungen bei der Lastverteilung
- Ausnutzung der hierarchischen Netzwerkstruktur
- Reaktion auf Zustandsänderungen (Prozessor-/Netzwerklast) während der Laufzeit
- Verwaltung einer großen Anzahl an Prozessoren

- Nutzung vorhandener Lastverteilungswerkzeuge, entweder die der Anwendung oder aber allgemeiner Werkzeuge (z. B. Jostle) muss weiterhin möglich sein
- Transparenz für den Benutzer, d. h. kein Zusatzaufwand für den Entwickler
- geringer Overhead, sowohl in Bezug auf Speicher (speicherintensive Anwendung) als auch auf Laufzeit

Um diese Anforderungen zu erfüllen, wurde ein Konzept entwickelt, das in Kapitel 4 vorgestellt wird. Die Umsetzung dieses Konzepts mit anschließender Validierung wird darlegen, ob bzw. inwieweit es den obigen Anforderungen entspricht.

3.2 Anforderungen an die Monitoring-Umgebung

Für das Monitoring-System müssen folgende Anforderungen erfüllt werden, um den Forderungen Skalierbarkeit, Effizienz und Verfügbarkeit auf Benutzerebene nachzukommen:

- **Skalierbarkeit:**
  Das Monitoring muss verteilt erfolgen. Es darf keine zentrale Instanz notwendig sein, weder bei der Messwerterzeugung noch bei der Verarbeitung der Messwerte.
- **Verfügbarkeit:**
  - Es muss Online-Monitoring eingesetzt werden, da die Anwendung in der Lage sein muss, auf aktuelle Systemzustände zu reagieren.
  - Das Monitoring muss auf Anwendungsebene stattfinden. Die Informationen auf Betriebssystemebene liefern nur allgemeine Informationen, nicht jedoch jene, die über die Leistungskapazitäten für die Anwendung Auskunft geben.
  - Aktive Sensoren können ermitteln, welche Kapazitäten der Anwendung zur Verfügung stehen. Wichtig ist, sie so zu integrieren, dass sie das Laufzeitverhalten der Anwendung nicht verändern.
- **Effizienz:**
  Um Effizienz zu ermöglichen, müssen hinreichende Informationen bei minimalem Overhead ermittelt werden. Dieses kann erreicht werden, indem Anzahl und Güte der Messwerte so groß wie nötig, aber so klein wie möglich werden. Dazu dienen die folgenden Strategien:
  - **Anzahl der notwendigen Informationen:**
    Es werden nur relevante Informationen über die zu beobachtende Anwendung gesammelt.

- **Aussagekraft der Informationen:**
  Die für die Aussagekraft notwendige Granularität der Information bestimmt die Genauigkeit der Messungen und die Höhe des Datenaufkommens. Diese sollte möglichst gering sein, um den Overhead minimal zu halten. Daher werden folgende Ziele angestrebt:
  * **Monitoring der Recheneinheiten:**
    Die Informationen müssen ausreichend sein, um die Last entsprechend der Leistungsfähigkeit der einzelnen Recheneinheiten zu verteilen.
  * **Monitoring des Netzwerks:**
    Wesentlich sind hier die Zeiten für die Dauer eines Nachrichtenaustausches innerhalb der Anwendung. Mit Hilfe der Messwerte muss eine Einschätzung der Übertragungsgeschwindigkeiten von Nachrichten möglich sein und damit auch eine Vergleichbarkeit der Verbindungen. Dieses betrifft aber nur Verbindungen, die die Anwendung benutzt.
- **Effizienz durch Dynamik:**
  Das Monitoring selbst sollte auf das System reagieren, indem die Anzahl der Monitoring-Aufrufe abhängig vom Zustand des Systems ist.

Das entwickelte Monitoring berücksichtigt die Forderungen Gropps in [GL99]. Dabei stellt Gropp besonders die Schwierigkeit heraus, exakte, reproduzierbare Messungen im parallelen Umfeld durchzuführen, da Parallelismus auch immer ein wenig Nichtdeterminismus beinhalte. Auch Kielmann et al. [KBM+02] sehen diese Schwierigkeit und begegnen dieser, indem sie einen Netzwerksimulator entwickelten.

Gropp sieht als besondere Gefahren bei Performance-Messungen:

1. Der Verbindungsaufbau erfolgt in manchen Systemen dynamisch. Daher kann die erste Kommunikation viel länger dauern.
2. Ein im Hintergrund laufendes Dateisystem kann viel der verfügbaren Kommunikationsbandbreite in Anspruch nehmen.
3. In einigen Anwendungen wird Latency-Hiding durch Überlappen von Berechnung und Kommunikation eingesetzt.
4. Dedizierte, geschaltete (switched) Netzwerke haben eine ganz andere Leistung als gemeinsam genutzte Netzwerkstrukturen.
5. Zeitereignisse, die im Verhältnis zur Uhrenauflösung kurz sind, sind schwierig zu messen.
6. Die Messwerte sind abhängig vom Kommunikationsmuster.
7. Punkt-zu-Punkt-Kommunikation verhält sich anders als Kommunikation zwischen mehreren Prozessoren

Als Lösungen zu den oben genannten Problemen schlägt Gropp vor:

ad 1 Die erste Kommunikation darf bei den Messungen nicht berücksichtigt werden.

ad 2 Konkurrenzsituationen mit nicht in Bezug stehenden Anwendungen oder Jobs sollten ebenfalls betrachtet werden.

ad 3 Überlappen von Berechnung und Kommunikation sollten mitberücksichtigt werden.

ad 4 Die Gesamtbandbreite sollte nicht mit Punkt-zu-Punkt-Bandbreite verwechselt werden.

ad 5 Sehr kurzen Zeitereignissen sollte Beachtung geschenkt werden.

ad 6 Messe nicht mit einem einzigen Kommunikationsmuster.

ad 7 Messe mit mehr als zwei Prozessoren.

Auf Basis dieser Regeln entwickelten Gropp et al. das MPI-Performance-Analyse-Werkzeug *perftest*. Es misst die Leistungsfähigkeit einiger grundlegender MPI-Routinen in unterschiedlichen Situationen (siehe [per]).

Diese Forderungen bildeten die Grundlage bei der Entwicklung eines eigenen Monitoring-Systems, das in Abschnitt 4.4 beschrieben wird.

Kapitel 4

Lastverteilung in Grid-Umgebungen

Nachdem im letzten Kapitel Anforderungen für eine umgebungsspezifische Unterstützung für Lastverteilung aufgezeigt wurden, wird nun ein Lösungsansatz beschrieben.

4.1 Grundkonzept der adaptiven Steuerung der Lastverteilung

Die im Folgenden präsentierte Strategie zielt darauf ab, dem Anforderungskatalog aus dem letzten Kapitel gerecht zu werden, indem sie

1. die Recheneinheiten bewertet und die Lastmenge ihren Kapazitäten anpasst (Adaption an heterogene Rechenleistung),

2. die Netzwerke bewertet und schnelle Verbindungen den langsameren bei Lastverteilung und Datenmigrationen vorzieht.

   Für eine Anwendung, die auf einem System mit heterogenem Netzwerk läuft, bedeuten die stark unterschiedlichen Netzwerkgeschwindigkeiten, dass sich die Transferraten um das Hundertfache unterscheiden können.

   Datenmigrationen bei Lastverteilung sind davon besonders betroffen. Sind die Migrationszeiten sehr hoch, so steigt der Overhead für Lastverteilung und somit sinkt auch seine Effizienz. Übersteigt der Overhead den Laufzeitgewinn durch Lastverteilung (Kosten-Nutzen-Analyse), so verschlechtert sich die Gesamtlaufzeit der Anwendung und das Ziel von Lastverteilung ist verfehlt.

   Ein Ziel der Lastverteilungsstrategie ist es daher, Datenmigrationen über sehr langsame Verbindungen zu minimieren. Lastverteilung zwischen Recheneinheiten mit hohen Übertragungsraten werden hingegen bevorzugt.

3. diese Bewertungen zur Laufzeit durchführt und Entscheidungen basierend auf ihren aktuellen Werten trifft und damit auf die dynamische Umgebung reagiert.

4. die Strategie aus einer lokalen Sicht heraus (dezentrale Strategie) umsetzt und damit Skalierbarkeit ermöglicht.

Dadurch sollen die Hauptziele der Lastverteilung erreicht werden:

1. Die Laufzeit der Anwendung wird optimiert durch gleichmäßige Auslastung der Recheneinheiten und damit auch schnellere Terminierung.

2. Der Zusatzaufwand für die Lastverteilung wird minimiert durch Nutzung schneller Verbindungen und damit auch kürzerer Datenmigrationszeiten.

In den folgenden Abschnitten werden Konzepte vorgestellt, mit denen diese Ziele erreicht werden sollen.

4.2 Kapazitätenabhängige Lastverteilung

Ein wichtiger Aspekt, um eine gleichmäßige Auslastung der Recheneinheiten zu gewährleisten, ist eine Verteilung der Last in Abhängigkeit von der Leistungsfähigkeit der einzelnen Recheneinheit. Aus Anwendersicht heraus bezieht sich die Leistungsfähigkeit auf die dem Anwendungsprozess (AP) zur Verfügung stehenden Leistung.

Die Leistung einer Recheneinheit ist bestimmt durch ihre Berechnungsgeschwindigkeit (siehe Abschnitt 4.4.1). Sei *calc_speed* die Berechnungsgeschwindigkeit einer Recheneinheit, dann ist ihre Leistung = $1/calc\_speed$.

Seien AP_i und AP_j Prozesse der Anwendung, $0 \leq i, j < n$, $n = |Anwendungsprozesse|$. Seien $Leistung_i$ und $Leistung_j$ die Leistungsfähigkeit der Recheneinheiten RE_i bzw. RE_j, AP_i wird auf RE_i bearbeitet, AP_j auf RE_j, $Last_i$ und $Last_j$ die Last von AP_i bzw. AP_j, dann ist ein Ziel der Lastverteilung, dass zu jedem Zeitpunkt t gilt:

$$\frac{Last_i}{Leistung_i} = \frac{Last_j}{Leistung_j}, \quad \forall\ AP_i, AP_j \tag{4.1}$$

Um die optimale Last eines Anwendungsprozesses AP_i zu berechnen, wird zunächst die optimale Lastrate $optimale\ Lastrate_i$, d. h. der optimale Anteil an der Gesamtlast, bestimmt.

Dabei ist die Gesamtlast

$$Last_{tot} = \sum_{j=0}^{n-1} Last_j \tag{4.2}$$

$$optimale\ Lastrate_i = \frac{Leistung_i}{\sum_{j=0}^{n-1} Leistung_j} \tag{4.3}$$

Damit gilt:

$$\sum_{j=0}^{n-1} optimale\ Lastrate_j = 1 \tag{4.4}$$

Die optimale Last für AP_i, $optimale\ Last_i$, ist damit die anteilige Lastmenge der Gesamtlast $Last_{tot}$ für den optimalen Fall und berechnet sich wie folgt:

$$optimale\ Last_i = optimale\ Lastrate_i * Last_{tot} \tag{4.5}$$

Durch die Adaptivität der Anwendung verändert sich ihre aktuelle Last während der Laufzeit und kann zu einem Abweichen von der optimalen Last führen. Weicht die aktuelle Last von der optimalen Last ab, so existiert ein Ungleichgewicht, das die Effizienz der Berechnung gefährden kann.

Das Ungleichgewicht von AP_i, $Ungleichgewicht_i$, berechnet sich aus:

$$Ungleichgewicht_i = |1 - \frac{aktuelle\ Last_i}{optimale\ Last_i}| \tag{4.6}$$

wobei $aktuelle\ Last_i$ die aktuelle Last von AP_i ist.

Ein geringes Ungleichgewicht, bei dem die zu erwartenden Kosten für seine Behebung höher sind als der zu erwartende Laufzeitgewinn, wird toleriert. Kann dieses Ungleichgewicht jedoch zu Effizienzverlusten führen, so muss eine Umverteilung der Last durchgeführt werden.

Sollen Kosten und Nutzen von Lastverteilung bewertet werden, setzt dieses das Vorhandensein von Laufzeit-Informationen über zukünftige Berechnungen voraus. Da diese Informationen i. A. nicht vorliegen (keine Vorhersehbarkeit), basiert die Feststellung eines Ungleichgewichts auf einem festen Schwellwert *balance_tolerance*. Wird *balance_tolerance* überschritten, d. h.

$Ungleichgewicht > balance\_tolerance$,

so wird Lastverteilung initiiert.

4.3 Netzwerkabhängige Lastverteilung

Die Betrachtung des Netzwerks ist ein wesentlicher Bestandteil dieser Arbeit. Andere Entwickler sehen zwar die Wichtigkeit dieses Aspekts, berücksichtigen sie jedoch nicht in ihren Konzepten.

Durch das Vorhandensein unterschiedlich schneller Kommunikationsnetzwerke entstehen Klassen von Kommunikationsgeschwindigkeiten. Werden Recheneinheiten, die durch Verbindungen einer Geschwindigkeitsklasse gekoppelt sind, zu Subsystemen zusammengefasst, so entsteht eine Hierarchie von Subsystemen. Werden z. B. MPP-Systeme mit einem im Vergleich zu ihrem internen Netzwerk um Faktoren langsameren Netzwerk gekoppelt, so bildet jedes MPP-System für sich ein Subsystem. Falls einige Recheneinheiten der MPP-Systeme über einen schnelleren Weg als das SAN kommunizieren, z. B. über ihren gemeinsamen Speicher („shared memory“-Kommunikation), bilden diese Knoten

ein Subsystem auf niedrigster Ebene. Auf höchster Hierarchieebene bildet das Gesamtsystem ein Subsystem.

Das folgende Beispiel (siehe Abbildung 4.1) zeigt die Kopplung zweier PC-Cluster mittels Ethernet (Übertragungsrate bis zu 100 Mbit/s). Die Knoten jedes PC-Clusters kommunizieren über SCI (Scalable Coherent Interface [IEE93], maximal 4 GBit/s) bzw. Infiniband (maximal 10 GBit/s). Innerhalb jedes Knotens erfolgt die Kommunikation über den gemeinsamen Speicher (Geschwindigkeit bestimmt durch Speicherzugriffszeiten). Somit bilden sich drei Hierarchiestufen.

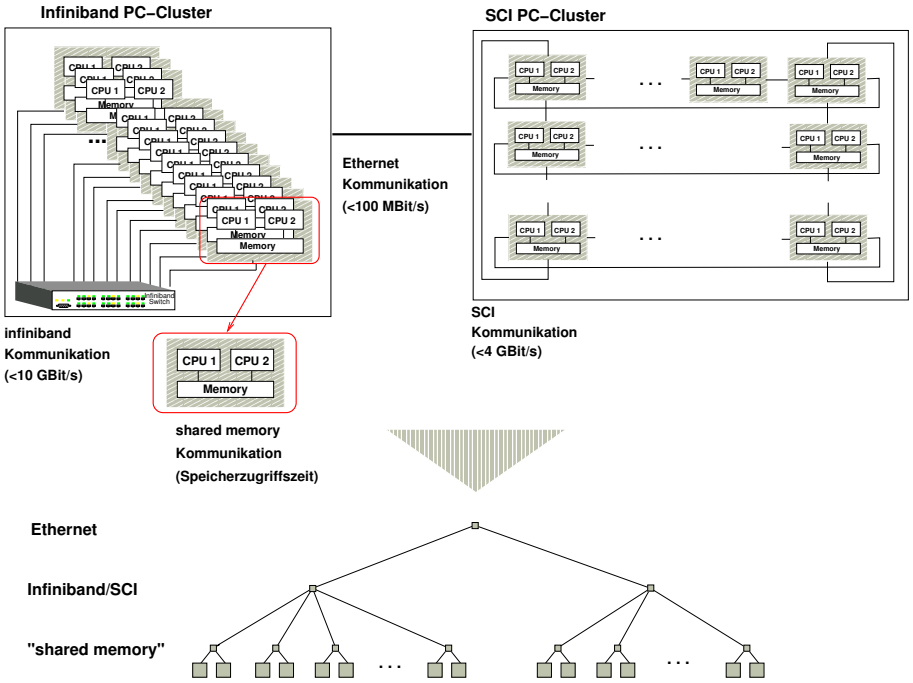


Abbildung 4.1: Beispiel: Abbildung einer Systemkonfiguration auf eine Hierarchie

Die Lastverteilungsstrategie macht sich diese Hierarchie zu Nutze, indem Lastverteilung auf möglichst niedriger Hierarchieebene, und damit mit möglichst niedrigen Kommunikationskosten, durchgeführt wird.

4.3.1 Hierarchie

Die Hierarchie von Subsystemen wird durch einen Baum dargestellt, dessen Knoten jeweils ein System mit folgenden Eigenschaften repräsentieren:

- die Wurzel repräsentiert das ganze System
- ein innerer Knoten repräsentiert ein Subsystem, bestehend aus allen PEs des darunter liegenden Teilbaums
- die Blätter repräsentieren einzelne PEs
- für jedes Subsystem gilt, dass die Kommunikation zwischen den PEs innerhalb eines Subsystems schneller ist als die Kommunikation zwischen PEs über Subsystemgrenzen hinweg

Abbildung 4.1 zeigt ein statisches Netzwerk. Da sich während der Laufzeit die Kommunikationsgeschwindigkeiten ändern können und dieses Auswirkungen auf die Struktur der Hierarchie hat, wird eine Hierarchieerkennung zur Laufzeit durchgeführt.

4.3.2 Hierarchieerkennung und -aktualisierung

Das Ziel der Hierarchieerkennung ist, unterschiedliche Klassen von Kommunikationsgeschwindigkeiten zu eruieren und diese auf eine hierarchische Struktur abzubilden.

Die Subsysteme auf unterster Ebene sind dabei durch die niedrigsten Kommunikationszeiten gekennzeichnet. Höhere Ebenen stellen Kommunikationsgeschwindigkeiten zwischen den einzelnen Subsystemen der darunter liegenden Ebene dar. Somit repräsentiert jede Ebene der Hierarchie eine Klasse von Kommunikationsgeschwindigkeiten. Je höher die Hierarchieebene, desto langsamer die Kommunikationsverbindungen zwischen den PEs.

Ein definiertes Ziel der Lastverteilung ist, schnelle Verbindungen zu bevorzugen. Um dieses zu gewährleisten, kann die hierarchische Struktur dahingehend genutzt werden, dass Lastverteilung innerhalb von Subsystemen auf möglichst niedriger Ebene durchgeführt wird. Dieses setzt die Annahme voraus, dass seitens der Anwendung nicht alle PEs am Lastverteilungsprozess beteiligt sein müssen.

Ein Lastgleichgewicht des Systems kann oftmals hergestellt werden, wenn lediglich in einem Subsystem, anstatt auf dem kompletten System, Lastverteilung durchgeführt wird. Aufgabe ist es also, einen Teilbaum auf niedrigster Hierarchiestufe zu finden, durch den ein allgemeines Lastgleichgewicht hergestellt werden kann, indem seine zugehörigen PEs ihre Last untereinander umverteilen. In dem Beispiel aus Abbildung 4.2 reicht es aus, wenn zwei PEs Last untereinander austauschen, um ein Gesamtgleichgewicht zu erreichen.

Hierarchieerkennung

Die Basis bei der Erstellung der Hierarchiestruktur bilden die Kommunikationsgeschwindigkeiten zu anderen PEs. Dabei ist es unerheblich, wie die Struktur

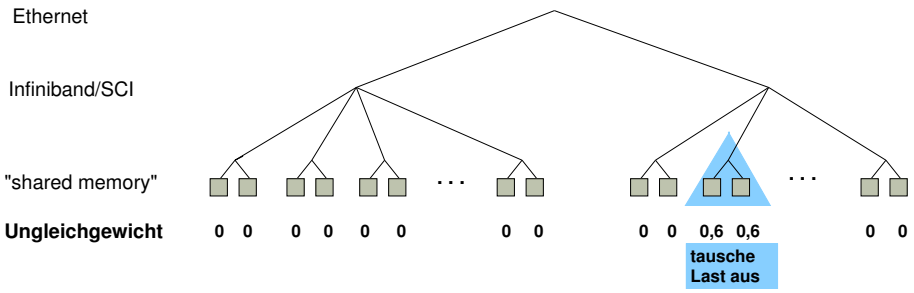


Abbildung 4.2: Wiederherstellung des Lastgleichgewichts

der Hardware aufgebaut ist. Es werden keine Annahmen über Konfigurationen gemacht. Stattdessen basieren alle Entscheidungen auf Messungen, die während der Laufzeit stattfinden.

Die Erkennung der hierarchischen Struktur erfolgt verteilt. Jedes PE arbeitet aus seiner lokalen Sicht heraus. Eine zentrale Instanz ist nicht vorhanden.

Zunächst werden alle PEs, zu denen die schnellsten Kommunikationsverbindungen bestehen, zu einem (Basis-) Subsystem zusammengefasst. Die nächste Hierarchieebene wird aus PEs mit den zweithöchsten Geschwindigkeiten gebildet, usw. Die letzte Ebene umfasst alle PEs des Systems. In Bezug auf Lastverteilung kann somit sichergestellt werden, dass stets ein Gleichgewicht erzielt werden kann, da im schlechtesten Fall alle PEs beteiligt werden.

Für die Ermittlung der hierarchischen Struktur werden folgende Phasen durchlaufen (siehe Abbildung 4.3):

1. Ermittlung potenzieller Subsystempartner: Berechnung einer Kandidatenliste
2. Ergebnisaustausch der PEs untereinander
3. Bestimmung der endgültigen Subsystempartner

Berechnung der Kandidatenliste

Die Klassenaufteilung erfolgt anhand einer aufsteigend sortierten Liste aller Kommunikationszeiten. Für die unterste Ebene bildet jedes PE_k eine Kandidatenliste für ein Subsystem $Kand_k^1$ aus allen PEs $PE_j, 0 \leq j,k < n$ mit Kommunikationszeiten $t(PE_k, PE_j) \in \mathbb{R}^+$, durch:

Sei

$$\begin{aligned} T_{com}^k \quad &= (t(PE_k, PE_{i_1}), ..., t(PE_k, PE_{i_{n-1}})), \\ &\quad wobei \;\; t(PE_k, PE_{i_j}) \leq t(PE_k, PE_{i_{j+1}}) \\ &\forall\ 1 \leq j < n-1,\ i_j \neq i_r,\ falls \;\; i \neq r, i_j \neq k, \end{aligned}$$

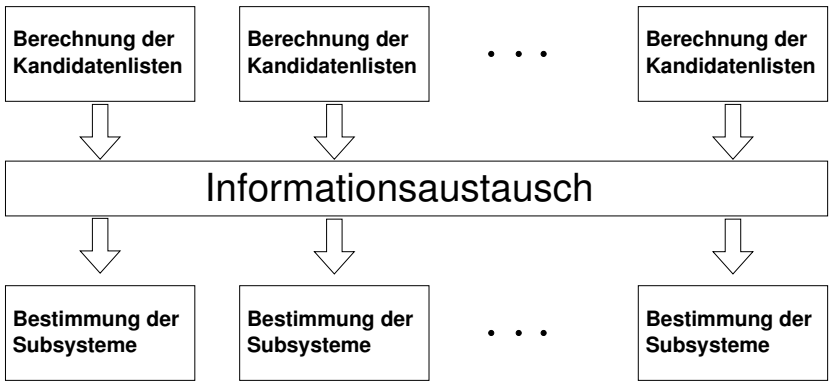


Abbildung 4.3: Phasen der Strukturermittlung

(4.7)

Sei weiterhin $DIFF\_TOLERANCE$ ein fester, aber beliebiger Wert größer als 1. Sei $D = DIFF\_TOLERANCE$.

Dann ist eine Kandidatenliste $Kand_k^1$ definiert durch:

$$Kand_k^1 = \{PE_k\} \cup \{PE_{i_j} \mid 1 \leq j < r, \tag{4.8}$$

$$mit \quad \left\{ \begin{array}{ll} & r = n, \ falls \ \frac{t(PE_k, PE_{i_s+1})}{t(PE_k, PE_{i_s})} < D \ \ \forall \ 1 \leq s < n-1 \\ oder & r = min\{s | 1 \leq s < n-1, \ \frac{t(PE_k, PE_{i_s+1})}{t(PE_k, PE_{i_s})} \geq D, i_j \neq k\} \end{array} \right\}$$

Durch die Verhältnisbildung zweier aufeinanderfolgender Kommunikationszeiten werden große Unterschiede zwischen diesen Zeiten aufgedeckt. Nur PEs, deren Kommunikationszeiten um einen kleinen Faktor (= *DIFF_TOLERANCE*) abweichen, gehören somit zu der Kandidatenliste.

Damit besteht eine Kandidatenliste für ein Subsystem aus PEs mit den geringsten Kommunikationszeiten. Das Verhältnis des PEs mit der höchsten Kommunikationszeit innerhalb des Subsystems zu demjenigen mit der geringsten Kommunikationszeit ausserhalb des Subsystems ist größer als der Wert von *DIFF_TOLERANCE*. Dieser Wert kennzeichnet also einen Sprung in der sortierten Liste der Kommunikationszeiten, und damit auch die nächste Ebene an Kommunikationszeiten.

Für die Erkennung der höheren Hierarchiestufen werden nicht die einzelnen PEs, sondern die Subsysteme betrachtet. Dabei wird jedes Subsystem durch ein beliebiges, aber festes PE repräsentiert. Nur diese PEs sind an der Subsystembildung für die aktuelle Ebene beteiligt.

Seien die Kommunikationszeiten $t(Sub_k^l, Sub_j^l)$ zwischen Subsystemen auf Ebene l gegeben. Sei n_l die Anzahl der Subsysteme auf Ebene l.

Sei

$$\begin{aligned} T_{com}^k \quad &= (t(Sub_k^l, Sub_{i_1}^l), ..., t(Sub_k^l, Sub_{i_{n_l-1}}^l)), \\ &wobei \;\; t(Sub_k^l, Sub_{i_j}^l) \leq t(Sub_k^l, PE_{i_{j+1}}^l) \\ &\forall\, 1 \leq j < n_l - 1,\; i_j \neq i_r,\; falls \;\; i \neq r, i_j \neq k, \end{aligned}$$

Dann ist eine Kandidatenliste für ein Subsystem höherer Ebene $(l+1)$ $Kand_m^{l+1}$ definiert durch:

$$Kand_k^{l+1} = \{Sub_k^l\} \cup \{Sub_{i_j}^l \;| 1 \leq j < r, \tag{4.9}$$

$$mit \quad \left\{ \begin{array}{c} r = n_l,\; falls \;\; \frac{t(Sub_k^l, Sub_{i_s+1}^l)}{t(Sub_k^l, Sub_{i_s}^l)} < D \;\; \forall\, 1 \leq s < n_l - 1 \\ oder \quad r = min\{s | 1 \leq s < n_l - 1,\; \frac{t(Sub_k^l, Sub_{i_s+1}^l)}{t(Sub_k^l, Sub_{i_s}^l)} \geq D, i_j \neq k\} \end{array} \right\}$$

Da auf diese Weise in jedem Schritt der Subsystembildung mindestens zwei Systeme zusammengefasst werden, terminiert dieses Verfahren.

Durch diese Methode werden Größenordnungen von Kommunikationszeiten herausgestellt. Netzwerkverbindungen, die um ein Vielfaches schneller sind als andere, werden von weniger leistungsstarken abgegrenzt. Diese Abgrenzung erfolgt durch den Wert *DIFF_TOLERANCE*.

Ergebnisaustausch

Da die Berechnung verteilt erfolgt, bestimmen die PEs die Subsysteme aus ihrer eigenen Sicht heraus. Es muss jedoch gewährleistet sein, dass die Hierarchie auf jedem PE in gleicher Weise aufgebaut wird. Das liegt darin begründet, dass alle an Lastverteilung beteiligten PEs Lastverteilung zur gleichen Zeit initiieren müssen. Dieses erfordert die Kenntnis über ihre Subsystemmitglieder.

Die Tatsache, dass die Bestimmung der Subsysteme nicht auf absoluten Geschwindigkeiten, sondern auf deren Verhältnisse zueinander basiert, und diese abhängig von der Größe des ersten Elements in der Liste der Kommunikationszeiten sind, kann zu unterschiedlichen Listen führen (siehe Abbildung 4.4).

Beispiel 4.3.2.1:

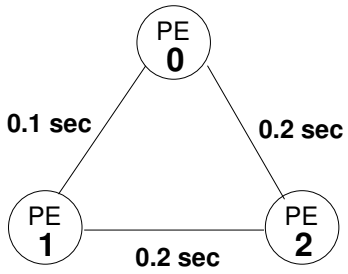


Abbildung 4.4: Gemessene Kommunikationszeiten in einem Beispielsystem

Die Listen der Kommunikationsgeschwindigkeiten sehen für Beispiel 4.3.2.1 wie folgt aus:

T^0_{com} : (0,1 0,2)
T^1_{com} : (0,1 0,2)
T^2_{com} : (0,2 0,2)

Der erste Wert der Liste von PE_2 (= 0,2) ist wesentlich höher als der der anderen PEs. Er entspricht vielmehr ihren zweiten Werten. Dieses zeigt, dass PE_2 auf unterster Ebene keine Subsystempartner hat. Da die globale Sicht für die einzelnen PEs nicht existiert, d. h. PE_2 verfügt über keine Informationen über die Werte der anderen PEs, müssen aus den lokalen Informationen die Subsysteme entsprechend ihres Kommunikationsverhaltens gebildet werden.

Die obigen Listen führen bei $DIFF\_TOLERANCE < 2$ zu folgenden Kandidatenlisten:

$Kand^1_0$: $\{PE_0, PE_1\}$
$Kand^1_1$: $\{PE_0, PE_1\}$
$Kand^1_2$: $\{PE_0, PE_1, PE_2\}$

Hierbei ist die eigene PE-Nummer immer Bestandteil der Liste.

Um in solchen Fällen zu einem einheitlichen Ergebnis zu gelangen, wird eine Berechnung der Subsystemzusammenstellungen durchgeführt.

Hierfür müssen jedem PE alle Kandidatenlisten vorliegen. Dieses erfordert einen Datenaustausch der PEs untereinander und damit eine kollektive Operation. Dieser Abgleich der Ergebnisse ist zwingend erforderlich, um identische Resultate aller PEs zu gewährleisten und damit eine Voraussetzung für eine spätere Lastverteilung zu schaffen.

Bestimmung der Subsystempartner

Nach dem Austausch der Listen erfolgt die Berechnung wiederum verteilt.

Dabei werden zunächst alle eindeutig erkannten Subsysteme als solche in die Hierarchie aufgenommen. Dies bedeutet, dass alle in der Kandidatenliste auftretenden PEs über die gleichen Listen verfügen (in Beispiel 4.3.2.1 die Liste von PE_0 und PE_1).

Werden auf diese Art und Weise nicht alle PEs erfasst, so werden die Subsysteme auf Basis von Schnittmengenbildung der Ergebnislisten bestimmt.

Die am häufigsten auftretenden Schnittmengen bilden die neuen Subsysteme. Kann ein Element nicht zugeordnet werden, dann bildet es ein Subsystem mit nur einem Element (isolierter Knoten). In Beispiel 4.3.2.1 bildet PE_2 auf unterster Ebene ein Subsystem mit nur einem Knoten.

Durch das Prinzip der Schnittmengenbildung von Kandidatenlisten werden nur PEs zusammengefasst, zwischen denen die für die aktuelle Ebene schnellsten Verbindungen bestehen.

Das Vorgehen zeigt das folgende Beispiel:

Beispiel 4.3.2.2:

Aufgrund der Kommunikationszeiten wurden folgende Kandidatenlisten ermittelt:

$$\begin{array}{ll} Kand_0^1: & \{PE_0,\, PE_2,\, PE_3\} \\ Kand_1^1: & \{PE_1,\, PE_2,\, PE_3\} \\ Kand_2^1: & \{PE_0,\, PE_1,\, PE_2,\, PE_3\} \\ Kand_3^1: & \{PE_0,\, PE_1,\, PE_2,\, PE_3\} \end{array}$$

Die Verbindung zwischen PE_0 und PE_1 ist um Faktoren langsamer als die zu den anderen Knoten. Daher tauchen diese Elemente nicht gegenseitig in ihren Listen auf. Bildet man die Schnittmenge über alle Kandidatenlisten, so ist

$$Kand_0^1 \cap Kand_1^1 \cap Kand_2^1 \cap Kand_3^1 = \{PE_2, PE_3\}.$$

PE_2 und PE_3 haben die schnellsten Verbindungen zueinander und können zu einem Subsystem kombiniert werden. Dieses ist jedoch sehr klein. Für die Lastverteilung ist es vorteilhafter, die Größe eines Subsystems zu maximieren, da dann mehr PEs bei einer potenziellen Lastumverteilung zur Verfügung stehen. So kann die obige Liste durch die Elemente PE_0 oder PE_1 erweitert werden, ohne dass die Regel zur Subsystembildung verletzt wird. Um dieses zu erreichen, werden Schnittmengen zwischen jeweils zwei Kandidatenlisten gebildet. Die am häufigsten auftretenden Schnittmengen bilden dann neue Subsysteme.

Zu Beispiel 4.3.2.2:

Schnittmenge	$Kand_0^1$	$Kand_1^1$	$Kand_2^1$	$Kand_3^1$
$Kand_0^1$	–	$\{PE_2, PE_3\}$	$\{PE_0, PE_2, PE_3\}$	$\{PE_0, PE_2, PE_3\}$
$Kand_1^1$	–	–	$\{PE_1, PE_2, PE_3\}$	$\{PE_1, PE_2, PE_3\}$
$Kand_2^1$	–	–	–	$\{PE_0, PE_1, PE_2, PE_3\}$

Da die Operation kommutativ ist, muss nur die obere Hälfte der Tabelle angegeben werden.

Die am häufigsten auftretenden Schnittmengen sind $\{PE_0, PE_2, PE_3\}$ und $\{PE_1, PE_2, PE_3\}$ (je zwei Mal). Wird die erste Menge gewählt, so entsteht ein Subsystem aus den PEs PE_0, PE_2 und PE_3. PE_1 bleibt isoliert und bildet sein eigenes Subsystem (siehe Abbildung 4.5).

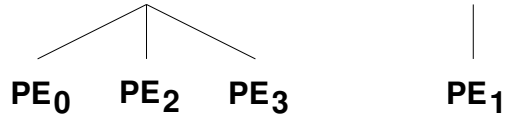


Abbildung 4.5: Ergebnishierarchie zu Beispiel 4.3.2.2

Subsystembildung auf höherer Ebene

Auf unterster Ebene sind alle PEs am Prozess der Subsystembildung beteiligt. Auf höheren Ebenen übernimmt ein beliebiges, aber festes PE jedes Subsystems die Funktion eines Masterknotens. Beim Erkennungsprozess der Hierarchie auf nicht unterster Ebene werden nur zwischen diesen Masterknoten Entscheidungen über neue Subsystembildungen getroffen. Nur die Masterknoten gehören zur Struktur der nächst höheren Ebene. Jedes PE hat somit nur Informationen über jene Subsysteme, zu denen es zugehörig ist.

Wird das Beispiel 4.3.2.2 um weitere Subsysteme auf unterster Ebene erweitert, so könnte es folgendermaßen aussehen:

Beispiel 4.3.2.3:

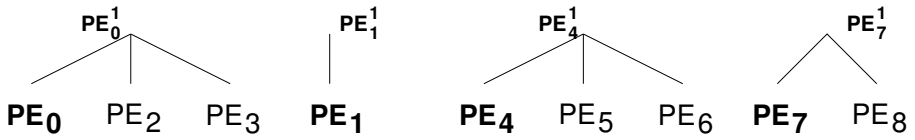


Abbildung 4.6: Erweiterung des Beispiels 4.3.2.2

Alle PEs kennen die Mitglieder ihrer Subsysteme. Die PEs PE_0, PE_1, PE_4 und PE_7 sind Masterknoten (hier gekennzeichnet durch „^{1} “) und sind an der Bildung der nächsten Hierarchieebene beteiligt. Diese erfolgt wie bereits für die unterste Ebene beschrieben nach Bildungsvorschrift 4.9.

Angenommen, die Masterknoten PE_0^1 und PE_1^1 sowie PE_4^1 und PE_7^1 bildeten jeweils Subsysteme auf der nächsten Ebene (Ebene zwei), dann verfügen nur diese PEs über Informationen über diese neuen Subsysteme. Ihre Subsystempartner auf Ebene eins haben keinerlei Informationen darüber.

Dieses hat zum einen den Vorteil, dass die nicht beteiligten PEs (PE_2, PE_3, PE_5, PE_6, PE_8), andere Aufgaben lösen können (z. B. Rechnungen im Rahmen der Anwendung), und zum anderen, dass sich die Anzahl an notwendigen Kommunikationen von Ebene zu Ebene verringert.

Die Hierarchie sieht nun folgendermaßen aus:

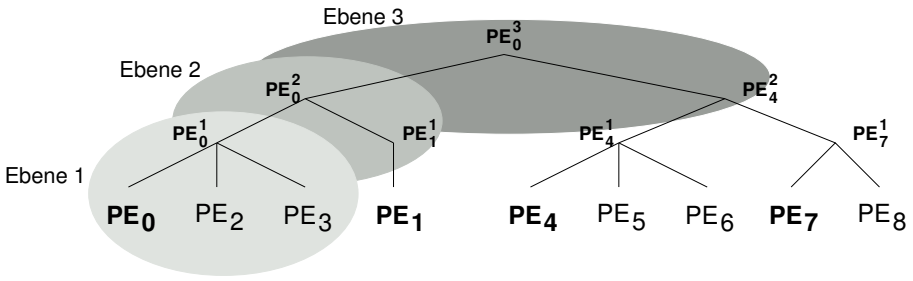


Abbildung 4.7: Erweiterung des Beispiels 4.3.2.2

Hierarchieaktualisierung

Während der Laufzeit können sich Kommunikationsgeschwindigkeiten stark verändern. Die bestehende hierarchische Struktur repräsentiert dann nicht mehr die aktuelle Netzwerksituation. Sie muss daher an die aktuell vorhandenen Messwerte angepasst werden. Damit reagiert das System auf sich verändernde Netzwerkstrukturen bzw. -qualitäten.

Es gibt mehrere Fälle, bei denen die hierarchische Struktur aufgrund von veränderten Kommunikationsgeschwindigkeiten angepasst werden muss:

1. Subsysteme müssen zu einem Subsystem zusammengefasst werden (siehe Abbildung 4.8), da die gemessenen Geschwindigkeiten zwischen zwei oder mehreren Subsystemen jenen innerhalb der Subsysteme ähneln (z. B. durch Ausfall oder Umschalten von Netzwerken).

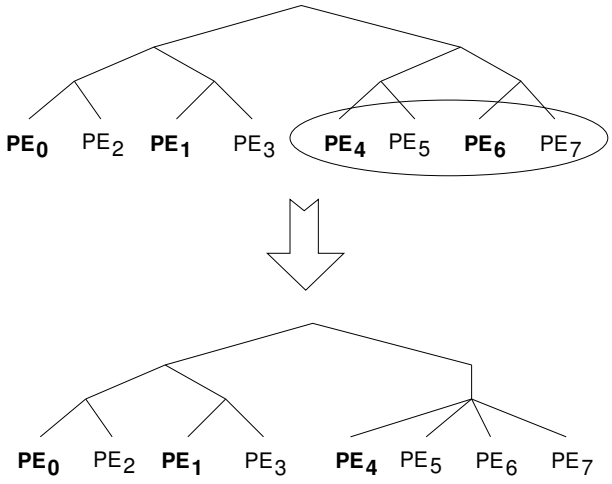


Abbildung 4.8: Zusammenfassen zweier Subsysteme

2. Ein oder mehrere PEs bzw. Subsysteme müssen zu anderen Subsystemen wandern (siehe Abbildung 4.9), da sich die Kommunikationsgeschwindigkeiten zwischen einzelnen PEs oder aber Subsystemen verändert haben (z. B. durch Netzwerkstörungen).

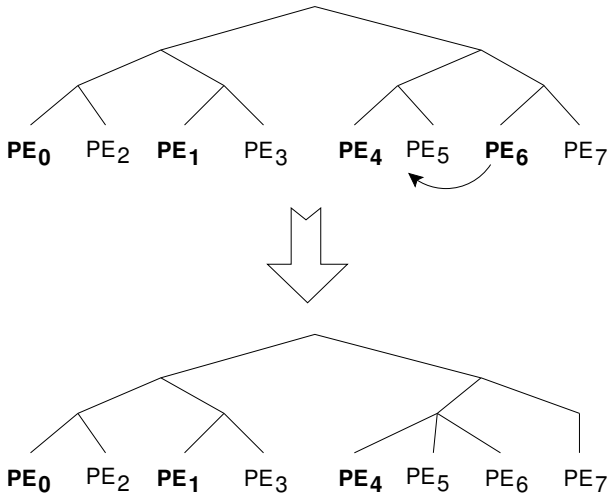


Abbildung 4.9: Wechsel zu einem anderen Subsystem

3. PEs oder Subsysteme werden zu isolierten Knoten (siehe Abbildung 4.10).

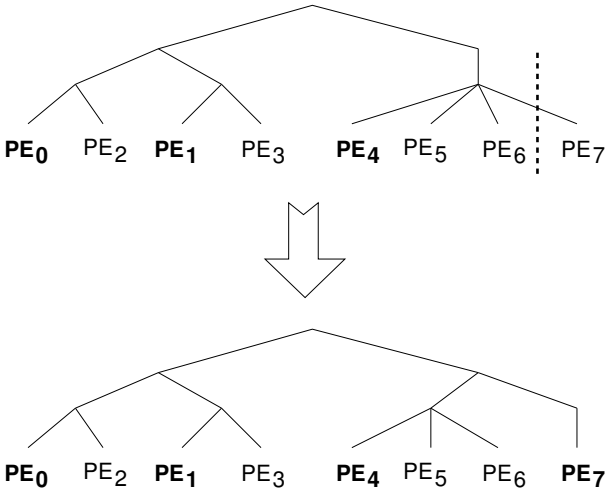


Abbildung 4.10: Bildung eines isolierten Knotens

Diese Situationen haben folgende Auswirkungen auf die Struktur der Hierarchie:

Fall 1: Müssen Subsysteme zusammengefasst werden, so sind die betroffenen PEs bzw. Subsysteme beteiligt. Die Hierarchiestruktur muss nur auf der betroffenen Ebene verändert werden. Um diese Änderungen jedoch aufzudecken, muss jede Ebene überprüft werden. Dafür ist es stets notwendig:

1. eine Kandidatenliste für die aktuelle Ebene zu bestimmen
2. diese mit der Kandidatenliste aus der letzten Aktualisierung zu vergleichen

Dieses entspricht einem deutlichen Mehraufwand zu dem des Hierarchieaufbaus, da hier zur Bestimmung der Kandidatenliste noch der Vergleich hinzukommt.

Fall 2: Müssen Subsysteme bzw. PEs verschoben werden, so hat dieses nicht nur Auswirkungen auf die betroffenen Subsysteme selber, sondern möglicherweise auch auf die komplette hierarchische Struktur (siehe Abbildung 4.11).

Fall 3: Wird ein Knoten isoliert, so kann sich seine eigene Subsystemstruktur, die des vorherigen Masters und die des Masters auf höherer Ebene ändern. Bei starken Veränderungen der Kommunikationszeiten, können sich diese Modifikationen durch den ganzen Baum ziehen. So kann der isolierte Knoten z. B. auch direkter Nachfolger der Wurzel werden.

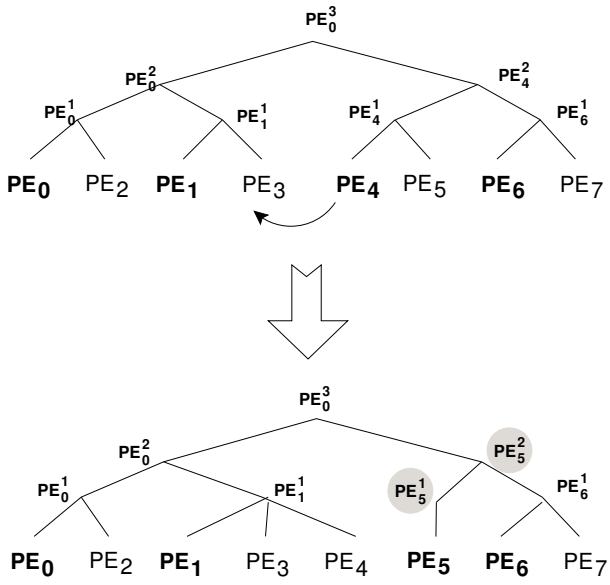


Abbildung 4.11: Strukturänderung durch Verschiebung eines Elements

In allen diesen Fällen ist es möglich, dass die Struktur auf jeder Ebene geändert bzw. überprüft werden muss. Da dieses einen größeren Aufwand gegenüber dem eines Neuaufbaus darstellt, wird statt einer Aktualisierung ein Neuaufbau der Hierarchie durchgeführt.

Die hierarchische Struktur kann nun dazu benutzt werden, bei einem bestehenden Ungleichgewicht geeignete Lastverteilungspartner zu finden.

4.3.3 Lokalisierung geeigneter Lastverteilungspartner

Ist ein Ungleichgewicht entdeckt worden, dann erfolgt die Suche nach den Lastverteilungspartnern, wobei deren Verbindungen zueinander möglichst schnell sein sollen. Das Ziel ist, ein Subsystem niedrigster Hierarchiestufe zu lokalisieren, durch dessen Re-Balancierung ein Lastgleichgewicht des Gesamtsystems erzielt werden kann. Das ist dann der Fall, wenn die Gesamtlast dieses Subsystems seiner optimalen Last entspricht, dieses aber nicht für seine Teilsysteme gilt.

Die optimale Last eines Subsystems Sub_j^l ($optimale\ Last_{Subj^l}$) wird durch Aufsummieren der einzelnen optimalen Lastwerte $optimale\ Last_{Sub_k^{l-1}}$ der zugehörigen Subsysteme, bzw. auf unterster Ebene PEs, bestimmt, d. h.

$$optimale\ Last_{Sub_j^l} = \sum_{Sub_k^{l-1} \in Sub_j} optimale\ Last_{Sub_k^{l-1}} \tag{4.10}$$

Wird ein Lastungleichgewicht lokal auf einem PE PE_i festgestellt (siehe Gleichung 4.6), so schickt PE_i eine Anfrage an seinen Masterknoten PE_j. Dieser überprüft daraufhin das Gleichgewicht seines Subsystems Sub_j^1.

- Besteht kein Ungleichgewicht, so ist der gesuchte Teilbaum gefunden. Innerhalb dieses Teilbaums wird nun das Ungleichgewicht jedes Subsystems überprüft. Alle PEs, die zu einem unbalancierten Subsystem gehören, bilden die Lastverteilungspartner für den aktuellen Zyklus.

- Besteht jedoch ein Ungleichgewicht in dem Subsystem Sub_j^l, so schickt PE_j eine Anfrage an seinen Masterknoten auf höherer Ebene $(l+1)$.

  Dieses Prozedere wird solange durchgeführt, bis ein balanciertes Subsystem gefunden wurde. Spätestens bei der Wurzel, die alle PEs des Systems repräsentiert, wird ein Gleichgewicht gefunden, da das Gesamtsystem stets in sich balanciert ist. Die aktuelle Last des Gesamtsystems entspricht immer seiner optimalen Last und damit ist:

$$Ungleichgewicht_{Wurzel} = |1 - \frac{aktuelle\ Last_{gesamt}}{optimale\ Last_{gesamt}}| = |1-1| = 0 \tag{4.11}$$

Abbildung 4.12 zeigt ein Beispiel zur Ortung von für Lastverteilung geeigneten Subsystemen. PE_0 entdeckt ein Lastungleichgewicht von 0,3 und schickt eine Anfrage an seinen Masterknoten. In diesem Falle ist PE_0 selbst Repräsentant auf nächst höherer Ebene (gekennzeichnet durch PE_0^1). PE_0^1 überprüft das Gleichgewicht seines Subsystems auf Ebene eins und stellt fest, dass ein Ungleichgewicht von 0,1 besteht. Es schickt ebenfalls eine Anfrage an den Masterknoten der nächst höheren Ebene (PE_0^2). Dieser stellt fest, dass sein Subsystem balanciert ist, d. h. die Gesamtlast seines Subsystems ist gleich seiner optimalen Last.

PE_3 entdeckt ebenfalls ein Ungleichgewicht (0,3) und stellt eine Anfrage an PE_4^1, das diese weiterreicht. Auf Ebene zwei wird festgestellt, dass innerhalb dieses Subsystems Lastverteilung erfolgen muss. Die unbalancierten Teilbäume mit den PEs PE_0, PE_5, PE_2, PE_4 und PE_3 können durch Austausch ihrer Last ein Gleichgewicht erzielen.

PE_7 und PE_8 bilden eine zweite Lastverteilungspartition, die von der ersten jedoch gänzlich unabhängig ist.

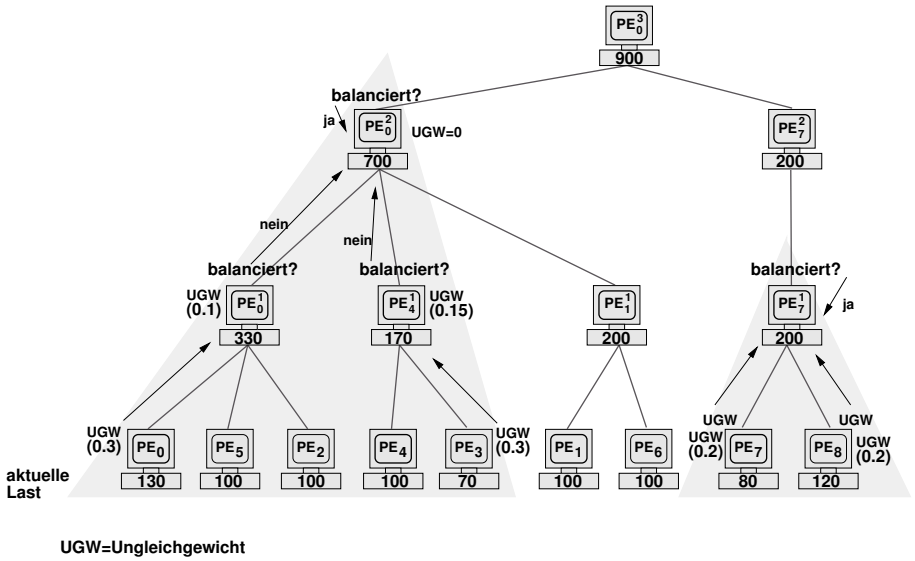


Abbildung 4.12: Lokalisierung von Lastverteilungspartnern

4.4 Leistungsmessungen

Als Basis für das Erkennen der Hierarchie zum einen und für die Bestimmung der Lastmenge je PE zum anderen, dienen Kommunikations- bzw. Berechnungsgeschwindigkeit. Diese werden aus Messwerten berechnet, die zur Laufzeit ermittelt wurden. Dadurch wird Aktualität gewährleistet.

Die Leistungsmessungen begründen sich auf die in Kapitel 3.2 beschriebenen Grundsätze Skalierbarkeit, Effizienz und Verfügbarkeit auf Anwenderebene.

Skalierbarkeit wird ermöglicht, indem das Monitoring vollständig verteilt erfolgt. Jedes PE führt das eigene Monitoring durch. Die Daten werden lokal gesammelt. Werte, die auf diesen Messergebnissen basieren, werden lokal berechnet.

Verfügbarkeit ist dadurch gegeben, dass innerhalb der Anwendung Leistungsmessungen durchgeführt werden. Auf Informationen des Betriebssystems wird nicht zugegriffen. Die Daten, die die Anwendung ermittelt, können auch von dieser genutzt werden.

Effizienz bedeutet, dass hinreichende Informationen mit möglichst wenig Overhead erzeugt werden.
Hinreichende Informationen bedeuten für den Netzwerkbereich ein Ermitteln von Größenordnungen von Kommunikationsgeschwindigkeiten im Hinblick auf die Anwendung. Die Messwerte dienen der Bildung der Subsysteme, werden also lediglich für die Aufteilung in unterschiedliche Ge-

schwindigkeitsklassen benötigt. Dieses erfolgt durch die Messung von Nachrichten an Synchronisationspunkten der Anwendung.
Für die Leistungsfähigkeit der Recheneinheiten müssen Berechnungsgeschwindigkeiten der Lasteinheiten bestimmt werden. Messen von Rechenphasen der Anwendung liefert die notwendigen Informationen.

Die Feststellung von Berechnungs- und Kommunikationsgeschwindigkeit erfolgt unabhängig voneinander und wird im Folgenden detailliert beschrieben.

4.4.1 Ermittlung der Berechnungsgeschwindigkeit

Die Leistungsfähigkeit einer Recheneinheit wird im Hinblick auf die Anwendung bestimmt. Diese dient dazu, die Last entsprechend dieser Leistungsfähigkeit auf die einzelnen PEs zu verteilen. Was benötigt wird, ist daher ein Maß für die Leistungsfähigkeit, auf dessen Grundlagen entschieden werden kann, wieviel Last innerhalb welcher Zeit bewältigt werden kann.

Da Lastverteilung in gitterbasierten Anwendungen die Menge der Zellen verteilt, bedarf es einer Aufwandsbestimmung je Zelle. In diesem Verfahren wird daher bestimmt, wie lange ein PE für die Berechnung einer Zelle benötigt. Diese Rechengeschwindigkeit zeigt nicht nur die Kapazitäten der CPU, sondern bezieht auch Faktoren wie Speicherauslastung mit ein. So erhöht sich z. B. die Laufzeit einer Anwendung, wenn der Hauptspeicher dermaßen belegt ist, dass in den Swap-Bereich ausgelagert werden muss.

Um die Rechengeschwindigkeit je Zelle zu bestimmen, wird zunächst die Berechnung mehrerer Zellen gemessen und anschließend diese Gesamtzeit durch die Anzahl der Zellen geteilt. Dieses Vorgehen erfüllt die fünfte Forderung Gropps, sehr kurze Zeitereignisse zu berücksichtigen. Der Aufwand für die Berechnung einer Zelle liegt im Millisekundenbereich, zu kurz, um aussagekräftige Messungen zu erhalten. Weiterhin wird der Ansatz damit der Forderung nach Effizienz gerecht. Anstatt vieler kurzer Zeiten werden wenige lange erfasst. Der Overhead wird dadurch minimal gehalten.

Ein wichtiger Aspekt ist die Aktualität der Messwerte, um auf Änderungen der für die Anwendung vorhandenen Leistungskapazität reagieren zu können. Dieses betrifft die Nutzung der Ressource durch andere Prozesse. Um jedoch abzufangen, dass kurzzeitige Auslastungsspitzen das Bild verfälschen, werden Messwerte über einen bestimmten Zeitraum gemittelt. Dadurch wird erreicht, dass die zu erwartende Leistungsfähigkeit, die für die Lastverteilung ausschlaggebend ist, erfasst wird.

4.4.2 Ermittlung der Kommunikationsgeschwindigkeit

Das Ausnutzen von Synchronisationspunkten der Anwendung bildet eine wichtige Basis für das Monitoring. Somit wird das Laufzeitverhalten der Anwendung nicht beeinflusst.

Synchronisationspunkte sind durch das Ablaufschema von FEM-Anwendungen vorgegeben. Wie in Abschnitt 2.3 beschrieben, folgen Rechenphasen jeweils Phasen des Datenaustausches und damit auch Kommunikation. Diese Kommunikation betrifft zum einen den Datenaustausch von benachbarten PEs, um Ergebnisse der Randzellen auszutauschen. Zum anderen werden diese Phasen auch für Lastverteilung genutzt. Viele FEM-Anwendungen benötigen auch den Austausch globaler Informationen. Dieses bedingt Kommunikation, an denen alle PEs beteiligt sind (kollektive Kommunikation). Diese kollektiven Kommunikationen stellen Synchronisationspunkte dar und werden genutzt, um Leistungsmessungen des Netzwerks durchzuführen.

Durch den Mangel einer globalen Uhr in einer verteilten Umgebung, wurde die Methode der Ping-Pong-Messungen gewählt. Hierbei wird die Laufzeit einer Nachricht gemessen, die von einem Sender zum Empfänger und wieder zurück geschickt wird (siehe Abbildung 4.13). Um die Nachrichtenlaufzeit ($= T_3 - T_0$) zu messen, ist es in diesem Verfahren erforderlich, dass der Empfänger empfangsbereit ist und die Nachricht direkt weiterschicken kann, d. h. $T_2 - T_1 < \epsilon$.

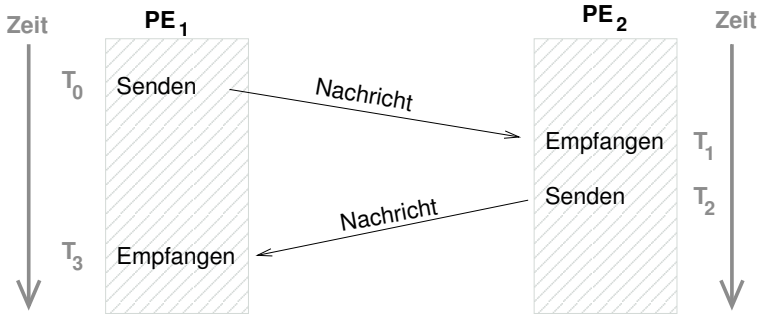


Abbildung 4.13: Ping-Pong-Schema

Um zu gewährleisten, dass die Empfangsbereitschaft gegeben ist, wird zu Beginn der Anwendungslaufzeit ein Kommunikationsplan berechnet, der die Reihenfolge der einzelnen Messungen wie auch die Kommunikationspartner bestimmt.

Anhand dieses Kommunikationsplans, der auf jedem PE berechnet wird, erkennt jedes PE, wann es mit welchem Partner kommunizieren muss.

Soll eine Messung zwischen den Kommunikationspartnern PE_i und PE_j stattfinden, so wird der erste Ping-Pong-Lauf nicht gemessen. Damit wird Gropp-Forderung Nummer eins, die erste Messung nicht zu berücksichtigen, erfüllt. Danach werden für eine Messung mehrere Nachrichtenläufe durchgeführt.

Wird eine Kommunikationsverbindung von PE_i zu PE_j gemessen, so initiiert nur einer der beiden PEs die Messung. Es wird also entweder $PE_i \rightarrow$

$PE_j \rightarrow PE_i$ gemessen oder aber $PE_j \rightarrow PE_i \rightarrow PE_j$. Das nicht-messende PE erhält den Messwert dann vom Initiator. Diese Methode geht davon aus, dass die Kommunikation symmetrisch ist, eine Grundvoraussetzung für die Subsystembildung.
Weiterhin werden die Kosten, und damit der Overhead für die Lastverteilung, minimiert, indem nur die Hälfte der Kommunikationsverbindungen gemessen wird.

Die Nachrichtengröße richtet sich zum einen nach der Anwendung, zum anderen soll sie so klein wie möglich sein, um einen hohen Zusatzaufwand zu vermeiden. Dieses trifft auch für die Häufigkeit der Messungen zu. Eine Evaluierung der optimalen Nachrichtengröße und der geeigneten Abstände der Messungen findet sich in Abschnitt 6.4.3.

Der Kommunikationsplan

Die Ping-Pong-Messungen werden von allen Prozessen gleichzeitig aufgerufen. Ziel ist es, alle Kommunikationswege „abzutasten“. Dabei muss die Laufzeit so minimal wie möglich sein, um den Overhead für die Lastverteilung zu minimieren. Um alle Kommunikationsvarianten (d. h. welches PE kommuniziert mit welchem anderen PE) abzudecken und Kollisionen zu vermeiden, wird ein Kommunikationsplan berechnet. Anhand dieses Plans führt jedes PE zu einer entsprechenden Zeit ein Senden bzw. Empfangen von Nachrichten zu bzw.von einem definierten anderen PE aus.

Durch diesen Plan wird gewährleistet, dass das empfangende PE empfangsbereit ist und die Nachricht direkt zurückschicken kann. Die Ergebnisse der Messungen werden allerdings von der darunter liegenden Technologie beeinflusst. Bei verschalteten Netzwerken, wie z. B. Infiniband, behindern sich die Nachrichten nicht. Die Ergebnisse zeigen die Übertragungsgeschwindigkeit zwischen den beiden betreffenden PEs, unabhängig vom übrigen Netzwerkverkehr. In Torus-Topologien hingegen, wie sie z. B. in SCI-Netzwerken zu finden sind, sind die Nachrichtenlaufzeiten stark abhängig von der Gesamtauslastung des aktuellen Rings. Bei den hier gemessenen Kommunikationsgeschwindigkeiten fließt auch die Auslastung des Gesamtnetzes mit ein. Dieses Phänomen ist beabsichtigt, da an den Synchronisationspunkten alle PEs Informationen austauschen und damit auch die Gesamtnetzwerklast beeinflussen.

Die Berechnungsgeschwindigkeit, die der Bearbeitungszeit je Zelle entspricht, und die Kommunikationsgeschwindigkeit, die Nachrichtenlaufzeiten zwischen PEs angibt, bilden die Basis für die Umsetzung des Konzepts.

4.5 Integration in den Lastverteilungszyklus

In den vorangegangenen Abschnitten wurden die einzelnen Konzepte der kapazitäten- und netzwerkabhängigen Lastverteilung vorgestellt. Wie diese Me-

thoden in den Lastverteilungszyklus integriert werden können, ist Thema des folgenden.

Lastverteilung gliedert sich, wie in Kapitel 2.2 beschrieben, in mehrere Schritte:

1. **Entscheidungsphase:**

   (a) Wann wird Lastverteilung durchgeführt?
   Diese Entscheidung obliegt der Endanwendung. Nur diese verfügt über Informationen, wann sich die Lastmenge ändert. In Abhängigkeit von der Höhe des Ungleichgewichts kann diese sich für bzw. gegen eine Umverteilung der Last entscheiden.
   Wird Lastverteilung durchgeführt, so werden folgende Fragestellungen von einem bestehenden Lastverteilungswerkzeug (z. B. Jostle, ParMETIS) oder aber von einer anwendungseigenen Lastverteilungsmethode bearbeitet:

   (b) Wieviel Last muss migriert werden?

   (c) Welche Last muss migriert werden?

   (d) Wohin/Woher muss Last migriert werden?

2. **Durchführungsphase:** Datenmigration
   Hier erfolgt die Verschiebung der Last durch die Anwendung.

Durch die neu entwickelte Strategie erweitert sich dieser Lastverteilungszyklus wie folgt:

1. **Entscheidungsphase:**

   (a) Wann wird Lastverteilung durchgeführt?
   Wird Lastverteilung von der Anwendung initiiert, so wird diese Entscheidung weitergereicht und Folgendes bearbeitet:

      i. Entscheidung über die prinzipielle Durchführung von Lastverteilung in Abhängigkeit vom Ungleichgewicht

   Wird Lastverteilung durchgeführt, dann:

      ii. Aktualisierung der Hierarchie und der Lastwerte

      iii. Lokalisierung der Lastverteilungspartner

      iv. Aufbereitung der Eingabedaten für das Lastverteilungswerkzeug

   Aufruf des Anwendungslastverteilers:
   In dieser Phase wird der Anwendungslastverteiler (ALB) (z. B. Jostle) mit dem zuvor bestimmten Subsystem aufgerufen. Dabei wird dem ALB angegeben, wieviel Last jeder PE erhalten soll (abhängig von seiner Berechnungsgeschwindigkeit). Der ALB hat dann die Aufgabe, zu berechnen:

   (b) Welche Last muss migriert werden?

(c) Wohin/Woher muss Last migriert werden?

2. **Durchführungsphase:** Datenmigration

   Die Kontrolle wird nun wieder der Anwendung übergeben. Diese führt die Verschiebung der Last, basierend auf den Ergebnissen des Lastverteilers, und die Einbindung dieser auf dem Zielprozessor durch.

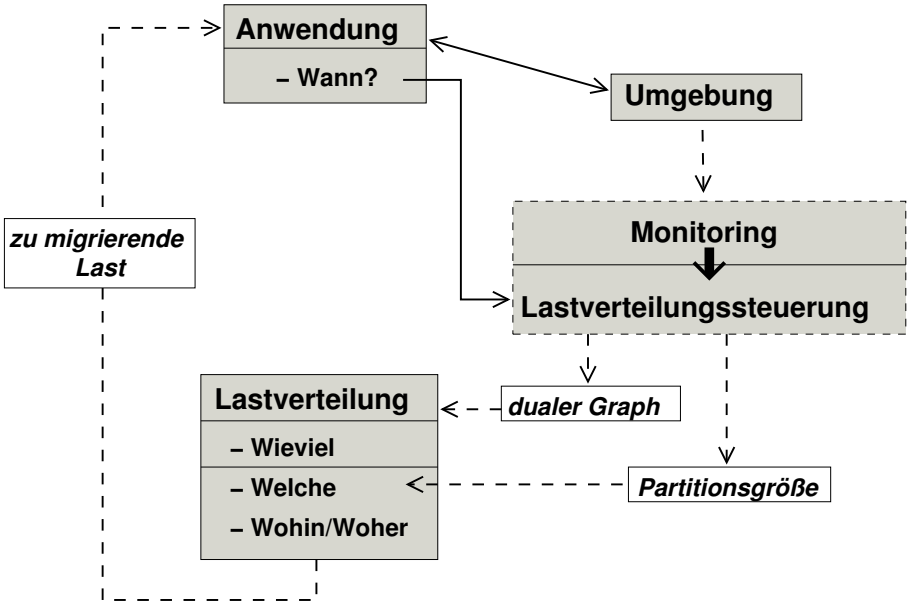


Abbildung 4.14: Platzierung des neuen Moduls

Abbildung 4.14 zeigt die Platzierung der Lastverteilungssteuerung. Die erforderliche Partitionsgröße soll nun durch die Steuerung bestimmt werden.

Die Umsetzung dieser Konzepte zusammen mit deren Integration in eine Anwendung ist Thema des nächsten Abschnitts.

Kapitel 5

Meta Load Balancer *mLB*

Die Konzepte aus dem letzten Kapitel wurden in dem Werkzeug *Meta Load Balancer* **mLB** umgesetzt. Die Entwicklungsziele sowie deren Umsetzung sind Thema dieses Kapitels.

5.1 Entwicklungsziele

Bedingt durch die Anforderungen der Anwendung zusammen mit den Gegebenheiten der Umgebung, mussten grundlegende Design-Richtlinien eingehalten werden. Diese ergaben sich aus folgenden Entwicklungszielen:

Skalierbarkeit: Die Effizienz einer parallelen Anwendung ist immer abhängig von ihrer Skalierbarkeit. Skalierbarkeit in einer Umgebung wie das Grid, in die Tausende an Ressourcen integriert sein können, kann nur eine verteilte Software ermöglichen.

Portabilität: Das Grid zeichnet sich nicht nur durch eine heterogene Rechenumgebung aus. Auch ist die zur Verfügung stehende Software nicht innerhalb jeden Zentrums gleich. Daher wurde nur Software eingesetzt, deren Verfügbarkeit sichergestellt ist.

Minimale Eingriffe in die Anwendung: Es darf nicht notwendig werden, bestehende Anwendungen in großem Umfang abändern zu müssen, um *mLB* einsetzen zu können. Ein einfaches Benutzen („easy to use“) von Software erhöht die Akzeptanz seitens der Anwender und damit auch ihren Einsatz.

(Gridfähigkeit): Hier geht es um softwaretechnische Möglichkeiten, die Anwendung in der Grid-Umgebung laufen zu lassen. Dieser Aspekt wird zwar berücksichtigt, ist aber nicht Aufgabe von *mLB*.

5.2 Architektur/Design

Die angestrebten Entwicklungsziele bestimmen das Design von *mLB*.

Skalierbarkeit wurde durch eine verteilte Architektur ermöglicht (siehe Abbildung 5.1). *mLB* läuft auf jedem Knoten der Anwendung. Entscheidungen werden verteilt getroffen. Lediglich wenige Informationsaustausche sind notwendig.

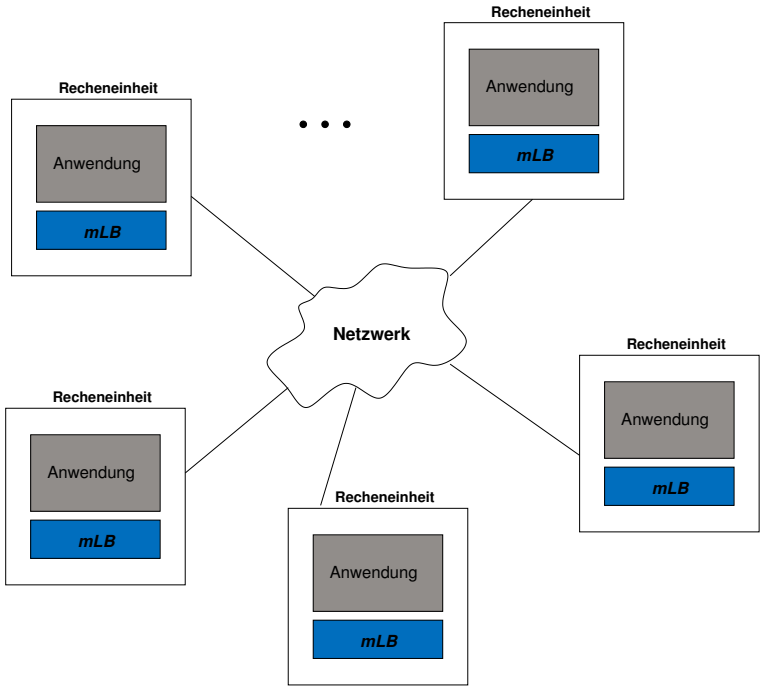


Abbildung 5.1: Verteilte Architektur von *mLB*

Portabilität: Ein heutiger Kommunikationsstandard bietet MPI (Message Passing Interface). Leistungsangaben von Netzwerktechnologien, wie z. B. Latenz, werden meist für die Kommunikation mit MPI angegeben (siehe auch Tabelle 2.1). Dieses ist ein Kennzeichen für seine starke Verbreitung.

MPI wurde entwickelt als ein Standard für Message-Passing-Funktionen, das ein Konsortium von Entwicklern und Hardware-Herstellern erstellte. Es beinhaltet die Definition von Kommunikationsroutinen. Die Spezifikation der Schnittstellen ermöglicht zum einen die Portierbarkeit von Anwenderprogrammen. Zum anderen können Hersteller und Entwickler eine effiziente, architekturspezifische Implementierung von Kommunikationsroutinen bereitstellen.

Die Kommunikation von MPI ist nur über Kommunikatoren möglich. Einem Kommunikator liegt eine Gruppe von Prozessen zugrunde. Bei Programmstart existiert eine einzige Gruppe, die alle Prozesse umfasst. Während der Laufzeit kann die Anwendung Prozessgruppen bzw. neue Kommunikatoren bilden. Damit ist es möglich, die Menge der Prozesse in Teilmengen zu zerlegen.

MPI bietet Routinen für Punkt-zu-Punkt-Kommunikation und kollektive Kommunikation, bei der alle Prozesse eines angegebenen Kommunikators beteiligt sind.

Eine Implementierung von MPI ist für jede Architektur verfügbar. Größtenteils gibt es für die jeweilige Netzwerktechnologie optimierte Versionen. MPI-Pakete gehören mittlerweile auch optional zu vielen Standard-Linux-Distributionen. Durch die freie Verfügbarkeit von MPICH, einer Implementierung des Argonne National Laboratory, besteht auch die Möglichkeit, diese Software selbst zu übersetzen.

MPICH verfügt über Schnittstellen für drei Programmiersprachen: C, C++ und Fortran. Für die Implementierung von *mLB* wurde die Sprache C gewählt. Fortran77 wurde nicht in Betracht gezogen, da es keine dynamische Speicherverwaltung unterstützt (siehe Abschnitt 6.5.1). Anwendungen, die in C++ geschrieben sind, können problemlos die C-Schnittstelle verwenden.

Minimale Eingriffe in die Anwendung: *mLB* nutzt das MPI-Profiling-Interface für die netzwerkabhängige Lastverteilung (siehe Abbildung 5.2). Mit Hilfe dieses Interfaces ist es bei Aufruf einer MPI-Routine seitens der Anwendung möglich, statt der entsprechenden MPI-Routine eine andere Funktion aufzurufen. Diese Möglichkeit wird von *mLB* genutzt, um Netzwerk-Monitoring anzustoßen.

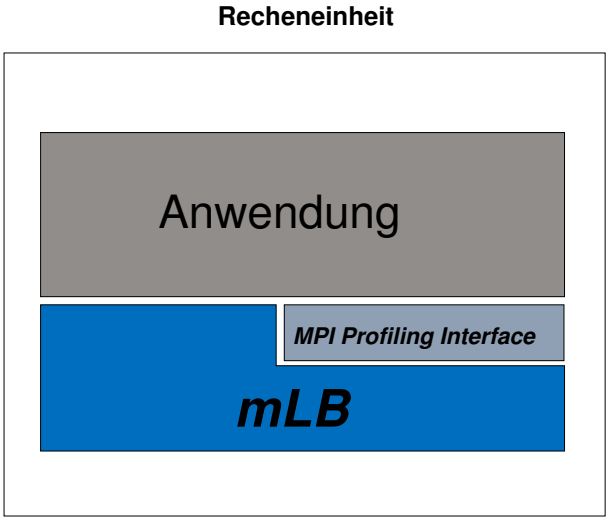


Abbildung 5.2: *mLB* mit MPI-Profiling Interface

Gridfähigkeit wurde durch den Einsatz von PACX-MPI [pac] realisiert. PACX-MPI ist eine Grid-Computing-Erweiterung von MPI. Es stellt die Kommunikation zwischen unterschiedlichen MPI-Implementierungen zur Verfügung. Damit können auf den einzelnen Systemen die optimierten MPI-Versionen genutzt werden, untereinander kommunizieren diese dann über PACX-MPI. Die Vorgehensweise von PACX-MPI zeigt Abbildung 5.3.

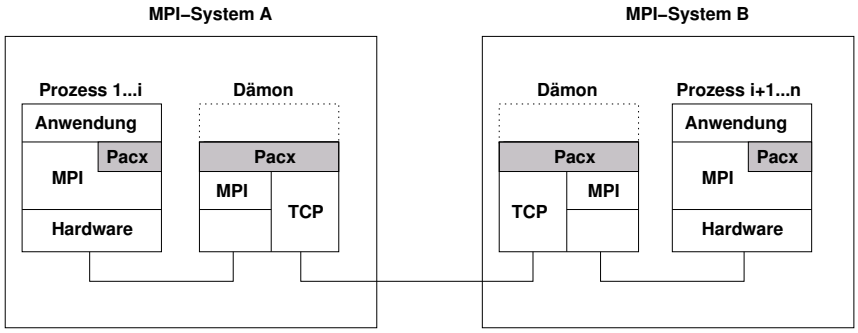


Abbildung 5.3: Kommunikationsprinzip von Anwendungen unter PACX-MPI (entnommen aus [pac])

5.2.1 Interne Architektur von *mLB*

mLB besteht aus zwei Monitoring-Komponenten und dem LBS-Modul (siehe Abbildung 5.4):

1. **Kommunikationsmonitor:** Dieser realisiert die netzwerkabhängige Lastverteilung und erhält seine Informationen aus der Umgebung.
2. **Anwendungsmonitor:** Dieser realisiert die kapazitätenabhängige Lastverteilung und erhält seine Informationen von der Anwendung.
3. **LBS- (Load Balancing Support, Lastverteilungssteuerungs-)Modul:** Dieses Modul verarbeitet die Informationen aus dem Kommunikationsmonitor und dem Anwendungsmonitor.

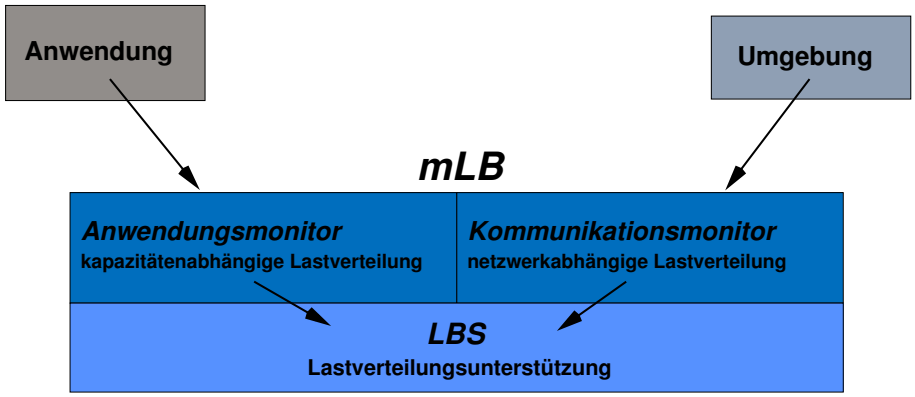


Abbildung 5.4: Internes *mLB* Design

5.2.2 Kommunikationsmonitor (KM)

Der KM beobachtet das Netzwerk, indem Kommunikationsgeschwindigkeiten gemessen werden. Er nutzt das MPI-Profiling-Interface. Damit ist er transparent für die Anwendung.

Der Aufruf einer kollektiven MPI-Routine aktiviert den KM. Anstatt ausschließlich die kollektive Kommunikation auszuführen, erfolgt nach Beendigung dieser Kommunikation eine Messung der Netzwerkverbindungen. Dieser Zeitpunkt ist prädestiniert für solche Messungen, da er einen Synchronisationspunkt darstellt. Der Ablauf der Anwendung ändert sich dadurch nicht.
Ein Beispiel dazu zeigt die Realisierung von Profiling der MPI-Barrier-Funktion:

Beispiel 5.1:

```
int MPI_Barrier(MPI_Comm kommunikator)
{
  ...
  result = PMPI_Barrier(kommunikator);
  Ping-Pong-Messungen();
  return result;
}
```

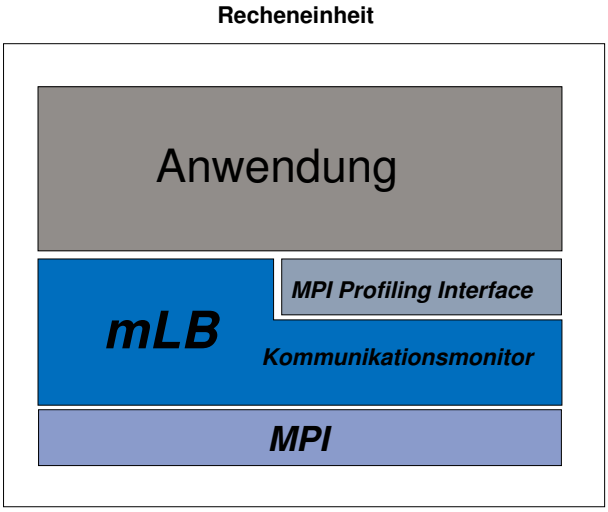


Abbildung 5.5: *mLB*-Architektur mit Kommunikationsmonitor

Der KM ist durch eine Bibliothek realisiert, die sowohl die Messung des Netzwerks über das MPI-Profiling-Interface als auch die Erkennung der hierarchischen Subsystemstruktur umfasst. Einfaches Hinzubinden dieser Bibliothek zum Objektcode der Anwendung ermöglicht die Nutzung des KM.

Der KM arbeitet autonom. Er ist nicht abhängig von den anderen Modulen. Durch seine Implementierung als Bibliothek kann er auch für andere Bereiche des Grid-Computings eingesetzt werden, bei denen die Systemstruktur eine wichtige Rolle spielt.

5.2.3 Anwendungsmonitor (AM)

Der AM erfasst anwendungsspezifische Daten. Dazu gehören insbesondere die Berechnungsgeschwindigkeit und die Anzahl der Lasteinheiten.

Weiterhin wird hier die Dauer von Lastverteilungszyklen gemessen, d. h. die Kosten der Lastverteilung. Diese besteht zum einen aus der Dauer für die Bestimmung der Last (wieviel, wohin, welche) und zum anderen aus der Zeit für die Migration der Last. Diese Informationen dienen der Kosten-Nutzen-Analyse, um festzustellen, ob der Gewinn durch Lastverteilung größer ist als die Lastverteilungskosten selber. Nur wenn dieses gegeben ist, ist die Durchführung von Lastverteilung effizienzsteigernd.

Durch die Erfassung der Lastverteilungskosten wurde eine Basis zur Kosten-Nutzen-Analyse geschaffen. Die zu erwartende Zeitersparnis ist mit einfachen Methoden in einem dynamischen System wie dem Grid kaum zu erfassen. Daher ist dieser Aspekt in dieser Arbeit nicht genauer untersucht worden.

Da der Anwendungsmonitor Daten der Anwendung erfasst, erfordert seine Nutzung die Integration einiger Funktionsaufrufe in den Anwendercode. Diese Funktionsaufrufe beinhalten das Messen von Zeit zwischen zwei Messpunkten oder aber die einfache Übermittlung von Werten, wie z. B. die Anzahl an Lasteinheiten (bzw. Zellen), zum *mLB*. Anhang B gibt eine Übersicht über die Funktionen der Benutzerschnittstelle.

5.2.4 Lastverteilungssteuerungs (LBS-) Modul

Das LBS-Modul nutzt die Werte des *KMs* und *AMs* und verarbeitet diese zu Informationen für die Lastverteilungsunterstützung der Anwendung. Aus den Berechnungsgeschwindigkeiten wird die optimale Lastrate bestimmt (siehe Abschnitt 4.2). Diese wiederum bestimmt zusammen mit der Anzahl an Lasteinheiten die optimale Lastmenge, die der anwendungseigene Lastverteiler für seine Entscheidungen benötigt.

Die Feststellung, welche PEs an Lastverteilung beteiligt werden, wird ebenfalls vom *LBS*-Modul getroffen.

Bei der Ortung des Ungleichgewichts in der Hierarchie muss sichergestellt sein, dass mindestens zwei Subsysteme gefunden werden, die ein Ungleichgewicht aufweisen. Durch den Toleranzwert für ein Lastungleichgewicht und auch durch Rundungsfehler kann es geschehen, dass nur ein Subsystem gefunden wird. Um diesem zu entgegnen, wird der Toleranzwert solange reduziert, bis mindestens zwei Subsysteme gefunden werden, die dann ihre Last untereinander austauschen können.

Die Entscheidung, ob Lastverteilung überhaupt initiiert werden muss, wird von der Anwendung selber auf das *LBS*-Modul übertragen. Da eine Kosten-Nutzen-Analyse derzeit noch nicht stattfinden kann, basiert diese Entscheidung auf einem festen, aber vom Anwenderprogramm einstellbaren Wert. Dieser Wert gibt an, welches Ungleichgewicht akzeptiert wird. Erst wenn jenes oberhalb dieses Schwellwertes liegt, wird Lastverteilung durchgeführt. Dadurch wird vermieden, dass bei minimalem Ungleichgewicht (derzeit liegt der Wert bei fünf Prozent) Lastverteilung durchgeführt wird.

Die Aufbereitung aller Informationen für den anwendungseigenen Lastverteiler (siehe Abschnitt 5.3) schließt die Arbeit des *LBS*-Moduls im aktuellen Lastverteilungszyklus ab.

5.2.5 Zusammenspiel der Module

Abbildung 5.6 zeigt das Zusammenspiel von Kommunikationsmonitor, Anwendungsmonitor und LBS-Modul.

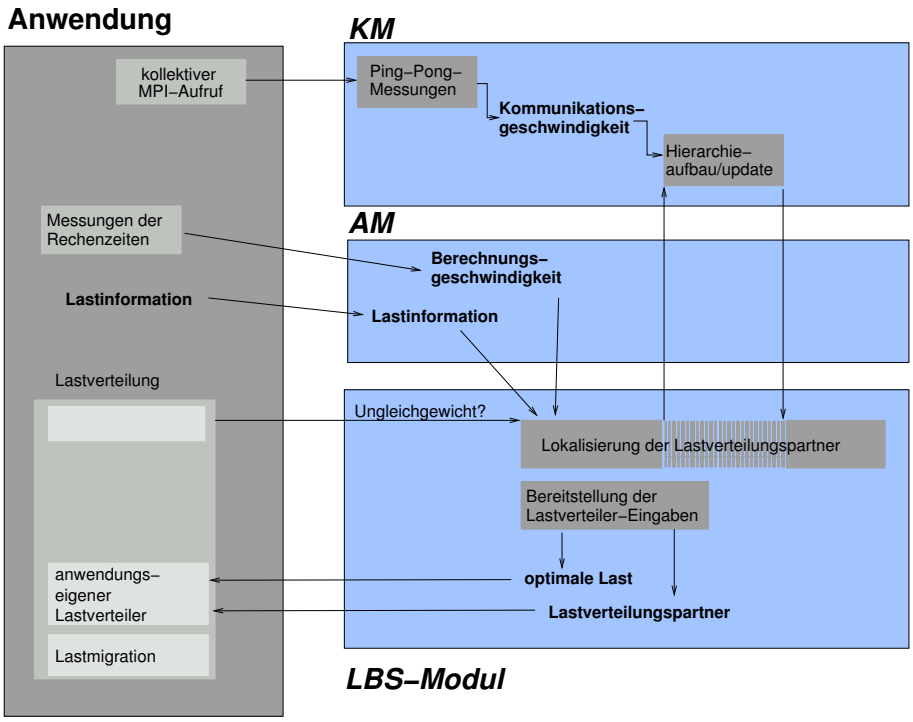


Abbildung 5.6: Zusammenspiel der *mLB*-Module

Erfolgt während der Berechnung ein kollektiver Aufruf seitens der Anwen-

dung, so wird dieser Aufruf an den **KM** weitergeleitet. Dieser führt daraufhin Messungen des Netzwerks durch und ermittelt die Kommunikationsgeschwindigkeiten zu anderen PEs.

Die Berechnungszeiten für lastrepräsentierende Rechenschritte werden vom **AM** erfasst, ebenso wie Lastinformationen (z. B. Anzahl Gitterzellen auf lokalem PE).

Wird von der Anwendung Lastverteilung initiiert, so wird dieser Aufruf an das **LBS**-Modul weitergereicht. Dieses berechnet aus den Berechnungsgeschwindigkeiten und der Lastinformation die optimale Last sowie das Lastungleichgewicht des lokalen PEs. Es stößt den Hierarchieaufbau an, der vom *KM* übernommen wird. Ist die Hierarchie aktuell, so erfolgt die Lokalisierung der Lastverteilungspartner.

mLB nutzt das MPI-Konzept der Kommunikatoren, indem es aus den PEs des Subsystems, in dem Lastverteilung durchgeführt werden soll, einen Kommunikator bildet. Dieser Kommunikator wird dem Anwendungslastverteiler (ALB) übergeben. Das Subsystem agiert somit aus Sicht des ALBs wie eine komplette Anwendung, der Teilgraph, der dem ALB übergeben wird, wie ein kompletter Graph.

5.3 Anwendungslastverteiler

Derzeit existiert eine Schnittstelle zu dem bereits vorgestellten Lastverteilungswerkzeug Jostle [WCE96]. Jostle wurde aufgrund seiner Flexibilität gewählt. Es ermöglicht der Anwendung, die Zielgrößen der Partitionen vorzugeben. Diese Fähigkeit ist eine Voraussetzung, um die kapazitätenabhängige Lastverteilung einsetzen zu können.

Die netzwerkabhängige Lastverteilung kann ebenfalls mit Jostle durchgeführt werden. Jostle nutzt MPI und akzeptiert einen von der Anwendung vorgegebenen MPI-Kommunikator.

Die Parameter, die für den Aufruf von Jostle notwendig sind, werden vom LBS-Modul aufbereitet und weitergereicht.

Jostle-Schnittstelle:

```
void pjostle(int nnodes, int offset, int *core, int *halo,
                int *index, int *degree, int *node_wt,
                int *partition, int local_nedges, int *edges,
                int *edge_wt, int *network,
                int output_level, int dimension, double* coords);
```

mLB-Schnittstelle:

```
void AppDoMetaLB(int nnodes, int offset, int *core, int *halo,
                 int *index, int *degree, int *node_wt,
                 int *partition, int local_nedges, int *edges,
                 int *edge_wt, int *network,
                 int output_level, int dimension, double* coords)
                 int lb_phase,
                 int *global_nodes_lb_part, int *local_nodes_opt);
```

Die Anwendung übergibt *mLB* als Parameter den kompletten dualen Graphen. Für den Jostle-Aufruf werden einige dieser Parameter folgendermaßen geändert:

nnodes: Gesamtanzahl der Knoten im gewählten Subsystem

halo: Die Schnittkanten, die Verbindungen zu PEs darstellen, die nicht zum Subsystem gehören, müssen eleminiert werden. Das Halo-Feld, in dem die entsprechenden Halos angegeben werden, muss entsprechend modifiziert werden.

index: In Jostle gibt es mehrere Möglichkeiten, die Graphknoten zu indizieren. Beim zusammenhängenden Format wird eine fortlaufende Nummerierung der Knoten vorausgesetzt. So befinden sich die Knoten $0, ..., |nodes_{PE_0}|-1$ auf PE_0, Knoten $|nodes_{PE_0}|, ..., |nodes_{PE_1}| - 1$ auf PE_1, usw., wobei $|nodes_{PE_i}|$ die Anzahl Graphknoten auf PE_i sind. Der Vorteil dieses Formats ist es, dass nicht jeder Knotenindex angegeben werden muss, wie es in einem anderen von Jostle unterstützten Format der Fall ist.

Ein Nachteil dieses Formats bei Benutzung von *mLB* ist es, dass dieses hergestellt werden muss, wenn nur Teilgraphen an Jostle übergeben werden, die sich nicht auf fortlaufend nummerierten PEs, beginnend mit PE_0, befinden.

degree: Dieses Feld beinhaltet den Grad jedes Graphknotens. Durch Wegfall einiger Halos erniedrigt sich der Grad (= Anzahl Nachbarn) der entsprechenden Knoten.

local_nedges: Die Anzahl an lokalen Kanten ändert sich durch die vorherigen Punkte ebenfalls.

edges: Die Kantenliste wird ebenfalls entsprechend modifiziert.

network: Die Anzahl an beteiligten PEs reduziert sich von der Gesamtanzahl zu der Anzahl der PEs im Subsystem.

processor_wt: In diesem Feld wird die Gewichtung der einzelnen PEs und damit auch die Menge an Ziellast angegeben. Die kapazitätenabhängige Lastverteilung wird durch dieses Feld ermöglicht.

Kapitel 6

Validierung

Nachdem die Konzepte und deren Umsetzung vorgestellt wurden, erfolgt nun ihre Validierung. Dabei muss überprüft werden, ob die Ziele aus den Anforderungskatalogen (siehe Kapitel 3) erreicht wurden.

6.1 Anforderungen an die Monitoring-Umgebung

Um die Monitoring-Umgebung zu validieren, müssen folgende Aspekte untersucht werden:

- Skalierbarkeit
- Verfügbarkeit
- Effizienz

Die Erfüllung dieser Forderungen wurde überprüft und, falls erforderlich, durch Tests validiert.

6.1.1 Skalierbarkeit

Durch seine verteilte Architektur ist eine Basis für die Skalierbarkeit des Monitorings geschaffen worden. Die Feststellung der Berechnungsgeschwindigkeiten erfolgt lokal und ist daher irrelevant für diesen Aspekt. Der bestimmende Faktor für die Skalierungseigenschaft von *mLB* sind die Ping-Pong-Tests. Daher wurden Messungen durchgeführt, bei denen diese Kommunikationstests jeweils auf einer unterschiedlichen Anzahl an Prozessoren durchgeführt wurden.

6.1.2 Verfügbarkeit

Diese ist gegeben, da das Monitoring nur für die Anwendung arbeitet und von dieser gesteuert wird. Damit kann zu jedem beliebigen Zeitpunkt auf alle notwendigen Informationen zugegriffen werden.

6.1.3 Effizienz

Hier muss überprüft werden, ob die gewonnenen Messwerte genau das liefern, was erzielt werden soll, wobei der Aufwand für ihre Feststellung so minimal wie möglich ist. In Bezug auf die Messwerte für die Kommunikationsgeschwindigkeiten ist festzustellen, ob die gewonnenen Ergebnisse das Kommunikationsverhalten der Anwendung repräsentieren und die Messungen nur einen geringen Overhead erzeugen.

Zur Feststellung des Overheads wurde die Laufzeit der Ping-Pong-Messungen erfasst. Ein wichtiger Aspekt ist hier die Größe der Nachrichten, die bei diesen Messungen verschickt werden. Diese sollte so klein wie möglich, aber so groß wie nötig gewählt werden. Werden zu viele Daten verschickt, dann wird sich der Overhead der Messungen, und damit auch des kompletten Monitorings, erhöhen. Bei Wahl einer zu geringen Datenmenge kann die Aussagekraft der Messungen zu gering sein.
Ziel bei der Wahl dieser Größe ist es, repräsentative Nachrichten auszutauschen, um auf Basis der Messwerte eine Hierarchie aufzubauen, die das Kommunikationsverhalten der Anwendung widerspiegelt. Dazu wurden Messungen mit unterschiedlichen Nachrichtengrößen durchgeführt.

Die Häufigkeit der Messungen bestimmt den Overhead entscheidend. Daher wurde bereits die Forderung nach Dynamik angeführt. Diese wurde bisher nicht explizit integriert. Zeitabhängiges Monitoring wurde stattdessen eingesetzt. Das bedeutet, erst nach Ablauf eines bestimmten Intervalls werden erneut Messungen durchgeführt. Angemessene Werte für dieses Intervall wurden bei Testläufen mit einer FEM-Anwendung evaluiert, mit dem Ziel wenige, aber ausreichend viele Messungen durchzuführen, um den aktuellen Netzwerkstatus zu ermitteln.

In Bezug auf die Messwerte für die Berechnungsgeschwindigkeiten muss geprüft werden, ob die Messung der Berechnungseinheiten ein gutes Maß für die Geschwindigkeit der Recheneinheiten darstellt. Die Ergebnisse der Lastverteilung, die auf diesen Messwerten basiert, werden diese Fragestellung klären.

6.2 Lastverteilungssteuerung mit *mLB*

Für die Validierung der Lastverteilungssteuerung mit *mLB* wurden folgende Aspekte überprüft:

- im Hinblick auf die kapazitätenangepasste Lastverteilung:
  die optimale Nutzung der Ressourcen,
- im Hinblick auf die netzwerkabhängige Lastverteilung:
  die Berücksichtigung unterschiedlicher Kommunikationsgeschwindigkeiten,
- die Reaktion auf Zustandsänderungen,
- Skalierbarkeit und
- Gesamteffizienz.

6.2.1 Kapazitätenangepasste Lastverteilung

Werden die Kapazitäten der Recheneinheiten optimal genutzt, so erreichen alle PEs ihre Synchronisationspunkte, an denen Nachrichten ausgetauscht werden, zur gleichen Zeit. Damit entstehen keine oder nur minimale Wartezeiten bei diesen kollektiven Kommunikationen. Um diese Wartezeiten festzuhalten, wurde die Zeit zwischen Initiierung und Zustandekommen der Kommunikation gemessen. Diese Messungen sind ein wichtiger Anhaltspunkt für die optimale Ausnutzung der Kapazitäten der Recheneinheiten.

6.2.2 Netzwerkabhängige Lastverteilung

Das Ziel war hier die Berücksichtigung unterschiedlicher Kommunikationsgeschwindigkeiten bei der Lastverteilung. Dafür werden schnellere Kommunikationswege langsameren gegenüber bevorzugt, indem Lastverteilung in Subsystemen durchgeführt wird.

Die Richtigkeit dieser Vorgehensweise kann zum einen überprüft werden, indem Zeitmessungen von Lastverteilung mit einer unterschiedlichen Anzahl von PEs und Lastmengen und zum anderen Messungen von Lastverteilung über unterschiedliche Kommunikationswege (z. B. Ethernet, Infiniband, gemeinsamer Speicher) durchgeführt werden.
Ein weiterer wichtiger Aspekt ist hier aber auch der richtige Aufbau der Hierarchie, d. h. einer Struktur, die die Geschwindigkeiten der von der Anwendung genutzten Kommunikationswege repräsentiert. Dafür spielt der Parameter DIFF_TOLERANCE (siehe Gleichung 4.8) eine wesentliche Rolle. Diese Variable bestimmt die Granularität der Hierarchiestruktur. Bei einem großen Wert werden wenige große Subsysteme gebildet, bei einem kleinen hingegen viele kleine Systeme. Die optimale Größe dieser Variablen muss in den Tests ermittelt werden.

Durch die Art der Bestimmung der Kandidatenliste, d. h. Verhältnisbildung der einzelnen Kommunikationszeiten zusammen mit dem Wert von DIFF_TOLERANCE, zeigen das erste Listenelement einer Hierarchiestufe zusammen mit dem letzten Element der vorhergehenden Ebene die Größenunterschiede der Kommunikationszeiten zwischen den Ebenen auf.

Um die korrekte Arbeitsweise des Hierarchieaufbaus zu überprüfen, wurde in einem dediziert genutzten System die erzeugte hierarchische Struktur mit den vorliegenden Netzwerkparametern verglichen.

6.2.3 Reaktion auf Zustandsänderungen

Durch Erfassung der Messwerte während der Laufzeit werden aktuelle Werte betrachtet. Diese dienen, bei der netzwerkabhängigen Lastverteilung, dem Hierarchieaufbau, sobald die Struktur benötigt wird. Die Hierarchie repräsentiert damit stets die Netzwerkkapazitäten.

Für die kapazitätenabhängige Lastverteilung werden Messwerte gemittelt. Die Anzahl der zu mittelnden Werte kann vom Anwenderprogramm bestimmt

werden. Verändert sich die zur Verfügung stehende Kapazität einer Recheneinheit, so findet dieses Berücksichtigung durch die zuletzt gewonnenen Werte.

Damit ist die Forderung der dynamischen Anpassung an Zustandsänderungen erfüllt.

6.2.4 Skalierbarkeit

Den Nachweis hierüber erbringen Laufzeitmessungen, die in einer Simulationsumgebung mit einer unterschiedlichen Anzahl von PEs erfolgten. Diese wurde so entwickelt, dass Skalierbarkeitsprobleme seitens des Simulationsprogrammes ausgeschlossen werden. Auf dieser Basis konnte die Skalierungseigenschaft von *mLB* gemessen werden.

6.2.5 Gesamteffizienz

Diese ergibt sich durch den Nutzen von *mLB* verglichen mit seinem Overhead. Hierfür wurden Vergleichsmessungen zwischen Testläufen mit *mLB* und jene ohne *mLB* durchgeführt.

Kontrolliert werden musste hier auch, ob die vom ALB erzeugten Partitionen von der Anwendung weiterverarbeitet werden können. Der Teilgraph, der von *mLB* erzeugt wird und der dem ALB übergeben wird, wird isoliert rebalanciert. Kanten, die zu nicht an Lastverteilung beteiligten PEs führen, werden nicht betrachtet. Dieses Vorgehen kann dazu führen, dass die Anwendung die derart erzeugten Partitionen nicht oder nur mit großem Aufwand weiterverarbeiten kann.

Dieser Aspekt kann durch eine Integration von *mLB* in eine vorhandene Anwendung analysiert werden.

Die Validierung erfolgte hauptsächlich mit Hilfe eines selbst entwickelten Simulators. Nur wenige Parameter wurden mit Hilfe einer bestehenden FEM-Anwendung evaluiert.

Der Vorteil dieses Vorgehens liegt darin, dass Probleme, die von der FEM-Anwendung ausgelöst werden können, ausgeschlossen werden. Dieses betrifft insbesondere Skalierbarkeitsprobleme, da viele FEM-Anwendungen nur auf einer begrenzten Anzahl an Prozessoren lauffähig sind. Ein vorzeitiges Terminieren aufgrund von Speicherplatzproblemen seitens der Anwendung (sehr verbreitet in diesem Bereich) konnte dadurch ebenfalls vermieden werden.

6.3 Beschreibung der Testplattform

Die Tests erfolgten auf einem PC-Cluster des parallelen Rechenzentrums der Universität Paderborn (PC^2) der Produktlinie hpc-Line. Seine 200 Rechenknoten, jeweils bestückt mit zwei Xeon Prozessoren mit 4 GByte Hauptspeicher

(genauere technische Informationen befinden sich im Anhang C), sind sowohl über ein Infiniband-Netzwerk verbunden, als auch über Ethernet.

Zur Kommunikation stehen dem Benutzer unterschiedliche MPI-Implementierungen zur Verfügung. Kommunikation über Infiniband wird von ScaMPI unterstützt. ScaMPI wurde von Scali Computer (www.scali.com) insbesondere für Gigabit Ethernet, Myrinet, Infiniband und SCI entwickelt. Beim Einsatz von ScaMPI hat die Anwendung die Möglichkeit, zwischen Netzwerktechnologien zu wählen. Das bedeutet für die eingesetzte Plattform, dass die Rechenknoten entweder über Ethernet oder über Infiniband kommunizieren können. Erfolgt Kommunikation zwischen Prozessoren, die sich auf einem Rechenknoten befinden, so erfolgt dies über ihren gemeinsamen Speicher (shared memory Kommunikation).

Diese unterschiedlichen Kommunikationsmöglichkeiten bieten eine Umgebung, in der „Distributed Supercomputing“, für das *mLB* entwickelt wurde, unter kontrollierten Bedingungen simuliert werden kann. Mehrere Rechenpartitionen, die jeweils aus mehreren über Infiniband kommunizierenden Rechenknoten bestehen, können mit Hilfe von PACX-MPI über Ethernet miteinander kommunizieren. Dabei stellt jede Rechenpartition ein paralleles System dar. Die Verschaltung dieser Partitionen mit PACX-MPI unterscheidet sich prinzipiell nicht von einer Koppelung mehrerer, geografisch verteilter Parallelrechner mit PACX-MPI.

Für die Rechnungen auf dem Infiniband-Cluster wurden Rechenknoten reserviert. Diese standen dadurch exklusiv zur Verfügung. Durch die Architektur des Infiniband-Netzwerks (ein zentraler Switch) stehen auch die Verbindungen zwischen den reservierten Rechenknoten dem Benutzer uneingeschränkt zur Verfügung. Einzig die Kommunikation über Ethernet findet nicht exklusiv statt. Die Übertragungsraten sind somit abhängig von der Auslastung des Ethernet-Switches durch den anderen Netzwerkverkehr.

6.4 Validierung durch eine Simulationsumgebung

6.4.1 Validierung der Monitoring-Umgebung

Zur Validierung der **Skalierbarkeit** der Kommunikationsmessungen (Ping-Pong-Messungen) wurde ein einfaches MPI-Programm, das kollektive Operationen durchführt, eingesetzt. Um auf die Funktionalitäten der netzwerkabhängigen Lastverteilung zugreifen zu können, wurde diesem Programm die entsprechende *mLB*-Bibliothek hinzugefügt. Es wurden Testläufe auf dem hpc-Line-Cluster durchgeführt. Um ausschließlich das entwickelte Monitoring zu testen, ohne dass die darunter liegende Konfiguration der Kommunikationstechnologie Einfluss nehmen kann, wurde Ethernet-Kommunikation gewählt. Gemessen wurde von zwei bis zu 180 Prozessen. Das Ergebnis zeigt Abbildung 6.1.

Es ist deutlich zu erkennen, dass die Kurve Sprünge aufweist. Der sprunghafte Anstieg der Messwerte ist auf den Kommunikationsplan (siehe Abschnitt 4.4.2) zurückzuführen. Bei acht Kommunikationspartnern existieren z. B. 28

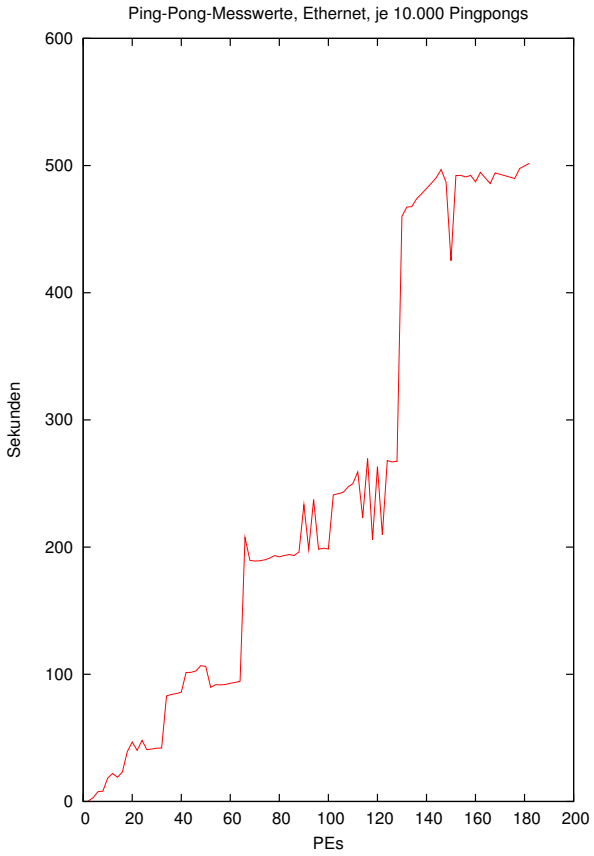


Abbildung 6.1: Ergebnisse von Kommunikationsmessungen auf einer unterschiedlichen Anzahl von PEs

Kommunikationsmöglichkeiten. Bei der Aufstellung des Kommunikationsplans werden diese Kommunikationspaare in sieben Durchläufe von jeweils vier parallelen Kommunikationen aufgeteilt. Bei neun bis 16 Kommunikationspartnern (36 bzw. 120 Kommunikationspaare) werden 15 Durchläufe mit ein bis vier parallelen Kommunikationen ermittelt, d. h. mehr als das Doppelte an Durchläufen ist notwendig. Abbildung 6.2 zeigt die Anzahl der Kommunikationsdurchläufe in Abhängigkeit von den PEs.

Dieses zeigt, dass der Algorithmus zur Erstellung des Kommunikationsplans nicht optimal ist. Die Skalierung der Ping-Pong-Messungen ist jedoch nicht gefährdet.

Die Sprünge der Kurve aus Abbildung 6.2 entsprechen denen aus Abbildung 6.1 (siehe Abbildung 6.3). Die Berechnung der Zeit pro Kommunikationsdurchlauf (siehe Abbildung 6.4) zeigt, dass diese sich nur leicht ansteigend ändert.

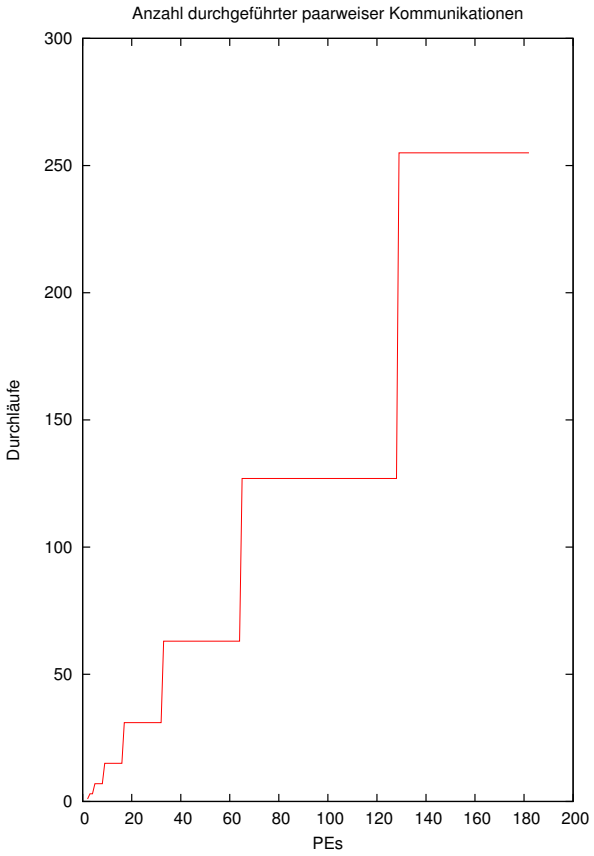


Abbildung 6.2: Kommunikationsdurchläufe

Ein optimierter Kommunikationsplan lässt daher eine gute Skalierbarkeit erwarten.

Die Gesamteffizienz des Monitorings zeigt sich bei der Laufzeitanalyse von Anwendungstestläufen. Eine geeignete Nachrichtengröße konnte in diesem Zuge ebenfalls festgestellt werden. Dynamik wird im Rahmen der Untersuchung einer realen FEM-Anwendung beleuchtet.

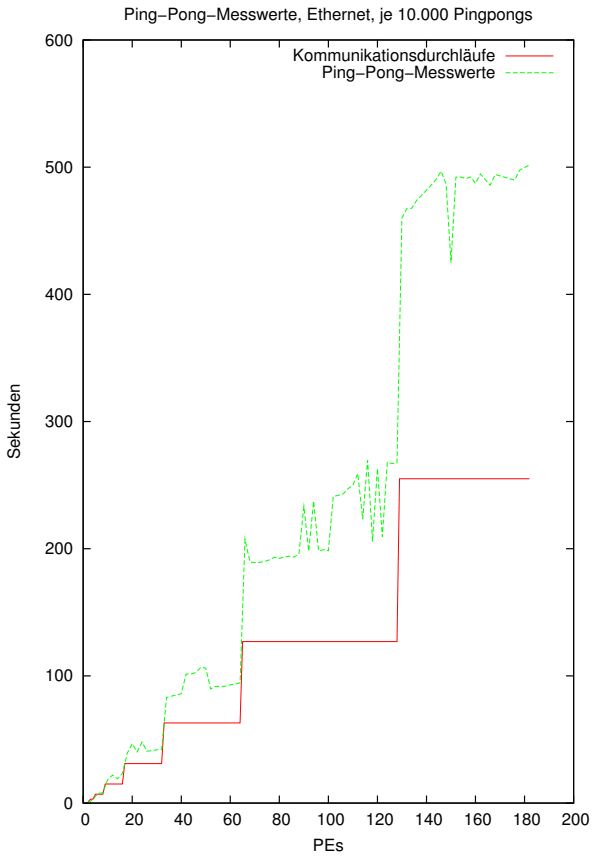


Abbildung 6.3: Kommunikationsdurchläufe mit Messergebnissen

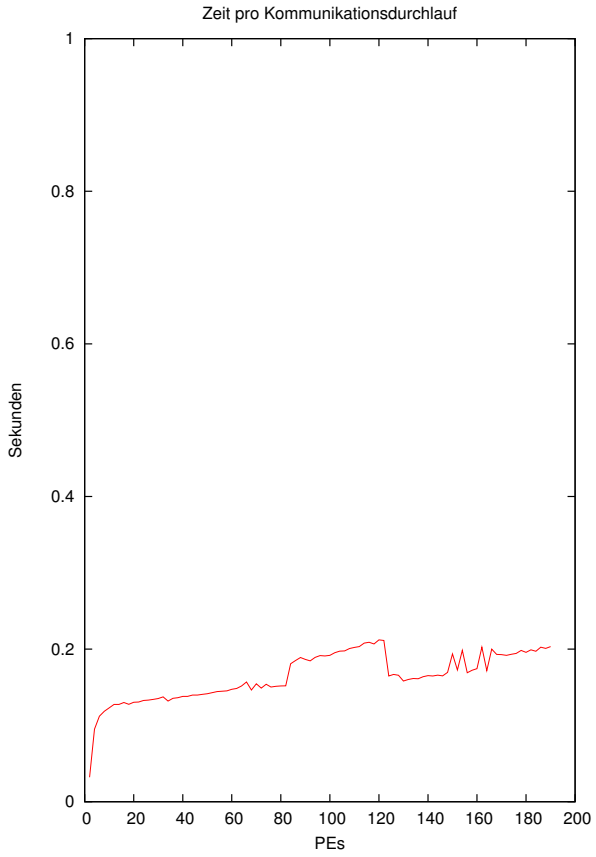


Abbildung 6.4: Zeit pro Kommunikationsdurchlauf in Abhängigkeit von der Anzahl an PEs

6.4.2 Beschreibung des FEM-Simulators

Der entwickelte FEM-Simulator repräsentiert das Verhalten einer parallelen FEM-Anwendung. Für jede Iteration wird eine Schleife durchlaufen, in der zunächst die Last erhöht wird (Nachbildung von Gitterverfeinerungen). Diese Lasterhöhung erfolgt auf einer bestimmten Quote von PEs. Diese erhalten jeweils eine unterschiedliche zusätzliche Last (=Anzahl Gitterzellen). Somit entsteht ein Ungleichgewicht zwischen den PEs.

Nach der Lasterhöhung findet Lastverteilung statt und damit der Aufruf des ALBs. Datenmigration, und damit die Herstellung einer balancierten Verteilung der Anwendungslast, wird nicht benötigt, da keine kompletten Datenstrukturen bei dieser Simulation zu verschieben sind.

Nach der Lastverteilung erfolgt die Berechnung. Dabei werden arithmetische Operationen durchgeführt, die das Lösen von Gleichungen innerhalb einer realen FEM-Anwendung nachbilden. Mit dem Austausch von Informationen zwischen den PEs schließt diese Rechenphase ab. Dabei werden MPI-Nachrichten zwischen benachbarten PEs ausgetauscht. Im realen Fall sind dieses PEs, die sich Schnittkanten des dualen Graphen teilen. Im Simulator werden die Nachbarn eines PEs festgelegt. Inhalt der zu verschickenden Nachrichten ist ein Feld mit doppelt genauen Gleitkomma-Werten der Länge „Anzahl lokaler Gitterzellen“. Es werden also Werte für jede lokale Gitterzelle ausgetauscht. Dieses entspricht der Arbeitsweise einer realen FEM-Anwendung.

Heterogenität in Bezug auf Leistungsfähigkeit der Prozessoren wird durch zusätzliche Last einzelner PEs simuliert.

Der Simulator erfüllt die Voraussetzungen einer Testumgebung für die Grundfunktionalitäten: Er arbeitet parallel, kommuniziert mittels MPI und ist adaptiv.

6.4.3 Validierung von mLB

Es wurden zwei Testszenarien erstellt:

- FEM-Simulator ohne *mLB*
- FEM-Simulator mit *mLB*

Für die Tests mit *mLB* wurde der Aufruf des ALBs durch den Aufruf einer Funktion von *mLB* ersetzt. Dadurch wird das LBS-Modul (siehe Abschnitt 5.2.4) aktiviert. Dieses führt die Überprüfung des Lastgleichgewichts durch und gegebenenfalls den Aufruf des ALBs mit den neu generierten Parametern.

Mit dem Simulator wurden folgende Aspekte untersucht:

- Die Laufzeiten des ALBs Jostle bei Einsatz:
  - auf unterschiedlicher Anzahl von PEs und

  - mit unterschiedlicher Anzahl von Graphknoten,

- die geeignete Nachrichtengröße für die Ping-Pong-Messungen und
- der Aufbau der hierarchischen Struktur in einem dedizierten System,
- die angemessene Verteilung der Last zusammen mit
- der Gesamteffizienz von *mLB*.

Jostle-Laufzeiten

Jostle wurde mit unterschiedlicher Anzahl an Gesamtgraphknoten und PEs getestet. Dabei wurden zum einen die Laufzeiten bei einer unterschiedlichen Anzahl von PEs, aber festen Anzahl an Graphknoten verglichen (siehe Abbildung 6.5).

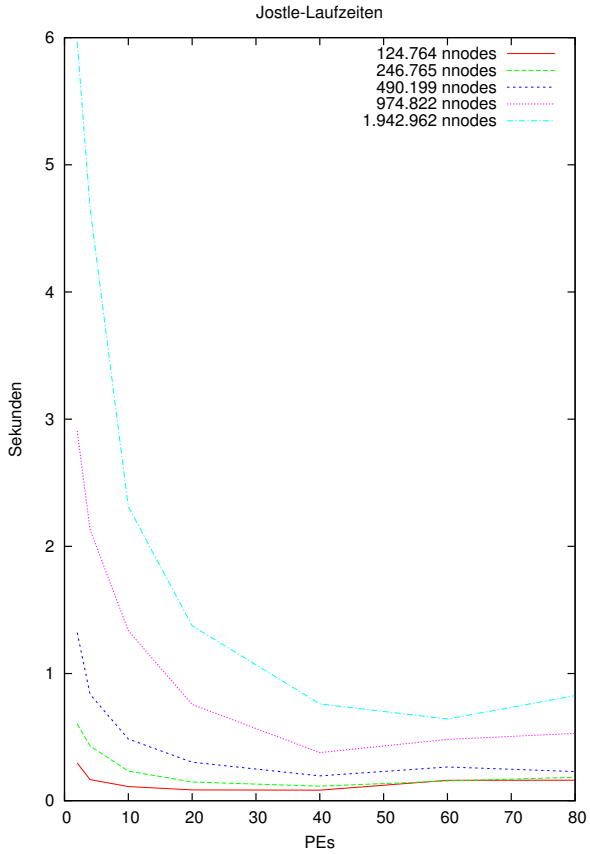


Abbildung 6.5: Jostle-Laufzeiten in Abhängigkeit von der Anzahl an PEs

Abbildung 6.6 zeigt den Vergleich bei fester Anzahl an PEs, aber unterschiedlicher Graphknotenzahl.

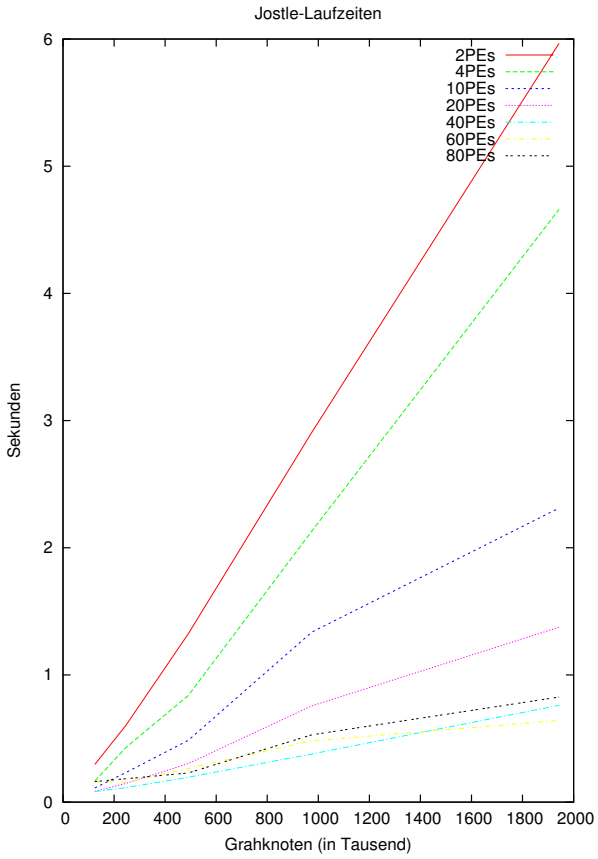


Abbildung 6.6: Jostle-Laufzeiten in Abhängigkeit von der Anzahl an Graphknoten

Die Ergebnisse zeigen, dass besonders deutliche Unterschiede bei der Berechnung einer großen Anzahl an Graphknoten bestehen.

Arbeitet Jostle z. B. mit 1,9 Millionen Graphknoten auf 20 PEs, so benötigt es circa 1,3 Sekunden. Verringert sich die Anzahl der PEs von 20 auf vier und die Anzahl an Graphknoten entsprechend, so beträgt die Laufzeit von Jostle weniger als 0,8 Sekunden. Das bedeutet eine Laufzeitersparnis von über 60 Prozent. Da Lastverteilung innerhalb einer Anwendung oftmals in jeder Iteration durchgeführt wird, summieren sich die Zeitersparnisse.

Reale FEM-Anwendungen rechnen oftmals mit mehreren Millionen Graphknoten. Lastverteilung, die nicht die gesamte Anzahl an Knoten verarbeiten muss, was durch *mLB* erreicht wird, lässt noch größere Laufzeitgewinne erwarten.

Erkennung der Hierarchiestruktur

In einem dediziert genutzten System, wie es durch die Reservierung von Rechenknoten vorlag, muss *mLB* den Aufbau des Netzwerks erkennen. Das bedeutet, dass PEs auf dem gleichen Rechenknoten, die über den gemeinsamen Speicher kommunizieren, über die schnellste Verbindung verfügen und damit auf unterster Ebene ein Subsystem bilden. Infiniband bildet somit die nächst höhere Ebene. Wird zudem noch Ethernet eingesetzt, so bildet diese die höchste Subsystemebene.

Die Evaluierung der geeigneten Nachrichtengröße, die im nächsten Abschnitt stattfindet, zeigt zum einen auch, dass auf Basis der gewonnenen Messwerte eine Hierarchie erkannt wird, und zum anderen welche Werte die Variable DIFF_TOLERANCE annehmen sollte.

Nachrichtengröße

Getestet wurden hier Nachrichten der Größe 1 Byte, 1.000 Byte und 64.000 Byte mit acht und mit achtzig PEs.

Werden Ping-Pong-Messungen durchgeführt, so besteht eine Messung aus mehreren Sende- und Empfangsaktionen. Der Grund liegt darin, dass auf einigen Plattformen die Zeiten für ein einziges Senden und Empfangen zu kurz sind, um messbar zu sein. Daher wurden jeweils fünf Nachrichtendurchläufe durchgeführt, um ein einzelnes Messergebnis zu erhalten.

Nachrichten-größe (Byte)	#PEs	Laufzeit (Sekunden)	1. Ebene (Sekunden)	2. Ebene (Sekunden)	Verhältnis $\frac{2.Ebene}{1.Ebene}$
1	8	0,0505	0,000002	0,00001	5
1.000	8	0,0833	0,000005	0,00002	4
64.000	8	1,1511	0,000075	0,0003	4
1	80	1,2148	0,000004	0,00001	2,5
1.000	80	1,9197	0,000007	0,00002	2,9
64.000	80	26,0948	0,000082	0,0003	3,7

Tabelle 6.1: Laufzeiten bei Ausführung von 100 Ping-Pong-Messungen

Tabelle 6.1 zeigt eine Übersicht über die Messergebnisse. Die Spalten vier und fünf mit den Überschriften „1. Ebene“ und „2. Ebene“ geben die durchschnittlichen Kommunikationszeiten zwischen PEs auf der entsprechenden Ebene an. In der verwendeten Testumgebung bedeutet dies, dass für Ebene eins Kommunikation über den gemeinsamen Speicher gemessen wurde, Ebene zwei zeigt die Zeiten für die Infiniband-Kommunikation. Die letzte Spalte zeigt die Verhältnisse der beiden Kommunikationszeiten. Anhand dieses Verhältnisses wird die

hierarchische Struktur erkannt. Die hohen Werte, die in der Tabelle auftreten, indizieren eine gute Erkennung dieser Struktur.

Deutlich zu erkennen ist, dass eine Nachrichtengröße von 64.000 Byte hohe Laufzeiten verursacht, der Informationsgehalt jedoch nicht höher ist als jener, der durch kleinere Nachrichten erzeugt wird. Nachrichtengrößen von einem und 1.000 Byte liefern einen hohen Informationsgehalt bei niedrigen Laufzeiten. Nachrichten der Länge ein Byte sind nicht repräsentativ für die Anwendung und wurden daher nicht für das Monitoring gewählt. Stattdessen wurden Nachrichten der Größe 1.000 Byte verschickt, da diese einen relativ niedrigen Overhead erzeugen und gleichzeitig aber einen hohen Informationsgehalt liefern.

Nachrichten der Größe von einem und 1.000 Byte wurden auch mit PACX-MPI getestet.

Nachrichten-größe (Byte)	#PEs	Laufzeit (Sek.)	1. Ebene (Sek.)	2. Ebene (Sek.)	3. Ebene (Sek.)
1	2x2+2x4	660	0,000003	0,000013	0,0003
1.000	2x4+2x4	240	0,000003	0,000014	0,0004

Tabelle 6.2: Laufzeiten bei Ausführung von 100 Ping-Pong-Messungen unter PACX-MPI

Tabelle 6.2 zeigen Messungen mit 12 und 16 PEs. Dabei wurden auf jedem Rechenknoten, wie oben auch, zwei Prozesse gestartet, die über gemeinsamen Speicher kommunizierten. Dieses zeigen die Werte für Ebene eins. Die beiden PACX-MPI-Partitionen kommunizierten über Ethernet (zu erkennen durch die Messwerte für Ebene 3). In der 16er Konfiguration wurden je PACX-MPI-Partition vier Rechenknoten eingesetzt, bei der 12er Konfiguration waren es zwei und vier Rechenknoten. Die Messungen zeigten hohe Kosten, die den gleichen Informationsgehalt wie die MPI-Messungen lieferten. Durch den Einsatz von Ethernet war die Gesamtlaufzeit abhängig von der aktuellen Netzauslastung. Daher konnten keine direkten Zusammenhänge von Nachrichtengröße und Laufzeit festgestellt werden. Die hohen Laufzeiten zeigen jedoch, dass Monitoring in dieser Konfiguration teuer ist. Die Anzahl der Ping-Pong-Messungen sollte daher minimal sein.

Der Wert für DIFF_TOLERANCE muss für die obigen Ergebnisse kleiner als 2,5 (niedrigster Wert der letzten Spalte aus Tabelle 6.1) gewählt werden. Da es sich bei den Messwerten jedoch um gemittelte Werte handelt, und somit die Verhältnisse der Kommunikationszeiten zwischen den Ebenen zum Teil geringer sind, wurde DIFF_TOLERANCE auf den Wert 1,6 gesetzt. Damit konnte die Hierarchie in allen Tests erkannt werden.

Gesamteffizienz und Skalierbarkeit

Die Aspekte, die in den vorangegangenen Abschnitten analysiert wurden, zeigen geringe Zusatzkosten bei Anwendung von *mLB*. Der Nutzen und damit auch seine Effizienz kann nur durch Vergleichsmessungen einer Anwendung ohne *mLB* festgestellt werden.

Abbildung 6.7 zeigt Messungen mit und ohne *mLB* mit unterschiedlich vielen PEs. Die Berechnung startete jeweils mit 150.000 Zellen, bzw. Graphknoten, und durchlief 35 Iterationen. In jeder Iteration wurde die Zellanzahl um circa 50.000 Zellen erhöht.

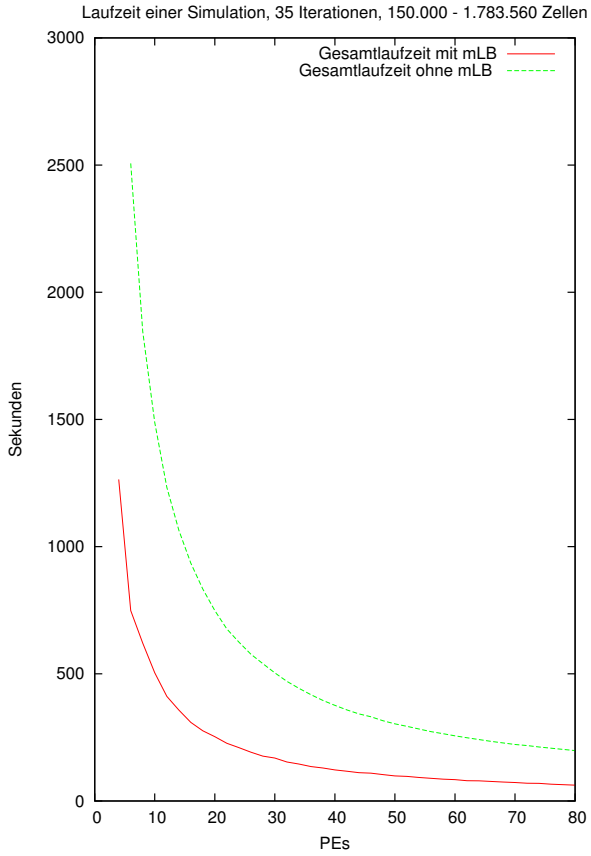


Abbildung 6.7: Laufzeitvergleich mit/ohne *mLB*

Es ist deutlich zu sehen, dass mit *mLB* wesentlich bessere Ergebnisse als ohne *mLB* erzielt werden. Teilweise reduziert sich die Laufzeit um über zwei Drittel.

Diese große Effizienzsteigerung durch *mLB* ist zum einen auf optimierte Last-

verteilung und zum anderen auf niedrige Zusatzkosten zurückzuführen.

In den nächsten Abschnitten wird daher sowohl der Effekt einer angemessenen Lastverteilung, als auch die Höhe der Zusatzkosten von *mLB* gezeigt.

Angemessene Lastverteilung

Ein Indiz für eine Lastverteilung entsprechend der vorhandenen Ressourcen ist, neben dem Laufzeitgewinn, die Wartezeit bei einer kollektiven Kommunikation. Je kürzer diese ist, desto gleichmäßiger konnte die Last von den Prozessoren abgearbeitet werden.

Eine Analyse der Wartezeiten zeigt große Unterschiede bei Einsatz mit bzw. ohne *mLB*. Abbildung 6.8 zeigt die maximalen Wartezeiten von PEs bei ei-

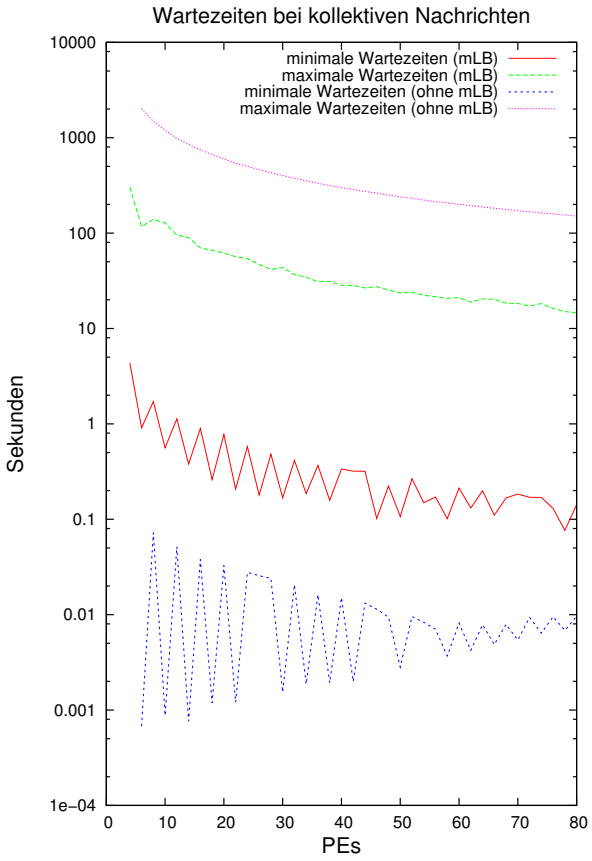


Abbildung 6.8: Vergleich der Wartezeiten bei kollektiven Nachrichten

ner kollektiven Kommunikation. Die gemessenen Werte ohne *mLB* sind durchschnittlich um das Zehnfache höher als jene mit *mLB*.

Diese niedrigen Wartezeiten zeigen eine gute Anpassung der Last an die aktuelle Leistungsfähigkeit der Prozessoren.

Kosten von *mLB*

Zur Analyse der Kosten von *mLB* wurden folgende Messwerte erfasst:

1. Monitoring-Kosten (Ping-Pong-Messungen)
2. Kosten für die Lokalisierung des Ungleichgewichts in der Hierarchie
3. Gesamtkosten für die angepasste Lastverteilung
4. Durchschnittliche Jostle-Laufzeiten

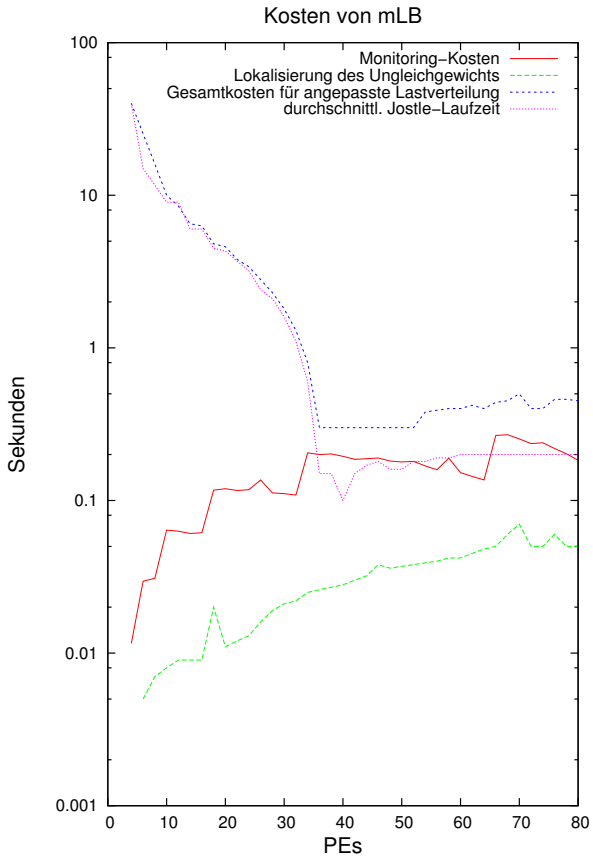


Abbildung 6.9: Kosten von *mLB*

Abbildung 6.9 zeigt die Kosten von *mLB*.
Deutlich zu erkennen ist, dass die Monitoring-Kosten (wenige Zehntel Sekunden), ebenso wie die Kosten zum Auffinden eines Ungleichgewichts (wenige Hundertstel Sekunden) sehr niedrig sind. Die Gesamtkosten für die angepasste Lastverteilung schließen die Kosten für den Aufbau der Hierarchie, die Bestimmung der optimalen Last eines PEs sowie die Durchführung von Jostle mit ein. Auch hier sind die geringen Kosten deutlich zu erkennen. Die Laufzeiten für Jostle bilden den Hauptanteil bei den Ergebnissen. Zieht man diese ab, so erhält man ebenfalls sehr kleine Werte für den Aufwand, verursacht durch *mLB*.

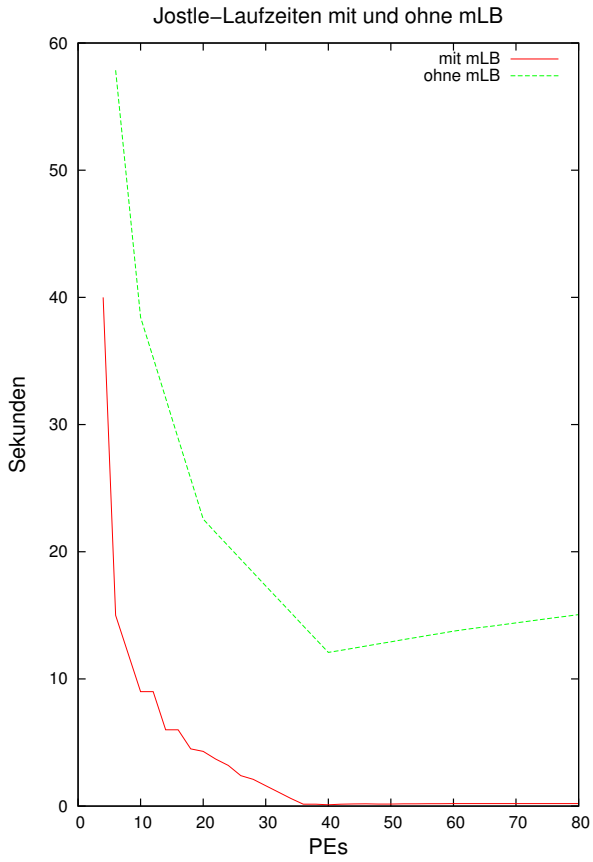


Abbildung 6.10: Jostle-Laufzeiten bei Einsatz mit und ohne *mLB*

Abbildung 6.10 zeigt den Vergleich von Jostle-Laufzeiten der gleichen Testläufe mit und ohne *mLB*. Eine deutliche Reduktion der Laufzeiten durch *mLB* ist auch hier zu erkennen. Interessant ist aber auch der Kurvenverlauf. Zunächst sinken die Jostle-Laufzeiten, zwar etwas stärker, aber ähnlich denen ohne Ein-

satz von *mLB*. Ab 36 PEs liegen die Jostle-Laufzeiten nur noch im Zehntel-Sekunden-Bereich.

Das liegt darin begründet, dass meist nur noch Zweiergruppen von PEs Lastverteilung untereinander durchführen. Dabei sind jene PEs beteiligt, die sich jeweils auf einem Rechenknoten befinden und über ihren gemeinsamen Speicher kommunizieren. Diese parallellaufenden Zweiergruppen bearbeiten jeweils wesentlich weniger Graphknoten als es bei einem Aufruf von Jostle mit allen PEs der Fall wäre. Betrachtet man in Abbildung 6.6 den steilen Anstieg der Kurve für zwei PEs, so wird deutlich, dass besonders bei dieser niedrigen Anzahl an PEs die Anzahl der Graphknoten eine entscheidende Rolle für die Laufzeit von Jostle spielt. Dieses erklärt den steilen Abfall der Jostle-Laufzeiten, da mit zunehmender Anzahl an PEs die Größe je Lastverteilungsprozess kleiner wird.

Beim Einsatz von Netzwerken mit geringeren Übertragungsraten, wie z. B. Ethernet, ist die Minimierung von Jostle-Aufrufen und der an Lastverteilung beteiligter PEs, die jeweils Kommunikation implizieren, ein wichtiger Aspekt. Die Messungen mit PACX-MPI (siehe Tabelle 6.2) zeigten die höheren Kommunikationskosten bei Nutzung von Ethernet. Daher kann erwartet werden, dass die Lastverteilungskosten in solch einer Umgebung durch *mLB* noch stärker reduziert werden.

Generell ist in Abbildung 6.9 zu beobachten, dass bei Erhöhung der Anzahl an PEs die Kosten von *mLB* nur gering steigen. Eine Skalierbarkeit der Anwendung ist somit nicht gefährdet.

Der Aufwand von *mLB* im Verhältnis zur Gesamtlaufzeit der Anwendung ist in Abbildung 6.11 zu sehen.

Hier wird besonders deutlich, dass die Zusatzkosten von *mLB* nur einen sehr geringen Anteil an der Gesamtlaufzeit darstellen. Diese Tatsache zusammen mit der großen Laufzeitverbesserung zeigt die hohe Effizienz von *mLB*.

6.5 Validierung durch eine bestehende FEM-Anwendung

Einige Parameter konnten erst durch Beobachtung des Verhaltens einer realen FEM-Anwendung eingestellt werden. Dieses betrifft insbesondere die Fähigkeit, auf den Teilpartitionen, die durch die modifizierte Lastverteilung entstanden sind, weiterzuarbeiten und den Wert für das zeitabhängige Monitoring.

6.5.1 Voraussetzungen für die Nutzung einer FEM-Anwendung

Um *mLB* in ein bestehendes Finite Elemente Werkzeug zu integrieren, muss dieses die folgenden Voraussetzungen erfüllen:

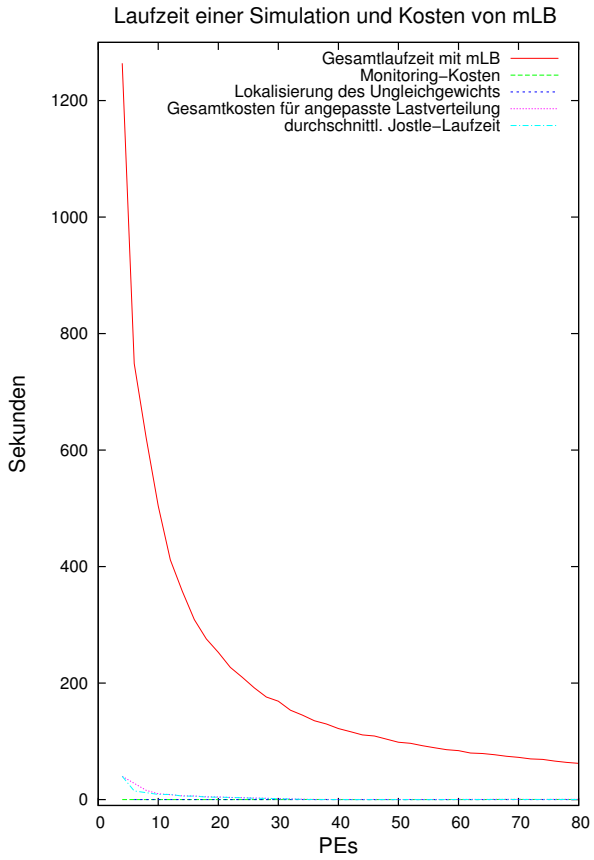


Abbildung 6.11: Laufzeiten und Kosten von *mLB*

- Parallelität: Die Software muss bereits in einer parallelen Version vorliegen.
- MPI: Die Kommunikation wird mittels MPI realisiert.
- Adaptivität: während der Laufzeit sollte sich die Last ändern, da ansonsten Lastverteilung nicht benötigt wird.
  Die Anzahl der adaptiven Werkzeuge ist jedoch stark begrenzt.
- Datenmigration: Es muss möglich sein, zur Laufzeit Last zu verschieben. Programmiersprachen wie z. B. Fortran77 bieten keine dynamische Speicherverwaltung. Implementierungen, die diese Sprache einsetzen, können daher nicht genutzt werden. Dieses reduziert die Anzahl der in Frage kommenden Werkzeuge, da auch heutzutage diese Programmiersprache aufgrund ihrer Unterstützung der Mathematik verbreitet ist. Ein weite-

rer Grund liegt darin, dass viele FEM-Werkzeuge über eine lange Historie verfügen. Diese hochkomplexe und sehr umfangreiche Software zu reimplementieren würde zu einem großen Aufwand führen, ihr Nutzen ist im Verhältnis jedoch zu gering.

Die aktuelle Implementierung von *mLB* wurde speziell auf gitterbasierte Anwendungen abgestimmt. Die Schnittstellen zur Anwendung sind angepasst an deren Erfordernisse. Um diese Schnittstellen nutzen zu können, müssen folgende Punkte erfüllt sein:

- Messbarkeit von Berechnungsabschnitten, die Berechnungseinheiten repräsentieren, die im Verhältnis zur Anzahl der Zellen stehen.
  Als Lasteinheit dient eine Gitterzelle. Es muss daher möglich sein, den Aufwand für die Berechnung einer Zelle zu messen, und diese durch die Schnittstelle *mLB* zu übergeben.
- Vorliegen des dualen Graphen

Diese Voraussetzungen werden von dem Werkzeug *padfem*^{2} [BKM03] erfüllt, das an der Universität Paderborn von Dr. Stephan Blazy und Oliver Marquardt entwickelt wurde.

Für Parameterstudien wurde *mLB* in *padfem*^{2} integriert. Die erzielten Ergebnisse werden im Folgenden vorgestellt.

6.5.2 Parameterstudien

Zeitabhängiges Monitoring:

Monitoring bei jedem kollektiven Aufruf führt zu sehr hohen Laufzeiten. Die gewonnenen Messwerte dienen der Darstellung des aktuellen Netzwerkzustandes und sollen auch jenen in der nahen Zukunft darlegen. Die Aktualität der Messwerte ist also zeitabhängig. Daher wurde ein Zeitintervall gesucht, innerhalb dessen die Messungen höchstens durchgeführt werden.

Es wurde eine Variable *MONITORING_INTERVAL* eingeführt, die den minimalen Abstand in Sekunden zwischen zwei Messungen angibt. Soll eine Messung erfolgen, so wird zunächst überprüft, ob zwischen der letzten Messung und der anstehenden mindestens *MONITORING_INTERVAL* Sekunden liegen. Ist dieses der Fall, so erfolgt eine neue Messung, ansonsten unterbleibt sie.

Testläufe mit *padfem*^{2} haben ergeben, dass ein Monitoring-Intervall von vier Sekunden eine gute Größe darstellt. Der Overhead bleibt damit gering, die Aussagekraft der Messungen ist vorhanden.

Die Nutzung einer festen Variablen stellt keine endgültige Lösung dar. Die Abstände zwischen den Messungen sollten abhängig vom Verhalten der Anwendung sein.

Arbeiten auf neu erstellten Partitionen:

Mit *mLB* werden oftmals nur Teilgitter neu partitioniert. Abhängigkeiten zwischen diesen Teilgittern und dem Gesamtgitter werden nicht betrachtet.
Bei den Testläufen mit *padfem*^{2} ergaben sich jedoch keine Schwierigkeiten bei der Fortführung der Berechnung auf dem neu partitionierten Gitter.

6.6 Geplante Funktionalitäten

Alle Konzepte konnten weitgehend umgesetzt werden. Manche Entscheidungen basieren jedoch auf festen Schwellwerten, anstatt auf dynamisch angepassten Parametern.

Insbesondere sind davon folgende Bereiche betroffen:

DIFF_TOLERANCE: Dieser Parameter bestimmt die Granularität der Subsystem-Bildung. Er ist derzeit auf einen festen Wert (1.6) gesetzt. Hier muss geprüft werden, ob und gegebenenfalls wann eine dynamische Anpassung sinnvoll ist und wie diese aussehen kann.

Anzahl gemittelter Messwerte: Kommunikations- und Berechnungsgeschwindigkeiten bilden sich aus gemittelten Messwerten. Derzeit bilden je 50 Messwerte dafür die Basis. Es muss überprüft werden, inwiefern dieser Wert für die Darstellung des Systemzustands geeignet ist. Ein großer Wert repräsentiert einen durchschnittlichen Zustand. Je kleiner dieser Wert ist, desto mehr repräsentiert er die aktuelle Situation. Eine Untersuchung, ob dieser Wert abhängig von der Dauer der Anwendung oder anderen Parametern ist, ist wünschenswert.

Monitoring-Intervall: Derzeit wird ein zeitabhängiges Monitoring eingesetzt, um ein Überfluten von Messungen zu vermeiden. Geeigneter erscheint jedoch ein Monitoring abhängig vom Verhalten der Anwendung.

In einer FEM-Anwendung werden die Rechenzeiten zwischen zwei Lastverteilungszyklen immer länger, da sich die Anzahl an Zellen im Laufe der Berechnung immer wenig ändert. Monitoring sollte daher auch immer seltener oder aber zeitnah zu dem Lastverteilungszyklus stattfinden. Ein Messen der Kommunikationsverbindungen nur direkt vor dem Lastverteilungsaufruf ist eine weitere Alternative, die überprüft werden müsste.

Neben der Analyse dieser Parameter, muss noch ein Konzept zur Kosten-Nutzen-Analyse entwickelt werden.

Die entstehenden Kosten für Lastverteilung können zum Teil abgeschätzt werden und auf entstandenen basieren, vorausgesetzt die Kosten der letzten Lastverteilungszyklen liegen vor.

Der Nutzen lässt sich jedoch nur schwer feststellen. Es müssten dafür Vorhersagen getroffen werden über eventuelle Laufzeitverbesserungen. Nur dann kann entschieden werden, ob Lastverteilung durchgeführt wird oder nicht.

Derzeit wird als Schwellwert für Lastverteilung der Wert 0,09 gesetzt. Dieser reduziert sich jedoch bei der Lokalisierung des Ungleichgewichts automatisch um 0,01, falls nicht ausreichend Lastverteilungspartner gefunden werden konnten.

6.7 Zusammenfassung

In diesem Kapitel wurde die Erfüllung der Anforderungen an die Lastverteilungssteuerung durch *mLB* überprüft. Dabei wurden zwei Teilaspekte getrennt evaluiert: das Monitoring des Netzwerks und die darauf aufbauenden Funktionen von *mLB* selbst.

Durch Messungen des Aufwandes von Ping-Pong-Kommunikationen wurde die Skalierbarkeit der Monitoring-Funktionen nachgewiesen. Die Evaluierung einer geeigneten Nachrichtengröße für diese Messungen und der zeitlichen Abstände zwischen ihnen, führte zu einem guten Kosten-Nutzen-Verhältnis und bestätigt damit auch die Effizienz des Monitorings.

Die Wirksamkeit von *mLB* wurde durch Laufzeitmessungen eines FEM-Simulators gezeigt. Durch Anpassung der Last an die zur Verfügung stehenden Ressourcen einer Recheneinheit und minimierter Lastverteilung (in Bezug auf Anzahl ihrer Aufrufe, beteiligte PEs und Anzahl zu verteilender Graphknoten) konnten große Laufzeitgewinne erzielt werden. Diesen hohen Gewinnen stehen nur niedrige Zusatzkosten entgegen. Die Effizienz von *mLB* ist daher sehr groß.

Kapitel 7

Resümee

In der vorliegenden Arbeit wurden Konzepte und deren Umsetzung für die Steuerung von Lastverteilung in Grid-Umgebungen vorgestellt. Dabei wurde der Fokus auf die Verteilung von Last datenparalleler Anwendungen gesetzt.

Zunächst wurden dafür die Merkmale von Umgebung, Anwendung und Lastverteilung herausgestellt und analysiert.

Bei der Analyse der Grid-Umgebung zeigte sich, dass ihre Leistungsfähigkeit aus Sicht der Anwendung stark abhängig von der Auslastung ist. Nur durch Monitoring zur Laufzeit kann daher eine Bewertung des Systems erfolgen. Das Monitoring muss dabei Effizienz, Skalierbarkeit und Verfügbarkeit bieten, um die Effizienz der Endanwendung nicht zu gefährden.

Die Untersuchung unterschiedlicher Monitoring-Werkzeuge endete mit dem Resultat, dass keines der vorgestellten Werkzeuge den Anforderungen entsprach. Aus diesem Grund wurde ein eigenes Monitoring entwickelt, das zum einen durch Ping-Pong-Messungen das Netzwerk analysiert und zum anderen die Dauer von Berechnungsschritten der Anwendung misst.
Die Evaluierung dieses Monitorings zeigte seine hohe Effizienz durch geringe Laufzeiten und hohen Informationsgehalt. Gesteuert durch das Verhalten der Anwendung, konnte Verfügbarkeit der Monitoring-Daten gewährleistet werden. In Bezug auf Skalierbarkeit zeigten sich Schwächen des Netzwerkmonitorings. Es stellte sich jedoch heraus, dass diese durch Optimierung eines Algorithmus, der die Kommunikationspartner festlegt, hergestellt werden kann.

Im Mittelpunkt standen adaptive, parallele FEM-Anwendungen. Ihre Arbeitsweise kennzeichnet sich dadurch, dass nach Rechenphasen immer wieder Kommunikationsphasen stattfinden, in denen Informationen zwischen den Prozessoren synchron ausgetauscht werden. Diese Synchronisationspunkte werden unter anderem dazu genutzt, Lastverteilung durchzuführen.

Für Lastverteilung werden dabei meist vorhandene Werkzeuge eingesetzt, die effizient die zu verschiebende Last bestimmen. Ohne Steuerung der Anwendung zielen diese darauf ab, die Last in gleichen Anteilen auf die Prozessoren zu verteilen.

Das im Rahmen dieser Arbeit entwickelte Werkzeug *mLB* greift in diesen Prozess ein, indem es die Höhe der optimalen Last als Eingabeparameter für den Lastverteiler der Anwendung bestimmt. Diese basiert auf den gewonnenen Monitoring-Daten und zielt darauf ab, die Anwendungslast der zur Verfügung stehenden Leistungsfähigkeit anzupassen.

Testergebnisse zeigten, dass dieses Vorgehen zu extremen Laufzeitverbesserungen führt. Durch Anpassung der Last werden die Synchronisationspunkte von allen Prozessen nahezu gleichzeitig erreicht. Wartezeiten an diesen Punkten reduzierten sich um mehrere Faktoren.

Neben dieser effizienten kapazitätenabhängigen Verteilung der Last, wird auch das Netzwerk berücksichtigt. Basierend auf den Ergebnissen des Netzwerkmonitorings wird zur Laufzeit eine Hierarchie aufgebaut, die den aktuellen Status der Kommunikationsverbindungen repräsentiert. *mLB* bestimmt mit Hilfe dieser Hierarchie, welche Prozesse Lastverteilung durchführen und gegebenenfalls, mit welchen anderen Prozessen sie ihre Last umverteilen, stets mit dem Ziel, die schnellsten Verbindungen zu nutzen. Dadurch können mehrere parallel laufende Gruppen von Prozessen, die jeweils untereinander Lastverteilung durchführen, entstehen.

Messungen haben gezeigt, dass auch dieses Konzept ein hohes Leistungspotenzial birgt. Lastverteilung wird oftmals nur auf wenigen Prozessoren, und damit auch mit geringerer Last, aufgerufen. Enorme Zeitersparnis zeigt sich, wenn Lastverteilung nur zwischen Prozessoren, die über gemeinsamen Speicher kommunizieren, durchgeführt wird. Die Laufzeiten des Lastverteilungswerkzeuges reduzieren sich dann drastisch, auch durch die geringere Last, deren Verteilung es neu zu bestimmen gilt.

Mit *mLB* wurde ein verteiltes, skalierbares, adaptives Werkzeug entwickelt, mit dessen Hilfe es gelingt, die Effizienz von Anwendungen in Grid-Umgebungen zu steigern. Durch die minimal notwendigen Eingriffe in den Code, lässt sich *mLB* problemlos in vorhandene Anwendungen integrieren. Das macht es nicht nur zu einem leistungsfähigen, sondern auch anwendbaren Werkzeug für effiziente Berechnungen in der Grid-Umgebung.

Literaturverzeichnis

[AJC00] R.J. Allan, J.M.Brooke, and F. Costen. Grid-based High-Performance Computing. Technical report, UKHEC Collaboration, http://www.dl.ac.uk/TCSC/UKHEC/metacomputing/metacomputing.pdf, May 2000.

[Amd67] G.M. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, page 483. AFIPS Press, 1967.

[APK+03] S. Agarwala, C. Poellabauer, J. Kong, K. Schwan, and M. Wolf. System-Level Resource Monitoring in High-Performance Computing Environments. *Journal of Grid Computing*, 1:273 – 289, 2003.

[Bar95] Stephen T. Barnard. PMRSB: Parallel Multilevel Recursive Spectral Bisection. In *Proceedings of the Supercomputing '95*, December 1995.

[BDH+99] Erik Boman, Karen Devine, Robert Heaphy, Bruce Hendrickson, William Mitchell, Matthew St. John, and Courtenay Vaughan. Zoltan home page. http://www.cs.sandia.gov/Zoltan, 1999.

[BKM03] S. Blazy, O. Kao, and O. Marquardt. padfem – An Efficient, Comfortable Framework for Massively Parallel FEM-Applications. In *Proceedings of EuroPVM/MPI 2003*, pages 681–685, 2003.

[BL02] Christopher A. Bohn and Gary B. Lamont. Load Balancing on a Heterogeneous Cluster of PCs. *Future Generation Computer Systems*, 18:389 – 400, January 2002.

[BS94] Stephen T. Barnard and Host D. Simon. A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. *Concurrency: Practice and Experience*, 6(2):101–17, April 1994.

[Cyb89] George Cybenko. Dynamic Load Balancing for Distributed Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 7:279 – 301, 1989.

[DHB02] Sajal K. Das, Daniel J. Harvey, and Rupak Biswas. MinEX: a latency-tolerant dynamic partitioner for grid computing applications. *Future Generation Computer Systems*, 18:477–489, 2002.

[DMP97] Ralf Diekmann, Burkhard Monien, and Robert Preis. Load Balancing Strategies for Distributed Memory Machines. In H. Satz F. Karsch, B. Monien, editor, *Multi-Scale Phenomena and Their Simulation*, pages 255 – 266. World Scientific Singapore, New Jersey, London, Hong Kong, 1997.

[EHR+98] Th. Eickermann, J. Henrichs, M. Resch, R. Stoy, and R. Völpel. Metacomputing in Gigabit Environments: Networks, Tools and Applications. *Parallel Computing*, 24:1847 – 1872, 1998.

[EMP00] Robert Elsässer, Burkhard Monien, and Robert Preis. Diffusive Load Balancing Schemes on Heterogeneous Networks. In G. Bilardi et al, editor, *12th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, volume 1461, pages 30 – 38, 2000.

[FK03] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 2003.

[FWM94] Geoffrey C. Fox, Roy D. Williams, and Paul C. Messina. *Parallel Computing Works!* Morgan Kaufmann Publishers, Inc., 1994.

[Gen99] Wolfgang Gentzsch. Metacomputing: from workstation clusters to Internet computing. *Future Generation Computer Systems*, 15:537 – 538, 1999.

[ggf] The global grid forum home page. http://www.ggf.org.

[GL99] William Gropp and Ewing L. Lusk. Reproducible Measurements of MPI Performance Characteristics. In Jack Dongarra, Emilio Luque, and Tomas Margalef, editors, *6th European PVM/MPI Users' Group Meeting*, pages 11–18, 1999.

[Glo] The Globus Alliance: Globus Toolkit.

[Gro91] W. Gropp. Parallel Computing and Domain Decomposition. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[GTC+00] D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee. NetLogger: A Toolkit for Distributed System Performance Analysis. In *Proceedings of the IEEE Mascots 2000 Conference*, 2000.

[HB93] Y. F. Hu and R. J. Blake. Initial Experience with Multilevel Recursive Spectral Bisection Algorithm. Information Quaterly for High Performance Computing in Engineering 4, SERC Daresbury Laboratory, October 1993.

[HB99] Y. F. Hu and R. J. Blake. An improved diffusion algorithm for dynamic load balancing. *Parallel Computing*, 25:417–444, 1999.

[hip] Webpage High-Performance Parallel Interface (HIPPI) Standards Activities. http://www.hippi.org.

[IEE93] IEEE. IEEE Standard for the Scalable Coherent Interface (SCI). In *IEEE Std*, pages 1596–1992. IEEE Computer Society, 1993.

[inf] Infiniband. http://www.infinibandta.org.

[JLK+99] H. Jasak, J.Y. Luo, B. Kaludercic, A.D. Gosman, H. Echtle, Z. Liang, F. Wirbeleit, M. Wierse, S. Rips, A. Werner, F. Fernstöm, and A. Karlsson. Rapid CFD Simulation of Internal Combustion Engines. In *International Congress and Exposition, Detroit, Michigan*, number 1999-01-1185. SAE, March 1999.

[jos] Jostle homepage. http://staffweb.cms.gre.ac.uk/∼c.walshaw/jostle/.

[KBM+02] Thilo Kielmann, Henri E. Bal, Jason Maassen, Rob van Nieuwpoort, Lionel Eyraud, Rutger Hofman, and Kees Verstoep. Programming environments for high-performance Grid computing: the Albatross project. *Future Generation Computer Systems*, 18:1113–1125, 2002.

[KDF+03] Ken Kennedy, Jack Dongarra, Geoffrey Fox, William Gropp, and Dan Reed. Parallel Programming Considerations. In Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White, editors, *Sourcebook of Parallel Computing*, pages 43 – 71. Morgan Kaufmann Publishers, 2003.

[KK97] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. Technical report, University of Minnesota, August 1995, updated in 1997.

[KK98] George Karypis and Vipin Kumar. Multilevel Algorithms for Multi-Constraint Graph Partitioning. Technical Report 98-019, University of Minnesota, May 1998.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291 –307, February 1970.

[LÖ1] R. Löhner. *Applied CFD Techniques*. J. Wiley & Sons, 2001.

[Loe] R. Löhner. Pictures from Simulations. http://www.scs.gmu.edu/∼rlohner/pages/pics/compflows.html.

[LMK+03] B. Lowekamp, N. Miller, R. Karrer, T. Gross, and P. Steenkiste. Design, Implementation, and Evaluation of the Remos Network Monitoring System. *Journal of Grid Computing*, 1:75 – 93, 2003.

[LTT] M. Leese, R. Tyer, and R. Tasker. Network Performance Monitoring for the Grid. UK e-Science, 2005 All Hands Meeting, http://gridmon.dl.ac.uk.

[MCC04] Matthew L. Massie, Brent N. Chun, and David E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30, July 2004.

[Mes03] Paul Messina. Distributed Supercomputing Applications. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[MP01] Thierry Monteil and Patricia Pascal. Task Allocation Using Processor Load Prediction on Multiprocessors Cluster. In Dan Grigoras, Alex Nicolau, Bernard Toursel, and Bertil Folliot, editors, *Advanced Environments, Tools, and Applications for Cluster Computing, IWCC 2001*, pages 126 – 135, 2001.

[mpia] MPICH homepage. http://www-unix.mcs.anl.gov/mpi/mpich.

[mpib] The Message Passing Interface (MPI) standard. http://www-unix.mcs.anl.gov/mpi.

[myr] Myrinet. http://www.myri.com/myrinet/overview.

[nws06] Informationsseite über $SGI^{®}NUMAlink^{TM}$ Interconnect Fabric. http://www.sgi.com/products/servers/altix/numalink.html, February 2006.

[OA02] A. Osman and H. Ammar. Dynamic load balancing strategies for parallel computers. In *Proceedings of the First International Symposium on Parallel and Distributed Computing (ISPDC'2002)*, 2002.

[pac] PACX-MPI homepage. http://www.hlrs.de/organization/pds/projects/pacx-mpi.

[par] ParMETIS homepage. http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

[per] Description of MPPTEST. http://www-unix.mcs.anl.gov/mpi/mpich/perftest.

[PK03] N. Podhorszki and P. Kacsuk. Monitoring Message Passing Applications in the Grid with GRM and R-GMA. In *Proceedings of EuroPVM/MPI'2003*, Venice, Italy, 2003.

[pvm] PVM homepage. http://www.csm.ornl.gov/pvm/.

[qua] Qsnet von Quadrics. http://www.quadrics.com.

[RBB+98] M. Resch, T. Beisel, H. Berger, K. Bidmon, E. Gabriel, R. Keller, and D. Rantzau. Clustering T3Es for Metacomputing Applications. In *Cray User Group Proceedings. Available online at http://www.cug.org*, 1998.

[Res01] Michael Resch. *Metacomputing in Simulationsanwendungen*. PhD thesis, Universität Stuttgart, 2001.

[RN01] Tiberiu Rotaru and Hans-Heinrich Nägeli. Heterogeneous Dynamic Load Balancing. In Dan Grigoras, Alex Nicolau, Bernard Toursel, and Bertil Folliot, editors, *Advanced Environments, Tools, and Applications for Cluster Computing, IWCC 2001*, pages 136 – 144, 2001.

[RRS99] Michael M. Resch, Dirk Rantzau, and Robert Stoy. Metacomputing Experience in a Transatlantic Wide Area Application Testbed. *Future Generation Computer Systems*, 15:807 – 816, 1999.

[Sim00] Jens Simon. *Werkzeugunterstützte effiziente Nutzung von Hochleistungsrechnern*. PhD thesis, Paderborn University, Berlin, Februar 2000. Blunk-Verlag, ISBN 3-934445-03-9.

[SKK97] Kirk Schloegel, George Karypis, and Vipin Kumar. Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes. Technical report: 97-013, University of Minnesota, Department of Computer Science, June 1997.

[SKK03] Kirk Schloegel, George Karypis, and Vipin Kumar. Graph partitioning for high-performance scientific simulations. In Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White, editors, *Sourcebook of Parallel Computing*, pages 491 – 541. Morgan Kaufmann Publishers, 2003.

[spe] SPEC (Standard Performance Evaluation Corporation) homepage. http://www.spec.org/.

[spe06] Webpage: All SPEC CPU2000 Results Published by SPEC. http://www.spec.org/cpu2000/results/cpu2000.html, February 2006.

[SSLK00] Arjen Schoneveld, Peter M.A. Sloot, Martin Lees, and Erwan Karyadi. A framework for dynamic load balancing: A case study on explosive containment simulation. *Parallel Computing*, 26:737 – 751, 2000.

[ter] The teragrid home page. http://www.teragrid.org.

[Tur96] S. Turek. Recent Benchmark Computations of Laminar Flow Around a Cylinder. In *Proc. 3rd World Conference in Applied Computational Fluid Mechanics, Freiburg*, 1996.

[WC01] C. Walshaw and M. Cross. Multilevel Mesh Partitioning for Heterogeneous Communication Networks. *Future Generation Comput. Syst.*, 17(5):601 – 623, 2001.

[WCE+95] C. Walshaw, M. Cross, M. G. Everett, S. Johnson, and K. McManus. Partitioning & Mapping of Unstructured Meshes to Parallel Machine Topologies. In A. Ferreira and J. Rolim, editors, *Proceedings of Irregular '95: Parallel Algorithms for Irregularly Structured Problems*, volume 980 of LNCS, pages 121 – 126. Springer, 1995.

[WCE96] C. Walshaw, M. Cross, and M. G. Everett. Parallel Partitioning of Unstructured Meshes. In P. Schiano et al, editor, *Parallel Computational Fluid Dynamics: Algorithms and Results Using Advanced Computers*, pages 174–181. Elsevier, Amsterdam, 1997. (Proc. Parallel CFD'96, Capri, 1996).

[WCE97] C. Walshaw, M. Cross, and M. G. Everett. Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. *Journal of Parallel and Distributed Computing*, 47(2):102–108, 1997. originally published as Univ. Greenwich Tech. Rep. 97/IM/20.

[WLR93] Marc H. Willebeek-LeMair and Anthony P. Reeves. Strategies for Dynamic Load Balancing on Highly Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979 – 993, 1993.

[Wol03] Rich Wolski. Experiences with Predicting Resource Performance On-line in Computational Grid Settings. *ACM SIGMETRICS Performance Evaluation Review*, 30(4):41 – 49, March 2003.

[WSH99] Rich Wolski, Neil T. Spring, and Jim Hayes. A network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15:757 – 768, 1999.

[xt3] Produktbeschreibung Cray XT3. http://www.cray.com/products/xt3/.

Anhang A

Abkürzungsverzeichnis

ALB: Anwendungslastbalancierer

AP: Anwendungsprozess

LAN: Local Area Network

MPP: Massively Parallel Processing

SAN: System Area Network

SMP: Symmetrisches Multiprozessorsystem

WAN: Wide Area Network

Anhang B

Benutzerschnittstelle

Anwendungsmonitor

Für das Monitoring der Anwendung werden folgende Funktionen eingesetzt:

- AppCurrentTime()
- AppMonitor()

```
void AppCurrentTime(void)
```

Ausgabe:

return value elapsed time on the calling processor

Funktion für die Zeitmessungen: gibt die Zeit in Sekunden an, die seit einem bestimmten Zeitpunkt vergangen ist;
derzeit realisiert durch MPI_Wtime() (= gettimeofday())

```
void AppMonitor(char *entry, void *value)
```

Übergabe von Parametern an den AM,
entry gibt dabei an, um welchen Übergabewert es sich handelt, *value* den Wert an sich

Eingabe:

char *entry	one of following strings:	
	"nprocs"	number of processes
	"ncells"	number of global nodes
	"proc"	own ID (MPI rank)
	"lcells"	number of local cells
	"mig_num"	number of cells to migrate
	"mig_proc"	proc ID to which to migrate
	"mig_time"	time for cell migration
	"lb_time"	time for lb tool
	"new_local_nodes"	number of current local cells
	"lb"	0 (=no lb done) or 1 (=lb done)
	"calc_time"	time between 2 lb cycles
	"print"	own proc ID (MPI rank)
	"end_lb_cycle"	indicates next lb cycle
	"string"	prints any string to output file
	"cell_calc_time"	calculation time for one cell
	"get_avg_cell_time"	returns average time for calculation of one cell
void *value	value, depends on entry	

LBS-Modul

Für die Nutzung der Lastverteilungssteuerung stehen der Anwendung folgende Funktionen zur Verfügung:

- AppBeforeLoadBalancing()
- AppDoMetaLB()
- AppAfterLoadBalancing()
- AppAfterLoadTransfer()

```
int AppBeforeLoadBalancing (int myrank, int nproc,
                            int global_nodes, int local_nodes,
                            MPI_Comm com)
```

Eingabe:

int myrank	own rank
int nprocs	number of processes
int global_nodes	total number of load elements (cells)
int local_nodes	number of local load elements (cells)
MPI_Comm com	communicator

Ausgabe:

return value	2 in case of initial partitioning (lb necessary)
	1 if load balancing necessary
	0 if not

Hier wird die Zeitmessung eines Berechnungszyklus gestoppt, Lastparameter erfasst (Lastmenge etc.) und das Lastgleichgewicht bestimmt.

```
void AppDoMetaLB (int nnodes, int offset, int core, int halo,
                  int *index, int *degree, int *node_wt,
                  int *partition, int local_nedges, int *edges,
                  int *edge_wt, int *network, int *processor_wt,
                  int output_level, int dimension, double* coords,
                  int lb_phase, int *global_nodes_lb_part,
                  int *local_nodes_opt)
```

Eingabe:

	alle Jostle Parameter
int lb_phase	load balancing phase (2 = initial lb, 1 = runtime lb)

Ausgabe:

int *global_nodes_lb_part	total number of cells of each load balancing partition
int *local_nodes_opt	optimal number of local cells (based on calculation speed)

Aufruf des Meta-Lastverteilers. Diese Schnittstelle ist angelehnt an den Aufruf zum Anwendungslastverteiler Jostle. Die ersten Parameter sind identisch zu denen von Jostle, nur die letzten drei sind *mLB*-spezifisch.

Durch diesen Aufruf wird das Ungleichgewicht lokalisiert, Datenstrukturen für den ALB (Jostle) vorbereitet und der ALB (Jostle) für das entsprechende Subsystem aufgerufen.

```
void AppAfterLoadBalancing(int myrank, int procs)
```

Eingabe:

int myrank	own rank
int nprocs	number of processes

Hat Lastverteilung stattgefunden, so wird hier lediglich die Zeit für den letzten Lastverteilungszyklus festgehalten und die Erfassung der Migrationszeit gestartet. Das Einbinden dieser Funktion ist optional, da eine Kosten-Nutzen-Rechnung derzeit nicht durchgeführt wird.

```
void AppAfterLoadTransfer(int myrank, int procs)
```

Eingabe:

int myrank	own rank
int nprocs	number of processes

Erfassung der Migrationszeit nach Beendigung der Durchführung. Das Einbinden dieser Funktion ist optional, da eine Kosten-Nutzen-Rechnung derzeit nicht durchgeführt wird.

Anhang C

Architektur der Testumgebung

Folgendes stammt von der Homepage des PC^2 (www.upb.de/pc2):

```
Architecture of the ARMINIUS cluster

   * System Configuration
         o 400 processors 64-bit INTEL Xeon
         o 16 processors AMD Opteron
         o 2.6 TFLOPS peak performance
         o 900 GByte main memory
   * Compute Node Configuration (200 nodes)
         o Dual INTEL Xeon 3.2 GHZ EM64T
         o 4 GByte main memory
         o 80 GByte local disk
         o InfiniBand HCA PCI-e
   * Visualization Node Configuration (8 nodes)
         o Dual AMD Opteron 2.2 GHz AMD64
         o 8 GByte main memory
         o nVidia Quadro FX 4500 PCI-e
         o InfiniBand HCA PCI-e
   * Infiniband Switch Fabric Configuration
         o 216 port InfinIO 9200
   * Disk Storage Configuration
         o 5 TByte Fibre Channel RAID
         o 10 TByte parallel file system
   * Software Configuration
         o 64-bit Linux Redhat AS 4
         o GNU Tools
         o INTEL Compiler C/C++, Fortran
         o Message Passing Interfaces
```

```
      + Scali MPI-Connect
      + NCSA MPICH-vmi
      + OSU MvAPICH
      + INTEL MPI
o Scientific libraries
      + INTEL MKL
      + ATLAS
      + Goto Lib
o Scientific Visualization
      + AMIRA
```