Institut für Parallele und Verteilte Höchstleistungsrechner (IPVR)

Fakultät für Informatik

Universität Stuttgart

Breitwiesenstr. 20-22, D-7000 Stuttgart 80

Globale dynamische Lastbalancierung in datenintensiven Anwendungen

Wolfgang Becker

Fakultätsbericht Nr. 1992 / ??
- vorläufige Version Stand 9/92 -

CR-Klassifikation C.2.4, C.4, D.4.8

Wolfgang.Becker@informatik.uni-stuttgart.de

Globale dynamische Lastbalancierung in datenintensiven Anwendungen

Inhaltsverzeichnis

1 Das HiCon Projekt **4**

1.1 Motivation 4

1.2 Anwendungsgebiete 5

1.3 Die Rechnerumgebung 6

1.4 Das Ausführungsmodell 6

1.5 Der Lastbalancierungsansatz - Konzept und Klassifikation 7

1.6 Literaturüberblick 8

2 Die Lastbalancierung im HiCon Modell **9**

2.1 Auftragsbeschreibung 9

2.2 Lastmessung 10

2.3 Die Planungs- und Zuweisungsstrategie 11

2.3.1 Die Ebene der Auftragsplazierung 14

2.3.2 Die Ebene der Datenverteilung 16

2.3.3 Vorplanung von Auftragsgruppen - die Konfigurationsebene 16

2.4 Hierarchisch verteilte Lastverwaltung 18

3 Realisierung und Ergebnisse **19**

3.1 Simulation des HiCon Modells 19

3.2 Loadman - Prototyp eines Lastbalancierers nach dem HiCon Modell 20

3.2.1 Systemstruktur und Anwendungsschnittstelle 20

3.2.2 Verwaltung und Synchronisation der Klassenkontexte 23

3.2.3 Fehlerbehandlung 24

3.2.4 Verfügbare Strategien 24

3.2.5 Evaluierung eines Anwendungsbeispiels 25

3.2.6 Ausblick 28

Literaturverzeichnis **29**

1 Das HiCon Projekt

Dieser Bericht stellt einen Ansatz zur dynamischen Lastbalancierung paralleler und verteilter Programme vor. Im ersten Kapitel wird das Projekt anhand der betrachteten Anwendungen, Rechnersysteme und Lastbalancierungsverfahren charakterisiert. Das Balancierungsprinzip wird in Kapitel 2 ausführlich erklärt, während Kapitel 3 Realisierungen und Ergebnisse des Modells vorstellt.Kapitel 3.2.6 gibt schließlich eine abschließende Betrachtung.

1.1 Motivation

In der Vielzahl in der Literatur existierender Untersuchungen zur Lastbalancierung zeichnen sich einige grundsätzliche Probleme ab, von denen wir einige durch den in diesem Bericht vorgestellten Ansatz bewältigen wollen. Die Realität stellt Lastbalancierungsverfahren vor ein bestehendes System, dessen Struktur und Schnittstellen meist für eine automatische Lastverteilung schlecht geeignet sind. Es besteht kaum Information über das zu erwartende Verhalten von Anwendungen, Betriebssystemschnittstellen schränken die Funktionalität und Effizienz der Lastbalancierung ein und zwischen den einzelnen Schichten der Software findet kein Informationsaustausch statt. Die transparente Verteilung von Abläufen und Daten ist ein wesentlicher Fortschritt für Programmierer, Lastbalancierungsverfahren müssen jedoch um den damit verbundenen Synchronisations- und Nachrichtenaufwand wissen.

Wir haben daher ein Verarbeitungsmodell entwickelt, das dem Programmierer transparente Verteilung und Replikation paralleler Ausführungen und Daten in heterogenen Systemen ermöglicht, dem Lastbalancierungsmechanismus aber eine Kostenabschätzung erlaubt. Die explizite Integration von Datenbeständen im Konzept erlaubt nunmehr eine Balancierung aufgrund der drei relevanten Faktoren - des Rechenaufwandes, der Datenlokalität und des Kommunikationsbedarfs. Unter dem Gesichtspunkt der mehrschichtigen Lastbalancierung kann der Zusammenhang zwischen den Ebenen (grob differenziert etwa in Konfiguration, Auftragszuweisung, Datenverteilung und Nachrichten-Routing) genutzt werden. Die Kontextverwaltung (siehe unten) ermöglicht weiterhin sowohl die dynamische Migration und Replikation von Daten als auch die Migration von Servern bzw. laufenden kooperierenden Teilausführungen in heterogenen Architekturen auf einer höheren Ebene.

Das hier vorgestellte Lastbalancierungskonzept zeichnet sich vor allem dadurch aus, daß es sowohl eine vollständigere Menge relevanter Faktoren und Größen berücksichtigt als auch einen komplexeren Zuweisungsmechanismus benutzt. Dadurch kann eine breitere Klasse von Applikationen und Systemkonfigurationen durch ein anwendungsunabhängiges automatisches Lastbalancierungsverfahren effizient betrieben werden.

An dieser Stelle wollen wir in einer kurzen Entwicklung von Lastbalancierungsverfahren zeigen, daß in der Tat ein Bedarf an fortgeschrittenen Methoden besteht:

Primitive Balancierungsverfahren arbeiten ohne Messung des Systemverhaltens und ohne Vorwissen über die Aufträge. Die wichtigsten Beispiele sind das Random- sowie das Round Robin-Verfahren, die sich in der Realität bewähren (siehe etwa [Trippner92]). Sie basieren auf der Annahme, Aufträge und Rechner seien völlig homogen. Weitere implizite Annahmen werden wir im folgenden noch sehen. Diese Verfahren versagen bereits bei einer schwankenden Grundlast auf dem System (Mehrbenutzerbetrieb).

Bessere Verfahren berücksichtigen das Systemverhalten. Üblicherweise mißt man die Run Queue Length der Prozessoren (siehe [Ferrari86]). Sind mehr Prozessoren (bzw. Server) frei als Aufträge anstehen, so bevorzugt man Prozessoren mit kurzer Run Queue, andernfalls gibt man den Auftrag dem ersten Server, der sich frei meldet (erfolgreicher Einsatz etwa im *Supervisor* [Schiele91], auf dem das *HiCon*-Projekt aufbaut). Auch hier unterstellt man unter anderem allen Aufträgen denselben Aufwand. Verfahren dieser Kategorie geraten in Schwierigkeiten, wenn man etwa datenintensive Berechnungen betrachtet. Der erhebliche Aufwand zum Heranschaffen der Daten bzw. die Beauftragung des dortigen Servers zerstört oft die geplante Lastverteilung.

Zieht man außer der Systemlast auch noch Wissen über die Aufträge in Betracht, so erhält man günstigere theoretische Ergebnisse. Das HiCon Modell betrachtet neben der Länge der Server-Warteschlangen und der Sekundärlast auch den Rechenumfang und Datenbedarf der Aufträge. In der Praxis stößt man jedoch auf massive Probleme (Aufwand der Balancierung selbst, Ungenauigkeit der Angaben). Man ignoriert hier Reihenfolgeabhängigkeiten und Kommunikation zwischen Aufträgen. Diese Verfahren zeigen ihre Schwäche bei Anwendungen, die nebst Parallelität einen kritischen sequentiellen Pfad enthalten und in Fällen intensiv kooperierender paralleler Teilaufträge.

Balancierungsalgorithmen, die auch diese Situationen meistern, findet man in der Praxis nicht. Jedoch existieren, vor allem im Bereich der statischen Planung (Scheduling, siehe etwa [Ma82], [Blazewicz86], [Thomasian86], [Towsley86], [Li90] oder [Kanet91]), Ansätze für Teilprobleme. Hier zwingt der große, exponentiell mit Aufträgen und Rechnern wachsende Balancierungs-Overhead zu sehr einfachen Heuristiken.

1.2 Anwendungsgebiete

Im HiCon Projekt untersuchen wir im wesentlichen datenintensive Berechnungen. Darunter fallen der Bereich der Verwaltung großer Datenmengen (Informationssysteme) sowie Applikationen, die umfangreiche Transformationen auf relativ stabilen, globalen Datenstrukturen durchführen, an. Diese Charakterisierung soll die betrachtete Domäne einerseits von Datenflußanwendungen abgrenzen, bei welchen man typischerweise kleine Datenmengen durch eine Reihe von Bearbeitungsfunktionen schleust, und andererseits von stark dialogorientierten Applikationen trennen, die viele sehr kurze, unvermittelt auftretende Verarbeitungsschritte aufweisen.

Weiterhin sollen hauptsächlich mäßige bis grobe Parallelisierungen betrachtet werden. Das bedeutet, daß die sequentiellen Teiloperationen sowohl einen gewissen Rechenumfang haben als auch eine größere Menge von Daten bearbeiten. In unserem Umfeld ist es daher nicht das Ziel, die maximal mögliche Parallelität eines Problems zu nutzen; oft ist es sinnvoller, Algorithmen anzuwenden, die ein relativ grobes Granulat aufweisen und somit besser auf die Fähigkeiten der betrachteten Hardware und auf die Methoden der Lastbalancierung abgestimmt sind.

Die Auswahl des Anwendungsgebietes bzw. der Algorithmenklasse hat Einfluß auf das gewählte Ausführungsmodell sowie auf die Klasse der zu untersuchenden Lastbalancierungsmethoden. So sind neben der Rechenkapazität die Verfügbarkeit und der Lagerort von Daten relevante Faktoren. Aufwendige, zentralisierte Ansätze zur Lastbalancierung sind erfolgversprechend.

1.3 Die Rechnerumgebung

Heutige Rechensysteme verfügen typischerweise über sehr starke zentrale Rechenleistung und vergleichsweise langsame und aufwendige Kommunikationsmöglichkeiten. Massiv parallele Architekturen sind (Software-seitig) äußerst schwer zu beherrschen und werden sich langfristig in Richtung der zentralen Prozessoren mit leistungsfähiger Vektor- und Pipeline-Unterstützung bewegen. Diese Überlegungen und der oben gewählte Fokus betrachteter Applikationen lassen es sinnvoll erscheinen, ein Rechnermodell bestehend aus relativ wenigen, sehr starken und über eigene Ressourcen verfügenden, lose gekoppelten Prozessoren zugrunde zu legen. Die Entscheidung für dieses Modell schließt die Einbindung anderer Architekturen in HiCon nicht aus, jedoch wird die Lastbalancierungsstruktur andere Systeme (z.B. SIMD oder Shared Memory) nicht optimal nutzen.

1.4 Das Ausführungsmodell

Applikationen nutzen die Parallelverarbeitung, indem der Algorithmus auf mehrere kooperierende Prozesse verteilt wird. Das Granulat der Aufteilung in Teilberechnungen sowie die Kooperationsform zwischen diesen wird durch die jeweilige Anwendung eingeschränkt und sollte außerdem auf die verwendete Rechenumgebung angepaßt werden.

Eine automatische Lastbalancierung benötigt aber neben dem Wissen über das Verhalten der Hardware auch Wissen über die Struktur und den Ablauf der Anwendungen. Wir wollen uns bewußt nicht darauf beschränken, Prozesse unbekannten Verhaltens bezüglich ihrer Rechenarbeit, des Kommunikationsverhaltens und ihrer Datenzugriffe zu balancieren. Solche Vorabschätzungen können über die Beschreibung einzelner sequentieller Einheiten hinaus auch die Kooperationsform, die gegenseitigen Abhängigkeiten zwischen diesen oder Ablaufhäufigkeiten der Einzelberechnungen umfassen.

Im HiCon Projekt entwickeln wir ein Programmier- und Ablaufmodell, das eine auf die Hardware angepaßte, relativ flexible Implementierung von Algorithmen erlaubt und der Lastbalancierung entscheidende Informationen zur günstigen Abwicklung zur Verfügung stellen kann. Eine Anwendung gliedert sich in Teilfunktionen, die durch je eine Serverklasse modelliert werden. Diese Funktionen können sich beliebig gegenseitig Nachrichten bzw. Aufrufe und Resultate zusenden. Die Funktionalität einer Serverklasse wird durch mehrere Serverinstanzen realisiert, die als je ein Prozeß auf einem Prozessor statisch aufgesetzt werden. Obwohl Serverinstanzen zur Laufzeit generiert und gelöscht werden können, liegt hier der Gedanke einer statischen Server-Konfiguration zugrunde. Diese Entscheidung ergibt sich aus den auf heutigen Betriebssystemen hohen Prozeßstartkosten, aus dem relativ groben Granulat der Parallelität (Verwendung von Klassen- und Instanz-Warteschlangen; maßvolle Parallelarbeit erweist sich als effektiv) sowie aus dem nicht zu vernachlässigenden Datenbestand der Serverinstanzen (siehe unten). Die Anzahl und Verteilung der Serverinstanzen ist für den Programmierer nicht sichtbar, wohl aber die Existenz einzelner Instanzen. So können Nachrichten an eine Funktion (Klasse) oder an eine spezielle Instanz gerichtet werden (Resultatrückgabe oder stehende Verbindung zum Zwecke intensiver Kommunikation).

Wir betrachten keine Prozeßmigration auf Betriebssystem-Ebene, obwohl dies in einigen Projekten erfolgreich angewandt wird (siehe etwa [Ezzat86], [Douglis91]), da unser Lastbalancierungsmodell zum einen für heterogene Systeme ausgelegt ist, zwischen denen es in absehbarer Zeit noch keinen Standard geben wird, welcher das Verschicken der Prozeßkontrollblöcke sowie die

transparente ‘Verlängerung’ von I/O-Verbindungen erlaubt; zum anderen ist Prozeßmigration hauptsächlich in Modellen mit dynamischer Prozeßerzeugung interessant (Migrationskosten bewegen sich in der Größenordnung eines Prozeßstarts). In Client-Server Umgebungen sind Prozeßstart und -stopp längerfristige Rekonfigurationsoperationen der Server-Struktur.

Durch den im folgenden vorgestellten Kontext der Serverklassen ist jedoch eine Migration auf höherer Ebene durch ‘Sessions’ möglich (die Aufspaltung langlaufender Funktionen unter Weiterverwendung der Daten verlangt hier keine feste Bindung an eine Serverinstanz).

Funktionen sind im HiCon Modell nicht notwendigerweise kontextfrei, sondern jede Funktion (Serverklasse) verfügt über einen statischen, lokalen Datenbestand. Der Datenbestand einer Funktion kann sich aus mehreren Partitionen zusammensetzen. Hier kann der Programmierer die daten-parallele Ausführung von Funktionen ermöglichen, denn er kann Partitionen zum Zugriff sperren und nach Abschluß eines kritischen bzw. zusammengehörigen Abschnittes wieder freigeben. Die Verteilung, Duplizierung und Konsistenterhaltung der Datenpartitionen ist für den Programmierer nicht zu sehen, sondern wird vom Laufzeitsystem unter Mitarbeit der Lastbalancierung abgewickelt. Der - im weiteren auch mit Kontext bezeichnete - Datenbestand einer Funktion umfaßt beliebige Datenstrukturen im Hauptspeicher und auf dem jeweils lokalen Plattenspeicher (flüchtige und nichtflüchtige Daten) und wird von den Instanzen einer Klasse als gemeinsamer Kontextspeicher betrachtet.

Das prozedurale Programmiermodell wird durch gegenseitige Aufrufe von Serverklassen unterstützt, wobei die Interaktion zwischen Client und Server im Prinzip nicht auf die Aufrufparameter und das Resultat beschränkt ist. Das in *Fortran* bzw. im Datenbankbereich übliche Arbeiten auf globalen Datenstrukturen ist im HiCon Modell nur eingeschränkt möglich: die zunehmend allgemein akzeptierten objektorientierten Konzepte verlangen, daß Datenstrukturen in Objekte gekapselt werden und nur über die dort vorhandenen Methoden (durch das Objekt) bearbeitet werden können. Im HiCon Modell wird eine Datenstruktur (in Gestalt des Kontextes) durch eine Serverklasse mit deren Aufrufinterface gekapselt. Parallelarbeit wird durch Replikation von Serverinstanzen erreicht, auf die auch die Daten verteilt sind.

1.5 Der Lastbalancierungsansatz - Konzept und Klassifikation

Im HiCon Projekt untersuchen wir globale zentrale Lastbalancierungsverfahren. Dabei können die Entscheidungsträger der Balancierung durchaus in Form einer hierarchischen Struktur über das Rechnersystem verteilt sein. Unter den oben umrissenen Vorgaben und Bedingungen erscheint eine Zentralstelle zur Verwaltung der Zustandsinformation und der darauf basierenden Entscheidungsfindung sinnvoll. Um zu vermeiden, daß diese einen Engpaß bildet, ist eine Einschränkung ihres Wirkungsbereiches notwendig. Lokale Balancierungskomponenten koordinieren sich auf einer abstrakteren Ebene. Offensichtlich kann das Ziel der global optimalen Lastverteilung in unserer Umgebung aufgrund des Aufwandes für die Lastbalancierung selbst nicht vollständig erreicht werden.

Ein zentraler Balancierungsansatz ist nicht selbstverständlich, zumal sich in der Literatur viele erfolgversprechende dezentrale Verfahren finden (siehe etwa [Barak85], [Eager85], [Eager86], [Hsu86], [Lin87], [Kale88], [Cybenko89], [Hosseini90] und [Kuchen90]). Im MIMD-Bereich werden Nachrichten durch schnell wachsende lokale Prozessorleistung und sehr beschränkt steigende Kommunikationsgeschwindigkeit immer teurer. Das erfordert eine eher grobgranulare Parallelisierung der Anwendungen. Bei der Balancierung größerer Einheiten lohnt es sich jedoch,

genauere Informationen über Aufträge und Systemzustand einzusetzen, was durch einen (logisch) zentralisierten Ansatz ermöglicht wird (siehe auch Kapitel 2.4).

Für SIMD-Systeme sowie Vektor- und Pipeline-Einheiten ist automatische dynamische Lastbalancierung weniger relevant, da man weder Mehrbenutzerbetrieb noch funktionale Parallelität beobachtet. Die auftretende massive Datenparallelität wird derzeit am besten durch einfache, statische Algorithmen balanciert.

Die Aufgabe der Lastbalancierung besteht im wesentlichen darin, Teilaufgaben bestimmten Instanzen zur Ausführung zuzuordnen. Die Verteilung von Daten steht nicht unter dem unmittelbaren Zugriff der Balancierung, wird aber bei der Zuweisungsentscheidung berücksichtigt und damit durch die Plazierung der Teilaufgaben in vorhersehbarer Weise beeinflußt (siehe Kapitel 3.2.2). Die zu balancierenden Objekte sind hier sowohl einzelne Aufrufe zwischen Servern als auch ganze Gruppen von Aufrufen, d.h. die Ausführung komplexer Funktionen.

Das Optimierungskriterium der Lastbalancierung ist die im Mittel möglichst schnelle Ausführung (Antwortzeit) aller im System ablaufenden Aufträge. Das bedeutet, daß ein neuer Auftrag nicht isoliert betrachtet wird, sondern so in das aktuelle Systemgeschehen eingebunden wird, daß seine Laufzeit einschließlich der Laufzeitzunahme der übrigen Aufträge kleinstmöglich ausfällt (soziale Lastbalancierung). Bei der Balancierung von Auftragsgruppen wird lediglich die Antwortzeit des Hauptauftrages minimiert, da die anderen zugehörigen Teilaufgaben als Unteraufträge angesehen werden, deren einzelne Laufzeit nicht interessiert (die Ausführung der Unteraufträge wird also auf schnellstmögliche Bearbeitungszeit des Hauptauftrages ausgelegt).

Das Balancierungsverfahren agiert dynamisch, d.h. mißt zur Laufzeit Ressourcenbelastung und Antwortzeitverhalten berücksichtigt diese im weiteren Vorgehen. Wir denken befassen uns hingegen weniger mit adaptiven Verfahren, die aufgrund von Laufzeitauswertungen dynamisch ihre Balancierungsstrategie (d.h. nicht nur die Parameter der Entscheidungsfunktion sondern den Algorithmus selbst) ändern. Die Grenzen dieser Aufgliederung sind jedoch fließend.

1.6 Literaturüberblick

Wegen der Vielzahl veröffentlichter Studien zur Lastbalancierung im weiten Sinne beschränken wir uns auf komplexere zentrale Verfahren im Bereich datenintensiver Anwendungen. Für eine allgemeinere Einführung sei auf [Becker92], [Casavant88] oder [He89] verwiesen.

Yu, Balsamo, Ciciani, Dias, Lee und Leff stellen in [Yu86], [Ciciani88] und [Yu91] Verfahren vor, um Transaktionen an verteilte Datenbank-Server zuzuweisen. Die eigentliche Datenzugriffe werden vom Server am Ort der Daten durchgeführt. Lastbalancierung besteht hier in der Abwägung zwischen Ausnutzung der Rechenkapazitäten und dem Aufwand für Remote-Datenzugriffe.

Copeland, Alexander, Boughter und Keller [Copeland88] betrachten die Partitionierung und Verteilung von Daten über Platten in Anwendungen mit exzessiven Datenzugriffen. Sie gehen davon aus, daß Operationen direkt am Ort der Daten durchzuführen sind. Gavish und Sheng [Gavish90] geben einen Überblick diverser Datei-Allokationsverfahren.

Varadarajan und Ma [Varadarajan88] untersuchen Datei-Migration in verteilten Datenbanken. Anhand der Datenverteilung und Zugriffsmuster wird erwogen, ob Datenmigration oder Ausführung des Auftrages am Ort der Daten günstiger ist. Dabei werden Daten nur zwischen und Aufträge nur innerhalb von Regionen bewegt. Innerhalb einer Region wird der Auftrag bzw. der

Datenzugriff am Ort der Daten durchgeführt. Optimierungskriterium ist die Minimierung der Migrationskosten, wobei die Ausführungszeiten ein Zeitlimit einhalten müssen.

Smith beschreibt in [Smith80] ein komplexes Verfahren der Interaktion zwischen Clienten und Servern zur optimalen Lastverteilung. Es ist eine Variante des allgemeinen Bidding-Ansatzes (siehe [He89], Kapitel 2.2.A und 2.2.B).

Thomasian [Thomasian86] analysiert den Verlauf von Anwendungen, die durch Ressourcenbedarf und Reihenfolgebeziehungen charakterisiert werden, auf dem Modell eines verteilten Rechnersystems anhand des Queueing Network Ansatzes.

2 Die Lastbalancierung im HiCon Modell

Dieses Kapitel gibt eine detaillierte Beschreibung der Struktur und Funktionsweise des im HiCon Projekt entwickelten Lastbalancierungsverfahrens.

2.1 Auftragsbeschreibung

Ohne eine statische Voreinschätzung des Verhaltens eines Auftrages muß die Lastbalancierungsstrategie ein festes Standardprofil für alle Aufträge annehmen, was oft zu gravierenden Fehlentscheidungen führt. Dennoch begnügen sich nahezu alle bekannten Verfahren damit, weil in den meisten Anwendungen keine exakten Vorhersagen über das Laufzeitverhalten möglich sind; sie hängen stark von aktuellen Laufzeitdaten ab. Statische Profile werden vor allem in Scheduling-Verfahren für Produktionsplanung und Batch-Betrieb verwendet. Für dynamische Lastbalancierung sollte das Profil eines Auftrages so spät wie möglich, d.h. erst unmittelbar bei seiner Zuweisung, festgestellt und genutzt werden. Zu diesem Zeitpunkt sind relative verläßliche Abschätzungen über die Ressourcenbedürfnisse möglich. Im HiCon Modell kann ein Aufrufer beim Versenden eines Unterauftrags das Profil durch folgende Größen charakterisieren:

- Ein Maß für die benötigte CPU-Rechenzeit sowie die Anzahl der notwendigen Plattenzugriffe. Unser Modell kennt dazu keine weiteren Aufgliederungen, da eine Serverinstanz auf einem Prozessor arbeitet, nicht migriert und ausschließlich auf lokale Platten zugreift. Eine explizite Unterscheidung zwischen mehreren lokalen Platten wurde unterlassen, da solche auf längere Sicht in Form von Disk-Arrays als eine logische Platte organisiert werden.

  Anhand dieser beiden Größen kann man neben absoluten Laufzeitabschätzungen anhand des Verhältnisses zwischen Rechenzeit und Zugriffswartezeit auch die Auslastung der Platte sowie die prozentuale Prozessorauslastung bestimmen. Dabei sind allerdings noch synchronisationsbedingte Wartezeiten zu beachten.

- Eine nach lesendem und änderndem Zugriff differenzierte Auflistung der angefaßten Datensätze. Während es in einer allgemeinen Programmierumgebung nahezu unmöglich ist, dies in einheitlicher Form zu beschreiben, können diese Angaben im HiCon Modell relativ einfach ermittelt werden, da jede Serverklasse über einen eigenen Datenbestand verfügt und nur auf diesen zugreift (siehe Kapitel 1.4). Der Datensatz einer Klasse ist in eine ‘überschaubare’ Anzahl von Partitionen aufgeteilt, die einzeln spezifiziert werden können und gleichzeitig im Ausführungsmodell die Einheiten der Kontextsynchronisation darstellen. Die Betrachtung einzelner Partitionen setzt ein relativ grobes Granulat voraus, bei feinerer Aufspaltung der Daten sind statistische Angaben sinnvoller.

- Eine Abschätzung der Unteraufträge, die während der Auftragsbearbeitung entstehen werden. Dabei ist nicht allein interessant, wieviel Aufrufe an welche Serverklassen gehen, sondern auch die zeitliche Verteilung der Unteraufrufe und die zu erwartenden Profile. Bei synchronen Unteraufrufen ist die dadurch entstehende Abhängigkeit zu vermerken.
- Die Genauigkeit bzw. Sicherheit der statischen Vorabschätzung. Das ermöglicht der Lastbalancierung, die Profile entsprechend ihrer Relevanz zu berücksichtigen (siehe Kapitel 2.3).

Aufgrund dieser Angaben kann der Balancierungsmechanismus Zuweisungsentscheidungen treffen.

Eigentlich sollte nicht der Aufrufer (Client), sondern der aufgerufene Server diese Vorabschätzungen an die Lastbalancierung liefern, denn für ihn ist es leichter, aus den aktuellen Aufrufparametern den vermutlichen Bearbeitungsaufwand zu ersehen. Zum Aufrufzeitpunkt steht die Serverinstanz jedoch noch nicht fest, sodaß in einer Art Ausschreibung alle Instanzen befragt werden müßten. Das ist ein erheblicher Zeit- und Nachrichtenaufwand (siehe etwa [Smith80]).

Mit den oben vorgestellten Größen lassen sich Reihenfolgebeziehungen oder Schleifen (Wiederholungen) innerhalb eines Aufrufes nur schlecht ausdrücken. Solche Angaben müssen jeweils für ganze Auftragsgruppen spezifiziert werden. Beziehungen zwischen Aufträgen können sehr wesentlich sein und werden daher im HiCon Modell berücksichtigt:

- Zu einem Auftrag können die bei der Abarbeitung involvierten Unteraufträge in Form einer Menge von Teilaufträgen angegeben werden. Wir verwenden im HiCon Modell jedoch keine hierarchische Aufrufkette sondern eine ‘flache’ Menge kooperierender Teilaufträge. Zu jedem Teilauftrag wird daher die Intensität der Kooperation (die Anzahl und Häufigkeit der Nachrichten) zwischen den verschiedenen Teilaufträgen sowie die entstehende Parallelität abgeschätzt. Zusätzlich können Reihenfolgeabhängigkeiten zwischen den Teilaufträgen spezifiziert werden. All diese Angaben sind abgekoppelt von der tatsächlichen Synchronisation durch die Serverinstanzen zur Laufzeit; es sind lediglich Vermutungen, welche die Lastbalancierung unterstützen.

  Offensichtlich eignet sich diese Beschreibungsform nicht für jede Struktur und jedes Granulat in parallelen und verteilten Algorithmen. An dieser Stelle fehlt uns bislang die notwendige Vielfalt an realisierten Anwendungen, sodaß wir die Profilangaben zunächst auf die Bedürfnisse des Balancierungsverfahrens zugeschnitten haben.

Eine Beschreibung von Auftragsgruppen kann zur Laufzeit vor Absendung der einzelnen Aufträge geschehen. Daraufhin reserviert der Balancierungsalgorithmus entsprechende Kapazitäten (Rekonfiguration der Serverinstanzen) und nimmt eine vorläufige grobe Zuweisung der Einzelaufträge vor. Oft wird auf die Information über Abhängigkeiten, Kooperation und Parallelität innerhalb von Auftragsgruppen verzichtet, um die Komplexität und die Selbstkosten der Balancierung gering zu halten. Diese Angaben sind ohnehin nur wertvoll, wenn sie eine gewisse Genauigkeit und Wahrscheinlichkeit aufweisen.

2.2 Lastmessung

Dynamische Lastbalancierungsverfahren messen zur Laufzeit die Auslastung ihrer Ressourcen (Prozessoren, Platten und Verbindungsnetze bzw. -Busse) und benutzen diese Information zur Umverteilung laufender, zur Zuweisung neuer Aufträge und zur Verteilung der Daten. Im HiCon Modell werden prinzipiell keine in Bearbeitung befindlichen Aufträge mehr migriert und Daten werden implizit durch Auftragszuweisung verschoben (diskutiert in Kapitel 1.4).

Für die Lastbalancierung ist weder die maximale Leistung einer Ressource noch die Systemlast von primärem Interesse, sondern die Zeit, die eine Instruktion (bzw. eine Nachricht, eine I/O) derzeit auf der Ressource in Anspruch nehmen würde.

Der Aufwand zur Messung des aktuellen Zustandes während des laufenden Betriebes ist eine Zusatzbelastung sowohl für die Ressourcen als auch für den zentralen Balancierungsagenten, sodaß man sich auf die notwendigen Messungen beschränken muß. So verzichten wir etwa bewußt auf die Messung der durch einzelne Serverinstanzen induzierten Ressourcenbelastung (Lastmessung einzelner Prozesse). Üblicherweise wird die Zahl der ausführbaren Prozesse auf einem Prozessor (Run Queue Length) als einzig entscheidender Lastfaktor angesehen (siehe [Ferrari86]), während die 'CPU Busy Time', die Plattencontrollerauslastung sowie die Belastung des Netzwerkes für dynamische Lastbalancierung weniger relevant scheint: Für eine Lastbalancierungsentscheidung ist die Leistung interessant, die ein Prozessor dem neuen Auftrag zur Verfügung stellen kann (beachte dabei das Konzept der sozialen Lastbalancierung, Kapitel 3.2.1). Bei CPU-gebundenen Aufträgen kann man dazu die Prozessorleistung durch die Anzahl der aktiven Aufträge (Run Queue Length) dividieren.

Üblicherweise läuft auf jedem Prozessor ein Messprozeß, der den lokalen Systemzustand periodisch zum Lastbalancierer sendet. Um das Nachrichtenaufkommen zu minimieren sendet er nur im Falle einer signifikanten Laständerung. Alternativ ist denkbar, daß die Serverinstanzen jeder Nachricht lokale Lastinformationen beifügen, was aber komplexer ist und starke Schwankung der Messintervalllänge zur Folge hat.

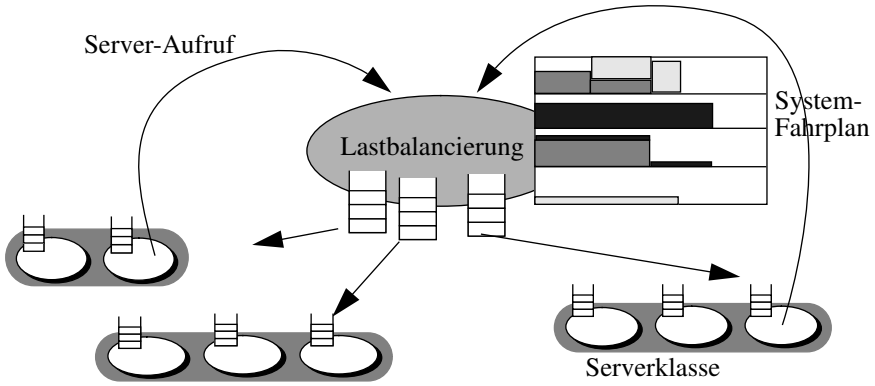
Im HiCon Modell verfügen wir über eine relativ statische Konfiguration von 'single threaded' Serverinstanzen, sodaß deren aktueller Arbeitszustand zu berücksichtigen ist. Aufträge erzeugen weder einen neuen Prozeß noch einen neuen Thread, sondern werden einer Serverinstanz zugewiesen, die ihn bearbeitet, sobald sie ihren derzeitigen und alle schon in ihrer Warteschlange befindlichen Aufträge erledigt hat. Der Zustand einer Serverinstanz hängt mit der aktuellen und zu erwartenden Prozessor- und Plattenauslastung zusammen, wird aber getrennt aufgenommen und versandt. Die Ressourcen-Lastmessung dient hier zur Korrektur der vom Lastbalancierer aus dem Anwendungsprofil vermuteten Auslastungen.

Jede Serverklasse hat einen Kontext, der in Partitionen auf die Instanzen verteilt ist und zur Laufzeit durch Zugriffsanforderungen zwischen den Instanzen ausgetauscht und kopiert wird. Der aktuelle Lagerort der Kontextpartitionen stellt damit eine weitere dynamische Systemzustandsbeschreibung dar. Da im derzeitigen Modell alle Nachrichten den Lastbalancierer passieren, ist die Balancierungsinstanz stets über die aktuelle Datenverteilung informiert; es sind keine zusätzlichen Nachrichten zur Übermittlung der Instanzenzustände oder der Kontextlokationen notwendig.

2.3 Die Planungs- und Zuweisungsstrategie

Im HiCon Projekt geschieht Lastbalancierung bislang in Form eines zentralen Prozesses, der sowohl die Zuweisung (Ort und Zeitpunkt) als auch die Einplanung von Auftragsgruppen (Server-Konfiguration, Reservierung) durchführt. Dieser Prozeß soll später physisch verteilt werden (siehe Kapitel 2.4). Der Lastbalancierer hat einen Eingangspuffer in welchen er neu entstandene Aufträge, Ergebnisse und Kontextverschiebungen gesendet bekommt. Darunter laufen auch Mitteilungen über den aktuellen Systemzustand ein. Anhand seines Systemfahrplanes (siehe unten) schätzt er die Eignung der verfügbaren Serverinstanzen zur Übernahme des Auftrages ab. Der

Auftrag wird daraufhin in den Puffer der Serverklasse abgelegt, wo er verbleibt bis er an eine Serverinstanz verschickt wird. Ein neuer Auftrag muß nicht unmittelbar fest an eine Instanz vergeben werden, braucht andererseits aber auch nicht zu warten, bis die gewünschte Serverinstanz frei wird. Die Serverinstanzen verwalten einen Puffer anstehender Aufträge, den sie sequentiell abarbeiten. Die Prozeßstruktur einer Anwendung ist im folgenden Bild noch einmal veranschaulicht.



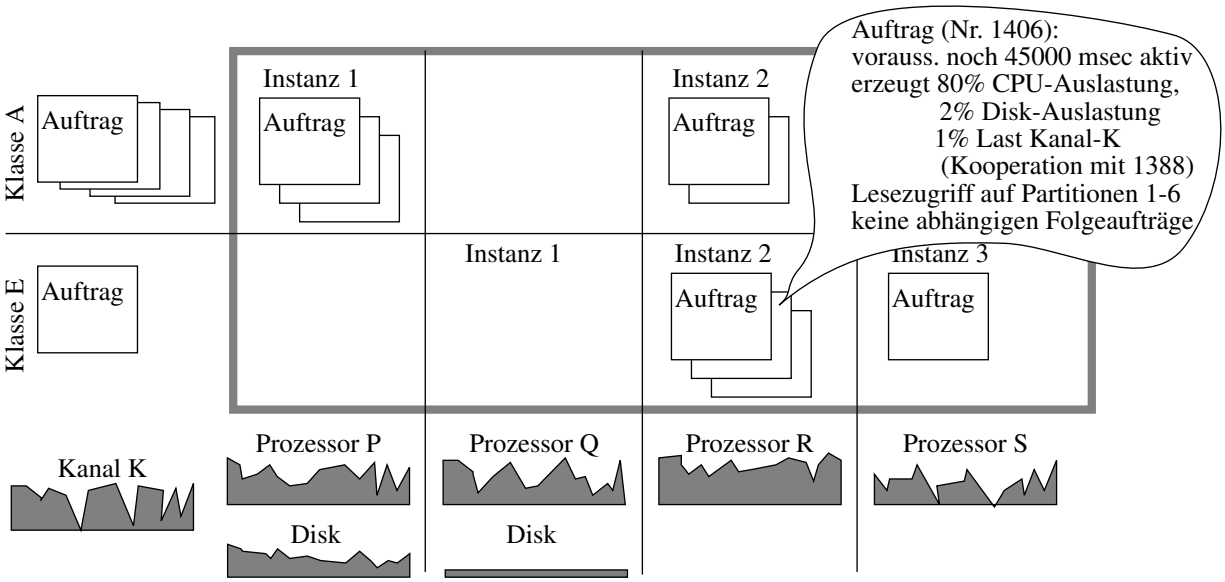
Das häufig praktizierte Verfahren, bei dem sich ein Server frei meldet und daraufhin einen neuen Auftrag bekommt, hat den Vorteil, daß die endgültige Festlegung der Serverinstanz so spät als möglich stattfindet, bringt aber das Problem mit sich, daß die Serverinstanz nach Abarbeitung des Auftrages stets solange Leerlauf hat, bis die Lastbalancierung mit Sendung eines weiteren Auftrages reagiert. Das ist nur erträglich, solange auf jedem Prozessor ständig mehrere Aufträge parallel bearbeitet werden, da diese in der Wartezeit der einen Instanz die Knotenleistung nutzen können. In jedem Falle müßte der Lastbalancierer die Leerlaufzeiten berücksichtigen.

Die Warteschlangen an den einzelnen Instanzen bieten weiterhin die Möglichkeit, Aufträge gruppenweise zu behandeln. Der Lastbalancierer kann jeweils einen Puffer voll von Aufträgen am Stück an eine Serverinstanz senden. Diese Optimierung wird oft bei hoher Parallelität und feinem Auftragsgranulat verwendet (siehe etwa [Schiele91]), wenn die Nachrichtenkosten sowie der Verwaltungsaufwand pro Auftrag durch den zentralen Scheduling-Prozeß relativ hoch sind. Es ist offensichtlich eine Vergröberung des von der Anwendung vorgegebenen Granulats. Da im HiCon Konzept das Granulat vollständig dem Anwender überlassen bleibt, wird diese Optimierung derzeit nicht angewandt.

Zentrale Klassen-Auftragswarteschlangen ermöglichen es dem Lastbalancierer, die Parallelität zu kontrollieren: Rekursive Anwendungen erzeugen oft eine exponentiell wachsende Anzahl parallel ablauffähiger Aufträge. Die Bereitstellung bzw. dynamische Erzeugung einer entsprechenden Anzahl von Serverinstanzen wäre ineffizient und unrealistisch (großer Prozeßwechselaufwand, Hauptspeicherüberlastung/Swapping, Nachrichtenfluten, Überlastung des Lastbalancierers). Die sofortige Zuweisung entstehender Aufträge an Instanzen führt zu langen lokalen Warteschlangen, was dem Balancierungsprinzip der möglichst späten Zuweisung widerspricht. Bis nämlich der Auftrag zur Bearbeitung an die Reihe kommt, ist die Instanz evtl. nicht mehr optimal für ihn.

Der Lastbalancierungsmechanismus benutzt eine Tabelle (Systemfahrplan), um die Systembelastung durch laufende und geplante Aufträge zu verwalten. Das Hauptproblem einer zentralen Datenbasis besteht darin, die unterschiedlichen, aber nicht unabhängigen Objekte wie Aufträge, Serverinstanzen, Datenbestände und Ressourcen geeignet darzustellen sowie die verschiedenen Zustandsübergänge wie Zeitintervalle, Auftragsbearbeitungsdurchlauf und logische Reihenfolgeabhängigkeiten möglichst auf eine Achse abzubilden.

Im HiCon Modell basiert der Systemfahrplan primär auf den Dimensionen Server, Ressourcen und Aufträgen (siehe Bild).



Für jede Serverinstanz gibt es eine Liste von anstehenden Aufträgen, versehen mit einer Beschreibung ihrer verursachten Last. Dabei ist zu unterscheiden, ob der Auftrag bereits an eine Serverinstanz abgeschickt wurde, d.h. in deren Warteschlange liegt, ob er noch in der zentralen Warteschlange steht oder ob er lediglich eingeplant ist. Im dritten Fall kann es passieren, daß dieser Auftrag nie Wirklichkeit wird, oder aber daß er mit anderen Lastparametern entsteht.

Man beachte, daß sich die Serverinstanzen in unserem Modell nicht notwendigerweise beim Lastbalancierer zurückmelden, wenn sie einen Auftrag abgeschlossen haben (Resultatsendungen werden vom Lastbalancierer identifiziert, genaueres siehe Kapitel 3.2.1). Vielmehr beginnen sie unmittelbar mit der Bearbeitung des nächsten Auftrages in ihrer Warteschlange. Dieser Ablauf ist notwendig, da ein zentraler Lastbalancierer ansonsten eine große Leerlaufzeitspanne verursachen würde. Serverinstanzen können jedoch dem Balancierer am Ende einer Bearbeitung eine freiwillige Zustandsmitteilung senden.

Genaue Vorab-Informationen sind nur für größere Aufträge sinnvoll. Zukünftig eventuell auftretende Aufträge (im Rahmen der Vorplanung einer Auftragsgruppe) werden überhaupt nur eingetragen, falls sie eine Mindestgröße und eine Mindestsicherheit an geschätztem Bedarf aufweisen oder eine Reihenfolge-Vorbedingung für andere, eingetragene Aufträge sind.

Für die Bewertungsfunktion (siehe unten) der Lastbalancierungsstrategie ist es sinnvoll, die durch laufende Aufträge hervorgerufene Auslastung pro Ressource aufzuaddieren. Das erlaubt weiterhin eine Korrektur (Neuskalierung) der Lastangaben von Aufträge durch Messungen, die in unserem Modell auf Ressourcenebene durchgeführt werden. Die Lastmessung einzelner Prozesse würde zu einer erheblichen Grundlast auf dem System führen.

Im HiCon Lastbalancierungscerfahren unterscheiden wir drei Vorgänge (Balancierungsebenen), die Einplanung von Auftragsgruppen, die Zuweisung einzelner Aufträge und die Verteilung von Datensätzen.

2.3.1 Die Ebene der Auftragsplazierung

Jeder Aufruf, den eine Serverinstanz an eine Serverklasse durchführt, wird vom Lastbalancierungssystem einer bestimmten Instanz zugewiesen. In der hier vorgestellten Version wird jede Nachricht über die zentrale Balancierungskomponente abgewickelt. Das ist offensichtlich für große Mengen kleiner Aufträge nicht sinnvoll, da dort die zentrale Komponente überlastet wird und jede Nachricht allein durch die Zwischenstation die doppelte Laufzeit benötigt. Der Lastbalancierer könnte in solchen Fällen im Rahmen der statischen Vorplanung (siehe unten) feinkörnige Aufträge zur dezentralen Balancierung freigeben. Daraufhin vergibt der Aufrufende die Aufträge nach einem einfachen Verteilungsverfahren direkt an die Instanzen der Klasse. Man beachte, daß dazu Verbindungen zwischen allen Instanzen der beteiligten Klassen bestehen müssen. Für große Systeme wird der Balancierer hierarchisch verteilt (Kapitel 2.4).

Der zentrale Balancierer reiht den ankommenden Auftrag zunächst in die Warteschlange der Zielklasse ein. Dort befinden sich alle Aufträge für diese Serverklasse, bis sie fest einer Instanz zugewiesen und dorthin abgesandt werden. Der Zeitpunkt der Zuweisung ist beliebig (wie oben besprochen), er kann durch die Ankunft eines neuen Auftrages, durch die Zustandsänderung einer Serverinstanz oder durch einen *Timeout* angestoßen werden.

Jeder Auftrag in der Warteschlange bekommt (vorläufig bei seiner Ankunft) eine Bewertung, welche Instanz zu seiner Bearbeitung in welchem Maße geeignet ist. Diese Bewertungsfunktion wird unten genauer erläutert. Der Balancierungsmechanismus versucht, die Klassen-Auftragsschlange beginnend beim ältesten Auftrag abzuarbeiten. Er weist den Auftrag einer Serverinstanz zu, sobald er sicher ist, daß sie aus derzeitiger Sicht den Auftrag am ‘besten’ bearbeitet. Dabei ist im Prinzip die Antwortzeit unter der momentan verfügbaren Rechenleistung entscheidend. Die genaue Bewertungsfunktion wird unten erläutert. ‘Sicher sein’ bedeutet, ausreichend genaue Angaben zu haben, daß die Instanz innerhalb eines Toleranzbereiches die bestgeeignete ist.

Das Verfahren verlangt einen Kompromiß zwischen möglichst später Zuweisung und möglichst geringer Verzögerung zwischen Aufruf und Beginn der Bearbeitung. Die verfrühte Bindung eines Auftrages an eine Instanz kann dazu führen, daß diese Instanz, bis sie tatsächlich mit der Abarbeitung dieses Auftrages beginnt, nicht mehr optimal ist. Verzögert man die Zuweisung lange (im Extremfall verbleibt der Auftrag in der zentralen Warteschlange bis die bevorzugte Instanz frei ist), so haben die Server zwischen zwei Bearbeitungen viel Leerlauf. Es verlangt außerdem sehr zuverlässiges Wissen über Auftrag und Instanzenzustand, will man freie Instanzen verschmähen, um weiter auf die optimale zu warten.

Die Fähigkeit zur Migration laufender Aufträge erlaubt es, Aufträge mit relativ geringer Vorplanung zuzuweisen und Fehlplanungen später durch Migration zu korrigieren. Das würde obiges Problem entschärfen; Auftragsmigration wird jedoch im HiCon Modell nicht unterstützt, da sie in heterogenen Systemen sehr aufwendig ist und durch die Kontextverwaltung innerhalb von Serverklassen auf einer höheren Ebene vollzogen werden kann (feineres Auftragsgranulat mit kontextsensitiven Server-Aufrufen).

Die Bewertungsfunktion bestimmt für einen gegebenen Auftrag und eine zur Bearbeitung in Frage kommende Serverinstanz, wie geeignet die Instanz sein wird, den Auftrag abzuwickeln. Als Kriterium gilt hier zweifellos die minimale Antwortzeit, man muß jedoch unterscheiden, ob die Lastbalancierung den Auftrag isoliert betrachtet oder die Gesamtheit der unter ihrer Regie laufenden Aufträge berücksichtigt (soziale Balancierung). Im ersten Fall liefert die

Bewertungsfunktion die vermutete Antwortzeit für diese Instanz zurück, im zweiten Fall eine Linearkombination der Antwortzeit auf dieser Instanz und der dabei entstehenden Verzögerungen, die anderen Aufträgen durch Benutzung derselben Ressourcen entstehen (wir sprechen hier noch nicht von Reihenfolgeabhängigkeiten). Die Summanden können nach der Priorität der Aufträge gewichtet werden.

Im HiCon Modell betrachten wir soziale Lastbalancierungsansätze, denn nur diese garantieren, daß bereits zugewiesene Aufträge weiterhin einen gewissen Ressourcenanteil erhalten und nicht zugunsten neu ankommender Aufträge beliebig 'ausgebremst' werden. Der Systemfahrplan ist dahingehend ausgelegt, die Bewertungsfunktion mit geringem Aufwand durchführen zu können: wir addieren die vermutete neue Last zur jeweiligen Ressourcenlast. Bei Überlastung (Lasteintrag übersteigt 100%) werden die betroffenen Aufträge entsprechend abgebremst und dieser Zeitverzug in der Bewertungsfunktion berücksichtigt. Selbstverständlich modifiziert die Bewertungsfunktion den Systemfahrplan nicht wirklich.

Der Aufwand zur Bewertung eines Auftrages steigt linear mit der Instanzenzahl der Serverklasse und bei hoher Auslastung im Extremfall linear mit der Anzahl der in Arbeit befindlichen Aufträge, wächst aber nicht exponentiell mit den Skalierungsfaktoren paralleler Systeme (Ressourcen und Aufträge). Die Abschätzung gilt für die Einplanung einzelner Aufträge; die Behandlung von Auftragsgruppen wird unten detailliert untersucht.

Der Zuweisungszeitpunkt ist im Prinzip unabhängig von der Ankunft eines neuen Auftrages oder der Freimeldung einer Serverinstanz. Dennoch geben diese Ereignisse Anlaß, die Auftragswarteschlange der Klasse daraufhin zu überprüfen, ob nun Aufträge fest an eine Instanz gebunden werden können. Unter der Voraussetzung, daß sowohl der geschätzte Ressourcenbedarf eines Auftrages als auch der Zustand einer Serverinstanz mit Ungenauigkeiten behaftet sind, wird der Auftrag zugewiesen, sobald der Balancierer eine Instanz findet, deren restliche Beschäftigungsdauer ein gewisses Maximum nicht überschreitet und im Rahmen der Ungenauigkeiten die für den Auftrag Bestgeeignete sein kann. Durch diese Strategie nutzen wir die Vorteile der späten Zuweisung (eine zentrale Warteschlange wird oft als die ideale Lastbalancierungstechnik betrachtet) als auch die ununterbrochene Auslastung des Systems durch nichtleere lokale Warteschlangen bei den Instanzen.

Der ideale Zuweisungszeitpunkt muß also nicht mit der Ankunft weiterer Aufträge oder der expliziten Zustandsänderungsmeldung einer Instanz zusammenfallen. Der Balancierer hat im Systemfahrplan Abschätzungen über den Verlauf der Bearbeitung, sodaß er den Zeitpunkt vorbestimmen kann, an dem ein Auftrag an eine bestimmte Instanz zu vergeben ist (sofern keine abweichenden Informationen aus dem laufenden Betrieb gemeldet werden). Das läßt sich beispielsweise durch *Timeouts* realisieren. Im HiCon System beschränken wir uns vorerst auf ein einfaches Verfahren: bei Ankunft eines Auftrags, Zustandsänderung einer Instanz oder wenn für eine Serverklasse eine Zeit lang keine Änderungen eintraten, wird grundsätzlich eine Auftragszuweisung der vorrätigen Aufträge versucht.

Bei Sender-initiierten Lastbalancierungstechniken versucht der Empfänger eines Auftrags bei Überlastung einen Teil seiner Bürde an minderbelastete Instanzen abzugeben, in etwas allgemeineren Verfahren gibt er Aufträge an Nachbarn ab, sobald er feststellt, daß sie weniger belastet ist als er. Bei Empfänger-initiierten Strategien übernimmt ein Server, wenn er einen Auftrag erledigt hat und andere höher belastet sind, Aufträge von Nachbarn. Man kann diese Verfahren leider nicht unmittelbar mit dem des HiCon Modells verglichen, da sie dezentral sind.

2.3.2 Die Ebene der Datenverteilung

Obwohl der vorgestellte Balancierungsalgorithmus nur die Verteilung der Funktionsausführungen direkt beeinflußt, berücksichtigt und steuert er indirekt den Ort und die Verteilung der benötigten Kontextdaten. Der Lastbalancierung bleiben auf der Datenebene zwei Freiheiten: Wenn eine Instanz eine Daten-Partition lesen möchte, so kann sie eine Kopie erhalten (das ist der Normalfall) oder aber das Original. Das ist vorteilhaft, wenn die Wahrscheinlichkeit für einen nachfolgenden Schreibzugriff hoch ist. Hat eine Instanz auf einer Kopie gearbeitet und kaum Aussichten, sie in naher Zukunft noch einmal zu verwenden, so kann sie die Kopie sofort (freiwillig) invalidieren. Beide Optimierungen mindern den Aufwand der Datenverwaltung, falls die Vorabschätzungen des Lastbalancierers zutreffen. Das Konzept der Datenkapselung durch Serverklassen erlaubt die einfache Integration beider Aspekte (funktionale Parallelität und Datenparallelität) im HiCon Lastbalancierungsverfahren.

Zuletzt soll noch die Möglichkeit der ungleichen Lastaufteilung innerhalb einer Ressource erwähnt werden, den wir im derzeitigen Modell noch nicht explizit nutzen. Serverinstanzen können, je nach Wichtigkeit ihres aktuellen Auftrages (siehe unten) mit unterschiedlichen Prozeßprioritäten ablaufen. Um einen eiligen Auftrag durchzusetzen, auf den etwa viele andere warten, kann es sinnvoll sein, andere laufende Aufträge zu bremsen bzw. warten zu lassen. Bislang teilen sich im HiCon Konzept alle laufenden Aufträge die gemeinsamen Ressourcen gleichmäßig untereinander auf. Der Lastbalancierer kann jedoch wichtigere Aufträge vorrangig bzw. an ‘schnellere’ Instanzen vergeben, oder er kann Prozessoren, welche wichtige Aufträge abwickeln, von weiteren Aufträgen verschonen.

2.3.3 Vorplanung von Auftragsgruppen - die Konfigurationsebene

Neben der oben vorgestellten Aufgabe der Lastbalancierung bietet das HiCon System die Möglichkeit, zur Laufzeit zusammenhängende Gruppen von Aufträgen (Anwendungen) anzumelden. Durch Reservierung und Bereitstellung von Ressourcen und einer vorläufigen Zuweisung der einzelnen Aufträge unter Ausnutzung des Wissens über den Gesamtablauf der Gruppe können wir wichtige Aspekte wie wiederholte bzw. parallele Aufrufe einer Serverklasse, Lokalität in aufeinanderfolgenden Datenzugriffen sowie Reihenfolgeabhängigkeiten in die Balancierung einbeziehen, was bei der isolierten Betrachtung einzelner Aufrufe (siehe oben) noch nicht möglich ist.

Die Probleme dieses Ansatzes liegen zum einen bei der Beschreibung der Auftragsgruppe, die möglichst kompakt und einfach, aber dennoch komplexen Anwendungen verschiedenen Granulats gerecht werden soll, zum anderen in den hohen Laufzeitkosten für den Vorgang der Einplanung an sich. Diese Grenzen sind aus dem Gebiet des statischen Scheduling hinreichend bekannt. Wir beschränken uns auf die Spezifikation weniger Zusammenhänge, die von besonderem Interesse für die Lastbalancierung sind, d.h. streben keine Programmiersprachen-ähnliche filigrane Ablaufbeschreibung (wie etwa für parallelisierende Compiler) an. Daher betrachten wir zunächst, welche Charakteristiken einer Anwendung, die über einzelne, isolierte Auftragsprofile hinausgehen, für eine Vorausplanung wesentlich sein können:

- Die in Petrinetz- und Datenflußbeschreibungen bekannten Reihenfolge-Abhängigkeitsgraphen erscheinen im prozeduralen Modell als Wartezeiten auf synchrone Aufrufe. Interessant ist zu wissen, in welchem Maße die Ausführungszeit des (Haupt-) Auftrages von der Bearbeitungszeit eines Unteraufrufes bzw. von der Geschwindigkeit eines Kooperationspartners abhängt. Daraus kann der Balancierer die notwendigen relativen Verarbeitungszeiten (und

damit Prioritäten, sobald der Auftrag zugewiesen wurde) abschätzen. In der statischen Lastbalancierung ist der *Bottleneck-Path*-Algorithmus ein bekanntes Verfahren (siehe [Bokhari81], [Iqbal86] und [Towsley86]). Diese Reihenfolgebeziehungen können freilich nur für kleinere Gruppen diskreter Aufträge genutzt werden, nicht für gesamte komplexe Anwendungen, die zumeist Schleifen, Rekursionen und Verzweigungen enthalten.

- Für größere Mengen anstehender Aufträge kann Lastbalancierung die Anzahl von und das Verhältnis zwischen Lese- und Änderungszugriffen auf Kontext (-Partitionen) ausnutzen, um einen Kompromiß zwischen Parallelität und Kontextverwaltungsaufwand zu finden. Genauere Datenzugriffs-Angaben können jedoch meist erst zum Aufrufzeitpunkt gemacht werden.
- Wenn für eine Serverklasse innerhalb der Auftragsgruppe der vermutlich entstehende Grad an möglicher Parallelität abgeschätzt werden kann, so hat der Balancierer die Möglichkeit, die Server-Konfiguration diesen Anforderungen anzupassen. Mit möglicher Parallelität ist damit die Anzahl der Aufträge einer Klasse gemeint, die tatsächlich zugleich ausführungsbereit sind.
- Die Angabe des Nachrichtenaufkommens (Dauer und Intensität) zwischen einzelnen Teilaufträgen bzw. zwischen zwei Serverklassen schlechthin erlaubt die Berücksichtigung von schnellen Kommunikationsverbindungen bei der Instanzenauswahl bzw. bei der Server-Konfiguration (siehe [Ma82], [Berger87], [Bowen88], [Lo88] und [Lo88/2]).

Um die Beschreibung derartiger Informationen über Auftragsgruppen einfach zu halten, verzichten wir auf hierarchisch verschachtelte Auftragsgruppen sowie auf die explizite Behandlung von Schleifen. Eine Auftragsgruppe ist eine flache Gruppe von Auftragstypen, zu denen die Anzahl gegenseitiger Aktivierungen (Aufrufe) samt Kommunikationsintensität und Reihenfolgebeziehungen angegeben ist.

Diese Angaben bewirken lediglich eine vorläufige Einplanung der zu erwartenden Aufträge sowie Änderungen der Server-Konfiguration. Die endgültige Zuweisung der Aufträge geschieht nach wie vor erst zum tatsächlichen Aufrufzeitpunkt. Wir folgen damit dem in der dynamischen Balancierung wichtigen Prinzip der möglichst späten Zuordnung und können außerdem Fehler und Ungenauigkeiten in den Profilangaben tolerieren.

Das Optimierungskriterium des hier vorgestellten Balancierungsmechanismus ist die möglichst schnelle Ausführung eines Gesamtauftrages (unter Rücksicht auf andere unabhängige Aufträge, ‘soziale Balancierung’, siehe oben). Daher ist nur die Antwortzeit des Hauptauftrags zu minimieren; alle Teilaufträge sind unkritisch und ihre Dringlichkeit ergibt sich allein aus den Abhängigkeitsbeziehungen zum Hauptauftrag.

Auf der Balancierungsebene der Server-Konfiguration spielen nicht nur Profile von Auftragsgruppen eine Rolle. Konfigurationsänderungen können auch durch längerfristige Beobachtungen aggregierter Daten ausgelöst werden. Wir wollen es hier nur anhand zweier Beispiele motivieren:

- Wenn die Klassen-Auftragswarteschlange ständig sehr lang ist, sollten evtl. neue Instanzen aufgesetzt werden. Umgekehrt können Instanzen gelöscht werden, wenn die mittlere Wartezeit auf einen neuen Auftrag groß ist.

- Wenn die Wartezeit auf Daten im Verhältnis zur eigentlichen Rechenzeit der Instanzen zu groß ist, so scheint das Anlegen von Lesekopien aufgrund des hohen Anteils an Schreibzugriffen unwirtschaftlich (siehe [Weinmann92]) oder die Balancierung achtet bei der Auftragszuweisung zu wenig auf die Datenlokalität. Im Zweifelsfall empfiehlt sich eine Reduktion der Serverinstanzen.

2.4 Hierarchisch verteilte Lastverwaltung

In völlig dezentralen Lastbalancierungsverfahren sprechen die Server die Auftragsverteilung selbstständig untereinander ab. Der Server, welcher neue Unteraufträge hat oder überlastet ist, versucht Aufträge an andere Server abzugeben. Unterbelastete Server sind bestrebt, von anderen Aufträge zu übernehmen. Man unterscheidet oft (siehe [Casavant88], [Lin92]), ob nur diese Entscheidung dezentral ist oder auch die Zustandsinformationen über die Server verteilt sind (ein Server kennt nur die Lastsituation seiner Nachbarn). Im HiCon Modell wird sowohl die Zustandsinformation als auch die Entscheidung in einer Komponente zentralisiert.

Eine zentrale Informations- und Entscheidungsstelle bietet den Vorteil einer global optimalen Balancierung. Dezentrale Verfahren optimieren nur lokal und basieren auf relativ einfachen Meß- und Entscheidungsgrößen; die einzelnen Instanzen sind nicht in der Lage, komplexere Zusammenhänge zu erfassen (etwa Reihenfolgebeziehungen, soziale Lastverteilung). Die Verwaltung zentraler Auftragsschlangen ermöglicht späte Zuweisung ohne zugeteilte oder laufende Aufträge migrieren zu müssen.

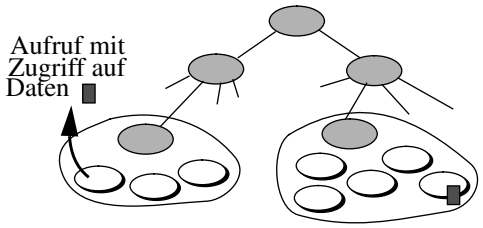
Zentrale Balancierung hat aber den Nachteil, selbst zum Engpaß zu werden. Die Balancierungskomponente muß sehr schnell viele Informations- und Auftragsnachrichten bearbeiten; wenn sie überfordert ist, arbeitet sie mit veralteten Informationen bzw. zögert die Auftragsbearbeitungen hinaus, da sie nicht schnell genug Zuweisungsentscheidungen treffen kann. Auch aus Gründen der Fehlertoleranz ist eine zentralisierte Lösung bedenklich.

In großen Systemen ist also, sofern man einen logisch zentralen Ansatz favorisiert, eine hierarchische Abstraktion der Lastbalancierung notwendig. Lokale Komponenten balancieren autonom ein (sinnvollerweise physisch zusammenhängendes) Teilsystem. Sie betrachten den übergeordneten Balancierer als eine weitere Komponente ihres Teilsystems, der eine abstrakte Sicht seines Zuständigkeitsbereichs zeigt. Eine Balancierungskomponente kann also von ihrer übergeordneten wie von einer Unterkomponente Aufträge und Lastinformationen erhalten und an diese abgeben.

Die Abstraktion erfolgt auf Ebene der Instanzen. Ein Balancierer bietet Anderen eine Instanz pro Serverklasse, deren Leistung dem Mittel- oder Bestwert seiner lokalen Instanzen entspricht. Das Teilsystem (Hardware) und die lokalen Aufträge verbirgt er vollständig.

Hierarchische Balancierung funktioniert nur dann effektiv, wenn eine gewisse Lokalität vorliegt und von der Balancierung berücksichtigt wird. Lastbalancierer verteilen Aufträge (und damit auch Daten) möglichst lokal. Die ‘Pseudo’-Instanz des übergeordneten Balancierers muß daher um einiges besser sein, damit sie gewählt wird. Wenn aufgrund der Anwendung keinerlei Lokalität in den Datenzugriffen vorhanden ist und sich auch nicht durch Datenmigration oder Replikation einstellt, so ist die hierarchische Balancierung ungünstig (das Bild zeigt solch einen Fall).

Allgemein lassen sich Änderungsoperationen auf einem gemeinsamen Datensatz nicht beliebig verteilen und die Lastbalancierung sollte dies auf höherer Ebene feststellen.



Alternativ zur oben vorgestellten homogenen Hierarchie bietet es sich an, die unteren Ebenen dezentral zu balancieren und für die höheren Ebenen zentrale Balancierungskomponenten einzusetzen. Beispielsweise könnten die Instanzen die Datenverwaltung direkt untereinander, ohne den Umweg über eine Balancierungsinstanz, regeln. Der für Auftragszuweisung verantwortliche Balancierer hat dann aber kein vollständiges Wissen mehr über die Datenverteilung und kann sie entsprechend schlecht einbeziehen. Zudem legt man sich durch die Trennung nach Ebenen auf eine bestimmte Systemgröße fest, während die homogene Hierarchie - eine gewisse Lokalität in der Anwendung vorausgesetzt - beliebig skalierbar bleibt.

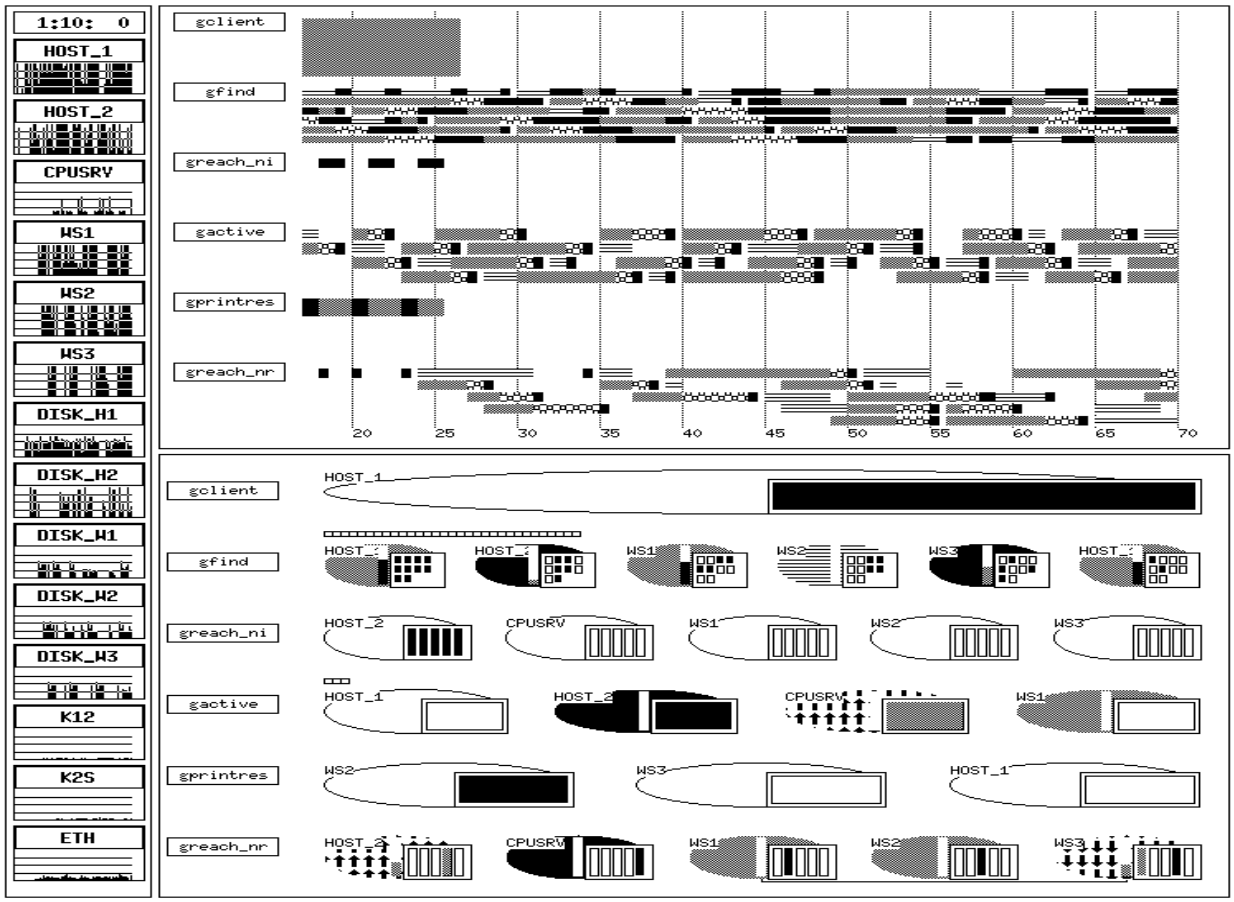
3 Realisierung und Ergebnisse

Eine Lastbalancierungsumgebung nach dem HiCon Modell wurde als Prototyp realisert; daneben wurde ein Simulator entwickelt, der uns eine schnellere und einfachere Evaluierung der vorgeschlagenen Konzepte ermöglicht. Wir werden diese Implementierungen im folgenden kurz betrachten.

3.1 Simulation des HiCon Modells

Um die prinzipielle Tauglichkeit der entwickelten Lastbalancierungsverfahren zu untersuchen und ein Gefühl für die verschiedenen Faktoren, ihr Zusammenspiel und ihre Auswirkungen zu bekommen, wurde ein Simulator zur Nachbildung des HiCon-Ausführungsmodells entwickelt (siehe [Staib92]). Nach Definition eines Systems, einer Server-Konfiguration und einer Anwendung (Auftragsprofile für die Serverinstanzen) sowie der Auswahl von Lastbalancierungsverfahren für zwei Eingriffspunkte (die Entstehung eines Auftrags und die Zustandsänderung einer Serverinstanz) kann man die Bearbeitung mitverfolgen und erhält eine abschließende Gesamtauswertung. Während der Simulation wird der Verlauf der Anwendung (Abarbeitung der Funktionsausführung, Unteraufrufe und Datenzugriffe) sowie die Auslastung aller Ressourcen detailliert graphisch dargestellt (siehe Bild). In einem Anwendungsbeispiel mit relativ feinkörniger Parallelität, der Suche nach einem kürzesten Weg in einem Graphen (siehe dazu auch Kapitel 3.2.5), zeigte [Staib92] bereits positive Resultate für einfache Lastbalancierungstechniken nach dem

HiCon-Konzept. Wir wollen jedoch nicht hier, sondern im Rahmen des *Loadman*-Prototyps näher auf diese Experimente eingehen.



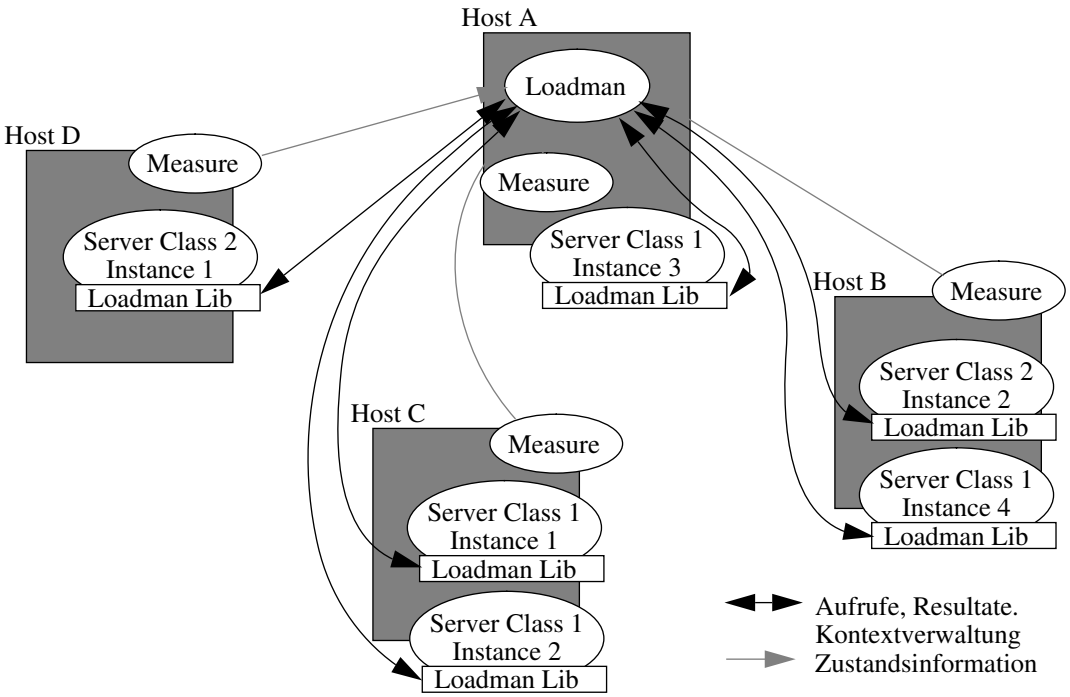
3.2 Loadman - Prototyp eines Lastbalancierers nach dem HiCon Modell

Die Lastbalancierungsumgebung *Loadman* ist eine Realisierung der in diesem Bericht vorgestellten Ansätze, um ein weites Spektrum datengestützter Anwendungen effizient auf heterogenen parallelen und verteilten Systemen zu bearbeiten. Dazu beschreiben wir kurz das Design, die bisher implementierten Balancierungsverfahren und vorläufige Messungen.

3.2.1 Systemstruktur und Anwendungsschnittstelle

Wie im Bild veranschaulicht besteht die Balancierungskomponente (Systemlast-Informationsverwaltung und Balancierungsentscheidung) derzeit aus einem zentralen Prozeß und einem Lastmessungsagenten auf jedem teilnehmenden Prozessor. Sie wird in späteren Versionen nach dem in Kapitel 2.4 beschriebenen Konzept verteilt sein. Die Anwendung ist durch einen Prozeß pro Ser-

verinstanz auf das System verteilt und verkehrt über angebundene Libraries mit der Balancierungskomponente.



Die eigentliche Interprozeß-Kommunikation wird mittels angebundener Funktionen der *coin*-Library durchgeführt. Diese Library wählt automatisch das schnellste verfügbare Nachrichtenprotokoll. So werden Nachrichten innerhalb eines Hauptspeichers (UNIX-Workstation oder shared memory Multiprozessor) über *pipes*, innerhalb von Parallelrechnern durch proprietäre Mechanismen (etwa *Guardian*s Kommunikationssystem) und zwischen verschiedenen Rechnern mithilfe des *TCP* Protokolls versandt. Die transparente Nutzung dieser verschiedenen Nachrichtensysteme ist für die Lastbalancierung in heterogenen Systemen unerläßlich, da unterschiedlich effiziente Kommunikationswege (nicht prinzipielle Beschränkung auf ein allgemeines, langsames Medium wie *TCP*) und damit verschiedene 'Entfernungen' zwischen Prozessen einen wesentlichen Faktor für die Lastverteilung darstellen.

Loadman ermöglicht derzeit verteilte Anwendungen auf einer aus Sun-, DEC-, HP840- UNIX Workstations, Sequent shared-memory-Multiprozessoren sowie Tandem shared-nothing-Parallelrechnern beliebig gemischten Basis. Multiprozessoren mit gemeinsamem Hauptspeicher werden von *Loadman* derzeit als je ein Prozessor betrachtet, da die Lastbalancierung im *HiCon* Modell relativ grobgranular konzipiert ist und auf verteilten Ressourcen basiert, sodaß man hier besser die vorhandenen Scheduling-Strategien (meist gemeinsame Run-Queue) nutzt.

Dynamische Lastmessung erfolgt auf UNIX-Systemen durch Ablesen bestimmter Datenstrukturen aus dem Betriebssystemkern, auf *Guardian*-Rechnern durch das *Measure*-Subsystem. Die statische Beschreibung des Systems (Prozessorleistungen, Platten, Netzstruktur und -Leistung) wird in einer Konfigurationsdatei spezifiziert.

Die *Loadman*-Library stellt den Server-Programmen eine komfortable Schnittstelle zur Verfügung, die asynchrone Serverklassen-Aufrufe sowie Zugriffs- und Konsistenzschutz-Operationen auf Klassenkontexte ermöglicht:

- *LoadmanInit (connectionDataToLoadman, directoryForLogging, timeout)*

Mit diesem Aufruf meldet sich die Serverinstanz bei *Loadman* an. *Loadman* setzt die in einer Datei spezifizierte Server-Konfiguration auf und teilt jeder Instanz beim Prozeßstart ihre Instanzennummer sowie ihre Datei-Zugriffspfade mit.

- *LoadmanCall (class, parameter, expectedCpuInstructions, expectedUsedPartitions, timeout)*

  Die Serverinstanz setzt einen Aufruf an eine Serverklasse ab. Dabei kann sie den vermutlichen Ressourcenbedarf für diesen Aufruf angeben. Die Spezifikation beschränkt sich bislang auf den Rechenaufwand und die benötigten Kontextpartitionen, jeweils versehen mit der Wahrscheinlichkeit für schreibenden Zugriff. Man beachte, daß diese Angaben unabhängig von Prozessor und Instanz sind, die den Auftrag einmal ausführen werden. Die Vor- und Nachteile der Ressourcenabschätzung am Aufrufzeitpunkt durch den Aufrufer wurden in Kapitel 2.1 diskutiert.

- *LoadmanResult (class, instance, parameter, serverState, timeout)*

  Die Serverinstanz schickt ein Resultat an den Aufrufer zurück. Dieser Aufruf kann auch von Client und Server innerhalb einer 'Session' verwendet werden, wo ja die Zielinstanz feststeht. Der Server sendet lediglich Zwischenergebnisse, der Aufrufer gibt daraufhin evtl. weitere Parameter. Durch den Parameter *serverState* kann der Server die Lastbalancierung informieren, welchen Anteil des Gesamtauftrages er mit Sendung des Resultats erledigt hat. Die *Loadman*-Library fügt als weitere Information die momentane Länge der Auftragswarteschlange der Instanz bei. Weiterhin kann eine Instanz bei langlaufenden Aufträgen Zustandsinformationen abgeben, indem sie eine Resultats-Nachricht ohne Zielklasse und -instanz schickt.

- *LoadmanRecv (class, instance, parameter, timeout)*

  Die Serverinstanz wartet auf einen Auftrag oder ein Resultat. Dabei kann sie eine spezielle Klasse und / oder Instanz vorgeben.

- *SlockContext (partition, timeout)*

  Die Serverinstanz meldet Lesezugriffe auf eine Kontext-Partition an. Dabei bekommt sie über die Library evtl. eine Kopie oder das Original dieser Partition von der Instanz gesandt, welche diese Partition momentan besitzt. Kapitel 3.2.2 geht näher auf die Kontextverwaltung ein.

- *XlockContext (partition, timeout)*

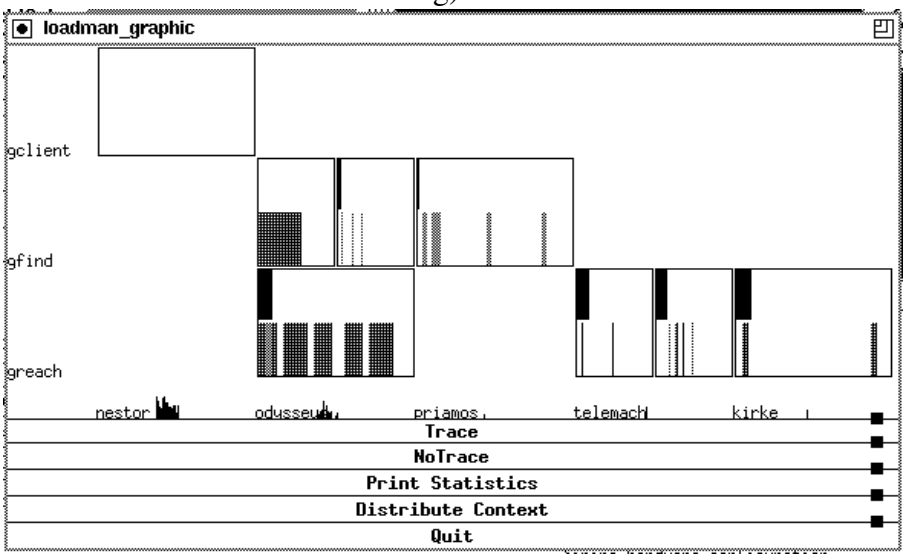
  Die Serverinstanz möchte Änderungsoperationen an dem Kontext-Datensatz durchführen. In der vorläufigen Realisierung wird sie dadurch gleichzeitig zum neuen Besitzer und Verwalter dieser Partition (siehe Kapitel 3.2.2).

- *UnlockContext (partition, timeout)*

  Die Serverinstanz gibt eine zuvor gesperrte Partition nach Durchführung einiger Zugriffe wieder frei.

Loadman kann wahlweise mit oder ohne graphischer Oberfläche (siehe Bild) ablaufen. Die graphische Darstellung informiert zur Laufzeit über den Verarbeitungszustand: die Server-Konfiguration, die Längen der Klassen- und Instanzen-Auftragswarteschlangen, die Verteilung der Kontextpartitionen und deren Replikate sowie die Auslastung der Ressourcen. Diese Darstellung bewährte sich, um Anwendungen bezüglich ihres Kontext- und Auftragsgranulats zu justieren, geeignete Server-Konfigurationen zu ermitteln und die Auswirkungen verschiedener Balancierungsstrategien zu beobachten. Für exakte Zeitmessungen verzichtet man freilich auf das graphische Interface, weil es gewisse Störungen und Verzerrungen verursacht. Während oder nach

Ablauf einer Anwendung kann der Benutzer statistische Informationen über das Verhalten der Serverklassen abrufen (die mittlere Leerlaufzeit der Instanzen einer Klasse zwischen zwei Auftragsbearbeitungen, die mittlere Bearbeitungszeit eines Auftrags ohne Datenverwaltungskosten und der mittlere Zeitbedarf für Datenverwaltung).



3.2.2 Verwaltung und Synchronisation der Klassenkontexte

Im HiCon Modell besitzt jede Klasse einen Kontext, der nach Partitionen aufgetrennt wird. Die Datenstrukturen und Einteilung in Partitionen wird durch die Anwendung bestimmt; ein Kontext kann sich beispielsweise aus Hauptspeichervariablen und Dateien zusammensetzen. Die Konvertierung der Daten von einem bzw. in ein über Nachrichten verschickbares Format geschieht durch von der Anwendung zur Verfügung gestellte Rückrufprozeduren:

SendContext (partition, contextMessage, size), RecvContext (partition, contextMessage, size)

Eine Partition ist zugleich die Einheit der Synchronisation, der Migration und der Replikation. Die Serverinstanzen einer Klasse betrachten den Kontext als gemeinsamen Speicher (man spricht daher von einer *shared memory* bzw. *shared disk* Semantik).

Zugriffen auf eine Kontextpartition muß stets eine Sperranforderung vorangehen. Zugleich mit dem Erwerb der Sperre wird bei Bedarf die Partition bzw. eine Kopie von der Instanz besorgt, welche momentan diese Partition verwaltet (besitzt). Im derzeitigen Prototyp übernimmt eine Instanz mit Anforderung einer Exklusivsperre auch die Verwaltung der Partition, bei Anforderung einer Lesesperre erhält sie lediglich eine Kopie. Ohne Lastbalancierung, welche die Kopien und den Aufwand für diese Kontextverwaltung miteinbezieht, ist diese Entscheidung nicht generell vorteilhaft (siehe etwa die Untersuchungen von [Weinmann92]). Im HiCon Modell kann das Wissen über die Verwaltungsstrategie erfolgreich zur Lastbalancierung genutzt werden.

Um eine Exklusivsperre durchzusetzen schickt die *Loadman*-Library selbstständig Invalidierungsnachrichten an die Kopienbesitzer. Änderungen an dieser Partition durch irgendeine Instanz der Klasse werden erst nach Freigabe dieser Sperre wieder zugelassen. Da der Freigabezeitpunkt durch die Anwendung bestimmt wird, sind verschiedene Stufen der Konsistenzerhaltung möglich (z.B. Operationssperren, Cursor-Stabilität oder serialisierbare Transaktionen).

Dieses Konzept zur Synchronisation des Zugriffs auf globale Daten und der Verteilung globaler Daten in heterogenen Systemen ermöglicht die Balancierung datenintensiver Anwendungen und erfüllt die von der Objektorientierung geforderte Kapselung von Daten durch Serverklassen.

3.2.3 Fehlerbehandlung

In parallelen und verteilten Systemen besteht verstärkter Bedarf an Fehlerbehandlungsmechanismen und Unterstützung bei der Fehlersuche. Gegenüber sequentiellen Programmen erhält man weitere Fehlerquellen durch Rechnerausfall, Prozeßabsturz, Nachrichtenprobleme und Synchronisations- bzw. Protokollfehler. Da wir im HiCon Modell den Schwerpunkt auf Lastbalancierung setzen, bieten wir lediglich zwei Konzepte zur Fehlererkennung an: *Timeouts* und Protokolldateien.

Jeder Aufruf einer *Loadman*-Library-Funktion kann mit einem *Timeout* versehen werden. Dies ist die einfachste und häufig eingesetzte Methode, um Protokollfehler und Ausfälle in der Anwendung zu erkennen. Allerdings sind die Funktionsaufrufe nicht atomar; die *Timeout*-Spezifikation wird beim Senden und Empfang von Nachrichten verwandt und die Funktionsausführung wird im Fehlerfall an dieser Stelle abgebrochen.

Alle am Ablauf teilnehmenden Prozesse schreiben in Protokolldateien. Dabei benutzt eine Serverinstanz ihre Protokolldatei, in die also auch Anwendungsinformation geschrieben werden kann, gemeinsam mit der *Loadman*-Library und der darunterliegenden *coin*-Nachrichten-Library. Nach Ablauf der Anwendung lassen sich Fehler durch Vergleiche der Protokolldateien feststellen. In Fehlerfällen fügen die Library-Funktionen grundsätzlich neben Rückgabe des Fehlercodes eine textuelle Fehlerbeschreibung in die Protokolldatei.

3.2.4 Verfügbare Strategien

Wir haben zunächst nur einfache Strategien, die jeweils auf maximal einer dynamischen Meßgröße basieren, untersucht. Derzeit sind folgende Verfahren auf *Loadman* verfügbar:

- **Round Robin**: in jeder Serverklasse werden die Aufträge reihum verteilt. Dabei sollen die Auftragswarteschlangen der Instanzen möglichst bis zu einer festen Länge gefüllt sein. Grundsätzlich wird der älteste Auftrag in der Klassenwarteschlange zuerst zugewiesen. Wenn in einer Klasse viel mehr Aufträge anfallen als die Serverinstanzen abarbeiten können, d.h. die Klassen-Warteschlange lang ist, wird der älteste Auftrag der Instanz zugeordnet, die als erste wieder Platz in ihrer Warteschlange hat. Das entspricht nicht mehr der reinen Reihum-Verteilung sondern eher der dritten Strategie, ist jedoch sehr lukrativ. Unter der Annahme gleicher Auftragsgrößen und Prozessorleistungen und Mißachtung sonstiger Einflußfaktoren (siehe Kapitel 1.1) realisiert dieses Verfahren Lastbalancierung zugunsten minimaler Auftragsausführungszeit.
- **Daten-Lokalität**: Aufträge werden stets an die Instanz abgegeben, welche die ‘meisten’ der benötigten Daten bereits lokal verfügbar hat. Für Lesezugriffe genügen Kopien, für Änderungsoperationen bedarf es der Originaldaten. Beim Aufruf werden Wahrscheinlichkeiten für Änderungsoperationen angegeben, nach denen das Verfahren die Kontext-Beschaffungskosten gewichtet. Man beachte, daß jeder Auftrag auch dann wartet, bis seine bevorzugte Instanz genügend Raum in ihrer lokalen Warteschlange aufweist, wenn andere Instanzen arbeitslos sind (im Gegensatz zu der im Round Robin Verfahren gewählten Lösung). Die Entscheidung

für eine bestimmte Instanz erfolgt aber nicht bei der Entstehung des Auftrages; vielmehr wird, sobald eine Instanz verfügbar wird, überprüft, ob sie für Aufträge aus der Klassen-Warteschlange die bestgeeignete ist. Diese späte Zuweisung ist notwendig, da sich die Datenlokationen ständig ändern. Grundsätzlich wird der älteste Auftrag, welcher diese Instanz beansprucht, zuerst abgegeben. Wie alle hier betrachteten Verfahren wird hier die Ausführungszeit der Einzelaufträge unter Mißachtung weiterer Faktoren minimiert.

- **Kürzeste Auftrage-Warteschlange**: der älteste Auftrag wird an die Instanz verwiesen, die gerade die wenigsten Aufträge in ihrer Warteschlange hat. Wie bei allen anderen Strategien werden auch hier die Warteschlangen der Instanzen nur bis zu einem festen Maximalwert gefüllt. Die Motivation hierfür sind nicht begrenzte Pufferfähigkeiten der Instanzen sondern die möglichst späte Bindung eines Auftrages an eine feste Instanz sowie die Kontrolle der entstehenden Parallelität (siehe Kapitel 2.3 bzw. 2.3.1).

  Beim Verfahren 'kürzeste Auftrage-Warteschlange' muß man einen Kompromiß zwischen häufiger Zusendung der aktuellen Schlangenlängen und der Abschätzung durch den Lastbalancierer selbst finden. Wenn Instanzen nach oder mehrmals während der Abarbeitung eines Auftrages Zustandsnachrichten an den Balancierer schicken entsteht großer Zusatzaufwand. Wenn der Balancierer hingegen allein aus den Aufruf-Angaben und der Rechnerbelastung (Fremdlast und bekannte Last durch weitere arbeitende Instanzen) den Abarbeitungszustand vermutet, kann es sehr schnell zu Fehleinschätzungen und dadurch zu Fehlentscheidungen kommen. Im derzeitigen Prototyp bleibt die Entscheidung dem Anwender überlassen; Server können zu beliebigen Zeitpunkten Zustandsinformation senden. Bei Rücksendung eines Resultates wird sie stets automatisch beigefügt.

- **Leistungsfähigste Instanz**: die momentane Leistungsfähigkeit eines Prozessors wird abgeschätzt, indem man seine Maximalleistung durch die Zahl der aktiven Prozesse (Run Queue Length) - zuzüglich dem gerade zu vergebenden - teilt. Der älteste Auftrag wird an eine der Instanzen auf diesem Prozessor vergeben, wenn ihre lokale Auftragsschlange nicht zu voll ist.

Offensichtlich wurden bisher nur sehr primitve Methoden implementiert. Der *Loadman* Prototyp soll auf längere Sicht über eine Strategie verfügen, die unter anderem obige Einzelaspekte integriert. Dazu ist es jedoch notwendig, zunächst die Relevanz der verschiedenen Größen und Verfahren isoliert zu untersuchen, um sie geeignet kombinieren und gewichten zu können.

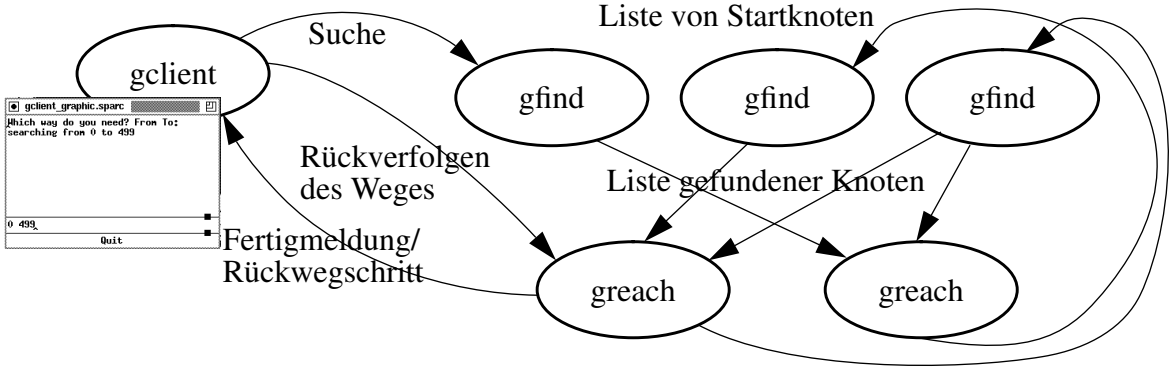
3.2.5 Evaluierung eines Anwendungsbeispiels

Für eine erste Validierung des Lastbalancierungsprinzips durch den Prototyp *Loadman* wurde eine Wegesuche als Anwendung gewählt. Sie findet den kürzesten Weg zwischen zwei vorgegebenen Knoten in einem gerichteten Graphen mit gewichteten Kanten. Da in Client-Server Architekturen durch synchrone rekursive Aufrufe binnen kurzer Zeit alle Instanzen belegt wären (nämlich auf die Ergebnisse ihrer rekursiven Aufrufe warten würden - im HiCon Modell verzichten wir gewöhnlich auf Multi-Threading, obwohl auch 'multi-threaded' Instanzen teilnehmen können), müssen die Aufrufe asynchron und ohne Resultatrückgabe abgesetzt werden. Das hierbei auftretende Terminierungsproblem wird gelöst, indem eine Serverklasse (unter anderem) die Zahl der noch zu untersuchenden Knoten bzw. die Zahl der noch zu bearbeitenden Suchaufträge verwaltet. Der *Supervisor* [Schiele91] stellt dazu im zentralen Fall das Konzept des Klassen-Events zur Verfügung.

Die Wegesuche wurde in Gestalt dreier Serverklassen realisiert (siehe auch im Bild):

- *gclient*: führt die Benutzerinteraktion durch, startet die Suche und verfolgt an deren Ende den kürzesten Weg zurück.
- *gfind*: bekommt im Aufruf jeweils eine Menge von Startknoten (aus derselben Partition). Sie sucht im Graph alle zugehörigen Folgeknoten samt Kosten heraus, eliminiert bei Duplikaten den teureren Weg. Sie teilt die Menge der gefundenen Zielknoten nach Partitionen auf und verschickt jeweils die Knoten einer Partition als Auftrag an die Klasse *greach*. Die Graphenbeschreibung ist in Form je einer Datei pro Partition abgelegt.
- *greach*: verwaltet die Liste der bisher erreichten Knoten. Sie bekommt im Aufruf jeweils eine Liste neu erreichter Knoten und trägt die Neuigkeiten bzw. Verbesserungen in ihrer Liste ein. Die anderen Knoten eliminiert sie aus der Aufrufliste. Dann schickt sie die Restliste als Aufruf an die Klasse *gfind*. Die Klasse verwaltet dabei auch die Zahl der noch zu untersuchenden Knoten (die Zählung ist nicht trivial, da sie nicht weiß, wieviel Folgeaufträge eine *gfind*-Instanz aus einem Aufruf von einer *greach*-Instanz generieren wird). Der Datensatz zur Terminierungszählung stellt sich als 'Hot Spot'-Kontextpartition heraus, da er bei jedem Aufruf modifiziert wird.

  Wenn man die Partitionierungsfunktion der Knoten des Graphen für die Serverklassen unterschiedlich wählt (d.h. der Knotenbereich einer *gfind*-Partition entspricht nicht dem einer *greach*-Partition), so muß *greach* im allgemeinen einige Partitionen pro Aufruf anfassen (und dabei häufig auch aktualisieren). Man sieht anhand der Wartezeiten zwischen Aufträgen und der Kontextverwaltungskosten, daß *greach* deshalb bei (sowohl bezüglich Auftragsgröße als auch der Größe einer Partition) feinerem Granulat zum Engpaß der Anwendung wird. Bei größeren Graphen werden die *gfind*-Instanzen zum Engpaß, d.h. zum bestimmenden Faktor.



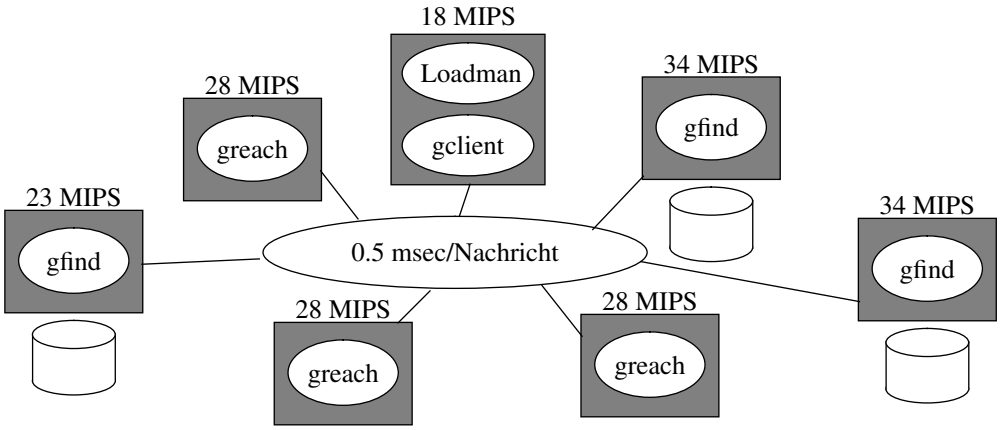
Die Ressourcenbedürfnisse (hier nur Prozessorleistung) der Serverklassen-Aufrufe wurden statistisch ermittelt; sie hängen in erster Näherung linear von der Anzahl der Eingabeknoten und der Größe des Graphen ab. Die 'Zahl der Instruktionen' ist dabei architekturunabhängig und nur auf die in Benchmarks übliche Leistungskenngröße MIPS (Millionen Instruktionen pro Sekunde) bezogen. Die Plattenzugriffe und Kommunikationskosten wurden außer Acht gelassen:

gfind: #Instruktionen = 182755 + #Kanten * 26 + #Startknoten * 30000

greach: #Instruktionen = 6808 + #Kanten * 2 + #Erreichte_Knoten * 16200

Die Aufrufe an die Klassen *greach* und *gfind* beziehen sich stets nur auf eine *gfind*-Partition. Diese Entscheidung wurde getroffen, weil *gfind*'s Partitionen große Dateien sind und neben dem Zugriff auch die Replikation und Migration sehr aufwendig machen. Da es sich jedoch ausschließlich um Lesezugriffe handelt, verschwindet der Verwaltungsaufwand im Verlaufe der Anwendung (die Instanzen sind mit Kopien versorgt).

Im folgenden betrachten wir Messungen der Wegesuche in einem kleinen und in einem relativ großen Graphen, wobei sowohl die Anzahl der Instanzen je Serverklasse als auch die Balancierungsstrategien variiert werden. Der kleine Graph besteht aus 500 Knoten und 10000 Kanten. Letztere sind auf 10 Kontext-Partitionen der Klasse *gfind* und damit auf Dateien der mittleren Größe von 6 KB verteilt. Der Graph weist eine Lokalität von 95% auf (Lokalität bezeichnet hier den Anteil der Kanten, die zwei Knoten derselben Partition verbinden). Das Bild zeigt die maximal verwendete Server-Konfiguration, die Ergebnisse sind in der nachfolgenden Tabelle aufgeführt.



Instanzen je Klasse	statische Balancierung	dynamische Balancierung	Laufzeit [sec]	Instanzenverhalten [msec]			Engpaß
				busy	*context*	*idle*	
1	gleichgültig	gleichgültig (Round Robin)	15.2	61	0	0	gfind.0
				2	0	16	
2	Data Distribution	Data Locality	9.9	67	0	6	loadman
				* 2	3	6	
		Run Queue Length	13.5	66	8	56	gfind.0 / . gfind.1
				* 2	2	23	
		Instance Queue Length	45.9	67	6	368	greach.0-1
				2	95	3	
		Round Robin	48.0	62	7	277	greach.0-1
				2	93	4	
	None	Round Robin	49.9	84	# 12	284	greach.0-1

die Kontextwartezeiten entstanden nur bei Instanz 1
+ Wartezeiten entstanden nur bei Instanz 0
* nur Instanz 0 wurde genutzt

Das Instanzenverhalten schlüsselt die Zeitanteile einer Auftragsbearbeitung einer Serverinstanz auf. Sie sind jeweils über die Instanzen einer Klasse und über deren bearbeitete Aufträge gemittelt. Die Angabe „*busy / context / idle*“ gibt die Zeit, welche die Instanz für den Auftrag gerechnet hat (ohne Kontextwartezeiten), die Zeit, die sie auf Kontextpartitionen gewartet hat und die Zeit, welche sie nach Ende eines Auftrags bis zum Erhalt des nächsten warten mußte, wieder. Die obere Zeile bezieht sich jeweils auf die Klasse *gfind*, die untere auf *greach*. Die Zeilen der Tabelle sind jeweils gemittelte Werte aus mehreren gleichartigen Messreihen.

Die zweite Tabelle enthält die Ergebnisse bei Durchsuchung eines relativ großen Graphen, bestehend aus 100000 Kanten, die auf 30 Partitionen zu je 20 KB verteilt sind. Die übrigen Eigenschaften entsprechen denen des kleinen Graphen.

Instanzen je Klasse	statische Balancierung	dynamische Balancierung	Laufzeit [sec]	Instanzenverhalten [msec]			Engpaß
				busy	*context*	*idle*	
1	gleichgültig	gleichgültig (Round Robin)	127.5	156	0	0	gfind.0
				4	0	40	
2	Data Distribution	Run Queue Length	105.7	213	22	44	gfind.0
				4	61	30	
		Data Locality	109.8	212	0	41	gfind.1
				* 4	2	31	
	None	Round Robin	124.1	209	# 18	92	greach.0-1
				4	86	2	
	Data Distribution	Instance Queue Length	127.3	213	23	107	greach.0-1
				4	86	2	
		Round Robin	131.4	215	21	79	greach.0-1
				4	85	2	
3	Data Distribution	Data Locality	98.7	230	0	+ 81	gfind.1-2
				* 4	2	23	
		Run Queue Length	132.1	227	49	231	greach.0
				5	84	32	
		Instance Queue Length	155.0	266	47	316	greach.0-2
				4	156	2	
	None	Round Robin	157.1	227	38	284	greach.0-2
				4	157	3	
	Data Distribution	Round Robin	164.0	261	41	232	greach.0-2
				4	157	3	

die Kontextwartezeiten entstanden nur bei Instanz 1
+ Wartezeiten entstanden nur bei Instanz 0
* nur Instanz 0 wurde genutzt

Je nach Strategie und Granulat wird die Klasse *gfind* bwz. die Klasse *greach* zum kritischen Faktor, d.h. Engpaß (eine eher ausgewogene Belastung deutet auf relativ starke Auslastung des Lastbalancierers *Loadman* hin). Man erkennt dies auch leicht an den durchschnittlichen Wartezeiten. Der Balancierer selbst wird - ohne Verwendung der graphische Anzeige - lediglich zu 5..20% beansprucht (für die Messungen wurde er gemeinsam mit *gclient* auf einem separaten Prozessor konfiguriert). Das Abschneiden der unterschiedlichen Strategien in den verschiedenen Situationen wollen wir nicht genauer betrachten, da sie nur Teilaspekte berücksichtigen. Die Messungen sollen im wesentlichen die Einsatzfähigkeit des Lastbalancierungsprototyps belegen und ihn damit für zukünftige Realisierungen des HiCon-Konzeptes qualifizieren. Wichtig ist uns die Abschätzung, inwiefern die zentrale Balancierungskomponente in der Lage ist, auch bei relativ feinem Auftragsgranulat den Durchsatz bei Erhöhung der Parallelität zu steigern ohne selbst ein Engpaß zu werden. Ein optimaler *Speedup* ist bei Operationen auf gemeinsamen Daten durch die resultierenden Reihenfolgeabhängigkeiten, Konsistenzbedingungen und Kommunikationskosten (die Grenzen der Parallelisierung) ohnehin nicht erreichbar. Deshalb verzichten wir an dieser Stelle auch auf genauere Auswertungen und auf Optimierung der Beispielanwendung bzw. der Balancierungsvarianten.

3.2.6 Ausblick

Zur Einschätzung des Anwendungsspektrums, für welches das vorgestellte Modell zur dynamischen Lastbalancierung geeignet ist, sind sicherlich noch weitere Applikationen zu untersuchen. Der nächste Schritt zur Realisierung des bisher entwickelten Balancierungsverfahrens besteht in der Kombination der oben betrachteten Einzelmethoden, wobei die Schwierigkeit vor allem in der geeigneten Gewichtung der diversen Größen zueinander liegt. Als weitere Schritte stehen die genaue Definition und Realisierung der Balancierungshierarchie (siehe Kapitel 2.4) sowie die Konkretisierung der Planungskomponente für Auftragsgruppen (siehe Kapitel 2.3.3) an. Von weitergehender Optimierung unseres Prototyps wie etwaigen Verkürzungen von Nachrichtenwegen oder der Nutzung niedrigerer Software-Ebenen sehen wir derzeit ab. Stattdessen werden wir tiefergehende Untersuchungen prinzipieller Design-Alternativen genauer erwägen: die Einführung einer Balancierungsinstanz je Serverklasse verspricht Vereinfachungen im Modell, bereitet jedoch Schwierigkeiten bei auftragsübergreifender Einplanung. Ebenso ist eine inhaltliche Dezentralisierung der Komponente, wie etwa die lokale Zuweisung kurzer, unwichtiger oder vom Profil her unbekannter Aufträge zu erwägen. Schließlich ist das derzeit gewählte Programmiermodell in seinen Vorzügen und Nachteilen mit Varianten zu vergleichen. So werden häufig (unter anderem im *Supervisor* [Schiele91]) Petrinetze zur Spezifikation und Synchronisation paralleler Abläufe bevorzugt; weiterhin kann man ‘multi-threaded’ Serverinstanzen (siehe Kapitel 2.2) in das Balancierungsmodell aufnehmen oder in Anlehnung an Datenbanksysteme globale Datensätze einführen. Schließlich läßt sich die Palette der bisher beleuchteten Lastbalancierungsfaktoren noch um Berücksichtigung der Verzögerungen durch Datensperren oder um Beachtung des verfügbaren Hauptspeicherplatzes bereichern.

Literaturverzeichnis

[Barak85] A. Barak, A. Shiloh, *A Distributed Load-balancing Policy for a Multicomputer*, Software-Practice and Experience, Vol. 15, No. 9, 1985.

[Becker92] W. Becker, *Lastbalancierung in heterogenen Client-Server Architekturen*, Fakultätsbericht 1992 /1, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, 1992.

[Berger87] M. Berger, S. Bokhari, *A Partitioning Strategy for Nonuniform Problems on Multiprocessors*, IEEE Transactions on Computers, Vol. 36, No. 5, 1987.

[Blazewicz86] J. Blazewicz, M. Drabowski, J. Weglarz, *Scheduling Multiprozessor Tasks to Minimize Schedule Length*, IEEE Transactions on Computers, Vol. 35, No. 5, 1986.

[Bokhari81] S. Bokhari, *A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in a Distributed Processor System*, IEEE Transactions on Software Engineering, Vol. 7, No. 6, 1981.

[Bowen88] N. Bowen, C. Nikolaou, A. Ghafoor, *Hierarchical Workload Allocation for Distributed Systems*, Proceedings Parallel Processing, 1988.

[Casavant88] T. Casavant, J. Kuhl, *A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems*, IEEE Transactions on Software Engineering, Vol. 14, No. 2, 1988.

[Ciciani88] B. Ciciani, D. Dias, P. Yu, *Load Sharing in Hybrid Distributed - Centralized Database Systems*, Proceedings Distributed Computing Systems, 1988.

[Copeland88] G. Copeland, W. Alexander, E. Boughter, T. Keller, *Data Placement in Bubba*, Proceedings SIGMOD, 1988.

[Cybenko89] G. Cybenko, *Dynamic Load Balancing for Distributed Memory Multiprocessors*, Journal of Parallel and Distributed Computing, No. 7, 1989.

[Douglis91] F. Douglis, J. Ousterhout, *Transparent Process Migration: Design Alternatives and the Sprite Implementation*, Software-Practice and Experience, Vol. 21, No. 8, 1991.

[Eager85] D. Eager, E. Lazowska, J. Zahorjan, *A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing*, ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Austin, Texas, 1985.

[Eager86] D. Eager, E. Lazowska, J. Zahorjan, *Adaptive Load Sharing in Homogeneous Distributed Systems*, IEEE Transactions on Software Engineering, Vol. 12, No. 5, 1986.

[Ezzat86] A. Ezzat, *Load Balancing in NEST: A Network of Workstations*, Proceedings Fall Joint Computer Conference, Dallas, Texas, 1986.

[Ferrari86] D. Ferrari, S. Zhou, *A Load Index for Dynamic Load Balancing*, Proceedings Fall Joint Computer Conference, Dallas, Texas, 1986.

[Gavish90] B. Gavish, O. Sheng, *Dynamic File Migration in Distributed Computer Systems*, Communications of the ACM, Vol. 33, No. 2, 1990.

[He89] X. He, *Eine Übersicht über die Lastverteilung in verteilten Systemen*, Bericht 190/89, Universität Kaiserslautern, Fachbereich Informatik, 1989.

[Hosseini90] S. Hosseini, B. Litow, M. Malkawi, J. Mc Pherson, K. Vairvan, *Analysis of a Graph Coloring Based Distributed Load Balancing Algorithm*, Journal of Parallel and Distributed Computing, No. 6, 1990.

[Hsu86] C. Hsu, J. Liu, *Dynamic Load Balancing Algorithms in Homogeneous Distributed Systems*, Proceedings Distributed Computing Systems, 1986.

[Iqbal86] M. Iqbal, J. Saltz, S. Bokhari, *A Comparative Analysis of Static and Dynamic Load Balancing Strategies*, Proceedings Parallel Processing, 1986.

[Kale88] L. Kale, *Comparing the Performance of two Dynamic Load Distribution Methods*, Proceedings Parallel Processing, 1988.

[Kanet91] J. Kanet, V. Sridharan, *PROGENITOR: A generic algorithm for production scheduling*, Wirtschaftsinformatik, Heft 4, August 1991.

[Kuchen90] H. Kuchen, A. Wagener, *Comparison of Load Balancing Strategies*, Bericht 5/90, Lehrstuhl für Informatik II, RWTH Aachen, 1990.

[Li90] K. Li, K. Cheng, *Static Job Scheduling in Partitionable Mesh Connected Systems*, Journal of Parallel and Distributed Computing, No. 10, 1990.

[Lin87] F. Lin, R. Keller, *The Gradient Model Load Balancing Method*, IEEE Transactions on Software Engineering, Vol. 13, No. 1, 1987.

[Lin92] H. Lin, C. Raghavendra, *A Dynamic Load-Balancing Policy With a Central Job Dispatcher (LBC)*, IEEE Transactions on Software Engineering, Vol. 18, No. 2, 1992.

[Lo88] V. Lo, *Algorithms for Static Task Assignment and Symmetric Contraction in Distributed Computing Systems*, Proceedings Parallel Processing, 1988.

[Lo88/2] V. Lo, *Heuristic Algorithms for Task Assignment in Distributed Computing Systems*, Proceedings Parallel Processing, 1988.

[Ma82] P. Ma, E. Lee, M. Tsuchiya, *A Task Allocation Model for Distributed Computing Systems*, IEEE Transactions on Computers, Vol. 31, No. 1, 1982.

[Schiele91] G. Schiele, *Kontrollstrukturen zur Ablaufsteuerung massiv paralleler Datenbankanwendungen in Multiprozessorsystemen*, Dissertation, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, 1991.

[Smith80] R. Smith, *The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver*, IEEE Transactions on Computers, Vol. 29, No. 12, 1980.

[Staib92] R. Staib, *Simulator zur Lastbalancierung in Client / Server Architekturen*, Diplomarbeit, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, 1992.

[Thomasian86] A. Thomasian, P. Bay, *Analytic Queueing Network Models for Parallel Processing of Task Systems*, IEEE Transactions on Computers, Vol. 35, No. 12, 1986.

[Towsley86] D. Towsley, *Allocating Programs Containing Branches and Loops within a Multiple Processor System*, IEEE Transactions on Software Engineering, Vol. 12, No. 10, 1986.

[Trippner92] R. Trippner, *Lastbalancierung einer exemplarischen Anwendung auf vernetzten Workstations*, Diplomarbeit, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, 1992.

[Varadarajan88] R. Varadarajan, E. Ma, *An Approximate Load Balancing Model with Resource Migration in Distributed Systems*, Proceedings Parallel Processing, 1988.

[Weinmann92] T. Weinmann, *Datenparallelität durch Partitionierung und Replikation*, Studienarbeit, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, 1992.

[Yu86] P. Yu, S. Balsamo, Y. Lee, *Dynamic Load Sharing in Distributed Database Systems*, Proceedings Fall Joint Computer Conference, 1986.

[Yu91] P. Yu, A. Leff, Y. Lee, *On Robust Transaction Routing and Load Sharing*, ACM Transactions on Database Systems, Vol. 16, No. 3, 1991.

[Zhou87] S. Zhou, D. Ferrari, *An Experimental Study of Load Balancing Performance*, Report No. 86.8, Computer Science Division, University of California, Berkeley, 1987.