# Lastbalancierung in heterogenen Client - Server Architekturen

Wolfgang Becker

Bericht Nr. 1992 / 1

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

# Lastbalancierung in heterogenen Client-Server Architekturen

**Fakultätsbericht Nr. 1992 / 1**

Wolfgang Becker

Institut für Parallele und Verteilte Höchstleistungsrechner (IPVR)

Fakultät für Informatik

Universität Stuttgart

Breitwiesenstr. 20-22, D-7000 Stuttgart 80

Wolfgang.Becker@informatik.uni-stuttgart.de

# Inhaltsverzeichnis

# 1 Einleitung

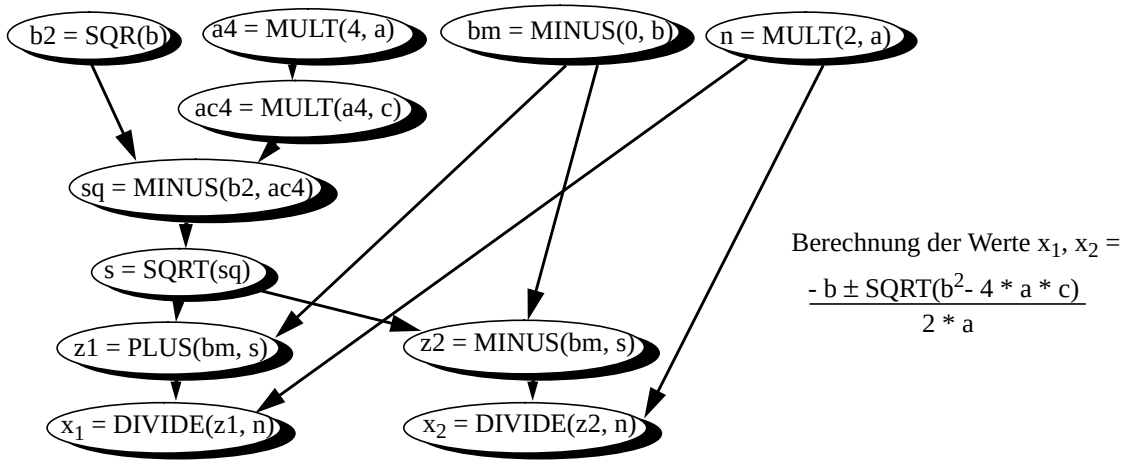
Dieser Bericht erarbeitet zunächst allgemein das Gebiet der Lastbalancierung. Dabei werden Anforderungen, Grundelemente und Verfahren gesammelt. Im zweiten Teil (Kapitel 7) wird das Grundkonzept unseres Lastbalancierungssystems (*HiCon*) vorgestellt. HiCon besteht aus Komponenten, die in heterogenen Client-Server Architekturen dynamische Lastbalancierung durchführen.

## 1.1 Mögliche Parallelarbeit in Anwendungen

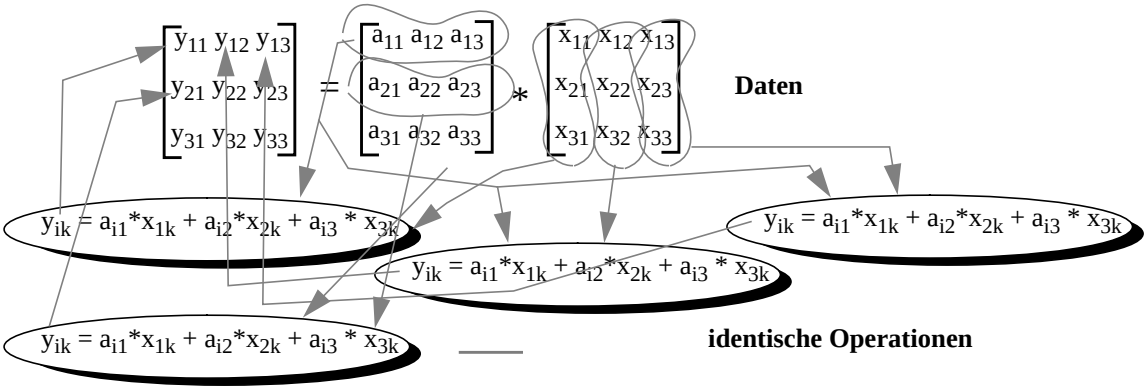
Viele Anwendungen kann man beschleunigen, indem man die Parallelität, die in ihnen steckt, ausnutzt. Man betrachtet die einzelnen Operationen der Anwendung und legt deren notwendige Reihenfolgebeziehungen fest. Ein System, das eine solche Anwendung ausführt, kann dann die einzelnen Operationen in beliebiger Reihenfolge, an beliebigem Ort und auch parallel durchführen; es muß lediglich die vorgegebenen Reihenfolgebeziehungen einhalten.

Gewöhnlich unterscheidet man zwei Arten der Parallelarbeit:

- **Funktionale Parallelität** besteht darin, verschiedene Funktionen zugleich auszuführen, die voneinander reihenfolge-unabhängig sind. Das hängt von den Daten ab, die eventuell von beiden Funktionen gelesen oder geändert (bzw. erzeugt) werden. Das Bild zeigt ein Beispiel. Abhängigkeiten zwischen Funktionsausführungen sind durch Pfeile dargestellt. Jede Funktion kann ablaufen, sobald die Daten, die sie benötigt, vorhanden (bzw. im benötigten Bearbeitungszustand) sind.



- **Daten-Parallelität** nutzt man, indem man eine Funktion parallel auf verschiedene Daten ansetzt. Diese Ausführungen müssen reihenfolge-unabhängig sein. Im Bild werden zwei Matrizen multipliziert; man kann dabei die eine Funktion $y_{ik} = a_{i1}*x_{1k} + a_{i2}*x_{2k} + a_{i3}*x_{3k}$ auf verschiedene Matrixelemente parallel anwenden. Die Pfeile deuten den Datenfluß an.




Prinzipiell können beide Formen der Parallelität zugleich eingesetzt werden. Die Unterscheidung zwischen funktionaler Parallelität und Parallelität in den Daten entfällt, wenn man verschiedene Ausführungen derselben Funktion auch als verschiedene Objekte ansieht, d.h. genauso als wären dies Ausführungen verschiedener Funktionen gewesen.

*Reihenfolge-Unabhängigkeit* von Daten bezüglich einiger Funktionsanwendungen bedeutet, daß das Ergebnis der Funktionsabläufe auf diesen Daten unabhängig von deren Reihenfolge und eventueller Parallelarbeit richtig ist (es muß nicht unbedingt identisch, aber im Sinne der Anwendung korrekt sein).

| | | |
|---|---|---|
| Beispiel1: Berechnung von D = B-2*A. | A = 2*A<br>D = B-A<br>richtiges Ergebnis | D = B-A<br>A = 2*A<br>falsches Ergebnis |
| Beispiel2: Berechnung von A = A+B+C. | A = A+B<br>A = A+C | A = A+C<br>A = A+B |

Die Reihenfolge ist beliebig, nur parallel sollten die Operationen nicht ablaufen (das ist typisch für Datenbankanwendungen).

Man kann Anwendungen unterschiedlich fein in Grundfunktionen (die jeweils in sich sequentiell ablaufen) und Datenobjekte (die nicht weiter aufgegliedert werden) zerlegen. Das Granulat ist ein entscheidender Faktor für den Nutzen der Parallelisierung.

## 1.2 Parallele und verteilte Rechnersysteme

Lastbalancierung ist bereits auf zentralen Systemen möglich, sie kommt jedoch erst auf parallelen und verteilten Rechnersystemen entscheidend zum Tragen. Zeitgewinn bringende Parallelarbeit wird durch folgende Phänomene möglich:

- Programmausführungen beinhalten *Wartezeiten*. Das Programm kann oft erst fortgesetzt werden, wenn bestimmte Daten zur Verfügung stehen. Diese Daten müssen von langsamen Geräten (Platten, Benutzerterminals) geladen oder von anderen Funktionsausführungen erzeugt werden. In der Zwischenzeit kann auf dem Prozessor eine andere Funktion durchgeführt werden.
- *Mehrprozessorsysteme* können auf allen Prozessoren gleichzeitig arbeiten. SIMD-Rechner können eine Operation gleichzeitig auf verschiedene Daten anwenden, MIMD-Rechner können sogar unterschiedliche Operationen auf verschiedene Daten zugleich anwenden.
- Viele Anwendungen, vor allem im Datenbankbereich, greifen intensiv auf Daten zu, die auf stabilen Speichermedien (Platten) gespeichert sind. Hat man mehrere solcher Medien, so kann man die Daten darüber verstreuen und dadurch parallel auf die *physisch verteilten Daten* zugreifen.

Wie weit man parallele Hardware tatsächlich für parallele Abläufe nutzen kann, hängt nicht allein von der prinzipiell in der Anwendung steckenden Parallelität ab. Je nachdem, wie zeitaufwendig der Transport von Daten zwischen den Prozessoren (bzw. zwischen den Funktionen auf den Prozessoren) ist, wird der Geschwindigkeitsgewinn des parallelen Rechnens durch die Wartezeiten während Datentransporten beeinträchtigt. Ein wichtiger Faktor sind dabei die Verbindungen (Busse, Netzwerke) zwischen den verschiedenen Prozessoren und Datenträgern (siehe Kapitel 3.1).

## 1.3 Beschreibung paralleler und verteilter Anwendungen

In sequentiellen Programmen benutzt man einen Ablaufplan und eine Datenbeschreibung. In der allgemeinsten Form paralleler Programmierung geht man mit Funktionsausführungen und Datenversionen um; dazu gibt es eine Ablaufbeschreibung, die angibt, welche Funktionsausführung auf welche Daten (in welchem Bearbeitungszustand der Daten) angewendet werden soll. Daten und Funktionen sind keine festen Objekte mehr, denn unterschiedliche Versionen eines Datenobjekts können gleichzeitig an verschiedenen Orten existieren und mehrere Ausführungen einer Funktion finden gleichzeitig an verschiedenen Orten statt.

Auf der untersten Ebene paralleler und verteilter Anwendungen laufen (sequentielle) Prozesse parallel ab und versenden untereinander Nachrichten. Die Nachrichten dienen zum Datenaustausch sowie zur Synchronisation (auf zentralen oder speichergekoppelten Systemen kann beides auch mithilfe globaler Variablen geschehen). Darauf aufbauende höhere Ebenen bieten übersichtlichere Programmiermodelle an:

Datenflußsprachen, Petri-Netze und Prädikat-Transitionsnetze ermöglichen die Beschreibung von Einzeloperationen und deren gegenseitigen Abhängigkeiten ohne daß man die genauen Synchronisationsvorgänge oder die Verteilung der Funktionen und Daten spezifizieren muß. Diese Modelle stellen Reihenfolgebeziehungen sehr klar heraus (in Datenflußsprachen beschreibt man sogar die eigentlichen Datenabhängigkeiten), sind aber für Iterationen und Rekursionen schlecht geeignet.

Das Modell des *Remote Procedure Call* ist eine Erweiterung der prozeduralen Programmierung. Ein Prozeduraufruf muß nicht lokal, sondern kann irgendwo im System ausgeführt werden. Parallelität erreicht man durch asynchrone Aufrufe; man muß die Ausführung eines Funktionsaufrufs nicht sofort abwarten, sondern kann (parallel) weiterrechnen, bis man die Ergebnisse der Funktionsausführung benötigt.

Im Client-Server Modell stehen einige Dienste (Server) zur Verfügung, die von Anwendungen (Clients) aufgerufen werden können; dabei können mehrere Server denselben Dienst anbieten. Die Aufrufe geschehen nach dem Prinzip des *Remote Procedure Call*.

In diesen Modellen muß der Programmierer selbst Reihenfolgebeziehungen in Aufrufsequenzen umsetzen; andererseits kann man viele Vorteile der zentralen sequentiellen Programmierung übernehmen (hierarchische Abstraktion von Datentypen, hierarchische Abstraktion des Kontrollflusses durch Unterprozeduren und Fehlerbehandlung). Der Datenfluß wird nicht explizit angegeben, jede Operation übernimmt die Daten in dem Zustand, in dem sie sich gerade befinden.

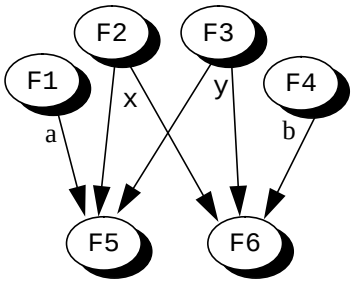
Als Beispiel asynchroner Funktionsaufrufe soll obige Formelberechnung in diesem Modell beschrieben werden (siehe Bild). Man bekommt beim Aufruf einer Funktion eine Nummer. Mit dieser Nummer kann man später die Funktionsergebnisse abholen (bzw. abwarten).

```
call1 = SQR(b)
call2 = MULT(4, a)
call3 = MINUS(0, b)
call4 = MULT(2, a)
call5 = MULT(result(call2), c)
call6 = MINUS(result(call1), result(call5))
call7 = SQRT(result(call6))
bm = result(call3)
s = result(call7)
call8 = PLUS(bm, s)
call9 = MINUS(bm, s)
n = result(call4)
```

```
call10 = DIVIDE(result(call8), n)
call11 = DIVIDE(result(call9), n)
x1 = result(call10)
x2 = result(call11)
```

Um dieselbe Mächtigkeit wie Datenflußsprachen zu erreichen, muß der Programmierer auch auf die Vollendung irgendeiner laufenden Ausführung warten können (siehe Bild).

Datenabhängigkeiten:

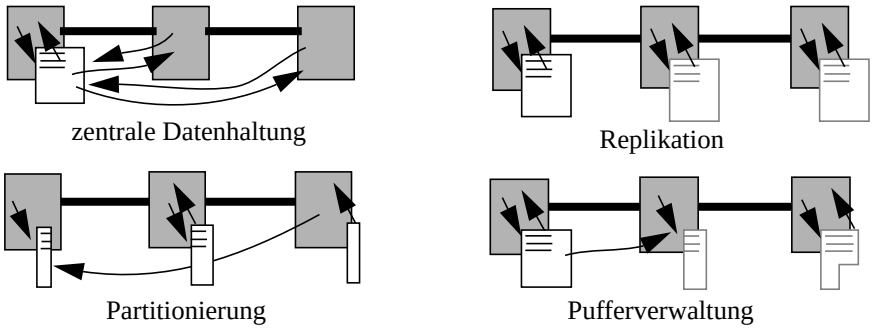



Beschreibung durch asynchrone Aufrufe:

```
x = result(F2call)
y = result(F3call)
ab = result(&any)
if any=F1call then
   call5 = F5(ab,x,y)
   b = result(F4call)
   call6 = F6(b,x,y)
else
   call6 = F6(ab,x,y)
   a = result(F1call)
   call5 = F5(a,x,y)
```

## 1.4 Replikation und Partitionierung von Daten

Datenparallelität benutzt man, um durch parallele Bearbeitung von Daten auf verschiedenen Prozessoren die Bearbeitungsgeschwindigkeit zu steigern. Dazu kann man einen Datensatz über die Prozessoren aufteilen (*Partitionierung*) oder Kopien des Datensatzes auf die Prozessoren verteilen (*Replikation*). Dazwischen sind einige Mischformen möglich. Das Bild zeigt einige gebräuchliche Verfahren (die Symbole werden in Kapitel 3.1 erklärt).




Das Problem der Partitionierung liegt darin, daß Zugriffe nur auf diejenige Partition der Daten billiger sind, die lokal auf dem Prozessor liegt. Bei der Replikation hat man hingegen das Problem, daß eine Änderung an einer Kopie sofort in allen anderen Kopien nachvollzogen werden muß. Dazu sind Sperrverfahren notwendig. Allgemein lohnt sich Partitionierung bei einer gewissen Lokalität der Datenzugriffe, während Replikation bei einer relativ großen Häufigkeit von Lesezugriffen Gewinn bringt.

## 1.5 Lastbalancierung

Lastbalancierung soll Aufträge so geschickt verteilen und zur richtigen Zeit bearbeiten lassen, daß sie möglichst schnell erledigt werden. Dazu versucht man, die Hardware-Ressourcen voll zu nutzen und Engpässe zu vermeiden.

Der Lastbalancierer verfügt über einige Arbeitskräfte (*Server*), die ihm Aufträge (**Funktionen**) ausführen können. Die Arbeiter, welche dieselbe Funktion ausführen können, faßt man durch die Bezeichnung *Serverklasse* zusammen. Der Lastbalancierer kann beliebig viele Kopien eines Servers (*Instanzen* der Serverklasse) erzeugen und beschäftigen, soweit genug Betriebsmittel verfügbar sind. Da aber die Rechenkapazitäten für die Arbeiter begrenzt sind und

die notwendige Absprache zwischen den Arbeitern irgendwann stärker wächst als die Produktivität, muß ein Lastbalancierer ein sinnvolles Maß für ihre Anzahl finden.

**Aufträge** verlangen, daß die Funktionen bestimmte **Daten** bearbeiten. Prinzipiell kann man sich (im Rahmen der vorhandenen Betriebsmittel) beliebig viele Kopien der Datensätze herstellen (*Replikation*). Doch der Speicherplatz sowie der wachsende Aufwand, um die Kopien der Daten untereinander in Übereinstimmung zu halten, zwingt zu einer angemessenen Anzahl von Datenkopien.

# 2 Aufgaben und Ziele der Lastbalancierung

## 2.1 Welche Dienste erwartet man von der Lastbalancierung?

Wir zeigen eine Sammlung von Aufgaben, die mit der Lastbalancierung im Zusammenhang stehen:

- **Funktionsausführungs-Management**. Der Lastbalancierer soll Funktionsausführungen starten, stoppen, fortsetzen, verlagern (*migrieren*), beschleunigen und abbremsen. Er bekommt (im allgemeinsten Fall) von den Anwendungen Gruppen von Funktionsaufrufen, die unter bestimmten *Reihenfolgebeziehungen* abzuarbeiten sind. Seine Menge von bereitstehenden Funktionen muß er nun so einsetzen, daß er, mit Rücksicht auf weitere Anwendungen und andere Funktionen, die dieselben Ressourcen benutzen, alle geforderten Aufrufe in minimaler Zeit abwickelt.

  Auf Betriebssystemebene bedeutet das die Verwaltung der Prozesse, die zur Ausführung von Funktionen bereitstehen; die Zuweisung von Aufträgen, Einstellung der Prozeßprioritäten und das Stoppen sowie Fortsetzen von Prozessen.

  In Echtzeitanwendungen müssen oft Zeitlimits eingehalten werden. Hier ist das Kriterium nicht unbedingt möglichst schnelle, sondern die fristgerechte Ausführung der Aufträge.

- **Funktions-Management**. Der Lastbalancierer hält sich eine Menge von Funktionen (*Serverinstanzen*), die über das System verteilt bereitstehen. Jede Funktion kann mit der Durchführung eines Aufrufes beauftragt werden; sie steht dann wieder zur Verfügung, sobald sie diesen Aufruf abgearbeitet hat (bei Server-Multitasking verkörpert ein Server entsprechend mehrere Instanzen). Der Lastbalancierer soll seine Funktionen stets in geeigneter Anzahl auf geeigneten Prozessoren verfügbar halten, um eine schnelle Bearbeitung aller Aufträge durch optimale Ausnutzung der Ressourcen zu garantieren. Dabei sind die Ressourcen zu berücksichtigen, die das Bereithalten einer Funktion erfordert sowie der Aufwand, um gemeinsame Daten zwischen den Funktionen einer Klasse konsistent zu halten.

  Das Bereithalten von Funktionen ist nicht prinzipiell notwendig, aber der Aufwand, um eine Funktion wegen eines einzelnen Aufrufs zu installieren und danach wieder abzubauen, ist meist unrentabel groß (Prozeßstartzeiten, Verbindungsaufbauzeiten sowie das Kopieren der Daten und Kontexte von anderen Instanzen der Klasse).

- **Durchsatz- und Systemlastmessung**. Der Lastbalancierer sollte die Ausführungszeiten der Aufträge messen und daran die Wirkung seiner Strategien bewerten. Weiterhin muß er die Belastung der Funktionen und Ressourcen messen, um Überlastung einzelner Komponenten (Server, Prozessoren, Datensätze, Platten und ähnliches) abzufangen und Aufträge geschickt an unbelastete Ressourcen zu verteilen. Aus statistischen Meßreihen könnte er selbstständig seine Strategie verbessern.

- **Beobachtung der Hardware-Konfiguration**. Der Lastbalancierer sollte stets über die aktuelle Konfiguration seines Systems informiert sein, damit er bei Umkonfigurierung und Komponentenausfällen seine Funktionen, Daten und Funktionsausführungen entsprechend umplanen kann. Die Erhöhung der Systemverfügbarkeit ist sehr eng mit der Lastbalancierung verbunden, wobei ein Auftrag nicht nur grundsätzlich, sondern sogar möglichst schnell erledigt werden soll.

- **Ablaufüberwachung**. Der Lastbalancierer soll die Durchführung der Funktionsaufrufe kontrollieren und Fehler sowie Ausfälle behandeln (in einer für die Anwendung möglichst transparenten Weise). Bei wichtigen, langlebigen Daten (wie Datenbanksätzen) sollte er Konsistenz garantieren (atomare Funktionsausführungen). Bei langlaufenden Anwendungen sollte er durch *Checkpoints* die Ausmaße der durch Fehler und Ausfälle verlorenen Arbeit klein halten.

- **Wahl des Funktionsgranulats**. Wie gut der Lastbalancierer die Aufträge auf sein System anpassen kann, hängt davon ab, in wieviel unabhängige Funktionen eine Anwendung aufgegliedert wird. Prinzipiell sind feinere Granulate leichter zu balancieren. Andererseits wirkt der zunehmende Aufwand für Verwaltung und Kommunikation dem Leistungsgewinn der Parallelarbeit entgegen. Der Lastbalancierer könnte das jeweils beste Granulat bestimmen.
- **Wahl des Datengranulats**. Da ein Datenobjekt meist nur von einer Funktionsausführung zugleich bearbeitet (vor allem modifiziert) werden kann, entstehen Engpässe durch andere Funktionsausführungen, die auf das Objekt zugreifen möchten. Wenn Funktionen nur Teile des Objekts benötigen, erlaubt eine Zerteilung des Objekts, daß an den Einzelteilen parallel gearbeitet werden kann. Der Lastbalancierer könnte entscheiden, welches Datengranulat geeignet ist.
- **Replikation und Verteilung der Daten** (Daten-Management). Der Lastbalancierer sollte Daten und Kopien bzw. Ausschnittskopien (Puffer) von Daten so geschickt über das System verteilen, daß die Funktionen möglichst lokal auf die Daten zugreifen können, während der Aufwand, um die Datenkopien konsistent zu erhalten, gering bleibt.
- **Kommunikations-Management**. Der Lastbalancierer soll dafür sorgen, daß Nachrichten möglichst effizient verschickt werden. Dazu kann er das *Routing* (Bestimmung der Pfade durch das System), die Paketverpackung und Bündelung der Nachrichten übernehmen bzw. geeignet einstellen. Darüber hinaus könnte er stehende Verbindungen überwachen und in Fehlerfällen bzw. nach Umkonfigurierungen wiederherstellen.
- **Ortstransparenz**. Da der Lastbalancierer die Zuordnung (*Mapping*) von Aufträgen zu Prozessoren, Prozessen und Platten übernimmt, sollten Anwendungen unabhängig von der Systemkonfiguration formuliert werden können. Der Ausführungsort von Funktionen, der Ort der Daten und der Zielort von Nachrichten sollte also für den Anwender transparent sein (er stellt sich ein zentrales System vor).

Dies alles sind nur Möglichkeiten; kein Lastbalancierer wird alles in Form einer Komponente übernehmen. Eventuell werden Teilaufgaben von separaten Komponenten erledigt. Alle Vorgänge und Informationen aber, welche die Balancierungsstrategie berücksichtigt, sollten zumindest an ihr vorbeilaufen, nicht über sie hinweg verarbeitet werden.

## 2.2 Was kann der ideale Lastbalancierer?

Der ideale Lastbalancierer verarbeitet alle Aufträge so, daß im Mittel die Summe aller Ausführungszeiten der Aufträge minimal ist. Bei der Summation kann er die Aufträge nach ihrer Priorität gewichten. Die Ausführungszeit eines Auftrages startet, sobald der Benutzer den Auftragswunsch zur sofortigen Ausführung äußert und endet mit der letzten dazu notwendigen Funktionsausführung.

Der ideale Lastbalancierer verfügt stets über alle aktuellen Informationen; er weiß sie sogar im voraus. Genau gesagt, weiß er bereits beim Start des Rechnersystems, welche Anwendungen wann laufen werden. Er kann daher jede Anwendung statisch (nicht bei der Übersetzung des Programms, aber bei jedem Start der Anwendung) aufgliedern und den Komponenten (nicht den Funktionen, aber jeder Funktionsausführung) und Daten feste Orte auf dem System sowie feste Aktivierungszeitpunkte zuweisen.

Da der ideale Lastbalancierer zudem beliebig schnell planen kann, geht er folgendermaßen vor: Er bildet zunächst alle Aufträge fest und zur frühest möglichen Zeit auf das System ab. Für diesen Plan kann er ja die Ausführungszeit je Auftrag berechnen und damit auch die Summe aller Ausführungszeiten. Nun variiert er sowohl das Granulat der Auftragszerlegung (samt Replikation und Partitionierung) als auch die Zuordnung (der Funktionsausführungen zu Prozessoren mit Prioritäten und der stabilen Daten zu den Platten) als auch die Zeitpunkte der Funktionsausführungen. Zu jeder Variante berechnet er die Summe der Laufzeiten, und am Ende wählt er die Variante, bei der die Summe minimal ist.

*Reale Lastbalancierer* haben ein schwierigeres Leben. Einerseits wissen sie die zukünftigen Aufträge nicht im voraus, haben bereits einige Aufträge am Laufen und bekommen zu neuen Aufträgen nur ungenaue Information über deren Bedürfnisse. Daher sollten sie das System während der Laufzeit beobachten, zur Laufzeit Funktionsaufrufe und Daten zuweisen und notfalls auch die Zuordnung laufender Aktivitäten korrigieren. Andererseits können sie nicht beliebig schnell planen, ihre Denkzeit sowie auch das Sammeln von Laufzeitinformationen beansprucht die Ressourcen, die ihnen zur Bearbeitung der Aufträge zur Verfügung stehen.

Die Kunst der Lastbalancierung besteht nunmehr darin, statisch möglichst viel Wissen über die Aufträge zu erwerben und viele der ‘Drehknöpfe’ der Lastbalancierung bereits gut einzustellen. Zur Laufzeit ist es dagegen wichtig, anhand weniger kritischer Meßgrößen wenige ‘Drehknöpfe’ geschickt nachzuregeln und so aus der Situation das Beste zu machen.

# 3 Elemente und Größen in der Lastbalancierung

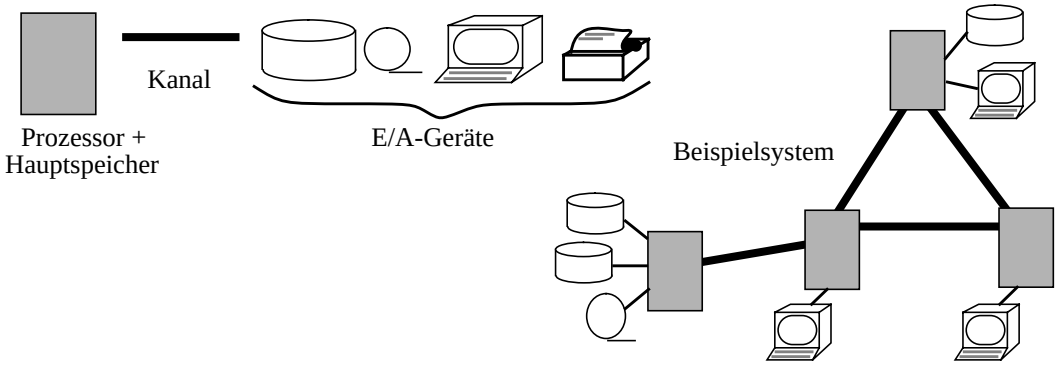
Wie sieht der Lastbalancierer das System, die Daten und die Aufträge, mit denen er zu tun hat? Im folgenden sammeln wir einige Elemente und ihre Eigenschaften.

## 3.1 Physische Ressourcen

Die physischen Ressourcen, mit denen der Lastbalancierer arbeitet, sind die Hardware-Komponenten des Systems. Dazu gehören Prozessoren (CPUs), Hauptspeicher, Ein-/Ausgabegeräte (Platten, Bandlaufwerke), Busse, Netzwerke (Kanäle) und eventuell Steuereinheiten (Controller). In heterogenen Systemen ist es wichtig, daß der Lastbalancierer ein einheitliches Modell zur Beschreibung der Ressourcen kennt.

Um ein Rechnersystem optimal ausnutzen zu können, braucht der Balancierer möglichst detaillierte Informationen; andererseits sollte die Systembeschreibung möglichst einfach sein, um die Lastbalancierungsstrategie schnell und einfach zu halten. Die Wahl einer geeigneten Abstraktionsstufe für das Systemmodell ist daher wichtig.

Wir wollen uns auf die Elemente Prozessoren, E/A-Geräte und Kanäle im Systemmodell beschränken. Das Bild zeigt die verwendeten Symbole und Kombinationsmöglichkeiten. Kanäle verbinden nur Prozessoren untereinander, E/A-Geräte sind direkt (ohne Kanal) an einen Prozessor gebunden.




Das Systemmodell weist folgende Merkmale auf:

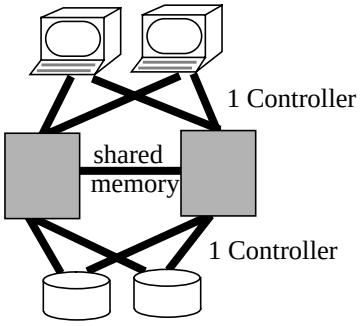
- Es gibt keine explizite Darstellung von gemeinsamem Speicher zwischen Prozessoren. Das muß man durch einen sehr schnellen Kanal zwischen diesen Prozessoren nachbilden. Wie realistisch das ist, hängt unter anderem davon ab, wie groß die lokalen Pufferspeicher der Prozessoren sind.
- SIMD-Architekturen und Pipeline-Rechner kann man nicht adäquat modellieren.
- Es gibt keine expliziten Steuereinheiten, Adapter, Hilfsprozessoren, DMA-Einheiten oder Bus-Umschalter, obwohl man diese bei der Lastbalancierung einbeziehen könnte, da das Systemmodell für den Lastbalancierer sonst zu kompliziert würde.
- Geräte, die an mehrere Prozessoren angeschlossen sind (etwa in *Tandem* Systemen) sind schlecht zu modellieren.
- Busse und Netzverbindungen werden nicht unterschieden. Sie tauchen entweder gar nicht (etwa Busse zwischen Prozessoren und ihrem Hauptspeicher oder ihren Platten) oder als Kanal auf (etwa Busse zwischen Prozessoren oder Leitungen zwischen Knoten).
- Kanäle können mehrere Prozessoren untereinander verbinden (nicht nur Punkt zu Punkt Verbindungen).
- Eine Verbindung zur ‘Außenwelt’, für die man kein genaues Systembild hat, muß man durch ein Gerät oder einen Prozessor (oder beides) modellieren, wenn sie in der Lastbalancierung berücksichtigt wird.

Die Ressourcen haben statische (bauartbedingte) und dynamische Eigenschaften. Die Tabelle listet mögliche Eigenschaften für Ressourcen auf:

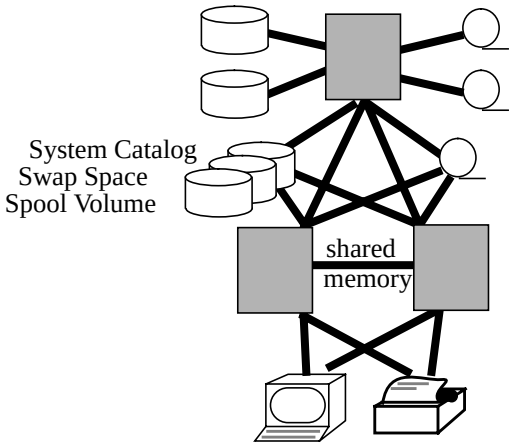
| **Ressource-Typ** | **statische Eigenschaften** | **dynamische Eigenschaften** |
|---|---|---|
| Prozessor | Geschwindigkeit | laufende / wartende Prozesse |
| | Betriebssystem-Typ | Auslastung (*busy-time*) |
| | Multitasking / Queueing | Hauptspeicherbelegung/*Swapping*-Rate |
| | max. Anzahl an Prozessen | |
| | Verfügbarkeit | |
| | Hauptspeichergröße | |
| | zugehöriger virtueller Speicher | |
| E/A-Gerät | Gerätetyp | wartende Prozesse |
| | Geschwindigkeit | Auslastung (*busy-time*) |
| | Blockgröße | |
| | Speicherplatz | |
| | Verfügbarkeit | |
| | Prozessorlast pro Zugriff | |
| Kanal | Geschwindigkeit | Auslastung (*busy-time*) |
| | Paketgröße | Länge der Nachrichtenwarteschlangen |
| | Verfügbarkeit | |

Zu diesen Daten für die einzelnen Komponenten gehört natürlich noch eine Verbindungstabelle. Sie ist statisch, solange sie keine schaltbaren Kanäle enthält oder häufig Elemente zu- und abgeschaltet werden.

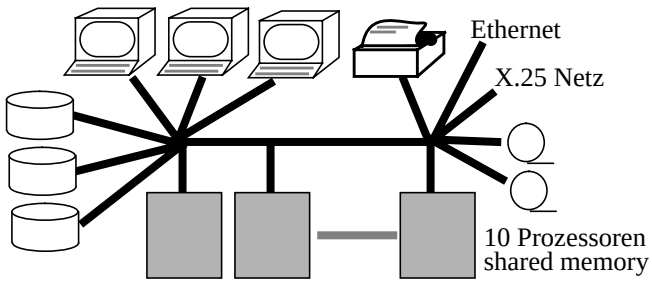
Wir stellen einige Rechnersysteme als Beispiele in diesem Ressourcenmodell dar:



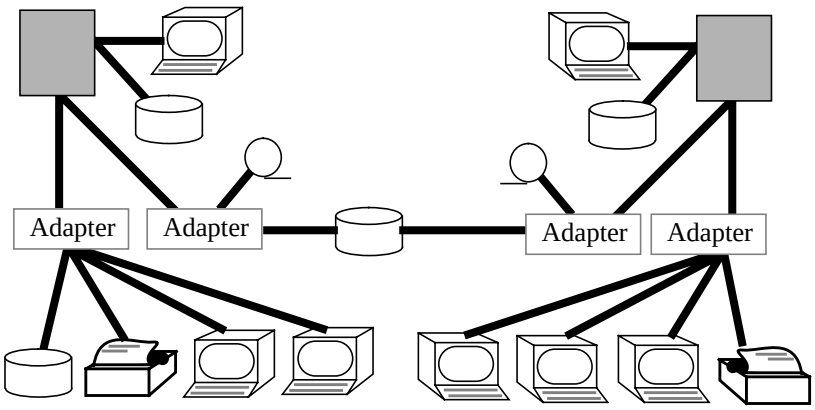

IBM 370/168 MP



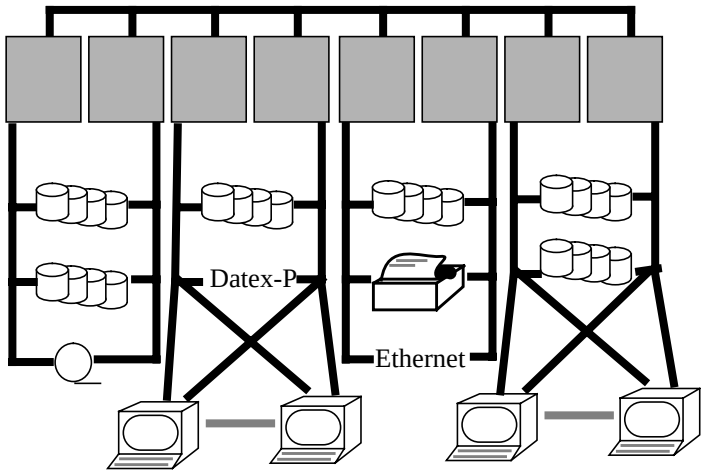

IBM 379/168 AP




Sequent Symmetry Multiprozessor



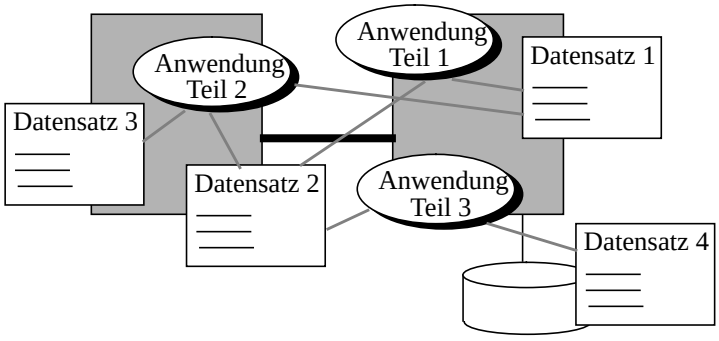

zwei gekoppelte VAX 11 Rechner




Tandem TXP Rechnersystem

## 3.2 Datenobjekte der Anwendungen

Die Ressourcen beherbergen Datenobjekte, mit denen die Anwendungen umgehen. Da man normalerweise nicht eine Platte oder den gesamten Hauptspeicher eines Prozessors als ein Datenobjekt betrachtet, und genausowenig die Gesamtmenge der Daten, mit denen eine Anwendung umgeht, als atomaren Datensatz behandeln will, gibt es eine Ebene über den physischen Ressourcen die Datenobjekte (siehe Bild, die gestrichelten Linien deuten Datenzugriffe an).




Aus Programmiersprachen kennt man die hierarchische Abstraktion und Zusammenfassung von Daten durch Typen. Es gibt Daten mit unterschiedlichen Sichtbarkeits- und Gültigkeitsbereichen. Dabei unterscheidet man auch flüchtige Daten, die solange wie die Anwendung leben und meist im Hauptspeicher gehalten werden, von nichtflüchtigen, die

ewig leben und gewöhnlich auf einer Platte gespeichert sind. Oft haben Daten auch Zugriffs- und Konsistenzbedingungen, die von den Anwendungen beachtet werden müssen.

Für die Lastbalancierung interessiert vor allem der Ort und das **Zugriffsverhalten** auf die Daten. Entscheidend für die Zugriffskosten ist die Nähe der Daten; daher legt man Kopien (**Replikate**) oder Puffer (Teil-Replikate) der Daten an verschiedene Orte (siehe auch Kapitel 1.4). Prinzipiell verbessern Replikate die Performance bei parallelen Lesezugriffen und beeinträchtigen sie bei parallelen Änderungsoperationen.

Weiterhin ist das **Granulat** der Daten für die Lastbalancierung entscheidend. Je feiner die Daten aufgeschlüsselt werden können, um so größer wird die potentielle **Datenparallelität**. Die Konsistenthaltung replizierter Daten verursacht dann aber in der Regel höheren Aufwand (kleine Puffer, viele Nachrichten, viele kleine Sperren).

Daten haben (wie Ressourcen) statische und dynamische Eigenschaften:

| **statische Eigenschaften** | **dynamische Eigenschaften** |
|---|---|
| Speicherbedarf | Ort der Daten, Ort von Kopien und Puffern |
| Zugriffsverfahren/ -Pfade | Zugriffshäufigkeit und -art (Lesen / Änderungen) |
| Konsistenzbedingungen | |
| geforderte Verfügbarkeit | |

## 3.3 Kommunikation

Das Nachrichtenaufkommen zwischen Prozessen und Prozessoren (auf höherer Ebene zwischen Funktionen) ist für die Lastbalancierung interessant, da die Kanäle zum Engpaß werden können und Kommunikations-Wartezeiten bei der Ausführungsdauer von Aufträgen mit einzurechnen sind.

Nachrichten werden oft paketweise in gebündelter Form verschickt. Wesentlich ist daneben, ob es nur Punkt-zu-Punkt Verbindungen oder auch einen effizienten Multicast- bzw. Broadcast-Mechanismus gibt. Der Lastbalancierer sollte eventuell auch wissen, ob Nachrichten verbindungsorientiert verschickt werden, denn meist ist dort die mehrmalige Wiederbenutzung einer stehenden Verbindung schneller als der ständige Neuaufbau nach dem Umschalten auf einen anderen Partner.

Der Lastbalancierer kann Wissen über die Semantik der Nachrichten ausnutzen. Beispielsweise wartet der Aufrufer eines synchronen Prozeduraufrufs solange, bis er die Ergebnisnachricht erhält. Wenn Nachrichten semantisch ‘call by reference’ Daten übertragen, so entstehen dadurch nebenbei Sperren, Synchronisationsaufwand zwischen Aufrufer und Bearbeiter.

Auch die Kommunikation zwischen Funktionen hat statische und dynamische Eigenschaften:

| **statische Eigenschaften** | **dynamische Eigenschaften** |
|---|---|
| geforderte Geschwindigkeit | bestehende Verbindungen |
| verbindungsorientiert? | Nachrichtenaufkommen (Menge, Häufigkeit) |
| unidirektional / abwechselnd / duplex | Warteschlangen beim Sender / Empfänger |
| geforderte Verfügbarkeit | |

## 3.4 Aufträge

Die Aufträge sind das Optimierungskriterium der Lastbalancierung; sie sollen im Mittel schnellstmöglich ausgeführt werden. Ein Auftrag besteht gewöhnlich aus vielen kooperierenden Teilaufgaben. Für die Lastbalancierung interessieren die Bedürfnisse der Teilaufgaben und deren gegenseitige Abhängigkeiten. Man möchte wissen, wann welcher Teilauftrag ablaufen soll und welche Reihenfolgebeziehungen zu beachten bzw. auszunutzen sind. Ein Lastbalancierer, der keinerlei Vorabinformation über die Aufträge hat (weder über die einzelnen noch über Auftragsgruppen), kann nur *reaktiv* wirken (siehe Kapitel 4.2).

Wir betrachten als **Lastprofile** von Einzelaufträgen Angaben, zwischen denen wir keine Reihenfolgebeziehungen mehr unterscheiden (siehe auch Kapitel 3.6). Lastprofile sind Abschätzungen über den typischen Ressourcenbedarf von Einzelaufträgen (Funktionsausführungen).

Die tatsächliche Last und Kommunikation steht erst zur Laufzeit fest (aktuelle Daten und Benutzereingaben sind maßgeblich), aber die Größenordnung läßt sich im voraus abschätzen. Dazu kann man, wie in der Komplexitätstheorie üblich, die Menge (Länge) der Eingabedaten betrachten oder die mittleren Kosten eines beispielhaften Programmlaufs (wahrscheinlichster Pfad) berechnen. In Datenbankanwendungen kann man anhand des Schemas der verwendeten Relationen die entstehende Last kalkulieren. Abweichungen werden Laufzeit festgestellt und die Balancierung kann entsprechend die statischen Entscheidungen korrigieren.

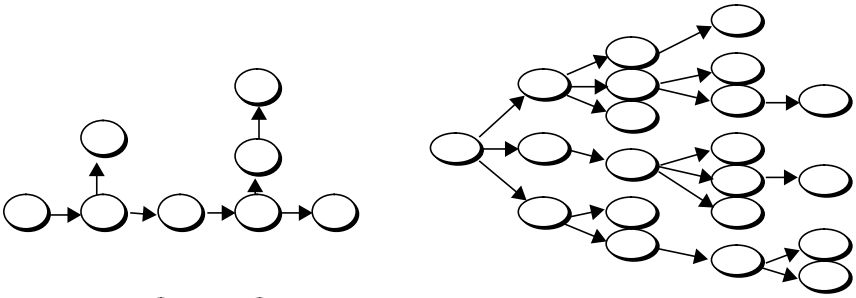
Entscheidend sind das Granulat, die Einheiten und die Genauigkeit (bzw. statistische Relevanz) der Profile. Die folgende Tabelle gibt einige Eigenschaften, die ein Lastprofil für einen einzelnen Auftrag enthalten kann (Attribute von Aufträgen, nicht von einzelnen Auftragsausführungen):

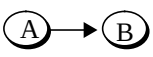
| **statische Eigenschaften** | **dynamische Eigenschaften** |
|---|---|
| mittlerer Rechenzeitbedarf, Varianz | Zahl und Ort der ausführenden Instanzen |
| maximal zulässige Laufzeit | Zahl und Frequenz der Ausführungen |
| Hauptspeicherbedarf | Zustände/Auslastung der ausführenden Instanzen |
| Menge der lokalen Daten | Zustand und Verteilung ihrer lokalen Daten |
| E/A-Zugriffe auf feste Geräte (+ Lese / Schreibverhältnis) | bisherige Datenzugriffe |
| Zugriffe auf feste Datensätze (+ Lese / Schreibverhältnis) | bisherige Kommunikation mit anderen |
| geforderte Verfügbarkeit | bisherige Unteraufrufe |
| Startzeitpunkte der Aufträge | mittlere Ausführungszeit |
| maximale Parallelität | genutzte Parallelität |

Die Kooperation zwischen einzelnen Teilaufträgen kann man, außer durch Reihenfolgeabhängigkeiten, auch durch Verzweigungen, Iterationen, Aufrufbeziehungen und Sessions in beliebig komplexer Form spezifizieren (bis hin zu eigenen Programmiersprachen). Über das Lastverhalten von **Auftragsgruppen** sind (neben den obigen Eigenschaften für Einzelaufträge) folgende Informationen wissenswert:

| **statische Eigenschaften** | **dynamische Eigenschaften** |
|---|---|
| Graph (möglicher) gegenseitiger Aufrufe | bisherige Kooperation (Menge, Frequenz) |
| Reihenfolgebeziehungen | |
| Geschwindigkeits-Abhängigkeiten | |
| synchrone / asynchrone Aufrufe | |
| Kommunikationsaufwand pro Aufruf | |
| Häufigkeit der Aufrufe | |
| Dauer/Kontextgröße von Sessions | |

Die Beschreibung der Auftragsbeziehungen enthält teilweise noch Probleme: Man kann komplexe prozedurale Abläufe (Schleifen, Verzweigungen, Rekursionen) kaum einfacher darstellen, ohne wesentliche Informationen zu verlieren (siehe etwa [Thomasian86] bzw. Kapitel 3.4). Die meisten Beschreibungsmodelle eignen sich für diskrete, grobkörnige Auftragsstrukturen (langlaufende, sequentielle Funktionen, links im Bild), weniger für rekursive, stark verzweigende Aufträge (rechts im Bild). Letztere lassen sich eher durch die maximale Parallelität, den Aufspaltungsgrad und Periodizität charakterisieren.



Bedeutung: 

‘A ruft B auf’ (oder ‘A aktiviert B’)

## 3.5 Verarbeitungsmodell

Die Lastbalancierung basiert auf einem bestimmten Modell der Auftragsbearbeitung auf dem System. Wir betrachten hier lediglich ein für heutige Systeme typisches Verarbeitungsmodell:

- Auf einem Prozessor können mehrere Funktionen bereitstehen und auch quasi-parallel, d.h. im Zeitscheibenwechsel nach Prioritäten gewichtet oder durch Wechsel bei I/O-Wartezeiten, ablaufen. Das entspricht nicht dem klassischen Warteschlangenmodell ([Thomasian86] bzw. Kapitel 3.4), da unter Einhaltung der Reihenfolgebeziehungen alle Aufträge gleichzeitig bearbeitet werden.
- Laufende Funktionsausführungen kann man nicht vom derzeitigen auf einen anderen Prozessor verlagern, sondern muß sie komplett auf einem Prozessor abwickeln. Man kann aber laufende Funktionsausführungen vorübergehend stoppen und später fortsetzen (*local preemptive scheduling*).
- Funktionen können auf Prozessoren bereit stehen und benötigen keine Ressourcen (zumindest keine Rechenzeit, eventuell Hauptspeicherplatz), solange sie keinen Auftrag durchführen.

## 3.6 Bearbeitungszeit von Aufträgen

Um die Zeit auszurechnen, die ein gegebener Satz von Aufträgen (Teile eines Gesamtauftrages) auf einem bestimmten System benötigt, muß man einiges beachten. Ein einfacher Ansatz (*bottleneck path*) besteht darin, pro Ressource alle Auftrags-Anforderungen zu summieren und damit die erforderliche Zeit auszurechnen. Die Bearbeitungszeit ergibt sich dann als Maximum dieser Zeiten:

$$\text{Bearbeitungszeit} = \text{MAX}_{\text{Resourcen } r}(\text{Kosten}[r] * \sum_{\text{Aufträge } a}(\text{Bedarf}[a, r])) \qquad (\textbf{Kostenmodell 1})$$

Das setzt aber implizit beliebig feinkörnige Parallelität der Aufträge und das Fehlen jeglicher Abhängigkeiten zwischen Aufträgen und zwischen den Ressource-Bedürfnissen eines Auftrags voraus. Wir wollen daher ein System mit den oben beschriebenen Modellen für Ressourcen, Aufträge (Funktionsausführungen) und Verarbeitung betrachten:

- Die Prozessoren, Geräte und Kanäle sind charakterisiert durch ihre Zeitkosten, d.h. die Kehrwerte der Leistungen (Prozessorkosten in sec/Mill.Instruktionen, Gerätekosten in sec/Block, Kanalkosten in sec/Nachricht). Geräte sind fest jeweils einem Prozessor zugeordnet, ebenso sind die Verbindungskanäle zwischen den Prozessoren festgelegt.
- Funktionen sind charakterisiert durch ihre Ressourcenbedürfnisse, d.h. den Rechenzeitbedarf (Mill.Instruktionen), Gerätebedarf (Blöcke) und Kooperationsbedarf (Nachrichten).
- Als Auslöser einer Menge von Funktionsausführungen wird eine Funktionsausführung (Auftrag) angefordert.
- Schließlich muß festgelegt werden, welche Funktionsausführung auf welchem Prozessor abläuft und wo sich die Daten befinden (das ist eine der Aufgaben statischer oder dynamischer Lastbalancierung).
- Sollen mehrere Instanzen zur Ausführung einer Funktion eingesetzt werden, so müßte der Synchronisationsaufwand zwischen den Instanzen berücksichtigt werden, um ihren gemeinsamen Kontext (Zustand und Daten) konsistent zu erhalten. Wir vernachlässigen es an dieser Stelle, da sich ein entsprechendes Kostenmodell derzeit noch in Entwicklung befindet.
- Die Hauptspeichergröße wird hier nicht beachtet. Durch die virtuelle Speicherverwaltung können erhebliche Verzögerungen entstehen.
- Die verschiedenen Ressourcen-Bedürfnisse eines Einzelauftrags werden über dessen gesamte Laufzeit ‘verschmiert’. Er hat also zu Beginn alle Bedürfnisse, nach einer gewissen Zeit noch jeweils 80% der anfänglichen Bedürfnisse und irgendwann hat er genau alle Anforderungen erhalten. Man kann also nicht etwa zuerst alle Plattenzugriffe, dann alle Rechenzeit und schließlich alle Kommunikation eines Auftrags erledigen. Das ist notwendig, da wir Reihenfolgebeziehungen innerhalb einzelner Aufträge nicht weiter auflösen.
- Aus demselben Grunde sind die Zeitpunkte, zu denen Unteraufrufe (Kooperation) getätigt werden, gleichmäßig über die gesamte Bearbeitungszeit des Auftrages verteilt. Die Ausführungen eines Auftrags teilen die Unteraufrufe gleichmäßig untereinander auf. Wir schränken hier Kooperation auf Unterfunktionsaufrufe ein (siehe Kapitel 7.1).

- Wir nehmen an, daß genügend Funktionen zur parallelen Ausführung bereitstehen (also überall beliebig viele Serverinstanzen warten). Funktionsinstanzen werden nicht explizit modelliert. Auch das soll in einem späteren Kostenmodell berücksichtigt werden.

Unter diesen Vorgaben berechnet der folgende Algorithmus die Gesamtausführungszeit der Aufträge. Wir betrachten jeweils kurze Zeitabschnitte t, in denen die Last als feststehend angenommen wird:

| wiederhole bis alle Ausführungen erledigt sind: (je ein Zeitschritt) | |
|---|---|
| | wiederhole einige Male: (je eine Iteration) |
| | die Ressourcen teilen ihre in diesem Zeitraum noch übrige Leistung unter die anfordernden Aufträge auf (gleichmäßig bzw. bei Prozessoren nach Prioritäten) |
| | die Aufträge verbrauchen von den angebotenen Leistungen soviel, wie das schlechteste Angebot einer Ressource erlaubt (wegen der ‘verschmierten’ Bearbeitung, siehe oben) |
| | es bleiben Ressource-Leistungen übrig, die in weiteren Iterationen noch verteilt werden können |
| | die Ausführungen werden fortgesetzt entsprechend der Ressourcen, die sie erhalten haben<br>dabei werden anstehende Unteraufrufe angestoßen |

Die Bearbeitungszeit ergibt sich dann aus der Zahl der Zeitschritte:

Bearbeitungszeit = Anzahl_Zeitschritte * Zeitschrittdauer (**Kostenmodell 2**)

Diese Berechnungsvorschrift wurde auch als Simulationsprogramm realisiert (siehe Kapitel 7.4). Bei [Ferrari86] findet man eine Untersuchung geeigneter Meßgrößen zur Lastbalancierung; siehe dazu Kapitel 6.8.

# 4 Organisation und Techniken der Lastbalancierung

## 4.1 Ebenen der Lastbalancierung

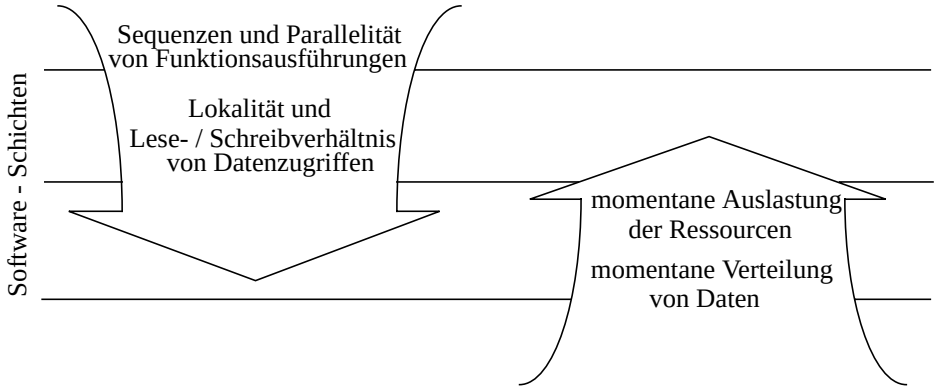
Schichtenmodelle ermöglichen es, immer komplexere Anwendungen zu realisieren. Eine Schicht stellt eine bestimmte Funktionalität zur Verfügung, auf der man aufbaut, ohne sich um die Realisierung des darunterliegenden zu kümmern. Wichtige Beispiele sind die Netzwerkprotokolle, das *Network File System*, die virtuelle Speicherverwaltung und die höheren Programmiersprachen. Der Programmierer erhält auf höheren Ebenen ein immer einfacheres Modell des Systems, kümmert sich immer weniger um Lagerorte und Struktur der Daten, Ausführungsorte und Arbeitsweise von Funktionen oder Fehlerbehandlung.

Für die Lastbalancierung werden solche Schichtenarchitekturen zum Problem, wenn zwischen den einzelnen Schichten keine Informationen über das Lastverhalten ausgetauscht werden können. So sollte beispielsweise ein Lastbalancierer, der über den Ausführungsort von Funktionen entscheidet, wissen, wo die zur Durchführung der Funktionen notwendigen Daten liegen. Wenn er auf einer Schicht basiert, die verteilte Daten auf transparente Weise lokal zur Verfügung stellt, so kann er nicht wissen, wieviel Sekundärlast eine Funktionsausführung auf anderen Knoten verursacht, um an die entfernt liegenden Daten zu kommen.

Umgekehrt können Lastbalancierungsmechanismen auf unteren Ebenen aufgrund einzelner isolierter Anforderungen schlecht wirken, ohne den größeren Zusammenhang zu kennen. Zum Beispiel benötigt man, um zu entscheiden ob sich eine lokale Datenkopie lohnt, eine Vorabinformation über das weitere Zugriffsverhalten auf diese Daten.

Im Prinzip wäre eine ‘flache’ Lastbalancierung optimal, d.h. ein Algorithmus, der über alle Informationen verfügt und alle Entscheidungen trifft. Nur ist ein solcher Ansatz so komplex, daß er weder in kurzer Zeit Entscheidungen treffen könnte (siehe Kapitel 5.7) noch für Programmierer überschaubar wäre. Daher erscheint es sinnvoll, Lastbalan-

cierung gemeinsam mit den Softwareebenen zu schichten, wobei zwischen den Ebenen Informationen ausgetauscht werden müssen (siehe Bild).




## 4.2 Statische und dynamische Lastbalancierung

**Statische Lastbalancierung** geschieht bei der Übersetzung einer Anwendung oder unmittelbar vor dem Start der Anwendung. Der statische Lastbalancierer betrachtet die auszuführenden Aufträge und die dazu notwendigen Ressourcen. Auf der anderen Seite betrachtet er die ihm zur Verfügung stehende Hardware. Nun verteilt er Aufträge und Daten derart, daß die mittlere Ausführungsdauer der Aufträge minimal wird. Er kann entweder Funktionen und Daten an feste Orte (Prozessoren, Platten) binden oder einen Plan erstellen, welche Funktionsaufrufe wo durchgeführt und wann sich die Daten an welchen Orten befinden sollen. Neben der Plazierung der Aufträge und Daten kann er auch die Reihenfolgen bzw. Zeitpunkte der einzelnen Auftragsbearbeitungen festlegen (freilich im Rahmen der vom Programmierer festgelegten Reihenfolge-Beziehungen, d.h. er fügt nur zusätzliche Beziehungen ein).

Zur Plazierung von Funktionsausführungen gehört die Bereitstellung replizierter Funktionsserver in geeigneter Anzahl; die Plazierung der Daten beinhaltet geeignete Partitionierung und Replikation von Datenobjekten.

Wenn statische Lastbalancierung lediglich zur *Vorbereitung der dynamischen Lastbalancierung* dient, so wird der statische Balancierer wenige Alternativen bestimmen, zwischen denen der dynamische Balancierer zur Laufzeit wählen kann. Außerdem kann er die kritischen Größen festlegen, die der dynamische Balancierer beobachten und zur Entscheidung heranziehen soll. Eine andere Möglichkeit besteht darin, daß der statische Balancierer eine feste Plazierung und einen festen Ablaufplan bestimmt, den der dynamische Balancierer zur Laufzeit modifiziert (auf die tatsächliche Situation hin optimiert).

**Dynamische Lastbalancierung** findet zur Laufzeit der Anwendungen statt. Der Balancierer bekommt zur Laufzeit neue Aufträge oder Auftragsgruppen. Diese Funktionsaufrufe weist er, der Situation angepaßt, geeigneten Serverinstanzen zu. Dazu muß er über den aktuellen Zustand seiner Ressourcen und Serverinstanzen informiert sein. Ein dynamischer Balancierer kann auch die Plazierung bzw. Partitionierung von Daten und Serverinstanzen ändern, wenn dies sinnvoll ist, oder sogar laufende Funktionsausführungen auf andere Prozessoren verlegen (*migrieren*).

## 4.3 Zentrale und dezentrale Lastbalancierung

Für statische Lastbalancierung ist es unwesentlich, ob sie von einer zentralen Komponente durchgeführt wird oder auf mehrere verteilt ist. Der Balancierer muß das gesamte System überblicken und möglichst große Gruppen von Aufträgen im Zusammenhang einplanen. Der Geschwindigkeitsgewinn (der Balancierung selbst, nicht der Auftragsdurchführung) durch physische Aufteilung ist gewöhnlich nicht gefragt.

Im dynamischen Fall hat entstehen bei der zentralen Lastbalancierung folgende Probleme:

- Der Balancierer verbraucht selbst einen Teil der Ressourcen, die ihm eigentlich für die Durchführung der Aufträge zur Verfügung stehen.
- Die Zeit, die verstreicht bis er für einen Auftrag den geeigneten Ort berechnet hat, zählt bereits mit zur Ausführungsdauer.

- Es ist sehr aufwendig, ständig die aktuellen Informationen übers Gesamtsystem an eine Stelle zusammenzutragen.
- Der zentrale Balancierer wird zum Engpaß, wenn sehr viele Aufträge kommen, da alles durch seine Hände gehen muß.

Daher ist es in sehr großen Systemen und in Anwendungen mit relativ feinkörnigen Aufträgen notwendig, die Aufgabe der Lastbalancierung zumindest physisch, wenn nicht sogar logisch über die Prozessoren zu verteilen.

**Physisch dezentrale Lastbalancierung**: Aufträge werden von einer lokalen Komponente entgegengenommen. Diese abstrahiert den Auftrag sowie ihren lokalen Systemzustand (Ressourcenbelastung) und gibt das an die ihr übergeordneten Komponente. Der Balancierer an der Spitze der Hierarchie entscheidet nun über die Plazierung des Auftrags oder er entscheidet eine abstrakte Plazierung, die dann auf dem Abwärtsweg durch die Balancierer-Hierarchie präzisiert wird.

Diese Struktur hat den Vorteil, daß nicht alle Informationen und Aufträge in jeder Einzelheit zur zentralen Komponente gebracht werden brauchen, sondern lokal vorverarbeitet werden. Durch die letztlich globale Entscheidung hat man immer noch eine systemglobale Lastbalancierung, was prinzipiell optimal ist (gäbe es nicht die oben erwähnten Probleme).

**Logisch dezentrale Lastbalancierung**: Aufträge werden lokal entgegengenommen und möglichst lokal verteilt. Die Lastbalancierer kooperieren auch hier (hierarchisch oder auf einer Ebene), d.h. tauschen Informationen über lokale Systemzustände aus. Ein Balancierer kann auch Aufträge an Partner abgeben oder ihnen Last abnehmen.

In größeren Systemen hat Lastbalancierung notwendigerweise diese Struktur; ohne weitgehend dezentrale Überwachung und Entscheidung würden Engpässe entstehen. Man verzichtet dabei allerdings (durch lokale Entscheidungen und abstrakte Systemsicht) auf global optimale Lastbalancierung, die offensichtlich ab einer gewissen Systemgröße nicht mehr möglich ist.

## Heuristische Verfahren

Da die allgemeine Lastbalancierung ein komplexes Problem ist (sie ist NP-vollständig, d.h. der Aufwand steigt exponentiell mit der Zahl der Elemente wie Ressourcen und Aufträge), werden oft vereinfachte Verfahren entwickelt. Sie beschränken sich auf spezielle Hardwarestrukturen oder Anwendungsklassen, betrachten nur eine kleine Auswahl relevanter Lastinformationen oder nehmen nur einen kleinen Teil der Funktionen wahr, die eine Lastbalancierung durchführen könnte (siehe Kapitel 2.1): man hat nur Aufträge ohne Reihenfolgebeziehungen, man vernachlässigt die Orte von Daten, vernachlässigt den Kommunikationsaufwand, besitzt nur Prozessoren gleicher Leistung, vernachlässigt Beziehungen zwischen Aufträgen (Sekundärlast, Unteraufrufe, Kooperation) oder nimmt für alle Aufträge gleichen Ressourcenbedarf an. In Kapitel 6 werden einige solcher Verfahren vorgestellt.

Anwendungsklassen mit charakteristischem Verhalten der Aufträge sind etwa Datenbanksysteme, numerische Berechnungen, graphische Applikationen oder Echtzeitprobleme. Auch Such- und Sortieralgorithmen weisen bestimmte Lastmuster auf. Allgemein unterscheiden sich die Anwendungsklassen durch das Granulat der Parallelität und anhand des Lastschwerpunktes (*bottleneck resource*), der auf der Rechenzeit, auf den Datenzugriffen oder auf der Kommunikation liegt (*CPU bound, Disk bound*). Manche Anwendungsbereiche weisen reguläre, vorhersehbare Verhaltensmuster auf, andere lassen sich lediglich statistisch erfassen.

# 5 Probleme der Lastbalancierung

Wir betrachten typische Schwierigkeiten, mit denen Lastbalancierer konfrontiert werden.

## 5.1 Vorhersage von Aufträgen

Der Lastbalancierer benutzt bei der Einplanung von Aufträgen Annahmen über deren Laufzeitverhalten. Dazu zählen Größen wie Rechenzeitbedarf und weitere Last (Sekundärlast), welche die Aufträge induzieren. Differenzen zwischen den Prognosen und dem tatsächlichen Lastverhalten bewirken, daß die Lastbalancierung nur noch statistisch gute Resultate ergibt.

Da man meist nur sehr grobe statische Bedarfsabschätzungen machen kann, ist dynamische Lastbalancierung sehr wichtig. Diese kann zur Laufzeit auf das tatsächliche Verhalten der Aufträge reagieren und die Lastverteilung korrigieren oder zumindest die Plazierung weiterer Aufträge auf die momentane reale Situation anpassen.

## 5.2 Schnelle Schwankungen der Systemlast

Wenn größere Aufträge zu Ende gehen, viele neue Aufträge starten oder einige Aufträge ihr Lastverhalten ändern (etwa Wechsel zwischen E/A-Phase und Rechenphase), so sind die Aufträge auf dem bisher gut balancierten System normalerweise jetzt ungünstig verteilt. Darauf muß ein Lastbalancierer reagieren:

- Wenn er laufende Aufträge migrieren kann, so sollte er solche Aufträge, die wohl noch länger laufen werden, auf unbelastete Prozessoren verlegen. In manchen Fällen kann er auch laufende Ausführungen abbrechen und an besserer Stelle erneut starten. Dabei sind jedoch Konsistenzbedingungen zu beachten (eventuell das bisher berechnete rückgängig machen oder die Idempotenz des Auftrags prüfen).
- Bei feinkörnig parallelen Anwendungen gibt es keine allzu lang laufenden Aufträge. Der Lastbalancierer braucht nur weitere Aufträge der neuen Situation angepaßt zu verteilen, dann wird sich die Situation innerhalb kurzer Zeit entschärfen. Kurze Lastspitzen sind ohnehin kaum vermeidbar.
- Wenn der Balancierer mit statistischen Daten arbeitet oder die Auftragsprofile statistisch ermittelt, so kann er die Schwankungen aufzeichnen, um diese Aufträge beim nächsten Mal von vornherein richtig einzuschätzen.

## 5.3 Kurzfristige Systemüberlastung

Durch Ausfall von Knoten oder plötzlichen Auftragsansturm können einige Knoten überlastet werden. Zusätzlich zur hohen Last machen sich dann die häufigen Prozeßwechsel und die Aus- und Einlagerung von Speicherblöcken negativ bemerkbar. Die Zahl der Fehler und Programmabbrüche erhöht sich erfahrungsgemäß und erzeugt weitere Last in Form von Prozeßende-, Prozeßstart- und Recovery-Aktionen.

Ein Balancierer kann versuchen, unwichtige Aufträge vorübergehend zu stoppen und neu hinzukommende zurückzuweisen. Dabei ist zu beachten, daß dadurch nicht auch wichtige Aufträge lahmgelegt werden (durch Stoppen einer Funktion, auf die sie warten oder Stoppen einer Ausführung, die wichtige Daten gesperrt hält).

In solchen Situationen ist weniger die Vollbeschäftigung der Prozessoren, sondern mehr die Hauptspeicherausnutzung, Anzahl der Prozesse je Prozessor und die Priorität von Aufträgen für die Lastbalancierung maßgeblich.

## 5.4 Momentan freier Knoten wird im nächsten Augenblick überlastet

Ein Knoten, der zur Zeit als unterbeschäftigt angesehen wird, ist das bevorzugte Ziel von neuen Aufträgen, die ja nicht auf andere, stark belastete Knoten geladen werden sollen. Dadurch passiert es oft, daß er kurz darauf überlastet ist. Dies ist vor allem bei (logisch) dezentralen Verfahren der Lastbalancierung (siehe Kapitel 4.3) ein Problem.

Ein zentraler Lastbalancierer vermeidet so etwas, indem er, wenn er einem Prozessor einen Auftrag erteilt, sogleich die momentane Lastangabe dieses Prozessors um die neue (zu erwartende) Last erhöht. Dezentrale Verfahren können das Problem etwas mildern, indem sie nicht alle Aufträge zum am wenigsten belasteten Knoten schicken, sondern statistisch an alle Knoten Aufträge vergeben, wobei die Wahrscheinlichkeiten nach der jeweiligen Belastung der Knoten gewichtet sind (siehe [Hsu86]).

## 5.5 Notwendigkeit einer globalen Reorganisation

Dynamische Lastbalancierer passen neue Aufträge gemäß der momentanen Situation gut in das System ein. Was bisher läuft, wird so belassen, obwohl einige Aufträge ‘einst’ unter einer völlig anderen Situation eingeplant wurden. Oft könnte man, wenn man jetzt alles noch einmal erneut verteilen würde, eine viel bessere Systemnutzung erreichen. Der Lastbalancierer sollte also erkennen, was eine globale Reorganisation bringen würde und ob sich der Aufwand lohnt. Unter globaler Reorganisation versteht man etwa die Migration von Dateien, die Repartitionierung von Daten

oder die Umverteilung von Serverinstanzen. Es ist offensichtlich, daß solche Aktionen die Systemlast vorübergehend stark erhöhen und man hofft, daß die Verteilung hinterher auch wirklich besser ist.

Wenn Lastbalancierungsentscheidungen lokal getroffen werden (bei dezentraler Lastbalancierung, siehe Kapitel 4.3), so kann der lokale Balancierer einfach die Kontrolle an eine zentralere Instanz abgeben, die globale Aktionen zentral überblicken und durchführen kann. Das kann der lokale Balancierer tun, wenn die Last in seinem lokalen System einen gewissen Schwellwert übersteigt oder wenn er feststellt, daß er seit längerem gegenüber anderen (lokalen) Systemen sehr stark (bzw. sehr wenig) belastet ist.

## 5.6 Gegensatz Parallelarbeit und Kommunikationsbedarf

Ein Problem, warum man in parallelen Systemen keinen proportionalen Speedup erreicht, ist die mangelnde Parallelität in den Anwendungen. Das andere Problem ist die teure Kommunikation zwischen Prozessoren. Je feiner das Granulat der Parallelität, umso größer wird die Menge der Nachrichten, die zwischen den einzelnen Funktionen und damit oft zwischen verschiedenen Prozessoren ausgetauscht werden.

Der Lastbalancierer muß also den Kommunikationsaufwand bei der Verteilung berücksichtigen. Er kann zwar gewöhnlich das Prozeßgranulat nicht ändern, aber da Nachrichten zwischen Prozessen auf demselben Prozessor relativ billig sind, sollten Funktionen, die stark kooperieren, eventuell auf demselben Prozessor laufen (obwohl das keine echte Parallelität ergibt).

## 5.7 Overhead durch Lastbalancierung

Dynamische Lastbalancierer beanspruchen das System durch ihre häufigen Lastmessungen, durch ihre Kommunikation und durch ihre eigentlichen Balancierungsberechnungen. Die Lastbalancierer verständigen sich untereinander, sammeln Informationen von Lastmessungsprozessen und Funktionsaufrufe gehen den Umweg über einen Lastbalancierer zur Serverinstanz. Die Überlegungen des Lastbalancierers zur Verteilung der Aufträge brauchen nicht nur Rechenzeit, sondern zögern auch die eigentliche Funktionsbearbeitung hinaus.

Der Vorteil der Lastbalancierung wird dadurch wieder verringert, in ungünstigen Fällen können die Verarbeitungszeiten auch größer werden als im unbalancierten Ablauf. Daher muß bei dynamischer Lastbalancierung folgendes beachtet werden:

- Der Überwachungs- und Meßaufwand sollte so klein wie möglich sein; man muß die wenigen entscheidenden Daten und Größen im System herausfinden und sich auf diese beschränken.
- Die Lastbalancierung sollte möglichst dezentral abgewickelt werden. Dadurch verringert man einerseits den Nachrichtenverkehr, andererseits sind die Balancierungsalgorithmen schon wegen der Größe der lokalen Systeme einfacher und schneller.
- Der Aufwand für die Balancierung sollte zum Nutzen proportional sein. So lohnt sich für große, langlaufende Prozesse ein viel höherer Planungsaufwand als für feinkörnig parallele Aufträge. Bei gut verteilter, hoher Systemlast sollten die Lastbalancierer wenig tun (und damit wenig stören); Planungen können dagegen auf wenig belasteten Knoten durchgeführt werden.
- Die Meßperioden und Zyklen für Umstrukturierungen der Lastbalancierer müssen gut mit den Lastwechselzyklen der Aufträge übereinstimmen, damit nicht zu viel gemessen und umverteilt wird, aber noch rechtzeitig auf Belastungswechsel reagiert werden kann.
- Optimale Zuweisungsalgorithmen sind NP-vollständig, haben also exponentielle Laufzeit. Daher müssen geeignete heuristische Verfahren angewandt werden, die annähernd optimale Lösungen liefern.

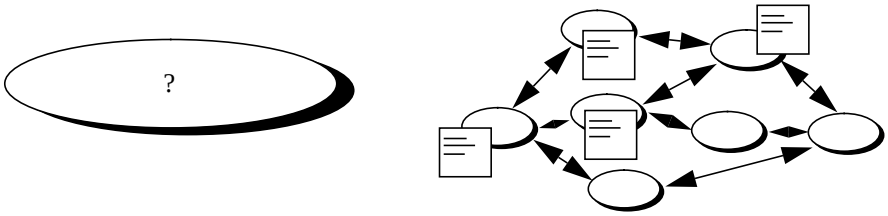
## 5.8 Einbindung in bestehende Systeme

Lastbalancierungsverfahren werden meistens in vorhandene Betriebssysteme oder Anwendungsprogramme eingebaut. Dabei fehlen geeignete Schnittstellen zur effizienten Messung und Regelung des Systems. Das Hauptproblem aber liegt darin, daß das Betriebssystem und die Software vielschichtig aufgebaut sind. Jede Schicht bietet eine

abstraktere Sicht auf das darunterliegende System. Auf höheren Schichten kümmert man sich unter anderem nicht mehr um die Lagerorte der Daten oder die Ausführungsorte von Funktionen. Das vereinfacht und vereinheitlicht die Programmierung sehr, aber man verliert den Überblick über die tatsächlichen Kosten der Datenzugriffe und Operationen. Beispiele sind das Network File System, die virtuelle Speicherverwaltung und Pufferverwaltung im allgemeinen (siehe auch Kapitel 4.1).

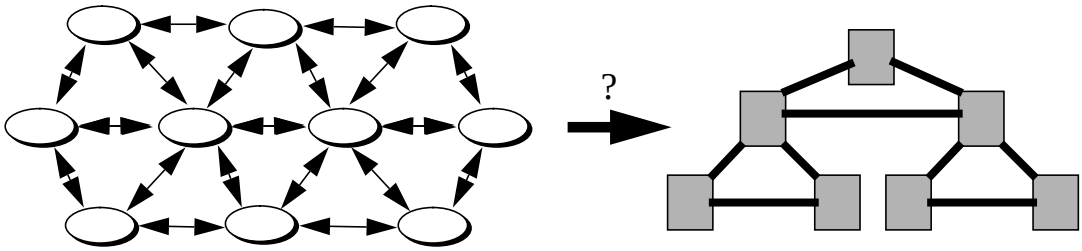
## 5.9 Geeignete Beschreibung paralleler Anwendungen

- Anwendungen, die als langlaufende, komplexe, sequentielle Prozesse realisiert sind, über deren Lastverhalten nichts bekannt ist, sind der Normalfall. Für die Lastbalancierung wären aber kurze, kleine Prozesse ideal, deren Kommunikation und externe Datenzugriffe explizit angegeben sind (siehe Bild). Die Lastprofile sollten das Granulat haben, mit dem die Lastbalancierung arbeitet. Größere Anwendungen sollten als Kette vieler kurzer Funktionen oder als Netz kooperierender Funktionen realisiert sein.




  Sequentielle Programme kann man teilweise durch parallelisierende Compiler aufgliedern, Lastprofile kann man durch Messungen von Testläufen erhalten oder vom Programmierer bzw. Benutzer abschätzen lassen. Gewöhnlich lassen sich sequentielle Anwendungen relativ gut statisch einplanen, sofern man ihren Ressourcenbedarf kennt.

- Auch das andere Extrem verursacht der Lastbalancierung Probleme: Anwendungen, die auf eine ganz spezielle Systemkonfiguration zugeschnitten sind und nur dort effizient ablaufen. Numerische Verfahren sind oft für eine Prozessor- oder Verbindungstopologie geeignet (im Bild ist eine finite-Elemente-Anwendung auf ein Mehrprozessorsystem zu verteilen). Wenn der Lastbalancierer sie aufgrund von Lasterwägungen anders verteilt als der Algorithmus voraussetzt, sinkt die Ablaufgeschwindigkeit sehr stark und der Kommunikationsbedarf wächst unnötig an.




  Man kann das Problem angehen, indem man solche Topologien explizit in Lastprofilen darstellt. Vorteilhaft ist es, gemeinsame Lastprofile jeweils für Gruppen kooperierender Prozesse zu erstellen. Solch komplexe Planungen sollten nur statisch durchgeführt werden, siehe etwa Vorschläge in [Bowen88, Lo88] bzw. Kapitel 6.2 und 6.3.

## 5.10 Veraltete Last- und Zustandsinformation

Wenn sich die Lastsituation auf dem System schnell ändert, dann erhalten die Lastbalancierer veraltete, falsche Daten über die Lastverteilung im System. Damit werden sie auch falsche Entscheidungen treffen. Andererseits würde eine schnelle und häufige Informationsverteilung das System so sehr abbremsen, daß sich die Lastbalancierung nicht mehr lohnt (siehe Kapitel 5.2). Wie balanciert man große, sich schnell ändernde Systeme?

- Die Lastbalancierungsentscheidungen müssen möglichst lokal getroffen werden, ohne globale Informationen oder eine zentrale Planungsinstanz einzubeziehen. Dazu braucht man eine geeignete, auf das System angepaßte Hierarchie an Balancierungskomponenten.

- Lastinformationen müssen auf ein Minimum reduziert werden und möglichst gebündelt und zu günstiger Zeit übers Netz ausgetauscht werden, um den Ablauf der Anwendungen nicht zu bremsen. Die Länge der Meßintervalle sollte in derselben Größenordnung sein wie die Wechselperioden der Last im System (d.h. proportional zum Auftragsgranulat).
- Durch geeignete statische Prognosen oder Anwendungswissen aus höheren Ebenen der Lastbalancierung kann man sich einige Messungen ersparen und Entscheidungen vorbereiten.

## 5.11 Häufige Zugriffe auf zentrale Datenobjekte

Viele Funktionen arbeiten mit einem Datenobjekt, auf das sie alle oft ändernd zugreifen (*Hot Spot* Daten). Beispiele sind Sperrentabellen und Protokolldateien. Wenn man das Datenobjekt auf die vorhandenen Ressourcen verteilt, gewinnt man meist keine Geschwindigkeit, denn die Konsistenzbedingungen der Daten erfordern eine zentrale Synchronisation und die Übereinstimmung aller Kopien. Die Verwaltung verteilter Kopien lohnt sich ja nur dann, wenn relativ viel lesend zugegriffen wird oder die Zugriffe so rechenintensiv sind, daß die Kosten für den Sperrenerwerb und das verteilen der Änderungen durch den Gewinn der verteilten Berechnung aufgefangen werden.

Hier kann die Lastbalancierung wenig helfen. Das Problem sollte auf Anwendungsebene gelöst werden, indem entweder die zentralen Daten feiner aufgegliedert werden, die Konsistenzanforderungen aufgeweicht werden oder spezielle Sperrstrategien angewandt werden (siehe dazu [Peinl88]). Ein Lastbalancierer kann höchstens entscheiden, inwiefern sich eine Replikation solcher Daten lohnt. Die Funktionen, die solch zentrale Zugriffe ausführen, sollten auf jeden Fall mit hoher Priorität bearbeitet werden ([Borr90] beschreibt das *Client-Server Priority Inversion* Problem, siehe Kapitel 6.10).

# 6 Existierende Ansätze zur Lastbalancierung

Wir sehen uns einige interessante Ansätze zur Lastbalancierung aus der Literatur an. Dabei versuchen wir, sie anhand folgender Kriterien zu charakterisieren:

- Sind die vorgeschlagenen Strategien statisch oder dynamisch; sind sie zentral oder verteilt?
- Welches Granulat an Parallelität wird betrachtet, mit welcher Art von parallelen Systemen wird gearbeitet?
- Welche Lastinformationen werden berücksichtigt (Prozessorlast, Ressourcen-Belastung, Kommunikation, Reihenfolgebeziehungen, einzelne Lastprofile / Profile von Auftragsgruppen, Datenobjekte oder Serverklassen)?
- Welche Aufgaben bzw. Fähigkeiten hat die Lastbalancierung (Auftragszuweisung, Prozeßmigration, Ressource-Migration, Replikation von Funktionen / Daten, Daten-Partitionierung)?

## 6.1 Preemptive Scheduling

Strategie: statisches Verfahren, parallelisiert.

Granulat: nicht festgelegt; nur Simulation.

Lastinformation: statisch vorgegebene Rechenzeitanforderungen und Prozessorleistungen.

Aufgaben: Plazierung, Stoppen, Migration und Fortsetzung von Aufträgen.

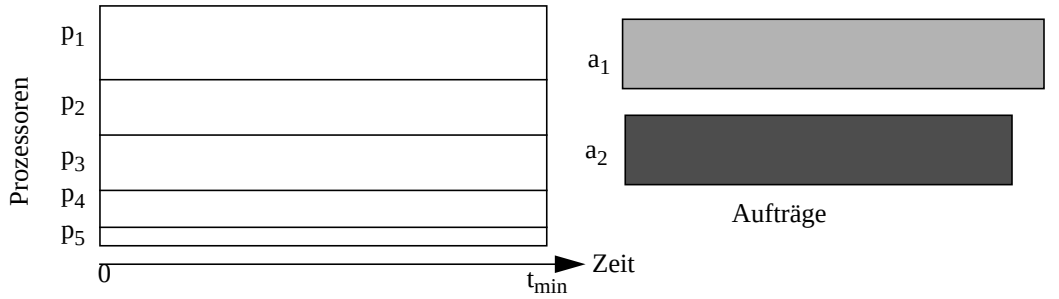
[Gonzalez78] beschreibt einen Algorithmus, der eine Anzahl von Aufträgen mit vorgegebenen Rechenzeitbedürfnissen auf eine Menge von gleichartigen Prozessoren so verteilt, daß die möglichst schnell abgearbeitet werden. Dieser Algorithmus wird in [Martel88] parallelisiert.

Gegeben sind #a Aufträge mit den Rechenbedürfnissen $a_i$ und #p Prozessoren mit den Leistungen $p_i$. Die Aufträge sollen so verteilt werden, daß die Zeit, bis der letzte fertig ist, minimal wird. Dabei kann jeder Auftrag zerteilt werden und die Teile können irgendwann auf irgendwelchen Prozessoren ablaufen (*preemptive Scheduling*), nur nicht parallel. Das Stoppen, Migrieren und Fortsetzen von Aufträgen wird als kostenlos angenommen. Die Aufträge und Prozessoren seien absteigend sortiert ($a_1 \geq a_2 \geq ... \geq a_{\#a}$ und $p_1 \geq p_2 \geq ... \geq p_{\#p}$). $A_i = \Sigma_{j=1..i} a_j$ ist die Summe der $i$ größten

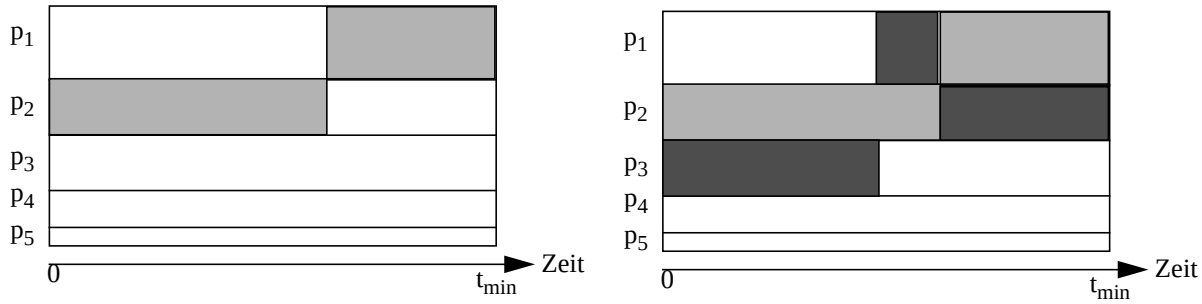
Anforderungen und $P_i = \Sigma_{j=1..i} p_j$ die Leistung der `i` schnellsten Prozessoren. Die minimale Gesamtlaufzeit beträgt dann

$$t_{min}=MAX(a_1/p_1,\ a_2/p_2,\ ...,\ a_{\#p-1}/p_{\#p-1}, a_{\#a}/p_{\#p}).$$

Wir betrachten nun das folgende Bild mit verfügbaren Rechenleistungen, in das die Aufträge eingeplant werden sollen.




Im ersten Schritt werden nun die `#p-1` großen Aufträge nach der Reihe eingeplant, jeder so, daß er maximal $t_{min}$ Laufzeit hat und möglichst langsame Prozessoren benutzt. Dazu setzt man den Auftrag links bei $p_1$ ein und schiebt ihn solange nach rechts (er rutscht dann in $p_2$ hinein, u.s.w.), bis er sich über $t_{min}$ erstreckt. Die folgenden Bilder zeigen die Einplanung der beiden größten Aufträge.




Im zweiten Schritt werden die restlichen Aufträge nach demselben Prinzip eingeplant. Der Algorithmus zur Plazierung der großen Aufträge ist von der Komplexität O(#p*log(#p)), zur Einplanung der Kleinen bedarf es O(#a). Der von [Martel88] auf #p Prozessoren parallelisierte Algorithmus hat eine Zeitkomplexität der Größe $O(\#p*\log^3(\#p))$ und O(#a/#p*log(#a)).

## 6.2 Statische hierarchische Auftragsverteilung

Strategie: statisches, zentrales Verfahren.

Granulat: parallele Prozesse.

Lastinformation: Kommunikationsaufwand pro Prozeßlauf und Häufigkeit des Prozeßablaufs wird zu einer Kenngröße gemischt. Die vermutlich anfallende Prozessorlast je Prozeß wird zur Einhaltung der Mindest- und Höchstlastgrenzen der Prozessoren benutzt.

Aufgaben: Die Prozesse werden auf die Prozessoren verteilt.

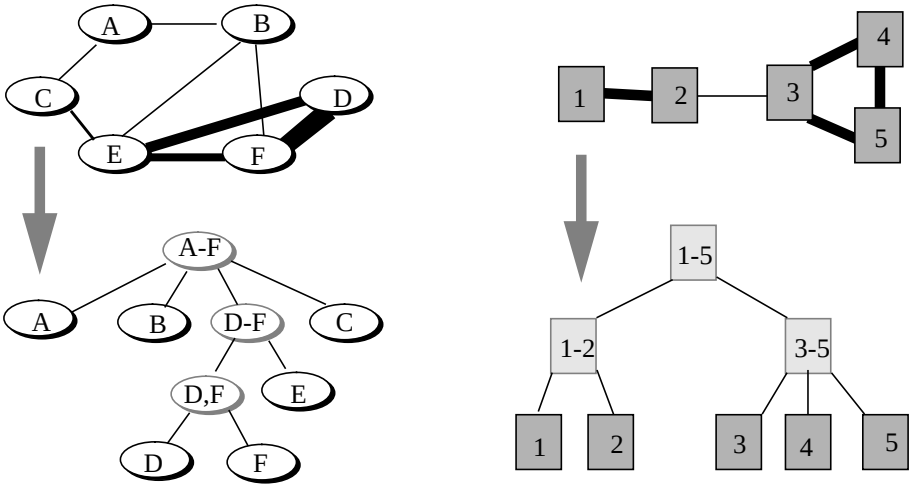
[Bowen88] beschreibt ein statisches Lastbalancierungsverfahren, das eine Gruppe von Aufträgen unter Berücksichtigung der Kommunikation zwischen den Aufträgen auf ein System verteilt. Dabei soll der Kommunikationsgraph der Aufträge möglichst gut auf den Verbindungsgraph der Prozessoren angepaßt werden. Man erreicht, daß Aufträge, die stark kooperieren, über schnelle Kommunikationskanäle verbunden sind.

Man sucht also die Verteilung `assign[a]` für die Aufträge `a` auf Prozessoren, bei der die Gesamtkommunikationskosten minimal sind:

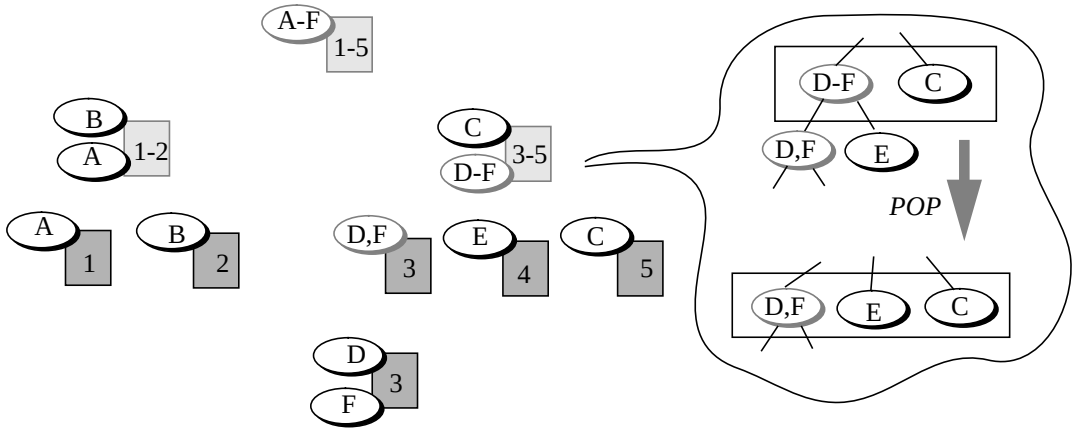
$$MIN_{assign}(\Sigma_{\text{Aufträge a}} \Sigma_{\text{Aufträge b}}(\text{Kommunikationsbedarf[a,b]} * \text{Kommunikationskosten[assign[a],assign[b]]})).$$

Als Randbedingung ist zu beachten, daß die Last auf einem Prozessor innerhalb gewisser Grenzen bleibt, damit kein Knoten überlastet wird und die Last nicht völlig ungleich verteilt ist. Man hat damit das Problem des *quadratic assignment*, das sehr aufwendig zu lösen ist. [Bowen88] schlägt deshalb folgenden heuristischen Algorithmus vor:

- Der Auftragsgraph (und völlig analog der Prozessorgraph) wird in einen Baum umgewandelt. Dabei sind Aufträge die Blätter; sie werden schrittweise durch Zwischenknoten zusammengefaßt, die dann wieder als ein Auftrag betrachtet werden. Für einen Schritt der Zusammenfassung nimmt man sich den Auftrag mit der dicksten Kante (d.h. dem größten Kommunikationsbedarf zu einem Nachbar) und faßt ihn mit den Aufträgen zusammen, mit denen er stark verbunden ist (eventuell noch mit deren Nachbarn, zu denen diese stark verbunden sind, u.s.w.). Dann berechnet man neue Kantenstärken zu den Nachbarn des zusammengefaßten Knotens. Das Bild zeigt, wie solch eine Umwandlung stattfinden könnte.



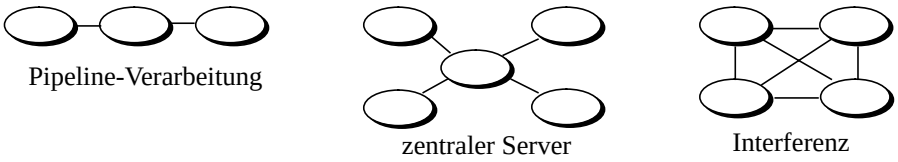

- Der Auftragsgraph wird auf den Prozessorgraph angepaßt. Dabei beginnt man mit dem Vergleich der Wurzeln und paßt dann rekursiv die Teilbäume einander an. In jedem Schritt ist eine Menge von Aufträgen auf eine Menge von Prozessoren zu verteilen:




Die Aufträge bzw. Prozessoren sind eventuell nur Zwischenknoten (d.h. keine Blätter), aber man kennt für jeden Auftragsknoten die Gesamtlast und für jeden Prozessorknoten das gesamte Lastminimum und -maximum sowie die bisher dort angehäufte Last. Man teilt nun dem Prozessor, der für seine Verhältnisse am wenigsten Last hat, den Auftrag zu, der am meisten Last beinhaltet. So verteilt man alle Aufträge (dieser Stufe) auf die Prozessoren (dieser Stufe). Die auf diese Weise an Prozessorgruppen verteilten Auftragsgruppen werden rekursiv genauer zugeordnet (auf der nächsten Stufe).

- Im Beispiel ist bei der Verteilung der Aufträge D-F und C auf die Prozessoren 3, 4 und 5 eine *POP*-Operation notwendig, da sonst einer der Prozessoren keinen Auftrag erhielte (und damit seine Minimallast nicht erreichen würde). Grundsätzlich wird, falls auf einer Stufe keine Zuweisung möglich ist, der Auftrag, der die meiste Last beinhaltet, aufgegliedert (*POP*); danach wird noch einmal eine Zuweisung versucht.

[Bowen88] stellt typische Auftragsgraphen vor, sie sind im Bild kurz erläutert. Mit Interferenz sind vor allem atomare Aktionen (z.B. Setzen und Freigabe von Sperren) gemeint, während deren Durchführung der Prozeß alle anderen anhält.




## 6.3 Statische Auftragszuweisung in Broadcast-Netzen

Strategie: statisches, zentrales Verfahren.

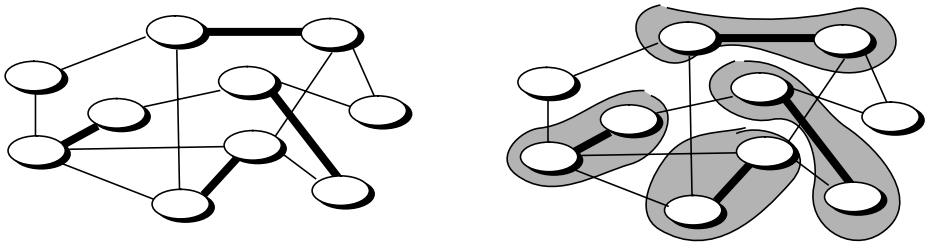
Granulat: parallele Prozesse, Kommunikation über Ethernet-Medium.

Lastinformation: Kommunikationsintensität zwischen Prozessen. Durch Beschränkung der Prozesse je Prozessor begrenzt man die Last.

Aufgaben: Die Prozesse werden zu Gruppen zusammengefaßt, die je auf irgendeinem Prozessor laufen können.

Die beiden Verfahren von [Lo88] plazieren Aufträge auf Prozessoren so, daß die Interprozeßkommunikation minimal wird. Hier wird ein Broadcast-Medium vorausgesetzt (z.B. Ethernet). Dadurch ist die Kommunikation zwischen allen Prozessoren gleich teuer und jede Nachricht zwischen zwei Prozessoren belastet das gesamte Verbindungsnetz. Das Problem beschränkt sich darauf, die Aufträge geeignet in Gruppen zusammenzufassen, sodaß die Summe aller Nachrichten zwischen allen Gruppen am geringsten ist. Diese Gruppen können dann beliebig auf je einen Prozessor verteilt werden. Der Lastbalancierung wird Rechnung getragen, indem man die Zahl der Prozesse je Prozessor begrenzt. Wir betrachten kurz die beiden Verfahren:

- Das erste Verfahren läuft in polynomieller Zeit und ergibt die optimale Lösung, solange kein Prozessor mehr als zwei Aufträge erhalten darf. Man konstruiert zuerst ein maximales *Matching* auf dem Graph der Aufträge, d.h. eine Menge von Kanten, die keinen Knoten gemeinsam haben (das maximale *Matching* ist die Kantenmenge, bei der die Summe der Kantengewichte am größten ist). [Lo88] nimmt alle Kommunikation als gleich an, d.h. hier enthält das maximale *Matching* am meisten Kanten. Das Bild (links) zeigt ein *Matching* (bestehend aus den breiten Kanten). Man packt nun die im *Matching* verbundenen Aufträge je auf einen Prozessor, wie im Bild (rechts) dargestellt. Die übrigen Aufträge verteilt man dann je zu zweit auf die freien Prozessoren.



- Der zweite Algorithmus verteilt in polynomieller Zeit auch mehr als zwei Aufträge je Prozessor, garantiert allerdings kein optimales Ergebnis. Der Auftragsgraph wird so weit zusammengeschrumpft (indem man mehrere Aufträge zu je einem Knoten zusammenfaßt), daß er nur noch aus zwei Aufträgen je Prozessor besteht; darauf kann der erste Algorithmus angesetzt werden.

  Beim Zusammenfassen sucht man immer wieder die Kante mit dem größten Gewicht und vereinigt die beiden Knoten (sofern sie zusammen nicht mehr Aufträge haben, als je Prozessor erlaubt ist). Das Kantengewicht zwischen zwei Auftragsgruppen ist die Summe der Einzelkantengewichte zwischen ihnen.

## 6.4 Ressourcen-Migration

Strategie: statisches, zentrales Verfahren.

Granulat: Datenbankanfragen (Queries).

Lastinformation: die Standorte der Aufträge, ihre Ressourcenzugriffe und die anfängliche Verteilung der Ressourcen sind bekannt. Beachtet wird Migrationsaufwand, Aufwand zur Konsistenthaltung von Kopien und Kommunikationsaufwand bei Remote-Zugriffen auf Ressourcen.

Aufgaben: Umverteilung der Ressourcen.

[Varadarajan88] stellt ein statisches Lastbalancierungsverfahren vor, das Ressourcen zu den Aufträgen migriert, die diese benötigen. Es migriert keine Aufträge, berücksichtigt dafür die Kosten der *Remote*-Zugriffe auf Ressourcen, die nicht auf ihren Prozessor gebracht wurden. Die Lastbalancierungsaufgabe besteht für [Varadarajan88] darin, die gesamten Migrationskosten zu minimieren, wobei die Ausführungszeit der Aufträge in einem gewissen Zeitlimit bleiben muß.

Als Anwendung wird die Migration von Dateien in Datenbankanwendungen betrachtet. Man will eine feste Anzahl von Kopien je Datei möglichst geschickt auf die Prozessoren verteilen. Der Aufwand zur Konsistenthaltung der Kopien wird implizit in den Migrationskosten mitberücksichtigt. Da man annimmt, daß Dateien zwischen verschiedenen Prozessorpaaren parallel migriert werden können, wird der Migrationsaufwand desjenigen Prozessorpaares minimiert, bei dem er maximal ist.

Da dieses Problem bereits NP-vollständig ist, wird ein heuristisches Verfahren vorgestellt. Dabei werden nur Gruppen von Prozessoren betrachtet; Prozessoren, deren Aufträge ähnliche Ressourcenbedürfnisse haben, kommen jeweils zusammen in eine Gruppe. Dateien werden nur zwischen Gruppen migriert. Da die Aufträge einer Gruppe ähnliche Ressourcenbedürfnisse haben, geht man davon aus, daß *Remote*-Zugriffe auf Ressourcen innerhalb der Gruppe bleiben. Der Lastbalancierer soll nun die optimale Migrationsvorschrift für die Dateien finden, d.h.:

$$\text{minimiere } \mathtt{MAX}_{\mathtt{Gruppen\ g,\ h}}(\mathtt{Migrationskosten[g->h]}),$$

wobei in jeder Gruppe `a` gelten muß

$$\mathtt{Nachrichtenkosten[a]} + \frac{\mathtt{\#Aufträge[a]\ *\ Rechenzeit\_pro\_Auftrag}}{\mathtt{\#nach\_a\_zu\_migrierende\_Kopien}} \leq \mathtt{Zeitlimit}.$$

Dazu hat man Näherungsformeln für die Migrationskosten und die Nachrichtenkosten.

Die Simulationen ergeben bei geeigneter Gruppenbildung Ergebnisse, die sehr dicht an der Ideallösung liegen.

## 6.5 Vier Strategien im Vergleich

Strategie: statische und dynamische, zentrale Verfahren.

Granulat: kurze Berechnungen mit Pipeline-Struktur. Es werden Pipeline-Topologien betrachtet.

Lastinformation: die statischen Strategien kennen Reihenfolgebeziehungen und Rechenzeitbedarf. Die dynamische Strategie berücksichtigt einzelne Datenabhängigkeiten zwischen Teilschritten von Aufträgen und den momentanen Bearbeitungszustand der Aufträge.

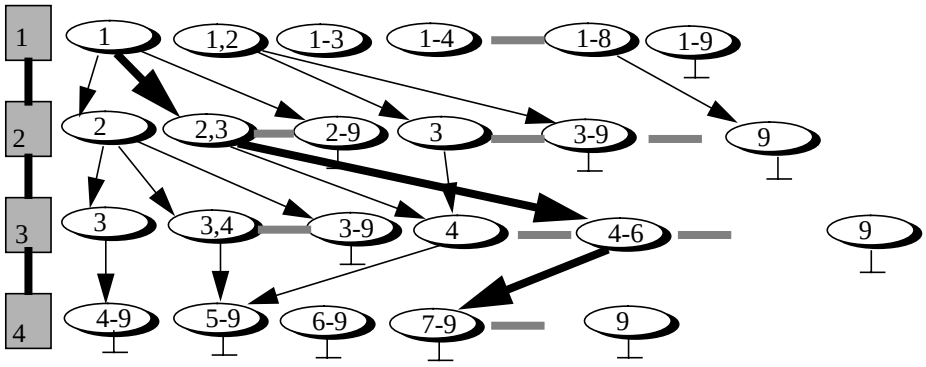
Aufgaben: Zuweisung von Aufträgen an Prozessoren. Das dynamische Verfahren kann laufende Aufträge umverteilen.

[Iqbal86] vergleicht durch Simulation drei statische Lastbalancierungsverfahren und ein dynamisches Verfahren, welches Reihenfolgebeziehungen zwischen Aufträgen ausnutzt. Dabei betrachtet man kettenartige Aufträge (Pipeline-Verarbeitung), die auf einer Kette von Prozessoren ablaufen sollen. Es folgen die Verfahren:

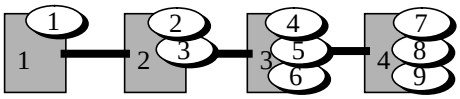
1. Der Algorithmus erzeugt eine optimale, statische Zuweisung einer Auftragskette auf eine Prozessorkette. Dazu werden Teilketten so auf je einen Prozessor gelegt, daß die Last des am stärksten beladenen Prozessors minimal wird. Dazu schreibt man alle Möglichkeiten der Auftragsverteilung neben die Prozessoren; man rechnet für jede

Auftragsgruppe auf einem Prozessor die erzeugte Prozessorlast aus. Dann sucht man die Kombination heraus, bei der die größte Prozessorlast minimal ist. Im Bild ist eine Kette von 9 Aufträgen auf eine Kette von 4 Prozessoren zu verteilen.

Reihenfolgebeziehungen der Aufträge




Die dick eingezeichnete Kombination bezeichnet diese Verteilung:




2. Diese Heuristik teilt die Auftragskette in zwei Teile auf, sodaß die Differenz zwischen den beiden Ausführungszeitsummen möglichst klein wird. Diese Hälften werden rekursiv weiter halbiert. Das Verfahren eignet sich für Systeme mit einer Kette von $2^n$ Prozessoren.

3. Man gibt sich eine Grenze für das maximale Ungleichgewicht `u` der Last zwischen zwei Prozessoren vor. Dann tastet man sich mit binärer Suche so nahe man will an das kleinstmögliche u heran, für das die Funktion *Probe* noch eine Zuweisung findet. Die Funktion läuft folgendermaßen ab:

```
function PROBE (a Aufträge, p Prozessoren, Ungleichgewicht u): boolean;
begin
  i= 1, j= 1
  for each processor:
     repeat j++ until j=p or Gewicht der Kette [i..j]>u
     if j=m then return TRUE (alle Prozesse sind zugewiesen)
     assign Kette [i..j] to current processor
     i= j;
  next processor
  return FALSE (es sind Prozesse übrig geblieben)
end
```

4. Die mögliche Arbeit soll bei allen Prozessoren etwa gleich sein. Wenn ein Prozessor (im Vergleich zu seinen Nachbarn) zuwenig mögliche Arbeit hat, so übernimmt er einen Auftrag von seinem höchstbeladenen Nachbarn. Der Lastunterschied muß so groß sein, daß sich die Migration des Auftrages lohnt. Die mögliche Arbeit eines Prozessors ist die Zeit, die er braucht um all seine Aufträge soweit zu erledigen, wie momentan die Daten dazu verfügbar sind.

   Zur Bestimmung der möglichen Arbeit eines Prozessors hat man die Aufträge jeweils in Schritte unterteilt. Zu jedem Schritt kennt man die Daten, die er benötigt. Er kann ausgeführt werden, sobald die Schritte der anderen Aufträge, die diese Daten produzieren, beendet sind. So kann man aufsummieren, wieviele Schritte der Prozessor in seinen Aufträgen jetzt bearbeiten kann. Der Wert muß jedesmal, wenn Daten ankommen oder ein Auftrag verschoben wurde, aktualisiert werden.

   Die Simulationsergebnisse zeigen, daß die dynamische Strategie wesentlich erfolgreicher arbeitet, wenn sie auf einer statisch lastbalancierten Zuordnung startet. Man kann sie als Verfeinerung der statischen Strategien einsetzen.

## 6.6 Warteschlangenmodelle

Strategie: dynamische, zentrale Verfahren können eingebaut werden.

Granulat: nicht festgelegt, reine Simulation.

Lastinformation: Rechenzeitbedarf, Bedarf an passiven Ressourcen, Kommunikationsmenge und Reihenfolgebeziehungen (deterministische und nichtdeterministische). Auftragslaufzeiten exponentialverteilt.

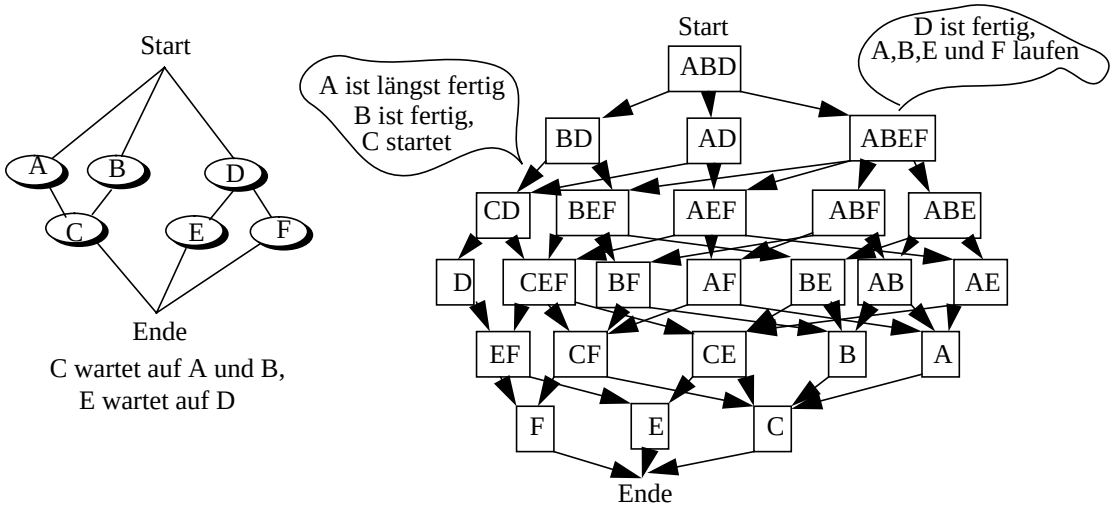
Aufgaben: Die Aufträge werden zur Laufzeit Prozessoren zugewiesen.

[Thomasian86] analysiert Antwortzeiten von Aufträgen mithilfe von Zustands-Wahrscheinlichkeiten. Das Computersystem wird durch die Geschwindigkeiten der Prozessoren, Kanäle und Platten beschrieben. Dazu spezifiziert man eine Menge von Aufträgen, die abzuarbeiten sind, mit ihren Ressourcenbedürfnissen. Die wirklich von bestimmten Geräten beanspruchte Zeit wird berechnet, sobald der Auftrag diesen Geräten zugewiesen wurde. Die Zuweisung führt ein *Scheduler* zur Laufzeit aus, dessen Strategie austauschbar ist.

In der Auftragsbeschreibung können Reihenfolgebeziehungen zwischen Aufträgen *probabilistisch* angegeben werden: zu jedem Auftrag gibt man eine Liste von Folgeaufträgen mit Wahrscheinlichkeiten an. Ist der Auftrag beendet, so werden alle Folgeaufträge, unter der jeweils angegebenen Wahrscheinlichkeit, gestartet. Daneben sind auch *deterministische* Reihenfolgebeziehungen erlaubt; man spezifiziert zum Auftrag eine Liste von Aufträgen, auf deren Vollendung er warten muß.

Der Simulator baut nun schrittweise (nicht in Zeitschritten!) einen Zustandsgraph (*Markov-Kette*) auf. In jedem Schritt berechnet er die möglichen Ablaufzustände samt der Wahrscheinlichkeit, mit der sie auftreten und der Zeit, um sie zu erreichen. Ein Ablaufzustand besteht einfach aus einer Menge von Aufträgen, die gerade bearbeitet werden. Aus den Zuständen eines Schrittes berechnet er die Zustände des nächsten Schrittes: die laufenden Aufträge werden mit exponentialverteilter Wahrscheinlichkeit fertig und starten dadurch Folgeaufträge.

Das Bild zeigt links die (deterministische) Reihenfolgeabhängigkeiten zwischen Tasks und rechts die Markovkette, die der Simulator daraus erzeugt. In den Zuständen stehen jeweils die Aufträge, die derzeit laufen




## 6.7 Dynamische Lastbalancierung in Datenbankanwendungen

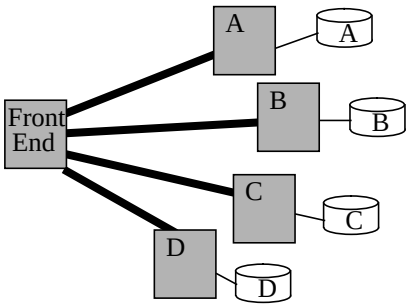
Strategie: dynamisches, zentrales Verfahren.

Granulat: Transaktionen auf Datenbankrechnern, eingeteilt in Ressourcenzugriffe.

Lastinformation: genaue statistische Lastprofile, die Rechenzeit und Datenbankzugriffe auf bestimmte Datenbanken enthalten. Last wird anhand der Kosten gemessen, welche die laufenden Transaktionen durch Rechenzeit und Kommunikation erzeugen.

Aufgaben: Zuweisung von Transaktionen an je einen Datenbankrechner.

[Yu86] stellt vier dynamische Strategien vor, die den Ausführungsort von Transaktionen in Datenbankanwendungen bestimmen. Die Transaktionen greifen dabei auf mehrere verteilte Datenbanken zu (siehe Bild). Ein Front-End Rechner nimmt Aufträge entgegen und weist sie je einem Datenbankrechner zu, der sie ausführt und das Ergebnis an den Front-End Rechner zurückgibt.




Die einzelnen Datenbankzugriffe werden jedoch immer von dem lokalen Datenbankrechner durchgeführt (unabhängig davon, welcher Rechner die Transaktion bearbeitet). Die vier Strategien funktionieren folgendermaßen:

1. Ein ankommender Auftrag wird an den Knoten geschickt, bei dem gerade am wenigsten Aufträge laufen. Dabei werden Aufträge, die auf I/O warten, nicht mitgerechnet, sondern nur solche, die derzeit Rechenleistung in Anspruch nehmen. Das Verfahren eignet sich gut, falls die Aufträge hauptsächlich lokal auf dem einen Prozessor abgearbeitet werden.

2. Diese Strategie schätzt für den ankommenden Auftrag die zu erwartenden Antwortzeiten der verschiedenen Knoten ab und sendet den Auftrag zu dem Knoten mit der kürzesten Antwortzeit. Eine Transaktionsverarbeitung wird hier modelliert durch ein Anwendungsprogramm, das hin und wieder Datenbankzugriffe tätigt. Die Datenbankzugriffe bestehen aus einem Teil Rechenzeit des lokalen Datenbankrechners und aus einem Teil Platten-Wartezeit. In diesem Modell kann man die Antwortzeit folgendermaßen abschätzten:

$$\text{Antwortzeit(Transaktionstyp k auf Knoten i)} = \frac{\text{Anwendungsbedarf(k)}}{\text{Prozessorleistung/Last(i)}} + \sum_{\text{DBs j}} \frac{\#\text{DB\_Zugriffe(k, j)}}{\text{Prozessorleistung/Last(i)}} * \begin{pmatrix} \text{lokale\_Zugriffskosten(k, j)} \\ + \text{Kommunikationskosten(k, i)} \\ + \text{Kommunikationskosten(k, j)} \end{pmatrix} + \sum_{\text{DBs j}} \#\text{DB\_Zugriffe(k, j)} * \text{I/O\_Wartezeit\_pro\_Zugriff}$$

   Dabei sind die Kommunikationskosten = 0, falls j=i, d.h. der Zugriff lokal geschieht.

   Nun hängt die Antwortzeit davon ab, welches maß man für die Last der Prozessoren verwendet. In der zweiten Strategie wird hier (wie in der ersten Strategie) die Zahl der laufenden Aufträge verwendet.

3. Die dritte und auch die vierte Strategie verwenden dieselbe, oben beschriebene, Formel zur Abschätzung der Antwortzeit. Das dritte Verfahren schätzt aber die momentane Last eines Prozessors anders ab: als Maß summiert man die diesem Prozessor von allen derzeit (irgendwo) laufenden Transaktionen drohende Arbeit auf. Analog zur obigen Formel summiert man den Anwendungsbedarf sowie den Aufwand der lokalen DB-Zugriffe auf für die Transaktionen, die lokal laufen; dazu addiert man die Kommunikationskosten und den Aufwand der DB-Zugriffe für alle Transaktionen, die anderswo ablaufen.

4. In der vierten Strategie wird (gegenüber der dritten) nicht die Arbeit aufsummiert, welche die derzeitigen Aufträge dem Prozessor wohl aufhalsen, sondern die Zeit, die diese Aufträge in dem einen Prozessor zubringen.

Die Simulationsergebnisse stellen die beiden letzten Strategien als am erfolgreichsten heraus; meist sind alle vier Strategien besser als eine optimale statische Zuweisung.

## 6.8 Meßgrößen für die Knotenbelastung

Strategie: dynamisches, zentrales Verfahren.

Granulat: UNIX-Prozesse.

Lastinformation: zu erwartende Antwortzeiten aufgrund momentaner Knotenlasten. Dabei werden verschiedene Auftragstypen und alle Ressourcen der Knoten berücksichtigt.

Aufgaben: Zuweisung eines neuen Auftrages an einen Knoten.

[Ferrari86] untersucht verschiedene Größen, die in der Lastbalancierung verwendet werden, um die Last der Knoten zu bestimmen. Als Grundelemente nennt er die Prozessor-Nutzung (*busy-time*), die Zahl der ausführbaren Prozesse (*Run Queue Length*) und den Streck-Faktor (Verhältnis der Ausführungszeit auf dem belasteten Knoten zur Zeit auf demselben Knoten, wenn er frei ist).

Dabei werden Aufträge nur für sich alleine betrachtet (keine Beziehungen zwischen Aufträgen). Als Maß für die Bearbeitungsgeschwindigkeit wird die Antwortzeit des Auftrags gewählt (und nicht der mittlere Durchsatz aller Aufträge). [Ferrari86] leitet ein Maß zur Abschätzung der Knotenlast her. Man gibt einen Auftrag A vom Typ Typ(A) vor und erfährt, *um wieviel die Antwortzeit für Auftrag* A (bei allen momentan auf dem Knoten laufenden Aufträgen B samt unserem neuen) *größer ist als die Antwortzeit für* A, *wenn der Knoten frei wäre*. Die Knotenlast hängt also davon ab, welchen Auftragstyp man starten möchte:

$$\texttt{Antwortzeit}(\text{A allein auf dem Knoten}) = \sum_{\text{Ressourcen r}} \begin{pmatrix} \#\text{Ressourcen\_Zugriffe}(\text{Typ(A)}, r) \\ * \text{ Bearbeitungszeit}(\text{Typ(A)}, r) \end{pmatrix}$$

$$\texttt{Antwortzeit}(\text{A und B auf dem Knoten}) = \sum_{\text{Ressourcen r}} \begin{pmatrix} \#\text{Ressourcen\_Zugriffe}(\text{Typ(A)}, r) \\ * \text{ Bearbeitungszeit}(\text{Typ(A)}, r) \\ * (\text{mittlere\_Schlangenlänge\_an\_Ressource}(r) + 1) \end{pmatrix}$$

Dabei wurden die mittleren Schlangenlängen im laufenden System gemessen, d.h. durch die Aufträge B verursacht.

Knotenlast(Typ(A)) = Antwortzeit(A und B auf dem Knoten) - Antwortzeit(A allein auf dem Knoten)

Die Ressourcen eines Knotens sind hauptsächlich der Prozessor (bzw. die Prozessoren bei Multiprozessorknoten), die Platten und die Kanäle. In der Praxis sind die meisten Anwendungen an eine Ressource gebunden (*CPU bound*, selten *Disk bound*), sodaß man nicht immer die Summe über alle Ressourcen der Knoten bilden muß. Wenn man auch noch die unterschiedlich großen Ressourcenbedürfnisse der Auftragstypen vernachlässigt, so erhält man die oft verwendete Zahl der laufenden Aufträge (*Run Queue Length*) als Maß für die Belastung eines Knotens.

Die Messungen bestätigen das, zeigen aber auch das Problem, daß die momentanen Schlangenlängen sich sehr schnell ändern und daher nicht besonders repräsentativ sind.

## 6.9 Lastbalancierung im PROSPECT-Projekt

Strategie: dynamisches, zentrales Verfahren.

Granulat: Server-Aufrufe. Es wird ein busgekoppeltes, *shared-nothing* Mehrprozessorsystem benutzt.

Lastinformation: momentane Prozessorlast, freie Serverinstanzen.

Aufgaben: Zuweisung eines Aufrufs an eine Serverinstanz.

Das *PROSPECT* Projekt [Reuter86, Duppel87, Duppel87b, Duppel88, Duppel88b, Duppel89, Duppel89b, Reuter90] befaßt sich mit der Organisation von transaktionsinterner Parallelität. Die Schwerpunkte liegen auf parallelen Join-Algorithmen, parallelen deduktiven Datenbanksystemen, Behandlung komplexer Objekte und der dynamischen Lastbalancierung. Als Beschreibungsmittel und Laufzeitumgebung paralleler Abläufe wurde ein *Scheduler* entwickelt. Anwendungen werden in Aktionen zerlegt und durch Ereignisse synchronisiert; der Scheduler bietet Ortstransparenz

sowie Parallelität durch Serverklassenverwaltung und Nachrichten-Routing. Für den Einsatz in großen Systemen kann der Scheduler in Form hierarchisch vernetzter, kooperierender Komponenten konfiguriert werden.

Lastbalancierung wird im *Scheduler* realisiert, indem er die Instanzen einer Serverklasse jeweils reihum benutzt oder indem er jeweils eine freie Instanz auf dem am wenigsten belasteten Prozessor wählt.

## 6.10 Lastbalancierung in lose gekoppelten Systemen

Strategie: dynamische Verfahren.

Granulat: Server-Aufrufe. Es wird ein busgekoppeltes, *shared-nothing* Mehrprozessorsystem benutzt.

Lastinformation: momentane Prozessorlast, freie Serverinstanzen.

Aufgaben: Zuweisung eines Aufrufs an eine Serverinstanz.

[Borr90] beschreibt die Probleme der Lastbalancierung von Systemen, deren Knoten keine gemeinsamen Ressourcen besitzen und die über Nachrichten kommunizieren. In solchen Systemen läßt sich keine gemeinsame *Run Queue* realisieren und die Migration laufender Prozesse ist sehr aufwendig. Wegen den Kosten für Zugriffe auf nicht-lokale Ressourcen ist es oft wichtiger, die Aufträge zu den Ressourcen zu legen, als die Prozessoren gleichmäßig zu beladen.

Da ein Prozeßstart sehr teuer ist, werden für die Aufträge Serverklassen (eine Serverklasse je Auftragstyp) bereitgestellt. Ein Aufruf besteht dann nur noch aus einer Nachricht an eine bereits wartende Instanz der Serverklasse. Die Zahl der Instanzen einer Serverklasse begrenzt allerdings die Zahl der parallel ablaufenden Aufträge dieses Typs.

Man stößt auf das *Client-Server Priority Inversion* Problem: ein Server, der einen stark benutzten Datensatz bedient, läuft mit sehr hoher Priorität, da alle anderen Server, die auf die Daten zugreifen, ständig auf ihn warten müssen. Ansonsten würde dieser Server zum Engpaß. Leider bedient er auch die Server, welche sehr niedrige Priorität besitzen, mit hoher Priorität und bremst dadurch andere wichtige Abläufe auf demselben Prozessor.

Als Ausweg schlägt [Borr90] vor, daß dieser Server unwichtige Zugriffe zurückstellt, wenn wichtigere warten, bzw. seine Priorität senkt, wenn er durch einen unwichtigen Zugriff einen wichtigeren Prozeß auf seinem Prozessor ausbremst. Aufträge an den Server sollten sehr kurz sein, sodaß er schnell auf *Priority Inversion*-Situationen reagieren kann.

Wenn sich mehrere Instanzen einer Serverklasse auf einem Prozessor befinden, so kann man zumindest für diese eine gemeinsame *Run Queue* realisieren, um die Last unter diesen Instanzen optimal zu verteilen.

## 6.11 Lastbalancierungsprobleme im Datenbankbereich

Strategie: dynamisches, hierarchisches Verfahren (es wird aber keines vorgestellt).

Granulat: Datenbankanwendungen.

Lastinformation: der Schwerpunkt liegt auf der Beachtung von parallelen Zugriffen auf gemeinsame Datenobjekte.

Aufgaben: nicht genau spezifiziert.

[Härder87] erläutert unter anderem die speziell in Datenbankanwendungen auftretenden Schwierigkeiten bei der Lastbalancierung. Viele Aufträge greifen auf wenige gemeinsame Datenobjekte zu; das schränkt die Parallelarbeit einerseits durch den zentralen Ort der Daten und andererseits durch die Synchronisation der Zugriffe (logische Konsistenzbedingungen der Daten) stark ein.

Man muß einen geeigneten Grad an Parallelität finden: zu wenig Parallelarbeit nutzt die Ressourcen des Systems (vor allem die Prozessorkapazitäten) nicht aus; bei wenig Parallelität erreicht man die besten mittleren Bearbeitungszeiten; zu hohe Parallelarbeit führt zu großen Zeitverlusten sowie Verklemmungen durch die Sperrprotokolle und damit zu unnötig häufigem Abbruch von Transaktionen.

{Härder87] schlägt für große Systeme eine hierarchische Lastbalancierer-Struktur vor. Globale Entscheidungen sollen auf hoher Ebene getroffen und in den niedrigeren Ebenen verfeinert und angepaßt werden.

## 6.12 Dynamische, verteilte Lastbalancierung

Strategie: dynamisches, verteiltes Verfahren.

Granulat: Prozesse auf vernetzten Workstations.

Lastinformation: bei der momentanen Prozessorauslastung wird die Zahl der laufenden Prozesse und die *busy-time* berücksichtigt. Der Aufwand zur Migration eines Prozesses wird beachtet.

Aufgaben: Weitergabe ankommender Aufträge (*Remote Execution*) und Migration laufender Aufträge.

[Ezzat86] beschreibt ein dynamisches, verteiltes Lastbalancierungsverfahren. Jeder Knoten mißt periodisch seinen Lastzustand und sendet diesen, falls er sich stark geändert hat, den Lastbalancierern der anderen Knoten zu. Auf diese Weise ist jeder Knoten über die aktuelle Last der anderen informiert und kann entscheiden, wann seine Last so weit über dem Durchschnitt liegt, daß sich die Migration eines Auftrages zu dem am wenigsten belasteten Knoten lohnt (Prozeßmigration). Vor allem kann er, sobald er einen neuen Auftrag erhält, entscheiden, ob er diesen selbst ausführt oder sofort an einen anderen abgibt (durch einen *Remote-Shell* Aufruf).

Die Last eines Knotens wird anhand der Zahl der laufenden Prozesse sowie anhand der *busy-time* beurteilt:

$$\texttt{lokale\_Last} = f_1(\#\texttt{laufende\_Prozesse}) + f_2(\#\texttt{laufende\_Prozesse/busy\_time\_Anteil})$$

## 6.13 Globale Lastbalancierung auf Broadcast-Systemen

Strategie: dynamisches, dezentrales Verfahren.

Granulat: Prozesse. Betrachtet werden vernetzte Workstations.

Lastinformation: momentane Prozessorauslastung (Auftragsschlangenlänge).

Aufgaben: Migration von Aufträgen.

[Baumgartner88] untersucht eine globale, dynamische Lastbalancierungsstrategie. Auf einem System mit Broadcast-Bus (vernetzte Workstations) ermitteln die Prozessoren den Knoten mit minimaler sowie den mit maximaler Last. Der höchstbelastete gibt dann einen Auftrag aus seiner Warteschlange (kein Multitasking) an den am wenigsten beladenen Prozessor ab. Als Lastmaß wird die Länge der Warteschlange verwendet. Das Problem besteht nun darin, die Knoten mit maximaler und minimaler Last effizient zu ermitteln.

Solange man unbeschäftigte Knoten im Netz hat, ist es wichtig, den Auftrag des höchstbelasteten Knotens sofort an einen unbelasteten zu senden, damit der direkt mit der Bearbeitung beginnen kann. Sind alle Knoten beschäftigt, so landet der migrierte Auftrag sowieso in einer Warteschlange. Dann kann die Minimum-Maximum Suche mit höherer Priorität ablaufen. Die Suche nach dem Lastminimum verläuft folgendermaßen (analog die Suche nach dem Maximum):

> Alle Knoten kennen das Intervall, in dem sich die minimale Last befindet. Zudem wissen sie, ob sich ein Knoten in diesem Intervall befindet. Das Intervall wird solange eingeschränkt, bis der Knoten mit minimaler Last feststeht. Jedesmal, nachdem einer die neuen Intervallgrenzen bestimmt hat, versuchen alle diejenigen, die noch in diesem Intervall liegen, sich zu melden, d.h. ihre Last und die nächsten Intervallgrenzen zu *broadcasten*. In Ethernet-Netzen (Kollisionserkennung) kommt dabei genau einer durch, die anderen kollidieren oder empfangen schon vorher die Broadcast-Nachricht des schnellsten Knotens. Die Nachricht des ersten Knotens ist damit für alle gültig.
>
> Jeder Knoten durchläuft also folgenden Algorithmus:

```
Intervall_l= 0; Intervall_r= 1; found= false
repeat
  Intervall_r?= (Intervall_l + Intervall_r) / 2 /*versuche die linke Hälfte des Intervalls*/
  if Last ≤ Intervall_r? then                  /*Knoten ist selbst noch in der linken Hälfte*/
```

```
      try_broadcast(Last)
    if not any_broadcast() then              /*niemand ist mehr in der linken Intervallhälfte*/
       Intervall_l= Intervall_r?             /*Minimum muß in der rechten Hälfte liegen*/
    else            /*jemand ist noch in der linken Hälfte. Sein Wert wird als neue rechte Grenze benutzt*/
       Intervall_r= read_broadcast() /*das kann der eigene Lastwert oder der eines anderen sein*/
    if Intervall_l=Intervall_r then found= true
  until found
```

Die Migration von Aufträgen ist billig, solange alle Knoten auf denselben Sekundärspeicher zugreifen. Das ist bei Workstations mit zentralem Plattenserver der Fall. Eine Auftragsmigration wird realisiert, indem der Quellknoten durch ein Remote-Shell Kommando den Auftrag auf dem Zielknoten ausführen läßt.

## 6.14 Die dynamische, dezentrale Gradientenmethode

Strategie: dynamisches, dezentrales Verfahren.

Granulat: nicht festgelegt, reine Simulation.

Lastinformation: momentane Prozessorauslastung.

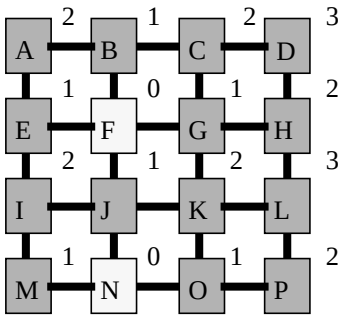
Aufgaben: Migration von Aufträgen.

[Lin87] stellt ein Verfahren vor, bei dem die Prozessoren nur die lokale Last sowie die ihrer Nachbarn kennen. Allerdings pflanzt sich die Lastinformation durch das Netz fort, sodaß im Laufe der Zeit eine globale Gleichverteilung der Last erreicht wird.

Jeder Prozessor ist entweder wenig, mittel oder stark belastet. Die Idee besteht darin, daß ein stark belasteter Prozessor einen Auftrag in Richtung des nächsten wenig beladenen Prozessors schickt. Der Auftrag wird solange weitergegeben, bis er auf einem leicht beladenen Prozessor ankommt, der ihn dann behält. Dazu hat jeder Prozessor ein Maß für seinen Abstand zum nächsten leicht belasteten Prozessor.

Wenn das System voll belastet ist, d.h. kein wenig belasteter Prozessor existiert, so werden auch keine Aufträge mehr migriert. Dies wird durch einen Maximalabstand realisiert. Ein Knoten, der Maximalabstand als Entfernungsmaß hat, weiß keinen Weg zu einem leicht beladenen Knoten.

Die Berechnung bzw. Aktualisierung der Abstandsmaße geschieht wie folgt: Periodisch senden die Prozessoren allen Nachbarn ihre aktuellen Entfernungsmaße. Ein Knoten, der wenig belastet ist, hat den Lastwert Null. Jeder andere Knoten nimmt als Lastwert den kleinsten seiner Nachbarn und erhöht ihn um Eins. Im Bild sind die Knoten `F` und `N` wenig belastet, alle anderen mittel oder schwer:




Mit dem Versenden des Lastwertes Null signalisiert der Knoten zugleich die Bereitschaft, Aufträge von anderen zu übernehmen. Da überbelastete Prozessoren einen Auftrag an den Nachbarn mit dem kleinsten Lastwert abgeben, wandert der automatisch auf einen wenig belasteten Knoten zu.

[Lin87] untersucht auch die Probleme, die bei Anwendung des Algorithmus auf heterogene Systeme entstehen. Dabei stößt er auf drei interessante Punkte:

1. Nicht jeder Auftrag kann auf jedem Prozessor ablaufen. Wenn also ein Prozessor einen Auftrag erhält, der einen anderen Maschinentyp verlangt, so muß er ihn ins Netz zurückgeben. Eine Alternative besteht darin, für die verschiedenen Prozessortypen separate Lastwerte anzulegen.
2. Wenn die Prozessoren unterschiedliche Leistung haben, so müssen sie verschiedene Maße für leichte, mittlere und schwere Belastung verwenden.
3. In heterogenen Netzen sind oft die Verbindungen zwischen den Prozessoren ungleich schnell. Daher sollte ein Prozessor nicht einfach den kleinsten Lastwert seiner Nachbarn um Eins inkrementieren, sondern ein Maß für die Verbindungskosten zu diesem Nachbarn zuaddieren (dadurch kann evtl. ein Nachbar mit höherem Lastwert attraktiver werden).

## 6.15 Vergleich dreier dynamischer Strategien

Strategie: dynamische, dezentrale Verfahren.

Granulat: Teile von Prozessen. Betrachtet wird eine Hypercube-Architektur, ist aber nicht Voraussetzung.

Lastinformation: Lastzustände der Nachbarknoten (nicht nur direkt verbundene, sondern in einem gewissen Radius). Last wird an der Zahl der laufenden Prozesse gemessen.

Aufgaben: Migration von laufenden Aufträgen.

[Hwang87] stellt drei dynamische, dezentrale Lastbalanicerungsalgorithmen vor und vergleicht sie bezüglich Skalierbarkeit, Granulat der Aufträge und der Wahl der Aktivierungsschwelle. Die Strategien verfahren wie folgt:

Ein separater Prozeß je Prozessor vergleicht periodisch seine Lastsituation mit denen der Nachbarn (nicht unbedingt nur diejenigen, mit denen er direkt verbunden ist). Jeder Prozessor ist entweder wenig, mittel oder stark belastet. Stark belastete Prozessoren können laufende Aufträge an wenig belastete Prozessoren abgeben. Die drei Verfahren unterscheiden sich in der Initiative:

1. Beim ersten Verfahren löst ein unterbelasteter Knoten eine Lastbalancierungsaktion aus (*receiver initiated*), indem er seine Nachbarn nach ihrem genauen Lastzustand fragt (als Maß verwendet man hier die Zahl der aktiven Prozesse). Vom höchstbeladenen Nachbarn fordert er dann einen Prozeß an. Dieses Verfahren zeigt bei allgemein hoher Systemlast gute Ergebnisse. Ein Auftrag kann hier nur einmal migrieren; so verhindert man, daß er endlos im Kreise herum gereicht wird.
2. Umgekehrt lösen beim *sender initiated* Verfahren die überlasteten Knoten Lastbalancierungsaktionen aus. Das Verfahren ist die in [Lin87] vorgestellte Gradientenmethode (siehe auch Kapitel 6.14): Überbelastete Prozessoren geben einen Prozeß an den Nachbarn mit dem kleinsten Lastwert ab; so wandert der Prozeß automatisch auf einen wenig belasteten Knoten zu.Die Strategie ist bei leicht beladenen Systemen erfolgreich.
3. Das dritte Verfahren benutzt die Sender-initiierte Methode bei leichter Last und die Empfänger-initiierte Methode bei hoher Systemlast. Jeder Knoten arbeitet nach dem Verfahren, das aufgrund der Last seiner Umgebung angemessen scheint. Die Knoten arbeiten also auch zusammen, wenn sie gerade mit verschiedenen Strategien operieren.

## 6.16 Partnerwahl bei verteilter Lastbalancierung

Strategie: dynamisches, dezentrales Verfahren.

Granulat: sehr kleine Aufträge, keine Prozesse im Sinne von UNIX. Betrachtet werden Hypercubes.

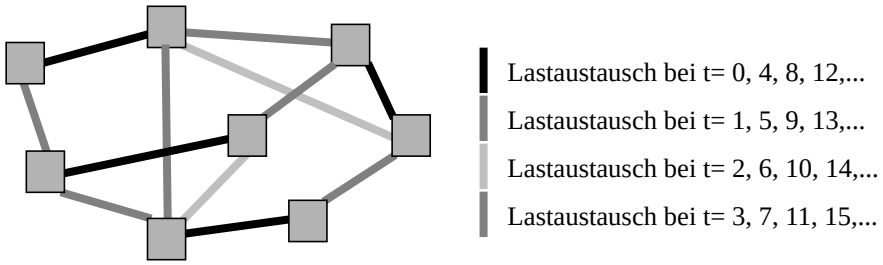
Lastinformation: momentane Prozessorauslastung.

Aufgaben: Migration von Aufträgen.

In [Hosseini90] werden zwei dynamische, verteilte Lastbalancierungsalgorithmen vorgestellt, wobei die Kommunikationspartner der Prozessoren nach der Methode der Graphenfärbung gewählt werden. Das Verfahren basiert auf

einem synchronen System, verfügt also über einen zentralen Takt für alle Prozessoren. Als Vereinfachung wird die Gesamtlast auf dem System als konstant angenommen.

Zunächst wird statisch bestimmt, welcher Prozessor wann mit welchem anderen Lastinformationen austauscht und bei Bedarf Last übernimmt bzw. abgibt. Dazu betrachtet man den Graph des Systems (siehe Bild), gibt sich `k` Farben vor und färbt nun die Kanten so ein, daß kein Knoten zwei gleichfarbige Kanten erhält. Zur Laufzeit spricht jeder Prozessor zum Zeitpunkt `t` mit dem Kollege, mit dem er über eine Kante der Farbe (`t mod k`) verbunden ist. Pro Zeitschritt unterhält sich also jeder mit maximal einem Partner, nach jeweils `k` Zeitschritten hat sich jeder mit allen seinen Nachbarn ausgetauscht.




Der Lastaustausch zwischen `i` und `j` erfolgt nach der Formel $Last_i(t+1) = (Last_i(t) + Last_j(t))/2$. Das garantiert, bei gleichbleibender Gesamtlast, die Konvergenz aller Einzellasten gegen den Durchschnitt. Durch Gewichtungsfaktoren $\alpha Last_i(t)+(1-\alpha)Last_j(t)$ kann man die Konvergenzgeschwindigkeit beeinflussen.

Das zweite Verfahren trägt der Tatsache Rechnung, daß man Last nicht beliebig fein aufteilen kann, weil sie aus unzerteilbaren Aufträgen endlicher Größe besteht. Man betrachtet daher $Last_i(t)$ als ganzzahlige Größe. Der Lastaustausch sieht nun folgendermaßen aus: $Last_i(t+1) = \lceil (Last_i(t) + Last_j(t))/2 \rceil$; bei Prozessor `j` wird entsprechend abgerundet. Auch hierfür kann man Konvergenz nachweisen. Die Konvergenzgeschwindigkeit hängt natürlich von der Verbindungsstruktur ab; gute Ergebnisse wurden mit Hypercubes erzielt.

## 6.17 Probabilistische dynamische Lastbalancierung

Strategie: dynamisches, dezentrales Verfahren.

Granulat: nicht festgelegt, ebensowenig ein bestimmtes System; reine Simulation.

Lastinformation: Auftragsankunft und Bearbeitungszeit nach einer Wahrscheinlichkeitsverteilung. Abschätzung der Restlast bzw. Betrachtung der Zahl der anliegenden Aufträge.

Aufgaben: Ankommende Aufträge werden zugewiesen.

[Hsu86] untersucht drei dynamische, dezentrale Lastbalancierungsverfahren. Jeder Knoten veröffentlicht periodisch seinen Lastzustand (*Broadcast*) und vergleicht ihn mit den Zuständen der anderen. Aus den Differenzen berechnet er die Wahrscheinlichkeit, mit der er einen ankommenden Auftrag an einen der Partner weitergibt. In dem verwendeten Simulationsmodell werden die Kosten für das Verschicken eines Auftrags zwischen Knoten mitberücksichtigt.

1. Als Maß für die Last eines Knotens wird die derzeit noch anstehende Arbeit verwendet. Sei `f` der Anteil, zu dem ein Knoten einen Auftrag pro Zeitschritt abarbeiten kann (man nimmt an, alle Aufträge haben dieselbe Laufzeit). Dann berechnet sich die durchschnittliche Bearbeitungszeit `T` der Aufträge des Knotens im Zeitabschnitt t als

   ```
   T(t) = f*Rechenzeit(t)/Zahl_der_beendeten_Aufträge(t) + (1-f)*T(t-1).
   ```

   Die noch anstehende Arbeit im Knoten ergibt sich zu

   ```
   Arbeit(t) = T(t)*Zahl_der_wartenden_Aufträge(t).
   ```

   Trifft nun ein Auftrag ein, so prüft der Knoten, ob er überdurchschnittlich belastet ist (abzüglich eines Schwellwertes). Wenn das der Fall ist, so wählt er zufällig einen anderen Knoten, an den er den Auftrag weitergibt (falls dieser unterbelastet ist). Falls er nach einigen Versuchen keinen unterbelasteten Kollegen erwischt hat, behält er den Auftrag bei sich.

2. Die Last der Knoten wird zunächst wie im obigen Verfahren bestimmt. Der Knoten berechnet nun die Wahrscheinlichkeiten, mit denen er ankommende Aufträge an andere weitergibt, proportional zu den Lastdifferenzen. Kommt ein Auftrag an, so bestimmt er wie oben zufällig einen Kollegen, der den Auftrag bekommt. Allerdings ist der Zufall nach den Wahrscheinlichkeiten gewichtet.

   Um zu vermeiden, daß momentan unterbelastete Knoten mit Aufträgen überschwemmt werden, verringert der Knoten jeweils die Wahrscheinlichkeit des Kollegen, nachdem er einen Auftrag in ihn abgeschoben hat.

3. Um den Overhead zur Berechnung der noch anstehenden Arbeit zu verringern, nimmt man die durchschnittliche Bearbeitungszeit T der Aufträge als konstant an. Die Last wird also nur anhand der momentanen Anzahl anstehender Aufträge gemessen.

## 6.18 Vergleich zweier dynamischer Verfahren

Strategie: dynamisches, dezentrales Verfahren.

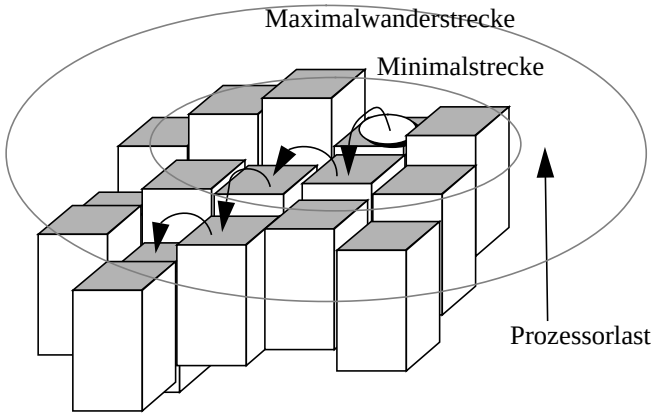
Granulat: kurze Berechnungen als Prozesse. Man betrachtet in Gitterstrukturen verbundene Prozessoren.

Lastinformation: lokale Prozessorauslastung (gering/mittel/hoch) und Entfernung zum nächsten gering belasteten Prozessor.

Aufgaben: Abgabe von Prozessen, die in der lokalen Warteschlange stehen.

[Kale88] stellt zwei dynamische, verteilte Lastbalancierungsalgorithmen vor und vergleicht Simulationsergebnisse:

1. Jeder Prozessor kennt die momentane Last seiner Nachbarn. Sobald auf einem Prozessor ein neuer Auftrag entsteht, schickt er ihn an seinen am wenigsten beladenen Nachbarn. Dieser gibt ihn wiederum an seinen am wenigsten belasteten Nachbarn ab. Das setzt sich fort, bis der Auftrag in einem (lokalen) Minimum der Last angelangt ist und dort angenommen wird (siehe Bild). Natürlich ist die Zahl der Schritte, die ein Auftrag weitergegeben werden kann, begrenzt, da er sonst zuviel Nachrichtenlast erzeugen und sein Start verzögert würde. Andererseits setzt man auch eine Mindestanzahl an Schritten fest, damit der Auftrag über lokale Minima hinaus zu besseren Minima gelangen kann.

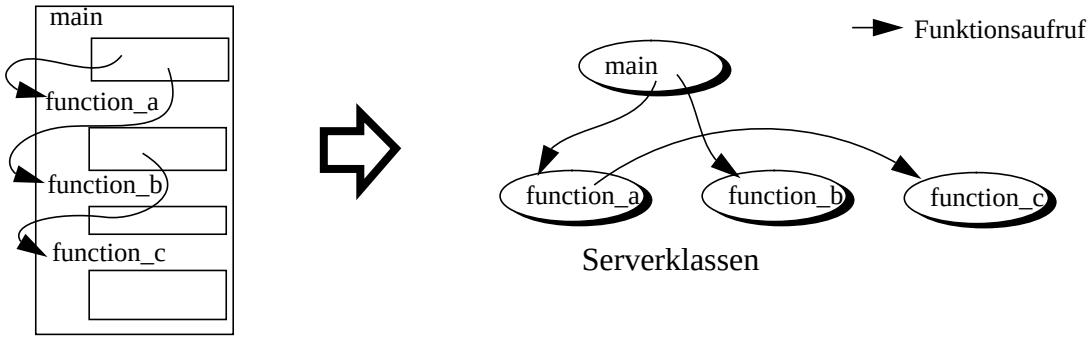



2. Das andere Verfahren ist die in [Lin87] vorgestellte Gradientenmethode (siehe auch Kapitel 6.14). Ankommende Aufträge werden in eine Warteschlange eingereiht, bis sie bearbeitet werden können (kein Multitasking). Es werden auch nur Aufträge aus Warteschlangen verschickt.

Die Simulation der Verfahren berücksichtigt Prozessorlasten und Kommunikationskosten. Als Verbindungstopologien wurden ein zweidimensionales Gitter und ein doppelt vernetztes Gitter gewählt. Als Anwendung wurden die Fibonacci-Zahlen nach der *Divide and Conquer* Methode berechnet.

# 7 Das *HiCon* System

*HiCon* unterstützt Lastbalancierung paralleler und verteilter Anwendungen. Ein Anwendungsprogamm wird in Funktionen (bzw. Module) aufgespalten, die jeweils als eigener Prozeß (*Serverklasse*) realisiert werden (siehe Bild). Die Funktionen kooperieren unter Verwendung von *HiCon*-Libraryfunktionen. Der Programmierer gibt zu jeder Anwendung eine globale Ablaufbeschreibung (*Skript*) sowie eine Abschätzung des Lastverhaltens jeder Funktion (*Lastprofil*) an.

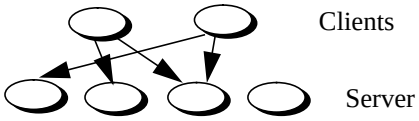



Ein *HiCon* Werkzeug gewinnt aus dem Skript eine geeignete Startkonfiguration und statische Informationen für die dynamische Lastbalancierung. Der Ablauf der Anwendung wird durch eine *HiCon* Scheduling-Komponente und die an die Server gebundenen Libraryfunktionen gesteuert.

## 7.1 Das *HiCon* Programmiermodell

*HiCon* verwendet das Modell asynchroner Funktionsaufrufe. Die als Serverklassen realisierten Funktionen rufen einander auf und erwarten bei passender Gelegenheit die Resultate. Es gibt kein Ereigniskonzept; Funktionsaufrufe sind das einzige Mittel zur Ablaufsteuerung einer Anwendung. Das Modell der *Remote Procedure Calls* bietet viele der Vorzüge sequentieller Programmiersprachen (etwa Aufrufparameter, Ergebnisrückgabe, Fehlerbehandlung, Rekursion und hierarchische Abstraktion des Kontrollflusses, siehe Kapitel 1.3).

Gewöhnlich versteht man das Client-Server Modell als flache Struktur (siehe Bild). Wir lassen jedoch auch hierarchische Aufrufstrukturen samt Rekursion zu. Natürlich werden durch synchrone rekursive Aufrufe bereits ohne Parallelarbeit Serverinstanzen belegt gehalten und können nicht zur Bearbeitung ihrer eigenen rekursiven Aufrufe benutzt werden (sie warten nicht auf neue Aufträge sondern auf das Resultat des rekursiven Aufrufes. *HiCon* Serverinstanzen kennen kein automatisches *Multithreading*).




Das HiCon System stellt für jede Serverklasse (Funktion) einige Instanzen zur Verfügung. Für den Programmierer spielt es keine Rolle, an welche Instanz der adressierten Serverklasse der Aufruf geht. Im *HiCon* Modell gilt dies auch für kontextsensitive Aufrufe (*Sessions*), da die Instanzen einer Klasse ihre gemeinsamen Daten untereinander konsistent halten. Ob eine Sequenz von Aufrufen innerhalb einer Session stets zur selben Serverinstanz geleitet wird, ist allein eine Lastbalancierungsentscheidung. Da jeder Datensatz grundsätzlich durch eine Serverfunktion gekapselt wird, genügt die Synchronisation der Instanzen je einer Klasse untereinander (siehe Kapitel 1.4).

Aus der Sicht des *HiCon* System sind der Zustand eines Servers, der Kontext eines Servers und die Daten, die ein Server verwaltet, dasselbe; der Lastbalancierer kennt nur Server mit ihren Daten. Auch Dateien sind lokale Daten des Servers, der sie verwaltet. Dieser Ansatz vereinfacht und vereinheitlicht den Umgang mit Kopien von Daten, Serverklassen und Kontexten.

Dieses Modell ermöglicht es, die Zuweisung eines (kontextsensisitven) Aufrufes an eine spezielle Serverinstanz als reinen Lastbalancierungsaspekt zu betrachten. Es entstehen keine Bindungen durch Kontexte auf Seiten des Clienten oder des Servers.

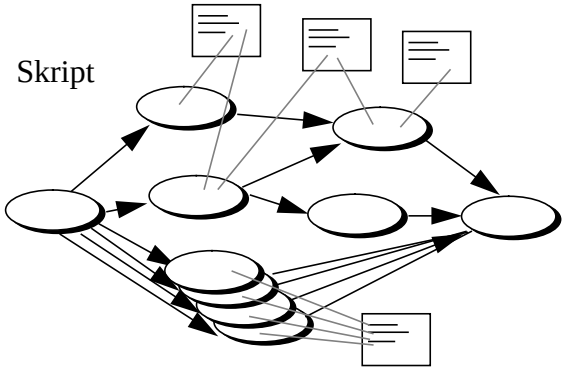
Eine Funktionsausführung muß an einer einzigen Stelle ablaufen, da die Funktion das Granulat der Parallelität darstellt (und keine Migration laufender Ausführungen möglich ist). Außerdem wird in einer Ausführung ausschließlich auf die lokalen Daten der Funktion zugegriffen. Das ist keine Einschränkung, denn die Ausführung kann andere Funktionen aufrufen (die eventuell woanders laufen) und auf diese Weise Zugriffe auf externe Daten durchführen.

## 7.2 Die Struktur der *HiCon* Lastbalancierung

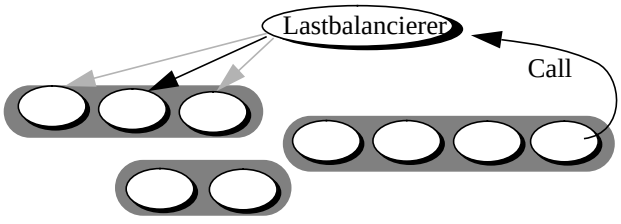
Wir unterscheiden drei Ebenen der Lastbalancierung, die untereinander Informationen austauschen (siehe auch Kapitel 4.1):

1. **Skriptebene**: Gegeben ist eine Ablaufbeschreibung des Gesamtauftrages als Kooperation der Einzelfunktionen (siehe Bild). Die Aufgabe der (statischen) Lastbalancierung besteht darin, sequentielle Pfade (*bottleneck paths*) in der Ausführung zu erkennen, die maximale Parallelität einzelner Funktionen abzuschätzen sowie die Lokalität, Frequenz und Charakteristik von Datenzugriffen festzustellen. Daraus kann man die Zeitgrenzen einzelner Ausführungspfade und die erforderliche Anzahl und Plazierung von Serverinstanzen gewinnen.

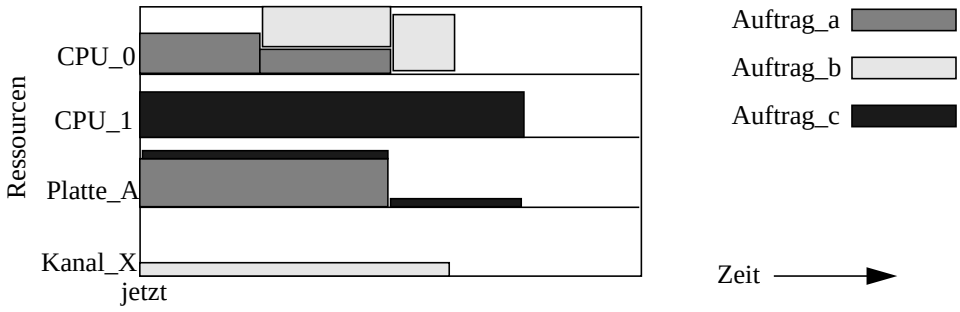
   Außerdem werden diese Informationen an die dynamische Balancierung der unteren Ebenen weitergegeben: Abschätzungen über die zu erwartende Sekundärlast (Unteraufrufe) einzelner Funktionsaufrufe und das weitere Zugriffsverhalten auf Daten.




2. **Aufrufebene**: Zur Laufzeit werden Funktionsaufrufe gepuffert und Serverinstanzen zugewiesen (siehe Bild). Dazu wird jeweils das Lastprofil des Auftrages und die momentane Auslastung der benötigten Ressourcen bei den Instanzen betrachtet. Außerdem sind (von der Skriptebene gegebene) Sekundärlasten und erforderliche Kontextmigrationen innerhalb der Serverklasse zu berücksichtigen. Bei Bedarf werden Instanzen gelöscht, migriert oder erzeugt.

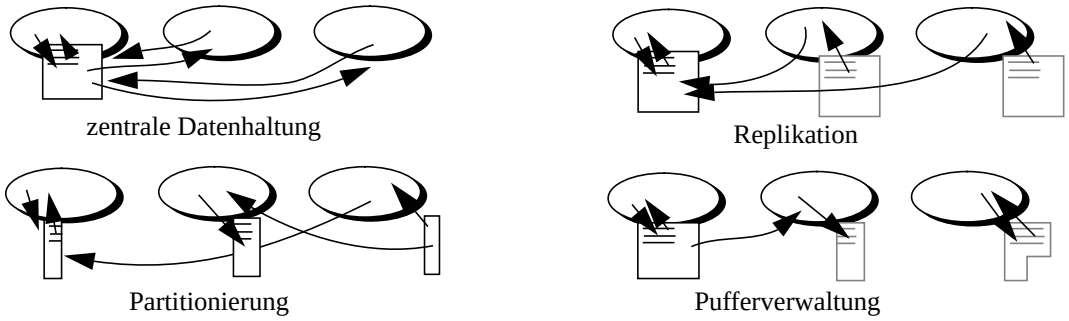



Dazu führt die Lastbalancierungskomponente einen Systemfahrplan (siehe Bild), worin die Aufträge und die entstehende Ressourcenbelastung geplant wird. Diese Tabelle wird anhand periodischer Lastmessungen korrigiert.

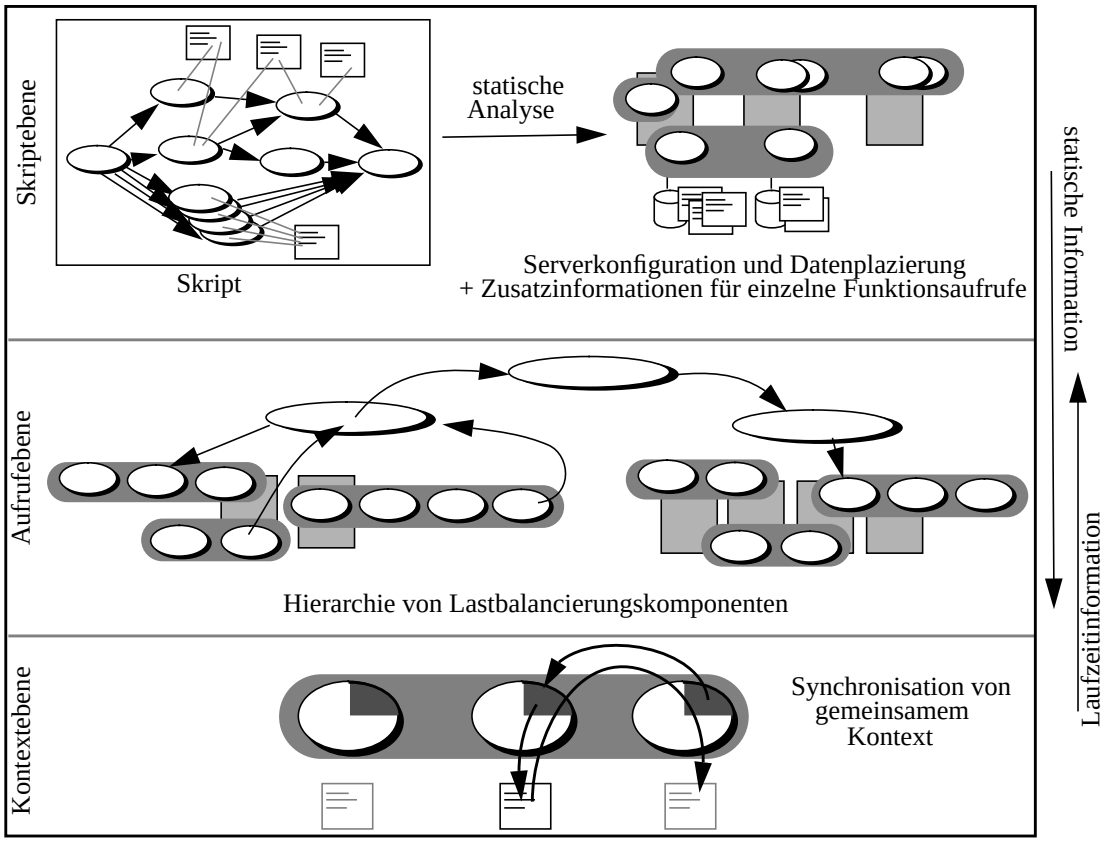


Da eine zentrale Instanz auf großen Systemen zum Engpaß wird, sollen die Komponenten dieser Balancierungsebene hierarchisch strukturiert werden. Lokale Komponenten treffen weitmöglichst autonome Entscheidungen und nutzen die Hierarchie nur in Fällen großer Aufträge oder stark ungleicher Systembelastung.

3. **Kontextebene**: Wird eine Serverinstanz mit einer Funktionsausführung beauftragt, so muß sie den Kontext ihrer Klasse mit den übrigen Instanzen konsistent halten. Für die Lastbalancierung ist es wichtig, anhand der (von höheren Ebenen) prognostizierten längerfristigen Zugriffsmustern (Lese-/ Schreibverhältnis, Wiederverwendung derselben Instanz) zu entscheiden, welche Teile des Kontextes bei welcher Instanz liegen sollen und wieviel Kopien wo sinnvoll sind (siehe Bild bzw. Kapitel 1.4).



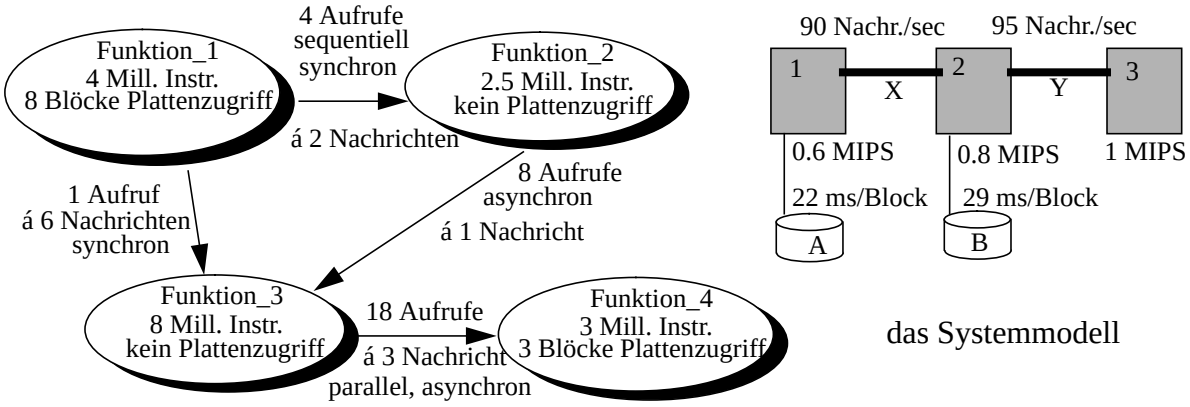

Das folgende Bild zeigt noch einmal die Lastbalancierungsstruktur im Zusammenhang:



## 7.3 Der *HiCon* Simulator

Um die theoretischen Möglichkeiten und Grenzen des Lastbalancierungsmodells zu erforschen wurde ein Simulationsprogramm realisiert, das den Ablauf von Anwendungen anhand ihrer Skriptbeschreibungen, Lastprofile und eines Systemmodells durchspielt. Es implementiert das zweite in Kapitel 3.6 vorgestellte Kostenmodell. Der Verlauf der Simulation läßt sich graphisch aufbereiten. Wir wollen im folgenden den Simulator anhand einer Beispielanwendung vorstellen.

Das Bild zeigt die Skript-Beschreibung der Anwendung und das Modell des verfügbaren Systems:



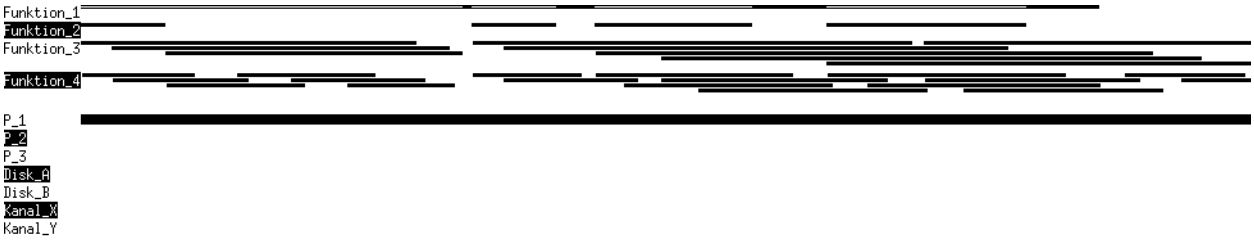

die Auftragsbeschreibung

das Systemmodell

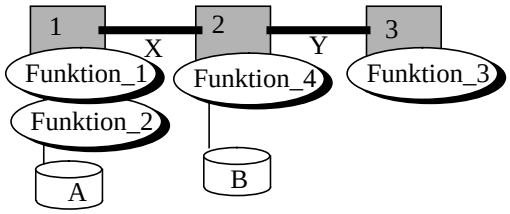
- Die Anwendung wird durch Aufruf der Funktion 1 aktiviert.
- Die Anzahlen der Funktionsaufrufe sind absolute Angaben und nicht pro Ausführung der rufenden Funktion gemeint. Auf diese Art kann man auch rekursive Aufrufe beschreiben ohne Verzweigungen oder Abbruchkriterien modellieren zu müssen.
- Da es in *HiCon* keine ‘losen’ Daten gibt (sie sind jeweils durch einen lokalen Server gekapselt) beziehen sich die Plattenzugriffe jeweils auf eine lokale Platte.
- In den Lastprofilen und den Aufrufen werden bisher keine verschiedenen Aufruftypen unterschieden. In der Realität kann ein Server verschiedene Aufträge durchführen, deren Lastprofile differieren.

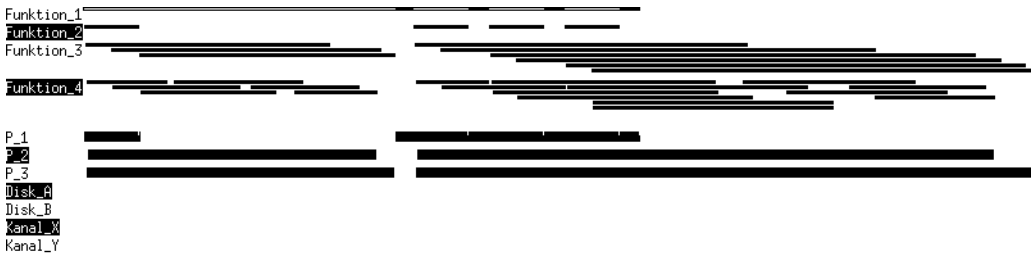
Verschiedene Methoden der Lastbalancierung und Asynchronität der Verarbeitung ergeben folgendes:

1. **Ohne Lastbalancierung**: die Anwendung läuft nur auf Prozessor 1. Wir wählen Zeitschritte der Größe 0.1 sec. Der Zeitbedarf zur Ausführung von Funktion_1 ergibt sich zu 233.5s (im Bild sieht man das Ablaufverhalten und die Ressourcenauslastung).



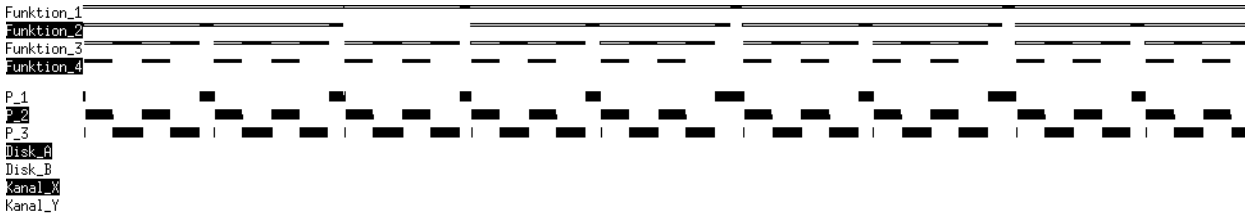
2. **Einfache statische Verteilung**: die Funktionen werden fest je einem Prozessor zugeordnet, wie im Bild dargestellt. Funktion 4 läuft wegen der Plattenzugriffe auf Prozessor_2. Wir erhalten eine Laufzeit von = 73.9s (das untere Bild zeigt wiederum den Ablauf und die Belastung).



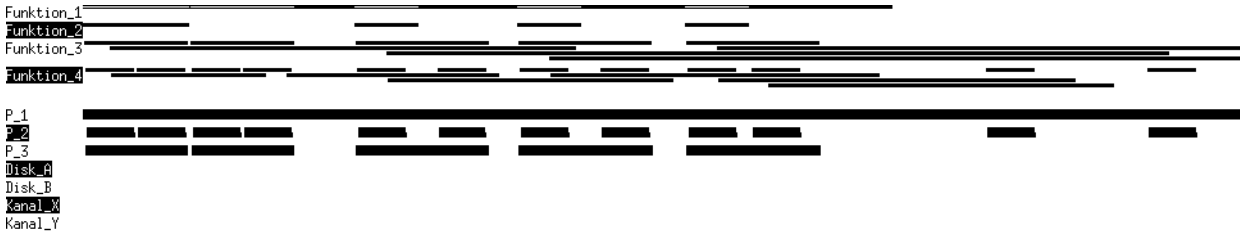




Die Verteilung der Funktionen ergibt also bereits eine Beschleunigung um den Faktor 3.24 (ein Speedup von 4 wäre aufgrund der Prozessor-Leistungen möglich). Die Prozessoren 2 und 3 stellen sich zeitweise als Engpässe heraus.

3. Wären alle Aufrufe synchron, d.h. keine Parallelarbeit möglich, so erhielte man (bei der einfachen statischen Verteilung) eine Laufzeit von 164.8s (siehe Bild). Könnte man alle Aufrufe asynchron realisieren, so ergäbe sich eine Zeit von 72.1s (zweites Bild).




4. **Einfache dynamische Balancierung**: jede Funktionsausführung wird auf dem Prozessor gestartet, der gerade am wenigsten belastet ist (Funktionen, welche Plattenzugriffe beinhalten, dürfen nur auf Prozessoren ablaufen, die über Platten verfügen). Weiterhin erhöht man den Belastungswert eines Prozessors, nachdem man ihm eine Ausführung zugewiesen hat. Das vermeidet, daß einem unbelasteten Prozessor auf einen Schlag viele Ausführungen zugedacht werden. Wir erreichen eine Bearbeitungszeit von 94.2s (siehe Bild).




Der Simulator soll auf das vollständige *HiCon* Verarbeitungsmodell hin erweitert werden. Das verlangt einerseits die explizite Modellierung von Serverinstanzen und andererseits die Berücksichtigung gemeinsamer Daten (Kontexte) innerhalb von Serverklassen.

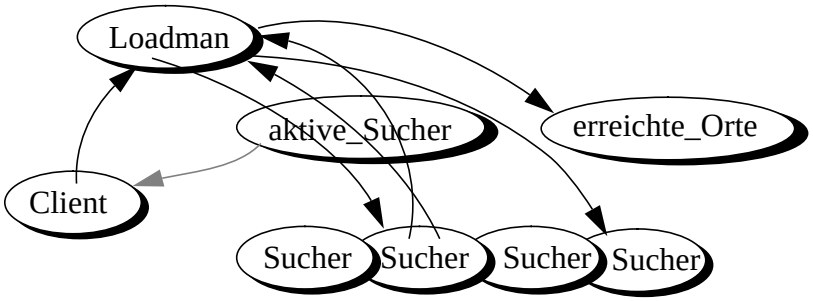
## 7.4 Die Realisierung des *HiCon* Systems

- In ‘bottom up’ Vorgehensweise wurde als Basis eine Nachrichtenschnittstelle COIN (*CO*mmunication *IN*terface) geschaffen, mittels derer Prozesse in heterogenen, verteilten Systemen durch Nachrichten kooperieren können. Dieses Interface ermöglicht die Prozeßerzeugung über Knotengrenzen hinweg in uniformer Weise. Sie erlaubt auch die Verarbeitung asynchron eintreffender Nachrichten. Die COIN Schnittstelle basiert auf unterschiedlichen Nachrichtenmechanismen, die Betriebssysteme zur Verfügung stellen. Daher werden Konstrukte wie Verbindungen, Ports und File-Handles durch die COIN Library verdeckt. Der Anwendungsprogrammierer benutzt lediglich abstrakte Prozeß-Identifikatoren der Partner, anhand derer die COIN Schnittstelle automatisch das günstigste (proprietäre) Kommunikationsmedium wählt. Derzeit werden UNIX Pipes, Guardian Messages und das TCP Protokoll unterstützt.
- Die COIN Schnittstelle verbindet verschiedene Hardware- und Betriebssystem-Plattformen, derzeit vernetzte UNIX-Workstations von SUN und DEC, einen *shared memory* Multiprozessor Sequent Symmetry S27, zwei UNIX Rechner HP845 und einen *shared nothing* Multiprozessor Tandem TXP.
- Auf dieser Basis wurde eine Scheduling-Komponente (*Loadman*) implementiert, die das HiCon Betriebsmodell realisiert. Sie übernimmt die Pufferung und Verteilung von Aufrufen an Serverinstanzen, die Rückgabe von Resultaten und unterstützt die Kontextverwaltung der Serverinstanzen einer Klasse. *Loadman* verwaltet die Serverklassen, die dynamisch von Serverinstanzen aus rekonfiguriert werden können. Zur Vereinfachung der Programmierung von Servern wurde eine C-Library geschaffen, die asynchrone *Remote Procedure Calls* als Funktionsaufrufe anbietet.

  Derzeit ist *Loadman* eine zentrale Komponente, die in Zukunft (entsprechend der in Kapitel 7.2 vorgestellten Struktur) verteilt werden soll.

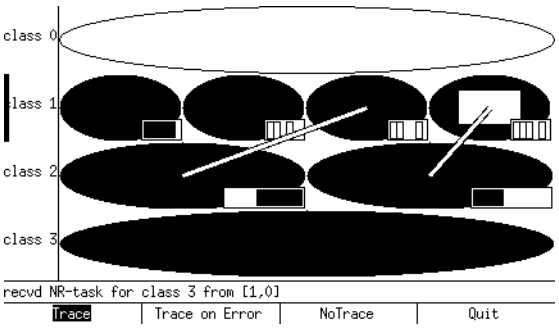
  *Loadman* kennt Serverklassen-Kontexte, d.h. Datensätze, die die Instanzen einer Serverklasse gemeinsam benutzen. Der Kontext einer Serverklasse kann bezüglich der Synchronisation in Partitionen aufgespalten werden. Loadman und die Library-Funktionen verwalten die Orte, die Verteilung der Partitionen und bieten dem Server-Programmierer durch Lese-/Schreibsperraufrufe einen sehr einfachen ortstransparenten Zugriff auf Kontextteile an.

- Als Anwendungsbeispiel wurde unter *Loadman* eine verteilte Wegesuche in gewichteten Graphen implementiert. Dabei haben wir auch kontextsensitive Serverfunktionen repliziert, die zentrale Daten gemeinsam verwalten. Das Bild zeigt die Serverstruktur der Anwendung:




  Hier werden asynchrone rekursive Aufrufe eingesetzt, da jeder *Sucher* zu einem Knoten die direkten Folgeknoten ermittelt und die Reststrecke von dort rekursiv suchen läßt. Synchrone Aufrufe würden schnell alle Instanzen blockieren (siehe Kapitel 7.1). Daher verwaltet *aktive_Sucher* die Zahl der arbeitenden Instanzen und erkennt die Terminierung der Suche. Die Serverklasse *erreichte_Orte* verwaltet die bisher im Graphen erreichten Knoten.

  Alle Serverklassen können ohne Änderung der Instanzen-Programme beliebig repliziert auf das heterogene System verteilt werden. Die Größe der Kontextpartitionen kann durch einfache Modifikation von Konstanten angepaßt werden. Das folgende Bild stellt die graphische Oberfläche des *Loadman* während einer Wegesuche vor. Dabei werden Instanzen durch Ellipsen, deren Kontext durch Rechtecke symbolisiert:




  Die Klasse 0 ist hier der Client, die Serverklasse 1 die Suchprozeduren, die Klasse 2 verwaltet die Liste der bisher erreichten Orte und die letzte Klasse die Zahl der aktiven Sucher. Der Kontext der Sucher-Klasse besteht aus der Graphenbeschreibung. Dies ist je Partition eine Datei, die einige Kanten enthält. Im Beispiel mit fünf Partitionen und einem 100 Knoten enthaltenden Graphen befinden sich in der ersten Partition alle Kanten, die von den Knoten 1 bis 20 starten. Eine schwarz eingezeichnete Partition gibt an, daß die Instanz derzeit das Original dieser Partition besitzt, ein Rahmen stellt eine Kopie der Partition dar.

  Im Beispiel legen sich die Sucher lediglich Kopien der Graphenbeschreibung an (physische Kopien der Dateien auf ihre lokalen Platten), da sie diese Kontextdaten nur lesen. Bei den Verwaltern der erreichten Orte (Klasse 2) beobachtet man hingegen ein ständiges Wechseln der Originalpartitionen , da die Verwalter die Liste modifizieren. Selbstverständlich kann man auch die Klasse 3 durch mehrere Instanzen realisieren, ohne deren Programmcode zu modifizieren. Der Kontext besteht hier aus einer einzigen Zahl, der Anzahl der zur Zeit aktiven Sucher-Instanzen.,

  Eine schwarz gefüllte Instanz führt gerade einen Aufruf durch; bei synchronen Aufrufen wird eine Linie zum (wartenden) Aufrufer eingezeichnet. Instanzen, die gerade auf die Zusendung einer Kontextpartition (entweder einer Kopie oder aber des Originales) warten, sind durch ein weisses Rechteck markiert. Der die Klassennummer verdeckende Balken ist ein Maß für die Zahl der anstehenden Funktionsaufrufe an diese Klasse.

Eine Komponente zur statischen Planung auf Skriptebene wurde bislang noch nicht realisiert.

## 7.5 Die *HiCon* Nachrichten- und *RPC*-Schnittstellen

Abschließendbetrachten wir kurz die Definitionen der *HiCon* Libraryfunktionen um die Klarheit und Einfachheit des Programmiermodells unserer Lastbalancierungsumgebung herauszustellen. Serverfunktionen sollten nur diese RPC Library verwenden, die automatisch mit der Schedulingkomponente kooperiert. Alle Funktionen liefern als Ergebnis einen Fehlercode.

- Der Client macht einen Funktionsaufruf an eine Serverklasse. Das Resultat wird im Aufrufstring zurückgegeben:
```
int loadman_call(int   class           (in),
                 int   tag             (in),
                 char  *param_result   (in),
                 int   param_size      (in),
                 int   max_result_size (in),
                 int   timeout         (in))
```
- Der Client ruft eine Funktion auf, ohne je deren Ausführung abzuwarten (dies ermöglicht die Verwendung von Ereignissen wie in Petrinetzen):
```
int loadman_call_no_result(int   class       (in),
                           char  *params     (in),
                           int   param_size  (in),
                           int   timeout     (in))
```
- Der Server wartet auf einen Auftrag:
```
int loadman_await_call(int   *tag        (out),
                       char  *params     (out),
                       int   *param_size (in-out),
                       int   timeout     (in))
```
- Der Server sendet das Resultat an den Aufrufer zurück:
```
int loadman_result(int   tag          (in),
                   char  *result      (in),
                   int   result_size  (in),
                   int   timeout      (in))
```
- Der Client wartet auf Abarbeitung des Aufrufes und Ergebnisrückgabe. Ist *tag* undefiniert, so erwartet man das Resultat eines beliebigen Aufrufs:
```
int loadman_await_result(int   *tag         (in-out),
                         char  *result      (out),
                         int   *result_size (out),
                         int   timeout      (in))
```
- Der Server möchte die Kontext-Partition lesen. Dazu sperrt er sie und bekommt sie, falls notwendig, kopiert. Der Aufruf wartet, bis die Sperre gewährt wurde:
```
int slock_context( int  partition  (in),
                   int  timeout    (in))
```
- Der Server möchte die Kontext-Partition ändern. Dazu sperrt er sie und bekommt sie, falls notwendig, geschickt. Der Aufruf kehrt zurück, sobald die Sperre gewährt wurde:
```
int xlock_context( int  partition  (in),
                   int  timeout    (in))
```
- Der Server hat die (im Sinne einer Transaktion) zusammengehörigen Zugriffe auf die Kontext-Partition abgeschlossen und gibt sie wieder frei:
```
int unlock_context(int  partition  (in),
                   int  timeout    (in))
```

Das Programm einer Serverinstanz muß folgende Funktionen (*callbacks*) bereitstellen, die die Aufgabe eines Stubs realisieren, d.h. die Kontextpartition in einen Nachrichtenstring verpacken und eine Nachricht entsprechend wieder auszupacken. Die *Loadman*-Library ruft diese Funktionen auf, um Kontextpartitionen zwischen den Instanzen einer Klasse auszutauschen:

`send_context(partition, context, size)` und
`recv_context(partition, context, size)`.

Serverkontexte können daher beliebige Datenstrukturen wie etwa verkettete Listen oder auf Platte abgelegte Dateien sein; sie sind nicht auf einen zusammenhängenden Speicherblock beschränkt.

Die *Loadman*-Library baut auf der COIN Nachrichtenschnittstelle auf, die folgende Funktionalität bereitstellt:

- Erzeuge einen Prozeß auf einem Hostrechner mit einer Kommadozeile:
```
int coin_create_process(char     *host            (in),
                        int      processor        (in),
                        char     *program         (in),
                        char     *options         (in),
                        coin_pid *son_process_id  (out)
                        int      timeout          (in))
```
- Sende eine Nachricht zum Partnerprozeß. Der Aufruf ist asynchron in dem Sinne, daß er bereits zurückkehrt, wenn die Nachricht in einem Puffer aufgezeichnet ist. Der Partnerprozeß muß sie noch nicht unbedingt gelesen haben:
```
int coin_send(coin_pid  partner_process_id  (in),
              char      *msg                (in),
              int       msg_len             (in),
              int       timeout             (in))
```
- Warte auf eine Nachricht vom Partner (oder von irgendeinem Partner, wenn kein Partner angegeben ist):
```
int coin_recv(coin_pid  *partner_process_id  (in-out),
              char      *msg                 (out),
              int       *msg_len             (in-out),
              int       timeout              (in))
```

# 8 Literaturverzeichnis

[Baumgartner88] K.Baumgartner, B.Wah, *A Global Load Balancing Strategy for a Distributed Computer System*, Workshop on the Future Trends of Distributed Computing Systems in the 1990's, 1988.

[Borr90] A.Borr, *Guardian 90: A Distributed Operating System Optimized Simultaneously for High-Performance OLTP, Parallelized Batch/Query and Mixed Workloads*, Tandem Technical Report, Cupertino, Juli 1990.

[Bowen88] N.Bowen, C.Nikolaou, A.Ghafoor, *Hierarchical Workload Allocation for Distributed Systems*, Parallel Processing Volume 2, 1988.

[Duppel87] N.Duppel, P.Peinl, G.Schiele, H.Zeller: *Progress Report #1 of PROSPECT*, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, 1987.

[Duppel87b] N.Duppel, P.Peinl, A.Reuter, G.Schiele, H.Zeller: *Progress Report #2 of PROSPECT*, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, 1987.

[Duppel88] N.Duppel, A.Reuter, G.Schiele, H.Zeller: *Progress Report #3 of PROSPECT*, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, 1988.

[Duppel88b] N.Duppel, D.Gugel, A.Reuter, G.Schiele, H.Zeller: *Progress Report #4 of PROSPECT*, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, 1988.

[Duppel89] N.Duppel, D.Gugel, A.Reuter, G.Schiele: *Progress Report #5 of PROSPECT*, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, 1989.

[Duppel89b] N.Duppel, D.Gugel, A.Reuter, G.Schiele: *Progress Report #6 of PROSPECT*, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, 1989.

[Ezzat86] A.Ezzat, *Load Balancing in NEST: A Network of Workstations*, IEEE CH23457 7 86 0000 1138, 1986.

[Ferrari86] D.Ferrari, S.Zhou, *A Load Index for Dynamic Load Balancing*, IEEE CH2345 7 86 0000 0684, 1986.

[Gonzalez78] T.Gonzalez, S.Sahni, *Preemptive Scheduling of Uniform Processor Systems*, Journal Assoc. Comput. Mach. 25, Jan. 1978.

[Härder87] T.Härder, *On Selected Performance Issues of Database Systems*, Informatik Fachberichte 154, Messung, Modellierung und Bewertung von Rechnersystemen, 1987.

[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 10, 1990.

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

[Hwang87] K.Hwang, R.Chowkwanyun, *Dynamic Load Balancing for Distributed Supercomputing and AI Applications*, Technical Report CRI-87-04, University of Southern California, Los Angeles, 1987.

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

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

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

[Lo88] V.Lo, *Algorithms for Static Task Assignment and Symmetric Contraction in Distributed Computing Systems*, Parallel Processing Volume 2, 1988.

[Martel88] C.Martel, *A Parallel Algorithm for Preemptive Scheduling of Uniform Machines*, Journal of Parallel and Distributed Computing, May 1988.

[Peinl88] P.Peinl, A.Reuter, H.Sammer, *High Contention in a Stock Trading Database: A Case Study*, ACM SIGMOD, Juni 1988.

[Reuter86] A.Reuter, N.Duppel, P.Peinl, G.Schiele, H.Zeller: *An Outlook on PROSPECT*, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, 1986.

[Reuter90] A.Reuter: *Proceedings of PROSPECT Workshop*, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, 1990.

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

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

[Yu86] P.Yu, S.Balsamo, Y.Lee, *Dynamic Load Sharing in Distributed Database Systems*, IEEE CH2345 7 86 0000 0675, 1986.