Prof. Dr. Jörg Keller

Modul 31331
Computersysteme

01727 Parallele Programmierung und Grid Computing

LESEPROBE

mathematik
und
informatik

FernUniversität in Hagen

Der Inhalt dieses Dokumentes darf ohne vorherige schriftliche Erlaubnis durch die FernUniversität in Hagen nicht (ganz oder teilweise) reproduziert, benutzt oder veröffentlicht werden. Das Copyright gilt für alle Formen der Speicherung und Reproduktion, in denen die vorliegenden Informationen eingeflossen sind, einschließlich und zwar ohne Begrenzung Magnetspeicher, Computerausdrucke und visuelle Anzeigen. Alle in diesem Dokument genannten Gebrauchsnamen, Handelsnamen und Warenbezeichnungen sind zumeist eingetragene Warenzeichen und urheberrechtlich geschützt. Warenzeichen, Patente oder Copyrights gelten gleich ohne ausdrückliche Nennung. In dieser Publikation enthaltene Informationen können ohne vorherige Ankündigung geändert werden.

Leseprobe zum Kurs 1727

Parallele Programmierung und Grid Computing

Bei diesem Kurs werden vorwiegend aufgezeichnete Vorlesungen genutzt. Als Leseprobe finden Sie hier die transkribierten Folien der einführenden Vorlesung.

FernUniversität in Hagen

Department of Mathematics and Computer Science

Parallelism and VLSI Group
Prof. Dr. Jörg Keller

Herzlich Willkommen zum ersten Teil des Kurses Parallele Programmierung und Grid Computing!
Wir beschäftigen uns zu Beginn mit den Grundlagen und Modellen der Parallelverarbeitung.

Übersicht

- **Begriffsbestimmung – Anwendungen – Bedeutung**
- **Klassifikationen - Architekturen**
- **Modelle**
- **Maße**
- **Graphen und Einbettungen**

Zunächst wollen wir eine Begriffsbestimmung vornehmen, danach mögliche Anwendungen und die Bedeutung der Parallelverarbeitung überblicksartig betrachten.
Nach der Behandlung verschiedener Klassifikationen werden wir uns den Architekturen von Parallelrechnern zuwenden.
Nach einer Betrachtung der verschiedenen Modelle, die für Parallelverarbeitung vorgeschlagen wurden, wenden wir uns den Maßen zu, mit denen wir parallele Programme bewerten, allen voran die Beschleunigung.
Schließlich betrachten wir Graphen und deren Einbettungen ineinander. Diese Betrachtung ist hilfreich, wenn wir eine Datenstruktur auf einem Rechner einbetten wollen, der eine davon abweichende Struktur hat.

FernUniversität in Hagen

Department of Mathematics and Computer Science

Begriff Parallelverarbeitung I

- **Parallelverarbeitung = Verarbeitung mit mehr als einer Ausführungseinheit**
- **Viele Ebenen der Hardware:**
  Mehrere Ausführungseinheiten in superskalaren Proz.
  Mehrere Threads in Multithreaded Proz.
  Mehrere Cores in Multicore Proz. (z.B. auch GPU)
  Mehrere Proz. auf einem Board in Multiprozessoren
  Mehrere Multiproz. in einem Rack als Cluster
  Mehrere Cluster über Netzwerke als Grid

Parallelism and VLSI Group
Prof. Dr. J. Keller

Unter Parallelverarbeitung verstehen wir die Verarbeitung eines Programms mit mehr als einer Ausführungseinheit. Dieser Begriff ist vielgestaltig, denn Parallelverarbeitung kommt auf vielen Ebenen eines Parallelrechners vor.
In einem heutigen superskalaren Prozessor finden wir mehrere Ausführungseinheiten, so dass in einem Takt mehr als ein Befehl ausgeführt werden kann.
In Multithreaded Prozessoren wie dem Intel Pentium Hyper-Threading gibt es Unterstützung für die quasi-parallele Ausführung mehrerer Threads, die sich die Vielzahl von Ausführungseinheiten teilen.
In einem Multicore Prozessor z. B. auch einem Graphikprozessor befinden sich in einem Prozessorchip mehrere Cores wobei auch jeder dieser Cores wiederum ein superskalarer Multihreaded Prozessor sein kann.
Auf einem Motherboard können sich mehrere Multicore Prozessoren befinden. Mehrere dieser Boards oder Multiprozessoren können sich als Cluster in einem Rack befinden und werden dann über ein Netzwerk miteinander verbunden.
Mehrere solcher Cluster können über Kommunikationsnetzwerke als Grid oder Cloud zusammenarbeiten.

Begriff Parallelverarbeitung II

- **Parallelverarbeitung: mehrere Proz. arbeiten zusammen an einem Problem**
- **Verteiltes Rechnen:**
  - kooperierende Rechner gehören organisatorisch nicht mehr zu einer Instution
  - Rechner sehr lose gekoppelt
- **Mit Grid- bzw. Cloud-Computing verwischt sich Grenze**

Unter Parallelverarbeitung verstehen wir also die Tatsache, dass mehrere Prozessoren an einer Problemstellung arbeiten.
Wir unterscheiden davon das Verteilte Rechnen. Beim Verteilten Rechnen gehören entweder die kooperierenden Rechner organisatorisch nicht zu einer Institution, sondern zu vielen verschiedenen oder aber die Rechner sind sehr lose gekoppelt, so dass wir nicht mehr unbedingt davon sprechen können, dass sie zusammen an einem Problem arbeiten.
Mit dem Aufkommen des Grid- und Cloud-Computing mit dem sich die zweite Hälfte dieses Kurses befassen wird, verwischt sich allerdings zunehmend die Grenze zwischen Parallelem und Verteiltem Rechnen.

FernUniversität in Hagen Department of Mathematics and Computer Science

Anwendungen I

- **High-Performance Computing**
  klassisch: große Problemstellungen in Wissenschaft und Technik

  neu: große Datenbanken, Suchmaschinen

- **Beispiel: Wettervorhersage**
  Lösung von diskretisierten Differential-Gleichungen
  Zellgröße 1 mile^3, 200 FLOP pro Zelle pro Zeitschritt
  10-Tage Vorhersage mit 10-Minuten-Schritten: 10^{15} FLOP
  braucht mehrere Tage auf PC

  FLOP=Gleitkomma-Operation

Die Anwendungen des Parallelen Rechnens finden sich klassisch im Bereich des so genannten High-Performance Computing, bei dem große Problemstellungen in Wissenschaft und Technik verarbeitet werden.
Neue Anwendungen für Parallelrechner sind auch große Datenbanken oder Suchmaschinen.
Als bekanntes Beispiel des High-Performance Computing wollen wir die Wettervorhersage nehmen. Hierbei werden Differential-Gleichungen gelöst, die im Ort und in der Zeit diskretisiert werden.
Nehmen wir an, dass die Lufthülle über der Erdoberfläche in Zellen der Größe 1 Meile^3 aufgeteilt wird. Dann benötigt diese Berechnung etwa 200 Gleitkomma-Operationen pro Zelle und pro Zeitschritt. Wollen wir nun eine 10 Tage Vorhersage berechnen und dabei die Simulationszeit in 10-Minuten-Schritten vorwärts schalten, benötigen wir dafür 10^{15} Gleitkomma-Operationen. Auf einem PC braucht eine solche Rechnung mehrere Tage, ganz abgesehen davon, dass die Vielzahl der benötigten Zellen einen großen Speicherbedarf hat, den wir in einer 32 Architektur nicht unbedingt mit einem Prozessor befriedigen können.

FernUniversität in Hagen

Department of Mathematics and Computer Science

Anwendungen II

- **Weitere Beispiele zeigen große Diversität:**

  Suchmaschine google: einer der weltgrößten Parallelrechner

  Bioinformatik: Simulation von Proteinfaltung 10^{25} Instr.

  Computergrafik: rendering, virtuelle Realität

  Crash-Simulationen von Fahrzeugen

  Atomwaffen-Simulation

Die jetzt folgenden weiteren Beispiele zeigen eine sehr große Diversität. Einer der größten Parallelrechner der Welt wird von der Firma Google für deren Suchmaschine betrieben.

In der Bioinformatik simuliert man unter anderem zur Entwicklungen neuer Medikamente die Faltung von Proteinen, wobei diese Rechnungen größenordnungsmäßig 10^{25} Instruktionen benötigen.
In der Computergrafik werden Szenerien, die künstlich generiert sind, durch Rendering in bewegte Bilder überführt. Mittlerweile so exakt, dass eine virtuelle Realität entsteht. Auch diese Berechnungen sind so aufwendig, dass sie in Echtzeit nicht von einem einzelnen Prozessor bewältigt werden können.
Bei der Crash-Simulation von Fahrzeugen versucht man vor dem Bau von Prototypen herauszufinden, ob eine bestimmte Fahrzeugkonstruktion Unfallsituationen gut übersteht. Auch in dieser Simulation werden vorwiegend Differentialgleichungen gelöst, so dass sie in eine ähnliche Kategorie von Anwendungen wie die vorher genannte Wettervorhersage fällt.
Als letztes Beispiel nennen wir die Atomwaffen-Simulation.

Bedeutung I

- **Jährlicher Leistungszuwachs von Mikroprozessoren wird kleiner**
- **Energieverbrauch bzw. Kühlung zur Energieableitung immer schwieriger**
- **Moore's Gesetz: Chipfläche wächst 40-50% pro Jahr**
- **Mehr Platz, aber kaum mehr Geschwindigkeit**
- **Zusätzliche Caches oder Funktionseinheiten ausgereitzt**

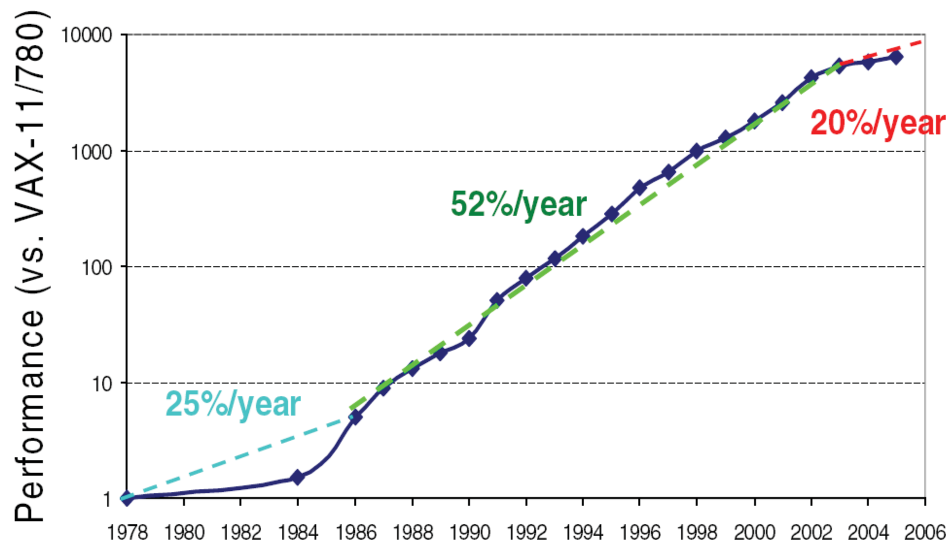
Weshalb wird Parallelverarbeitung immer wichtiger?
Wir stellen fest, dass der jährliche Leistungszuwachs von Mikroprozessoren immer kleiner wird.
Gleichzeitig wird der Energieverbrauch immer höher und dadurch die Kühlung zur Ableitung der in Wärme verwandelten aufgenommenen Energie immer schwieriger. Gordon Moore´s Gesetz, das eigentlich eine empirische Tatsache darstellt, gibt an, dass die verfügbare Chipfläche eines Prozessors pro Jahr um 40-50 % wächst.
So haben wir auf einem Prozessorchip immer mehr Platz, aber beim Bau eines einzelnen Prozessors können wir dieses "Mehr" an Platz kaum noch dazu nutzen, mehr Leistung zu erzielen, denn der Zuwachs durch zusätzliche Caches oder Funktionseinheiten gilt weitestgehend als ausgereizt.

Bedeutung II

- **Extraktion von Parallelität aus sequentiellen Programmen nur sehr beschränkt**
- **Ausweg: parallele Hardware + parallele Programme nicht nur für Spezialanwendungen, sondern auch für Desktops und Notebooks!**
- **Parallele Hardware: Multi- und Manycores**
- **Parallele Programmierung: Betrifft plötzlich (fast) alle Programmierer und Anwender**

Man könnte nun auf den Gedanken kommen, dass man einen großen Prozessorchip dazu benutzen kann, um nicht nur einen Prozessor, sondern mehrere Prozessoren in einen Chip zu implementieren. Dies stellen wir derzeit gerade fest. 2fach, 4fach, 6fach-Prozessoren treffen auf den Markt. Allerdings sind die verfügbaren Programme weiterhin sequentiell.
Selbst wenn wir den Quellcode dieser Anwendungsprogramme hätten gibt es keine Werkzeuge, die solche Programme automatisch und effizient parallelisieren, denn die Extraktion von Parallelität aus sequentiellen Programmen ist nur sehr beschränkt möglich.
Um also weiterhin steigende Leistungen von Rechnern vorweisen zu können, gibt es nur einen Ausweg: Parallele Hardware und parallele Programme.
Und dies nicht mehr nur für Spezialanwendungen wie die des zuvor genannten High-Performance Computing, sondern auch auf Desktoprechnern und Notebooks.
Damit betrifft, nachdem parallele Hardware in Form von Multi- und Manycores vorliegt, die parallele Programmierung nun fast alle Programmierer und Anwender. Damit Sie dieser Herausforderung begegnen können belegen Sie diesen Kurs.

Bedeutung III

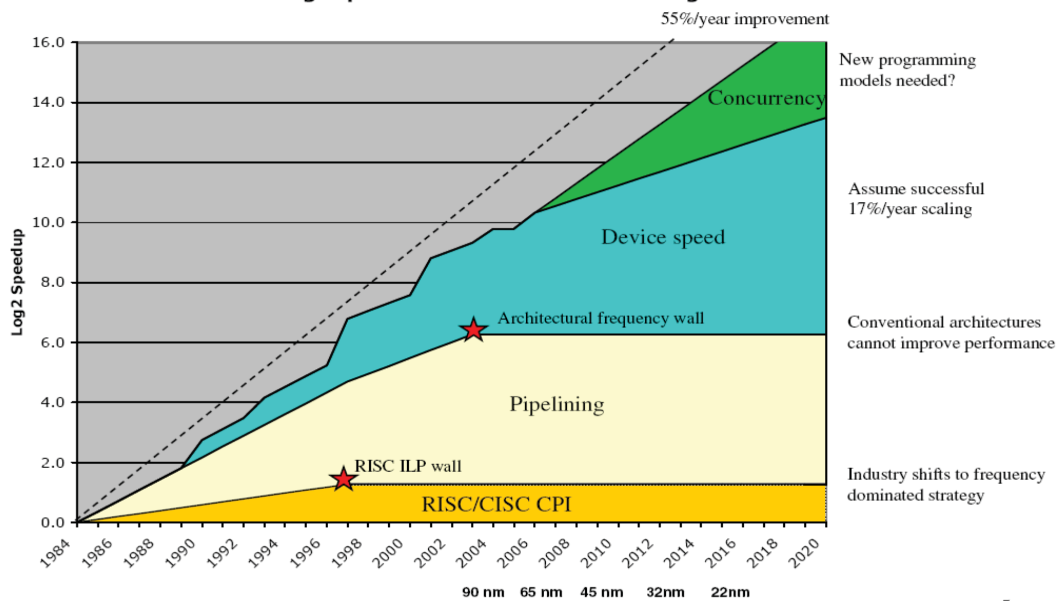


Wir wollen die zuvor geführte Diskussion um die Bedeutung des Parallelen Rechnens an einigen Graphen verdeutlichen:
Wir sehen hier in der Graphik die Leistung von Prozessoren vom Jahr 1978 – 2006 gemessen relativ zu einer damals verfügbaren Maschine, der VAX-11/780. Wir stellen fest, dass zunächst der Anstieg sehr langsam ging, dann über eine Vielzahl von Jahren (nämlich etwa bis 2002 oder 2003) sehr schnell nämlich um 52 % pro Jahr, dass aber in den vergangenen 5-6 Jahren dieser Leistungsanstieg deutlich abgeflacht ist. Dies gilt uns als Beleg, dass an der Front der Parallelverarbeitung tatsächlich Handlungsbedarf herrscht.

FernUniversität in Hagen

Department of Mathematics and Computer Science

Bedeutung IV



Slide 10 Course 01727 Parallel Programming

Parallelism and VLSI Group

Prof. Dr. J. Keller

Die folgende Folie illustriert, woher in den verschiedenen Jahren die Leistungszuwächse kamen.
Einen kleinen Teil des Leistungszuwachses ersehen wir aus verbesserten RISC und CISC Architekturen. Dies ist der unten sichtbare, kleine, gelbe Streifen.
Einen relativ großen Sprung schaffte das Pipelining und insbesondere auch die Superskalarität. Allerdings stellen wir fest, dass ab einem gewissen Punkt, der hier im Jahre 2004 markiert ist, diese Techniken nicht weiter zu einer Erhöhung führen, da ein "mehr" an Caches oder ein "mehr" an Funktionseinheiten in sequentiellen Prozessoren mit den Abhängigkeiten sequentieller Programme keinen Zugewinn mehr bringt.
Der in blau gezeichnete Zuwachs wird erhalten durch die Geschwindigkeit des Prozessors. Durch zunehmende Verkleinerung der Transistoren war es möglich, mehr Transistoren auf einem Prozessor unterzubringen. Und diese Transistoren mit einer höheren Frequenz betreiben zu können. Allerdings führt diese Erhöhung der Geschwindigkeit mittlerweile zu großen Problemen. Die Erhöhung der Frequenzen treibt den Energieverbrauch und damit das Kühlungsproblem in die Höhe. Die Erhöhung der Frequenzen verbunden mit der gleichzeitigen Vergrößerung von Chips führt aber auch zu dem Problem, dass es immer schwieriger wird, auf einem Prozessor überhaupt eine synchrone Clockdomäne aufrecht zu erhalten, d.h. den Takt so zu verteilen, dass der Prozessor insgesamt noch als synchroner Automat arbeiten kann, ohne dass die maximal mögliche Taktrate hierdurch beeinflusst wird. Wir sehen also, dass etwa ab 2008 neue Wege notwendig sind, um den bisherigen Zugewinn von 55 % Leistung pro Jahr

aufrecht zu erhalten. Die einzige Technik, die sich derzeit anbietet um dies aufrecht zu erhalten ist die konkurrierende oder Parallelverarbeitung. Um diese nutzen zu können, ist es allerdings notwendig, explizit parallele Programme zu schreiben und entsprechende Programmiermodelle hierfür zu entwickeln.

FernUniversität in Hagen — Department of Mathematics and Computer Science

Klassifikation Flynn

- **Anzahl der Instruktions- und Datenströme**
- **Single/Multiple Instruction/Data Streams**
- **SISD: sequentielle Rechner**
- **SIMD: Vektorprozessoren, VLIW Prozessoren**
- **MIMD: Jeder Prozessor oder Thread hat Programmzähler**
- **Spezialfall: SPMD single program multiple data**

Parallelism and VLSI Group
Prof. Dr. J. Keller

Um die Vielzahl von Vorschlägen für verschiedenste Rechner überhaupt vergleichen zu können, hat Flynn bereits relativ früh eine Klassifikation von Rechnern vorgelegt, die sich an der Anzahl der Instruktions- und der Datenströme in einem Rechner orientiert. Beide Arten, Instruktions- und Datenströme, können entweder einfach oder mehrfach vorhanden sein.
Man kürzt demzufolge die einzelnen Klassen mit vier Buchstaben ab.
SISD steht hierbei für Single Instruction Single Data Stream. Das ist ein sequentieller Rechner. Es wird eine Instruktion nach der anderen ausgeführt und gleichzeitig geht eine Instruktion auf ein Datenwort ein.
Die zweite von Flynn betrachtete Klasse sind SIMD Rechner (Single Instruction Multiple Data Streams). Ein erstes Beispiel für solche Rechner sind Vektorprozessoren. Hier arbeitet eine Instruktion wie eine Addition nicht nur um zwei Zahlen zu addieren, sondern hiermit werden zwei Vektoren aus Zahlen addiert, so dass man also eine Vielzahl von Datenströmen hat.
Als zweites Beispiel gelten die VLIW Prozessoren. In deren Very Long Instruction Word das gemeinsam eine Instruktion darstellt, sind viele Teilinstruktionen spezifiziert, die auf verschiedenen Daten arbeiten können.
Die dritte Klasse MISD (Multiple Instruction Single Data Stream) gilt als leer, so dass als letzte Möglichkeit, MIMD (Multiple Instruction Multiple Data Streams) bleibt. Wir haben mehrere Instruktionsströme, d.h. entweder mehrere Prozessoren oder mehrere Threads, die gleichzeitig ablaufen und von denen jeder einen eigenen Programmzähler hat und von denen jeder auf eigenen Daten arbeitet.

Ein populärer Spezialfall der MIMD Rechner sind die SPMD. SPMD bedeutet Single Program Multiple Data. Jeder der Prozessoren in der MIMD Maschine führt also das gleiche Programm aus, wenn auch jeder auf verschiedenen Daten. Allerdings kann jeder Prozessor in diesem Programm an einer verschiedenen Stelle sein. Damit lässt sich dieser Spezialfall auch in sein Gegenteil verkehren. Hätten wir einen MIMD Rechner, bei dem jeder Prozessor ein anderes Programm ausführt, dann könnten wir auch alle diese Programme in ein Programm zusammenfassen und zum Start dieses Programms eine Abfrage schalten, die abhängig von der Prozessornummer das gewünschte Programm nun als Unterprogramm aufruft.

FernUniversität in Hagen — Department of Mathematics and Computer Science

Klassifikation Speicherstruktur I

- **Getrennte Speicher (Distributed Memory)**
  Jeder Prozessor hat eigenen Speicher
  Prozessoren kommunizieren über Verbindungsnetzwerk mittels Nachrichten (Message passing)

- **Gemeinsame Speicher (Shared Memory)**
  Variante 1: symmetrische Multiprozessoren (SMP) auch UMA (uniform memory access) genannt

  Variante 2: verteilter gemeinsamer Speicher
  Distributed shared memory (DSM)
  auch NUMA (non-uniform memory access) genannt

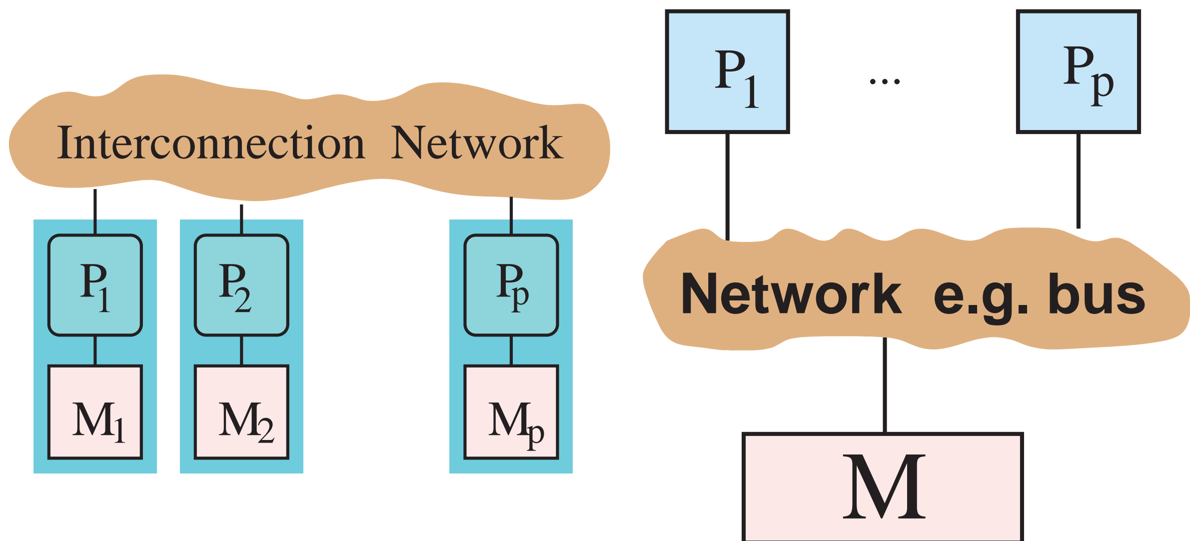
Während Flynn sich nur Gedanken über die Datenströme macht, gibt es eine weitere Klassifikation, die Rechner gemäß ihrer Speicherstruktur unterteilt.
Bei den so genannten Distributed Memory Rechnern hat jeder Prozessor seinen eigenen Speicher. Diese Prozessorspeicherpaare kommunizieren über ein Verbindungsnetzwerk explizit mittels Nachrichten. Die Programmierung solcher Rechner wird deshalb auch als Message passing programming bezeichnet. Je nach der Bandbreite und der Latenz dieses Verbindungsnetzwerkes ist die Kopplung dieser Prozessoren enger oder loser.
Die zweite Möglichkeit ist es, einen gemeinsamen Speicher zu haben. Dieses bezeichnet man als Shared Memory.
Variante 1 der Shared Memory Prozessoren sind die so genannten symmetrischen Multiprozessoren (SMP), die auch UMA Maschinen genannt werden. UMA steht hier für uniform memory access, dies bedeutet dass ein Speicherzugriff immer etwa gleich lang dauert egal von welchem Prozessor er ausgeführt wird und egal auf welche Speicherzelle er sich bezieht.
Die Variante 2 ist der so genannte verteilte gemeinsame Speicher, auch Distributed Shared Memory (DSM) genannt. Hierbei existiert nur noch ein gemeinsamer Adressraum, der aber verteilt implementiert ist. Solche Rechner werden auch NUMA-Rechner genannt. NUMA steht hier für non-uniform memory access, d.h. die Zeit zum Speicherzugriff ist nicht mehr gleichmäßig, sondern hängt davon ab, welcher Prozessor gerade auf welche Speicheradresse zugreift.

Klassifikation Speicherstruktur II

- **Bei DSM wird Zugriff auf nicht-lokale Adresse in Netzwerk-Kommunikation übersetzt, deshalb auch stark variierende Zugriffszeit**
- **Wenn bei DSM nicht-lokale Daten auch im Cache gehalten werden, Wahrung der Konsistenz notwendig: CC-NUMA**
- **Wenn Caches nicht konsistent: NCC-NUMA (eigentlich Mischform zwischen distributed und shared memory)**

Bei den Distributed Shared Memory Maschinen wird der Zugriff auf eine Adresse, die sich nicht im lokalen Speicher befindet, in eine Netzwerkkommunikation übersetzt, d.h. eine Anfrage an den Speicher, der die gesuchte Adresse enthält, der dann eine Antwort mit im Lesefall dem gewünschten Datum folgt. Deshalb variiert die Zugriffszeit abhängig von der Kommunikationszeit im Netzwerk sehr stark.
Da bei DSM Maschinen auch auf nicht-lokale Daten geschrieben werden kann, kommt es zur Notwendigkeit, die Konsistenz zu wahren, denn damit ist es möglich, dass Daten die man erst angefragt hat und sich lokal in der Form eines Caches behält, mittlerweile an ihrem Originalstandort nicht mehr gültig sein können. Maschinen, die die Konsistenz wahren, werden auch als CC-NUMA bezeichnet. CC steht hier für Cache Coherency.
Wenn die Caches nicht konsistent sind, dann spricht man von NCC-NUMA. Eigentlich ist NCC-NUMA eine Mischform zwischen distributed und shared memory Maschinen, denn das Einzige was hier noch shared ist, ist der Adressraum und das Einzige was automatisch passiert ist die Umsetzung des Zugriffs in eine Netzwerkkommunikation.

Klassifikation Speicherstruktur III



Wir sehen hier links das Beispiel einer Distributed Memory Maschine. Wir haben mehrere Prozessor-Speicher-Paare, d. h. im einfachsten Fall mehrere PCs, die über ein Verbindungsnetzwerk verbunden sind. Über die Hardware hinaus werden wir bei einem Parallelrechner aber auch annehmen, dass alle diese PCs mit ihrem Verbindungsnetzwerk unter einer gemeinsamen organisatorischen Kontrolle stehen, eine gemeinsame Betriebssystemversion haben und dass es möglich ist, von einer Stelle aus ein paralleles Programm auf diesem Parallelrechner zu starten.
Auf der rechten Seite sehen wir ein Shared Memory System. Wir haben hier eine Reihe von Prozessoren, die über ein Netzwerk, im einfachsten Falle ein Bus mit einem gemeinsamen Speicher verbunden sind. Dies zeigt auch schon eine Einschränkung der UMA-Architekturen. Damit der Bus nicht zum Engpass wird, ist die Anzahl der Prozessoren hier beschränkt.

Klassifikation Speicherstruktur IV

- **Beispiel UMA: bus-basierter Multiprozessor ohne Cache**
- **Beispiel CC-NUMA: SGI-Origin, Multicore Prozessoren**
- **Beispiel NCC-NUMA: Cray T3E**
- **DSM auch in Software möglich**
  seitenbasiert oder objektbasiert
  heißt auch: VSM, virtual shared memory

Eine weitere Einschränkung von UMA ist die Möglichkeit des Caching, denn im reinen Fall muss ein solcher bus-basierter Multiprozessor ohne Caches auskommen. Busbasierte Multiprozessoren mit Caches gelten immer noch als UMA, aber müssen bereits Cache Kohärenz realisieren und die Speicherzugriffszeit ist nicht mehr ganz gleichmäßig, sie hängt jetzt, wie allerdings auch in einem sequenziellen Rechner auch, von der Frage Cache-Zugriff oder Cache-Fehler ab.
Ein bekanntes Beispiel für eine CC-NUMA Maschine ist der SGI-Origin oder auch Multicore-Prozessoren, die sich über mehrere Boards erstrecken. Damit sind nämlich ihre Speicher getrennt und es ist notwendig, dass sie über ein Netzwerk kommunizieren.
Als Beispiel eines NCC-NUMA gilt die Cray T3E, die allerdings mehr als zehn Jahre alt ist.
Distributed Shared Memory muss nicht unbedingt in Hardware realisiert werden, auch eine Realisierung in Software ist möglich und es gibt Beispiele hierfür.
Entweder übersetzt man die Adressen seitenbasiert oder objektbasiert, wobei die erste Variante häufiger realisiert wurde. Eine solche Software Distributed Shared Memory Realisierung wird auch als Virtual Shared Memory bezeichnet.

Klassifikation Speicherstruktur V

- **Achtung!**

  Shared Memory und Message Passing sowohl als Hardware-Struktur oder als Sicht der Software möglich

- **Mix ist möglich, erhöht Portierbarkeit**

  Message passing Programmierung auf Shared memory Hardware: Nachrichtenpuffer im SM

  Shared memory Programmierung auf Message Passing Hardware: user-space VSM

Es ist bei der Betrachtungsweise, die Shared Memory und Message Passing unterscheidet, allerdings Vorsicht geboten.

Man kann diese Unterscheidung nämlich sowohl auf der Hardware-Struktur als auch auf der Software durchführen, d.h. man kann einen Rechner haben, der physikalisch einen Shared Memory bereitstellt bzw. physikalisch verteilten Speicher und Message Passing anbietet oder man kann eine Programmierumgebung haben, die dem Benutzer die Programmierung mit einem Shared Memory oder die Programmierung als einer Reihe von Prozessen die Nachrichten austauschen darstellt.

Weiterhin ist ein Mix möglich. Mit einem Mix meinen wir hier nicht, dass man Shared Memory und Message Passing gleichzeitig anwenden kann, auch wenn dies ebenfalls vorkommt. Wir meinen mit einem Mix, dass die Sicht der Software und die Hardware-Struktur nicht übereinstimmen, z. B. kann man auf einer Shared Memory Hardware auch Message Passing Programmierung durchführen. Offensichtlich ist es in einem solchen Rechner kein Problem, mehrere Prozesse gleichzeitig laufen zu lassen und die Puffer, über die die Nachrichten ausgetauscht werden, befinden sich eben im Shared Memory.

Andersrum kann auch auf einer Message Passing Hardware eine Shared Memory Programmierung durchgeführt werden, d.h. dem Benutzer wird lediglich noch ein gemeinsamer Adressraum vorgegaukelt, der dann durch die Software aufgeteilt wird und Speicheranfragen in diesem gemeinsamen Adressraum in Nachrichten umgesetzt

werden, d.h. Shared Memory Programmierung auf einer Message Passing Hardware ist nichts anderes als ein Virtual Shared Memory im User-Space der Software. Dass man auf jeder möglichen Hardware im Prinzip beide Arten der Programmierung durchführen kann, erhöht die Portierbarkeit eines Programms auf eine größere Anzahl von Plattformen. Gleichzeitig erweitert es auch die Einsetzbarkeit jeder Plattform, allerdings, und hiermit kommen wir auf das oben betrachtete Achtung zurück, dies gilt funktional. Im Bereich der Performance gilt typischerweise, dass die gemixte Benutzung wesentlich langsamer ist als die Originalbenutzung. Eine Ausnahme könnte Message Passing auf Shared Memory Hardware bieten, weil der Nachrichtenaustausch über das Shared Memory schneller gehen könnte als der Austausch einer Nachricht über ein Verbindungsnetzwerk, wenn dieses (wie z. B. beim 100 MBit Ethernet der Fall) keine sehr guten Latenzeigenschaften hat.

Modelle für Parallelrechner I

- **Parallel Random Access Machine PRAM [Fortune,Wyllie'78]**
- **Äquivalent der Random Access Machine (RAM) der Algorithmenanalyse**
- **Zweck: Entwurf und Analyse paralleler Algorithmen unabhängig von konkreter Hardware**
- **UMA mit synchron laufenden Prozessoren, konstante Zugriffszeit trotz beliebig vieler Prozessoren**

Wir wollen im nächsten Teil unserer Vorlesung über Modelle für Parallelrechner reden.

Eines der ersten Modelle aus der Theoretischen Informatik ist die Parallel Random Access Machine abgekürzt PRAM. Sie wurde von Fortune und Wyllie bereits 1978 eingeführt, ist also schon sehr alt.

Sie ähnelt sehr und nicht nur im Namen der Random Access Machine RAM aus der Algorithmenanalyse.

Auch der Zweck ist ähnlich. Man möchte parallele Algorithmen entwerfen und analysieren können, ohne dabei an die Besonderheiten einer konkreten Hard- und Softwarestruktur gebunden zu sein. Gleiches gilt auch für die Random Access Machine, die, obwohl es in ihr weder Caches noch Superskalarität gibt, sehr erfolgreich bei der Analyse sequenzieller Algorithmen gewirkt hat.

Die Parallel Random Access Machine ist eine UMA mit synchron laufenden Prozessoren, d.h. wir haben eine Vielzahl von Prozessoren, die auf einen gemeinsamen Speicher zugreifen. Die Prozessoren führen je nach Variante Maschineninstruktionen oder sogar Hochsprachenstatements synchron aus. Die Zugriffszeit ist für alle Prozessoren und alle Speicherzellen gleich und zwar ist sie konstant, obwohl die Zahl der Prozessoren beliebig groß sein kann. Wir sehen also, dass die PRAM genauso wie die RAM stark von einer konkreten parallelen Maschine abstrahiert.

Modelle für Parallelrechner II

- **Viele PRAM-Varianten bezüglich erlaubter Zugriffsmuster**
- **Exclusive Read Exclusive Write (EREW-PRAM)**
  in einem Schritt darf eine Speicherzelle nur von einem Prozessor gelesen oder geschrieben werden
- **Concurrent Read Exclusive Write (CREW-PRAM)**
  Lesen durch mehrere Proz.en in einem Schritt erlaubt
- **Concurrent Read Concurrent Write (CRCW-PRAM)**
  Auch Schreiben durch mehrere Prozessoren erlaubt

Im Laufe der Zeit haben sich viele PRAM-Varianten ausgebildet. Die meisten unterscheiden sich bzgl. der erlaubten Zugriffsmuster im Speicher.

Offensichtlich ist die Exclusive Read Exclusive Write-PRAM. Dies bedeutet, dass in einem Schritt eine Speicherzelle nur von einem Prozessor gelesen oder geschrieben werden darf.

Etwas komfortabler ist die Concurrent Read Exclusive Write-PRAM. Hier ist auch das Lesen durch mehrere Prozessoren in einem Schritt bei einer Zelle erlaubt. Egal wie viele Prozessoren eine Speicherzelle in einem Schritt auslesen, die Zeit für diesen Schritt bleibt gleich. Wir sehen bereits wie die Abkürzung der Varianten zusammengesetzt wird. Der erste Buchstabe sagt Exclusive oder Concurrent "E" oder "C", der zweite Buchstabe sagt, diese Variante gilt für Read also lesende Zugriffe, der dritte Buchstabe in diesem Fall jedes Mal "E" gibt wieder die Variante an und zwar dieses Mal für "W" Write die Schreibzugriffe.

Die nächste Variante, die man sich vorstellen könnte, wäre ERCW allerdings hat man für diese Variante bisher keine Algorithmen bisher gefunden, so dass als vierte Variante die Concurrent Read Concurrent Write-PRAM CRCW bleibt. In dieser Variante ist es erlaubt, dass eine Speicherzelle in einem Schritt durch mehrere Prozessoren beschrieben wird.

Modelle für Parallelrechner III

- **Welcher Prozessor gewinnt bei Concurrent Write?**
  Arbitrary = irgendeiner, Algorithmus muss damit umgehen können
  Common = alle konkurrierend schreibenden Prozessoren schreiben gleichen Wert (damit ist egal wer gewinnt)
  Priority = Prozessoren haben lineare Ordnung (Priorität), der Proz mit höchster Priorität gewinnt

- **Auch eingeschränkte Modelle wie**
  Owner write (z.B. CROW-PRAM): jede Speicherzelle hat einen Prozessor als Eigentümer, nur der darf schreiben

Damit stellt sich die Frage welcher der Prozessoren bei einem solchen Concurrent Write gewinnt. Auch hier haben sich mehrere Varianten ausgebildet. Bei der Arbitrary CRCW-PRAM gewinnt irgendeiner der Prozessoren. Man weiß nicht welcher, bei einer Wiederholung könnte es auch ein anderer sein. Der ausgeführte Algorithmus muss mit diesem Verhalten umgehen können.

Bei der Common CRCW-PRAM müssen alle Prozessoren, die in einem Schritt die gleiche Speicherzelle beschreiben den gleichen Wert schreiben. Damit ist es egal, welcher Prozessor gewinnt, Hauptsache einer schreibt überhaupt. Dafür sorgen, dass alle Prozessoren jeweils im gleichen Schritt in eine Zelle den gleichen Wert schreiben, muss der Algorithmus.

Bei der Priority CRCW-PRAM gibt es unter den Prozessoren eine lineare Ordnung, die man Priorität nennt, und der Prozessor mit der höchsten Priorität gewinnt beim Concurrent Write. Im einfachsten Fall kann man sich vorstellen, dass die lineare Ordnung auf den Prozessor IDs definiert ist und dass der teilnehmende Prozessor mit der größten Prozessor ID gewinnt.

Schließlich gibt es auch eingeschränkte PRAM-Modelle. Bekannt geworden ist ein Modell, bei dem das Schreiben eingeschränkt ist, nämlich die Owner Write-PRAM, typischerweise zusammen mit einem Concurrent Read also CROW. In dieser Variante hat jede Speicherzelle einen Prozessor als Eigentümer und nur dieser Prozessor darf in

die Speicherzelle schreiben. Man kann sich dieses eingeschränkte Modell so vorstellen, als wäre jede Speicherzelle lokal zu einem Prozessor, d.h. der Speicher tatsächlich aufgeteilt. Lesende Anfragen aus der Ferne darf es geben, geschrieben werden darf nur lokal.

Aus der Sicht eines parallelen Algorithmus hat selbstverständlich das aufwendigste Modell nämlich die CRCW-PRAM die meisten Vorteile, denn man kann viele Dinge nun einfacher formulieren. Allerdings ist bei diesen Varianten auch am fraglichsten, wie man sie in die Praxis umsetzen soll.

FernUniversität in Hagen

Department of Mathematics and Computer Science

Modelle für Parallelrechner IV

- **Beispiele**
  Logisches Oder von n binären Variablen mit n Proz in O(1) Schritten:
  Jeder Proz liest eine Variable, falls Wert=1, dann schreibt er 1 in Resultat. Geht mit common ERCW

  Summierung von n Zahlen mit n/log n Prozessoren:
  Jeder Prozessor summiert log n Zahlen auf, damit n/log n Zwischenergebnisse
  Prozessoren bilden Baum und summieren in O(log n) Schritten

Slide 20 Course 01727 Parallel Programming

Parallelism and VLSI Group

Prof. Dr. J. Keller

Wir wollen nun zwei Beispiele für PRAM-Algorithmen angeben. Nehmen wir an wir haben n binäre Variablen, d.h. Variablen die nur die Werte 0 oder 1 annehmen können, in n Speicherzellen unseres Shared Memory und wir möchten nun von diesen n Variablen das logische Oder bilden. Wenn wir n Prozessoren haben können wir dies in einer konstanten Anzahl von Schritten tun, dabei nutzen wir das konkurrierende Schreiben. Nämlich jeder der Prozessoren liest eine Variable. Falls der Wert dieser Variable = 1 ist, dann schreibt er eine 1 in die Resultatzelle. Natürlich muss die Resultatzelle dafür mit 0 vorinitialisiert sein. Sind nun alle n Variablen gleich 0, so schreibt kein Prozessor und das Ergebnis bleibt 0. Hat mindestens eine dieser binären Variablen den Wert 1 dann schreibt ein Prozessor eine 1 ins Resultat. Haben mehrere Variablen den Wert 1, dann kommt es zum konkurrierenden Schreiben. Alle Prozessoren die schreiben, schreiben aber den gleichen Wert 1, d.h. wir benutzen hier eine Common CRCW-PRAM Variante. Wenn wir genau hinschauen, stellen wir fest, dass es allerdings gar kein konkurrierendes Lesen gibt, d.h. eigentlich kommen wir mit einer ERCW-PRAM aus.

Das zweite Beispiel bezieht sich auf die Bildung der Summe aus n Zahlen, die sich in n Speicherzellen des Shared Memory befinden. Wir haben in diesem Fall allerdings nicht n Prozessoren zur Verfügung, sondern nur n/log n. Hierbei wollen wir annehmen, dass n eine Zweierpotenz ist, so dass log n, genauer Logarithmus zur Basis 2 von n, eine Ganzzahl ist und schließlich dass auch n/log n eine Ganzzahl ist. In einer Vorberechnung summiert jeder der Prozessoren log n der Zahlen auf. Damit erhalten wir n/log n

Zwischenergebnisse. Nun haben wir so viele Zwischenergebnisse wie Prozessoren. Nun bilden die Prozessoren einen Baum in der Zeit und summieren, d.h. am Anfang brauchen wir n/(2*log n) Prozessoren, also halb so viele wie Zwischenergebnisse vorliegen, jeder Prozessor addiert zwei der Zwischenergebnisse auf. Nun haben wir noch n/(2*log n) Zwischenergebnisse. Im nächsten Schritt brauchen wir dann nur noch n/(4*log n) Prozessoren, die jeweils wiederum zwei der Zwischenergebnisse aufaddieren und so weiter und so weiter bis am Schluss nur noch ein Ergebnis übrig ist. Da sich die Anzahl der Zwischenergebnisse in jedem Schritt halbiert, brauchen wir insgesamt logarithmisch viele Schritte.

FernUniversität in Hagen

Department of Mathematics and Computer Science

Modelle für Parallelrechner V

- **Bewertung von PRAM-Algorithmen**
  Laufzeit
  Processor-Time-Product = Laufzeit*Anzahl von Proz.
- **Algorithmus optimal falls PTP = O(seq. Laufzeit)**
- **Kritik an PRAM:**
  gut für Algorithmenanalyse, aber nicht sehr realistisch
  da Proz nicht synchron und kein skalierbares SM
- **Heute: GPUs ähneln PRAMs**

Zur Bewertung von PRAM-Algorithmen dient zunächst einmal die Laufzeit und auch die Anzahl der Prozessoren, die zum Erreichen dieser Laufzeit gebraucht werden. Man fasst diese beiden Maße zusammen im Processor-Time-Product (PTP). Wenn man nämlich die Laufzeit des Algorithmus mit der Anzahl der Prozessoren multipliziert, erhält man die insgesamt von der PRAM aufgewendete Rechenzeit. Diese Rechenzeit kann niemals besser sein als die sequentielle Laufzeit eines Programms für das gegebene Problem, denn andernfalls könnte man den parallelen Algorithmus sequentiell simulieren.

Erreicht der parallele Algorithmus mit seinem Processor-Time-Product die sequentielle Laufzeit oder zumindest O(sequentieller Laufzeit), d.h. bis auf einen konstanten Faktor, dann ist der Algorithmus PTP-optimal.

An der PRAM gab es viel Kritik. Sie sei gut für die Algorithmenanalyse, aber nicht sehr realistisch, da Prozessoren in echten Parallelrechnern nicht synchron seien und es kein skalierbares Shared Memory gäbe.

Heute stellt sich allerdings heraus, dass massiv parallele Grafikprozessoren, die GPUs, in ihrer Struktur einer PRAM sehr stark ähneln. Allerdings steckt die Überführung von PRAM-Algorithmen auf GPUs noch in den Kinderschuhen, so dass wir uns diesem Thema in der gegenwärtigen Auflage der Vorlesung noch nicht zuwenden können.

Modelle für Parallelrechner VI

- **BSP-Modell [Valiant 1990]**
- **BSP = Bulk Synchronous Parallelism**
  sollte ähnlich wie PRAM arbeiten, aber besser realisierbar sein
- **Zeit unterteilt in Reihe von Supersteps**
  In jedem Superstep arbeitet Prozessor auf lokalem Speicher und kann Anfrage an andere Speicher stellen
  Ergebnis der Anfrage erst im nächsten Superstep verfügbar

Ein Modell, das als Antwort auf die Kritik an der PRAM von Leslie Valiant 1990 publiziert wurde, ist das BSP-Modell. BSP steht hier für Bulk Synchronous Parallelism. Es sollte ähnlich wie die PRAM arbeiten, aber besser realisierbar sein.

Im BSP-Modell ist die Zeit unterteilt in einer Reihe von so genannten Supersteps. In jedem Superstep arbeitet ein Prozessor auf seinem lokalen Speicher und kann Anfragen sowohl lesend als auch schreibend an andere Speicher stellen. Das Ergebnis einer solchen Anfrage ist allerdings erst am Anfang des nächsten Supersteps verfügbar. Das BSP-Modell modelliert also die Zeit, die man zum Zugriff auf einen fernen Speicher braucht und das BSP-Modell honoriert Arbeiten auf lokalem Speicher. Gleichzeitig erlaubt die Unterteilung von Supersteps aber trotzdem eine Vielzahl von Synchronisierungspunkten, an denen von allen Prozessoren wieder klar ist, dass sie sie erreicht haben.

Modelle für Parallelrechner VII

- **BSP-Modell hängt sehr von Dauer des Superstep ab**
- **Entwicklung von Bibliotheken für grundlegende parallele Algorithmen**
- **Entwicklung eigener Algorithmen schwierig**

Die konkrete Ausgestaltung des BSP-Modells hängt sehr von der Dauer des Supersteps ab. Man ist darauf angewiesen, dass jeder der Prozessoren in einem Superstep etwa gleich viel zu tun hat und dass die Zeit, die die Prozessoren dafür brauchen, jedenfalls so groß ist dass die Kommunikation zwischen den Prozessoren in dieser Zeit auch abgewickelt werden kann.

Man hat in den 90-er Jahren Bibliotheken für grundlegende parallele Algorithmen im BSP-Modell entwickelt. Für Experten ist dieses Modell also ein guter Ansatzpunkt, um auf realistischen Parallelrechnern portierbare parallele Algorithmen unterzubringen. Allerdings ist die Entwicklung eigener Algorithmen schwierig und aufwendig.

FernUniversität in Hagen — Department of Mathematics and Computer Science

Modelle für Parallelrechner VIII

- **LogP-Modell [Culler et al 1993]**
- **Message-Passing Rechner beschrieben durch 4 Parameter:**
  L = Latenz, Zeit zur Übermittlung einer Nachricht (1 Wort)
  o = Overhead, Berechnungszeit zum Absetzen einer Nachr.
  g = Gap, Zeitspanne die zwischen dem Absetzen zweier Nachrichten vergehen muss
  P = Anzahl der Prozessoren
- **Beschreibung halbwegs realistisch, allerdings steigt in der Praxis Latenz nur langsam mit Nachrichtengröße**

Ein weiteres Modell, das sich noch näher an realistische Parallelrechner anlegt ist das so genannte LogP-Modell von David Culler und Kollegen, etwa 1993 veröffentlicht. Ein Message Passing Rechner wird in diesem Modell durch vier Parameter beschrieben. Message Passing Rechner wurden gewählt, weil man diese Rechner zu jener Zeit für am praktikabelsten und am besten in großen Prozessorzahlen realisierbar hielt. Der erste Parameter "L" ist die Latenz, d.h. die Zeit zur Übermittlung einer Nachricht, die aus einem Wort besteht, von einem Prozessor zum Anderen. Der zweite Parameter ist "o" der Overhead, das ist die Zeit, die ein Prozessor braucht um eine Nachricht absetzen zu können. Der dritte Parameter "g" ist der Gap, dies ist die Zeitspanne, die zwischen dem Absetzen zweier Nachrichten vergehen muss. Schließlich "P" die Anzahl der Prozessoren.

Der Parameter "L" modelliert die Leistungsfähigkeit des Verbindungsnetzwerkes zwischen den Prozessoren. Allerdings nur gemittelt, denn in der Praxis hängt die Latenz einer Nachricht natürlich auch davon ab, wie viele andere Nachrichten zur gleichen Zeit in diesem Netzwerk unterwegs sind. Der Overhead modelliert den Softwareaufwand, der notwendig ist, sei es durch Umkopieren von Puffern, sei es durch Betriebssystemaufrufe, sei es durch Code der in der Netzwerkkarte ausgeführt wird. Die Gap modelliert die Bandbreite, die ein Prozessor zum Netzwerk hat, denn sicherlich kann ein Prozessor in einer bestimmten Zeit nicht beliebig viele Nachrichten ins Netzwerk absetzen. Man sieht, dass die Beschreibung halbwegs realistisch ist. Allerdings gibt es in diesem Modell nur Nachrichten aus einem Wort. Will man eine Nachricht aus mehreren Worten absenden, muss man sie in Einzelnachrichten verpacken und für das Versenden jeder Nachricht die Latenz und den Overhead einberechnen und die Gap. In der Praxis allerdings steigt die Latenz einer Nachricht nur sehr langsam mit deren Größe. Dies liegt daran, dass in vielen Verbindungsnetzwerken wie z. B. Ethernet die Rahmengröße einer Nachricht relativ groß sein kann.

Modelle für Parallelrechner IX

- **Zur Modellierung langer Nachrichten weiterer Parameter: LogGP-Modell**
- **Analyse von Algorithmen im LogP-Modell schwierig**
- **Laufzeitoptimierung von Algorithmen ist unintuitiv**

Als Reaktion hat man einen weiteren Parameter G eingeführt, um damit lange Nachrichten zu modellieren. Man hat also jetzt ein klein g die kleine Gap für kurze Nachrichten und das große G für die große Gap für große Nachrichten.

Die Analyse von Algorithmen in diesem LogP-Modell ist schwierig und die Laufzeitoptimierung von Algorithmen ist nicht sehr intuitiv. Sie erfordert damit sehr viel Erfahrung. Damit ist auch dieses Modell, obwohl es die Realität schon sehr gut wieder spiegelt, nur für Experten zu handhaben.

Modelle für Parallelrechner X

- **Fazit:**
  kein Modell hat sich bisher in der praktischen Analyse durchgesetzt

  PRAM-Modell hat große Verbreitung bei der theoretischen Analyse paralleler Algorithmen für Shared-Memory Maschinen
  s. auch Kurs 01824 Parallele Algorithmen

Zusammenfassend können wir sagen, dass sich Modelle für Parallelrechner bisher in der praktischen Analyse nicht wirklich durchgesetzt haben.

Das PRAM-Modell hat zwar eine große Verbreitung bei der theoretischen Analyse paralleler Algorithmen gefunden, speziell für Shared Memory Maschinen, z. B. ist in unserem Curriculum diesem Modell ein ganzer Kurs, nämlich 01824 Parallele Algorithmen, gewidmet. Gleichzeitig stellt sich aber die Übertragung dieser Algorithmen in die Praxis als schwierig da, weil die Prozessoren ständig synchronisiert werden müssten, was in gegenwärtigen Rechnern sehr aufwendig ist. Meistens benutzt man bei der Analyse eine mehr oder weniger sequentielle Sicht, die aber beim Zugriff auf das Shared Memory, z. B. zusätzlicher Aufwand dafür, dass mehrere Rechner gleichzeitig zugreifen, nur beschränkt tauglich sind. Wir werden also im Weiteren sehen, dass die Entwicklung paralleler Programme auch daran krankt, dass das Nachdenken über ihr zeitliches Verhalten schwierig ist. Das funktionale Verhalten ist in den beiden Kategorien Shared Memory und Message Passing bereits gut abgedeckt.