2. Parallele Rechnerarchitekturen

Parallele Algorithmen hängen stark von der verfügbaren parallelen Hardware ab ⇒

- kurze Übersicht über Parallelrechner
- theoretische Überlegungen zur Struktur paralleler Prozessoren
- Einfluss der Architektur auf die Programmierung
- Komponenten eines Parallelrechners
- Klassifizierungsschemata

<u>Gliederung</u>

2.1 Ebenen der Parallelität

Parallelität kann auf unterschiedlichen Ebenen auftreten und ist Teil jedes heutigen Rechners

Man unterscheidet

- implizite Parallelität im Prozessor
- explizite Parallelität von Parallelrechnern

<u>Parallelität in Mikroprozessoren</u>

Taktrate von Mikroprozessoren steigt um ca. 30% pro Jahr

Messungen an Benchmarks zeigen

- Steigerung der Integer Leistung von ca. 55% pro Jahr
- Steigerung der Floating Point Leistung von ca. 75% pro Jahr

⇒ zusätzliche Leistungssteigerung zu großen Teilen durch prozessorinterne Parallelität

Automatische Nutzung dieser Parallelität bei Abarbeitung von sequentiellem Code

<u>Parallelität auf Bitebene</u>

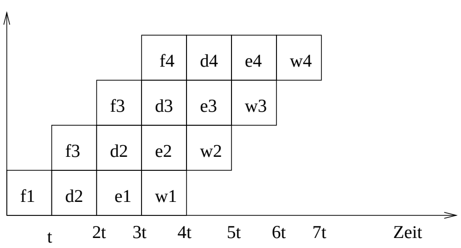
- Wortbreite eines Prozessors (heute i.d.R. 64 Bit)

<u>Parallelität durch Pipelining</u>

- Zerlegung von Instruktionen in Teilaufgaben und
- überlappende Abarbeitung der Teilaufgaben

Typische Zerlegung

- laden (fetch)
- dekodieren (decode)
- Adressbestimmung und Ausführung (execute)
- Zurückschreiben der Resultate (write back)



Einschränkungen

- Pipeline-Stufen mit ähnlicher Bearbeitungszeit
- Zahl möglicher Stufen beschränkt
  - Datenabhängigkeit
  - Sprungbefehle

<u>Parallelität durch mehrere Funktionseinheiten</u>

Superskalare Prozessoren

mehrere unabhängige Funktionseinheiten wie

- ALUs (arithmetic logical units)
- FPUs (floating point units)
- Speicherbänke
- Sprungeinheiten

⇒ parallele Ausführung von Instruktionen (inkl. paralleles laden/schreiben der Daten)

Grenzen des Einsatzes

- Datenabhängigkeit der Instruktionen (zur Laufzeit zu ermitteln!)
- Sprünge im sequentiellen Kontrollfluss

⇒ Messungen zeigen, dass maximal 4 Instruktionen pro Maschinenzyklus sinnvoll sind!

Werte heutiger Prozessoren

- AMD Opteron max. 2 parallele Instruktionen
- Intel Itanium 2 max. 4 parallele Instruktionen
- DEC Alpha 21264 max. 6 parallele Instruktionen

Parallelität im Prozessor wird in den sequentiellen Kontrollfluss eingefügt

⇒ höchstens indirekte Einflussnahme durch Programmierer

- explizite Parallelität im Kontrollfluss und
- deren Ausnutzung durch Parallelrechner/verteilte Systeme

⇒ Parallelrechner

Versuch einer Definition

"Ein Parallelrechner ist eine Ansammlung von Berechnungseinheiten (Prozessoren), die durch koordinierte Zusammenarbeit große Probleme schnell lösen können" [RaRü00]

sehr vage Definition (umfasst zahlreiche Architekturen)

Kennzeichen der Parallelität auf dieser Ebene
(im Gegensatz zur Prozessorebene)

Abarbeitung eines parallelen Programms

explizite (oder implizite) Berücksichtigung unabhängiger Teile im Algorithmus und deren (zumindest potenziellen) parallelen Abarbeitung

Zur konkreten Darstellung potenzieller Parallelität ist weitere Klassifizierung der Architektur notwendig

⇒ zahlreiche Klassifizierungen existieren, am weitesten verbreitet (wenn auch heute nicht mehr aktuell) ist Flynns Klassifizierung

Flynns Klassifizierung (1972)

Unterscheidung nach Struktur des Kontroll- und Datenflusses jeweils in sequentiell (single) - parallel (multiple)

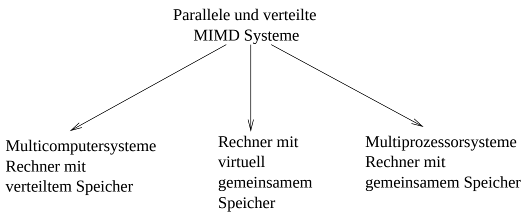
Folgende 4 Klassen entstehen

- SISD (Single Instruction Single Data)
  - klassischer von-Neumann Rechner
- MISD (Multiple Instruction Single Data)
  - ein Datum wird von mehreren Funktionseinheiten parallel bearbeitet
  - keine praktische Relevanz
- SIMD (Single Instruction Multiple Data)
  - synchrone Abarbeitung einer Instruktion auf unterschiedlichen Daten
  - Struktur der ersten Parallelrechner (Illiac IV, CM-1/2, MP-1/2)
  - einfach programmierbar durch Vektoroperationen
  - Spezialprozessoren
- MIMD (Multiple Instruction Multiple Data)
  - mehrere unabhängige Funktionseinheiten greifen unabhängig auf (gemeinsamen oder geteilten) Speicher zu
  - Architektur (fast aller) heutigen Parallelrechner
  - Standardprozessoren

Flynns Klassifizierung kaum noch hilfreich, da im wesentlichen MIMD Systeme vorhanden

⇒ weitere Unterteilung ist notwendig

Möglicher Ansatz



Weitere Kriterien einer Klassifizierung:

- Eigenschaften der verwendeten Prozessoren (Zahl, Typ, homogen-inhomogen)
- Speicherzugriff (verteilt-geteilt)
- Struktur der Verbindung zwischen Prozessoren (feste Verbindung, geschaltete Verbindung LAN)
- Art des Betriebssystems (Prozessoren transparent-nicht transparent)
- Organisation der Prozessoren (Master-Slave, Client-Server, Prozessorpool)

⇒ Strukturen beeinflussen verwendbare Programmiermodelle

2.2 Speicherorganisation von Parallelrechnern

weiter Unterteilung von MIMD Rechnern⇒ Speicherorganisation

physikalische Organisation -
Sicht des Programmierers auf den Speicher
(Sichten müssen nicht identisch sein!)

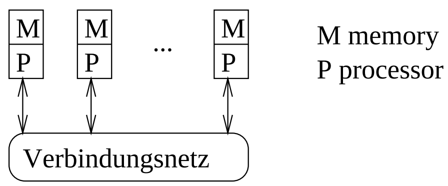
Sicht des Programmierers:
verteilter oder gemeinsamer Adressraum

Physikalische Struktur:
von eng gekoppelten Prozessoren
bis zu verteilten Workstations

Rechner mit physikalisch verteiltem Speicher

Physikalische Struktur:

- mittels Verbindungsnetzwerk verbundene Knoten
- jeder Knoten besteht aus Prozessor + Speicher + evtl. E/A-Peripherie



- Daten und Programme werden in lokalen Speichern abgelegt
- Zugriff auf lokale Speicher privat (d.h. nur vom zugehörigen Prozessor) oder per DMA (*direct memory access*) d.h. direkter Datentransfer zwischen Speicher und Netz ohne Prozessorbelastung
- Zugriff auf Daten im Speicher anderer Prozessoren über Nachrichtenaustausch (*message passing*)

Zur Struktur des Verbindungsnetzes siehe 2.3

Rechner mit verteiltem Speicher sind einfach zu realisieren

- aus Standardprozessoren mit Verbindungsnetzwerk
- aus Workstations/PCs mittels LAN verbunden
  (man spricht von einem *Cluster*, falls alle Rechner als Gesamtheit benutzt werden)

Auch bei struktureller Gleichheit zwischen Parallelrechnern und Workstationnetzen als MIMD Rechner ergeben sich Unterschiede durch

- Struktur des Betriebssystems
  ein Betriebssystem für den gesamten Parallelrechner - lokale (u.U. heterogene) Betriebssysteme für Workstationnetze
- Einprogrammbetrieb (pro Prozessor) auf dem Parallelrechner - Mehrbenutzerbetrieb im Workstationnetz

- Prozessor-Prozessor Verbindung im Parallelrechner
  geteiltes Kommunikationsmedium im Workstationnetz
- schnelle Kommunikation auf dem Parallelrechner - langsamere Kommunikation auf dem Workstationnetz
  (u.U. bis zu Faktor 100 langsamer!)

näheres zu Workstationnetzen siehe 2.5

Programmiermodell für diese Systeme im message passing Paradigma

- Programmierer muss lokale Verfügbarkeit von Daten sicherstellen
- Datentransfer durch explizite Sende- und Empfangsoperationen
- Datenverteilung beeinflusst Leistung des parallelen Algorithmus
- teilweise Architekturunabhängigkeit der Programmierung durch Verwendung von Standardbibliotheken

$\Rightarrow$ komplexe Programmierung (näheres in Kapitel 3)

<u>Rechner mit</u> <u>physikalisch gemeinsamem Speicher</u>

- gemeinsamer Adressraum, auf den alle Prozessoren lesend und schreibend zugreifen können
- Speicher kann aus mehreren Modulen bestehen
- Verbindung Prozessoren - Speicher über Verbindungsnetz

Variante SMP-Systeme (Symmetric Multiprocessor) mit
(geringer Zahl $\approx 32 - 64$) identischen Prozessoren
mit Busverbindung zum gemeinsamen Speicher

Speicherzugriffszeit ist oftmals Flaschenhals

Steigerung zwischen 1980 und 2000 pro Jahr

- Prozessorleistung FP 75%
- Prozessorleitung INT 55%
- Speicherkapazität 60%
- Speicherzugriffszeit 25%

Umgehung des Speicherengpasses durch

mehrere Threads auf einem Prozessor

- Wartezeit auf Speicherzugriff wird von anderem Thread als Rechenzeit genutzt
- Overhead durch Kontextwechsel und Speicherbedarf der Threads

Alternative: lokale Caches

- schnelle Speicher zwischen Prozessor und Speicher
- Ablage häufig benutzter Daten im Cache
- Blockweises lesen von Daten in den Cache

Problem der Cache-Kohärenz

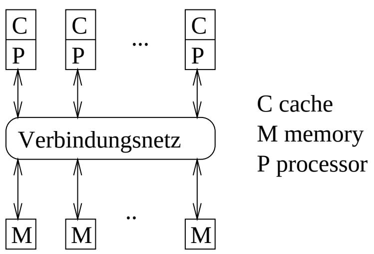
- falls Daten im Speicher/in einem Cache geändert werden, muss dies "gleichzeitig" in allen Caches, wo das Datum liegt geschehen ($\Rightarrow$ später mehr dazu)

Programmiermodelle bei gemeinsamem Speicher:

- gemeinsame Variablen, Threads etc.
- i.a. einfacher als Nachrichtenaustausch

Realisierungsvarianten für gemeinsamen Speicher

Uniform Memory Access (UMA)



Mögliche Verbindungsnetze

- Bus
- geschaltete Verbindungen (siehe 2.3)

Falls Prozessoren identisch und Verbindungsnetz Bus ⇒ SMP

- Speicherzugriffszeit unabhängig von Lage der Daten
- Schreibkonflikte müssen per Software ausgeschlossen werden

Nachteile von UMA Systemen

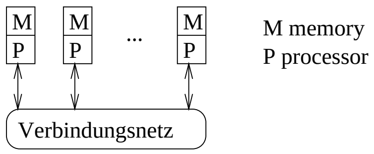
- geringe Skalierbarkeit (Verbindungsnetz wird zum Flaschenhals)

Non-uniform Memory Access (NUMA)

Gemeinsamer Adressraum aber Verteilung der Daten auf unterschiedliche Speichermodule

⇒Zugriffszeit abhängig von Datenlage

Oftmals Software-Realisierung auf
Systemen mit verteiltem Speicher



Programmiermodell der gemeinsamen Variablen
prinzipiell anwendbar auf NUMA Systemen

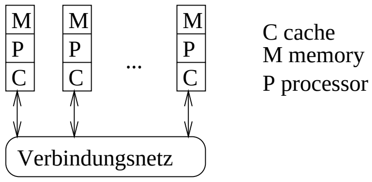
Gefahr langsamer Zugriffszeiten durch
ungünstige Lage der Daten

⇒ Optimierung der Datenverteilung durch Übersetzer-
Laufzeitsystem (oft problematisch)

Erweiterung des NUMA-Modells zu

Cache-Coherent NUMA (CC-NUMA)

- Knoten mit lokalen Speichern und Caches
- Cache nimmt Daten auf
  - aus lokalem Speicher
  - aus anderen Prozessoren zugeordneten Speichern
- Protokoll stellt Kohärenz der Daten im Cache sicher



- Problem der Lageabhängigkeit beim Speicherzugriff wird gemildert
- aber zusätzlicher Aufwand durch Kohärenzprotokoll

Teilweise existieren auch
Non-Coherent NUMA (NC-NUMA) Systeme

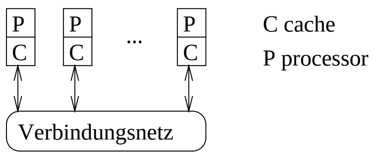
- Kohärenz (genaue Definition später) des Caches wird nur für lokale Daten sichergestellt
- bei Daten aus anderen Speicherbereichen muss Konsistenz oder Kohärenz vom Programm explizit gefordert werden (z.B. per message passing)

Gemeinsamkeit aller Architekturen:
feste Zuteilung von Daten zu Speicherbereichen

Falls dies nicht gefordert wird entstehen

Cache Only Memory Access (COMA) Systeme

- gesamter Speicher besteht aus Cache Speichern der einzelnen Knoten
- lokale Speicher enthalten
  - Teil der Daten
  - Cache Directory mit Angabe, welche Datenbereiche in welchem Cache liegen



Beim Datenzugriff werden

- logische Adressen in physikalische (Cache-)Adressen umgewandelt
- Daten in Cache des zugreifenden Prozessors geladen

von jedem Datum muss mindestens eine Kopie
in einem Cache sein ⇒

eine Kopie wird als primär markiert und kann nicht verdrängt werden

COMA für viele Anwendungen effizienter als NUMA aber

- Kohärenzprotokoll komplex
- ineffizient falls zwei Prozessoren auf Daten eines Blocks zugreifen
- da primäre Kopien nicht verdrängt werden dürfen, kann es zu Speicherengpässen kommen

<u>Caches und Speicherhierarchien</u>

schneller Speicher ist teuer und
kann nicht in beliebiger Größe realisiert werden

⇒ Verringerung der mittleren Zugriffszeit
durch Speicherhierarchien

(hier betrachte 2 Stufen (Cache-Hauptspeicher)
aber im Prinzip für beliebige Stufenzahl realisierbar)

Ziel:

Daten befinden sich beim Zugriff im schnellsten Speicher

⇒ möglichst vom Cacheprotokoll sicherzustellen

Bei Multiprozessoren zusätzlich:

Daten im Cache und Hauptspeicher müssen kohärent sein

Vorgehen beim Zugriff auf hierarchische Speichersysteme

- Prozessor setzt Lese- oder Schreibbefehl ab
- Speicherhierarchie für Prozessor unsichtbar
- Cache Controller führt Lese-/Schreibbefehl auf Cache aus
- falls Daten vorhanden direkter Zugriff im Cache (*cache hit*)
  - beim Schreiben evtl. Rückschreiben der geänderten Daten in Hauptspeicher
- falls Daten nicht vorhanden (*cache miss*)
  - lesen der Daten in den Cache
  - Ausführung der Lese-/Schreiboperation des Prozessors
  - beim Schreiben evtl. Rückschreiben der geänderten Daten in Hauptspeicher

Zugriffszeit Cache < Zugriffszeit Hauptspeicher

(Faktor 10 bei seq. Rechnern,
deutlich größere Unterschiede bei Multiprozessoren)

Effizienz abhängig vom Verhältnis $miss/hit$, damit abhängig von

- Cachegröße
- Zeitlicher-räumlicher Zugriffslokalität eines Programms

Blockweises Lesen von Daten in den Cache

Cache Assoziativität legt fest, wo welche Blöcke im Cache liegen dürfen

- *Direkt abgebildet*: jeder Speicherblock gehört in einen Cacheblock
- *Voll-assoziativ*: jeder Speicherblock kann in jedem Cacheblock liegen
- *Mengen-assoziativ*: Zwischenlösung

Blockersetzungsmethoden

Laden eines neuen Blocks in den Cache erfordert
u.U. ersetzen eines vorhandenen Blocks

Auswahl des zu ersetzenden Blocks:

- least recently used (LRU) Block auf den am längsten nicht zugegriffen wurde
- least frequently used (LFU) Block auf den am wenigsten oft zugegriffen wurde

in beiden Fällen müssen Zähler für Zugriffszeit/-zahl
realisiert werden (Zähler werden periodisch auf 0 gesetzt)

Rückschreibestrategien:

Geänderte Daten im Cache müssen auch im Hauptspeicher geändert werden

- Write-through Strategie: Jede Änderung sofort in den Hauptspeicher schreiben
  - Hauptspeicher enthält möglichst schnell aktuellen Wert
  - Aufwand höher durchlangsames Schreiben in den Hauptspeicher
- Write-back Strategie: Erst bei Blockersetzung in den Hauptspeicher schreiben
  - Reduktion der Zahl der Schreiboperationen
  - veraltete Daten im Hauptspeicher

Cache Kohärenz

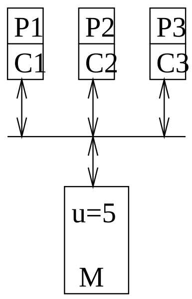
In Multiprozessoren kann ein Datum in mehreren Caches gleichzeitig sein

⇒ Werte in den Caches können sich unterscheiden

⇒ Threads auf unterschiedlichen Prozessoren rechnen mit unterschiedlichen Werten für eine Variable

Problem der Cache Kohärenz

Beispiel:



write-through Strategie

Ablauf eines Berechnung

t_1 P1 liest u
Block mit $u = 5$ wird in C1 geladen
t_2 P3 liest u
Block mit $u = 5$ wird in C3 geladen
t_3 P3 schreibt $u = 7$
Block mit $u = 7$ wird in M geschrieben
t_4 P1 liest $u = 5$ aus C1

⇒ zum Zeitpunkt t_4 liest $P1$ den Wert 5 statt 7 beim Zugriff auf u

Gleiches Ergebnis bei Verwendung von write-back

⇒ Speichersystem ist nicht kohärent!

Ein Speichersystem ist kohärent, wenn folgende Bedingungen gelten

1. Falls ein Prozessor eine Speicherzelle x zum Zeitpunkt t_1 beschreibt und zum Zeitpunkt $t_2 > t_1$ liest und kein anderer Prozessor den Wert von x im Intervall $[t_1, t_2]$ verändert hat $\Rightarrow$ Prozessor liest den von ihm geschriebenen Wert
2. Falls ein Prozessor P_1 zum Zeitpunkt t_1 Speicherzelle x schreibt, ein Prozessor P_2 Speicherzelle x zum Zeitpunkt $t_2 > t_1$ liest und kein anderer Prozessor den Wert von x im Intervall $[t_1, t_2]$ verändert hat $\Rightarrow$ Prozessor P_2 liest den von P_1 geschriebenen Wert, falls $t_2 - t_1$ genügend groß
3. Falls zwei Prozessoren die gleiche Speicherzelle schreiben, werden die Schreibzugriffe so sequentialisiert, dass alle Prozessoren die gleiche Schreibreihenfolge sehen

Methoden zur Sicherstellung von Cache-Kohärenz in busbasierten Systemen:

Bus-Snooping (für busbasierte SMP-Systemen)

- alle Prozessoren überwachen den Bus
- write-through Rückschreibestrategie
- jedes Schreiben in den Hauptspeicher führt zur Datenaktualisierung in allen relevanten Caches

Cache-Kohärenz in nicht-busbasierten Systemen

deutlich schwieriger zu realisieren

i.d.R. über verteilte Directory Struktur mit Zustandsbeschreibung der einzelnen Blöcke

Speicherkonsistenz

cache Kohärenz sagt aus, dass jeder Prozessor das gleiche Bild des Speichers hat

es wird nichts ausgesagt, wann und in welcher Reihenfolge Operationen sichtbar werden $\Rightarrow$ Speicherkonsistenz

Verschiedene Konsistenzmodelle existieren

Unterscheidungskriterien

- Werden Speicherzugriffsoperationen der einzelnen Prozessoren in deren lokaler Reihenfolge ausgeführt?
- Sehen alle Prozessoren die ausgeführten Speicherzugriffsoperationen in der gleichen Reihenfolge?

Beachte: Im Gegensatz zur Kohärenz werden nun alle und nicht mehr eine Speicherzelle betrachtet!

Konsistenzmodelle:

Sequentielles Konsistenzmodell (SC-Modell)

- jeder Prozessor führt lokal seine Operationen in der im Programm vorgegebenen Reihenfolge aus
- alle Prozessoren sehen die gleiche Ordnung von Zugriffsoperationen, die aus der Mischung der lokalen Operationsströme resultiert (totale Ordnung)

Hinreichende Bedingungen für sequentielle Konsistenz

1. Jeder Prozessor setzt seine Speicherabfragen in Programmreihenfolge ab
2. Nach Absetzen einer Schreiboperation wartet Prozessor bis zum Ausführungsende, bevor die nächste Speicherabfrage abgesetzt wird
3. Nach Absetzen einer Leseoperation wartet ein Prozessor bis die Leseoperation und alle daraus resultierenden Schreiboperationen abgeschlossen und für alle anderen Prozessoren sichtbar sind

$\Rightarrow$ Bedingungen leicht realisierbar aber ineffiziente Programme

(langes Warten bis zur Beendigung von Speicherzugriffsoperationen)

$\Rightarrow$ Abgeschwächte Konsistenzmodelle

- $R \rightarrow R$ nur Lesezugriffe erfolgen in Programmreihenfolge
- $R \rightarrow W$ Lese- und nachfolgende Schreiboperation erfolgen in Programmreihenfolge
- $W \rightarrow W$ Schreibzugriffe erfolgen in Programmreihenfolge
- $W \rightarrow R$ Schreib- und nachfolgende Leseoperation erfolgen in Programmreihenfolge

Abgeschwächte Konsistenz verzichtet auf einzelne Bedingungen

TSO-Modell verzichtet auf 4. Bedingung

- keine Garantie, dass Schreiboperation beendet, wenn Leseoperation erfolgt
- Reduktion der Latenzzeit von Schreiboperationen
- Realisierung bei Sparc Prozessoren von Sun

PSO-Modell verzichtet auf 3. und 4. Bedingung

- keine Garantie, dass Schreiboperationen reihenfolgetreu erfolgen
- Ausnahme Schreiboperationen für eine Speicherzelle (Kohärenz gilt)

2.3 Verbindungsnetzwerke

Realisierung der physikalischen Verbindung der Knoten eines Parallelrechners durch Verbindungsnetzwerk

Man unterscheidet

- statische Verbindungsnetze
  (Punkt zu Punkt Verbindungen)
- geschaltete Verbindungsnetze (Busse, Switches, LANs)

<u>Statische Verbindungsnetze</u>

Gestaltungkriterien: Topologie und Routingtechnik

- Topologie beschreibt die geometrische Struktur, mit der Prozessoren und Speicher miteinander verbunden sind
- Routingtechnik beschreibt entlang welchen Pfades Nachrichten von einem Sender zu einem festen Ziel übertragen werden

Topologie wird durch Graphen $G = (V, E)$ mit

- Knotenmenge V (Prozessoren)
- Kantenmenge E (Leitungen zwischen Prozessoren)

Also $(u, v) \in E \Rightarrow$

es existiert eine direkte Verbindung zwischen u und v

Verbindungen sind bidirektional (Graph ungerichtet)

Falls eine Nachricht von u nach v gesendet werden soll und keine direkte Verbindung zwischen den Prozessoren existiert $\Rightarrow$

$$\text{Pfad } (v_0, v_1, \ldots, v_k)$$

mit $v_0 = u$, $v_k = v$ und $(v_i, v_{i+1}) \in E$ ist zu wählen

Für jedes sinnvolle Verbindungsnetz muss zwischen je zwei Knoten ein Pfad existieren

Bewertungskriterien und -maße

für statische Verbindungsnetze (auf Basis des Graphen G)

- Kantenzahl $k(N)$:= Anzahl benötigte Leitungen
- Symmetrie := symmetrische Topologien sehen aus Sicht jedes Knotens identisch aus
- Grad $g(G)$:= maximale Kantenzahl eines Knotens
- Durchmesser $\delta(G)$:= maximale Distanz zwischen zwei beliebigen Knoten
  (manchmal auch durchschnittliche Weglänge)
- Konnektivität $ec(G)$:= minimale Anzahl Kanten (oder Knoten), die entfernt werden muss, damit der Graph in zwei unzusammenhängende Teile zerfällt
- Bisektionsbreite := $B(G)$ minimale Anzahl Kanten, die entfernt werden müssen, damit Graph in zwei gleichgroße Teile zerfällt

- Skalierbarkeit := Möglichkeit Topologie mit unterschiedlichen Prozessorzahle zu realisieren bzw. die Prozessorzahl zu verändern
- Flexibilität := Möglichkeit andere Topologien einzubetten oder mit geringem Aufwand zu simulieren

Anforderungen an ein ideales Verbindungsnetz:

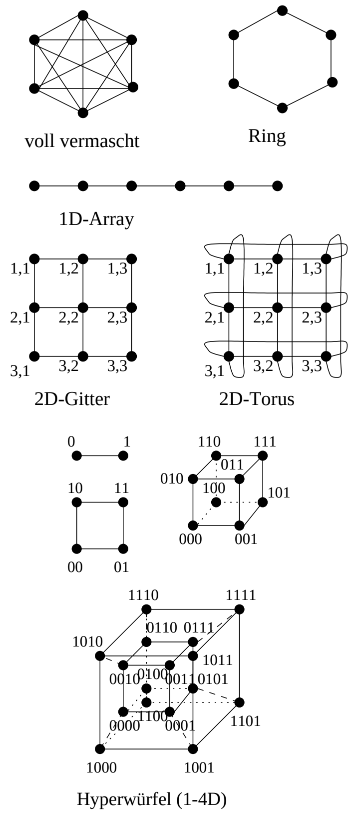
- kleine Kantenzahl
- symmetrisch
- kleiner Durchmesser
- kleiner Grad
- hohe Bisektionsbreite
- hohe Konnektivität
- gute Skalierbarkeit
- hohe Flexibilität

$\Rightarrow$ Anforderungen widersprechen sich teilweise

Gesucht ist Kompromiss zwischen

- Komplexität des Verbindungsnetzes
- Effizienz der darauf realisierbaren Algorithmen

Typische Topologien



Eigenschaften der Topologien (mit $n = 2^k = d^l$ Knoten)

Topologie	$k(G)$	$g(G)$	$\delta(G)$	$ec(G)$	$B(G)$
1-D Array	$n-1$	2	$n-1$	1	1
Ring	n	2	$\lfloor \frac{n}{2} \rfloor$	2	2
dD -Array	$n \cdot d - n^{1/d} \cdot d$	$2 \cdot d$	$d \cdot (n^{1/d} - 1)$	d	$n^{\frac{d-1}{d}}$
dD-Torus	$n \cdot d$	$2 \cdot d$	$d \cdot \lfloor \frac{n^{1/d}}{2} \rfloor$	$2 \cdot d$	$2 \cdot n^{\frac{d-1}{d}}$
Hyperwürfel	$\frac{n \cdot \log_2 n}{2}$	$\log_2 n$	$\log_2 n$	$\log_2 n$	$\frac{n}{2}$
voll. Graph	$\frac{n \cdot (n-1)}{2}$	$n-1$	1	$n-1$	$\frac{n^2}{4}$

Dynamische Verbindungsnetze

- keine direkte Punkt zu Punkt Verbindung, sondern dynamische Schaltung von Verbindungen
- aus Sicht eines Knotens (Prozessors) ist ein dynamisches Verbindungsnetz eine Einheit, welche Verbindung auf Anforderung herstellt

Busnetzwerke

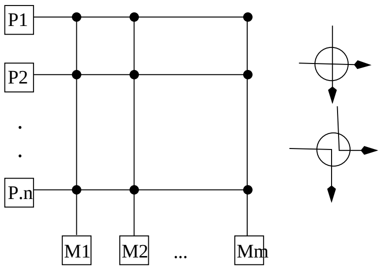
Busse mit großer Bandbreite aber keine simultanen Übertragungen möglich ⇒ nur für kleine Strukturen geeignet

Auswahl des Senders durch Busarbiter

⇒ Anwendung für Systeme bis ca. 64 Knoten

Crossbar-Netze

insbesondere zur Verbindung von Prozessoren mit Speichermodulen



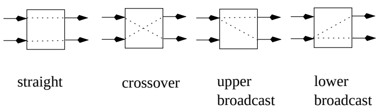
hohe maximale Bandbreite aber aufwendige und teure Realisierung

Mehrstufige Schaltnetze

Schaltung in mehreren Stufen

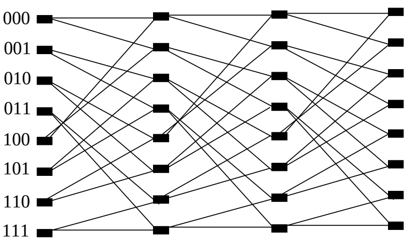
⇒ weniger Schaltelemente aber auch längere Übertragungszeiten

Basiselemente 2×2 Schalter

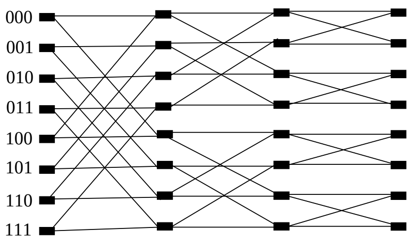


Zusammenfassen von Basiselementen zu Schaltenetzen

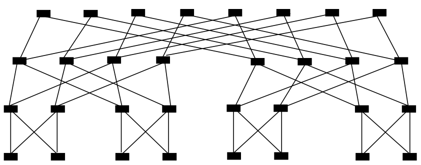
Typische Netzstrukturen



Omega Netz



Butterfly-Netz



Fat Tree

Routingverfahren

Soll eine Nachricht zwischen zwei Knoten ohne direkte Verbindung verschickt werden, so muss ein indirekter Weg gewählt werden

⇒ Routingalgorithmus

Routingalgorithmen in

- Punkt zu Punkt Verbindungsnetzen durch Auswahl der benutzten Leitungen
- geschalteten Netzen durch Auswahl der Schalterstellung

Ziele des Routingalgorithmus

- Kostenminimierung (Minimierung Übertragungzeit, Maximierung Bandbreite)
- gleichmäßige Auslastung der Leitungen
- Deadlockfreiheit

Einflussfaktoren

- Topologie des Verbindungsnetzes
- Netzwerk-Contention (konkurrierende Anforderungen bei Sendung über eine Verbindung)
- Congestion (Verlust von Nachrichten durch Pufferüberläufe)

Klassifizierungskriterien für Routingalgorithmen

- Pfadlänge
  - minimale Pfadlänge
  - nichtminimale Pfadlänge
- Pfadauswahl
  - adaptiv
  - deterministisch

Beispiele für Routingalgorithmen

Dimensionsgeordnetes Routing

Anwendung in Gittern, Torus-Netzen und Hyperwürfeln

XY-Routing im 2D-Gitter:

Jeder Knoten hat 2-dimensionale Adresse (X, Y)

Übertragung von Knoten A mit Adresse (X_A, Y_A)
nach B mit Adresse (X_B, Y_B)

Routingalgorithmus (deterministisch, minimal)

- schicke Paket in X-Richtung bis X_B erreicht
- schicke Paket in Y-Richtung bis Y_B erreicht

$\Rightarrow$ Pfadlänge $|X_A - X_B| + |Y_A - Y_B|$

Routing in Hyperwürfeln:

Jeder Knoten ist mit k Nachbarn verbunden

Knotenzahl $n = 2^k$

Adresse jedes Prozessors ist k-dimensionale Bitfolge

- Adresse Sender A: $\underline{a} = (a_0, \ldots, a_{k-1})$
- Adresse Empfänger B: $\underline{b} = (b_0, \ldots, b_{k-1})$

Übertragung über Knoten
$A = A_0 \to A_1 \to \ldots \to A_l = B$

sei $\underline{a}^i = (a_0^i, \ldots, a_{k-1}^i)$ Adresse von Knoten A_i

Operation $\bigoplus$ (XOR) für Bitvektoren $\underline{a}$ und $\underline{b}$ liefert

- eine 1, wenn $a_j \neq b_j$
- eine 0, falls $a_j = b_j$

Routingalgorithmus:

Nachfolger von Knoten A_i bei Sendung zu B

- bilde $\underline{c} = \underline{a}^i \bigoplus \underline{b}$
- erzeuge $\underline{a}^{i+1}$ aus $\underline{a}^i$ indem das am weitesten rechts stehen Bit, welches in c gleich 1 ist, in $\underline{a}^i$ negiert wird

$\Rightarrow$ Anzahl Schritte entspricht der Anzahl 1 in $\underline{a} \bigoplus \underline{b}$

Deadlockgefahr:

Falls mehrere Nachrichten im Netze sind und Ressourcen nur jeweils von einer Nachricht genutzt werden können

$\Rightarrow$ Gefahr von Deadlocks

Beispiel:

Verbindungskanäle können nur von einer Nachricht genutzt werden und werden erst freigegeben, wenn nächster Kanal frei

$\Rightarrow$ Verhinderung durch Routingalgorithmus

(beiden vorgestellten Algorithmen sind deadlockfrei)

Adaptives Routing

Anpassung der Wahl des Pfades an die aktuelle Lastsituation

$\Rightarrow$ Aufbau virtueller Kanäle

Auswahl eines Kanals auf Basis der beobachteten Last

Beispiel (adaptives Routing in Hyperwürfeln)

Routing von A mit Adresse $(a_0, \ldots, a_{k-1})$
nach B mit Adresse $(b_0, \ldots, b_{k-1})$

$E_1 = \{i | a_i = 1 \wedge b_i = 0\}$ und $E_2 = \{i | a_i = 0 \wedge b_i = 1\}$

- erste Phase: Auswahl eines Nachbarknotens durch Negation eines Adressbits aus E_1 ($\Rightarrow$Dim. E_1 nimmt ab)
- zweite Phase: Auswahl eines Nachbarknotens durch Negation eines Adressbits aus E_2 ($\Rightarrow$Dim. E_2 nimmt ab)

Switchingstrategien

Festlegung, wie eine Nachricht den vom Routingalgorithmus gewählten Pfad durchläuft

- Zerlegung in Pakete oder als vollständige Nachricht
- Allokation der Ressourcen auf dem Pfad,
  vollständig oder teilweise
- Übergabe vom Eingang eines Schalters/Routers
  zum Ausgang

$\Rightarrow$ Switchingstrategie bestimmt
Zeitaufwand der Übertragung

Übertragungszeit zwischen benachbarten Prozessoren

Sender der Nachricht

- Nachricht in Sendepuffer kopieren
- BS berechnet Prüfsumme und fügt Header hinzu und startet Timer
- BS veranlasst Übertragung (per Hardware)

Empfänger der Nachricht

- BS kopiert empfangene Nachricht in Systempuffer
- BS berechnet Prüfsumme und sendet Empfangsbestätigung (bei korrekter Übertragung) oder verwirft Nachricht (bei fehlerhafter Übertragung)

- korrekt empfangene Nachricht wird in den Puffer des Anwendungsprogramms kopiert

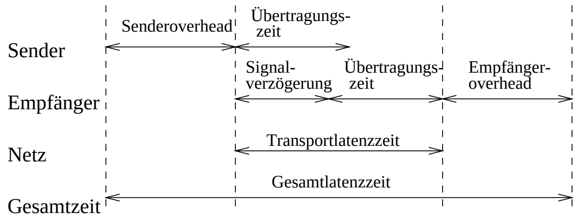
Weitere Schritte beim Sender

- bei Empfang der Empfangsbestätigung Freigabe des Systempuffers
- bei Ablauf des Timers erneute Sendung der Nachricht

Latenzzeit $T(m)$ zur Sendung einer Nachricht der Länge m in Byte

$$T(m) = t_S + t_B \cdot m$$

mit t_S Startzeit (unabhängig von der Nachrichtenlänge)
t_B Übertragungszeit für ein Byte



Übertragung zwischen nicht benachbarten Knoten durch Realisierung von Switchingstrategien

Beispiele für Switchingstrategien

Circuit-Switching:

- gesamter Pfad wird allokiert
- Nachricht wird in Teile (phits) zerteilt und übertragen
- Übertragungspfad wird durch Sendung einer kurzen Kontrollnachricht aufgebaut

Übertragungszeit einer Nachricht der Länge m auf einem Pfad der Länge l

$$T_{cs}(m) = t_S + t_C \cdot l + t_B \cdot m$$

mit t_S Startzeit (unabhängig von der Nachrichtenlänge)
t_B Übertragungszeit für ein Byte
t_C Übertragungszeit Kontrollnachricht

Paket-Switching:

- Nachricht wird in Pakete unterteilt
- Pakete werden unabhängig voneinander übertragen
  jedes Paket erhält
  - Header mit Adressinformation
  - Trailer zur Fehlererkennung
- verschiedene Varianten existieren

Store-and-Forward Routing:

- Pakete werden über eine Verbindung übertragen
- jeder Zwischenknoten speichert Paket und leitet es weiter
- Verbindung zwischen Knoten wird nach Paketübertragung freigegeben

Übliche Strategie für WANs und frühe Parallelrechner
- schnelle Ressourcenfreigabe
- aber hoher Speicherbedarf und Kommunikationszeit

Übertragungszeit einer Nachricht der Länge m auf einem Pfad der Länge l

$$T_{sf}(m,l) = t_S + l \cdot (t_H + t_B \cdot m)$$

mit t_H Übertragungszeit für Header

Cut-Through Routing:

- pipeline-artige Verschachtelung der Paketübertragung
- erste Bits eines Pakets enthalten Zieladresse
- auf Basis der Zieladresse Weiterleitung des Pakets zu einem Nachbarknoten
  - bei freier Verbindung direkte Weiterleitung von Paketen (ohne Zwischenspeicherung)
  - falls Verbindungen belegt sind, so müssen Pakete zwischengespeichert werden (bei virtuellen Verbindungen)

Cut-through Routing heute übliches Verfahren für Parallelrechner

Übertragungszeit einer Nachricht der Länge m auf einem Pfad der Länge l (ohne Zwischenspeicherung)

$$T_{cs}(m,l) = t_S + l \cdot t_H + t_B \cdot (m - m_H)$$

mit m_H Größe des Headers

Variante des cut-through Routings ist Wormhole-Routing

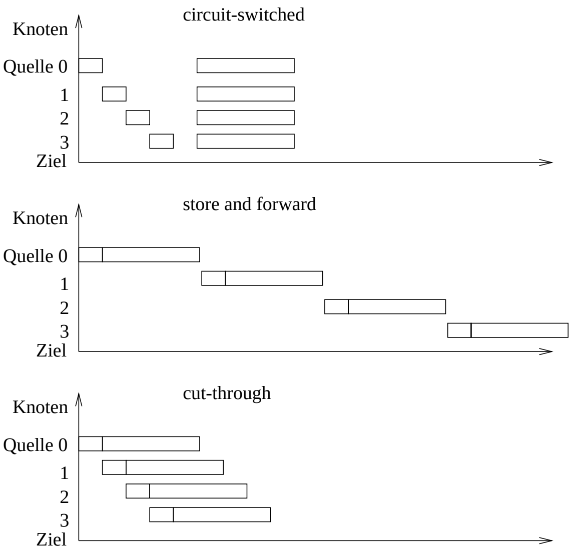
- Zerlegung der Nachricht in kleine Einheiten (flits (*flow control units*), 1-8 Bytes)
- Header-flits werden durch das Netz geleitet (mittels Routingalgorithmus)
- weitere flits folgen dem Header
- falls eine Verbindung belegt blockiert Übertragung (d.h. flits bleiben in ihren Knoten)
- Verfahren durch Hardware unterstützt

Vorteil gegenüber cut-through Routing

- weniger Speicherbedarf

aber Deadlockgefahr

Darstellung Latenzzeit (ohne Contention im Netz)



Verhinderung von Congestion durch Flusskontrolle

- *request-acknowledgement handshake* zwischen Sender und Empfänger

2.4 Beispiele realer Systeme_

Wir betrachten MIMD Systeme
typische Beispiele für unterschiedliche Architekturvarianten
Übersicht über realisierte Varianten keine aktuelle Übersicht!!

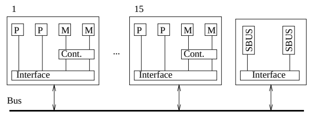
Busbasierte Systeme

auch als SMP-Systeme bezeichnet

- UMA Speicherzugriff
- wenige Prozessoren
- Einsatz im Serverbereich

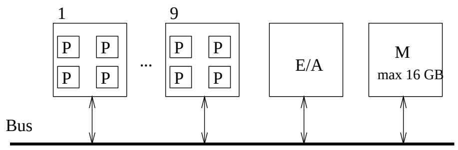
Beispiel Sun Enterprise 6000

- Parallelrechner mit gemeinsamem Speicher
- bis zu 30 UltraSPARC Prozessoren
- je zwei Prozessoren und zwei Speichemodule (max. 1 GB pro Module) pro Prozessorboard
- Busnetzwerk mit 2.67 GB/s Bandbreite
- pro Prozessor ein L2-Cache (Zugriffszeit 40ns)
- Speicherzugriffszeit 300ns (alle Zugriffe, auch lokale, laufen über den Bus ⇒UMA)



Beispiel SGI Power Challenge

- 36 MIPS R4400 (4 pro Board) oder 18 MIPS R8000 (2 pro Board)
- max. 16 GB Hauptspeicher auf eigenem Board
- Busnetzwerk mit 1.2 GB/s Bandbreite
- pro Prozessor ein L2 Cache (Zugriffszeit 75ns)
- Speicherzugriffszeit 1150ns (UMA)

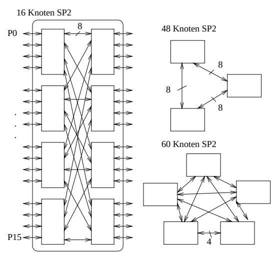


Rechner mit verteiltem Adressraum

- einfache Realisierung "großer" Systeme
- sehr gutes Preis-/Leistungsverhältnis
- aber schwierige Programmierung durch verteilten Adressraum
- primäres Einsatzgebiet numerische Simulation

Beispiel IBM SP2

- System aus Power2 Prozessoren (4-128, in Ausnahmefällen 512 Knoten)
- Verbindung der Prozessoren über mehrstufige geschaltete Verbindungsnetze (8×8 Crossbar Switches)
- jeweils 16 Knoten werden zu einem Rahmen zusammengefasst
- bis zu 80 Knoten durch Verbindung von 5 Rahmen
- größere Prozessorzahlen durch zusätzliches geschaltetes Verbindungsnetz zwischen den Rahmen



Erweiterungen der SP2:

ASCI Blue Pacific

- 512 Prozessorknoten
- jeder Prozessorknoten ist ein SMP mit 4/8 Prozessoren

Rechner mit gemeinsamen Adressraum

Beispiel Cray T3E

- DEC Alpha Prozessoren (bis zu 2048) mit 300-600 MHz
- Knoten mit Prozessor, lokalem Speicher, 8 KB L1 und 96 KB L2 Caches
- 3D-Torus Verbinungsnetz
  (Bandbreite 600 MB/s bidirektional)
- NUMA Zugriff auf Daten in externen Speichern
- Speicherzugriffszeiten (300 MHz Prozessoren):
  L1 Cache 6.6ns, L2 Cache 26.6 ns, lokaler Speicher 283 ns, externer Speicher ca. 1500 ns

Beispiel SGI Origin 2000

- MIPS R10000 Prozessoren (bis zu 1024)
- Knoten mit 2 Prozessoren, bis zu 4GB Hauptspeicher, 4MB L2 Cache, Directory Controller (zur Verwaltung der Cache-Kohärenz), Schnittstellen E/A und Verbindungsnetz
- Verbindungsnetz
  - Hyperwürfel (bis zu 4D) bis zu 32 Knoten (64 Prozessoren) mit 1.56 GB/s Bandbreite bidirektional, je zwei Berechnungsknoten und ein Router pro Knoten im Hyperwürfel

  - bei mehr als 64 Prozessoren Metarouter zur Verbindung der Hyperwürfel
- Speicherzugriffszeiten:
  L1 Cache 5.1ns, L2 Cache 56.4 ns, lokaler Speicher 310 ns, externer Speicher (4 Prozessoren) ca. 540 ns, externer Speicher (128 Prozessoren) ca. 945 ns

Beispiel KSR1

- bis zu 1088 superskalare Prozessoren (Eigenentwicklung)
- Knoten mit 1 Prozessor, 32 MB als Cache organisierter Hauptspeicher, 512 KB Subcache
  (⇒ COMA Architektur)
- Verbindungsnetz
  - Ringverbindung mit 1 GB/s Bandbreite
  - Ring der Stufe 0 mit 32 Prozessoren
  - Verbindung von bis zu 34 Ringen über Ring der Stufe 1
- Speicherverwaltung
  - Sicherung sequentieller Konsistenz
  - Seitengröße 16 KB
  - logische Anordnung des Caches als zweidimensionales Feld

2.5 Workstationnetze und -cluster

Ursprünglicher Aufbau von Parallelrechnern

- spezielle Hardware (Prozessoren, Verbindungsnetzwerke)
- spezielle Software (Betriebssystem, Compiler)

Dies führt zu Parallelrechnern, die

- leistungsfähig und
- für parallele Berechnungen optimiert sind

aber auch

- teuer,
- nur beschränkt zugreifbar und
- aufwändig zu programmieren sind

Billige und verbreitete Alternative Netz von Workstations (NOWs)

- eigenständige Maschinen über ein Netzwerk verbunden
- vollständige Kopie des BS auf jedem Rechner
- verteilte/parallele Programmerstellung durch Basisbibliotheken PVM/MPI

Falls Maschinen eines Netzes zur gemeinsamen Lösung von Problemen genutzt werden, spricht man von einem Cluster

Größte Vorteile von NOWs:

- weite Verbreitung
- Basis der Kommunikation TCP/IP
- Nutzung von Standardkomponenten
- extrem hohe Peak Leistung
- vergleichsweise preiswert
- universell verwendbar
- einfach (und fast beliebig) erweiterbar
- teilweise Verwendung von Standardsoftware

Größte Nachteil von NOWs gegenüber Parallelrechner

- langsame Kommunikation mit geringer Bandbreite und hoher Latenzzeit
- i.d.R. NUMA Architekturen
- in NOWs nur geringe Unterstützung zur Parallelisierung
- durch Mehrbenutzerbetrieb Leistungsverluste und Schwankungen in Ressourcenverfügbarkeit
- u.U. Notwendigkeit der Datenkonvertierung

Verbesserung durch geschaltete Verbindungsnetze in NOWs und explizite Clusterbildung

Ziel: Aufbau eines "Parallelrechners" aus homogenen PCs oder Workstations

Vorteile:

- homogene Leistung der Prozessoren
- homogene Datenformate
- angepasstes Kommunikationsnetz
- Softwareunterstüzung paralleler Anwendungen ist möglich

Erste Realisierungen Beowulf Cluster 1994

Oft Vernetzung von PCs unter Linux

Cluster umspannen einen weiten Bereich

- von universell genutzten verteilten Rechnern
- zu dedizierten Parallelrechnern

Wesentliche Unterschiede

- Einbenutzer- - Mehrbenutzerbetrieb
- Verfügbarkeit von Werkzeugen zur Verwaltung, Steuerung und zum Management großer paralleler Anwendungen

Kritische Komponente: Kommunikation zwischen Prozessen

Verfügbare Realisierungen

- Ethernet (Bandbreite: 100Mb/sec oder 1Gb/sec, Latenzzeit $100\mu s$)
- Myrinet (Bandbreite 100Mb-2Gb/sec, Latenzzeit $7-9\mu s$)
- Infiniband (Bandbreite 2.5Gb/sec, Latenzzeit $7-9\mu s$)
- SCI (Bandbreite 320Mb/sec, Latenzzeit $2\mu s$)

$\Rightarrow$ Latenzzeit von Ethernet zu groß für viele parallelen Anwendungen

Andere Ansätze operieren aus dem Benutzerprozess heraus (spezielle Implementierung von MPI/PVM sind notwendig und verfügbar) und erreichen Leistung von Verbindungsnetzen in Parallelerechnern

Preis der Netze

- deutlich unterhalb von properitären Netzen, die in Parallelrechnern verwendet werden
- deutlich oberhalb von Standard-Ethernet

$\Rightarrow$ viele heutige Parallelrechner basieren auf der Clustertechnik (siehe auch TOP 500 Liste)

Beispiel Myrinet

- Erste Realisierungen 1994
- geschaltete Verbindungen (verfügbar Switches mit 8-256 Anschlüssen, bis 32 Anschlüsse als Crossbar realisiert)
- Duplexverbindungen mit 2+2Gb/sec Bandbreite für aktuelle Karten
- Aufbau von Clustern durch Verbindung mehrerer Switches (Realisierungen mit mehreren Tausend Rechnern existieren)
- Verbindung durch Lichtwellenleiter
- cut-through Routing
- Kommunikationsschnittstellen (GM properitär für Myrinet, MPI, Sockets, TCP/IP)
  für TCP/IP Latenzzeit größer und Bandbreite geringer, da BS involviert
- Open Source Software verfügbar
- Kommunikation direkt aus dem Adressraum des Benutzerprozesses
- Hohe Zuverlässigkeit (MTBF 5 Mio. Std.)
- Weite Verbreitung
  (38% der Rechner in der TOP 500 Liste benutzen Myrinet)

Computer-Grids

Beobachtung:

Durch die heutige Vernetzung steht eine immense Menge teilweise ungenutzter Rechnen- und Speicherkapazität zur Verfügung, die prinzipiell für die Lösung großer Probleme nutzbar ist

Durch den temporären Aufbau virtueller Parallelrechner lassen sich diese Ressourcen nutzen ($\Rightarrow$ Computer-Grids)

Probleme und Herausforderungen dabei

- heterogene Hardware
- hohe Latenzzeit und geringe Bandbreite der Kommunikationsverbindungen
- wechselnde Verfügbarkeit
- Zugriffsrechte und Sicherheitsaspekte

Aber

- enormes Potenzial
- sehr kostengünstig
- für lose gekoppelte Problem gibt es schon erfolgreiche Realisierungen

$\Rightarrow$ aktuelles Forschungs- und/oder Anwendungsgebiet

2.7 Die Top 500 Liste

Leistungsmaße für Parallelrechner

- Angabe von Taktfrequenz Prozessor
- MIPS (millions of instructions per second)
- MFLOPS
  (millions of floating point operations per second)

⇒ Leistungsmaße zum Vergleich von Systemen

Sind Systeme auf Basis der Maße vergleichbar???

Sicherlich nicht allgemein! Auftretende Probleme:

- Angabe von Taktfrequenz sagt wenig darüber aus, welche Instruktionen in einem Takt ausführbar sind ⇒ Vergleich nur innerhalb einer Prozessorfamilie
- Wie sollen MIPS/MFLOPS bestimmt werden?
  - Einfachste Form: Messung pro Prozessor ⇒MIPS/MFLOPS pro Prozessor
  - Peak Rate: Anzahl Prozessoren * MIPS/MFLOPS pro Prozessor
    Aussagekraft der Peak Rate???
    Offensichtlich kann die Peak Rate bei Parallelrechnern nicht erreicht werden
    (Speicherzugriffszeiten, Kommunikationszeiten!)

Gibt es objektivere Vergleichsmaße?

Naheliegende Idee:

Vergleich der Laufzeit einer typischen Anwendung!

Was ist eine typische Anwendung?

Dies ist sicherlich nicht unabhängig vom Rechner definierbar!

Für Parallelrechner/Supercomputer gilt

- Anwendung muss "groß" sein
- Anwendungen oft aus dem numerischen Bereich

⇒ LINPACK Programm wird als Benchmark eingesetzt

LINPACK ist

- eine Sammlung von Routinen der linearen Algebra
- Teil der Netlib
- in FORTRAN geschrieben

Zum fairen Vergleich von Supercomputern wurde folgendes Vergleichsverfahren festgelegt:

- Lösung eines dichtbesetzten linearen Gleichungssystems mit n Variablen
- Lösung mit Hilfe der LU-Dekomposition mit Aufwand $2/3 \cdot n^3 + O(n^2)$ Fließkommaoperationen

Ermittelte Resultate

- Rmax LINPACK Leistung (in MFLOPS)
- Rpeak theoretische Maximalleistung (in MFLOPS)
- Nmax Problemgröße, bei der Rmax erreicht wurde
- N1/2 Problemgröße bei der Rmax/2 erreicht wurde

Daten können von Herstellern oder Anwendern übermittelt werden und werden überprüft

Seit 1993 wird zweimal im Jahr eine Liste der
500 "schnellsten" Rechner (auf Basis von Rmax) publiziert

⇒ Liste ermöglicht einen Überblick über Entwicklungen
im Umfeld der Supercomputer

Auch dieses Bewertungsmethode ist nicht unumstritten, da

- ein sehr reguläres Problem verwendet wird
  (reale Probleme sind oft irregulär)
- Rechner, Compiler, Laufzeitsystem bzgl. des Benchmarks optimiert werden (dürfen)
- Rangfolge auf Basis von Rmax ohne Berücksichtigung von Nmax problematisch ist

Aber

Vergleichswerte sind deutlich aussagekräftiger als Vergleich der Peak Rate!

Einige Ergebnisse aus der aktuellen Liste

Rang	Hersteller-Typ	Inst.-Ort	Land	Jahr	#Proz.	Rmax	Rpeak	N
1	IBM BlueGene	IBM/DOE	USA	2004	32768	70720	91750	933887
2	SGI Altix	NASA/Ames	USA	2004	10160	51870	60960	1.29&+6
3	NEC Vector	Earth-Sim.-Center	Japan	2002	5120	35860	40960	1.08&+6 266240
5	Intel NOW	Law.-Liv.-Lab.	USA	2004	4096	19940	22938	975000 110000
30	IBM SP	Jülich	Germ.	2004	1312	5568	8921	660000
50	IBM Cluster	DBank Frankf.	Germ.	2004	1064	3755	6511	
85	IBM SP	MPI München	Germ.	2003	822	2198	4274	
94	AMD NOW	Uni Wuppertal	Germ.	2004	1024	2083	3686	
	Pentium 3GHz				1	3.2	6	1000