Vorlesung
Rechnerarchitektur II
Parallelrechner

Sommersemester 1994

Dieses Skript enstand mit Hilfe von Studenten, die als Hörer an der Vorlesung teilnahmen. Die Namen sind im Kopf der jeweiligen Vorlesungsstunde vermerkt. An dieser Stelle möchte ich ihnen danken.

Das Skript kann und will die Vorlesung nicht ersetzen, es ist als Hilfe und Ergänzung gedacht. Trotz Korrekturlesens kann nicht ausgeschlossen werden, daß Fehler im Skript verblieben sind.

Saarbrücken, im Juli 1994

Jörg Keller

Inhaltsverzeichnis

Grundlagen 1

Einbettung in Würfel 7

Netzwerke und Routing 10

Shared Memory Systeme 16

Fluent Machine Routing 22

Fluent Machine Routing II 27

Distributed Shared Memory Systeme 31

Message Passing Systeme 34

LogP Modell 38

LogP Modell II, SIMD Architekturen I 41

SIMD Architekturen II 46

Datenfluß– und VLIW Maschinen 52

1. Übungsblatt 56

2. Übungsblatt 57

3. Übungsblatt 58

4. Übungsblatt 59

5. Übungsblatt 60

6. Übungsblatt 61

7. Übungsblatt 62

8. Übungsblatt 63

9. Übungsblatt **64**

10. Übungsblatt **65**

11. Übungsblatt **66**

Vorlesung Rechnerarchitektur II (Parallelrechner)
Dr. Jörg Keller

SS 94, Termin Nr. 1: 14.04.94
Skript: Andreas Crauser

Grundlagen

1.1 Einleitung

Was ist ein Parallelrechner ?

Versuch einer Definition:
Ein Parallelrechner ist ein Computer mit mehreren Prozessoren, die irgendwie zusammenarbeiten.

Man unterscheidet folgende Parallelrechner:

- homogene : alle Prozessoren sind hardwaremäßig gleich
- heterogene : die Prozessoren dürfen sich hardwaremäßig unterscheiden
- symmetrisch: die Prozessoren sind bzgl. ihrer Rolle im System austauschbar (symmetrisch $\Rightarrow$ homogen)
- lose gekoppelt:Netzwerk von eigenständigen Systemen, z.B. Netzwerk von Workstations, die gemeinsam an einem Problem arbeiten
  Vorteil : System als Einzelcomputer laufähig, geographisch verteilbar
  Nachteil : Kommunikation verglichen mit Rechengeschwindigkeit ist langsam
- eng gekoppelt : physikalisch 1 Rechner, Prozessoren sind alleine nicht lauffähig

Wie mißt man den Vorteil eines Parallelrechners ?

$$\text{speedup} = \frac{\text{Laufzeit des sequentiellen Rechners}}{\text{Laufzeit des parallelen Rechners}}$$

Bei einem Rechner mit n Prozessoren wäre ein Speedup von n erwünscht. Dies wird jedoch wegen Overhead meist nicht erreicht.

Effektivität $= \frac{speedup}{n} \epsilon [0,1]$ (normalerweise)

Theoretisch gibt es superlinearen Speedup (speedup >n). Dieser Effekt läßt sich wie folgt begründen:
Simuliert man s Schritte des parallelen Programms auf einen sequentiellen Rechner, so ergibt sich als Speedup $n * s/s = n$. Ist jedoch der Simulationsoverhead amortisiert nicht null, sondern c dann folgt:
$c + N * s/s = n + c/s > n$ (superlinearer speedup)

Bemerkung: Der Begriff des Speedups ist mit Vorsicht zu genießen, da der sequentielle Algorithmus optimal sein muß.

1.2 Klassifizierung von Parallelrechnern (nach Flynn, 1966)

Man unterscheidet 4 Klassen:

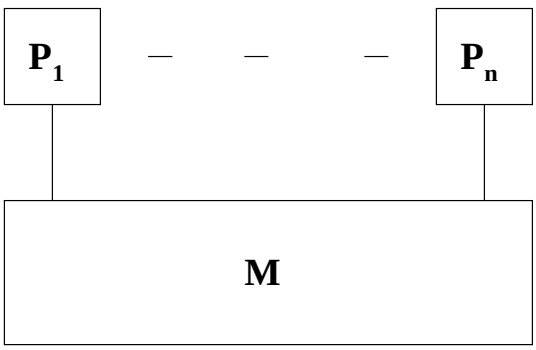
- SISD (single instruction multiple data):
  entspricht den heutigen von Neumann Rechnern
- MISD (multiple instruction single data):
  Die Klasse wird als leer betrachtet. Manche Informatiker ordnen gepipelinete Vektorrechner dieser Klasse zu.
- SIMD/MIMD (single/multiple instruction multiple data):
  Diese beiden Klassen beinhalten die heute bekannten Parallelrechner

Auf einem **SIMD**-Rechner arbeiten alle Prozessoren ein Programm auf verschiedenen Daten ab (insbesondere sind sie stets bei der gleichen Instruktion). Es ist möglich, Prozessoren gezielt auszublenden. Anwendungsbeispiele für solche Rechner sind Matrixprobleme.
Im **MIMD**-Modell kann jeder Prozessor machen, was er will. Insbesondere können sogar mehrere Programme gleichzeitig abgearbeitet werden. Unter MIMD versteht man heute meist **SPMD** (Single Program Multiple Data), d.h. es wird ein Programm auf verschiedenen Daten ausgeführt, die Prozessoren müssen jedoch nicht alle die gleiche Instruktion abarbeiten.

Bemerkung: SIMD ist auf MIMD simulierbar, MIMD ist allgemein flexibler

Eine weitere Klassifizierung ist nun durch unterschiedliche **Speicheranbindung** möglich:

1. gemeinsamer Speicher (shared memory)



   Die Kommunikation der Prozessoren erfolgt über den Speicher, alle Prozessoren können alle Daten sehen.

2. privater Speicher (distributed memory)

   Jeder Prozessor hat eigenen Speicher

Der Austausch von Daten verschiedener Prozessoren erfolgt über message-passing
Programmierproblem: Welcher Prozessor erhält welche Daten ? Das Netwerk bei privatem Speicher läßt eine weitere Unterteilung zu:

(a) switch-network: Jeder Prozessor kann über das Netzwerk direkt mit jedem kommunizieren

(b) network architecture: Eine Kommunikation ist nur mit Nachbarn möglich

Es ist zu beachten, daß diese Klassifizierung sowohl auf die Hardware als auch auf die Software anwendbar ist. Dies ist jedoch nicht das gleiche, eine Abbildung von Software auf Hardware erfolgt entweder über zusätzliche Hardware oder durch den Compiler.
Man spricht dann von
virtual
distributed shared memory $\simeq$Hardware distributed, Programmiermodell shared
emulated

wobei bei virtual,distributed Datenreplikation erfolgt, bei emulated nicht.

1.3 Netzwerke

kurze Wiederholung der wichtigsten Begriffe der Graphentheorie:
Ein Graph ist ein Tupel G=(V,E) mit V endl. Knotenmenge, E endl. Kantenmenge Ist der Graph gerichtet, so ist E $\subseteq$ V$\times$V. Ist der Graph ungerichtet, so gilt: E$\subseteq$ {{a,b} |a,bϵV}.
Bei Netzwerken entsprechen den Knoten Schaltern, den Kanten Leitungen bzw. Kabel. Im ungerichteten Graphen sind die Kanten (links) bidirektional, im gerichteten Graphen unidirektional.
Desweiteren unterteilt man in Netzwerken die Knotenmenge in Teilmengen:
V_S $\subseteq$V : Sender
V_E $\subseteq$V : Empfänger

1.3.1 Eingenschaften von Graphen

Durchmesser D = $\max_{v,w}$\{kürzester Pfad$v \rightarrow w$\}

Grad g = $\max_v$\{max\{ingrad(v),outgrad(v)\}\}

Sei | V | = N, ein Netzwerk heißt *eindeutig*: $\forall$v,w Pfad v $\rightarrow$ w eindeutig

1.3.2 Kommunikationsmuster

h-Relation: jeder Sender sendet $\leq$ h Nachrichten, Empfänger empfängt $\leq$ h Nachrichten
1-Relation: Permutation

Broadcast: 1 Sender, alle empfangen (entweder n-1 Nachrichten bei n Teilnehmern oder das Netzwerk dupliziert die Nachricht)

1.3.3 Vermittlungsarten

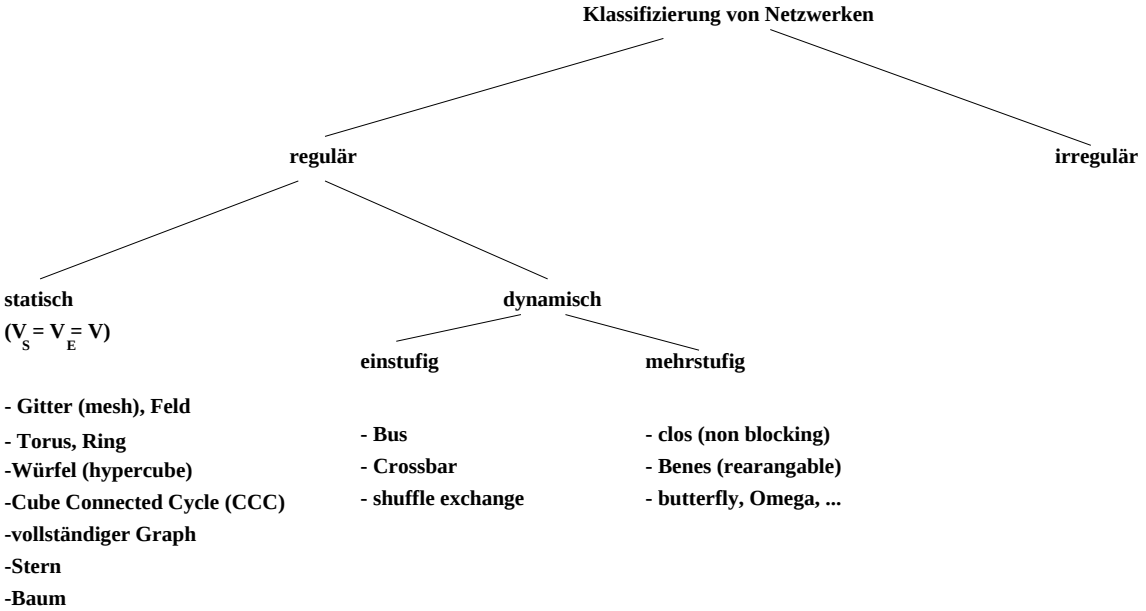
- paketvermittelnd:
  Die Nachricht wird in einem Paket verpackt. Das Paket enthält die Codierung des Empfängers. Es wird kein Pfad vom Sender zum Empfänger aufgebaut, das Paket wird immer zum Nachbarn geschickt, d.h. es springt von Knoten zu Knoten (hops)

- leitungsvermittelnd:
  Es wird im Netzwerk ein Pfad von Sender zum Empfänger geschaltet, über den alle Nachrichten versendet werden.
  Vorteil: schnelle Übertragung großer Datenmengen
  Nachteil: geschalteter Pfad blockiert vielleicht andere
  großer Overhead für viele kleine Datentransfers

Bei Leitungsvermittlung gibt es zwei wichtige Eingenschaften:

- nonblocking:
  Eine Permutation kann im Netzwerk ohne Konflikt eingestellt werden
- rearrangable:
  Mit Umordnen anderer Pfade aus einer Teilpermutation kann jede neue Verbidung bis zur vollständigen Permutation ohne Konflikt eingestellt werden

1.3.4 Klassifizierung

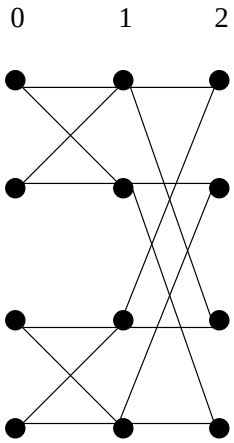


Definitionen:
Ein d-dimensionales Gitter (Torus) mit Seintenlänge n ist ein Graph G=(V,E) mit V = $\{0,\ldots,n-1\}^d$
$\{(a_0,\ldots,a_{d-1}),(b_0,\ldots,b_{d-1})\}\epsilon E \Leftrightarrow \exists$! i mit $\mid a_i - b_i \mid = 1, a_j = b_j, j \neq i$
beim Thorus $\mid a_i - b_i \mid = 1 \bmod n$

Der d-dimensionale Würfel H_d ist ein d-dimensionales Gitter mit Seitenlänge 2.

Das n-dimensionale Butterfly ist ein Graph G=(V,E) mit V = $\{0,\ldots,2^n-1\} \times \{0,\ldots,n\}$
$\{(a_0,a_1),(b_0,b_1)\}\epsilon E \Leftrightarrow b_1 - a_1 = 1 \wedge (a_0 = b_0 \vee b_0 = a_0 \oplus 2^{a_1})$

Beispiel:

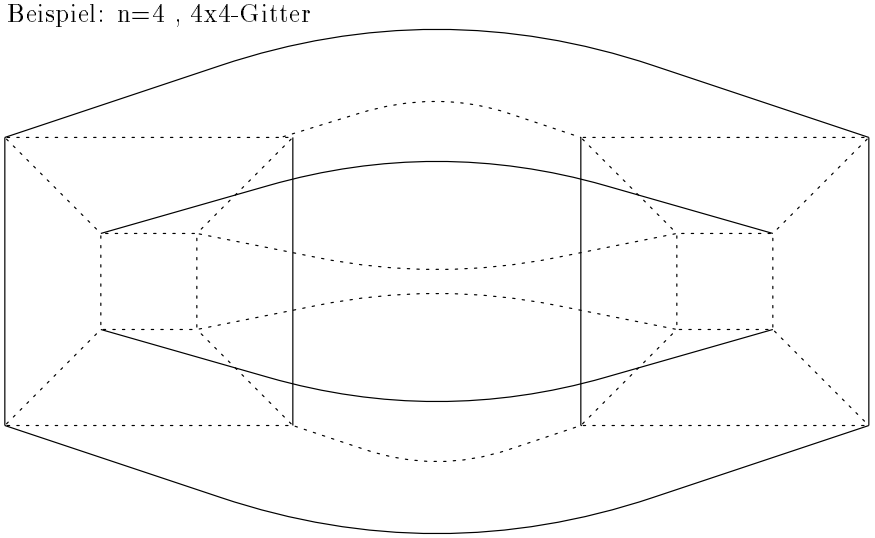


1.3.5 Einbettung auf Gittern

Eines der bekanntesten Netzwerke ist der Hypercube. Er ist vor allem deshalb beliebt, weil man Gitter gut auf ihm einbetten kann. Gitter sind eine oft benutzte Datenstruktur im wissenschaftlichen parallelen Rechnen (Physik, Chemie).

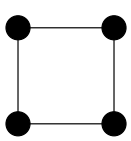
Satz:
Der n-dimensionale Würfel enthält jedes Gitter mit N=2^n Knoten als Subgraph.
(Subgraph G' von G : V'=V E'$\subseteq$ E)

Beispiel: n=4 , 4x4-Gitter



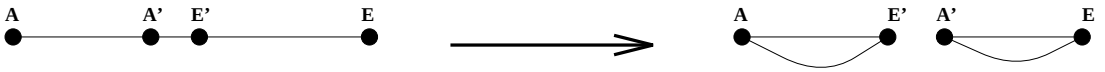
Beweis:
1.) Einbettung von 1-dimensionalen Gittern
2.) Abbildung von mehrdimensionalen auf 1-dimensionale Gitter
Lemma 1:
Der n-dimensionale Würfel enthält ein 1-dimensionales Gitter (Torus) der Seitenlänge N=2^n als Subgraph.

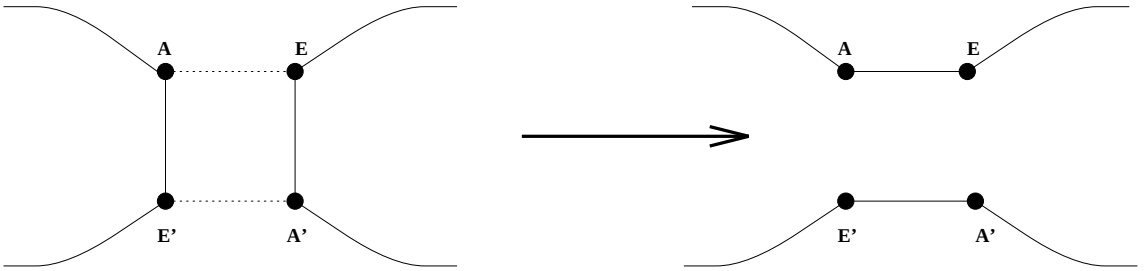
Beweis durch Induktion über n:

n=2: 

n $\rightarrow$ n+1:
Spalte den n+1-dimensionalen Würfel in zwei n-dimensionale Würfel (= entferne die Kanten entlang Dimension

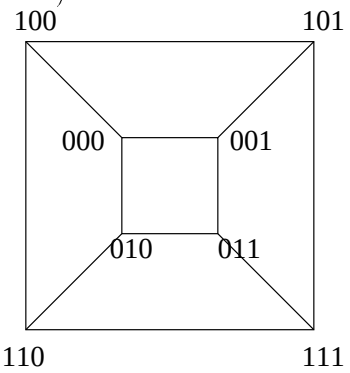
n)
Zerschneide das 1-dimensionale Gitter in der Mitte
Konstruiere identische Einbettungen(geht nach Induktionsvorausssetzung)
Lösche die Kanten AE' und A'E und ersetze sie durch Kanten AE und A'E' entlang Dimension n.





q.e.d. Lemma

Laufe entlang des Gitters(Thorus)
Man trifft alle Knoten des Hyercubes und die sukzessiven Nummern unterscheiden sich nur in einem Bit (Gray code).



Vorlesung Rechnerarchitektur II (Parallelrechner)
Dr. Jörg Keller

SS 94, Termin Nr. 2: 21.04.94
Skript: Cédric Lichtenau

Einbettung in Würfel

Satz: $n-$dim. Würfel enthält jedes Gitter mit 2^n Knoten als Subgraph.

Wir wissen bereits:

1. Lemma 1: $n-$dim Würfel enthält $1-$dim Gitter mit 2^n Knoten als Subgraph.
2. Üb 1.3: $M_1 \times \ldots \times M_k$ Gitter $= M_1$-Gitter $\otimes$ M_k-Gitter.
3. Üb 1.3: $H_r = H_{r_1} \otimes \ldots \otimes H_{r_k}$ mit $r_1 + \ldots + r_k = r$.

Lemma 2: Sei $G = G_1 \otimes \ldots \otimes G_k$, $G' = G'_1 \otimes \ldots G'_k$ und sei G_i Subgraph von G'_i für $1 \leq i \leq k$.
Dann ist G ein Subgraph von G'.

Bew.: (Satz)
$M_i = 2^{r_i}$
$G = M_1 \times \ldots \times M_k$ Gitter $\overset{\text{nach2)}}{\Rightarrow}$ $G_i = M_i$-Gitter.
$G' = H_r \overset{\text{nach3)}}{\Rightarrow} G'_i = H'_{r_i}$
$\overset{\text{nach1)}}{\Rightarrow} G_i$ Subgraph von G'_i
$\overset{\text{nachLemma2)}}{\Rightarrow} M_1 \times \ldots \times M_k$ Gitter ist Subgraph von H_r $\square$

Bew.: (Lemma 2)
Ist G_i Subgraph von G'_i, dann gibt es ein Mapping $\delta_i : V_i \longrightarrow V'_i$ bijektiv,
so daß Kanten von E_i auf Kanten in E'_i abegebildet werden.
Definiere $\delta : V \longrightarrow V'$ $\qquad \delta(v_1 \ldots v_k) = (\delta_1(v_1), \ldots, \delta_k(v_k))$
da δ_i bijektiv $\Rightarrow \delta$ bijektiv
Kanten in E werden auf Kanten in E' abegebildet, wegen Definition der δ_i
$\Rightarrow G$ ist Subgraph von G' $\square$

Def:
- Die **Einbettung** eines Graphen G (Guest) auf einem Graphen H (Host) ist definiert durch (α, ρ)
  mit $\alpha : V_G \longrightarrow V_H$
  $\rho : E_G \longrightarrow \text{Pfade}(H)$
- **Load:** $L = \max_{v \in V_H} |\alpha^{-1}(v)|$
  Max. Anzahl von Knoten in G, die auf einen Knoten in H eingebettet werden.
- **Diltation:** $D = \max_{\{u,v\} \in E_G} \text{Länge}(\rho(\{u, v\}))$
  Längster Pfad von H auf den eine Kante von G eingebettet wird.
- **Congestion:** $C = \max_{e \in E_H} |\{e' \in E_G : e \in \rho(e')\}|$
  Max. Anzahl von Kanten aus G, die auf eine Kante in H eingebettet werden.
- **Expansion:** $E = \frac{|V_H|}{|V_G|}$

Bsp.: Siehe Abbildung 2.1.

Simulation: Arbeite T Schritte eines Programmes für G auf H in Zeit T' ab.

- **Slowdown:** $\frac{T'}{T}$
- **Effizienz:** $\frac{1}{\text{Slowdown} \cdot \text{Expansion}}$

Routing:

- Bestimmung des Pfades von Startknoten zu Zielknoten.
- Scheduling = Reihenfolge von Paketen bei Konflikten.

Pfadbestimmung in Butterfly

- In Stufe i benutze Bit i der Zieladresse.

Bsp.: Siehe Abbildung 2.2.

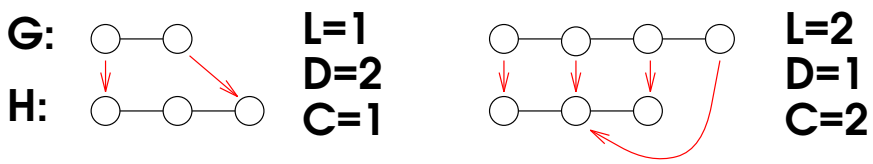


Abbildung 2.1: Einbettung eines Graphen.

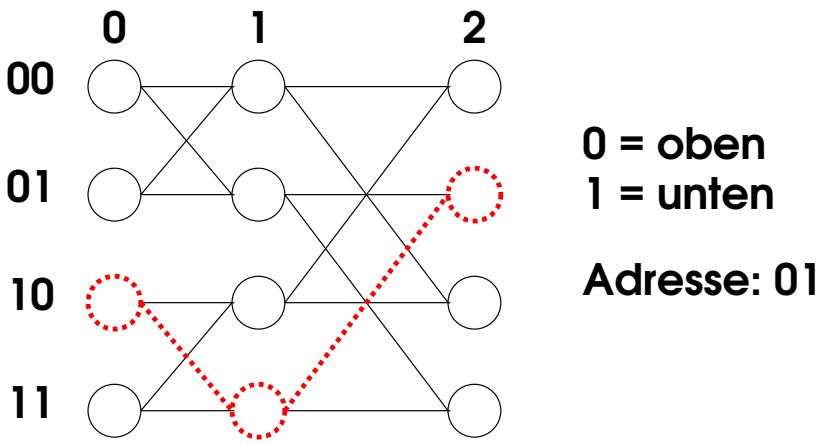


Abbildung 2.2: Butterfly routing.

Vorlesung Rechnerarchitektur II (Parallelrechner)
Dr. Jörg Keller

SS 94, Termin Nr. 3: 28.04.94
Skript: Peter Bach

Netzwerke und Routing

3.1 Graphen

Die besprochenen Graphen sind auch auf dem Sonderblatt vom 28.04.94 zu finden. Es handelt sich um **Cube Connected Cycles (CCC)**, **Shuffle-Exchange**, **Bus**, **Stern**, **Bênes–Netzwerk**, **Crossbar** und **Binärer Baum** (Abbildung 3.1 bis 3.7).

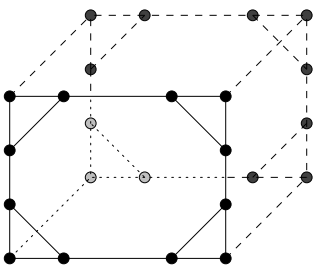


Abbildung 3.1: Cube Connected Cycle, 3 Dimensionen

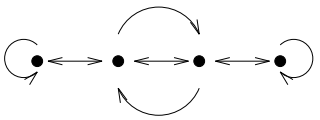


Abbildung 3.2: Shuffle-Exchange, n=4

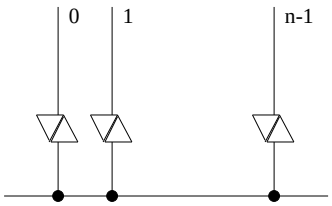


Abbildung 3.3: Bus

Jeder Graph hat Vor– und Nachteile. So besitzen **Stern** und **Baum** einen Flaschenhals. Wenn man davon ausgeht, daß von allen Knoten zufällig verteilt Pakete an zufälliges Ziel geschickt werden sollen, so wird der Knoten an der Wurzel des Baums stark belastet. Dieses Problem führte zum **Fat-Tree**. Die stark belasteten Verbindungen sind hier mehrfach vorhanden.

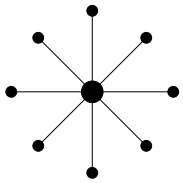


Abbildung 3.4: Stern

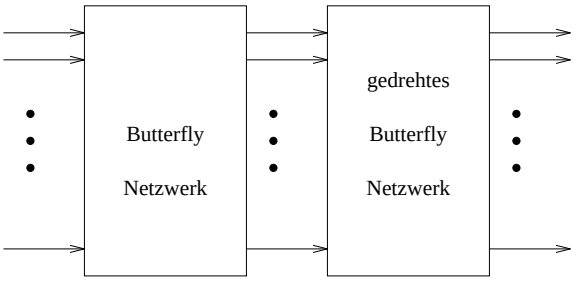


Abbildung 3.5: Bênes–Netzwerk

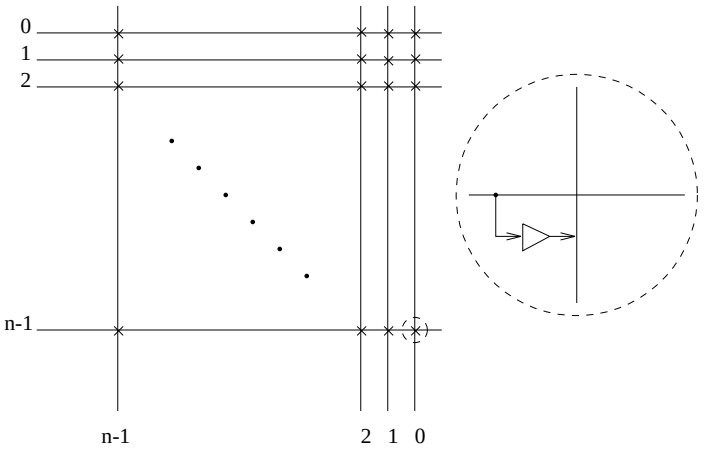


Abbildung 3.6: Crossbar

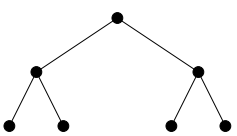


Abbildung 3.7: Binärer Baum, 2 Dimensionen

3.2 Kommunikationsarten

Wie kann man nun auf diesen unterschiedlichen Graphen kommunizieren? Man unterscheidet:

- **Routing**: Das Berechnen des Weges zwischen Quellknoten und Zielknoten und das Versenden der Information selbst.
- **Scheduling**: Das Verhalten beim Kollisionsfall. Zwei oder mehrere Pakete treffen in einem Knoten aufeinander. Hier muß die Entscheidung getroffen werden, ob ein Paket gespeichert oder weggeworfen wird.

Die Art der Kommunikation kann entweder **synchron** oder **asynchron** erfolgen. Man versteht darunter:

- **synchron**: Der Datenaustausch geschieht zu bestimmten Zeiten. Ein besonders synchroner Ablauf ist das **compute-communicate** Modell. Während der **compute**-Zeit rechnen die Knoten lokal, während der **communicate**-Zeit tauschen sie Informationen aus.
- **asynchron**: Zu beliebiger Zeit werden Nachrichten ins Netzwerk eingespeist.

Die Kontrolle über die Kommunikation:

- **global**: Es existiert ein Scheduler, der die Wege der Pakete zentral verwaltet und bestimmt.
- **lokal**: Jeder Knoten kontrolliert den Weg der Pakete, die durch ihn durch geroutet werden.

Während die globale Variante bei **leitungsvermittelnden Netzen** mit großem Datenaufkommen Verwendung findet, wird die lokale Kontrolle oft bei **paketvermittelnden Netzen** eingesetzt. Es müssen viele Routingentscheidungen getroffen werden, was eine zentrale Kontrolle überlasten würde.

Um ein Routing auszurechnen, gibt es verschiedene Möglichkeiten:

- **deterministisch**: Der Weg ist immer reproduzierbar. Ein Spezialfall hiervon ist das **oblivious Routing**. Hier hängt der Weg nur von Start und Ziel ab.
- **randomisiert**: An bestimmten Stellen des Algorithmus werden zufällige Entscheidungen getroffen.

  Beispiel: **Valiants Paradigma** oder **2-Phase-Routing** auf Hypercubes: Viele Permutationen lassen sich gut routen, aber nicht alle.

  1. Route zu zufälliger Zwischenadresse.
  2. Route zum Ziel.

  Der Trick hierbei ist, daß ein deterministischer Schritt in zwei zufällige Teilschritte zerlegt wird. Mit sehr hoher Wahrscheinlichkeit ist die Wahl der Zwischenadresse gut.
- **adaptiv**: Der Algorithmus verhält sich im Normalfall deterministisch, in Ausnahmefällen (Konflikt) randomisiert. Dies ist nur bei **nichteindeutigem** Weg (z.B. Netz) möglich. Unterschieden wird noch zwischen:
  - **minimal adaptiv**: nur kleine Umleitungen werden genommen, der Abstand zum Ziel wird nicht größer.
  - **maximal adaptiv**: beliebige Umleitungen werden genommen.

3.3 Kommunikationskriterien

Die Art des Routings ist abhängig von der Art der Kommunikation und vom Aufbau der Knoten (haben Puffer oder nicht, besitzen FIFO oder Stack).

- Hierbei gibt es sog.**Online**– und **Offline–Routing–Algorithmen**. Beim heute aktuellen Online–Routing wird das Start–Ziel–Muster erst bei Verschicken des Paketes errechnet. Beim Offline–Routing findet die Berechnung schon vorher statt. Dies ist problematisch, wenn die Wege von erst zu bestimmenden Ergebnissen abhängen. Hierzu folgendes Beispiel:

  ```
  for i=1 to p pardo
  F[F[i]] = F[i]
  ```

- **Delay des Routingverfahrens**: Wie lange benötigt ein Paket maximal von Start nach Ziel? Der Durchmesser des Netzes ist eine untere Schranke, da im schlimmsten Fall ein Paket vom einen 'Ende' zum anderen 'Ende' verschickt werden muß.

- **Durchsatz des Netzwerkes**: Hierunter versteht man die Auslastung des Netzwerkes. Wieviele Pakete können gleichzeitig verschickt werden? Beim Bus z.B. ist der Durchsatz schlecht (1).

3.4 Übersicht

Hier eine kleine Übersicht über die Leistungsfähigkeit der verschiedenen Netzwerktypen anhand von Beispielen:

- Gitter, d–dim, Seitenlänge N, routet Permutationen deterministisch in **O(dN)**
- Hypercube, n–dim, routet Permutationen randomisiert in **O(n)** (Valiant)
- Benês–Netzwerk, routet Permutationen offline deterministisch in **O(n)**
- Butterfly, n–dim, routet n–Relationen deterministisch offline oder randomisiert online in **O(n)** und besitzt Durchsatz T = n*2^n Pakete, das sind $\frac{T}{O(n)}$ = O(2^n) $\frac{Pakete}{Zeit}$.

Auf das Routen auf Butterfly–Netzwerken werden wir jetzt noch etwas näher eingehen:

3.5 Routing auf Butterflies

Zum Routen auf einem Butterfly–Netzwerk nimmt man die Binärdarstellung des Zielknotens und geht in Stufe i nach oben, falls Bit i der Binärdarstellung 0 ist, sonst geht man nach unten. Betrachte dazu das Beispiel in Abbildung 3.8:

Das Routing beginnt in Knoten 10. Zielknoten ist 01. Also wird in der ersten Stufe nach unten (0) und in der zweiten Stufe nach oben (1) geroutet.

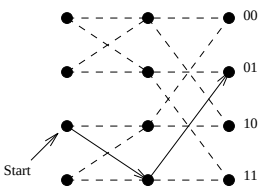


Abbildung 3.8: Bsp: Routen auf Butterflies

Der Korrektheitsbeweis wird durch Induktion über die Dimension des Netzes geführt.

- n=1, n=2: $\surd$
- n $\rightarrow$ n+1: Wir setzen voraus, daß das Verfahren auf einem n-dimensionalen Butterfly korrekt funktioniert und schließen nun auf ein n+1-dimensionales. Betrachte Abbildung 3.9.
  Ein Paket soll an die Adresse '0x' oder '1x' geroutet werden. Da ein n+1-Butterfly rekursiv aus zwei n-Butterflies aufgebaut werden kann, steht das Paket in beiden Fällen an Adresse 'x' der n-Butterflies.
  - falls das Ziel '1x' ist, wird im letzten Schritt nach unten
  - falls das Ziel '0x' ist, wird nach oben geroutet

  Das Ergebnis ist also richtig. □

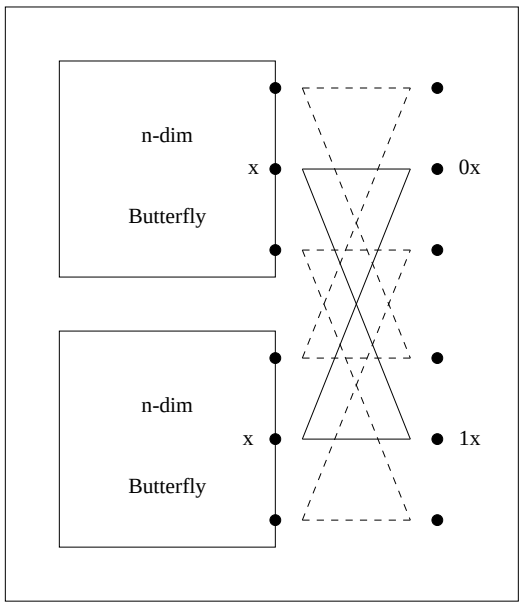


Abbildung 3.9: Routen auf Butterflies

3.6 Routing auf Hypercubes

Hier ein Algorithmus zum Routen auf n-dim Hypercubes:

- Nimm bitweise **xor** von Quelle und Ziel: $mask_i := q_i \oplus d_i$, wobei q_i die Adresse der Quelle und d_i die, des Zieles ist.
- Suche niedrigstes Bit i mit $mask_i = 1$. Gehe entlang Kante der i-ten Dimension.
- Lösche $mask_i$.
- Falls $mask = [0...0]$ ist das Ziel erreicht.

Betrachte Abbildung 3.10: Hier soll von Knoten '0000' zu Knoten '1011' geroutet werden.

Beweis: $mask_i = 1 \rightarrow q_i \neq d_i$, entlang Kante dim-i $\rightarrow$ ändere Bit i der Kantennummer. □

Das Routing auf CCC's ist ähnlich. Wenn $mask_i = 0 \rightarrow$, gehe entlang des Rings. □

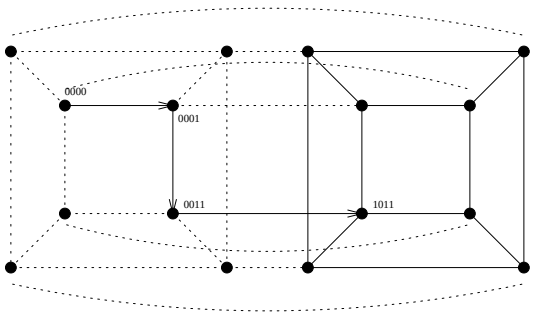


Abbildung 3.10: Routing auf Hypercubes

3.7 Routing auf eindimensionalen Gittern

Greedy-Route: Jeder Knoten sendet ein Paket 'greedy'. Das heißt: Das Paket läuft nach links (rechts) bis das Ziel erreicht ist.

- Bei Greedy-Route ist maximal ein Paket in einer Richtung in einem Knoten.
- Die Pakete behindern sich niemals gegenseitig → Zeit: $\leq$ N-1

3.8 Routing auf zweidimensionalen Gittern

Wegwahl:

- Phase 1: zuerst zur richtigen Spalte
- Phase 2: dann zur richtigen Zeile

Folgendes Problem tritt dabei auf: Zwei Pakete können sich in einem Knoten treffen und nicht gleichzeitig weitergeschickt werden (Abbildung 3.11). In solchen Fällen schafft das **farthest-first**-Protokoll Abhilfe. Wer

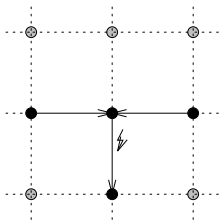


Abbildung 3.11: Problem beim Routen auf zweidimensionalen Gittern

weiter muß, darf zuerst den Knoten verlassen.

Der obige Algorithmus routet eine Permutation auf NxN-Gittern in Zeit 2N-2.
Beweis:

- Phase 1 benötigt Zeit N-1, wegen Greedy-Routing auf eindimensionalen Gittern.
- Phase 2: Lemma: Lineares Array, N Pakete, zu Beginn beliebig verteilt, am Ende jeder Knoten ein Paket. Falls Kollisionen: farthest-first. Dann dauert das Routing **N-1** Schritte. Aus dem Lemma folgt dann der Satz. □
  Beweis Lemma: Wende ∀ Pakete gleichzeitig Aussage in Übung 3.2 an. Gibt es ein Paket, das nach N-1 Schritten zwar einen Knoten in M, aber nicht im Ziel erreicht hat. → Widerspruch. □

Vorlesung Rechnerarchitektur II (Parallelrechner)
Dr. Jörg Keller

SS 94, Termin Nr. 4: 05.05.94
Skript: Carsten Scholtes

Shared Memory Systeme

4.1 Systeme mit gemeinsamem Speicher (shared memory)

4.1.1 Benutzersicht

Aus der Sicht eines Benutzers (Programmierers) bestehen Systeme mit gemeinsamem Speicher aus n Prozessoren $P_1, \ldots, P_n$, die alle auf den gleichen Speicher M (shared memory) zugreifen können. Die zugehörige Hardware kann anders aussehen (s.u.). Abbildung 4.1 zeigt die Benutzersicht auf solch ein System.

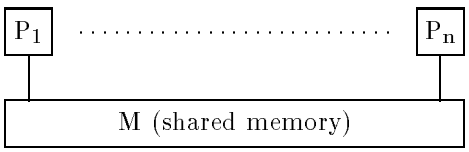


Abbildung 4.1: Benutzersicht auf ein System mit gemeinsamem Speicher

Vorteile

Kommunikation über den Speicher: Der Datenaustausch zwischen den Prozessoren erfolgt über den gemeinsamen Speicher. Diese Methode ist einfacher und schneller als das Versenden und Empfangen von Nachrichten in Systemen ohne gemeinsamen Speicher.

keine Datenverteilung: Alle Daten befinden sich im selben Speicher. Man muß sich also beim Programmentwurf nicht überlegen, welcher Prozessor welche Daten erhält.

Zeitverhalten

Je nach ihrem Zeitverhalten beim *Speicherzugriff* unterscheidet man bei Systemen mit gemeinsamem Speicher die folgenden Klassen:

UMA — Uniform Memory Access: Die Speicherzugriffszeit ist für alle Prozessoren und für alle Speicheradressen *gleich*.

Die *synchrone* Variante heißt PRAM (Parallel Random Access Machine). Insbesondere ist bei dieser Variante die Zugriffszeit 1, d.h. Zugriffe erfolgen genau so schnell wie bei einem Einzelprozessor.

NUMA — Non Uniform Memory Access: Die Speicherzugriffszeiten sind *variabel*. Deshalb müssen bei der Programmierung solcher Systeme besondere Strategien angewandt werden, um die Speicherzugriffe zu optimieren.

4.1.2 Realisierung

Zur Realisierung eines gemeinsamen Speichers, genauer gesagt, eines gemeinsamen Adreßraums, gibt es mehrere Möglichkeiten. Der gemeinsame Adreßraum kann durch einen *physikalisch* gemeinsamen Speicher verwirklicht werden. Er kann aber auch durch ein Netzwerk *emuliert* werden.

Physikalischer Speicher gemeinsam

Für kleinere Prozessorzahlen kann der gemeinsame Speicher physikalisch realisiert werden. Dabei sind die Prozessoren über einen *Bus* mit dem gemeinsamen Speicher verbunden. Für sehr schnelle Bussysteme und kleine Prozessorzahlen erreichen diese Systeme die UMA-Kriterien. Abbildung 4.2 zeigt den typischen physikalischen Aufbau solcher Systeme: Die Prozessoren $P_1,\ldots,P_n$ sind über einen Bus mit einem (oder mehreren) Speicher(n) $M_1,\ldots,M_i$ verbunden.

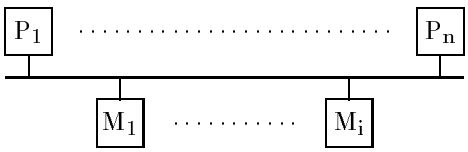


Abbildung 4.2: physikalisch realisierter gemeinsamer Speicher

Ein System, das dieser Architektur sehr genau folgt, ist die in Abbildung 4.3 dargestellte *Alliant FX/8* (1985-1991). Ihre 8 Prozessoren $P_1,\ldots,P_8$ sind über ein 8×2 Crossbar-Netzwerk mit 2 Caches C_1 und C_2 verbunden. Diese sind wiederum über einen Bus mit 8 Speichermodulen $M_1,\ldots,M_8$ verbunden.

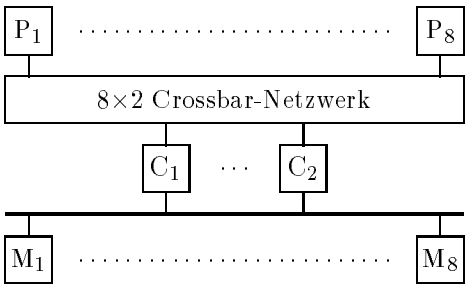


Abbildung 4.3: Architektur der Alliant FX/8

Anfangs wurden Motorola Prozessoren verwendet, später dann Intel i860-Prozessoren. Der Datendurchsatz des Crossbar-Netzwerks von den Prozessoren zu einem der Caches beträgt 376 MB/s, das entspricht im Mittel ca. 94 MB/s für jeden Prozessor (bei Verwendung beider Caches). Die Gesamt-Transferrate zwischen den Caches und den Speicherbausteinen beträgt 188 MB/s.

Neuere Architekturen (nach 1990), wie z.B. die *Sequent Symmetry* oder einige Maschinen von *SGI* und *SUN* verwenden für jeden ihrer 16 (bzw. 32) Prozessoren einen *eigenen* Cache. Die Caches selbst sind, wie in Abbildung 4.4 dargestellt, wieder über einen Bus mit dem Speicher verbunden.

Probleme treten auf, wenn mehrere Prozessoren über die Caches auf die gleichen Daten zugreifen. Deshalb sind Mechanismen für die *Cache-Kohärenz*[1] erforderlich.

Verteilter Speicher ohne Datenreplikation

Architektur Der gemeinsame Adreßraum wird über ein Netzwerk auf die Speichermodule verteilt. Bei diesem Modell können wesentlich mehr Prozessoren verwendet werden. Die Architektur socher Systeme folgt dem Schema in Abbildung 4.5.

[1] wird auch als *Cache-Konsistenz* bezeichnet

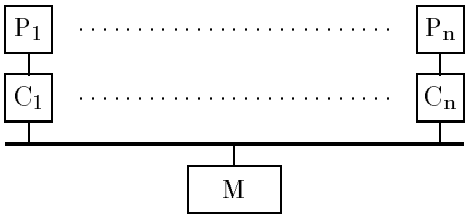


Abbildung 4.4: Architektur von Maschinen wie der Sequent Symmetry

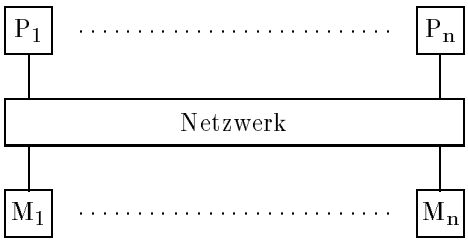


Abbildung 4.5: durch Netzwerk emulierter gemeinsamer Speicher

Die Verteilung eines Adreßraums $\{0,\ldots,m-1\}$ auf n Prozessoren wird mit einer *Hashfunktion* h realisiert.

$h : \{0,\ldots,m-1\} \mapsto \{0,\ldots,n-1\} \times \{0,\ldots,\frac{m}{n}-1\}$, mit

$h(x) = (i,j) \iff$ Speicherzelle x des gemeinsamen Adreßraums befindet sich auf (Speicher-) Modul i und hat dort lokale Adresse j.

Für eine Hashfunktion ergeben sich verschiedene *Forderungen*:

- Sie muß *einfach* sein. D.h.: Sie muß schnell von jedem Prozessor berechnet werden können.
- Es dürfen nicht zu viele Anfragen auf *ein* Modul abgebildet werden. Beispielsweise wäre die Funktion $h(x) = (0,x)$ sehr einfach zu berechnen. Da aber alle Anfragen auf das Modul 0 gehen, ist sie eine schlechte Hashfunktion.

In der obigen Definition der Hashfunktion h wird angenommen, daß h eine bijektive Funktion ist. Es sind auch Hashfunktionen denkbar, die nur injektiv, aber nicht surjektiv sind. In diesem Fall muß die Definition von h so abgeändert werden, daß sie Zieladressen,j, bis zu einer Größe $l \geq \frac{m}{n} - 1$ erlaubt. l ist die höchste Adresse eines Speichermoduls:

$h : \{0,\ldots,m-1\} \mapsto \{0,\ldots,n-1\} \times \{0,\ldots,l\}$

Routing Der Zugriff auf eine bestimmte Adresse des gemeinsamen Speichers erfolgt mittels eines *Anfragepakets*, das über das Netzwerk verschickt wird (*Routing*). Dabei kann man die folgenden Schritte unterscheiden:

1. Berechnen des Zielmoduls
2. Senden des Pakets an das Zielmodul
3. Ausführen des gewünschten Zugriffs
4. Rücksenden des Pakets (nur bei Lesezugriffen)

Als Beispiel wird im folgenden das Routing der *Fluent Machine* (Ranade 1987) vorgestellt. Es handelt sich hierbei um ein statisches[2] Butterfly-Netzwerk. An jedem Knoten befinden sich ein Prozessor und ein Speichermodul. Der Zugriff erfolgt über 1 Butterfly-Netzwerk mit bidirektionalen Links in 6 Phasen (siehe Abbildung 4.6). Dabei wird das Netzwerk in den Phasen 1, 3 und 5 vorwärts betrieben, während es in den Phasen 2, 4 und 6 rückwärts arbeitet. Im folgenden werden die Phasen einzeln beschrieben:

[2]jeder Knoten ist immer Sender und Empfänger

1. Die Pakete werden an den Rand des Netzes geschoben. Sie verlassen die Zeile ihres Startknotens nicht.
2. Die Pakete werden entlang dem eindeutigen Pfad von der Reihe ihres Startknotens in die Reihe ihres Zielknotens verschickt.
3. Die Pakete werden in der Reihe ihres Zielknotens weitergeschoben, bis sie ihr Ziel erreicht haben. Dort wird der entsprechende Zugriff ausgeführt. Bei einem Lesezugriff wird das Paket mit dem gelesenen Wert an den Rand geschoben.
4. Die Pakete werden an den anderen Rand geschoben, bleiben aber immer noch in der Reihe ihres Ziels. Sie befinden sich jetzt in denselben Knoten, in denen sie auch zum Ende der Phase 2 waren.
5. Die Phase 5 läuft im Gegensatz zur Phase 2 vorwärts. Deshalb können die Pakete jetzt auf dem bereits durchlaufenen Pfad in die Reihe ihres Startknotens zurückkehren.
6. Die Pakete werden bis zu ihrem Startknoten zurückgeschoben.

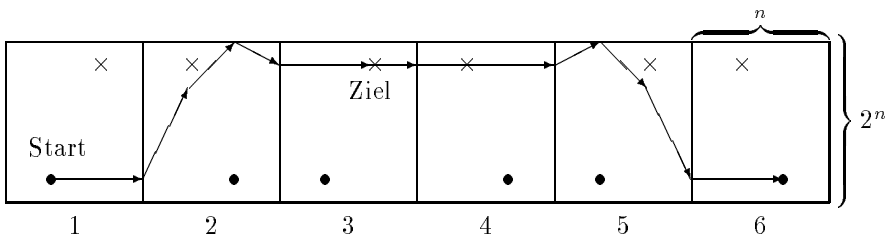


Abbildung 4.6: Routing im Butterflynetzwerk

Die Phase 4 (Dummy-Phase) ist notwendig, damit die Phase 5 vorwärts arbeitet, also umgekehrt wie die ihr entsprechende Phase 2. Außerdem befinden sich die Pakete nach Phase 4 am richtigen Ende des Netzwerks (In den selben Knoten wie am Ende der Phase 2). Der Rückweg eines Pakets in den Phasen 5 und 6 verläuft identisch zu dem in den Phasen 1 und 2 zurückgelegten Weg.

Deshalb müssen die Pakete keine Information über ihren Startknoten enthalten. Es genügt, wenn in den Knoten, die sie passieren, vermerkt wird, über welchen Eingang die Pakete eintreffen, und über welchen Ausgang sie den Knoten wieder verlassen.

Bei einem geschickten Scheduling (s.u.) passieren die Pakete die Knoten in der Phase 5 in der gleichen Reihenfolge wie in Phase 2. Deshalb genügt zur Speicherung der Wege-Informationen eine FIFO pro Knoten. Ein Element dieser FIFO besteht aus 2 Bits: Das erste Bit, besagt, ob ein Paket über den oberen oder unteren Eingang angekommen ist. Das zweite enthält die entsprechende Information für den Ausgang.

Scheduling Jeder Netzwerk-Knoten besitzt, wie aus Abbildung 4.7 ersichtlich, zwei gepufferte Eingänge.

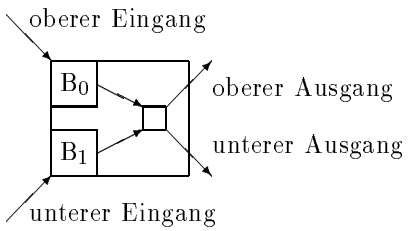


Abbildung 4.7: ein Netzwerkknoten

Steht in beiden Eingangspuffern, B_1 und B_2, ein Paket, wird das mit der kleineren (gehashten) Adresse weitergesendet. Haben beide die gleiche Adresse, werden sie *kombiniert*. Dabei kann man die folgenden Fälle unterscheiden:

konkurrierendes Schreiben : Eines der beiden Pakete wird *entfernt*. Dazu gibt es eine Prioritätsfunktion auf den Prozessornummern. Das Paket mit der höheren Priorität bleibt erhalten. (Oft wird als Priorität die Prozessornummer selbst verwendet.)

konkurrierendes Lesen : Nur eins der Pakete wird zum Zielknoten weitergeleitet. Auf dem Rückweg muß die Antwort verdoppelt werden. Um diese Information zu speichern, genügt es die FIFO zur Speicherung der Weginformationen um ein 3. Bit pro Eintrag zu erweitern. Dieses Bit besagt, ob das gerade zurückkehrende Paket verdoppelt werden muß oder nicht.

Lesen und Schreiben : Dieser Fall ist normalerweise nicht erlaubt.

Diese Art des Scheduling sortiert die Pakete, die den Knoten verlassen nach aufsteigender Adresse.

Ein Problem entsteht, wenn einer der Eingangspuffer leer ist. Abbildung 4.8 veranschaulicht das Problem an einem Beispiel.

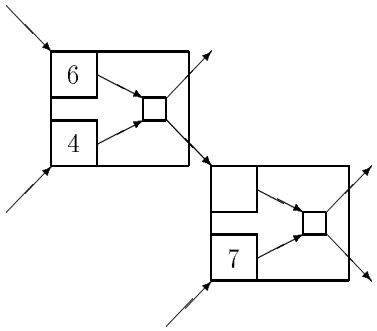


Abbildung 4.8: Beispiel für eine notwendige Verzögerung beim Versenden eines Pakets

Es sind zwei Knoten abgebildet. Die Eingangspuffer des linken Knotens sind mit Paketen belegt, die an die Adressen 4 und 6 gerichtet sind. Der (eindeutige) Weg des Pakets mit der Adresse 6 führt über den rechten Knoten. Das Paket mit der Adresse 4 muß nach oben. Da das Paket mit der kleineren Adresse Vorrang hat, ist das Paket mit der Adresse 6 blockiert. (Es könnte z.B. noch ein Paket mit der Adresse 5 zum unteren Eingang des linken Knotens gelangen.) Der rechte Knoten hat nur ein Paket mit Zieladresse 7. Er kann in diesem Fall nicht entscheiden, ob noch ein Paket ankommen wird, das eine kleinere Adresse als 7 hat. Das bedeutet, daß der rechte Knoten warten muß (unter Umständen sehr lange).

Es gibt aber auch ähnliche Situationen, in denen ein Paket schon versendet werden kann, obwohl der andere Puffer des Knotens, in dem es sich befindet, leer ist. Dies ist der Fall, wenn bekannt ist, daß kein weiteres Paket mit einer kleineren Adresse mehr in den anderen Puffer gelangen kann. Solch ein Fall ist in Abbildung 4.9 dargestellt.

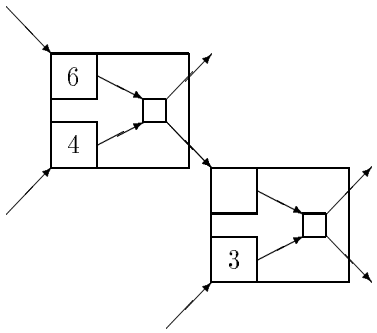


Abbildung 4.9: Beispiel für eine unnötige Verzögerung beim Versenden eines Pakets

Die Pakete mit den Adressen 4 und 6 verhalten sich wie im vorigen Beispiel. Im nächsten Schritt wird also das Paket mit der Adresse 4 nach oben gesendet. Der rechte Knoten muß warten, obwohl in seinen oberen Puffer

nur noch Pakete mit Adressen größer als 4 gelangen können. Das Paket mit der Adresse 3 könnte also gefahrlos weitergesendet werden.

Um unnötige Wartezeiten zu vermeiden, werden *Ghost*-Pakete versendet. Diese Pakete enthalten keine Speicheranfragen. Sie liefern lediglich eine untere Schranke für die zukünftigen Adressen von Paketen, die über den gleichen Eingang wie das Ghost-Paket ankommen können. Die Puffer eines Knotens enthalten dadurch immer entweder eine Speicheranfrage oder ein Ghost-Paket.

Im Beipiel würde der linke Knoten ein Ghost-Paket mit Adresse 4 an den oberen Eingang des rechten Knotens schicken. Mit dieser Information kann der rechte Knoten das Paket mit der Adresse 3 weiterschicken.

Ghost-Pakete, die nicht verschickt werden können, werden gelöscht. Dadurch behindern sie nicht die eventuell nachfolgenden Speicheranfrage-Pakete.

Ein Paket muß also nur noch warten, wenn es allein in einem Knoten ist, und das Ghost-Paket im anderen Puffer eine kleinere Adresse hat. Gelangt in solch einen Knoten kein weiteres Paket mit einer höheren Adresse, bleibt das Paket in diesem Knoten hängen.

Abhilfe schafft hier das Versenden spezieller Ghost-Pakete mit der Adresse ∞, sogenannte *EOS*-Pakete (End Of Stream). Solch ein Paket wird von jedem Knoten direkt nach seiner Anfrage ins Netzwerk geschickt. Im Gegensatz zu normalen Ghost-Paketen werden EOS-Pakete nie gelöscht. Dies garantiert, daß kein Paket im Netzwerk hängen bleibt.

4.1.3 Multipräfix

Definition 4.1 (Multipräfix) *Sei $\circ$ ein assoziativer Operator und A eine Adresse mit Inhalt D. Führen die Prozessoren $i_1 < i_2 < \ldots < i_k$ die Operation $MP(A, \circ, D_{i_j})$ (Multipräfix) aus für $j = 1, \ldots, k$, dann gilt am Ende des Schritts:*

$Inhalt(A) = D \circ D_{i_1} \circ \ldots \circ D_{i_k}$ und

Prozessor i_j erhält: $D \circ D_{i_1} \circ \ldots \circ D_{i_{j-1}}$

Vorlesung Rechnerarchitektur II (Parallelrechner)
Dr. Jörg Keller

SS 94, Termin Nr. 5: 19.05.94
Skript: Peter Zorn

Fluent Machine Routing

5.1 Routing (Fortsetzung)

Definition 5.1 (Multipräfix)
Sei $\circ$ ein assoziativer binärer Operator, A eine Adresse mit Inhalt(A)=D.
Die Prozessoren $i_1, \ldots, i_k$ $(i_1 < i_2 < \ldots < i_k)$ enthalten die Daten $D_{i_1}, \ldots D_{i_k}$.
$Proz_{i_j}$ führt $MP(A, \circ, D_{i_j})$ aus $(j = 1, \ldots, k)$. Dann gilt am Ende des Schrittes:
Inhalt(A)=$D \circ D_{i_1} \circ \ldots \circ D_{i_k}$ und $Proz_{i_j}$ enthält $D \circ D_{i_1} \circ \ldots \circ D_{i_{j-1}}$

Beispiel:
$\circ = +$, Inhalt(A)=2, $k = 3$
$i_1 = 4,\ i_2 = 7,\ i_3 = 18,\ D_4 = 8,\ D_7 = -3,\ D_{18} = 6$
$\Rightarrow Inhalt(A) = 13 = 2 + 8 - 3 + 6$ und es gilt:
$Proz_4$: enthält: $D = 2$
$Proz_7$: enthält: $D \circ D_{i_1} = 2 + 8 = 10$
$Proz_{18}$: enthält: $D \circ D_{i_1} \circ D_{i_2} = 2 + 8 - 3 = 7$

Anwendung: Sei A ein Zeiger auf den Anfang des freien Speichers, D_{i_j} die Größe des von $Proz_{i_j}$ angeforderten Speicherblocks. $\Rightarrow$ am Ende des Schrittes gilt:

- A zeigt wieder auf den Anfang des freien Speichers.
- Jeder Prozessor hat einen Pointer auf den ihm zugewiesenen Speicher.

Bemerkung: Nur das Allokieren des Speichers funktioniert mit Multipräfix. Die Freigabe muß anders geregelt werden.

Problem: Die Prozessoren sind 2-dimensional numeriert, der Speicher aber 1-dimensional. Daraus ergibt sich folgende Implementierung.

5.1.1 Implementierung

Der Prozessor in Zeile i, Spalte j hat die Nummer $i * n + j$. Siehe dazu 5.2.

Die Knoten in den Stufen 1,2,5,6 (= Phasen des Scheduling) werden erweitert. (vgl. Kombination von Paketen (Phase 1,2) und Rückgängigmachen der Kombination (Phase 5,6)). Siehe dazu Abbildung 5.3.

Korrektheit: Beweis durch Induktion über Tiefe (Übung 5.1)

5.1.2 Korrektheit

Die Korrektheit des Routing wird durch folgende Schritte bewiesen:

- Jedes Paket läuft über den richtigen Pfad:
  Ja, da Pfad eindeutig ist. Ist klar für Phase (1,3,4,6) da 'Geradeausbewegung'. Gilt auch für Phase 2,5 wegen Eindeutigkeit der Wege im Butterfly-Netz.

- Korrektheit der Kombinierung: Übung 4.1
- Beginn und Ende des Routing:
  Phase 1,2: Problem: Anfangs ist Puffer leer – in Phase 1 kommt ein Paket ins Netz, allerdings ist nur ein Puffer gefüllt.

  Lemma 3 *Der Knoten in Tiefe i (wobei von links an mit 0 beginnend gezählt wird)*
  i) hält ab Takt i immer Pakete (incl. GHOST-Pakete) in beiden Puffern und
  ii) verschickt in jedem Takt ein Paket

  [$\Rightarrow$ spätestens nach n Takten haben alle Knoten in Phase 1 zwei Pakete $\Rightarrow$ Routing $\surd$]

  Beweis: Induktion über Tiefe
  i) i=0: Sonderfall (da Knoten nur 1 Puffer hat, da er am linken Rand liegt) $\Rightarrow \surd$
  i=i+1: nach der Induktionsvor. wird zur Zeit i von Knoten in Tiefe i ein Paket versandt
  $\Rightarrow$ dieses ist zur Zeit $i+1$ in Knoten der Tiefe $i+1$
  Phase 1: der andere Puffer enthält Paket vom Prozessor.
  Phase 2: gilt für beide Eingangspuffer
  ii) Versenden folgt nach Scheduling-Vorschrift.
- **Lemma 4** *GHOST-Pakete warten nie (das Netz wird nie 'zugemüllt')*

  Beweis: GHOST-Pakete werden entweder verschickt oder nach Definition gelöscht.

  Phase 3,4: Die Pakete wandern nur $\surd$
- Phase 5,6: [z.z. Pakete kommen in gleicher Reihenfolge an, wie sie abgeschickt wurden $\Rightarrow$ FIFO Richtungsqueue korrekt]

  Lemma 5 *FIFO-Queues für Richtungsbestimmung reichen aus, d.h. die Antworten kommen in der gleichen Reihenfolge wie die Pakete.*

  Beweis: Übung 5.2

Daraus folgt die Korrektheit des Routing.

5.1.3 Zeitverhalten

Satz 5.1 *Bei geeignet gewählter Klasse von Hash-Funktionen dauert ein Schritt der Fluent-Machine Zeit $O(n)$ mit Wahrscheinlichkeit $\geq 1 - \frac{1}{n2^n}$ ($n2^n$ = Anzahl der Prozessoren) bei Puffern der konstanten Länge b.*

Bemerkung:

- Konstante Pufferlänge hat zur Folge, daß nur ein Puffer gebaut werden muß und dieser für verschieden skalierte Maschinen eingesetzt werden kann.
- Das Verfahren kann länger dauern, aber funktioniert trotzdem korrekt.

Beweis: Es ist zu zeigen:

1. Wenn der Schritt lange dauert, dann gibt es lange *polarisierte Sequenz* von Paketen (d.h. Paket A hält B auf, B hält C auf, usw. $\Rightarrow$ wenn der Schritt x Takte dauert, gibt es Sequenz von x Paketen)
2. bei geeigneter Klasse von Hash-Funktionen sind lange polarisierte Sequenzen selten.

Definition 5.2 *Pfad S im Netzwerk ist eine Folge $S = \{S(i)\}_i$, so daß $S(i)$ und $S(i+1)$ verbunden sind und damit $S(i) \neq S(i+1)$.*

Bemerkung: es wird nur eine Aussage über *paarweise* Ungleichheit gemacht.

Definition 5.3 *Ein Pfad von Spalte 0 bis Spalte 6n ($\Rightarrow$ Tiefe $6n+1$) heißt I/O-Pfad.*
Für Paket m sei tag(m) die gehashte Adresse.
Sei $M = m_1, \ldots m_\alpha$ Folge von Paketen mit $tag(m_i) < tag(m_{i+1})$, sei S Pfad im Netzwerk, so daß m_i (beim Routing) durch die Knoten $S(j_i)$ kommt und daß $j_i \leq j_{i+1}$. Dann heißt M polarisiert entlang *S.*

Problem: auf welche Art kann Paket aufgehalten werden → lange polarisierte Sequenz finden.

Lemma 6 *Wenn ein Paket im Knoten s zur Zeit t warten muß, dann gilt entweder:*

1. *m' mit $tag(m') < tag(m)$ wurde verschickt, oder*
2. *Puffer eines Knoten s' in der nächsten Stufe ist voll und enthält Pakete $m''_1, \ldots, m''_b$ mit $tag(m''_b) < tag(m''_{b-1}) < \ldots < tag(m''_1) < tag(m)$, oder*
3. *Ein Puffer in s ist leer und $level(s) \geq t$. (Es gilt $level(s)$ = Tiefe(Knoten s))*
4. *Das sind alle.*

Notation: 1. heißt *m-delay* und 2. heißt *b-delay*

Beweis:

1. gilt wegen Scheduling
2. folgt aus Lemma 2 (siehe Übung 4)
3. folgt aus Lemma 3
4. Scheduling:
   Fall i: m ist vorne in Queue ⇒ Fall 3.-1.
   Fall ii: nein ⇒ Fall 1. oder 2.

Konstruiere polarisierte Sequenz
m_1: Paket, das am meisten aufgehalten wurde.
m_2: Paket, das m_1 aufhält.
m_3: etc. ...
Achtung: Paket kann auch von GHOST aufgehalten werden.

m'_1 wird abgeliefert nach $t_0 = 6n + \delta$ Takten. ($6n$ = Länge des längsten Pfades im Netzwerk, δ = Verzögerung)

Zeige, wie aus m'_i und t_{i-1} Paket m'_{i+1} berechnet wird:
$i = 1$: Induktionsanfang war m'_1
$i \rightarrow i + 1$: Falls $m'_i \neq$ GHOST, dann $m_i := m'_i$. Sonst verfolge Weg des GHOST zurück bis zu Knoten, wo GHOST durch m_i erzeugt wurde. Verfolge Weg vom m_i zurück bis zum Knoten s_i, indem m_i zur Zeit t_i wartete. Muß m_i warten:

Fall 1: *m-delay*: $m'_{i+1} = m'$, das anstatt m_i verschickt wurde.
Fall 2: *b-delay*: $m_{i+j} = m'_{i+j} = m''_j$, $1 \leq j \leq b-1$
$t_{i+j} = t_i \qquad s_{i+j} = s'$ (Pakete, die im Puffer stehen, aber nicht am Anfang des Puffer)
$m'_{i+b} = m''_b$
Fall 3: Puffer in s_i leer oder falls $t_i = 0$, beende Verfahren.

Wende Konstruktion an, bis $\frac{\delta}{b}$ b-delays vorkommen (⇒ Sequenz hat Länge δ), oder bis Verfahren beendet.

Definition 5.4 *Für Paket m in Knoten s zur Zeit t sei $lag(m, s, t) = t - level(s)$ (= Verzögerung(s)) ⇒ lag ist untere Schranke für bisherige Verzögerung von m.*

Lemma 7

1. *Falls m_i m-delayed, dann gilt:*
   $lag(m'_i, t_{i-1}) = lag(m_i, t_i) + 1 = lag(m'_{i+1}, t_i) + 1$
2. *Falls m_i b-delayed, dann gilt:*
   $lag(m'_i, t_{i-1}) = lag(m_i, t_i) + 1 = lag(m'_{i+b}, t_i) + 2$

Beweis: m_i wartet zur Zeit $t_i \Rightarrow lag(m_i, t_i + 1) = lag(m_i, t_i) + 1$
von t_{i+1} bis t_{i-1} wartet m_i und m'_i nicht !
$\Rightarrow lag(m'_i, t_{i-1}) = lag(m_i, t_i + 1)$ dann gilt für 1. und 2. der linke Teil der Gleichung.

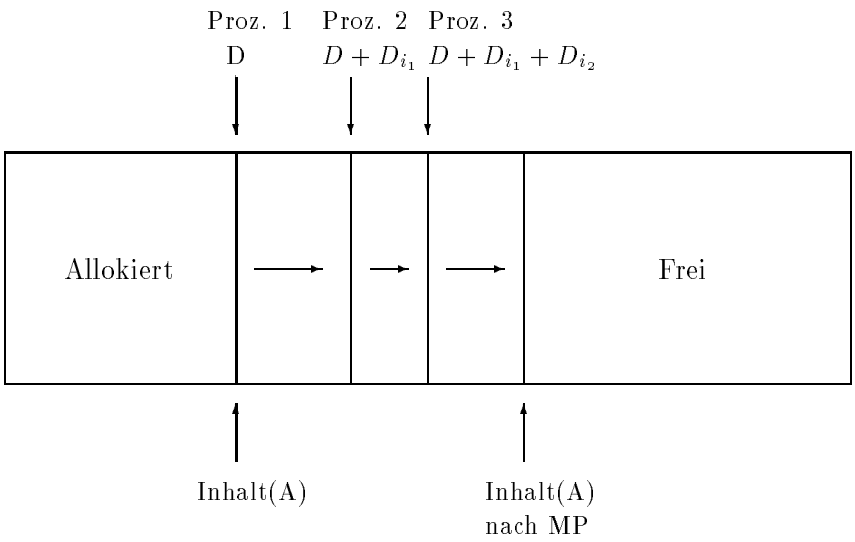


Abbildung 5.1: Speicherallozierung mit MultiPräfix

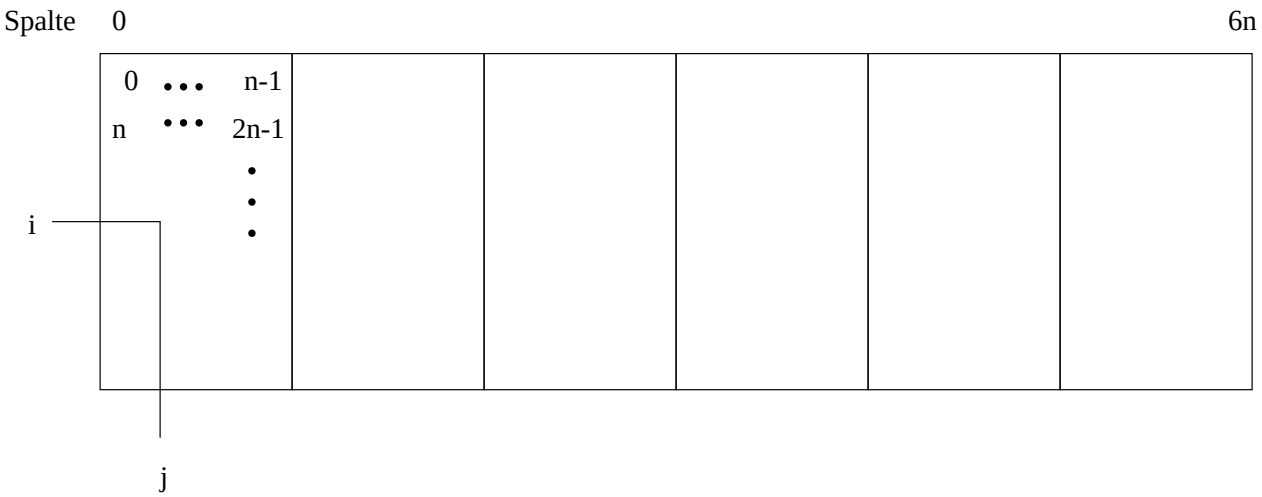


Abbildung 5.2: Butterfly mit Prozessornummern

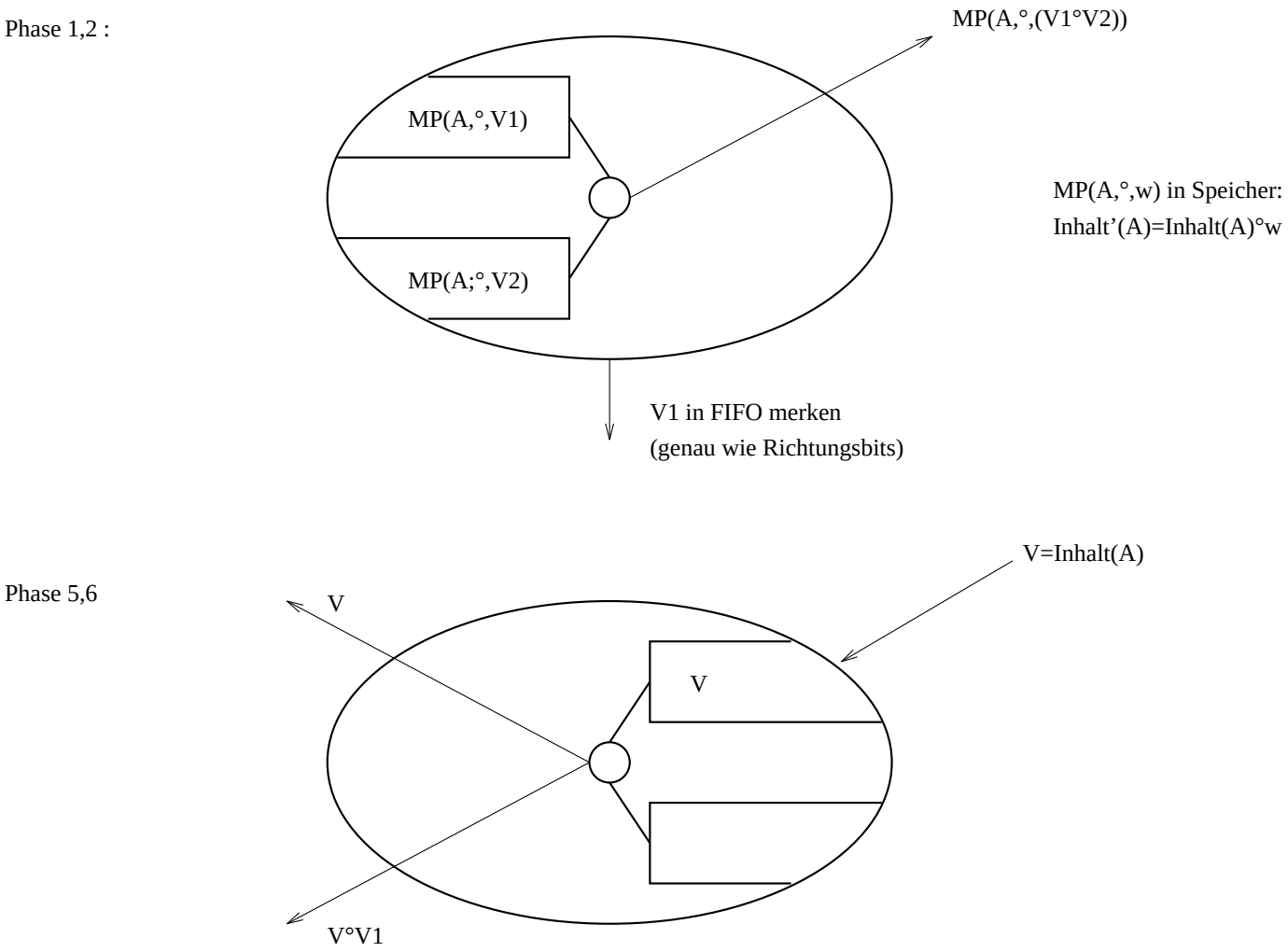


Abbildung 5.3: Implementierung von Multipräfix

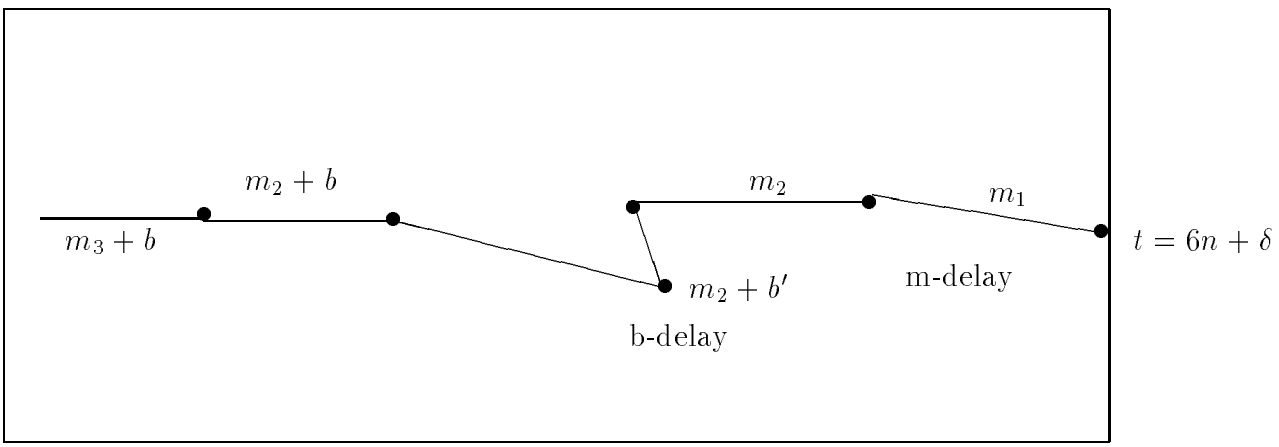


Abbildung 5.4: Polarisierte Sequenz

Vorlesung Rechnerarchitektur II (Parallelrechner)
Dr. Jörg Keller

SS 94, Termin Nr. 6: 26.05.94
Skript: Stefan Franziskus

Fluent Machine Routing II

6.1 Fluent Machine (Fortsetzung)

Ende Beweis Lemma:

1. m_i und m'_{i+1} sind zur gleichen Zeit im gleichen Knoten $\Rightarrow lag(m_i, t_i) = lag(m'_{i+1}, t_i)$
2. m'_{i+b} ist im nächsten Level, $\Rightarrow lag(m_i, t_i) = lag(m'_{i+b}, t_i) + 1$

□

Lemma 6.1 *Die konstruierte Sequenz hat Länge* $v \geq \delta$

Beweis: Sei j die Anzahl der Konstruktionsschritte, f sei die Anzahl der b-Delays
$f \leq \frac{\delta}{b}, j - f \geq \#m$-Delays.

1. Fall: $f = \frac{\delta}{b}$
$v \geq f * b = \delta$

2. Fall: $f \leq \frac{\delta}{b}$
$\Rightarrow lag(m_v, t_v) \leq 0$
$lag(m'_1, t_0) \geq \delta$

Lemma 7 $\underset{j-mal}{\Longrightarrow}$ Differenz der $lags = j - f + 2f = j + f$
$\Rightarrow j + f \geq \delta$
$\Rightarrow v = j - f + f * b = j + f(b-1) \geq j + f \geq \delta$, für $b = 2$
□

Lemma 6.2 *Der Pfad* $S_1, \cdots, S_v$ *hat höchstens die Länge* $6n + 2\frac{\delta}{b}$

Beweis: $\leq 6n$ Levels. $\Rightarrow$ # Kanten $\leq 6n + 2\frac{\delta}{b}$
□

Theorem 6.1 *Braucht ein Schritt der Fluentmachine* $6n + \delta$ *Takte, dann gibt es I/O-Pfade der Länge* $6n + 2\frac{\delta}{b}$ *und eine Sequenz der Länge* δ*, die entlang des Pfades polarisiert ist.*

Beweis: Übung

Theorem 6.2 *Mit Wahrscheinlichkeit* $\geq 1-(n2^n)^k$ *dauert ein Schritt Zeit* $\leq 6n+kn$.

Beweis: Annahme: 1 Schritt dauert Zeit $\geq 6n+k_1 n$
$\Rightarrow \exists$ polarisierte Sequenz der Länge $\delta = (k_1-6)n$

a) Wähle Pfad der Länge: $2^{7n}+2f\binom{6n+2f}{f}$ Möglichkeiten nach Übung 6.2

b) Wähle Knoten auf Pfad, $f = \#b$-delays
$\binom{6n+2f+\delta}{\delta}\binom{s+r-1}{r} < \binom{s+r}{r}$ Möglichkeiten

c) Wähle Source-Prozessoren der δ Pakete
$\Rightarrow$ Adresse in PRAM-Adressraum und n Bits in gehashter Adresse festgelegt.

d) Wähle gehashte Adressen: P Möglichkeiten $\Rightarrow \frac{P}{2^n}$ Möglichkeiten für verbleibende Bits
$\binom{\frac{P}{2^n}+\delta}{\delta}$ Möglichkeiten für verbleibende Bits der polarisierten Sequenz.
$\Rightarrow (c+d)(n2^n)^\delta\binom{\frac{P}{2^n}+\delta}{\delta}$ Möglichkeiten.
Hash: Polynom vom Grad $\delta-1$ über $\mathbf{Z}/P\mathbf{Z}$ P Primzahl, > Speichergröße
Polynom des Grades $\delta-1$ ist durch δ Funktionswerte (gehashte Adressen) eindeutig bestimmt. P^δ Polynome, $\Rightarrow$ die Wahrscheinlichkeit einer bestimmten Sequenz gehashter Adressen ist $= \frac{1}{P^\delta}$

$$\begin{aligned} Prob(\geq 6n+\delta) &\leq \frac{2^{7n+2f}\binom{6n+2f}{f}\binom{6n+2f+\delta}{\delta}(n2^n)^\delta\binom{\frac{P}{2^n}+\delta}{\delta}}{P^\delta} \\ &\leq N^{-k} \\ \Rightarrow Prob(\leq 6n+\delta) &\geq 1-N^{-k} \end{aligned}$$

□

Übung 6.3 $\Rightarrow \delta = kn$

6.2 Realisierung

6.2.1 SB-PRAM

- Statt Polynomen benutzt man $h(x) = ax \bmod m$;
  a ungerade
  m Speichergröße, 2er Potenz
  Vorteil: bei Load/Store mit Multiplikation ist Hashing gratis.
- Phasen 3 und 4 gibt es nicht
- Phasen 1 und 6 benutzen lineare Sortierfelder
- Prozessoren einer Zeile werden zu einem physikalischen Prozessor mit $c*n$ Registersätzen
  $c*n$ wegen Netzwerklatenz $\Rightarrow c \approx 12$;
  Delayed Load; Netzwerkknoten doppelt so schnell $\Rightarrow c \approx 3$
- Problem: bei Sortierfeldern der Länge $c*n$ dauert es eine Runde bis die Pakete sortiert sind.
  $\Rightarrow$ Länge $\frac{c*n}{x}$, $x = 4$
- Rundentrennung: Sortierfeld schickt ein End Of Round nach den Paketen der Runde
- 128 Prozessoren, 32 virtuelle Prozessoren, 8 MHz Clock, 2 GByte = 512 MWord, (1 Word = 4 Byte)

6.2.2 TERA

- 3D-Torus (depopulated)
- 16^3 Knoten = 4096 Knoten
- 256 Prozessoren + I/O-Prozessoren
- 512 MM (Memory Module), 256 I/O-Memories)
- Prozessor superskalar, mehrere Registersätze
- Hashing: Tabellen "Memory Maps", Multiprefix "+" auf MM; kein Combining
- Trick: LD-Instruktion spezifiziert in 3-Bit-Feld wieviele Instruktionen danach unabhängig ausführbar sind. ($\Rightarrow$ optimierender Compiler)

6.3 Shared-Memory mit Caches

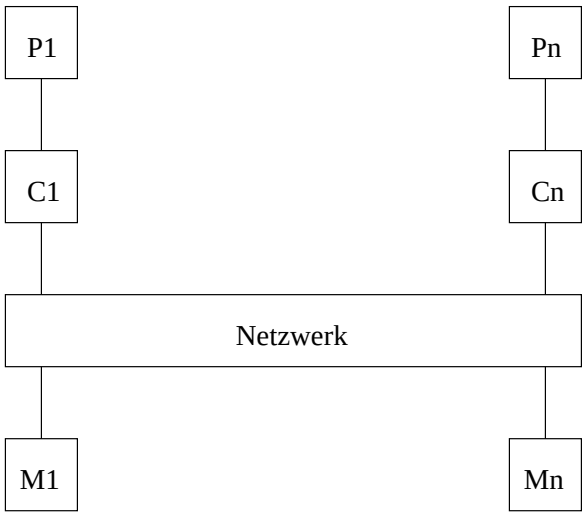


Abbildung 6.1: Shared-Memoy mit Caches

zu vermeiden:

1. *Stale memory*, Proz. A cachet Wert und verändert Wert, Proz. B lädt gleiches Datum aus dem Hauptspeicher und erhält alten Wert.
2. *Stale Cache*, Proz B las Wert schon vorher auch wenn A Wert im Hauptspeicher aktualisiert, hat Cache von Proz B alten Wert.

6.3.1 Netzwerk-Bus

Write-Through Cache $\Leftrightarrow$ Update im Cache geht auch zum Hauptspeicher $\Rightarrow$ Fehler 1 kommt nicht vor.
Fehler 2 beheben durch

1. *write-broadcast*: Caches hören den Bus ab, schreibt ein Cache ein Update zum Hauptspeicher updaten alle anderen Caches (wenn sie diesen Wert haben)
2. *invalidieren*: schreibt ein Cache ein Update zum Hauptspeicher, dann löschen alle anderen Caches diesen Wert (falls sie ihn haben)

6.3.2 “Snooping”-Protokolle

Write-Back-Caches ⇒ nur wenn ein veränderter Wert aus dem Cache gelöscht wird wird der Hauptspeicher aktualisiert. Den Fehler 1 muß man zusätzlich beheben: schickt B eine Anfrage auf den Bus, so muß der Cache von A antworten statt der Hauptspeicher.

6.3.3 “Split-Phase”-Protokolle

+ höherer Durchsatz
- Speicher ist jetzt aktiv

Vorlesung Rechnerarchitektur II (Parallelrechner)
Dr. Jörg Keller

SS 94, Termin Nr. 7: 9.6.94
Skript: Markus Jakob

Distributed Shared Memory Systeme

7.1 PRAM (Wiederholung)

Bei einer PRAM-Emulation wird der Adreßraum gleichmäßig und zufällig verteilt.

Dies führt zur einer *a priori Kenntnis der Latenz* des verwendeten Netzwerks; diese kann durch einen *einfachen Mechanismus kaschiert* werden.

Leider wird dadurch *keine Lokalität in den Zugriffen ausgenutzt.*

Insgesamt hat man aber ein klares, aus der Theorie enstandenes Schema, das für alle Anwendungsbereiche gleich gut geeignet ist.

7.2 VSM (Virtual Shared Memory)

7.2.1 Einführung

VSM Systeme wurden aus der Praxis heraus entwickelt, um technisch-wissenschaftliche Programme schnell ausführen zu können. Der Adreßraum ist hier seitenweise verteilt, wobei eine Seite einer Cache Line entspricht.

Daten werden dorthin gebracht, wo sie benötigt werden. Man erreicht dadurch eine *örtliche Lokalität*, wodurch der Einsatz von *Caches* ermgöglicht wird. Allerdings wird auch *Datenreplikation* nötig, weshalb diese Systeme auch als *Shared Memory mit Replikation* bezeichnet werden.

Da immer ganze Seiten kopiert werden und da in Programmen auf einen Zugriff auf Adresse x oft ein Zugriff auf eine Adresse in der Nähe von x folgt, wird die *zeitliche Lokalität* von Speicherzugriffen ausgenutzt.

Im Unterschied zu bus-basierten Architekturen existiert das shared memory nur noch virtuell als Adreßraum.

7.2.2 Konsequenzen

1. Stale memory kann nicht mehr vorkommen.
2. Es muß gesagt werden, wer die Kontrolle über welche Seite hat: Festlegen des Ownership von Seiten.
3. Man muß zwischen Schreib- und Lesezugriffen unterscheiden; es dürfen mehrere Lesekopien, aber nur eine Schreibkopie existieren.
   - Schreiben auf Seite ohne Schreibrecht:
     - write-invalidation aller anderen Seiten
     - evtl. Kopie besorgen, Zugriffsrecht auf Schreiben setzen
   - Lesen aus nicht lokaler Seite:
     - falls Prozessor mit Schreibrecht auf diese Seite existiert, setze dieses zurück
     - Kopie besorgen, Zugriffsrecht auf Lesen setzen

Der letzte Punkt führt zum Problem des *false Sharing*. Es tritt auf, wenn zwei unabhängige Objekte auf einer Seite liegen und mehrere Prozessoren um Zugriffsrechte konkurrieren. Es müssen dann laufend die Zugriffsrechte verändert werden, womit der Vorteil des Nichtpartitionierens fast weg ist; die Software muß also doch auf die Architektur Rücksicht nehmen.

Zur Verwaltung von Ownership, Kopien und Zugriffsrechten ist ein *Manager* nötig. Damit dieser nicht zum Bottleneck der Maschine wird, muß er i. allg. verteilt sein. Für bestimmte strukturierte Anwendungen ist auch ein zentraler Manager brauchbar.

7.2.3 Seitenverwaltung

Bei der Seitenverwaltung gibt es prinzipiell zwei Möglichkeiten:

- *fest*: realisiert durch eine Hasfunktion
- *dynamisch*: realisiert durch Tabellen, wodurch der Speicherbedarf erhöht wird

Bei den Realisierungen für den Manager gibt es zwei Varianten:

- Bei den *VSM Workstation Clustern* übertrifft die Rechen- die Kommunikationsgeschwindigkeit bei weitem; daher wird der *Mananger hier softwaremäßig* realisiert. Ein Beispiel ist das *Amoeba System* von Tannenbaum und anderen aus Amsterdam.
- Bei *eng gekoppelten Parallelrechnern* wird der *Manager hardwaremäßig in der Cacheverwaltung integriert.* Beispiele hierfür sind die Standford Dash, die KSR 1 und die MIT Alewife.

Bei der Hardwarelösung verwendet man *directory-based coherence protocols.* Dazu wird die Verwaltung jeder Cache Line um ein *dirty bit* ergänzt, das anzeigt, daß die Seite vom Prozessor verändert worden ist (dirty bit=1). Zusätzlich wird noch Information über den Aufenthaltsort der Seiten gespeichert.

Beim sog. *full-map scheme* (in Dash) wird jede Cache Line um je ein Bit pro Prozessor erweitert. Ist ein solches Bit gesetzt, so hat der zugehörige Prozessor eine Kopie dieser Seite,

Alternativ kann man auch für jede Seite x Pointer auf die Prozessoren speichern, die eine Kopie der Seite haben. Dazu benötigt man $x \cdot \lg n$ Bits. Wollen mehr als x Prozessoren eine Kopie derselben Seite, so wird ein Broadcastbit gesetzt, das dafür sorgt, daß bei einem Schreibzugriff ein Broadcast statt einer write-invalidation durchgeführt wird. Eine andere Lösung ist, beim $x+1$-ten Zugriff eine andere Kopie zu invalidieren.

Eine Einschränkung auf x Kopien erscheint deshalb sinnvoll, weil Vielfach-Sharing außer bei Total-Sharing nur selten vorkommt.

Ein Beispiel ist die Alewife mit $x = 4$. Bei einem Overflow wird ein Trap erzeugt und die Software entscheidet über eine Konfliktlösung (*Limitless Directory*). Die oben geschilderten Hardwarelösungen für dieses Problem scheinen die Konstrukteure der Alewife also nicht überzeugt zu haben.

7.2.4 Realisierungen

Stanford DASH Es handelt sich hierbei um eine 2-level Architektur mit einem 2-dimensionalen Gitter[1] von Clustern. Ein Cluster ist dabei ein bus-basierter Multiprozessor mit Snooping Cache, nämlich eine SGI Power Station 4D/340 mit 4 MIPS R3000 Prozessoren. Die Routing Chips stammen von Caltech.

Kann eine Speicheranfrage nicht im eigenen Cluster beantwortet werden, so wird sie an den sog. Home Cluster der gewünschten Seite weitergeleitet, das ist der Cluster, der die Informationen von diese Seite hat, wobei ein festes Mapping zwischen Seiteninformationen und Clustern vorgegeben ist. Wird nun im Home Cluster festgestellt, daß ein dritter Cluster (Remote Cluster) Schreibberechtigung auf diese Seite hat, so wird die Anfrage dorthin weitergeleitet.

Damit ergeben sich 4 verschiedene Antwortzeiten für eine Speicheranfrage[2]:

[1] Prototyp: 4×4

[2] eigentlich 5, da die Prozessoren gewöhnliche first-level Cache haben, die aber mit dem Cache Coherency Mechanismus nichts zu tun haben

- Seite im lokalen Cache
- Seite im lokalen Cluster
- Seite im Home Cluster
- Seite in anderem Cluster

KSR 1 Die KSR 1 (Kendall Square Research) ist ein kommerzielles, hierarchisches VSM System.

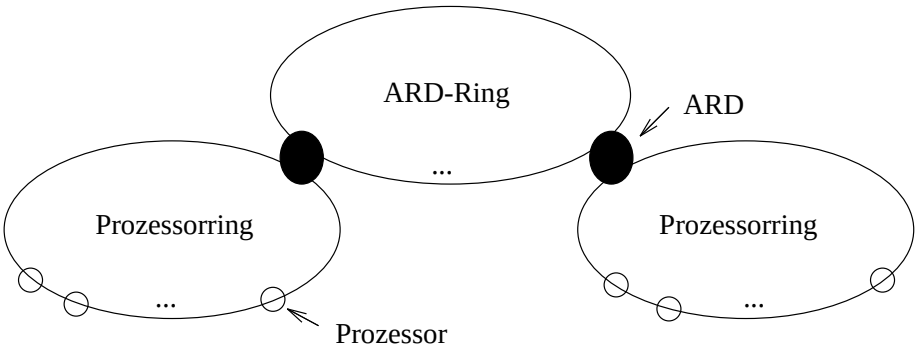


Abbildung 7.1: Systemaufbau der KSR 1

Das System (siehe Abbildung 1) besteht aus bis zu 34 *Prozessorringen* mit 32 Prozessoren und einem ARD sowie einem *ARD-Ring* mit bis zu 34 ARD's. ARD steht fÜur ALLCACHE Routing & Directory; hierin sind alle Informationen über alle Cache Lines im eigenen Prozessorring gespeichert.

Die verwendeten Prozessoren sind ein Eigendesign in 4 Chips mit einem first level Cache; es sind superskalare 64-bit Prozessoren. Die Bandbreite auf dem Prozessorring beträgt 1 GB/s. Maximal gibt es 1088 Prozessoren im System.

Kann eine Speicheranfrage nicht lokal beantwortet werden, so wird sie zum lokalen ARD weitergeleitet. Wird hier festgestellt, daß die Seite im lokalen Prozessorring ist, wird die Anfrage dort weitergeleitet. Ansonsten läuft sie über den ARD-Ring, bis der entsprechende Prozessorring gefunden ist, läuft dann zum entsprechenden Prozessor und von dort zum Startprozessor zurück.

Somit ergeben sich hier 3 verschiedene Antwortzeiten für eine Speicheranfrage:

- Seite lokal
- Seite im eigenen Prozessorring
- Seite in anderem Prozessorring (dabei sind 2 Prozessorringe und der ARD-Ring zu durchlaufen)

Vorlesung Rechnerarchitektur II (Parallelrechner)
Dr. Jörg Keller

SS 94, Termin Nr. 8: 16.6.94
Skript: Bernd Spengler

Message Passing Systeme

8.1 Distributed Shared Memory (Forts.)

Bezeichnungen:

ALLCACHE: Die KSR1 besitzt keinen eigentlichen Hauptspeicher, sondern 32 MB/Prozessor in DRAMs, die als Cache organisiert sind. Zusätzlich gibt es einen gewöhnlichen *first–level–cache* zwischen jedem Prozessor und dem ALLCACHE.

COMA: ähnlich dem ALLCACHE an der Universität Stockholm entwickelte *cache only memory architecture*.

Nachteile cache–basierter Systeme:

- Um Caches auszunutzen, müssen Anwendungen Lokalität aufweisen (z.B. Matrizenrechnungen, nicht jedoch graphbasierte Algorithmen).
- Bei einem *cache–miss* muß die Latenz versteckt werden, was zusätzliche (teure) Hardware erfordert, die selten ausgelastet ist.
- Die Zugriffszeit variiert sehr stark, sodaß *latency–hiding* schwierig ist.
- Kontextwechsel ohne spezielle Hardwareunterstützung ist langsam.

Message Passing Systeme

8.2 Allgemeines

MPS haben mehrere Prozessoren, jeder mit lokalem Speicher. Sie sind untereinander durch ein Netzwerk verbunden. Das Verbindungsnetzwerk (VN) kann statisch oder dynamisch sein, statische VN sind jedoch weiter verbreitet (Hypercube, 2D/3D-Gitter).

Die jeweiligen Knoten arbeiten mit Standardprozessoren. Programmiert werden sie in konventionellen Hochsprachen mit parallelen Erweiterungen (z.B. mit parallel loop). Die Kommunikation erfolgt über *send–receive*-Befehle, synchron (senden und warten bis acknowledge kommt) oder asynchron (send und sofort weiterarbeiten).

Für den Programmierer ergeben sich drei mögliche Sichtweisen:

1. *neighbour–to–neighbour:* Statisches VN, jeder Prozessor darf nur seinem direkten Nachbarn Pakete schicken.

2. Vollständiges VN, jeder Prozessor darf an jeden anderen Pakete schicken, System übernimmt *Routing* und *Scheduling*.

3. System von Prozessen mit virtuellen Kommunikationskanälen (KK). System plaziert Prozesse und übersetzt KK in Kommunikationsmuster der zweiten Sichtweise (vermutlich geringste Effizienz, da System hoch beansprucht, aber leicht zu programmieren)

8.3 Kommerzielle Message Passing Systeme

8.3.1 Überblick

MPS werden mit unterschiedlichen VN angeboten. Die bekanntesten sind:

1. VN = Hypercube
   - **Cosmic Cube** (Caltech) mit i8086-Prozessoren, bietet Sichtweise 1 (für Physiker) und Sichtweise 3 (für Informatiker)
   - **iPSC, iPSC/2, iPSC/860** (Intel) mit Verkabelungsproblemen, Sichtweise 2
   - **nCUBE, nCUBE2** (nCUBE)
2. VN = Gitter
   - **PARAGON, Touchstone Delta** (Intel)
3. VN = Baum (genauer: Fat-Tree, von C. Leiserson, Bandbreite der Links wächst von Blättern zur Wurzel)
   - **CM5** (Thinking Machines Corp. und MIT), mit Sparc-Prozessoren (MIMD)
4. VN = Omega–Netzwerk (topologisch äquivalent zum Butterflynetzwerk)
   - **SP1** (IBM)

8.3.2 Probleme bei der Modellierung der Performance

Häufig wird die Performance eines MPS durch die Zeit ausgedrückt, die zum versenden einer Nachricht erforderlich ist:

$$t_S = \text{Setup} + f_1(\text{Entfernung}) + f_2(\text{Größe des Paketes})$$

f_1, f_2 sind linear; Setup enthält viele Systemcalls usw. und dominiert bei gegenwärtigen Systemen (somit ist Sicht 2 gerechtfertigt). Die Performance hängt also nicht nur von der verwendeten Hardware, sondern auch vom Betriebssystem ab.

8.4 Das LogP-Modell

Das LogP-Modell ist ein maschinenunabhängiges Performance-Modell von Culler (Berkeley), Karp (ICSI) et. al., 1993 entwickelt. Es abstrahiert von der zugrunde liegenden Hardware. Folgende Parameter werden modelliert:

P Anzahl der Prozessoren, mit lokalem Speicher, verbunden über vollständiges VN

L Latenz; max. Routing–Zeit

o overhead; Zeit, die Prozessor zum send/receive mit Instruktionen selbst aufwendet.

g gap; minimale Zeit zwischen zwei Kommunikationsanweisungen $\Rightarrow\ g \geq 2o$.

Weiter Eigenschaften des Modells:

- Die Kommunikationsbandbreite eines Prozesors ist durch $1/g$ festgelegt, die Kapazität beträgt dabei max. L/g, d.h. L/g Nachrichten dürfen von oder zu einem Prozessor unterwegs sein.
- L ist obere Schranke, Nachrichten können auch früher ankommen. Es gibt keine FIFO-Ordnung, Nachrichten können sich überholen.
- Die Kommunikation erfolgt wortweise (Kritik: heute werden unterschiedliche Paketlängen hardwaremäßig unterstützt).

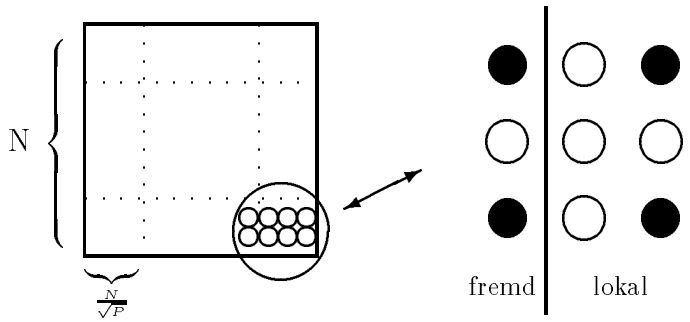
Anwendungsbeispiel: 5–Punktestern zur Lösung 2–dimensionaler partieller Differentialgleichungen:

$$M = \{(x,y) | 0 \leq x,y \leq N-1, x,y \in \mathbf{N}_0\} \qquad |M| = N^2$$

Berechne iterativ u(x,y):

$$u_{t+1}(x,y) = f(u_t(x,y), u_t(x+1,y), u_t(x-1,y), u_t(x,y+1), u_t(x,y-1))$$

Teile $N \times N$–Gitter in quadratische Blöcke mit Seitenlänge $\frac{N}{\sqrt{P}}$. #Blöcke = P.



Prozessoren kommunizieren in Form eines $\sqrt{P} \times \sqrt{P}$–Gitters. Vor jeder Iteration ist Kommunikation notwendig.

$$t_{Schritt} = \underbrace{\left(\frac{N}{\sqrt{P}}\right)^2}_{\text{Anzahl der Punkte im Block}} \cdot t_{5Punkt} + \underbrace{\frac{4N}{\sqrt{P}}}_{\text{Größe des Randes}} \cdot (L + o + g)$$

Bei Algorithmen mit *compute–communicate*–Runden kann der Kommunikationsdelay teilweise versteckt werden. Trick: Überlappen von Kommunikation und Berechnung sowie Prefetch von nichtlokalen Daten.

Überlappung mittels Beobachtung:

1. im inneren Teil des Blocks kann ohne Kommunikation berechnet werden. Überlappung von innerer Berechnung und Kommunikation.
2. Kommunikation wird in Pipeline betrieben.

$$\Rightarrow t_{comm} = o + L + (o+g) \cdot \left(\frac{4N}{\sqrt{P}} - 1\right)$$

Die Kapazität ist eingehalten, wenn alle Prozessoren erst nach rechts und dann nach oben etc. senden. $(o+g)$ kann zu g reduziert werden, wenn $g > 2o$ ist. Während L und jedem $(g-o)$ kann der Prozessor arbeiten. Wenn also gilt:

$$\left(\frac{N}{\sqrt{P}}\right)^2 \cdot t_{5Punkt} \geq L + (g-o) \cdot \left(\frac{4N}{\sqrt{P}} - 1\right)$$

ist $t_{Schritt}$ für die gegebenen Parameter $LogP$ optimal, da der Prozessor nie idle ist.

Lösung und Beispiel in Übung 9.1.

Vorlesung Rechnerarchitektur II (Parallelrechner)
Dr. Jörg Keller

SS 94, Termin Nr. 9: 22.6.94
Skript: Andreas Döring

LogP Modell

Anwendung des LogP-Modelles

Modellierung eines *butterfly*-Netzwerkes [Sahay]

Bei gewissen Anwendungen tauchen Berechnungsabhängigkeiten in Form von *butterfly*-Netzwerken auf. Beispiele dafür sind *fast fourier transformation* (FFT), die z.B. in der Bildverarbeitung Verwendung findet, oder *bitonic sort*. Es soll nun untersucht werden, wie gut sich solche Probleme durch das LogP-Modell bearbeiten lassen.
Man betrachte ein n-dimensionales *butterfly*-Netzwerk. Wir numerieren die Stufen des Netzwerkes umgekehrt durch, d.h. die „Sprünge“ im *butterfly*-Netz werden mit steigender Nummer der Stufe kleiner. Der zu verarbeitende *input* wird in das *level* 0 eingespeist, der berechnete *output* aus dem *level* n ausgegeben.
Möchte man eine möglichst hohe Effizienz erreichen, so ist folgendes zu beachten:

- Es sollte möglichst wenig Kommunikation zwischen den Prozessoren stattfinden,
- Damit es zu keiner Überschreitung der maximalen Kommunikationskapazität kommt, sollte die Kommunikation zwischen den Prozessoren möglichst „gleichmäßig“ stattfinden. Es sollte z.B. darauf geachtet werden, daß zu einem Zeitpunkt niemals von mehreren Prozessoren Daten zum gleichen Prozessor gesendet werden.
- Nach Möglichkeit sind Kommunikation und Berechnung miteinander zu überlappen, d.h. die Wartezeit während der Kommunikation sollte für Berechungen genutzt werden.

In einem ersten Ansatz verzichten wir auf den letzten der genannten Punkte, beschreiben also zunächst einen Algorithmus ohne Überlappung. Es sei $2^n \geq P^2$, $m := 2^n/P$, $l := m/P = 2^n/P^2$.

Welcher Knoten auf welchen Prozessor?

Unterteilung in 2 Teile:

- Teil 1: Die *levels* $c \in \{0, \cdots, \log m\}$: Bette den x-ten Knoten ($x \in \{0, \cdots, 2^n - 1\}$) von Level c auf Prozessor $x \bmod P$ ein (zyklische Einbettung).
- Teil 2: Die *levels* $c \in \{\log m + 1, \cdots, n\}$: Bette den x-ten Knoten ($x \in \{0, \cdots, 2^n - 1\}$) von Level c auf Prozessor $\lfloor x/m \rfloor$ ein (blockweise Einbettung mit Blockgröße m).

Aus (Übung 8.2) und (Übung 8.3) folgt:

- Teil 1 zerfällt in $\log m$-dimensionale *butterfly*-Netzwerke, wobei auf jeden Prozessor genau eines dieser *butterfly*-Netzwerke abgebildet wird,
- Teil 2 zerfällt in $n - \log m - 1 = \log P - 1$-dimensionale *butterfly*-Netzwerke, wobei auf jeden Prozessor genau $2 \cdot l$ dieser *butterfly*-Netzwerke abgebildet werden. Wie interpretieren diese Netzwerke als $\log P$-dimensionale Netzwerke, bei denen die Stufe 0 fehlt.

Aus (Übung 10.1) folgt: Die P *inputs* für die *butterfly*-Netzwerke in Teil 2 kommen alle von verschiedenen Prozessoren.

Wie berechnet man, wie kommuniziert man?

Es wird in folgender Reihenfolge simuliert:

- Berechnung:
  Die Prozessoren simulieren die Knoten von Teil 1. Für diese Simulation ist keinerlei Kommunikation notwendig, da die Teilgraphen jeweils vollständig auf einen Prozessor abgebildet wurden.
- Kommunikation: (erst nach Abschluß der Berechnung von Teil 1)
  Es wird in $P-1$ Runden kommuniziert: In Runde i sendet Prozessor p an Prozessor $(p+i) \bmod P$ die ihn betreffenden Daten. Diese Kommunikationsordnung ist regelmäßig, da jeder Prozessor in jeder Runde gleichviele Daten empfängt und sendet. Damit sendet jeder Prozessor insgesamt $m-l$ Pakete, die letzten l gehen an ihn selbst.
- Berechnung: (erst nach Abschluß der Kommunikation)
  Die Prozessoren simulieren die Knoten von Teil 2.

Es ergibt sich:

$$\begin{aligned}
\text{Zeit} &= \underbrace{m \cdot (\underbrace{\log m}_{n-\log P} + 1)}_{\text{Teil 1}} + \underbrace{\underbrace{l \cdot P}_{m} \cdot \log P}_{\text{Teil 2}} + \underbrace{L + g \cdot (m - l - 1)}_{\text{Kommunikation}} \\
&= \underbrace{m}_{2^n/P} \cdot ((n - \log P + 1) + \log P) + L + g \cdot \left(\frac{2^n}{P} - \frac{2^n}{P^2} - 1 \right) \\
&= \underbrace{\frac{2^n \cdot (n+1)}{P}}_{\text{Rechnen}} + \underbrace{L + g \cdot \left(\frac{2^n}{P} - \frac{2^n}{P^2} - 1 \right)}_{\text{Kommunikation}}
\end{aligned}$$

Nun sollen Kommunikation und Berechnung miteinander überlappt werden. Es wird das selbe *mapping* der Knoten des *butterfly*-Netzwerkes auf die Prozessoren wie oben verwendet, jedoch mit einer neuen „output-driven"–Berechnungsordnung: „Versende *outputs*, sobald erzeugt!"

Für jeden Prozessor i ($i \in \{0, \cdots, p-1\}$) sei $\pi^{(i)}$ eine Permutation über $\{0, \cdots, m-1\}$, der *outputs* eines Prozessors. $\pi^{(i)}$ gibt vor, in welcher Reihenfolge Prozessor i seine Ausgaben berechnet. Um möglichst früh kommunizieren zu können, berechnet jeder Prozessor stets gerade so viele Knoten, wie für die nächste Ausgabe nötig.

Beispiel

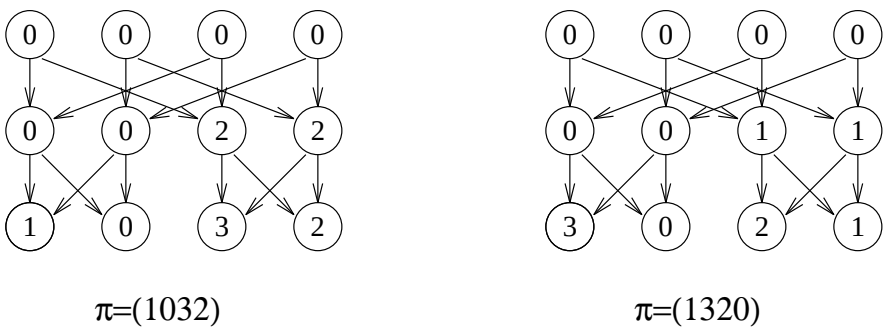


Abbildung 1: Beispiele für Reihenfolgen, in denen Knoten simuliert werden müssen.

Die linke Permutation führt zu früherer Kommunikation und ist daher der rechten vorzuziehen. Optimal sind die binären Permutationen, die im folgenden definiert werden.

Definition (binäre Permutation)

Sei $S_i := \{0, \cdots, 2^i - 1\}$, π Permutation auf S_i, dann heißt π „binär“, genau dann wenn $i = 0$, oder $i > 0$ und

1.
$$\left\lfloor \frac{\pi(0)}{2^{i-1}} \right\rfloor = \left\lfloor \frac{\pi(j)}{2^{i-1}} \right\rfloor \qquad \text{für } j = 1, \cdots, 2^{i-1} - 1$$

2. π_L und π_R sind binäre Permutationen auf S_i mit
$$\begin{aligned} \pi_L(j) &= \pi(j) \bmod 2^{i-1} \\ \pi_R(j) &= \pi(j + 2^{i-1}) \bmod 2^{i-1} \end{aligned}$$
für $j = 0, \cdots, 2^{i-1} - 1$.

Beispiele

$\pi = \mathrm{id}$ ist (trivial) binär.

$\pi = (45673201)$ ist binär, weil $\pi_L = (0123)$ und $\pi_R = (3201)$ binär sind;
$\pi_L = (0123)$ ist binär, weil $\pi_{LL} = (01)$ und $\pi_{LR} = (01)$ binär sind;
$\pi_R = (3201)$ ist binär, weil $\pi_{RL} = (10)$ und $\pi_{RR} = (01)$ binär sind;
$\pi_{LL} = \pi_{LR} = \pi_{RL} = (01)$ sind binär, weil $\pi_{LLL} = \pi_{LLR} = \pi_{LRL} = \pi_{LRR} = \pi_{RLL} = \pi_{RLR} = (0)$ binär sind;
$\pi_{RR} = (10)$ ist binär, weil $\pi_{RRL} = \pi_{RRR} = (0)$ binär sind.

$\pi = (45673021)$ ist nicht binär, da π_R nicht binär ist.

Vorlesung Rechnerarchitektur II (Parallelrechner)
Dr. Jörg Keller

SS 94, Termin Nr. 10: 30.06.94
Skript: Jochen Röhrig

LogP Modell II, SIMD Architekturen I

10.1 Algorithmus von Sahay (2. Teil)

Der betrachtete Algorithmus überlagert Berechnung und Kommunikation. Die Verteilung der Knoten des Butterflynetzwerks auf die P Prozessoren erfolgt wie beim einfachen Algorithmus, es ändert sich aber die Berechnungsreihenfolge des ersten Teils der Berechnung:

- Prozessor p_i berechnet seine Ouptuts gemäß einer Permutation $\pi^{(i)}$ in der Reihenfolge $\pi^{(i)}(0)$, $\pi^{(i)}(1), \ldots, \pi^{(i)}(m-1)$.
- p_i versendet seine Outputs so früh wie möglich.
- Es wird immer nur die minimale Anzahl von Knoten ausgewertet, die zur Berechnung eines Outputs notwendig sind („output driven schedule“).

Folgende Forderungen werden an $\pi^{(i)}$ gestellt:

- Die Outputs sollen optimal früh berechnet werden.
- Die Kommunikation soll regelmäßig erfolgen, d.h. $\pi^{(i)} \neq \pi^{(j)}$ für $i \neq j$.

Man betrachte nun einen balancierten binären Baum (vgl. Abb. 10.1) mit m Blättern und Permutationen π der Blätter für die gilt:

$$\begin{array}{rrcl} \text{entweder} & \max\{\pi(i), \ldots, \pi(j)\} & < & \min\{\pi(j+1), \ldots, \pi(k)\} \\ \text{oder} & \min\{\pi(i), \ldots, \pi(j)\} & > & \max\{\pi(j+1), \ldots, \pi(k)\} \end{array}$$

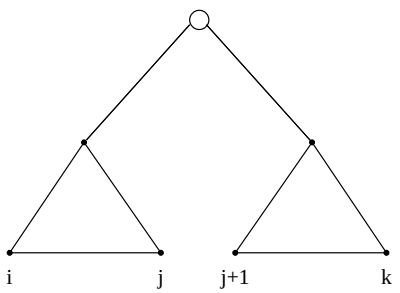


Abbildung 10.1: Balancierter binärer Baum

Man macht folgende Beobachtung:

- Diese Permutationen sind gerade die binären Permutationen.

- Die zeitliche Erzeugung der Outputs ist von der konkreten Permutation unabhängig.
- Die Erzeugungszeiten der Outputs sind optimal.

Beispiel:

$m = 8, \pi^{(1)} = (23016745), \pi^{(2)} = (45673201)$ (vgl. Abb. 10.2)

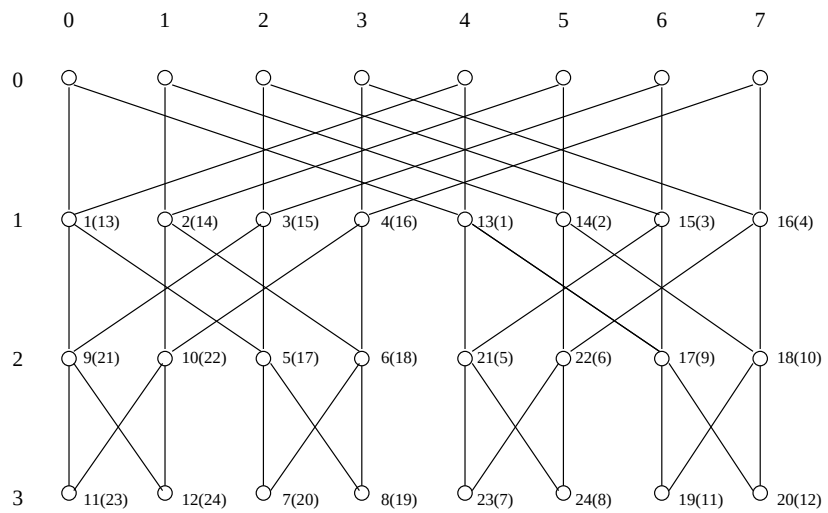


Abbildung 10.2: Mögliche optimale Berechnung der Outputs für $\pi^{(1)}$ und $\pi^{(2)}$

Die Berechnungszeitpunkte der Knoten sind neben den Knoten angegeben (die Zahlen in Klammern entsprechen den Zeiten fuer $\pi^{(2)}$). Man sieht an diesem Beispiel, daß beim Berechnen der 8 Outputs unabhängig von der verwendeten Permutation π und unabhängig vom Berechnungsschema für die Berechnungszeitpunkte t_B der Outputs gilt:

Output	t_B
$\pi(0)$	7
$\pi(1)$	8
$\pi(2)$	11
$\pi(3)$	12
$\pi(4)$	19
$\pi(5)$	20
$\pi(6)$	23
$\pi(7)$	24

Um die P verschiedenen Permutationen auszuwählen geht man nun wie folgt vor:

Seien $q, r \geq 0, w = q + r$. Wie in Übung 9.3 gezeigt wird ist jede binäre Permutation π über S_w bestimmt durch eine binäre Permutation σ auf S_q und durch 2^q binäre Permutationen $\varphi^{(x)}, 0 \leq x < 2^q$, auf S_r.
Wir wählen nun $w = \log m, q = \log P$ und $r = \log l$. Für einen Prozessor p_i sei die binäre Permutation $\pi^{(i)}$ bestimmt durch die binäre Permutation $\sigma^{(i)}$ auf $S_{\log P}$, wobei

$$\sigma^{(i)}(j) = \overline{i} \oplus j, 0 \leq i, j < P,$$

und die binären Permuationen $\varphi^{(x)}$ auf $S_{\log l}$, wobei

$$\varphi^{(x)} \equiv id_{S_{\log l}}, 0 \leq x < P$$

Durch die $\pi^{(i)}$ werden die Outputs eines Prozessors p_i in P Blöcke der Größe l gruppiert (wobei ein Block den Nachrichten entspricht, die an einen bestimmten Prozessor p_j gesendet werden): durch $\sigma^{(i)}$ wird die Reihenfolge festgelegt, in der p_i die Blöcke an die entsprechenden Prozessoren p_j sendet, und $\varphi^{(x)}$ gibt die Reihenfolge

innerhalb eines Blocks an.
Da $\sigma^{(i)} \neq \sigma^{(k)}, i \neq k$, entspricht jeder Kommunikationsschritt einer Permutation von $0, \ldots, P-1$, d.h. in jedem der Kommunikationsschritte wird an jeden Prozessor genau eine Nachricht gesendet (*Regelmäßigkeit* der Kommunikation).
Man beobachtet ausserdem, daß gemäß Definition von $\sigma^{(i)}$ Prozessor p_i erst im letzten Kommunikationsschritt an sich selber „sendet".

10.1.1 Laufzeit des komplizierten Algorithmus

Da die zeitliche Erzeugung der Outputs von i unabhängig ist, können wir o.B.d.A. $i = P-1$ wählen ($\Rightarrow \sigma^{(i)} = Id$).
Teil 1 braucht Zeit $m \log m$ zur Berechnung. Zur Zeit T werde die letzte Nachricht verschickt. Teil 2 braucht Zeit $m \log P$. Die Gesamtzeit T_g beträgt also:

$$T_g = \max\{T + L, m \log m\} + \log P$$

Berechnung von T:

Für Output $a, 0 \leq a < m$, von Prozessor $P-1$ sei t_a die Zeit, zu der a berechnet ist und s_a die Zeit, zu der a versendet wird. Dann ist $s_0 = t_0$ und $s_a = \max\{t_a, s_{a-1} + g\}$ für $a > 0$. Sei a^* maximal, sodaß $s_{a^*} = t_{a^*}$. Dann ist

$$T = t_{a^*} + g * (m - l - a^* - 1)$$

Lemma 10.1 *Es ist $t_0 = m-1$, und für $a = q * 2^{r-1}$ mit q ungerade gilt: $t_a = t_{a-2^{r-1}} + r * 2^{r-1}$*

Beweis:

- Sei v die Anzahl der Vorgänger des Knotens der Output 0 berechnet im Berechungsbaum für Output 0. Dann gilt:
$$v = \sum_{i=1}^{\log m - 1} 2^i = 2^{\log m} - 1 - 1 = m - 2$$
t_0 ist gleich der Zeit zur Berechnung der Vorgänger von Output 0 plus die Zeit zur Ausgabe von Output 0, d.h.
$$t_0 = v + 1 = m - 1$$
- Für $a = q * 2^{r-1}$ sind zur Zeit $t_{a-2^{r-1}}$ die Vorgänger in den Levels $1, \ldots, \log m - r$ berechnet. In den r Levels $\log m - r + 1, \ldots, \log m$ müssen je 2^{r-1} Kanten berechnet werden. Also ist
$$t_a = t_{a-2^{r-1}} + r * 2^{r-1}$$

Lemma 10.2 *Falls $a \equiv 0 \mod 2^{q-1}$, dann ist $s_a = t_a$, sonst ist $s_a > t_a$.*

Beweis: Übung 11.1

Theorem 10.1 *Seien n, P Zweierpotenzen mit $n \geq P^2$*

1. *Falls $g \geq \log m$, dann ist $T = (m-1) + (m-l-1) * g$*
2. *Falls $\log \underbrace{\frac{n}{P^2}}_{=l} + 1 < g < \log m$, dann ist $T = m \log m + 2^g - g - 1 - l * g$*
3. *Falls $g \leq \log l + 1$, dann ist $T = m \log m + 2^g - 1 - l * (\log l + 1) \leq m * \log m$*

Beweis: Übung 11.2

10.1.2 Speedup des komplizierten Algorithmus

Vgl. Übung 11.3

10.2 SIMD-Maschinen

SIMD = **S**ingle **I**nstruction **M**ultiple **D**ata

SIMD-Maschinen arbeiten auf einem einzigen Programm, haben nur einen Programmcounter und einen Stack. Man unterscheidet zwei Arten von SIMD-Maschinen:

- *Vektorprozessoren* (z.B. Cray, CDC Cyber 205, NEC SX)
- *Arrayprozessoren* (z.B. Connection Machine (CM1 und CM2))

Beide Rechnerarten arbeiten nach dem Prinzip der Ausführung einer einzigen Operation auf vielen Daten (z.B. Vektoraddition: $A[1\ldots100] = B[1\ldots100] + C[1\ldots100]$). Sie unterscheiden sich aber in ihrem Aufbau:

- *Arrayprozessoren* haben viele CPUs.
- *Vektorprozessoren* haben nur eine CPU, die gepipelined ist und über mehrere Arithmetikeinheiten verfügt. Dies hat den Vorteil, daß pro Berechnungsschritt nur einmal ein Fetch und ein Decode ausgeführt werden muß.
  Die Vektoralu ist i.a. gepipelined. (Sei t die Tiefe der ALU. Bei skalarer Berechnung benötigen i Operationen Zeit $i * t$. Mit x Pipelinestufen wird für i Operationen nur Zeit $t + (i-1) * (\frac{t}{x} + \delta)$ benötigt. Vgl. hierzu auch Vorlesung *Rechnerarchitektur 1*).

  Aufbau eines Vektorrechners: (vgl. Abb. 10.3)

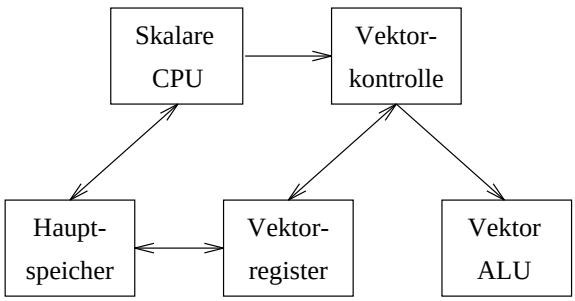


Abbildung 10.3: Aufbau eines Vektorrechners

  - Die skalare CPU erledigt Sprünge etc.
  - Die Vektoralu benutzt schnelle Vektorregister (Problem: Registerallokierung durch den Compiler)
  - **„Chaining“**: Ein Vektorregister ist Output einer Operation und dient gleichzeitig als Input einer anderen Operation ⇒ Die zweite Operation darf beginnen, sobald das erste Ergebnis der ersten Operation da ist.

  Beispiele für Vektorrechner:

  - Cray 1 (12.5 ns Zykluszeit) (vgl. Abb. 10.4)
  - Cray YMP (4 ns Zykluszeit)

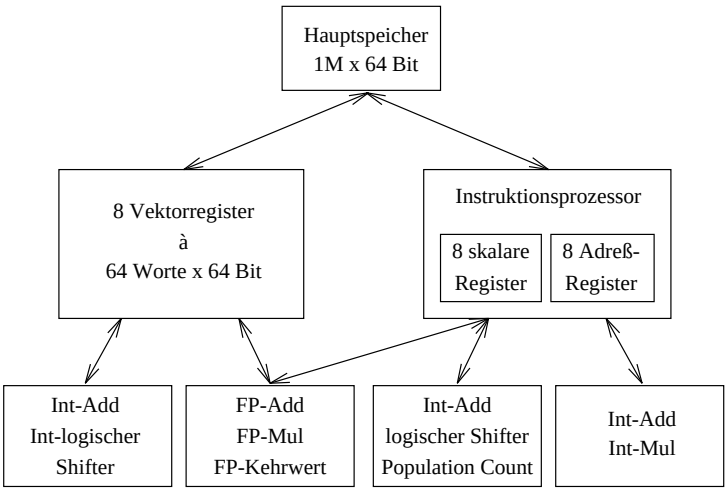


Abbildung 10.4: Aufbau der Cray 1

Vorlesung Rechnerarchitektur II (Parallelrechner)
Dr. Jörg Keller

SS 94, Termin Nr. 11: 07.07.94
Skript: Andreas Paul

SIMD Architekturen II

11.1 Parallele SIMD–Maschinen

Hierbei werden statt einer aufwendigen CPU viele, in der Regel einfach gehaltene Prozessoren verwendet. Vertreter dieser Kategorie sind die Connection Machine 1 ("CM1"), die CM2, die MasPar MP1 und MP2, sowie die "Systolic Arrays".
Im Folgenden werden die CM1, die CM2 und Vertreter der Systolic Arrays betrachtet.

11.1.1 Die CM1

Sie wurde 1986 von D. Hillis am MIT entwickelt. Wie die SB–PRAM handelt es sich um einen Backend–Rechner, welcher mittels eines Host–Rechners (Vax oder Symbolics Lisp) bedient wird.

Aufbau

- 65536 Prozessoren, wobei jeder Prozessor ein 1–Bit–Prozessorelement (**"PE"**) ist ⇒ schnell auf symbolischen Rechnungen, aber langsam auf Arithmetik ⇒ Einsatzgebiet war hauptsächlich in der künstlichen Intelligenz.
- 16PE auf einem Chip
- 32 Chips auf einem Board
- 16 Boards bilden einen **Subcube**
- 2 Subcubes bilden einen **Quadranten**
- Quadranten wahlweise voneinander unabhängig (⇒ bis zu 4 Programme gleichzeitig) oder — evtl. teilweise — zusammengeschaltet konfigurierbar
- Die 2^{12} Prozessorchips sind sowohl zu einem zwölfdimensionalen Hypercube verbunden, als auch zu einem 256x256–Gitter. Die Gitterverbindungen sind aber im Vergleich zu den Cube–Kanten erheblich schneller.
- 32MByte Speicher

Die Komponenten des Subcubes sehen schematisch folgendermaßen aus:

Programmierung

Die Programmierung des Rechners erfolgt in mehreren Stufen:

1. Der User schreibt sein Programm (recht) bequem in C, Fortran oder Lisp mit zusätzlichen parallelen Konstrukten.

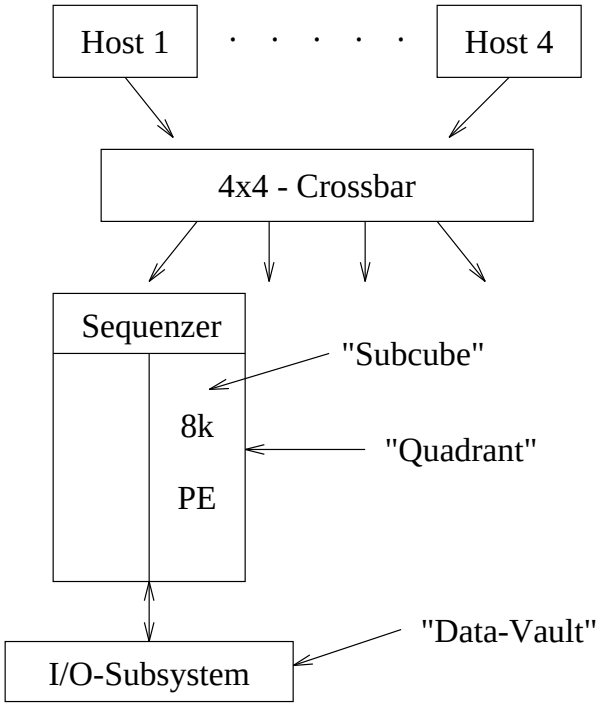


Abbildung 11.1: Schematischer Aufbau der CM1

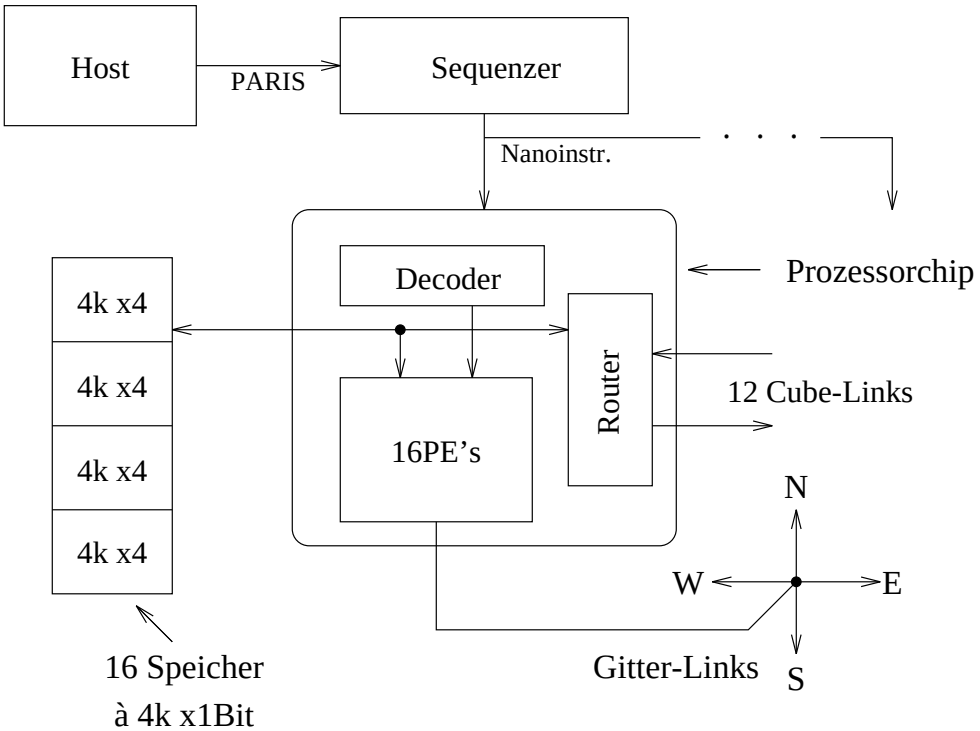


Abbildung 11.2: Aufbau eines Prozessorchips

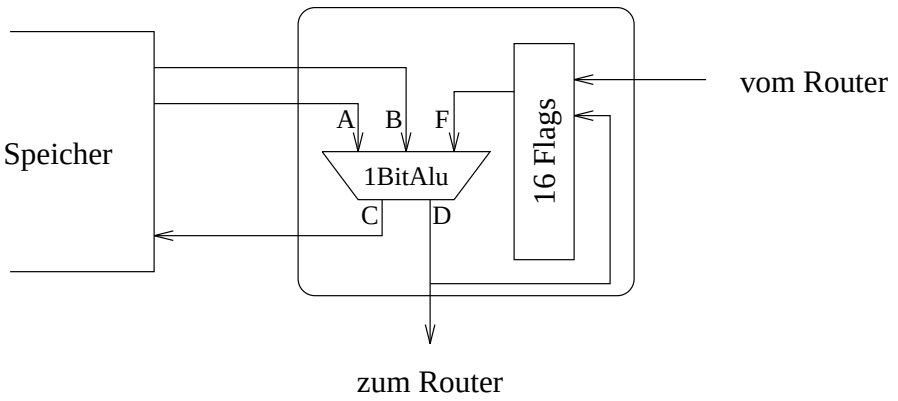


Abbildung 11.3: Aufbau eines PEs

2. Dies wird auf dem jeweiligen Host-Rechner in ein **PARIS–Programm** (→ **PAR**allel **I**nstruction **S**et) übersetzt. Hierbei handelt es sich um eine Folge von Makroinstruktionen wie MOVE, +, -, ·, ∨, ∧, TST (auf verschiedenen Datentypen), Vektor–und Matrixoperationen, globale Summe, globales Maximum, globales Minimum, globales AND, globales OR, Float–Operationen, Sortier–Operationen, I/O, Kommunikation zwischen den Prozessoren, Kommunikation mit dem Host usw.

3. Das PARIS–Programm wird an den Sequenzer geschickt, welcher es in eine Folge von sog. **Nanoinstruktionen** an alle PE's des Quadranten schickt.

Die Nanoinstruktionen haben eine feste Länge von 55 Bit:

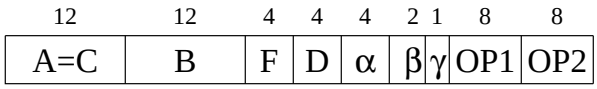


Abbildung 11.4: Format der Nanoinstruktionen

A/C und B bilden die Speicheradressen für den ersten Operanden (und gleichzeitig das Resultat) bzw. den zweiten Operanden einer ALU–Operation an; in α werden Bedingungen für das Abspeichern des Ergebnisses der Operation codiert, in β, dem "Condition–Bit" wird festgelegt, ob in α die Maskierung des Abspeicherns bei gesetzten oder gelöschten Bits erfolgen soll. γ legt die **NEWS–Direction** (→ **N**ord, **E**ast, **W**est, **S**outh) fest, also in welche Richtung z.B. das Carry bei einer Additon propagiert wird. Die OP–Bits geben die Funktionstabelle einer ALU–Operation an.

Beispiel für die Belegung der OP–Bits: 1–Bit–Addition Der Operand F entspreche dem Carry–In, C, also OP1 entspreche der Summe und D, also OP2, entspreche dem Carry-Out.

A	B	F	C	D
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Die schrittweise Konvertierung sieht beispielsweise folgendermaßen aus:
Die PARIS–Instruktion "ADD 100,200,64" ("Addiere die beiden 64–Bit Integers, welche in den Zellen 100–163 bzw. 200–263 gespeichert sind") wird von Sequenzer umgewandelt in einige (≈ 3) Init–Instruktionen, z.B. zum

Zurücksetzen der Flags, "Addiere 100 ($\rightsquigarrow$ A), 200 ($\rightsquigarrow$ B), F=D, Abspeichern erlaubt,..", "Addiere 101, 201,..." usw. Aus einer PARIS–Instruktion werden hier also etwa 67 Nanoinstruktionen.
Dauert ein 1–Bit–ADD 750ns, so dauert die 64–Bit–Addition dann 50 μs. Da diese Operation aber parallel auf allen 64k PE's ausgeführt werden kann, ergibt dies dann die Zahl von umgerechnet $65536/50\mu\text{s} = 1.2 \cdot 10^9$ 64–Bit–Additionen pro Sekunde!

Eine andere Möglichkeit, die Addition durchzuführen, wäre die, zu pipelinen: Bei allen PE's seien die NEWS-Bits auf N gestellt. Ein PE addiere die Bits 0 in Adresse 100 und 200, das Carry–Out wird nach oben propagiert, das PE über dem betrachteten addiere mit Carry–In Bit 1 in Adresse 101 bzw. 201, das Carry–Out wird weiter nach oben propagiert usw.

11.1.2 Die CM2

Hier gibt es für je 32 PE's einen FP–Coprozessor von Weitek; der Einsatz verlagert sich mehr von symbolischer auf numerische Berechnung. Die Sichtweise ist nun die einer 2048–Prozessor–Maschine mit 32–Bit–Prozessoren und FP–Coprozessoren. Entsprechend wurde die Software dieser Sichtweise angepaßt.
Außerdem wurde der Speicher von 32 MByte auf 8 GByte erweitert.

11.1.3 Systolic Arrays

Hierbei handelt es sich weniger um *eine* bestimmte Maschine als mehr um eine Gruppe von Parallelrechnern, die für gewisse Aufgaben optimiert sind. "Systolic" steht als Analogie für das "Pumpen" von Daten durch das Array wie das Pumpen von Blut durch das Herz. Das Ziel bei der Entwicklung der Systolic Arrays war das Design von Algorithmen und Architekturen, so daß diese sehr gut auf zweidimensionale Gebilde aus VLSI–Chips abbildbar sind. Die Hauptanwendung dieser Rechner liegt in der Signalverarbeitung. Sie wurden von Kung und Leiserson entwickelt.

Zwei Beispiele werden im folgenden betrachtet:

FFT–Array

Für eine FFT (Fast Fourier Transformation) der Länge n werden $n \log n$ VLSI–Chips gleichen Aufbaus eingesetzt; sie sind als Butterfly–Netzwerk (Breite n, Tiefe $\log n$) miteinander verdrahtet. Der Delay für eine FFT ist also in der Größenordnung $\log n$. Es ist aber stets nur eine Stufe des Netzwerkes in Betrieb, so daß ein Pipelining von mehreren FFT's sehr gut möglich ist. x FFT–Operationen bräuchten somit die Zeit $\log n + x - 1$.

Der Aufbau der Knoten ist sehr einfach:

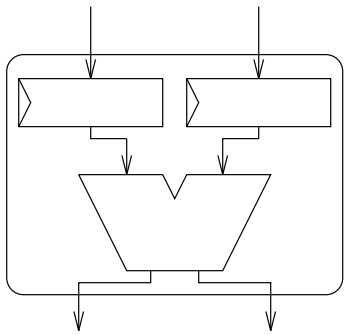


Abbildung 11.5: Butterfly–Knoten

Bandmatrixmultiplikation

Bandmatrizen sind Matrizen, welche nur entlang eines "Bandes" um die Hauptdiagonale der Matrix Einträge ungleich 0 besitzt. Bei der Multiplikation zweier Bandmatrizen entsteht wiederum eine Bandmatrix. Die Multiplikation läßt sich leicht mittels eines Netzwerkes von "Produktaddierern" (PA) realisieren. Ein solcher PA sei ein Knoten, durch den 3 Datenströme folgendermaßen fließen:

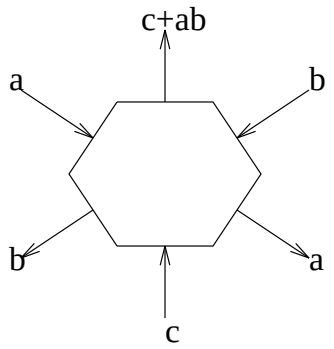


Abbildung 11.6: Datenfluß durch Netzwerkknoten

Als komplettes Netzwerk sähe das dann so aus:

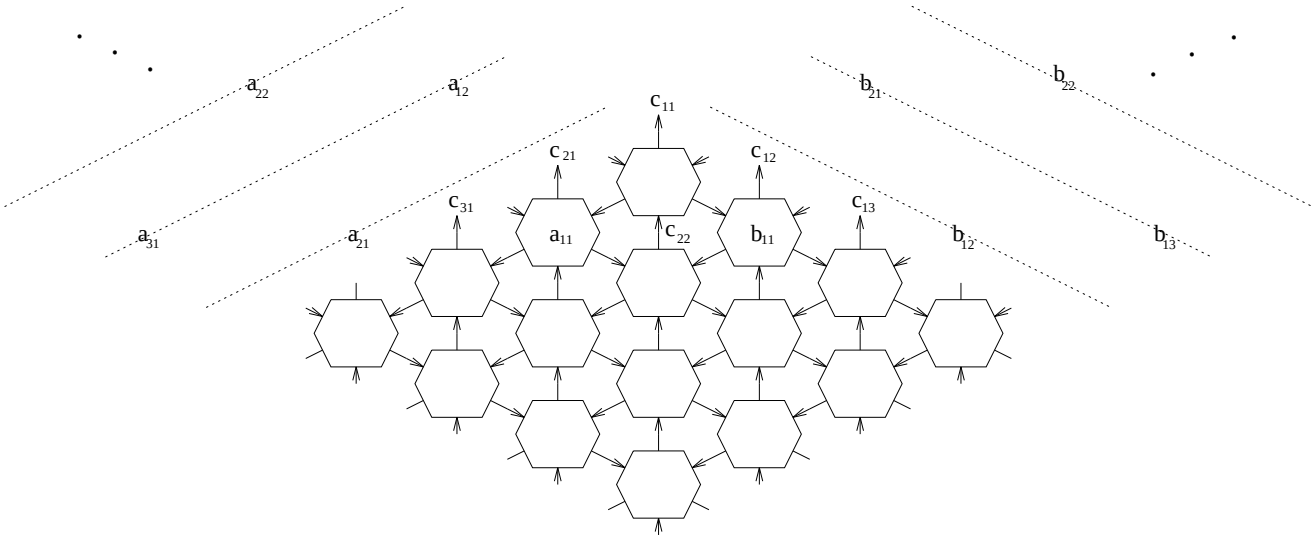


Abbildung 11.7: Netzwerk für Bandmatrixmultiplikation

11.2 Datenflußrechner

Definition 11.1

- **Datenflußrechner** *sind Parallelrechner, die in einer sequentiellen Programmiersprache programmiert werden und* **automatisch** *den inhärenten Parallelismus von Programmen auf Instruktionsebene nutzen.*
- *Ein* **Datenflußgraph** *ist ein gerichteter, azyklischer Graph, in dem Knoten Instruktionen und Kanten Datenabhängigkeiten darstellen.*

Beispiel:

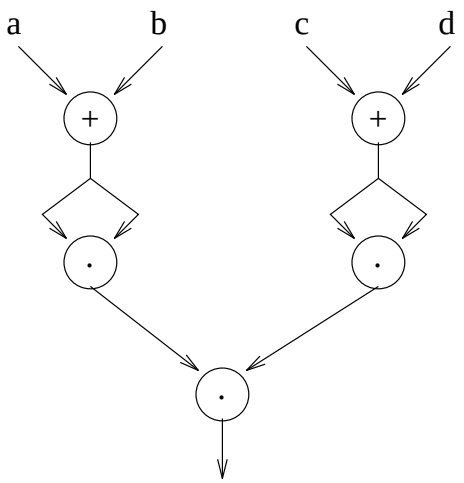


Abbildung 11.8: Datenflußgraph für $(a+b)^2 \cdot (c+d)^2$

Vorlesung Rechnerarchitektur II (Parallelrechner)
Dr. Jörg Keller

SS 94, Termin Nr. 12: 17.06.94
Skript: Stefan Burkhardt

Datenfluß– und VLIW Maschinen

12.1 Datenflußrechner

Datenflußprinzip:
Instruktion kann dann ausgeführt werden, wenn Tokens auf allen Eingabekanten vorhanden sind, bei Ausführung werden Tokens auf den Ausgangskanten erzeugt.

⇒ zeitunabhängiges Ausführungsmodell, Instruktionen können in beliebiger Reihenfolge bearbeitet werden, wenn Datenabhängigkeiten berücksichtigt werden.

⇒ Parallelausführung möglich, „impliziter Parallelismus“

Gegenteil: „expliziter Parallelismus“, d.h. Parallelverarbeitung wird durch Hochsprachenkonstrukte ausgedrückt.

Gegenteil zum Datenflußprinzip:
Kontrollflußprinzip:
echter Kontrollfluß (Befehlszähler) steuert die Ausführung von Befehlen.

Problem:
„scheinbare Abhängigkeiten“, z.B:

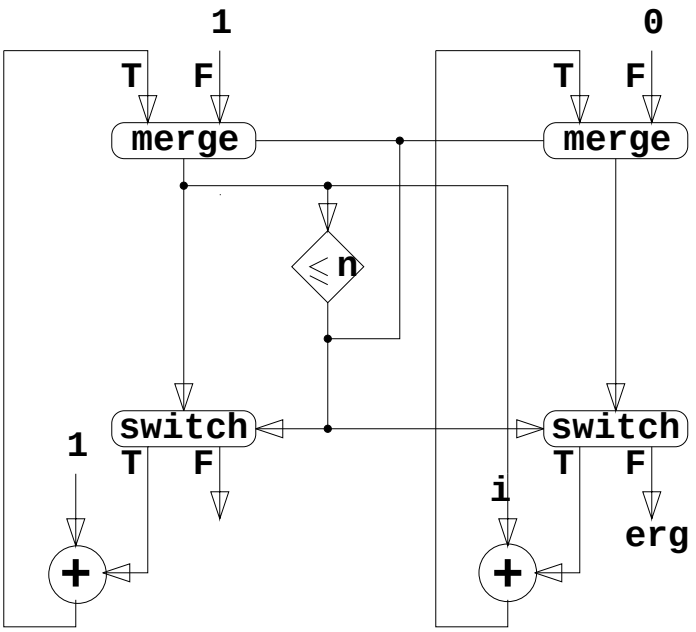
```
A:=B+C;
B:=D/2;
```

Abhilfe:
Single-Assignment-Languages, d.h. jede Variable darf nur einmal beschrieben werden. Solche Sprachen sind in der Regel funktional, z.B. ID.

Beispiel:
Programm:

```
erg:=0;
for i:=1 to n do
erg:=erg+i;
```

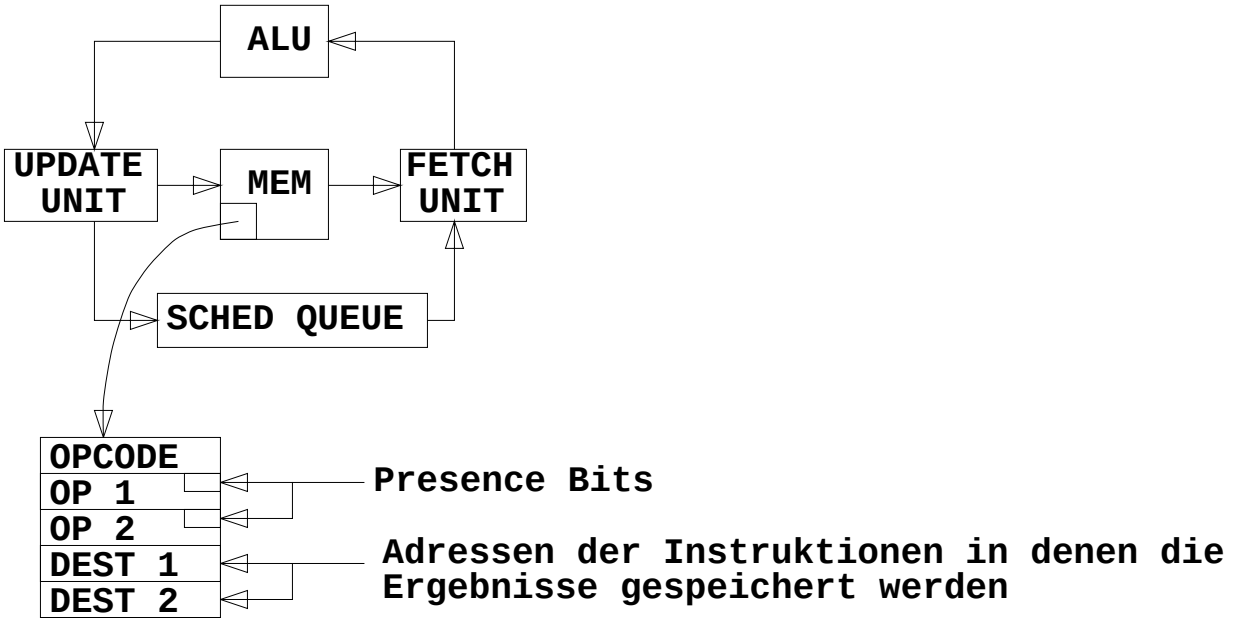
Datengflußgraph:



Problem:
Mehrere Token pro Kante sind möglich

statische Datenflußrechner: maximal ein Token pro Kante
dynamische Datenflußrechner: mehrere Tokens pro Kante erlaubt

Beispiel Statisch:
Cell Block Architecture [Dennis, MIT]



Parallel:
Netzwerk zwischen ALUs und Update Units

Problem mehrerer Tokens pro Kante:

- entweder in Software FIFO-Queues auf den Kanten implementieren
- oder im Datenflußgraph zu jeder Kante Signalkante in umgekehrter Richtung

Problem:

Dies verhindert oft die Parallelausführung, da hierdurch Schleifen sequentiell werden.

weiteres Problem:
Prozeduraufrufe

Beispiel Dynamisch:
TTDA (Tagged Token Dataflow Architecture) [Arvind, Papadopoulos, MIT]
Trennung von Programm- und Operandenspeicher :
Der Programmspeicher enthält die Templates ohne die Operandenstructs, der Operandenspeicher ist dynamisch allokierbar und enthält die Operandenstructs. Für mehrere Instanzen einer Operation existieren mehrere Operandenstructs. Da die Tokens auch die Instanz kennen müessen, werden Tags fuer die Tokens eingeführt. Tokens mit gleichen Tags werden im Operandenspeicher zusammengeführt.

Probleme:
1. Erzeugen von Tags
2. Zusammenführung

zu 1:
Softwareproblem, lösbar

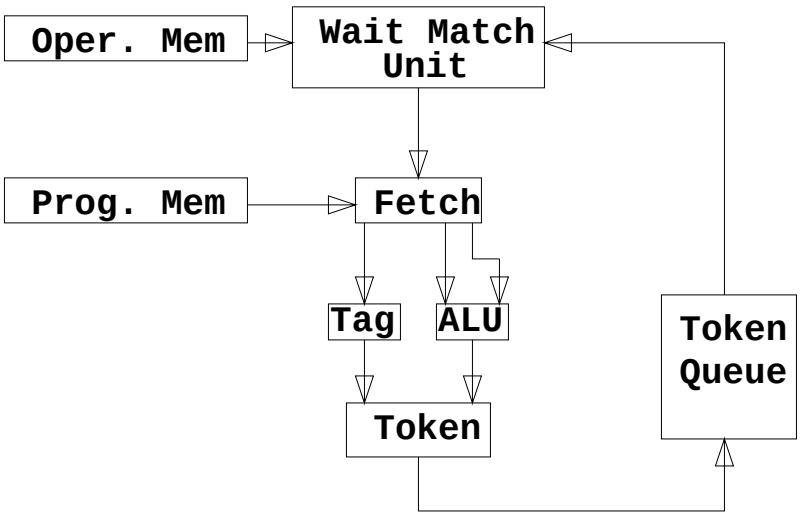
zu 2:
prinzipiell:
Assoziativspeicher

Praxis:

- Hashtabellen
- Adressen teilweise zur Compilezeit feststellen, ETS (Explicit Token Store)

Beispiel:
Monsoon [1990 am MIT gebaut, 16 Prozessoren], basiert auf TTDA

TTDA:



Monsoon: Wait match unit wird durch ETS ersetzt.

Feststellung: Reines Datenflußprinzip zu langsam wegen Overhead.
⇒ hybride Datenflußrechner:
Kombination aus Kontroll- und Datenfluß;
Kontrollfluß für threads (Programmstücke), Datenfluß für die Initiierung der threads.

12.2 VLIW-Rechner (Very Long Instruction Word)

Ziel: Impliziter Parallelismus + Kontrollfluß, d.h. Mischung aus Datenfluß und SIMD

Beispiel:
ELI-512 [Fisher, 1983, Yale]
Firma Multiflow, Rechner Trace, Firma existierte bis 1990, baute kommerziell einige hundert Rechner.

Daten:
512 bit Instruktionen,
16 funktionale Einheiten
⇒ Parallelismus muß zur Compilezeit erkannt und genutzt werden.
16 Cluster verbunden als Ring
Prozessor i ist mit Prozessor i+6 mod 16 und mit i-6 mod 16 verbunden.
8 M-Cluster bestehend aus Memory Modul, Integer ALU, Multiport Register Bank,
8 F-Cluster bestehend aus FP ALU, Multiport FP Register Bank,
jeweils abwechselnd auf dem Ring angeordnet.

pro Instruktion:

- 16 ALU Operationen (8 Integer, 8 FP)
- 8 Speicherzugriffe
- Tests
- Jump

Hauptproblem:
Compiler muß all dies berücksichtigen um überhaupt alle diese Sachen gleichzeitig zu ermöglichen.

Postfach 151150 — 66041 Saarbrücken Tel.: 0681–302–2576 Fax: 0681–302–4290 Email: jkeller@cs.uni-sb.de

14.04.94

1. Übungsblatt zur Vorlesung
Rechnerarchitektur II — Parallelrechner

1. Aufgabe
Gegeben sei ein sequentieller Algorithmus mit Laufzeit t_s. Nehmen Sie an, daß ein Teil des Algorithmus mit einem Anteil α der Laufzeit, wobei $0 \leq \alpha \leq 1$, nicht parallelisierbar ist. Für die restliche Laufzeit wird ein Speedup s_1 erreicht. Geben Sie eine Formel für den Speedup s des Gesamtproblems an. Geben Sie eine obere Schranke für s bei einem beliebig großen Parallelrechner an ($s_1 \to \infty$). Was folgt für s, wenn α eine Konstante ist?
Bemerkung: Die Antwort zur letzten Frage nennt man *Amdahl's Law*.

2. Aufgabe
Welcher Zusammenhang besteht zwischen n–dimensionalem Würfel und n–dimensionalem Butterfly? Tip: Ordnen Sie Kanten nach den Dimensionen.

3. Aufgabe
Ein Graph $G = (V, E)$ heißt *Kreuzprodukt* der Graphen $G_i = (V_i, E_i)$, $i = 1, 2, \ldots, k$, falls $V = \{(v_1, v_2, \ldots, v_k) | v_i \in V_i \text{ für } 1 \leq i \leq k\}$. und $E = \{\{(u_1, u_2, \ldots, u_k), (v_1, v_2, \ldots, v_k)\} |$ es gibt j mit $\{u_j, v_j\} \in E_j$ und $u_i = v_i$ für $i \neq j\}$. Wir bezeichnen das Kreuzprodukt mit $G = G_1 \otimes G_2 \otimes \cdots \otimes G_k$.
Zeigen Sie, daß ein $M_1 \times M_2 \times \cdots \times M_k$ Gitter das Kreuzprodukt von linearen Gittern der Größen $M_1, M_2, \ldots, M_k$ ist, und daß für $r = r_1 + r_2 + \cdots r_k$ der r–dimensionale Würfel H_r als Kreuzprodukt der r_i–dimensionalen Würfel H_{r_i} ist, $i = 1, 2, \ldots, k$.

Postfach 151150 — 66041 Saarbrücken Tel.: 0681–302–2576 Fax: 0681–302–4290 Email: jkeller@cs.uni-sb.de

21.04.94

2. Übungsblatt zur Vorlesung
Rechnerarchitektur II — Parallelrechner

1. Aufgabe
Wie läßt sich ein $M_1 \times M_2 \times \cdots \times M_k$ Gitter auf einen d–dimensionalen Würfel einbetten, wenn die M_i keine Zweierpotenzen sind? Wie groß ist die Dimension d des Würfels bei dieser Einbettung?

2. Aufgabe
Um in einem n–dimensionalen Butterfly Netzwerk ein Paket $\mathcal{P}$ von einem Knoten A in Level 0 zu einem Knoten B in Level n zu routen, benötigt man nur die Nummer des Zielknotens B, die in n Bit kodiert werden kann. Was muß ein Netzwerkknoten zusätzlich zur Routingfunktion tun, damit eine Antwort $\mathcal{P}'$ von B zu A geschickt werden kann, ohne daß in $\mathcal{P}$ weitere Bits benötigt werden?

3. Aufgabe (Einbettung von großen in kleine Gitter)
Zeigen Sie, daß man ein r–dimensionales Gitter der Seitenlänge N auf einem s–dimensionalen Gitter der Seitenlänge N, wobei $s < r$, mit Slowdown N^{s-r} simulieren kann. Zeigen Sie, daß man ein r–dimensionales Gitter der Seitenlänge N auf einem r–dimensionalem Gitter der Seitenlänge N', wobei $N' < N$, mit Slowdown $(\lceil N/N' \rceil)^r$ simulieren kann.

Postfach 151150 — 66041 Saarbrücken Tel.: 0681–302–2576 Fax: 0681–302–4290 Email: jkeller@cs.uni-sb.de

28.04.94

3. Übungsblatt zur Vorlesung

Rechnerarchitektur II — Parallelrechner

1. Aufgabe
Zeigen Sie, daß man von einem Knoten in Stufe $n/2$ eines n–dimensionalen Butterfly Netzwerkes aus je $\sqrt{2^n}$ Knoten in Stufe 0 und Stufe n erreichen kann. Konstruieren Sie daraus eine Permutation von Sendern (Knoten in Stufe 0) zu Empfängern (Knoten in Stufe n), deren Routing im Butterfly Netzwerk mindestens Zeit $\sqrt{2^n}$ benötigt. Benutzen Sie dazu die Eindeutigkeit von Wegen im Butterfly Netzwerk.

2. Aufgabe
Gegeben sei ein lineares Gitter der Länge N. Zu Beginn seien N Pakete beliebig auf die Knoten verteilt, jeder Knoten ist Ziel genau eines Paketes. Zeigen Sie:
Verwendet man den greedy Algorithmus mit farthest-first Protokoll, dann erreichen alle Pakete mit einem Ziel aus $M = \{N-i, \ldots, N-1\}$ einen der Knoten in M nach höchstens $N-1$ Schritten.

3. Aufgabe
Wie groß können die Queues werden, in denen beim greedy Algorithmus zum Routing von Permutationen auf 2–dim. Gittern Pakete zwischen Phase 1 und Phase 2 gespeichert werden,
(a) wenn die Phasen nicht überlappt werden?
(b) wenn die Phasen überlappt werden?

Postfach 151150 — 66041 Saarbrücken Tel.: 0681–302–2576 Fax: 0681–302–4290 Email: jkeller@cs.uni-sb.de

05.05.94

4. Übungsblatt zur Vorlesung
Rechnerarchitektur II — Parallelrechner

Übung am 10.05.94 und Vorlesung am 12.05.94 fallen aus.
Nächste Übung am 17.05.94, nächste Vorlesung am 19.05.94.

1. Aufgabe
Zeigen Sie, daß in der Fluent Machine zwei Pakete, die im gleichen Schritt zur gleichen Speicherzelle wollen, immer in Phase 1 oder Phase 2 kombiniert werden. Hinweis: Benutzen Sie die Sortierung nach gehashten Adressen.

2. Aufgabe
Um in einem Netzwerk-Knoten der Fluent Machine zu entscheiden, ob zwei Pakete kombiniert werden müssen, ist es mit Sicherheit ausreichend, die nicht gehashten Zieladressen zu vergleichen. Um Bandbreite zu sparen, möchte man aber nur die gehashten Zieladressen mitschicken, die man sowieso zum Routing braucht. Geben Sie eine notwendige und hinreichende Bedingung an, so daß ein Vergleich der gehashten Zieladressen ausreichend ist.

3. Aufgabe
Betrachten Sie einen Rechner vom Typ Sequent Symmetry. Gegeben sei eine Anwendung, bei der alle Prozessoren auf verschiedene Speicherzellen zugreifen. Konstruieren Sie eine Situation, in der trotzdem ein Check der Cache-Konsistenz notwendig ist.

Postfach 151150 — 66041 Saarbrücken Tel.: 0681–302–2576 Fax: 0681–302–4290 Email: jkeller@cs.uni-sb.de

19.05.94

5. Übungsblatt zur Vorlesung
Rechnerarchitektur II — Parallelrechner

1. Aufgabe
Zeigen Sie, daß die in der Vorlesung gezeigte Implementierung von Multipräfix auf der Fluent Machine korrekt ist, und daß die Routing Knoten nur um eine arithmetische Einheit zur Berechnung von $\circ$ und eine FIFO Queue zum Speichern von Zwischenergebnissen erweitert werden müssen.

2. Aufgabe
Zeigen Sie: das i-te Paket, das in einem Knoten der Phasen 5 bzw. 6 der Fluent Machine zum Verschicken selektiert wird, ist die Antwort auf das i-te Paket, das im korrespondierenden Knoten der Phasen 2 bzw. 1 zum Verschicken selektiert wurde.

3. Aufgabe
Die Fluent Machine ist unter anderem deshalb berühmt, weil die Puffer in den Knoten nur eine konstante Anzahl von Paketen fassen müssen, und deshalb der gleiche Entwurf für verschiedene Maschinengrößen verwendet werden kann. Die FIFO Queues für die Richtungsbits haben aber die Länge $O(n)$, da das Routing im Normalfalle so lange dauert. Wieso kann man behaupten, daß das kein Widerspruch ist?

Postfach 151150 — 66041 Saarbrücken Tel.: 0681–302–2576 Fax: 0681–302–4290 Email: jkeller@cs.uni-sb.de

26.05.94

6. Übungsblatt zur Vorlesung

Rechnerarchitektur II — Parallelrechner

Vorlesung am 02.06.94 fällt aus, nächste Vorlesung am 09.06.94.
Die Übung am 30.05.94 fällt nicht aus.

1. Aufgabe
Zeigen Sie: Benötigt eine Menge von Speicherzugriffen auf der Fluent Machine $6n + \delta$ Takte, dann gibt es einen I/O-Pfad der Länge $6n + 2\delta/b$ und eine Sequenz aus aus δ verschiedenen Paketen, die entlang dieses Pfades polarisiert ist. Hierbei ist b die Pufferlänge der Netzwerkknoten.

2. Aufgabe
Zeigen Sie: Es gibt $2^{7n+2f}\binom{6n+2f}{f}$ Möglichkeiten, einen I/O-Pfad aus der ersten Aufgabe zu wählen, wobei $f = \delta/b$.

3. Aufgabe
Leiten Sie aus der Abschätzung der Vorlesung her: Die Wahrscheinlichkeit, daß ein Schritt der Fluent Machine mindestens Zeit $6n + kn$ dauert, ist kleiner als $(n2^n)^{-k}$. Hinweis: Benutzen Sie die Formel $\binom{n}{r} \leq (ne/r)^r$, wobei e die Euler'sche Zahl darstellt.

4. Aufgabe
Warum kann man in der SB-PRAM die Sortierfelder nicht zur Länge 1 schrumpfen (sie also weglassen) und nach jedem Paket ein End-of-Round in das Netzwerk schicken?

5. Aufgabe
Welche Vor- und Nachteile hat das Split-Phase-Protocol bei bus-basierten Parallelrechnern mit Shared Memory gegenüber Protokollen, bei denen ein Prozessor die Kontrolle über den Bus behält, bis er das angeforderte Datum aus dem Speicher erhält?

Postfach 151150 — 66041 Saarbrücken Tel.: 0681–302–2576 Fax: 0681–302–4290 Email: jkeller@cs.uni-sb.de

09.06.94

7. Übungsblatt zur Vorlesung
Rechnerarchitektur II — Parallelrechner

1. Aufgabe
Virtual shared memory Systeme verwalten ihren Adreßraum meist seitenweise. Wie hängt die Seitengröße von der darunterliegenden physikalischen Maschine und von der Anwendung ab?

2. Aufgabe
Nehmen Sie bei einer KSR1 mit 1088 Prozessoren eine Cache Treffer Rate von x an, wobei $0 \leq x \leq 1$. Bei einem Cache Miss soll der Cache, der das Datum liefern kann, mit gleicher Wahrscheinlichkeit auf einem der anderen 1087 Prozessoren sein. Berechnen Sie die mittlere Zugriffszeit, wenn der Zugriff auf einen Cache Zeit t_a und ein Schritt auf einem Ring Zeit t_b braucht. Nehmen Sie dabei an, daß keine Konflikte beim Routing auftreten.

3. Aufgabe
Wie Aufgabe 7.2, aber für Stanford DASH mit 16 Clustern. Der Parameter t_b beschreibt hier die Zeit für einen Schritt im Gitter. Wie verändert sich die Zugriffszeit, wenn mit einer Wahrscheinlichkeit von y, wobei $0 \leq y \leq 1$, der Home Cluster der gesuchten Page die Anfrage an einen weiteren Cluster weiterleitet, der mit gleicher Wahrscheinlichkeit einer der anderen 14 Cluster ist?

Postfach 151150 — 66041 Saarbrücken Tel.: 0681–302–2576 Fax: 0681–302–4290 Email: jkeller@cs.uni-sb.de

16.06.94

8. Übungsblatt zur Vorlesung
Rechnerarchitektur II — Parallelrechner

1. Aufgabe
Welche Vor– und Nachteile bieten die drei verschiedenen Programmiermodelle Nachbar-Nachbar, vollständige Vernetzung, System von Prozessen auf Message Passing Maschinen?

2. Aufgabe
Zeigen Sie:
Schneidet man ein n–dimensionales Butterfly Netzwerk zwischen Level j und $j+1$ durch (d.h. man entfernt alle Kanten zwischen diesen Levels), wobei $0 \leq j < n$, dann gilt:

- Die Stufen 0 bis j zerfallen in 2^{n-j} verschiedene j–dimensionale Butterfly Netzwerke.
- Die Stufen $j+1$ bis n zerfallen in 2^{j+1} verschiedene $(n-j-1)$–dimensionale Butterfly Netzwerke.

3. Aufgabe
Bestimmen Sie die jeweiligen Knotenmengen der neuen Butterfly Netzwerke in Aufgabe 2.

Postfach 151150 — 66041 Saarbrücken Tel.: 0681–302–2576 Fax: 0681–302–4290 Email: jkeller@cs.uni-sb.de

22.06.94

9. Übungsblatt zur Vorlesung
Rechnerarchitektur II — Parallelrechner

1. Aufgabe
Lösen Sie die Ungleichung

$$\left(\frac{N}{\sqrt{P}}\right)^2 \cdot t_{5\text{punkt}} \geq L + (g-o) \cdot \left(\frac{4N}{\sqrt{P}} - 1\right)$$

aus der letzten Vorlesung nach $N/\sqrt{P}$ auf. Bestimmen Sie eine minimale Datengröße N, so daß für $L = 300\ \mu s$, $o = 1\ \mu s$, $g = 2.3\ \mu s$, $t_{5\text{punkt}} = 2.75\ \mu s$ die Dauer eines Iterationsschrittes minimiert wird. (Die Zeiten wurden nach dem nCube 2 modelliert.)

2. Aufgabe
Was ist die optimale Zeit zur Simulation eines n–dimensionalen Butterfly Netzwerkes auf einem Rechner mit P Prozessoren? Um welchen Faktor ist Sahay's Simulation im LogP Modell, die Zeit $(n2^n)/P + ((n2^n)/P - (n2^n)/P^2 - 1)g + L$ braucht, von der optimalen Zeit maximal entfernt?

3. Aufgabe
Sei $i = q + r$ mit $q, r \geq 0$ und sei π eine binäre Permutation von $S_i = \{0, \ldots, 2^i - 1\}$. Zeigen Sie:

- Es gilt $\left\lfloor \frac{\pi(k2^r)}{2^r} \right\rfloor = \left\lfloor \frac{\pi(k2^r + j)}{2^r} \right\rfloor$ für alle j und k mit $1 \leq j < 2^r$ und $0 \leq k < 2^q$.
- Für jedes k mit $0 \leq k < 2^q$ ist π^k mit $\pi^k(i) = \pi(k2^r + i) \bmod 2^r$ eine binäre Permutation von S_r.
- Die Folge $\left(\left\lfloor \frac{\pi(k2^r)}{2^r} \right\rfloor\right)$, wobei $k = 0, 1, \ldots, 2^q - 1$, ist eine binäre Permutation von S_q.

Hinweis: benutzen Sie Induktion.

Postfach 151150 — 66041 Saarbrücken Tel.: 0681–302–2576 Fax: 0681–302–4290 Email: jkeller@cs.uni-sb.de

30.06.94

10. Übungsblatt zur Vorlesung
Rechnerarchitektur II — Parallelrechner

1. Aufgabe
Zeigen Sie:
Bei der in der Vorlesung vorgestellten Einbettung des (umgedrehten) Butterfly Netzwerkes nach Sahay kommen die P Inputs jedes Butterfly Netzwerkes des zweiten Teils aus P verschiedenen Prozessoren.

2. Aufgabe
Sei $t_0 = m - 1$ und $t_a = t_{a-2^{r-1}} + r2^{r-1}$ für $a = q2^{r-1}$ und q ungerade. Zeigen Sie:
Für Werte a mit Binärdarstellung $b_t b_{t-1} \ldots b_0$ ist $t_a = (m-1) + \sum_{i=0}^{t} b_i(i+1)2^i$.

Postfach 151150 — 66041 Saarbrücken Tel.: 0681–302–2576 Fax: 0681–302–4290 Email: jkeller@cs.uni-sb.de

07.07.94

11. Übungsblatt zur Vorlesung
Rechnerarchitektur II — Parallelrechner

1. Aufgabe
Beweisen Sie das folgende Lemma aus der Vorlesung:
Falls $a \equiv 0 \bmod 2^{g-1}$ dann ist $s_a = t_a$, sonst ist $s_a > t_a$.

2. Aufgabe
Beweisen Sie das Theorem aus der Vorlesung:
Seien N und P Zweierpotenzen mit $N \geq P^2$, dann gilt für die Laufzeit des komplizierten Algorithmus von Sahay

$$T = \begin{cases} m - 1 + g \cdot (m - l - 1) & \text{falls} \quad g \geq \log m \\ m \log m + 2^g - g - 1 - l \cdot g & \text{falls} \quad \log(N/P^2) + 1 < g < \log m \\ m \log m + (2^g - 1) - l(\log l + 1) & \text{falls} \quad g \leq l + 1 \ . \end{cases}$$

3. Aufgabe
Um welchen Faktor ist der zweite Algorithmus von Sahay, dessen Laufzeit in Aufgabe 11.2 bewiesen wurde, von der optimalen Laufzeit entfernt?

Vollständiger Beweis von Lemma 1 aus der Vorlesung vom 30.06.94

Lemma 1 Für den zweiten Algorithmus von Sahay gilt $t_0 = m - 1$. Für $a = q \cdot 2^{r-1}$ mit ungeradem q gilt $t_a = t_{a-2^{r-1}} + r2^{r-1}$.

Beweis: Zur Zeit t_0 ist Output 0 berechnet. Da dies der erste berechnete Output ist, müssen alle Vorgänger berechnet werden. Der Output liegt in Level $\log m$. Er hat 2 Vorgänger in Level $\log m - 1$, 4 Vorgänger in Level $\log m - 2$, allgemein 2^x Vorgänger in Level $\log m - x$, wobei $x = 1, 2, \ldots, \log m - 1$, da alle Vorgänger bis einschließlich Level 1 berechnet werden müssen. Die Berechnung jedes Vorgängers braucht einen Schritt, die Anzahl der Vorgänger ist $\sum_{x=1}^{\log m - 1} 2^i = m - 2$. Die Berechnung des Outputs selbst dauert ebenfalls einen Schritt, also ist er nach $m - 2 + 1 = m - 1$ Schritten berechnet.

Den allgemeinen Teil beweisen wir mittels folgender Beobachtung über Butterfly Netzwerke. Die Knoten $a' = a - 2^{r-1}$ und a unterscheiden sich in ihrer Binärdarstellung nur in Bit $r-1$, dieses ist bei a' Null und bei a Eins. Die Bits 0 bis $r-2$ haben den Wert Null. Deshalb sitzen a' und a als Knoten ganz links in Sub-Netzwerken mit 2^{r-1} Eingängen wie in Abb. 12.1 zu sehen. Die Vorgänger von a' sind schraffiert. Damit ist klar, daß die Vorgänger von a, a' und allen Knoten dazwischen in den Levels 1 bis $\log m - r$ identisch sind. Haben wir Output a' berechnet, so müssen wir für die Outputs zwischen a' und a in Level $\log m - x$ des linken Butterfly Netzwerkes $2^{r-1} - 2^x$ Knoten berechnen, wobei $x = 0, 1, \ldots, r-1$. Um Output a zu berechnen, müssen wir in Level $\log m - x$ des rechten Butterfly Netzwerkes 2^x Knoten berechnen, wobei $x = 0, 1, \ldots, r-1$. Insgesamt müssen wir also in r Levels jeweils 2^{r-1} Knoten berechnen, nachdem wir a' berechnet haben. Damit gilt $t_a = t_{a'} + r2^{r-1}$. □

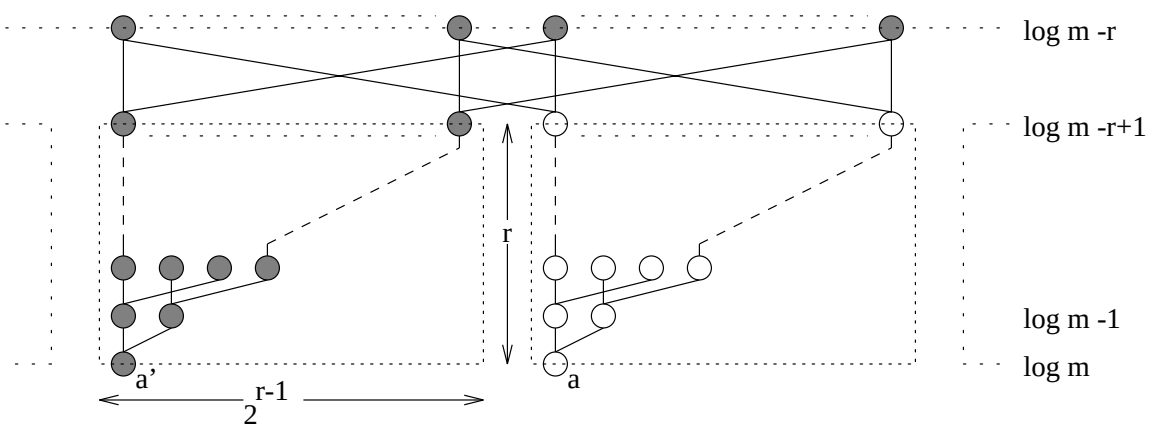


Abbildung 12.1: Position von a' und a