

Universität Karlsruhe (TH)

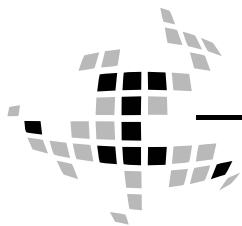
Forschungsuniversität · gegründet 1825

Programmiermodelle

Prof. Dr. Walter F. Tichy

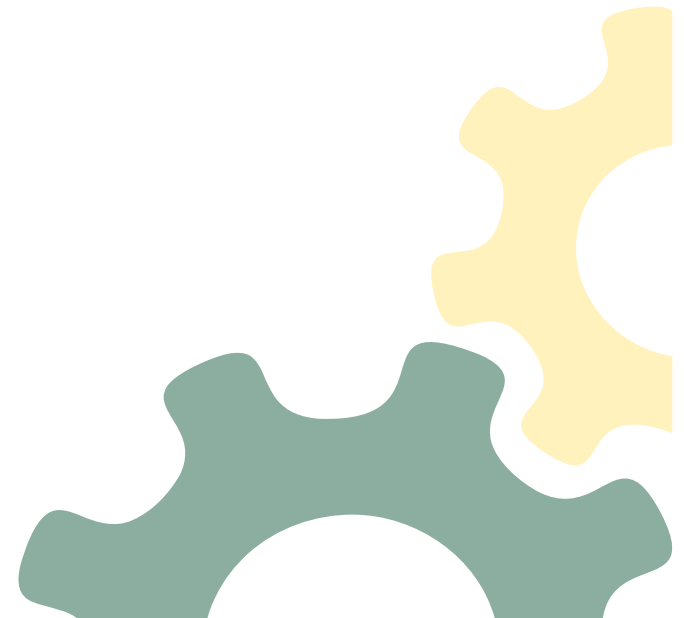
Dr. Victor Pankratius

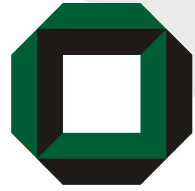
Ali Jannesari



Fakultät für **Informatik**

Lehrstuhl für Programmiersysteme



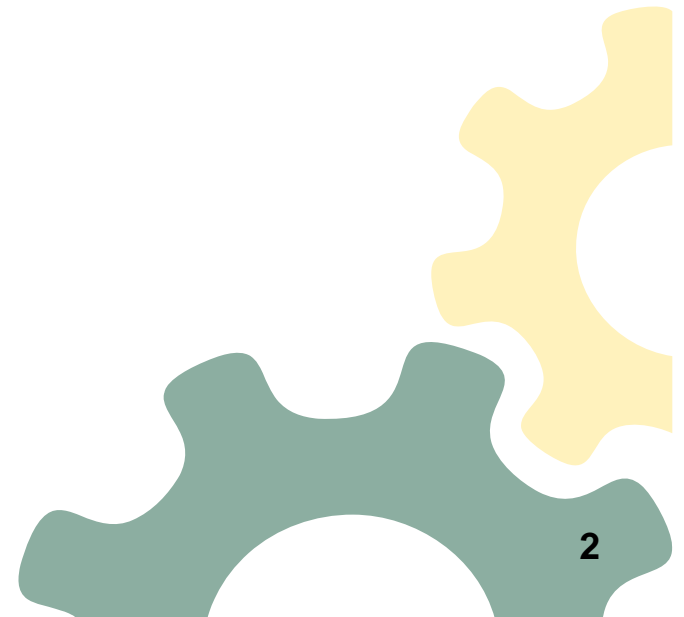


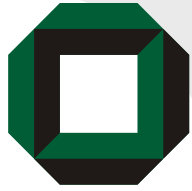
Vorlesung „Cluster Computing“

- Architektur von Rechnerbündeln
- Betrieb von Rechnerbündeln
- Nutzung und Programmierung von Rechnerbündeln

→ Programmiermodelle

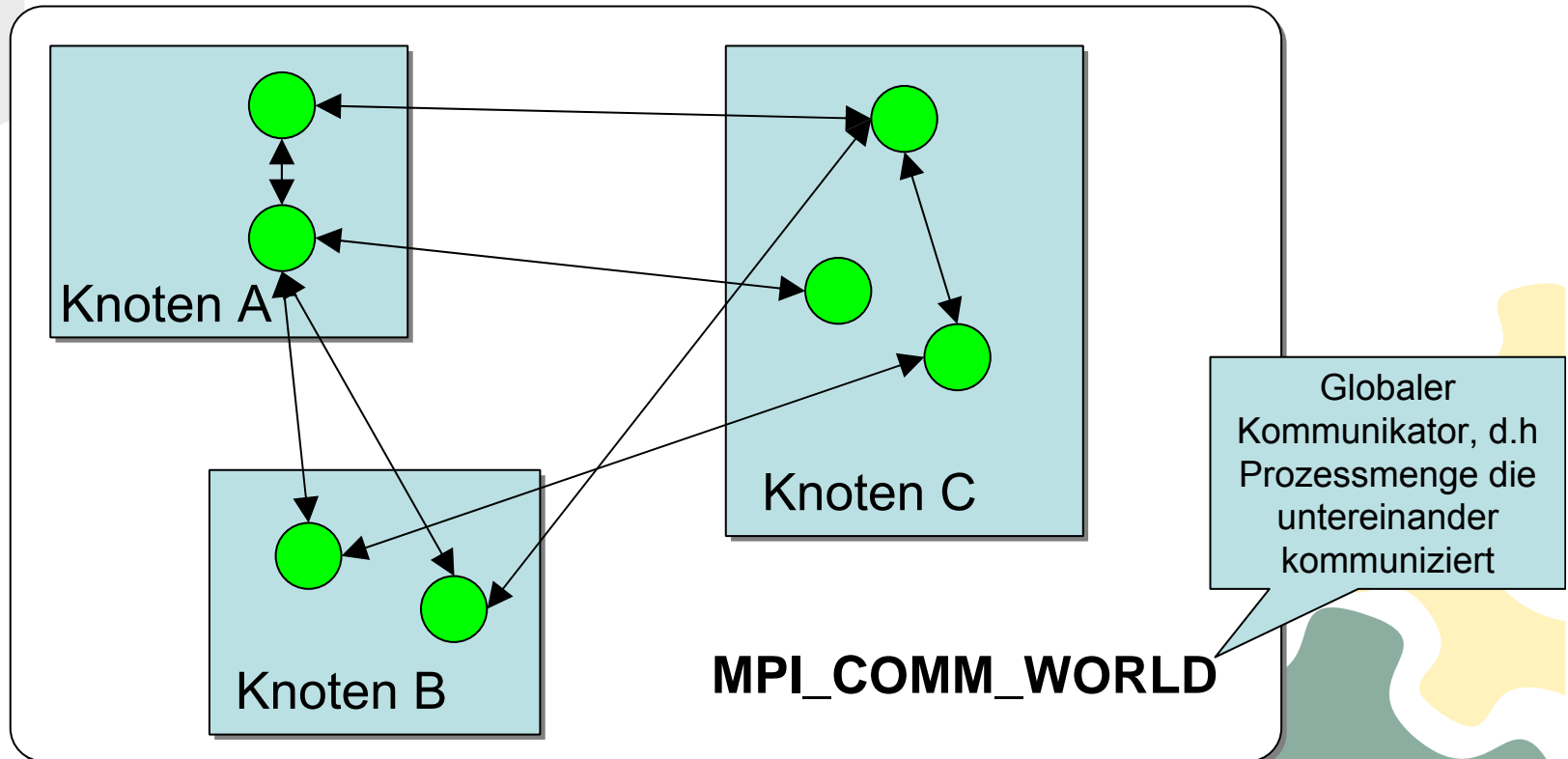
- MPI - Message Passing Environment
- Linda (Tupel Space)
- Netzhauptspeicher (Network Ram)
- DSM - Distributed Shared Memory
- MPI (ausführlich)
- JavaParty
- Platzierung

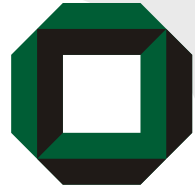




Message Passing Interface MPI (1)

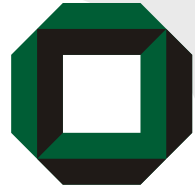
- Bibliothek zum Botschaftenaustausch (synchron und asynchron, kollektive Operationen wie Barriere, Rundruf, Einsammeln, Verteilen)
- eine beliebige Menge an Rechnerknoten spannt eine virtuelle Maschine auf
- Auf jedem Knoten dürfen auch mehrere Prozesse laufen (Multiprozessorknoten)





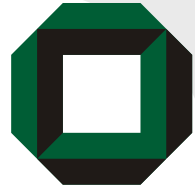
MPI (2)

- Einrichten der MPI-Umgebung:
 - Eintragen der beteiligten Knoten in eine statische Knotenliste
 - Prozesse werden dann auf den einzelnen Knoten gestartet (mpirun -np 8 ./appl -param)
- Eigenschaften:
 - Konzept des Kommunikators: Jeder Knoten befindet sich in einem (oder mehreren) Kommunikatoren, wobei die Knoten innerhalb eines Kommunikators von 0 bis n-1 nummeriert sind.
 - dynamisches Erzeugen von Prozessen erst mit MPI-2
 - bisher keine Möglichkeit, Prozesse zu migrieren
- Genaueres über MPI in einer separaten Präsentation.

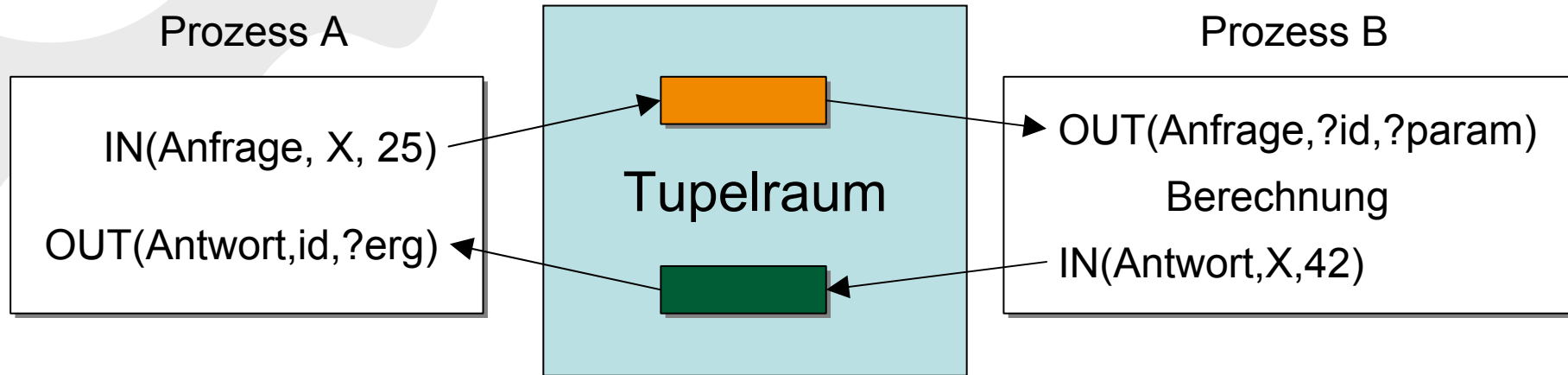


Linda (1)

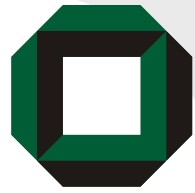
- zentrales Konzept ist der Tupelraum, der Objekte enthält
 - Tupelraum als gemeinsamen Speicher (repliziert oder verteilt realisiert)
- Operationen:
 - IN(t): fügt Element in Tupelraum ein
 - OUT(t): entfernt Element aus Tupelraum
 - READ(t): liest Element im Tupelraum (kein Entfernen)
 - EVAL(t): erzeugt neuen Prozess (mit entsprechendem Tupel)
- Keine neue Sprache, sondern eine Bibliothek für eine Sprache, z.B. C-Linda oder Fortran-Linda.



Linda (2)

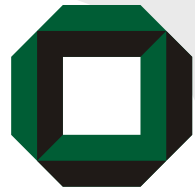


- Der Tupelraum ist ein großer, gemeinsamer, potenziell verteilter Speicher.
- Jeder Prozessor kann Tupel ablegen, lesen oder herausnehmen.
- Ein Tupel besteht aus einem oder mehreren Elementen der Grunddatentypen wie integer, float, etc, aber auch Feldern, Zeichenreihen, Verbunden und Objekten, aber nicht aus anderen Tupeln.



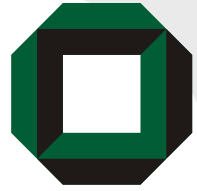
Linda (3)

- Die Operationen OUT, READ und EVAL geben ein Muster an, um ein Tupel zu suchen.
 - OUT("abc", 2, ?i) sucht nach einem Tupel, das aus der Zeichenreih "abc", der Zahl 2 und einer weiteren Ganzzahl (falls i ganzzahlig) besteht. Zahl, Typen und Werte, falls vorhanden, der Elemente eines Musters müssen mit denen des gesuchten Tupels übereinstimmen. Variablen mit ? werden die entsprechenden Werte des passenden Tupels zugewiesen.
 - Wenn es gleichzeitig mehrere OUTs für das gleiche Tupel gibt, dann bekommt es nur einer der Prozesse (es sei denn, es gibt mehrere Kopien). Wenn es kein passendes Tupel gibt, blockiert der Prozess, der OUT ausführt, bis ein geeignetes Tupel abgelegt wird.



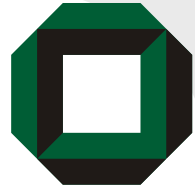
Linda (4)

- Eine Warte- oder P-Operation an einem Semaphor S könnte man so verwirklichen:
 - $OUT(\text{„Semaphor } S\text{“})$
- Die Freigabe- oder V-Operation wäre
 - $IN(\text{„Semaphor } S\text{“})$.
- Die Anzahl der Tupel („Semaphore S “) gibt an, wie viele Prozesse gleichzeitig aktiv sein können.
- Andere Kommunikationsmuster, wie Senden/Empfangen oder Austausch sind leicht zu programmieren.



Linda (5)

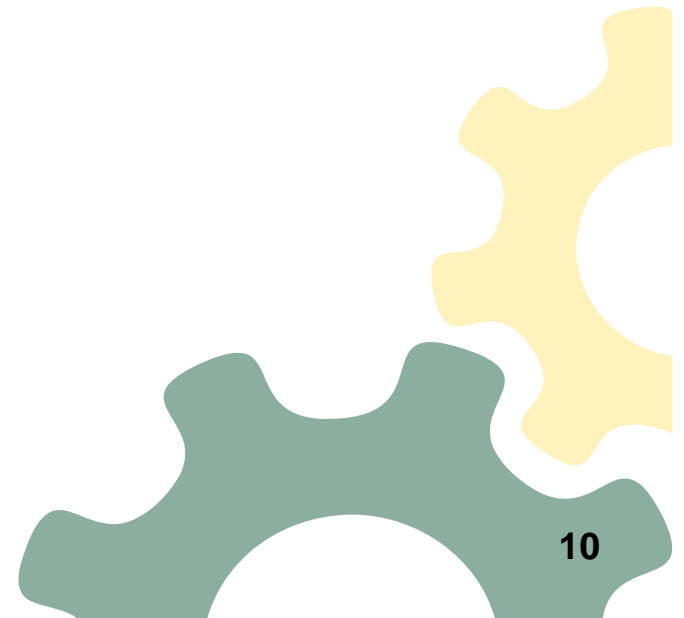
- Linda-Programme sind oft nach dem Master/Slave-Prinzip aufgebaut. Ein Auftraggeberprozess legt neue Aufgaben mit
 - IN(„Aufgabengruppe“, Aufgabe)
- Arbeiter holen sich die Aufgabe mit
 - OUT(„Aufgabengruppe“, ?Aufgabe)

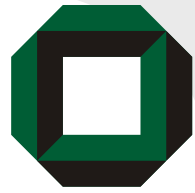


Linda (7)

Implementierungsaufgaben:

1. schnelle, assoziative Suche nach Tupeln
2. Partitionierung des Tupelraumes, Suche auf mehreren Maschinen

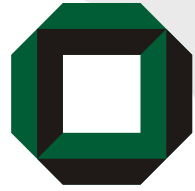




Linda (8)

Zu 1 (assoziative Suche):

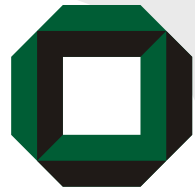
- Tupel werden nach ihrer Typsignatur in mehreren Teilmengen geordnet (z.B. (string, int), (string, int, int), (string, int, float), etc. Innerhalb der Teilmengen werden Hashtabellen aufgrund der Werte gebildet.
- Wenn eine Tupelanfrage ankommt, dann wird die entsprechende Typsignaturmenge ausgewählt und dann eine Hashsuche mit einem geg. Wert durchgeführt. So muss selten lange gesucht werden.



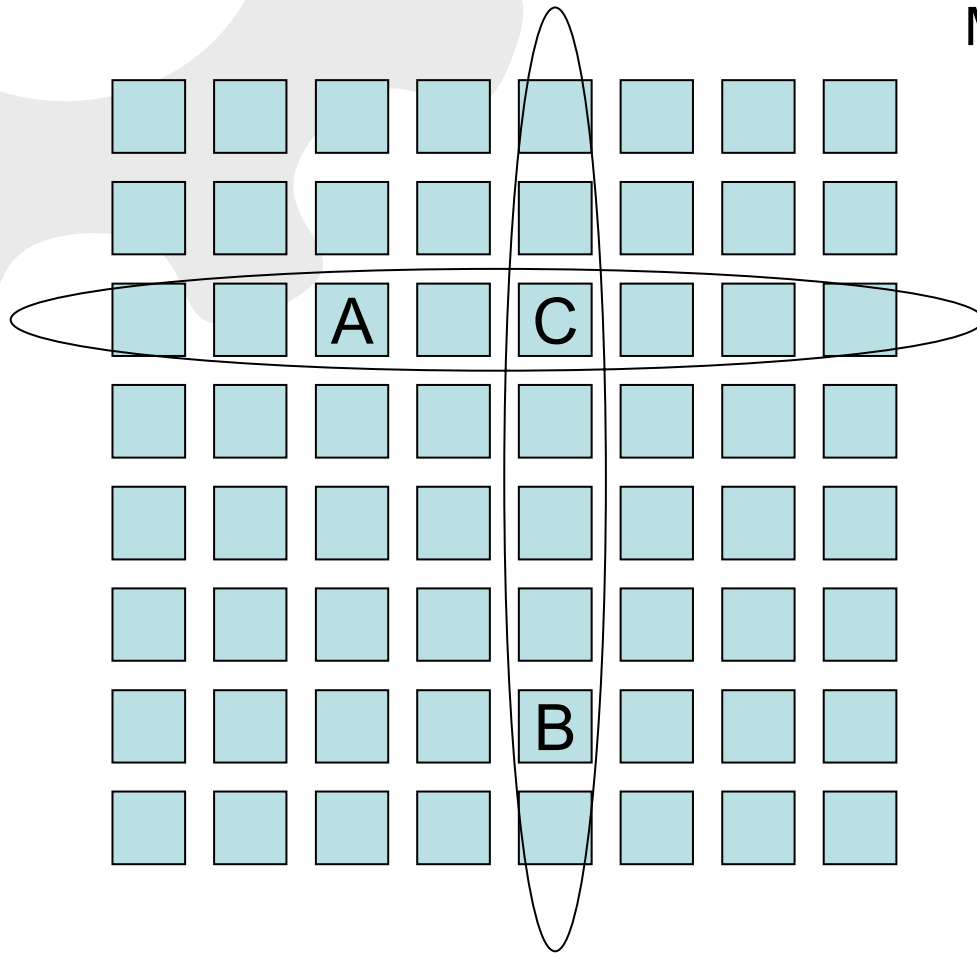
Linda (9)

Zu 2 (Partitionierung): Jeder Prozessor hat einen lokalen Tupelraum.

- Bei schnellem Rundruf: volle Replizierung des Tupelraumes. Ein IN wird an alle teilnehmenden Prozessoren geschickt. Das Lesen von Tupeln geschieht lokal. Die Wahl und Entnahme eines Tupels geschieht ebenfalls lokal, aber die Entnahme muss per Rundruf an alle geschickt werden (Synchronisation bei mehreren gleichzeitigen Zugriffen auf das gleiche Tupel notwendig).
- Alternativ: IN geschieht nur lokal. Anfrage an den Tupelraum geschieht per Rundruf. Alle passenden Tupel werden an den Anforderer geschickt. Dieser wählt eines aus, belässt den Rest in seinem lokalen Speicher (Probleme mit Gleichzeitigkeit!). Wenn nichts zurückkommt, dann wird der Rundruf in größer werdenden Abständen wiederholt.

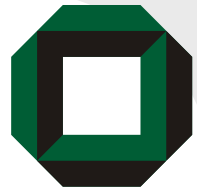


Linda (10)



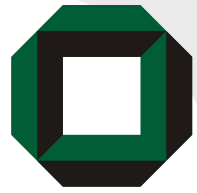
Mischform: teilweise Replizierung

- Prozessoren schicken ihre Tupel an alle Prozessoren in ihrer Zeile. Anfrage nach Tupel geht an alle in der gleichen Spalte. A schickt IN an alle in Zeile 3. B schickt OUT an all in Spalte 5. C sieht sowohl OUT als auch zugehöriges IN. Danach muss das entnommene Tupel in Zeile 3 gelöscht werden (Gleichzeitigkeit beachten!) (anstelle von n Prozessoren sind nur noch $2\sqrt{n}$ involviert)



Netz Hauptspeicher (1) (engl. Network RAM)

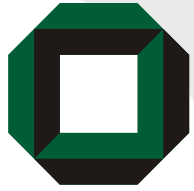
- **Idee:** Pufferung des Hintergrundspeichers im gesamten Hauptspeicher eines verteilten Systems; Zugriff auf die Puffer via Netz.
- Interessant für:
 - **Anwendungen** mit großem Speicherbedarf
 - simuliert großen und schnellen virtuellen Speicher (Seitenwechsel auf andere Rechner)
 - füllt die Lücke zwischen Speicher und Auslagerungsdatei (Festplatte)
 - **Dateisysteme** (Netz Hauptspeicher als Dateipuffer)
 - **Datenbanken** (Geschwindigkeit, Redundanz)



Netzhauptspeicher (2) Voraussetzungen

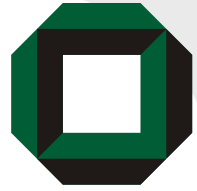
Zugriffszeit (Latenzzeit) und Durchsatz in der Speicherhierarchie

Speicherhierarchie	Latenz	Durchsatz	Kapazität
Cache	<0.005 μ s	2000 MB/s	2 MB
DRAM	~0.1 μ s	700 MB/s	1 GB
Network RAM	~30 μ s	100 MB/s	p * 1 GB
Disk	~7000 μ s	20 MB/s	100 GB



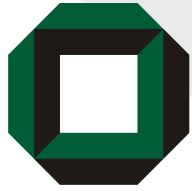
Netz Hauptspeicher (3) Verwendung

- **Seitenwechsel mit entferntem Hauptspeicher**
Speicherseiten werden nicht auf die lokale Platte sondern in den Speicher eines entfernten Knotens geschrieben (geht nur, wenn dort Speicher frei ist).
- **Netzdateisystem (network file system)**
verteilter Hauptspeicher wird als Zwischenspeicher für Dateien verwendet (entweder als Puffer oder ganze Dateien werden in Speicher abgebildet)
- **Netzdatenbanken (network data base)**
verteilter Hauptspeicher wird als Hauptspeicherdatenbank oder großer Datenbank-Puffer verwendet. Entfernter Hauptspeicher schneller als lokale Platte; Redundanz möglich (mit ununterbrechbarer Stromversorgung).

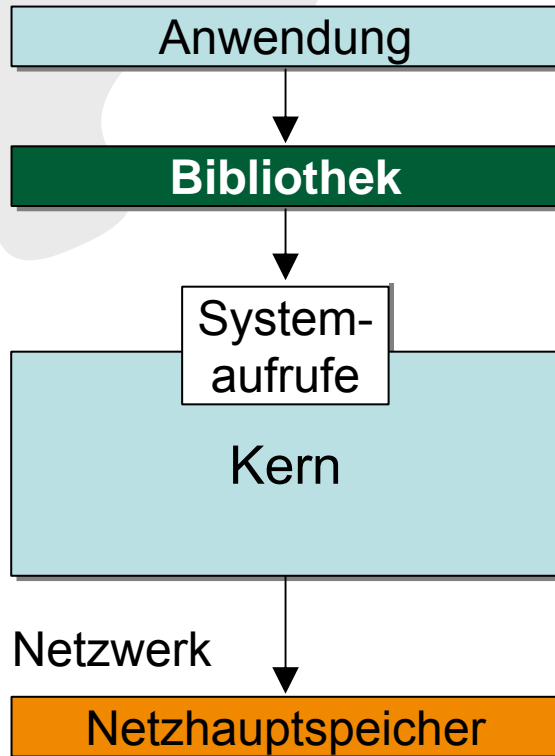


Netz Hauptspeicher (4) Implementierungsalternativen

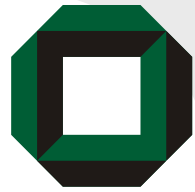
- neue Speicherverwaltungsroutinen innerhalb einer Bibliothek (malloc, free)
- neuer Gerätetreiber (Swap Device), der Seiten über das Netz auslagern kann
- Kernerweiterungen, die den gesamten im Cluster verfügbaren Speicher als eine Ressource verwalten.



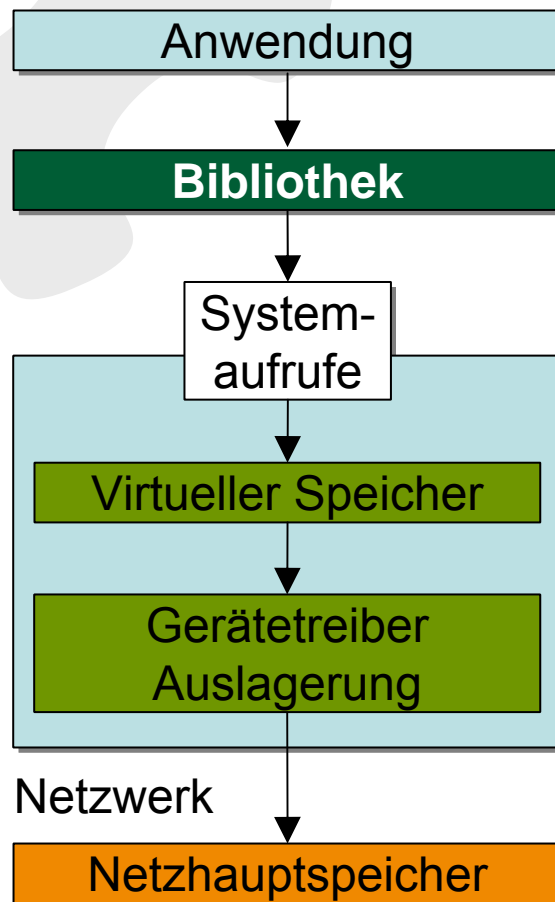
Netzhauptspeicher (5) als Bibliothek



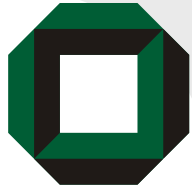
- Vorteil:
 - Einfach zu implementieren
- Nachteil:
 - Anwendungen müssen mit der neuen Bibliothek gebunden oder auf die neue Bibliothek angepasst werden.
 - andere Anwendungen oder Systemdienste sind vom Netzhauptspeicher ausgeschlossen



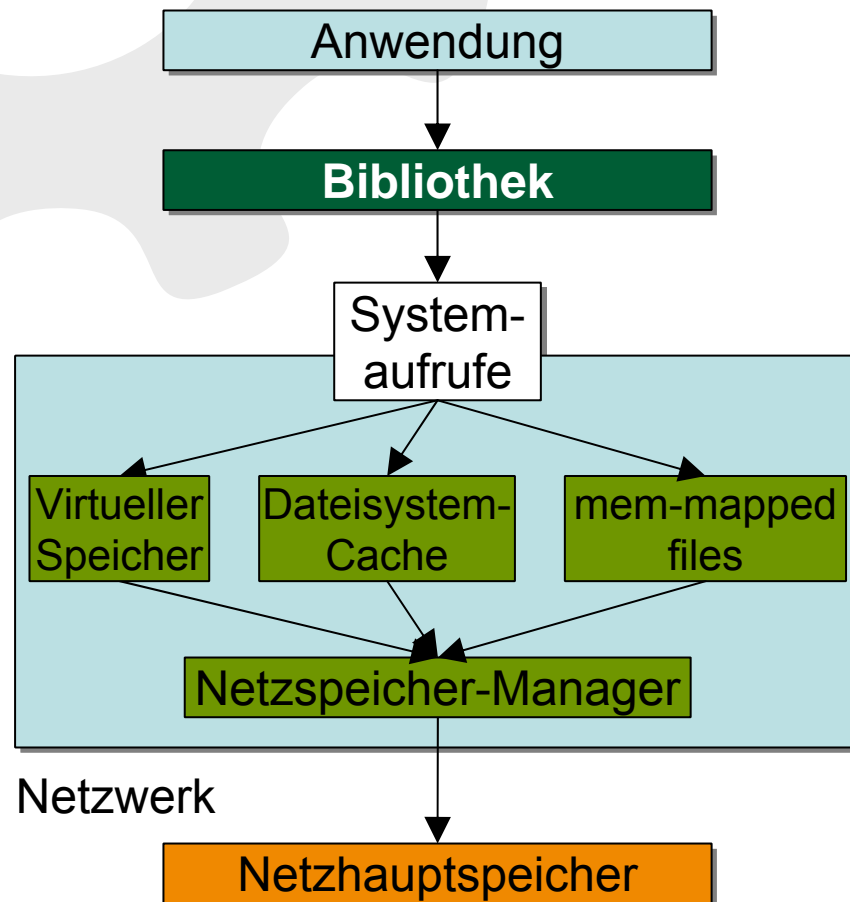
Netzhauptspeicher (6) als Gerätetreiber



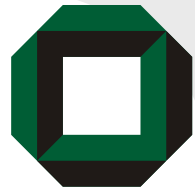
- Vorteil:
 - alle Applikationen können den Netzhauptspeicher nutzen
- Nachteil:
 - Implementierung des Gerätetreibers
 - Kerndienste (Dateisystem) können Netzhauptspeicher nicht nutzen



Netzhauptspeicher (7) als Kernerweiterung

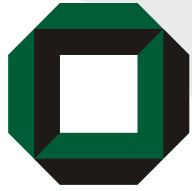


- Vorteil:
 - volle Systemintegration des Netzhauptspeicher
- Nachteil:
 - Implementierung der Kernerweiterungen

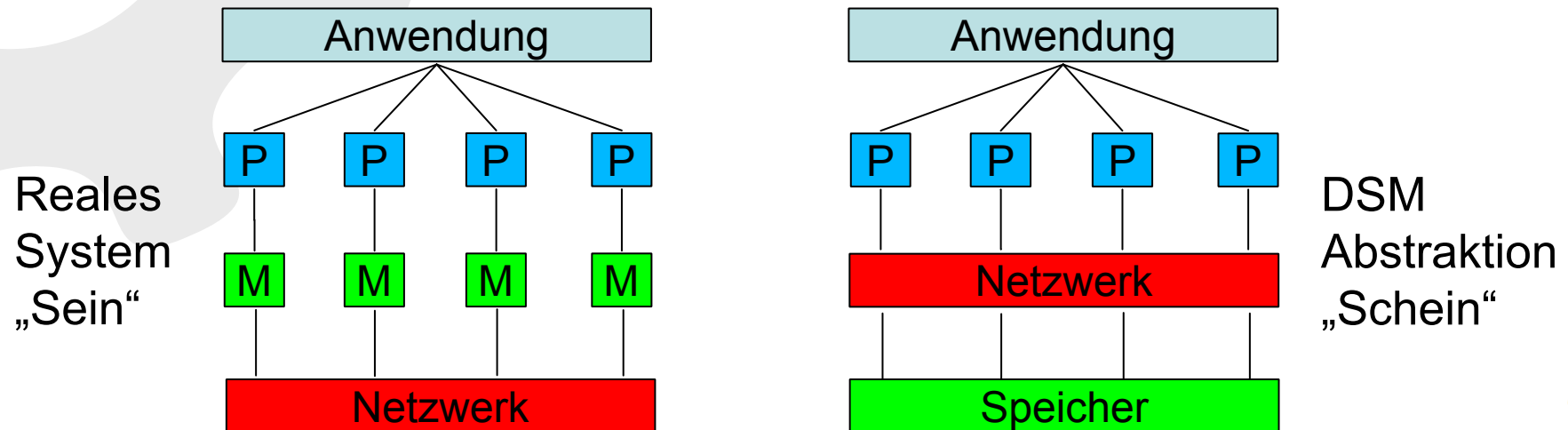


Netz Hauptspeicher (8)

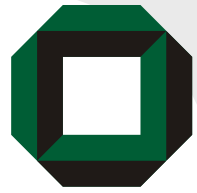
- Offene Probleme:
 - **Sicherheit:**
wie wird unbefugter Zugriff auf den Netz Hauptspeicher bzw. auf die übers Netz wandernden Speicherseiten verhindert?
 - **Zuverlässigkeit & Fehlertoleranz:**
was passiert beim Ausfall eines Knotens?
- Allgemeinere Lösung: Verteilter gemeinsamer Speicher (Netz Hauptspeicher puffert in diesem Fall nur Hintergrundspeicher im Netz.)



Verteilter, gemeinsamer Speicher Distributed Shared Memory (DSM)



- Ziel: Vorspiegelung eines gemeinsamen Speichers auf einem System mit physikalisch verteiltem Speicher (in Software)
- Seitenaustauschmechanismen: Zugriff auf entfernte Adresse führt dazu, dass die zugehörige Seite in den Hauptspeicher des Rechners, der den Zugriff ausführt, geholt wird.



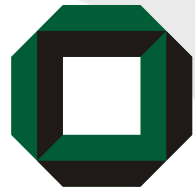
Distributed Shared Memory (2) Designentscheidungen

- **Datenmigration:**

Wie und unter welchen Bedingungen »wandern« die Daten von einem Prozessor auf einen anderen? Wo befinden sich meine Daten aktuell?

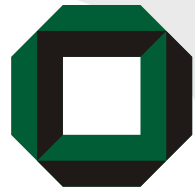
- **Datenreplikation:**

Unter welchen Bedingungen werden Daten auf den Prozessoren repliziert? Wie werden die Kopien verwaltet, insbesondere wie werden die Kopien konsistent gehalten? (Das ist einfach für Seiten mit konstantem Inhalt.)



Distributed Shared Memory (3) Datenmigration

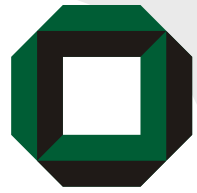
- Frage: Wem gehört eine Seite des DSM?
 - einem bestimmten (eindeutigen) Knoten?
(wird bei vielen Schreibzugriffen zum Flaschenhals)
 - einem beliebigen Knoten?
 - derjenige, der die Seite gerade benutzt?
(was geschieht, wenn es Replikate gibt?)
 - derjenige, der das Schreibrecht hält?
- Wie findet man den aktuellen Eigentümer einer Seite?
 - zentraler Koordinator oder
 - verteilte Informationsbasis



Distributed Shared Memory (4)

Datenreplikation

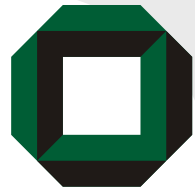
- Replikate sollen die Leistung steigern (so wie die lokalen Prozessorcaches in SMP Systemen)
- **Aber:** Wie werden die Replikate konsistent gehalten?
 - **strikte Konsistenz:** jede Leseoperation auf Variable x liefert den Wert, der von der letzten Schreiboperation auf x gespeichert wurde.
 - **schwache Konsistenz:** man gewährleistet den Zustand einer strikten Konsistenz nur zu bestimmten Zeitpunkten (Konsistenz bei Eintritt oder Austritt aus kritischem Abschnitt: entry consistency, release consistency)
 - **keine Konsistenz:** man stellt aber Mechanismen zur Synchronisation zur Verfügung.



Distributed Shared Memory (5)

Konsistenzerhaltung

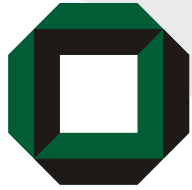
- Nur interessant für Seiten, die repliziert sind und sowohl gelesen als auch geschrieben werden.
- **Methode 1:** Aktualisierung. Hier müsste das geänderte Speicherwort gefunden und an alle Replikate geschickt werden. Das ist teuer für ein DSM-System und erzielt keine sequentielle Konsistenz bei mehreren, gleichzeitigen Änderungen (Reihenfolge der Änderungen kann nicht garantiert werden).
- **Methode 2:** Invalidierung: Immer, wenn auf eine replizierte Seite geschrieben wird, holt der entsprechende Prozessor die Seite zu sich (wird Eigentümer) und alle anderen Knoten müssen ihre Replikate invalidieren. Nur der Eigentümer darf schreiben. Replikate (zum Lesen) können danach wieder gebildet werden.



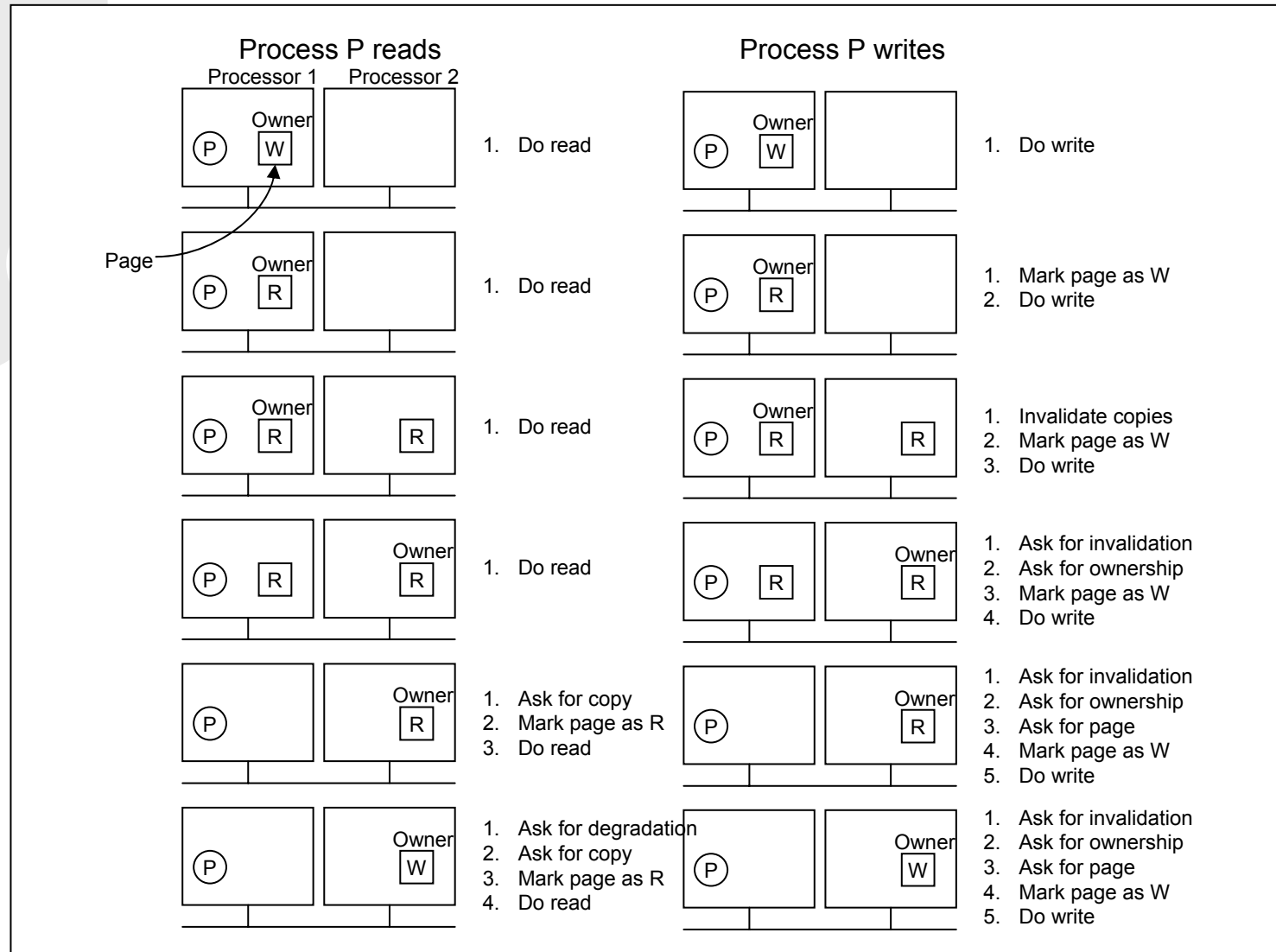
Distributed Shared Memory (6a)

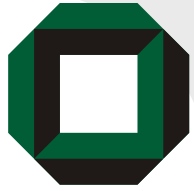
Konsistenzerhaltung

- Beispiel eines Protokolls für Methode 2
 - Jede Seite hat einen Besitzer, das ist der Prozess, der zuletzt darauf geschrieben hat.
 - Seite hat Status „W“: es gibt nur eine Kopie beim Besitzer; diese kann beschrieben werden.
 - Seite hat Status „R“: es gibt Kopien beim Besitzer und bei anderen; diese sind nur lesbar.
 - Es sind jeweils sechs Fälle zu unterscheiden, wenn ein Prozess eine Seite schreiben oder lesen will (vergl. Abbildung auf der nächsten Folie.) Es sind jeweils die Situation und die Schritte angegeben, die unternommen werden müssen.



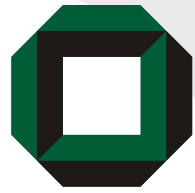
Distributed Shared Memory (6b)





Distributed Shared Memory (7) False Sharing Problem

- Angenommen, zwei Datenobjekte befinden sich auf der gleichen Seite. Ein Prozessor schreibt das eine, der andere das andere Objekt. Nun muss diese Seite bei jedem Zugriff von einem zum anderen Prozessor wandern.
- Dies ist ein ernstes Problem für verteilten, gemeinsamen Speicher. Es ist kaum zu vermeiden (man denke an dynamisch angelegte Objekte oder an Felder, bei denen benachbarte Elemente von unterschiedlichen Prozessoren bearbeitet werden) und bedeutet eine Zugriffsverlangsamung um mindestens 4 Größenordnungen.
- Je größer die Granularität (Seitengröße und Anzahl Seiten die auf einmal migriert werden), desto höher die Wahrscheinlichkeit für false sharing.



Distributed Shared Memory (8)

Beispiele

- IVY (Yale University, 1989)
- Mirage (erweitert IVY, 1989)
- Mether (1989)
- Munin (1990) (release consistency)
- Midway (1993) (entry consistency)
- TreadMarks (1996)
 - mittlerweile weit verbreitet
 - <http://www.cs.rice.edu/~willy/TreadMarks/overview.html>
- Literatur
 - Andrew S. Tanenbaum, Distributed Operating Systems, Prentice Hall, 1995.