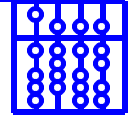


# Werkzeuggestützte Entwicklungsmethodik für hochparallele Programme

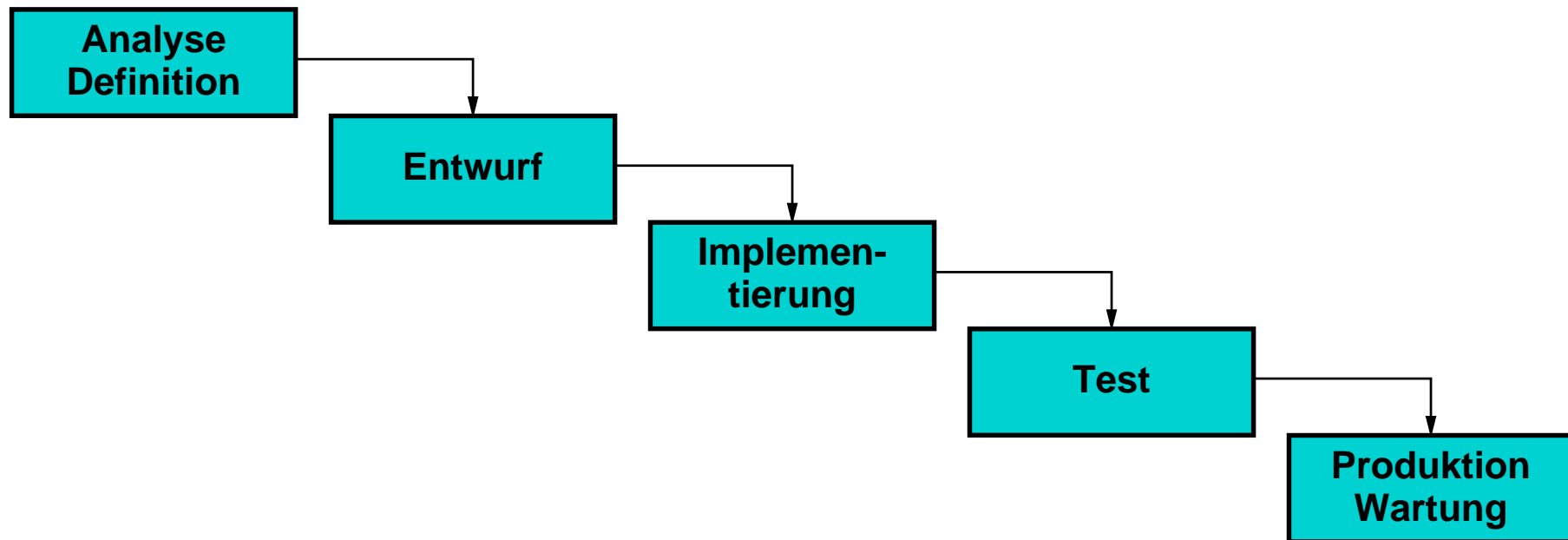
**Thomas Ludwig**

Lehrstuhl für Rechnerarchitektur und  
Rechnerorganisation (LRR-TUM)  
Institut für Informatik  
Technische Universität München

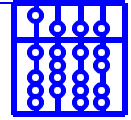
[ludwig@in.tum.de](mailto:ludwig@in.tum.de)  
[www.in.tum.de/~ludwig](http://www.in.tum.de/~ludwig)



- ↳ **Entwicklungsmethodik sequentieller Programme**
- ↳ **Entwicklungsmethodik paralleler Programme**
- ↳ **Die Entwicklungsmethodik im Detail**
- ↳ **Entwicklungswerkzeuge für parallele Programme**
- ↳ **Methodische Werkzeugkonstruktion**
- ↳ **Ein verteiltes Monitorsystem**
- ↳ **Beispiele parallelisierter Anwendungen**
- ↳ **Zusammenfassung**



Phasenmodell: linear oder zyklisch

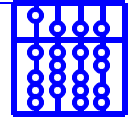


## Ausgangspunkt

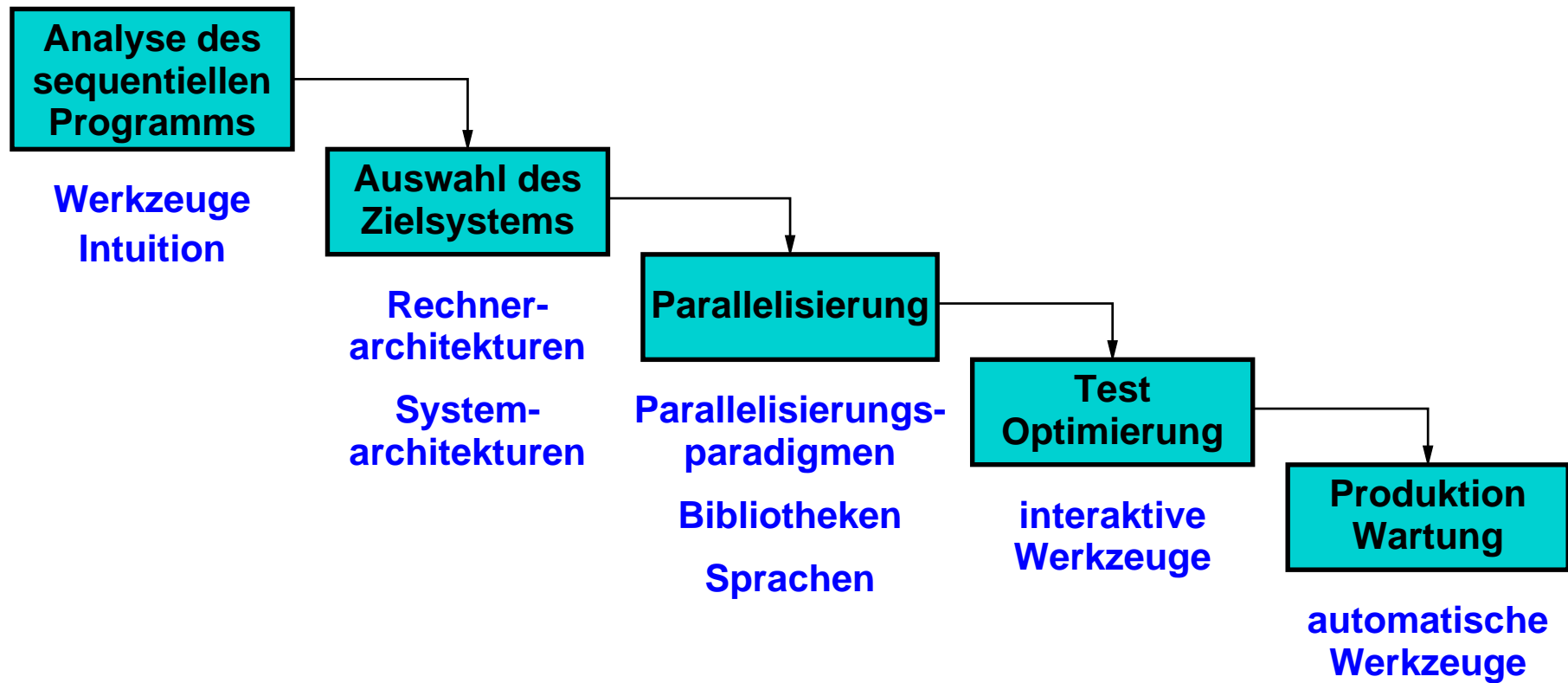
- ➔ Neuentwicklungen (10%)
- ➔ Aufarbeitung sequentieller Programme (90%)  
Sogenannter Legacy-Code oder Dusty-Deck-Code  
Meist nur Quellcode vorhanden; kein Know-How

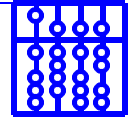
## Bei Neuentwicklungen

- ➔ Möglicherweise Spezifikation des Parallelismus
- ➔ Einsatz neuer Paradigmen  
z.B. objektorientierte und skelettorientierte Programmierung



## Aufarbeitung sequentieller Programme



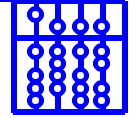


## Profiling charakteristischer Programmläufe

- ➔ Ermittlung des potentiellen Parallelismus
- ➔ Sequentielle Programmanteile ermitteln
- ➔ Leistungsgewinn abschätzen (Gesetz von Amdahl)

## Analyse charakteristischer Datenstrukturen

- ➔ Partitionierbarkeit — Replizierbarkeit — Größe
- ➔ Zeiten für Ein-/Ausgabe ermitteln

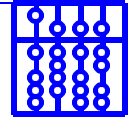


## Abhängigkeiten

- ➔ Verfügbarkeit der Systeme
- ➔ Leistungsbedarf der Anwendung
- ➔ Adäquanz für die Anwendung

## Architekturklassen

- ➔ Parallelrechner mit verschiedenen Architekturen  
Verteilter Speicher — gemeinsamer Speicher — Mischformen
- ➔ Workstation-Cluster  
Standardvernetzung — Spezialkopplungen (SCI)



## Parallelisierungsparadigmen

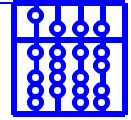
↳ Codeaufteilung — Datenaufteilung — Mischformen

## Sprachkonzepte

↳ Nachrichtenaustausch: MPI, PVM

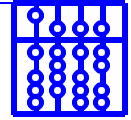
↳ Gemeinsamer Speicher: OpenMP

## Codeumstrukturierungen und -erweiterungen



## Methodische Fragen

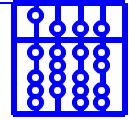
- ➔ Welches Paradigma für welche Anwendungsklasse?
- ➔ Ausnutzung von Asynchronität und Nichtdeterminismus
- ➔ Verhältnis sequentielles Programm zu parallelem Programm
- ➔ Überlappung von Berechnung und Kommunikation
- ➔ Einbau von Lastausgleich



## Fehlersuche — Deterministische Programmausführung — Leistungsanalyse

### Probleme

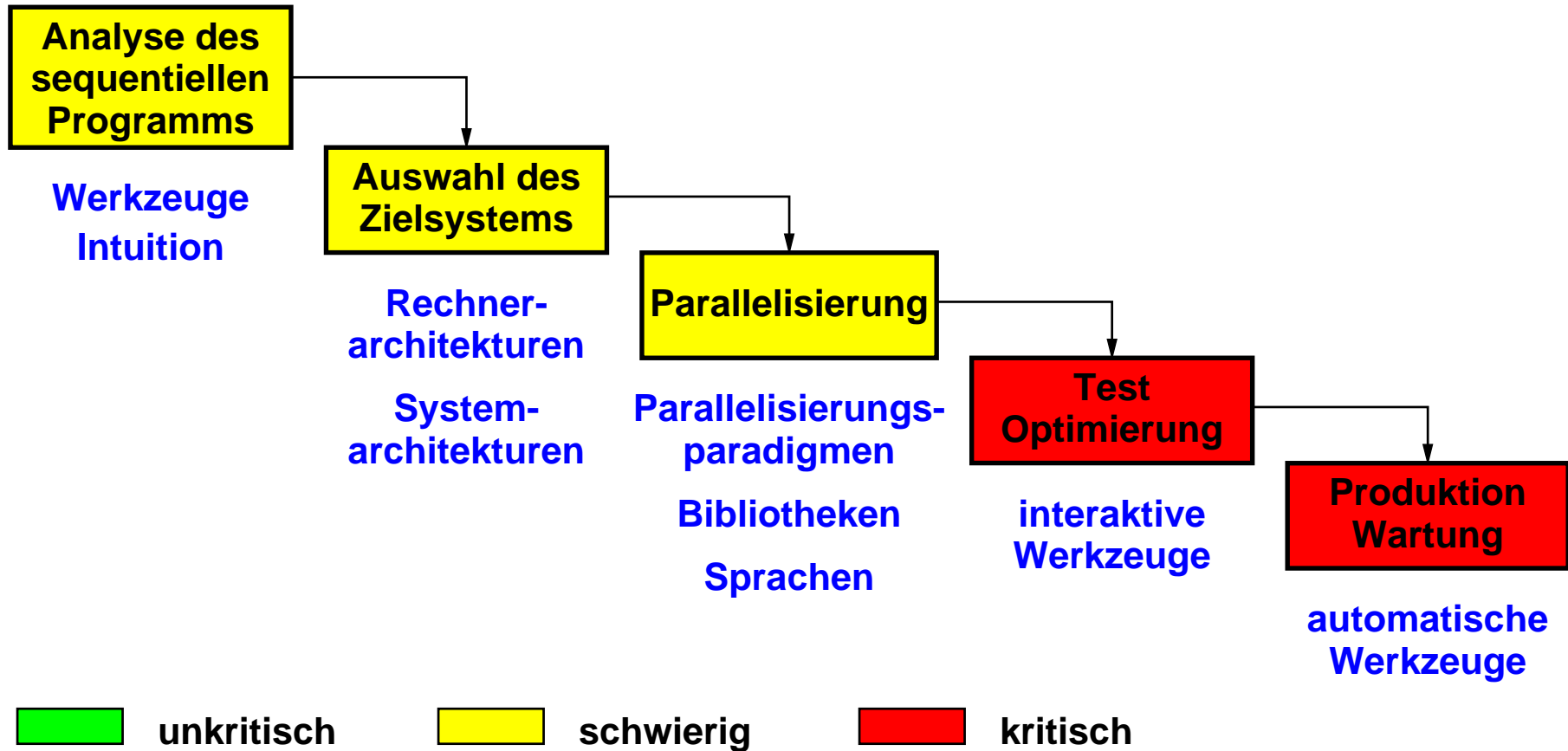
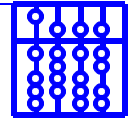
- ➔ Räumliche Verteilung
- ➔ Keine globale Zeit
- ➔ Absichtlicher Nichtdeterminismus
- ➔ Nichtreproduzierbarkeit von Ergebnisse



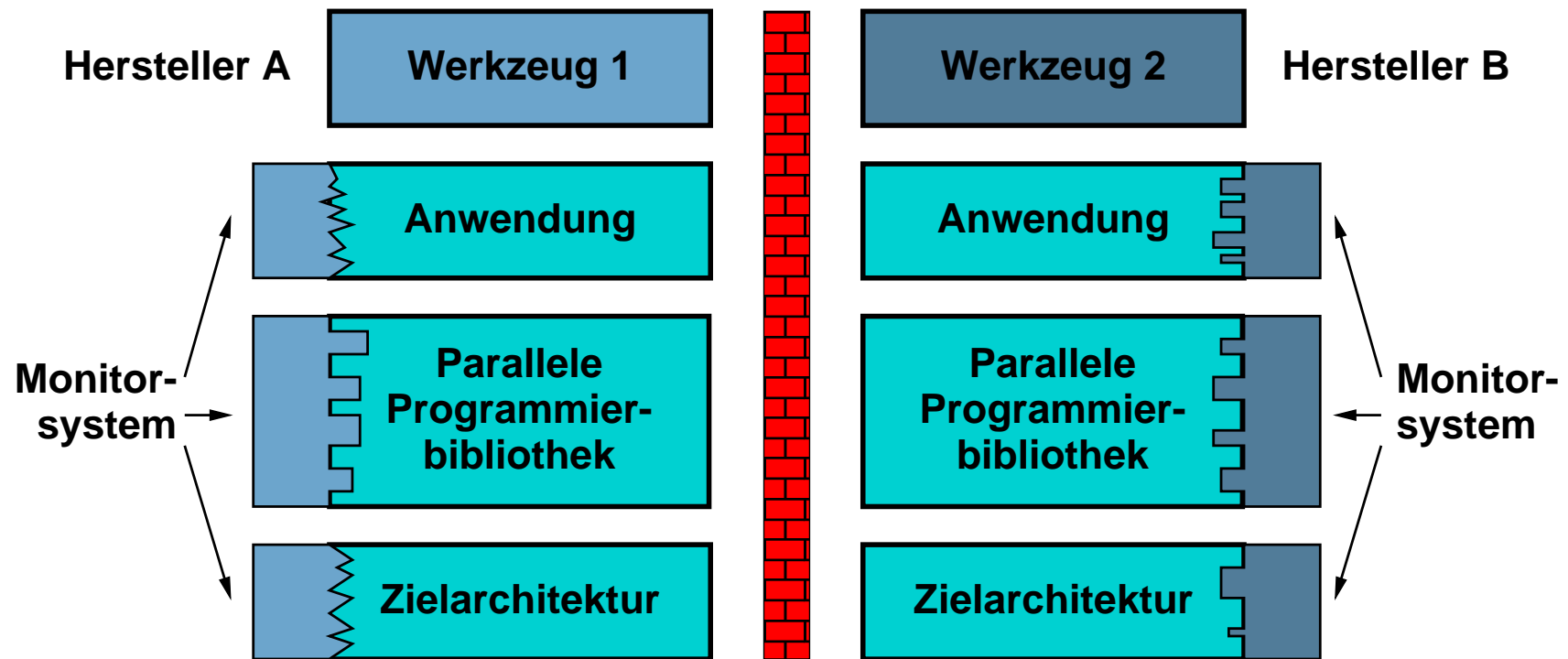
## Lastausgleich — Ressourcenverwaltung — Programmsteuerung

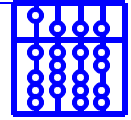
### Probleme

- ➔ Abhängigkeit vom Betriebssystem
- ➔ Abhängigkeit von der Hardware



- ➡ Kein theoretisches Fundament (Betriebssysteme, verteilte Systeme)
- ➡ Ad-hoc-Entwurf mit monolithischem Aufbau



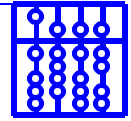


## Aufwendige Werkzeugkonstruktion

- ➔ Ständige Neuentwicklung bereits vorhandener Funktionalitäten
- ➔ Benutzer hat schlechte Auswahl
- ➔ Konstruktion paralleler Software erschwert

## Mangelhafte Konzepte bei den Werkzeugen

- ➔ Keine Kooperation möglich  
Grund: Werkzeuge wissen nichts voneinander
- ➔ Keine gemeinsame Verwendung möglich  
Grund: Werkzeuge verwenden verschieden modifizierte Systemkomponenten



**Prinzip:** Schnittstelle zwischen Werkzeug und Monitorsystem einziehen

**Folge:** Monitorsystem nur einmal realisieren; beliebige Werkzeuge aufsetzen

**Die früheren Probleme sind damit eliminierbar**

- ↳ Werkzeuge verwenden inkompatible Systemmodifikationen
- ↳ Werkzeuge wissen nichts voneinander

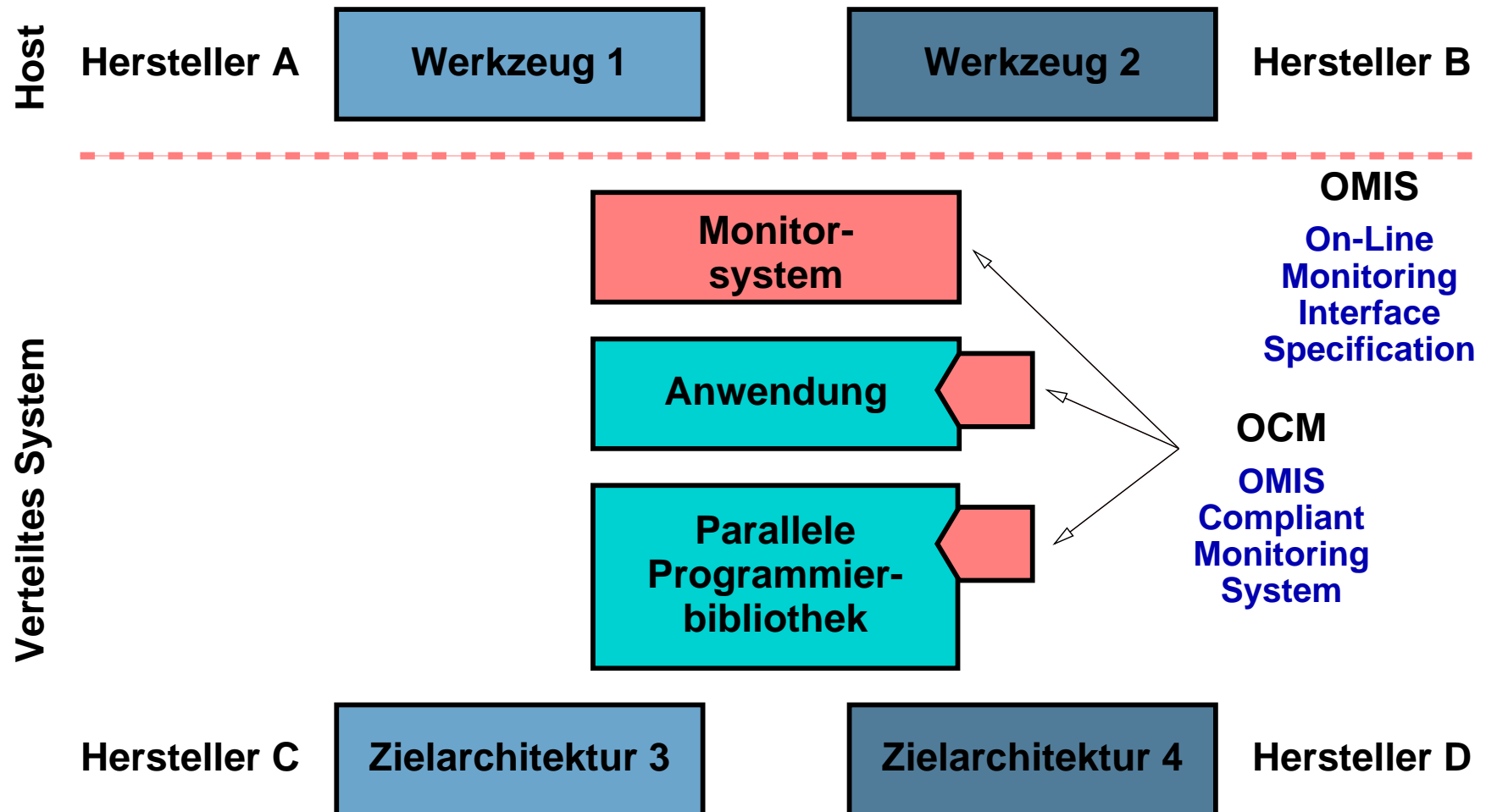
**Gemeinsame Schnittstelle für alle Werkzeuge**

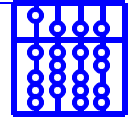
- ↳ Basis für Werkzeugintegration und Werkzeuginteroperabilität

**Gemeinsames Monitorsystem auf verschiedenen Architekturen**

- ↳ Basis für universelle Werkzeuge

## Schnittstellenbasierte Werkzeugkonstruktion





## **Unterstützung von Interoperabilität und Universalität**

- ➔ Mehrere Werkzeuge nebenläufig anwendbar
- ➔ Werkzeuge auf mehreren Zielarchitekturen ablauffähig

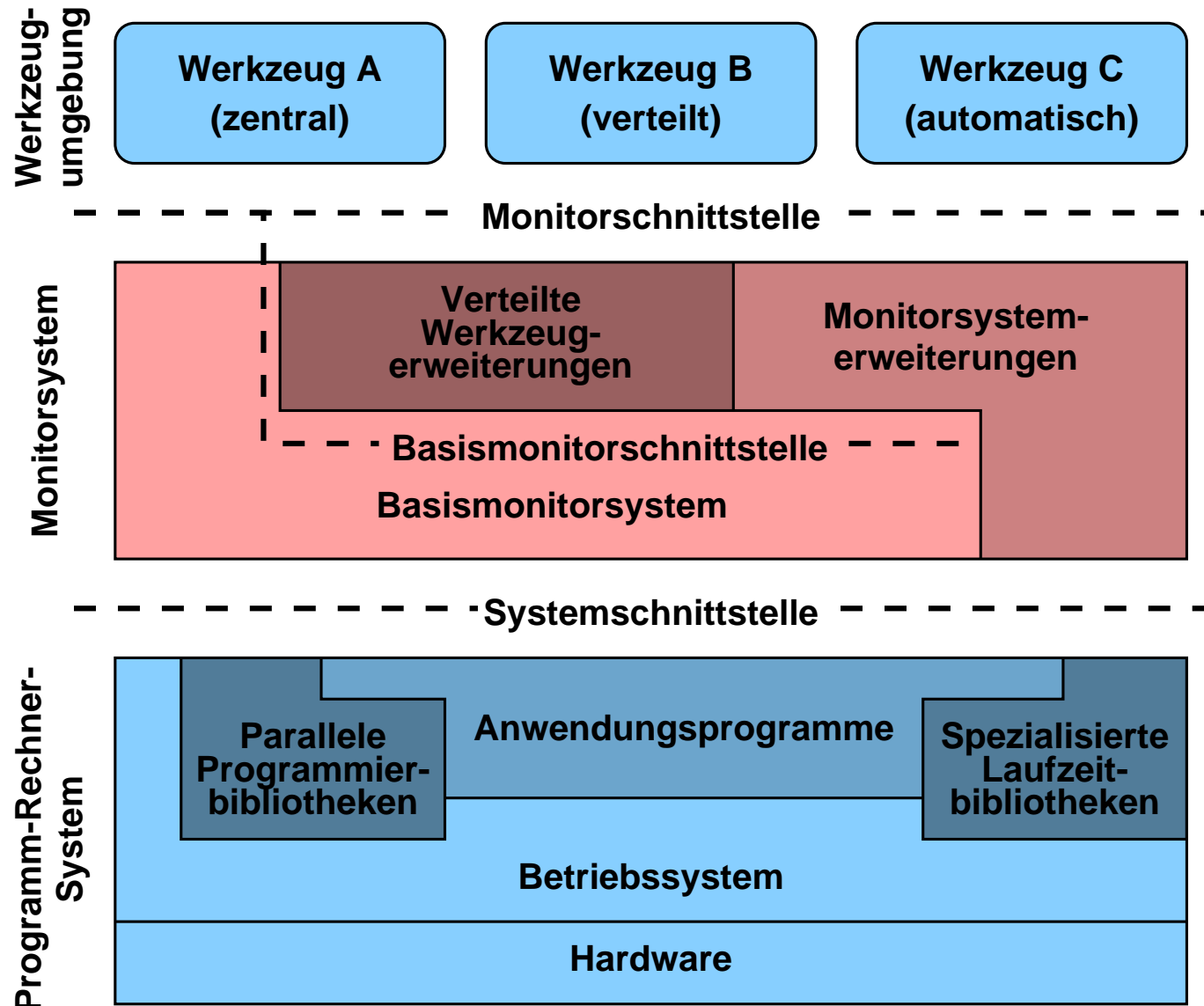
## **Allgemeingültigkeit und Flexibilität**

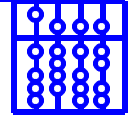
- ➔ Verschiedene Werkzeuge zu unterschiedlichen Zwecken
- ➔ Verschiedene Werkzeugarchitekturen
- ➔ Verschiedene programmiersprachliche Objekte

## **Erweiterbarkeit**

- ➔ Neue Werkzeugkonzepte
- ➔ Neue zu überwachende Objekte

# Das Systemmodell





## Objektklassen

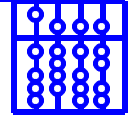
- ➔ System, Knoten
- ➔ Prozesse, Threads
- ➔ Nachrichtenpuffer, Nachrichten
- ➔ Monitorobjekte

## Dienstklassen

- ➔ Informationsdienste (z.B. liefere Liste der Prozesse)
- ➔ Manipulationsdienste (z.B. halte Prozeß an)
- ➔ Benachrichtigungsdienste (z.B. melde, daß Prozeß terminiert)

## Sicht des Werkzeugs

- ➔ Monitorsystem ist paralleler Server, der Dienste an Objekten anbietet

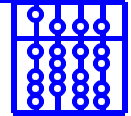


## Arbeitsprinzip: Ereignis/Aktions-Modell

- ➔ **Ereignis** = interessierender Zustandsübergang im überwachten System
- ➔ **Aktion** = erwünschte Beobachtung oder Manipulation des System

## Dienstanforderungen

- ➔ Verhalten des Monitorsystems ist durch Ereignis/Aktions-Relationen bestimmt
- ➔ Programmierung der Relationen über die definierte Schnittstelle
- ➔ Aktionen jeweils ausgelöst, wenn das Ereignis erkannt wird
- ➔ Leere Ereignisdefinition  $\Rightarrow$  unbedingte Aktion



## Syntax der Schnittstellenfunktion

Omis\_reply

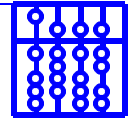
```
omis_request( char * request ,
              void (* callback)(Omis_reply reply, void *param),
              void *param,
              Omis_flags flags )
```

## Syntax der Ereignis/Aktions-Relationen

```
request          ::= [ event_definition ] : action_list
event_definition ::= service_name ( parameters )
action_list      ::= action | action [ ; ] action_list
action          ::= service_name ( parameters )
```

## Semantische Details

- ➔ Dienste arbeiten auf Einzelobjekten und Objektmengen
- ➔ Dienste arbeiten auf Objekten beliebiger Hierarchiestufen
- ➔ Aktionen können potentiell nebenläufig ausgeführt werden



**Gewünschte Information:** Verweilzeit im Sendeaufruf und Anzahl gesendete Datenelemente

## Was ist zu tun

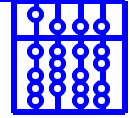
- ➔ Starten und Stoppen einer Stoppuhr
- ➔ Verwendung eines inkrementierenden Zählers

```
thread_has_started_lib_call([p_21],"MPI_Send") :
    pt_integrator_start(pt_i_1) pt_counter_add(pt_c_1,$par5)
```

- ☞ Wenn Prozeß p\_21 einen Sendeaufruf startet: aktiviere einen integrierenden Zähler und addiere die Anzahl Datenelemente (\$par5) auf einen Zähler

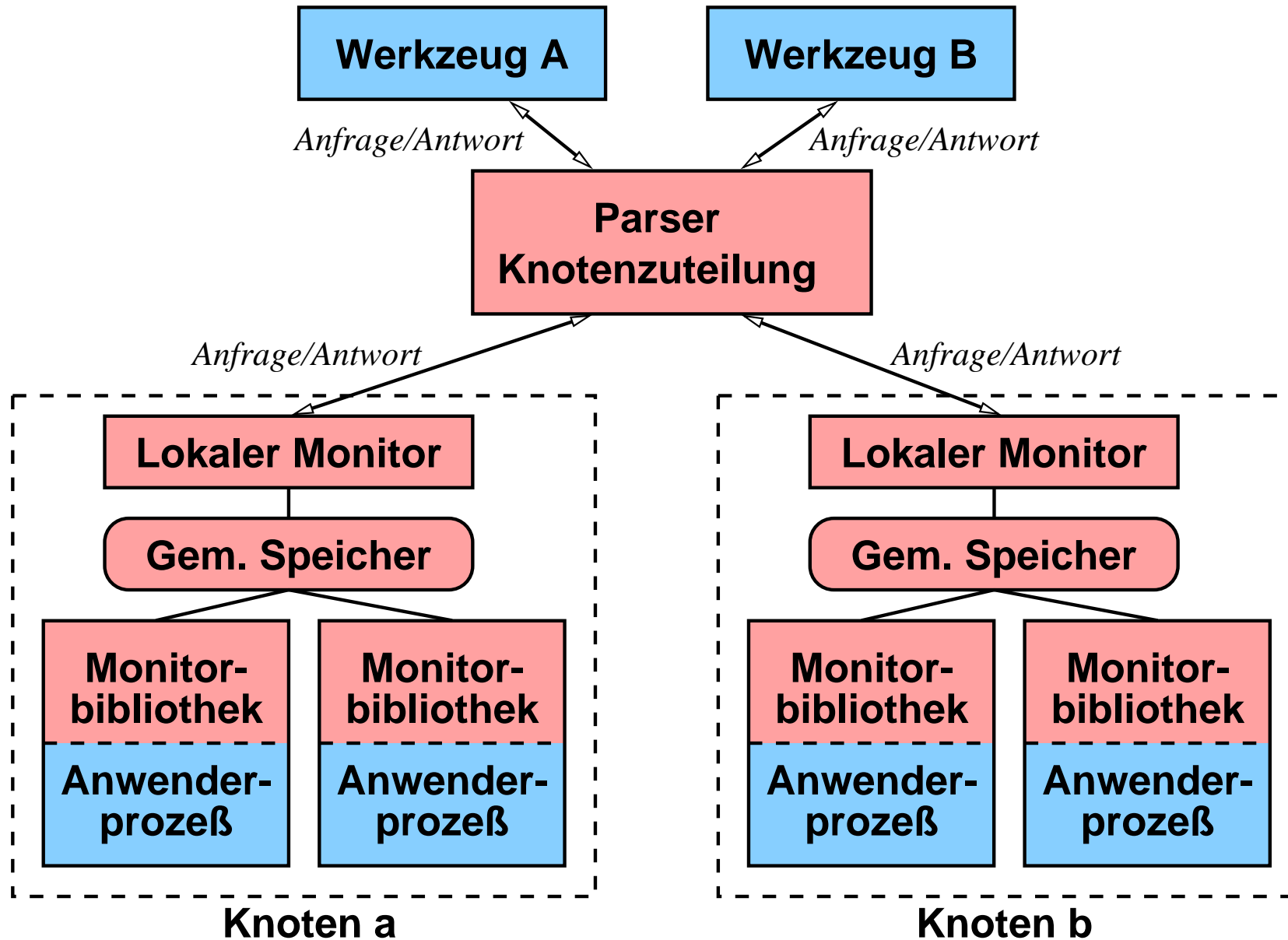
```
thread_has_ended_lib_call([p_21],"MPI_Send") : pt_integrator_stop(pt_i_1)
```

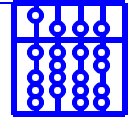
- ☞ Wenn der Prozeß den Aufruf beendet: halte den integrierenden Zähler an



- ➔ Das Monitorsystem wird aus kommunizierenden Monitoren gebildet, jeweils einem pro Rechnerknoten der virtuellen Maschine
- ➔ Globale Aktionsspezifikationen werden zur Definitionszeit an die Knoten verteilt.
- ➔ Kommunikation zwischen Monitoren mittels Nachrichtenaustausch
- ➔ Im ersten Schritt: Zentrales Parsen der Dienstanforderungen
- ➔ Ereignisgetriebenes Ausführungsmodell der (Einzelknoten-)Monitore
- ➔ Ereigniserkennung und Aktionsausführung sowohl im Monitor als auch in der Anwendung
- ➔ Kommunikation zwischen knotenlokalen Monitorkomponenten über gemeinsamen Speicher und Signale

# Grobstruktur des Monitorystems



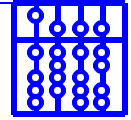


## Einordnung im Bereich der Informatik

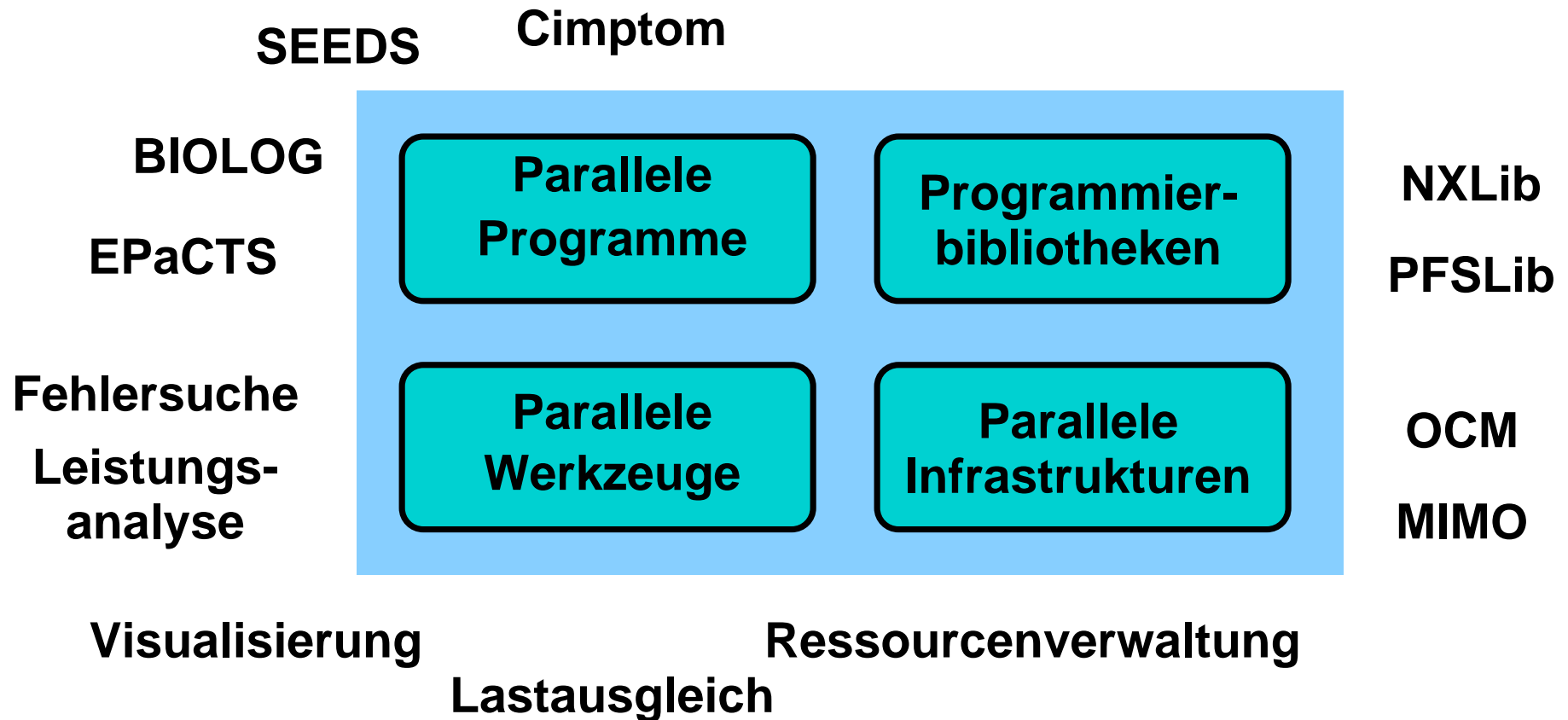
- ➔ Werkzeugentwicklung: Parallele Programmierung
- ➔ Schnittstelle OMIS: Softwareentwurfsmethodik
- ➔ Monitorsystem OCM: Verteilte Betriebssysteme

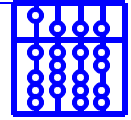
## Ausbau

- ➔ OMIS/OCM für verteilten gemeinsamen Speicher
- ➔ Übertragung der Schnittstellenkonzepte in den Bereich der Middleware-Systeme (CORBA)



- ➔ **Bildgebende Verfahren in der Computertomographie (PET, fMR)**  
 Bildrekonstruktion, Spektralanalyse, statistische Modellierung der aufgenommenen Bilder und Bildreihen  
 Vergleich verschiedener Programmiermodelle und Integration von dynamischem Lastausgleich in die Anwendungsprogramme  
 Graphische Darstellung der gewonnenen Daten über geeignete Benutzerschnittstellen. Einsatz in der klinischen Routine.
- ➔ **Kooperation mit BMW: Parallelisierung von Pamcrash**  
 Simulation einer Kollision mittels Finite-Elemente-Methoden  
 Vergleich verschiedener Programmiermodelle und Integration von dynamischem Lastausgleich  
 Kontrolle des Nichtdeterminismus durch geeignete Online-Werkzeuge (Deterministische Programmausführung)





- ➡ Eine effiziente Methode zur Konstruktion hochparalleler Programme hängt entscheidend von der Verfügbarkeit leistungsfähiger Werkzeuge ab
- ➡ Dies betrifft hauptsächlich die späten Phasen der Programmentwicklung, bei denen der Code bereits ablauffähig ist
- ➡ Parallele Entwicklungswerkzeuge sind selber wieder parallele Programme und/oder stützen sich auf parallele Infrastrukturen ab
- ➡ Ein Ansatz mit einer gemeinsamen Schnittstelle unterstützt die leichter Erstellung solcher Werkzeuge sowie ihre Integration bei der parallelen Programmierung