→ „Vom Problem zur parallelen Lösung“

2.1 Überblick (I)

Übergang vom Problem zur parallelen Lösung wird beeinflusst von:

- Struktur des Problems
- Struktur der zu verarbeitenden Daten

Typ. Vorgehensweise [nach Foster: „Designing and Building Parallel Programs“]

1. Partitionierung
   = Aufteilung des Problems in viele Tasks
2. Kommunikation
   = Spezifikation des Informationsflusses zwischen den Tasks
   und Festlegen der Kommunikationsstruktur
3. Agglomeration
   = Leistungsbewertung (Tasks, Kommunikationsstruktur)
   ggf. Zusammenfassung von Tasks zu größeren Tasks
4. Mapping
   = Abbildung der Tasks auf Prozessoren

2.1 Überblick (II)

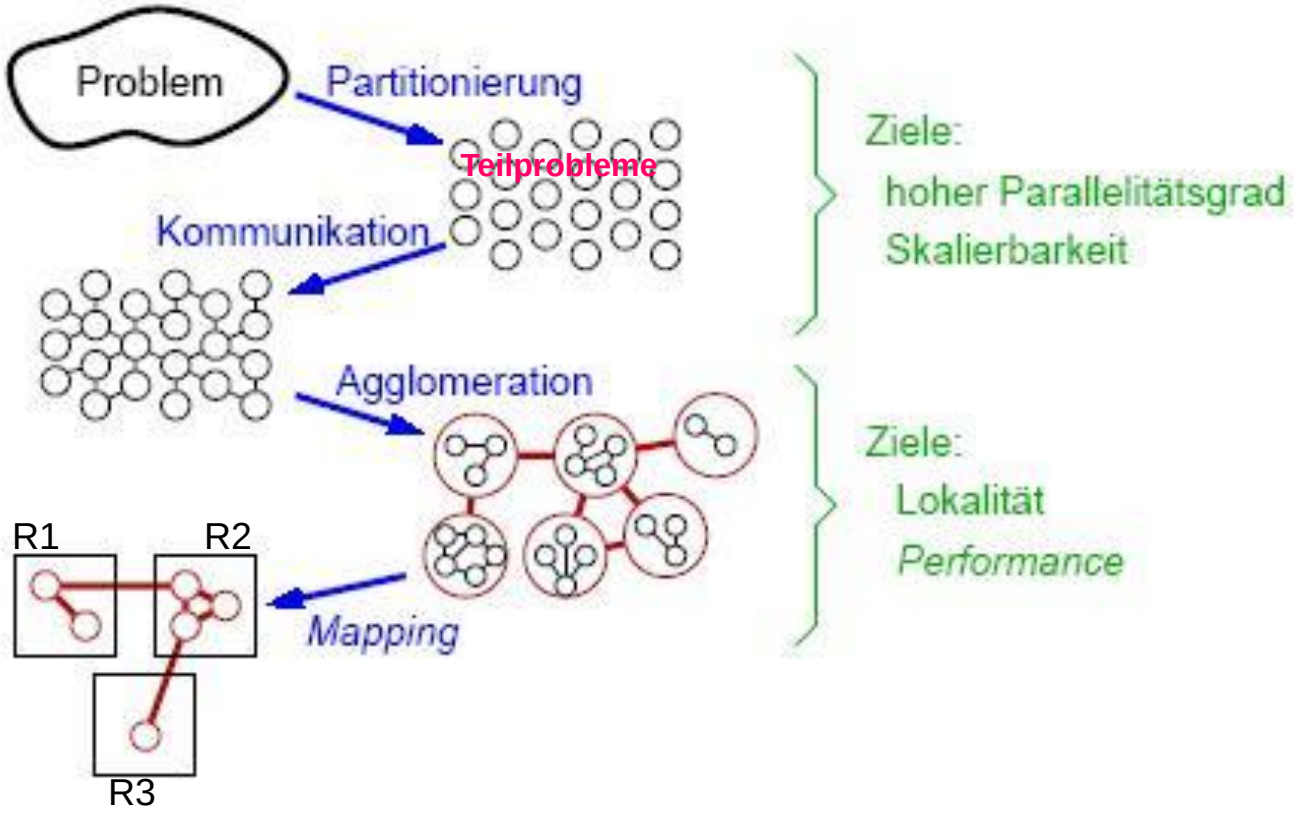


Bild-Quelle: Wismüller: VO Parallelverarbeitung. Univ. Siegen, 2006/07 (angepasst)
Siehe auch: Bengel, u.a.: Masterkurs parallele und verteilte Systeme. Vieweg/Teubner, 2008

2.1 Überblick (III)

Sonderfall: Inhärente Parallelität

- falls das Problem nur aus mehreren voneinander unabhängigen (!) Teilproblemen besteht
- effiziente Lösung auf p Knoten/Prozessoren möglich, indem man (bei n Teilproblemen) jeweils n/p Teilprobleme auf jeden Knoten verteilt und danach parallel berechnet
- dabei ist (fast) keine Kommunikation zwischen den Teilprobl. nötig (nur vor Berechnung verteilen und nach Berechnung einsammeln)
- damit ist fast linearer Speedup möglich, „perfekte Parallelisierung“
- 2 Fälle möglich:
  - Anzahl der Rechenoperationen für die Teilprobleme variiert stark
  - Prozesse können auf unterschiedlich schnellen Knoten laufen Gesamtlaufzeit des parallelen Programms hängt dann bei statischer Lastverteilung vom aufwändigsten Teilproblem bzw. vom langsamsten Prozessor ab

2.2 Partitionierung (I)

Ziel: Aufteilung des Problems in möglichst viele kleinere Teilprobleme, alle Möglichkeiten der Parallelausführung erkennen

Typische Technik: „Teile und Herrsche“ (divide & conquer)

- Problem in mind. 2 (mögl. gleich große) Teilprobleme zerlegen
- rekursive Anwendung der Technik auf die Teilprobleme
- Ende der Rekursion, wenn keine weitere Zerlegung möglich ist
- erlaubt im Idealfall fast linearen Speedup

Zerlegung eines sequ. Problems für eine parallele Lösung durch:

- funktionale Zerlegung
- Datenzerlegung
- Kombination aus beiden

2.2 Partitionierung (II)

a) Funktionale Zerlegung (function decomposition)

- Teilung des Problems in mehrere Arbeitsschritte, Aufgaben oder Funktionen mit Zerlegung des Programmcodes
- Bsp.: Klimasimulationsmodell wird zerlegt in
  - Atmosphären-Modell
  - Hydrologisches Modell
  - Landoberflächen-Modell
  - Ozean-Modell
- Teilfunktionen können ggf. an mehreren voneinander unabh. Teilen des Gesamtproblems parallel arbeiten (kommt evtl. der inhärenten Parallelität nahe)
- falls Teilfunktionen datenabhängig sind, führt das zu einer Pipeline-Verarbeitung, die Teilprobleme (und damit die Stufen der Pipeline) können dabei gleichartig oder verschieden sein für viele gleichartige Fkt. in der Pipeline kann fast linearer Speedup erreicht werden

2.2 Partitionierung (III)

b) Daten-Zerlegung (domain decomposition)

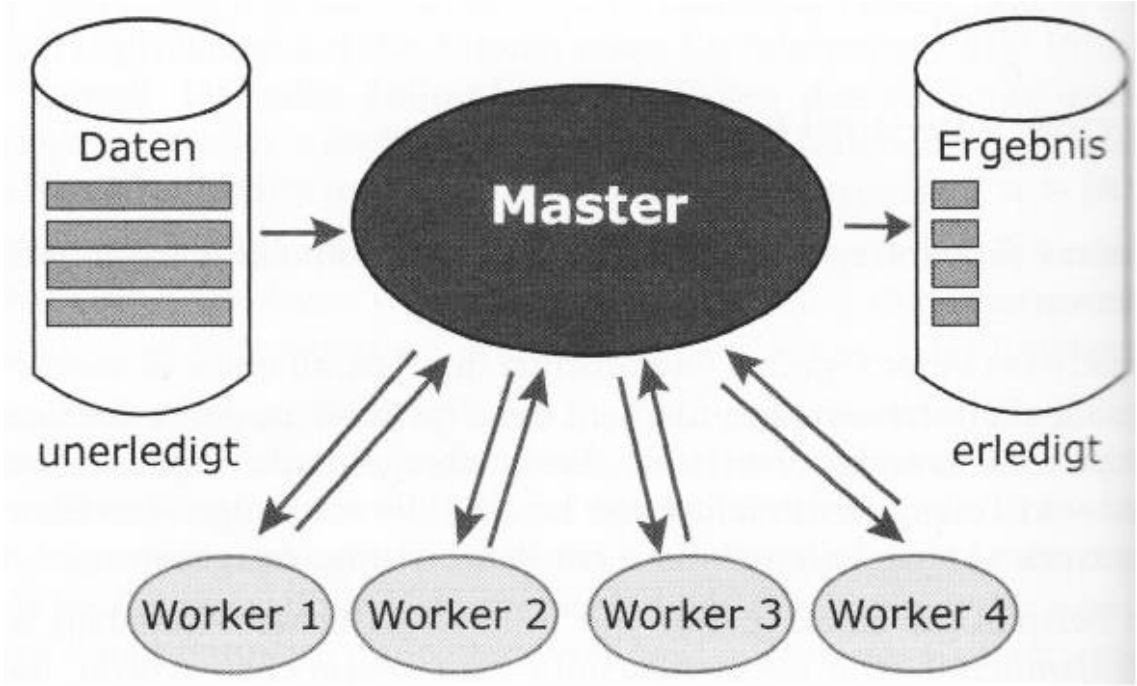
- Zerlegung der Daten (Eingabedaten, Ausgabedaten oder Zwischendaten)
  (mindestens Zerlegung der größten oder am häufigsten benutzten Datenstruktur)
- führt bei einmaliger Zerlegung i.allg. auf ein *„Master-Worker-Schema“,* bei mehrfach rekursiver Zerlegung ergeben sich *Berechnungsbäume*
- Bsp.: 3D–Aufteilung von Raumdaten bzgl. 3D-Gitter

2.2 Partitionierung (IV)

b) Daten-Zerlegung (domain decomposition)

„Master-Worker-Schema“

- Master verteilt versch. Datenbereiche auf Anfrage an mehrere Worker
- Master nimmt Ergebnisse der Worker entgegen



Quelle: Bengel, u.a.: Masterkurs Parallele und verteilte Systeme. Vieweg/Teubner, 2008

2.2 Partitionierung (V)

b) Daten-Zerlegung (domain decomposition)

„Berechnungsbäume“

- Datenzerlegung mehrmals und rekursiv mittels Divide&Conquer
- Bsp.: mehrfache Zerlegung eines zu sortierenden Feldes führt damit zu einem binären Baum
  - Merge-Sort: beim Absteigen im Baum: Feld zweiteilen
    Blätter: paralleles Sortieren der Teilfelder
    nach dem Sortieren: je zwei sort. Teilfelder verschmelzen
  - Quick-Sort: laufende Vertauschung der Feldelemente
    führt evtl. zu unausgeglichenem Baum
  - weitere Bsp.: Summation der Elemente eines Feldes
    lexikalische Analyse einer Zeichenkette
    …

c) Kombination von Funktions- und Daten-Zerlegung möglich

2.2 Partitionierung (VI)

„Checkliste“ zur Partitionierung:

- gibt es mindestens eine Größenordnung mehr Teilaufgaben als parallele Prozessoren? → Flexibilität für weitere Entwurfsschritte
- Alle Teilaufgaben etwa gleich groß?
  → einfacher Lastausgleich mittels Mapping
- Wurden redundante Daten bzw. Berechnungen vermieden?
  (aber ggf. Replikation von Daten sinnvoll, um aufwändige Kommunikation zu vermeiden!)
- Skaliert die Anzahl der Teilaufgaben mit der Problemgröße?
- Wurden verschiedene alternative Partitionierungen untersucht?

2.3 Kommunikation (I)

Ziel: möglichst effiziente Kommunikation zwischen den parallelen Einheiten (Prozesse/Threads),
möglichst Vermeidung von Blockierungen

Festlegung der benötigten Kommunikationsstruktur und Algorithmen
→ wer muss mit wem kommunizieren?
- bei Datenpartitionierung z.T. komplex
- bei Funktionspartitionierung meist einfach

… und Festlegung der Nachrichten
→ welche Daten müssen wann ausgetauscht werden?
- Datenabhängigkeiten beachten, aber unnötige Synchronisation vermeiden, krit. Abschnitte kurz halten

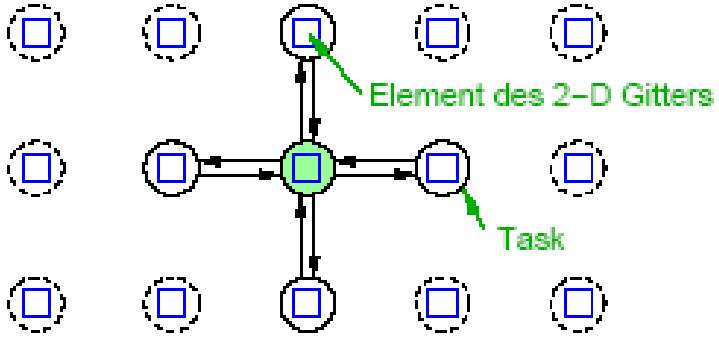
2.3 Kommunikation (II)

Anwendung sehr unterschiedlicher Kommunikationsmuster:

- Lokale oder globale Kommunikation

  lokal: Task kommuniziert nur mit wenigen Tasks (z.B. Nachbarn)
  Bsp: Stencil-Algorithmen (Gauss-Seidel-Verfahren, dig. Bildverarbeitung/Filter)

  global: Task kommuniziert mit vielen Tasks
  Bsp: N-Körper-Problem



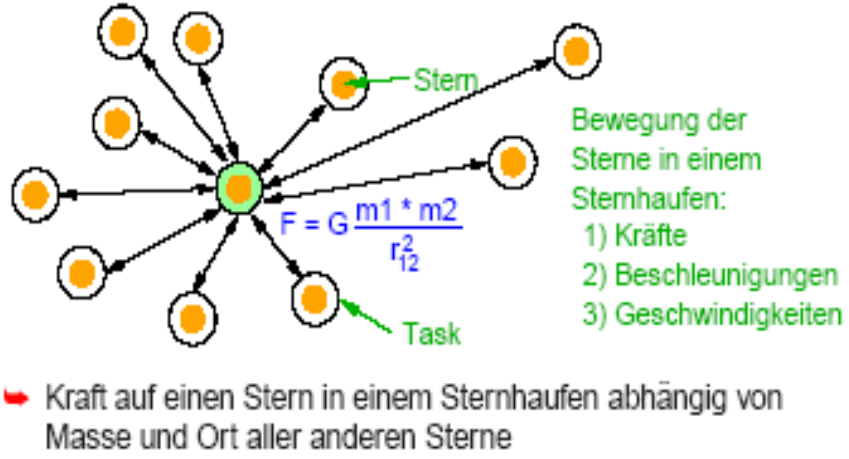


Bild-Quelle: Wismüller: VO Parallelverarbeitung. Univ. Siegen, 2006/07

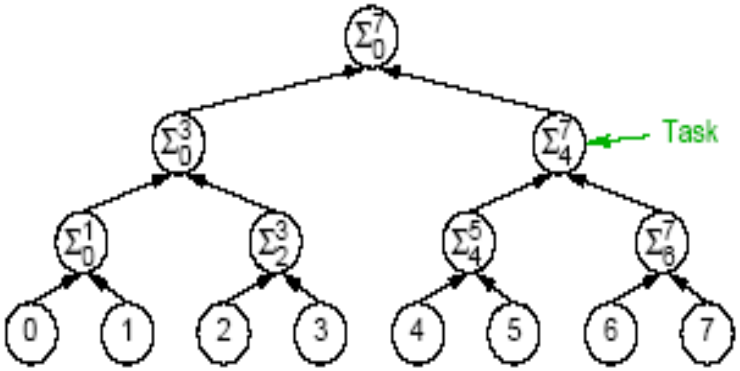
2.3 Kommunikation (III)

Anwendung sehr unterschiedlicher Kommunikationsmuster:

- Strukturierte oder unstrukturierte Kommunikation

strukturiert: regelmäß. Struktur,
z.B. Gitter, Baum
Bsp: Stencil-Algorithmen,
hierarch. Summenbildung

unstrukturiert: unregelmäßig
z.B. Gitterpunkte untersch. eng
Bsp: Schadstoffausbreitung



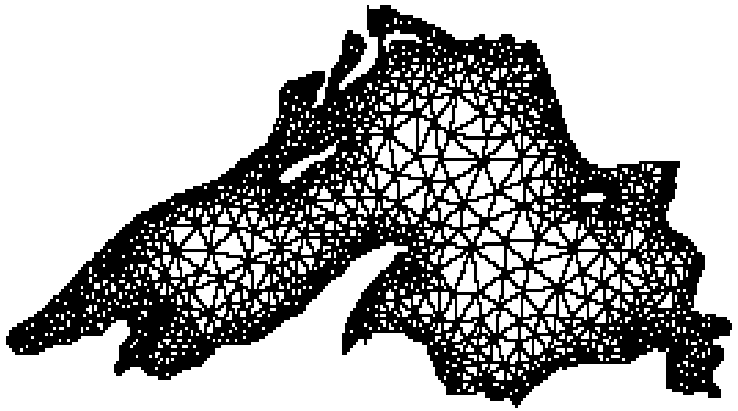


Bild-Quelle: Wismüller: VO Parallelverarbeitung. Univ. Siegen, 2006/07

2.3 Kommunikation (IV)

Anwendung sehr unterschiedlicher Kommunikationsmuster:

- Statische oder dynamische Kommunikation
  - statisch: z.B. Stencil-Algorithmen (verarbeiten Array-Elemente anhand einer Schablone)
  - dynamisch: Kommunikationsstruktur ändert sich zur Laufzeit, abhängig von berechneten Daten
    z.B. N-Körper-Problem: wenn nur nahe Sterne betrachtet werden, aber Sterne bewegen sich, Entfernung ändert sich…

- Synchrone oder asynchrone Kommunikation
  - synchron: z.B. Erzeuger + Verbraucher wissen, wann Kommunikation nötig ist
  - asynchron: Task, die Daten besitzt, weiß nicht, wann andere Task die Daten braucht;
    Lösungen: Polling oder gemeinsamer Speicher

2.3 Kommunikation (IV)

„Checkliste“ zur Kommunikation

- etwa gleich große Anzahl von Kommunikationspartnern pro Task (Teilaufgabe)? → ggf. Verteilung bzw. Replikation von Daten
- Kommunikation nur mit kleiner Anzahl von Nachbarn?
- Können die Kommunikationsoperationen parallel ausgeführt werden?
- Können die Berechnungen der verschied. Teilaufgaben parallel ausgeführt werden?

2.4 Agglomeration (I)

Ziel: In Abhängigkeit von der zur Verfügung stehenden HW (Kosten, Leistungsfähigkeit, CPU-Anzahl)

- Teilaufgaben/Tasks bündeln und/oder
- Daten bzw. Berechnungen replizieren,

um die Kommunikationskosten zu minimieren und trotzdem noch eine genügende Granularität zu bewahren.

Bsp.: Kommunikations/Berechungs-Verhältnis

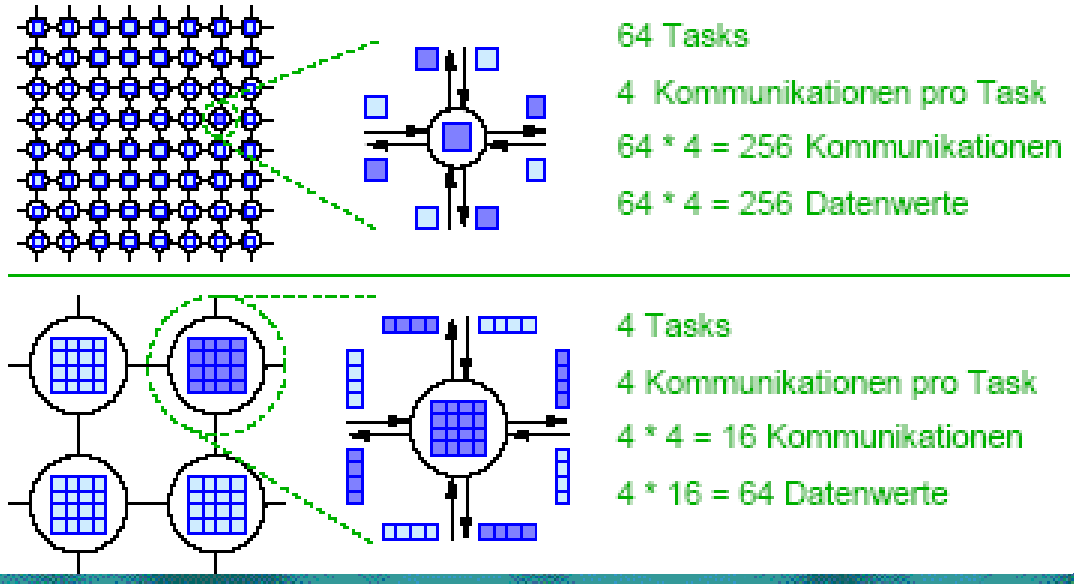
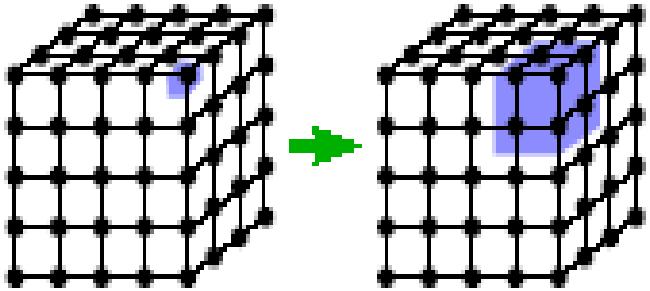


Bild-Quelle: Wismüller: VO Parallelverarbeitung. Univ. Siegen, 2006/07

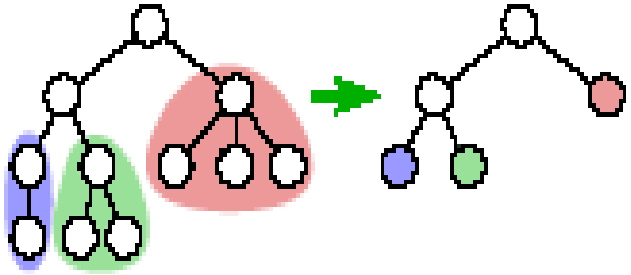
2.4 Agglomeration (II)

Beispiele:

Bild-Quelle: Wismüller: VO Parallelverarbeitung. Univ. Siegen, 2006/07



Kombination benachbarter Tasks



Zusammenfassen von Unterbäumen
bei Divide-and-Conquer-Algorithmen

Tasks zusammenfassen, die

- nicht parallel abgearbeitet werden können
- und miteinander kommunizieren

2.4 Agglomeration (III)

„Checkliste“ zur Agglomeration

- Wurden Kommunikationskosten durch Lokalität verringert?
- Bei replizierter Berechung: überwiegen Vorteile die Kosten?
- Bei replizierten Daten: Skalierbarkeit?
- Tasks mit ähnlichem Rechen- und Kommunikationsaufwand?
- Skaliert die Anzahl der Tasks noch mit Problemgröße?
- Ist das Parallelitätspotential noch ausreichend?
- Ist eine weitere Vergröberung der Tasks möglich?
- Programmieraufwand für Parallelisierung?

2.5 Mapping (I)

Aufgabe: (opt.) Zuweisung von Tasks an verfügbare Prozessoren

Ziel: Minimierung der Ausführungszeit

Strategien:

- parallel ausführbare Tasks auf unterschiedliche Prozessoren legen
  → hoher Parallelitätsgrad
- kommunizierende Tasks auf denselben Prozessor bringen
  → höhere Lokalität (weniger Kommunikation)

Nebenbedingung: Lastausgleich

- (etwa) gleicher Rechenaufwand für jeden Prozessor
- Mapping-Problem ist NP-vollständig ;-(

2.5 Mapping (II)

Mapping-Varianten

a) Statisches Mapping

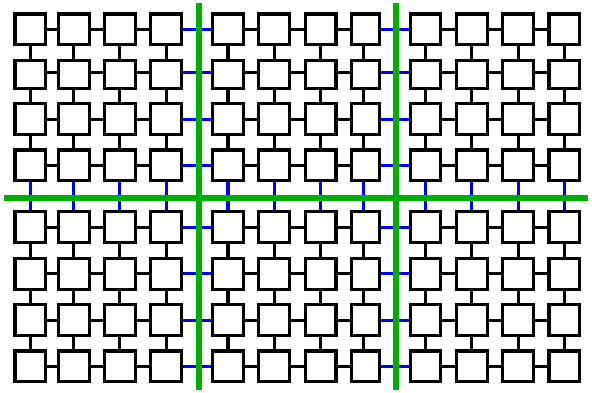
- feste Zuweisung von Tasks zu Prozessoren beim Programmstart
- bei Algorithmen auf Arrays bzw. kartesischen Gittern:
  - Mapping oft manuell festgelegt
  - blockweise Zuteilung bzw. zyklische Zuteilung (kann Lastausgleich verbessern)
- bei unstrukturierten Gittern:
  - Anwendung von Algorithmen zur Graph-Partitionierung, z.B.:
    - Greedy-Algorithmus
    - Recursive Bisection

    …

2.5 Mapping (III)

Beispiele zum statischen Mapping

kartesisches Gitter:
- gleiche Last für jeden Prozessor
- begrenzte Kommunikation nur mit (max.) 4 Nachbarn
- kleine Datenmenge f. Kommunik.



unstrukturiertes Gitter:
- etwa gleich viele Gitterpunkte je Prozessor
- kurze Grenzlinien = wenig Kommunikation

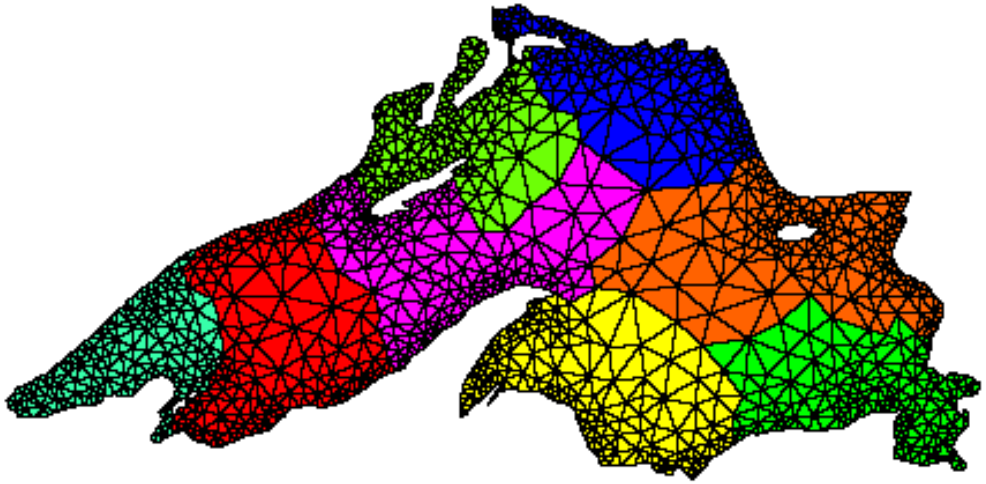


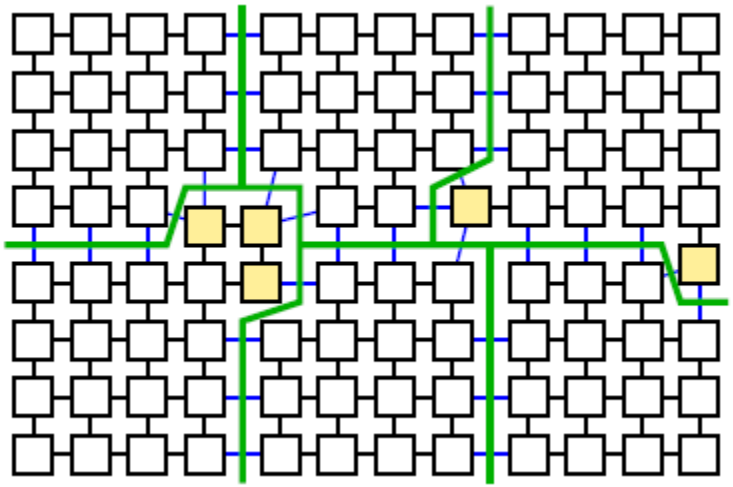
Bild-Quelle: Wismüller: VO Parallelverarbeitung. Univ. Siegen, 2006/07

2.5 Mapping (IV)

b) Dynamisches Mapping (dynamischer Lastausgleich)

- Zuweisung von Tasks an Prozessoren zur Laufzeit
- Varianten:
  - Tasks können während der Ausführung zwischen Prozessoren verschoben werden
  - Tasks bleiben bis zum Ende auf ihrem Prozessor

(vgl. Work-Pool bzw. Master/Slave-Modell)



Bsp.
für einen lokalen Algorithmus:
wenn zur Laufzeit Lastungleichheit festgestellt wird, geben Prozessoren einige Tasks an Nachbarn ab

[Weitere Verfahren später]

Bild-Quelle: Wismüller: VO Parallelverarbeitung. Univ. Siegen, 2006/07

2.5 Mapping (V)

„Checkliste“ zum Mapping

- Vor- und Nachteile von SPMD gegenüber dynamischer Task-Erzeugung abgewogen?
- reicht statisches Mapping oder ist dynamischer Lastausgleich nötig?
  - statisches Mapping: einfach, effizient
  - dynamischer Lastausgleich: oft komplex, teuer
    aber nötig, wenn Anzahl oder Größe der Tasks während der Ausführung variiert (oder bis zur Laufzeit unbekannt ist)
- verschiedene Varianten des Lastausgleichs evaluiert?

2.6 Lastausgleich

2.6.1 Grundlagen (I)

Im Folgenden: Last = Rechenzeit (zwischen zwei Synchronisationen)
Ziel: Zwischen zwei Synchronisationen sollten möglichst alle Prozessoren gleich lange rechnen
Synchronisation hier: z.B.: Programmstart/-ende, Nachrichtenempfang o.ä.

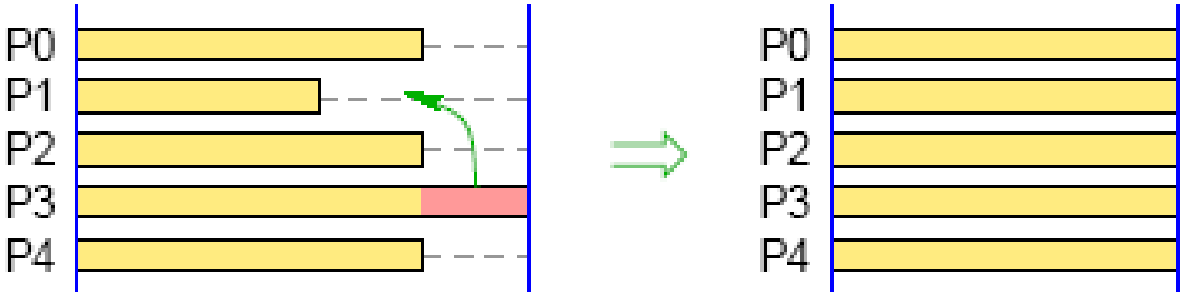


Bild-Quelle: Wismüller: VO Parallelverarbeitung. Univ. Siegen, 2006/07

Hinweise: Lastausgleich ist eines der Ziele von Mapping.
Es gibt auch andere Kriterien zum Lastausgleich, z.B. Kommunikationslast, Gesamtausführungskosten, …

2.6.1 Grundlagen (II)

Woher kommt ungleiche Last?

- ungleicher Rechenbedarf für die einzelnen Teilaufgaben
  z.B. bei Atmosphärenmodell: Teile über/unter Wasser
- verschiedenartige Ausführungshardware
  z.B. unterschiedlich schnelle CPUs
- dynamische Laständerung bei den Teilaufgaben
  z.B. bei Atmosphärenmodell: Einfluss der Tageszeit
- evtl. zusätzliche Hintergrundlast auf den Prozessoren
  z.B. bei PC-Clustern

2.6.1 Grundlagen (III)

Beispiel: Atmosphärenmodell
Kontinente verursachen statisches Lastungleichgewicht
Tag/Nacht-Grenze verursachen dynam. Lastungleichgewicht

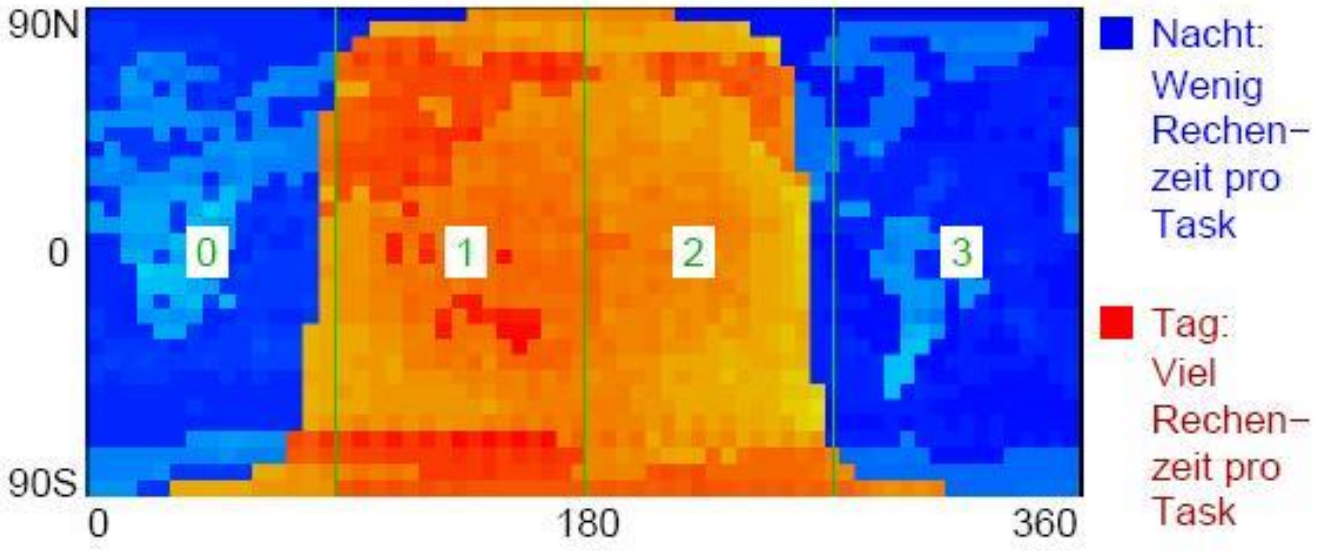


Bild-Quelle: Wismüller: VO Parallelverarbeitung. Univ. Siegen, 2006/07

2.6.1 Grundlagen (IV)

(gewünschte) Eigenschaften von Lastverteilungsverfahren:

- möglichst ohne Vorwissen über die zu verteilenden Aufgaben
- dynam. Verfahren zwecks Berücksichtigung des akt. Zustandes
- schneller Algorithmus
- geringer Overhead für Informationsbeschaffung
- Stabilität: System sollte nicht nur mit Lastausgleich befasst sein
- Skalierbarkeit beim Hinzufügen neuer Knoten/Prozessoren
- Ortstransparenz usw.

Lastverteilung (load sharing):

→ Prozesse so verteilen, dass kein Prozessor unbeschäftigt bleibt (solange noch passende Prozesse warten)

Lastausgleich (load balancing):

→ Geht weiter: Lastverteilung im System ausgeglichen halten

2.6.1 Grundlagen (V)

Globaler Lastausgleich in einem Parallelrechner-/verteilten System:

- statisch (Zuordnung der Last vor der Laufzeit anhand bis dahin bekannter Informationen)
- dynamisch (Zuordnung während der Laufzeit anhand des aktuellen Systemzustandes)
  - zentrale Kontrolle (von einem Rechner aus)
  - dezentrale Kontrolle (verteilt von mehreren Rechnern aus)
    - ohne Prozess-Migration
    - mit Prozess-Migration

2.6.2 Statischer Lastausgleich (I)

Ziel: Tasks *vor* oder *zum* Programmstart so auf CPUs aufteilen, dass Rechenlast der CPUs möglichst identisch ist

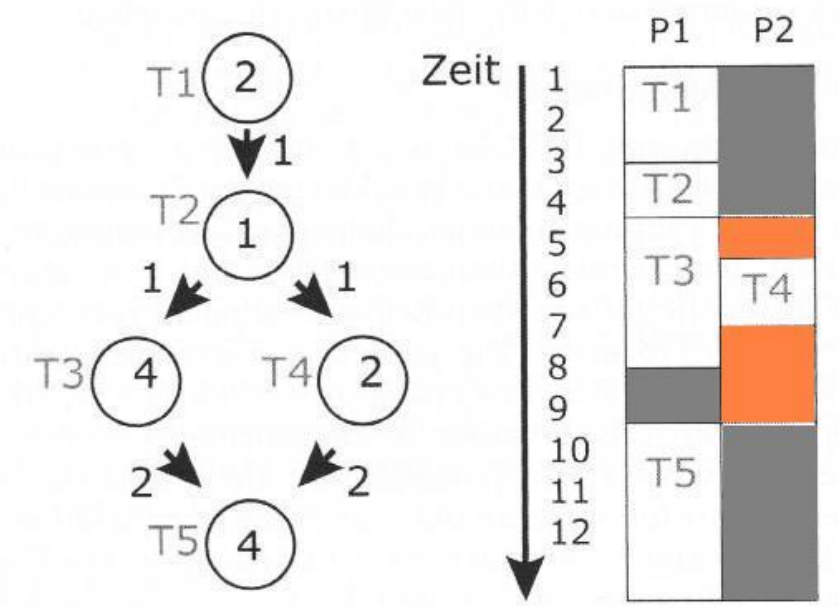
Vorgehensweisen:

a) Berücksichtigung des unterschiedl. Rechenbedarfs der Teilaufgaben bei der CPU-Zuteilung
   - Lösung z.B. mittels (Erweiterung der) Graph-Partitionierung
   - aber Abschätzung des Anforderungsprofils oft schwierig
   - greift nicht bei dynam. Laständerung

b) Feingranulare zyklische oder zufällige CPU-Zuordnung
   - oft gute Lastbalancierung auch bei dynam. Änderung
   - erfordert aber höheren Kommunikationsaufwand

2.6.2 Statischer Lastausgleich (II)

Beispiele zur Graph-Partitionierung
a1) Task-Präzendenzgraph und Gantt-Diagramm (hier: 2 CPUs)



Quelle: Bengel, u.a.: Masterkurs Parallele und verteilte Systeme. Vieweg/Teubner, 2008 (angepasst)

Task-Präzendenzgraph:

Knoten: Prozesse inkl. Dauer

Kanten: Präzedenz inkl. ggf. externer Komm.-zeiten

Gantt-Diagramm:

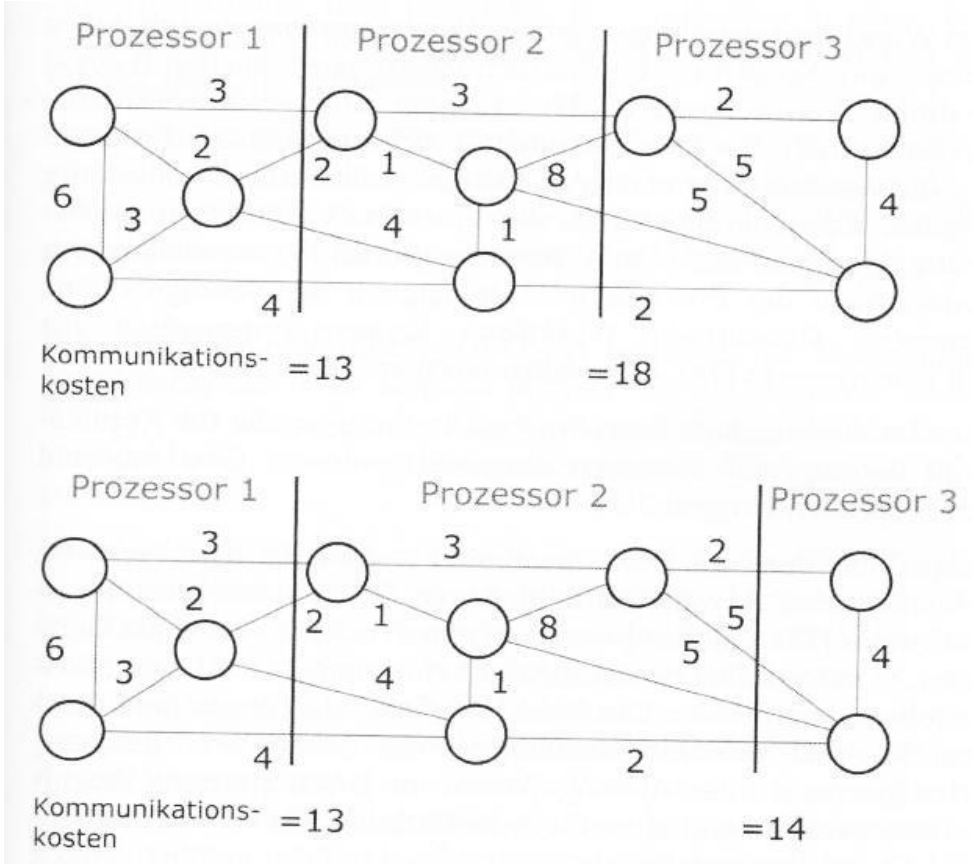
Zuordnung von Task zu CPUs

evtl. Freiräume, weil kein Prozess ready oder externe Komm. nötig

2.6.2 Statischer Lastausgleich (III)

Beispiele zur Graph-Partitionierung

a2) Task-Interaktionsgraph



Quelle: Bengel, u.a.: Masterkurs Parallele und verteilte Systeme. Vieweg/Teubner, 2008

Task-Interaktionsgraph:

zeitl. Ablauf und zeitl. Abhängigkeiten unbeachtet

Knoten = Prozess/Job

Kanten = Komm.-kosten, falls die verbundenen Knoten nicht auf denselben Prozessor zugeteilt werden

Graph-Aufteilung hier nach minimalen Komm.-kosten

Bsp. hier: im unteren Graphen geringere Komm.-kosten, obwohl Aufteilung ungleichmäßig

2.6.2 Statischer Lastausgleich (IV)

Beispiele zur feingranularen zyklischen CPU-Zuordnung

b1) mögliche Formen
- blockweise
- zyklisch
- block-zyklisch

Im Bsp.-Bild:
a) für eindimensionale Felder

b) für zweidimensionale Felder mit streifenweiser Datenverteilung

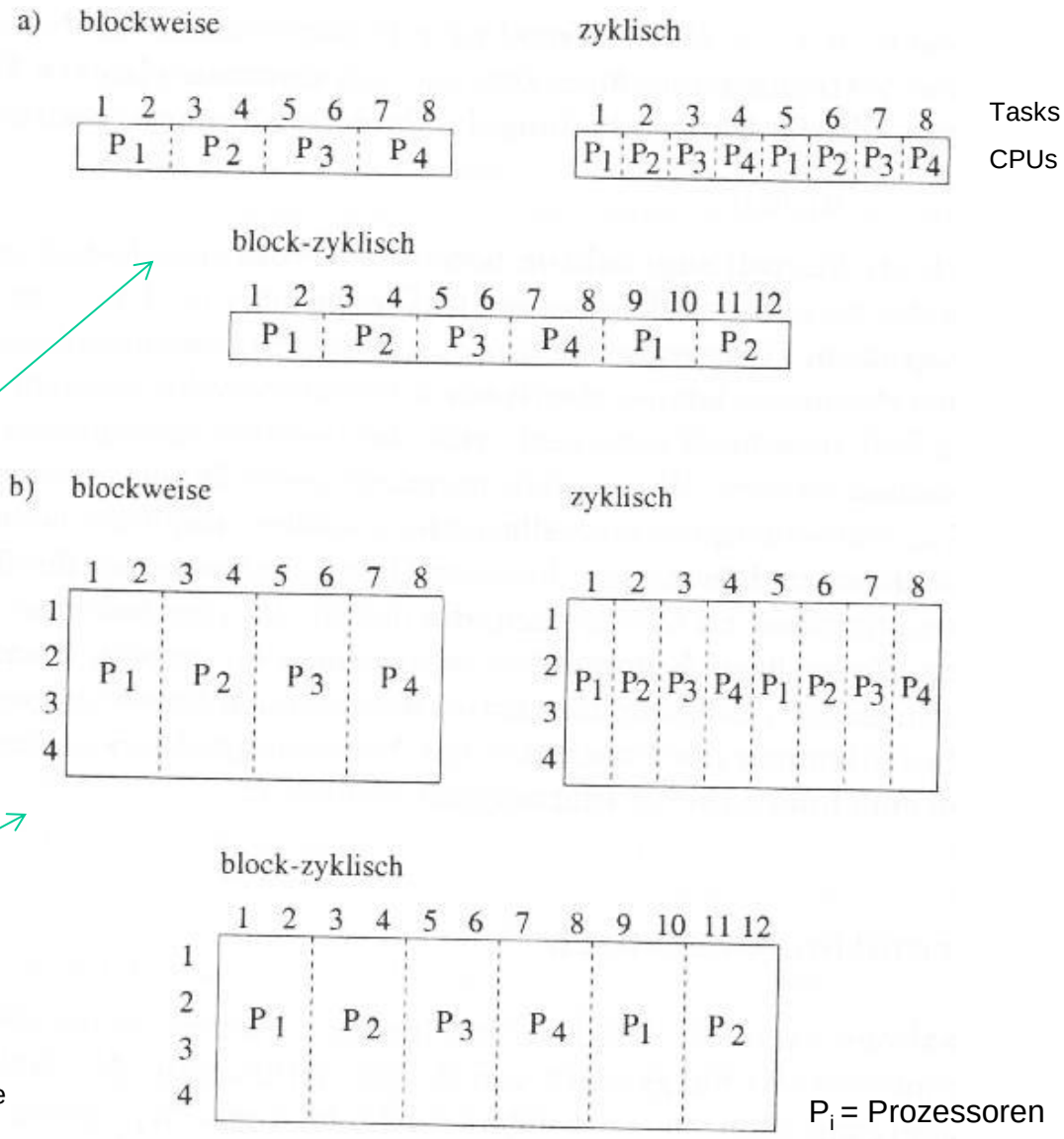
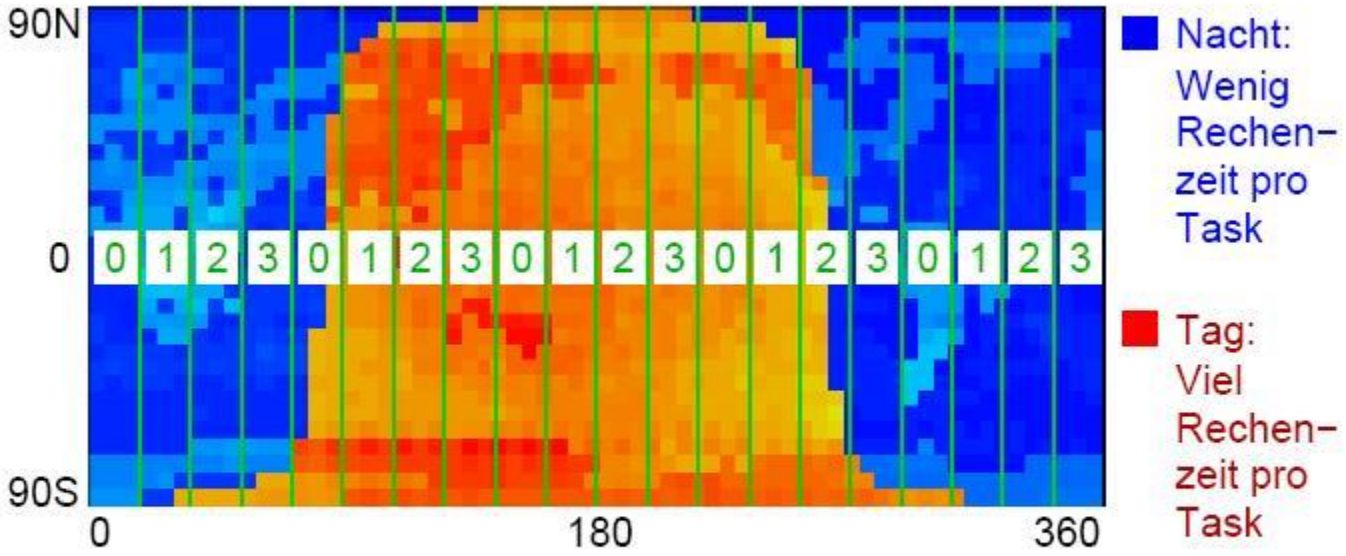


Bild-Quelle: Rauber/Rünger: Parallele und verteilte Programmierung. Springer, 2000

2.6.2 Statischer Lastausgleich (V)

Beispiele zur feingranularen zyklischen CPU-Zuordnung

b2) Bsp. Atmosphärenmodell bei 4 Prozessoren



- Jeder Prozessor bekommt Tasks mit hoher und niedriger Rechenlast

Bild-Quelle: Wismüller: VO Parallelverarbeitung. Univ. Siegen, 2006/07

2.6.3 Dynamischer Lastausgleich

2.6.3.1 Überblick (I)

Im Vordergrund steht nicht mehr das Lastprofil der Anwendungen, sondern die aktuelle Last im Gesamtsystem

Ziel: Verbesserung der Systemleistung durch dynam. Lastverteilung

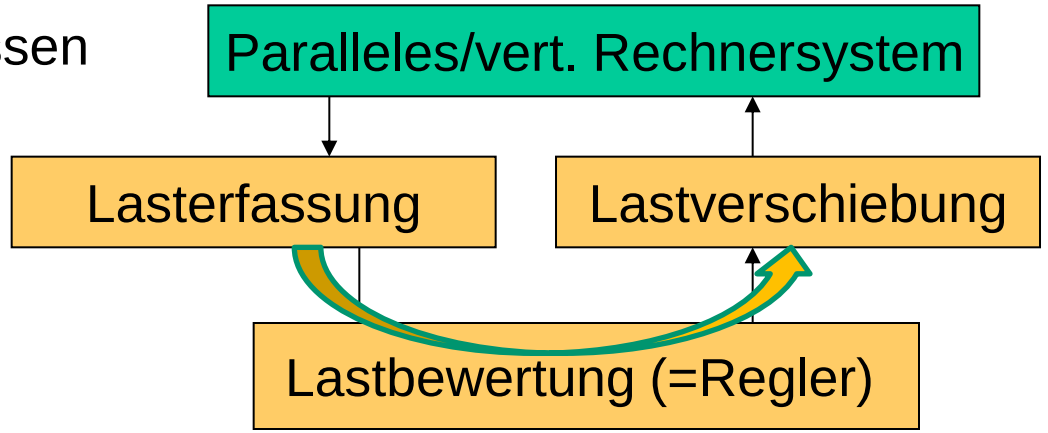
Bsp. für Anwendungsfälle:

- Workstation-Modell:
  - Rechner mehrerer Benutzer sind vernetzt
  - Rechner ist entweder in Benutzung oder idle
  - Lastverteilung soll Prozesse von aktiven Rechnern auf unbenutzte Rechner umlagern (und ggf. wieder von dort weg, falls der Rechner von „seinem“ Nutzer gebraucht wird)
- Prozessor-Pool:
  - Zentraler Server nimmt Aufträge versch. Nutzer entgegen und verteilt sie auf die im Pool verfügbaren Prozessoren/Rechner
  - Nutzer erhält „passende“ Anzahl von Prozessoren

2.6.3.1 Dynamischer Lastausgleich – Überblick (II)

Aufgaben der Lastverteilung (entspricht Regelkreis!):

- Last erfassen bzw. messen
- Last bewerten
- Last verschieben

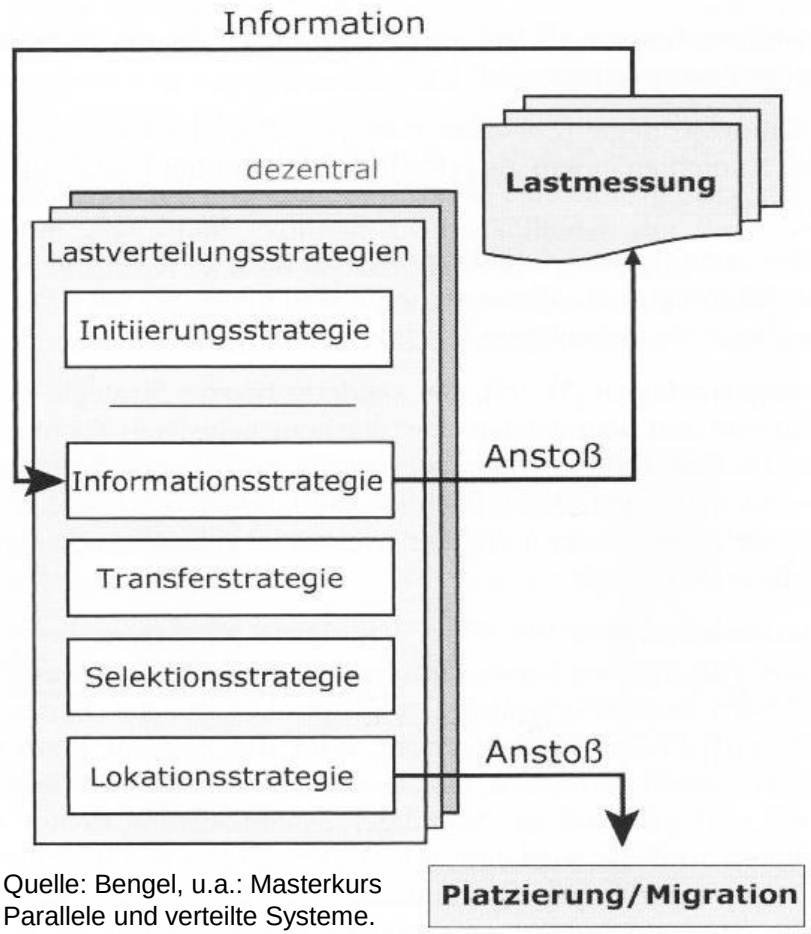


Je nach Problemlage:

- werden Tasks dyn. an CPUs *zugeteilt* und bleiben dort bis zum Ende
  → Ziel: untätige CPUs vermeiden
- werden Tasks ggf. während ihrer Arbeit zwischen CPUs *verschoben*
  → Ziel: gleiche Rechenzeit auf den CPUs zwischen zwei Synchronis.

2.6.3.1 Dynamischer Lastausgleich – Überblick (III)

Architektur eines (dezentralen) Systems zur dynam. Lastverteilung



Quelle: Bengel, u.a.: Masterkurs Parallele und verteilte Systeme. Vieweg/Teubner, 2008

Initiierungsstrategie: welcher Knoten startet die Lasterverteilung (Sender [will Last abgeben] oder Empfänger [will Last aufnehmen]?

Informationsstrategie: wann werden Informationen ausgetauscht, welcher Knoten fordert diese an bzw. gibt sie weiter?

Transferstrategie: Ist Lastausgleich erforderlich?

Selektionsstrategie: welcher Prozess soll transferiert werden?

Lokationsstrategie: welche Knoten sollen als Sender oder Empfänger von Last am Ausgleich teilnehmen?

2.6.3.2 Zentraler dynam. Lastausgleich (I)

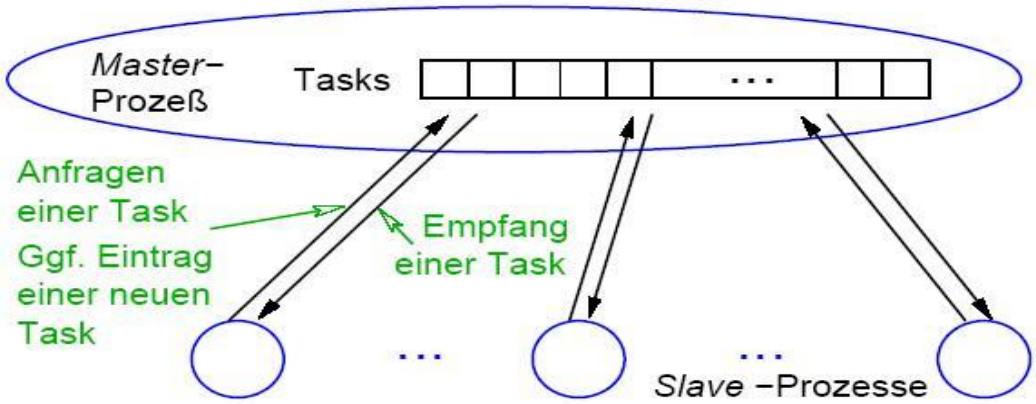
- wird mittels Taskzuteilung durch zentrale Komponente erledigt
- Initiierungsstrategie hier unnötig
- ggf. auch als dedizierter Lastausgleichsserver

Bsp.: Task-Pool-Modell / Prozessor-Pool / Prozessor-Farm
auch: Master-Worker-Schema (vgl. 2.2)

- Master führt sequ. Programmteile aus und verteilt die parallelen Teile an die Worker
- Master hält (Last-)Informationen der Worker zentral
- Initiierung der Lastverteilung durch die Worker, die nach Bearbeitung ereignisgesteuert neue Last vom Master anfordern
- ermöglicht effiziente Entscheidung bzgl. geeigneter Platzierung von Prozessen
- meist periodische Status-Aktualisierung
- Problem: Ausfall des Masters
- schlechte Skalierung

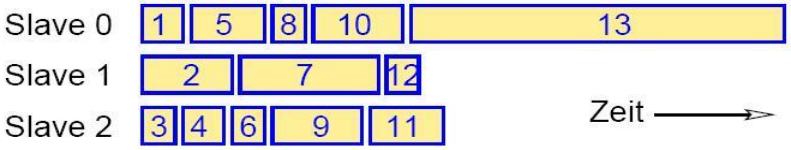
2.6.3.2 Zentraler dynam. Lastausgleich (II)

Beispiel Task-Pool:



Beispiel-Strategie für einen Lastausgleich durch Task-Zuteilung:

- Problemfall: große Task, die am Schluß zugeteilt wird



- Stategie zum bestmöglichen Lastausgleich: teile die jeweils längste Task zuerst zu

Bild-Quelle: Wismüller: VO Parallelverarbeitung. Univ. Siegen, 2006/07

2.6.3.3 Dezentraler dynam. Lastausgleich (I)

- mehrere oder alle Rechner verfügen über Lastbewertung inkl. Lastverteilungsstrategien
  - kooperativ (Lastverteilungseinheiten kommunizieren miteinander, mehr Komm.-Overhead, aber stabiler)
  - nicht-kooperativ (Lastverteilungseinheiten arbeiten autonom)
- Priorisierung bei Zuteilung von Prozessen nötig: Bevorzugung lokaler oder fremder Prozesse (fremde Proz. = günstiger)
- höhere Ausfallsicherheit und Flexibilität als zentrale Verfahren
- aber insges. höhere Systembelastung

2.6.3.3 Dezentraler dynam. Lastausgleich (II)

Beispiel:
Task-Pool bzw. Master/Slave-Modell mit verteiltem Ansatz

- Vermeidung des Engpasses des zentralen Masters durch Hierarchie („Untermaster“) = verteilter Work-Pool
- Bei dynam. Taskerzeugung ggf. Verschiebung zwischen Untermastern nötig

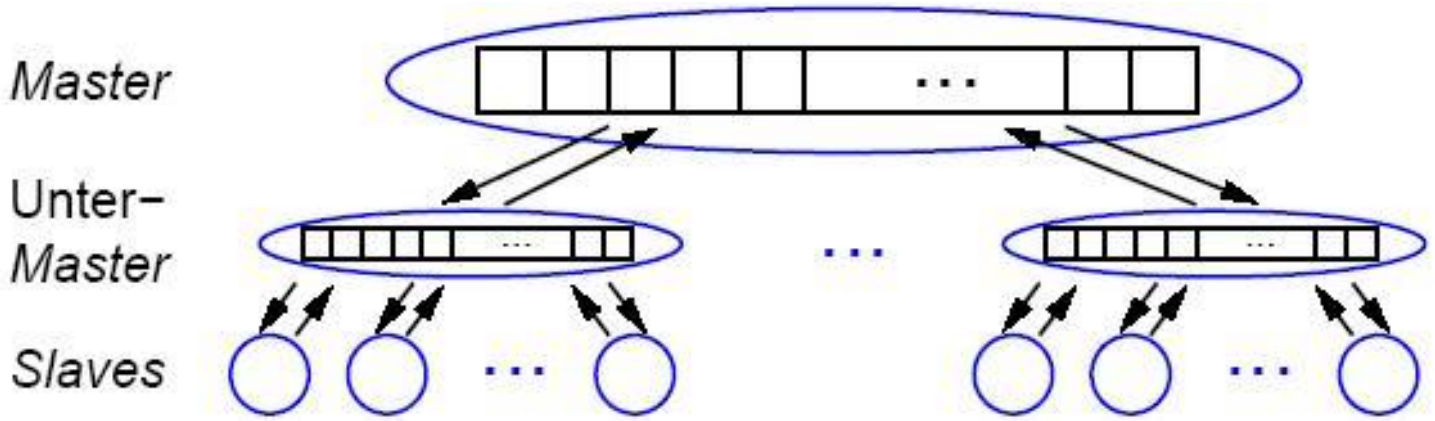
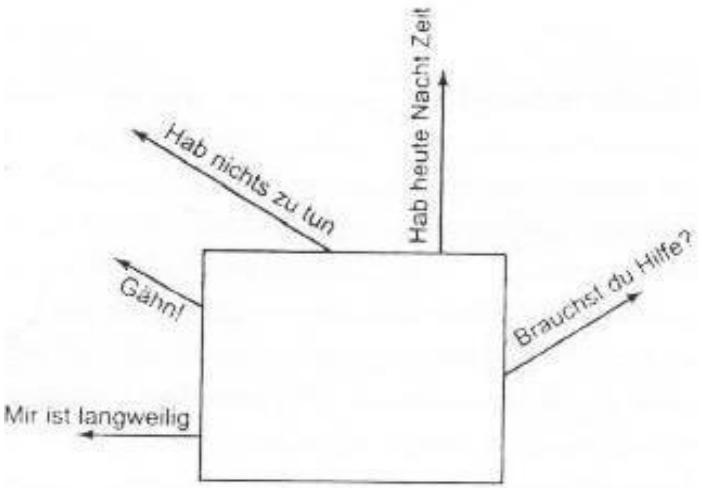


Bild-Quelle: Wismüller: VO Parallelverarbeitung. Univ. Siegen, 2006/07

2.6.3.3 Dezentraler dynam. Lastausgleich (III)

Initiierungsstrategien

a) Empfänger-initiiert
- Untätiger Prozess fordert Last an
- Vorteil: Arbeit wird durch unbelasteten Prozess erledigt
- Nachteil: ggf. zeitw. Untätigkeit
- gut bei hoher Gesamtlast



b) Sender-initiiert
- Überlasteter Prozessor gibt Last ab
- gut bei geringer Gesamtlast

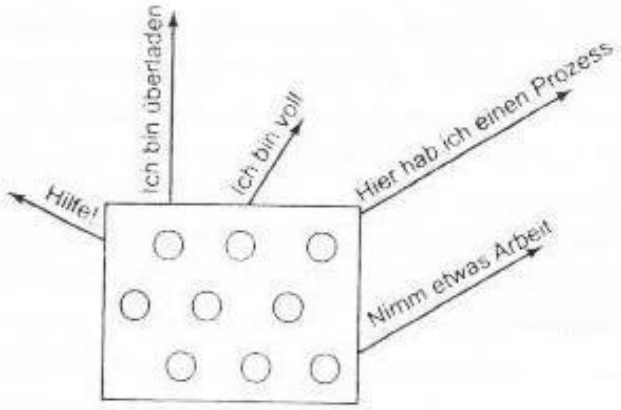


Bild-Quelle: Tanenbaum: Moderne Betriebssysteme. München: Pearson Studium, 2002

2.6.3.3 Dezentraler dynam. Lastausgleich (IV)

Beispiele für weitere Implementierungsentscheidungen:

- Systemkenntnis bzgl. Last:
  - lokal: Prozess kennt nur Lastinformation einiger Nachbarn
    skalierbar, aber evtl. nur langsamer Lastausgleich
  - global: Prozess kennt Lastsituation aller anderen

- Austausch/Verlagerung von Tasks
  - lokal: nur mit Nachbarn
  - global: mit allen Prozessen

- Kriterien zur Auswahl des Partnerprozesses und der zu verschiebenden Task(s)
  - Tasklast
  - Kommunikationsaufwand
  - …

2.6.3.3 Dezentraler dynam. Lastausgleich (V)

Beispiele für Algorithmen zum verteilten dyn. Lastausgleich

- *Tiling*: Zerlegung des gesamten vert. Systems in nicht überlappenden Teilbereiche; innerhalb der Bereiche erfolgt vollst. Lastausgleich, danach leichte Verschiebung der Teilung, so dass die Last langsam durch das gesamte System „wandert“
- *Average Neighbor* (auch: *Diffusion*): Zerlegung in überlappende Teilbereiche (Inseln); ein Rechner in jeder Insel verteilt die Last, wegen Überlappung „diffundiert“ die Last gleichmäßig durch das Gesamtsystem
- *Nearest Neighbor*: nur zwei direkte Nachbarn tauschen Last aus
- Relativ neu: „*ökonomische Verfahren*“ für hochkomplexe parallele und verteilte Rechnersysteme = Wettbewerbsprinzip („Markt“): Agenten für Kunden (= Anwendungen, wollen hohe Performance) und Agenten für Lieferanten (= Computersysteme, wollen Gewinn).

2.6.4 Terminierungserkennung (I)

Problem bei Anwendungen mit dynamischer Task-Erzeugung:

→ wann ist die Berechnung abgeschlossen?

- bei zentralem Task-Pool relativ einfach:
  - Task-Pool ist leer
  - alle Slaves haben neue Task angefragt und warten auf Antwort

- bei verteiltem Task-Pool schwieriger:
  - jeder Prozess hat lokale Terminierungsbedingung erreicht
  - es sind keine Nachrichten mehr zwischen Prozessen unterwegs

  Nachrichten könnten neue Tasks beinhalten

2.6.4 Terminierungserkennung (II)

Ein Algorithmus zur verteilten Terminierungserkennung

- Jeder Prozess hat zwei Zustände: aktiv, inaktiv
- Wenn inaktiver Prozess eine Task empfängt: Sender wird ”Vater“
- Aktiver Prozess bestätigt Empfang einer neuen Task sofort, außer sie kommt vom Vater
- Bestätigung an Vater erst dann, wenn Prozess inaktiv wird:
  - lokale Terminierungsbedingung erfüllt (alle Tasks beendet)
  - alle Bestätigungen für empfangene Tasks versendet
  - alle Bestätigungen für versendete Tasks erhalten
- Berechnung beendet, wenn erster Prozess inaktiv wird

  → erster Prozess: hat initiale Task(s) erzeugt

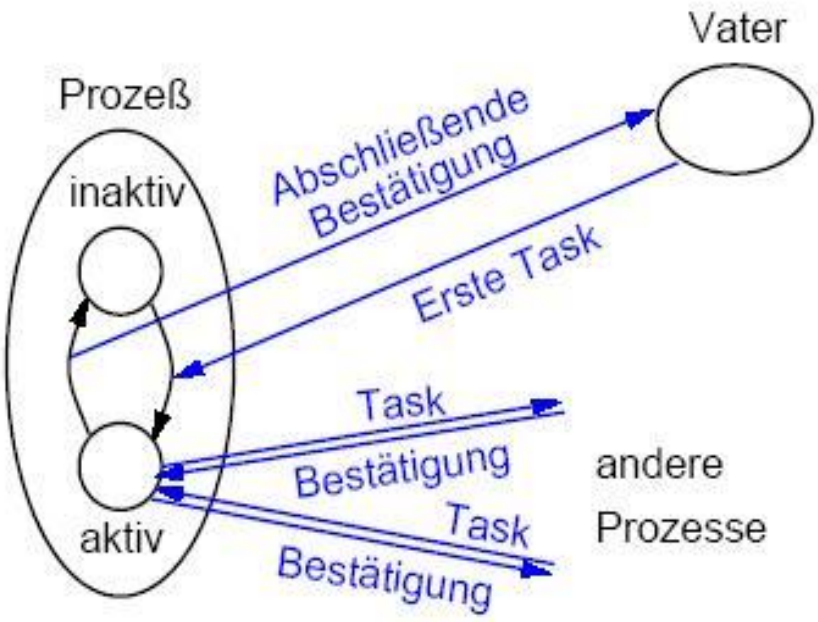


Bild-Quelle: Wismüller: VO Parallelverarbeitung. Univ. Siegen, 2006/07