Paralleles Rechnen I

Einführung in das Hochleistungsrechnen – SS 2012
Design paralleler Algorithmen

Thorsten Grahs, 7. Juni 2012

Design paralleler Algorithmen

- **PCAM-Ansatz**
- **Task-Zuordnung**
- **Dekompositionstechniken**
- **Lastverteilungsverfahren**

Überblick: Design Paralleler Algorithmen

Ian Forster: Designing and Building Parallel Programs

http://www.mcs.anl.gov/ itf/dbpp/

- Ein sequentielles Programm besteht aus einer Folge elementarer Schritte zur Lösung eines Problems.
- Ein paralleles Programm muss zusätzlich Aspekte berücksichtigen:
  – Zerlegung in Teilaufgaben
  – Datenzuweisung/Kommunikation
  – . . .

Ziel: Methodische Vorgehensweise

Oft gibt es für jeden Aspekt mehrere Realisierungsmöglichkeiten, die auf unterschiedlichen Parallelrechnerarchitekturen zu unterschiedlichen Laufzeiten führen können.

Methodologie: Design Paralleler Algorithmen

Foster's Methodology

PCAM-Methode

- **Partitioning**
  Zerlegung in Teilaufgaben
- **Communication**
  Kommunikation/Datenaustausch zwischen den Teilen
- **Agglomeration**
  Zusammenfassung mehrerer ähnlicher Teilaufgaben
- **Mapping**
  Abbildung der Teilaufgaben auf Prozessoren

Partitionierung

Ziele

- möglichst feinkörnige problemabhängige Zerlegung der Berechnung und der Daten in Teilaufgaben ohne Berücksichtigung der zur Verfügung stehenden Prozessoren

  $\Rightarrow$ inhärente Parallelität, Skalierbarkeit
- Bestimmung der maximal vorhandenen Parallelität
- Vermeidung der Duplizierung von Daten/Berechnungen

Partitionierung

Grundtechniken

- Bereichszerlegung (Domain decomposition) $\Rightarrow$ Datenparallelität
- funktionale Zerlegung $\Rightarrow$ Kontrollparallelität

im Beispiel Matrixmultiplikation

- Bereichszerlegung

  Ausgabematrix $\Rightarrow$ n^2 Aufgaben
- funktionale Zerlegung

  arithm. Operationen $\Rightarrow$ n^3 Multip., $n^2(n-1)$ Additionen

Checkliste Partitionierung

- # Teilaufgaben $\gg$ # Prozessoren? $\Rightarrow$ Flexibilität
- keine redundanten Berechnungen
  $\Rightarrow$ Skalierbarkeit Speicheranforderungen? $\Rightarrow$ Skalierbarkeit
- vergleichbare Taskgröße? $\Rightarrow$ Lastausgleich
- Steigt die Anzahl der Tasks mit der Problemgröße?
  (nicht die Größe der Tasks!) $\Rightarrow$ Skalierbarkeit
- alternative Partitionierungen? $\Rightarrow$ Flexibilität

Agglomeration

Ziele

- Minimierung der Kommunikationskosten
  – Zusammenfassung von stark interagierenden Teilaufgaben
- Vergrößerung der Aufgaben
- Verbesserung der Skalierbarkeit

Methoden

- Replikation von Berechnungen
- Überlappung von Kommunikation und Berechnung

Beispiele für Agglomeration

Matrixmultiplikation

- Submatrizen (Teilblöcke) statt einzelner Matrixelemente multiplizieren
- Verhältnis Kommunikationsaufwand/Berechnungsaufwand sinkt

  $\Rightarrow$ gute Skalierbarkeit

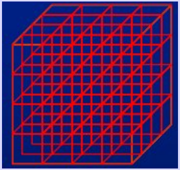
Bei regulären mehrdimensionalen Strukturen, wie Gittern, Würfeln etc. ist eine Agglomeration auf mehrere Weisen möglich.

- Dimensionsreduktion
- Blockaufteilung

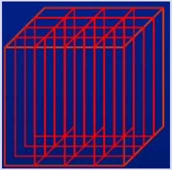
Dimensions/Blockreduktion

Dimensionsreduktion

n^3 Tasks mit
Volumen: 1
Oberfläche: 6



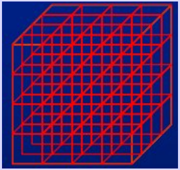
$\rightarrow$



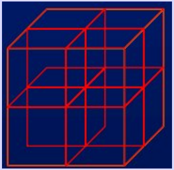
n^2 Tasks mit
Volumen: n
Oberfläche:
4n+2

Blockaufteilung

n^3 Tasks mit
Volumen: 1
Oberfläche: 6

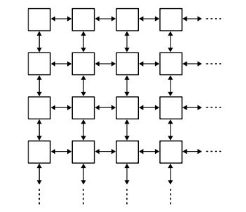


$\rightarrow$



$(n/k)^3$ Tasks:
Volumen: k^3
Oberfläche: $6k^2$

Beispiel 8×8-Gitter



Jeder Knoten verwaltet ein Element und schickt dieses an alle Nachbarn. $64 * 4 = 256$ bidirektionale Kommunikationen von ebenso vielen Datenelementen.

- **Dimensionsreduktion:** 8 Tasks verwalten je 8 Datenelemente
  $8 * 2 = 16$ bidirektionale Kommunikation
  Austausch von $16 * 8 = 128$ Datenelementen

- **Blockaufteilung** (2×2 Gitter)
  $4 * 4 = 16$ bidirektionale Kommunikation
  Austausch von $16 * 4 = 64$ Datenelemente.

Checkliste Agglomeration

Checkliste

- Reduktion der Kommunikationskosten durch Erhöhung der Lokalität?
- Mehraufwand durch Replikation von Daten/ Berechnungen gerechtfertigt?
- Skalierbarkeit?
- Verhältnis Kommunikations-/Berechnungsaufwand?
- Task-Komplexität ausgeglichen?
- weitere Zusammenfassungen?

Mapping

Ziele

- Zuordnung der parallelen Aufgaben zu Prozessoren (rechnerabhängig)
- Platzierung unabhängiger. Tasks auf versch. Rechnern
  (Platzierung häufig komm. Tasks auf denselben Proz.)

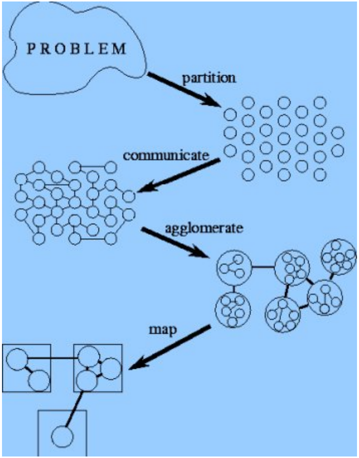
Methoden

- statische Aufgabenverteilung
- dynamische Lastbalancierung (task scheduling)

Design paralleler Algorithmen - Task

Tasks als kleinste Teilaufgaben

- **Partitioning**
  Zerlegen des Problems in Tasks
- **Communication**
  Kommunikation/Datenabhängigkeiten klären
- **Agglomaration**
  Zusammenfassen von verwandten Tasks
- **Mapping**
  Abbilden der Aufgaben auf Prozessoren



Tasks

Definition

- Berechnungseinheiten, die aus der Partitionierung resultieren.
- Kleinste identifizierbare Teilaufgabe der Berechnung

- Durch parallele Ausführung der Tasks wird Beschleunigung erzielt.
- Zwischen Tasks können **Datenabhängigkeiten** bestehen.
  Task benötigt Daten, die ein anderen Task berechnet.

- **Statische Task-Erzeugung**
  – Tasks werden vor Beginn der Berechnung festgelegt
- **Dynamische Task-Erzeugung**
  – Tasks werden (fortlaufend) während der Berechnung erzeugt.
  – Algorithmus legt fest, wann ein neuer Task mit welchen Eigenschaften erzeugt werden soll.

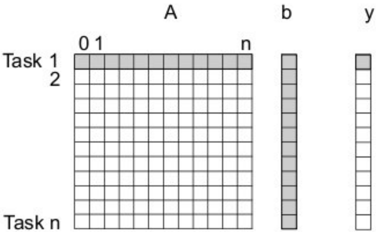
Eigenschaften von Tasks

- Die Größe eines Tasks definiert sich durch seine Berechnungsdauer.
- Oftmals besteht eine Berechnung aus Tasks von sehr unterschiedlicher Größe:
  - Dekomposition erzeugt Tasks mit unterschiedlicher Größe.
  - Größe der Tasks ist a priori nicht bekannt.

Dekomposition ist durch ihre Granularität gekennzeichnet

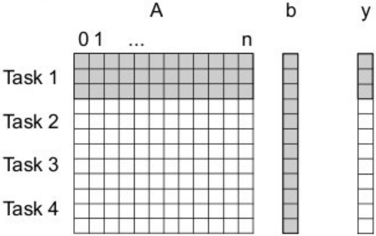
- fein-granular (fine-grained): viele kleine Tasks
- grob-granular (coarse-grained): wenige große Tasks

Beispiel Matrix-Vektor-Multiplikation



$$y[i] = \sum_{j=1}^{n} A[i,j].b[j]$$

Fein–granulare Dekomposition



Grob-granulare Dekomposition

Task-Abhängigkeit

Task-Abhängigkeitsgraph

- Gerichteter, azyklischer Graph
- Knoten repräsentieren Tasks
- Gewicht eines Knotens ist die Größe des Tasks
- Kanten geben Datenabhängigkeiten an

- Datenabhängigkeiten zwischen den Tasks werden durch den Task-Abhängigkeitsgraph angezeigt:
- Der Task-Abhängigkeitsgraph bestimmt die Ausführungs-reihenfolge der Tasks: Ein Task kann dann ausgeführt werden, wenn alle Tasks ausgeführt wurden, die über eingehende Kanten mit ihm verbunden sind.

Eigenschaften von Task-Abhängigkeitsgraphen

- **Maximaler Grad der Nebenläufigkeit**

  Maximale Anzahl an Tasks, die zu einem Zeitpunkt gleichzeitig ausgeführt werden können.

- **Kritischer Pfad**

  Längster vorkommender gerichteter Pfad zwischen Start- und End-Knoten.

  — **Pfadlänge**

    Summe der Gewichte der Knoten entlang des Pfades.

- **Durchschnittlicher Grad der Nebenläufigkeit**
  Verhältnis des Gesamtgewichts der Tasks zur Länge des kritischen Pfads.

Beispiel Datenbank

ID	Model	Year	Color	Price
4523	Civic	2002	Blue	$18,000
3476	Corolla	1999	White	$15,000
7623	Camry	2001	Green	$21,000
9834	Prius	2002	Green	$18,000
6734	Civic	2001	White	$17,000
5342	Altima	2001	Green	$19,000
3845	Maxima	2001	Blue	$22,000
8354	Accord	2000	Green	$18,000
4395	Civic	2001	Red	$17,000
7352	Civic	2002	Red	$18,000

Abfrage

MODEL=„CIVIC“ AND YEAR=„2001“ AND (COLOR=„WHITE“ OR COLOR=„GREEN“)

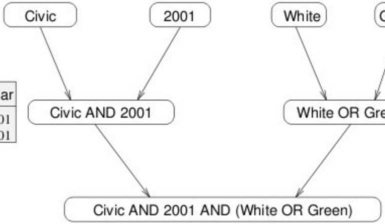
Datenbank Abfrage 1

ID#	Model
4523	Civic
6734	Civic
4395	Civic
7352	Civic

ID#	Year
7623	2001
6734	2001
5342	2001
3845	2001
4395	2001

ID#	Color
3476	White
6734	White

ID#	Color
7623	Green
9834	Green
5342	Green
8354	Green

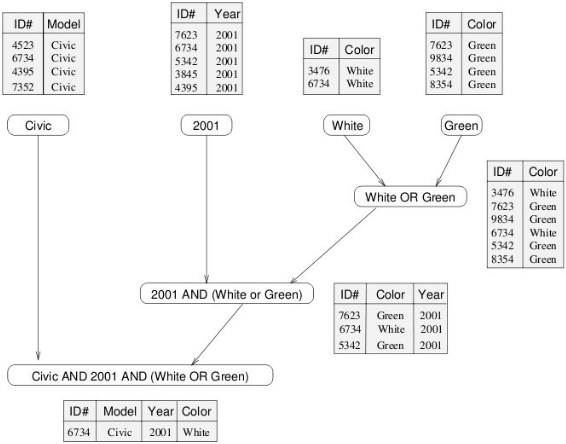


ID#	Model	Year
6734	Civic	2001
4395	Civic	2001

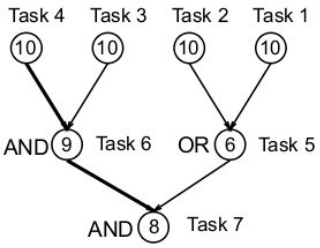
ID#	Color
3476	White
7623	Green
9834	Green
6734	White
5342	Green
8354	Green

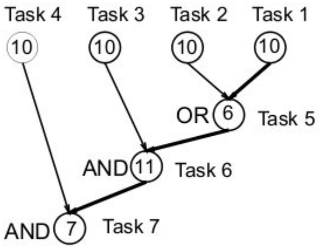
ID#	Model	Year	Color
6734	Civic	2001	White

Datenbank Abfrage 2



Datenbank Abhängigkeitsgraphen





Dekomposition A

- Länge des kritischen Pfads:27
- Durchschnittlicher Grad der Nebenläufigkeit: 63/27=2,33...

Dekomposition B

- Länge des kritischen Pfads:34
- Durchschnittlicher Grad der Nebenläufigkeit:64/34=1,88...

Interaktion zwischen Tasks

Maximal erzielbare Beschleunigung wird bestimmt von

- dem durchschnittlichen Grad der Nebenläufigkeit
  - der Granularität der Dekomposition
  - der Länge des kritischen Pfades
- und der Interaktion der Tasks

Oftmals sind Interaktionen zwischen Tasks nicht im Task-Abhängigkeitsgraph berücksichtigt

- Interaktionen sind abhängig vom Programmiermodell und/oder der Architektur des Parallelrechners.
- Beispiel: der Eingabevektor b bei einer Matrix-Vektor Multiplikation muss allen Prozessen zur Verfügung stehen.

Task-Interaktionsgraph

Repräsentiert Interaktionsmuster zwischen Tasks

- Knoten repräsentieren Tasks
- Kanten zeigen Interaktionen zwischen den Tasks an
- Kantenmenge des Task-Interaktionsgraphs ist eine Obermenge der Kantenmenge des Task- Abhängigkeitsgraphs.

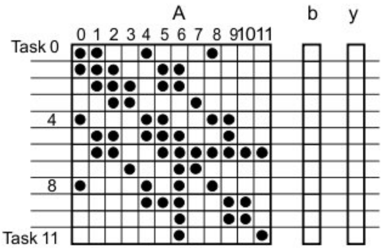
- **Task-Abhängigkeitsgraph**
  erfasst Problem spezifische Aspekte.

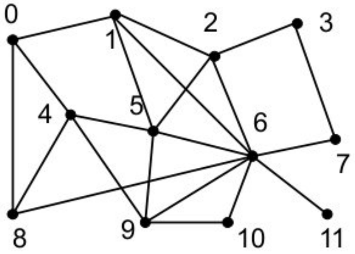
- **Task-Interaktionsgraph**
  erfasst (zusätzlich) Aspekte der Abbildung auf eine konkrete Parallelrechnerarchitektur.

Beispiel: Task-Interaktionsgraph

Sparse Matrix-Vektor Multiplikation

- Dünnbesetzte (sparse) Matrix: viele Einträge sind 0.
- Daten-Dekomposition auf Nachrichten basierter Architektur: Task *i* berechnet y[i] und speichert A[i,∗] und b[i].
- Task-Abhängigkeitsgraph enthält keine Kanten.





Task-Interaktionsgraph

Eigenschaften von Task-Interaktionen

Statisches Interaktionsmuster

- Interagierende Tasks stehen vor Beginn der Berechnung fest.
- Interaktionen treten zu vordefinierten Zeitpunkten auf.

Dynamisches Interaktionsmuster

- Interagierende Tasks und/oder Zeitpunkte der Interaktion können nicht vorherbestimmt werden.
- Im Message-Passing-Programmiermodell schwierig zu realisieren:
  - Sinnvolle Platzierung von send/recieve Paaren schwierig.
  - Zusätzliche Synchronisation oder Polling erforderlich.

Eigenschaften von Task-Interaktionen

Reguläres Interaktionsmuster

- Struktur des Interaktionsmusters kann für effiziente Implementierung genutzt werden.
- Interagierende Tasks werden so auf Prozesse abgebildet, dass sie effizient kommunizieren können.
- Beispiel: Sparse Matrix-Vektor Multiplikation, bei der die von 0 verschiedenen Elemente der Matrix ein Muster aufweisen

Irreguläres Interaktionsmuster

- Interaktionsmuster weist keine verwertbare Struktur auf.
- Beispiel: Sparse Matrix-Vektor Multiplikation, bei der die 0 Elemente der Matrix zufällig verteilt sind.

Dekompositionstechniken

Ziel

- Zerlegung von Berechnungsaufwand und Daten in kleinste sinnvolle Teilaufgaben
- Verbunden mit der Identifikation der Tasks
- Identifizieren von Möglichkeiten der parallelen Ausführung
  (fein-Granulare Zerlegung des Problems)
- Maximaler Speedup bei paralleler Ausführung

Fokus der Dekomposition

Schwerpunkt bei der Zerlegung

- **Daten**

  $\Rightarrow$ Gebietszerlegung (domain decomposition)

  – Zerlegung der Daten

  – anschließende Verknüpfung mit Berechnungsvorschriften (Datenaustausch)

- **Berechnung**

  $\Rightarrow$ Funktionale Zerlegung

  – Zerlegung des Berechnungsaufwand

  – anschließende Verknüpfung der Tasks mit den Daten

Gebietszerlegung

Sichtweise auf das zu berechnende Gebiet, bzw. die aus dem Berechnungsalgorithmus resultierenden Daten (Matrix)

- Zerlege die Daten in Blöcke von $\equiv$ gleicher Größe
- Unterteilung induziert Dekomposition des Problems in verschiedene Tasks
  (Berechnungsoperationen, welche auf den Datenblöcken operieren)
- Partition (Daten, Operationen) bestimmen die Menge der Tasks
- Datenaustausch/Kommunikation, sofern Daten von (benachbarten) Taks benötigt werden
- Domain Decomposition typisch für Probleme mit großen zentralen Datenstrukturen

Zerlegung der Ausgangsdaten

Owner-Computer-Regel

Ein Task führt alle Berechnungen auf dem ihm zugewiesenen Datenbereich aus.

- Partitionierung kann auf Eingabe-, Ausgabe- und/oder Zwischen-Datenstrukturen vorgenommen werden.
  – Kombination führt oft zu fein-granularer Dekomposition
- Partitionierung der Ausgangsdaten anwendbar, wenn die Elemente der Ausgabedatenstruktur unabhängig voneinander berechnet werden können.
- Owner-Computes Regel bedeutet hier: Jeder Task berechnet einen Teil der Ausgabe.
- Eine Partitionierung der Ausgabedaten kann zu unterschiedlichen Dekompositionen in Tasks führen

Beispiel Gebietszerlegung

Matrixmultiplikation

Formulierung als Blockoperation auf $2 \times 2 Blcken$

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Task resultierend aus Blockzerlegung der Matrix C

- **Task 1:** $C_{11} = A_{11}B_{11} + A_{12}B_{21}$
- **Task 2:** $C_{12} = A_{11}B_{12} + A_{12}B_{22}$
- **Task 3:** $C_{21} = A_{21}B_{11} + A_{22}B_{21}$
- **Task 4:** $C_{22} = A_{21}B_{12} + A_{22}B_{22}$

Beispiel Gebietszerlegung

Resultierende Tasks nicht notwendig eineindeutig

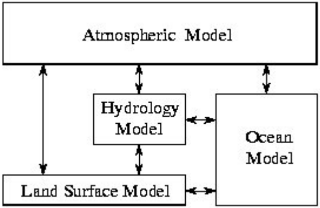
Zerlegung 1

Task 1: $C_{11} = A_{11}B_{11}$
Task 2: $C_{11} = C_{11} + A_{12}B_{21}$
Task 3: $C_{12} = A_{11}B_{12}$
Task 4: $C_{12} = C_{12} + A_{12}B_{22}$
Task 5: $C_{21} = A_{21}B_{11}$
Task 6: $C_{21} = C_{21} + A_{22}B_{21}$
Task 7: $C_{22} = A_{21}B_{12}$
Task 8: $C_{22} = C_{22} + A_{22}B_{22}$

Zerlegung 2

Task 1: $C_{11} = A_{11}B_{11}$
Task 2: $C_{11} = C_{11} + A_{12}B_{21}$
Task 3: $C_{12} = A_{12}B_{22}$
Task 4: $C_{12} = C_{12} + A_{11}B_{12}$
Task 5: $C_{21} = A_{22}B_{21}$
Task 6: $C_{21} = C_{21} + A_{21}B_{11}$
Task 7: $C_{22} = A_{21}B_{12}$
Task 8: $C_{22} = C_{22} + A_{22}B_{22}$

Funktionale Zerlegung



Zerlegung nach funktionalem Zusammenhang

- Modell (PDEs) für Ozeanmodell
- Modell (PDEs) für Atmospärensimmulation
- ...
- Kopplung der Modelle ergibt Kommunikationsbedarf

Rekursive Zerlegung

Divide-and-Conquer Schema

Soll Nebenläufigkeit induzieren:

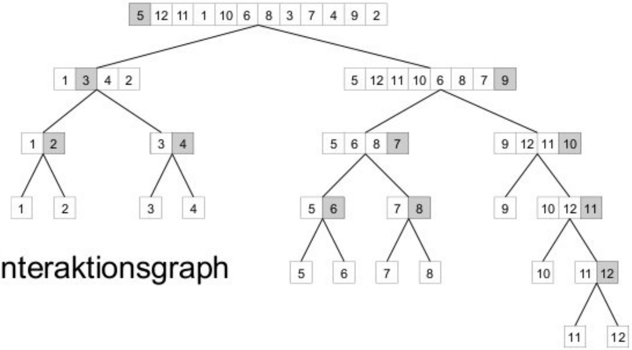
- **Divide-Schritt**

  Aufteilung in Menge von unabhängigen Subproblemen
- **Conquer-Schritt**

  Nach Lösung der Subprobleme werden die Ergebnisse

- Jedes Subproblem wird gelöst, indem es rekursiv weiter unterteilt wird, bis Trivialfall erreicht ist.
- Oftmals können Algorithmen neu strukturiert werden, um sie für rekursive Dekomposition zugänglich zu machen.

Beispiel: Quicksort



Task-Interaktionsgraph

- Dynamische Task-Erzeugung. In der Regel resultieren Tasks mit unterschiedlicher Größe.
- Dynamisches, reguläres, two-way Interaktionsmuster

Lastverteilung – Overhead

Quellen von Overhead bei Parallelisierung

- Overhead durch Task Interaktion
  – Latenzzeiten
  – Beschränkte Bandbreiten
- Overhead durch Leerlauf von Prozessen
  Unterschiede in der Größe der Tasks
  Datenabhängigkeiten blockieren die Ausführung von Tasks

Lastverteilung (load balancing)

Optimale Zuordnung von Tasks zu Prozessen

Lastverteilung – Ziele

Ziele der Lastverteilung

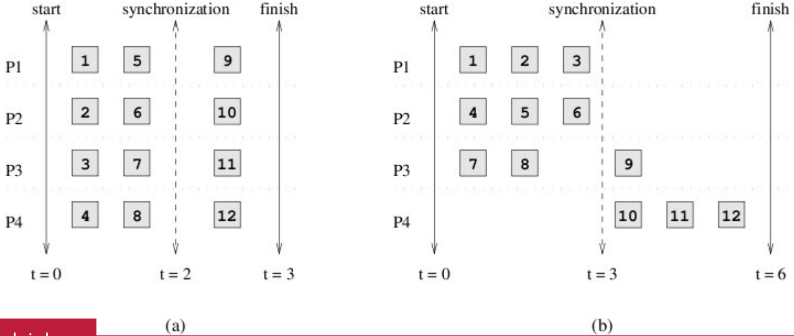
- Minimierung des Overheads bei der parallelen Ausführung der Tasks.
- gleicher work load für alle Prozessoren
  - Minimierung der Zeit für die Task Interaktion
  - Minimierung der Leerlaufzeit von Prozessen

Oft können nicht beide Teilziele gleichzeitig erreicht werden

Beispiel: Vermeiden von Warten/Leerlauf (idling)

Mapping muss auch Leerlauf (idling) vermeiden

- 12 Tasks gleichmäßig auf 4 Prozesse (also 3 pro Prozess) verteilt.
- Datenabhängigkeiten: Tasks 9-12 können erst nach Beendigung von Tasks 1-8 gestartet werden.



Klassifizierung von Lastverteilungsverfahren

Statische Lastverteilungsverfahren

- Die Zuordnung von Tasks zu Prozessen ist vor der Programmausführung bekannt.
- Meistens integraler Bestandteil des Algorithmus.
- Statische Task-Dekomposition erforderlich.

Dynamische Lastverteilungsverfahren

- Tasks werden während der Programmausführung den Prozessen zugeordnet.
- Benötigt zusätzliche Systemkomponente zur Task-Migration.
- Bei dynamischer Task-Dekomposition erforderlich.

Statische vs. dynamische Lastverteilungsverfahren

Statische Verfahren

- Technisch einfacher zu realisieren.
- Erfordern Kenntnis über die Größe der Tasks und die vorkommenden Task-Interaktionen.
- Optimale Zuordnung ist bei unterschiedlicher Task-Größe NP-vollständig, aber es gibt gute Heuristiken.

Dynamische Verfahren

- Erforderlich, wenn die Größe der Tasks stark unterschiedlich und/oder unbekannt ist.
- Oft ineffizient, falls die Übertragungszeit der Tasks im Vergleich zu deren Berechnungszeit groß ist.

Überblick: Statische Lastverteilungsverfahren

- Statische Lastverteilungsverfahren werden meistens im Zusammenhang mit Daten-Dekompositionsverfahren oder Problemen mit statischem Task-Interaktionsgraph verwendet.

- **Lastverteilung mittels Daten-Partitionierung**

  — Blockverteilung

  – zyklische Blockverteilung

  – randomisierte Blockverteilung

- **Lastverteilung mittels Task-Partitionierung**

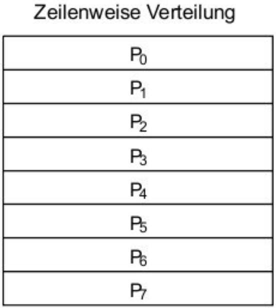
  – Partitionierung des Task-Interaktionsgraphs

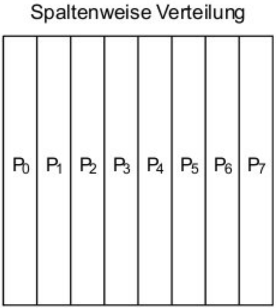
Block-Verteilungsverfahren

- Block-Verteilungsverfahren sind besonders gut geeignet, wenn die Interaktionen der Berechnung hohe Lokalität aufweisen, z.B.
  - alle Elemente lassen sich unabhängig berechnen.
  - die Berechnung eines Elements hängt nur von seinen Nachbarelementen ab.
- Wir betrachten im Folgenden beispielhaft 2-dimensionale Arrays der Größe n x n.
- Beachte: Owner-Computes Regel assoziiert Tasks und Daten
  - Abbildung von Daten zu Prozessen ist hier gleichbedeutend mit Abbildung von Tasks zu Prozessen.

1-dimensionale Blockverteilung

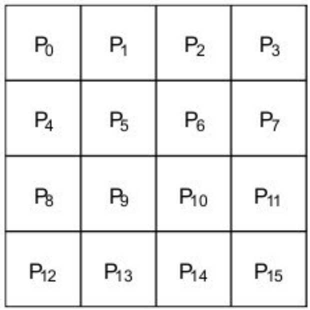
- Jeder Prozess erhält zusammenhängenden Datenblock aus n/p Zeilen bzw. Spalten.
- Beispiel (p=8):

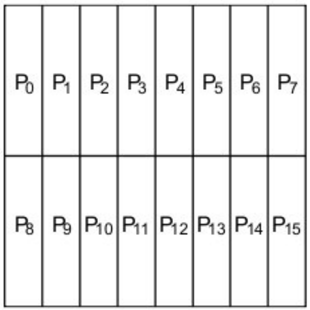




2-dimensionale Blockverteilung

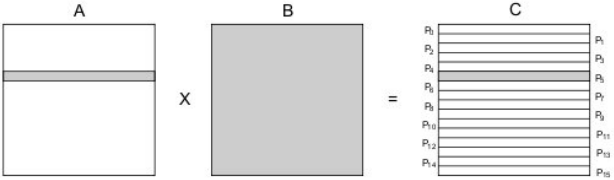
- Jeder Prozess erhält zusammenhängenden Datenblock der Größe $n/p1 \times n/p2$ mit $p = p1 \times p2$
- Beispiel ($p = 4 \times 4$ und $p = 2 \times 8$):

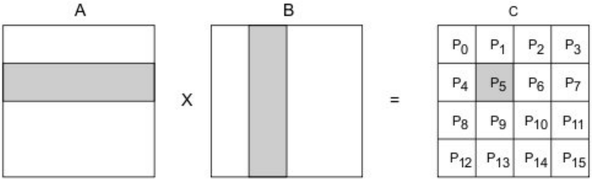




Beispiel: Matrix-Multiplikation

Partitionierung der
Ausgabe-Matrix





Beispiel: Matrix-Multiplikation

- Höher-dimensionale Partitionierung/Verteilung ermöglicht die Verwendung einer größeren Anzahl von Prozessen
  - 1 dimensional: max. n Prozesse
  - 2 dimensional: max. n^2 Prozesse

- Höher-dimensionale Partitionierung/Verteilung reduziert die Anzahl der Interaktionen
  - 1 dimensional:
    - Jeder Prozess greift auf alle Elemente der Matrix B zu
    - Gemeinsamer Datenbereich hat die Größe $O(n^2)$
  - 2 dimensional:
    - Gemeinsamer Datenbereich hat die Größe $O(n^2/\sqrt{p})$

Zyklische Blockverteilung

Problem

Falls die Berechnung der Elemente des Arrays unterschiedliche Zeit erfordert, kann durch Blockverteilung eine ungleichmäßige Lastverteilung resultieren.

Ansatz: Zyklische Verteilung

- Array wird in wesentlich mehr Blöcke partitioniert als Prozesse vorhanden sind.
- Blöcke werden reihum auf Prozesse verteilt, so dass jeder Prozess mehrere nicht-zusammenhängende Blöcke erhält.

Zyklische Blockverteilung

1-dimensional:

αp Blöcke aus

$n/(\alpha p)$ Zeilen/Spalten

mit $1 \leqslant \alpha \leqslant n/p$

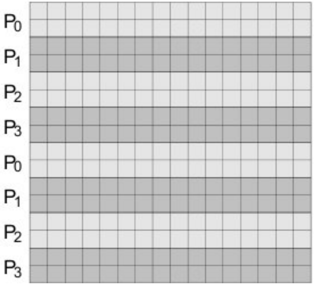
Block b_i wird Prozess $P_{i\%p}$ zugew.

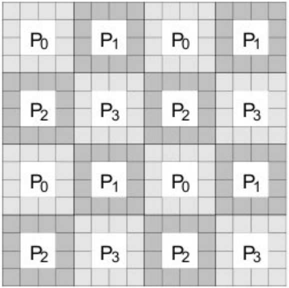
2-dimensional:

$\alpha\sqrt{p} \times \alpha\sqrt{p}$ Blöcke

der Größe $n/(\alpha\sqrt{p})$

mit $1 \leqslant \alpha \leqslant n/\sqrt{p}$





Beispiel LU-Zerlegung

Lösen eines linearen Gleichungssystems Ax=b

Verfahren

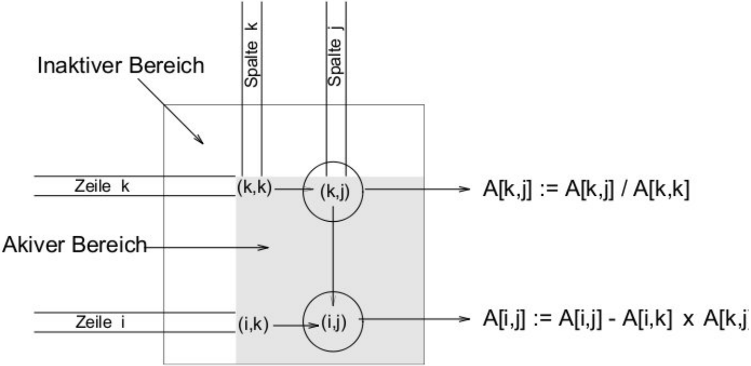
- Bestimme Matrix L und Matrix U mit
  – A=LU
  – L untere Dreiecksmatrix mit Einheitendiagonale
  – U obere Dreiecksmatrix
- Löse zunächst Ly=b und dann Ux=y
  – Lösungen lassen sich „ablesen“ (Dreicksmatrizen)

$$\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 0 & 0 \\ L_{2,1} & 1 & 0 \\ L_{3,1} & L_{3,2} & 1 \end{pmatrix} \bullet \begin{pmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ 0 & U_{2,2} & U_{2,3} \\ 0 & 0 & U_{3,3} \end{pmatrix}$$

Beispiel LU-Zerlegung

```
procedure LU Factorization (A)
begin
   for k := 1 to n do
      for j := k to n do
         A[ j, k] := A[ j, k]/A[k, k];
      endfor;
      for j := k + 1 to n do
         for i := k + 1 to n do
            A[i, j] := A[i, j] - A[i, k] × A[k, j ];
         endfor;
      endfor;
   /* After this iteration, column A[k + 1 : n, k] is logically the kth
   column of L and row A[k, k : n] is logically the kth row of U. */
   endfor;
end
```

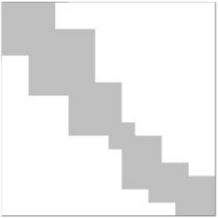
Beispiel LU-Zerlegung



- Der aktive Teil der Matrix ändert sich
- Durch block-zyklische Zuordnung erhält jeder Prozessor verschiedene Teile der Matrix

Randomisierte Blockverteilung

- In manchen Fällen erzeugt auch eine zyklische Blockverteilung eine ungleichmäßige Lastverteilung.
- Beispiel: Prozesse auf Diagonale (P0, P5, P10 und P15) erhalten mehr Tasks als die anderen Prozesse.



P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}	P_{12}	P_{13}	P_{14}	P_{15}
P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}	P_{12}	P_{13}	P_{14}	P_{15}

Lösung

Randomisierte Verteilung: Zufallspermutation der Blöcke.

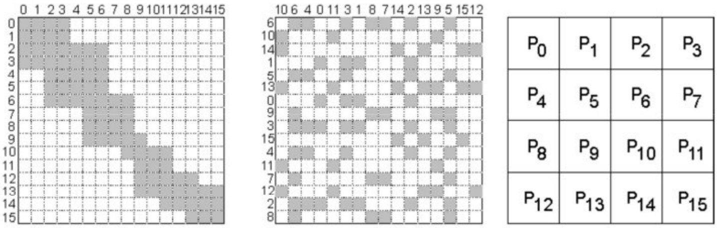
Randomisierte Blockverteilung

$$V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]$$

$$random(V) = [8, 2, 6, 0, 3, 7, 11, 1, 9, 5, 4, 10]$$

$$Zuordnung = \underbrace{8, 2, 6}_{P0} \quad \underbrace{0, 3, 7}_{P1} \quad \underbrace{11, 1, 9}_{P3} \quad \underbrace{5, 4, 10}_{P4}$$

Beispiel



Lastverteilung durch Task-Partitionierung

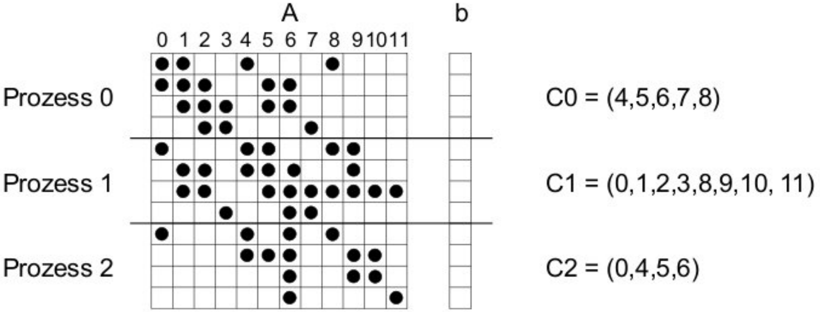
- Lastverteilung durch Task-Partitionierung beruht auf Partitionierung des Task-Interaktionsgraphs mittels Graphpartitionierungsverfahren.
  – Knotenmenge des Graphs soll so in p Teile partitioniert werden, dass
    - alle Partitionen möglichst gleich groß sind (bzgl. Der Summe der Task Größen) und
    - die Anzahl der durchtrennten Kanten minimiert wird.
  – NP vollständiges Problem, es gibt aber gute Heuristiken.

- Statischer Task-Interaktionsgraph erforderlich.

- Task Größe muss bekannt sein.

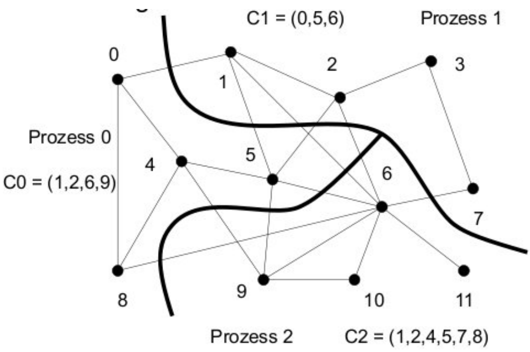
Beispiel: Sparse Matrix-Vektor Multiplikation

- Lastverteilung durch 1D Blockverteilung
- Liste Ci zeigt Interaktionen der Tasks von Prozess i mit Tasks die auf andere Prozesse abgebildet sind.



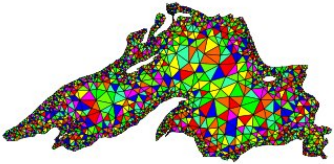
Beispiel: Sparse Matrix-Vektor Multiplikation

- Lastverteilung durch Task-Partitionierung
- Task-Interaktion über Prozessgrenzen ist geringer als bei Blockverteilung

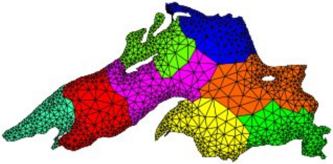


Beispiel: Graphpartitionierung

Diskretisierung: Lake Superior



Random Partitioning



Partitioning for minimum edge-cut.

Dynamische Lastverteilungsverfahren

Überblick

- Dynamische Lastverteilungsverfahren sind erforderlich, falls
  - statische Verfahren zu einer ungleichmäßigen Lastverteilung führen oder
  - der Task-Interaktionsgraph nicht statisch bekannt ist.

- Dynamische Lastverteilungsverfahren können
  - zentral oder
  - verteilt realisiert werden.

Zentraler Ansatz für dynamische Lastverteilung

- Alle ausführbaren Tasks werden in einer zentralen Datenstruktur (Task-Pool) gehalten.

  – Wenn ein Prozess keinen Task zur Ausführung verfügbar hat, entnimmt er einen Task aus dem Task-Pool.

  – Dynamisch erzeugte Tasks werden in den Task-Pool eingestellt.

- Soll die Verwaltung des Pools über einen spezielle Prozess zuständig, so kann dieser dem Master-Prozess zugeordnet werden, die ausführenden Prozesse der Worker-Gruppe.

Zentraler Ansatz für dynamische Lastverteilung

Beispiel: Sortieren der Zeilen einer nxn Matrix

```
for (i=0; i<n; i++)
  sort(A[i], n);
```

- Je nach den Werten der Einträge kann das Sortieren der Zeilen unterschiedlich lange dauern.
  – Statische Zuordnung führt dann zu ungleichmäßiger Lastverteilung.

- Dynamischer Ansatz: **Self-Scheduling** von Schleifen.
  Task-Pool enthält Indizes von noch nicht sortierten Zeilen.
  Prozesse entnehmen Indizes aus Task-Pool und führen den zugehörigen Sortier-Task aus.

Zentraler Ansatz für dynamische Lastverteilung

Problem bei zentralem Ansatz: Schlechte Skalierbarkeit

Bei einer großen Anzahl von Prozessen wird der Zugriff auf den zentrale Task-Pool zum Flaschenhals.

Abhilfe: Chunk-Scheduling

- Es wird pro Anfrage eine Gruppe von Tasks (Chunk) aus dem Task-Pool entnommen.
- Bei vielen Tasks pro Chunk kann wiederum eine ungleichmäßige Lastverteilung auftreten.
- Vermeidung von ungleichmäßiger Lastverteilung durch dynamische Reduzierung der Chunk-Größe zum Ende der Berechnung.

Verteilter Ansatz für dynamische Lastverteilung

Prinzip: Jeder Prozess hat lokalen Task-Pool

Tasks können von anderen Prozessen empfangen, bzw. zu anderen Prozessen geschickt werden.

Parameter

- Wer initiiert einen Task-Transfer?
  – Sender oder Empfänger
- Wann wird ein Task-Transfer vorgenommen?
  – Schwellenwert für Größe des lokalen Task Pools
- Wie werden Sender- und Empfängerprozess gepaart?
  –z.B. round-robin oder randomisiert
- Wie viele Tasks werden auf einmal transferiert?