7 Lineare Algebra

- Allgegenwärtig im wissenschaftlichen Rechnen
- Prozessoren sind darauf optimiert
- Andere Algebren (z.B. boolsch) $\leadsto$ Graphenalgorithmen, ...

7.1 Matrizenrechnung

- $\mathbf{A} \in \mathbf{R}^{r\times s}$, $\mathbf{B} \in \mathbf{R}^{s\times t}$ Matrizen mit $\mathbf{A} = (a_{ij})$ und $\mathbf{B} = (b_{ij})$
- $\mathbf{R}$ Ring (später: Körper)

$$c_{ij} = \sum_{k=1}^{s} a_{ik} \cdot b_{kj}$$

7.1.1 Geblockte Schreibweise

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \text{ und } \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$\mathbf{C} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

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

Allgemein:

$$C_{ij} = \sum_{k} A_{ik} \cdot B_{kj}$$

(nicht notwendig quadratisch)

Anwendungen der Zerlegung

- Besserer sequentieller Code (warum?)
- Asymptotisch schnellere Algorithmen (Strassen $O(n^{2.81})$, Winograd ...)
- Parallelisierung

Bibliotheksfunktionen

Basic Linear Algebra Subroutines (BLAS)

- Meist **viel** effizienter als direkte Implementierung.
  - Blockung an Speicherhierarchie anpassen
  - Alinierung
  - Prefetching
  - Schleifen ausrollen
  - Software-Pipelining
  - Register nutzen
  - Instruction-Scheduling
- Aufgabe 5.2. Eigenes Programm versus Bibliothek
  `praktikum/ATLAS/atlas.ps,README`
  Architektur `PII` Compiler `pgcc -O6` oder `pcc -fast -O4`.
- Weitere Info
  `http://www.netlib.org/atlas,blas`
- Auch Parallel

7.1.2 Matrixmultiplikation: Ein trivialer Algorithmus

„Unfaire“ Ein- Ausgabebedingungen

- Eingabe: Alle PEs besitzen Kopien beider Matrizen
- Berechnung/Ausgabe: PE i berechnet Spalten $in/P \leq j < (i+1)n/P$

Realistischer: PE 0 macht Eingabe und Ausgabe

1. Broadcast **A**
2. Scatter Spalten von **B** (MPI-Datentypen!)
3. PE i berechnet Spalten $in/P \leq j < (i+1)n/P$
4. Gather **C**

7.1.3 Analyse

- $n \times n$ Matrizen
- P teilt n
- $\approx 2n^3/PT_{\text{flop}}$ für arithm. Operationen
- $T_{\text{gather}} = T_{\text{scatter}} \geq 8n^2 T_{\text{byte}}$
- $T_{\text{broadcast}} =\geq 2 \cdot 8n^2 T_{\text{byte}}$
- $T_{\text{par}} \geq 2n^3/PT_{\text{flop}} + 32n^2 T_{\text{byte}}$
- Gute Effzienz nur wenn

$$\begin{aligned} & 2n^3/PT_{\text{flop}} \gg 32n^2 T_{\text{byte}} \\ \Leftrightarrow \quad & n \gg 16P\frac{T_{\text{byte}}}{T_{\text{flop}}} \end{aligned}$$

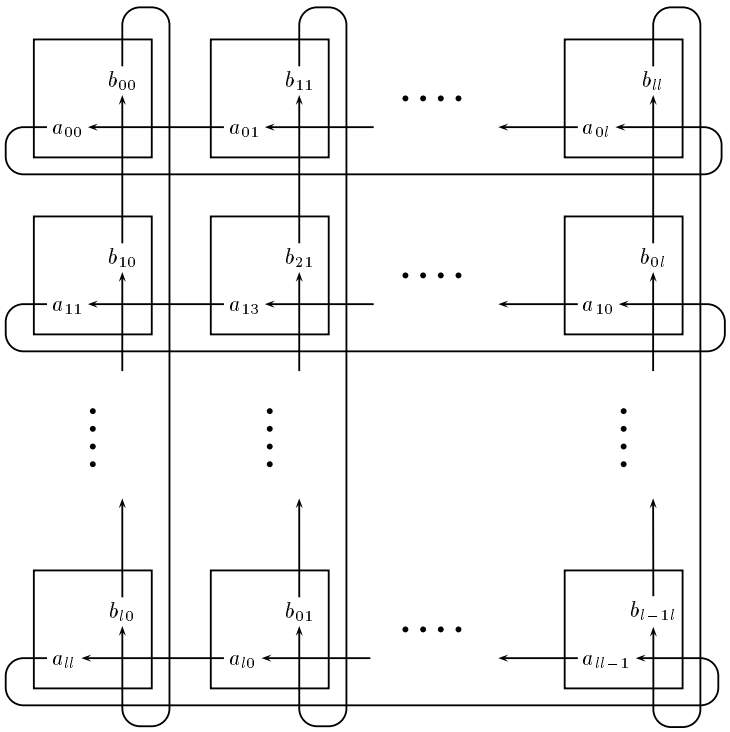
Beispiel $T_{\text{flop}} = 4\text{ns}$, $T_{\text{byte}} = 300\text{ns}$, $P = 16$:

$$n \gg 19200$$

Speicherbedarf: mehr als 8GB!

7.1.4 Matrixmultiplikation nach Cannon

$$c_{ij} \leftarrow c_{ij} + a_{i,(i+j+k) \bmod n} \cdot b_{(i+j+k) \bmod n, j}$$



Analyse

- P Quadratzahl
- Blockgrößen $n/\sqrt{P} \times n/\sqrt{P}$
- $\sqrt{P}$ Schritte
- $\approx 2(n/\sqrt{P})^3 T_{\text{flop}}$ für arithm. Operationen
- Kommunikation a $4(T_{\text{start}} + (8n/\sqrt{P})^2 T_{\text{byte}})$
- $T_{\text{par}} = 2n^3/PT_{\text{flop}} + 4\sqrt{P}T_{\text{start}} + 32n^2 T_{\text{byte}}/\sqrt{P})$
- Gute Effzienz falls
  - $2n^3/PT_{\text{flop}} \gg 4\sqrt{P}T_{\text{start}}$
  - $2n^3/PT_{\text{flop}} \gg 32n^2 T_{\text{byte}}/\sqrt{P})$
  - $n \gg \sqrt{P}\max(\sqrt[3]{2T_{\text{start}}/Tflop}, 16T_{\text{byte}}/T_{\text{flop}})$

Beispiel $T_{\text{flop}} = 4$ns, T_{start} =300 000ns,
T_{byte} =300ns, $P = 16$:

$$n \gg \max(212.5, 4800)$$

Also ist Bandbreite das Hauptproblem.
Speicherbedarf bei $n = 4800$:
$3 \cdot 8 \cdot 4800^2 \approx 550MB$

Systolische Algorithmen allgemein

- Immer die gleichen Befehle (im wesentlichen)
- Wandernde Daten
- Synchrone Arbeitsweise
- Vorzugsweise einfache Kommunikationsmuster
- Gut für Hardwareentwurf

Verfeinerungen

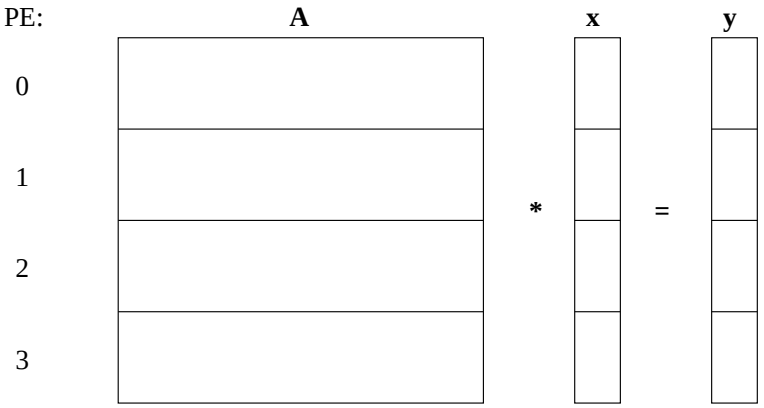
- MPI Prozesstopologien
  - `MPI_Cart_create` numeriert die PEs so um daß tatsächlich Nachbarschaftskommunikation stattfindet
  - `MPI_Dims_create` hilft geeignete Dimensionen finden
  - Diverse andere Hilfsfunktionen. Koordinatenumrechung etc. `MPI_Cart_rank`, `MPI_Cart_shift`, ...

7.2 Matrix-Vektormultiplikation

- Degenerierte Matrix-Matrixmultiplikation ...
- Basis für iterative Gleichungslöser, Eigenwertprobleme, etc.

$$T_{\text{seq}} = 2n^2 T_{\text{flop}} + \mathrm{O}(n)$$

Zeilenweise Zerlegung



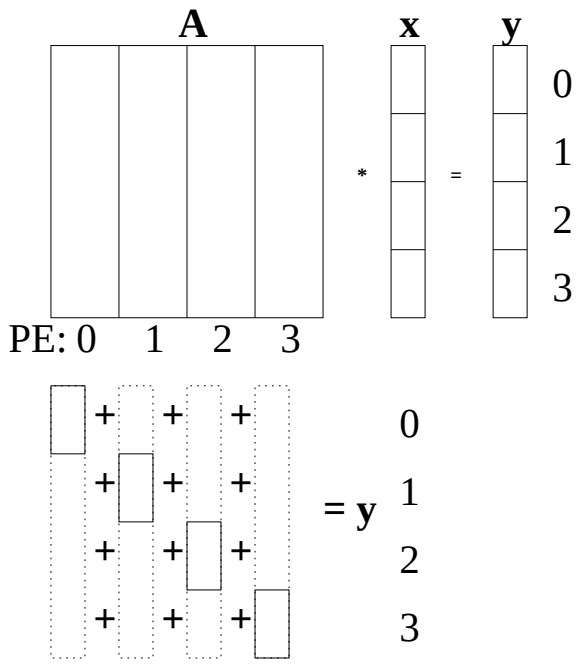
Analyse

$$\begin{aligned} T_{\text{par}} &\approx 2\frac{n^2}{P}T_{\text{flop}} + T_{\text{all-gather}}(n/P, P) \\ &\approx 2\frac{n^2}{P}T_{\text{flop}} + 2 \cdot 8nT_{\text{byte}} \end{aligned}$$

Gute Effizienz für

$$n \gg 8nP\frac{T_{\text{byte}}}{T_{\text{flop}}}$$

Spaltenweise Zerlegung



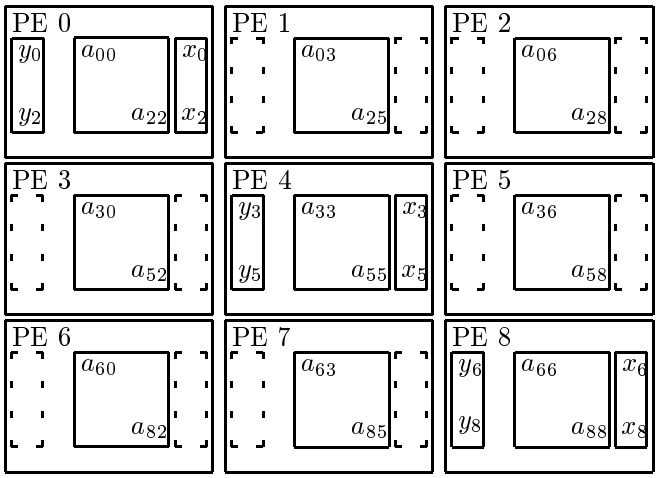
Analyse

- Jeder kriegt ganzes Ergebnis:
  $T_{\text{par}} \approx 2\frac{n^2}{P}T_{\text{flop}} + T_{\text{allreduce}}(n, P)$
- P Teilsummationen:
  $T_{\text{par}} \approx 2\frac{n^2}{P}T_{\text{flop}} + PT_{\text{reduce}}(n/P, P)$
- Jeder sammelt seinen Teil und summiert dann:
  $T_{\text{par}} \approx 2\frac{n^2}{P}T_{\text{flop}} + T_{\text{all-to-all}}(n/P, P)$
  Nur bei vollstaändiger Verknüpfung konkurrenzfähig.
- Die maßgeschneiderte MPI Funktion:
  $T_{\text{par}} \approx 2\frac{n^2}{P}T_{\text{flop}} + T_{\text{reduce-scatter}}(n/P, P)$

Gute Effizienz für

$$n \gg 16nP\frac{T_{\text{byte}}}{T_{\text{flop}}}$$

7.2.1 Zweidimensionale Zerlegung



7.2.2 Analyse

$$\begin{aligned} T_{\text{par}} &\approx 2\frac{n^2}{P}T_{\text{flop}} + T_{\text{broadcast}}(n/\sqrt{P}, \sqrt{P}) + \\ &\quad T_{\text{reduce}}(n/\sqrt{P}, \sqrt{P}) \\ &\approx 2\frac{n^2}{P}T_{\text{flop}} + 4 \cdot 8nT_{\text{byte}}/\sqrt{P} \end{aligned}$$

Gute Effizienz für $n \gg 16\frac{T_{\text{byte}}}{T_{\text{flop}}}\sqrt{P}$.

7.2.3 `MPI_Comm_split`

```
int MPI_Comm_split(
  MPI_Comm comm, int color, int key,
  MPI_Comm *newComm);
```

- Kollektiv auf `comm`
- Jede Farbe gibt einen Kommunikator (zurückgegeben in `newComm`)
- $\texttt{key}_i < \texttt{key}_j \implies \texttt{rank}_i < \texttt{rank}_j$
- `MPI_UNDEFINED`

Beispiel:

```
MPI_Comm_size(MPI_COMM_WORLD, &P);
MPI_Comm_rank(MPI_COMM_WORLD, &i);
MPI_Comm_split(MPI_COMM_WORLD, i >= P/2, 0,
               &myHalf);
```


7.2.4 Der DNS-Algorithmus zur Matrixmultiplikation

- Dekel, Nassimi und Sahni
- KumEtAl94 Abschnitt 5.4.4.
- Blockgröße $n/\sqrt[3]{P} \times n/\sqrt[3]{P}$
- $T_{\text{par}} \approx$ $2n^3/PT_{\text{flop}} + 2T_{\text{broadcast}}(n^2/P^{2/3}, P^{1/3}) +$ $T_{\text{reduce}}(n^2/P^{2/3}, P^{1/3})$
- Effiziente Broadcast und Reduce-Operationen wichtig

\+ Kommunikationsvolumen nur $\mathrm{O}\left(n^2/P^{2/3}\right)$ pro PE

– Speicherbedarf insgesamt $\Omega\left(n^2P^{1/3}\right)$

– Mindestens $3D$-Gitter nötig (zumindest asymptotisch)

7.3 Lösen Linearer Gleichungssysteme

- Viele Verfahren
- Abhängig von
  - Struktur der Matrix
  - Erwägungen bzgl. Numerische Stabilität
- Hier: Gausselimination mit Zeilen-Pivot-Suche
- Ein *direktes* Verfahren
- Verwandte Probleme:
  - Gauss-Jordan-Elimination
  - LU Zerlegung
  - Matrixinversion
  - Cholesky Zerlegung

Hier: Gausselimination mit Zeilen-Pivot-Suche

7.3.1 Mathematische Formulierung

$n \times n$ Matrix

$$\mathbf{A} = (a_{ij})$$

$(0 \leq i, j < n)$ löse

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$$

7.3.2 Algorithmus in Vektorschreibweise

PROCEDURE GaussElimination($\mathbf{A}' = [\mathbf{A}|b]$) : n-Vector
for $k := 0$ **to** $n-1$ **do**

(* Pivoting *) find $k \leq i < n$ such that a_{ik} is maximi
exchange rows k and i

(* make a 1 *)
multiply row k by $1/a_{kk}$

(* Zero out column below *)
$\mathbf{a}_{j*} = \mathbf{a}_{j*} - a_{jk}/a_{kk}\mathbf{a}_{k*}$ for $k < j < n$

(* Back substutution *)
return backSubstitute($\mathbf{A}'$)

$$\begin{aligned} T_{\text{seq}} &= \sum_k 2(n-k-1)^2 T_{\text{flop}} + \mathrm{O}(n-k) \\ &= \frac{2}{3}n^3 T_{\text{flop}} + \mathrm{O}\big(n^2\big) \end{aligned}$$

7.3.3 Spaltenweise Blockverteilung

- PE l is für Spalten mit $l(n+1)/P \leq i < (l+1)(n+1)/P$ zuständig.
- Broadcast von Spalte j (a_{kj} für $k \leq j < n$)
- Kommunkationskosten
  $\sum_k T_{\text{broadcast}}(n-k+1, P - \frac{kP}{n})$
  $\approx 8nT_{\text{byte}}$
- Lokale Berechnung:
  $\sum_k 2(n-k-1)\frac{n}{P}T_{\text{flop}}$
  $= \frac{n^3}{P}T_{\text{flop}} + \mathrm{O}\Big(\frac{n^2}{P}\Big)$. Oops.
- Problem: Eine zunehmende Zahl von Prozessoren wird arbeitslos.

7.3.4 Spaltenweise zyklische Verteilung

PE l ist für Spalten mit $i \bmod P = l$ zuständig.

$$T_{\text{par}} = \frac{2}{3}n^3 T_{\text{flop}} + \mathrm{O}\big(n^2 P T_{\text{flop}} + n T_{\text{broadcast}}(n,P)\big)$$

Entkopplung

- Spaltenweise Blockverteilung:
  - Fasse m Iterationen zusammen
  - Bringe $m \times m$ Matrix in Dreiecksform $\leadsto 2/3m^3 T_{\text{flop}}$ Wartezeit
  - Ein Broadcast m facher Länge statt m einfache Broadcasts $\leadsto \mathrm{O}(f(P)m)$ gesparten Startups ($f(P)$ zwischen $\log p$ und P.
  - Optimales m in $\sqrt{\frac{T_{\text{start}} f(P)}{2T_{\text{flop}}}}$
- So nicht bei zyklischer Verteilung
- Lösung block-zyklische Verteilung mit Blöcken der Größe m Bedingung: $mP \ll n$.