

Algorithmen für vollbesetzte Matrizen II

Stefan Lang

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg
INF 368, Raum 532
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

WS 11/12



Themen

Datenparallele Algorithmen für vollbesetzte Matrizen

- Matrix-Vektor Multiplikation
- Matrix-Matrix Multiplikation



Matrix-Vektor Multiplikation

Berechne $y = Ax$, Matrix $A \in \mathcal{R}^{N \times M}$ und Vektor $x \in \mathcal{R}^M$

- Unterschiedliche Möglichkeiten zur Datenaufteilung.
- Verteilung der Matrix und des Vektors müssen aufeinander abgestimmt sein
- Verteilung des Ergebnisvektor $y \in \mathcal{R}^N$ wie bei Eingabevektor x

Beispiel:

- Matrix sei blockweise auf eine Feldtopologie verteilt
- Eingabevektor x entsprechend blockweise über die Diagonalprozessoren verteilt
- das Prozessorfeld ist quadratisch
- Vektorsegment x_q wird in jeder Prozessorspalte benötigt und ist somit in jeder Spalte zu kopieren (einer-an-alle).
- lokale Berechnung des Produkts $y_{p,q} = A_{p,q}x_q$.
- komplettes Segment y_p ergibt sich erst durch die Summation $y_p = \sum_q y_{p,q}$. (weitere alle-an-einen Kommunikation)
- Resultat kann unmittelbar für weitere Matrix-Vektor-Multiplikation benutzt werden



Matrix-Vektor Multiplikation: Aufteilung

Aufteilung für das Matrix-Vektor-Produkt



Matrix-Vektor Multiplikation: Parallele Laufzeit

Parallele Laufzeit für eine $N \times N$ -Matrix und $\sqrt{P} \times \sqrt{P}$ Prozessoren mit cut-through Kommunikationsnetzwerk:

$$\begin{aligned} T_P(N, P) &= \underbrace{\left(t_s + t_h + t_w \frac{\overbrace{N}^{\text{Vektor}}}{\sqrt{P}} \right) \text{Id } \sqrt{P}}_{\text{Austeilen von } x \text{ über Spalte}} + \underbrace{\left(\frac{N}{\sqrt{P}} \right)^2 2t_f}_{\text{lokale Matrix-Vektor-Mult.}} \\ &+ \underbrace{\left(t_s + t_h + t_w \frac{N}{\sqrt{P}} \right) \text{Id } \sqrt{P}}_{\text{Reduktion } (t_f \ll t_w)} = \\ &= \text{Id } \sqrt{P} (t_s + t_h) 2 + \frac{N}{\sqrt{P}} \text{Id } \sqrt{P} 2t_w + \frac{N^2}{P} 2t_f \end{aligned}$$

Für festes P und $N \rightarrow \infty$ wird der Kommunikationsanteil beliebig klein, es existiert also eine Isoeffizienzfunktion, der Algorithmus ist skalierbar.



Matrix-Vektor Multiplikation: Arbeit/Overhead

Berechnen wir Arbeit und Overhead:

Umrechnen auf die Arbeit W :

$$W = N^2 2t_f \text{ (seq. Laufzeit)}$$

$$\Rightarrow N = \frac{\sqrt{W}}{\sqrt{2t_f}}$$

$$T_P(W, P) = \text{ld} \sqrt{P} (t_s + t_h) 2 + \frac{\sqrt{W}}{\sqrt{P}} \text{ld} \sqrt{P} \frac{2t_w}{\sqrt{2t_f}} + \frac{W}{P}$$

Overhead:

$$\begin{aligned} T_O(W, P) &= P T_P(W, P) - W = \\ &= \sqrt{W} \sqrt{P} \text{ld} \sqrt{P} \frac{2t_w}{\sqrt{2t_f}} + P \text{ld} \sqrt{P} (t_s + t_h) 2 \end{aligned}$$



Matrix-Vektor Multiplikation: Isoeffizienz

und nun die Isoeffizienzfunktion:

Isoeffizienz ($T_0(W, P) \stackrel{!}{=} KW$): T_0 hat zwei Terme. Für den ersten erhalten wir

$$\begin{aligned} \sqrt{W}\sqrt{P} \text{Id} \sqrt{P} \frac{2t_w}{\sqrt{2t_f}} &= KW \\ \Leftrightarrow W &= P(\text{Id} \sqrt{P})^2 \frac{4t_w^2}{2t_f K^2} \end{aligned}$$

und für den zweiten

$$\begin{aligned} P \text{Id} \sqrt{P}(t_s + t_h)2 &= KW \\ \Leftrightarrow W &= P \text{Id} \sqrt{P} \frac{(t_s + t_h)2}{K}; \end{aligned}$$

somit ist $W = \Theta(P(\text{Id} \sqrt{P})^2)$ die gesuchte Isoeffizienzfunktion.



Algorithmus von Cannon

Es ist $C = A \cdot B$ zu berechnen.

- Zu multiplizierende $N \times N$ -Matrizen A und B sind blockweise auf eine 2D-Feldtopologie ($\sqrt{P} \times \sqrt{P}$) verteilt
- Praktischerweise soll das Ergebnis C wieder in derselben Verteilung vorliegen.
- Prozess (p, q) muss somit

$$C_{p,q} = \sum_k A_{p,k} \cdot B_{k,q}$$

berechnen, benötigt also Blockzeile p von A und Blockspalte q von B .



Matrix-Matrix-Multiplikation

Die zwei Phasen des Algorithmus von Cannon:

- 1 *Alignment-Phase*: Die Blöcke von A werden in jeder Zeile zyklisch nach links geschoben, bis der Diagonalblock in der ersten Spalte zu liegen kommt. Entsprechend schiebt man die Blöcke von B in den Spalten nach oben, bis alle Diagonalblöcke in der ersten Zeile liegen. Nach der Alignment-Phase hat Prozessor (p, q) die Blöcke

$$A_{p, \underbrace{(q+p) \% \sqrt{P}}} \quad (\text{Zeile } p \text{ schiebt } p \text{ mal nach links})$$

$$B_{\underbrace{(p+q) \% \sqrt{P}}, q} \quad (\text{Spalte } q \text{ schiebt } q \text{ mal nach oben}).$$

- 2 *Rechenphase*: Offensichtlich verfügt nun jeder Prozess über zwei passende Blöcke, die er multiplizieren kann. Schiebt man die Blöcke von A in jeder Zeile von A zyklisch um eine Position nach links und die von B in jeder Spalte nach oben, so erhält jeder wieder zwei passende Blöcke. Nach \sqrt{P} Schritten ist man fertig.



Cannon's Algorithmus

- basiert auf blockweise Partitionierung der Matrizen
- Setup-Phase
 - ▶ Rotation der Matrizen A und B
- Iteration über \sqrt{p}
 - ▶ Berechne lokales Block-Matrix-Produkt
 - ▶ Shift A horizontal und B vertikal

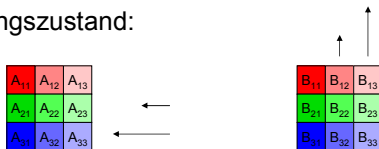
$$\begin{array}{|c|c|c|} \hline C_{11} & C_{12} & C_{13} \\ \hline C_{21} & C_{22} & C_{23} \\ \hline C_{31} & C_{32} & C_{33} \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline A_{11} & A_{12} & A_{13} \\ \hline A_{21} & A_{22} & A_{23} \\ \hline A_{31} & A_{32} & A_{33} \\ \hline \end{array} \times \begin{array}{|c|c|c|} \hline B_{11} & B_{12} & B_{13} \\ \hline B_{21} & B_{22} & B_{23} \\ \hline B_{31} & B_{32} & B_{33} \\ \hline \end{array}$$

C A B



Cannon's Algorithmus - Rotation

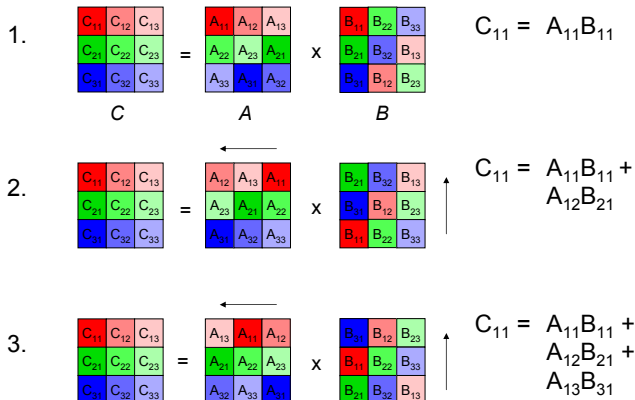
- Anfangszustand:



- Verdrehen: Rotieren der i. Zeile (Spalte) von A (B) um i-Schritte:



Cannon's Algorithmus - Iteration



Cannon's Algorithm - Performance Analysis

- A, B verdrehen: $2 * s * (t_{\text{startup}} + t_{\text{word}} N^2/p)$
- Iteration (s-mal):
 - dgemm: $2 * t_{\text{flop}} * (n/s)^3 = 2 * t_{\text{flop}} * n^3/p^{1.5}$
 - A, B rollen: $2 * (t_{\text{startup}} + t_{\text{word}} N^2/p)$
- Gesamt: $t_{\text{cannon}}(p) = 4t_{\text{startup}} * s + 4t_{\text{word}} * N^2/s + 2t_{\text{flop}} * N^3/p$
- Effizienz = $2 t_{\text{flop}} * N^3 / (p * t_{\text{cannon}}(p))$
= $1 / (2t_{\text{startup}} * (s/N)^3 + 2t_{\text{word}} * s/N + t_{\text{flop}})$
 $\approx 1 / O(1 + \text{sqrt}(p) / N)$
- Effizienz $\rightarrow 1$, wenn $(N/s) \rightarrow \infty$
 - $N / s = N / \text{sqrt}(p) = \text{sqrt}(\text{Daten pro Prozessor})$



Cannon mit MPI (Init)

```
/* Baue Gitter und hole Koordinaten */
int  dims[2], periods[2] = {1, 1};
int  mycoords[2];

dims[0] = sqrt(num_procs);
dims[1] = num_procs / dims[0];

MPI_Cart_create(MPI_COMM_WORLD, /* kollektiv */
                2, dims, periods,
                0, &comm_2d);
MPI_Comm_rank(comm_2d, &my2drank);
MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);

/* Lokale Blöcke der Matrizen */
double      *a, *b, *c;

/* Lade a, b  und c entsprechend der Koordinaten */
...
```



Cannon mit MPI (Rotate)

```
/* Matrix-Verdrehung A */
MPI_Cart_shift(comm_2d, 0, -mycoords[0],
               &shiftsource, &shiftdest);
MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
                    shiftdest, 77, shiftsource, 77,
                    comm_2d, &status);

/* Matrix-Verdrehung B */
MPI_Cart_shift(comm_2d, 1, -mycoords[1],
               &shiftsource, &shiftdest);
MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
                    shiftdest, 77, shiftsource, 77,
                    comm_2d, &status);
```



Cannon mit MPI (Iteration)

```
/* Finde linken und oberen Nachbarn */
MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &lefttrank);
MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);

for (i=0; i<dims[0]; i++)
{
    dgemv(nlocal, a, b, c); /* c= c + a * b */

    /* Matrix A nach links rollen */
    MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
                        lefttrank, 77, righttrank, 77,
                        comm_2d, &status);

    /* Matrix B nach oben rollen */
    MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
                        uprank, 77, downrank, 77,
                        comm_2d, &status);
}

/* A und B zurück in Ursprungs-Zustand */
...
```



Cannon - Praktische Aspekte

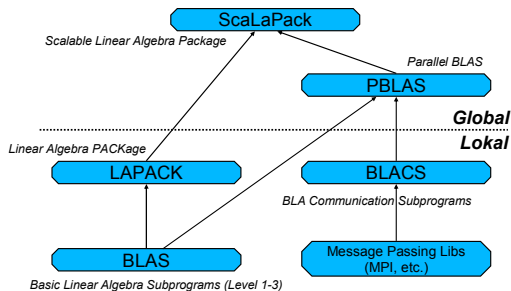
- effiziente, aber nicht einfache Verallgemeinerung, falls
 - ▶ Matrizen nicht quadratisch sind
 - ▶ Dimensionen sind nicht ohne Rest durch p teilbar
 - ▶ andere Matrix Partitionierungen gebraucht werden
- Isoeffizienzfunktion von Cannon's Algorithmus: $O(P^{3/2})$,
 $N/\sqrt{P} = \text{const}$ → Effizienz bleibt konstant für feste Blockgrößen pro Prozessor und anwachsender Prozessorzahl
- Dekel-Nassimi-Salmi-Algorithmus erlaubt die Nutzung von N^3 Prozessoren (Cannon N^2) mit besserer Isoeffizienz Funktion.



Standard Bibliotheken für Lineare Algebra

ATLAS, BLITZ (expression templates), ISTL (generic programming),
ScaLaPack (klassisches Paket), Trilinos (Riesige Code Familie),

<http://www.netlib.org>



Matrix-Matrix-Multiplikation: Isoeffizienzanalyse

Betrachten wir die zugehörige Isoeffizienzfunktion.

Sequentielle Laufzeit :

$$W = T_S(N) = N^3 2t_f$$

$$\Rightarrow N = \left(\frac{W}{2t_f} \right)^{\frac{1}{3}}$$

parallele Laufzeit:

$$\begin{aligned} T_P(N, P) &= \underbrace{(\sqrt{P} - 1) \left(t_s + t_h + t_w \frac{N^2}{P} \right)}_{\text{alignment}} \underbrace{\text{send/recv } A/B}_{4} \\ &+ \sqrt{P} \left(\underbrace{\left(\frac{N}{\sqrt{P}} \right)^3 2t_f}_{\text{Multiplik. eines Blockes}} + \left(t_s + t_h + t_w \frac{N^2}{P} \right) 4 \right) \approx \\ &\approx \sqrt{P}(t_s + t_h)8 + \frac{N^2}{\sqrt{P}} t_w 8 + \frac{N^3}{P} 2t_f \\ T_P(W, P) &= \sqrt{P}(t_s + t_h)8 + \frac{W^{\frac{2}{3}}}{\sqrt{P}} \frac{8t_w}{(2t_f)^{\frac{1}{3}}} + \frac{W}{P} \end{aligned}$$



Matrix-Matrix-Multiplikation: Isoeffizienzanalyse

Overhead:

$$T_O(W, P) = PT_P(W, P) - W = P^{\frac{2}{3}}(t_s + t_h)8 + \sqrt{P}W^{\frac{2}{3}} \frac{8t_w}{(2t_f)^{\frac{1}{3}}}$$

Ergebnis:

- Somit ist $W = \Theta(P^{3/2})$.
- Wegen $N = \left(\frac{W}{2t_f}\right)^{1/3}$ gilt $N/\sqrt{P} = \text{const}$
- Somit ist bei *fester* Grösse der Blöcke in jedem Prozessor und wachsender Prozessorzahl die Effizienz konstant.
- Beschränken wir uns beim Algorithmus von Cannon auf 1×1 -Blöcke pro Prozessor, also $\sqrt{P} = N$, so können wir für die erforderlichen N^3 Multiplikationen nur N^2 Prozessoren nutzen.
- Dies ist der Grund für die Isoeffizienzfunktion der Ordnung $P^{3/2}$.



Matrix-Matrix-Multiplikation: Dekel-Nassimi-Salmi-Alg.

Dekel-Nassimi-Salmi-Algorithmus

- Nun betrachten wir einen Algorithmus der den Einsatz von bis zu N^3 Prozessoren bei einer $N \times N$ -Matrix erlaubt.
- Gegeben seien also $N \times N$ -Matrizen A und B sowie ein 3D-Feld von Prozessoren der Dimension $P^{1/3} \times P^{1/3} \times P^{1/3}$.
- Die Prozessoren werden über die Koordinaten (p, q, r) adressiert.
- Um den Block $C_{p,q}$ der Ergebnismatrix C mittels

$$C_{p,q} = \sum_{r=0}^{P^{1/3}-1} A_{p,r} \cdot B_{r,q} \quad (1)$$

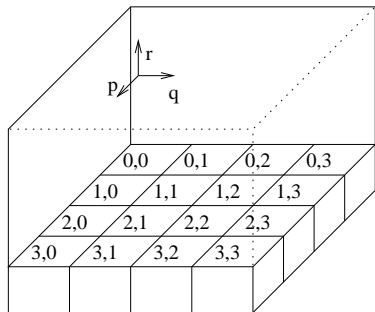
zu berechnen, setzen wir $P^{1/3}$ Prozessoren ein, und zwar ist Prozessor (p, q, r) genau für das Produkt $A_{p,r} \cdot B_{r,q}$ zuständig.

- Nun ist noch zu klären, wie Eingabe- und Ergebnismatrizen verteilt sein sollen.
- Sowohl A als auch B sind in $P^{1/3} \times P^{1/3}$ -Blöcke der Größe $\frac{N}{P^{1/3}} \times \frac{N}{P^{1/3}}$ zerlegt.
- $A_{p,q}$ und $B_{p,q}$ wird zu Beginn in Prozessor $(p, q, 0)$ gespeichert, auch das Ergebnis $C_{p,q}$ soll dort liegen.
- Die Prozessoren (p, q, r) für $r > 0$ werden nur zwischenzeitlich benutzt.

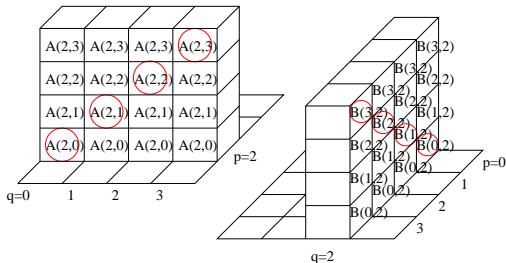


Matrix-Matrix-Multiplikation: Dekel-Nassimi-Salmi-Alg.

Verteilung von A , B , C für $P^{1/3} = 4$ ($P=64$).



Verteilung der Blöcke von A und B
(zu Beginn) und C (am Ende)



Verteilung von A und B für die
Multiplikation



Matrix-Matrix-Multiplikation: Dekel-Nassimi-Salmi-Alg.

- Damit nun jeder Prozessor (p, q, r) „seine“ Multiplikation $A_{p,r} \cdot B_{r,q}$ durchführen kann, sind die beteiligten Blöcke von A und B erst an ihre richtige Position zu befördern.
- Alle Prozessoren benötigen $(p, *, r)$ den Block $A_{p,r}$ und alle Prozessoren $(*, q, r)$ den Block $B_{r,q}$.
- Sie wird folgendermaßen hergestellt:

*Prozessor $(p, q, 0)$ sendet $A_{p,q}$ an Prozessor (p, q, q) und dann sendet (p, q, q) das $A_{p,q}$ an alle $(p, *, q)$ mittels einer einer-an-alle Kommunikation auf $P^{1/3}$ Prozessoren.*

Entsprechend schickt $(p, q, 0)$ das $B_{p,q}$ an Prozessor (p, q, p) , und dieser verteilt dann an $(, q, p)$.*

- Nach der Multiplikation in jedem (p, q, r) sind die Ergebnisse aller $(p, q, *)$ noch in $(p, q, 0)$ mittels einer alle-an-einen Kommunikation auf $P^{1/3}$ Prozessoren zu sammeln.



Matrix-Matrix-Multiplikation: Dekel-Nassimi-Salmi-Alg.

Analysieren wir das Verfahren im Detail (3D-cut-through Netzwerk):

$$W = T_S(N) = N^3 2t_f \Rightarrow N = \left(\frac{W}{2t_f} \right)^{\frac{1}{3}}$$

$$T_P(N, P) = \underbrace{\left(t_s + t_h + t_w \left(\frac{N}{P^{\frac{1}{3}}} \right)^2 \right)}_{(p,q,0) \rightarrow (p,q,q), (p,q,p)} \underbrace{2}_{\substack{A_{p,q} \text{ u.} \\ B_{p,q}}} + \underbrace{\left(t_s + t_h + t_w \left(\frac{N}{P^{\frac{1}{3}}} \right)^2 \right) \text{Id } P^{\frac{1}{3}}}_{\text{einer-an-alle}} \underbrace{2}_{A,B}$$

$$+ \underbrace{\left(\frac{N}{P^{\frac{1}{3}}} \right)^3 2t_f}_{\text{Multiplikation}} + \underbrace{\left(t_s + t_h + t_w \left(\frac{N}{P^{\frac{1}{3}}} \right)^2 \right) \text{Id } P^{\frac{1}{3}}}_{\substack{\text{alle-an-einen} \\ (t_f \ll t_w)}} \approx$$

$$\approx 3 \text{Id } P^{\frac{1}{3}} (t_s + t_h) + \frac{N^2}{P^{\frac{2}{3}}} 3 \text{Id } P^{\frac{1}{3}} t_w + \frac{N^3}{P} 2t_f$$

$$T_P(W, P) = 3 \text{Id } P^{\frac{1}{3}} (t_s + t_h) + \frac{W^{\frac{2}{3}}}{P^{\frac{2}{3}}} 3 \text{Id } P^{\frac{1}{3}} \frac{t_w}{(2t_f)^{\frac{2}{3}}} + \frac{W}{P}$$

$$T_O(W, P) = P \text{Id } P^{\frac{1}{3}} 3(t_s + t_h) + W^{\frac{2}{3}} P^{\frac{1}{3}} \text{Id } P^{\frac{1}{3}} \frac{3t_w}{(2t_f)^{\frac{2}{3}}}$$



Matrix-Matrix-Multiplikation: Dekel-Nassimi-Salmi-Alg.

- Aus dem zweiten Term von $T_0(W, P)$ nähern wir die Isoeffizienzfunktion an:

$$W^{\frac{2}{3}} P^{\frac{1}{3}} \text{ld } P^{\frac{1}{3}} \frac{3t_w}{(2t_f)^{\frac{2}{3}}} = KW$$

$$\Leftrightarrow W^{\frac{1}{3}} = P^{\frac{1}{3}} \text{ld } P^{\frac{1}{3}} \frac{3t_w}{(2t_f)^{\frac{2}{3}} K}$$

$$\Leftrightarrow \boxed{W = P \left(\text{ld } P^{\frac{1}{3}} \right)^3 \frac{27t_w^3}{4t_f^2 K^3}}$$

- Also erhalten wir die Isoeffizienzfunktion $O(P(\text{ld } P)^3)$ und somit eine bessere Skalierbarkeit als für den Cannon'schen Algorithmus.
- Wir haben immer angenommen, dass die optimale sequentielle Komplexität der Matrixmultiplikation N^3 ist. Der Algorithmus von Strassen hat jedoch eine Komplexität von $O(N^{2.87})$.
- Für eine effiziente Implementierung der Multiplikation zweier Matrixblöcke auf einem Prozessor muß auf Cacheeffizienz geachtet werden.

