

Letzte Vorlesung:

Format für dicht besetzte Matrizen.

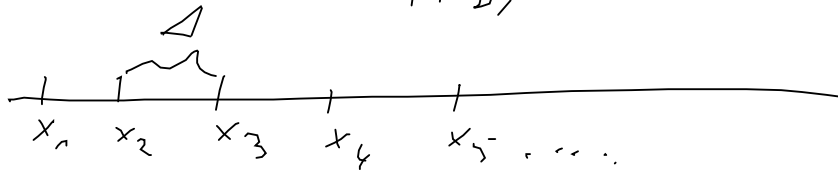
Jetzt: Format für dünn besetzte Matrizen

* Problem: (Sehr) oft gibt es dünn besetzte Matrizen

Ein Beispiel

$$\partial_x^2 \varphi(x)$$

Wir diskretisieren $\varphi(x_n)$



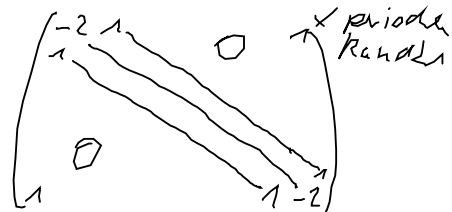
$$\partial_x^2 \varphi(x) \approx \varphi(x_{n-1}) - 2\varphi(x_n) + \varphi(x_{n+1})$$

Die zugehörige Matrix hat die Form

Vektor der Längen N

Also von N^2 Einträgen sind

nur $3N$ besetzt!



$$\frac{3N}{N^2} = \frac{3}{N} \leftarrow \text{Bei großen } N \text{ ist dies sehr klein!}$$

Das naive Matrixformat ist sehr schlecht bei großen N !

Anders Beispiel $V(x_n) \varphi(x_n) \rightarrow \begin{pmatrix} V(x_1) & & 0 \\ & \ddots & \\ 0 & & V(x_N) \end{pmatrix}$

(Literatur: LinAlg book netlib.org)

Dünn besetzte Matrizen

Naivste Format

Ein Array mit Fließkommazellen $v[i]$

Array mit ganzen Zahlen $i[j]$ $j[i]$

Beispiel: $i = 0 \ 1 \ 2 \ 3$

$$j = \begin{pmatrix} 0 & 2 & 6 \\ 1 & & \\ 2 & & 8 \\ 3 & 7 & \\ 4 & & \end{pmatrix}$$

$$v = \{ 2, 6, 8, 7 \}$$

$$i = \{ 0, 2, 3, 1 \}$$

$$j = \{ 0, 0, 2, 3 \}$$

funktioniert zwar, ist aber ineffektiv!

Compressed Row Storage Format aka Val sparse format, general sparse A[] format

Array val []

speichert die Elemente die nicht null sind

Array col_ind []

enthält die Spalten indices

Array row_ptr []

enthält den Index, in der beiden andere Arrays, beginn der Reihe gespeichert wird.

Beispiel (von netlib.org):

$$M = \begin{pmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{pmatrix}$$

\downarrow
 \downarrow
 \downarrow

$$val = \{ 10, -2, 3, 9, 3, 7, 8, 7, 3, \dots, 9, 13, 4, 2, -1 \}$$

$$col_ind = \{ 0, 4, 0, 1, 5, 1, 2, 3, 0, \dots, 4, 5, 1, 4 \}$$

$$row_ptr = \{ 0, 2, 5, 8, 12, 16, 15 \}$$

Vorteil: Speicherplatzersparnis, Nachteil: schlechter memory access pattern Vektor

Compressed Column Storage:

das gleiche mit vertauschten Zeilen und Spalten (z.B. Transponierte zu berechnen)

Weitere Formate für spezielle Anwendungen existieren.

Für die typischen Formate 1x. opt. Fassung für Matrixmultiplikation.

Tipps: Zusammenbau der sparse Matrix aufwendig.

In der Praxis muss man vor der Speicherallozierung der Struktur die Elemente abzählen. Da die Anpassung der Arraygröße ineffektiv ist,

Blockmatrizen: (z. B. als multiplizierte Probe)

$$\begin{pmatrix} A_{aa} & A_{ab} & A_{ac} \\ A_{ba} & A_{bb} & A_{bc} \\ A_{ca} & A_{cb} & A_{cc} \end{pmatrix}$$

Zwei Möglichkeiten

1) monolithische Matrix

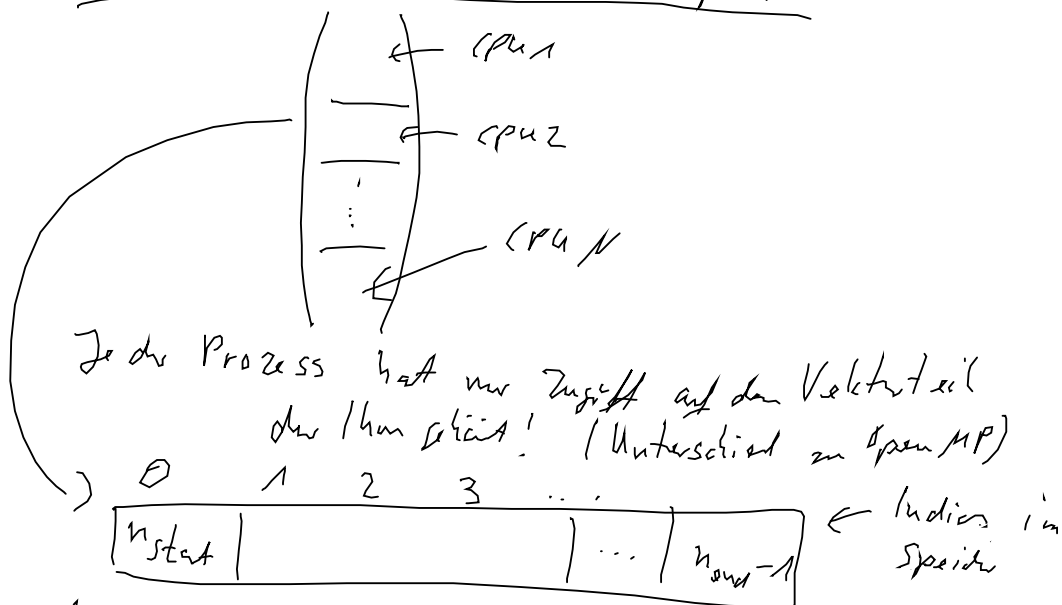
oder

2) verschaltete Matrizen

Es kann organisierter Vorteil haben (verschiedene Gruppen) oder aber auch Vorteil in Algorithmen, daher ist das auch eine Speichermöglichkeit.

Parallele Berechnung der Matrizen

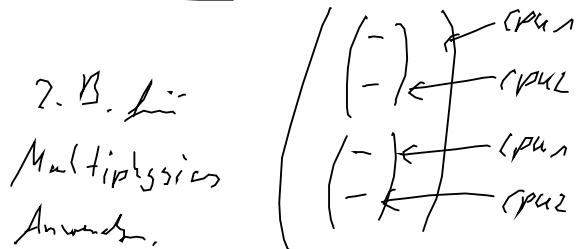
Zunächst Vektoren für MPI (Beispiel)



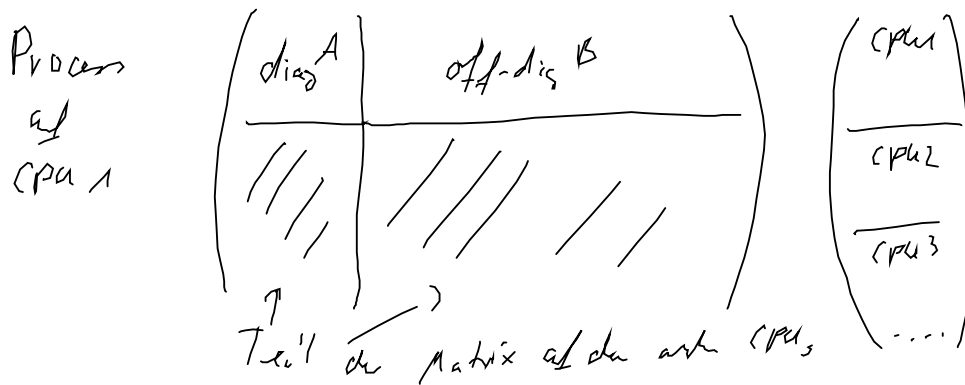
Möchte man auf ein Teil des Vektors zugreifen, der mir nicht zugehört, so muss dieser gescaffert werden (Stichwort Scatter, Scatter) übertragen werden

Bemerkung: Vektoren müssen während des Zugriffs sortiert werden
 (bei Petsc z.B. ist es auf der Spätkante, muss der Vektor übertragen werden)
 (0,0, 1, per der, Matrixwerte etc.)

Alternativ für geblockte Anwendungen:



Wie funktioniert die Matrixmultiplikation bei Parallelisierung?



z.B. Vorgehen von Petsc bei Matrixmultiplikation (UPC)

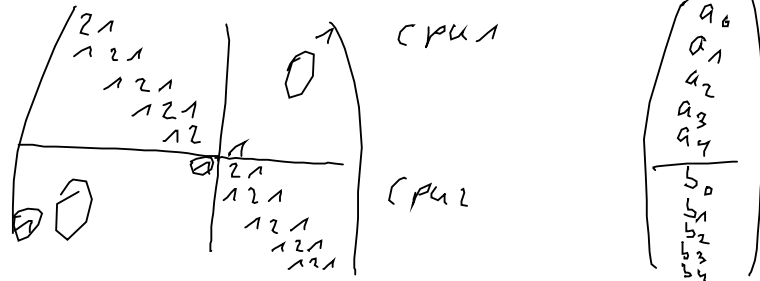
- 1) Schicke eigen Vektoren an andere CPUs
 (aber nur die Einträge die verwendet werden, da Spalten sind von Matrixschicht in den Transfer minimieren!)
- 2.) Währenddessen: Berechne $\underline{x} = \underline{A} \cdot \underline{a}$
- 3) Empfangen der Vektoranteile von den anderen CPUs \underline{b}

4) Berechne $\underline{b} = \underline{B} \cdot \underline{b} + \underline{x}$

↑↑ Ergebnis Nuller Spalten entfernt.

Dies werden CPUs machen das entspricht.

Beispiel:



1) Schicke (a_0) und (a_4) an CPU2

2) Berechne $\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} = \underline{x}$

3) Empfang $\begin{pmatrix} b_0 \\ b_4 \end{pmatrix}$

4) Berechne $\begin{pmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ \vdots & \vdots \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} b_0 \\ b_4 \end{pmatrix} + \underline{x} = \dots$

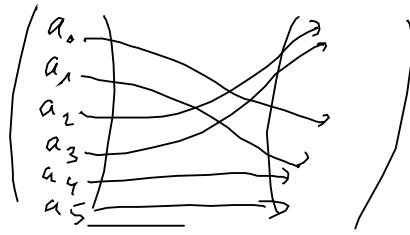
Tipp: Unbedingt Matrix so abfangen dass

der offdiagonale Part kaum besetzt ist.

- Besondere Anordng von Spalten in Speicher (Nächste VL)

- Es ex. für bel Matrizen (BSL) oder unstrukturierte Spalten, Bibliotheken, die die Einträge im Vektor so umsortieren, dass es nur wenige offdiagonal Einträge gibt

z.B. permatis (auch im Petsc eingebaut)



Sind viele Matrixeinträge gleich oder lässt sich die Matrix schnell berechnen (z.B. bestimmte Finite Elemente Methode)

Das heißt man implementiert die Funktion mit

Matrix Shell Mat Mult (vecin, vecout)

die $\underline{v_{out}} = \underline{M} \cdot \underline{v_{in}}$ berechnet.

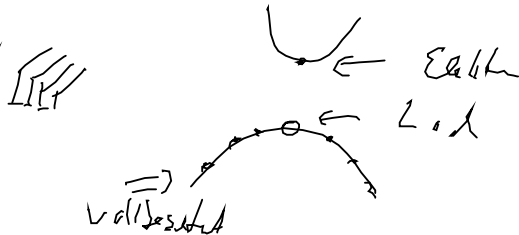
Dies ist entweder schneller als die Verwendung einer Matrix, schon implementiert, braucht weniger Speicher

Nachteil: Parallelisierte Implementierung aufwendig

II.4 Lösung der Schrödingergleichung über finite Differenz z.B. für Exziton und Triplet, Biexziton

Nur ein Beispiel aus der Physik wie man partielle Differentialgl. anwenden kann:

Halbleiter z.B. Quantenwell



Elektron-Ladungpaar wird z.B. optisch erzeugt



Coulomb Wechselwirkung führt zu ein Exziton, ein gebundenes Loch bei Wasserstoffatomproblem.