Lösungen zu Übungsblatt 5 (Parnum)

Aufgabe 18

zu 1)

Dicht besetzte Matrizen üblicher Benchmark für numerische Algorithmen. Zahlreiche Anwendungen führen aber zu spärlich besetzten Problemen (z.B. diskretisierte DGLs, Makov Modelle)

Matrizen, bei denen nur die Diagonale und die beiden Nebendiagonalen besetzt sind, resultieren aus der Diskretisierung 2-dimensionaler Probleme.

Matrizen bei denen K Nebendiagonalen besetzt sind, resultieren aus der Linearisierung höher dimensionaler Probleme. Z.B. Linearisierung K-dimensionales Problem mit Werten $i_k = 0, \ldots, n_k - 1$ kann durch lineares Feld der Länge $\prod_{k=1}^{K} n_k$ dargestellt werden. Dabei wird Element $(i_1, \ldots, i_K)$ auf Index $\sum_{k=1}^{K} i_k \cdot \prod_{l=k+1}^{K} n_l$ abgebildet (mixed Radix Nummerierung). Falls Übergänge in nur einer Dimension zwischen benachbarten Elementen möglich sind, d.h. $(i_1, \ldots, i_K)$ und $(j_1, \ldots, j_K)$ kommunizieren, falls ein k existiert, so dass $|i_k - j_k| = 1$ und $i_l = j_l$ für $l \neq k$, so entstehen in der Matrix Nichtnmullelement an Stelle (i, j) falls $|i - j| = n_k$ für $k = 1, \ldots, K$.

Spärlich besetzte Matrizen unregelmäßiger Struktur entstehen bei vielen Anwendungen, z.B. Markov Ketten. Wegeoptimierung, ..

zu 2)

a) Dicht besetzte Matrix 2-dimensionales Array

```
double a[X_DIM][Y_DIM] ;
```

sei `X_DIM` gleich n und `Y_DIM` gleich m

Speicherplatzbedarf nmd (mit d Größe double in Bytes)

b) Matrix mit Haupt- und Nebendiagonalelementen

b1) Speicherung zeilenweise mit je 3 Elementen pro Zeile

```
double a [X_DIM][3] ;
```

Element 0 ist untere Nebendiagonale, 1 Diagonale, 2 obere Nebendiagonale

b2) Speicherung von 3 Vektoren der Länge n

```
double a[3][X_DIM] ;
```

Vektor 0 untere Nebendiagonale, Vektor 1 Hauptdiagonale und Vektor 2 obere Nebendiagonale

Speicherbedarf in beiden Fällen $3 \cdot n \cdot d$ bei $3 \cdot n - 2$ Nichtnullelementen

c) Speicherung ähnlich zu 2

sei $2 \cdot K$ Anzahl belegter Nebendiagonalen (+ zus. Hauptdiagonale) c1) Speicherung von $2 \cdot K + 1$ Elementen pro Zeile

```
double a [X_DIM][2·K+1] ;
```

c2) Speicherung von $2 \cdot K + 1$ Diagonalen der Länge n

```
double a [2·K+1][X_DIM] ;
```

Speicherplatzbedarf in beiden Fällen $(2 \cdot K + 1) \cdot n \cdot d$ bei $(2 \cdot K + 1) \cdot n - 2 \cdot n \cdot \sum_{k=1}^{K} 1/n_k$ Nichtnullelementen

d) Allgemeine spärlich besetzte Matrizen existieren im wesentlichen in 2 Strukturvarianten

d1) Fortran geprägt ohne Zeiger

Hier am Beispiel erläutert

$$A = \begin{pmatrix} 11 & 12 & 0 & 14 & 0 & 0 \\ 0 & 22 & 23 & 0 & 0 & 0 \\ 31 & 0 & 33 & 34 & 0 & 0 \\ 0 & 42 & 0 & 44 & 45 & 46 \\ 0 & 0 & 0 & 0 & 55 & 56 \\ 0 & 0 & 0 & 0 & 65 & 66 \end{pmatrix}$$

sei nz Anzahl Nichtnullelemente in der Matrix, im Beispiel 16 (also 44% besetzt, dies ist nicht dünn besetzt).

COO-Format: Koordinatenformat

Dies ist ein sehr einfaches Format, daß für jedes Nichtnullelement zwei Integer Werte zur Speicherung des Zeilen- und Spaltenindex, sowie ein Fließkommazahl zur Speicherung des Matrixeintrags verwendet. Die Werte werden in Arrays der Länge nz gespeichert. Für die gesamte Matrix müssen $2 \cdot nz$ Integer und nz Fließkommazahlen gespeichert werden. Nichtnullelemente werden in beliebiger Reihenfolge gespeichert.

Womit obiges Beipsiel z.B. wie folgt gespeichert werden könnte:

val	65	56	46	34	44	45	22	66	11	12	31	55	23	33	144	42
row	6	5	4	3	4	4	2	6	1	1	3	5	2	3	1	4
col	5	6	6	4	4	5	2	6	1	2	1	5	3	3	4	2

Damit erfordert das Suchen eines Elements einen Aufwand in $O(nz)$. Zeilen- oder spaltenweise Durchläufe durch die Matrix sind unmöglich, bzw. erforderten das Suchen jedes einzelnen Elements. Dies ändert sich auch nicht, wenn Elemente zeilen- oder spaltenweise sortiert gespeichert werden, da die Anzahl der Nichtnullelemente in einer Spalte oder Zeile nicht a priori bekannt ist. Das Einfügen von Elementen ist im COO-Format recht einfach, da neue Elemente nur hinten an das Array angehängt werden müssen.

C Deklaration:

double val [NZ] ;

int col [NZ], row [NZ] ;

CRS- und CCS-Format: Kompremierte Zeilen-/Spaltenspeicherung

Die beiden Formate unterscheiden sich dadurch, daß in einem Fall zeilenweise, im anderen Fall spaltenweise gespeichert wird. Wir betrachten hier die zeilenweise Speicherung. Dies entspricht der spaltenweisen Speicherung der transponierten Matrix. Werte der Nichtnullelemente und deren Spaltenindizes werden in zwei Arrays konsekutiv gespeichert werden. In einem zusäzlichen Integer Array `row` der Länge $n+1$ werden die Indizes des ersten Elements der korrespondierenden Zeile in den beiden anderen Arrays gespeichert. Im letzten Element des Arrays wird der Wert $nz+1$ gespeichert. Die Speicherstruktur wird am Beispiel deutlich.

val	11	12	14	22	23	31	33	34	42	44	45	46	55	56	65	66
col	1	2	4	2	3	1	3	4	2	4	5	6	5	6	5	6

row	1	4	6	9	13	15	17

Die Nichtnullelemente der Zeile `i` sind also an den Positionen `row[i]` bis `row[i+1]-1` zu finden. Damit ist ein zeilenweiser Durchlauf durch die Matrix leicht zu realisieren und auch der Zugriff auf ein Element wird deutlich schneller, da nur im Intervall [`row[i]`,`row[i+1]-1`] gesucht werden muß, wenn auf ein Element aus Zeile `i` zugegriffen wird. Ein spaltenweiser Durchlauf oder Modifikationen der Matrix sind allerdings aufwendig. Aufwendig ist auch das Einfügen neuer Elemente, da die gesmate Struktur zu modifizieren ist.

Zur Speicherung einer Matrix im CRS-Format werden jeweils ein Vektor der Länge nz mit Integer- und Real-Werten benötigt. Zusätzlich wird ein Integer-Vektor der Länge n gebraucht. Da normalerweise $n < nz$ gilt, benötigt das CRS-Format weniger Speicherplatz als das COO Verfahren.

C Deklaration:

```
double val [NZ] ;

int col [NZ], row [X_DIM] ;
```

d2) C- Version mit Zeigern

C-Deklaration

```
typedef struct {
    unsigned no_of_elements ;
    int *colind ;
    double *val ;
} Row ;
```

```
typedef struct {
    unsigned dim ;
    Row **rowind ;
} Sparse_matrix ;
Sparse_matrix *a ;
```

Speicherplatz allokieren:

```
Sparse_matrix *new_sparse_matrix (int dim, int *entries)
{
    int i;
    Sparse_matrix *a ;

    a = (Sparse_matrix *) calloc (1,sizeof(Sparse_matrix)) ;
    a->dim = dim ;
    a->rowind = (Row *) calloc(dim, sizeof(Row *)) ;
    for (i = 0; i < dim; i++) {
       no_elements = entries[i] ;
       a->rowind[i]->colind = (int *) calloc(entries[i], sizeof(int)) ;
       a->rowind[i]->val = (double *) calloc(entries[i], sizeof(double)) ;
    }
    return(a) ;
}
```

Speicherbedarf $nz \cdot d + nz \cdot i + nz * adr$

zu 3)

Annahme quadratische Matrix mit $n = m$

a) Aufwand dichtbesetzte Matrix

Transponieren keine neue Datenstruktur notwendig

```
for (i = 0; i < n; i++)
for (j = i+1; j < n; j++) {
    h = a[i,j] ;
    a[i,j] = a[j,i] ;
    a[j,i] = h ;
}
```

$3 \cdot n \cdot (n-1)/2$ Zuweisungen

Multiplikation mit einem Vektor in eine neue Vektorstruktur

```
for (i = 0; i < n; i++)
for (j = 0; j < n; j++) {
    v[i] += a[i,j] * w[j];
}
```

n^2 Additionen und n^2 Multiplikationen

Multiplikation mit Matrix in neue Matrixstruktur

```
for (i = 0; i < n; i++)
for (j = 0; j < n; j++)
for (k = 0; k < n; k++) {
    c[i][k] += a[i][j] * b[j][k];
}
```

n^3 Additionen und n^3 Multiplikationen

b) Aufwand Bandmatrix

Transponieren (Struktur b1)

```
for (i = 0; i < n-1; i++) {
    h = a[i][2] ;
    a[i][2] = a[i+1][0] ;
    a[i+1][2] = h ;
}
```

$3 \cdot (n-1)$ Zuweisungen

Transpoinieren Struktur b2

```
for (i = 1; i < n; i++)
    a[0][i-1] = a[0][i] ;
for (i = n-1; i >= 0; i--)
    a[2][i] = a[2][i+1] ;
h = a[0] ;
```

a[0] =a[2] ;

a[2] = h ;

$2 \cdot (n-1) + 3$ Zuweisungen

Multiplikation mit Vektor in neue Vektorstruktur (Variante b1, b2 ähnlich)

```
for (i = 1; i < n-1; i++)
for (j = 0; j < 3 ; j++) {
    v[i] += a[i][j] * w[i-1+j];
```

}

v[0] = a[0][1]*w[0] + a[0][2]*w[1] ;

v[n-1] = a[n-1][0]*w[n-2] + a[n-1][1]*w[n-1] ;

$3n-2$ Additionen und $3n-2$ Multiplikationen (+ Adressberechnungen)

Multiplikation mit Matrix in neue Matrixstruktur bei der vier Nebendiagonalen Nichtnullelemente beinhalten!

=> Transformation in andere Darstellung!

c) Aufwand Pentadiagonalmatrizen

Transponieren, Struktur bleibt erhalten (hier Struktur c1, Struktur c2 ähnlich zu b2)

sei `nk` Array mit `nk[i]=`$\prod_{k=i}^{K} n_k$

```
for (i = 0; i < n-1; i++)
for (j = 0; j < K; j++)
if (i > nk[j+1]) {
      h = a[i][j] ;
      a[i][j] = a[i-nk[j+1]][j+K+1] ;
      a[i-nk[j+1]][j+K+1] = h ;
}
```

$3/2 \cdot ((2 \cdot K + 1) \cdot n - 2 \cdot n \cdot \sum_{k=1}^{K} 1/n_k)$ Zuweisungen (+ Adressberechnungen)

Multiplikation mit Vektor neue Vektorstruktur (Variante c1, c2 ähnlich)

```
for (i = 0; i < n-1; i++) {
   for (j = 0; j < K; j++)
   if (i > nk[j+1])
    v[i] += a[i][j]*w[i-nk[j+1]];
    v[i] += a[i][K] * w[i] ;
   for (j = K+1; j <= 2*K+1; j++)
   if (i + nk[j-K+1] < n)
    v[i] += a[i][j]*w[i+nk[j-K+1]];
}
```

$((2 \cdot K + 1) \cdot n - 2 \cdot n \cdot \sum_{k=1}^{K} 1/n_k)$ Additionen und Multiplikationen (+ Adressberechnungen)

Multiplikation mit Matrix neue Matrixstruktur bei der vier Nebendiagonalen Nichtnullelemente beinhalten!

=> i.d.R. Transformation in andere Darstellung!

d) Aufwand allgemeine spärlich besetzte Matrix (hier d1 und d3)

Transponieren

d1) ein Durchlauf vertauschen Zeieln und Spaltenindex

d2) neue Datenstruktur

Vorgehen relativ komplex!

sei `pos` integer Array der Länge n mit 0 initialisiert

Bestimmung der Anzahl der Nichtnullelemente

```
for (i = 0; i < n; i++)
for (j = 0; j < a->rowind[i]->no_of_entries; j++)
    pos[a->rowind[i]->colind[j]]++ ;
```

Aufwand nz Additionen

Generierung einer neuen Matrix

```
at = new_sparse_matrix (n, pos) ;
```

Aufwand $O(n)$

Aufbau der Matrix

```
for (i = 0; i < n; i++)
   pos[i] = 0 ;
for (i = 0; i < n; i++)
for (j = 0; j < a->rowind[i]->no_of_entries; j++) {
    k = a->rowind[i]->colind[j];
    at->rowind[k]->colind[pos[k]] = i ;
    at->rowind[k]->val[pos[k]] = a->rowind[i]->val[j] ;
    pos[k]++ ;
}
```

Aufwand $3nz$ Zuweisungen und nz Additionen

Multiplikation mit Vektor neue Vektorstruktur (Variante d1)

```
for (i = 0; i < nz; i++)
   w[row[i]] += val[i]*v[col[i]] ;
```

Aufwand nz Multiplikationen und Additionen

Multiplikation mit Vektor neue Vektorstruktur (Variante d3)

```
for (i = 0; i < n; i++)
for (j = 0; j < a->rowind[i]->no_of_entries; j++) {
    w[i] += a->rowind[i]->val[j]*v[a->rowind[i]->colind[j]];
```

Aufwand nz Multiplikationen und Additionen (+ Adressberechnungen)

Matrixmultiplikation mit d1 realisterischerweise nicht möglich

Matrixmultiplikation mit d3 schwierig (eine Variante)

Vektor dim Dimension n mit 0 initialisiert

```
bt = mat_transpose(b) ;
for (i = 0; i < n; i++)
for (j = 0; j < n; j++) {
     i1 = 0 ;
     i2 = 0 ;
     while (i1 < a->rowind[i]->no_of_entries && i2 < bt->rowind[j]->no_of_entries) {
         if (a->rowind[i]->colind[i1] == bt->rowind[j]->colind[i2]) {
             dim[i]++ ;
             break ;
         }
         if (a->rowind[i]->colind[i1] < bt->rowind[i]->colind[i2])
             i1++ ;
         if (a->rowind[i]->colind[i1] > bt->rowind[i]->colind[i2])
             i2++ ;
     }
}
c = new_sparse_matrix(n,dim) ;
for (i = 0; i < n; i++)
    dim[i] = 0 ;
for (i = 0; i < n; i++) {
    k = 0 ;
    for (j = 0; j < n; j++) {
        i1 = 0 ;
        i2 = 0 ;
        found = 0 ;
        while (i1 < a->rowind[i]->no_of_entries && i2 < bt->rowind[j]->no_of_entries) {
             if (a->rowind[i]->colind[i1] == bt->rowind[j]->colind[i2]) {
                found++ ;
                c->rowind[i]->colind[k] = j ;
                c->rowind[i]->val[k] += a->rowind[i]->val[i1] * bt->rowind[j]->val[i2] ;
            }
```

```
            if (a->rowind[i]->colind[i1] < bt->rowind[i]->colind[i2])
                i1++ ;
            if (a->rowind[i]->colind[i1] > bt->rowind[i]->colind[i2])
                i2++ ;
        }
        if (found > 0)
            k++ ;
        if (k >= c->rowind[i]->no_of_entries)
            break ;
    }
}
```

Aufgabe 19

Parallelisierung auf einem Bussystem

Ziele

- gleichmäßige Verteilung der Arbeit
- geringe Kommunikationslast
- geringer Speicherplatzbedarf pro Prozess

Algorithmus für dichtbesetzte Strukturen kann übernommen werden. Bei spärlich besetzten Matrizen ist auf eine gleichmäßige Lastverteilung zu achten. D.h. ein Prozessor erhält nicht mehr $\approx n/p$ Zeilen, sondern $\approx nz/p$ Nichtnullelement zugewiesen.

Parallelisierung auf einem System mit verteiltem Speicher (z.B. Hyperwürfel)

a) Nur Nebendiagonalen und Diagonalen enthalten Nichtnullelemente

zeilenweise Speicherung der Matrix auf p Prozessoren

$\Rightarrow$ jeder Prozessor bekommt $k = \lceil n/p \rceil$ Zeilen zugewiesen

Annahme: k sei ganzzahlig

Prozessor benötigt Vektor c_local der Länge $k+2$ (bzw. $k+1$ falls erster oder letzter Prozessor)

Schritte des Algorithmus für Prozessor i

- Multiplikation Matrix - Vektor wobei Elemente 0 und $k+1$ initial 0 sind
- Verschicken
  - von Element $c\_local[0]$ an Prozessor mit Index $i-1$
  - von Element $c\_local[k+1]$ an Prozessor mit Index $i+1$

- Empfangen
  - von $c1$ von Prozessor $i-1$
  - von ck von Prozessor $i+1$
- Addition $c\_local[1]+=c1$ und $c\_local[k]+=ck$

Algorithmus für Prozessor $pi \in \{1,\dots,p\}$

- $A\_local[i][j]$ der Größe $k \times k$
- $b[k+2], bp[k+2]$

Lokaler Algorithmus auf Prozessor p

for ($i=0$;$i<k+2$; $i++$)
  $bp[i]=0.0$;
for ($i=0$; $i<k$; $i++$)
  for ($j=0$; $j<3$; $j++$)
    $bp[i]+=A\_local[i][j]*b[i-1+j]$;
    exchange(b, bp) ; /* tausche Pointer */
if ($pi>1$)
  send($b[0]$,$p-1$) ;
if ($pi<p$)
  send($b[k+1]$,$pi+1$) ;
if ($pi>1$) {
  receive(&$b[0], p-1$) ;
  $b[1]+=b[0]$; }
if ($pi<p$) {
  receive(&$b[k+1], pi+1$) ;
  $b[k]+=b[k+1]$; }

Ähnlich für spaltenweise Speicherung

b) Pentadiagonalmatrizen

Pentadiagonalmatrizen resultieren aus der Diskretisierung mehrdimensionaler Strukturen.

Darstellung der Struktur (i.d.R. gitterförmig) und zusammenfassen benachbarter Elemente (Bild!).

Hier am Beispiel zweidimensionaler Gitter, jeder Punkt kann mehrdimensional beschrieben sein!

Sei zur Vereinfachung die x- und y-Dimension gleich N und sei $NI = N/\sqrt{p}$. Prozessoren bekommen Nummer (px, py) mit $1 \le px, py \le \sqrt{p}$.

Prozessor (px, py) speichert Elemente $A[i][j][k][l]$ mit $(px-1)\cdot NI \leq i < px \cdot NI$, $(py-1)\cdot NI \leq i < py \cdot NI$, $(px-1)\cdot NI - 1 \leq i \leq px \cdot NI$ und $(py-1)\cdot NI - 1 \leq i \leq py \cdot NI$. Grenzen werden in den beiden letzten Fällen für Prozessoren am Rand nicht benötigt.

Lokale Datenstrukturen

- Matrix $A\_local[NI][NI][5]$ (0 Diagonale, 1 loben, 2 rechts, 3 unten, 4 links)
- Vektoren $b[NI+2][NI+2]$, $bp[NI+2][NI+2]$, $c[NI]$

Schritte des Algorithmus für Prozessor px, py

- Multiplikation Matrix - Vektor wobei Elemente $0, y$ und $x, 0$ initial 0 sind
- Verschicken
  - von Vektor $b[0][y]$ an Prozessor $px, py-1$
  - von Vektor $b[NI+1][y]$ an Prozessor $px, py+1$
  - von Vektor $b[x][0]$ an Prozessor $px-1, py$
  - von Vektor $b[x][NI+1]$ an Prozessor $px+1, py$
- Empfangen
  - von Vektor $b[0][y]$ an Prozessor $px, py-1$
  - von Vektor $b[NI+1][y]$ an Prozessor $px, py+1$
  - von Vektor $b[x][0]$ an Prozessor $px-1, py$
  - von Vektor $b[x][NI+1]$ an Prozessor $px+1, py$
- Addition $b[1][y]+=b[0][y]$, $b[NI][y]+=b[NI][y]$, $b[x][1]+=b[x][0]$ und $b[x][NI]+=b[x][NI+1]$

Lokaler Algorithmus auf Prozessor px, py

for $(i = 0; i < NI + 2; i++)$
    for $(j = 0; j < NI + 2; j++)$
        $bp[i][j] = 0.0;$
for $(i = 1; i < NI + 1; i++)$
    for $(j = 1; j < NI + 1; j++)$ {
        $bp[i][j] += b[i][j] * A\_local[i][j][0]$
        if $(px > 1)$
            $bp[i-1][j] += b[i][j] * A\_local[i][j][1]$;
        if $(py < NI)$

$bp[i][j+1]+=b[i][j]*A\_local[i][j][2]$;

if $(px < NI)$

$bp[i+1][j]+=b[i][j]*A\_local[i][j][3]$;

if $(py > 1)$

$bp[i][j-1]+=b[i][j]*A\_local[i][j][4]$; }

exchange (b, bp) ;

if $(px > 1)$

send($b[0]$,$px-1$, py) ;

if $(px < NI)$

send($b[NI]$,$px+1$, py) ;

if $(py > 1)$ {

for $(i = 0; i < NI+2; i++)$

$c[i] = b[i][0]$;

send(c,px, $py-1$) ; }

if $(py < NI)$ {

for $(i = 0; i < NI+2; i++)$

$c[i] = b[i][NI+1]$;

send(c,px, $py+1$) ; }

if $(px > 1)$ {

receive(&c,$px-1$, py) ;

for $(i = 1; i < NI+1; i++)$

$b[1][i] = c[i]$; }

if $(px < NI)${

receive(&c,$px+1$, py) ;

for $(i = 1; i < NI+1; i++)$

$b[NI][i] = c[i]$; }

if $(py > 1)$ {

receive(&c,px, $py-1$) ;

for $(i = 1; i < NI+1; i++)$

$b[i][0] = c[i]$; }

if $(py < NI)${

receive(&c,px, $py+1$) ;

for $(i = 1; i < NI+1; i++)$

$b[i][NI] = c[i]$; }

Ähnlich bei anderen Diskretisierungen

c) Allgemeine spärlich besetzte Matrizen

Problem deutlich komplexer, da nicht klar ist wer mit wem kommuniziert

Einfachstes Vorgehen, blockweise Verteilung wie bei dicht-besetzter Matrix

Problem: u.U. ungleiche Lastverteilung

Nächster Ansatz, blockweise Verteilung, so dass Zahl der Nichtnullelemente pro Prozessor ungefähr gleich groß

Problem: u.U. viel mehr Kommunikation als notwendig, da viele Prozessoren miteinander kommunizieren wobei u.U. nur einzelne Werte verschickt werden.

Bester Ansatz: Zuteilung der Zeilen (oder Spalten), so dass Last ungefähr gleich verteilt und möglichst wenige Prozessoren miteinander kommunizieren.

Vorgehen: Versuch der Minimierung der Bandbreite der Matrix (d.h. des maximalen Abstands von Nichtnullelementen von der Hauptdiagonalen)

Exakte Lösung des Problems ist NP-hart, aber zahlreiche Approximationen existieren, die passable Ergenisse liefern (insbesondere, wenn sie mit Kenntnissen über die Systemstruktur verbunden werden können).

Ergebnis der Bandbreitenminimierung: Matrix in Untermatrizen unterteilt

wobei $A[I, J]$ eine Untermatrix ist

die Zahl der Nichtnullelemente in den Matrizen $A[I, J]$ $(J = 1, \ldots, N)$ für alle I ungefähr gleich ist

Algorithmus:

sei N_I Anzahl Elemente für Prozessor I $(I = 1, \ldots, N)$

Datenstrukturen für Prozessor I:

Vektoren $b[N_I]$, $bp[N^I_{\max}]$ mit $N^I_{\max} = \max_{J=1,\ldots,N, A[I][J] \neq 0}(N_J)$

Matrizen $A[I][J]$ der Dimension $N_I \times N_J$

for $(J = 1;\ J <= N;\ J++)$

if $(I! = J \&\& A[I][J]! = NULL\ )$ {

$bp = mult(A[I][J], b)$;

send(bp, J) ; }

$bp = mult(A[I][J], b)$;

copy_vektor(bp, b, 0, N_I) ; /* kopiert Elemente $0, \ldots, N_I - 1$ von bp nach b */

for $(J = 1;\ J <= N;\ J++)$

if $(I! = J \&\& A[J][I]! = NULL\ )$ {

receive(&bp, J) ;

add_vektor(bp, b, 0, N_I) ; /* addiert Elemente $0, \ldots, N_I - 1$ von bp zu b */}

Realisierung auf Hyperwürfel durch Einbettung der Strukturen in den Hyperwürfel $\Rightarrow$ falls dies gelingt, Kommunikation nur zwischen benachbarten Prozessoren