Implementation of parallel Krylov space algorithms

Uwe Küster

University of Stuttgart
High-Performance Computing Center Stuttgart (HLRS)
www.hlrs.de

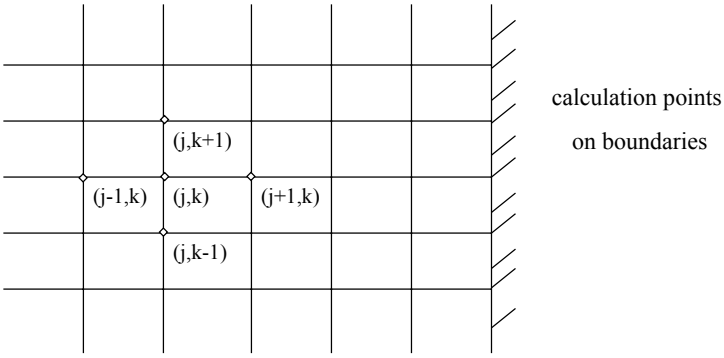
overview

- motivation
- examples of Krylov space algorithms
- building blocks
- domain decomposition as parallelization approach
- CG program code
- vector class
- sparse matrix class
- exchange mechanism
- work definition, load balancing, load distribution
- communication setup

overview

- **motivation**
- examples of Krylov space algorithms
- building blocks
- domain decomposition as parallelization approach
- CG program code
- vector class
- sparse matrix class
- exchange mechanism
- work definition, load balancing, load distribution
- communication setup

Finite Differences 1
(Approximation of Differential Operators by Differences)



Finite Differences 2

$$(\Delta\phi)_{jk} = \frac{1}{\Delta x^2}\left(\Phi_{j-1k} - 2\Phi_{jk} + \Phi_{j+1k}\right) + \frac{1}{\Delta y^2}\left(\Phi_{jk-1} - 2\Phi_{jk} + \Phi_{jk+1}\right)$$

- diskrete Laplace operator on an equidistant rectangular grid
- can be represented as a product of a matrix with the vector of all states
- coefficients vary on location and may vary on solution
- the matrix will be stored in an explicit way
- example of (implicit) local neighbourhood
  - typical for implementations of other physical / technical models

Finite Differences 2

$$(\Delta\phi)_{jk} = \frac{1}{\Delta x^2}\left(\Phi_{j-1k} - 2\Phi_{jk} + \Phi_{j+1k}\right) + \frac{1}{\Delta y^2}\left(\Phi_{jk-1} - 2\Phi_{jk} + \Phi_{jk+1}\right)$$

- diskrete Laplace operator on an equidistant rectangular grid
- can be represented as a product of a matrix with the vector of all states
- coefficients vary on location and may vary on solution
- the matrix will be stored in an explicit way
- example of (implicit) local neighbourhood
  - typical for implementations of other physical / technical models

overview

- motivation
- **examples of Krylov space algorithms**
- building blocks
- domain decomposition as parallelization approach
- CG program code
- vector class
- sparse matrix class
- exchange mechanism
- work definition, load balancing, load distribution
- communication setup

conjugate gradient procedure for SPD matrices

projection method

solution is searched in Krylov space

$$r_0 = b - Ax$$

$$U_m = span\left\{ r_0, Ar_0, \ldots, A^{m-1} r_0 \right\} = span\left\{ r_0, r_1, \ldots, r_{m-1} \right\}$$

$$r_{m+1} - r_m \in AU_{m+1}$$

$$r_{m+1} \perp U_{m+1}$$

conjugate gradient procedure for SPD matrices

starting values

$$x_0 = b$$

$$v_0 = Ax_0 \qquad \leftrightarrow$$

$$r_{m+1} = b - v_0$$

$$p_0 = r_0$$

$$\alpha_0 = \|r_0\|_2^2 \qquad \leftrightarrow$$

$$do\ m = 0, n-1$$

$$v_m = Ap_m \qquad \leftrightarrow$$

$$\lambda_m = \frac{\alpha_m}{(v_m, p_m)_2} \qquad \leftrightarrow$$

$$x_{m+1} = x_m + \lambda_m p_m$$

$$r_{m+1} = r_m - \lambda_m v_m$$

$$\alpha_{m+1} = \|r_{m+1}\|_2^2 \qquad \leftrightarrow$$

$$p_{m+1} = r_{m+1} + \frac{\alpha_{m+1}}{\alpha_m} p_m$$

$\leftrightarrow$ data exchange between processors

preconditioning

- convergence properties depend on the condition of the matrix
- reduce condition by preconditioning

$$\hat{b} = Lb$$

$$LAU\hat{x} = b$$

$$x = U\hat{x}$$

- scaling the diagonal
- incomplete LU factorization on the matrix pattern
- multilevel

Conjugate Gradient Squared (CGS)

start values

$r_0 = L^{-1} A x_0 - b \quad \leftrightarrow$

$\bar{r}_0 = ? \quad \leftrightarrow$

$q_0 = 0$

$p_{-1} = 0$

$\rho_{-1} = 1$

Iteration

$do\ k = 1, k\max$

$\rho_k = \langle \bar{r}_0, r_k \rangle \quad \leftrightarrow$

$\beta_k = \rho_k / \rho_{k-1}$

$u_k = r_k + \beta_k q_k$

$p_k = u_k + \beta_k (q_k + \beta_k p_{k-1})$

$v_k = L^{-1} A U^{-1} p_k \quad \leftrightarrow$

$\sigma_k = \langle \bar{r}_0, v_k \rangle \quad \leftrightarrow$

$\alpha_k = -\rho_k / \sigma_k$

$q_{k+1} = u_k + \alpha_k v_k$

$w_k = \alpha_k U^{-1} (u_k + q_{k+1}) \quad \leftrightarrow$

$r_{k+1} = r_k + L^{-1} A w_k \quad \leftrightarrow$

$x_{k+1} = x_k + w_k$

$enddo$

$\leftrightarrow$ data exchange between processors

more complicated BiCGSTAB(2)

starting values

x_0 *is an initial guess*; $r_0 = b - A x_0$;

$\bar{r}_0$ *is an arbitrary vector, such that* $(r, \bar{r}_0) \neq 0$,

e.g. $\bar{r}_0 = r$;

$\rho_0 = 1; u = 0; \alpha = 0; \omega_2 = 1;$

even Bi-CG step

$for\ i = 0,2,4,6,\cdots$

$\rho_0 = -\omega_2 \rho_0$

$\rho_1 = (\bar{r}_0, r_i) \quad \leftrightarrow$

$\beta = \alpha\, \rho_1 / \rho_0; \rho_0 = \rho_1$

$u = r_i - \beta u;$

$v = Au \quad \leftrightarrow$

$\gamma = (v, \bar{r}_0) \quad \leftrightarrow$

$\alpha = \rho_0 / \gamma;$

$r = r_i - \alpha v;$

$s = Ar \quad \leftrightarrow$

$x = x_i + \alpha u;$

odd Bi-CG step

$\rho_1 = (\bar{r}_0, s) \quad \leftrightarrow$

$\beta = \alpha\, \rho_1 / \rho_0; \rho_0 = \rho_1$

$v = s - \beta v;$

$w = Av \quad \leftrightarrow$

$\gamma = (w, \bar{r}_0) \quad \leftrightarrow$

$\alpha = \rho_0 / \gamma;$

$u = r - \beta u$

$r = r - \alpha v$

$s = s - \alpha w$

$t = As \quad \leftrightarrow$

GCR(2) - part

$\omega_1 = (r,s); \mu = (s,s); v = (s,t); r = (t,t)$

$\omega_2 = (r,t) \quad \leftrightarrow$

$r = r - v^2 / \mu;$

$\omega_2 = (\omega_2 - v\omega_1 / \mu) / r;$

$\omega_1 = (\omega_1 - v\omega_2) / \mu$

$x_{i+2} = x + \omega_1 r + \omega_2 s + \alpha u$

$r_{i+2} = r - \omega_1 s - \omega_2 t$

$if\ x_{i+2}\ accurate\ enough\ then\ quit$

$u = u - \omega_1 v - \omega_2 w$

end

$\leftrightarrow$ data exchange between processors

Krylov space algorithms

CG

Lanczos

BiCO

CGS

BiCGSTAB(l)

TFQMR

ORTHOMIN

GMRES

GMRESR

all have the same buildung blocks

overview

- motivation
- examples of Krylov space algorithms
- **building blocks**
- domain decomposition as parallelization approach
- CG program code
- vector class
- sparse matrix class
- exchange mechanism
- work definition, load balancing, load distribution
- communication setup

building blocks of Krylov space algorithms

- long vectors are used
- some scalar operations
- scalarproduct / modulus of vectors
- a = b + alpha*c (daxpy)
- vector = matrix * vector (time critical)
- preconditioning (not handled here); may have severe influence on mechanism
- set up of the procedure
- set up of matrix (e.g. Finite Elements)
- all these operations may be vectorized / parallelized

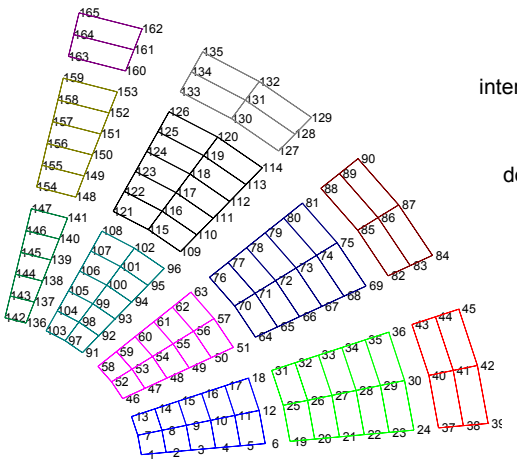
overview

- motivation
- examples of Krylov space algorithms
- building blocks
- **domain decomposition as parallelization approach**
- CG program code
- vector class
- sparse matrix class
- exchange mechanism
- work definition, load balancing, load distribution
- communication setup

parallelization approach

- most examples arize from discretized partial differential equations
- suitable approach is domain decomposition
- decompose complete domain in
  - non-overlapping domains (not handled here)
  - overlapping domains
- overlapping domains have ghost/slave/halo points to simplify the algorithm
- the values at the halo points have o be updated by the values of the master points by communication
- set up of discretization / Finite Elements may be handled on the same decomposition

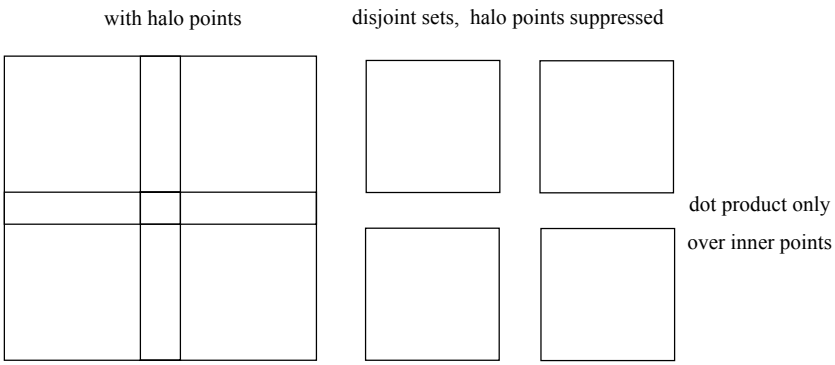
15 x 11 grid on 3 x 4 domains



inter domain edges suppressed

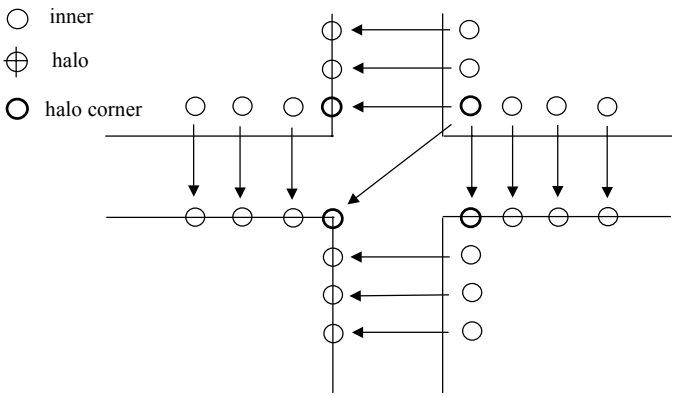
domains of different sizes

disjoint sets of points, halo, dot product



local matrices acting on inner points, but addressing the halo

data exchange near a corner



data are send from the related inner points to the halo of the neighbours

multiple sends may be necessary

overview

- motivation
- examples of Krylov space algorithms
- building blocks
- domain decomposition as parallelization approach
- **CG program code**
- vector class
- sparse matrix class
- exchange mechanism
- work definition, load balancing, load distribution
- communication setup

conjugate gradient iteration code // changes for parallelization

```
for ( ii = 0; ii < itermax;  ii++) {
    rTr_old = rTr ;
  // halo of   pp  used by matrix   mat_A   has to be updated here
    matrix_vector(App, mat_A, pp);     // App = A p
    dotproduct(&dot, App, pp);         //  dot = < A p , p >      // global operation
     if ( sqrt(dot) < epsilon ) { break ; }
    lambda = rTr / dot;
    vector_update(xx,  pp,   lambda ) ;   //   x = x + lambda * p
    vector_update(rr, App, -lambda ) ;   //    r = r  - lambda * App
    dotproduct(&rTr, rr, rr);            // rTr = < r , r >          // global operation
    vector_update_r(pp, rr, rTr / rTr_old ) ;
  }
```

object oriented technique with fat objects (vector, sparse matrix)

all Krylov space algorithms may be formulated by this way

calling overhead decreases performance (inlining)

parallelized conjugate gradient iteration code

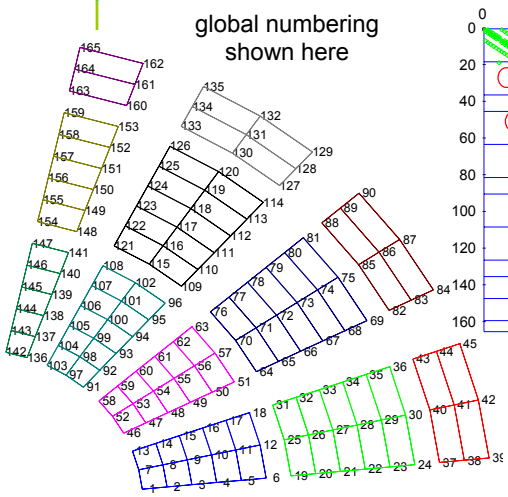
```
for ( ii = 0; ii < itermax;  ii++) {
    rTr_old = rTr ;
    exch_vector(&exch_data,  pp
        , to_buffer_values,          to_buffer_index
        ,    loc2glob_offset_from_array,  to_buffer_length_array
        , from_buffer_values,   from_buffer_index
        ,      from_buffer_offset_array, from_buffer_length_array );
    matrix_vector(App, mat_A, pp);
    dotproduct(&dot,  App,  pp);            // changed procedure
    if ( sqrt(dot) < epsilon ) {break ;}
    lambda = rTr / dot;
    vector_update(xx,   pp,  lambda ) ;
    vector_update(rr, App, -lambda ) ;
    dotproduct(&rTr, rr, rr);               // changed procedure
    vector_update_r(pp, rr, rTr / rTr_old ) ;
}
```

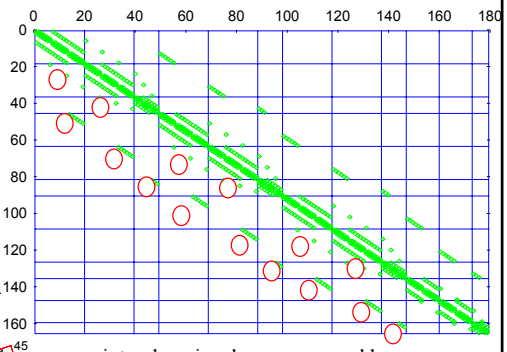
update halo

critical parts for parallelization of algorithm

- sparse matrix vector multiplication $y = Ax$
  - distributed over the domains
  - differentiate between local and global numbering
  - local representation by local indices $y_{local} = Ax_{local+halo}$
  - matrix may be handled without communication
  - only vector has to be communicated
  - halo has to be updated before
- dot product
  - dot product for inner points; exclude halo
  - sum over all domains
  - communication to all processors
  - MPI_Allreduce

15 x 11 grid on 3x4 domains





inter domain edges suppressed here

contigous global enumeration in domains

values to be communicated; upper part omitted

overview

- motivation
- examples of Krylov space algorithms
- building blocks
- domain decomposition as parallelization approach
- CG program code
- **vector class**
- sparse matrix class
- exchange mechanism
- work definition, load balancing, load distribution
- communication setup

data structure vector_type

- we assume a contiguous memory buffers, no linked list, no tree, no sparse array (performance breakdown)
- data access by array indices
- suitable for local numbering
- not reasonable global numbering
- fast by implementation with restrict qualifier (C99)
- procedures should be compiled with '-noalias' option

```
typedef struct {
        int length ;            // total length
        int inner_length ;      // length of the inner values
        double * values ;       // the values
            } vector_type;
```


vector_update: incrementing the left vector

```
void vector_update ( vector_type aa, vector_type bb, double  alpha ) {
  if   ( aa.length != bb.length) {
        printf ("PE%i:\t error in routine vector_update:\n", my_rank);
        ....
        exit_MPI(1);
        exit(1);
  }
   vector_update_kernel( aa.values, bb.values, alpha, aa.length) ;
}
```

error handling to be suppressed for production

inline call by hand for production; use retrict

```
void vector_update_kernel ( double * aa, double * bb, double  alpha, int length) {
      int i ;
      for (i = 0; i < length; i++) {
            aa[i] = aa[i] + alpha * bb[i] ;
      }
}
```

dotproduct of two vectors

```
void dotproduct( double * result, vector_type aa, vector_type bb){

      if (aa.length != bb.length){
        printf("PE%i:\t error in routine dotproduct:");
        printf("PE%i:\t aa.length= %d ",my_rank, aa.length);
        printf("PE%i:\t is different from ");
        printf("PE%i:\t bb.length= %d \n",my_rank, bb.length);
      exit_MPI(1);
        exit(1);
        }

  // length for vectors is aa.inner_length to exclude overlapping boundaries
    *result=dot_pro( aa.values, bb.values, aa.inner_length);
}
```

suppress second call

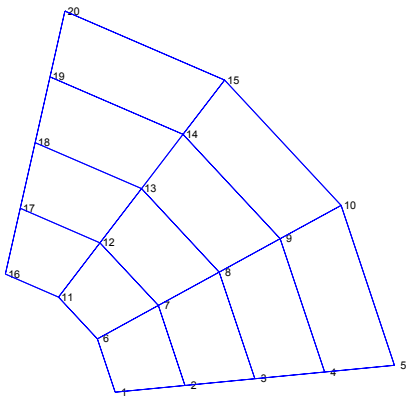
parallel dot_product

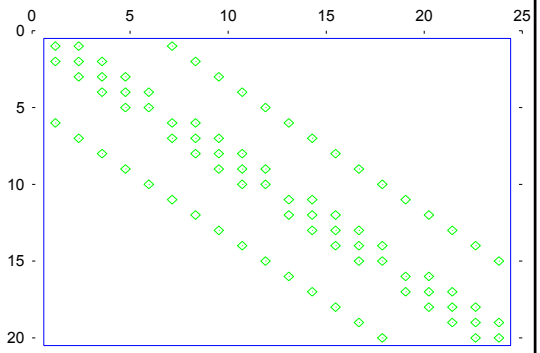
```
double dot_pro( double * values_1, double * values_2, int length )
 {
 int ii;
 int count=1;
 double sum;
 double received_sum;

    sum=0.;
 for (ii=0; ii< length ; ii++ ) {
    sum=sum+values_1[ii]*values_2[ii];   // local sum
 }

          MPI_Allreduce(&sum,&received_sum, count,
                        MPI_DOUBLE,MPI_SUM,comm_comm);

    return(received_sum);
 }
```

overview

- motivation
- examples of Krylov space algorithms
- building blocks
- domain decomposition as parallelization approach
- CG program code
- vector class
- **sparse matrix class**
- exchange mechanism
- work definition, load balancing, load distribution
- communication setup

5x4 grid with regular enumeration



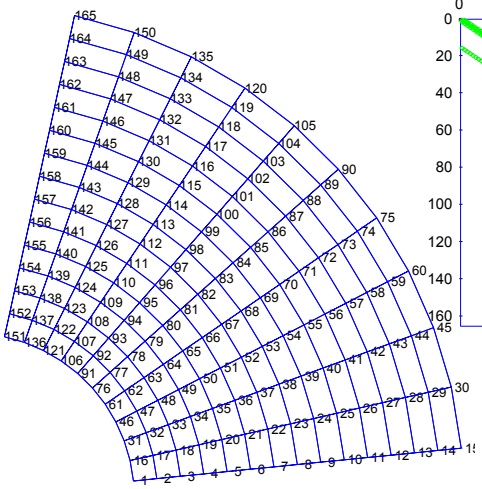


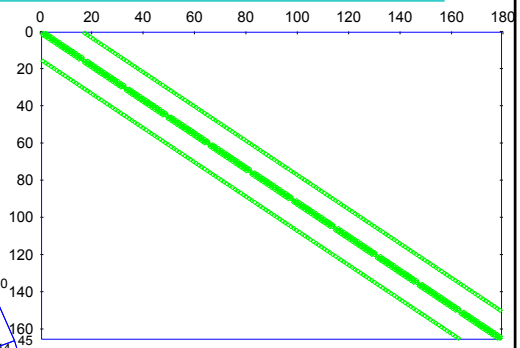
simplest instance of a matrix

implicit neighbourhood

lexicographic ordering

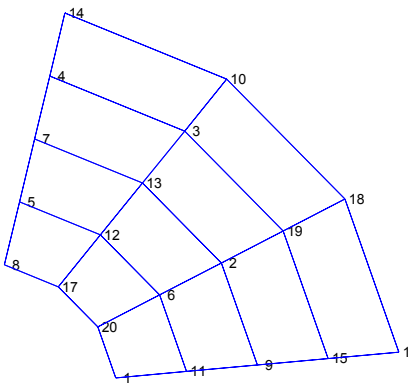
15x11 grid with regular enumeration

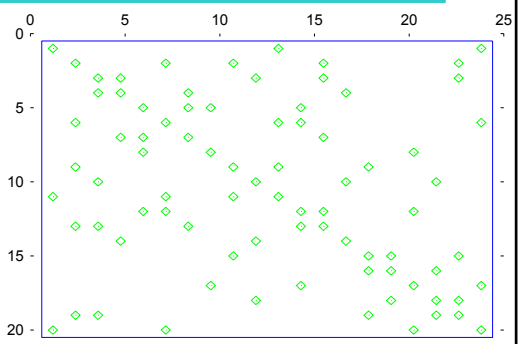




we see a relatively small bandwidth

5x4 irregulary enumerated grid and matrix



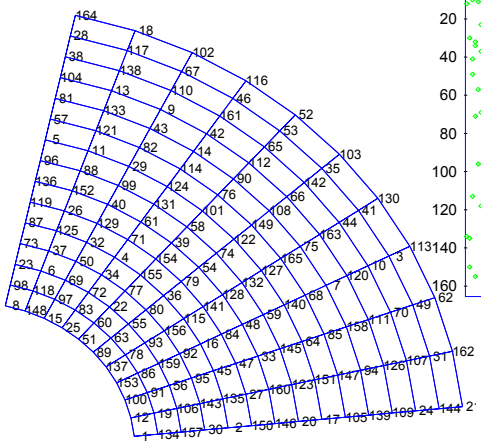


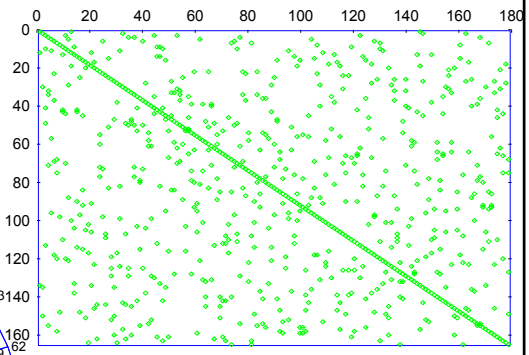
neighbourhood has to be stored explicitly

decrease of performance

flexible; covers all needs

irregularly enumerated 15x11 grid

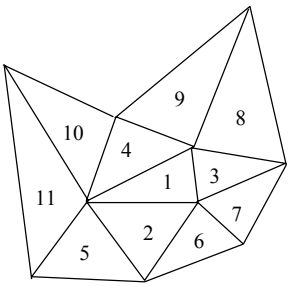




here 5 entries per row

neighbourhood has to be stored

triangle grid with relation matrix



triangles with common

sides are neighbours

no implicit neighbourhood possible

position of matrix elements

	1	2	3	4	5	6	7	8	9	10	11	Σ
1	•	•	•	•								4
2	•	•			•	•						4
3	•		•				•	•				4
4	•			•					•	•		4
5		•			•						•	3
6		•				•	•					3
7			•			•	•					3
8			•					•	•			3
9				•				•	•			3
10				•						•	•	3

why sparse matrix storage schemes

- avoid storing zero elements (but some of them may be stored)
- key data structure using Krylov space algorithms
- useful also for particle methods, general graphs
- fast under special circumstances
- general formulation of flexible neighbourhoods
- may replace recursive data structures as trees, linked lists
- if possible use small dense matrices of fixed size as elements instead of real numbers

matrix with zero elements

matrix

$$\begin{bmatrix} a_{11} & 0 & a_{13} & 0 \\ 0 & a_{22} & a_{23} & 0 \\ a_{31} & 0 & a_{33} & a_{34} \\ 0 & a_{42} & 0 & a_{44} \end{bmatrix}$$

values $[a_{11}, a_{13}, a_{22}, a_{23}, a_{31}, a_{33}, a_{34}, a_{42}, a_{44}]$

index $[\ 1\ ,\ 3\ ,\ 2\ ,\ 3\ ,\ 1\ ,\ 3\ ,\ 4\ ,\ 2\ ,\ 4\ ]$ points to column number

row_start ⇑ ⇑ ⇑ ⇑

sparse matrices: row ordered example

elements

0	5
1	11
2	7
3	8
4	2
5	4
6	16
7	9
8	12

number_of_row_elements

0	3
1	2
2	3
3	3
4	2
5	3
6	3
7	2
8	3

index

0	11	1	7	2	8
3	4	4	5		
5	5	6	16	7	8
8	5	9	7	10	2
11	8	12	12		
13	11	14	16	15	9
16	4	17	12	18	7
19	4	20	12		
21	2	22	16	23	9

working direction

rows may have different length

arrays beginning with 0

matrix elements organized as index array

data structure sparse_matrix_type

```
typedef struct {
        double * mat_aa ;                    //  matrix values
        int * mat_index ;                    // column address
        int * mat_elements ;                 // row address
        int * mat_number_of_row_elements ;   // number of elements of row
        int mat_number_of_rows ;             // number of rows
        int mat_number_of_entries ;          // total number of index entries
          } sparse_matrix_type;
```

- row ordered sparse matrix
- data access by array indices
- extra array for addressing elements for use with global numbering
- with small changes also direct addressing of elements
- with small changes also for jagged diagonal

sparse matrix vector multiplication, row ordered

```
void  sparse_matrix_vector_kernel ( double * yy
                        ,double * mat_aa, int * mat_index, int * mat_elements
                        ,int * mat_number_of_row_elements
                        ,int mat_number_of_rows, int mat_number_of_entries
                        ,double * xx  ) {
int ii, nn, kk, num, nn_max;
double sum;
                                        num = -1;
for( ii = 0 ; ii < mat_number_of_rows; ii++ ) {
          sum = 0; nn_max  =  mat_number_of_row_elements[ii];
 for( nn = 0;  nn <  nn_max ; nn++) {
   num = num+1; sum = sum + mat_aa[num] * xx[mat_index[num]];   // incrementing
 }                                                                a variable
          yy[mat_elements[ii]] = sum;
}
}
```


sparse matrix vector multiplication, row ordered, direct result access

```
void  sparse_matrix_vector_kernel ( double * yy
                        ,double * mat_aa, int * mat_index, int * mat_elements
                        ,int * mat_number_of_row_elements
                        ,int mat_number_of_rows, int mat_number_of_entries
                        ,double * xx  ) {
int ii, nn, kk, num, nn_max;
double sum;
                                        num = -1;
for( ii = 0 ; ii < mat_number_of_rows; ii++ ) {  sum = 0;
          nn_max  =  mat_number_of_row_elements[ii];
 for( nn = 0;  nn <  nn_max ; nn++) {   num = num+1;
          sum = sum + mat_aa[num] * xx[mat_index[num]];   incrementing a variable
 }
          yy[                ii  ] = sum;   // no index vector ;
                                            // if contigous access to data
}
}
```

sparse matrices: jagged diagonal example

row_number

0	5
1	7
2	8
3	4
4	16
5	12
6	11
7	2
8	9

pseudo_col

0	0
1	9
2	18
3	24

index

0	11	9	7	18	8
1	5	10	16	19	8
2	5	11	7	20	2
3	11	12	16	21	9
4	4	13	12	22	7
5	2	14	16	23	9
6	4	15	5		
7	8	16	12		
8	4	17	12		

working direction

end

0	8
1	17
2	23

may be useful

decreasing number of pseudo columns

arrays beginning with 0

matrix elements organized as index array

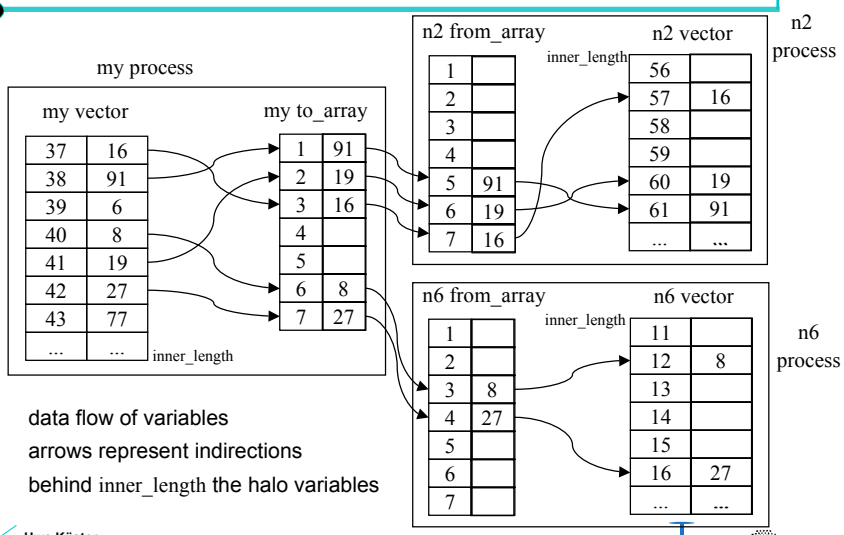
sparse matrix vector multiplication, jagged diagonal

```
 void sparse_matrix_vector_jagged_diagonal_kernel(
 double * yy, double * y_temp,double * mat_aa, int * mat_index,int * mat_elements
 ,int * pseudo_col,int * mat_number_of_row_elements,int mat_number_of_rows
 ,int mat_number_of_pseudo_columns,int mat_number_of_entries
 ,int *matrix_type,double * xx,int * total_number_of_operations) {
int ii,nn,kk,num,nn_max; double sum;
num=0;                                                    short loop
for( nn = 0 ; nn < mat_number_of_pseudo_columns; nn++ ) {
        ii=0;                                                     long loop
        for( kk = pseudo_col[nn]; kk < pseudo_col[nn+1] -1; kk++ ) {
                y_temp[ii] = y_temp[ii]+mat_aa[kk]*xx[mat_index[kk]];
                ii = ii+1;  }                                incrementing an array
        num=num+ii;
}                                          reordering  loop
 for( ii = 0 ; ii < mat_number_of_rows; ii++ ) {
        yy[mat_elements[ii]] = y_temp[ii];  }
*total_number_of_operations=2*num;}
```

overview

- motivation
- examples of Krylov space algorithms
- building blocks
- domain decomposition as parallelization approach
- CG program code
- vector class
- sparse matrix class
- **exchange mechanism**
- work definition, load balancing, load distribution
- communication setup

propagating the values by buffering



data flow of variables
arrows represent indirections
behind inner_length the halo variables

intermediate summary

- the iteration mechanism is completely described
- the formulation seems to be quite simple
- problems with preconditioning
- there is still one topic missing:
  set up of the communication mechanism

overview

- motivation
- examples of Krylov space algorithms
- building blocks
- domain decomposition as parallelization approach
- CG program code
- vector class
- sparse matrix class
- exchange mechanism
- **work definition, load balancing, load distribution**
- communication setup

How to set up the complete mechanism

- input of data
- output of data
- treatment of boundary conditions
- work definition
- load balancing
- load distribution
- set up of neighbourhood communication

treatment of Dirichlet boundary conditions

- input of data (not handled)
- output of data (not handled)
- **treatment of boundary conditions**
- work definition
- load balancing
- load distribution
- set up of neighbourhood communication

handling of Dirichlet boundary conditions

- boundary conditions may have a complicated pattern
- problem of defining the boundary conditions on the proper processor
- should be given in the global enumeration system
- the boundary values could be defined by a sparse array with global indices. This ensures independence on the special distribution.

work definition and load balancing

- input of data (not handled)
- output of data (not handled)
- treatment of boundary conditions
- **work definition**
- load balancing
- load distribution
- set up of neighbourhood communication

work definition

- the definition of the system is the main problem
  - set up of global matrix has to be done in parallel
  - has his own decomposition
  - complicated
    - **e.g. Finite Element Method**
      - set up of a lot of small matrices (simple to parallelize)
      - to be added to the stiffness matrix (data communication needed)
  - defines (implicitly or explicitly) the graph/grid of the problem
  - includes IO
    - **should also be parallel**
    - **parallel input/output of domains of the decomposition**
    - **halos to be calculated during the input procedure ensures independence on processor number**
- ideally the solver uses this domain decomposition; otherwise the initialization of the solver has to repartition the data
- still the problem of switching between local and global enumeration

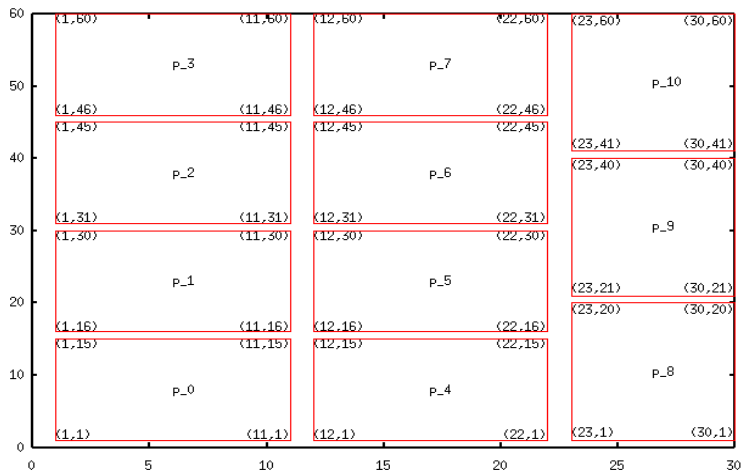
load balancing

- input of data (not handled)
- output of data (not handled)
- treatment of boundary conditions
- work definition
- **load balancing**
- load distribution
- set up of neighbourhood communication

load balancing

- load balancing to ensure an identical load for all processors in the
  - work definition phase
  - solving phase
  - domains should have identical sizes for equivalent processors

11 patches



load distribution

- input of data (not handled)
- output of data (not handled)
- treatment of boundary conditions
- work definition
- load balancing
- **load distribution**
- set up of neighbourhood communication

load distribution

- load distribution has to repartition the data
- may be complicate to program because all moved references (graph edges) are renumerated and the communication has to be redefined

communication setup

- input of data (not handled)
- output of data (not handled)
- treatment of boundary conditions (here Dirichlet)
- work definition
- load balancing
- load distribution
- **set up of neighbourhood communication**

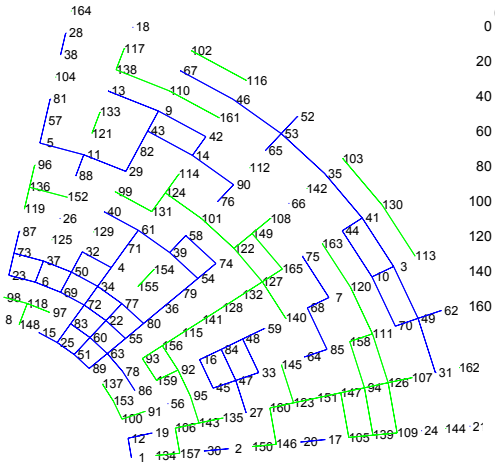
set up of neighbourhood communication: requirements

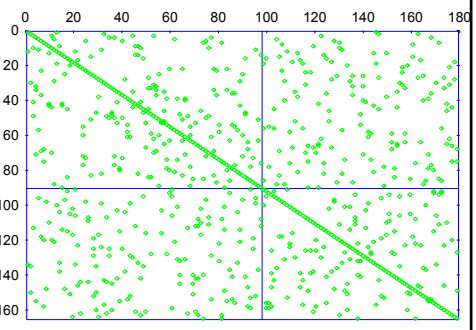
- data exchange mechanism should use information only based on the matrix related graph;
- the graph connectivity is given by the global enumeration
- a dense local enumeration in each domain will be needed; otherwise performance will be lost
- halo variables have to be defined in the local enumeration scheme consistent to the master variables in the other domain
- references from local to global enumeration have to be maintained; result has to be given in the global enumeration

simple setup mechanism

- PetSC approach:
  - partition the global enumeration in intervals of identical size solves load balancing problem
  - mark starting, ending points and process number and distribute this array to all processors (array has less than 10000 entries)
  - any processor recognizes the master process for any index by searching his interval in this array (by hashing)
  - the processors interval is simply to be transformed to a local enumeration by substracting the interval starting number

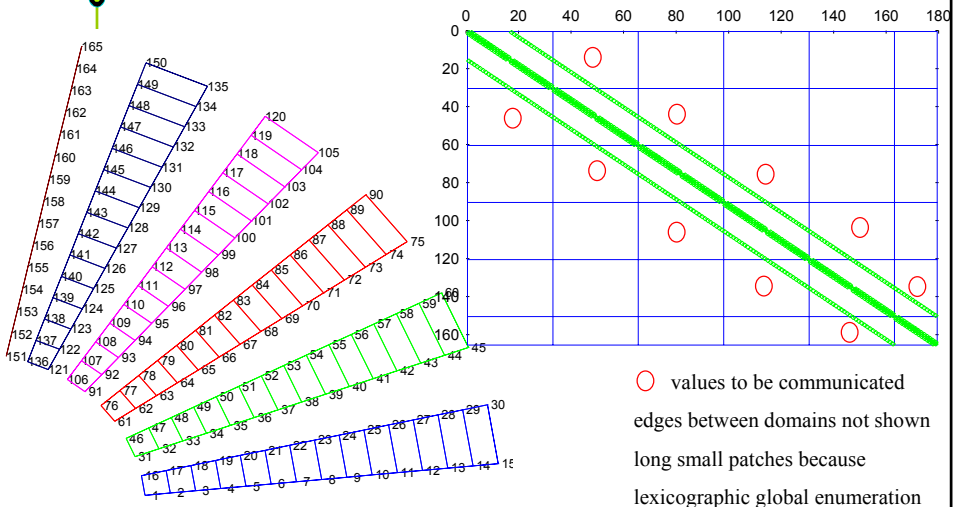
15 x 11 grid randomly enumerated with 2 domains





calculation/communication is small!

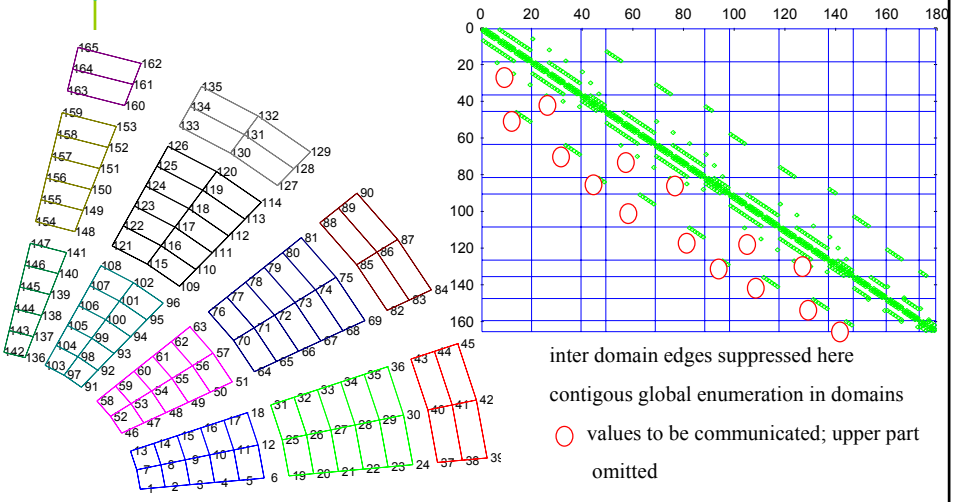
15 x 11 grid on 6 domains



simple setup mechanism

- the technique depends on the enumeration
  - lexicographical ordering generates long small domains
  - random ordering may generate non connected domains
- connected domains with small numbers for interdomains edges needed
- renumeration step has to be done before reducing the number of interdomain edges
- problem:
  each processor knows the owner processes of his halo data; but the owner does not know the neighbours with halo data;
  (one sided communication will help)
  otherwise a non scalable global communication step is needed;
  not necessary if graph/matrix pattern is symmetric

15 x 11 grid on 3 x 4 domains

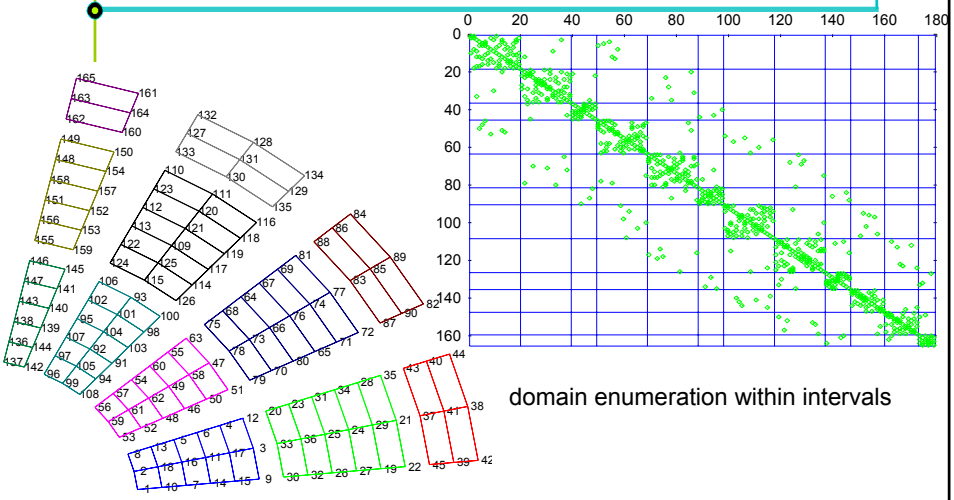


inter domain edges suppressed here

contigous global enumeration in domains

values to be communicated; upper part omitted

15 x 11 grid with irregulary enumerated 3x4 domains



domain enumeration within intervals

end