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)

Calculation points on boundaries
Finite Differences 2

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

- discrete 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

\[ r_0 = b - Ax \]
\[ U_m = \text{span}\{r_0, Ar_0, \ldots, A^{m-1}r_0\} = \text{span}\{r_0, r_1, \ldots, r_{m-1}\} \]
\[ r_{m+1} = r_m - A U_{m+1} \]
\[ r_{m+1} \perp U_{m+1} \]
conjugate gradient procedure for SPD matrices

starting values
\[ x_0 = b \]
\[ v_0 = Ax_0 \]
\[ r_{m+1} = b - v_0 \]
\[ p_0 = r_0 \]
\[ \alpha_0 = \|r_0\|^2 \]

\[ do \ m = 0, n - 1 \]
\[ v_m = Ap_m \]
\[ \lambda_m = \frac{\alpha_m}{(v_m, p_m)_2} \]
\[ x_{m+1} = x_m + \lambda_m p_m \]
\[ r_{m+1} = r_m - \lambda_m v_m \]
\[ \alpha_{m+1} = \|r_{m+1}\|^2 \]
\[ p_{m+1} = r_{m+1} + \frac{\alpha_{m+1}}{\alpha_m} p_m \]

\[ \leftrightarrow \] data exchange between processors

---

preconditioning

- convergence properties depend on the condition of the matrix
- reduce condition by preconditioning
  \[ \tilde{b} = Lb \]
  \[ LA\tilde{x} = b \]
  \[ x = U\tilde{x} \]
- scaling the diagonal
- incomplete LU factorization on the matrix pattern
- multilevel
Conjugate Gradient Squared (CGS)

\[ r_0 = L^{-1} Ax_0 - b \leftrightarrow \text{do } k = 1, k \text{ max} \]
\[ r_0 =? \leftrightarrow \rho_0 = \langle r_0, r_0 \rangle \leftrightarrow \alpha_0 = -\rho_0 / \sigma_0 \]
\[ q_0 = 0 \]
\[ p_0 = 0 \]
\[ rho = 1 \]
\[ \beta_0 = \rho_0 / \rho_{-1} \]
\[ u_0 = r_0 + \beta_0 q_0 \]
\[ p_0 = u_0 + \beta_0 (q_0 + \beta_0 p_{-1}) \]
\[ v_0 = L^{-1} A U^{-1} p_0 \]
\[ \sigma_0 = \langle r_0, v_0 \rangle \leftrightarrow \text{enddo} \]

\[ \leftrightarrow \text{data exchange between processors} \]

more complicated BiCGSTAB(2)

starting values \[ x_0 \text{ is an initial guess, } r_0 = b - Ax_0; \]
\[ r_0 \text{ is an arbitrary vector, such that } (r_0, r_0) \neq 0, \]
e.g., \[ r_0 = r; \]
\[ p_0 = \alpha_0 = 0; \]

even Bi-CG step
\[ \text{for } i = 0, 2, 4, \ldots \]  
\[ \alpha_i = \beta_i / \rho_i \]
\[ \beta_i = \alpha_i / \rho_i; \rho_i = \rho_i \]
\[ \beta_i = \gamma_i / \rho_i; \gamma_i = \gamma_i \]
\[ v_i = A v_i \]
\[ \gamma_i = (v_i, r_i) \leftrightarrow u_i = r_i - \beta_i v_i \]
\[ \alpha_i = \beta_i / \gamma_i \]
\[ r_i = r_i - \alpha_i v_i \]
\[ s_i = A s_i \leftrightarrow t_i = A t_i \leftrightarrow \text{end} \]

\[ \leftrightarrow \text{data exchange between processors} \]
Krylov space algorithms

- CG
- Lanczos
- BiCO
- CGS
- BiCGSTAB(l)
- TFQMR
- ORTHOMIN
- GMRES
- GMRESR

All have the same building 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
- scalar product / 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 arise 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 to 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

with halo points  disjoint sets, halo points suppressed

dot product only over inner points

local matrices acting on inner points, but addressing the halo

data exchange near a corner

inner  halo  halo 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

```c
for ( ii = 0; ii < itermax; ii++) {
    rTr_old = rTr;
    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
parallelized conjugate gradient iteration code

```c
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 ) ;
}
```

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
  - 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
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

---

15 x 11 grid on 3x4 domains

inter domain edges suppressed here
contiguous global enumeration in domains
values to be communicated; upper part omitted
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

```c
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

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

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] ;
    }
}
```

error handling to be suppressed for production
inline call by hand for production; use restrict
**dotproduct of two vectors**

```c
void dotproduct( double * result, vector_type aa, vector_type bb) {
    if (aa.length != bb.length) {
        printf("PE%i: error in routine dotproduct:");
        printf("PE%i: aa.length= %d \".my_rank, aa.length);
        printf("PE%i: is different from \".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**

```c
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
### 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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a_{11}</td>
<td>0</td>
<td>a_{13}</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>a_{22}</td>
<td>a_{23}</td>
<td>0</td>
</tr>
<tr>
<td>a_{31}</td>
<td>0</td>
<td>a_{33}</td>
<td>a_{34}</td>
</tr>
<tr>
<td>0</td>
<td>a_{42}</td>
<td>0</td>
<td>a_{44}</td>
</tr>
</tbody>
</table>

**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**

\[ \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \]
sparse matrices: row ordered example

<table>
<thead>
<tr>
<th>elements</th>
<th>number_of_row_elements</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 5</td>
<td>0 3</td>
<td>0 11</td>
</tr>
<tr>
<td>1 11</td>
<td>1 2</td>
<td>1 4</td>
</tr>
<tr>
<td>2 7</td>
<td>2 3</td>
<td>2 5</td>
</tr>
<tr>
<td>3 8</td>
<td>3 3</td>
<td>3 5</td>
</tr>
<tr>
<td>4 2</td>
<td>4 2</td>
<td>4 5</td>
</tr>
<tr>
<td>5 4</td>
<td>5 3</td>
<td>5 5</td>
</tr>
<tr>
<td>6 16</td>
<td>6 3</td>
<td>6 5</td>
</tr>
<tr>
<td>7 9</td>
<td>7 2</td>
<td>7 5</td>
</tr>
<tr>
<td>8 12</td>
<td>8 3</td>
<td>8 5</td>
</tr>
</tbody>
</table>

Arrays beginning with 0

Arrays elements organized as index array

Working direction

Rows may have different length

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

```c
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++ ) {  // incrementing a variable
        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

```c
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++ ) {  // incrementing a variable
        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 contiguous access to data
    }
}
```
### Sparse Matrices: Jagged Diagonal Example

<table>
<thead>
<tr>
<th>row_number</th>
<th>pseudo_col</th>
<th>index</th>
<th>end</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>2</td>
<td>23</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>22</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>4</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>6</td>
<td>19</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>7</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>8</td>
<td>17</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>9</td>
<td>16</td>
</tr>
</tbody>
</table>

Arrays beginning with 0. Matrix elements organized as index array.

Working direction decreasing number of pseudo columns.

May be useful.

### Sparse Matrix Vector Multiplication, Jagged Diagonal

```c
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;  
  for( nn = 0 ; nn < mat_number_of_pseudo_columns; nn++ ) {  
    ii=0;  
    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;  
    }  
    num=num+ii;  
  }  
  for( ii = 0 ; ii < mat_number_of_rows; ii++ ) {  
    yy[mat_elements[ii]] = y_temp[ii];  
  }  
  *total_number_of_operations=2*num;} 
```

Short loop

Long loop

Incrementing an array

Reordering loop

The code snippet demonstrates the implementation of a function for sparse matrix vector multiplication using a jagged diagonal approach.
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

---

values to be communicated
edges between domains not shown
long small patches because
lexicographic global enumeration
15 x 11 grid on 3 x 4 domains

inter domain edges suppressed here
contiguous global enumeration in domains
values to be communicated; upper part
omitted

15 x 11 grid with irregularly enumerated 3x4 domains

domain enumeration within intervals
end