Algebraic Multigrid Methods on GPU-Accelerated Hybrid Architectures

Manfred Liebmann
Technische Universität München
Chair of Optimal Control
Center for Mathematical Sciences, M17
manfred.liebmann@tum.de

January 19, 2016

(1) Parallel Toolbox Software

- **Parallel Toolbox Solver Library**
  - The toolbox is implemented as C++ template library
  - The central piece of the library is the communicator class
  - The communicator class handles all data exchange for parallel linear algebra kernels
  - This design enables the reuse of sequential linear algebra kernels
  - The library includes optimized parallel CPU/GPU solver components: PCG, AMG
  - A flexible and modular design for building complex parallel solvers

Communicator Class

The communicator is derived from a domain decomposition based parallelization approach.

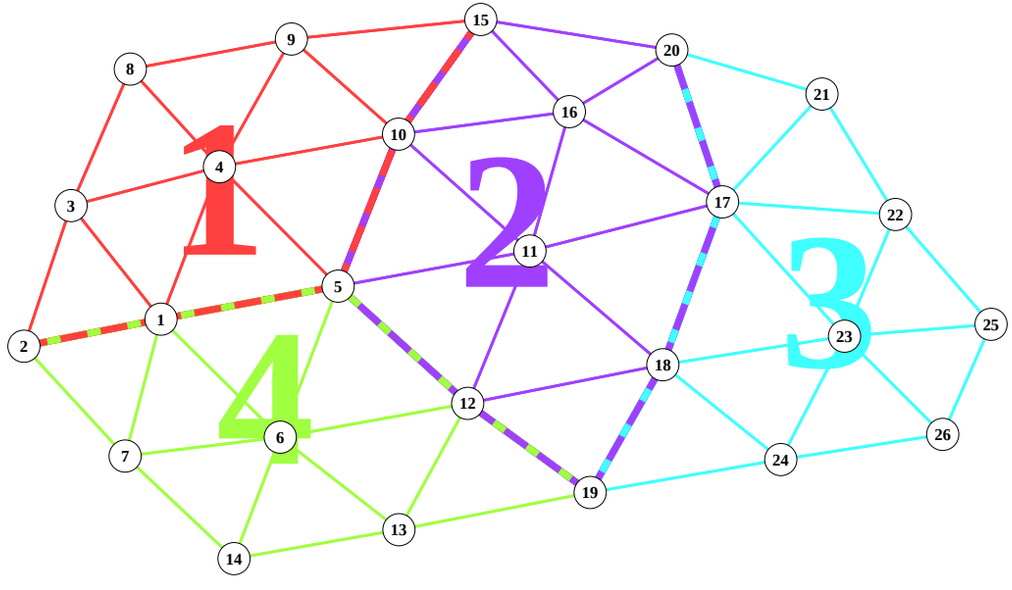

Figure 1: Simple finite element mesh distributed to four processors with global node numbers and color-coded processor domains.

Parallel communication is handled by a communicator object using MPI all-to-all communication patterns. Basic parallel linear algebra routines can be build with the sequential routines and the communicator object.

- **Parallel linear algebra basics**
  - Accumulated vector: $\mathfrak{r}, \mathfrak{s}$ (fraktur font)
  - Distributed vector: r, s (sans-serif font)
  - Accumulated matrix: $\mathfrak{A}, \mathfrak{B}$
  - Distributed matrix: A, B
- **Matrix-vector multiplication and scalar product**
  - Multiplication: $\mathsf{r} \leftarrow \mathsf{A}\mathfrak{s},\ \mathsf{s} \leftarrow \mathfrak{B}\mathsf{r}$
  - Scalar product: $\sigma \leftarrow \mathfrak{S}(\mathfrak{r}, \mathsf{s}) \equiv \mathfrak{S}(\mathsf{r}, \mathfrak{s})$
- **Accumulation and distribution**
  - Accumulation: $\mathfrak{r} \Leftarrow \mathsf{r}$ *Communication!*
  - Distribution: $\mathsf{r} \Leftarrow \mathfrak{r}$

Essential Communication Routines

Accumulation $\mathfrak{r} \Leftarrow \mathsf{r}$ is the most important communication routine in the Parallel Toolbox. This is the only place where MPI all-to-all communication takes place within linear algebra calculations. The accumulation routine provides a single point to optimize communication performance.

Furthermore, distribution of a vector $\mathsf{r} \Leftarrow \mathfrak{r}$ does not require any communication and is a local operation.

Calculating the global value of a scalar product $\sigma \leftarrow \mathfrak{S}(\mathfrak{r}, \mathsf{s})$ requires a simple MPI all-gather operation and accumulation of a single value. Scalar products are expensive because they enforce a synchronization point in the parallel code path.

(2) Sequential Algebraic Multigrid Algorithm

- **Main ingredients of the algebraic multigrid setup**
  - Coarse and fine node selection process: $I = C \cup F$
  - Construction of prolongation P and restriction R operators
  - Triple matrix product: $A_c = RAP$

Coarsening Algorithm

Simplified Ruge-Stüben based coarsening algorithm using the strong connection concept.

$C \leftarrow \emptyset, F \leftarrow \emptyset, T \leftarrow I$
while $T \neq \emptyset$ **do**
  Find next node $i \in T$
  $C \leftarrow C \cup \{i\}$
  $F \leftarrow F \cup \{j \in I | j \notin C \cup F \wedge i \neq j \wedge |A_{ij}| > \epsilon |A_{ii}|\}$
  $T \leftarrow T \backslash (C \cup F)$
end while

Prolongation Operator

$$P = \begin{pmatrix} \mathbf{1}_{CC} \\ P_{FC} \end{pmatrix} \tag{1}$$

Define the number of strongly coupled coarse grid nodes with respect to the fine grid node $i \in F$:

$$n_i := \#\{j \in C || A_{ij}| > \epsilon |A_{ii}|\} \tag{2}$$

The matrix P_{FC} is then defined as:

$$(P_{FC})_{ij} := \begin{cases} 1/n_i, & |A_{ij}| > \epsilon |A_{ii}| \\ 0, & \text{else} \end{cases} \tag{3}$$

Matrix Graph Representations

$$A = \begin{pmatrix} 2 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 \end{pmatrix} \tag{4}$$

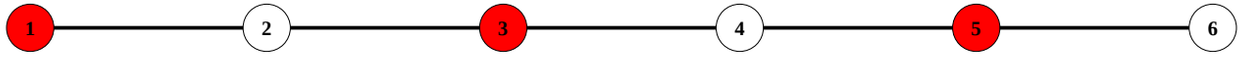

Figure 2: The undirected matrix graph of the 1D Laplace operator.

Figure 3: The directed graph of the prolongation operator.

Sequential Classic Multigrid Algorithm

Require: $f_l, u_l, r_l, s_l, A_l, R_l, P_l, S_l, T_l, \quad 0 \leq l < L$

$f_0 \leftarrow f$

if $l < L$ **then**

$u_l \leftarrow 0$ {Initial guess}

$u_l \leftarrow S_l(u_l, f_l)$ {Presmoothing}

$r_l \leftarrow f_l - A_l u_l$ {Calculation of residual}

$f_{l+1} \leftarrow R_l r_l$ {Restriction of residual}

multigrid(l+1) {Multigrid recursion}

$s_l \leftarrow P_l u_{l+1}$ {Prolongation of correction}

$u_l \leftarrow u_l + s_l$ {Addition of correction}

$u_l \leftarrow T_l(u_l, f_l)$ {Postsmoothing}

else

$u_L \leftarrow A_L^{-1} f_L$ {Coarse grid solver}

end if

$u \leftarrow u_0$

ω-Jacobi and Forward/Backward Gauss-Seidel Smoother

ω-Jacobi Smoother

Require: $A_l, D \leftarrow \mathrm{diag}(A_l), \omega \in (0, 1]$

return $u_l \leftarrow u_l + \omega D^{-1}(f_l - A_l u_l)$

Forward/Backward Gauss-Seidel Smoother

Require: $A_l, D \leftarrow \mathrm{diag}(A_l)$

return $u_l \overset{\text{forward}}{\longleftarrow} u_l + D^{-1}(f_l - A_l u_l)$

Require: $A_l, D \leftarrow \mathrm{diag}(A_l)$

return $u_l \overset{\text{backward}}{\longleftarrow} u_l + D^{-1}(f_l - A_l u_l)$

Preconditioned Conjugate Gradient Algorithm

Require: $m \in \mathbb{N}, \alpha, \beta, \sigma, \sigma_f, \sigma_l, \tau \in \mathbb{R}, r, s, v, w$
$r \leftarrow f - Au$ {Calculation of residual}
$s \leftarrow Pr$ {Apply preconditioner}
$\sigma \leftarrow S(s, r)$ {Scalar product}
$\sigma_f \leftarrow \sigma_l \leftarrow \sigma, \quad m \leftarrow 0$
while $m < M \wedge \sigma/\sigma_f > \epsilon^2$ **do**
$m \leftarrow m + 1, \quad v \leftarrow As$
$\tau \leftarrow S(s, v)$ {Scalar product}
$\alpha \leftarrow \sigma/\tau$
$u \leftarrow u + \alpha s$ {Update solution}
$r \leftarrow r - \alpha v$ {Update residual}
$w \leftarrow Pr$ {Apply preconditioner}
$\sigma \leftarrow S(w, r)$ {Scalar product}
$\beta \leftarrow \sigma/\sigma_l, \quad \sigma_l \leftarrow \sigma$
$s \leftarrow \beta s + w$ {Update search direction}
end while

(3) Parallel Algebraic Multigrid Algorithm

- **Challenges of the parallel algebraic multigrid setup**
  - Parallel coarse and fine node selection must be globally consistent
  - Skeleton based coarsening equivalent to sequential algorithm with reordering
  - Local prolongation and restriction operators without communication requirements

Parallel Coarsening Algorithm

Construct the boundary node set B and the submatrix A_{BB}

(a)

$C_B^p \leftarrow \emptyset, F_B^p \leftarrow \emptyset, T_B^p \leftarrow B$

while $T_B^p \neq \emptyset$ **do**

Find next node $i \in T_B^p$

$C_B^p \leftarrow C_B^p \cup \{i\}$

$F_B^p \leftarrow F_B^p \cup \{j \in B | j \notin C_B^p \cup F_B^p \wedge i \neq j \wedge |A_{BB;ij}| > \epsilon |A_{BB;ii}|\}$

$T_B^p \leftarrow T_B^p \backslash (C_B^p \cup F_B^p)$

end while

(b)

$C^p \leftarrow I^p \cap C_B^p, F^p \leftarrow I^p \cap F_B^p, T^p \leftarrow I^p \cap C_B^p$

while $T^p \neq \emptyset$ **do**

Find next node $i \in T^p$

$F^p \leftarrow F^p \cup \{j \in I^p | j \notin C^p \cup F^p \wedge i \neq j \wedge |A_{ij}^p| > \epsilon |A_{ii}^p|\}$

$T^p \leftarrow T^p \backslash \{i\}$

end while

(c)

$T^p \leftarrow I^p \backslash (C^p \cup F^p)$

while $T^p \neq \emptyset$ **do**

Find next node $i \in T^p$
$C^p \leftarrow C^p \cup \{i\}$
$F^p \leftarrow F^p \cup \{j \in I^p | j \notin C^p \cup F^p \wedge i \neq j \wedge |A^p_{ij}| > \epsilon |A^p_{ii}|\}$
$T^p \leftarrow T^p \backslash (C^p \cup F^p)$
end while

Parallel Coarsening: Step (a)

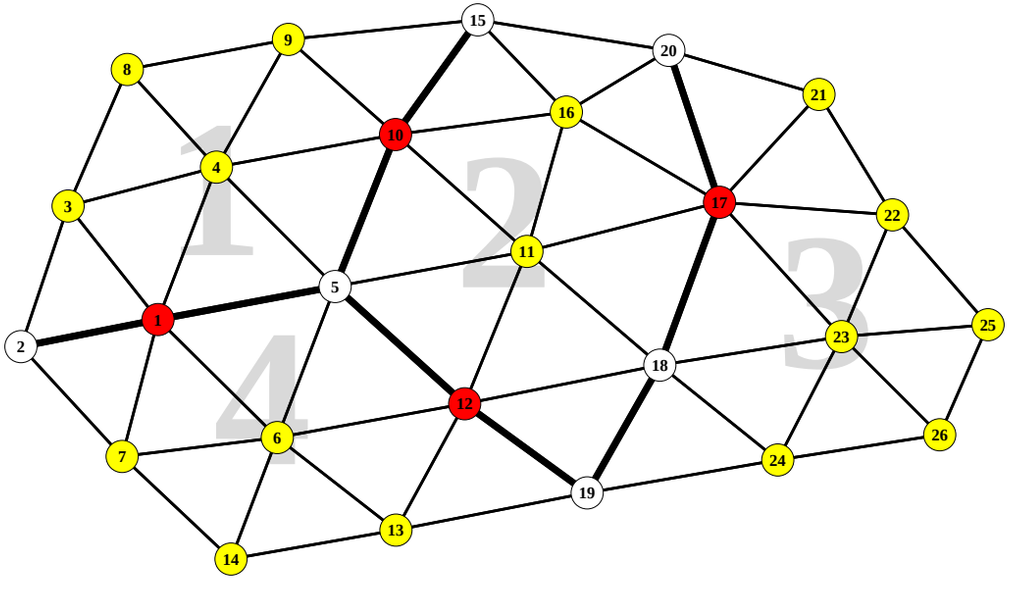

Figure 4: Parallel Coarsening Step (a): The boundary nodes are assigned either red coarse grid nodes or white fine grid nodes. The unassigned nodes in the inner of the processor domains are depicted in yellow color.

Parallel Coarsening: Step (b)

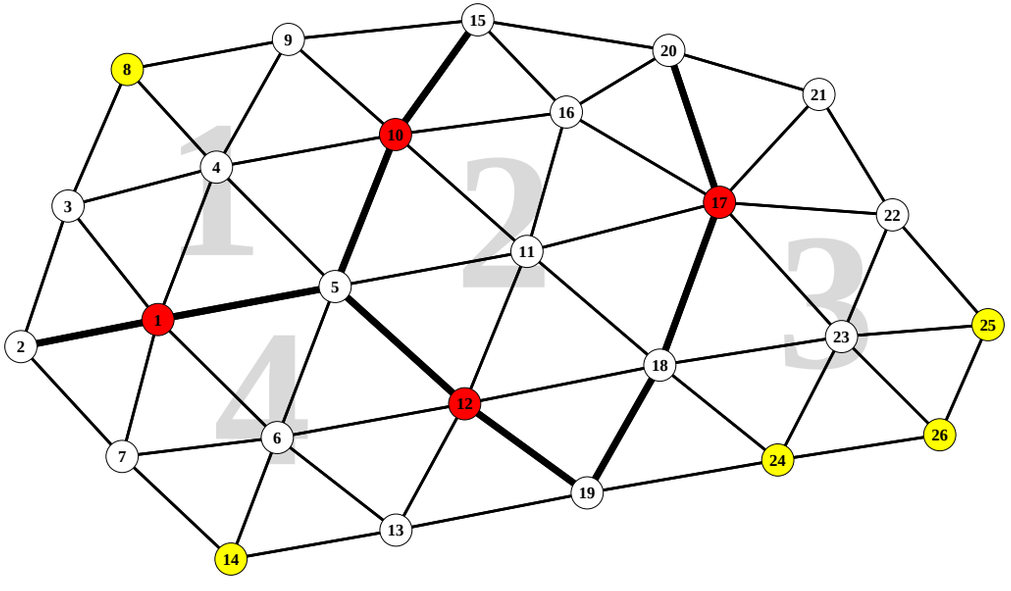

Figure 5: Parallel Coarsening Step (b): Fine grid nodes in the inner of the processor domains depending on the coarse boundary nodes are assigned.

Parallel Coarsening: Step (c)

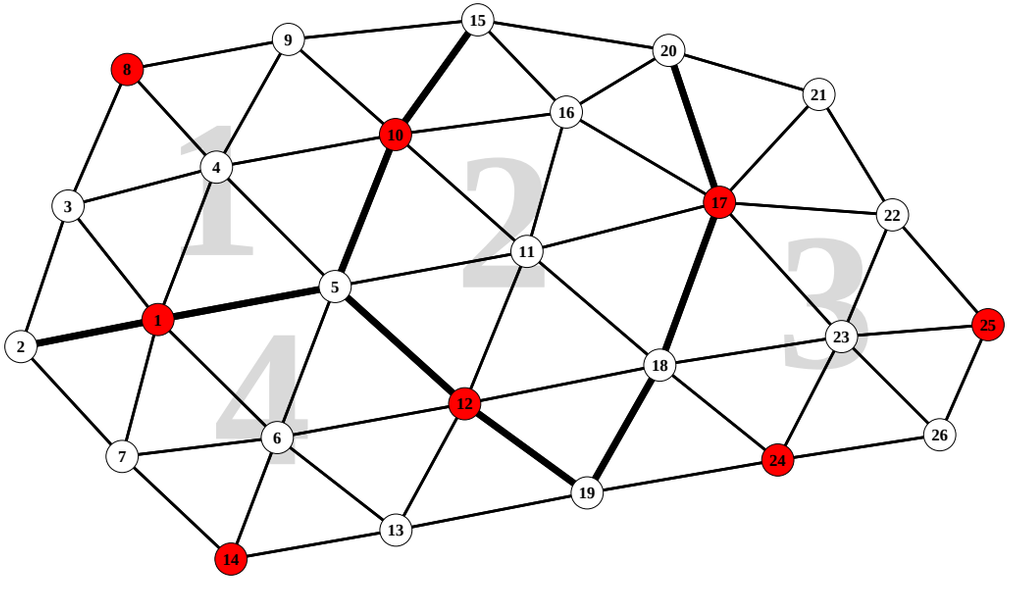

Figure 6: Parallel Coarsening Step (c): The coarsening is completed on the remaining nodes in the inner of the processor domains.

Parallel Prolongation Operator

The local prolongation operator on processor p requires *foreign* coarse grid nodes C^{*p} to interpolate the boundary fine grid nodes F^{*p}.

$F^{*p} \leftarrow F^p \cap B^p$

Construct the submatrix $A_{F^{*p}C}$

$C^{*p} \leftarrow \{j \in C | \exists i \in F^{*p} : |A_{F^{*p}C;ij}| > \epsilon |A_{F^{*p}C;ii}|\} \backslash C^p$

$J^p \leftarrow C^p \cup C^{*p}$

Construct $P^p := P_{I^p J^p}$ using $A_{F^{*p}C}$ and A^p

Interpolation on the Finite Element Mesh

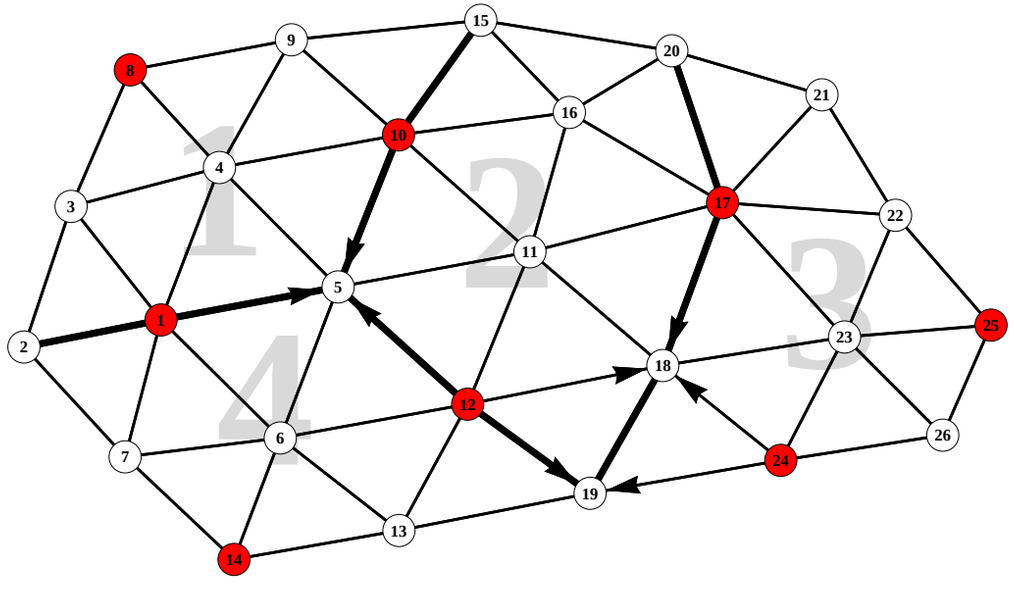

Figure 7: White fine grid nodes are interpolated by red coarse grid nodes. The black arrows show the boundary crossing interpolation for the three fine grid nodes 5, 18, and 19.

Global Prolongation Operator

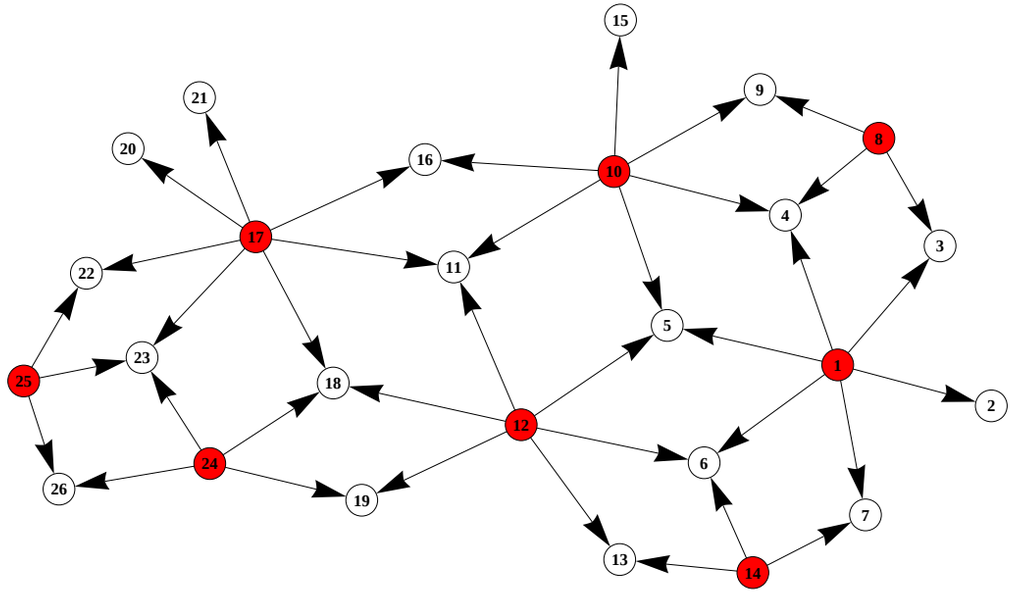

Figure 8: The global prolongation operator on the whole simulation domain represented as a directed graph.

Interpolation on the Color-Coded Finite Element Mesh

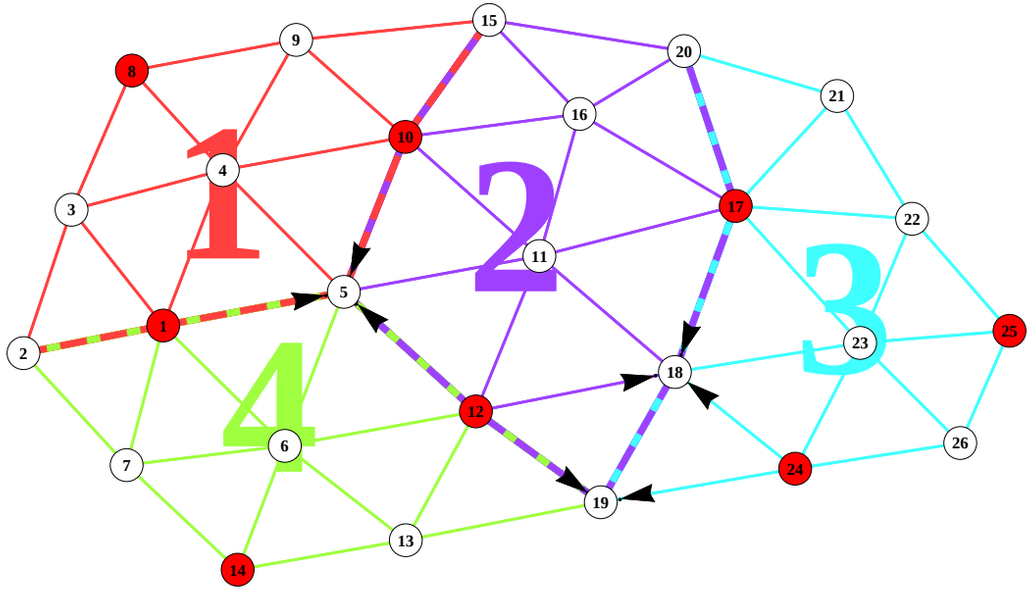

Figure 9: The processor domains embedded in the finite element mesh are shown in color. The cross-boundary interpolation is depicted for the fine grid nodes 5, 18, and 19.

Color-Coded Global Prolongation Operator

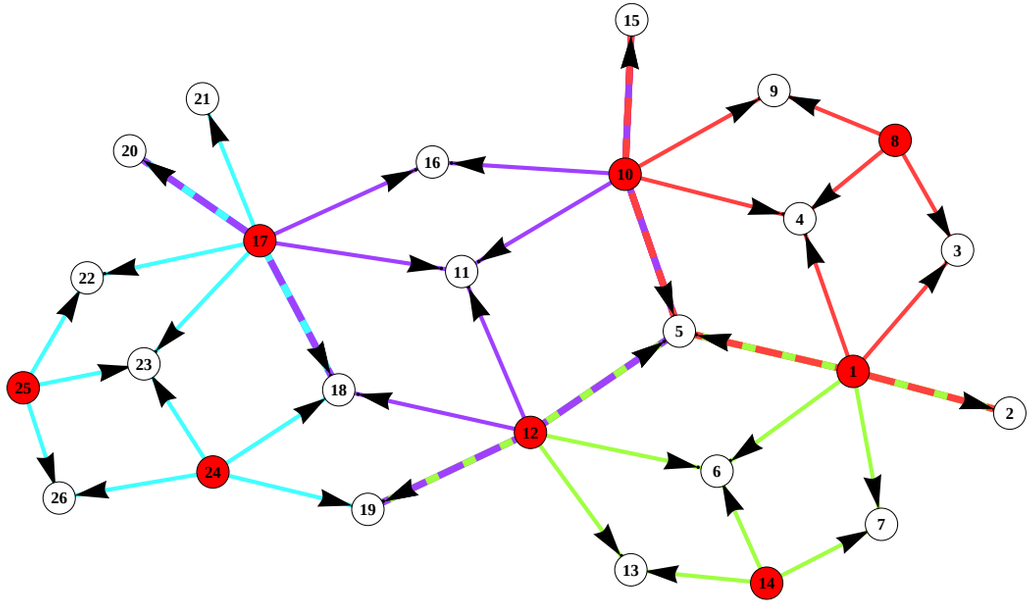

Figure 10: The global prolongation operator on the whole simulation domain represented as a color-coded directed graph.

Local Prolongation Operator on Processor 1

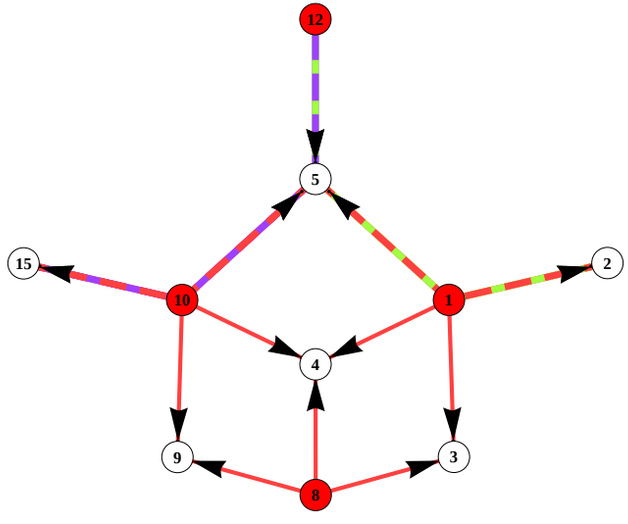

Figure 11: The local prolongation operator on processor one represented as colorcoded directed graph. Note the foreign coarse grid node 12 and the crossboundary interpolation of node 5.

Local Prolongation Operator on Processor 2

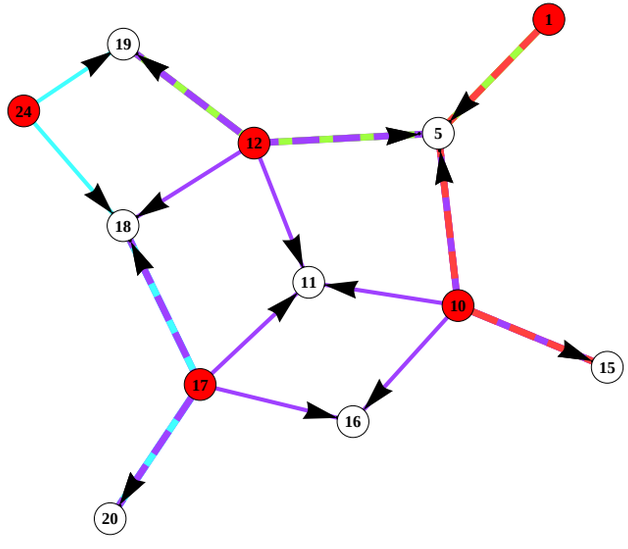

Figure 12: The local prolongation operator on processor two represented as colorcoded directed graph. Note the foreign coarse grid nodes 1 and 24 and the cross-boundary interpolation of the nodes 5, 18, and 19.

Local Prolongation Operator on Processor 3

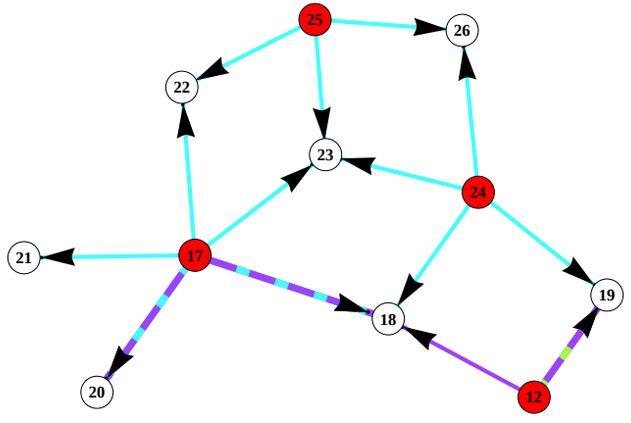

Figure 13: The local prolongation operator on processor three represented as colorcoded directed graph. Note the foreign coarse grid node 12 and the crossboundary interpolation of the nodes 18 and 19.

Local Prolongation Operator on Processor 4

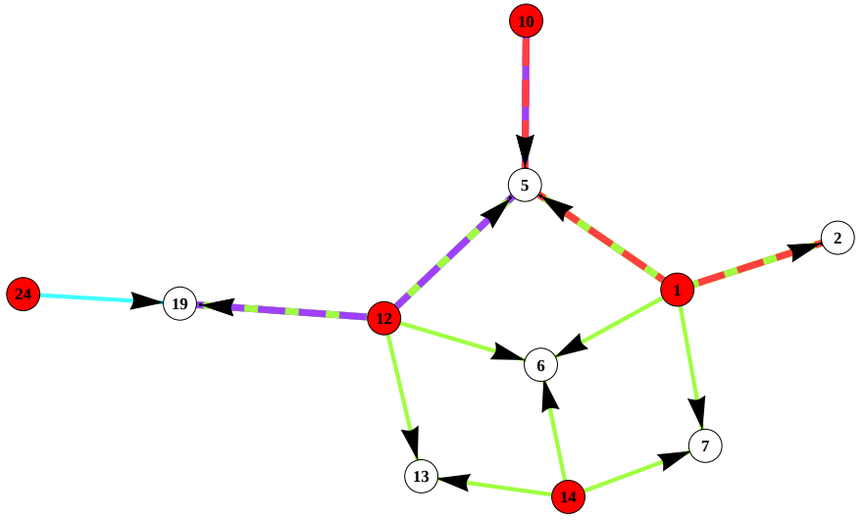

Figure 14: The local prolongation operator on processor four represented as colorcoded directed graph. Note the foreign coarse grid nodes 10 and 24 and the cross-boundary interpolation of the nodes 5 and 19.

Parallel Classic Multigrid Algorithm

Require: $\mathsf{f}_l, \mathfrak{u}_l, \mathsf{r}_l, \mathfrak{s}_l, R_l, P_l, S_l, T_l, \quad 0 \leq l < L$

$\mathsf{f}_0 \leftarrow \mathsf{f}$

if $l < L$ **then**

$\mathfrak{u}_l \leftarrow 0$ {Initial guess}

$\mathfrak{u}_l \leftarrow S_l(\mathfrak{u}_l, \mathsf{f}_l)$ {Presmoothing}

$\mathsf{r}_l \leftarrow \mathsf{f}_l - \mathsf{A}_l\mathfrak{u}_l$ {Calculation of residual}

$\mathsf{f}_{l+1} \leftarrow R_l\mathsf{r}_l$ {Restriction of residual}

multigrid(l+1) {Multigrid recursion}

$\mathfrak{s}_l \leftarrow P_l\mathfrak{u}_{l+1}$ {Prolongation of correction}

$\mathfrak{u}_l \leftarrow \mathfrak{u}_l + \mathfrak{s}_l$ {Addition of correction}

$\mathfrak{u}_l \leftarrow T_l(\mathfrak{u}_l, \mathsf{f}_l)$ {Postsmoothing}

else

$\mathfrak{u}_L \Leftarrow (\mathfrak{A}_L)^{-1}\mathsf{f}_L$ {Coarse grid solver}

end if

$\mathfrak{u} \leftarrow \mathfrak{u}_0$

Parallel ω-Jacobi and Forward/Backward Gauss-Seidel Smoother

Parallel ω-Jacobi Smoother

Require: $A_l^p, \mathfrak{D} \Leftarrow \mathrm{diag}(A_l^p), \omega \in (0,1]$
$\mathfrak{v}_l^p \Leftarrow (\mathfrak{f}_l^p - A_l^p \mathfrak{u}_l^p)$
return $\mathfrak{u}_l^p \leftarrow \mathfrak{u}_l^p + \omega \mathfrak{D}^{-1} \mathfrak{v}_l^p$

Parallel Forward/Backward Gauss-Seidel Smoother

Require: $A_l^p, \mathfrak{D} \Leftarrow \mathrm{diag}(A_l^p), \omega \in (0,1]$
$\mathfrak{v}_l^p \Leftarrow (\mathfrak{f}_l^p - A_l^p \mathfrak{u}_l^p)$ {On boundary nodes}
$\mathfrak{u}_l^p \leftarrow \mathfrak{u}_l^p + \omega \mathfrak{D}^{-1} \mathfrak{v}_l^p$ {On boundary nodes}
$\mathfrak{u}_l^p \overset{\text{forward}}{\longleftarrow} \mathfrak{u}_l^p + \mathfrak{D}^{-1}(\mathfrak{f}_l^p - A_l^p \mathfrak{u}_l^p)$ {On inner nodes}
return $\mathfrak{u}_l^p$

Require: $A_l^p, \mathfrak{D} \Leftarrow \mathrm{diag}(A_l^p), \omega \in (0,1]$
$\mathfrak{u}_l^p \overset{\text{backward}}{\longleftarrow} \mathfrak{u}_l^p + \mathfrak{D}^{-1}(\mathfrak{f}_l^p - A_l^p \mathfrak{u}_l^p)$ {On inner nodes}
$\mathfrak{v}_l^p \Leftarrow (\mathfrak{f}_l^p - A_l^p \mathfrak{u}_l^p)$ {On boundary nodes}
$\mathfrak{u}_l^p \leftarrow \mathfrak{u}_l^p + \omega \mathfrak{D}^{-1} \mathfrak{v}_l^p$ {On boundary nodes}
return $\mathfrak{u}_l^p$

Parallel Conjugate Gradient Algorithm

Require: $m \in \mathbb{N}, \alpha, \beta, \sigma, \sigma_f, \sigma_l, \tau \in \mathbb{R}$, $\mathsf{v}, \mathsf{r}, \mathfrak{s}, \mathfrak{w}$

$\mathsf{r} \leftarrow \mathsf{f} - \mathsf{A}\mathfrak{u}$ {Calculation of residual}

$\mathfrak{s} \leftarrow \mathfrak{P}\mathsf{r}$ {Apply preconditioner}

$\sigma \leftarrow \mathfrak{S}(\mathfrak{s}, \mathsf{r})$ {Scalar product}

$\sigma_f \leftarrow \sigma_l \leftarrow \sigma, \quad m \leftarrow 0$

while $m < M \wedge \sigma/\sigma_f > \epsilon^2$ **do**

$m \leftarrow m + 1, \quad \mathsf{v} \leftarrow \mathsf{A}\mathfrak{s}$

$\tau \leftarrow \mathfrak{S}(\mathfrak{s}, \mathsf{v})$ {Scalar product}

$\alpha \leftarrow \sigma/\tau$

$\mathfrak{u} \leftarrow \mathfrak{u} + \alpha\mathfrak{s}$ {Update solution}

$\mathsf{r} \leftarrow \mathsf{r} - \alpha\mathsf{v}$ {Update residual}

$\mathfrak{w} \leftarrow \mathfrak{P}\mathsf{r}$ {Apply preconditioner}

$\sigma \leftarrow \mathfrak{S}(\mathfrak{w}, \mathsf{r})$ {Scalar product}

$\beta \leftarrow \sigma/\sigma_l, \quad \sigma_l \leftarrow \sigma$

$\mathfrak{s} \leftarrow \beta\mathfrak{s} + \mathfrak{w}$ {Update search direction}

end while

(4) Parallelization on GPU-Accelerated Hybrid Architectures

- **Parallelization Concepts on CPU and GPU clusters for PCG-AMG**
  - Coarse grained MPI parallelization for CPU and GPU nodes
  - Additional fine grained parallelization on GPUs
  - Many-core implementation of sparse matrix-vector multiplication
  - Interleaved CRS data format for *coalesced memory access* on GPU
  - Nvidia CUDA Technology for C/C++ GPU programming

Sparse Matrix-Vector Multiplication Kernel

Let $A \in \mathbb{R}^{N \times N}$ be a matrix in compressed row storage format and $u, b \in \mathbb{R}^N$.

- **CRS Sparse Matrix-Vector Multiplication Kernel**
  - Schedule a thread for every sparse scalar product!
  - Thread i calculates $u_i = \sum_{j=1}^{N} A_{ij} b_j$

Looks nice! Performs very badly!

Problems and Solutions

- **Non-coalesced memory access!**
  - Rearrange CRS data structure for coalesced access
  - Interleave the sparse matrix rows for at least 16 consecutive rows
  - Holes in the data structure: Not critical! Typical 5-10% increase in storage
- **Random access to b vector!**
  - Use the texture unit of the GPU for random access to b vector
  - Texturing is optimized for spacial locality: Small read-only cache

GPU-Accelerated Sparse Matrix-Vector Multiplication

An efficient sparse matrix-vector multiplication is key to the PCG-AMG solver performance.

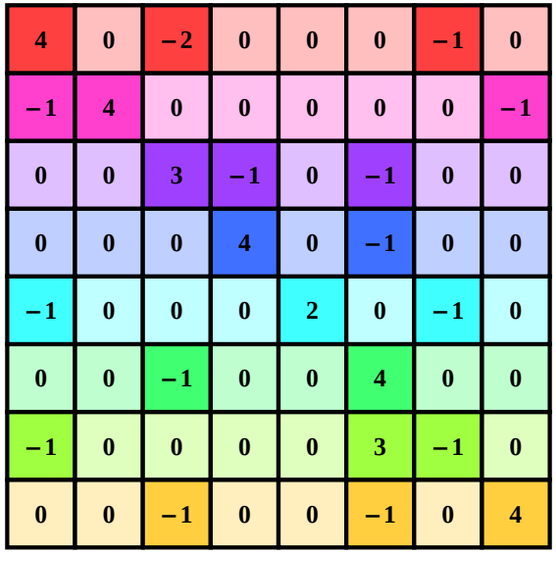

Figure 15: A sample matrix with the rows colored in different hues.

Compressed Row Storage Data Format (CRS)

A flexible data format for sparse matrices.

3	3	3	2	3	2	3	3
0	3	6	9	11	14	16	19

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

Figure 16: CRS data structure for the sample matrix with the count and displacement vector on top and the column indices and matrix entries below.

Interleaved Compressed Row Storage Data Format (ICRS)

Coalesced memory access patterns on GPUs are required to achieve high performance.

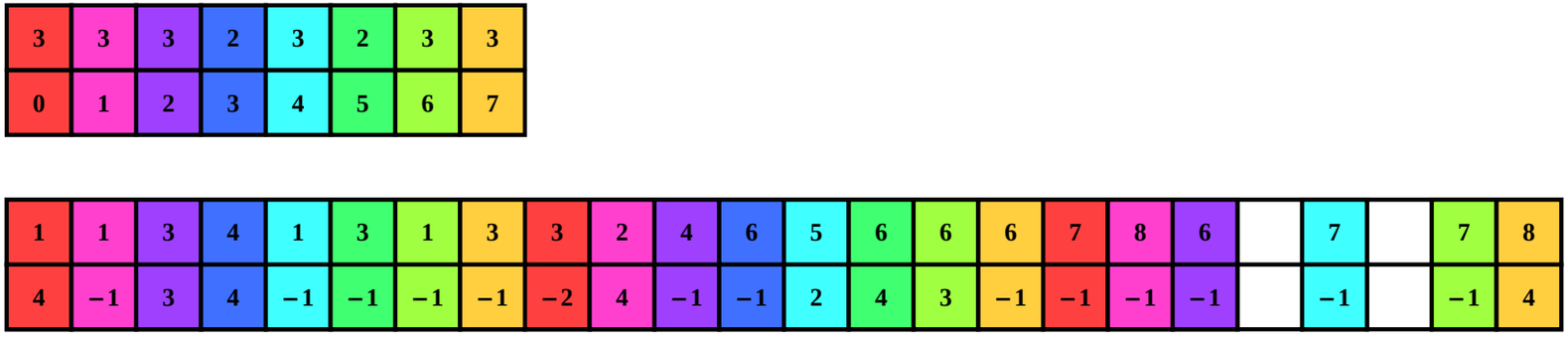

Figure 17: ICRS data structure for the sample matrix with the count and displacement vector on top and the interleaved column indices and matrix entries below. The eight interleaved matrix rows create holes in the data structure represented by the white boxes.

CUDA Code Sample for Sparse Matrix-Vector Multiplication

```
#define L 256
struct linear_operator_params {
    const int *cnt;       //ICRS count vector
    const int *dsp;       //ICRS displacement vector
    const int *col;       //ICRS column indices
    const double *ele;    //ICRS matrix entries
    const double *u;      //Input vector
    double *v;            //Output vector
    int n;                //Matrix dimension
};
texture<int2> tex_u;

void _device_linear_operator(int *cnt, int *dsp, int *col, double *ele,
    int m, int n, double *u, double *v)
{
    cudaBindTexture(0, tex_u, (int2*)u, sizeof(double) * m); //Bind the texture

    struct linear_operator_params parms;
    parms.cnt = cnt; parms.dsp = dsp; parms.col = col; parms.ele = ele;
    parms.u = u; parms.v = v; parms.n = n;
    __device_linear_operator<<< (n + N - 1)/N, N >>>(parms); //GPU kernel launch

    cudaUnbindTexture(tex_u);                                //Unbind the texture
}
```

```
__global__ void __device_linear_operator(struct linear_operator_params parms)
{
    unsigned int j = N * blockIdx.x + threadIdx.x;
    if(j < parms.n)
    {
        unsigned int blkStart = parms.dsp[j];
        unsigned int blkStop = blkStart + L * parms.cnt[j];
        double s = 0.0;
        for(unsigned int i = blkStart; i < blkStop; i += L)
        {
            unsigned int q = parms.col[i];          //Load column index
            double a = parms.ele[i];                //Load matrix entry
            int2 c = tex1Dfetch(tex_u, q);          //Load vector entry using texture mapping
            double b = __hiloint2double(c.y, c.x);  //Convert texture entries to double number
            s += a * b;                             //Calculate the sparse scalar product
        }
        parms.v[j] = s;                             //Store the sparse scalar product
    }
}
```