*P. Koumoutsakos*
*ETH Zentrum, CLT E 13*
*CH-8092 Zürich*

Fall semester 2018

# Set 9 - Sparse Linear Algebra with MPI

Issued: November 23, 2018
Hand in (optional): December 03, 2018 23:59

## Question 1: CSR format

Sparse matrices have a limited number of nonzero elements. Specialized formats are used for efficient storage and operations on them. The CSR format (compressed sparse row) represents a matrix $A_{ij}$ by three one-dimensional arrays:

- $A[k]$, nonzero coefficients in the row-major order, $0 \leq k < nnz$;
- $K[i]$, extents of rows, row $i$ consists of $A[K[i] : K[i+1]]$;
- $J[k]$, column indices.

Write down a representation of the following matrix in the CSR format

$$A = \begin{bmatrix} 2 & -1 & 0 & -1 \\ -1 & 2 & -1 & 0 \\ 0 & -2 & 4 & -2 \\ -1 & 0 & -1 & 2 \end{bmatrix}.$$

From that representation, compute the product $Au$ with column $u = [0, 1, 2, 3]^T$.

(**1 points**) The CSR representation:

```
A:  2  -1  -1  -1   2  -1  -2  4  -2  -1  -1   2
K:  0   3   6   9  12
J:  0   1   3   0   1   2   1  2   3   0   2   3
```

The matrix-vector product $b = Au$:

```
b: -4   0   0   4
```

## Question 2: Matrix-vector product with MPI

Sparse matrices and corresponding vectors can be distributed among multiple processors. Consider the matrix-vector product

$$b = Au$$

with a square matrix $A \in \mathbf{R}^{n \times n}$ and vectors (columns) $u, b \in \mathbf{R}^n$. Each of $p$ processors stores $n/p$ rows of the matrix, the corresponding chunks of vector $b$ and the same chunks of vector $u$. In the worst case, the communication pattern is all-to-all. However, for a banded matrix (e.g. tridiagonal) the communication is required only between neighbouring chunks. For instance, such matrices result from discretization of PDEs.

Here you will implement the sparse matrix-vector product and apply it for the solution of the two-dimensional diffusion equation. The skeleton code provides structures for storing the matrix and converting between local and global indices. Distributed CSR format uses global indices for columns and local indices for extents of rows. The algorithm should involve the following stages:

- traverse rows of the matrix, multiply the elements stored locally and collect global indices of columns that require communication; use `GlbToLoc()` to convert the global indices;
- use `GlbToRank()` to find the ranks of processors storing the required columns;
- use `MPI_Allreduce()` to compute the number of messages that every processor needs to receive;
- send and receive the indices of columns from other processors;
- send and receive the corresponding elements of vector $u$;
- add the received data to the product.

a) Implement the matrix-vector product in function `Mul()`. For testing use the provided discretization of the Laplace operator applied for solving the two-dimensional diffusion equation

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

as well as other operators found in `op.h`. For plotting the results use `make plot` which calls the Python script `plot`.

File `mul.h` in the solution code contains a reference implementation. The implementation depends only on functions `GlbToRank()` and `GlbToLoc()` that define the partitioning of matrices and vectors among processors. Grading of the subtasks:

1. product of local elements (**2 points**)
2. communication for remote elements of vector $u$ required for the product (**10 points**)
   - $-3$ points if the communication involves the whole vector $u$ from remote ranks, not only the required elements
   - $-1$ points if the implementation relies on specific features of the mapping (e.g. same number of equations on all ranks) or the matrix (e.g. exactly 5 elements in each row)
3. product of remote elements (**2 points**)

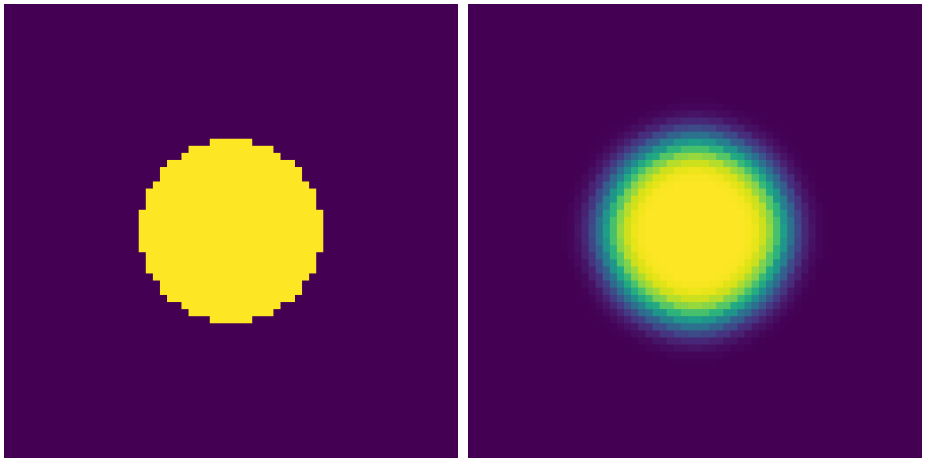

Figure 1: Solution of the diffusion equation on grid $64^2$: initial (left) and after 10 steps (right).

b) Implement function `WriteMpi()` in `io.h` with either MPI I/O or gathering the data on one processor.

File `io.h` in the solution code contains two reference implementations

- `WriteMpiGather()` which gathers the data on the master rank;
- `WriteMpiIo()` which uses the MPI IO routines.

Grading: **3 points** for correct implementation of any of the methods.

c) Report the weak scaling on Euler.

For the weak scaling the number of components (i.e. number of grid cells) per processor should be kept constant. Fig. 2 shows the weak scaling efficiency up to 36 ranks on Euler.

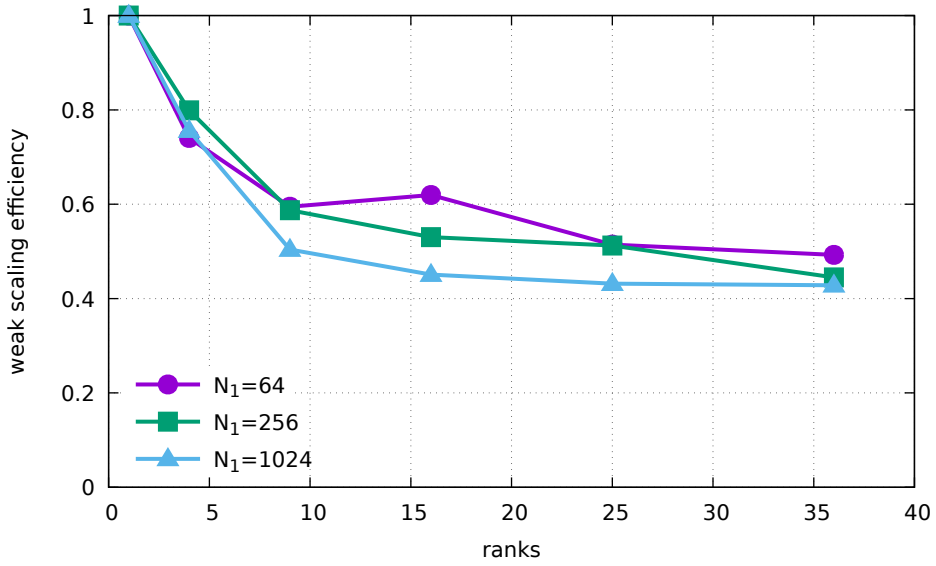


Figure 2: Weak scaling on Euler. $N_1$ is the grid size for one processor.

Grading: **2 points** for the weak scaling data with at least three different values of the number of processors.

d) (Optional) Add a new linear operator of your choice to `op.h` and apply for discretization of a different equation (e.g. advection equation). You can combine multiple operators in one time step to discretize non-linear equations.

Slides `solution09_slides.pdf` and movies in `mov` show other applications. Instructions for reproducing them are given in `README.md`.

Grading: **0 points**.