Verification Challenges in Sparse Matrix Vector Multiplication in High Performance Computing

Junchao Zhang
Argonne National Laboratory, Illinois, USA
jczhang@anl.gov

Sparse matrix vector multiplication is a crucial kernel in scientific codes using iterative solvers. We present a sequential and a basic MPI parallel implementations of the kernel on the CPU, to provide a challenge problem to the scientific software verification community. We describe the implementations in the context of the PETSc library.

1 Introduction

Solving sparse linear systems of $Ax = b$ lies in the heart of scientific computing. Iterative Krylov subspace methods [2] are a key algorithm for that, especially for large scale computations. Sparse matrix vector multiplication (SpMV) is a crucial kernel in these methods. The popular math library, PETSc (the Portable, Extensible Toolkit for Scientific Computation)[1], contains a suite of Krylov methods and SpMV implementations. Since the matrix is usually very sparse, it is wise to only store nonzeros to save memory. The compressed sparse row (CSR), known as `MATAIJ` in PETSc, is a commonly used sparse matrix storage format. It represents an `m` $\times$ `n` matrix with total `nnz` nozeros in three arrays: `a[nnz]`, `i[m]` and `j[nnz]`, where `a[]` and `j[]` store the nonzeros and their column indices row-wisely, respectively, and `i[]` stores indices pointing to the start of each row in `a[]` and `j[]`. Apparently, `i[0] = 0` and row `k` has `i[k+1] - i[k]` nonzeros. `i[m]` points to the next place after the last nonzero. Figure 2 shows a CSR example for a 4 $\times$ 12 sparse matrix.

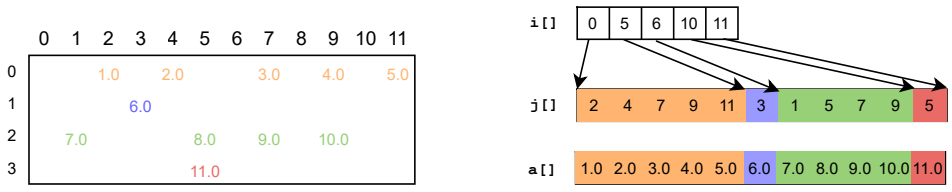

Figure 1: *Left*: A 4 $\times$ 12 sparse matrix. *Right*: its compressed sparse row (CSR) representation.

Matrix vector multplication has a very simple definition. Denote $A = \{a_{ij}\}$, $x = \{x_j\}$, $y = \{y_i\}$, then $y = Ax$ is defined by $y_i = \sum_{j=0}^{n-1} a_{ij}x_j$, with $0 \le i < m$, $0 \le j < n$. However, it is the storage format, data distribution, parallelization and optimizations that make implementations complicated. Among PETSc's many SpMV implementations, there is an MPI parallel one with sophiscated optimizations. We won't elaborate that one here. Instead, we present a sequential and an MPI basic implementations, which are simple but still share good ingredients with the MPI optimized one. We first describle the structure of the code, then the two implementations and the verification challenges within, and conclude at the end.

Submitted to:
VSS 2025

2 Code structure, input and output

We provide standalone implementations written in the C language in a public repository[1]. One only needs to specify a C or MPI compiler to compile them. The input data is hardwired in source code with global varibles. We provide matrix A of size `M` $\times$ `N` and `NNZ` nonzeros in three arrays `Gi[M+1]`, `Gj[NNZ]`, `Ga[NNZ]`, vector x in `Xa[N]`, and the correct answer $z = Ax$ in `Za[M]`. There is also an array `Ya[M]` to provide space for vector y, which will store the computed result of Ax. The code will compute the square of the 2-norm of $||y-z||$, which should be zero, and return a non-zero code if y is wrong. The input data was generated by the accompanied Python script csr.py. One can modify parameters in the script and re-run it to generate a different set of input.

3 Sequential implementation

In this implementation (seq.c), marix A, vectors x, y and z reside in one process. We directly build A, x, y and z from the global arrays mentioned above. Sizes of x, y and z must conform to A's shape.

```
typedef struct {
  int      m, n; // size
  int     *i, *j;
  double *a;     // values
} Mat;

typedef struct {
  int      n; // size
  double *a; // values
} Vec;
```

```
int i, j;
for (i = 0; i < A.m; i++) {
  y.a[i] = 0.0;
  for (j = A.i[i]; j < A.i[i + 1]; j++)
    y.a[i] += A.a[j] * x.a[A.j[j]];
}
```

Figure 2: *Left*: sequential matrix and vector types. *Right*: sequential SpMV kernel.

Figure 2 shows the sequential matrix and vector types and the SpMV kernel. One needs to verify the code computes $y = Ax$. In other words, assume those a_{ij} not in A's CSR representation are zero, then $y_i = \sum_{j=0}^{n-1} a_{ij}x_j$, for $0 \le i < m$, $0 \le j < n$, without counting floating point round-off errors.

4 MPI basic implementation

This implementation (mpibasic.c) uses MPI parallelsim. The global matrix is block-distributed by row. Each MPI process has a piece (submatrix) of the matrix, where the row size of the submatrix can be set by users or automatically computed by PETSc. We call it the matrix row layout. Though matrix columns are not distrubuted, they also have a layout that can be set by users. These row and column layouts are properties of the matrix, by which we distribute vectors y and x respectively. In PETSc, one can use `MatCreateVecs(A, &x, &y)` to create vectors with conforming size and layout suitable for doing $y = Ax$. We use `M`, `N` to represent the global size of the matrix, and `m`, `n` for the local size of the diagonal block of the matrix on this process. `m`, `n` are also the local size of y and x respectively. In this implementation, we first compute the matrix layouts. Suppose `size` is the size of `MPI_COMM_WORLD` and `rank` is the rank of the current MPI process, `m`, `n` are computed as `m = M/size + (M%size > rank ? 1 : 0)`, `n = N/size + (N%size > rank ? 1 : 0)`. This is the formula PETSc would use if users don't set

[1] `https://github.com/jczhang07/cs2`

m and n. With the layouts set, each process knows the indices of its first row rstart and first column cstart. Then it pulls out its data from the global arrays. A.j[] and A.a[] can share the space with Gi[] and Ga[], but we have to allocate A.i[] and populate it with shifted indices from Gi[]. Similarily, we can build x, y and z by pointing them to Xa[cstart], Ya[rstart] and Za[rstart] repectively.

```
typedef struct {
  int      m, n, M, N;
  int      rstart, cstart;
  int     *i, *j;
  double  *a;
} Mat;

typedef struct {
  int      n, N;
  double  *a;
} Vec;
```

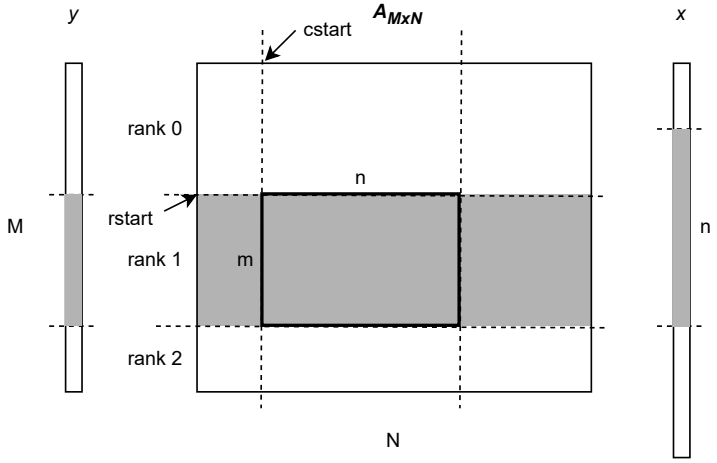

Figure 3: *Left*: MPI parallel matrix and vector types. *Right*: distributed matrix A and vectors x and y on three MPI ranks. The shadowed parts reside on rank 1.

To multiply the local submatrix A with vector x, we need remote entries of x. In theory, we only need entries of x which corresponds to nonzero columns in the local submatrix. But for simplicity, we gather the whole distributed x to a local vector X of size N. Depending on whether N is evenly distributed or not, we can use MPI_Allgather or MPI_Allgatherv. With our block distribution formula, this can be tested by N%size. Otherwise, one has to check whether all n's on processes are equal. With X, we perform a sequential SpMV as in Section 3 and get the local part of y. After that we compute the partial norm of $||y-z||$, and then the final norm with help of MPI_Allreduce.

For this MPI implementation, one needs to verify 1) the layouts are correct, i.e., $\sum m = M$, $\sum n = N$; 2) the y on each process is the result of a sub-SpMV for Ax where A is the local submatrix and as if x was not distributed.

5 Conclusion and future work

We described a sequential and an MPI basic implementations of the SpMV kernel in the context of the PETSc library, provided standalone C code and identified some challenges for the scientifc software verfication community. In the future, we'd like to add other variants, such as the MPI optimized implementation used by PETSc, and GPU implementations, to make the challenge code closer to real code.

References

[1] S Balay et al. (2024): *PETSc/TAO Users Manual*. Technical Report ANL-21/39 - Revision 3.22, Argonne National Laboratory, doi:10.2172/2205494.

[2] Yousef Saad (2003): *Iterative Methods for Sparse Linear Systems*, second edition. Society for Industrial and Applied Mathematics, doi:10.1137/1.9780898718003.