Sparse Distributed Matrix-Vector Multiplication

Matéo Pasquier

February 2025

1 Motivation

We want to implement the Power Iteration method in C. Power iteration is an eigenvalue algorithm used to find the dominant eigenvalue and its corresponding eigenvector of a matrix. It works by repeatedly multiplying an initial random vector b by the matrix and normalizing the result until convergence.
The method iteratively computes :

$$b_{k+1} = \frac{Ab_k}{||Ab_k||}$$

Here is a simple Python example that translates the above equation into code[1] :

```
import numpy as np

def power_iteration(A, num_iterations: int):

    b_k = np.random.rand(A.shape[1])

    for _ in range(num_iterations):
        # calculate the matrix-by-vector product Ab
        b_k1 = np.dot(A, b_k)

        # calculate the norm
        b_k1_norm = np.linalg.norm(b_k1)

        # re normalize the vector
        b_k = b_k1 / b_k1_norm

    return b_k
```

[1] Source

2 CSR format

We want to implement this using the *Compressed Sparse Row* format for the matrix A. Compressed Sparse Row (CSR) format is a memory-efficient way to store and operate on sparse matrices. It represents a matrix using three arrays:

- **nnz** : stores the non-zero values of the matrix.
- **col** : stores the corresponding column indexes for the values.
- **row** : Marks where each row of the matrix starts in the nnz array.

This format is very efficient for sparse matrices. While slightly trickier to set up it allows for lower memory consumption and faster computations. We will use predefined matrices from SuiteSparse Matrix Collection.

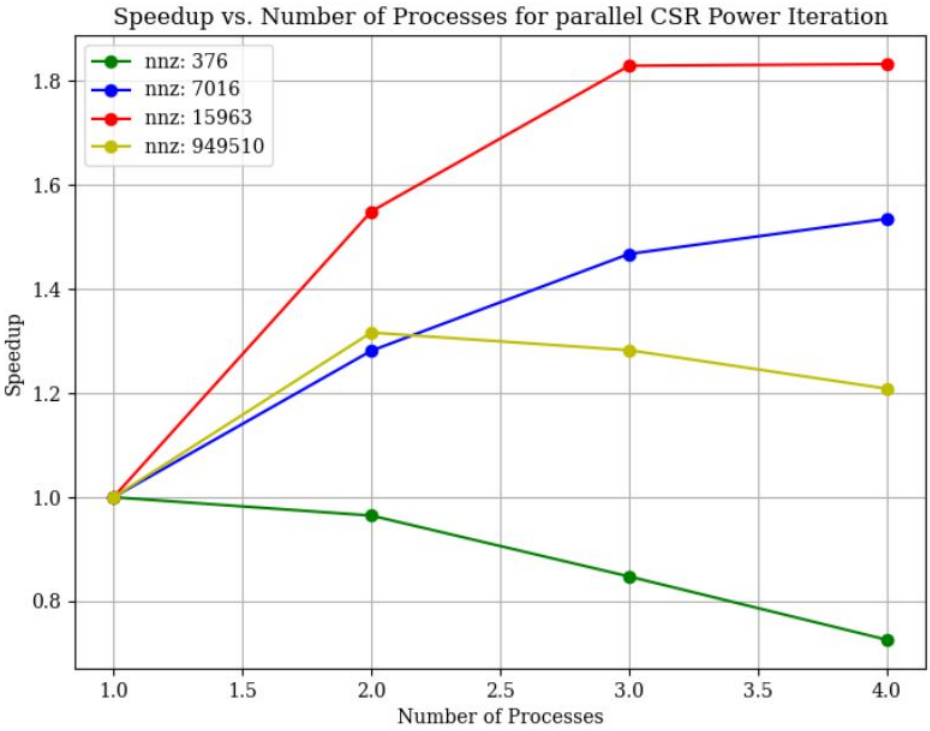
3 Parallelism

After having implemented a working serial version, we will parallelize this code using MPI and checking for speedups.
Before entering the *Power Iteration* method, all processes with read the matrix file and load into memory only their corresponding part of the data structure. We chose a more straightforward row-wise distribution but there may be other distribution patterns that could perform better (the primary bottleneck here probably will be communications during the *Power Iteration* method, a better distribution may help).
Each process will compute its own local matrix-vector product which produces a portion of the solution vector. These portions are then shared between processes so that they all possess the full vector, necessary for the next step. Another solution would be to have only one process gather the full vector and then broadcast only the calculated norm which would result in less data traveling around, but it would make all other processes idle while waiting for the value which isn't ideal either.

The initial random vector b will be generated _*only once before testing the methods* and we will keep using this same one, as to make sure the inputs always stay the same. To ensure correctness of the parallel version, we will simply compare the number of iterations until convergence with the serial version. Since *Power Iteration* is a deterministic method, we should observe the same convergence given that the inputs are strictly similar.

4 Results

We tested multiple matrix sizes with 4 different number of cores participating (strong scaling):



We can make several observations:

- With a matrix size that's too small ($nnz = 376$) there is no speedup and the execution time is increasing. This is a common effect with distributed programming where the synchronization penalty between processes outweighs the diminished workload gain.
- Medium-sized matrices perform better with a noticeable speedup despite the heavy communications of our implementation. The efficiency though is decreasing rapidly (as noted by the flattening of the curves), another expected result caused by the local workload decreasing while the communication cost remains.
- A too big of a matrix has its efficiency rapidly decreasing. While sharing the workload definitely reduced computational costs, a lot of data has to be sent every iteration and communications thus take a long time. This aspect is also limited by the bandwidth of the system; which is likely why

we see the speedup actually decreasing as the network is being "flooded" with data from every direction.

5 Conclusion and improvements

We successfully implemented a parallel **Power Iteration** method using the **CSR** matrix format and **MPI**. We profiled the code's performance using different matrix sizes, noticed speedups in different cases and addressed issues with rapidly decreasing efficiency. Another implementation would be to distribute the sparse matrix column-wise which could reduce the communication cost as only one array reduction would be needed during an iteration, as each process would compute a partial sum of the final vector.