Assignment 4

Jiannan Guo
Ramkumar Rajagopalan
Niklas Semmler
Christos Tsakiroglou

June 19, 2013

1 Question A

```
void multiply_vector_matrix(val, col_ind, row_ptr, vector, res) {
    /*
     * res = Matrix * vector
     *
         * Matrix in CSR format:
     * val - values
     * col_ind - column indices
     * row ptr - row pointers
     */
    int lower_bound = row_ptr[0]
    int upper_bound = 0
    int row = 0

    for (i = 1; i < len(row_ptr); i++)
        upper_bound = row_ptr[i]
        for (j = lower_bound; j < upper_bound; j++)
            res[row] += vector[col_ind[j]] * val[j]
        lower_bound = upper_bound
        row++

    return
}
```

2 Question B

One first idea is to distribute the work of the matrix multiplication $A \times b = c$ by row, meaning that each process gets n rows of A and the whole b vector. However, this process may result in highly unbalanced workload, since we dont take into account the actual number of non-zero elements in each row. Rather than dividing A matrix evenly by rows, we could distribute the rows based on their non-zero elements according to row_ptr, meaning that we calculate $\frac{length(val)}{p}$, where p is number of processes, and try to distribute a number of non-zero elements as close to that as possible without cutting a row into smaller pieces.

2.1 Two inconveniences

- Workload unbalanced
  In an extreme case of A:
  $$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$
  If first 2 rows are distributed to process 1, and last 2 rows for process 2, then, process 2 will have nothing to calculate at all.

- Since $N \times N$ matrix A is sparse but not banded matrix, the adjacency matrix of A might have a large bandwidth (in extreme case, N). But for a certain row of A, there might be plenty of zero gaps inside due to the sparseness. In this case, access to the memory where vector b is stored will be randomized to a large degree, which will reduce the speed.

To account for this inequality we evenly share the row_ptr instead of rows. Below you can find an example for two threads:

```
int lower_bound = row_ptr[0]
for (i = 1; i < len(row_ptr)/2; i++)
    upper_bound = row_ptr[i]
    for (j = lower_bound; j < upper_bound; j++)
        res[row] += vector[col_ind[j]] * val[j]
    lower_bound = upper_bound
    row++

lower_bound = row_ptr[len(row_ptr)/2+1]
for (i = len(row_ptr)/2+2; i < len(row_ptr); i++)
    upper_bound = row_ptr[i]
    for (j = lower_bound; j < upper_bound; j++)
```

```
    res[row] += vector[col_ind[j]] * val[j]
lower_bound = upper_bound
row++
```

3 Question C

3.1 Implications of the band structure on the multiplication and describe an improved version!

Here we have a band-structured matrix, trying to minimize the maximum index difference between non-zero elements, thus minimizing the bandwidth of the matrix. The smaller the bandwidth is, the less the complexity and the faster the calculation becomes.
Assuming an $N \times N$ band matrix with a bandwidth of B, we store it in memory as an $N \times B$ matrix. Thus, $N \times (N - B)$ amount of memory is already saved. Memory accesses are reduced and number of operations is also reduced.
As far as the parallelization is concerned, the non-zero part of the band matrix is divided into smaller sub-matrices, which are then distributed to different processes. However, there are always elements outside the diagonals which need to be communicated between processes. Moreover, each process only needs to know a part of vector b, which actually consists of values stored successively, so only needs to fetch a block of memory. On the contrary, at the simple sparse matrix case, vector b had to be accessed in various memory addresses, increasing the memory blocks accessed. A more detailed explanation follows.

3.2 Analyze and outline...

Analyze and outline the effects on the Jacobi implementation described above when using this improved version.
Take the equation below as an example (with band structure matrix $bw = 4$):

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & & & & \\ a_{21} & a_{22} & a_{23} & a_{24} & & & & \\ a_{31} & a_{32} & a_{33} & a_{34} & c_{31} & & & \\ a_{41} & a_{42} & a_{43} & a_{44} & 0 & c_{42} & & \\ & & d_{13} & 0 & b_{11} & b_{12} & b_{13} & b_{14} \\ & & & d_{24} & b_{21} & b_{22} & b_{23} & b_{24} \\ & & & & b_{31} & b_{32} & b_{33} & b_{34} \\ & & & & b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} = \begin{bmatrix} n_1 \\ n_2 \\ n_3 \\ n_4 \\ m_5 \\ m_6 \\ m_7 \\ m_8 \end{bmatrix} \tag{1}$$

It can be rewritten as:

$$\begin{bmatrix} A & C \\ D & B \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} N \\ M \end{bmatrix} \tag{2}$$

With this structure, assuming we parallelize Jacobi iterations with 2 processes, with first one holding $A \times X = N$ and second one holding $B \times Y = M$. But there are still sparse matrices C and D which are interface elements, where the communications occur. But domain decomposition storage, communication is reduced in a way that every block of vector will only need to exchange message with adjacent ones. In this case, m_5 and m_6 will be sent to second process for next iterative step and vice versa.