

# The Conjugate Gradient Method

December 6, 2012

## 1 The Conjugate Gradient

The Conjugate Gradient (CG) is an iterative method for the solution for sparse, symmetric and positive-definite linear systems. Remember, sparse matrices are matrices where most of the coefficients are equal to zero. The zero coefficients are not explicitly stored in memory and are not considered in all the operations involving the matrix; this allows to reduce the memory consumption as well as the complexity of matrix operations.

The CG method receives in input a starting solution  $x = x_0$  and refines it until the desired accuracy is reached (for example, until the norm of the residual is smaller than a certain threshold  $\|r\|_2 = \|b - Ax\|_2 < \varepsilon$  and/or when the number of iterations exceeds a maximum value). In its basic form, the CG is described by the algorithm below:

**Require:** a matrix  $A$ , a right-hand side  $b$  and a starting solution  $x_0$

- 1:  $d_0 = r_0 = b - Ax_0$
- 2: **while**  $\|r_i\|_2 > \varepsilon$  and  $i < itmax$  **do**
- 3:    $\alpha_i = \frac{r_i^T r_i}{d_i^T A d_i}$
- 4:    $x_{i+1} = x_i + \alpha_i d_i$
- 5:    $r_{i+1} = r_i - \alpha_i A d_i$
- 6:    $\beta_{i+1} = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i}$
- 7:    $d_{i+1} = r_{i+1} + \beta_{i+1} d_i$
- 8: **end while**

The method only requires four basic operations:

1. matrix-vector product  $y = \alpha Ax + \beta y$
2. dot-product  $v = x^T y$
3. vector 2-norm  $v = \|x\|_2$
4. vector sum  $y = \alpha x + \beta y$

where  $y$  and  $x$  are dense vectors,  $A$  is a sparse matrix,  $\alpha$ ,  $\beta$  and  $v$  are scalars.

The objective of this exercise is to parallelize each of these four operations independently using OpenMP in order to accelerate the Conjugate Gradient.

The mathematical details of the CG method are, in this context, not important and can be neglected.

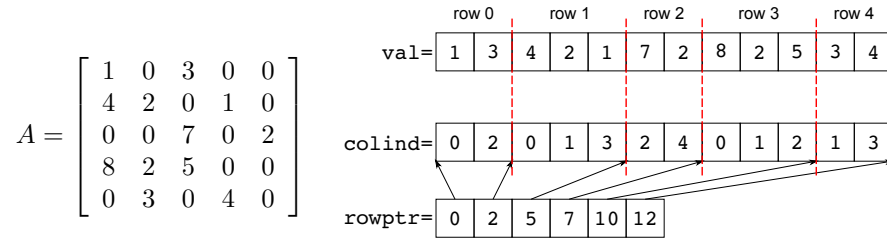
## 2 Sparse Matrix Format

Because the zero coefficients of the matrix must not be stored, a standard 2D array cannot be used for storing a sparse matrix. In this exercise, sparse matrices are represented in Compressed Sparse Row (CSR) format that consists of three arrays:

- **val**: this array of size **nz** contains the nonzero coefficients of the matrix sorted by rows (all the coefficients in the first row, then all those in the second row etc.)
- **colind**: this array of size **nz** contains the column indices for the coefficients in the **val** array
- **rowptr**: this array of size **n+1** contains pointer to the beginning of each row inside the **val** and **colind** arrays. For example **rowptr[3]=4** means that the first coefficient of row 3 is the 4th element of array **val** and its column index is equal to **colind[4]**. Therefore, all the coefficients of row **i** are between positions **rowptr[i]** and **rowptr[i+1]-1**.

where **n** is the size of the matrix and **nz** is the number of nonzero coefficients in the matrix.

Here is an example:



The sparse matrix-vector product  $y = \alpha Ax + \beta y$  is computed with the following routine:

```
void spmv(int n, int *rowptr, int *colind, double *val,
          double alpha, double *x, double beta, double *y){

    int i, j;

    for(i=0; i<n; i++){
        /* for each row... */
        y[i] = beta*y[i];
        for(j=rowptr[i]; j<rowptr[i+1]; j++){
            /* for each coefficient in the row... */
            y[i] += alpha*val[j]*x[colind[j]];
        }
    }
    return;
}
```

Note that each iteration of the outer loop handles one row of the matrix and each iteration of the inner loop handles one coefficient of a row.

### 3 Package content

In the `ConjugateGradient` directory you will find the following files:

- `conjugategradient.c`: this file contains the main program that reads a matrix  $A$  from a file, generates a right-hand side  $b$  and computes the solution  $x$  of the system  $Ax = b$  using the Conjugate Gradient method. **This file should not be modified.**
- `kernels.c`: this file contains the subroutines that perform the four basic operations described above. **This is the only file that must be modified** as described below.
- the remaining files contain auxiliary routines and can be safely ignored.

The code can be compiled with the `make` command: just type `make` inside the `ConjugateGradient` directory; this will generate a `main` program that can be run like this:



```
$ ./main matrix_file
```

where `matrix_file` can be `matrix1.rb`, `matrix2.rb` or `matrix3.rb`. When executed, the main program will read the matrix  $A$  from the file, generate the right-hand side  $b$  and find the solution  $x$  of the linear system  $Ax = b$  using the CG method. It will print the norm of the residual  $\|r\|_2 = \|b - Ax_i\|_2$  every 10 iterations and, at the end, will print the total number of iterations done, the residual of the final solution and the CG execution time.

The matrices can be found in the `/mnt/n7fs/ens/tp_abuttari/TP_SysCo/Matrices_BE/` directory. Before running the program for the first time, just make a copy of these files into the `ConjugateGradient` directory:

```
$ cp /mnt/n7fs/ens/tp_abuttari/TP_SysCo/Matrices_BE/matrix*.rb .
```

## 4 Assignment

-  Use OpenMP to parallelize the four subroutines in the `kernels.c` file. In case you used the `omp parallel for` construct, consider using different scheduling types.
-  Analyze and compare the performance of the parallel code using one or two threads. Report in the `responses.txt` file the number of iterations and the execution time for the three example matrices `matrix1.rb`, `matrix2.rb` and `matrix3.rb`. Comment on the observed results: did you observe any speedup (reduction of the execution time) using 2 threads instead of 1? did you observe any difference in the number of iterations using 2 threads instead of 1? Is the method still converging when using 2 threads? Did you observe any difference when using a dynamic scheduling instead of a static one? can you explain this difference?

### Advice.

- Note that some operations are easier to parallelize than others. The vector sum routine `axpby` is the easiest, the `dot` and `norm2` are a bit more difficult and, finally, the matrix-vector product `spmv` is the hardest. It is recommended to begin with the easier routines first and to validate the correctness of the result each time one routine is parallelized.
- It is reasonable to expect that the number of iterations to convergence is slightly different when using 2 (or more) threads instead of 1. The difference, however, should be very small (not more than 10 iterations for the given test matrices). Therefore, if you observe a big difference in the number of iterations or if the method does not converge anymore it means that the code was not correctly parallelized.