INSTITUTE FOR MICROELECTRONICS

nssc@iue.tuwien.ac.at

Numerical Simulation and Scientific Computing I

Exercise 3

Submission is due on Monday, December 16th 2019, 10am (submit to nssc@iue.tuwien.ac.at)

• Include the name of all group members and the group number in your submission

• The binaries should be callable with command line parameters as specified below

• Submit your answers (including the plots/visualizations) in the PDF format per task.

• Submit everything (Makefiles, Sources, Binaries, PDF) together as one zip-file per task.

General information

• Use the GNU compiler collection for the submitted binaries using at least the following compiler flags:

```
-std=c++11 -Wall -pedantic
```

• For OpenMP examples, additionally add the flag:

```
-fopenmp
```

• Use (and assume) the double precision floating point representation for all matrices/vectors and function evaluations.

INSTITUTE FOR MICROELECTRONICS

Task 1 – Conjugate Gradients Implementation (4 points)

Implement the non-preconditioned Conjugate Gradients method using your own data structures (i.e., only the C/C++ standard libraries).

In particular, you should

a) Implement a CCS data structure for symmetric matrices.
b) Implement a reader for s.p.d. matrices using the *Matrix Market* (.mtx) format.
c) Implement a matrix-vector multiplication for symmetric matrices while keeping the matrix in the CCS format and making use of the symmetry.
d) Prescribe the right-hand side to the given solution $x^* = [1, \dots 1]$, that is $b = Ax^*$.
e) Implement the non-preconditioned CG algorithm with starting guess $x_0 = [0, \dots, 0]$.
f) Have your program output to the terminal $\frac{\|r_k\|_2}{\|r_0\|_2}$ and the error on the A-norm $\|e_k\|_A = \|x^* - x_k\|_A = \sqrt{e_k^T A e_k}$ after all iterations.
g) Have your program be callable by: `./cg filename iterations` e.g. `./cg s3rmt3m1.mtx 10000`
h) Additionally, plot $\frac{\|r_k\|_2}{\|r_0\|_2}$ and $\|e_k\|_A$ as a function of the iteration number for the matrix *S3RMT3M1* from *Matrix Market* and discuss your results.

Additional information and hints:

-You can find the matrix *S3RMT3M1* here:
https://math.nist.gov/MatrixMarket/data/misc/cylshell/s3rmt3m1.html

-You are **not** allowed to use the *Matrix Market* C or FORTRAN libraries.

-The matrices are symmetric, which means only the lower triangular part is present in the files. You should also only store the lower triangular matrices in your CCS format.

-Pay attention so you don't calculate the same dot product or matrix-vector product twice.

-You can use the provided pseudocode for the matrix-vector multiplication as a guide (see next page).

INSTITUTE FOR MICROELECTRONICS

Pseudocode: Symmetric CCS Matrix * Vector product

$$\boldsymbol{Ax = y \rightarrow y_i = \sum_j A_{ij} x_j}$$

```
// initialize y to 0
y=[0,...,0]
// iterate over all rows
for i in [0,max_rows)
     // iterate over all columns in front of the diagonal
     for j in [0,i)
          // search in each column if there is an entry
          // with row_index= i
          for index in [JA[j],JA[j+1])
               if (i == IA[index])
                    y[i] += V[index]*x[j]
                    break
     // once the diagonal is reached,
     // do the calculation for the column (i.e., transposed)
     for index in [JA[i],JA[i+1])
          y[i] += V[index]*x[I[index]]
```

INSTITUTE FOR MICROELECTRONICS

Task 2 – Conjugate Gradients in Eigen (1 point)

Use *Eigen*'s Conjugate Gradients to solve $Ax = b$ where A is the matrix *S3RMT3M3*, $b = Ax^*$ where $x^* = [1, \dots 1]$. Compare your implementation to Eigen's implementation using two different Eigen preconditioners: The plot should show $\frac{\|r_k\|_2}{\|r_0\|_2}$ as a function of the number of iterations k for

(1) your implementation from Task 1,

(2) *Eigen*'s Conjugate Gradients using `DiagonalPreconditioner`, and

(3) *Eigen*'s Conjugate Gradients using `IncompleteCholesky`.

Discuss your results.

Additional information and hints:

- *S3RMT3M3 is* different matrix to Task 1! Search and download it by yourself from *Matrix Market.*

-You might consider using *Eigen*'s `selfadjointView()` function in order to correctly consider the symmetry of the matrix.

-In order to accurately control the number of iterations in *Eigen*, it is convenient to set the tolerance to a very low number.

INSTITUTE FOR MICROELECTRONICS

Task 3 – Monte Carlo Integration (3 points)

Implement a numerical integration for a one-dimensional function $f(x) \rightarrow y$ using a Monte Carlo technique. Parallelize your implementation using OpenMP.

In particular, you should

a) use an exclusive RNG for each thread
b) use a single shared RNG to generate the seeds used for the exclusive RNG of each thread
c) use a fixed seed for the shared RNG
d) print to the terminal:
   - the approximation of the integral computed by each thread
   - the final approximation of the integral
   - the runtime of the integration
   - the number of threads used
   - the number of total samples
e) make your program callable by:
   `./mcint FUNC XMIN XMAX YMIN YMAX SAMPLES`
   (e.g. `./mcint SINX 0 6.283185 -1 1 1e9`)
   where:
   `FUNC` defines the function to be integrated:
   - `SINX` stands for $\sin(x) \rightarrow y$
   - `SIN2XINV` stands for $\sin^2(\frac{1}{x}) \rightarrow y$
   - `XCUBE` stands for $x^3 \rightarrow y$

   `XMIN, XMAX, YMIN, YMAX` define the integration box boundary
   `SAMPLES` defines the number of samples per thread
f) compile and test your program on your machine **and on the IUE-cluster**
g) use your program to plot the runtime for 1, 5, 10, 20, 40, and 80 threads on a single node on the cluster when using `XMIN=`$-\pi$`, XMAX=`π`, YMIN=-1, YMAX=1` for each of the three functions and 1 million samples per thread (first plot). Create a second plot now using 100 million samples per thread for the same thread numbers and all three functions as before.

Additional information and hints:
-Make sure shared objects are not accessed from different threads if the access is not thread-save.
-Find a SLURM job submission example script in the handouts and additional information in the slides of lecture 8.

INSTITUTE FOR MICROELECTRONICS

Task 4 – Parallel Jacobi Solver (2 points)

Parallelize your Jacobi solver developed in Exercise 2 using OpenMP.

In particular, you should

a) identify the for loops eligible for OpenMP parallelization and discuss your choice
b) adopt the implementation of Exercise 2 so that not a fixed number of iterations is performed, but instead the program stops the iterations once the runtime exceeds 10 seconds
c) make your program callable by:
`./stencilomp resolution numthreads` e.g.
`./stencilomp 64 4`
where `resolution` defining the grid spacing (same as in Exercise 2) and `numthreads` referring to the number of OpenMP threads to be used
d) print the
   - number of performed iterations,
   - the total runtime of the iteration loop, and
   - the average runtime per iteration

   to the terminal
e) compile and test your program on your machine **and on the IUE-cluster**
f) investigate the scalability of your parallelization by plotting the speedup of the average iteration runtime for 1, 2, 4, 8, 10, 20, and 40 threads on the cluster for three different OpenMP scheduling policies for the parallelized loop:
   - `schedule(static)`
   - `schedule(static,1)`
   - `schedule(dynamic)`

   Plot and discuss the results for `resolutions` 256, 512, 1024, and 2048.

Additional information and hints:

-You can also use the reference implementation of the Jacobi solver distributed with the handout of this exercise as a starting point for the parallelization