INSTITUTE FOR MICROELECTRONICS

nssc@iue.tuwien.ac.at

Numerical Simulation and Scientific Computing I

Exercise 3

Submission is due on Thursday, **January 09th 2025, 14:00** (to **TUWEL**)

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

• Do not include binary executables but build instructions (for GNU/Linux) and a Makefile.

• Include any scripts you used to generate the plots from your recorded data.

• Submit your report (including the plots/visualizations) as one .pdf-file per task.

• Submit everything as one zip-file.

General information

• 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, i.e., only storing the lower triangular part.
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 $x^* = [1, \ldots, 1]$, that is $b = A x^*$.
e) Implement the non-preconditioned CG algorithm with starting guess $x_0 = [0, \ldots, 0]$.
f) Output to 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 the final iteration.
g) Have your program be callable by: `./cg filename iterations` e.g. `./cg bcsstk13.mtx 10000`
h) Plot $\frac{\|r_k\|_2}{\|r_0\|_2}$ and $\|e_k\|_A$ as a function of the iteration number for the matrix *BCSSTK13* from *Matrix Market* and discuss your results.

Additional information and hints:

-You can find the matrix *BCSSTK13* here:

https://math.nist.gov/MatrixMarket/data/Harwell-Boeing/bcsstruc1/bcsstk13.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 do not 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).

Pseudocode: Symmetric CCS Matrix * Vector product

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

```
// initialize y to 0
y=[0,...,0]
// iterate over all columns
for j in [0,max_columns)
    // extract the index for V and IA
    for index in [JA[j],JA[j+1])
        val = V[index]
        i = IA[index]
        // regular M-V multiplication
        y[i] += val*x[j]
        // check to not double-count diagonal
        if (i =/= j)
                // transposed multiplication
                y[j] += val*x[i]
```

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 *BCSSTK11*, $b=Ax^*$ where $x^*=[1,\dots 1]$. Compare your implementation to Eigen's with respect to convergence 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`.

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

Discuss your results.

Additional information and hints:

- *BCSSTK11 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.

-You can use *Eigen*'s `.error()`(check documentation for Conjugate Gradients) to easily calculate the residuals.

INSTITUTE FOR MICROELECTRONICS

Task 3 – Monte Carlo Integration (3 points)

Implement numerical integration for a one-dimensional function $f(x) \rightarrow y$ using Monte Carlo integration. 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 at the end of all iterations:
   - 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 SAMPLES`
   (e.g. `./mcint SINX 0 6.283185 1e9`)
   where:
   `FUNC` defines the function to be integrated:
   - `SINX` stands for $\sin(x) \rightarrow y$
   - `COS2XINV` stands for $\cos^2\left(\frac{1}{x}\right) \rightarrow y$
   - `X4M5` stands for $5x^4 \rightarrow y$

   `XMIN, XMAX,` define the integration boundaries, `SAMPLES` defines the total number of samples. The samples $x^{(k)}$ are uniformly distributed in (`XMIN`, `XMAX`) and $\widehat{I}_m = m^{-1}(XMAX - XMIN)\sum_{k=1}^{m} y^{(k)}$.
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=-`π`/2, XMAX=`π`/2,` for each of the three functions and 10 million samples.

Additional information and hints:
-Make sure shared objects are not accessed from different threads if the access is not thread-safe.
-Find a SLURM job submission example script in the handout.
-Do not forget to submit your functioning SLURM script.
-All your plots and discussions should be in a single PDF.

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) Adapt your program from Exercise 2
   (1) by disabling/removing the option to set the Dirichlet boundary condition and instead fixed values (-100 for u_W and 100 for u_E),
   (2) by disabling/removing the option to set the source region and instead use no source at all, and
   Finally your program should be callable using three command line parameters:
   ```
   # ./solver name resolution iterations, e.g.
   ./solver reference 100 10000
   ```
   where the meaning of the parameters is the same as in Exercise 2.
c) Print the total runtime to console.
d) Compile and test your program on your machine **and on the IuE-cluster.**
e) Investigate the scalability of your parallelization on the cluster using a resolution of 2048 and 100000 iterations. Plot the speedup of the total runtime for 1, 2, 4, 8, 10, 20, and 40 threads for three different OpenMP scheduling policies for the parallelized loop:
   ```
   schedule(static)
   schedule(static,1)
   schedule(dynamic)
   ```
   Discuss the scaling and the effect of the scheduling policies.

Additional information and hints:

- You can also use the reference implementation of the Jacobi solver distributed with this handout of this exercise as a starting point for your parallelization.
- Do not forget to submit a functioning SLURM script.
- All your plots and discussions should be in a single PDF.