High-Performance Computing

Exercises

Prof. Dr. Jan Dünnweber

Research Unit on Distributed Systems and Operating Systems
Faculty of Computer Science and Mathematics
Regensburg University of Applied Sciences

Solving Equation Systems in Parallel

Exercise 1: Include the Jacobi-Method from lecture 5 in your matrix code

Jacobi-Algorithm in Java

```
public void solve(Vector b, Vector xn, Vector xi, int iter)  {
  for (int approx = 0;  approx < iter;  ++approx)  {
    for (int i = 0;  i < y;  ++i)  {
      double sum = 0.0;
      for (int j = 0;  j < x;  ++j)
        if (i != j)
          sum += get(j, i) * xn.get(j);
      xi.set(i, (b.get(i) - sum) / get(i, i));  }
    xi.copy(xn);  }  }
// ...
```

- Use "MatLab"© to verify that your code computes

$$x_{m+1,i} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_{m,j} \right), \; i = 1, \ldots, n$$

- Parallelize your code *line-by-line* and perform some speedup measurements (process some blocks of lines sequentially)

The SOR-Method

Exercise 2: SOR can be implemented as follows

Successive Over-Relaxation

```
public final void runSOR(double omega, int nIter)  {
    double omegaOverFour = omega * 0.25;
    double oneMinusOmega = 1.0 - omega;
    for (int p = 0; p < nITer; ++p)
      for (int i = 1; i < m.x - 1; i++)
        for (int j = 1; j < m.y-1;  j++)
          m.set(i, j, omegaOverFour * (m.get(i - 1,j)
                    + m.get(i + 1, j) + m.get(i, j - 1)
                    + m.get(i, j + 1) + oneMinusOmega * m.get(i, j); }
```

- Verify that the code works by implementing a JUnit test that compares the results of the Jacobi and the SOR-approximation
- Describe the corresponding *polytope* and try to find an appropriate *schedule* and *placement* to perform a *space-time* mapping *manually*
- Port the *loop nest* to C (use arrays to represent the matrices) and run *Pluto* from `pluto-compiler.sourceforge.net` to generate a parallel version and conduct some performance experiments