Simulation and High-Performance Computing

University of Kiel, September 28th to Oktober 9th, 2020.

Exercises: Day Five

CSC Computational Science Center

General Remark

Today, we look at two state-of-the-art approaches to handling large systems of equations. These approaches are normally used in two distinct situations.

The first situation is that a partial differential equation is discretized on a series of nested grids or meshes that get progressively finer. This usually leads to a hierarchy of *sparse* matrices. For that kind of problem, the so-called multigrid methods are considered to be extremely effective. In principle, their order of convergence does *not* depend on the mesh size of the finest mesh.

The second situation is that a non-local problem - such as the gravitational problem we already touched on earlier during this course - leads to a *dense* matrix. For many problems of that kind, a very successful approach is to approximate the matrix with a lower rank one that is based on a tree of clusters that contain the degrees of freedom.

Main Exercises: Applications of Lecture 09

a) Several Linear Iterations With the 1D Model Problem

As preparation for implementing the multigrid method, we will first look at a few linear iterations that can be used to handle systems of linear equations. Those iterations do not converge very fast but require very little computational effort, are easy to implement and fairly flexible.

In practice, the linear iterations have two main uses: Firstly, they can be utilized as so-called *preconditioners* in conjunction with methods such as the CG method. Secondly, they play an important role in the implementation of the multigrid method - which is what we will look at in todays exercises.

Implement *one* step with the Richardson, the Jacobi and the Gauß-Seidel iterations for the specific matrix arising from the 1D model problem - see the exercises from day 4 for that matrix - in the functions `rich_step_1d`, `jac_step_1d`, `gs_step_1d` in the file `gridfunc1d.c`. Within the lectures, these iterations are described for an implicit multiplication with the matrix arising from the 2D model problem. Work out how to transfer the given algorithms to the 1D case. In case of the Richardson iteration use the damping parameter $\Theta = 0.5\,h^2$.

Test your linear iterations and observe their order of convergence.

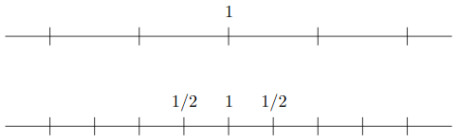
b) The Multigrid Method With the 1D Model Problem

As described in the lectures, the multigrid method is usually based on nested grids or meshes on which a partial differential equation is discretized - leading to different matrices, each one corresponding to a grid level. In all previous methods for these problems, we always only solved the problem on the *finest* grid. The multigrid method, however, involves *all* matrices, not just the one from the finest grid.

In order to make use of the whole hierarchy of grids, we need a *prolongation* and a *restriction* to transfer the values of gridfunctions between grids.

In the 1D case, you can think of the grid as just a line of equidistant points. Similarly to the

2D case described in the lectures, **implement** a prolongation and a restriction for the 1D model problem in the functions **prolong_gridfunc1d**, **restrict_gridfunc1d** in the file **gridfunc1d.c**. The prolongation should describe the following transfer from the top line to the bottom:



The restriction from the fine to the coarse grid should essentially be the adjoint of the prolongation, but scaled in such a way, that it takes a gridfunction that is exactly 1 at all points of the fine grid to a gridfunction that is exactly 1 at all points on the coarser grid. To keep in line with the lectures, implement the prolongation as *adding* values to a fine grid function and the restriction as *replacing* the values of a coarse grid function.

Now, **implement** a complete multigrid algorithm for the 1D model problem in the file **gridfunc1d.c**. The multigrid algorithm described in the lectures (slide 18) is independent of the problem dimension and can, thus, be used as orientation. Further Augment the algorithm by allowing for more than just one smoothing step at each grid level. Then **test** your algorithm.

Additional Exercises: Applications of Lecture 10

a) Cluster Tree Technique for the Gravitational Potential

Evaluating the gravitational force or the gravitational potential of a many body problem is a challenging task, since for $n \in \mathbb{N}$ bodies, the evaluation of the gravitational potential needs order of n many operations per evaluation. Evaluating all forces would lead to an $\mathcal{O}(n^2)$ algorithm and is not feasible for very large problems like simulating our milkyway.

Therefore, some kind of approximation technique is needed to reduce the amount of operations for such computations.

The approach of splitting the space into subsets called *clusters* and approximating the far-field interaction between two remote clusters has become famous because it reduces the complexity down to $\mathcal{O}(n \log(n))$.

Your task is to **implement** in the file `exercise_tree_gravitation.c` the function `eval_cluster` that will evaluate the gravitational potential at some point $x = (x_1, x_2, x_3) \in \mathbb{R}^3$ using the *cluster tree algorithm*.

Now three cases can occur:

- x is *admissible* to the cluster s, then use the approximation from `eval_approximation`, which uses tensor-interpolation.
- x is not admissible to the cluster s, but s has sons. Then proceed with recursion to the sons of s.
- x is not admissible to the cluster s and has no sons. In that case evaluate the potential directly with `eval_full`.

The functions `eval_approximation` and `eval_full` have to be **implemented** in the file `exercise_tree_gravitation.c` as well.

For the direct evaluation, it is necessary to permute the bodies stored in the structure `state`. More specifically, the position of the i-th body is

`st->y[0][s->idx[i]], st->y[1][s->idx[i]], st->y[2][s->idx[i]].`

When evaluating the tensor-interpolation approximation the weights are precomputed and can be loaded from the array `s->z`. Again, the correct indexing is the crucial part. Assume you have the interpolation point ξ_ν, where $\nu = (\nu_1, \nu_2, \nu_3)$ is a multi-index. Then the corresponding weight can be found at position

```
nu = nu1 * (m+1) * (m+1) + nu2 * (m+1) + nu3;
```

In both cases you can use the function `potential` to evaluate the potential between two points $x, y \in \mathbb{R}^3$.

Please provide feedback about the format/difficulty/length of the exercises so that we can adjust accordingly. Contact us via Email or on the OpenOLAT forum.