Exercise for Course

Parallel High-Performance Computing

Dr. S. Lang

Return: 21. January 2016 at the beginning of the exercise or earlier

---

Starting with this exercise we examine a variant of the gravitational N-body problem (*GNBP*), this is also introduced in the lecture in more detail. We are going to parallelize the problem with all techniques, that we learned up to now: tiling, *OpenMP*, *PThreads*, MPI and *CUDA* parallelisation and will each measure the acquired MFLOPs rate.

Please read first the section at the end of the exercise sheet, where the used variant of the N-body problem and the provided code is explained. On the homepage you find additional hints regarding the files. Get confident with the code, that calculates the movement of several bodies in empty space, that are attracting each other by gravitation.

For each parallel variant an own kernel exists. These can be compiled with the present Makefile by `make`. A kernel consists roughly out of the functions `accelerate()`, `leapfrog()` and `main()`. The function `leapfrog()` is nearly always equal and implements the time stepping scheme. The most important function is `accelerate()`, since it performs the main task. This function realises an algorithms with complexity of N^2, this function shall be implemented in different parallel variants. Some variants are already finished (Sequential, Tiling, *OpenMP*), for the others this function has to be worked out in the appropriate kernel.

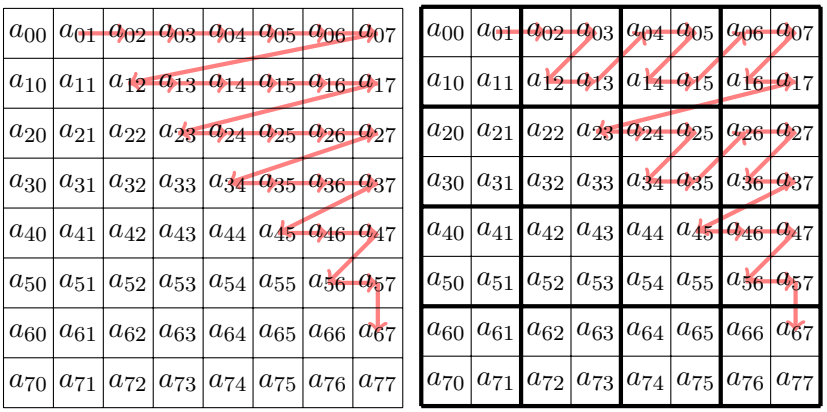
There are other more complicated approaches that reduce the complexity to $O(N \log N)$ or even $O(N)$. The $O(N \log N)$ variant will be discussed in the lecture. Please consider in every case the hints at the end of the exercise sheet and on the Homepage!

Task 23 N-Body Problem: Sequential, Tiling, *OpenMP* (5 points)

In the serial variant of the N-body problem the function `acceleration()` calculates the acceleration of all bodies. For a body i it has to iterate over all other bodies $j = i+1, \ldots, N-1$ and has to get access to their positions `x[j]` and masses `m[j]`. Then it can calculate the accelerations $a_{ij} = -a_{ji}$ and accumulates the values in `a[i]` and `a[j]`.

You find in the archive the sequential, tiling- and *OpenMP* variants implemented completely. The according kernel gets as parameter N, the number of time steps to be preformed as well as the choice of the output steps. The timestep dt is set to $dt = 0.001s$, this value should be appropriate for our purposes. Preset are initial conditions with equal distribution. With this run the following simulation once: `./nbody_vanilla 40 100000 100`, and consider the result with Paraview!

First we investigate the tiling approach. In the following figure the serial iteration order for the calculatin of the accelerations is presented on the left. On the right the optimised ordering for tiling is visualized with a block size of $B = 2$.



You find these variants in the kernel file `nbody_tiled.c`. Recognize how the original loop over `i` and `j` has been split into loops over the blocks and inner loops:

```
for(I = 0; I < N; I += B)
  for(J = I; J < N; J += B)
    for(i = I; i < MIN(N, I+B); ++i)
      for(j = MAX(i+1,J); j < MIN(N, J+B); ++j)
      {
        /* code goes here */
      }
```

The *OpenMP* parallelisation uses the tiled variant and is realized in the Kernel file `nbody_openmp.c`.

Task

You have to decide for one of the two initial conditions `plummer` (preset) or `cube`. Use the choosen condition for all following tasks. Now perform a perfomance analysis of the sequential variant and of the one with tiling and *OpenMP* by measureing the *MFLOPs* rate. Therefore use values of $N >= 100$ and different tiling sizes B. Adapt the number of time steps such that several time steps are executed and calculate the average *MFLOPs* rate for a single time step. For the test measurements in the pool (with the previous machines) with tiling no improvement of the *MFLOP*-Rate has been acchieved. On other architectures albeit a performance gain as expected could be gained. Now introduce a table with the measured rates over the problem size. This table will be extended with the values of the other parallelisation types and then we can compare each of them.

Free Willi Task

To investigate the reason for the missing performance gain in more detail, we want to change the data layout, to load the data in another sequence into the cache. For this define a data structure `Body`, that stores `m, x, v` and `a` for each body and create an array `Body[n]` for the `n` bodies. The temporary accelerations can be handled in an array as before. Now change the functionen `accelerate` and `leapfrog` such that they load bodies with complete information. In contradiction to the storage of individual arrays the components now reside consequtively in memory. Is an improvement measureable? As alternative realize your own ideas to improve the cache efficency.

Hint

To validate the correctness of parallel variants youd could compare the generated *VTK* files. Here, the simulation results can distinguish in accuracy at some positions (1e-14 to 1e-16). The provided script `fuzzy_diff` can compare two *VTK* files regarding a given tolerance, e.g.:

```
./fuzzy_diff sequential.vtk parallel.vtk 1e-14
```

Use the script to validate your parallel implementations.

Task 24 GNBP with MPI (10 points)

In this task we parallelise the N-body problem with *MPI*. In the MPI version of the code each process contains only a part of the coordinates and masses. For simplicity you can assume foŕ this task that $N \, mod \, p = 0$ holds.

Implement a solution scheme for the N-body problem using *MPI*. An approach would be that the involved processes commumicate blocking in a ring topology with edge coloring. The skeleton `nbody_mpi.c` is already provided, it contains routines to generate the initial conditions and reading / writing of the `VTK` files, that are already parallelised. The existing code initially works for a single process.

Parallelise now the routine `accelerate_mpi`, such that it implements the proposed MPI communication scheme. Perform then calculations with the same parameters and initial conditions as in the previous task. Measure the computing times and calculate the speedup of the parallel program. Check also the simulation results (time dependent positions) by comparing the results of the parallel computation with the sequential ones with the `fuzzy_diff` script. Now enrich your result table with the measured *MFLOPs* rates and establish a plot of the *MFLOPs* rate over N.