Assignment Sheet #10

Submission deadline: 21.06.2016, 12:00

Organizational hints and related background were presented in the lecture and also can be found in the lecture notes (uploaded to Stud.IP). Send your solution files in a compressed ZIP archive to sven.bingert@gwdg.de.
You can find the programs or templates referred in the assignments in the directory in Stud.IP.

Exercise 1 – Lottery (5 Points)

In `lottery.cpp` you will find a serial implementation of a box with bouncing balls.

a) Read it, understand the basic program flow, compile it with `g++ serial.cpp common.cpp -o serial`, run it and visualize the result with the program `./visualize` which you will find in `visualize.zip`. Answer in very general terms:

  1) What does init_particles() (line 28) do?

  2) What happens in Line 34, 39-43 and 49-50?

  3) Why do you think are the particle positions only saved every SAVEFREQ time steps?

b) Parallelize the program using MPI. Consider the following questions and guidelines

  1) The basic idea is to distribute the particles over the set of processors. So each processor applies the force and moves a subset of all particles in each time step.

  2) Is it sensible to parallelize the init_particles() routine?

  3) You can assume that the number of particles is always a multiple of the number of processors. You can abort the program if this is not the case.

  4) For the sake of simplicity you might keep an array of all particle positions in every process and exchange it after each time step. Which collective communication function fits best for this purpose? What is the drawback of this approach in terms of scalability?

  5) How shall the save routine behave in your MPI version? Implement it.

  6) Compile your program with `mpiCC`, run it and check whether it produces the same output like the serial one.

c) Discuss some possibilities to further optimize the algorithm and parallelization in terms of overall runtime.

Exercise 2 – *Conduction* (10 Points)

Write a parallel program that simulates a square metal plate that was heated once in one spot in the middle. You will find a rough guiding framework for the serial implementation in `conduct.c`. Watch out for "FIXME" strings in the code.

The plate is modeled as a NxN two dimensional grid of square cells. The cell at position (i,j) has the temperature T[i][j].

The simulation runs in discrete time steps. In each time step a cell changes its temperature influenced by its neighboring cells according to

We assume that heat just blows away at the borders and enforce a value of T[][]=0 at hypothetical neighbors *behind* the border of the grid i.e. T[*][-1]=T[N+1][*]=T[-1][*]=T[*][N+1]=0 at all times.

The temperature is a real number between 0 and 1:

In this exercise we will use the following initial conditions but your program shall be able to run from any set of initial conditions which might be imposed in `init_cells()`.

All cells have a temperature T[i][j]=0 except the 16x16 innermost cells. This is already implemented in the template.

You can test and visualize your serial implementation with the program `visualize_conduct`. An example-run of 1000 time steps on a 128x128 grid is given in `sample_conduct.txt`.

1) Complete the serial implementation given in `conduct.c` **(1 point)**
2) Why should one use MPI instead of a pure OpenMP parallelization? **(0.5 points)**
3) Parallelize it with MPI following the steps given below. The basic idea is to have each processor calculate a partition of the grid at each time step. A processor should decide on its own depending on its rank which part of the grid it computes. You can assume that the number of MPI processes *m* is a square of an integer, the grid a square and the number of rows *N* dividable by .
   a) As the calculation of each cell only depends on the value of its neighbor in the previous time step we only need to communicate the outer most border of each partition between the processors. How would you partition the grid between the processors in order to minimize the communication in this approach? Why? **(1 point)**
   b) Implement version 1: Each processor has a copy of the whole grid at every time but only calculates a part of it. After each step the changed data is distributed among everybody. **(2 points)**
   c) Implement version 2: Each processor only holds the part of the grid that it has to calculate. After each time step the borders are exchanged between neighboring grid processors. Remember the ghost-cell approach. **(2 points)**
   d) Version 3: Add a sensible OpenMP parallelization to one of your 2 MPI versions. Think of a computing model where you distribute MPI processes over a number of compute nodes with a shared memory CPU architecture. Each node handles one MPI process which itself shall be multithreaded. Explain your solution. **(1 point)**
   e) Try to use gcc auto vectorization on version 3 to further utilize vector units (eg SSE) of the CPU. Does it give a benefit? (0.5 point)
4) Add a save-routine to version 3 and visualize your result. Run your final MPI version on the PBS system of the Computer Science institute. Why did you choose to do the save-routine like the way you did? Are there other possibilities? Discuss this. **(1 point)**
5) Do some wall clock timings of your programs showing the dependency of the run time on the grid size, the number of nodes and the number of threads. Discuss the results. What is faster: n MPI processes or n threads on one node? Will the benchmark change with a different set of initial conditions? **(1 point)**

Everybody who implements a better visualization program will get a bunch of cookies (but no extra points). The following guidelines hold:

1. It shall be able to visualize at least a 1024x1024 grid on a desktop PC.
2. It has to have an open source license.

Extra cookies will be provided for:

1. The avoidance of java.
2. Cross platform implementations.
3. A new, non-ASCII and/ or compressed data format.
4. An input format that can be written by multiple MPI processors in parallel (i.e. might consists of several files that hold parts of the grid).
5. Usage of (for-this-task) atypical programming languages.
6. Fancy ideas – a web version? A console program? A hardware LED implementation? A Mobile App? A