Laboratory exercises (4)

Introduction HPC, TU Delft, course 2001-2002

dr. A.G.M. van Hees

December 4, 2001

4 N-body problem

Introduction

In the this exercise we do no longer aim to calculate values of a function at fixed points in space (the grid points). We consider a collection of particles that move due to their mutual forces. The computation can in principle be performed rather straightforward. Evaluate the total force on each particle by considering all the other particles. This can easily be done if all positions are known. So all we have to do is to choose an algorithm to let the particles move in time, i.e., keep track of their positions and velocities, and evaluate forces whenever necessary. This is typically a Verlet algorithm. The complexity of the force evaluation is of order N^2. Therefore the evaluation of of the total force on each particle is the computational bottleneck. Each particle has interactions with all the $N - 1$ other particles. Particles that are far apart are therefore, from this computational point of view, just as important as nearby particles. From a physical point of view this is of course nonsense. The main question therefore is how we can exploit in a numerical computation the fact far away particles are much less important than nearby particles.

Several methods are developed. A rather simple one is the ParticleInCell method. Space is divided in a large number of cells. Particles are member of one cell. Cells are geometrically arranged, and lets assume that they remain fixed in space, whereas particles can move. A group of adjacent cells can be assigned to one processor of a parallel computer. The information to be communicated between processors in such a method basically is the complete content of some cells. Here we can see the analogy with the previous exercises. The domain consists of cells, neighboring cells need each others information, just as neighboring grid points do. So every time-step information of those cells that are along the boundary of a domain has to be communicated to the processor that owns the neighboring domain. This is similar to the situation with Poissons solver where each iteration-step the information between grid points along a boundary had to be communicated. This approach works well if the interaction is short ranged, i.e., particles that are not in the same or in two neighboring cells do not feel a mutual interaction. The amount of computational work that has to be done to evaluate all non vanishing forces for short-ranged interactions is proportional to the number of particles N. The ParticleInCell method exploits this feature. Without the concept of cells one would have to test each pair of particles to determine whether they are close enough to feel their mutual interaction. So the evaluation of all forces remains of order N^2 also for short ranged interactions.

Several steps in this ParticleInCell algorithm are novel or somewhat more complicated than in the previous two exercises. The amount of information stored for each cell is not constant since particles may enter or leave cells. This means that particles must have the possibility to migrate between cells and thus also between processes. Of course one wants to keep the additional communication due to particles moving from a cell in one domain into another cell in another domain as small as possible. Reassigning particles to cells need not be done every time-step if the domains are allowed to overlap and the time-step is small enough.

Information stored in cells along the boundary of a domain has to be communicated every time-step to the neighboring domain, but the number of particles within a cell may change any time particles have been allowed to move. As a consequence, in the send-receive operations the receiving process does not know in advance how much information it will receive from each of its neighbors. With the Poisson solver the amount of information was fixed during the calculation. A consequence of this variable amount of data in a transfer is that there arises the problem of load balancing. How can one guarantee that all processes have more or less the same amount of work to do? With the ParticleInCell method we do not attempt any load balancing, but assume for simplicity that particles remain rather evenly distributed over the cells. In summary, parallelization of the simple order N^2 algorithm is straightforward, but the method in itself is very unattractive due to its scaling behavior. Parellelization of other methods such as ParticleInCell, that have a better scaling behavior, is less straightforward.

Another approach to the N-body problem is to build more complicated data structures than only the list of particles, with their mass, positions and velocities. A quadtree is an example of a two-dimensional N-body hierarchical data structure. Each node in a quadtree contains information such as the total mass, and the center-of-gravity of all the particles inside that node. Moreover it contains pointers to its nonempty subnodes, that are nodes themselves. Each node with more than one particle has subnodes. With such a data structure it is not necessary to traverse the quadtree to its full depth in order to evaluate the force on a particular particle, say p_i, with limited accuracy. Nodes that describe a region of space that is far enough from particle p_i need not be opened. The force of all the particles inside such a node on that particular particle can be approximated by the force on p_i of the total mass inside the node placed in the center-of-mass of that node. This way one need not to go into the full depth of a tree, but has a means to trade accuracy for performance. The parallelization strategy in this case is in principle very simple. Each process builds no more of the total quadtree than is necessary to evaluate the forces on the particles it owns. Particles are grouped such that those assigned to one process are in the same region of space. This can be done with the Orthogonal Recursive Bisection method. Similar to the ParticleInCell method, where cells change because of particles moving in and out, also the quadtrees have to be rebuilt after each time-step. Each process builds the quadtree for the particles it owns, and inserts the particles that it receives from all the other processes. Each process can deduce from its own quadtree and the region of space owned by any other processor which particles or clusters of particles it needs to send to that other processor. Here it should be emphasized that we can prevent that each process sends the information of all the particles it owns to any other process. This would lead to the construction of the global quadtree by each processor. Instead each process can construct a personalized message to any other process without communication. The remote process may receive in a message particles that in reality are clusters of many particles. However, the sending process knows in advance that the receiving process need not resolve the full structure of such a cluster. As far as communication is concerned, particles have the possibility to move from one processors domain to another. Otherwise formulated each time step one needs to (re)distribute all the particles over the processors and each processor builds a personalized message for all the other processors.

Each process now has the information to proceed for one time step without any further communication to the others. It simply builds its own quadtree, that must be consistent with the global quadtree (which is never built), and traverses it to the required depth for in order to evaluate the force on each of the particles it owns.

The exercises aim to investigate the performance of the ParticleInCell method or the Barnes-Hut method, and to illustrate the various types of communications that are required in these N-body methods.

The N-body problem

In this exercise we are going to study a many-body system. If all particles interact with each other there are $N \times (N+1)/2$ pairs of particles to consider. If all the forces are evaluated the complexity of the problem is $O(N^2)$,

However, for a short-range interaction, only the 'nearby' particles contribute to the force on a particular particle. In this exercise we want to exploit that by creating cells that are fixed in space. The cells have a known topological structure. We assume that they form a mesh in 2-d. Cells have a size that is related to the range of the interaction. Any particle only feels the interaction of particles that are within a distance R_c. Now we choose the cells such that particles in a cell do not feel the interaction with particles that are at a distance of more than 1 cell in either direction. Hence a particle in a cell only interacts with particles in 9 cells. The gain one has is that one does not even have to check whether two particles are close enough (this still has order N^2 complexity), but that one only has to run over cells. The number of particles in 9 cells is limited since their is in general a maximum density. Hence the number of interaction partners that is visited for any particle, is limited by a constant. The total complexity of such an algorithm is of order N.

This reasoning or algorithmic improvement still has nothing to do with parallelism. It is just a method that exploits knowledge about the system, knowledge about the behavior of the particle-particle interaction. If the interaction does not exactly vanish at distances larger than R_c, one has to decide whether the introduced errors are acceptable or not. In general the behavior of a many-particle system is chaotic anyway, and the smallest change in initial conditions (or algorithmic precision) may eventually lead to large differences in the trajectories of individual particles.

With an algorithm that scales as order N it is much easier to go to large systems than with algorithms that scale as N^2. Anyway, when the problem size becomes big enough, also the order N algorithm may benefit in performance by a parallelization effort. Next we will describe the strategy that is followed in order to create a parallel implementation.

Parallelization of the Particles-in-Cell N-body method

Just like with the Poisson solver discussed in the first exercise, the domain is divided into a large number of grid cells. In the Poisson solver the information, i.e., the value of a discretized function, lives on gridpoints. In this N-body problem the information – particles with position and velocity – lives within the gridcells.

The strategy that is followed in order to create a parallel version is to create subdomains that each contain many cells. Cells are non-overlapping rectangles that cover the complete computational domain, organized in a grid. This is similar to the situation for the Poisson-solver where the subdomains contained many gridpoints. Each subdomain owns cells for which it is responsible. A subdomain also has ghostcells from which it needs information to update the contents of its own cells. Information from ghostcells is received from the owner. The subdivision of the domain into subdomains with ghostcells is completely analog to the procedure in the first exercise where there were ghostpoints.

A domain has a particle-list with information about particles that occupy the cells within the domain. Particles in this list are all assigned to one cell. This can be a cell that is owned by the process or it can be a ghostcell. This list of particles (or better: pointers to particles) can grow or shrink. This can happen if a domain receives messages from its neighboring domains that say how many particles there are in each ghostcell. If this number of particles is larger than the number the process thinks there are in a ghostcell, apparently some particles have moved into the region that is important for this process.

If a process is aware of a particle, also if it is in a ghostcell, it can always find out when a particle is no longer important for the region that is maintained by the process.

Messages between (neighboring) processes that communicate the occupation numbers of the cells along the borders of a domain need not be sent every timestep, but can be sent for instance only once every 10 timesteps. The justification of this depends on application specific details, such as the range of the interaction, the size of the time-step, the size of the cells, etc.

Now we will describe the assignment of particles to cells in some more detail.

Re-assignment. This means that a particle is in the particle-list of a process and stays there. Upon reassignment it becomes member of the cell in which it belongs at that moment. The process that reassigns a particle must have up to date information about its position. For the particles that are owned by the process this is no problem. For particles that are not owned by the process, but that nevertheless are in the process' particle-list, the up to date information is obtained every timestep from a neighboring process that owns the particle.

Assignment. A particle must become member of a cell when it enters it. If this cell is a ghostcell of a particular process and if the particle just enters the region covered by that process, it is not in the particle-list of that process. Only a process that owns a particle can detect when it enters a region this that is important for another process. Each process therefore sends the occupation numbers of all its bordercells to any process that needs these cells. Note that the bordercells of a process are the ghostcell of the neighboring processes.

Assignment means that one entry is added to the particlelist of a process. This added particle becomes member of the cell to which it is assigned. This must be a ghostcell. New particles can only become assigned to a cell when the process that owns all the particles in that cell sends a message that contains the number of particles in that cell. The process that receives this message can check how much this number deviates from the number of particles it already has in that cell.

De-assignment. If a process does no longer need a particle for the evaluation of the force on its own particles, because the particle moved out of the region 'seen' by the process, the particle is deassigned. This implies that the particle is removed from the particle-list of the process. The last particle in the list is moved to the vacant position, and the particle counter of the cell from which the particle is removed is reduced by one.

All the possibilities for a process, that checks all the particles in its particlelist for reassignment are summarized below. 'From' means the cell to which the particle belongs, whereas 'to' means the cell to which a particle has to assign because its present position requests it.

- Particle is still within the limits of the cell to which it is assigned. Nothing changes.
- Particle has moved from own cell to another own cell or to a ghostcell. Reassign.
- Particle has moved from own cell to outside. **Not allowed!** This particle is going too fast. For this process it is easy to deassign the particle, but by design such a particle is not assigned to another cell in another process, therefore stop the program, and prevent this from happening again.
- Particle has moved from a ghostcell to own cell or to another ghostcell. Reassign.
- Particle has moved from a ghostcell to outside. De-assign. This particle is no longer needed.

The following three situations might occur for particles that are outside the view of a process.

- Particle moves from outside to own cell. **Not allowed!** The program is not designed to deal with such a situation. Domains do not administrate the particles that leave their cells and where they go to. They only count the number of particles in their border cells. Apparently a particle that behaves like this moves too fast. it should first appear in a ghostcell!
- Particle moves from outside to ghostcell. In this situation a particle p_i enters the region that is important for the particles that are owned by this process. However, this process is not yet aware of the existence of particle p_i. It is told by the process that owns p_i that it will

receive information about p_i – its coordinates – in the future. Therefore: **Assign p_i to the appropriate ghostcell.**

- Particle moves from outside to outside. The process is not aware of the existence of such particles. No problem.

The number of particles seen by a particular process may change in time. Next we will discuss the type of information that is to be exchanged between processes, and the actions that have to be undertaken by the processes in order to manage their tasks.

1. Exchange information about number of particles.

   After particles are assigned each process knows the number of particles in its own cells. For the cells that are ghostcells, i.e. owned by another domain, the number of particles is communicated. The cells involved in this communication are the cells along the border. This communication is similar to the `Exchange_Borders` function in the first exercise. The only difference is that there are now 4 additional neighboring processes (NE, SE, SW and NW) to/from where only 1 integer number has to be communicated. Note that the cell at one of the four cornerpoints of the region owned by a domain is a ghostcell of three neighboring domains.

2. Exchange information about the particles. After each time step each process has to send the new positions of all particles in its bordercells to those processes that maintain a copy of these cells as a ghostcell. Otherwise formulated, each process receives the new positions of particles that are assigned to its ghostcells from the domain that is the owner.

   The receiving process knows how much particle-data it must receive. The receiving process knows how many particles have to be updated in each ghostcell. For instance a domain receives 9 (x, y) pairs, positions of particles in its ghostcells. These ghostcells are owned by the process that sent the data.. Previously it had received from the same sender the message containing the occupation numbers of these ghostcells. For instance, with 2 integers that sum up to 9, e.g. 3 and 6.

   The receiving process now knows that the first 3 (x, y) pairs should be associated with 3 particles in the first ghost cell, whereas the other 6 (x, y)-pairs should be associated with 6 particles in the second ghostcell. Therefore the receiving process must store the right amount of pointers to particles in all its ghostcells.

Exercises and performance aspects of the N-body code

4.1

Check whether the PIC method in the serial N-body code speeds up the performance of the program as expected. How much do you expect to gain in speed for a system that contains 10×10 cells?

4.2

How many particles are expected in bordercells? Give an expression in terms of N_{part} (number of particles), N_{cell} (number of cells) and N_{proc} (number of processes). Assume that the domain is a unit square (in 2-D), that the cells are equally sized squares, and that the particle distribution is uniform.

4.3

Measure the extra time it takes to do the reassignment of particles. Compare this time with the time it takes to exchange the particle-data of the border-cells.

4.4

Plot trajectories of a single particle for 1×1, 5×5, 20×20 cells. Do this by writing (x,y) coordinates to a file and e.g. calling gnuplot. Within gnuplot use:
plot "filename' with lines

What can you conclude from this about the accuracy and behavior of the system of particles? Is the assumption justified that interaction with remote particles can be neglected?