ID1217: Parallelize Particle Simulation

Mathilda Strandberg von Schantz

March 2017

1 Introduction

This assignment is about taking a toy particle simulator that runs in O(n^2) and making it run in O(n), and then taking three parallel versions of the program, using Pthreads, OpenMP and MPI, and attempting to make them run in time close to $\frac{O(n)}{p}$ where p is the number of processors.

2 Sequential program

The sequential particle simulator uses a simple array of particles. For a certain number of simulation steps every particle is compared with every other particle, and if they are within a certain range a repulsive force between them is applied. Then the particles move given their position, velocity and acceleration. However, the density of the particles is set sufficiently low so that given n particles, only O(n) interactions are expected, making this O(n^2) comparison between every particle redundant. In order to reduce the number of comparisons the particles were divided into bins. The bins are divided so that each bin have approximately the size of the interaction radius. Thus, instead of comparing every particle with every other particle and then applying the force if they are in range, only the particles in the neighboring bins are gone through. The bins were modelled as a struct with four parameters: the number of surrounding bins, the number of particles in the bin, and two arrays, the first one with the id:s of the surrounding bins and the second one with the id:s of the particles in the bin. An array of these structs are then created, where the row and column of the bin is given by its index. There is naturally some overhead when using the bins: firstly the bins need to be created and populated by the particles and then, in every timestep of the simulation, the bins are updated with the new information of the particles' positions.

The run time of the sequential program was measured with the help of a bash script which ran the program for different number of particles, ranging from 1000 to 5000, where the execution time was given as the median value of five seperate executions. This was compared with the run time of the initial serial time, as seen in figure 1.

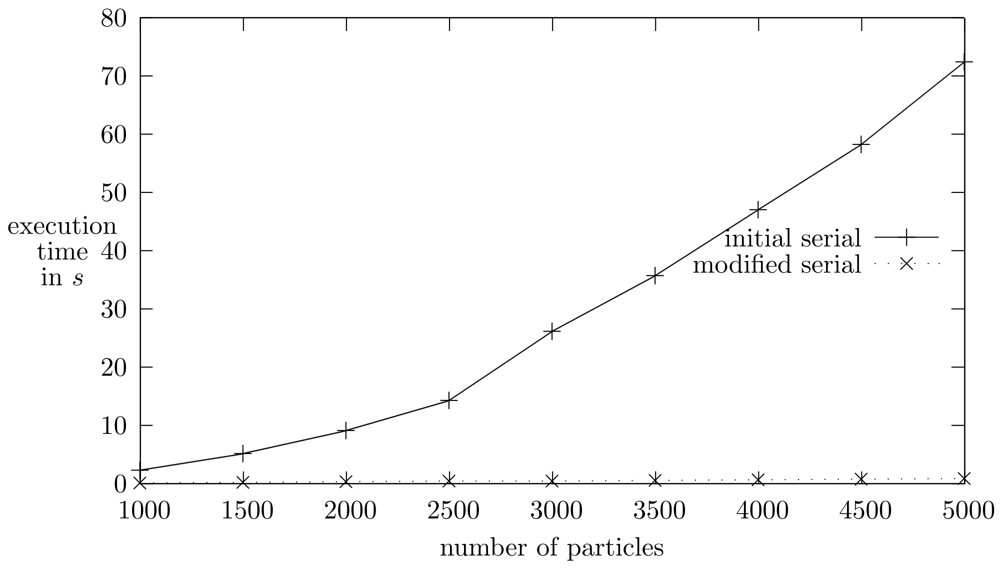

Figure 1: Plot that shows run time for initial vs modified serial implementations.

3 Parallel program: shared memory

In addition to the serial program there were three parallel implementations: one using the C-library Pthreads, one using the API OpenMP and one distributed version using message-passing with the message-passing system MPI.

In the Pthreads implementation the threads each get a subset of the particles to control. A barrier is used first after applying force to the particles, and then after the particles are moved. The bins are not divided up between the threads even though they too are a shared resource. This is not an issue when the particles move or when the force between them is applied because in these scenarios the bins are only read, not written to. It does however become an issue when the bins are updated, which is the last thing done in every timestep. Therefore this is done by only one thread: the root thread with id zero. To make the program slightly faster, a lock combined with a way of making sure only one thread gets the lock per timestep could have been used. Then the updating of the bins could have been done by an arbitrary thread and not specifically the thread with id zero. However, this was easier to program. After the updating of the bins is made there is another barrier, to make sure no thread starts in the next iteration before the bins are sufficiently updated. When debugging the program, an assertion was made in every timestep of the program, making sure that the number of particles was constant.

The OpenMP implementation is similar: for-loop parallelism is used when calculating the forces between particles and when moving the particles. In this way the array of particles and bins respectively are divided between the threads in these calculations. As with the Pthreads implementation, the updating of the bins is done by only one thread, through the directive #pragma omp master. In this program too, the assertion that the number of particles is constant was made in the debugging process. No explicit barriers are used in the OpenMP implementation, since there are implicit barriers after the parallel for loop constructs.

A benchmark was then done on the three programs done so far, with the number of particles ranging from 1000 to 10000. Here too, the execution time is the median of five separate executions. The plots are shown in log-log scale in figure 2. The Pthreads implementation is slower than the OpenMP implementation - for a smaller number of particles it is slower than the serial implementation as well. That is interesting since OpenMP is built on Posix threads, which makes it seem that they should be roughly equivalent speed-wise. Some overhead is introduced with the barriers in the Pthreads implementation: however, removing them doesn't result in a significant time decrease. A more convincing bottleneck seems to be in the updating of the bins. This is natural, since it is a critical section where every bin and particle are gone through. It is still linear by n, but nonetheless time-consuming. An alternative to this could be to not update bins, but rather the particles. This could also introduce some overhead in having to check for particles added to your bin.

It is also impossible the updating of the bins, that is the reinsertion of the particles into the bins, could have been made faster by using a more straightforward way of representing the bins, such as a two-dimensional integer array. (That was explored later in the MPI implementation.)

Speedup plots of the Pthreads and OpenMP implementations were made as well, as seen in figure 3. Here the superiority of the OpenMP implementation is obvious. However, it doesn't scale perfectly: the speedup with 8 cores borders on 3, not 8.

4 Parallel program: distributed memory

The MPI program was more of a challenge than the previous programs. The plan was to do the exact same thing as in the Pthreads program: divide the particles between the threads, here with the Scatterv operation, and then globally share the bins by using the Bcast operation. The thinking was then that, as before, the calculations are done on a subset of the particles and all the threads have a local copy of all of the bins. Then, after each thread has done their calculations on their portion of the particles, one of the thread updates the bins and rebroadcast them to the other threads. There is a lot of overhead here with broadcasting the bins at each timestep, especially given that Bcast is a blocking call. However, there needs to be a synchronization of the threads here anyway: all the threads need to have the same version of the bins before going to the next iteration.

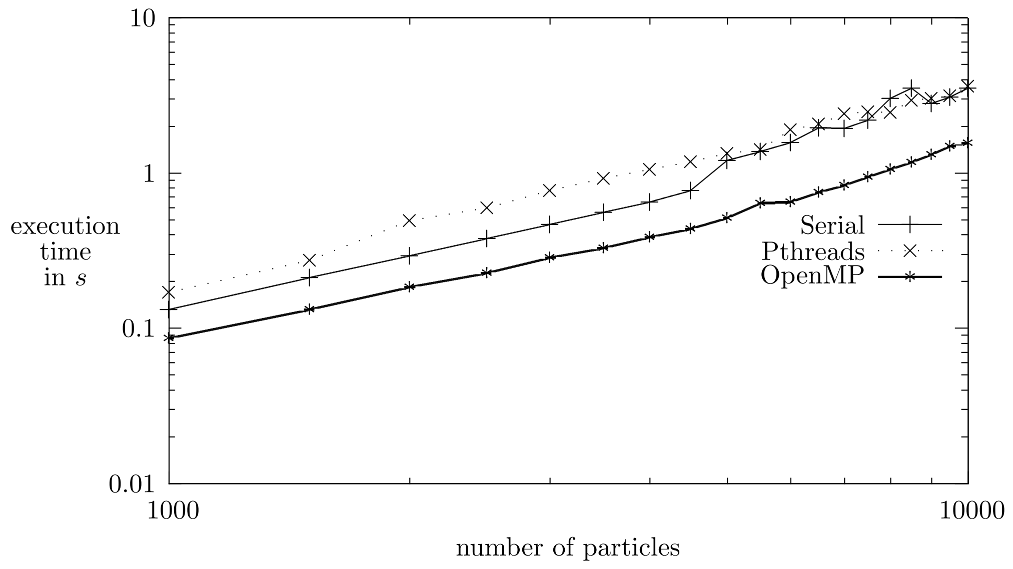

Figure 2: Execution time in log-log scale for both serial and parallel implementations. The parallel implementations are run on two cores.

One alternative to this method would be to divide the bins between the particles and then send the particles between the threads one by one as they move over into another thread's bin. This seems more natural: to divide the static bins between the threads and then send and receive the particles as they move between the processor's different territories. The sending of the particles could then be done with non-blocking point-to-point communication such as Isend and Irecv. However, this implementation seemed harder and more different from the Pthreads and OpenMP implementations, which is why the alternative with the collective communication mentioned above was chosen.

However, there were some problems implementing this version where the particles are divided between the threads and the bins "shared": the nature of MPI made it hard to broadcast the bins because of the structure of them. Since the bin struct had two arrays with a dynamic size there seemed to have to be some struct serialization of the struct involved to be able to broadcast the bins. This was attempted, but a lot of segmentation faults ensued. Therefore a new representation of the bins were used, were they different parts of the struct bin were divided into several arrays. This version was never completed however, because of lack of time.

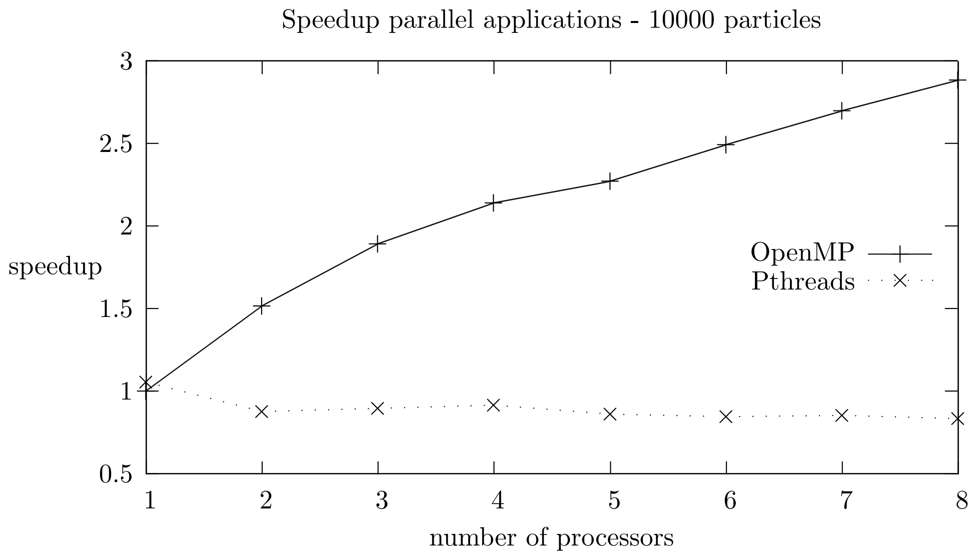

Figure 3: Speedup plots of Pthreads and OpenMP implementations. Benchmarks were done on a school Redhat computer with 8 cores.

5 Discussion

In this assignment, the OpenMP implementation showed better scalability and was easier to implement. Pthreads is more low level: it introduces more control over the threads but also more difficulties. OpenMP was grateful in this instance because it was a typical example of for-loop parallelism. MPI introduced a lot of hinders in the way the bins were represented. Whether it was better or not speedup-wise is not possible to say since that implementation wasn't completed.