Parallisation concept

Fatema Alkhanaizi

January 18, 2019

The first loop in updateBody() is for computing the force. The force for each particle is the sum of all forces acting on it from the rest of the particles in the system; a nested loop is used to implement this.
Each iteration of the outer loop i is independent and the inner loops deals with computing the value for the force vector for the ith particle using scalar variables. There are a lot of duplicated computations as the forces computation always start from 0 until N - number of bodies. It is possible to optimize this computation e.g. the force when particle i=0 and particle j=1 is the same as the force when particle i=1 and particle j=0 in terms of magnitude, so it is possible to evaluate them in one step and update the force vector for each particle. However, this could cause a race condition in which if running the nested loop in parallel two or more threads will try to modify the same particle at the same time. If a critical section is added this will slow the computation for the threads which would give worst results than if ran in serial due to the overhead introduced by parallization in OpenMP. To reduce the access to the force array, scalar variables were setup to compute the force vector and duplicated computations remained. The minimum distance is computed accross all iterations in the both loops. It could be determined in a reduction steps following Bulk Synchronous programming (BSP) method which OpenMP allows; it will be reduction for the minimum value,and it will add a bit of an overhead. As OpenMP performs on a shared memory, any variable initialized outside the parallel region will be shared so the scalar variables inside the loop that will be constantly modified will be set to private.

The second loop in updateBody() deals with updating the movement of the particles. Each iteration i is independent in terms of access to x and v arrays however computing the maxV creates a dependency. For maxV, a reduction step with OpenMP is possible as maxV is a scalar (reduction for max), thus it is possible to run this loop concurrently. The scalar variables inside the loop will be set to private; similarly to the previous loop.

For meaturing performance, relative speed and efficieny measurement will be used:

$$S(p) = \frac{t(1)}{t(p)}$$

$$E(p) = \frac{S(p)}{p}$$

By following Amdahl's law, an increase in speed and effiency is to be expected when the number of cores is increased while the number of bodies remains constant (the speed reaches a limit however as more cores are added).

```
    for (i = 0; i < NumberOfBodies; ++i) {
        xi = x[i][0];
        yi = x[i][1];
        zi = x[i][2];
        fx = 0.0;
        fy = 0.0;
        fz = 0.0;

        // compute force on particle i
        for (j = 0; j < NumberOfBodies; j++) {
            ...
            minDx = std::min(minDx, r2);
        }

        force[i][0] = fx;
        force[i][1] = fy;
        force[i][2] = fz;
    }
```

```
        for (i = 0; i < NumberOfBodies; i++) {

            x[i][0] = x[i][0] + timeStepSize * v[i][0];
            x[i][1] = x[i][1] + timeStepSize * v[i][1];
            x[i][2] = x[i][2] + timeStepSize * v[i][2];

            mt = timeStepSize / mass[i];

            v[i][0] = v[i][0] + mt * force[i][0];
            v[i][1] = v[i][1] + mt * force[i][1];
            v[i][2] = v[i][2] + mt * force[i][2];

            ...

            maxV = std::max(maxV, V);
        }
```