Parallisation concept

Fatema Alkhanaizi

January 12, 2019

Write a report which discusses which parts of the routine updateBody() are well-suited for parallelisation. Focus on OpenMP only. The solution to this step is to be handed in as a one-page PDF called solution-step5.pdf. Please phrase your statements in terms of well-known performance models, i.e. predict what efficiency you expect to obtain.

```
    for (i = 0; i < NumberOfBodies; i++) {
        force[i][0] = 0.0;
        force[i][1] = 0.0;
        force[i][2] = 0.0;
    }

```

It is possible to vectorize this loop. Each iteration is independent thus can be ran in its own thread (MIMD).

```
    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
        for (j = 0; j < NumberOfBodies; j++) {
            ...
        }

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

```

Computing the force applied on each particles required nested loops. The outer loop iterates over each particle, the inner loop is used to computed

and aggregate the force applied on the particle. As there is an inner loop parallization should be handled with care; all iterations for the outer loop must be to a degree independent. There aren't any complix conditions or early exits from the loop so it is possible to parralize it as long as the calls in the inner loop aren't dependent to any iteration from the outer loop. All variables must be treated as private variables for each iteration.

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

        dx = xi - x[j][0];
        dy = yi - x[j][1];
        dz = zi - x[j][2];

        const double r2 = dx * dx + dy * dy + dz * dz;

        fr2 = sigma2 / r2;
        fr6 = fr2 * fr2 * fr2;

        F = 48.0 * epsilon * fr6 * (fr6 - 0.5) / r2;

        fx += dx * F;
        fy += dy * F;
        fz += dz * F;

        minDx = std::min(minDx, r2);
    }

```

The inner loop contains a couple of complications that could affect parallization. firstly, the force is being aggregated; a reduction step would resolve this following the BSP architecture. minDx is a global variable, in other words it is changed even accros the outer loop iterations; it is the minimum value for the whole iteration for the outer loop - part of the outer loop scope. A fix for this is to do a two step reduction once for the inner loop above and once again for the outer loop; this can be handled by setting a temporary variable that will keep track of the minimum value for the inner loop, then comparing the temporary variable across the iterations of the outer loop.

```
    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];

```

```
9         v[i][0] = v[i][0] + mt * force[i][0];
10        v[i][1] = v[i][1] + mt * force[i][1];
11        v[i][2] = v[i][2] + mt * force[i][2];
12
13        const double V = std::sqrt(v[i][0] * v[i][0] + v[i][1] *
      v[i][1] + v[i][2] * v[i][2]);
14
15        maxV = std::max(maxV, V);
16    }
17
```

It is possible to run this loop in parallel as no value is accessed and modified for more than one iteration except maxV which would need to be reduced.