High Performance Computing for Science and Engineering I.

Exercise Set 2

Florian Mahlknecht

2019-10-21

1 Brownian Motion

Having a deeper look on the equation

$$x_i(t + \Delta t) = x_i(t) + \xi_i^{(t)} \sqrt{\Delta t} \tag{1}$$

we notice that the particles are completely independent and do not influence each other. So we can calculate the final position for each particle one by one by iterating over time. This flips the order of the for loops in the provided serial implementation. The following listings show the code in the steps requested in the exercise.

With OpenMP the wall time implementation becomes simply listing 1

```
31 // Returns current wall-clock time in seconds
32 double GetWtime() {
33   return omp_get_wtime();
34 }
```

Listing 1: GetWtime() implementation

fmahlknecht@student.ethz.ch
19-945-351

```
84  #pragma omp parallel shared(xx)
85      {
86        std::default_random_engine gen;
87
88        auto random = gen();
89        // Seed generator
90        int seed = (omp_get_thread_num() + 1) * random % gen.max();
91        gen.seed(seed);
92
93        // parallel initialization
94        std::uniform_real_distribution<double> dis(-0.5, 0.5);
95  #pragma omp for
96        for (size_t i = 0; i < N; ++i) {
97          xx[i] = dis(gen);
98        }
99
100 #pragma omp single
101       {
102         xx0 = xx;
103         wt0 = GetWtime();
104       }
105
106       std::normal_distribution<double> dis2(0., std::sqrt(dt));
107 #pragma omp for
108       for (size_t i = 0; i < N; ++i) {
109         for (size_t m = 0; m < M; ++m) {
110           xx[i] += dis2(gen);
111         }
112       }
113
114 #pragma omp single
115       {
116         wt1 = GetWtime();
117         wtime_walk = wt1 - wt0;
118       }
119     }
```

Listing 2: Parallelized time stepping

Listing 2 shows the parallel version of the time stepping calculations. Note that random number generators have been used *inside* the parallel region, guaranteeing independent random values for all threads. The timing measurments are moved inside a `#pragma omp single` region, which implies a barrier before, guaranteeing repeatable measurements.

The vector `xx` is shared over the whole region. The actual calculations are in the two for loops, which are not collapsed to guarantee a correct splitting of access ranges on the vector throughout all threads.

Florian Mahlknecht
fmahlknecht@student.ethz.ch
19-945-351

```
36 std::vector<double> GetHistogram(const std::vector<double>& xx) {
37   std::vector<double> hh(nb, 0);
38   std::vector<std::vector<double>> hh_t(omp_get_max_threads(),
39                                         std::vector<double>(nb, 0));
40 #pragma omp parallel shared(hh, hh_t)
41   {
42 #pragma omp for
43     for (size_t i = 0; i < xx.size(); ++i) {
44       int j = (xx[i] - xmin) / (xmax - xmin) * nb;
45       j = std::max(0, std::min(int(nb) - 1, j));
46       hh_t[omp_get_thread_num()][j] += 1;
47     }
48 #pragma omp for // no collapse here to avoid eventual race conditions
49     for (size_t i = 0; i < nb; ++i)
50         for (size_t j = 0; j < omp_get_max_threads(); ++j)
51             hh[i] += hh_t[j][i];
52   }
53   return hh;
54 }
55
```

Listing 3: Histogram parallel implementation

Finally, listing 3 shows the parallel histogram implementation.

In the serial version only one for loop was used, summing up all the occurrences of a particular x_i at the calculated position `j`. In the parallel version, the naive approach of putting this in a parallel for loop, would end up in race conditions.

Therefore, in the shown implementation, *two* for loops are used. The first one is extended to sum up in different arrays, one for each thread. In the second one the work from the different threads is merged into the final result, also in a parallel fashion.

For completeness, the make file (CMake) is shown in listing 4.

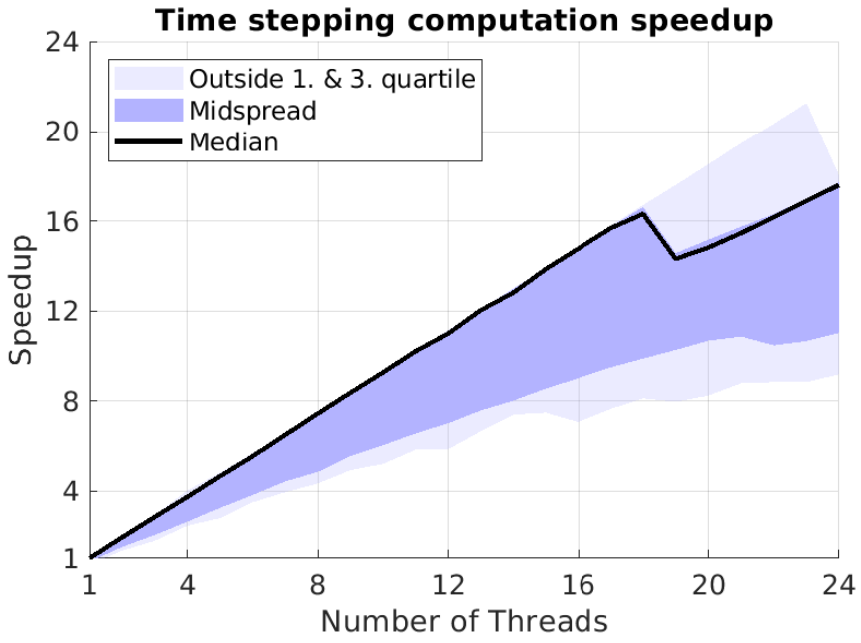
```
1  cmake_minimum_required(VERSION 2.8)
2  set(CMAKE_CXX_FLAGS_RELEASE "-O3")
3
4  project(brownian_motion)
5
6  set( project_sources
7     main.cpp
8     cacheflusher.cpp
9  )
10
11 add_executable(${PROJECT_NAME}
12   ${project_sources}
13 )
14
15 find_package(OpenMP)
16 target_link_libraries(${PROJECT_NAME} PUBLIC OpenMP::OpenMP_CXX)
```

Listing 4: Histogram parallel implementation

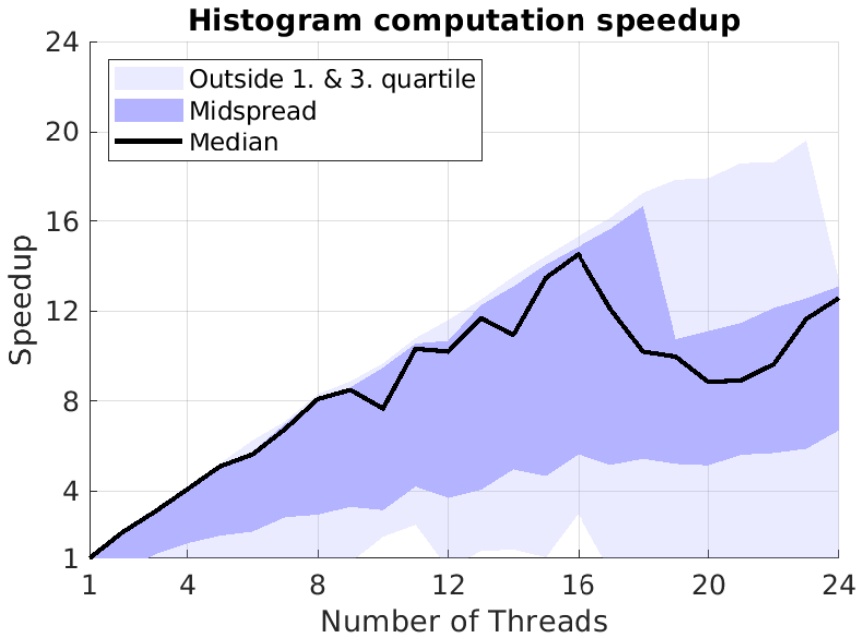
fmahlknecht@student.ethz.ch
19-945-351

1.1 Speedup

Figure 1 presents the obtained results in terms of speedup.



(a) Time stepping



(b) Histogram

Figure 1: Computation speedup evaluated on 100 executions

100 executions have been performed, each of them executing the complete algorithm 24 times, with the respective amount of threads. The parameters used were $N = 5000000$ particles and just $M = 10$ iterations. This increased the necessary time for the histogram generation to 1.2 ms to 14.6 ms with the respective 24 to 1 core, being more representative than us. This however, without

Florian Mahlknecht
fmahlknecht@student.ethz.ch
19-945-351

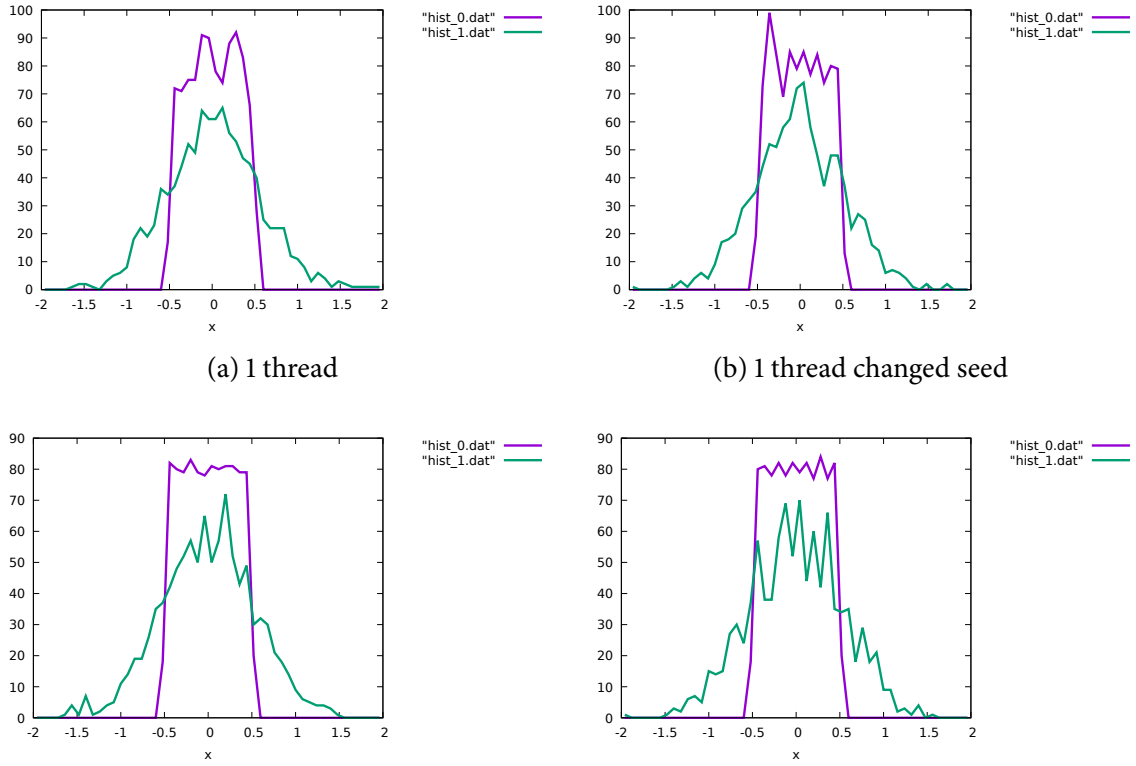
increasing the time stepping computation load to much, by keeping the iterations lower.

However, as fig. 1b shows, the numerical values still present a larger average deviation from the mean value in the computations for the histograms. Generally, the scaling is not perfect but rather good.

The workload in all threads is basically identical, as long as the benchmarked code sections are considered. The `#pragma omp parallel for` divides the particles equally among all threads. With the initializations being in the same form as the usages, i.e. done from the same threads, also the caches and memory accesses are optimized (see listing 2). This allows to obtain the almost perfect linear scaling. By then doing all the iterations of a single particle in a row, also the cache misses are reduced. In this way the code gets most likely computationally bound, especially for large M, which is why the parallel scaling behaves so well.

1.2 Sensitivity on Parameters

Figure 2 compares the different outcomes of the extreme case of just 1 iterations, 1000 particles and 1 or 500 threads.



(a) 1 thread

(b) 1 thread changed seed

(c) 500 threads

(d) 500 threads changed seed

Figure 2: Output histogram sensitivity on parameter changes

Note that in all cases, the random number generators are seeded with the thread id involved. In the 'changed seed' cases the seed is simply equal to the thread id, whereas the standard cases involve another random number and some arithmetic operations.

We can observe that the final histogram generated with 500 threads is less uniformly distributed than it is with just a single thread. This can be explained by the fact, that a one of the 500 threads just generates 2 numbers. Therefore the entropy is shifted away from the properly implemented

fmahlknecht@student.ethz.ch
19-945-351

random number generator into the initial seed implementation. This of course cannot guarantee anymore a uniformly distributed output, especially just after 1 iteration.

On the other hand, the single threaded implementation can use the same number generator for all the particles, and meets therefor the requirements of yielding a correctly distributed output.

Changing the way of seeding does not show any significant systematic effect.

2 Bug Hunting

```
1  //assume there are no OpenMP directives inside these two functions
2  void do_work(const float a, const float sum);
3  doublenew_value(int i);
4
5  void time_loop()
6  {
7    float t=0;
8    float sum=0;
9
10 #pragma omp parallel
11   {
12     for(int step = 0; step<100; step++) {
13 #pragma omp parallel for nowait
14       for(int i=1; i<n; i++){
15         b[i-1] = (a[i]+a[i-1])/2.;
16         c[i-1] += a[i];
17       }
18
19 #pragma omp for
20       for(int i=0; i<m; i++)
21         z[i] = sqrt(b[i]+c[i]);
22
23 #pragma omp for reduction(+:sum)
24       for(int i=0; i<m; i++)
25         sum=sum+z[i];
26
27 #pragma omp critical
28       {
29         do_work(t,sum);
30       }
31
32 #pragma omp single
33       {
34         t=new_value(step);
35       }
36
37     }
38   }
39 }
```

Listing 5: Bughunting Exercise 1

For the first exercise, given in listing 5, the following bugs can be spotted:

Florian Mahlknecht
fmahlknecht@student.ethz.ch
19-945-351

- Duplicate spawning of threads in line 10 and 13, the second should definitely be removed and turned into a `#pragma omp for`, i.e. without *nowait*, since that has no effect in this case
- the reduction for loop can be unified with the for loop before, calculating `z[i]` and reducing `sum` in just one loop.
- `m` and `n` are neither declared nor defined, but probably we can suppose that this is properly handled elsewhere.
- this holds for the critical and single section as well

```
1  void work(int i, int j);
2  void nesting(int n)
3  {
4    int i,j;
5  #pragma omp parallel
6    {
7  #pragma omp for
8      for(i=0; i<n; i++) {
9  #pragma omp parallel
10       {
11 #pragma omp for
12         for(j=0; j<n; j++) {
13           work(i,j);
14         }
15       }
16     }
17   }
18 }
```

Listing 6: Bughunting Exercise 2

In the second exercise, given in listing 6 the following issues arise:

- Duplicate spawning in line 6 and 10; line 10 an be removed completely.
- A `#pragma omp for collapse(2)` can be applied in line 8 afterwards. Of course we cannot know exactly what `work(i,j)` does, but guessing from the declaration, this should be the best performing optimization.

fmahlknecht@student.ethz.ch
19-945-351