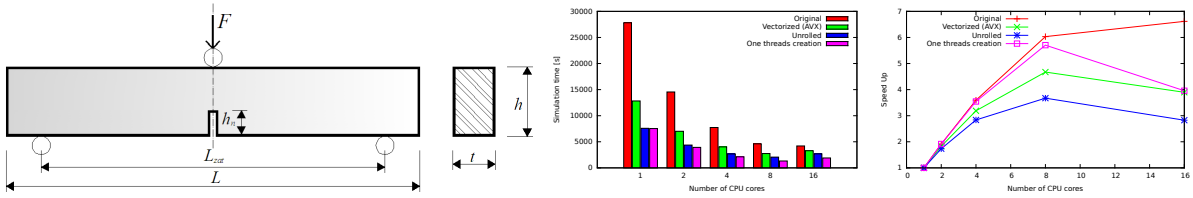
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2017
http://excel.fit.vutbr.cz

Optimization of Fracture Tests Simulation in Civil Engineering

Gabriel Bordovský*



Abstract

Precise simulations of many physical phenomena such as heat deposition, gravitation forces or molecular dynamics are very time-consuming. This paper describes a computer fracture test simulation of quasi-brittle materials and proposes techniques to reduce simulation time by effectively using given compute resources as CPUs and GPUs.

The existing code created by the Faculty of Civil Engineering was analyzed and the bottleneck of simulation was found. A code analysis by Alinea Map showed that 60% of the CPU work is done by a single line of code. Even the work is done by a single line, the code has to be refactored to get better performance. Vectorization, parallelization and loop unroll are used in this work. The GPU code was analyzed too. The main problem was found in the nonaligned access to the input data and in the need of exclusive writes of the output data.

The original source code can process 10 millions of iteration by one thread in 463 minutes. The CUDA realization processes the same amount of iterations 4.63 times faster (in 100 minutes). The optimized code described in this paper processes the same amount of iterations 3.85 times faster (in 120 minutes) by a single thread while 21 times faster (in 21 minutes) with the use of eight physical CPU cores.

Many comparisons of GPU and CPU implementations are often fogged by the fact that the CPU realization does not use resources effectively. Contrary, this work shows that some problems may be solved easier and potentially cheaper on a CPU. The cost of required hardware may be lower because the CPU solution waits on main memory. Because of that, processors with lower frequency are sufficient.

Keywords: Parallelization — Vectorization — CUDA — Fracture Test – Simulation

Supplementary Material: *N/A*

*xbordo04@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

[Motivation] When renovating historical buildings or bridges, some stones or bricks may need replacement. This replacement has to have the same visual and mechanic characteristics as the original material. In order to find the best replacement, some blocks of sandstone from the original building are taken a sample. This sample is then destroyed by the fracture test to investigate the material properties. Here, the computer simulation can make the whole process faster, cheaper and limit the need to extract large amount of the material from the original construction. However, there are

series of simulation with different parameters that have to be executed. This paper shows how the simulation can be accelerated on CPU with the use of multiple threads and vectorization.

[Problem definition] The fundamental problem of the existing implementation is a very low computational efficiency leading to very long simulation times. This is given by the representation of the analyzed sample in computer memory. The 3D domain covering the tested sample is divided into a 3D mesh of points connected by bonds together forming so called finite elements, or bricks, see Fig. 1. At every iteration, the force resultant inside the bricks is computed. If the force exceeds a predefined limit, the sample begin to break. Our goal is to redesign the underlying data structures and refactor the simulation code to offer high computational efficiency.

[Existing solutions] There are many different solutions of the fracture test simulation. These solution are mostly based on 2D models of the tested sample [1]. Limiting the model to two dimensions improves the simulation speed by a great deal, however, imposes many restriction on the model, e.g., the reaction forces have to be distributed linearly. The symmetry in the vector of load also provides a possibility to calculate only half of the model. However, not all models can be transformed into 2D without loss of precision. For example, the samples and the models used for our simulation are composed from quasi-brittle round material with a "V shaped" indentation. This does not allow the reduction to a 2D model.

[Contributions] In this work, two Intel® Xeon® Processor E5-2470 at 2.30 GHz were used on computer cluster Anselm from IT4Innovations. The proper use of processors vectorization, memory aligned accesses, and unrolled loops allowed to decrease the simulation time from seven hours to two hours. Furthermore, with use of eight processor cores on a single CPU, the execution time was lowered down to 21 minutes. The original GPU/CUDA solution handles the same amount of work in nearly 100 minutes.

The CPU version was found to be now memory bound. Up to 8 cores, the computational efficiency is about 70%. Distributing the work among more than 8 becomes inefficient due to cache coherence and NUMA (multi-socket) overhead. Also increasing the CPU frequency is not expected to yield better performance since the memory has already become saturated. Finally, the work on the CUDA implementation is still in progress, but multiple bricks sharing same point require atomic/exclusive access to memory and this leads to serialization of the process.

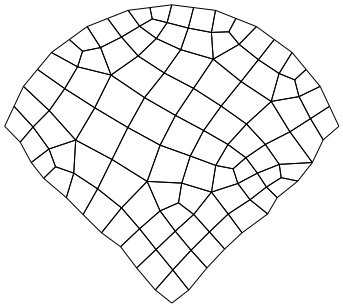


Figure 1. A Vertical slice through the V shaped indentation of used model with the mesh of finite elements.

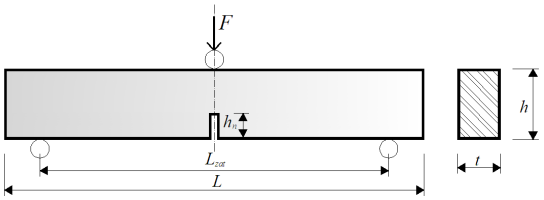


Figure 2. Example of a three point bending test with indentation.

2. Fracture Test

The fracture test is in general used for obtaining material characteristics. Builders in modern constructions have to stay safe from the point where constant load on the materials causes fractures and the construction collapse. This was an issue in the middle of 20th century [2] when buildings suddenly collapsed due to fatigue of material. Then the interest in fracture mechanics created a new field in science. One of the current interests in this domain is finding the right replacement for historical buildings. The material with different properties may behave differently and cause future aesthetic and structural damage.

The full name of this investigation is the fracture test by three point bending. As the name may suggest, a sample of material is put on two pads. Constant load is then applied in the middle between the pads until the material breaks in two.

A special form of the test is used in this work. The sample has an indentation on the side opposite to point of load, see Fig 2. The length of the indentation is called the crack mount opening. The change in this crack length may be observed. One of the results from this test it the load, commonly marked as P, by crack mount opening displacement, shortened as CMOD. A civil engineer may specify parameters as fracture toughness, energy or tensile strength of the sample from this P-CMOD diagram.

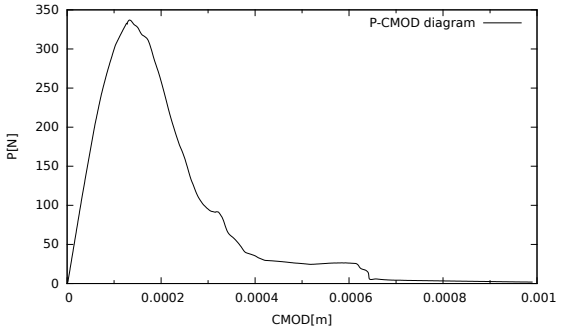


Figure 3. Example output of the test. P is force applied on pads. CMOD is Crack mount opening displacement from original position.

The diagram shape is given by the type of the material [3]. The material can return from elastic behaving to the original state. Fragile materials behave elastically, load is increasing with deformation, and then when maximal load is applied, it breaks, for example glass. Others, such as steel, behave elastically to the maximal load too. Then they start to behave plastically. Third is quasi-fragile type. It also behaves elastically in the beginning. Then micro-cracks are created in the material. In the point of maximal load, the main crack is selected from the micro-cracks and widened to the point of full separation.

The sandstone or steal-less concrete in the point of maximal load is currently analyzed at the Institute of Structural Mechanics at Brno University of Technology .

3. Fracture Test Simulation

Ing. Jan Bedaň from the Institute of Structural Mechanics developed a simulation tool for the fracture test. This simulation takes a model of analyzed material and assumed characteristics of this material. The result is the P-CMOD diagram. The tool has a limited use, because the time required for a single simulation is 463 minutes. This is mainly given by a single-threaded nature of the code. With the use of CUDA on an Nvidia Tesla C2050, it takes the reference simulation from the Institute 41 minutes to finish 4 millions of iteration. However, these 4 million iterations do not capture the whole P-CMOD diagram. Further in this work, the results of 10 millions of iteration will be used. It is safe to assume that the work in iteration is constant and therefore 10 million iterations would have taken circa 102 minutes.

This solution [4] takes Y-axis symmetry and represent only one half of the sample as the model. Instead of the other half, we may imagine immovable plain that is connected to the model via a cohesive function. This function provides additional reaction to the points

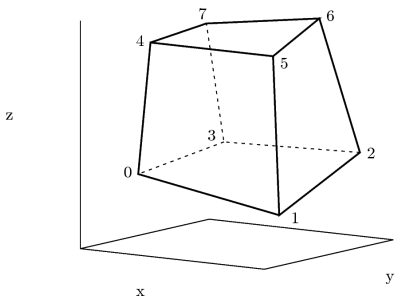


Figure 4. Example of the brick element.

in crack area. That slows detaching of the points from the plain. The points that are already detached from the plain are still influenced by the cohesive function and therefor they move slowly from the plain.

During the simulation, the point where the load is applied, is being moved in direction of the load. Based on the matrix of stiffness and the previous movement of the points, the reactions in bricks are computed. Next positions of all points are based on these reactions. The points creating the crack have to accumulate enough reactions to overcame cohesive function before they can move. The most critical place of the simulation is the computing of reactions. The inputs are the bricks that are composed from eight references to points, matrix of stiffness that is created for each brick, positions of points and original positions of points. Output is reaction of each point. Although, the term brick may suggest that the birck is a cube or cuboid, it can reach an arbitrary shape, see. Fig 4.

4. Analysis of the original solution

The first problem spotted is a huge number of iterations that has to be executed compared to a relative small number of bricks. The iteration has naturally be executed in a sequential way. In each iteration, reactions inside each brick are determined. There are implemented as nested for loops. For each brick, 576 (3 coordinate for 8 points to 3 coordinates for 8 points) loop iterations are done.

The second problem is that the points for each brick are not aligned in memory. This is a fundamental problem for CPU vectorization and GPU computing. Each brick is composed from references to eight points. This leads to ineffective load and store operations. It means, when a brick is assigned to one thread, some of the threads has to access the same memory location leading in racing condition. It would be ideal if each thread had its own location in main memory, where to read from and store to. Also, the threads/vector lines logically next to each other should load blocks from data from neighboring locations.

5. Implementation

We first focus on the CPU version in this work. The baseline of simulation time was set to 27 815 seconds for the reference single threaded code.

I would like to make a little side note here, in a case that someone would look for inspiration or solution for their own simulation here. It is bad idea to flush data in the main iteration process, if they are not required on the output during the iteration. In this case, every 10 000 iteration the data for P-CMOD diagram was printed into file and flushed. As the I/O operations are typically very slow, it is better to let the flush outside the loop and write all data at once.

The modern CPU has vectorization units [5] and registers that allows them to apply the same operation on multiple data elements at once. Such examples are the Streaming Single Instruction Multiple Data Extension (SSE) or more recent Advanced Vector Extension (AVX [6]). The number of processed elements is given by the capabilities of the extension used, the size of processed data type and size of the vectorization registers. The SSE uses 128 bit registers that may process four single precision floating point numbers. The version SSE2 was able to use those registers for four 32 bit integers, two double precision floating point numbers or others. The AVX has 256 bit wide registers and is capable to process eight single or four double precision floating point numbers.

The first question was if the AVX can be used. When the differences in the point position are computed, they are aligned in the memory [7]. This means that they may be simple loaded into AVX registers. Otherwise, they would have to be gathered from the different memory locations and lined up. The gathering would increased the overhead of vectorization and the speed up would be much lower. The matrix of stiffness is aligned too.

The original code has used double precision floating point numbers. When replaced with single precision ones, the result was computed 40 % faster but wrongly as may be seen in Figure 5. The reactions are so small that the change in position is not captured well when added to original positions stored in single precision floating point variables.

Moreover, the GNU Compilator assumed that overhead in double precision version is too big to apply vectorization meaningfully. When the vectorization was forced on, the simulation time was deceased from 27 815 seconds down to 12 827 seconds, yielding a speedup of 2.16.

Following optimizations was focused at thread level parallelization. Two processors, each with eight

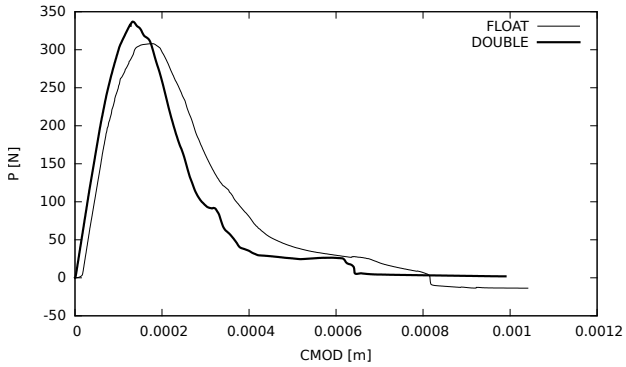


Figure 5. Comparison of result using double precision (correct-width) and single precision (incorrect-thin).

cores, was used in the tests. The work was distributed by using **#pragma omp parallel** region on the loop for bricks. In Fig. 7, you can see the scalability of developed solutions. This figure tells us how much faster the solution is compared to sequential version. In ideal case, the speed-up is same as the number of processor cores used. However, that is nearly impossible due to parallelization overhead. When the number of processors goes up from 8 to 16, two physical processors are used. The scaling on two processors is inefficient due to large amount of communication between these two processors. It could be removed by better decomposition over processors. For example, the Message Passing Interface (MPI) would allowed to distribute data to physical processors. However, finding the right way to split the work is not an easy task. In this simulation, one brick may use points at different memory locations and some threads will need to access memory on the other processor.

On the other hand, 8 cores on single processor presented further in text is already so fast that it has to wait on main memory. Because of that the distribution on more physical processors was not analyzed in depth.

The following phase in optimization was to unroll the critical loop [8]. As for now, there was a loop for each coordinate of the eight points creating the brick. When unrolled three times, all coordinates of the current point are calculated at once. This provides a lower overhead for vectorization. The unrolled loop iterates only 8 times instead of 24, but does three times more work in each iteration. Thanks to that, the processor may prepare data for next iteration during previous one (using memory pre-fetch, out-of-order execution, etc.). The simulation time achieved by applying this optimization stopped at 7 584 seconds. That is 1.69x faster than vectorized version and 3.66x faster than original code.

The code was then analyzed by the Intel VTune

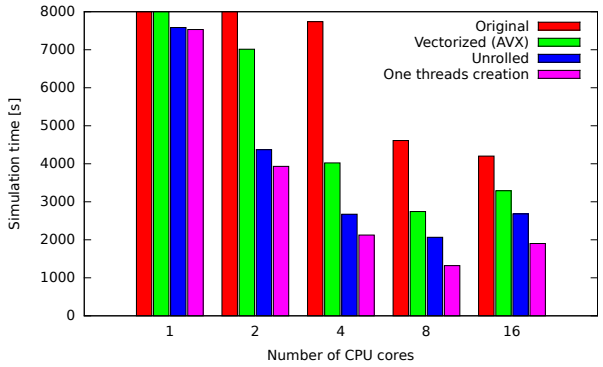


Figure 6. Execution times reached on different number of physical CPU cores. Y range from 8 000 seconds up is cut off. Precise times may be found in Table 1

tool. This tool showed that overhead of the **#pragma omp parallel for** begun to be the slowest part of the code. The process of creating a team of threads for every single brick slowed the computation significantly. In reaction to this analysis, the **#pragma omp parallel** was moved above the whole iteration process (ten million iterations). This way, all the threads are created once. The work is then distributed by **#pragma omp for**. This last code modification decreases the simulation time of the parallel version down to 1 320s.

6. Results

The time required for a single simulation executed by the sequential version went down from 27 815 to 7 531 seconds. That is 3.69 times faster with use of the same hardware. Other times may be seen in Table 1. If all 8 cores on processor are used, the final simulation time is 1 320 seconds and the solution is faster 21 times then original sequence program.

Table 1. Table of simulation times in different states of optimization. Each line contains optimizations from previous one.

Number of cores	1	2	4	8	16
Original	27 815s	14 545s	7 738s	4 611s	4 203s
Vectorized (AVX)	12 827s	7 014s	4 023s	2 744s	3 291s
Unrolled	7 584s	4 371s	2 673s	2 065s	2 681s
One threads creation	7 531s	3 931s	2 124s	1 320s	1 903s

The scaling of different versions of the code is shown in In Fig 7. For 16 cores, the simulation slows down due to the transfers between cache and memories of two CPUs. The original solution scales a bit better than others because a long time of processing allows to overleap the communication better. The final solution using vectorization, loop unroll and single thread creation is the second best in terms of scaling.

7. Conclusions

[Paper Summary] This paper analyzed an existing solution of computer simulation of fracture test for quasi-brittle materials and presented techniques to optimize the execution. These techniques cover vectorization, parallelization, loop unrolling and correct placing of parallel sections to reduce the overhead. The main problems with CUDA realization were presented. The main disadvantage for computing this problem on graphic card are the unaligned reads and exclusive writes into the memory.

[Highlights of Results] The original solution took 27 815 seconds (7.7 hours) on a Intel Xeon E5-2470 processor. The fully optimized solution now takes 7 531 seconds (2.1 hours) when executed on a single core. When all 8 physical cores of CPU are used, the simulation time is 1 320 seconds, almost 21 times faster on the same hardware. This solution is also 5 times faster than the CUDA solution currently used at the Institute of Structural Mechanics Brno University of Technology. The further performance improvements are limited by the latency of main memory.

[Paper Contributions] This work presented how different the execution time of program on the same processor may be. We show how much the code can be accelerated by careful optimization of the algorithm to the proper hardware.

[Future Work] In the near future, the scalability of the final solution will be analyzed on computer cluster Salomon. This cluster allows the programmer to specify frequency of processor. It should more exactly determinate the frequency with the processor becomes limited by the memory and this reduce the power consumption of the processor. Further experiments with the CUDA solution based on this work are planned as they are requested by the Institute of Structural Mechanics.

Acknowledgements

I would like to thank my supervisor Ing. Jiří Jaroš, Ph.D. for his help with my master's thesis.

I would also like to thank Ing. Jan Bedaň for his patience with this work and his expertise with original solution.

This work was supported by The Ministry of Education, Youth and Sports from the Large Infrastructures for Research, Experimental Development and Innovations project „IT4Innovations National Supercomputing Center – LM2015070“.

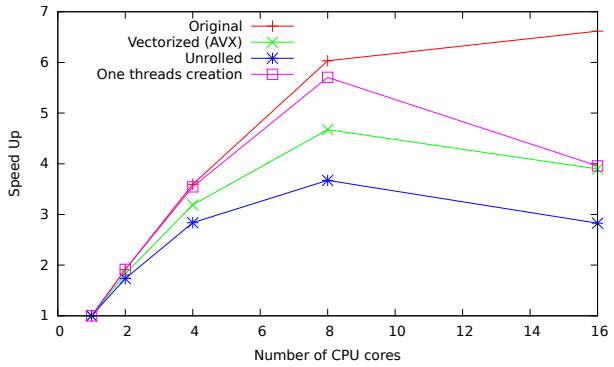


Figure 7. Scalability of the solution. Ideal scalability is linear and as close to function f(x)=x as possible.

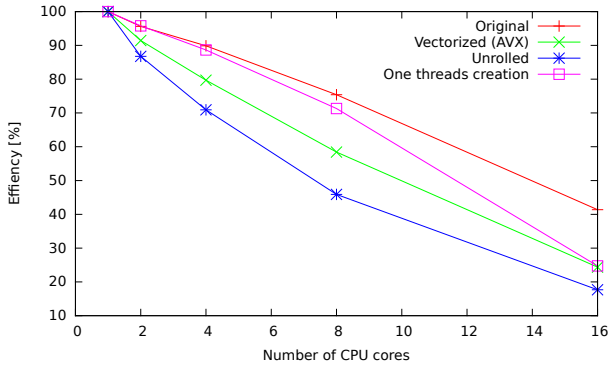


Figure 8. Efficiency of use multiple cores for produced solutions. As the overhead rises with more cores the efficiency goes down.

References

[1] Tomáš Pail, Petr Frantík, and Michal Štafa. Specializovaný mkp model lomu trámce. *International Scientific Conference on Structural and Physical Aspects of Civil Engineering*, pages 1–6, Slovenská republika: 2010. ISBN: 978-80-89284-68-9.

[2] A.A. Griffith. The phenomena of rupture and flow in solids. *Philosophical Transactionsm, Series A, Vol. 221*, pages 163–198, 1920.

[3] Drahomír Novák and Luděk Brdečko. *Pružnost a pevnost MO1 - Základní pojmy a předpoklady*. FAST VUT v Brně, 2004.

[4] Jan Bedáň. Simulace lomových procesů na superpočítači. *Pojednání k tématu disertační práce, FAST VUT v Brně*, 2015.

[5] Anoop Madhusoodhanan Prabha and Bob Chesebrough. Performance essentials using openmp 4.0 vectorization with c/c ++. https://software.intel.com/sites/default/files/managed/37/df/OpenMP4-performance-essentials-using-vectorization.pdf.

[6] Chris Lomont. Introduction to intel advanced vector extensions. https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf.

[7] Rakesh Krishnaiyer. Data alignment to assist vectorization. https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization.

[8] Maria J. Garzaran. Loop transformations. http://users.ices.utexas.edu/~lenharth/cs378/fall14/8%20-%20LoopOptimizations.pdf.

```
for (int iteration = 0; iteration < MAX_ITERATION; iteration++) {
    resetReactions();
        // For each brick in the model
    for (int i = 0; i < numberOfBricks; i++) {
        double deformations[] = { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
            // Get movement from original position for each point
        for (int j = 0; j < 8; j++)
        {
            deformations[j * 3] = x[indexesOfNodes[i * 8 + j]] - xStart[indexesOfNodes[i * 8 + j]];
            deformations[j * 3 + 1] = y[indexesOfNodes[i * 8 + j]] - yStart[indexesOfNodes[i * 8 + j]];
            deformations[j * 3 + 2] = z[indexesOfNodes[i * 8 + j]] - zStart[indexesOfNodes[i * 8 + j]];
        }

        int indexOfMassPoint = 0;
        int helpIndex = 0;
            // For each coordinate of each point (3x8)
        for (int j = 0; j < 24; j++)
        {
            if (j % 3 == 0 && j > 0)
            {
                helpIndex = 0;
                indexOfMassPoint++;
            }
                // Get reaction on the other coordinates in current brick
            double reaction = 0;
            for (int k = 0; k < 24; k++)
            {
                reaction += matrixOfStiffness[i * 24 * 24 + j * 24 + k] * deformations[k];
            }
                // Store reaction for correct coordinate
            if (helpIndex == 0) rx[indexesOfNodes[i * 8 + indexOfMassPoint]] -= reaction;
            else if (helpIndex == 1) ry[indexesOfNodes[i * 8 + indexOfMassPoint]] -= reaction;
            else if (helpIndex == 2) rz[indexesOfNodes[i * 8 + indexOfMassPoint]] -= reaction;

            helpIndex++;
        }
    }
    printResults(iteration);
    getNewPointPositions();
    applyCohesiveFunction();
}
```

Code sample 1. The original code computing the simulation. Full code computing reaction is displayed.

```
1       // One time creating threads
2     #pragma omp parallel
3     for (int iteration = 0; iteration < MAX_ITERATION; iteration++) {
4       resetReactions();
5         // Each thread takes same number of different bricks
6       #pragma omp for schedule(static)
7       for (int i = 0; i < numberOfBricks; i++) {
8         double valX = 0;
9         double valY = 0;
10        double valZ = 0;
11
12        double deformations[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
13
14        for (int j = 0; j < 8; j++) {
15          deformations[j * 3] = x[indexesOfNodes[i * 8 + j]] - xStart[indexesOfNodes[i * 8 + j]];
16          deformations[j * 3 + 1] = y[indexesOfNodes[i * 8 + j]] - yStart[indexesOfNodes[i * 8 + j]];
17          deformations[j * 3 + 2] = z[indexesOfNodes[i * 8 + j]] - zStart[indexesOfNodes[i * 8 + j]];
18        }
19          // Help compiler to assume matrix aligned for SIMD
20        double* stiffness = this->matrixOfStiffness;
21        __builtin_assume_aligned(stiffness, 64);
22
23        int indexOfMassPoint = 0;
24          // Go by point instead of by coordinate (3x unrolled)
25        for (int j = 0; j < 24; j+=3) {
26          double reactionX = 0;
27          double reactionY = 0;
28          double reactionZ = 0;
29
30          int baseX = i * 24 * 24 + j * 24;
31          int baseY = i * 24 * 24 + (j+1) * 24;
32          int baseZ = i * 24 * 24 + (j+2) * 24;
33
34           #pragma omp simd reduction(+: reactionX, reactionY, reactionZ)
35          for (int k = 0; k < 24; k++) {
36            valX = stiffness[baseX + k] * deformations[k];
37            valY = stiffness[baseY + k] * deformations[k];
38            valZ = stiffness[baseZ + k] * deformations[k];
39            reactionX += valX;
40            reactionY += valY;
41            reactionZ += valZ;
42          }
43
44          rx[indexesOfNodes[i * 8 + indexOfMassPoint]] -= reactionX;
45          ry[indexesOfNodes[i * 8 + indexOfMassPoint]] -= reactionY;
46          rz[indexesOfNodes[i * 8 + indexOfMassPoint]] -= reactionZ;
47          indexOfMassPoint++;
48        }
49      }
50      printResults(iteration);
51      getNewPointPositions();
52      applyCohesiveFunction();
53    }
```

Code sample 2. The modified code. The **#pragma omp parallel** , **#pragma omp simd** and **#pragma omp for** are correctly placed in the code and the code is unrolled.