Evolutionary Mechanics

Tobias Jacob, Raffaele Guillera, Ali Muddasar

December 6, 2020

Abstract

We developed an application that is able to develop mechanical structures using an evolutionary algorithm. This approach can be scaled efficiently across many different nodes.

1 Method

Our project is divided in two sections and corresponding layers of parallelism. The first one is solving the mechanical equations to check if a mechanical structure can withstand a force. Figure 1 shows the result of such a simulation. The shape of a figure is approximated through squares.

The second stage is the evolutionary algorithm. The best 10% survive in each round. The structures mutate and are simulated again. Even though we put a lot of effort into parallelizing the equation solver, it turned out, that in the end it is faster to parallelize just the evolutionary algorithm instead of parallelizing the evolutionary algorithm and the equation solver. This is, because the core count is usually lower than the generation size, which means that each core can process its own structure.

1.1 Speeding up the equation solver

`OpenMP` is used for parallelizing the equation solver. The reason for that is the tight coupeling between the data. `gprof` revealed that the equation solver spends most of it's time in the sparse matrix multiplication, however the whole equation solver class works parallelized.

- The equation setup process adds the local element striffness matrix into the global equation system. Depending on the mechanical structure, the position of different planes appears rather random. Therefore, a lock is needed to prevent a race condition. Because the matrix is sparse and each thread works on its own plane, that typically are not directly connected, it is unprobable that two threads operate on the same equation row at the same time. A global lock would introduce an unnecessary penalty, therefore each row uses it's own lock.

- All linear algebra operations work in parallel. There are two types of these operations. In Addition, scalar multiplication, matrix multiplication or assigning a constant value,

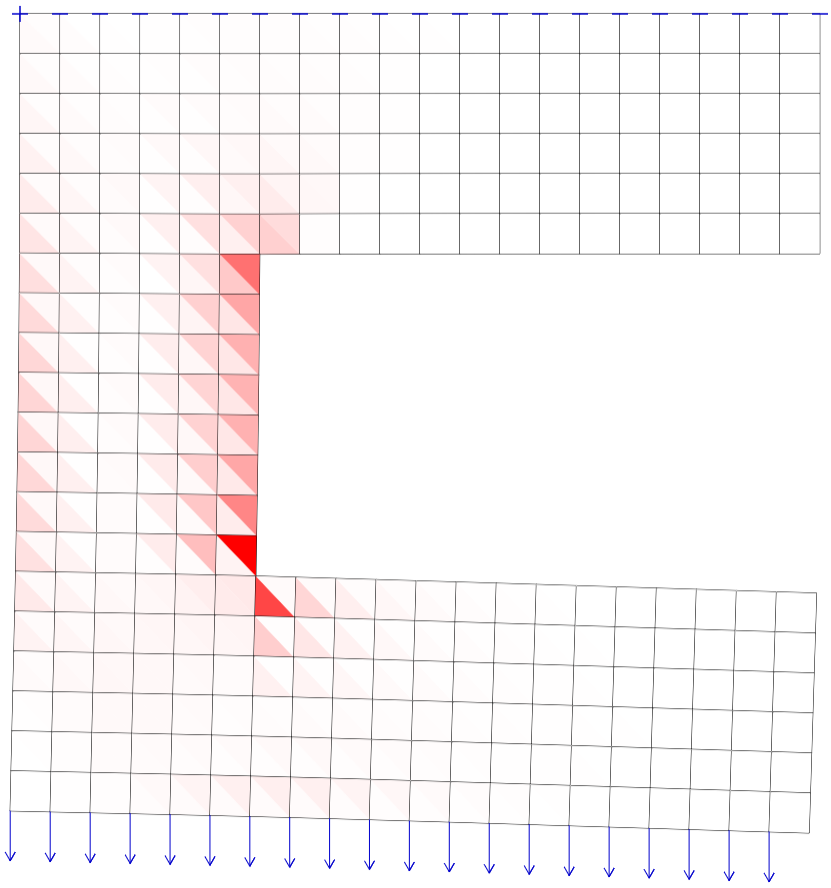

Figure 1: Result of a mechanical simulation

each thread processes its own chunk of rows. This means, that these operations to not require an implicit or explicit barrier. If for example, a vector addtion follows a scalar multiplication, it is fine if the first thread begins with the scalar multiplication before the second thread has finished the vector additon, since each of them operates on its own set of rows.

However, in the conjugate gradient method there are also operations like the scalar product or the norm of the vector. These operations require a reduction and have an implicit barrier.

- In the beginning, a unique index has to be assigned to each Plane and corner. This is not parallelizable, as the total number of planes and corners is unkown and does not follow a predictable pattern.

The equation solver has theoretically strong scalability, as the dominant part in the computational complexity, the sparse matrix multiplication, can be split accodring to rows.

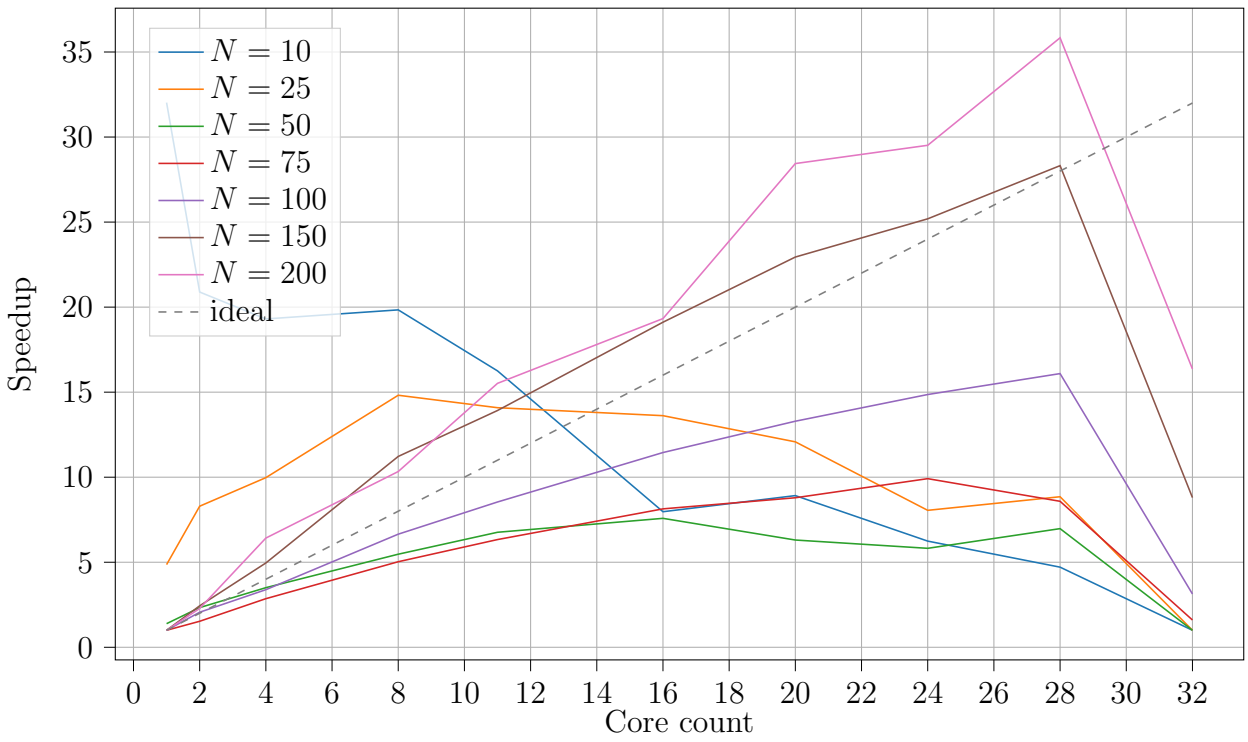

Figure 2: Execution time

In practice however, there is a significant overhead in creating the threads. This means, that the speedup becomes only notable for problem sizes above 50.

Also, the evolutionary algorithm easily solves several thousand equation systems per second, and for the problem sizes that we evaluated the algorithm on, the thread creation becomes the significant overhead. So while the program is still capable of utilizing multiple cores for solving the equation, we rather decided to let each core process one or multiple structures per generation.

Nevertheless, the Equation Solver is a powerful solver, that scales well with core-count. It is able to solve a mechanical structure that consists of 80600 equations in 2.732 s using 28 cores on bridges. It also showcases OpenMP and locks well, we therefore decided to leave the code there. The speeup is shown in figure 2.

1.2 Speeding up the evolutionary algorithm

Evolutionary algoirthms are very parallel by nature. Each organism can be evaluated independent. Then they have to be sorted, and the best 10% get redistributed. A MPI Message for an evaluated board consisits of

- 1 float for the score
- $R \times C$ bytes, each containing the boolean value of the field.

Algorithm 1 Evolute on node

```
Initialize all local fields fully set
for all epochs do
    Mutate the fields
    Evaluate the fields
    Gather all fields into the master
    Sort the fields on the master
    Broadcast the best 10% to everyone
    Replace local fields
end for
```

We typically simulate grid sizes of 20, meaning the message has a size of 404 bytes. Since all processes will be initialized with the same field size it has not to be stored in the message. Our evolution works like shown in algorith 1. The result of the finished evolution is shown on figure 3.

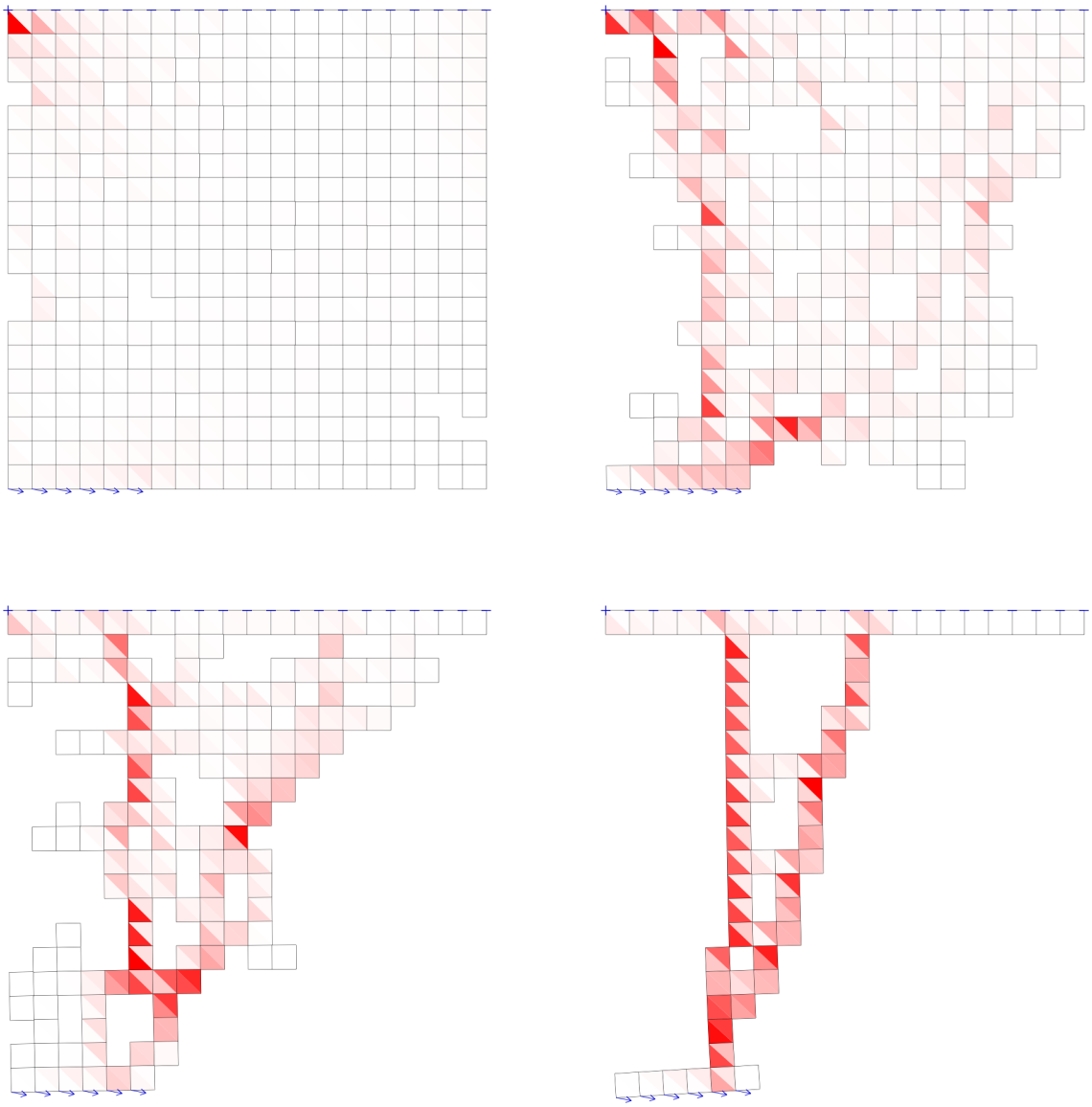

Figure 3: Result of a evoultion, using 20×20 gird, 100 organisms per epoch, 1000 epochs and a alterations decay of 0.995. Note, how the structure gets wider on the top to deal with the increased bending stress.