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. We used `OpenMP` for this part.

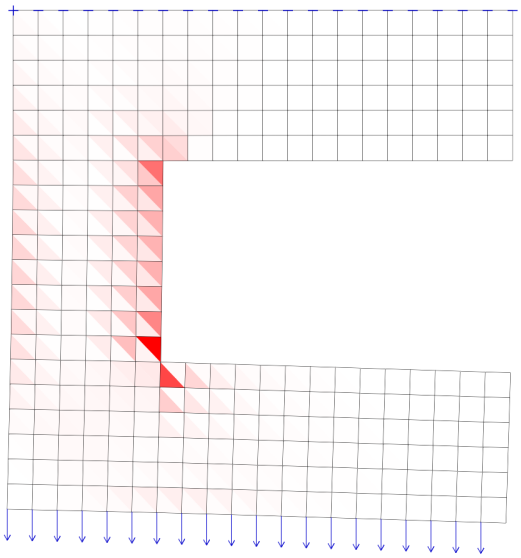

Figure 1: Result of a mechanical simulation

The second stage is the evolutionary algorithm. The best 10% survive in each round. The structures mutate and are simulated again. We used `MPI` for this part.

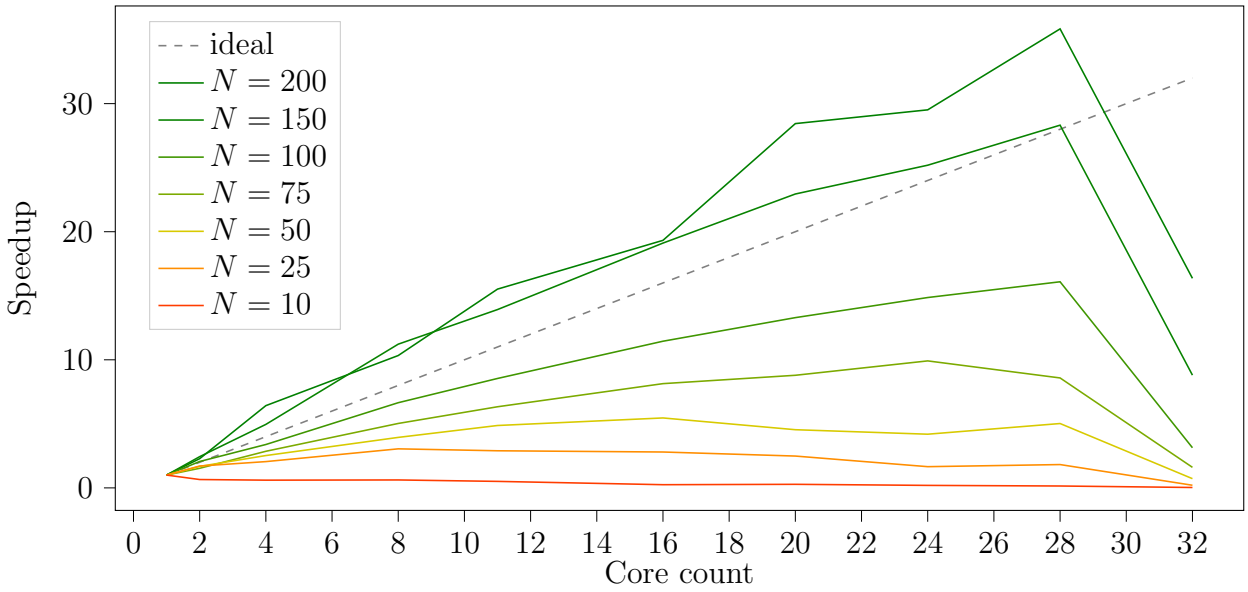

Figure 2: Speedup of the equation solver for different proplem sizes N.

	$N = 10$	$N = 25$	$N = 50$	$N = 75$	$N = 100$	$N = 150$	$N = 200$
Equations	230	1325	5150	11475	20300	45450	80600
$C = 1$	3	43	311	1153	3306	19869	97932
$C = 2$	4	25	185	754	1604	8149	42934
$C = 4$	5	21	123	403	976	4008	15251
$C = 8$	5	14	79	229	497	1771	9477
$C = 11$	6	15	64	182	387	1427	6311
$C = 16$	11	15	57	142	289	1040	5067
$C = 20$	10	17	69	131	249	866	3443
$C = 24$	14	26	74	116	222	789	3318
$C = 28$	19	24	62	134	205	702	2733
$C = 32$	91	209	432	718	1055	2257	5986

Table 1: Execution time for the equation solver for different field sizes N and cores C

1.1 Speeding up the equation solver

The reason for usign `OpenMP` is the tight coupeling between the data in the sparse matrix multiplication. `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.

- **Threads are spawned at the highest level** of the `PerformanceEvaluator` class. Subsequent calls to the equation setup, equation solver or linear algrabra operations will not spawn new threads. They require the threads to be set up already, and process only their their chunk using the `omp for` directive. All for loops are executed in chunks to **prevent false sharing**.

- 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 of the **equation solver** work in parallel. There are two types of these operations. For addition, scalar multiplication, matrix multiplication or assigning a constant value, each thread processes its **own chunk of rows**. These operations do 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 addition, 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 time complexity of the sequential equation solver is dominated by the sparse matrix multiplication, having a complexity of

$$O_{seqSolve}(N) = O(N^3) \tag{1}$$

as explained in the progress report. The sparse matrix multiplication is fully parallel. The indexing of the equation requires $O(N^2)$. Leaving the overhead for thread creation aside, the runtime complexity is

$$O_{parSolve}(N, C) = O\left(\frac{N^3}{C}\right) + O(N^2) \tag{2}$$

for a sufficiently large N. The efficency is

$$E = \frac{O_{seqSolve}}{O_{parSolve}C} = O\left(\frac{N^3}{N^3 + C^2}\right) \tag{3}$$

The equation solver has weak scalability, because efficency remains only at the same level if $N \propto C$. In practice however, there is a significant overhead in creating the threads. The speedup becomes only notable for problem sizes $N > 100$.

The Equation Solver is still a powerful solver. It is able to solve a mechanical structure with 80600 equations in 2.732 s using 28 cores on bridges. It also showcases OpenMP and locks well. The speedup of the execution time of table 2 are shown in figure 1.

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. We do the sorting on a node 0. `Gather` is used in that process. The best 10% get redistributed using `Bcast`. This is visualized in algorithm 1.

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.

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. The message size grows with

$$O_{msgSize} = O(N^2) \tag{4}$$

The runtime complexity of the single thread evolutionary algorithm is

$$O_{evolSeq}(N, G, A) = O_{seqSolve} \cdot O(GA) \tag{5}$$

with generations size G and A epochs. The parallel version has

$$O_{evolPar}(N, C, G, A) = O\left(\frac{N^3 GA}{C} + N^2 GA\right) \tag{6}$$

with $O(\frac{N^3 GA}{C})$ being the work for solving the equation system per core and $O(N^2 GA)$ being the communcation cost for sending $O(G)$ bytes of data over the network each epoch A. The efficency is

$$E = O\left(\frac{N^3 + N^2}{N^3 + CN^2}\right) \tag{7}$$

so this algorithm has also a weak scalability. Table 2 shows the execution times of the program. They are visualized in figure 3. The algorihm scales well for small core counts $C < 112$. Each generation has 112 organisims, so up to this part, the evolution scales very well. For $C > 112$, the mechanical solver starts to parallelize. This gives a speedup only if the problem size $N > 100$ is sufficient. The actual execution time starts to reach into hours for that problem size, but this is a typical for a evolutionary algorithm. The result of the finished evolution is shown on figure 4.

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

Cores	7	14	28	56	112	224
Cores per Task	1	1	1	1	1	2
Tasks	7	14	28	56	112	112
Nodes	1	1	1	2	4	8
N = 10	33.667	18.519	9,399	5.432	2.950	4.256
N = 20	296.997	164.875	91.958	43.114	25.199	29.429
N = 40	(2029.938)	(1343.408)	(631.448)	335.852	200.529	164.243
N = 80	-	-	-	-	(3729.680)	2163.535

Table 2: Execution time in seconds for the evoultion for different field sizes N and cores C. The generation size is set fixed to 112 and 1000 epochs are simulated. Values in brackets are extrapolated from a run with less epochs.

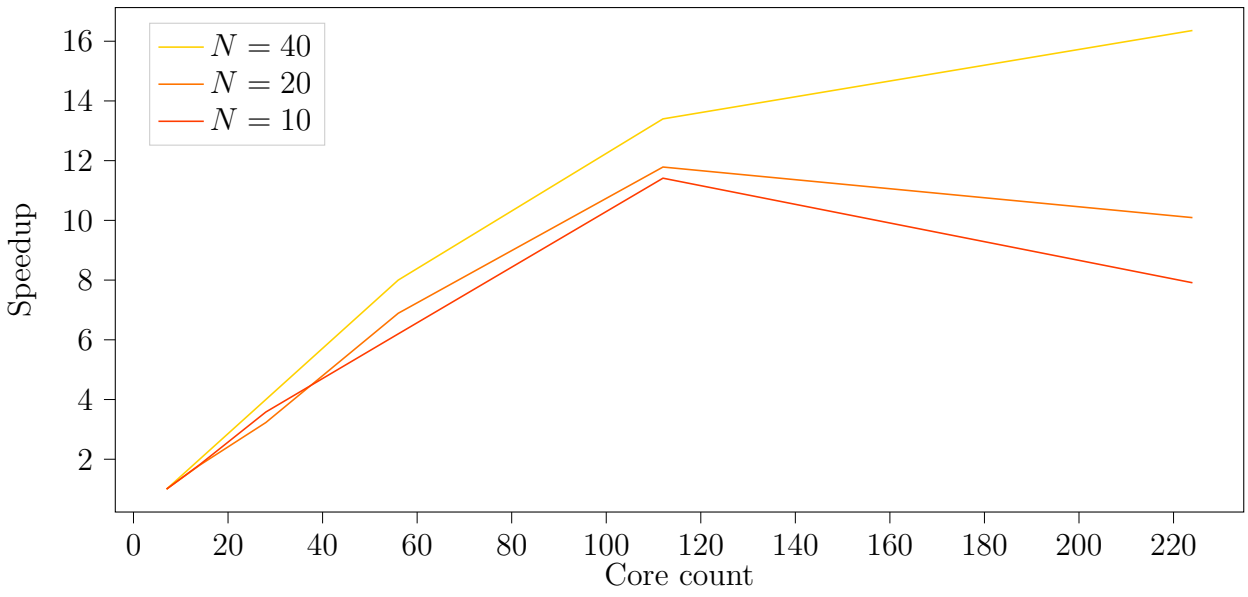

Figure 3: Speedup of the equation solver for different proplem sizes N.

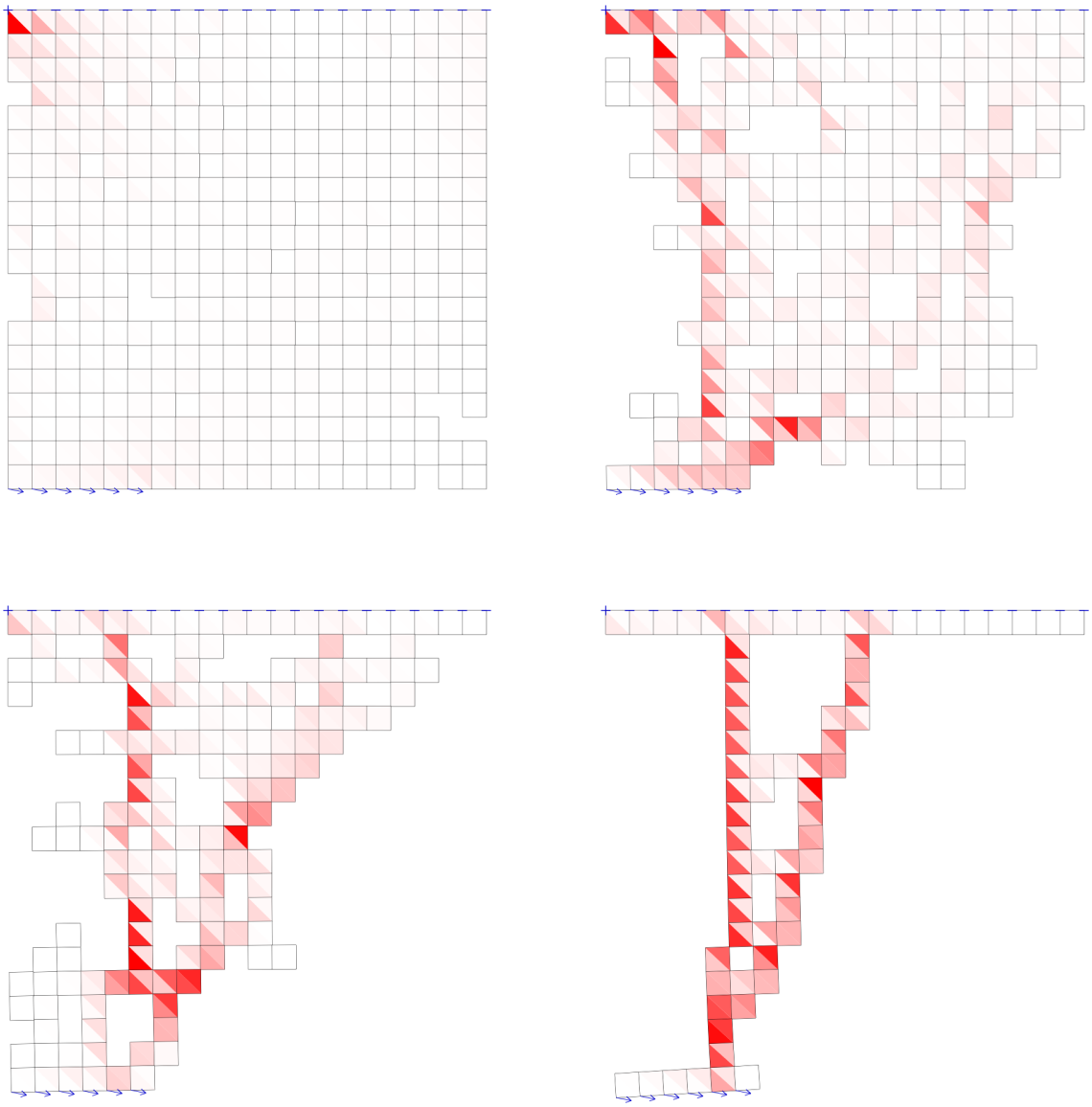

Figure 4: Result of a evoultion, using 20×20 gird, 100 organisms per epoch, 1000 epochs and a alterations decay of 0.995. Note, how the structure gets wider on the top to deal with the increased bending stress.