How to improve performance of Abaqus/Explicit models

Bastian Näser[1], Simon Mössner[1]

[1] BMW AG, 80788 München, Germany

***Abstract**: For a cost and time efficient virtual development process three main objectives can be identified: Minimize the turnaround time, maximize the throughput, and minimize license and hardware costs. Beside the fact, that these are contrary objectives, optimizing the model to reduce the runtime helps for every objective.*

***Keywords**: Abaqus/Explicit, Crash, Performance*

1. Introduction

Numerical simulation in general significantly helps to reduce the cost and turnaround time of a development process. In case of numerical crash simulation, the cost of a hardware test is between 1,000 to 10,000 times higher than the cost of a simulation. The cost for hardware crash tests increases, if tests are carried out at the early phase of the design process. At the end of the design process, hardware tests are comparatively inexpensive, but still significantly more expensive than numerical simulations.

The throughput of numerical crash simulations is between 1,500 and 2,000 times higher than hardware tests, depending on capacity of the hardware crash equipment and the HPC (High Performance Computing) cluster for the numerical simulation.

Independent of the fact, that a virtual design process is much more time and cost efficient than a purely hardware design process, optimizing these processes helps to reduce costs and to increase productivity.

To characterize the performance of a simulation process two quantities can be used: *Turnaround time* and *Throughput*.

1.1 Turnaround time

The turnaround time is the time between submitting a job and the final delivery of the results. This includes transfer times of the input date and the results files, queueing times, solving and automatic post processing. This is the most important quantity for the users (engineers). Depending on the simulation task, the overall process should be faster than 7 hours in order to carry out two jobs per day or faster than 16 hours to ensure a simulation overnight.

1.2 Throughput

Throughput describes the number of jobs processed in a given time frame. This is less important for the engineers, but a key indicator for the IT department, since this quantity determines the size of the HPC cluster and the license pool.

1.3 Conflict of interests

Both objectives are of opposing nature. Reducing the turnaround time (e.g. increasing the number of cores per job) reduces the throughput and increasing the throughput (by decreasing the number of cores per job) prolongs the turnaround time.

Defining a perfect core number per job or job type is not an easy task. Two facts further increase the complexity of this task.

1.3.1 Queueing time

The queueing time is an essential part of the turnaround time. Increasing the core number to reduce the turnaround time affects the queueing time in a negative manner due to a reduce throughput of the HPC cluster. Moreover, this effect strongly depends on the load of the HPC cluster, which varies over time. It is very hard to predict the impact on queuing time in advanced, thus there will always be a tradeoff in selecting the number of cores.

1.3.2 License cost

The second source of complexity are the license costs. Depending on the license cost, the license concept and the license usage a local cost minimum for core number per job can be found. This depends on the ratio between license cost and HPC hardware cost, and the nonlinear license cost scheme utilized by Abaqus.

2. Optimizing performance of crash simulation at BMW

Optimizing the simulation models to decrease the run time helps to increase the throughput and to reduce the turnaround time. Moreover, for special applications the focus can be shifted between turnaround and throughput. Three examples will be presented within this paper:

1. DOE like applications
    a. Reducing the number of cores per job and running more than one job per server.
    b. Identifying an optimal physical simulation time to avoid simulating unnecessary periods.
2. Optimizing crash models to run faster.

3. DOE like applications – Parallel execution on single server

DOE and DOE-like applications are widely used to observe results for varying ranges of input parameters. As an example, this work focuses on pedestrian protection. In contrast to occupant

safety simulations, the initial position of a pedestrian is unknown. To ensure an all-encompassing protection different positions have to be investigated.

One example is the Head Impact (HIP) simulation (see Figure 1). Depending on the size of the hood up to 400 impact points have to be investigated.

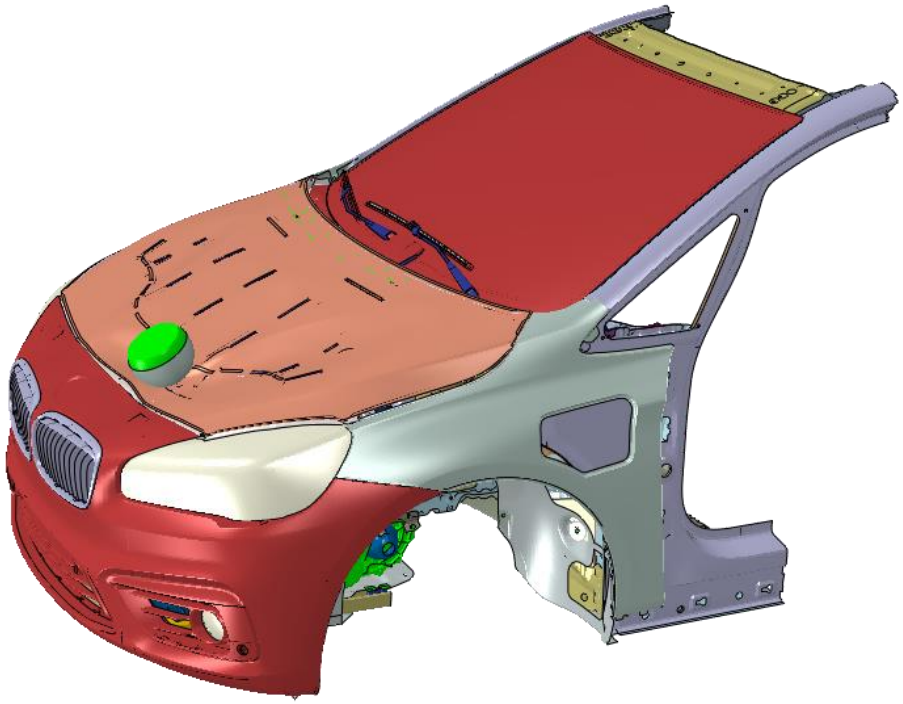

Figure 1. Hood deformation due to impact from a head impactor on a BMW 2 Series Active Tourer PHEV (12ms after impact).

For an engineer working on this kind of simulations, the turnaround time of the whole set of simulations is more important than the turnaround time of a single simulation. Thus, there is a shift of the focus from turnaround time to throughput. A straight forward approach is to reduce the number of cores used for these simulation jobs.

Only the front end of the vehicle is modeled, thus the simulation models are small compared to full vehicle crash models. In addition, the number of cores per server has increased in recent years. This leads to an increasingly inefficient usage of today's available compute servers for these pedestrian protection simulations.

To increase the throughput and address the mismatch of model size and computer size, more jobs per server have to run in parallel. Due to the imperfect scaling of run time with the number of cores, running 4 jobs with 5 cores each in parallel reduces the used core-hours to 72% on a 20

Core server compared to running 4 jobs with 20 Cores one after another. 4 jobs in parallel is a sweet spot for this job and this HPC hardware (see Figure 2).

The queueing system is taking care, that more jobs are distributed to a single server. The job control has to ensure, that the configuration is modified to allow non-exclusive usage of the hardware.

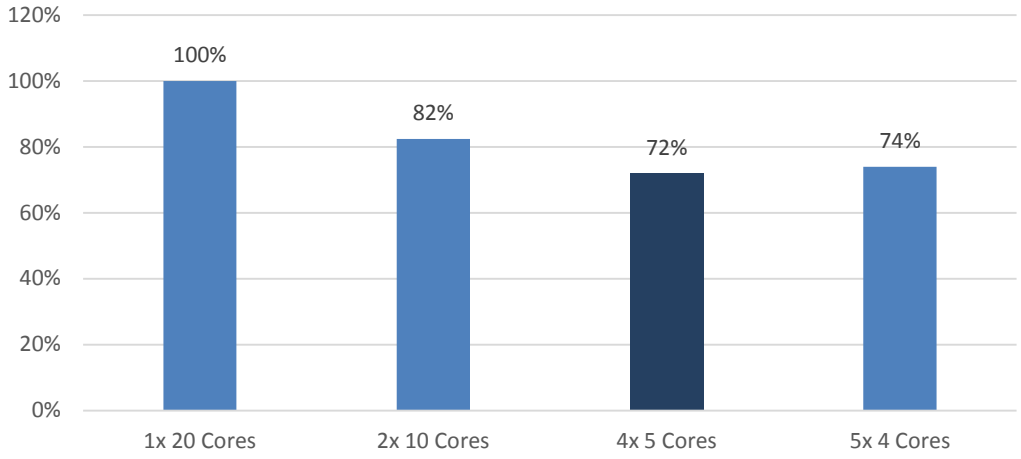

Figure 2. Savings from running parallel jobs on a 20 core server compared to a single job.

4. DOE like applications – Adaptive job termination

In the head impact workflow, the initial position of the impactor is parametrized, the remaining input file is identical for all impact positions. Even the physical simulation time, which has to consider the worst case, remains constant for all runs. However, the necessary simulation time varies depending on the impact position.

To evaluate the

$$HIC = \left\{ \left[\frac{1}{t_2 - t_1} \int_{t_1}^{t_2} a(t)dt \right]^{2.5} (t_2 - t_1) \right\}_{max}$$

(Head Injury Criterion), the highest average acceleration in the so-called HIC windows has to be computed. Acceleration of the impactor can mainly be observed during the impact phase, the rebound phase has no significant acceleration, but a contribution to the HIC value. Thus, the simulation has to be continued after detecting the rebound phase to compute sufficient data to evaluate HIC.

The duration of the impact phase depends on the impact position. In the middle of the hood, the stiffness is less compared to the edges or corners of the hood. Thus, the intrusion is much higher and the rebound phase starts later than on the edges or corners.

The necessary simulation time is hard to compute in advanced. Moreover, it is necessary to have a safety margin to compute the HIC for all impact positions. All these aspects result in a workflow, which computes plenty of unnecessary rebound time.

To overcome this waste of computational resources and to accelerate the development process, an adaptive job termination is used. This is implemented with the user subroutine `VUAMP`, which can be used to utilize user defined amplitudes. This user subroutine can monitor selected output quantities (sensors) and respond on the behavior of the model. One possible reaction is to trigger the end of the simulation.

```
      SUBROUTINE VUAMP(
     *     ampName, time, ampValueOld, dt, nprops, props, nSvars,
     *     svars, lFlagsInfo, nSensor, sensorValues, sensorNames,
     *     jSensorLookUpTable,
     *     AmpValueNew,
     *     lFlagsDefine,
     *     AmpDerivative, AmpSecDerivative, AmpIncIntegral)
      INCLUDE 'VABA_PARAM.INC'
     ..................................
     C parameter (iComputeDeriv     = 1,
     *           iComputeSecDeriv  = 2,
     *           iComputeInteg     = 3,
     *           iStopAnalysis     = 4,
     *           iConcludeStep     = 5,
     *           nFlagsDefine      = 5)
      ..................................
```

In the implementation we use at BMW, the acceleration and the relative vertical position of the impactor are monitored. If the logic implemented in the user subroutine assesses that the simulation can be terminated, the `iConcludeStep` flag will be set, thus the simulation terminates.

In Figure 3 the simulation time for a hood scan with 118 jobs are shown. The wave shaped variation of the simulation time can clearly by seen. Every wave corresponds to a scan line along the hood. By default, we use a simulation time of 30ms without adaptive job termination. In this example, the shortest simulation time was 12.6ms (Position A), and the longest 26.0ms (Position B). Compared to the default time of 30ms, the turnaround time (and the usage of simulation resources) could be reduced by 37.2%.

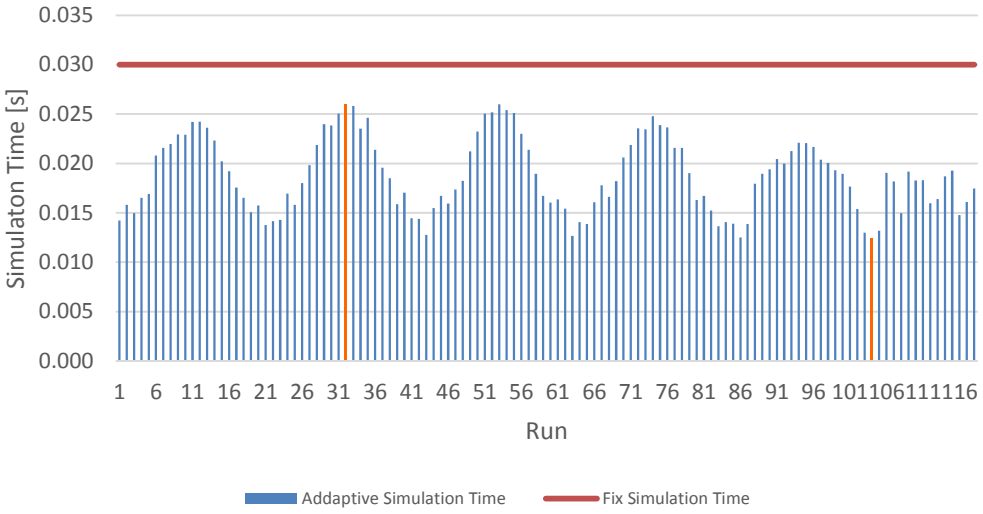

Figure 3. Simulation time used for different impact points of a head impact simulation (118 jobs).

The results for acceleration and displacement for two different impact positions are shown in Figure 4. Comparing the results with fixed simulation time and adaptive job termination shows that the rebound was detected and the jobs were terminated with sufficient data to consider the HIC window.

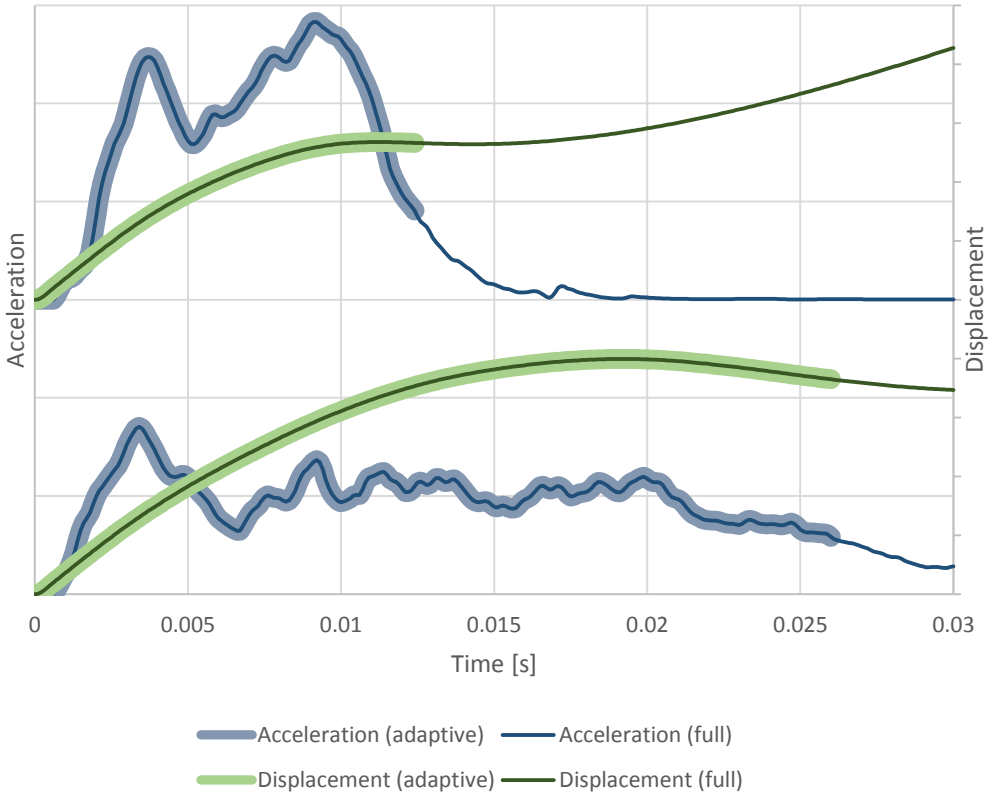

Figure 4. Results (acceleration and displacement of impactor) for simulation with fixed simulation time and adaptive job termination (top: Position A, bottom: Position B).

5. Model tuning

In this section the performance of a full vehicle crash model has to be improved. This model is composed by 10.5M elements and the simulation time is defined as 110ms. With mass scaling the time increment is predetermined to 5.7e-7s, which leads to approximately 200,000 increments. All jobs are run in double precision. This simulation represents an US-NCAP front crash (rigid wall).

All jobs presented in this paper were run on dual socket server, with 10 cores Ivy Bridge processors at 2.8GHz, (Intel® Xeon® E5-2680v2).

5.1 Measuring performance

To optimize the performance of a simulation model, a reliable way to measure the performance is needed. We rely on the timings documented in the sta-file of Abaqus/Explicit.

```
STEP 1  ORIGIN 0.0000

  Total memory used for step 1 is approximately 224.1 gigabytes.
  Element by element stable time estimation algorithm will be used.
  Scaling factor:  1.0000
  Variable mass scaling factor at zero increment: 1.0000
             STEP      TOTAL    DOMAIN      STABLE           CRITICAL    KINETIC       TOTAL    PERCENT
INCREMENT     TIME       TIME      TIME   INCREMENT  MSPEI     ELEMENT     ENERGY      ENERGY  CHNG MASS
        0  0.000E+00 0.000E+00  00:00:56 5.721E-07  0.000    68035612  2.857E+08  2.857E+08  3.546E+00
ODB Field Frame Number      0 at requested time intervals of  4.000000E-03 at increment zero
       53  3.032E-05 3.032E-05  00:02:00 5.721E-07   0.18    11800740  2.857E+08  2.857E+08  3.548E+00
     1085  6.207E-04 6.207E-04  00:04:00 5.721E-07  0.018    25097032  2.857E+08  2.857E+08  3.548E+00
     2073  1.186E-03 1.186E-03  00:06:00 5.721E-07  0.019    66194364  2.855E+08  2.857E+08  3.548E+00
     3002  1.717E-03 1.717E-03  00:08:03 5.719E-07  0.020    98029070  2.854E+08  2.857E+08  3.548E+00
     3948  2.258E-03 2.258E-03  00:10:02 5.720E-07  0.019    66363795  2.852E+08  2.857E+08  3.548E+00
     4834  2.765E-03 2.765E-03  00:12:02 5.720E-07  0.021    66241286  2.849E+08  2.857E+08  3.548E+00
     5698  3.259E-03 3.259E-03  00:14:02 5.718E-07  0.021    66004165  2.846E+08  2.857E+08  3.548E+00
     6583  3.765E-03 3.765E-03  00:16:02 5.716E-07  0.021    66114810  2.841E+08  2.857E+08  3.548E+00
     6995  4.001E-03 4.001E-03  00:17:06 5.716E-07  0.024    66357780  2.838E+08  2.857E+08  3.548E+00
ODB Field Frame Number      1 at requested time intervals of  4.000000E-03 at  4.000513E-03
     7623  4.359E-03 4.359E-03  00:18:57 5.720E-07  0.027    66003777  2.835E+08  2.857E+08  3.548E+00
     8469  4.843E-03 4.843E-03  00:20:57 5.716E-07  0.022    97001226  2.830E+08  2.857E+08  3.548E+00
     9318  5.328E-03 5.328E-03  00:22:57 5.708E-07  0.022    66026642  2.825E+08  2.857E+08  3.548E+00
.....................................
  CPU time of step corresponds to 2.35149E-02 MSPEI.
```

The wall clock time is denoted as domain time and does not include the time for the pre-phase and the packager. The sta-file includes the so-called MSPEI (micro second per element increment) value, which should be independent from the model size. Beside the fact that this quantity can vary during the simulation, it can be used for evaluating the performance without running the job till the end.

The output of the MSPEI value is switched off by default. To activate the MSPEI output the following setting has to be included in the environment file:

```
import os
os.environ['ABQ_XPL_MSPEI']='ON'
del os
```

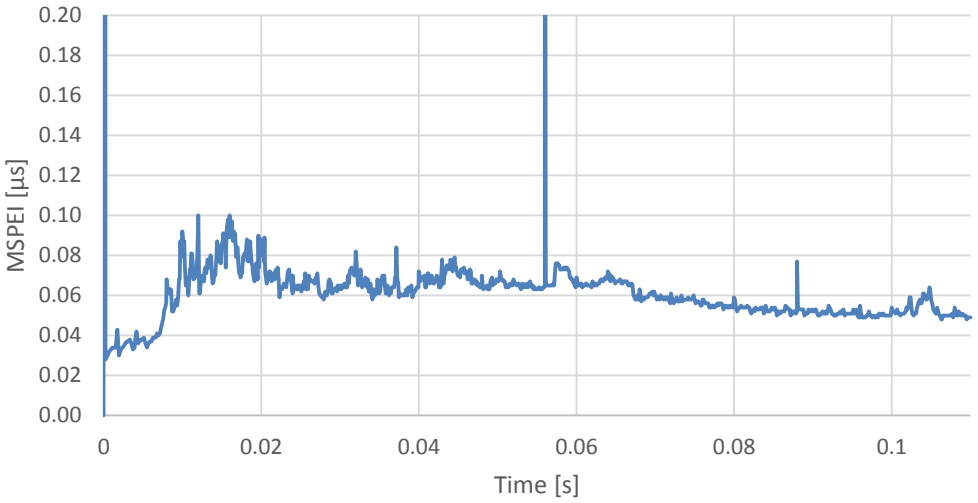

Figure 5. MSPEI value over simulation time on 100 cores, no dynamic load balancing, turnaround: 32:02:33.

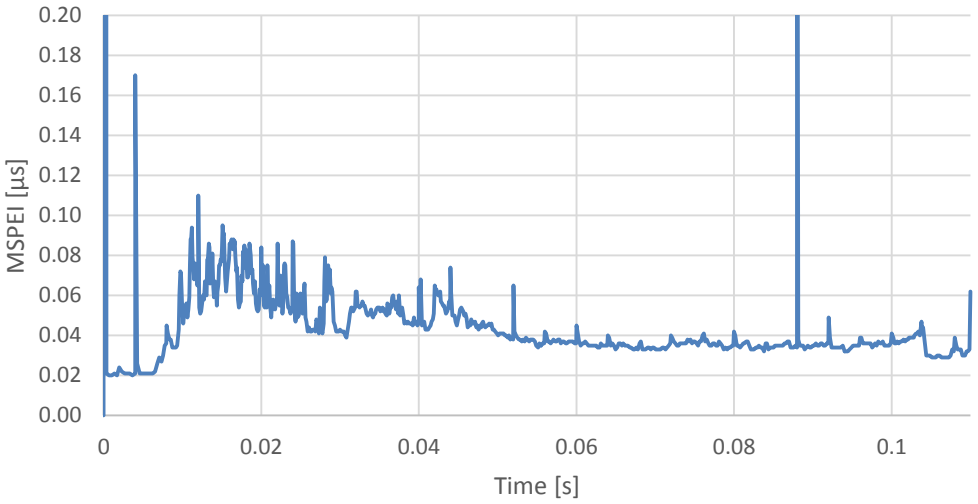

Figure 6. MSPEI value over simulation time on 200 cores, with 2x dynamic load balancing, turnaround: 22:22:02.

In Figure 5 and Figure 6 the MSPEI value is plotted over the simulation time for this job run with 100 cores without dynamic load balancing as well as 200 cores with 2x dynamic load balancing. The varying MSPEI over the simulation time can clearly be seen. At the beginning of the simulation the 100 core job has a MSPEI value of approx. 0.03 and the 200 core job of 0.02 which represents roughly the ratio of the final turnaround time.

An overview of all timings for the original model can be found in Table 1.

Table 1. Timing of the original model.

	100 Cores	*200 Cores*
no DLB	32:02:33	28:11:20
2x DLB	32:16:45	22:22:05

Summarizing these timings leads to the following conclusions:

1. Poor scaling without dynamic load balancing.
2. Fair scaling with 2x dynamic load balancing.
3. No effect of dynamic load balancing with 100 cores.
4. Significant effect of dynamic load balancing with 200 cores.

5.2 Identifying performance bottlenecks

To improve the performance of this model the performance bottlenecks have to be identified. Abaqus/Explicit helps with a report of timings for different computation phases for every domain. This report is printed in the log file.

Phase	DOMAIN 1	%	DOMAIN 2	%	DOMAIN 3	%	DOMAIN 4	%	DOMAIN 5	%
Element Calculations	4.249E+03	14.5	8.063E+02	2.7	1.338E+03	4.5	4.109E+03	13.8	3.731E+03	12.5
General Contact	1.057E+04	36.0	1.317E+04	44.2	1.184E+04	39.7	6.457E+03	21.7	9.510E+03	31.9
Constraint Solution	2.736E+02	0.9	9.857E+01	0.3	2.492E+02	0.8	4.473E+02	1.5	5.439E+02	1.8
Nodal Updates	3.552E+02	1.2	1.302E+02	0.4	1.577E+02	0.5	2.936E+02	1.0	3.452E+02	1.2
External Forces	6.386E+00	0.0	3.490E+00	0.0	5.957E+00	0.0	1.649E+01	0.1	2.803E+00	0.0
Fluid Cavity Volume	1.645E+01	0.1	1.818E+01	0.1	1.819E+01	0.1	3.595E+01	0.1	9.592E+00	0.0
Output Procedures	2.863E+01	0.1	5.333E+00	0.0	9.336E+00	0.0	2.225E+01	0.1	1.873E+01	0.1
Comm Fld & Hist Output	3.140E+02	1.1	2.919E+02	1.0	3.074E+02	1.0	3.119E+02	1.0	1.361E+02	0.5
Comm Nodes Elements	2.069E+03	7.0	1.425E+03	4.8	2.403E+03	8.1	2.180E+03	7.3	1.964E+03	6.6
Comm General Contact	6.460E+03	22.0	7.497E+03	25.2	8.061E+03	27.0	1.078E+04	36.2	8.813E+03	29.6
Comm Constraints	1.214E+03	4.1	1.387E+03	4.7	1.266E+03	4.2	1.020E+03	3.4	1.300E+03	4.4
Comm Reduction Ops	3.051E+03	10.4	3.893E+03	13.1	3.056E+03	10.3	2.978E+03	10.0	3.184E+03	10.7
Comm Fluid Cav Volume	9.266E+01	0.3	9.858E+02	3.3	9.900E+02	3.3	1.041E+03	3.5	1.384E+02	0.5
Comm DOMAIN Shuffling	6.630E+00	0.0	3.599E+01	0.1	3.748E+01	0.1	3.710E+01	0.1	3.485E+01	0.1
End Incr Output & Comm	6.730E+02	2.3	5.706E+01	0.2	5.828E+01	0.2	6.020E+01	0.2	6.037E+01	0.2
Total	2.938E+04		2.981E+04		2.980E+04		2.979E+04		2.979E+04	

In this table, the timings for different computation phases are listed for each domain. With this information at hand, type and location of performance bottlenecks can be identified.

This output will be activated by switching on the MSPEI output. To ensure a reporting based on domains instead of CPUs, which is essential in the case of dynamic load balancing, the following setting has to be added to the environment file:

```
import os
os.environ['ABA_XPL_DOMAINTIMERS'] = 'ON'
del os
```

In Table 2, the timings are summed up over all domains. In case of perfect scaling, the timings should not depend on the core count. This is true for *General Contact* and *Element Calculations*, which needs approx. 1.2M seconds and 2.0M seconds respectively. The scaling of the *Constraint Solution* is fair as well. There are some more phases starting with *Comm*, which denotes communication phases. These phases contain the time for exchanging information and data as well as waiting time. *Comm General Contact* is very interesting, since the scaling is very poor. In fact, for 100 cores Abaqus/Explicit takes approx. 7.5M seconds and for 200 cores (no dynamic load balancing) approx. 15M seconds. For this phase, doubling the number of cores has no positive effect, because the used CPU time doubles as well. The timings for *Constraint Solution* and *Comm Constraints* are small compared to the timing of the remaining phases. A reason for this is the usage of strict modeling guidelines and checks to avoid most of the overlapping constraints.

Table 2. Summary of timings for different computation phases.

	100 Cores no DLB	*100 Cores 2x DLB*	*200 Cores no DLB*	*200 Cores 2x DLB*
Comm Constraints	167.334	174.182	190.617	224.316
Constraint Solution	244.484	257.456	271.003	297.941
Comm General Contact	7.333.630	7.272.685	15.270.850	9.917.223
General Contact	1.155.905	1.184.108	1.214.679	1.292.754
Element Calculations	1.951.340	1.984.106	2.014.866	2.082.472
REST	854.818	994.904	1.670.926	2.783.396
Sum	**11.707.510**	**11.867.441**	**20.632.941**	**16.598.102**

Much more helpful is a graphical visualization of the timings of the computation phases. In Figure 7 and Figure 8 visualizations of the jobs with 100 and 200 cores (without dynamic load balancing) are shown. On the x-axis, the domain number can be found and on the y-axis the wall clock time. The approx. 115,000 seconds represents roughly 32 hours in Figure 7 and the 101,000 seconds represents 28 hours in Figure 8. In total, all domains take the same time to complete the computational tasks.

In both figures the peaks in *General Contact* at low domain numbers can be identified. These peaks result in a big amount of *Comm General Contact* for the remaining domains, which is most likely waiting time associated with completing contact computation on all domains. For *Element Calculations* and the *REST* the load is quite regularly distributed. *Constraint Solution* and *Comm Constraint* are regularly distributed as well and quite small.

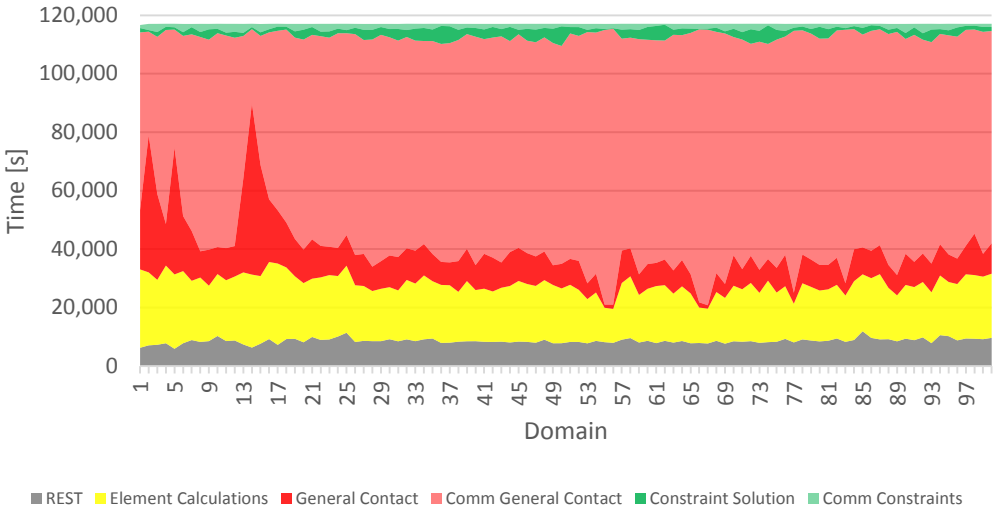

Figure 7. Visualization of load balancing (100 cores, without dynamic load balancing, turnaround: 32:02:33).

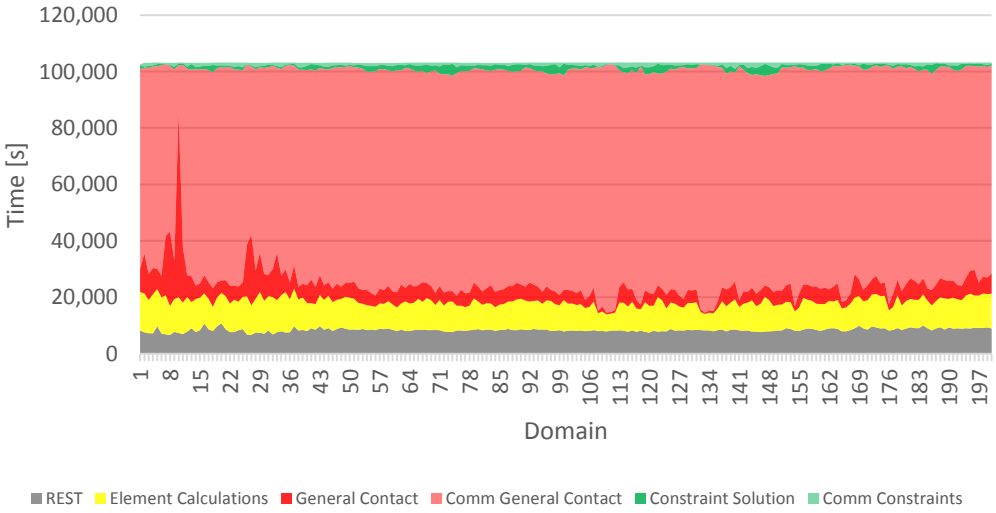

Figure 8. Visualization of load balancing (200 cores, without dynamic load balancing, turnaround: 28:11:20).

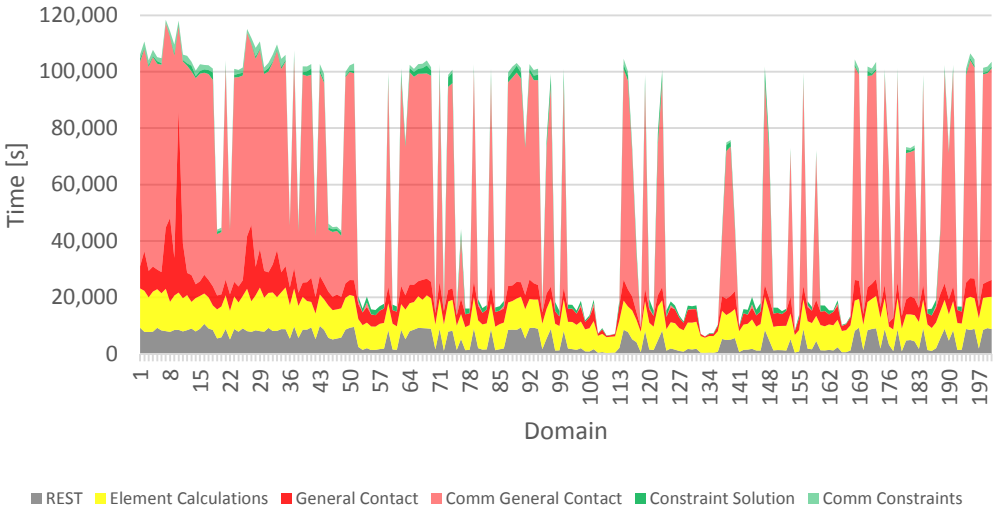

Figure 9. Visualization of load balancing (100 cores, 2x dynamic load balancing, turnaround: 32:16:45).

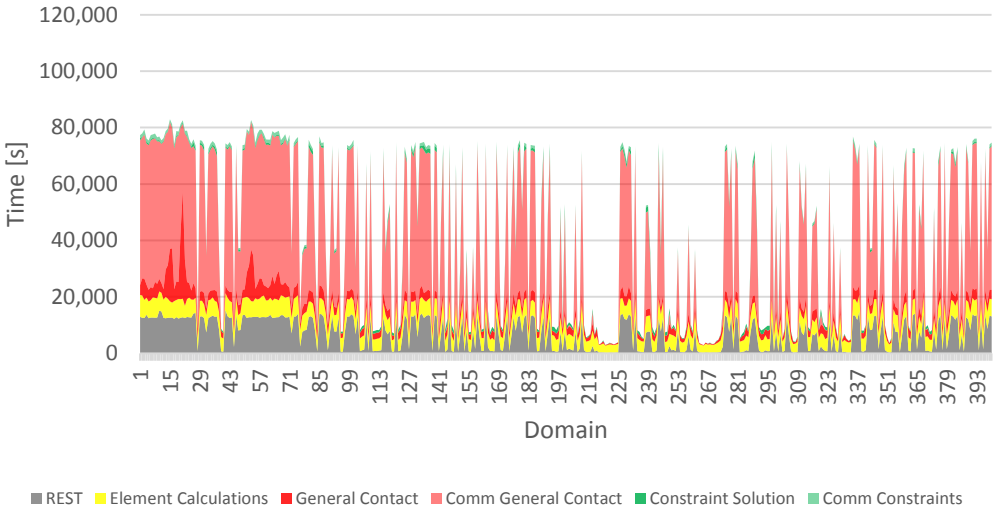

Figure 10. Visualization of load balancing (200 cores, 2x dynamic load balancing, turnaround: 22:22:05).

In Figure 9 and Figure 10 visualizations of the jobs with 100 and 200 cores (with 2x dynamic load balancing) are shown. In case of dynamic load balancing, the model is decomposed in more domains than available cores. Abaqus/Explicit distributes the domains on the cores in a way that the computational load is shared evenly between all the cores. This results in a comb shaped load distribution, since the domains do not take the same time to complete.

During the simulation, the timings of the domains are monitored and dynamically redistributed to achieve an optimal load balancing.

The type of visualization in Figure 9 and Figure 10 tends to be confusing for regular users, as such we decided to use an alternative visualization shown in Figure 11 and Figure 12. In this visualization, only the *Element Calculations*, *General Contact* and *Constraint Solution* are plotted. One advantage is that this visualization is independent of activating dynamic load balancing. Both figures show more or less the same load balancing, despite the fact that the load balancing in Figure 11 is from a job with 100 cores and 2x dynamic load balancing and in Figure 12 with 200 cores and without dynamic load balancing. Both configurations result in the same number of domains. The second advantage of this visualization is the fact that only computation phases are shown, which can be directly influenced by modeling techniques.

In both figures the characteristic peaks in *General Contact* can be seen. This peak dominates the overall turnaround time of this job. To address this, areas in the model with the high contact load have to be divided into smaller domains. This can be done by increasing the number domains (more cores or higher factor for dynamic load balancing). In both cases the whole model will be divided into smaller domains. This increases the communicational effort which probably offsets any potential gains from reducing the peaks.

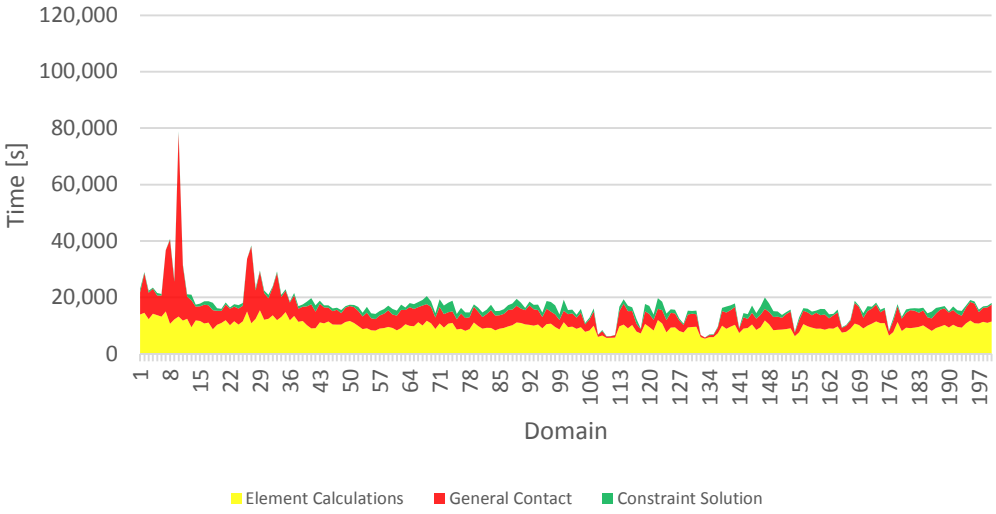

Figure 11. Alternative visualization of load balancing (100 cores, 2x dynamic load balancing).

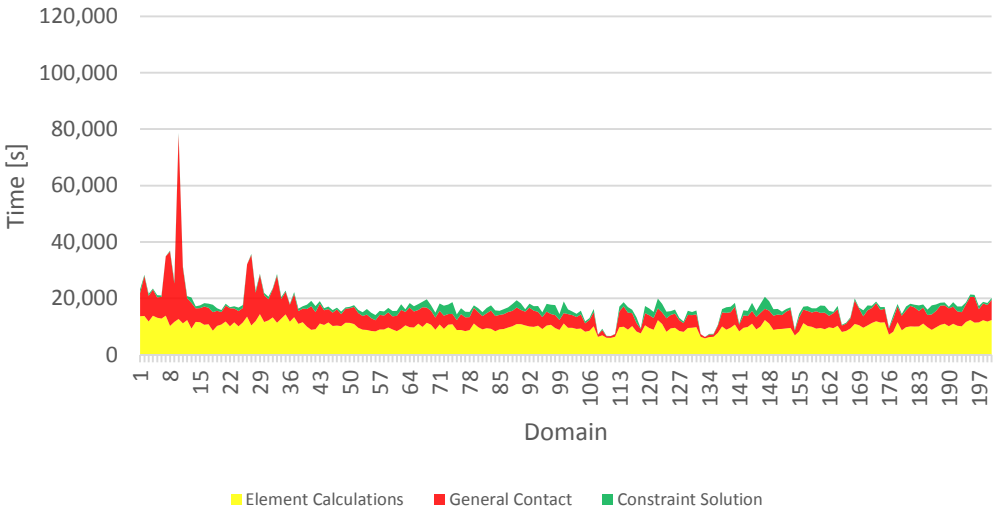

Figure 12. Alternative visualization of load balancing (200 cores, without dynamic load balancing).

5.3 Addressing performance bottlenecks

A smarter way to improve the performance is to create an additional set of small domains in a local area of the model. Since Abaqus 6.14 this can be done with the keyword `*DOMAIN DECOMPOSITION`. Two different ways can be used to define this local area:

1. Definition by box.
2. Definition by element set.

5.3.1 Definition by box

The definition by box has the advantage that this technique can be automated. We know in advance the impact zone of a crash model. The impact zone is typically the area of the model with the highest computational load. Unfortunately, modeling issues resulting in a performance penalty, which are not located in the impact zone, will not considered by this method. To use the dynamic load balancing feature, no additional over decomposition has to be defined on the command line during job submission, since there is an over decomposition due to additional domains from the user defined decomposition.

The turnaround time are listed in Table 3.

Table 3. Timing of the tuned model (`*DOMAIN DECOMPOSITION, TYPE=BOX`).

	100 Cores	*200 Cores*
no DLB	25:54:32	20:55:19
DLB	23:32:05	19:59:48

Summarizing these timings leads to the following conclusions:

1. Significantly improved performance compared to the original model.
2. Poor scaling with and without dynamic load balancing.
3. Small effect of dynamic load balancing with 100 cores and 200 cores.

5.3.2 Definition by element set

To apply the user defined domain decomposition with an element set, we need to know the load balancing information. With these information at hand all domains with a computational load above average can be identified and the element labels can be extracted.

The advantage of this method comes from a perfect adaption of the domain decomposition on the specific model. The disadvantage is the fact that we have to run the job at least once before. The behavior regarding performance after changing the mesh inside the refined zone with small domains is unpredictable.

The run to evaluate the load balancing can be carried out with different settings (number of cores, dynamic load balancing) than the final runs. In this example we tested two variants. Variant A uses the load balancing data from the original run without dynamic load balancing, Variant B uses the data from the run with dynamic load balancing. This results in slightly different sets, however

the performance results should be comparable between Variant A and B, since the choice of the size of the element set should not be super critical.

The turnaround times are listed in Table 4.

Table 4. Timing of the tuned model (*DOMAIN DECOMPOSITION, TYPE=ELSET).

	100 Cores	*200 Cores*
DLB, Variant A	20:36:25	16:10:25
DLB, Variant B	21:37:00	21:48:51

Summarizing these timings leads to the following conclusions:

1. Further improved performance compared to the original model and model with `*DOMAIN DECOMPOSITION, TYPE=BOX`.
2. Surprisingly big difference in performance for 200 cores for both variants.
3. No scaling for Variant B from 100 to 200 cores.

5.4 Identifying modeling issues

In Figure 13 and Figure 14 the two different areas of the model can be seen. On the right hand side the domain timing for the refined zone with small domains is plotted. With some exceptions, these timings are significantly smaller than the domain timings on the left hand side of the remaining model. With the dynamic load balancing technology, Abaqus/Explicit finds the optimal distribution among the cores of these domains to reduce the turnaround time.

On the right hand side of Figure 13 a number of peaks can be identified. On the right hand side of Figure 14 only one, but very high peak is visible. This leads to the significant difference in runtime, since the runtime is dominated by the highest peak.

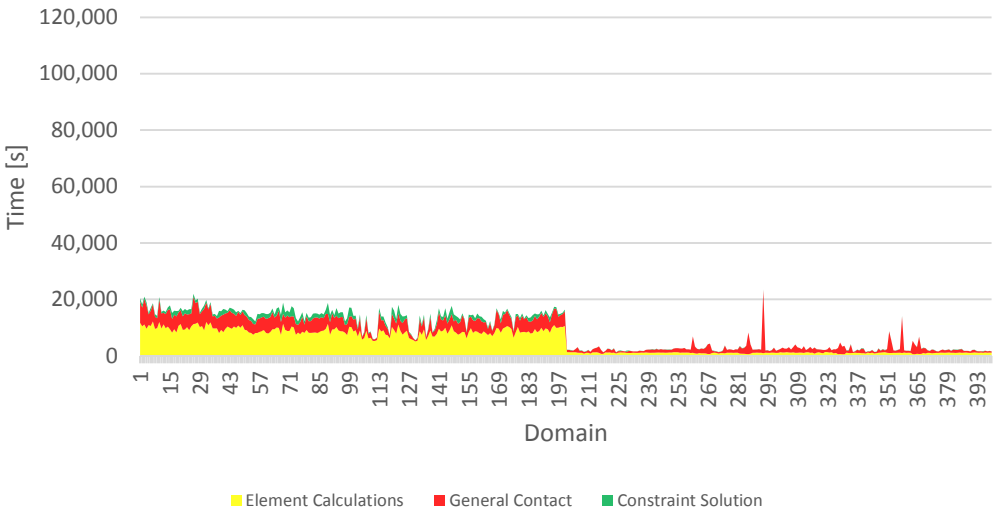

Figure 13. Visualization of load balancing (Variant A, 200 cores, dynamic load balancing, turnaround: 16:10:25).

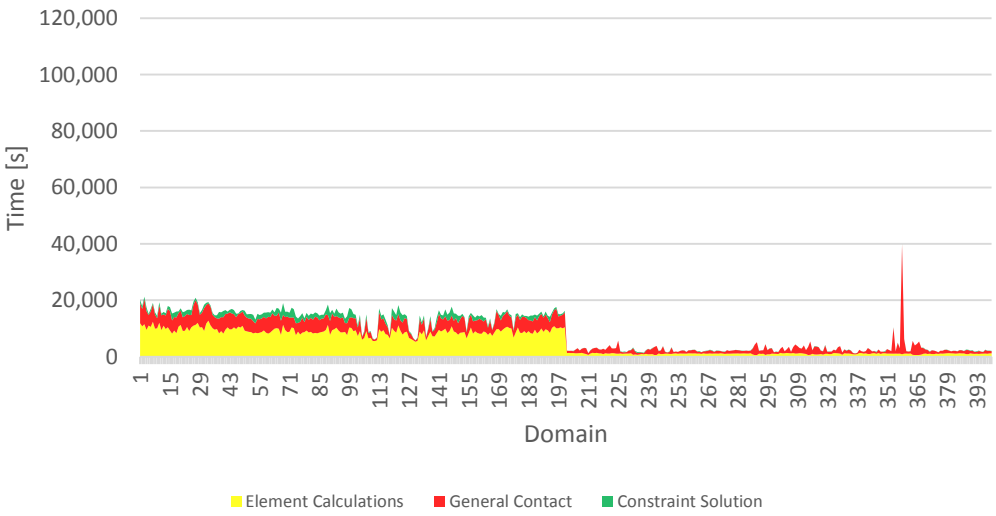

Figure 14. Visualization of load balancing (Variant B, 200 cores, dynamic load balancing, turnaround: 21:48:51).

After analyzing these visualization of the load balancing, the following points has to be concluded:

1. The peaks are in red, thus there is an issue with the contact computations.
2. The part of the model containing the issue tends to be very small, since the entire issue can be found in only one small domain.
3. The turnaround time can vary significantly depending on decomposing the modeling issue into multiple domains by chance.

In fact, investigating the domain with the excessive computational load for contact turns out, that there is a small component of the car (Spirit of Ecstasy or Emily) with a very fine mesh (0.5mm mesh size). The Emily is idealized as a rigid body, but with `*MPC` functionality of Abaqus/Explicit. It is recommended to use `*RIGID BODY` instead. `*RIGID BODY` can utilize the rigidity information in a more efficient algorithm for contact tracking. Moreover, `*RIGID BODY` probably avoids calling the element routines. In this case surface-elements without element routines are used, thus there is no yellow component in the peaks.

Table 5. Timing of the corrected model

	100 Cores	*200 Cores*
no DLB	25:25:15	15:24:47
2x DLB	18:47:01	13:53:59

In Table 5 the turnaround times for the corrected model are documented. These timings are much better than the timings from the uncorrected, but tuned model (Table 3 and Table 4). This specific performance issue with `*MPC` in combination with a very fine mesh was addressed and fixed in a maintenance release of Abaqus 2018. Nevertheless, the methodology holds for any kind of modeling issue affecting the performance.

Applying the `*DOMAIN DECOMPOSITION, TYPE=ELSET` technique to the corrected model leads to the turnaround timings in Table 6.

Table 6. Timing of the corrected and tuned model (`*DOMAIN DECOMPOSITION, TYPE=ELSET`).

	100 Cores	*200 Cores*
DLB, Variant A	17:55:27	12:11:48
DLB, Variant B	17:34:29	12:14:11

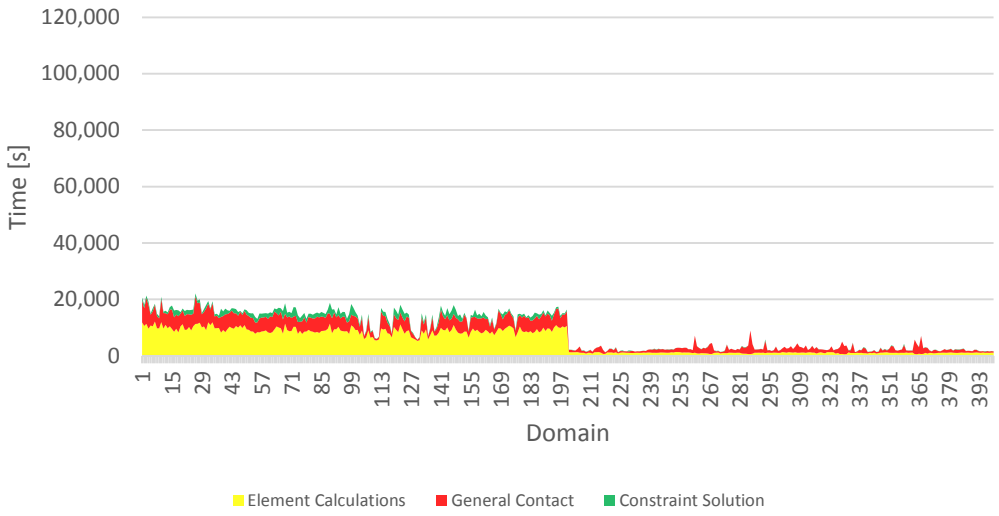

Figure 15. Visualization of load balancing (Variant A, 200 cores, dynamic load balancing, turnaround: 12:11:48).

In Figure 15 the visualization of the load balancing for the job (Variant A) with 200 cores, dynamic load balancing is shown. This load balancing is close to an optimal distribution which is also reflected by the significantly reduced turnaround time.

6. Summary

In this paper three methods to increase the simulation throughput and reducing the turnaround time are presented. This helps to reduce the cost and to increase the productivity.

Method one utilizes parallel execution of jobs, thus increasing turnaround time of a single job, but increases throughput to reduce the turnaround time of the whole set of simulation jobs (e.g. DOE).

The second method optimizes the physical time to be simulated to ensure, that the shortest possible time will be used without losing any important information. This approach can be adopted to almost every crash simulations we carry out at BMW to optimize the physical time to be simulated.

Finally, a method for tuning the performance of simulation models is presented. With this method at hand, performance bottlenecks can be identified and with existing technologies in Abaqus/Explicit addressed. Moreover, this method helps to identify and localize modeling issues affecting the performance.