# In Core Adaptivity

Brian Wainscott and Houfu Fan
*LSTC*

## Abstract

*Adaptivity is almost universally used in metal forming applications. As useful as it is, there are certain inefficiencies in its execution which are a result of the way in which it was implemented. The current approach requires a significant amount of I/O and code serialization. A new method is being developed in MPPDYNA which not only performs the adaptivity in parallel, but without exiting the solution loop. The current status of this work is presented.*

## Background

Adaptivity is a technique used primarily in metal forming applications, in which nodes and elements are automatically generated in areas of high curvature. This allows the user to input a coarsely meshed blank and at the same time capture fine details in the finished part. Early portions of the simulation run faster due to having fewer elements than the final closing requires, and the user doesn't have to know ahead of time where a dense element mesh will be needed.

The limitations of the current implementation of adaptivity are an unavoidable result of the state of the art in computing during the time in which it was developed. The technology of the time did not allow for easy or efficient dynamic memory allocation or memory management. One result of this is that almost all problem data in LS-DYNA® is stored in one large array which is partitioned up into individual pieces during input processing. As a result, it has been impractical to change the number of nodes or elements in the simulation during the solution loop. To perform an adaptive step, LS-DYNA writes the current solution information to disk, creates an input file for a refined version of the model, and "starts over." It goes back through the input phase and then initializes this new problem from the information written to disk earlier.

## A New Approach

The time for all this disk I/O can be significant for a large model, and the serialization involved in these steps limits scalability. The only way to avoid this would be to somehow create new nodes and elements "on the fly" without leaving the solution loop. Different approaches to this problem can be conceived. For example, pre-allocating extra nodes and elements which are inactive until needed. But this requires knowing ahead of time how many to pre-allocate, and implementation would be difficult and somewhat limited.

Some time ago, a new dynamic memory facility was added to LS-DYNA which makes use of some of the features of modern FORTRAN standards. By moving data out of the single large array and into their own dynamically allocated arrays, it becomes possible to change their size while the problem is running. This makes it possible to consider a new approach to adaptivity.

In order to be able to create a new node, every data array whose size depends on the number of nodes has to be moved into a new dynamic array. For each one, every place in the code which references the "large array" location of the data has to be modified to reference the new dynamic array. Shells of course have the same issue. The scope of work involved is substantial.

Over 40 node and 30 shell related arrays have so far been moved out of the "large array" memory, and it is now possible to create new nodes and shell elements during execution, with some limitations. The limitations are because of the scope of the current work, and the test problems so far encountered. Any feature which references nodes or shells (e.g. *ELEMENT_SEATBELT) could potentially need modification to work properly when new nodes and elements are created. Many such features exist which would probably not work correctly at the moment.

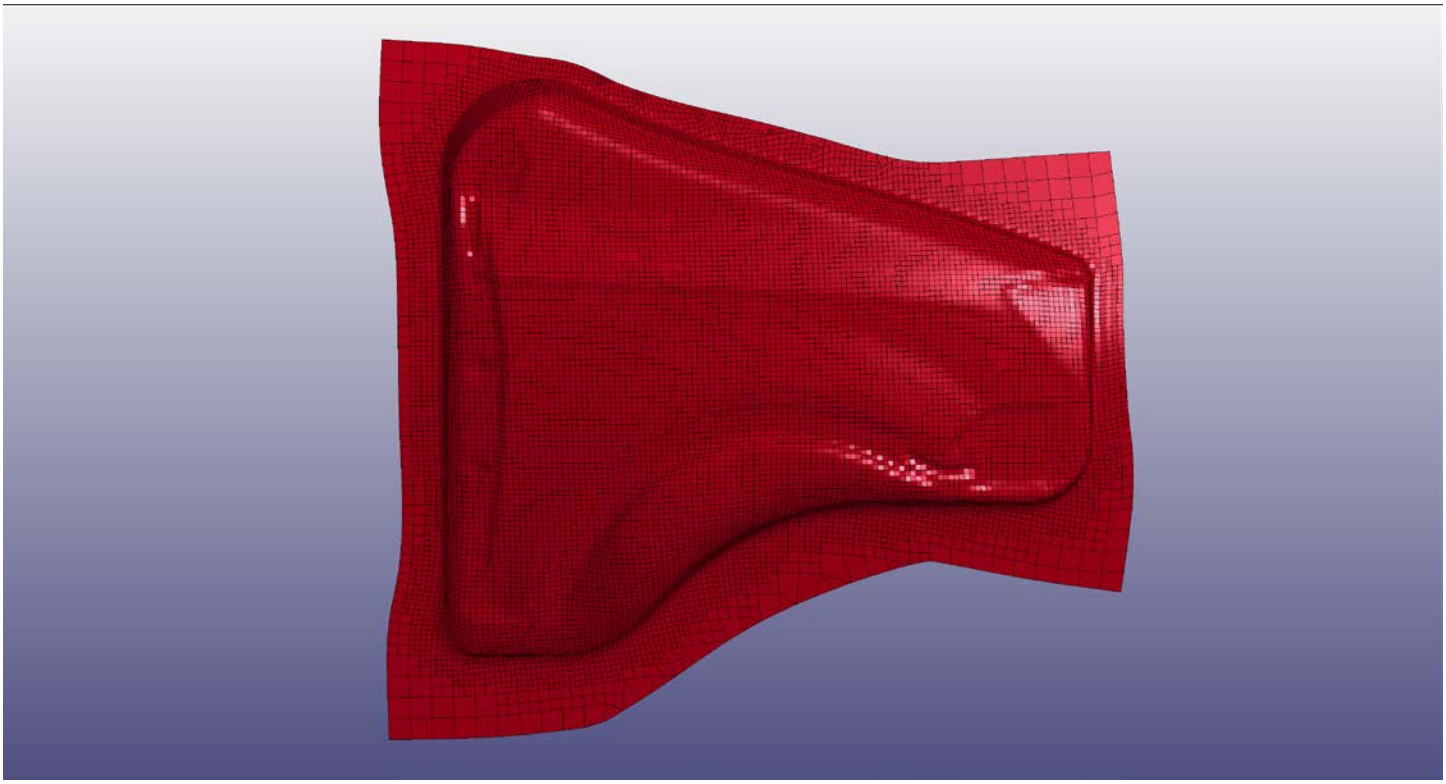
For the remainder of this paper, we will refer to the historical adaptivity approach as the "old" method, and the newly developed in-memory approach as the "new" method.

## Test Results

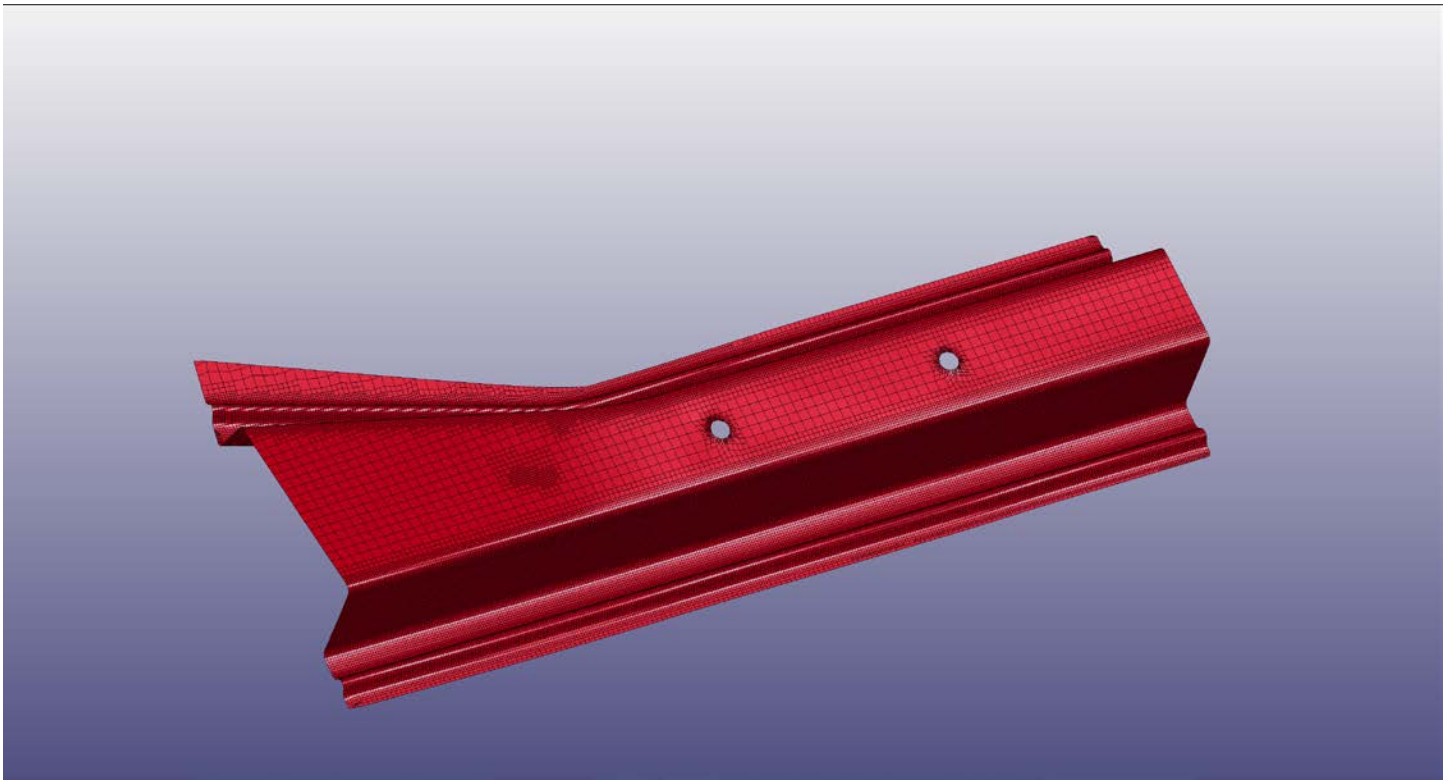
All the results shown here were obtained with the single precision development version of MPP LS-DYNA r123437 on January 19, 2018. They were run on our 768 core cluster (2 sockets per node, 6 cores per socket) with infiniband interconnect.

Two test problems are presented here.

Model 1 is a small fender stamping model that starts with 1008 elements in the blank, and ends with about 14K elements in the blank. It runs 107K cycles and has 132 adaptive steps.



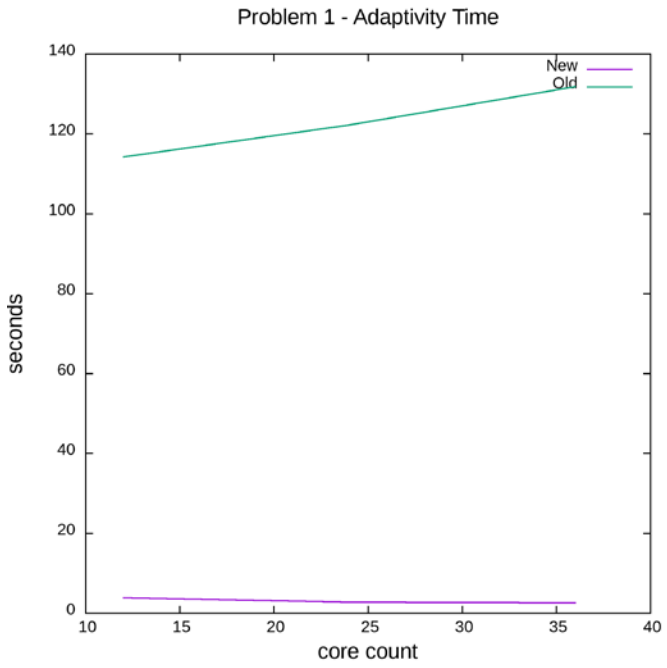
Model 2 is a simple piece of channel with 3977 shells in the initial blank. After 69K cycles and 101 adaptive steps, the final mesh has about 50K elements.

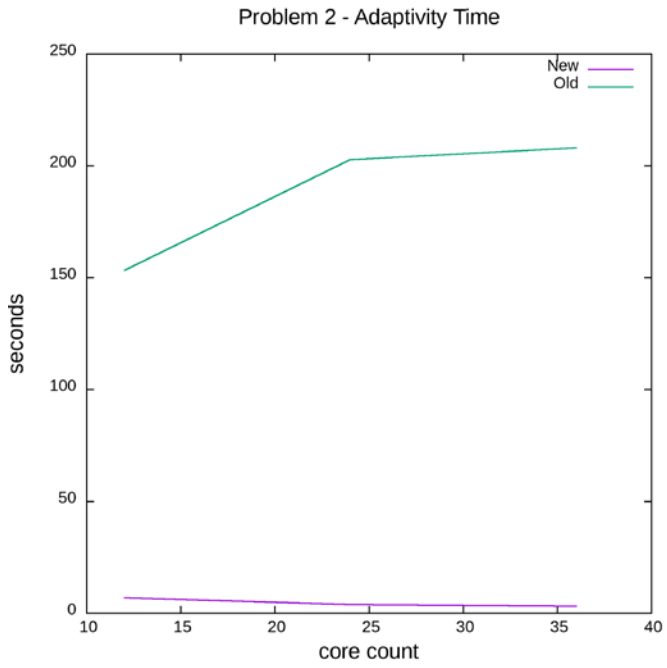


Each model was run on 12, 24, and 36 cores. Model 1 does not use mass scaling, and so the number of time steps changed from run to run. This makes simple time comparisons invalid, and so instead we looked at run times divided by the total number of simulation cycles. Time spent in adaptivity, on the other hand, does not depend on the time step size, and every simulation had the same number of adaptive steps, so that data is directly comparable. All timing data was taking from the table at the bottom of the d3hsp file from each run.

The first time measure of interest is the time spent doing adaptivity. For the old method, this is taken to include all the time spent in "Keyword Processing" plus "MPP Decomposition" plus "Initialization", as these are all repeated each adaptive step. As reported in the d3hsp, these times of course include the non-adaptive, "first time through" costs incurred before cycle 1. These should be identical whether using the old or new approach, so the sum of these times from the new method were subtracted out, leaving just the extra time associated with actual adaptivity. The actual time spent determining which elements to adapt and computing the new mesh are not broken out in the timing table, and so the numbers used here are known to be lower than the true cost of the old adaptive approach.

For the new approach, the actual time spent determining the target shells and modifying the mesh are broken out in the timing table, so those are available directly.

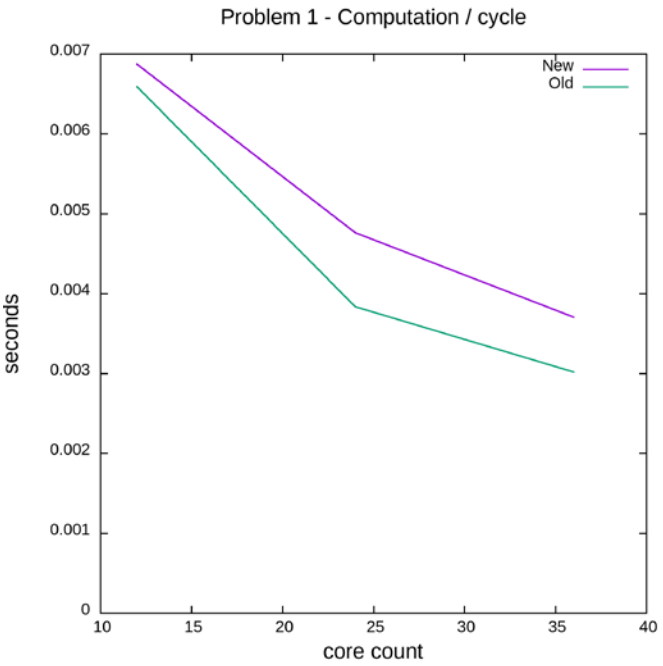


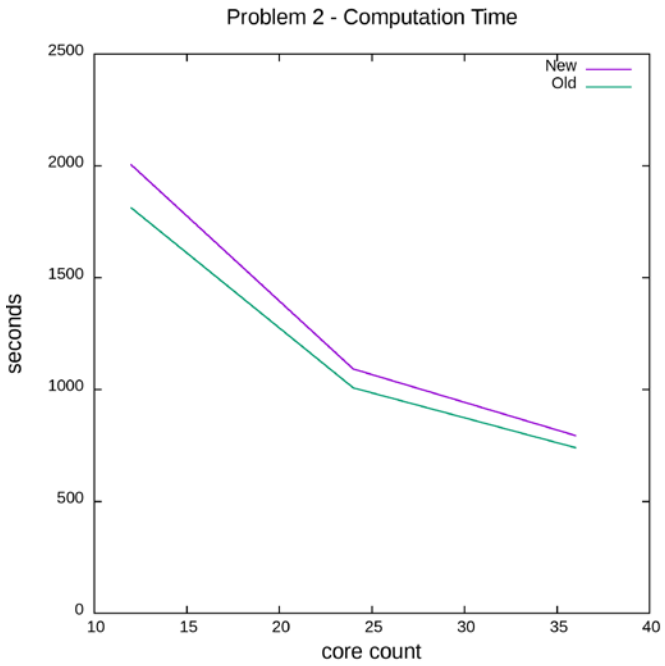




On problem 1, the old code spent between 114 and 131 seconds with this part of the solution, out of a total run time of 920 to 504 seconds. The adaptive time went up as the number of cores increased, while the run time dropped. As a result, the percentage of time spent doing this ranged from 12% to 26% of the total run time, getting worse as the number of cores increased. The new code spent between 3.9 and 2.6 seconds doing all the work related to adaptivity. This was about 0.53% to 0.66% of the total run time.

For problem 2 the results are similar. The proportion of time spent doing old method adaptivity increases with core count, from 7.8% up to almost 22% in this example. The new approach ranges from 0.34% to 0.39%

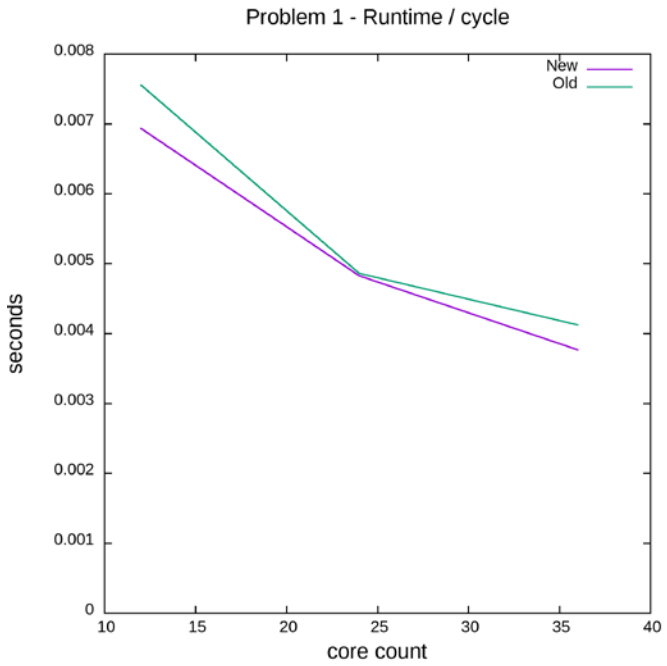
With such dramatic improvements in adaptive time, one would expect the total run times to show substantial improvement. Unfortunately, there is one important function that occurs in the old approach that is lost when the "stop and start over" cycle is abandoned: re-balancing the calculation. In the new approach, each processor generates new elements at a different density, according to the needs of the model. This results in a poor work load distribution, and less efficient use of the available CPU power.

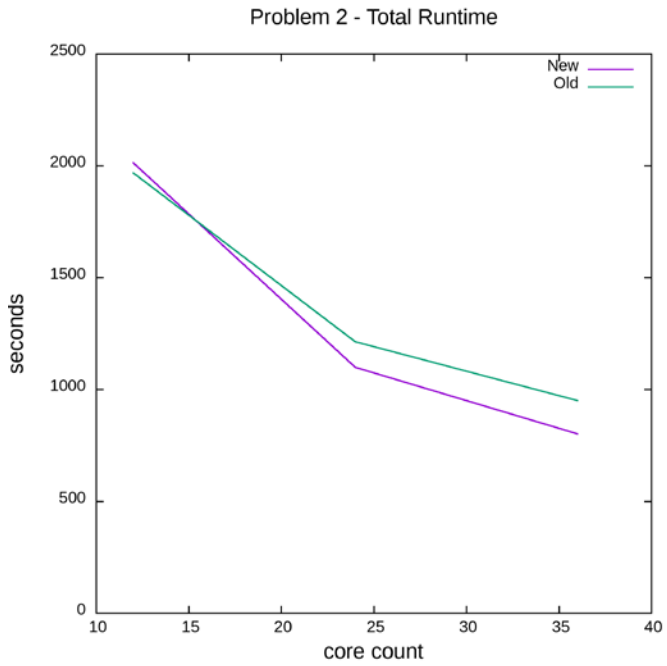



Computation time, as used here, is taken as total run time minus (initialization time + adaptivity time). As you can see, the new algorithm currently has longer computation times in all cases. The effect is no so great in problem 2, where the adaptivity is relatively uniform and the decomposition was chosen to take advantage of this fact. Even so, the differences in computation time are a result of the unbalanced calculation caused by unequal adaptivity on each processor.

Things are not quite as bad as could be, however. Even with this imbalance, the total run times were lower with the new approach, and the parallel scalability is substantially improved. The expectation is that this will only get better as the load balance issue is dealt with.




## Usage

At the time of this writing, only a limited subset of LS-DYNA capabilities are compatible with in-memory adaptivity. Only beam and shell elements may be present, and only contact types *CONTACT_DRAWBEAD and the *CONTACT_*_TO_SURFACE contact types (including the FORMING and SMOOTH variations). Turning on the in-memory adaptivity option will automatically switch to the GROUPABLE version of the contact routines for all contacts, and only one pass adaptivity (ADPASS=1) is currently supported.

To try this new feature, simply turn on the "INMEMORY" flag by placing a 1 in the first field of the 5th card of *CONTROL_ADAPTIVE

## Future work

A substantial amount of work remains to be done in terms of supporting the myriad of options currently available in LS-DYNA, and making sure they all work with this new capability. Work will be done on the re-balancing issue, which should increase the overall performance. Having removed the memory bottleneck and serialization of re-initialization with each adaptive step, we expect that adaptive metal forming problems can now be run more efficiently on large numbers of processors.