Processor Count Independent Results: Challenges and Progress

Brian Wainscott[1], Zhidong Han[1]

[1]LSTC

1 Introduction

When a different number of cores is used to run the same model in MPPDYNA, or a change is made in the decomposition, different results are generally produced. We discuss a couple of different reasons for this, and why it is to be expected in a parallel program. Work is in progress to offer options which address this issue, and results of some test problems are presented.

2 MPP: The Promise and the Problems

The promise of distributed parallel computing is of course improved efficiency and decreased runtime. In order to achieve the greatest possible performance, a parallel program should spend as little time as possible waiting. A straightforward MPP implementation of this principal naturally leads to a program where data is processed as soon as it arrives on a core, and the results are returned as soon as they are complete, so that other cores are not required to wait. When no data from other cores is waiting to be processed, a core should be working on its local data so that it is kept busy at all times.
This approach can have interesting consequences. In the early implementation of contact in MPP LS-DYNA, this approach was taken: data was processed on each processor as it arrived. Performance was good, but there was a "problem." If a model was run twice in a row with the same executable and the same number of cores, you would generally get a different answer. Due to minor fluctuations in the computing environment on each core, data would arrive in a different order each time, resulting in a different computational result. In one sense, this isn't really a problem at all: there is no a priori reason to prefer one of these results to the others, or to be able to claim one is correct and the others are wrong. From a practical standpoint however, it made engineering decisions difficult. And debugging the code was very time consuming. It is hard to find and fix bugs that only occur once every 10 times you run a problem.
The solution to this situation was to force the order in which data arriving on a core was processed. This was a simple matter of making the order of the data messages deterministic. Calculations on the same number of cores then became completely repeatable. But changing the decomposition or the number of cores changes everything.
There are two basic issues at play here. The first is floating point roundoff, and the second is algorithmic decision making.
Roundoff is an issue for any parallel program that deals with summation of floating point numbers. This is very easily illustrated with a simple example. Suppose we have a theoretical machine which stores and processes floating point values with 4 digits of precision, and we have to add up the these numbers: 101.0, 1.009, 0.0910 We can add them like this:

(101.0 + 1.009) + 0.0910 = 102.0+0.0910 = 102.0

or like this

101.0 + (1.009+0.0910) = 101.0 + 1.100 = 102.1

This happens whenever more than 2 real numbers must be added. Now suppose we are adding up a list of real numbers on a distributed parallel machine. The most natural thing to do is have each core sum its own data, and then sum the values between cores. This is the fastest thing to do, and requires the least amount of communication. But if the distribution of the data between the cores changes, the final result will change. This is the situation in MPP LS-DYNA whenever the number of cores is changed for a simulation, or even if the number of cores is the same but the decomposition is different. And the summation of real numbers happens in many, many places in the program.

It may seem at first glance that differences of a single bit should not be significant. But experience has shown that, for most explicit calculations, once such a difference occurs anywhere in the code, the effect usually propagates. Soon decisions are made differently: branch conditions in the code take a different path, and the differences build. Usually the different results are just minor variations of each other. But even small differences can make the design process much more difficult.
Algorithmic decision making can also be data distribution dependent, particularly in the contact algorithms. Take for example the simple problem of finding the closest contact segment to a node. The natural thing to do is loop through all the segments and pick the one that is the closest. But what happens in the case of a tie? Generally either the first one found or the last one found is used. But if the segments are processed in a different order, this again leads to different results. In short, no algorithmic decision anywhere in the program can depend on the order of the data, if that order can ever change with core count or decomposition.
None of this is new. These things have been know for a long time. Deciding how to deal with it, or even if it should be dealt with, has been a long standing question. It is expected that a full resolution of these issues will require a lot of work. LS-DYNA currently comprises about 9 million lines of code. Trying to insure all of it gives completely consistent results for any decomposition or core count is a daunting task. Is it worth even doing? Any solution to these two problems is going to decrease performance – extra work is being done to guarantee identical results are obtained. How much impact will it have? If it is too much, customers will not use it anyway. Sure, it would be great from a marketing perspective to be able to say we have this option you can turn on, and you'll always get the same result for any number of processors. But if it slows the code down by a factor of 2 and so no one ever uses it, is it worth the effort? These are the questions that have concerned us while we have instead worked on other features in the program.

3 Strategies Implemented

There are various ways of dealing with the problem of roundoff when summing a list of real values. If you know beforehand exactly how many values you will have, you can assign each one a position in an array, and store each in its proper location as it is computed, so that when you add them up the order is always the same. This is how the element force calculations are handled in the SMP code. Each element exerts forces on its nodes, and these forces have to be summed into the nodal force vector. But the number of elements attached to each node is known. So as each element is computed (in whatever order), the forces it applies to its nodes are stored in precomputed locations. At the end of the element loop, the forces are summed to the nodes from this array and the order, and hence the result, is always the same.
In the MPP code, each element exists on only one core, but a node may exist on any number of cores. In order to get repeatable results, all the element forces on a node have to be collected to some core and summed properly. One approach would be to have an "owning" core for each node, where all the forces are sent. The owner would compute the sum, and send the results back to every core which has a copy of the node. Alternately, every core could send forces to every other core sharing the node, and they can all compute the sum independently. Whichever approach is used, all the data has to arrive at some location before any of it can be summed. Then the same approach that the SMP code uses for element forces can be applied, the only difference being that not all the forces being considered originated from elements on the current core.
In some cases, contact for example, the number of forces being applied cannot be known ahead of time, and varies from cycle to cycle. The precomputed position approach cannot be used in situations like this. The simplest thing to do in this case is to order the forces in some way that results in the sum always being the same. Unfortunately, this means that not only do we have to communicate the forces between processors, but then take care to make sure they are properly ordered.
Repeatable decision making in the contact is achieved by giving each node and segment a global ID that depends only on the model, not the decomposition or distribution of the model. These values are then used as tie breakers, or to force the order of processing whenever there is a potential for order dependent results.
Implementing these strategies and others like them throughout the code can guarantee identical results independent of the core count or decomposition, but a significant effort will be needed to support all features.

4 Progress

Some work along these lines was first started in the MPP code in early 2011. By September of that year, enough progress had been made that a few simple metalforming problems could be run. But performance was a question, and without real customer experience it was hard to know whether it was

worth continuing the work. The project was set aside. In early 2016, the work was revived. Approaches were changed a bit, bugs were fixed, and a concerted effort was made to get at least a couple of real customer problems running and get some real timing data. All the results presented here were obtained with the single precision development version of MPP LS-DYNA r112254, between November 28, 2016 and December 1, 2016. All problems were run on our 768 core cluster (2 sockets per node, 6 cores per socket) with infiniband interconnect. They were run on a variety of core counts, and the nodout files (with 23-25 nodes distributed across the face of the blank) were compared to make sure the results were identical. No pictures of the actual models can be shown.

4.1 Model 1

This is an explicit adaptive forming simulation of a simple car fender. The initial configuration has 27,541 nodes and 30,030 elements. The consistent results finish with 226,335 nodes and 227,292 elements after 119,915 cycles and 324 adaptive steps.

4.2 Model 2

This is an explicit adaptive forming simulation of a simple channel like part. The initial configuration has 20,484 nodes and 19,459 elements. The consistent results finish with 210,067 nodes and 207,601 elements after 69,064 cycles and 100 adaptive steps.

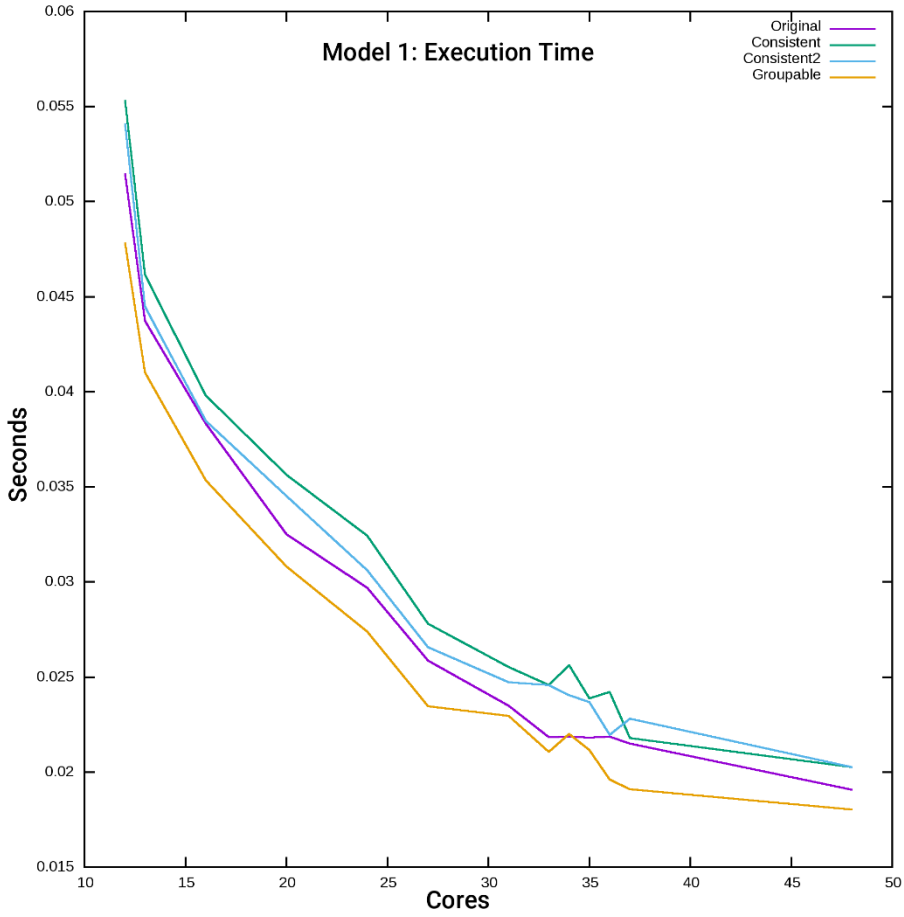
4.3 Description of Calculations

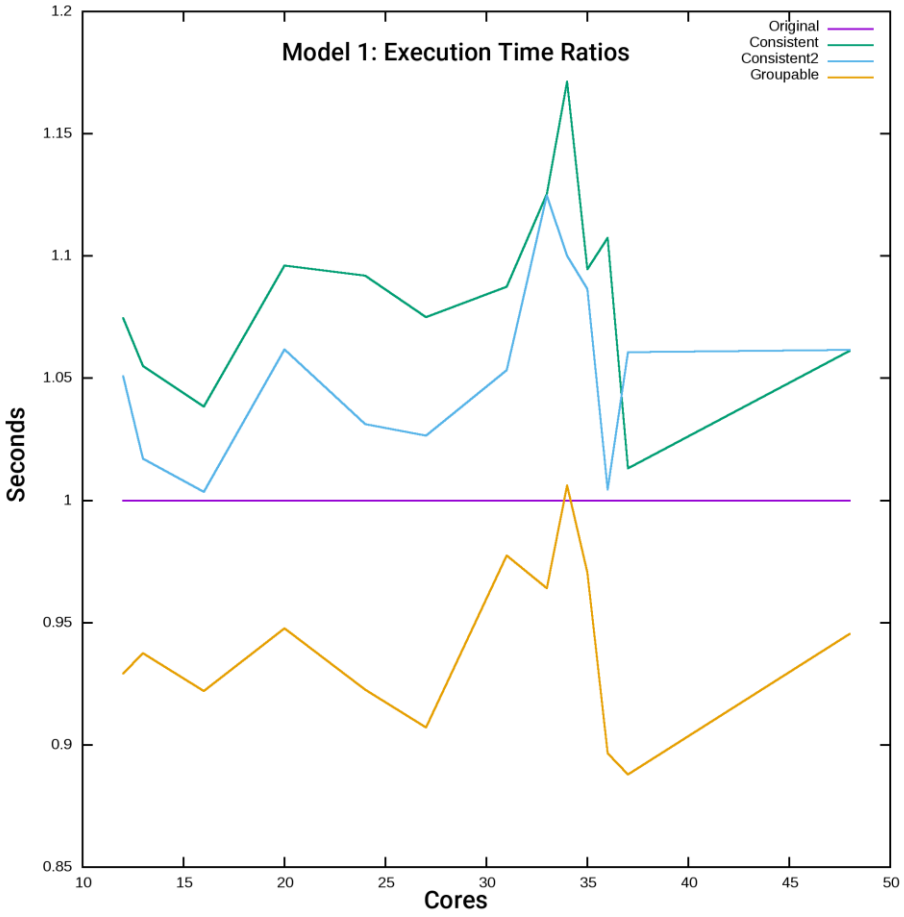
Each model was run on 12, 13, 16, 20, 24, 27, 31, 33, 34, 35, 36, 37 and 48 cores. Each was run on each core count in 4 different variations. The timings marked “Original” are from running the model with no special options. The “Consistent” execution times are using the consistent summation and decision making code described above. The “Consistent2” use the consistent summation and decision making, plus a newer approach to consistent element force summation. Finally, the times marked “Groupable” are the same as “Original” except with all the contact definitions having the “groupable” flag turned on. The nodout files for the Consistent group are all identical, as are the nodout files for the Consistent2 group. The Original and Groupable runs are all different, as expected. The Consistent and Consistent2 groups give different results, due to differences in the summation order for the elements.
What is this Groupable variation, and why is it included? Several years ago, a new version of the contact routines was introduced, dubbed the “groupable” contacts. They combine (or “group”) together the messages for several different contact interfaces, in order to reduce communication and minimize waiting during the contact calculations. Because this version of the contact routines is generally more efficient than the standard versions, the expectation is that they should be adopted over time and supplant the older versions. Consequently, most new developments in MPP contact are done in the groupable contact routines. In particular, turning on the consistency option switches all the contacts to their groupable versions, because the consistency modifications only exist in the groupable version of the routines. Thus, comparing the speed of execution with and without consistency should properly be done by comparing the Consistent (or Consistent2) version with the Groupable version.

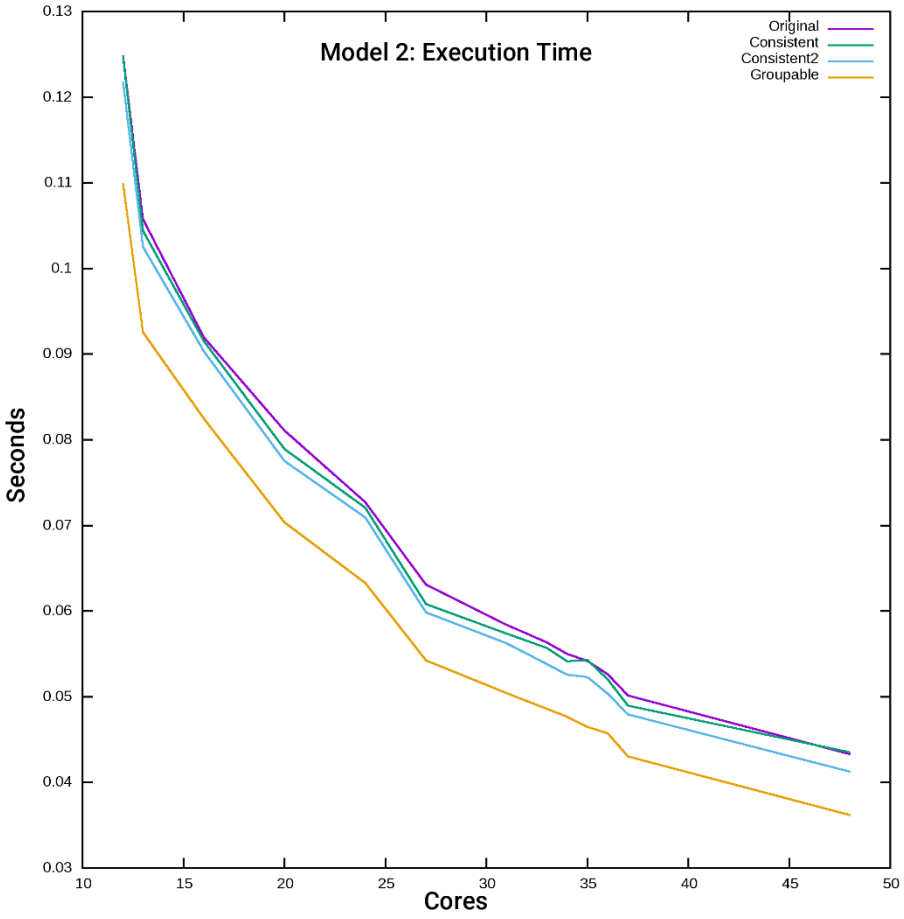
4.4 Timings

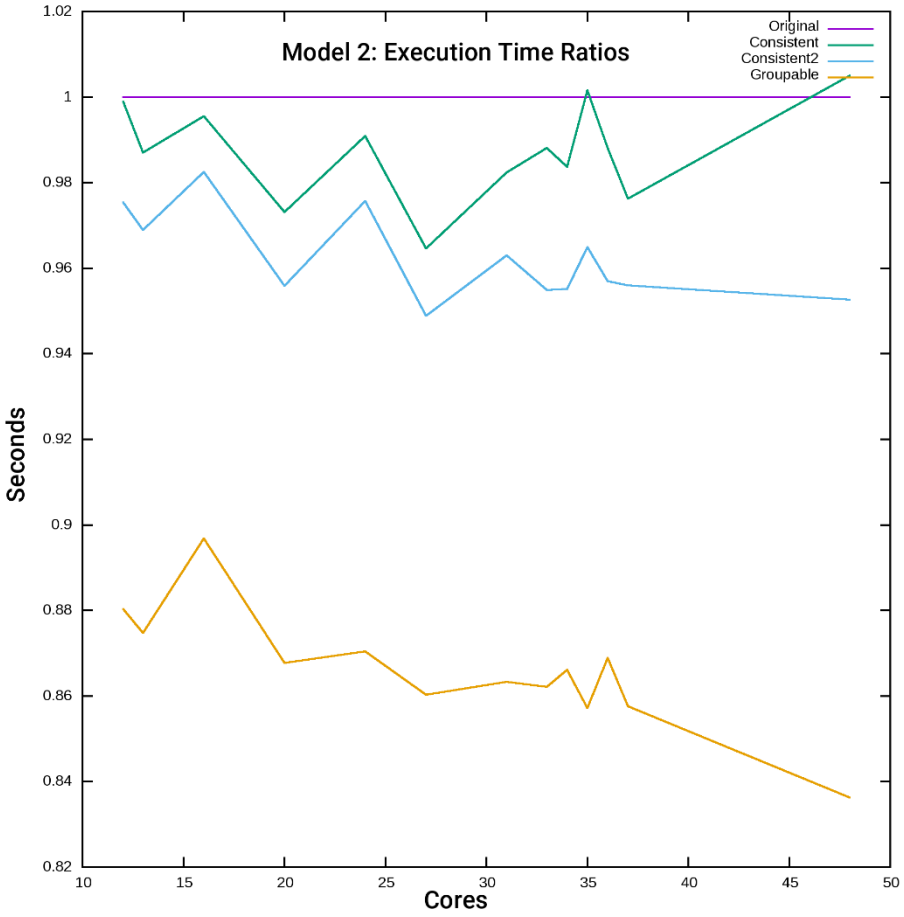
There are a number of ways to compare performance. The most natural would be to just measure the wall clock time for execution. But each variation of the model (Original, Consistent, etc) will run slightly differently and so, in the absence of mass scaling (which model 1 lacks), a different number of cycles. Furthermore, a significant portion of the wall clock time is spent during the adaptivity process which is identical between variations (in MPP, the adaptivity is done by a single core and so is not subject to the kinds of issues we are addressing). We are really only interested in the speed of a single computational cycle. So the timings presented here are computed by taking the total execution time and subtracting out the time spent in initialization and adaptivity. Specifically, using the “Timing information” table from the bottom of the d3hsp file, the execution time is computed from the values in the “Clock” column as: “Totals” - “Keyword Processing” - “MPP Decomposition” - “Initialization”. This value is then divided by the total number of execution cycles. These are the values presented in the “Execution Time” graphs. For the “Execution Time Ratios” graphs, each variant is simply divided by the value for the “Original” run, to give a simple measure of speed relative to the default option.
It must also be noted here that there is some noise expected in the results, and some noise here that we don’t claim to fully understand. The compute nodes used during testing were running no other (known) jobs, but the system as a whole was not dedicated to these timings, so disk I/O and other shared resources might affect the timings. Also, as the core count varies the distribution of cores to sockets and compute nodes also naturally varies, affecting cache and memory performance, as well as

communication patterns.. Also, for the non-consistent variations of the code, the number of elements in the model can vary from run to run due to differences in the adaptivity.









5 Analysis

For these models and these core counts, a few conclusions can be drawn. Turning on groupable contact has a significant impact on performance. For model 1, there is on average a 6.0% speed improvement, and for model 2 a 13.4% speed improvement. It is also clear that the new element consistency routines are faster than the old ones, with about a 2-3% overall speed improvement. The information of primary interest here, however, is a comparison of the Groupable results and the Consistent2 results. For model 1 this difference is 11.2%, and for model 2 it is 9.6%. These numbers were obtained by simply averaging the values shown in the “per cycle ratios” graphs, and taking their difference.

6 Summary

The issue of getting different results with different numbers of cores or different decompositions is complex and not easily resolved. But it can be done. The admittedly limited testing done for this study indicates that the run time performance penalty is about 10% with the current techniques.

7 Appendix

The various options mentioned above are enabled as follows. To turn on the groupable contact, the line “contact { groupable 1 }” was added to the MPP pfile. To turn on consistency, “general { consistent }” was added to the pfile. For the Consistent2 runs, “general { consistent }” is added to the pfile, along with “para=2” being added to the LS-DYNA command line.
The following is a complete list of the control cards that appear in these two test models. No claim is made that every option on each of these cards is currently supported for consistency, nor even that every combination of these cards is supported, nor that any other control card or combination of cards is supported. On the other hand, no doubt many other cards (most materials, for example) are expected to work without problems. This list is presented here simply for reference. We would like to encourage you to try your own models with the consistency option, and let us know how it works for you.

```
*BOUNDARY_PRESCRIBED_MOTION_RIGID
*CONSTRAINED_RIGID_BODIES
*CONTACT_DRAWBEAD
*CONTACT_FORMING_NODES_TO_SURFACE_SMOOTH
*CONTACT_FORMING_ONE_WAY_SURFACE_TO_SURFACE
*CONTACT_FORMING_ONE_WAY_SURFACE_TO_SURFACE_SMOOTH
*CONTROL_ACCURACY
*CONTROL_ADAPSTEP
*CONTROL_ADAPTIVE
*CONTROL_BULK_VISCOSITY
*CONTROL_CONTACT
*CONTROL_ENERGY
*CONTROL_FORMING_AUTOPOSION_PARAMETER
*CONTROL_FORMING_AUTOPOSITION_PARAMETER_SET
*CONTROL_HOURGLASS
*CONTROL_OUTPUT
*CONTROL_PARALLEL
*CONTROL_RIGID
*CONTROL_SHELL
*CONTROL_TERMINATION
*CONTROL_TIMESTEP
*DATABASE_ABSTAT
*DATABASE_BINARY_D3PLOT
*DATABASE_BNDOUT
*DATABASE_EXTENT_BINARY
*DATABASE_GLSTAT
*DATABASE_HISTORY_NODE
*DATABASE_MATSUM
*DATABASE_NODOUT
*DATABASE_RBDOUT
*DATABASE_RCFORC
*DATABASE_SLEOUT
*DEFINE_BOX_DRAWBEAD_TITLE
*DEFINE_CURVE
*DEFINE_CURVE_TITLE
*ELEMENT_BEAM
*ELEMENT_SHELL*ELEMENT_SHELL_THICKNESS
*END
*INCLUDE
*INCLUDE_AUTO_OFFSET
*INCLUDE_PATH_RELATIVE
*INITIAL_STRESS_SHELL
*INTERFACE_SPRINGBACK_LSDYNA
*KEYWORD
*LOAD_SURFACE_STRESS_SET
*MAT_3-PARAMETER_BARLAT
*MAT_KINEMATIC_HARDENING_TRANSVERSELY_ANISOTROPIC
*MAT_RIGID
*NODE
*PARAMETER
*PARAMETER_EXPRESSION
*PART
*PART_MOVE
*SECTION_BEAM_TITLE
*SECTION_SHELL
*SET_PART_LIST
*TITLE
```