A Tutorial on How to Use Implicit LS-DYNA®

Roger Grimes
Livermore Software Technology Corporation

Abstract

This talk will focus on the issues of using the Implicit features of LS-DYNA. Implicit has a profoundly different footprint regarding the computational resources required than an explicit simulation. This talk will especially focus on the management of those computational resources, especially for distributed memory computations. This talk will also include an overview of techniques for debugging models for use by Implicit.

Understanding the Computational Environment

Implicit has a fundamentally different set of computational requirements than the standard explicit computational approach in LS-DYNA. This difference is due to the requirement of assembling and solving the underlying linear system K * u = f for the displacements u. To best use Implicit LS-DYNA on your computational environment it is best to start with understanding your computational environment. This paper will assume that you are using a distributed memory computational cluster and are using the MPP Hybrid executable of LS-DYNA. But most of the discussion will apply to the non-hybrid MPP executable and the SMP executable. Furthermore we will assume that the computation is being performed with the double precision version of LS-DYNA and all storage for integer and real numbers is 8 bytes.

Most large scale LS-DYNA modeling is being performed on distributed memory computational clusters. These clusters are made up of some number of computational nodes which are usually similar in make-up. A computational node will have some number of sockets each holding some number of cores (aka cpus), the product is the maximum number of processes for a compute node; some amount of RAM (memory); and a local I/O system. A user needs to know the number associated with these resources to properly manage them to get the best performance for LS-DYNA Implicit.

The user also needs to understand that the MPP LS-DYNA executable can be spread across the computational nodes using a round robin assignment to have the same number of processes per computational node. For this paper the total number of processes is P. The number of processes that will execute on a single computational node is P_ℓ which should be limited by the number of cores on that compute node.

Memory Management

Past experience with LS-DYNA Implicit has shown that the amount of memory available to each MPP processes should be at least (440/P + 75)*N words where P is the total number of processes and N is global number of nodes in the model.

This memory requirement has a serial memory restriction that is a limiting factor. This is an algorithmic issue associated with the use of direct sparse linear equations solution technology and affects every implicit finite element software package. LSTC is actively researching and developing a new linear algebra technology to avoid this serial memory restriction. Until that technology is available this serial memory restriction will be a limiting factor in implicit. And memory per computational node will be the restriction.

Remember that the memory on a computational node not only has to hold the LS-DYNA work array whose length is given by the memory=xxxxM on the command line but also dynamic memory for MPP core utilities including Metis, the matrix assembly package and the LS-DYNA executable. It is recommended that the memory setting on the LS-DYNA command line uses no more than 75% of the memory available to that process. The authors always use the following memory specifications on 16 and 48 Gbytes per node clusters

16 Gbytes		48 Gbytes	
P_ℓ	Memory	P_ℓ	Memory
1	1500M	1	4000M
2	800M	2	2000M
4	400M	4	1000M
8	200M	8	500M

You can get these recommended specifications by first dividing the amount of memory for a computational node by 8 (to convert from bytes to words), multiplying by 0.75, and then dividing by P_ℓ, the number of processes running on a single compute node.

It is strongly recommended that user does NOT use the memory2 specification. The imbalance of memory between processes will cause an imbalance in the intensive computation during the linear algebra phase. It is also strongly recommended that the user does NOT use the AUTOMEMORY feature. AUTOMEMORY tells LS-DYNA just to allocate all of the memory it wants to. Implicit linear algebra needs to know how much memory it can use and when to switch to using disk for alternate storage.

Now the user has to do some work. They have to know the amount of available memory (MEM) and the global number of nodes in their model (N). They need to find the smallest P and P_ℓ such that $(440/P+75)*N < (0.75*MEM/8)/P_\ell$. The easiest approach is to start with 1 process per computational node ($P_\ell = 1$). Then select a small number such as 8 for P. This choice of P_ℓ keeps all of the available memory intact for the LS-DYNA executable to overcome the serial memory bottleneck.

To recover the computational power of each computational node it is recommended that you use the MPP Hybrid executable for LS-DYNA. This executable uses the MPI distributed memory computational paradigm for the processes, P, and then uses shared memory parallelism within the MPI process to recover the otherwise idle computational power of the hardware. For a computational node with 8 cores and only one MPI process for that computational node then use ncpu=+/-8 on the command line to have, as appropriate, up to 8 SMP threads operating in the context of that MPI process. If two MPI processes are running on that computational node use ncpu=+/-4.

Once you get the first execution to run then the user can try to reduce computational time by experimenting with process assignment. One can increase the number of processes per computational node, P_ℓ, and correspondingly reducing the number of SMP threads. Of course, another approach is to increase P, the number of computational nodes utilized. But before that experiment the user should understand how memory is being used and what are the options for increasing P and P_ℓ. During this experimental phase the user can reduce the computational time for the overall simulation by restricting the simulation to a single time step and, perhaps, restricting the job to a linear simulation with setting NSOLVR=1 (first field of *CONTROL_IMPLICIT_SOLUTION).

Understanding the Utilization of Memory

LS-DYNA Implicit provides information about how memory is being used, especially when LPRINT=2 (2nd field of *CONTROL_IMPLICIT_SOLVER) is utilized, to the mesxxxx files. Usually perusal of file mes0000 is sufficient although the user should check all of these files.

To begin with Implicit declares the start and end of static memory allocation for the non-linear algebra storage for implicit from the LS-DYNA work array with the output to the file mesxxxx file of

```
Start of implicit storage allocation - locend =    293181565
```

and

```
End   of implicit storage allocation - locend =   2708332159
```

This is the storage required for implicit, especially for the nonlinear solution process. It is allocated after explicit has finished allocating storage and prior to the start of the linear algebra requirements.

The next phase of Implicit that affects memory is the assembly of constraints, the assembly of the stiffness matrix, and the application of the constraints to the stiffness matrix. This is performed by a collection of software called LCPACK (Linear Constraint Package). This is performed using dynamic memory. Dynamic memory is allocated from some of the 25% of RAM not allocated to static memory with the memory= specification.

When this phase is completed the matrix is extracted into static memory. You will see this message for the expansion of static memory to hold the matrix data.

```
 expanding   memory to   2838019839 implicit matrix storage
```

At this point the linear equation solution phase starts with the first step being the symbolic processing. This is where the serial memory bottleneck for the symbolic processing will show up. Look for this block of output

```
ptr to start of wrkspc   =   2838019839
```

and later

```
storage currently in use =     94316328
storage needed           =   1011902889
factor speedup           =   7.5168E+00
solve speedup            =   7.6665E+00
```

The value of the memory pointer (ptr above) should be very close to the end of the implicit storage as the matrix storage is reused by the linear equation solver. The value for the pointer plus the amount for storage needed is the serial memory bottleneck for symbolic processing, here 3.85 Gwords. If this amount is more than the specification of memory on the command line the job will fail. If this amount is near the command line specification then the user cannot increase the number of processes per computational node nor decrease the amount of memory per process. If this amount is half of the command line specification then it is possible to double the number of processes per computational node, P_ℓ, and correspondingly reduce the memory specification and number of SMP threads (ncpu). On this run, using a computer with 96 Gbytes and memory=8000M the required memory is less than half so it would be possible to increase P_ℓ from 1 to 2. As we see later this may or may not be a good idea.

The symbolic processing examines the computational structure resulting from the symbolic processing and determines the amount of distributed memory parallel speed-up is available. This computation is limited to the actual number of processes being used for this run. If the speed-up number for the factorization is much smaller than the number of processes being used then the user should not increase the number of processes. All of the available performance gain has probably been achieved.

During the process of solving the system of linear equations there will be output of the form

```
in-core numerical storg 1 =     159821.76 Mw
out-of-core num.  storg 1 =      10936.73 Mw
expanding   memory to 8000000000 linear eqn. solver
```

With $P = 8$ and $P_\ell = 1$ and memory=8000M this job failed because there was not enough memory for an out-of-core factorization. So the first step is to increase P to get the out-of-core storage below 8000M. Of course, you can always request a memory increase for your computer. Like that is going to work.

If the memory expansion is less than the in-core required storage the factorization will be stored on disk. This will tend to increase the overall computational time. Storage of the factorization on disk may well be required to get the problem solved. Increasing the number of processes per computational node may cause a factorization that was stored in memory to be stored on disk. This will probably cause a degradation of performance.

If the factorization is in memory and only half of the static memory is being used then the user can probably double the number of processes per computational node, P_ℓ, and correspondingly decrease both the specification for memory and ncpu.

If the factorization is out-of-core (as many large simulations will be) it is best if the factorization is stored on the file system local to the compute nodes. It is almost never a good idea to use central file systems for these scratch files. Include in p=pfile the line

```
dir { local /my/local/disk/tmp rmlocal transfer_files transfer_scr }
```

Model Debugging

After memory management the most difficult part of using Implicit is debugging the actual model. Due to the requirement of solving the system of linear equations the stiffness matrix, after constraints have been applied, has to be non-singular. Over the years LSTC has added a number of automatic responses to removing such things as nodes with no stiffness and many types of redundant constraints. Alas, like the Maginot Line, innovative users can easily get past these defense mechanisms.

Two debugging tools are immediately available to the users. The first is setting LPRINT=3. This activates extra checking of each and every elemental stiffness matrix. Elemental stiffness matrices with negative eigenvalues, too many zero eigenvalues, and not enough positive eigenvalues are all flagged. If the user sees these warnings they should examine the associated elemental and material definitions.

The most powerful tool is the eigenvalue computations. Inserting

```
*CONTROL_IMPLICIT_EIGENVALUE
10
```

activates the eigenvalue computation which provides a very powerful tool. If rigid body modes are found they need to be removed from the model or a feature such as Implicit Dynamics or Inertia Relief should be used to remove them. A common problem is parts of the model that are not connected to the rest of the model due to a missing contact definition or constraint.

In some cases, the number of rigid body modes is so large the eigensolver fails. Look for the first occurrence of

```
Shift (in cycles)     :  8.8037D-04
No. modes to left     :         223
```

This indicates that there are 223 modes to the left of 8.8037D-4. This indicates a huge cluster of eigenvalues at zero. The eigensolver will not be able to resolve this huge cluster to determine the smallest 10. A common cause of this huge cluster of eigenvalues is the inclusion of elements or parts not tied to the rest of the model. The author saw one test case with 120 beam elements that were not connected with the rest of the model. This added 720 near zero eigenvalues. With the removal of the spurious beam elements the model ran just fine.

Another common cause of a huge cluster of zero eigenvalues is the common practice of wrapping a solid part with shell elements. This is usually to get surface stresses but sometimes is used for the purposes of contact definitions. Such shell elements tend to be very thin to add very little mass or stiffness to the model. So a common shell element would have no stiffness for the rotational dofs that are not present with the solid elements. This adds 3 near zero eigenvalues for each node attached to the shell elements. So it is required that these very thin shell elements use the membrane element formulation so as NOT to add the unnecessary rotational dofs.

Another aspect of the debug checking activated with LPRINT=3 is a linear algebra based examination of the stiffness matrix looking for separability. Due to algorithmic restrictions this is only available for the SMP version. This is the property that the overall stiffness matrix is actually made up of more than 1 independent problems or components. The user could use the SMP just to check for separable components if the MPP eigensolver fails. If the problem is separable it is almost always coupled with elements and parts not attached to the rest of the model. A report is generated on these components. Search the messag file for

```
   This implicit model has separable components
   this may lead to model singularities
   Number of components = 8873
   the first few smallest 6 components
   the list is limited to at most 100 rows for each component
        Component     size        node    dof
        ---------     ----        ----    ---
                1       72      220427    x-t
                                220427    y-t
                                220427    z-t
```

This says that the first (sorted by size) has 72 dofs in the component including the x-t, y-t, and z-t dofs for node 220427. The report lists the nodal information for the smallest components. This is another approach for finding parts that are not connected to the remainder of the model that would be causing singularities.

The linear algebra software will track the numerical stability of the numerical factorization of the stiffness matrix. If you see

```
 *** Warning: At least one small pivot detected during factorization
```

then you probably have a near singular matrix. It is recommend that you compute the near zero eigenvalues as discussed above. It may be that the singularity develops during the simulation. One could consider using the intermittent eigenvalue feature to compute the eigenvalues at the simulation time where the warning appears.

Another technique is to dump the nonlinear search directions to the d3iter database. You can then use LSPrePost to examine the search directions. If the search direction is going off in a strange direction it is probably due to a singularity in that direction. You can turn on the output to the d3iter database using D3ITCTL, the 6th field on the 2nd line of *CONTROL_IMPLICIT_SOLUTION.

A Parthian Shot

Implicit prefers modeling with constraints rather than the penalty treatment. For robust solution of the implicit problems it is recommended that all use of

*CONTACT_TIED_xxxxxx_OFFSET

or

*CONTACT_TIED_xxxxxx_BEAM_OFFSET

is switched to

*CONTACT_TIED_xxxxxx_CONSTRAINED_OFFSET.