Characterizing LS-DYNA® Implicit performance on SGI® Systems using SGI MPInside MPI profiling tool

Dr. Olivier Schreiber, Tony DeVarco, James Custer, Scott Shaw
SGI

Abstract

SGI delivers a unified compute, storage and remote visualization solution to manufacturing customers reducing overall system management requirements and costs. LS-DYNA integrates several solvers into a single code base.
The implicit solver is studied for better matching with the multiple computer architectures available from SGI, namely, Multi-node Distributed Memory Processor clusters and Shared Memory Processor servers, both of which are capable of running in Shared Memory Parallelism (SMP), Distributed Memory Parallelism (DMP) and their combination (Hybrid) mode.
The MPI analysis tool used is SGI MPInside featuring customary communication profiling and "on the fly" modeling to predict potential performance benefits of the different upgrades available from the latest Intel® Xeon® CPU, interconnect and its middleware, MPI library, and the underlying LS-DYNA source code. Profile-guided MPIplace component is used to minimize inter rank transfer times.

1.0 Versions Used

LS-DYNA/MPP R8.1 double precision HYBRID

ls-dyna_hyb_d_r8_1_105897_x64_suse111_ifort131_avx2_sgimpt203

OS: 2.6.32.13-0.4

Fortran : Intel(R) Fortran Intel(R) 64 Compiler XE for applications running on Intel(R) 64,Version 13.1.3.192 Build 20130607
C : Intel(R) C Version 8.1 Build 20050518
MPI : SGI MPT 2.03

2.0 Benchmarks Description

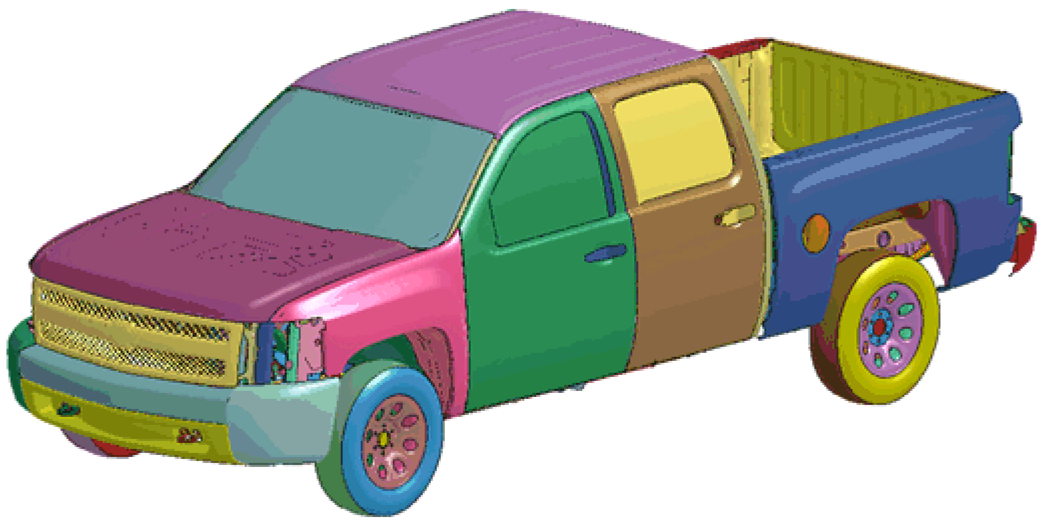
LS-DYNA/MPP/Implicit [1], [2] is being used on two sets of benchmarks.

2.1 AWE

Solid element model from Atomic Weapons Establishment benchmark suite as used in Refs [1] and [2] available in meshes of 100K, 500K, 1M, 1.5M, 2M, 4M, up to 20M nodes. The model represents 6 nested cylinders held together with surface to surface contact, meshed with single elastic material solid elements. A prescribed motion on the top and a load on the bottom are imposed for one nonlinear implicit time step with two factorizations, two solves and four force computations. The figure illustrates a 921,600 solid elements, 1,014,751 nodes problem leading to a 3,034,944 order linear algebra system.

2.2 Silverado

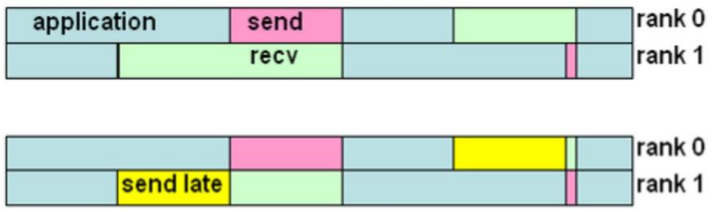
This National Center for Automotive Crash Chevrolet Silverado model original with .85M nodes [3], was refined by LSTC to have 1.7M, 3.4M and 10.5M nodes [2] to be used in statics, modal and nonlinear solver runs.



3.0 MPInside

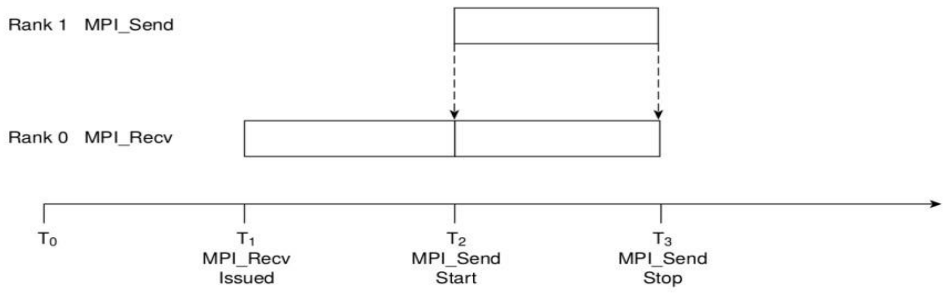
3.1 MPInside Introduction

MPInside is available with SGI MPI – High Performance MPI Environment along with MPIplace, an MPI profiling tool [4]. It can provide information to help MPI application developers optimize their application by finding out such instances where, for example, MPI Send/Receive pairs are not executed synchronously.



3.2 MPInside Terminology

MPI communication consists of non-necessarily synchronized Sends and Receives.'Send Late Time' (SLT) is defined as the delay between one process's MPI_Recv call and the process' MPI_Send call where the message is supposed to come from. The time it takes for data to actually be transferred is called Transfer Time (Tt). The sum of SLT and Tt is defined as Function time (FT). 'Receive Late Time' is defined as the delay between an MPI_ send blocked call and its remote receiver process' MPI_Recv eventual call.



Function Waiting Time FWT in above example is equal to the FT time because MPI_Recv is a blocking function but in case of a non-blocking function such as MPI_Irecv FWT would be the time of the MPI_Wait function that "finished" the request (in the MPI sense) corresponding to this function.

Non-blocking receives (MPI_Irecv, MPI_Recv_init) allow the application to do useful computations during Send Late Time as well. It is possible that this Send Late Time is still going by the time the application attempts to complete the non-blocking receive with MPI_Wait, MPI_Test or similar. In this case, the Send Late Time overlapped with computation is not counted. Only Send Late Time that is visible to the application as time spent blocked or delayed completing the request in a Wait or Test is counted.

4.1 MPInside Usage

4.1.1 MPInside Command

MPInside command doesn't require any change in the application or any re-link and only needs to be inserted as an argument of the mpirun or mpiexe_mpt command prepended to the target application executable.

4.1.2 MPInside Output

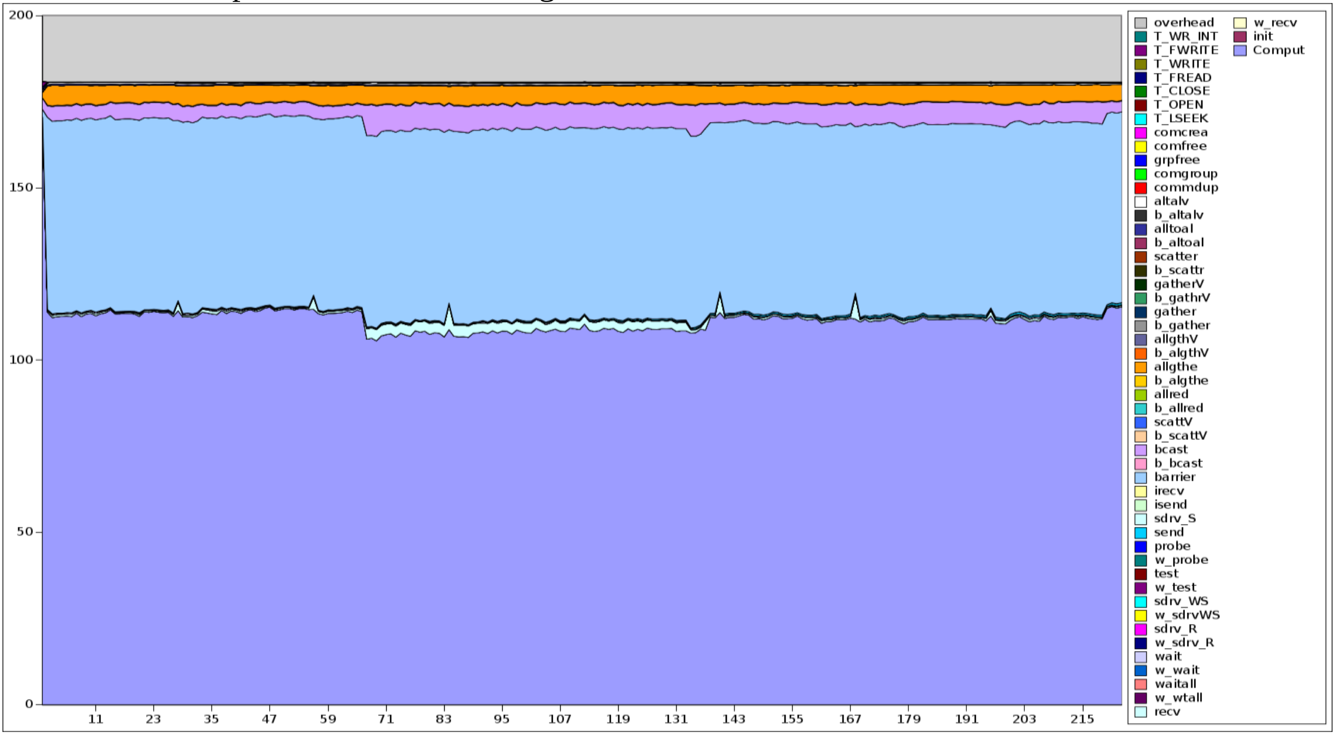
At end of run, five tables with one entry per rank over multiple columns of MPI functions labeled in abbreviated form are output to ASCII files.

4.1.2.1 Timing Table Format:

```
>>>> Communication time totals (s) 0 1<<<<
CPU    Compute       MPI_Init     w_MPI_Recv     Recv           w_MPI_Waitall  Waitall

0      868.484133    0.000232     0              322.801183     0              0
1      654.365446    0.000213     0              326.385665     0              0.348279
2      645.987836    0.000189     0              337.04429      0              0.270488
3      634.765585    0.000189     0              339.249457     0              0
4      648.41097     0.000214     0              333.377204     0              0
5      657.331095    0.000185     0              322.48984      0              0
```

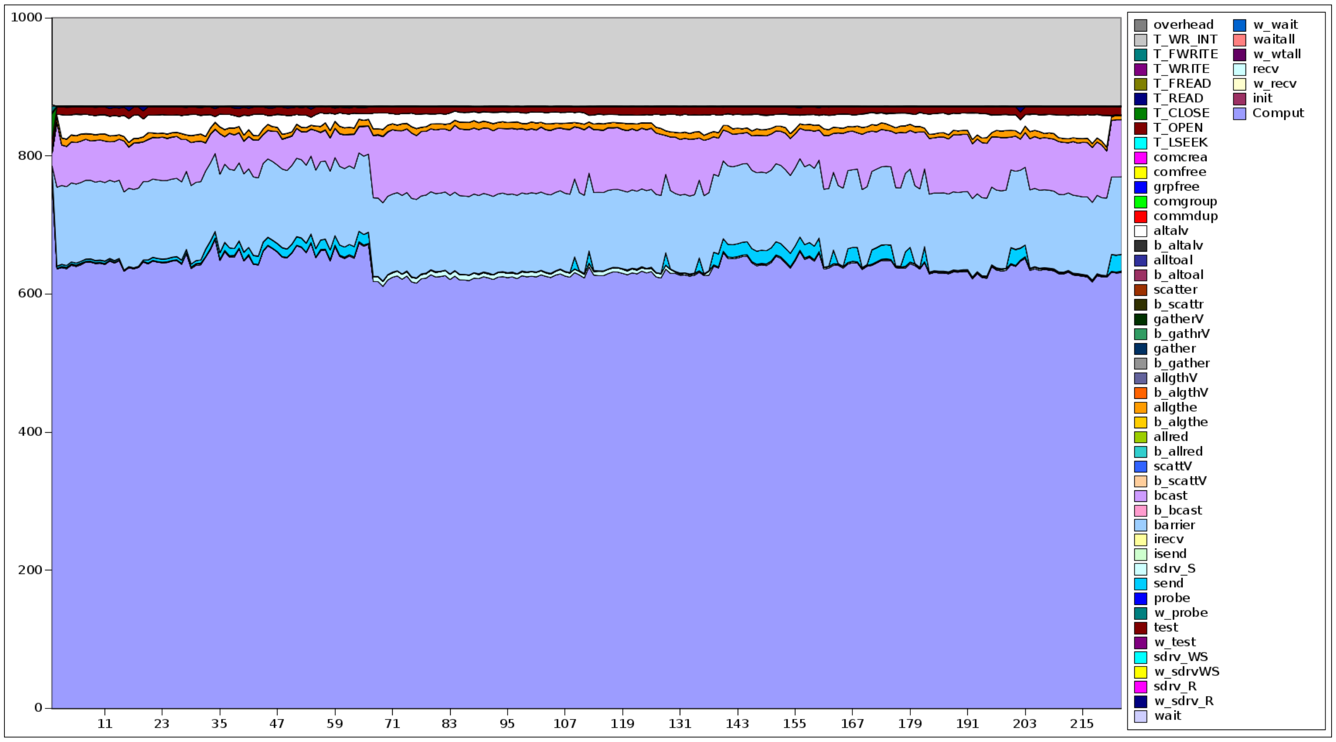
4.1.2.1.1 Timing table charted for linear implicit Silverado 4M solution

Main times: compute, barrier, bcast, allgthe



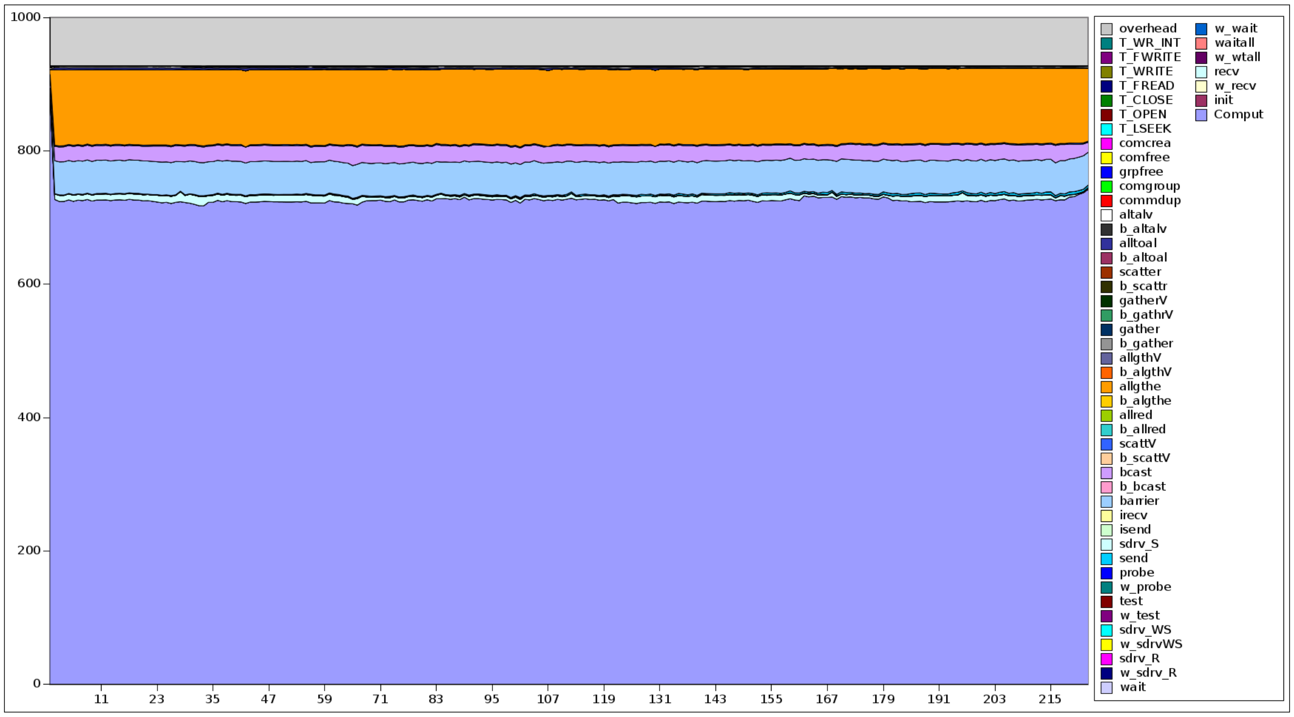
4.1.2.1.2 Timing table charted for eigenvalue implicit Silverado solution

Main: compute, barrier, bcast, allgthe, altalv and better compute to communication ratio.



4.1.2.1.3 Timing table charted for nonlinear implicit Silverado solution

Main times: compute, barrier, bcast, allgthe with yet better compute to communication ratio.



4.1.2.2 Bytes Sent Table format:

```
>>>> Bytes sent <<<<

CPU    Compute       MPI_Init     w_MPI_Recv     Recv           w_MPI_Waitall  Waitall

0      ------        0            0              0              0              0
1      ------        0            0              0              0              0
2      ------        0            0              0              0              0
3      ------        0            0              0              0              0
4      ------        0            0              0              0              0
```

4.1.2.3 Number of "Send" Calls Table format:

```
>>>> Calls sending data <<<<

CPU    Compute       MPI_Init     w_MPI_Recv     Recv           w_MPI_Waitall  Waitall

0      ------        1            0              0              0              0
1      ------        1            0              0              0              239981
2      ------        1            0              0              0              239981
3      ------        1            0              0              0              0
4      ------        1            0              0              0              0
```

4.1.2.4 Bytes Received Table format:

```
>>>> Bytes received <<<<
CPU    Compute       MPI_Init     w_MPI_Recv     Recv           w_MPI_Waitall  Waitall

0      ------        0            0              28953401700    0              0
```

```
1      ------        0            0             28939575772    0             0
2      ------        0            0             20038927680    0             0
3      ------        0            0             19903973196    0             0
4      ------        0            0             13668688376    0             0
```

4.1.2.5 Number of “Recv” Calls Table format:

```
>>>> Calls receiving data <<<<

CPU    Compute       MPI_Init     w_MPI_Recv    Recv           w_MPI_Waitall  Waitall

0      ------        0            0             14208346       0              0
1      ------        0            0             13966079       0              239981
2      ------        0            0             14222841       0              239981
3      ------        0            0             17384042       0              239981
4      ------        0            0             15638825       0              239981
```

4.1.2.6 Other Outputs

Function calls and their timings in histogram format in terms of message sizes for following quantities:

4.1.2.6.1 Number of requests distribution:

```
>>> Rank 0 Sizes distribution <<<

Sizes       Recv          Send          Isend         Irecv

65536       0             106           0             48558469
32768       0             38            0             0
16384       0             22            4356          0
[...]
128         489344        1199947       248494        1199934
64          1208805       1679905       245879        719961
32          487218        720068        2531          239989
0           3616833       3927          3636521 0
```

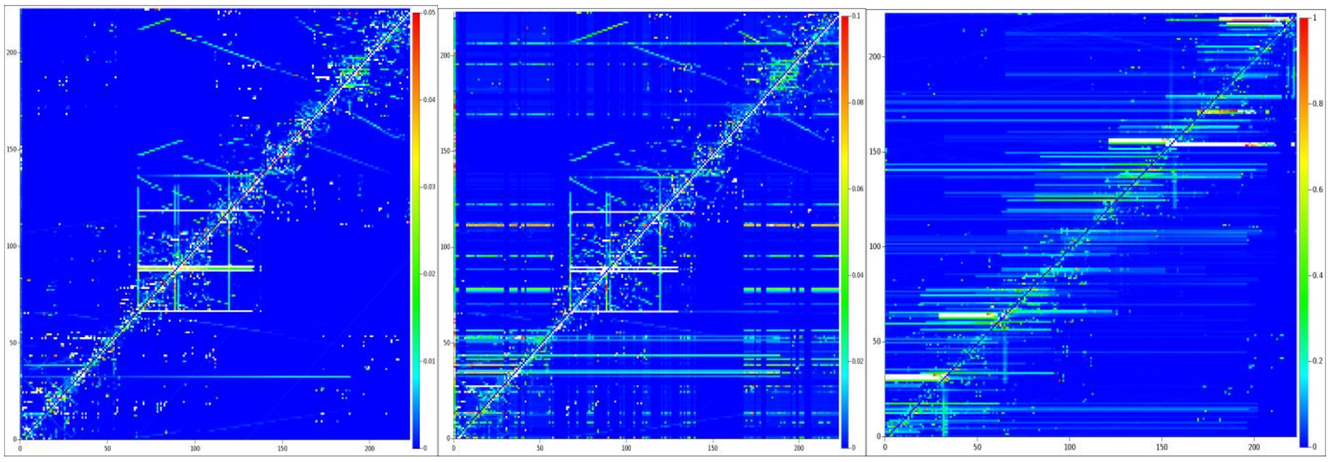
4.1.2.6.2 Times Distribution:

```
>>> Rank 0 Size distribution times<<<

Sizes       Recv          Send          Isend         Irecv

65536       0             0.357941      0             30.856745
32768       0             0.001294      0             0
16384       0             0.000713      0.002428      0
8192        16.060919     1.945468      0.005868      0
```

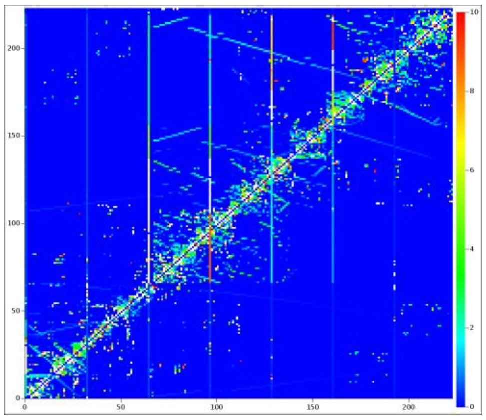
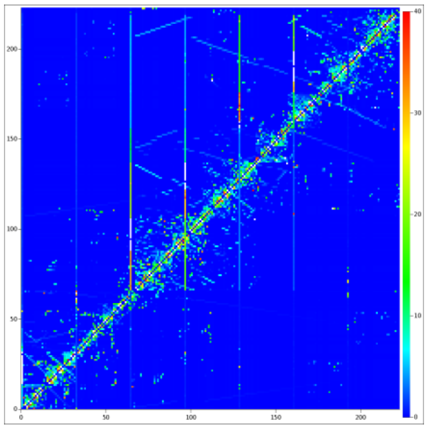
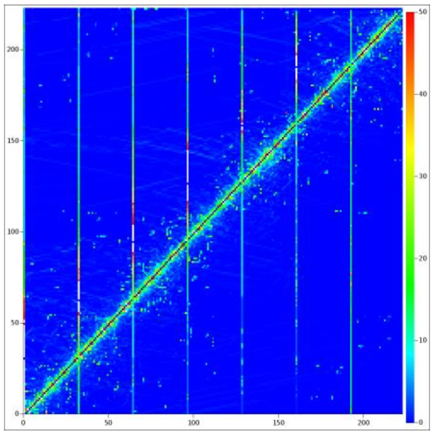
4.1.2.7 Transfer Matrices

One row and one column are also produced per rank for these three quantities:

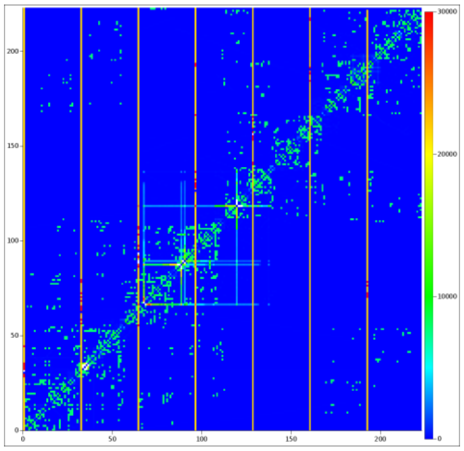
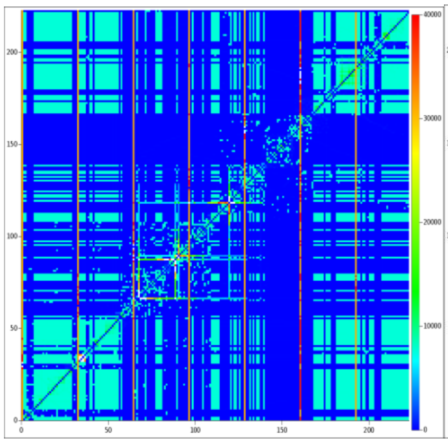
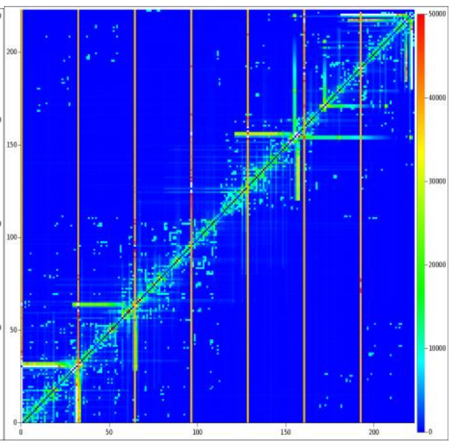
- TIME(i,j): the aggregate time rank “i” spent receiving data from rank “j” shown for example by linear 4M, eigenvalue 4M and nonlinear 2M benchmark charts respectively:



- SIZE(i,j): the amount of data transferred from "i" to "j" shown for example by linear 4M, eigenvalue 4M and nonlinear 2M benchmark charts respectively:

- REQUEST(i,j): the number of calls involved in these transfers shown for example by linear 4M, eigenvalue 4M and nonlinear 2M benchmark charts respectively:

4.1.3 Information Handled by MPInside

4.1.3.1 General

MPInside reports the number of bytes physically transferred, not the size specified on the receive side.

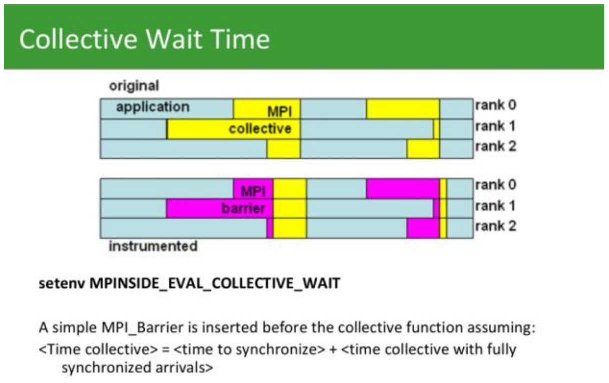
For collective operations such as MPI_Bcast or MPI_Alltoall, transfers are assigned as a send for the root of the broadcast and as a receive for the other ranks participating in the operation.

Sizes reported are computed as buffer size multiplied by number of ranks participating in the function.

“Compute” time as measured by MPInside is the time that a given rank spent that wasn’t attributable to a profiled MPI call including I/O time or separately if MPINSIDE_SHOW_READ_WRITE is set. In that case, number of characters and number of direct calls to libc I/O functions read(), write, open, and read or to MPI_File_xxx MPI I/O functions such as MPI_File_read_at() are reported in the same tables.

Non-communication-related times like I/O or system wait can also be captured by integrating with open source *perf* and *oprofile* or proprietary Intel® VTune™ profiling tools to drill further down into Compute time.

As mentioned earlier, a rank can be blocked on an MPI call waiting for some other rank to catch up. This is t h e case for collective operations such as MPI_Allreduce, where a fraction of the time in these MPI collective functions is spent waiting for the last rank to reach the rendezvous point. To evaluate the cost of these timing misalignments, a call to MPI_Barrier is inserted before each MPI collective to synchronize all ranks, and record its elapsed time thereby measuring the collective operation wait time only. The time shown in the subsequent MPI collective is about the physical transfer of data and its processing. This reporting is activated by setting MPINSIDE_EVAL_COLLECTIVE_WAIT.



Collective Wait Time

setenv MPINSIDE_EVAL_COLLECTIVE_WAIT

A simple MPI_Barrier is inserted before the collective function assuming:
<Time collective> = <time to synchronize> + <time collective with fully synchronized arrivals>

In the data tables and histograms, the column "b_xxx" will give the MPI_barrier time of the corresponding "xxx" MPI collective function and "xxx" column will show the remainder time.

For non-collective operations, setting MPINSIDE_EVAL_SLT directs MPInside to measure the time for all send calls that are late (SLT) with respect to Recv-Wait events. Such time will be labeled w_xxx in the tables where xxx could be MPI_Wait or MPI_Recv. It cannot be MPI_Irecv, because the Send late time, if any, will be, for this last function, accounted in an MPI_Wait-like function.

If neither profiling mode is enabled, (basic mode), times are shown as being spent by their respective MPI call.

If Collective Wait and SLT modes are enabled, time spent in MPI calls is subdivided into Transfer Time (Tt), and wait time. The latter is due to computational load imbalance or OS-related disturbance.

For MPI_Send or MPI_Isend/MPI_Wait couplets, receive-late time accounts for "late" receivers. With sufficient buffers, their impact can be minimized.

On the other hand, for MPI_Recv or MPI_Irecv/MPI_Wait couplets wait time is non-zero when matching sender is late (Send Late Time). This wait time cannot be avoided with any kind of buffering and hence is more important to monitor.

To summarize:

- All times in w_MPI_Recv, b_Bcast and b_Allreduce columns are wait times and all times in Recv, Bcast, and Allreduce columns are physical transfer times.

- Total elapsed time is the sum of the "Compute" column and all the MPI columns.

4.1.3.2 Name convention for MPI functions

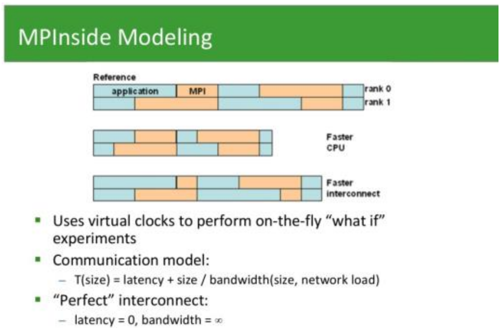
b_<Collective_function>: Artificial MPI_Barrier inserted before the collective function if MPINSIDE_EVAL_COLLECTIVE_WAIT set. Total time for collective function is b_<Collective_function> + <Collective_function>.

w_<receive_or_wait_func>: Artificial wait function accounting for time by which Sends were late with respect to matching MPI_Recv or MPI_Wait.

4.1.3.3 Further Capabilities

Instead of being measured, MPInside reports are also available with communication modeling the hypothetical 'Perfect Interconnect'. This asymptotic value can tell if enhancing communication hardware or library is worthwhile for a particular application and run case.

A 'Perfect' profile will not eliminate times for Recv and Bcast where what remains is overhead time in libmpi.so for MPI call argument passing, pushing and popping functions on the stack, allocating and deallocating memory, but more importantly, waiting on one or more ranks on the other end of the Recv or Bcast to catch up. These times might be similar to Send Late or Collective Wait times, but might be shorter because zero transfer times sometimes lead to better rank synchronization.



MPInside Modeling

- Uses virtual clocks to perform on-the-fly "what if" experiments
- Communication model:
  - T(size) = latency + size / bandwidth(size, network load)
- "Perfect" interconnect:
  - latency = 0, bandwidth = ∞

4.1.4 MPInside Inferences

As alluded to previously, large times in w_MPI_Recv column of MPInside tables correspond to Send Late Time (SLT) situations in Recv. Similarly, large times in b_Bcast column of MPInside tables correspond to synchronization waiting times before Bcast actually start.

Recv and Bcast nonzero times with Perfect Interconnect mode similarly points to waiting on one or more ranks on the other end of the Recv or Bcast to catch up as shown similarly with Send Late or Collective Wait time, but might be shorter if zero transfer time in between compute intervals leads to better synchronization between ranks.

Above three symptoms maybe the result of load imbalances if they correspond to similar irregularities in Compute time.

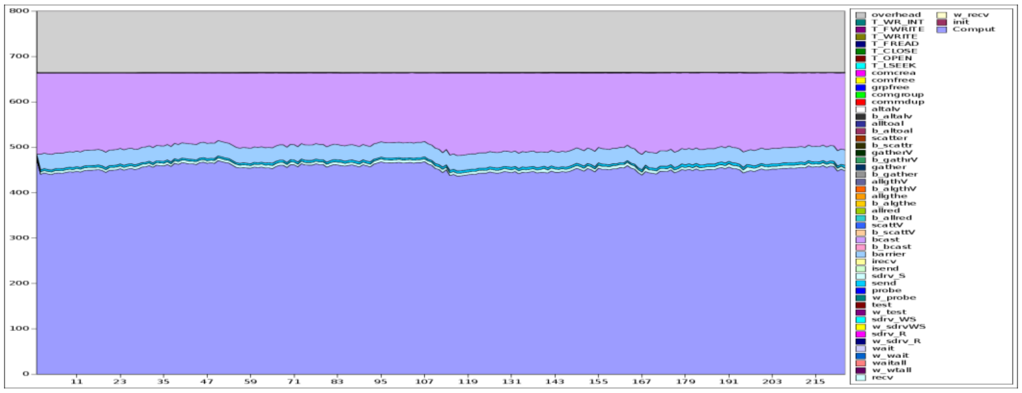
4.1.5 Remedying Load Imbalances

Using MPInside data domain decomposition can be modified by either analysts or LSTC to drive down load imbalances. If it cannot be improved, overlapping communication and computation with MPI_Ibcast (for MPT 2.10 and later) or MPI_Irecv and delaying blocking until work can't proceed without more data will help. Periodical MPI_Test* on the requests that come back from Ibcast or Irecv can further increase overlap. SGI MPI has a separate progress thread that proceeds once a request is initiated. MPI_Test will "kick" the progress engine to make it check for completion again.

4.2 Case Study of AWE 1.5M Benchmark

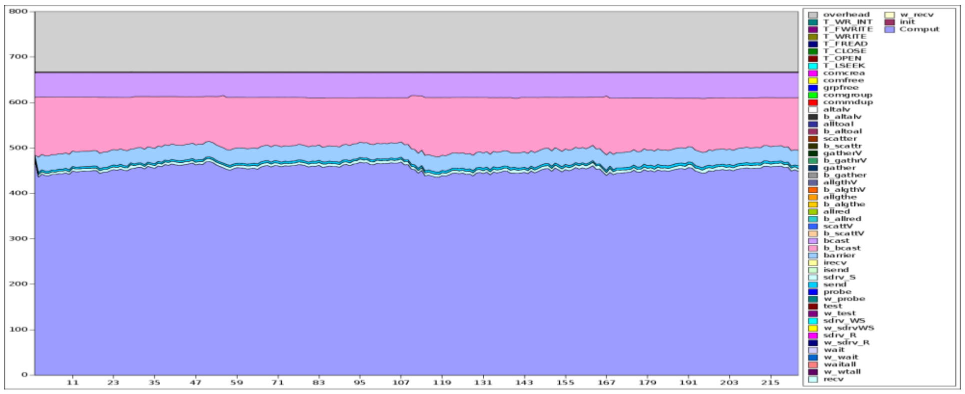
4.2.1 Basic Profiling

The bands shown across the 224 MPI processes are compute, barrier and bcast times:



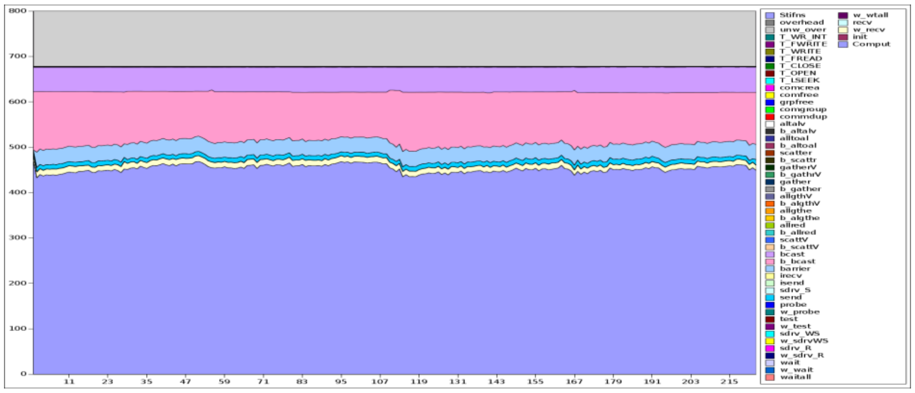
4.2.2 Collective Wait Profiling

Turning on Collectives Wait mode shows that the Bcast calls of previous graph are in fact made up of barrier-like times for ranks to synchronize--the Bcast itself being a third of that time:



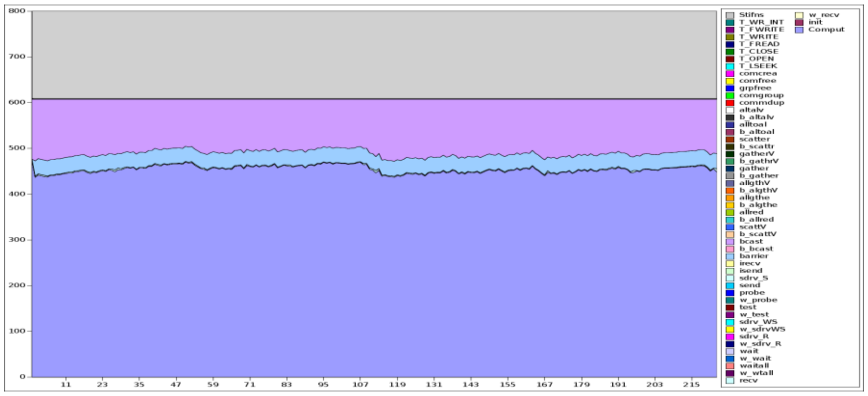
4.2.3 Send Late Time Profiling

Turning on Send Late Time mode affects Recv times and carve out a significant w_Recv portion meaning Send Late Time delays are present.



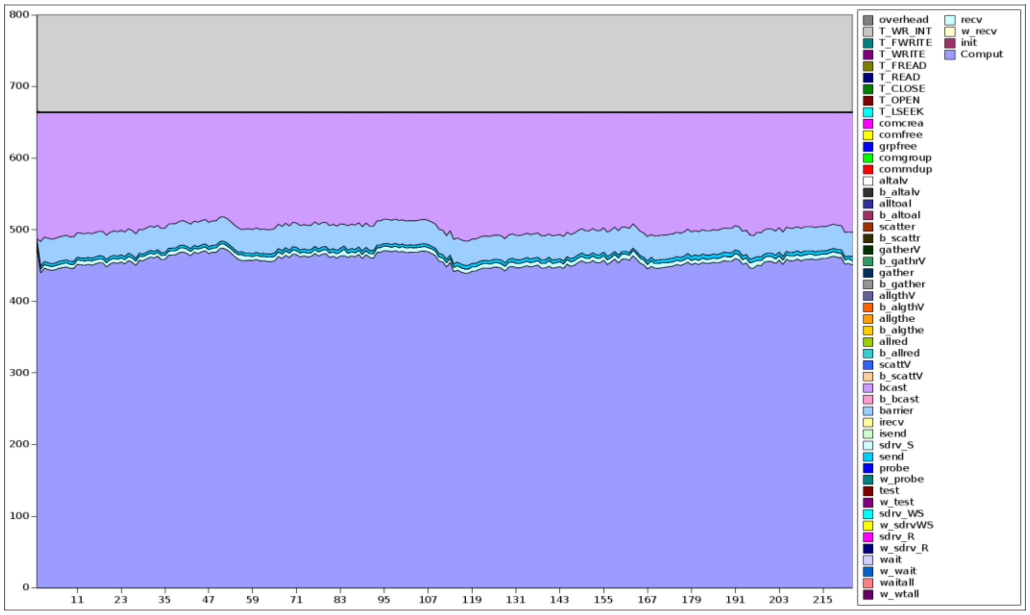
4.2.4 Perfect Interconnect Profiling

b_bcast constituted of wait times for synchronization are not erased out by a perfect interconnect as opposed to bcast.



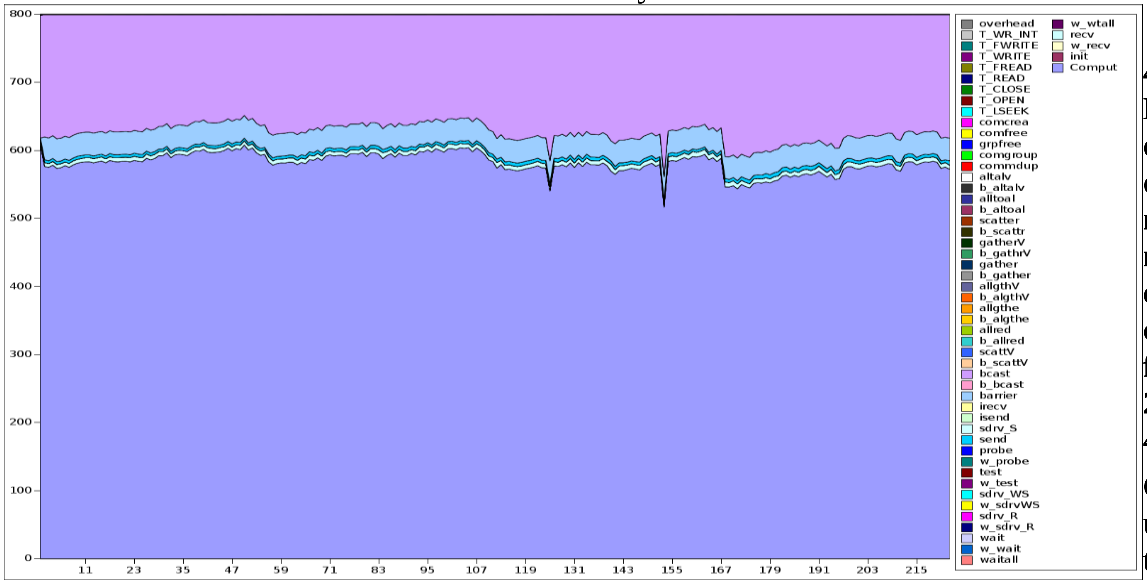
4.2.5 Adding 1 rail to the interconnect

No effect was observed by adding 1 rail to the infiniband fabric.



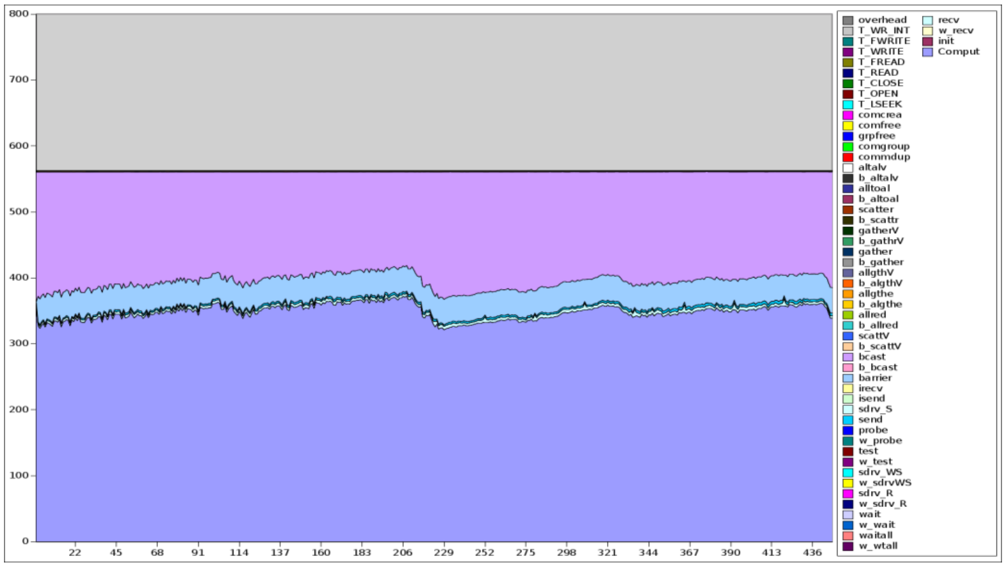
4.2.6 Effect of going out of core

Compute time increases but communication times stay the same.



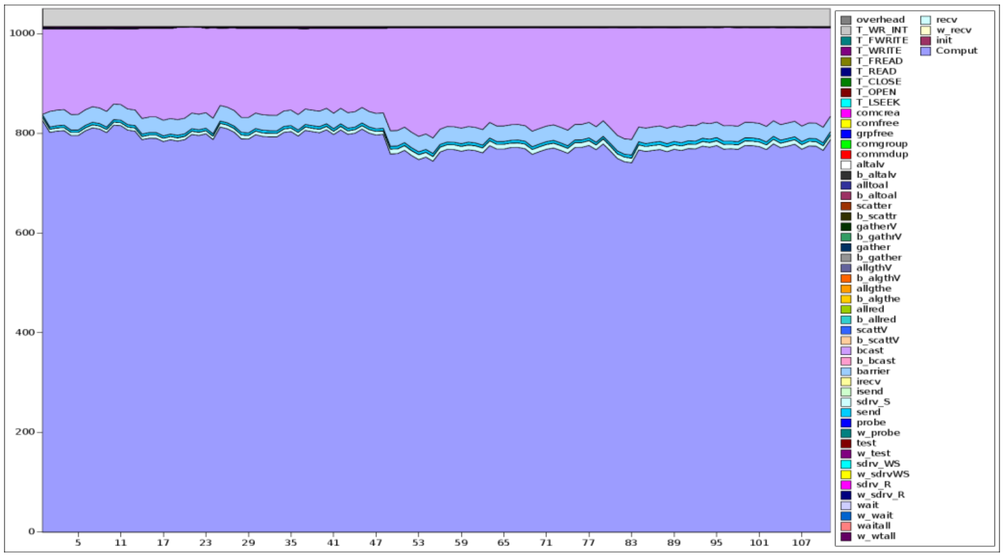
4.2.7 Effect of doubling number of cores from 224 to 448

Compute times decrease but communication stay the same.



4.2.8 Effect of halving number of cores from 224 to 112

Now the solver is processing out of core. Compute times increase but communication times stay constant.



4.3 SGI MPIplace Profile Guided Placement Tool for MPI

SGI MPIplace can speed up execution by mapping ranks to a different sequence of nodes based on rank to rank matrix signature of communications obtained by MPInside to minimize inter node and inter switch transfer costs. A file defining the permutation of ranks to node list is generated that can be used by a subsequent run of the application. MPIplace translates the system's InfiniBand topology information and the rank-to-rank matrices data into a form which the *Scotch* library applies heuristics to through it's implementation of the recursive bi-partitioning algorithm to obtain a nearly optimal mapping of ranks to nodes with respect to the observed transfer patterns as a truly optimal solution would be computationally intractable (NP-complete). [5]

4.3.1 Synopsis:

A. Run Application with MPT for performance baseline.

B. Run Application with MPT and MPInside with MPINSIDE_MATRICES=PLA:-B:S set to produce transfer matrices.

C. Run MPIplace using B) matrices and node list to produce a permutation of ranks along the list of nodes. For example node list n001, n002, n003with multiplicity of 24 cores might be reordered like this:

n001 n002 n002 n002 n003 n001 n003 [...]

D. Run Application again with MPT and permutation of ranks to get improved performance. So one would use the reordered list of nodes in mpirun command:

```
mpirun -v n001 1, n002 1, n002 1, n002 1,
n003 1, n001 1, n003 1, n002 1, n003 1, n003 1, n003 1, n003 1, n003 1, n003 1,
n003 1, n003 1, n003 1, n003 1, n003 1, n003 1, n003 1, n003 1, n003 1, n003 1,
n001 1, n001 1, n001 1, n001 1, n001 1, n001 1, n001 1, n001 1, n001 1, n002 1,
n002 1, n002 1, n002 1, n002 1, n002 1, n002 1, n002 1, n003 1, n003 1, n003 1,
n003 1, n003 1, n003 1, n003 1, n001 1, n001 1, n001 1, n001 1, n001 1, n001 1,
n001 1, n001 1, n001 1, n001 1, n001 1, n001 1, n001 1, n001 1, n001 1, n002 1,
n002 1, n002 1, n002 1, n002 1, n002 1, n002 1, n002 1, n002 1
omplace -vv -c 0-23 mpp971_s_R3.2.1_Intel_linux86-64_sgimpt i=neon.refined.rev01.k
ncpu=1 memory=40m p=pfile memory2=4m
```

and a modified mapping of ranks to nodes would be displayed:

```
wrank   grank   lrank   pinning    node name   cpuid

0       0       0       yes        n003        0
8       1       1       yes        n003        1
9       2       2       yes        n003        2
10      3       3       yes        n003        3
11      4       4       yes        n003        4
12      5       5       yes        n003        5
13      6       6       yes        n003        6
14      7       7       yes        n003        7
15      8       8       yes        n003        8
16      9       9       yes        n003        9
17      10      10      yes        n003        10
```

Or, reordered for clarity:

```
wrank   grank   lrank   pinning    node name   cpuid

0       0       0       yes        n003        0
1       24      0       yes        n002        0
2       25      1       yes        n002        1
3       26      2       yes        n002        2
4       27      3       yes        n002        3
5       28      4       yes        n002        4
6       29      5       yes        n002        5
7       30      6       yes        n002        6
8       1       1       yes        n003        1
9       2       2       yes        n003        2
```

5.0 Benchmark System

SGI ICE XA:

- 576 dual-socket compute nodes.
- 2 x Intel E5-2690 v4 CPU (14 core, 2.6 GHz).
    - 128 GB memory per node.
    - 5D enhanced hypercube interconnect (dual-rail).
- 4-4-3-3-3 topology.
    - FDR InfiniBand.
    - Standard switch blades.
    - SUSE Linux Enterprise Server 11.3.
- Intel compilers (version 15.0.4).
    - SGI MPT (version 2.13).

6.0 Summary

The implicit solver has been studied with MPI analysis tool SGI MPInside using its custom communication profiling and "on the fly" modeling to predict potential performance benefits of the different upgrades available from the latest Intel® Xeon® CPU, interconnect and its middleware, MPI library, and the underlying LS-DYNA source code. The profile-guided MPIplace component can be exercised to minimize inter rank transfer times.

7.0 References

1. Dr. C. Cleve Ashcraft, Roger G. Grimes, and Dr. Robert F. Lucas. "A Study of LS-DYNA Implicit Performance in MPP". In Proceedings of 7th European LS-DYNA Conference, Austria, 2009.
2. Dr. C. Cleve Ashcraft, Roger G. Grimes, and Dr. Robert F. Lucas. "A Study of LS-DYNA Implicit Performance in MPP (Update)", 2009.
3. http://www.ncac.gwu.edu/vml/models.html
4. Daniel Thomas, Jean-Pierre Panziera, John Baron: MPInside: a performance analysis and diagnostic tool for MPI applications. WOSP/SIPEW 2010: 79-86, ACM, (2010) http://www.sgi.com/products/software/ sps.html http://techpubs.sgi.com/library/manuals/5000/007-5780-002/pdf/007-5780-002.pdf.
5. Experimental Analysis of the Dual Recursive Bipartitioning Algorithm for Static Mapping. Research report RR-1138-96, LaBRI, September 1996. F. Pellegrini and J. Roman. https://www.labri.fr/perso/pelegrin/scotch/.

8.0 Acknowledgements

Expert help from John Baron and James Southern from SGI is gratefully acknowledged.

LSTC's generous access to development oriented information was invaluable for these tests.