WHITE PAPER

# Characterizing LS-DYNA® Performance on SGI® Systems Using SGI MPInside MPI Profiling Tool


## Authors

Dr. Olivier Schreiber*, Tony DeVarco**, Scott Shaw***



## Abstract

SGI delivers a unified compute, storage and remote visualization solution to manufacturing customers reducing overall system management requirements and costs with its multiple computer architectures available, namely, multi-node Distributed Memory Processor clusters and Shared Memory Processor systems. LS-DYNA integrates several solvers into a single code base.

In this paper, the LS-DYNA explicit solver is hereby profiled. The MPI analysis tool used is SGI MPInside. SGI MPInside features customary communication profiling and features “on the fly” modeling to predict potential performance benefits of the different upgrades available from the latest Intel® Xeon® processor, interconnect fabric and its middleware, SGI MPI library, and the underlying LS-DYNA source code. We also describe how the profile-guided MPIplace component of SGI MPI is used to minimize inter rank transfer times on a SGI system thus reducing the simulation run time of the TopCrunch “Car2car” standard benchmark by up to 10%.



* Senior Applications Engineer
** Director Manufacturing Solutions
*** Principal Applications Engineer

# TABLE OF CONTENTS

# 1.0 About SGI Systems

SGI systems used to perform the benchmarks outlined in this paper include the SGI® Rackable® standard depth cluster; SGI® ICE™ X integrated blade cluster and the SGI® UV™ 2000 shared memory system. They are the same servers used to solve some of the world's most difficult computing challenges. Each of these server platforms supports LSTC LS-DYNA with its Shared Memory Parallel (SMP) and Distributed Memory Parallel (DMP) modes [1].

## 1.1 SGI® Rackable® Standard-Depth Cluster

SGI Rackable standard-depth, rackmount C2112-4GP3 2U enclosure supports four nodes and up to 4TB of memory in 64 slots (16 slots per server). It also supports up to 144 cores per 2U with support of FDR InfiniBand, fourteen-core Intel® Xeon® processor E5-2600 v3 series and 2133 MHz DDR4 memory running SUSE® Linux® Enterprise Server or Red Hat® Enterprise Linux for a reduced TCO (Figure 1).

***Figure 1:*** *Overhead View of SGI Rackable Server with the Top Cover Removed*

## 1.2 SGI® ICE™ XA System

SGI® ICE™ XA is one of the world's fastest commercial distributed memory supercomputer. This performance leadership is proven in the lab and at customer sites including the largest and fastest pure compute InfiniBand cluster in the world. The system can be configured with compute nodes comprising Intel® Xeon® processor E5-2600 v3 series exclusively or with compute nodes comprising both Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors or Nvidia® compute GPU's. Running on SUSE® Linux® Enterprise Server and Red Hat® Enterprise Linux, SGI ICE XA can deliver over 191 teraflops of performance per rack and scale from 36 to tens of thousands of nodes.

SGI ICE XA is designed to minimize system overhead and communication bottlenecks, and offers, for example the highest performance and scalability above 2,000 cores for LS-DYNA topcrunch.org benchmarks with top-most positions six years running. SGI ICE X can be architected in a variety of topologies with choice of switch and single or dual plane FDR Infiniband interconnect. The integrated bladed design offers rack-level redundant power and cooling via air (currently ICE X provides air cooled racks not ICE XA), warm or cold water and is also available with storage and visualization options (Figure 2 shows the ICE X rack and blade enclosure).

SGI ICE XA configuration used in this paper:

- 576 sockets (13,823 cores)
- Intel® Xeon® 12 core 2.6Ghz E5-2690v3
- Mellanox® Technologies ConnectX® Industry standard Infiniband FDR integrated interconnect Hypercube
- 128GB of RAM/core Memory Speed 2133MHz
- Altair® PBS Professional
- SLES or RHEL, SGI Performance Suite

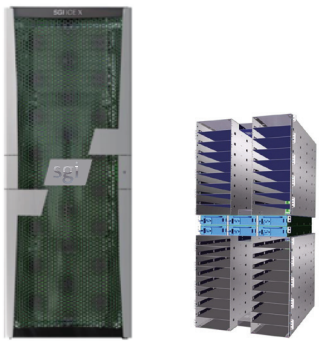


***Figure 2:*** *SGI ICE X Cluster with Blade Enclosure*

## 1.3 SGI® UV™ 3000

SGI UV 3000 server comprises up to 256 sockets (4,096 cores). Support for 64TB of global shared memory in a single system image enables efficiency of SGI UV for applications ranging from in-memory databases, to diverse sets of data and compute-intensive HPC applications all the while programming via the familiar Linux OS [2], without the need for rewriting software to include complex communication algorithms. TCO is lower due to one-system administration needs. Workflow and overall time to solution is accelerated by running Pre/Post-Processing, solvers and visualization on one system without having to move data (Figure 3).

Job memory is allocated independently from cores allocation for maximum multi-user, heterogeneous workload environment flexibility. Whereas on a cluster, problems have to be decomposed and require many nodes to be available, the SGI UV can run a large memory problem on any number of cores adapting to application license availability and with less concern about lack of memory resources killing the job.

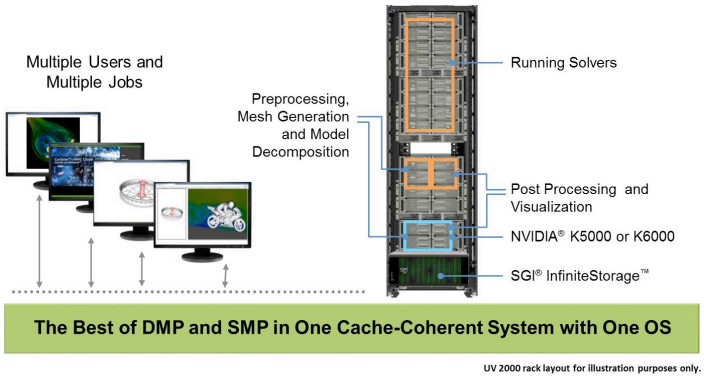



***Figure 3:*** *SGI UV CAE workflow running LSTC applications*

## 1.4 SGI Performance Tools

Utilizing the latest MPI compliant libraries and standard-distribution Linux, SGI® Performance Suite (Figure 4) fuels HPC applications to achieve breakthrough speed and scale. A feature-rich tool set optimizes application placement, enables application tuning at runtime without recompiling, and can boost performance up to 70%. Fine-grain metrics facilitate MPI analysis. Checkpoint Restart augments productivity. And hard real-time performance can be realized without special kernels on standard Linux. Coupled with world-class application expertise, SGI takes Linux to the next level. For detailed information: http://www.sgi.com/products/software/.

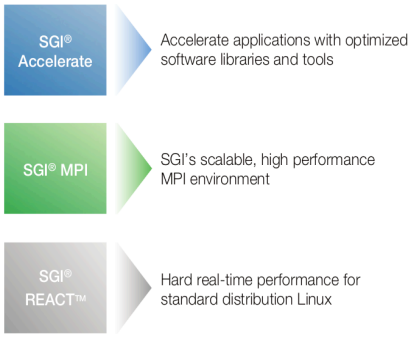



***Figure 4:*** *SGI Performance Suite Components*

## 1.5 SGI System Management Tools

Spanning bare-metal provisioning and protection against memory failure, to 24x7 systems monitoring, task automation, and innovative power optimization, SGI® Management Suite helps maximize productivity and achieve a high return on your investment. Administrators can deploy systems and upgrades with unparalleled speed, proactively manage system health and energy consumption, and deliver consistently high service levels — enabling users to run more jobs in less time and without interruption. For detailed information: http://www.sgi.com/products/software/smc.html

## 1.6 Resource and Workload Scheduling

Resource and workload scheduling allows one to manage large, complex applications, dynamic and unpredictable workloads, and optimize limited computing resources. SGI offers several solutions that customers can choose from to best meet their needs.

**Altair Engineering PBS Professional®** is SGI's preferred workload management tool for technical computing scaling across SGI's clusters and servers. Features:

- Policy-driven workload management which improves productivity, meets service levels, and minimizes hardware and software costs
- Integrated operation with SGI Management Center for features such as workload-driven, automated dynamic provisioning
- Altair PBS Professional Power Awareness integrates job-level power management with SGI Management Center 3

**Adaptive Computing Moab® HPC Suite Basic Edition**

Adaptive Computing Moab® HPC Suite enables intelligent predictive scheduling for workloads on scalable systems.

- Policy-based HPC workload manager that integrates scheduling, managing, monitoring and reporting of cluster workloads
- Includes TORQUE resource manager

## 1.7 SGI® VizServer® with NICE DCV

SGI VizServer with NICE DCV gives technical users remote 3D modeling tools through a web-based portal, allowing for GPU and resource sharing and secure data storage. (Figure 5)

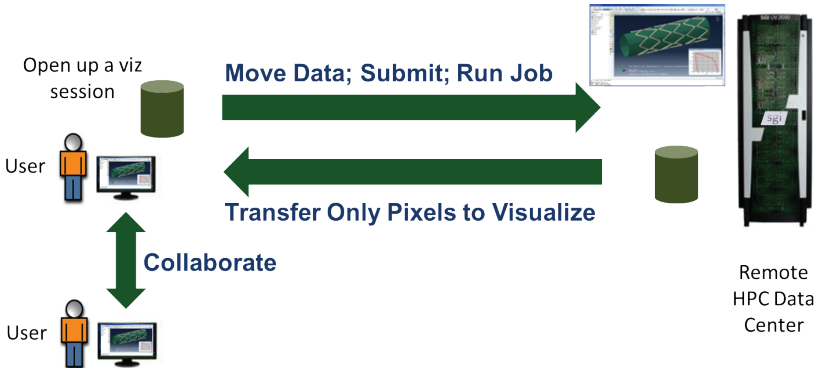



***Figure 5:*** *SGI VizServer workflow*

SGI VizServer with NICE DCV installed on a company's servers can provide LS-PrePost remote visualization capabilities through a software-as-a-service (SaaS) built in the company's private network. The LS-PrePost software is accessed through an easy-to-use web interface, resulting in simplicity for the end user. This solution provides intuitive help and guidance to ensure that less-experienced users can maximize productivity without being hindered by complex IT processes.

SGI VizServer with NICE DCV Components:

- Engineer-friendly self-service portal: The self-service portal enables engineers to access the LS-PrePost application and data in a web browser–based setting. It also provides security, monitoring, and management to ensure that users cannot leak company data and that IT managers can track usage. Engineers access the LS-PrePost application and data directly from their web browsers, with no need for a separate LS-PrePost software installation on their local client.
- Resource control and abstraction layer: The resource control and abstraction layer lies underneath the portal, not visible to end users. It handles job scheduling, remote visualization, resource provisioning, interactive workloads, and distributed data management without detracting from the user experience. This layer translates the user request from the browser and facilitates the delivery of resources needed to complete the visualization or HPC tasks. This layer has a scalable architecture to work on a single SGI Rackable cluster or SGI UV server, as well as a multi-site WAN implementation.
- Computational and storage resources: The SGI VizServer with NICE DCV software takes advantage of the company's existing or newly purchased SGI industry-standard resources, such as servers, HPC schedulers, memory, graphical processing units (GPUs), and visualization servers, as well as the required storage to host application binaries, models and intermediate results. These are all accessed through the web-based portal via the resource control and abstraction layer and are provisioned according to the end user's needs by the middle software.

The NICE DCV and EnginFrame software is built on common technology standards. The software adapts to network infrastructures so that an enterprise can create its own secure engineering cloud without major network upgrades. The software also secures data, removing the need to transfer it and stage it on the workstation, since both technical applications and data stay in the private cloud or data center. These solutions feature the best characteristics of cloud computing—simple, self-service, dynamic, and scalable, while still being powerful enough to provide 3D visualization as well as HPC capabilities to end users, regardless of their location.

# 2.0 LS-DYNA

## 2.1 Versions used

LS-DYNA/MPP ls971 R3.2.1 or later. At R4.2.1, coordinate arrays were coded to double precision for the simulation of finer time-wise phenomena thus incurring a decrease in performance of 25% (neon) to 35% (car2car).

Compilers: Fortran: Intel® Fortran Compiler 11.1 for EM64T-based applications.

MPI: IBM Platform MPI, Intel MPI, Open MPI and SGI MPI.

## 2.2 Parallel Processing Capabilities of LS-DYNA

### 2.2.1 Underlying Hardware and Software Notions

It is important to distinguish hardware components of a system and the actual computations being performed using them. On the hardware side, one can identify:

1. Cores, the Central Processing Units (CPU) capable of arithmetic operations.
2. Processors, the four, six, eight and up core socket-mounted devices.
3. Nodes, the hosts associated with one network interface and address.

With current technology, nodes are implemented on boards in a chassis or blade rack-mounted enclosure. The board may comprise of two sockets or more.

From the software side, one can identify:

1. Processes: execution streams having their own address space.
2. Threads: execution streams sharing address space with other threads.

Therefore, it is important to note that processes and threads created to compute a solution on a system will be deployed in different ways on the underlying nodes through the processors and cores' hardware hierarchy.

### 2.2.2 Parallelism Background

Parallelism in scientific/technical computing exists in two paradigms implemented separately but sometimes combined in 'hybrid' codes: Shared Memory Parallelism (SMP) appeared in the 1980's with the strip mining of 'DO loops' and subroutine spawning via memory-sharing threads. In this paradigm, parallel efficiency is affected by the relative importance of arithmetic operations versus data access referred to as 'DO loop granularity.' In the late 1990's, Distributed Memory Parallelism (DMP) Processing was introduced and proved very suitable for performance gains because of its coarser grain parallelism design. It consolidated on the MPI Application Programming Interface. In the meantime, Shared Memory Parallelism saw adjunction of mathematical libraries already parallelized using efficient implementation through OpenMP™ (Open Multi-Processing) and Pthreads standard API's.

Both DMP and SMP (with some limitations) programs can be run on the two commonly available types of hardware systems:

- Shared Memory systems or single nodes with multiple cores sharing a single memory address space and a single instance of the operating system.
- Distributed Memory systems, otherwise known as clusters, comprised of nodes with separate local memory address spaces and a dedicated instance of the operating system per node.

Note: SMP programs because of their single memory spaces cannot execute across clusters. Inversely, DMP programs can run perfectly well on a Shared Memory system. Since DMP has coarser granularity than SMP, it is therefore preferable, on a Shared Memory system, to run DMP rather than SMP despite what the names may imply at first glance. SMP and DMP processing may be combined together, in what is called 'hybrid mode'.

### 2.2.3 Distributed Memory Parallelism Implementations

Distributed Memory Parallelism is implemented through the problem at hand with domain decomposition. Depending on the physics involved in their respective industry, the domains could be geometry, finite elements, matrix, frequency, load cases or right hand side of an implicit method. Parallel inefficiency from communication costs is affected by the boundaries created by the partitioning. Load balancing is also important so that all MPI processes perform the same number of computations during the solution and therefore finish at the same time. Deployment of the MPI processes across the computing resources can be adapted to each architecture with 'rank' or 'round-robin' allocation.

### 2.2.4 Parallelism Metrics

Amdahl's Law, 'Speedup yielded by increasing the number of parallel processes of a program is bounded by the inverse of its sequential fraction' is also expressed by the following formula (where P is the program portion that can be made parallel, 1-P is its serial complement and N is the number of processes applied to the computation):

Amdahl Speedup=1/[(1-P)+P/N]

A derived metric is: Efficiency=Amdahl Speedup/N

A trend can already be deduced by the empirical fact that the parallelizable fraction of an application depends more on CPU speed, and the serial part, comprising of overhead tasks depends more on RAM speed or I/O bandwidth. Therefore, an application running on a higher CPU speed system will have a larger 1-P serial part and a smaller P parallel part causing its Amdahl Speedup to decrease. This can lead to a misleading assessment of different hardware configurations as shown by this example where, say System B has faster CPU speed than system A:

| N | System a elapsed seconds | System B elapsed seconds |
|---|---|---|
| 1 | 1000 | 810 |
| 10 | 100 | 90 |
| Speedup | 10 | 9 |

System A and System B could show parallel speedups of 10 and 9, respectively, even though System B has faster raw performance across the board. Normalizing speedups with the slowest system serial time remedies this problem:

| | | |
|---|---|---|
| **Speedup** | 10 | 11.11 |

A computational solution of a particular dataset is said to exhibit strong scalability if elapsed execution time decreases when number of processors increases. While computational solution of increasing dataset sizes is said to exhibit weak scalability when elapsed execution time can remain bounded through an increase of number of processors.

It may be preferable, in the end, to use a throughput metric, especially if several jobs are running simultaneously on a system:

Number of jobs/hour/system = 3600/(Job elapsed time)

The system could be a chassis, rack, blade, or any hardware provisioned as a whole unit.

## 2.3 Parallel Execution Control

### 2.3.1 Submittal Procedure

Submittal procedure must ensure:

1. Placement of processes and threads across nodes and also sockets within nodes.
2. Control of process memory allocation to stay within node capacity.
3. Use of adequate scratch files across nodes or network.

Batch schedulers/resource managers dispatch jobs from a front-end login node to be executed on one or more compute nodes so the following is a possible synoptic of a job submission script:

1. Change directory to the local scratch directory on the first compute node allocated by the batch scheduler.
2. Copy all input files over to this directory.
3. Create parallel local scratch directories on the other compute nodes allocated by the batch scheduler.
4. Launch application on the first compute node. The executable may itself carry out propagation and collection of various files between launch node and the others at start, and end of the main analysis execution. The launch script may also asynchronously sweep up output files like d3plot* files to free up scratch directory.

### 2.3.2 Run Command with MPI Tasks and OpenMP Thread Allocation Across Nodes and Cores

For LS-DYNA, the deployment of processes, threads and associated memory is achieved with the following keywords in the execution command below [1]:

- `-np:` Total number of MPI processes used in a Distributed Memory Parallel job.
- `ncpu=:` number of SMP OpenMP threads.
- `memory, memory2:` Size in words of allocated RAM for MPI processes.
  (A word is 4 or 8 bytes long for single or double precision executables, respectively.)

1. Pure MPI mode- on all or subset of cores available:
   ```
   mpirun -np #MPIprocesses HybridExec inputFile ncpu=1
   #MPIprocesses=#nodes x #CoresPerNode
   ```
2. Hybrid mode: combinations MPI processes & threads:
   ```
   mpirun -np #MPIprocesses HybridExec inputFile ncpu=#ThreadsPerProcess
   #MPIprocesses x ncpu = total#Threads = #nodes x #CoresPerNode
   ```
3. SMP mode (threads only):
   ```
   mpirun -np 1 HybridExec inputFile ncpu=#CoresOn1Node
   ```
4. SMP mode with SMP-only executable (threads only):
   ```
   SMPexec inputFile ncpu=#CoresOn1Node
   ```

## 2.4 Tuning

### 2.4.1 Input/Output and Memory

To achieve the best runtime in a batch environment, disk access to input and output files should be placed on the high performance filesystem closest to the compute node. The high performance filesystem could be either an in-memory filesystem (/dev/shm), a Direct (DAS) or a Network (NAS) Attached Storage filesystem. In diskless computing environments, in-memory filesystem or Network Attached Storage are the only options. In cluster computing environments with a Network Attached Filesystem (NAS), isolating application MPI communications and NFS traffic will provide the best NFS I/O throughput for scratch files. The filesystem nomenclature is illustrated in Figure 7.

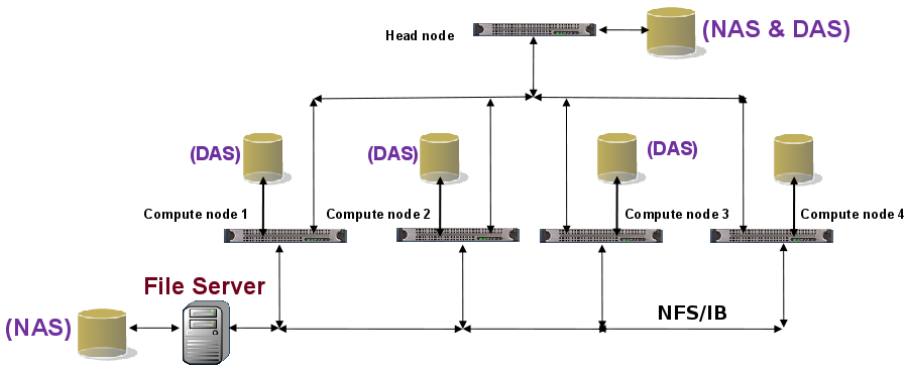



***Figure 6:*** *Example filesystems for Scratch Space*

Having more system memory per core will increase performance since it can be allocated for the analysis as well as the Linux kernel buffer cache to improve I/O efficiency. SGI's Flexible File I/O (FFIO) is a link-less library (which means it does not need to be linked to the application) bundled with SGI Accelerate. It implements user defined I/O buffer caches to avoid the operating system ones from thrashing when running multiple I/O intensive jobs or processes. This can be effective in Shared Memory Parallel systems or cluster computing environments using DAS or NAS storage subsystems. FFIO isolates user page caches so jobs or processes do not contend for Linux Kernel page cache. Hence, FFIO minimizes the number of system calls and I/O operations as echoed back by the eie_close sync and async values reflecting synchronous calls to disk—which should be as close to 0 as possible—to and from the storage subsystem and improves performance for large and I/O intensive jobs. (Ref [2], Chapter 7 Flexible File I/O).

### 2.4.2 Using Only a Subset of Available Cores on Dense Processors

Two ways of looking at computing systems are either through nodes which are their procurement cost sizing blocks or through cores which are their throughput sizing factors. When choosing metrics, because processors have different prices, clock rates, core counts and memory bandwidth, optimizing for turnaround time or throughput will depend on running on all or a subset of cores available. Since licensing charges are assessed by the number of threads or processes being run as opposed to the actual number of physical cores present on the system, there is no licensing cost downside in not using all cores available so this may provide performance increase possibilities. The deployment of threads or processes across partially used nodes should be done with consideration to the existence of shared resources among cores.

### 2.4.3 Intel® Hyper-threading

Intel Hyper-threading (HT) is a feature of the Intel® Xeon® processor family which can increase performance for multi-threaded or multi-process applications. It allows a user to run twice the number of OpenMP threads or MPI processes than available physical cores per node (over-subscription).

Note, beyond 2 nodes, with LS-DYNA, Hyper-threading gains are negated by added communication costs between the double-up numbers of MPI processes.

### 2.4.4 Intel® Turbo Boost

Intel Turbo Boost is a feature of the Intel® Xeon® processor family, for increasing performance by raising the core operating frequency within controlled limits constrained by the thermal envelope of the processor. The mode of activation is a function of how many cores are active at a given moment when MPI processes, OpenMP threads or Pthreads are running. Turbo Boost can improve performance for low numbers of cores used, up to the ratio of the maximum frequency over baseline value. As more cores are used, Turbo Boost cannot increase the frequencies on all of them as it can on fewer active ones. For example, for a base frequency of 3.0GHz, when 1-2 cores are active, core frequencies might be throttled up to 3.3GHz, but with 3-4 cores active, frequencies may be throttled up only to 3.2 GHz. For computational tasks, utilizing Turbo Boost often results in improved runtimes so it is best to leave it enabled, although the overall benefit may be mitigated by the presence of other performance bottlenecks outside of the processor.

### 2.4.5 SGI Performance Suite MPI and SGI PerfBoost

The ability to bind an MPI rank to a processor core is key to control performance on the multiple node/socket/core environments available. From [3], '3.1.2 Computation cost-effects of CPU affinity and core placement [...] HP-MPI currently provides CPU-affinity and core-placement capabilities to bind an MPI rank to a core in the processor from which the MPI rank is issued. Children threads, including SMP threads, can also be bound to a core in the same processor, but not to a different processor; additionally, core placement for SMP threads is by system default and cannot be explicitly controlled by users.[...]'.

In contrast, SGI MPI, through its 'omplace' option enforces accurate placement of Hybrid MPI processes, OpenMP threads and Pthreads within each node. SGI MPI's bundled PerfBoost facility linklessly translates IBM Platform MPI, Intel MPI, OpenMPI calls on the fly to SGI MPI calls.

### 2.4.6 SGI Accelerate LibFFIO

LS-DYNA/MPP/Explicit is not I/O intensive and placement can be handled by SGI MPI, therefore, libFFIO is not necessary. However, LS-DYNA/MPP/Implicit does involve larger I/O so libFFIO can compensate for bandwidth contention on NAS or slow filesystems.

## 3.0 Benchmarks Description

The benchmarks used belong to the three TopCrunch (http:www.topcrunch.org) datasets--created by National Crash Analysis Center (NCAC) at George Washington University. The TopCrunch project was initiated to track aggregate performance trends of high performance computer systems and engineering software. Instead of using a synthetic benchmark, an actual engineering software application, LS-DYNA/Explicit, is used with real data. Since 2008, SGI has held top performing positions on the three datasets. The metric is: Minimum Elapsed Time and the rule is that all cores for each processor must be utilized.

LS-DYNA/Implicit [4], [5] has been covered in [8][9].

## 3.1 Car2car

Angled 2 vehicle collision (Figure 8). The vehicle models are based on NCAC minivan model with 2.5 million elements. The simulation writes 201,854,976 Bytes d3plot and 101,996,544 Bytes d3plot[01-25] files at 26 time steps from start to end point (2624MB).

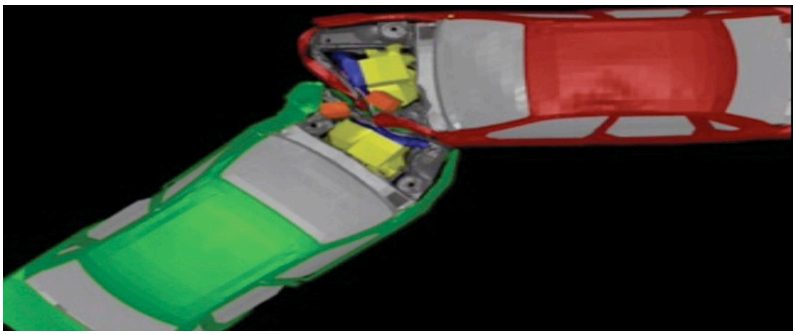
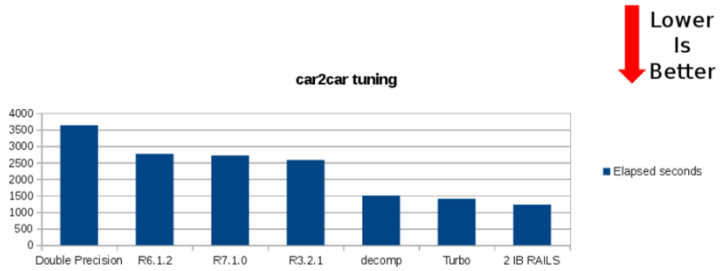


***Figure 8:*** *Car2car*

## 3.2 Car2car Tuning

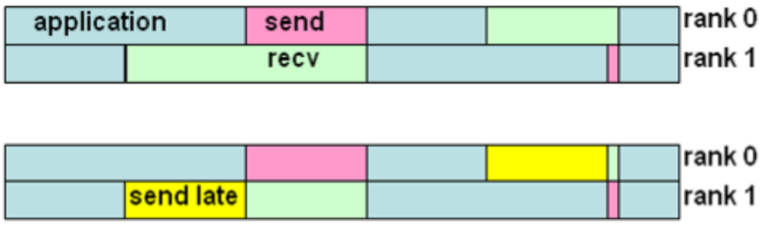
Going from Double to Single Precision when possible affects performance. LS-DYNA's version chosen also affects results going back to R3.2.1. Then adjusting the otherwise automatic decomposition can improve results. Intel® Turbo Boost and dual rail further improve performance where last entry cumulates Single Precision, R3.2.1, custom decomposition, Turbo Boost mode and dual rail.




# 4.0 MPInside

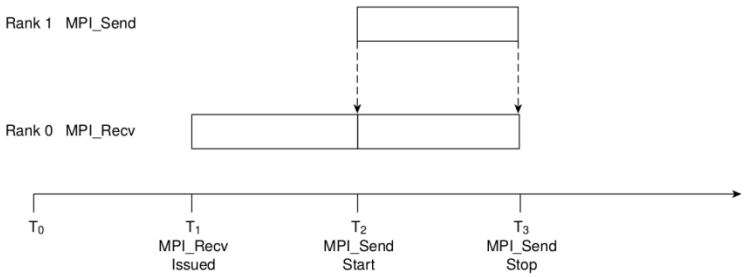
## 4.1 MPInside Introduction

MPInside is available with SGI MPI – High Performance MPI Environment along with mpiplace as an MPI profiling tool [10]. It can provide information to help MPI application developers optimize their application by finding out for example where MPI Send/Receive pairs are not executed synchronously.




## 4.2 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 overlap communication and computation. It is usually assumed that the communication time is transfer time, but in fact MPI_Irecv allows 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.

## 4.3 MPInside usage

### 4.3.1 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 command prepended to the target application executable.

### 4.3.2 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:

#### 4.3.2.1 Timing Table

```
>>>> 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 |

#### 4.3.2.2 Bytes Sent Table:

```
>>>> 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 |

#### 4.3.2.3 Number of "Send" Calls Table:

```
>>>> 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 |

#### 4.3.2.4 Bytes Received Table:

```
>>>> 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 |

#### 4.3.2.5 Number of "Recv" Calls Table:

```
>>>> 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 |

#### 4.3.2.6 Other Outputs

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

##### 4.3.2.6.1 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 |

#### 4.3.2.6.2 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 |

### 4.3.2.7 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”;
- SIZE(i,j): the amount of data transferred from “i” to “j”;
- REQUEST(i,j): the number of calls involved in these transfers.

## 4.4 Information Handled by MPInside

### 4.4.1 General

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

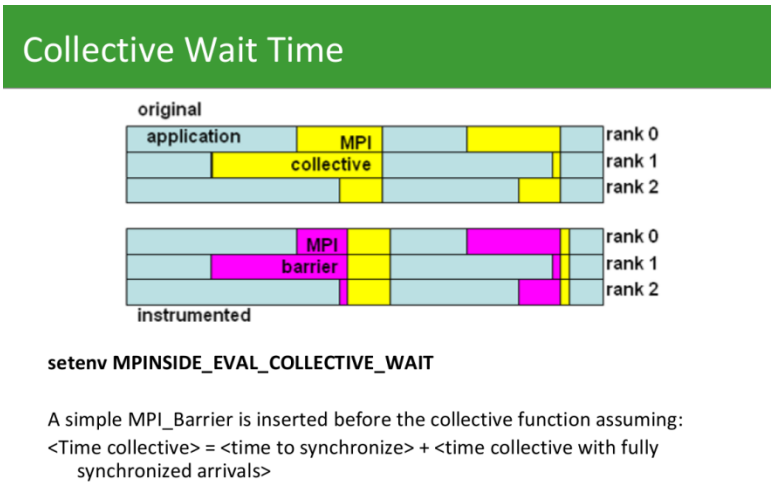
For collective operations such as MPI_Bcast or MPI_Alltoall, transfers are assigned as a send for the root of the broadcast and as a receive for the other ranks participating in the operation.

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 the 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.




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 modes are 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 nonzero when matching sender is late (Send Late Time, SLT). 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. The total elapsed time is the sum of the “Compute” column and all the MPI columns.

### 4.4.2 Shortened names for MPI functions

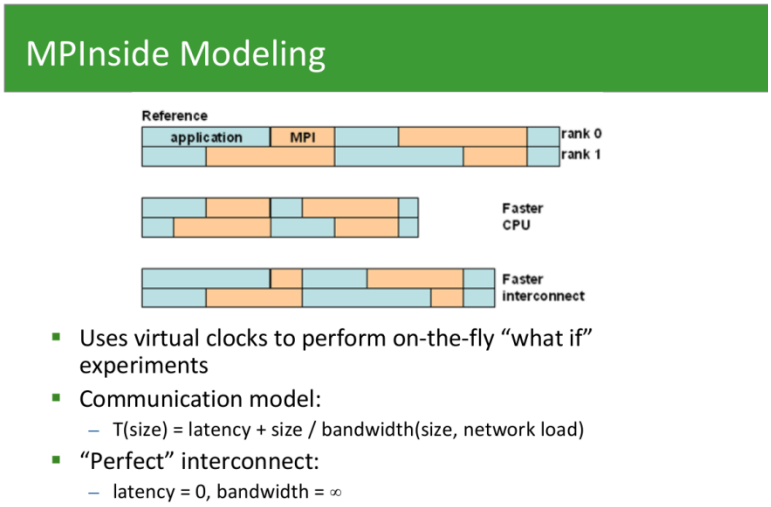
b_<Collective_function>: Artifical MPI_Barrier inserted before the collective function if MPINSIDE_EVAL_COLLECTIVE_WAIT set. Total time for collective function is b_<Collective_function> + <Collective_function>.

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.

### 4.4.3 Further Capabilities

Instead of being measured, MPInside reports are also available with the 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 SLT or collective waiting times, but might be shorter because zero transfer times sometimes lead to better rank synchronization.




## 4.5 MPInside Inferences

Large times in w_MPI_Recv column of MPInside tables correspond to Send Late Time (SLT) situations in Recv.

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 Time (SLT) 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.

## 4.6 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.

## 4.7 Case Study Car2car Topcrunch Benchmark

### 4.7.1 Basic Profiling

SGI MPInside was run to get basic profiling and construct the area plot stack across all 1992 MPI processes attributed to computation time and MPI calls. Figure 8 shows elapsed seconds on the Y axis for the complete range of ranks 0 to 1992.

The light purple, blue and dark blue bands indicate that across all ranks, a little less than half of the running time was compute, with the majority of the communication time spent in Recv and Bcast calls.

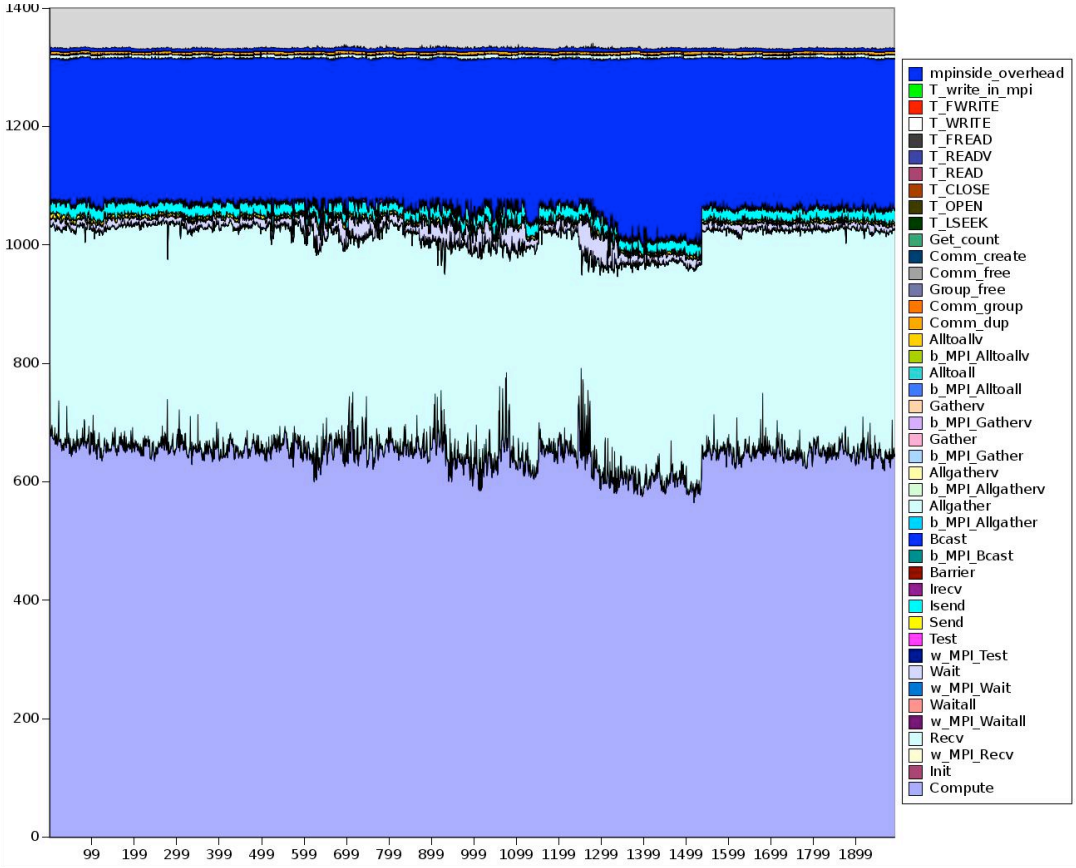



***Figure 8:*** *mpinside_basic_f2501_stats.xls*

### 4.7.2 Collective Wait Profiling

Figure 9 shows how 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 1% of that time.

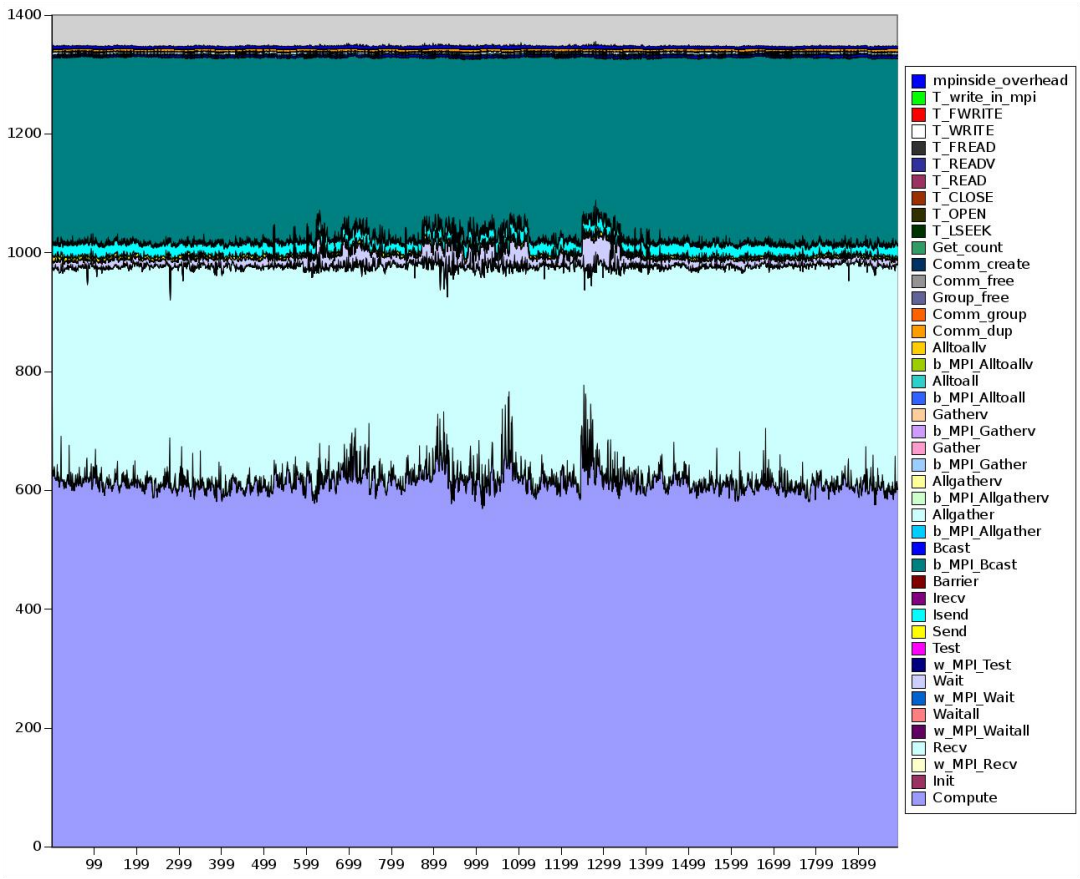



***Figure 9:*** *mpinside_collectivewait_f2501_stats.xls*

### 4.7.3 Send Late Time Profiling

Figure 10 shows how turning on Send Late Time mode does not affect Recv times and carve out a significant w_MPI_Recv portion which means Send Late Time delays are not significant.

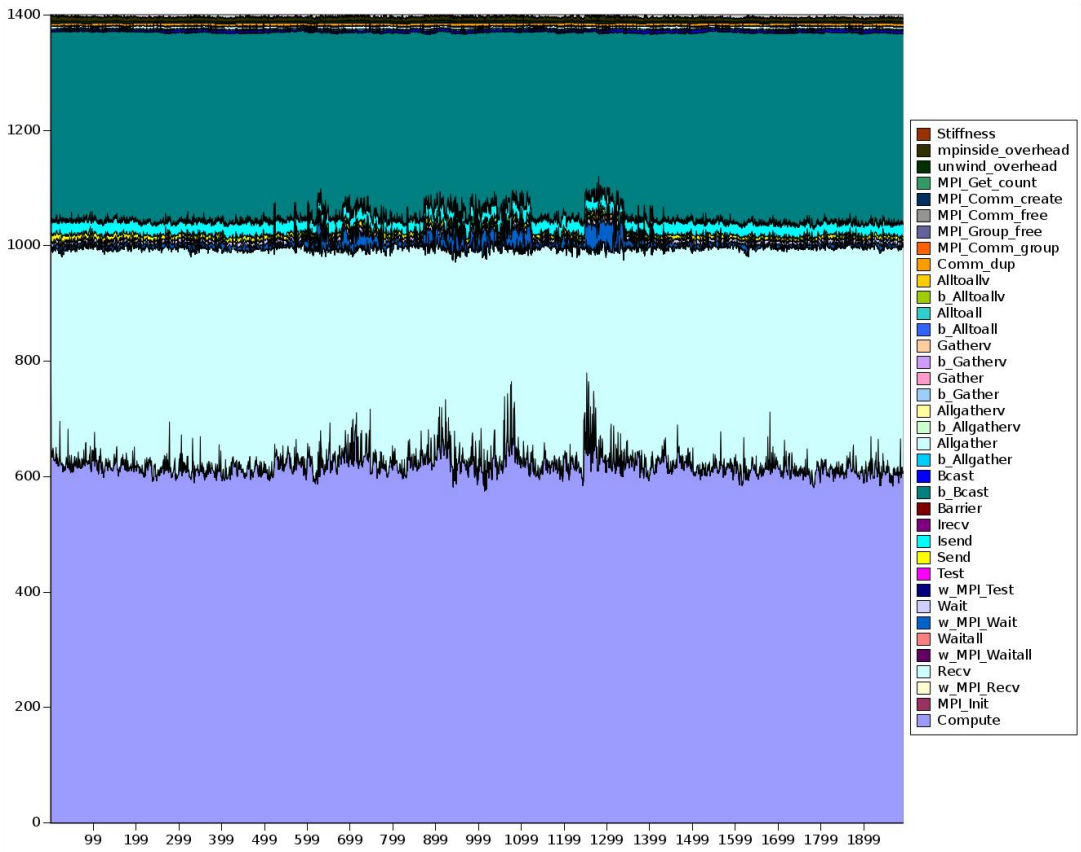



***Figure 10:*** *mpinside_slt_f2501_stats.xls*

### 4.7.4 Perfect Interconnect Profiling

Figure 11 shows the b_Bcast constituted of wait times for synchronization are not erased out by a perfect interconnect and appear as Bcast in yellow. In contrast, perfect interconnect modeling zero'ed out Recv times.

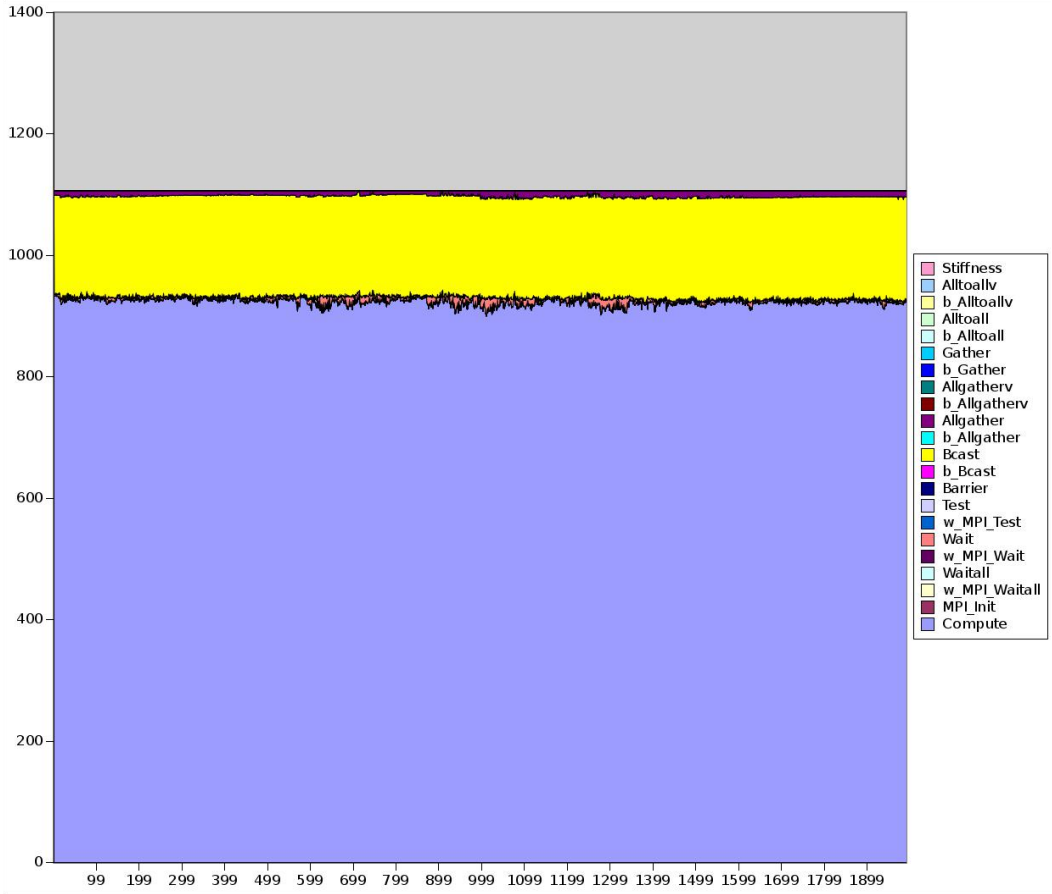



***Figure 11:*** *mpinside_perfect_f2501_stats.xls*

## 4.8 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 data in the rank-to-rank matrices into a form that can be understood by Scotch. Scotch is a library that can apply heuristics to a class of loosely related problems from static mapping, graph partitioning and mesh refinement to get a near-optimal map, partition or mesh in cases where a truly optimal solution would be computationally intractable (NP-complete). MPIplace then uses Scotch's implementation of the recursive bipartitioning algorithm to come up with a mapping of ranks to nodes that's nearly optimal with respect to the observed transfer patterns. [11]

### 4.8.1 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) step matrices and node list to produce a permutation of ranks along the list of nodes. For example node list `n001, n002, n003` with multiplicity of 24 cores maybe reordered like this:

```
n003
n002
n002
n002
n002
n002
n002
n002
n003
n003
n003
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 n003 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, 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
```

### 4.8.2 Case Study Car2Car Topcrunch Benchmark

Model with 1992 ranks over 83 24-core nodes was run with:

A. LS-DYNA with MPT 2.12-beta for performance baseline
   Elapsed: 3222 sec. (0 hours 53 min. 42 sec.) 239981 cycles

B. LS-DYNA with MPT 2.12-beta MPInside 3.6.6-beta to generate matrices.
   Elapsed: 3492 sec. (0 hours 58 min. 12 sec.) 239981 cycles

C. mpiplace to generate permutation file.

```
$ head mpiplace_perm

936
937
1560
1561
1562
1563
1564
1565
1566
1567
[...]
Elapsed: 56.18 sec.
```

D. LS-DYNA again MPT 2.12-beta, no MPInside with mpiplace_perm and mpirun
   Elapsed: 2992 sec. (0 hours 49 min. 52 sec.) 239981 cycles

Which is a gain of 7%

This 7% improvement in performance comes in addition to the load balance tuning done through domain decomposition. If domain decomposition wasn't used for the baseline run the performance improvement using MPInside and mpiplace would yield a 10% improvement vs. 7%.

## 5.0 Summary

The explicit solver has been studied with MPI analysis tool SGI MPInside using its features 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. The profile-guided mpiplace component was exercised to minimize inter rank transfer times and already brought some benefit encouraging further efforts in that area.

## 6.0 Literature


1. LS-DYNA®, KEYWORD USER’S MANUAL, VOLUME I, Appendix O, August 2012, Version 971 R6.1.0.
2. SGI. Linux Application Tuning Guide. Silicon Graphics International, California, 2009.
3. Yih-Yih Lin and Jason Wang. “Performance of the Hybrid LS-DYNA on Crash Simulation with the Multicore Architecture”. In 7th European LS-DYNA Conference, 2009.
4. 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.
5. Dr. C. Cleve Ashcraft, Roger G. Grimes, and Dr. Robert F. Lucas. “A Study of LS-DYNA Implicit Performance in MPP (Update)”, 2009.
6. Olivier Schreiber, Michael Raymond, Srinivas Kodiyalam, LS-DYNA® Performance Improvements with Multi-Rail MPI on SGI® Altix® ICE cluster, 10th International LS-DYNA® Users Conference, June 2008.
7. Olivier Schreiber, Scott Shaw, Brian Thatch, and Bill Tang. “LS-DYNA Implicit Hybrid Technology on Advanced SGI Architectures”. July 2010.
8. Olivier Schreiber, Tony DeVarco, Scott Shaw and Suri Bala, ‘Matching LS-DYNA Explicit, Implicit, Hybrid technologies with SGI architectures’ In 12th International LS-DYNA Conference, May 2012.
9. Leveraging LS-DYNA Explicit, Implicit, Hybrid technologies with SGI hardware, Cyclone Cloud Bursting and d3VIEW, Olivier Schreiber*, Tony DeVarco*, Scott Shaw* and Suri Bala† *SGI, †LSTC, 9th European Users Conference, 3-4th June 2013-Manchester, UK.
10. 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.
11. Experimental Analysis of the Dual Recursive Bipartitioning Algorithm for Static Mapping. Research report RR-1138-96, LaBRI, September 1996. F. Pellegrini and J. Roman. http://www.labri.fr/perso/pelegrin/scotch.


## 7.0 About SGI

SGI is a global leader in high performance solutions for compute, data analytics and data management that enable customers to accelerate time to discovery, innovation, and profitability. Visit sgi.com/manufacturing for more information.

**Global Sales and Support**: sgi.com/global