In-situ visualization using Damaris: the Code Saturne use case

Joshua C Bowden, François Tessier, Charles Deltel, Simone Bnà, Gabriel Antoniu

Available on-line at www.prace-ri.eu

Partnership for Advanced Computing in Europe

In-situ visualization using Damaris: the Code_Saturne use case

Joshua C Bowden[a,*], Francois Tessier[a], Charles Deltel[a], Simone Bnà[b], Gabriel Antoniu[a]

[a]Inria, Rennes - Bretange Atlantic, Campus de Beaulieu Rennes, 35042, cedex, France
[b]SuperComputing Application and Innovation Department, Cineca, Via Magnanelli 6/3, 40133, Casalecchio di Reno, Bologna, Italy

Abstract

As the exascale era approaches, maintaining scalable performance in data management tasks (storage, visualization, analysis, etc.) remains a key challenge in sustaining high performance for the application execution. To address this challenge, the Damaris middleware leverages dedicated computational resources in multicore nodes to offload data management tasks, including I/O, data compression, scheduling of data movements, in-situ analysis, and visualization. In this study we evaluate the benefits of Damaris to improve the efficiency of in-situ visualization for Code_Saturne, a fluid dynamics modeling environment. The experiments show Damaris to adequately hide the I/O processing of various Paraview processing pipelines in Code_Saturne. In all cases the Damaris enabled version of Code_Saturne was found to be more efficient than the identical non-Damaris capable version when running the same Paraview pipeline.

1. Introduction

Large-scale simulations running on leadership-class supercomputers generate massive amounts of data for subsequent analysis and visualization. Under heavy access, the performance of traditional HPC storage systems show their limitations and exhibits high variability. Damaris [1, 2, 3, 4, 5] is a middleware system that leverages dedicated cores in multicore nodes to offload data management tasks, including I/O, data compression, scheduling of data movements, in-situ analysis and visualization. Damaris scaled up to 16,000 cores on Oak Ridge's leadership supercomputer 'Titan' (which was first in the Top500 supercomputer list in Nov. 2012) and was tested on other top supercomputers (e.g. University of Tennessee 'Kraken'; Oak Ridge, Tennessee 'Jaguar'). Damaris is now an international reference system for I/O management and in-situ processing for extreme-scale systems. It has been distributed with VisIt, one of the most widely used visualization engines for HPC systems [6, 7]. Damaris has been successfully integrated into a number of large-scale simulation application environments, including: CM1, OLAM, Nek5000, GTC.

For asynchronous data storage, Damaris currently has built-in capability to output HDF5 data in file-per-process or collective modes and is integrated with Paraview Catalyst and VisIt visualization interfaces via use of VTK libraries [8, 9, 10]. Other methods of output specific to a client software can also be integrated with Damaris via a plug-in architecture that has access to its server side asynchronous data store. Damaris can be allocated a user configurable number of cores per node or alternatively a number of dedicated nodes at runtime to carry out its asynchronous processing. The resources allocated to Damaris are called Damaris dedicated cores (or nodes) or Damaris server cores (or nodes). Damaris relies on an XML configuration file to configure its functionality, to specify numbers of server cores or nodes, and to define the data structures that will be passed from an application to the processing cores. The data structures can be dynamically sized depending on XML file parameter values that can be set at runtime by the client software and used within the data structure definitions within the XML configuration file. This

*Corresponding author, e-mail: joshua-charles.bowden@inria.fr

makes the Damaris system flexible and convenient for both the application integrator and the end-user. Damaris uses a shared memory buffer (shmem) to store application data for processing. It is also possible for applications to use the buffer for client side memory allocation, thus allowing a 'zero-copy' method of memory management with a concomitant reduction in memory demands on the system. Furthermore, the use of Damaris with in-situ visualization processing capability requires the definition of a mesh type in the XML definition file. The mesh defines where in space the associated simulation variable (for example, velocity field data) is situated. As part of this work, an unstructured mesh type has been added to Damaris. This gives users the choice of rectilinear, curvilinear, point and unstructured mesh types. Readers are referred to the Damaris website [5] and the example code in the Damaris GitLab code repository [11] for examples of how to use the Damaris API and how the XML configuration file is used to define data structures to be used by the Damaris API. Damaris is linked at compile time to the simulation executable as either a static or dynamic library. The simulation is then run in single program multiple data (SPMD) mode and Damaris is passed the simulation global MPI communicator and uses `MPI_Comm_split()` to partition its desired processes from the initial pool for its use. The simulation code to must then request the client sub-communicator using the `damaris_client_comm_get()` API function that is then used by the simulation ranks for the remainder of the simulation.

This work has been done as part of the PRACE 6th Implementation Phase Project, Task 6.2, which has the goal of the prototyping and design of new services for the European Data Infrastructure (EDI). The group involved has selected the Damaris library to be trialed as a method for enhanced in-situ visualization within a code designed to be used in large scale computational problems on PRACE based infrastructure. To this end, an integration of the Damaris library with Code_Saturne [12, 13], a fluid dynamics modeling environment has been undertaken. This document discusses the current capabilities, documents performance of the combined codes and presents results for profiling and tuning the implementation. The timing results are obtained for the Code_Saturne integration running on the PRACE Tier-0 HAWK supercomputer that has been recently commissioned (Nov. 2020) and hosted at the High-performance Computing Center (HLRS), Stuttgart, Germany [14].

Code_Saturne is a finite volume computational fluid dynamics (CFD) simulation environment developed over the past 25 years at the French energy company EDF. It is an open-source, multi-capability CFD modeling environment which supports both single and multi-phase flow and includes modules for atmospheric flow, combustion modeling, electric modeling, and particle tracking. It is parallelised using both MPI [15] and OpenMP [16, 17] and has been shown to scale over large distributed memory computer systems such as IBM Blue-Gene/P to 32,768 cores [18]. Code_Saturne has previously been integrated with Paraview Catalyst in-situ visualization [19].

The aim of this work is to show that Damaris asynchronous I/O and in-situ visualization can further improve performance of Code_Saturne in a typical scenario with model sizes encountered in industry. This paper presents methods for allocation of resources for Damaris and shows what problems asynchronous processing can encounter due to under-allocation of resources. In addition, we will look at ways for predicting and mitigating any issues that could be encountered. The paper also presents an analysis of integration effort required to add the Damaris functionality to Code_Saturne (i.e. lines of code (LOC) added), which is another important consideration when deciding to integrate a library.

2. Experimental Setup

2.1. Notation for Nodes, Processes, Threads and Dedicated Cores

The text and graphs that follow will use the following scheme to represent nodes, MPI processes, OpenMP threads and dedicated cores.

For the number of nodes in an experiment, the number will be preceded with an 'n', e.g. 'n2' means the computation was performed on 2 nodes.

A combined notation is used for per-node resources as follows:

X.Y.Z

X : The number of *computational* MPI processes per node (i.e. ranks exclusive of Damaris dedicated cores)
Y : The number of OpenMP threads per Process

Z : The number of *Damaris dedicated core* processes per node (i.e. the ranks used for asynchronous processing of e.g. in-situ visuliazation tasks)

2.2. Hardware

A schematic of the overall architecture of HAWK is shown in Figure 1 [14].

Node Architecture:

The compute nodes of HAWK consist of 2 AMD EPYC 7742 CPUs (Zen2 aka Rome CPUs, 64 cores each) and are equipped with 256 GB RAM. The architecture of the Zen2 Rome CPU is an important consideration when configuring Damaris and will be discussed further in Section 2.5 'Process Placement' below. The hwloc library [20] tool `lstopo` provided further details of the compute node architecture, a feature of which is the grouping of 4 CPUs with their own level 3 (L3) cache. The group is termed a CCX (short for Core CompleX) by the manufacturer. The 128 cores are also grouped into 8 NUMA node divisions (numbered 0-7), with Infiniband network ports attached to NUMA node 1 and 6. Further details of the architecture are available through PRACE best practice guide 'Modern Processors' [21].

Interconnect:

HAWK has a total of 5632 nodes, connected via Infiniband HDR based interconnect with a 9-dimensional enhanced hypercube topology. Each path across the hypercube has an MPI latency ~1.3 µs, with 16 nodes sharing a single switch with 'single hop' locality.

I/O Scratch File System:

HAWK uses a Lustre file system, a high performance parallel file system. It is is managed through the 'Workspace' mechanism to obtain a temporary storage area to be used by applications. The resulting scratch area was used in its default configuration (1 MB stripe size, 8 Object Storage Targets (OSTs) [22]) and was on the original ws9 file system, not the ws10 file system delivered with HAWK (ws10 was available from 18/05/2021).

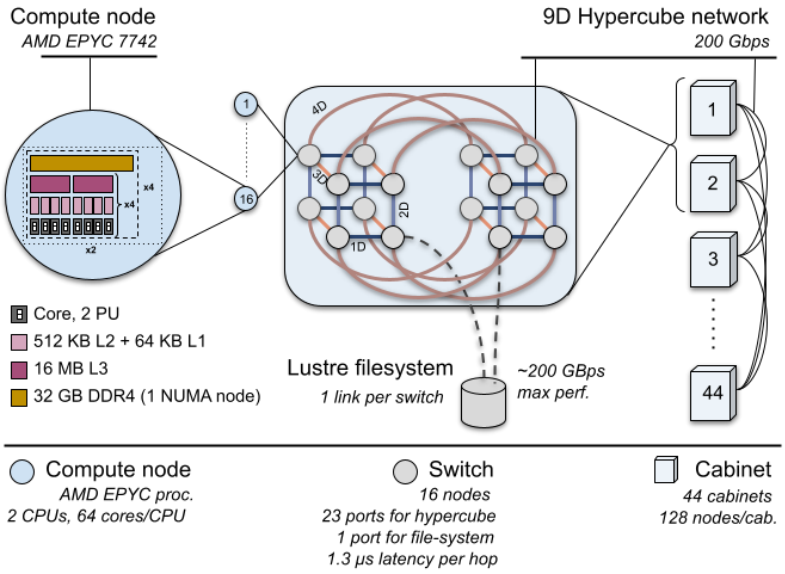

Figure 1. Diagram showing high level overview of HAWK Architecture

2.3. Software

A list of software used with version information is shown in Table 1.

Table 1: List of software used.

Software	Version
O.S.	Linux CentOS8 4.18.0-193.28.1.el8_2.x86_64
Compiler	gcc 9.2.0 Optimisation flags: -02 -march=znver2 -mtune=znver2
Resource batch system	PBS Professional v19.2.3
MPI	Hewlett Packard Enterprise (HPE) Message Passing Toolkit (MPT); mpt/2.23 libxmpi.so 'HPE MPT 2.23 08/26/20 02:56:08-root'
Code_Saturne	Version 6.2-alpha; https://gitlab.inria.fr/Damaris/Simulations/code-saturne-damaris-integration branch: cs_with_damaris SHA:ec8ff8a61fa023660e97ca3d1a3512697d2cb766
Damaris	v.1.3.3, dynamically linked; https://gitlab.inria.fr/Damaris/damaris-development branch: code_saturne SHA:ee3059c57b17ead9d852dac5adc9cdd21f069812
Paraview	paraview/server/v5.8.0-364-g17464f2efe
MED	v4.1.0 –with-med_int=int –with-int64=long
Scotch	scotch/6.0.9-int64-shared
CGNS	cgns/3.4.0-int64
HDF5	hdf5/1.10.5
Mesa	mesa/20.0.1
Python	v3.8.3
xerces-c	v3.2.2
xsd	v4.0.0

A table of user set environment variables is listed in Table 2.

Table 2: Standard user exported environment variables

Variable	Configuration Used
KMP_AFFINITY	disabled
OMP_NUM_THREADS	<variable>
MPI_DSM_VERBOSE	1
MPI_USE_IB	TRUE
MPI_COLL_OPT	TRUE
MPI_COL_OPT_VERBOSE	TRUE
MPI_MEM_ALIGN	32

The system default MPI configuration was modified by system administration part way through data collection (Change date, 23/02/2021). Variables changed are seen in Table 3. The changes were to improve performance for short messages.

Table 3: MPI environment variables that were modified during the course of the experiments by system administrators

Variable	Original Configuration	Modified Configuration
MPI_NUM_QUICKS	0	unset
MPI_IB_ADAPTIVE_ROUTING	unset	1

Code_Saturne did not compile using OpenMPI due to an incompatibility via the Mesa library that was compiled with MPT, and ParaView VTK libraries libvtkParallelMPI-pv5.8.so and libvtkIceTMPI-pv5.8.so.1 causing undefined references to 'mpi_sgi_status_ignore' and 'mpi_sgi_inplace'.

2.4. Code_Saturne Model and Simulation Setup

Model Creation

The geometric model used for Code_Saturne was generated using the Salome library [23] for mesh generation through its Python interface. Salome allows for interactive geometry creation and meshing and then the export of a Python script to automate model creation. The output file created by the script was in MED file format [24], which is one of several available input file types of Code_Saturne. The model created was a cuboid with a *moving face* boundary in the xz plane at the maximum y value. The exported script was modified so that the number of mesh elements in the x, y, and z directions and the total length of the model's side in the x direction could all be specified. The y side total length was set as 1/2 of the x side length and the z length was set proportional to the ratio of elements in the x and z direction (i.e. z elements / x elements * x length). The model allows for simulation of shear driven cavity flow, where the shear force was specified in the Code_Saturne graphical interface and values are seen in Table 7. The script to create the model is available from the Damaris GitLab wiki [25].

Three models are used in the following experiments, each one doubling the number of hexahedral cells of the prior one. The models names and dimensions are shown in Tables 4 - 6.

Table 4: Model cell dimensions in each direction.

model name	cell length x	cell length y	cell length z
29M	0.8333	0.41666	0.8333
58M	0.8333	0.41666	0.8333
115M	0.41666	0.41666	0.41666

Table 5: Model number of cells in each direction.

model name	total cells x	total cells y	total cells z	number of cells
29M	120	120	2000	28.8M
58M	120	120	4000	57.6M
115M	240	120	4000	115.2M

Table 6: Model dimensions along each direction.

model name	total length x	total length y	total length z
29M	100	50	1666.67
58M	100	50	3333.33
115M	100	50	1666.67

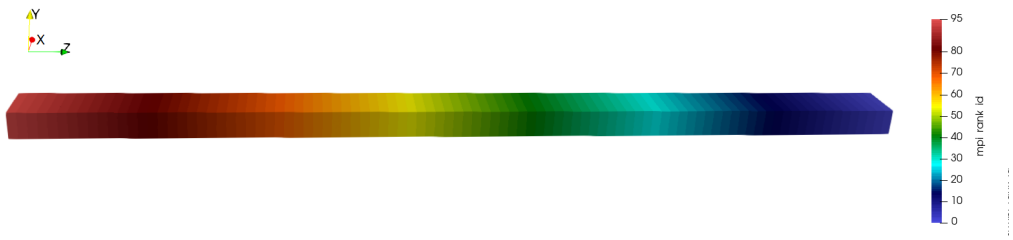

Figure 2. Example model 29M, mesh cells 120,120,2000 (x,y,z), showing typical block decomposition pattern, with colour mapped by MPI rank.

The partitioning of cells between MPI ranks used a simple block partitioning method, where the mesh model was dissected through the x-y plane in sections along the z direction into equivalent sized blocks as seen in Figure 2. It is thought that the simple, equal sized block partitioning should reduce the effect of data layout on parallel efficiency [26], so will reduce the influence of model differences on timing results.

Code_Saturne Model Paramaters

The values used in the Code_Saturne *setup.xml* input file are shown in Table 7.

Table 7: List of physical property values used by Code_Saturne in the simulation.

Property	Value	Units
Total pressure	101325.0	Pa
Time per timestep	0.1	s
Density	1.0	kg/m^3
Dynamic Diffusion	0.01	
Molecular Viscosity	0.001	$Pa \cdot s$
Shear force	0.1	m/s (along x)
Boundary conditions on moving_face	dirichlet=1 in x direction and 0 for all other walls and directions	

Post processing recording was only enabled for Pressure and Velocity field variables.

Issues found with model creation, partitioning and particular geometry.

Three issues were encountered during this work that were not thoroughly investigated due to time constraints.

- An initial 115M element model with 8000 cells in the z direction was found to have a serious performance degradation within Code_Saturne, which is thought to be due to the very high aspect-ratio of the model. Because of the performance degradation, the z=8000 cell model results are not presented and a model that increased the x cell number to 240 was used.

- Attempts at producing a 480 M cell model using the Salome script failed, even when using a high-memory (4 TB RAM) node. The reason for this is unclear as the amount of memory required was computed as approximately 33 GB, well under the capacity of the server.

- Trials were carried out with PT_Scotch partitioning; however, these runs were more likely to fail for reasons not investigated so this partitioning method was not followed.

2.5. Process Placement

Process placement on multi-core nodes is an important part of performance tuning and optimisation of parallel codes. When Damaris is used in dedicated core mode, the Damaris server cores are sharing node resources with the simulation computational ranks. Due to the large number of available cores per node and their NUMA arrangement in modern node architectures, the number and the placement of Damaris cores are parameters that need to be tuned to find a favourable setup. This setup should minimise the impact of the Damaris cores on the simulation processes and maximises the number of resources that Damaris cores could use. A further placement optimisation choice is to localise the I/O cores on NUMA nodes that also have network hardware, such as Infiniband cards attached. This optimisation was not explored in this work. No use of hyperthreading was trialed, as prior studies have shown the the memory bound nature of the Code_Saturne and the low memory bandwidth per core of the AMD Rome CPUs is a mix that is unlikely to benefit from further computational threads [21, 17].

The placement and pinning of MPI ranks to cores is a not well standardised task and in the following experiments it was carried out using a utility named `omplace`. This tool is part of the HPE MPI MPT library [27]. The tool is invoked on the (`mpirun`) command line and has a syntax to specify a start core (or more accurately a hardware thread), end core, step size (`st`) and block size (`bs`) (within a step). The syntax includes `-nt` to specify the number of threads each process will need a CPU allocation for, and `-ht spread` to indicate not to allocate on second hardware thread resources (i.e. hyperthread) until all primary hardware threads are allocated. The syntax has limitations in that `-nt` can only be specified once, even though the syntax can have multiple sections with differing step and block sizes.

Damaris has a process to core allocation policy that allocates the final MPI ranks on a node as the dedicated cores[1]. So, for example, if a user wants 4 dedicated cores per node and will be running 32 MPI ranks per node, then the first 28 MPI ranks will run the simulation and the final 4 will run asynchronous processing, as in the example in Listing 1 and Figure 3 shows. Damaris has another constraint in that the number of simulation ranks must divide exactly by the number of dedicated cores.

Segregated and Spread Dedicated Core Placement

Two placement options for dedicated cores were trialed and referred to as either *segregated* or *spread* process placement. The examples that follow in Listing 1 and 2 are chosen to show the differences between the placements and the features of the `omplace` syntax used. The choice in the example of using of 32 ranks and 3 threads is guided by results shown below (i.e. Fig. 5), with only three threads being chosen per rank so as to keep a core per rank free for use by Damaris ranks.

The *segregated* placement was implemented in a similar way as how Damaris allocates the dedicated cores from available processes. This method would reserve cores at the end of the CPU/rank range to run the dedicated cores. This placement strategy used a single `omplace` request similar to what is seen in Listing 1 below. The listing shows two lines: the first for the PBS resource request would request 2 nodes, running 32 processes per node and the second to run the Code_Saturne job using `mpirun`, which includes the `omplace` request. The placement has a shorthand of 28.3.4 and n2, using the nomenclature described above.

```
#PBS -l select=2:node_type=rome:node_type_mem=256gb:mpiprocs=32:ompthreads=3

mpirun -np 64 omplace -vv -c 0-127:bs=3+st=4 -nt 3 \
                                  -ht spread ./cs_solver --mpi --omp 3
```

Listing 1: Segregated process request and allocation statement.

The *segregated* placement is visualized in Figure 3. The command in Listing 1 requests 32 MPI processes per node and will place the first 28 compute processes out of the 32 in strides of 4 over the cores 0-111 on each node. Then the 3 OpenMP threads per process would be placed in a block of 3 cores, with one core left in the stride of 4 as free. The 4 Damaris server processes (requested within the Damaris XML configuration file) would be allocated on cores 112, 116, 120 and 124, having 3 free cores per block as in our case Damaris does not use OpenMP for processing. This allocation method was wasteful when used with Code_Saturne that used OpenMP and was a lot less flexible in terms of where cores could be allocated. This inflexibility resulted in the *spread* placement option being developed.

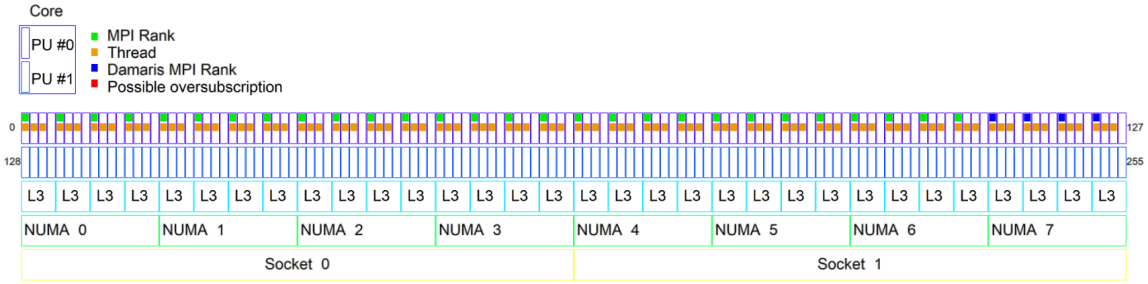

Figure 3. Schematic of HAWK Rome node CPU architecture, showing 'segregated' rank placement with 28 computational ranks and 4 Damaris I/O ranks. In the legend, PU stands for processing unit, and corresponds to a hardware thread or hyper-thread

The *spread* process placement method was trialed in order to make use of available cores remaining in a strided placement of the computational cores. An example PBS `qsub` resource request with corresponding MPI command line using `omplace` that was used for placement on the HAWK nodes cores is shown in Listing 2 below. The request asks for 2 nodes, running 36 processes per node and the 4 dedicated cores are requested within the Damaris XML configuration file. This placement has a shorthand of 32.3.4 and n2, using the nomenclature described above.

[1]This is hoped to be extended for more flexibility in a future release.

```
#PBS -l select=2:node_type=rome:node_type_mem=256gb:mpiprocs=36:ompthreads=3

mpirun -np 72 omplace -vv -c 0-126:bs=3+st=4,3-127:bs=3+st=32  -nt 3 \
                                    -ht spread ./cs_solver --mpi --omp 3
```

Listing 2: Spread process request and allocation statement

The resulting placement from Listing 2 is seen in Figure 4. In this case we have requested 36 processes per node `mpiprocs=36`) and the commands will place 32 compute processes in strides of 4 over the cores 0-126 on each node. The 3 OpenMP threads per process will be placed in a block of 3 cores, with one core left over in the stride of 4 as free or available to run a Damaris server processes. As all threads of the 32 compute cores will be allocated a CPU core within the first allocation (`0-126:bs=3+st=4`), the second block of 4 Damaris cores will be allocated from the second allocation section (`3-127:bs=3+st=32`). The stride of 32 will result in Damaris server cores being allocated on NUMA nodes 0, 2, 4 and 6, starting at CPU core 3 (0-based, so it is the 4th CPU core). The threads (0-2) of the Damaris core processes are then allocated resources, thread 0 being the root Damaris process allocated on a free core and threads 1 and 2 are allocated on top of previously allocated computational cores (shown as red blocks in Figure 4). This over-allocation will not result in any resource contention as the Damaris processing is single threaded in our implementation.

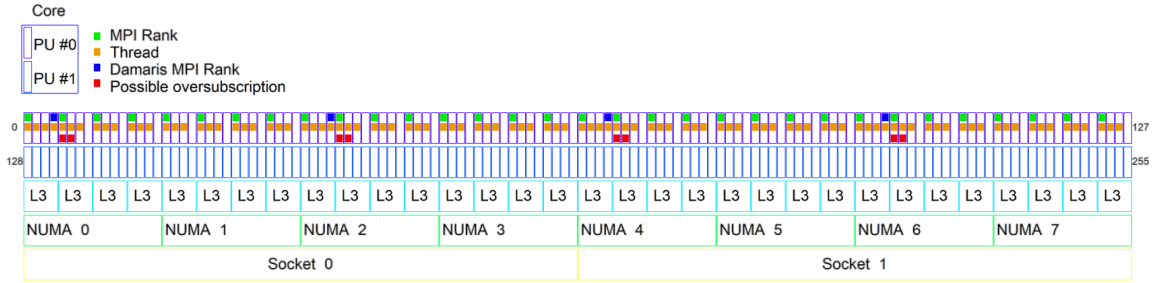

Figure 4. Schematic of HAWK Rome node CPU architecture, showing 'spread' rank placement with 32 computational ranks and 4 Damaris I/O ranks. In the legend, PU stands for processing unit, and corresponds to a hardware thread or hyper-thread

A single threaded version of Code_Saturne would not have extra cores allocated for threads (i.e. `-nt 1`), so the second `bs` placement element would be set to equal 1 and there would be no overlap of Damaris threads with computational threads. An interactive tool that produces the placement diagrams is available from the Damaris wiki site [25].

2.6. Paraview Catalyst Output

To create a range of computational loads to look at how they affect performance of the Damaris server cores, three versions of Paraview in-situ processing pipeline scripts were created. The pipeline name (how they are referred to in the text) and a description of their outputs are shown in Table 8 . The scripts themselves are available on the Damaris Wiki [25].

Table 8: The Paraview processing scripts used in subsequent experiments

Paraview Pipeline	Description of Processing
Single CSV	A single slice through the center of the model through the x-z plane with CSV output using the paraview.simple.CSVWriter and all fields output [28].
Single XML	A single slice through the center (velocity field only) and the full 3D data of a field (velocity only) using an XML based binary output paraview.simple.XMLMultiBlockDataWriter. This output format is stated to support parallel I/O and data compression [29].

Paraview Pipeline	Description of Processing
Double CSV	Two slices, one through the center and the other 10% of the depth of the model above the center (in the y direction), both slices are through the x-y plane, with CSV output (all fields output per slice)

The frequency of output of the data was also scripted and was typically set to output on odd iterations. To increase the computational requirements output was increased to every iteration (see Appendix A3). This results in 6 levels of computational intensity for the Paraview Catalyst workloads.

The above three scripts also output the full 3D MPI rank allocation before the 1st iteration, i.e. they output what cells were allocated to what ranks and used the Paraview XML binary output format.

2.7. Timing results

The timing results are presented in the following formats. Code_Saturne code runs an iterative solver, which causes some variation in per-iteration timing and the system (notably I/O) causes further jitter in the timings. Damaris is noted for its capacity to reduce the jitter due to I/O, so the spread in the timings was of interest, thus box and whisker plots are presented. The box and whisker plots show inter-quartile ranges of the per iteration computational timings of the 0.1 second simulation time steps of the Code_Saturne simulations. Outlier values are shown as the small open circles. Timing data are for single runs of the simulation. For CSV output data, each box represents 20 or 40 time points for odd iteration or every iteration output respectively. For the XML output, so as to reduce storage overheads[2], the number of iterations per run was reduced to 20. So, generally 10 time samples are represented per box plot for XML output runs.

Damaris server times are shown as large crossed circles. The values are the time between the first and last server EndOfIterationCallback() function calls, the values of which are stored in log file records and then averaged over the number of iterations. The results in section 'Damaris Server Log Analysis' shows time-series data of the Damaris server cores which are also computed from Damaris server log file timestamps. The values presented are times that server cores are not processing, i.e. they are server 'free time'. This is computed from the difference between the finish time of the previous iteration and the start of the next iteration.

Occasionally runs showed very high spread in the timing data and were re-run. The reason for the excessive jitter was unclear. Inter-node variation in timing was not tested rigorously; however, it is thought to be low as seen from some repeat measurements.

When Paraview output occurs only on odd iterations, the even iterations give the iteration timing without I/O, so they are displayed separately (usually in green)

3. Results

Overview

Section 3.1 gives an analysis of source code changes required for the integration with discussion of caveats to watch out for during integration. Section 3.2 show the initial experiments that were conducted using the segregated process placement. Two sets of results are presented, the first looked for an optimal combination of MPI ranks and OpenMP threads per rank to find a minimum in the per iteration runtime. The results are validated when looking at strong scaling results are shown for the larger 115M. Section 3.3 then shows results for the spread placement method. An increased complexity Paraview output (the single XML output I/O workload) running the 115M element model shows how the Damaris log timing analysis is used to check the efficiency of the Damaris core processing. A second study shows strong scaling performance differences due to differing Paraview pipeline I/O complexity using the 32.3.32 process placement configuration. Following this, a Damaris server log file timing analysis provides a view of the Damaris server timing data as a time-series plot and shows how the free time profiles change with added Damaris server core resources. Weak scaling data for the CSV and XML Paraview pipelines are then presented. The differing output pipelines show different scaling characteristics and reconfirms the

[2]These runs stored full 3D field data sets, along with the 2D slices.

differences as seen in strong scaling timing section. To end up, Section 3.4 presents results from the use of dedicated nodes, bypassing some of the placement issues and obtaining good performance.

3.1. Code changes

Tables 9-11 give some statistics on how many changes were required to support the integration of Code_Saturne with Damaris. The three main additions that were required are: Modification of the build system to optionally link Damaris; C/C++ code additions for Damaris library API calls for initialisation and data passing; and an XML file that describes the data (fields and meshes) that can be (not necessarily needing to be) passed to Damaris.

The predominant build system addition was the `m4/cs_damaris.m4` file that can be reused in other projects that require similar testing for the presence of Damaris.

The Code_Saturne FVM library architecture made integration relatively straight forward as there are standardised structures requiring function pointers to the I/O routines. Most of the C code added was in `base/cs_base.c` (62 LOC) and in the FVM library files that were added (`fvm/fvm_to_damaris.cxx` and `fvm/fvm_to_damaris.h`) at 692 LOC. The LOC summaries also include the addition of an `-omp` command line flag and associated OpenMP code for setting the number of threads to the `cs_solver` executable.

The number of Damaris API calls required is seen in Table 11. Field data required 2 API calls per field, and currently only 3 fields are passed to Damaris. Passing the unstructured mesh data structures to Damaris accounted for the most calls, and is another reusable piece of code that could be integrated into the Damaris API in the future.

One thing to look out for in a code is how it uses `MPI_COMM_WORLD`, as this communicator has to be replaced with a sub-communicator returned from Damaris to be used in any further simulation side communications. Code_Saturne did not have any issues in this regard due to its prior capability for use in coupled code, multiple program multiple data (MPMD) mode, using a sub-communicator in all MPI communications.

Finally, the XML file used contained 70 LOC and can be broken down into 25 LOC being needed for the unstructured mesh definition, and 3-4 LOC being required per field variable. The remaining 36 LOC is mostly constant and used for switching functionality, such as numbers of dedicated cores and setting paths to ParaView Catalyst scripts.

Table 9: Changes to Build System

Files		LOC	
modified	added	modified	added
5		3	25
	1		142

Table 10: Changes to C/C++ code

Files		LOC	
modified	added	modified	added
6		3	146
	2		692

Table 11: Damaris API Calls

Damaris Setup	Field Passing	Unstructured Mesh Passing
6	7	20

3.2. Segregated Process Placement

Use of Hybrid MPI-OpenMP Processing In this section we present the performance of Code_Saturne when run in hybrid MPI-OpenMP mode vs pure MPI modes. The hybrid mode is where

each MPI rank uses multiple OpenMP threads to carry out required work. The hybrid configuration can potentially improve performance of Code_Saturne when using low numbers of OpenMP threads per MPI rank [13]. The 29M element model was used for the tests and a range of MPI-OpenMP configurations were run, with and without the use of 1 or 2 Damaris cores for I/O. The Paraview single CSV slice pipeline script was used. The Damaris dedicated cores were allocated to CPUs in range of 116-127 (i.e. segregated placement)

Timing results for a 4 node configuration are shown in Figure 5. The results show that using less computational ranks per node can improve performance. The time per iteration values for the non-Damaris simulations (in red) at 128.1.0, 96.1.0 and 64.1.0 rank configurations show a continued reduction in time per iteration and then lowering the MPI rank count further while simultaneously adding extra OpenMP threads further improves performance. Reading from the graph, it is seen that having 32 ranks per node with each rank using 3 threads and 1 Damaris core (configuration 31.3.1) was the performance optimum. There were similar trends for both the Damaris and non-Damaris configurations. The 32 ranks per node placement equates to one MPI process per available L3 cache CCX, so the performance improvement is not surprising as OpenMP threads will have good data locality. The Damaris configurations (blue and yellow) all show better overall performance as the Paraview processing is completed asynchronously to the simulation computation. Further results for other node counts are shown in Appendix A (Figures 12-13).

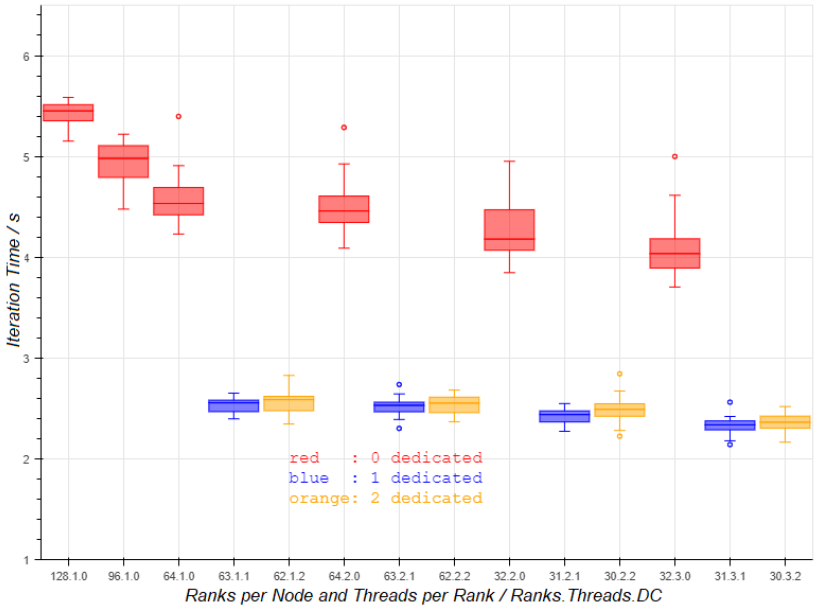

Figure 5. 29M model, 4 nodes, testing ranks per node and threads per rank, segregated Damaris core placement. DC in x axis label refers to 'dedicated cores'

The effect of using 1 vs 2 Damaris dedicated cores per node is also seen in Figure 5 and it is seen that adding the second Damaris core increases time per iteration slightly, although consistently (see also Appendix A). The reason being is that as extra cores are added in the segregated placement setup, the less cores are available for simulation ranks.

Strong Scaling 115M Model These results are shown to reaffirm the choice of optimal hybrid MPI-OpenMP configuration in the larger model (115M) and over extended node computations. Timing results in Figure 6 show a good strong scaling response. The 32 MPI process placement with three OpenMP threads has a slight advantage but only at higher node counts ($>$ n2). The 115M system would not run on a single node, ostensibly due to reaching node memory limits. The timing data in green are for the even iterations where no processing is done by Paraview. It shows that the overhead from using Damaris processing on the simulation computational runtime is low, as it simply consists of copying of data to the Damaris server shared memory areas.

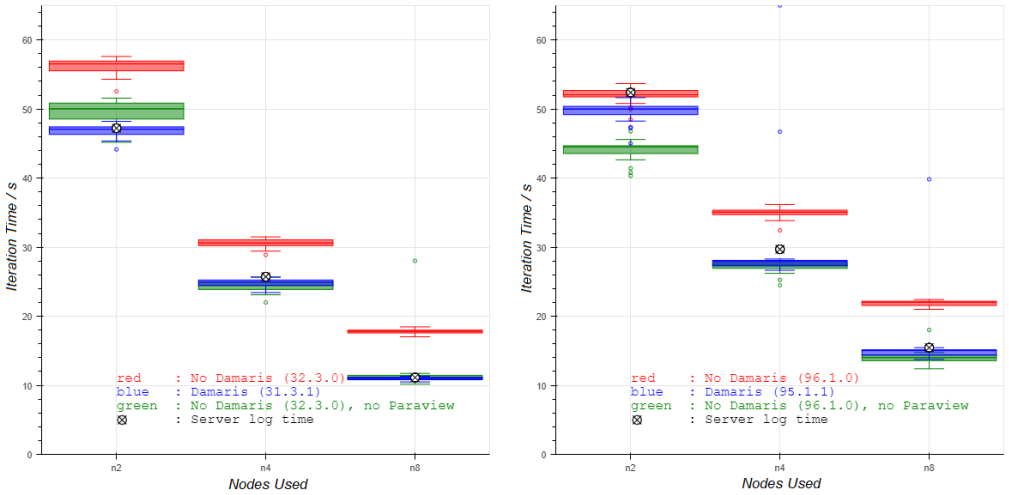

Figure 6. Strong scaling 115M model, single CSV output, 32 processes per node and 3 threads per process (left) and 96 ranks 1 thread per process (right)

The Damaris server iteration times (black crossed dots seen in Figure 6) show that the simple CSV output is not overloading the Damaris core, even though only a sinlge core has been allocated to the Damaris processing.

3.3. Spread Placement of Dedicated Cores

Increased Complexity Paraview Output The effects of changing the number of dedicated cores was investigated using the high complexity XML output on the large 115M model and results are seen in Figure 7. The Paraview processing produces a much larger amount of data that the single (or double) CSV slice scripts and hence has an (assumed) larger processing requirement. The black crossed dots on the graph are important parts to note as they are the average time per-iteration spent by the Damaris server cores processing the Paraview pipeline. The values reveal that the Damaris server cores are not keeping pace with the simulation processes (i.e are overloaded) until 32 Damaris cores are added to the configuration.

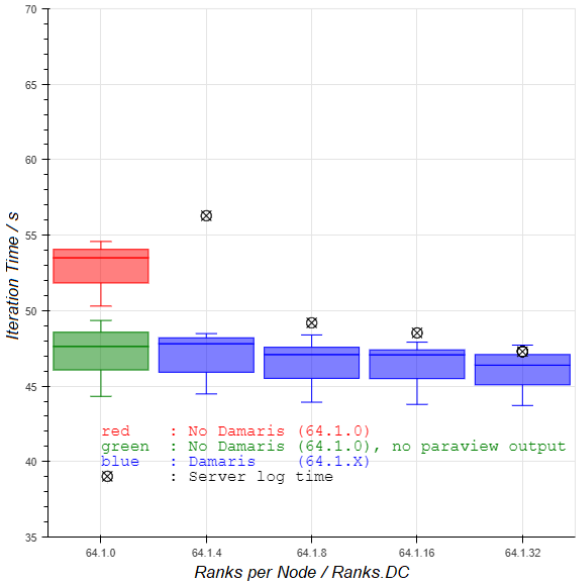

Figure 7. 115M Element model, XML VTK Output, 2 nodes, 64 MPI computational ranks per node, showing the effect of adding dedicated cores (DC).

Strong Scaling Timing A comparison of the three Paraview in-situ processing pipelines of differing complexity are shown in Figure 8 for the 58M element model. The Paraview pipelines were set to output on every odd iteration and the timing for these iterations, that are not using Damaris cores, are seen in red. The even iteration times (i.e. no Paraview output) are seen in green and the Damaris enabled pipeline iteration times are shown in blue. All processing is done with 32 MPI (processing) ranks per node, each with 3 OpenMP threads and either 0 or 32 Damaris server cores, using the spread process placement (i.e. 32.3.0 or 32.3.32 configurations).

In all three pipelines the Damaris enabled processing shows an advantage over the non-Damaris processed time results. At higher node counts, the speed-up is over 50%. The difference in the single vs double CSV outputs are obvious, as the non-Damaris enable times look to double when the amount of work performed doubles. An important thing to note is that the Damaris server times (large black crossed circles on the graphs) do not move much above the timing for the computational iterations (in blue). This means the Damaris server cores are not being over-allocated with work. This is the case except for the double CSV output pipeline at node counts n4 and above, where the Damaris servers are taking a greater time than the computational processes. This can be compared with the XML pipeline that does not show the discrepancy, even though it is outputting a much greater volume of data. The improved performance is apparently due to the scalability of the VTK binary XML writer used with a parallel file system as compared with the VTK CSV writer. The small but consistent reduction in iteration time of the no Paraview output iterations (the even iterations of the non-Damaris enabled code, shown in green) over the Damaris enabled code (shown in blue) are seen in these results. The overhead is likely caused by the memory copy from the Code_Saturne I/O routine to the Damaris server. This overhead could be removed by using the Damaris 'zero-copy' memory allocation methods; however, this method of memory allocation has not been implemented with the Code_Saturne integration.

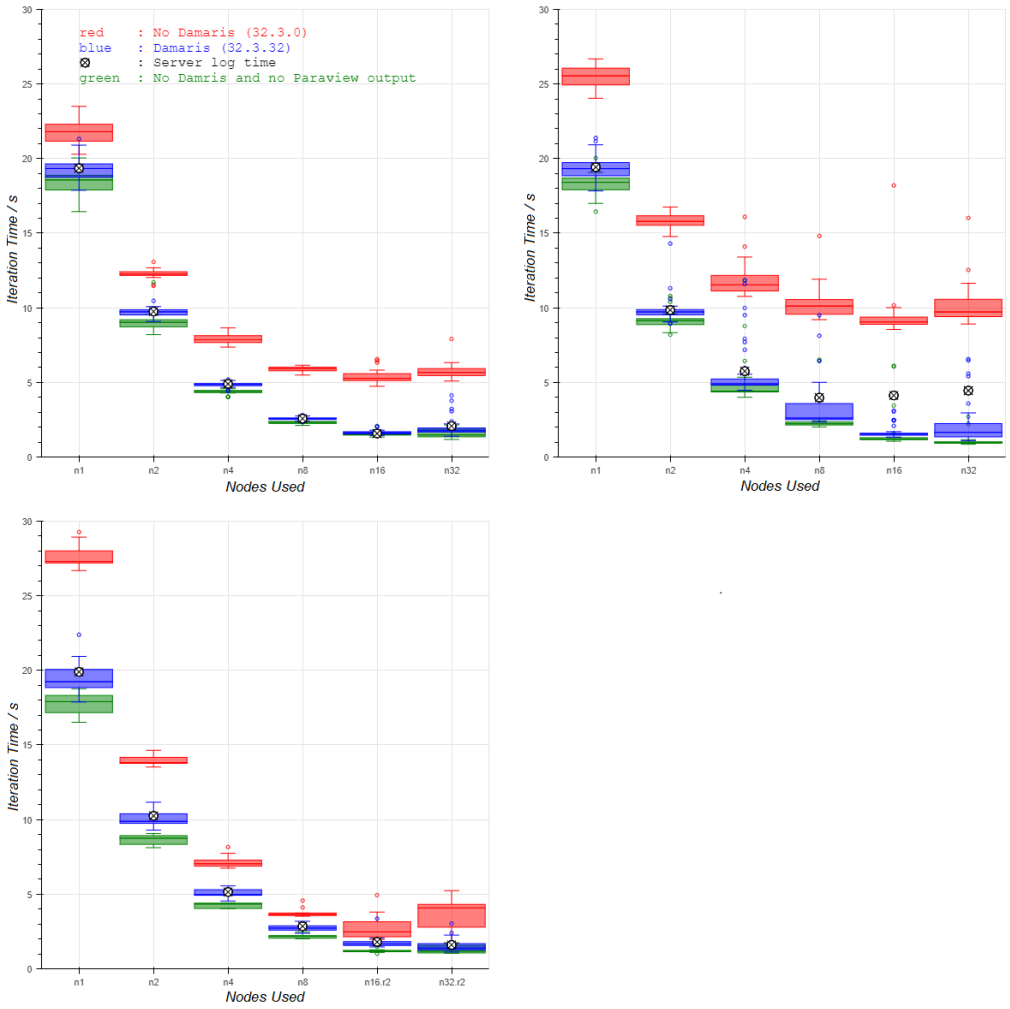

Figure 8. Strong scaling 58M model, single CSV output (top left), double CSV output (top right), XML output (bottom left). All runs are using 32 processes per node and 3 threads per process

Damaris Server Log Analysis Damaris server processes log the time on entry and exit to their processing routine. Figure 9 shows a time series plot from multiple runs of the same system (1 node, 96.1.X, single CSV cross section output using 29M model) with changing the number of Damaris server cores (X) used for asynchronous processing. The traces show an interesting effect due to the large I/O activity caused by the output of the MPI rank data on the 0th iteration. This output causes a high load on the Damaris servers and results in the computational cores completing multiple iterations while the Damaris servers complete the first iteration work. This results in a backlog of output data that the Damaris server cores must process which results in a lag time as seen in Figure 9 before the server iterations have any free time for processing. The server core free time values are shown to increase with number of Damaris server cores, which is expected when more processors are added to the system, the amount of work per Damaris server process is lowered. The peak in the data at iteration 13 is caused by the Damaris servers shared memory being filled so the server process also spends time freeing memory during the processing routine of that iteration. The server log analysis carried out does not compute the free time correctly in this case.

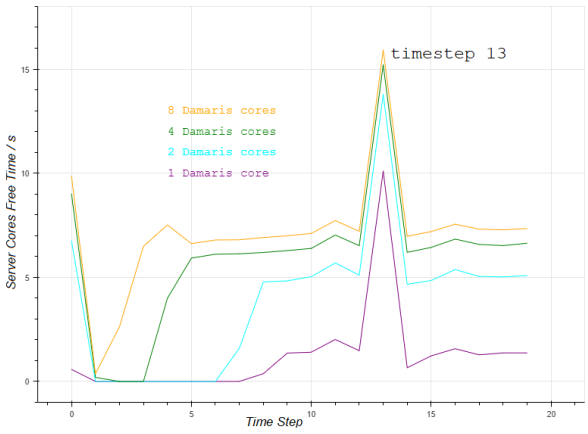

Figure 9. Timeseries showing changes due to the number of Damaris server cores, showing server free time (i.e. the time not processing). 1 node, 29M model, 96 processing ranks, 16 GB shmem.

Weak Scaling Figure 10 shows further interesting effects of the two processing pipelines (Single CSV vs Single XML). Both pipelines present similar timing results for the 3 model sizes, with the Damaris enabled simulation being more efficient than the non-Damaris capable version. Both Paraview pipelines show a big step in timing per iteration when run with the 115M sized model. The exact reason for this has not been determined; however, the difference possibly stems from the difference in the number of x dimension cells which would increase the number of surface cells needing to be shared between ranks as ghost cells. Another striking difference between pipelines is seen in the non-Damaris capable timing results (in red) of the CSV output pipeline. The timing per iteration of this pipeline is seen to increase proportionally to the data set size, whereas the time difference between the Damaris iterations and the non-Damaris iterations of the XML output stays mostly constant. This is attributed to efficient use of the parallel file system and the binary formatted and file-per-process output capability of the XMLMultiBlockDataWriter.

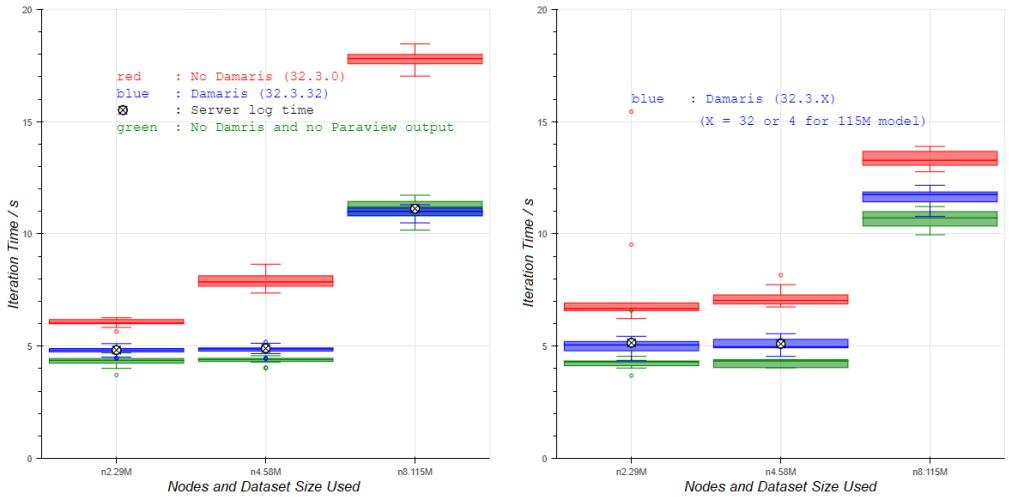

Figure 10. Weak Scaling, 29M model 2 nodes, 58M model 4 nodes and 115M model 8 nodes. Single CSV output (left) or Single XML output (right), all models use configuration 32.3.32 except the 8 node XML results, which used 32.3.4

3.4. Using Dedicated Nodes

Dedicated Nodes The ability to use dedicated nodes (either one or more) that are separate to the simulation computational nodes is another feature of the Damaris library. To look at how using a dedicated node can help offload excess I/O, a relatively computationally expensive Paraview pipeline was used, with the double CSV slice script outputting on every iteration. Figure 11 shows similar profiles as seen previously; however, the Damaris server timing results of the higher node count simulations are all a lot closer to the per-iteration time of the computational processes. This is indicating that even though

dedicated cores are scaling in number as the number of nodes increases, that having dedicated nodes (in this case a single one) that are specialized in I/O processing benefit the high load Paraview processing.

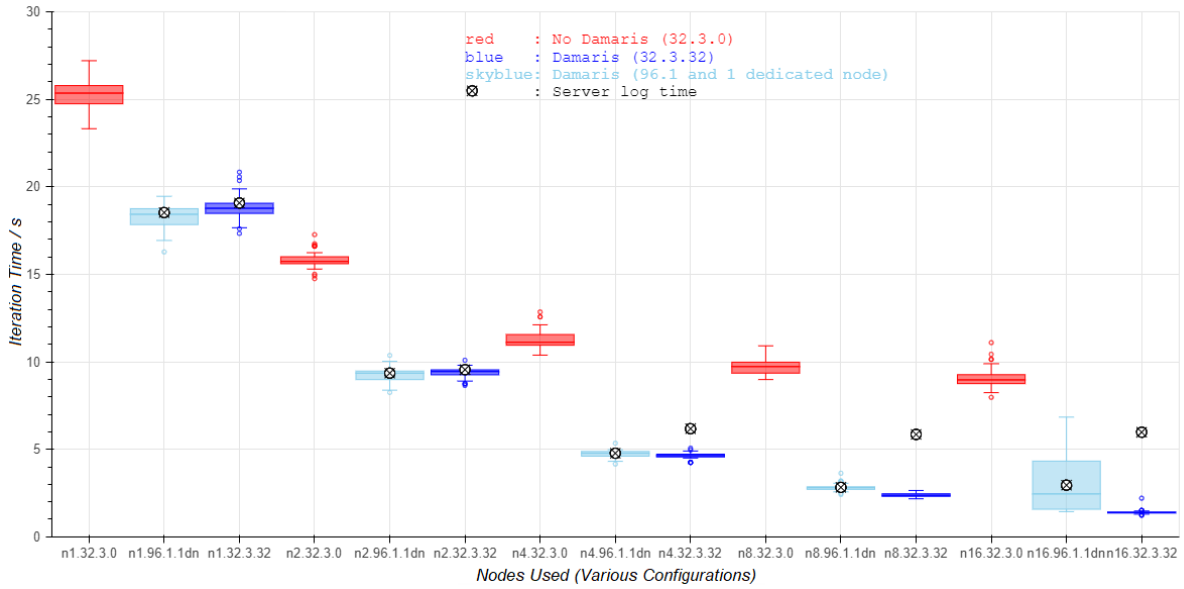

Figure 11. Dedicated cores compared with dedicated node, strong scaling 58M model, 2 slice CSV output every iteration, system setup is 32.3.0 no Damaris (red), 32.3.32 for dedicated cores (blue) and 96.1.1dn dedicated node (light blue)

4. Discussion

Damaris asynchronous processing work balance.

The various trials have tested various rank placement methods for the Damaris cores. The extension of the work to use of the dedicated node configuration also showed how the system may keep up with processing of the more complex pipelines, particularly at high node counts and lower work per computational rank.

As seen through the study on process and threads per process counts (Figure 5), the memory bound nature of Code_Saturne leaves multiple cores available in the node that cannot be of benefit to the simulation computation. This allows multiple possible resources for Damaris to use for its asynchronous tasks, and was found to levy negligible affect on the performance of the computation part of the code. Integration of Damaris with software that tends toward being compute bound may result in less opportunity for spare CPU resources and may detract from the performance of the code to a greater extent. This is one use-case of using the Damaris dedicated node configuration.

When using Damaris for asynchronous I/O it is seen that it is important to make best use of the finite time available. We see that choosing an in-situ visualization method that is efficient, such as the Paraview XML writer, is important and will improve efficiency, particularly at higher node counts. When high loads are foreseeable then the Damaris dedicated node configuration should be considered. This configuration has the ability to scale; however, this has not been assessed thoroughly in this work. In-situ processing can help reduce the post-processing requirements of larger workloads; however, data such as large restart files could possibly over-extend asynchronous methods. For heavy I/O loads, optimisation of the final I/O data movement to disk (i.e. Lustre partitioning) is still going to be important.

The Damaris shared memory buffer size.

The shared memory buffer size used by Damaris may also need to be tuned to find the balance between extra memory use by Damaris and the amount of memory available for the computation. The higher the amount of memory available to Damaris the better the ability to hide irregularities in I/O processing. If regular high I/O load is expected then the buffer should be increased if possible. User freeing of used shared memory should also be implemented (using plugins) as it can reduce the time required when Damaris has to clear the whole memory buffer when it is full (i.e. reduce the spike shown in Figure 9).

Limitations

To complete this work there has been support added for using unstructured mesh geometry types within Damaris. Code_Saturne has multiple advanced CFD modeling functionalities of which only a simple shear driven flow model has been tested. The Damaris implementation currently has limitations and untested use cases as outlined here.

- The VTK_POLYGON and VTK_POLYHEDRON mesh sections are not currently supported. This may affect multi-mesh models that are 'melded' to form a single mesh or other more unusual (or not) use cases where free-form geometry mesh types are required.
- Use of multiple meshes has not been tested.
- Use of Code_Saturne coupled computations has not been tested.
- Boundary faces have not been tested as output, only fluid zone fields are output.
- Damaris VisIt support with unstructured mesh data is not implemented.

5. Conclusion

The experimental results show Damaris to efficiently hide the I/O processing of various Paraview processing pipelines in Code_Saturne. In most cases the Damaris enabled version of Code_Saturne was found to be more efficient than the identical non-Damaris capable version when running the same Paraview pipeline. The efficiency is due to the asynchronous and parallel processing of I/O with the Damaris cores also possessing a time buffer that can help hide I/O variability. The efficiency gain is non-negligible and makes integration of asynchronous methods an attractive way to enhance computational codes. The number of code changes needed were not substantial and integration was aided by Code_Saturne's modular I/O library that is designed to allow easy switching between I/O options. Nevertheless, along with the improved performance comes some complexity in computational setup. The number allocated and the placement of Damaris cores are important parameters to optimise. Understanding the I/O load of an application using log file timestamp analysis or, for more detailed profiling analysis, using Darshan [30] is recommended. Damaris may hide I/O variability, however, finding ways to optimise I/O, such as with selected in-situ processing is important for the efficient running of large simulation systems at scale.

6. Future Work

The unstructured mesh integration with Damaris has been successful, although it is in a nascent state. Various improvements to the integration with Code_Saturne are targets for development as outlined in the 'Limitations' section above. Integration of unstructured mesh capabilities with the VisIt in-situ processing library is to be carried out.

The number of dedicated compute resources and the size of the shared memory buffer are the two main adjustable parameters of the Damaris library. The current Damaris mode of running can result in the under-allocation of these resources that can result in silent failures (i.e. no data output) and performance degradation when server cores cannot keep pace with the amount of processing needed. The modeling of the effect on run-time and development of diagnostics and tuning methods for these parameters for a running application are flagged as required future work.

The setup of process placement was found to be rather complex and as it is not governed by the MPI standard, requires differing methods as developed by the MPI system developers to carry out what is a regularly needed capability. The development within Damaris to set the placement by using either the hwloc library or a simplified MPI process mapping within the `MPI_Comm_split()` function and controlled through additions to the Damaris XML configuration file will be looked into.

As for the Code_Saturne code, its iteration timer functionality does not take into account the asynchronous work that Damaris carries out, i.e. the timers can be finalized while Damaris servers are processing a backlog of data. Damaris log file analysis has allowed us to get the average time per iteration for the Damaris server cores; however, the ability to view the timing directly in log files in real time would be

useful for end-users. Integration of the Damaris I/O routines with the Code_Saturne QT Python based user interface would also be a good addition.

Acknowledgments

This work was financially supported by the PRACE project funded in part by the EU's Horizon 2020 Research and Innovation programme (2014-2020) under grant agreement 823767. The work was achieved using the PRACE Research Infrastructure resource HAWK supercomputer based at the High-performance Computing Center (HLRS), Stuttgart, Germany. Development and testing of the code presented in this paper was carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). The support of M Puskaric from HLRS, Germany to the technical work is gratefully acknowledged. The support of Y Fournier to the technical work via the Code_Saturne user forum is gratefully acknowledged.

References

[1] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O," in *CLUSTER 2012 - IEEE International Conference on Cluster Computing*, (Beijing, China), IEEE, Sept. 2012.

[2] M. Dorier, *Addressing the Challenges of I/O Variability in Post-Petascale HPC Simulations*. Theses, Ecole Normale Supérieure de Rennes, Dec. 2014.

[3] M. Dorier, M. Dreher, T. Peterka, G. Antoniu, B. Raffin, and J. M. Wozniak, "Lessons Learned from Building In Situ Coupling Frameworks," in *ISAV 2015 - First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (held in conjunction with SC15)*, (Austin, United States), Nov. 2015.

[4] M. Dorier, R. Sisneros, T. Peterka, G. Antoniu, and D. B. Semeraro, "Damaris/Viz: a Nonintrusive, Adaptable and User-Friendly In Situ Visualization Framework," in *LDAV - IEEE Symposium on Large-Scale Data Analysis and Visualization*, (Atlanta, United States), Oct. 2013.

[5] INRIA, "Damaris website." https://project.inria.fr/damaris/, 2021. Visited 2021-02-30.

[6] Lawrence Livermore National Laboratory, "Visit website." https://wci.llnl.gov/simulation/computer-codes/visit/releases/release-notes-2.10.0, 2021. Visited 2021-07-27.

[7] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. M. Favre, and P. Navrátil, "Visit: An end-user tool for visualizing and analyzing very large data," in *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, pp. 357–372, Oct 2012.

[8] The HDF Group, "Hierarchical data format version 5." http://www.hdfgroup.org/HDF5, 2000-2021. Visited 2021-05-17.

[9] U. Ayachit, A. Bauer, B. Geveci, P. O'Leary, K. Moreland, N. Fabian, and J. Mauldin, "Paraview Catalyst: Enabling in-situ data analysis and visualization," in *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ISAV2015, (New York, NY, USA), p. 25–29, Association for Computing Machinery, 2015.

[10] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit (4th ed.)*. United States of America: Kitware, 2006.

[11] INRIA, "Damaris GitLab repository." https://gitlab.inria.fr/Damaris/damaris, 2021. Visited 2021-07-17.

[12] F. Archambeau, N. Méchitoua, and M. Sakiz, "Code_Saturne: A finite volume code for the computation of turbulent incompressible flows - industrial applications," *International Journal on Finite Volumes*, vol. 1, pp. http://www.latp.univ–mrs.fr/IJFV/spip.php?article3, Feb. 2004.

[13] EDF, "Code_Saturne website." https://www.code-saturne.org/cms/, 2021. Visited 2021-01-30.

[14] High Performance Computing Centre (HLRS), "HLRS website." https://kb.hlrs.de/platforms/index.php/HPE_Hawk_Hardware_and_Architecture, 2021. Visited 2021-01-30.

[15] Message Passing Interface Forum, "MPI: A message-passing interface standard version 3.0," tech. rep., USA, 2012.

[16] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[17] Y. Fournier, J. Bonelle, P. Vezolle, J. Heyman, B. D'Amora, K. Magerlein, J. Magerlein, G. Braudaway, C. Moulinec, and A. Sunderland, "Multiple threads and parallel challenges for large simulations to accelerate a general Navier–Stokes CFD code on massively parallel systems," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 6, pp. 843–861, 2013.

[18] Y. Fournier, J. Bonelle, C. Moulinec, Z. Shang, A. Sunderland, and J. Uribe, "Optimizing Code_Saturne computations on petascale systems," *Computers & Fluids*, vol. 45, no. 1, pp. 103–108, 2011. 22nd International Conference on Parallel Computational Fluid Dynamics (ParCFD 2010).

[19] B. Lorendeau, Y. Fournier, and A. Ribes, "In-situ visualization in fluid mechanics using catalyst: A case study for Code_Saturne," in *IEEE Symposium on Large Data Analysis and Visualization 2013, LDAV 2013 - Proceedings*, 10 2013.

[20] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in hpc applications," in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pp. 180–186, 2010.

[21] O. W. Saastad, K. Kapanova, S. Markov, C. Morales, A. Shamakina, N. Johnson, E. Krishnasamy, S. Varrette, and H. Shoukourian, "PRACE Best Practice Guide 2020: Modern Processors," tech. rep., 2020.

[22] M. Seiz, P. Offenhäuser, S. Andersson, J. Hötzer, H. Hierl, B. Nestler, and M. Resch, "Lustre I/O performance investigations on Hazel Hen: experiments and heuristics," *The Journal of Supercomputing*, 2021.

[23] EDF, "Salome website." https://www.salome-platform.org/user-section/faq, 2021. Visited 2021-01-30.

[24] EDF, "Salome website." https://www.salome-platform.org/user-section/about/med, 2021. Visited 2021-05-17.

[25] INRIA, "Damaris integration with Code_Saturne wiki." https://gitlab.inria.fr/Damaris/damaris/-/wikis/Damaris-Integration-with-Code_Saturne, 2021. Visited 2021-05-20.

[26] Z. Shang, "Performance analysis of large scale parallel CFD computing based on Code_Saturne," *Computer Physics Communications*, vol. 184, pp. 381–386, 2013.

[27] "HPE message passing interface (MPI) user guide," Tech. Rep. 007-3773-039, 2020.

[28] Kitware, "Online Paraview documentation." https://kitware.github.io/paraview-docs/v5.8.0/python/paraview.simple.CSVWriter.html, 2021. Visited 2021-05-18.

[29] Kitware, "Online Paraview documentation." https://kitware.github.io/paraview-docs/v5.8.0/python/paraview.simple.XMLMultiBlockDataWriter.html, 2021. Visited 2021-05-18.

[30] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *TOS*, vol. 7, p. 8, 10 2011.

A Appendix

A1. Mixed MPI-OpenMP timing results

29M model on 1 and 2 nodes.

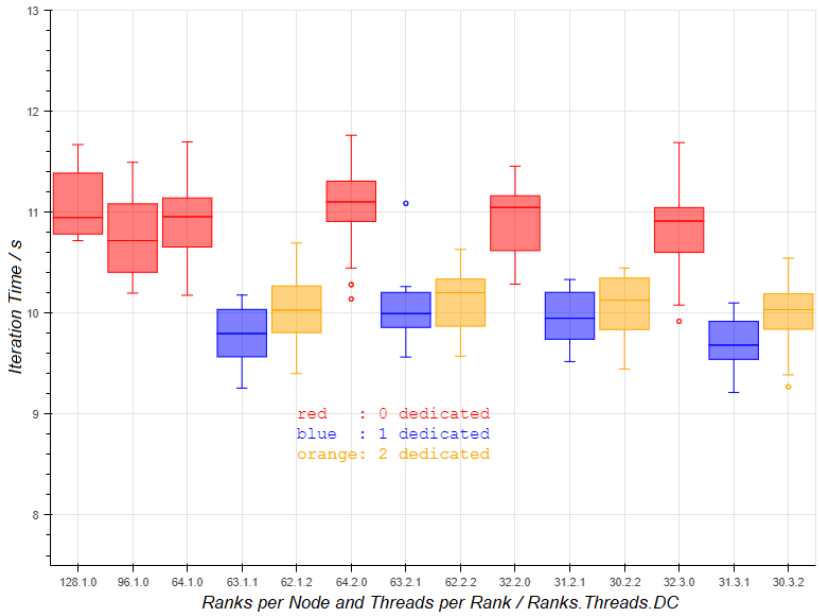

Figure 12. 29M model, 1 node, testing of threads per rank

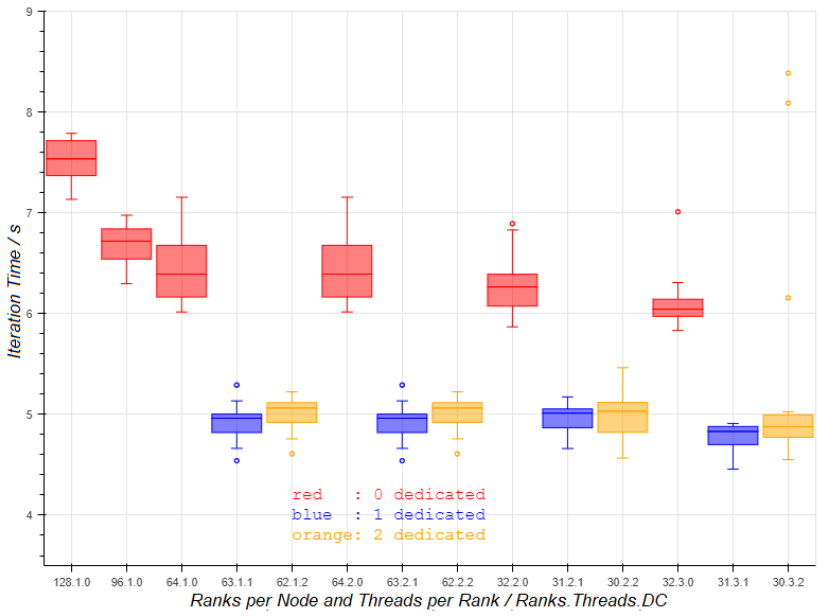

Figure 13. 29M model, 2 nodes, testing threads per rank

A2. Damaris shared memory use.

The Damaris XML configuration file alows a user to specify the amount of shared memory allocated (per node) to the Damaris server processes. Fig. 14 shows the effect on the strong scaling results of modifying the memory available between 16 GB and 32 GB per node. The n4.16GB example did not return data, possibly due to lack of output on the final iteration, which indicated a failing system, even

if other iterations completed without a problem. The results show that there is no conclusive change between the memory configurations, except perhaps the onset of unequal server vs computation iteration time at n2 problem distribution size.

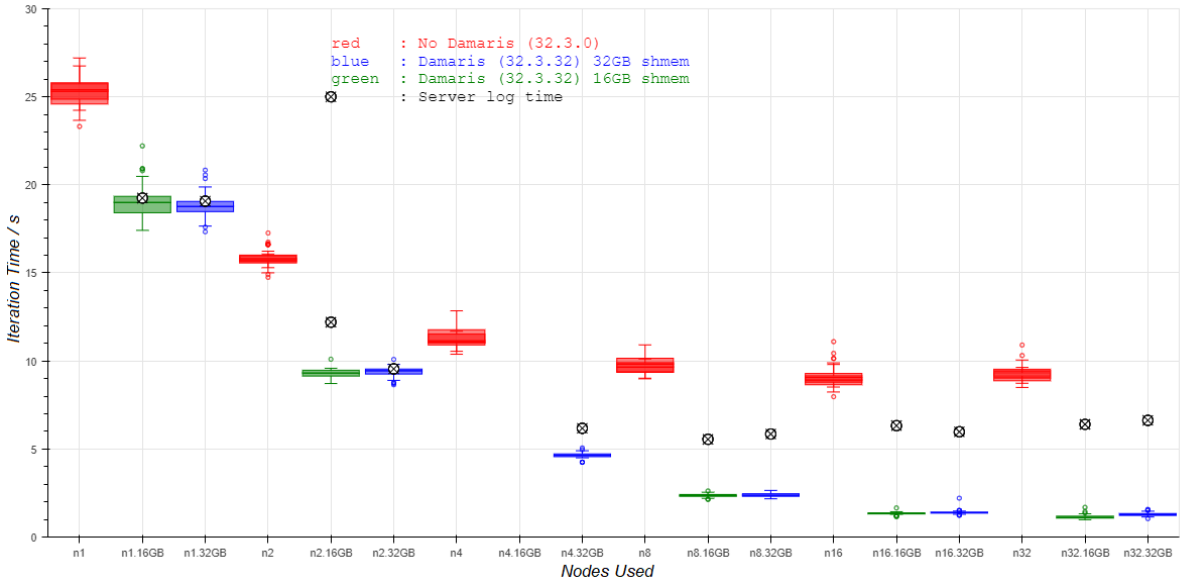

Figure 14. Strong scaling 58M model, Comparing shmem size 16 GM or 32 GB, 2 slice CSV output (every iteration), 32 proc/node and 3 threads/proc

A3. Changing Output Frequency

These results all use CSV Slice Output, and show the effects of changing the frequency of the CSV output. Fig. 15 shows changing frequency of output from odd iterations to every iteration. Fig. 16 shows changes due to changing the number of slices output, i.e. increasing the number of files output per iteration (either 1 or 2 slices) on every iteration. The effect on Damaris server iteration time is seen as rising time per iteration with the greater amount of processing required. The computational cores continue unaffected by server processing time. This is with the exception of the n4 result, which failed to produce a final iteration output.

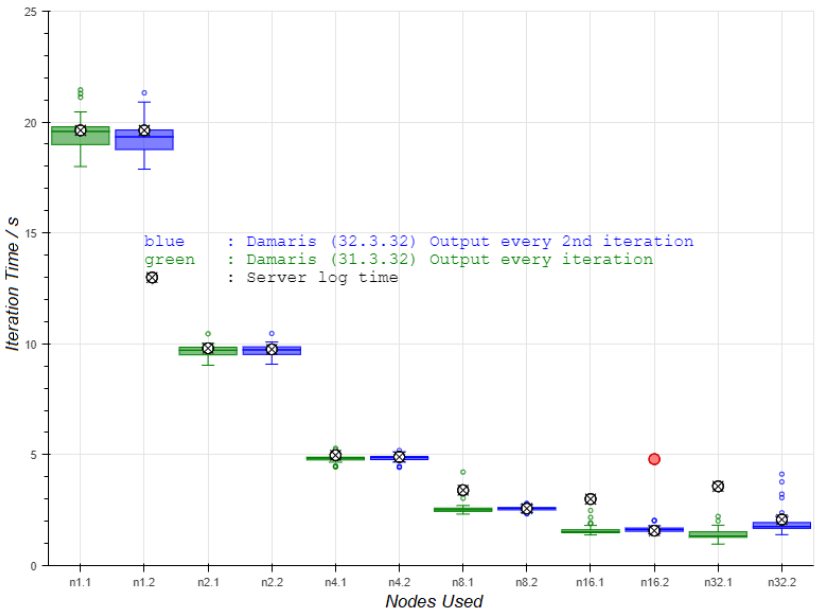

Figure 15. Strong scaling 58M model, 1 slice CSV output either every iteration (green) or odd iterations (blue), 32 processes per node and 3 threads per process. The red dot is Damaris server time for a repeat run.

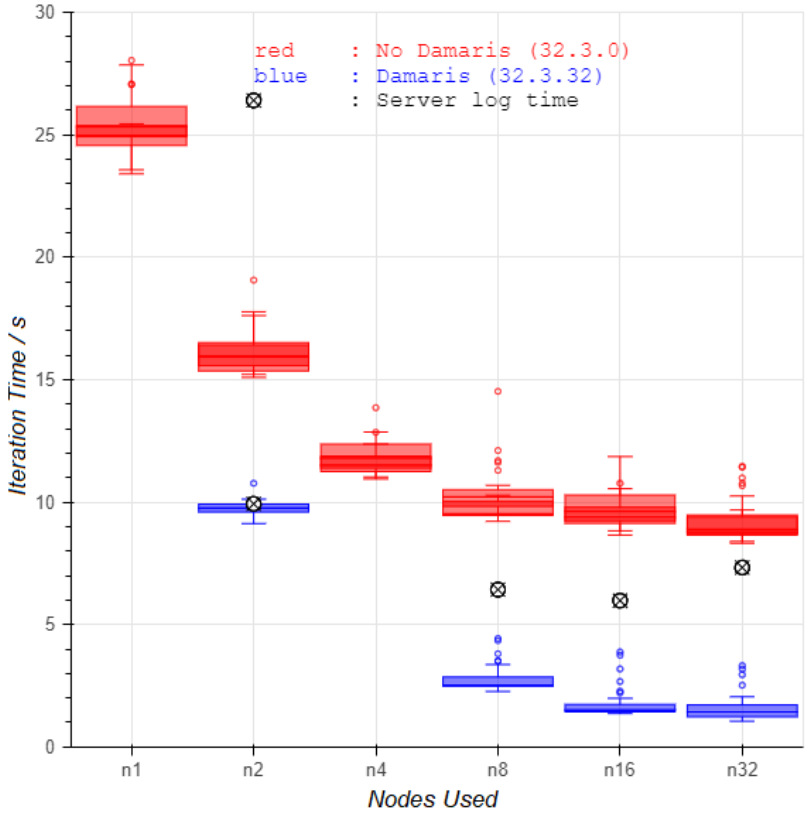

Figure 16. Strong scaling 58M model, 2 slice CSV output (every iteration), 32 proc/node and 3 threads/proc