H2020 FETHPC-1-2014

Enabling Exascale Fluid Dynamics Simulations
Project Number 671571

D2.4 – Final report on the ExaFLOW algorithms, energy efficiency and IO strategies

WP2: Efficiency improvements towards exascale

Document Information

Deliverable Number	D2.4
Deliverable Name	Final report on the ExaFLOW algorithms, energy efficiency and IO strategies.
Due Date	30/09/2018 (PM36)
Deliverable Lead	UEDIN
Authors	N. Johnson, UEDIN M.R. Bareford, UEDIN C. Cantwell, IMPERIAL A. Nielsen, EPFL N. Jansson, KTH N. Offermans, KTH J. Gong, KTH A. Peplinksi, KTH S. Jammy, SOTON J. Zhang, USTUTT B. Dick, USTUTT P. Vogler, USTUTT S. Sherwin, IMPERIAL M. Vymazal, IMPERIAL D. Moxey, IMPERIAL/ UNEXE
Responsible Author	N. Johnson, UEDIN, n.johnson@epcc.ed.ac.uk
Keywords	Efficiency; Efficient Implementation; Exascale; Energy Efficiency; Data compression
WP	WP2
Nature	R
Dissemination Level	PU
Final Version Date	30/09/2018
Reviewed by	Rupert Nash, UEDIN Michael Bareford, UEDIN Niclas Jansson, KTH Xavi Aguilar, KTH
MGT Board Approval	30/09/2018

Document History

Partner	Date	Comments	Version
UEDIN	14/06/2018	Skeleton	0.1
UEDIN	26/07/2018	Assemble all contributions	0.2
UEDIN	20/08/2018	Captions + refs to captions now sane	0.3
UEDIN	24/08/2018	Bibliography done so far	0.4
UEDIN	24/08/2018	Add code generation section	0.5
UEDIN	24/08/2018	Update to roofline section; author list done	0.6
UEDIN	29/08/2018	Add first intro/conclusion prior to initial review.	0.7
UEDIN	24/09/2018	Address internal review comments; add executive summary.	0.8
UEDIN	25/09/2018	Finalise pre-PMB checks.	0.9
KTH	30/09/2018	Final version	1.0

Executive Summary

This deliverable showcases the progress made in the ExaFLOW project regarding improving the scalability, performance and robustness of CFD software with a view to making it both ready for future exascale systems and suitable for use in commercial and industrial production environments.

We have concentrated on implementing the algorithms developed in work-package 1, such as compression of CFD data, in parallel with robustness and scaling improvements to areas such as IO and energy efficiency. For example:

- With regards compression, we see using an approach similar to that used for storing images in JPEG format, wavelet-based compression can result in a good balance between the time taken for compression and the reduction in data stored. Our method achieves similar results to the popular 7-zip algorithm but takes around 1/10th of the time.
- Our work on fault-tolerance is seen in the Llama library, which has been integrated to Nektar++ and ready for wider usage. We are able to recover from a failure of 28 cores in a 7,168 core simulation with no problems, and achieve a recovery rate of ~100GB/s by using in-memory checkpointing.
- We have improved the implementations of Adaptive Mesh Refinement (AMR) and Algebraic Multi-Grid (AMG) methods in the Nek5000 code, in the latter case, we have replaced the inbuilt library with an external library which has higher performance to leverage development effort being supplied there.
- We have improved the IO of the Nektar++ application in response to bottlenecks seen by the user community when running industrially relevant problems. By integrating a 3rd party library, SIONlib, from Julich Supercomputing Centre, we can again leverage development effort there, and improve our performance, achieving a 50% reduction in read times for a 5GB checkpoint in 6,144 cores.
- Also, we have developed a new communications library based on single-sided communication methods which is compatible with both Nektar++ and Nek5000, by offering the same API as the existing, MPI-1 based GS library. We can achieve an improvement of around 10% over the existing library, plus separating the communication library from the Nek5000 code allows others to write additional libraries targeted at new machines.
- With respect to energy efficiency, we see that we are able to reduce energy by around 20% in test-cases with only a 10% increase in run-time and our methodology for measurement allows tailoring of this trade-off on future machines to achieve a good balance between energy savings and runtime increase.
- We have explored alternative architectures, by showing that porting Nek5000 to use GPUs via OenACC directives offers a good solution to take advantage of accelerators. Also, partner code OpenSBLI, performs code-generation to target each hardware platform again allowing performant results on different machines.

- Finally, we have also developed a method of modelling the performance of codes by using an architecture independent roofline toolkit to understand where further development effort might be focused and where performance is limited by hardware performance.

We believe these improvements to the robustness of scalability of our co-design applications, and where applicable to other CFD applications which can benefit allow for a more efficiency usage of future exascale systems, regardless of their composition.

Contents

1 Introduction

This deliverable reports on the progress made in work-package 2 since the previous technical deliverable, D2.2 at project month 18, towards its aim of increasing the performance and efficiency of the co-design applications through improvements to algorithm implementations, libraries and systems configuration.

Currently, computation fluid dynamics applications do not scale to the point where they would make efficient use of an exascale machine, only of existing machines. To further the scaling, efficiency must be increased in many areas, algorithms must allow for greater scaling and decomposition, and implementations must do this sensibly and maximise the potential of codes at each phase of simulation, from hardware though libraries to solvers and post-processors.

Our ultimate aim to is to run industrially relevant, engineering quality CFD codes on exascale systems in a sustainable manner. For example, we do not currently take advantage of failure recovery and mitigation mechanisms, because the rate of node or core failure is too low to justify the overhead, however limited, of using mitigation techniques.

The first of two routes to improving performance is the efficient implementation of the algorithms from work-package 1. This material is presented in Section 2. As an example, we see that parallel preconditioning can either be implemented natively in the codes, or, by using a more efficient implementation from the hypre library, greater, parallelised, performance can be achieved. This improvement is dependent on the mesh used, but is applicable to many codes, as hypre is application agnostic.

The second of the two routes to improve performance is by considering systems factors; this work is detailed in Section 3. As reported in D2.3, the exact shape and configuration of any future exascale system is unknown, however some educated guesswork can inform the expected bottlenecks to be overcome to make effective use of such a machine. In Section 3.1 we see that changing the IO library can have either a positive or detrimental effect on overall runtime. This knowledge allows correct selection of the IO method and library to be made prior to large core-count runs. As with implementation improvements, these system-type improvements are tied to the application and test-case used, but the knowledge of the limitations and strengths of each method is application agnostic.

2 Efficient Implementations of WP1 Algorithms

In this section, we show the efficient implementations of algorithms developed as part of work-package 1.

2.1 Compression Algorithms

One of the hallmarks of high performance computing (HPC) is its ability to process complex numerical simulations by breaking down the intensive, computational workload into small, manageable junks that can be distributed onto highly parallel computer-clusters. This approach, combined with a steady increase in single core performance, has enabled the scientific community to study increasingly complex physical problems. As we approach the end of Moore's law, however, a significant gain in computing power may only be achieved by an increase in core counts. Yet, this path to exascale performance is blocked by an IO bottleneck. Modern high-performance systems are capable of producing and processing high amounts of data quickly, while the overall performance is oftentimes hampered by how fast a system can transfer and store the computed data. Considering that researchers invariably seek to study simulations with increasingly higher spatial and temporal resolution, the increase in core-count will consequently only exacerbate this problem [11].

Since most fluid flow problems are subject to diffusion, however, our numerical datasets are typically smooth and continuous, resulting in a frequency spectrum dominated by lower modes [11]. Thus, our best way forward is to use the otherwise wasted computing cycles by exploiting these inherent statistical redundancies to reduce the simulation file size. Since effective data storage is a pervasive IT problem, much effort has already been expended on refining compression algorithms. Prominent algorithms that allow for lossy compression can be found in the world of entertainment technology. In this context, Loddoch and Schmalzl [6][8] have extended the Joint Photographic Experts Group (JPEG) standard for volumetric floating-point arrays by applying the one-dimensional real-to-real discrete cosine transform (DCT) along the axis of each spatial dimension. The DCT coefficients are then quantized and encoded using a variable-length code similar to that proposed by the JPEG standard. Lindstrom [5], on the other hand, uses the fixed-point number format Q, which maps the floating point values onto the dynamic range of a specified integer type. A lifting based integer-to-integer implementation of the discrete cosine transform is then applied to the resulting integer array. Next, the DCT coefficients are encoded using an embedded coding algorithm to produce a quality-scalable code-stream. These compression algorithms are simple and efficient in exploiting the low frequency nature of most numerical datasets. The non-locality of the basis functions of the discrete cosine transform, however, will result in large scale blocking artifacts which heavily distort the dataset [1]. To circumvent this problem, we adapted the wavelet-based JPEG 2000 algorithm for volumetric floating-point arrays.

2.1.1 Taylor Green Vortex

A comparison between our wavelet-based compression algorithm (WBC), and the ZFP and 7-Zip encoder was carried out using a numerical simulation, with an in-house code, of a Taylor-Green vortex (TGV) decay at $Re = 1600$ (see Figure 1). The

Taylor-Green vortex represents a simple and well-defined hydrodynamics problem, characterized by its turbulent energy cascade. The flow field is initialized with an analytical solution containing a single length scale. The initial vortex quickly transitions into fully-turbulent dynamics, breaking down the vortices into increasingly smaller turbulent length scales until the turbulent energy is dissipated through viscous effects [4]. The vortex decay's broad turbulent scale spectrum allowed us to study the effects of length scale on the performance of the individual compressors. The evaluation was performed for the non-dimensional time-steps $t = 0, 2.5, 5, 7.5, 10, 12.5, 15, 17.5$ and 20. Each time-step contained the conservative variables ρ, ρu, ρv, ρw and ρE on a $nx \times ny \times nz = 260 \times 260 \times 260$ grid. The uncompressed file size of one time-step measured 703 MegaBytes.

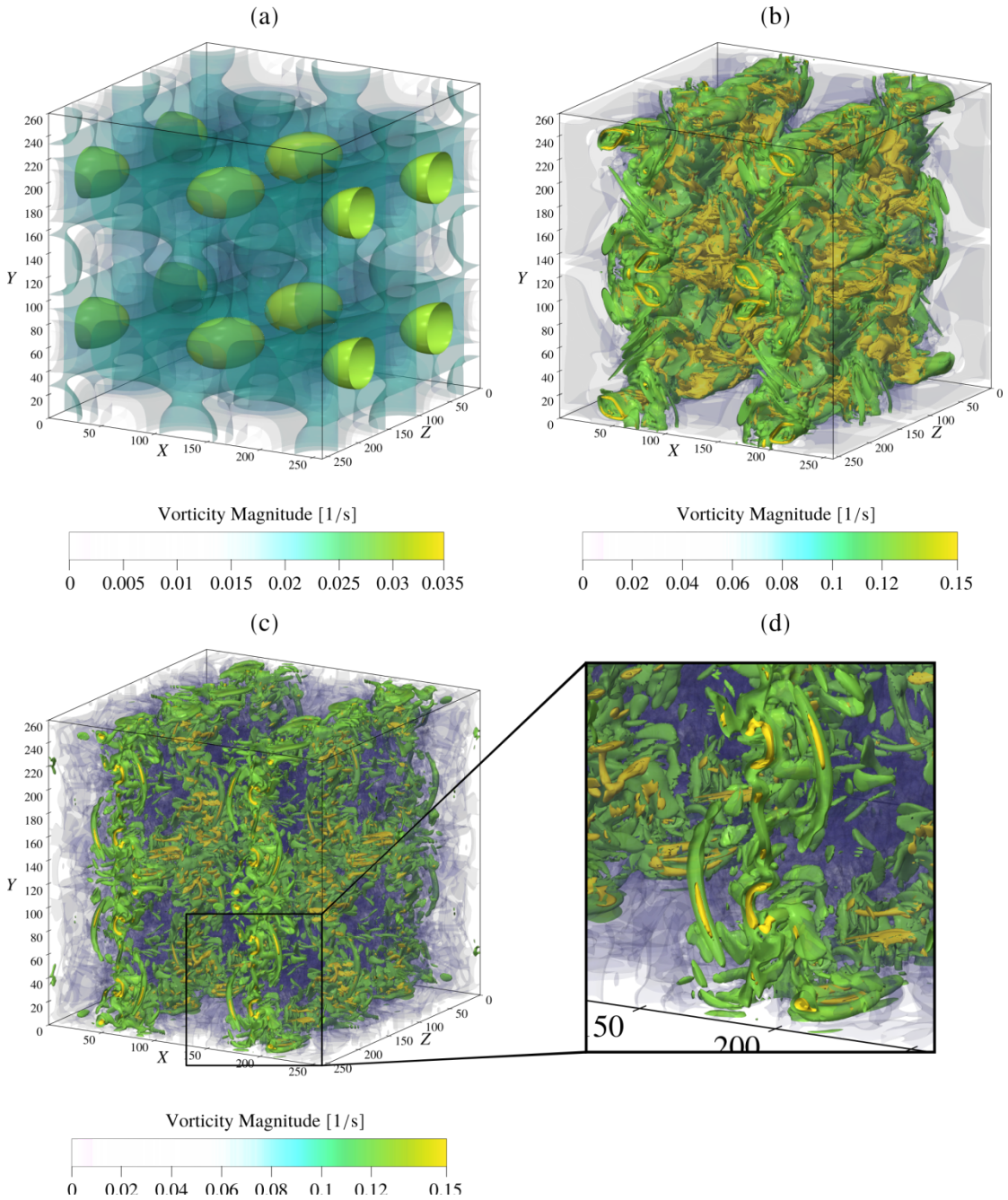


Figure 1: Visualization of the Taylor-Green vortex test case on a 256^3 grid. The vorticity magnitude of the decaying vortex is shown from top left to bottom left for the non-dimensional times t = 0, 7.5, 15.

Table 1 compares the compression ratio, compression time and peak signal-to-noise ratio for the WBC, ZFP and 7-Zip compressors. Multiple non-dimensional time-steps are listed to assess the effects of fluid structure size on the compression performance and induced distortion.

Table 1: Compression ratio, compression time and peak signal to noise ratio for the BWC, ZFP and 7-Zip compressor. The best results appear in bold.

Time-step	Compression Ratio			Compression Time (s)			PSNR		
	WBC	ZFP	7-Zip	WBC	ZFP	7-Zip	WBC	ZFP	7-Zip[a]
0	64.7	**65**	41.7	6.9	**1.7**	114.9	133.9	61.2	∞
2.5	**62.2**	62.1	6.0	4.4	**2.0**	160.4	123.6	60.0	∞
5	**61.6**	61.1	5.8	7.2	**2.0**	173.2	92.0	48.2	∞
7.5	61.0	**61.1**	5.6	8.8	**1.9**	164.0	73.8	44.3	∞
10	60.8	**61.1**	5.1	11.0	**1.8**	172.9	62.6	37.9	∞
12.5	60.8	**61.1**	4.6	11.4	**1.8**	172.8	63.3	40.6	∞
15	60.8	**61.1**	4.2	11.1	**1.8**	186.4	64.1	40.5	∞
17.5	60.8	**61.1**	3.9	10.7	**1.8**	183.9	65.7	43.0	∞
20	60.9	**61.1**	3.7	11.1	**1.9**	181.5	67.6	45.2	∞

[a] The PSNR for the 7-Zip compressor is always infinity due to its lossless nature

We found that all compressors offer good file size reduction for large-scale flow-field structures. However, the 7-Zip compressor is unable to provide significant compression ratios for datasets that contain small length scales. In contrast, both the WBC and ZFP algorithm can maintain large compression ratios throughout the time series.

The smallest compression time is achieved with the ZFP compressor, followed by our wavelet-based compression algorithm. This discrepancy can be attributed to WBC's more complex entropy encoding stage that has been derived from the JPEG 2000 algorithm. The 7-Zip algorithm, on the other hand, requires an excessive amount of time to reduce the file size of any time-step.

The defining difference between the WBC and ZFP is compression-induced distortion. While the WBC algorithm offers good reconstruction throughout, the ZFP codec suffers from large-scale blocking artifacts that decrease the PSNR of a decompressed file significantly. The 7-Zip codec offers perfect reconstruction for all time-steps due to its lossless nature.

To analyze the local distortion introduced by the lossy WBC and ZFP compression algorithms, we evaluated the relative error for the non-dimensional time-step $t = 10$ of the transient vortex decay. Here, the vortex decay has advanced to the point where most of the turbulent energy is stored in small-scale structures that are dissipated through viscous effects. The high-frequency information these small length scales contain will reduce the overall compression performance. This, in turn, requires an efficient compression strategy and aggressive truncation of the bit-stream to reach the specified compression ratio. The performance was

evaluated for the compression ratios *5:1*, *40:1*, *80:1* and *120:1*. The respective histograms for the relative error are shown in Figure 2.

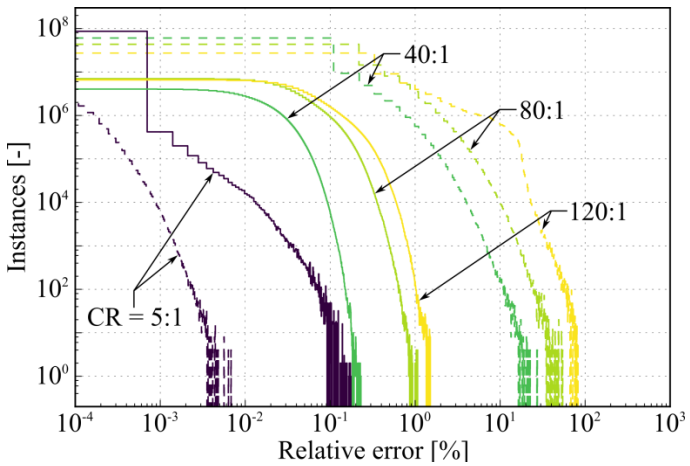


Figure 2: Histogram of relative error for the Taylor-Green vortex compressed with the WBC (—) and ZFP (- - -) algorithm. The compression ratios CR = 5:1, 40:1, 80:1 and 120:1 are shown for the non-dimensional time-step 10.

For the WBC encoder, the mean error for all compression ratios remains below *0.1%*. As expected, the histograms shift towards higher values for an increase in compression ratio. The smallest maximum relative error (*0.23%*) occurred for a size reduction of *5:1*, the largest maximum relative error (*1.4%*) for a size reduction of *120:1*. The biggest jump in relative error can be observed between the compression ratios *40:1* and *80:1*. In contrast, the infinite support of the discrete cosine transform and the minimalist entropy encoding stage of the ZFP codec result in heavily distorted fluid-flow structures for large compression ratios. The maximum relative error amounts to *27%* for the compression ratio *40:1*, *54%* for the compression ratio *80:1* and *80%* for the compression ratio *120:1*. Furthermore, the histograms exhibit an increase in the mean error, ranging from *0.3%* for the compression ratio *40:1* to *5%* for the compression ratio *120:1*. Merely the compression ratio *5:1* exhibits a smaller maximum relative error (*0.02%*) and mean error (*0.0002%*) for the ZFP algorithm. This is due to the choice of the fractional bit parameter value $Q_m = 28$ for the WBC encoder.

The evolution of the local distortion introduced by our lossy wavelet-based compression algorithm is evaluated using the relative error for the transient vortex decay. The histograms shown in Figure 3 mirror the evolution of the energy dissipation found in the original dataset. Both the maximum (5×10^{-3} %) and mean (8×10^{-5} %) error remain negligibly small while the vortex decay is in its initial stages. Most of the turbulent energy is stored in longer length scales that are easy to compress. However, the error histogram is shifted to higher values with an increase in the energy dissipation rate. The largest deviation can be observed at the non-dimensional time-step *10* with a maximum relative error of *2%* and a mean error of 8×10^{-2} %. Towards the end of the vortex decay, most of the turbulent energy has already been dissipated from the flow-field domain and thus an improvement in compression performance can be observed. For all time steps, the upper limit of the maximum relative error remains at *2%*.

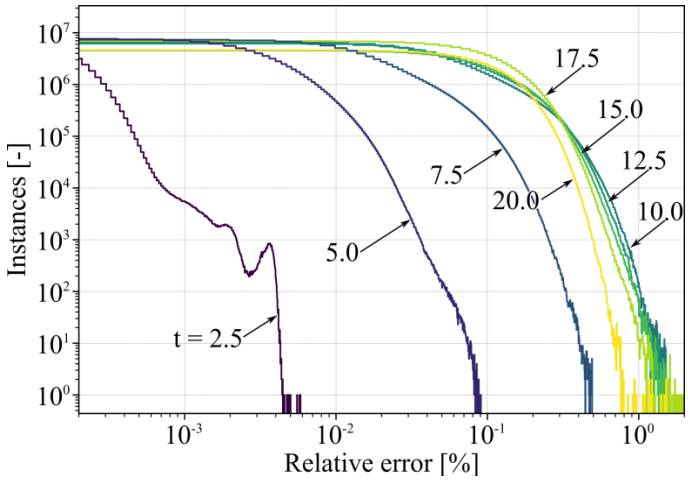


Figure 3: Histogram of relative error for the Taylor-Green vortex evolution compressed with the WBC algorithm. The non-dimensional times t = 2.5, 5, 7.5, 10, 12.5, 15, 17.5 and 20 are shown for a compression ratio of 120:1.

To gain insight into where the distortion is introduced in our numerical datasets, we compressed, suing the WBC method, the three distinct time-steps shown in Figure 1 with a compression-ratio of 120:1. The absolute error for the vorticity magnitude was evaluated for each data sample and normalized with the dynamic range of the vorticity magnitude. Figure 4 shows the decompressed, non-dimensional time-steps t = 0, 7.5 and 15. Isosurfaces for the absolute error are used to indicate regions of large, compression induced distortion.

Both time-step 0 as well as time-step 7.5 show a reasonably well reconstructed dataset with marginal deviation from the original file. This can be attributed to the large length-scale present in the dataset, which can easily be decorrelated by the discrete wavelet transform.

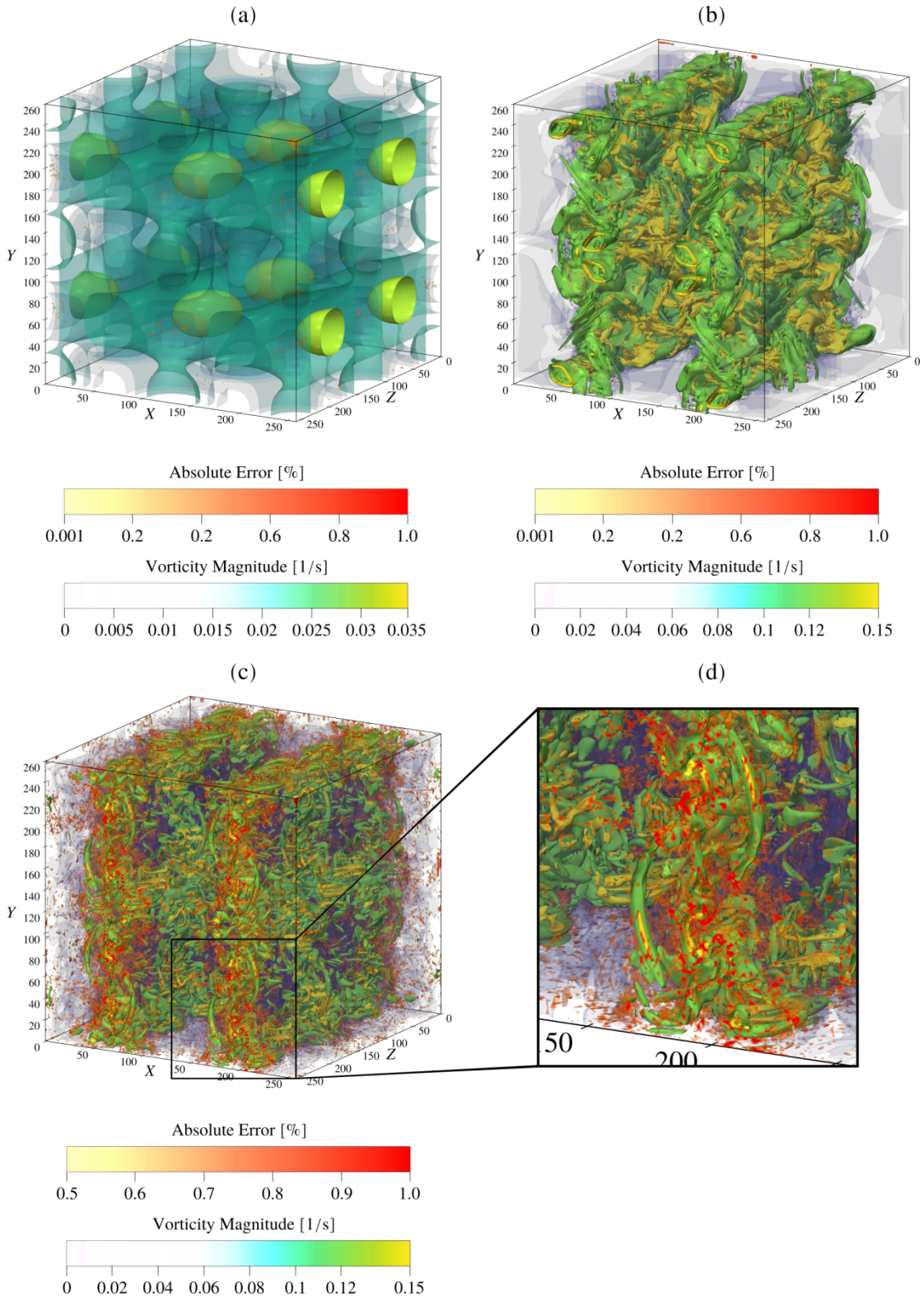


Figure 4: Visualization of the compressed Taylor-Green vortex test case on a 256^3 grid for a compression ratio of 120:1. The vorticity magnitude of the decaying vortex is shown from top left to bottom left for the non-dimensional times t = 0, 7.5, 15. Volume rendering of the relative error is used to indicate regions of large, compression-induced distortion.

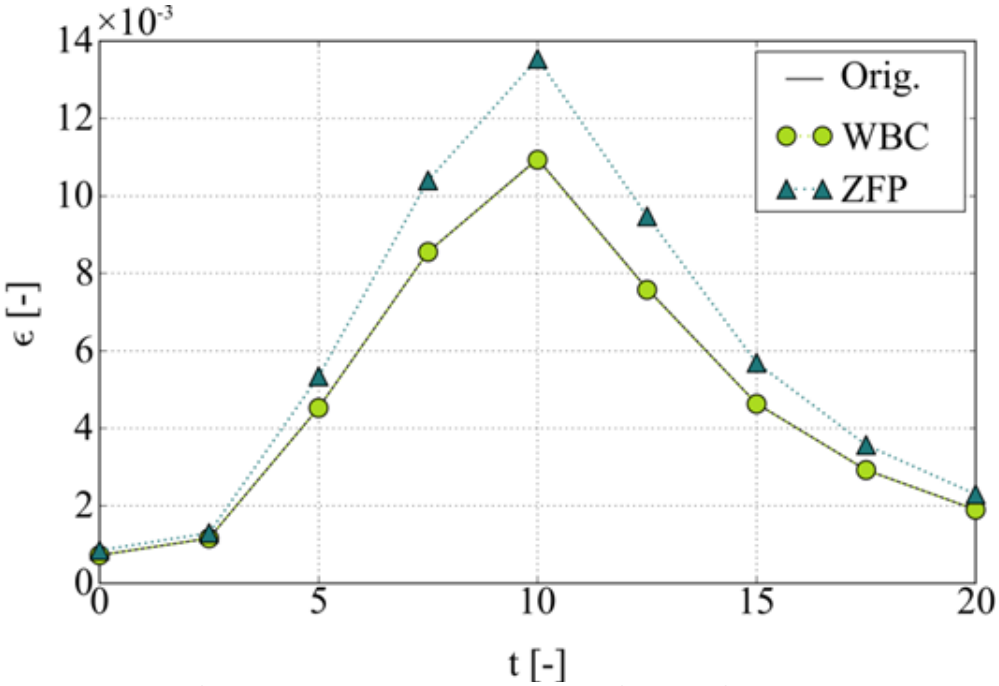


Figure 5: Comparison of normalized energy dissipation rate as a function of Taylor-Green vortex evolution at different compression ratios for ZFP and WBC compression algorithms.

For time-step 15 we can clearly observe areas of large, absolute errors present in regions defined by small scale vortex structures. These small-scale fluid-flow structures are synonymous with high-frequency information, which will accumulate in the high-frequency wavelet bands. This leads to a decrease in compression performance that must be compensated for by significant truncation of the compressed bit-stream, increasing the local, absolute error.

To analyze the amount of energy lost during the lossy compression stage, we compressed the time series of the Taylor-Green vortex test case with the WBC and ZFP algorithm at a compression ratio of 120:1. The energy dissipation rate was evaluated for each time-step and plotted against the energy dissipation rate of the original dataset. Figure 5 shows the evolution of the normalized energy dissipation rate for the original and uncompressed files. The figure shows that the WBC algorithm is capable of retaining most of the energy contained in the original datasets for large compression ratios. The compression stage of the ZFP algorithm, on the other hand, artificially dissipates large portions of the internal energy for high compression ratios. This behavior can be attributed to the fact that the ZFP compressor handles rate control without analyzing and selecting the bit-stream truncation points for the lowest possible field distortion.

2.1.2 Turbulent Flat-Plate Boundary Layer Flow

In contrast to the Taylor-Green vortex decay, boundary flows are defined by a large spectrum of length-scales that is present at all times throughout the entire flow field domain. The effects of a lossy wavelet-based compression stage on such a wall bounded flow were evaluated using a direct numerical simulation of a turbulent flat-plate boundary layer flow (TFBF) at $Ma_\infty = 0.3$ (see Figure 6). The spatial resolution of the numerical grid was set to $nx \times ny \times nz = 3300 \times 240 \times 512$ nodes in streamwise, wall-normal and spanwise directions, respectively. This amounts to a file size of *16 GigaBytes* for one time-step containing the conservative variables ρ, ρu, ρv, ρw and ρE.

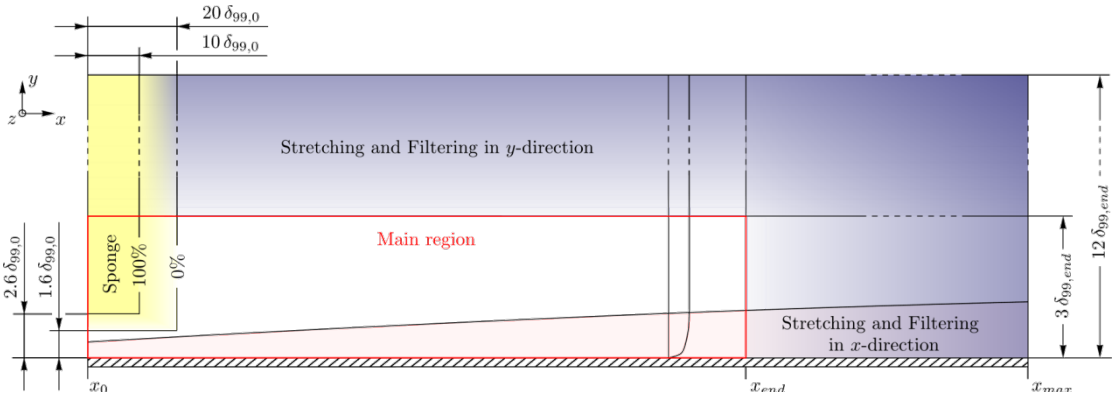


Figure 6: Numerical setup for the simulation of a turbulent flat-plate boundary layer flow at Ma∞ =0.3.

To understand the effects of a lossy compression stage on the TFBF test case we compressed the simulation file with our WBC algorithm at a compression ratio of 150:1. The PSNR amounted to 127.3 db for an average compression time of 161 s. The mean-velocity profile and the mean-velocity profile normalized by the wall shear velocity u_τ were evaluated for $Re_\theta = 2517$ and plotted in Figure 8 and Figure 7 respectively. While the mean-velocity profile shows good reconstruction with only a minor deviation from the original simulation file, the mean-velocity profile normalized by the wall shear velocity u_τ exhibits a significant discrepancy in the wake region. This error can attributed to a high susceptibility of the of the wall normal derivative of the velocity vector to loss of information in wall adjacent cells. This susceptibility can lead to a noticeable decrease in the wall sheer stress and, consequently, the friction coefficient.

The impact of high compression ratios on vortex detection and visualization is depicted in Figure 9. The figure demonstrates that for visualization purposes, the WBC algorithm is capable of a significant reduction in the overall file size without introducing any noticeable compression artifacts.

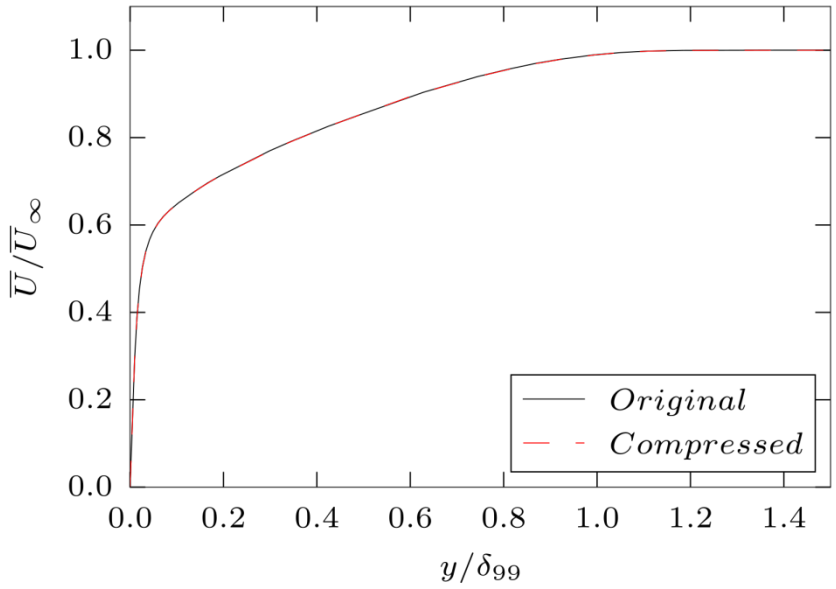


Figure 7: Comparison of the mean-velocity profile for the original and compressed DNS of a turbulent flat-plate boundary layer flow at Reθ = 2517 and Ma∞ =0.3.

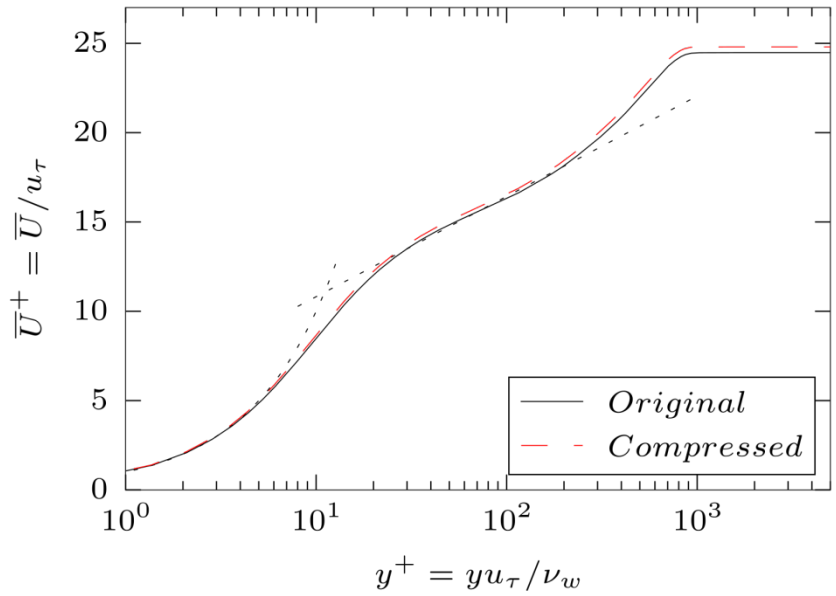


Figure 8: Comparison of the mean-velocity profile normalized by wall shear velocity for the original and compressed DNS of a turbulent flat-plate boundary layer flow at Reθ = 2517 and Ma∞ =0.3.

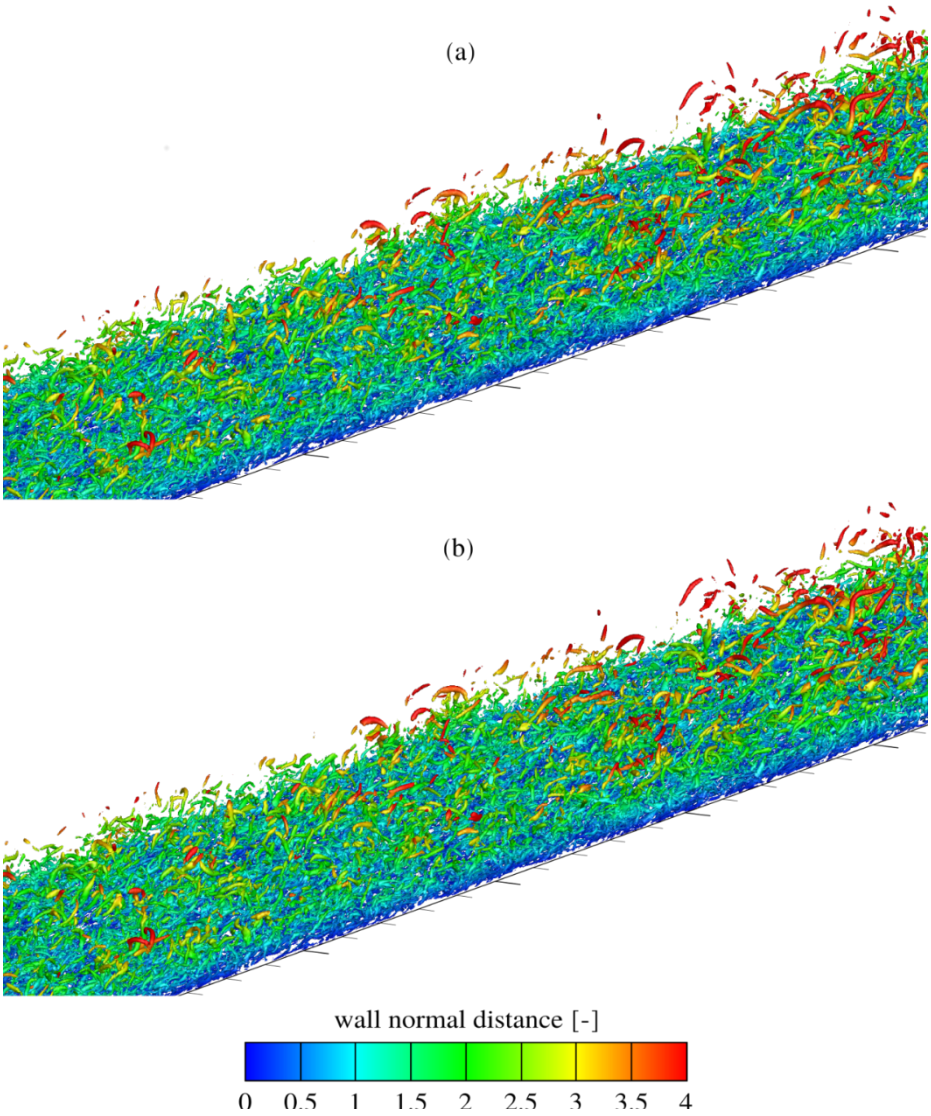


Figure 9: Original (a) and Compressed (b) DNS of a turbulent flat-plate boundary layer flow at Ma∞ = 0.3. Flow structures identified by the λ2 criterion for λ2 = −0.15. Coloring of isosurfaces according to wall normal distance y [15].

2.1.3 Jet in Crossflow

As a final test, the jet-in-crossflow test-case was used to evaluate how our compressor would fare on a dataset that is subdivided into zones of differing fluid flow activity in a well understood geometry. The numerical simulation was carried out at $Ma_\infty = 0.25$ for a jet-to-crossflow ratio of 3. The simulation file was 949 MegaBytes bytes in size, containing the conservative variables ρ, ρu, ρv, ρw and ρE on a $nx \times ny \times nz = 360 \times 800 \times 256$ numerical grid [9].

Figure 10 shows a close-up of the original (top) and compressed (bottom) DNS for a jet in crossflow at Ma ∞ = 0.25 and a jet-to-crossflow ratio of 3. The compression ratio measured 302:1 with a PSNR of 193.4. The average compression time amounted to 20 s. The compressed dataset exhibits excellent reconstruction with maximum error of 1%.

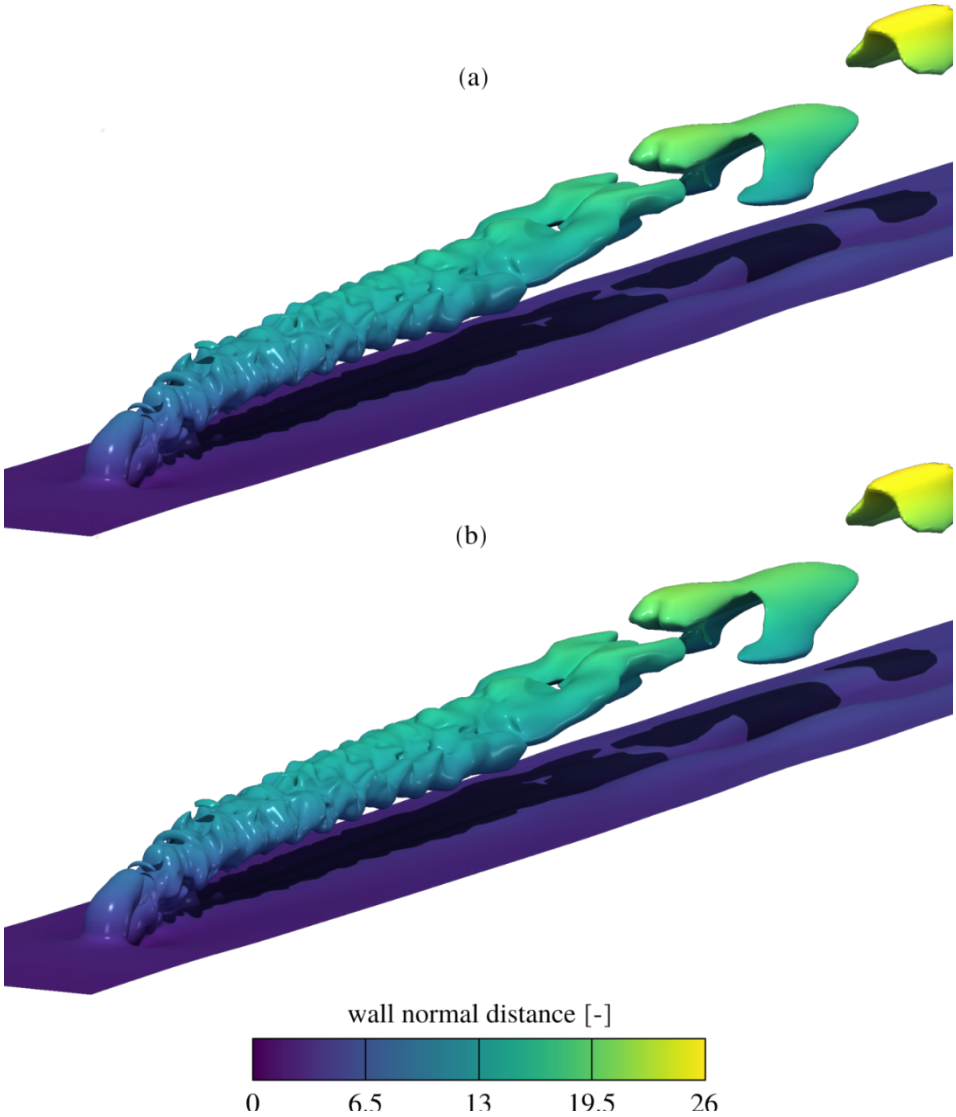


Figure 10: Original (a) and Compressed (b) DNS of a jet in crossflow at Ma∞ = 0.25 for jet-to-crossflow ratio of 3. Flow structures identified by non-dimensional, streamwise impulse = 0.92. Coloring of isosurface according to wall normal distance y.

2.2 Adaptive Mesh Refinement

2.2.1 Pre-conditioners for nonconforming SEM solver

A major source of stiffness for incompressible flow is a pressure operator making the pressure calculation the most expensive part of a simulation. That is why proper pre-conditioning of this problem has received much attention in the last decades and a number of approaches were proposed. Spectral Element Method (SEM) solver Nek5000 uses two possible pre-conditioners based on an additive overlapping Schwarz method [12][13], and a hybrid Schwarz-multigrid method [14][15]. In WP1 deliverables D1.2 and WP2 deliverables D2.2 we described the additive overlapping Schwarz approach and the modifications necessary to adapt this method for h-type Adaptive Mesh Refinement (AMR) framework. In this section we focus on the second alternative; the hybrid Schwarz-multigrid method. This method will be described in more detail in WP1 deliverables D1.3.

Both methods share number of features, so a big part of the code developed for the additive overlapping Schwarz method can be reused for its hybrid counterpart. The first method could be considered as a simplification of the second one, as it performs separately the Schwarz (local Poisson problems in overlapping sub-domains) and the global coarse grid problems and finally sums both contributions using relaxation coefficient. Alternatively, the hybrid pre-conditioner performs the whole multigrid V cycle with consecutive restriction and interpolation operators, and a coarse grid solver is executed only once after the whole coarsening stage is finished. At each multigrid level the restriction and Schwarz operators are executed performing global communication however their nonconforming communication routines differ, as the first one is based on transposed interpolation operator and the second one on the inverse one. Besides global communication we had to redefine the diagonal weight matrix that indicates the number of sub-domains sharing a given node and is used to accommodate for overlapping regions. Although in conforming case its definition is straightforward, the nonconforming case is more problematic as hanging nodes are not real degrees of freedom. In the current implementation the information about node multiplicity on the nonconforming faces is hidden to the parent element, so the parent element sees only one neighbor instead of two (four in 3D). Although this choice gives pre-conditioner that significantly reduces number of pressure iterations, its performance for the studied cases is slightly worse than performance of the additive Schwarz pre-conditioner. This can be caused by not-optimal value of the weight matrix, or by the fact that the hybrid pre-conditioner is superior over the additive one for high-aspect ratio elements, that are not present in our adaptive simulations. In the future we are going to investigate other definitions of the weight matrix, and our flagship run is performed with additive overlapping Schwarz preconditioner.

2.2.2 Introduction to AMR implementation

Here, we describe the progress made in implementing adaptive mesh refinement (AMR) in the co-design application Nek5000. This effort is closely related to work-package 1 task 1.1. We have developed a fully functional nonconforming solver of the incompressible Navier-Stokes equations which is capable of dynamical mesh modification at points during the simulation according to the estimated computational error. It allows running a fully adaptive 3D simulation of the turbulent flow around the NACA 4412 wing profile at Re_c = 200,000, which is one of ExaFLOW flagship runs. Figure 11 presents vortical structures of the velocity field around the wing and Figure 12 presents the part of the domain covered by refinement levels higher than 3 at simulation time 2.6s. This simulation is discussed in detail in the work-package 3 deliverables relating to the flagship runs.

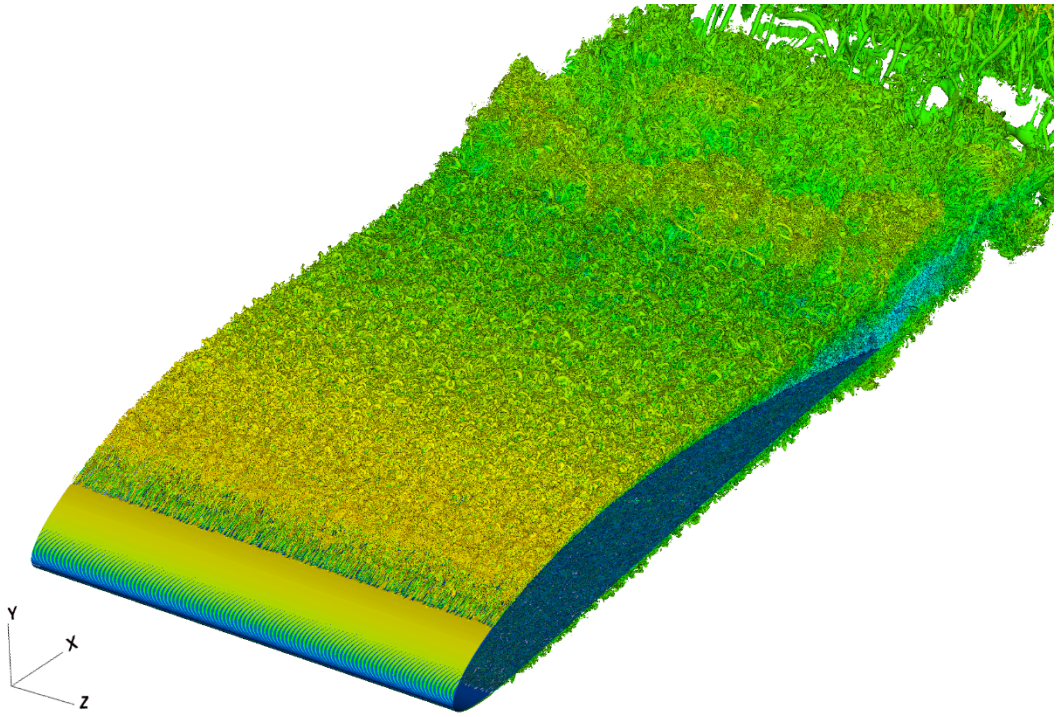


Figure 11: Vortical structures (λ2 criterion) of the velocity field around the wing in the flagship simulation at time 2.6s.

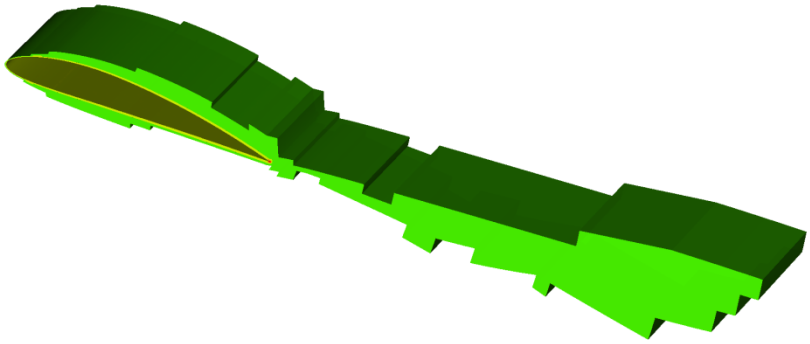


Figure 12: The part of the domain covered by refinement levels higher than 3 at simulation time 2.6s.

Since project month 18 many 2D and 3D tests have been performed and many refinements have been made to the code to increase speed and robustness. The main changes are detailed below. Apart from the adaptation of a hybrid Schwarz-multigrid preconditioner [14][15] for nonconforming meshes, which is described in Section 2.2.1.

2.2.3 Residual projection

The adaptation of residual projection [16] for both velocity and pressure fields for nonconforming meshes. To reduce the computational expense, Nek5000 uses previous solutions as a projection space that provides a proper initial guess for the iterative solver at the current time step. It requires an orthogonalization step using A-conjugate projection which involves action of the Helmholtz or pressure operators on the given vector. This code has been upgraded to deal with the nonconforming meshes using a proper version of the operator and resetting the projection space after each refinement step. This technique has been successfully applied to a number of test-cases including the flagship runs and we have observed a similar reduction in the iteration count for both the conforming and nonconforming runs.

2.2.4 High-pass-filter based stabilization

In implementing AMR in Nek5000 we followed the previous work of Fischer at all [17] concerning nonconforming SEM solvers and applied a conforming-space/nonconforming-mesh approach. In this approach the functional space is continuous across nonconforming faces. This simplifies the implementation as mortar elements can be replaced by a simple interpolation operator. However, this simplification led to instability in the case where the lower-resolution element (parent) is located downstream with respect to the high-resolution region (children). In this case the advected field could contain more information (a greater number of smaller scale structures) than can be represented by the

parent element. However, the filed values at the interface are set “upwind” by the low-resolution parent. This makes the interface partially reflective and leads to the oscillations which are visible in Figure 13 and Figure 14.

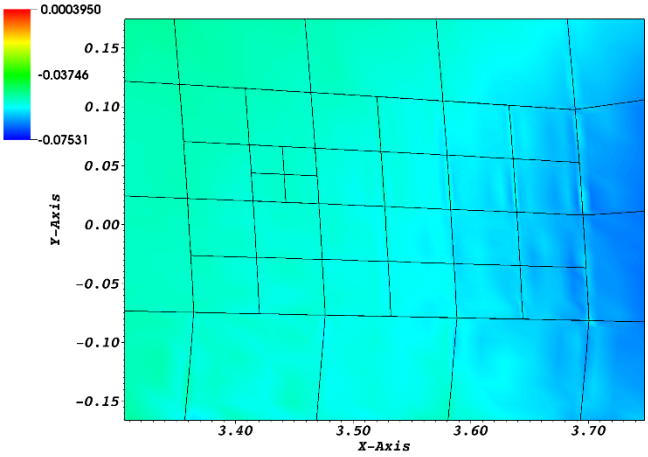


Figure 13: Y velocity component at the children-parent interface with the low-resolution region located downstream in 2D simulation of flow around NACA4412 wing. This shows additional element borders. Oscillations at the right interface of the children element caused by reflective properties of the interface are visible.

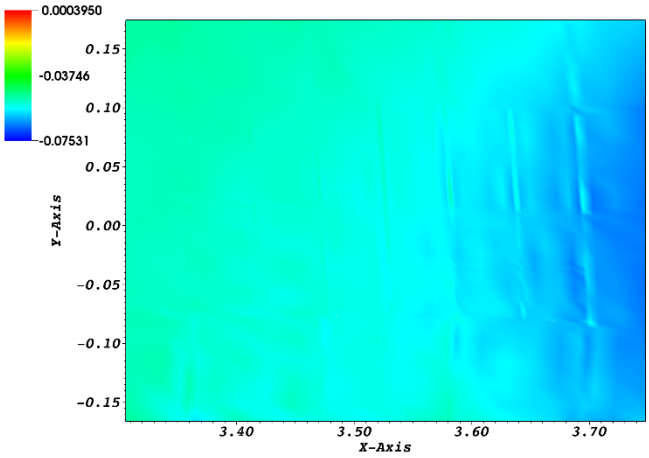


Figure 14: Y velocity component at the children-parent interface with the low-resolution region located downstream in 2D simulation of flow around NACA4412 wing. Oscillations at the right interface of the children element caused by reflective properties of the interface are visible.

In most cases these oscillations remain small and are suppressed by AMR itself, as it would take care of the unresolved regions. However, in some situations (e.g. user overwriting error indicator marks to additionally reduce resolution in not interesting regions) they can grow and even cause the simulation to crash. To avoid such a situation, we add a stabilization mechanism partially diffusing in the interface vicinity of small scale features that cannot be properly represented in the parent element. We achieve this by adding a high-pass filter to the forcing term:

$$\frac{du}{dt} = L(u) - \chi * m(r) * H(u),$$

where χ, m(r) and H(u) are relaxation coefficient, masking function and transfer function removing half of high frequency modes respectively. This method is often used for simulation stabilization [18] however in our case masking function

restricts forcing to the interior of the children elements touching nonconforming faces (Figure 15). The efficiency of this method depends on the value of relaxation coefficient and setting its value is not straightforward. Too low a value of χ would make the filter inefficient and on the other hand too strong a forcing could remove important flow features. Based on our current experience, we set χ=0.5, but more investigation of this problem is necessary. Future work will focus on removing the assumption of a conforming functional space and implementing mortar elements in the solver, which should increase stability.

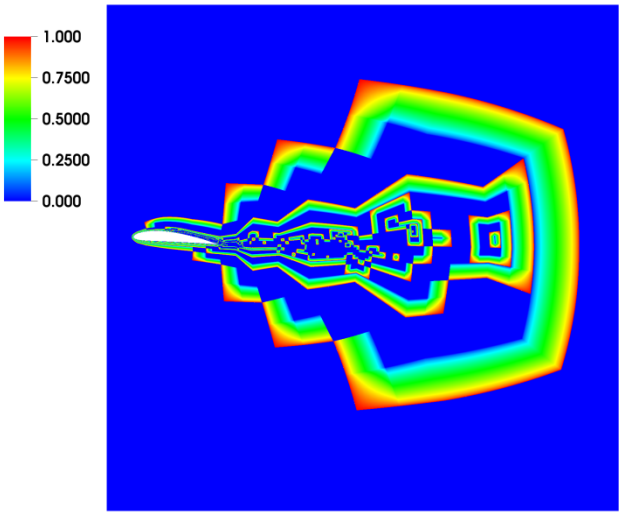


Figure 15: Linear masking function restricting action of the high-pass filter to the nonconforming interfaces. Notice, the masking function forms a set of closed loops at every interface between different refinement levels, as we do not distinguish between local inflow and outflow conditions. This however, does not influence significantly the solution, as the flow entering high resolution region does not contain small scale features that would be damped.

2.2.5 Modification to the Refine, transfer & coarsen strategy

After every refinement step, elements are constructed/destroyed and redistributed between processors to keep an optimal load balance. An important limitation of Nek5000 is the exclusive use of a static memory allocation that sets maximum number of elements per MPI rank. Although in most cases this number is sufficiently high, for some specific mesh partitioning it can be exceeded by the number of refined elements on a single core. This limitation was not significant for initial tests (e.g. 2D flow around NACA 4412 wing profile) but it became critical for 3D flagship tests. To remove this constraint we have refactored the algorithm. The required memory can now be limited by refining, transferring and coarsening one variable at a time and by performing the whole operation in a sufficiently large memory buffer.

2.2.6 Modifications to solver restart.

The multi-step time integration scheme applied by Nek5000 requires information from the last three time-steps to perform the new one. To minimize computation, Nek5000 utilizes additional arrays storing not only old velocity and pressure

fields, but also old right-hand sides of all linear problems. To retain time integration order, this information has to be available at every step including the beginning of the simulation and after each refinement step. Our initial implementation performed the refinement/transfer/coarsening operations on all right-hand sides, even though they could be recalculated after each refinement step. This approach was less computationally intensive but was not optimal with respect to communication and could not be used during simulation restart due to the volume of data that had to be stored on the disc. Hence, the new solver restart procedure was developed which interpolates only the current and previous variable fields and recalculates the right-hand sides after refinement. The same set of routines is used in the new simulation restart, which is significantly improved with respect to the original version. Unlike the original version it makes it feasible to save the full set of restart data in a single time step and to recreate final simulation state in the new run.

The implementation of trip forcing for nonconforming meshes. One of the goals of this project is to improve efficiency of turbulent flow simulations. In many such simulations turbulence has to be triggered by free-stream turbulence or some kind of forcing. In the conforming reference case of our flagship run the tripping line technique [19] was used to start turbulence and we implemented it in our AMR code. The tripping line technique applies to the filed a forcing, which is a sum of number of Fourier mode with constant amplitude and random phases, and is later localized with use of Gaussian profile$f(r) = e^{-\left(\frac{r}{\delta}\right)^2}$. Unfortunately, infinite support of Gaussian profile and its relatively slow decay led to discontinuities across interfaces of different refinement level, and we had to modify tripping formulation replacing Gaussian profile with similar function with limited support:

$$\boldsymbol{f(r)} = \begin{cases} \left(\mathbf{1} - \left(\frac{\boldsymbol{r}}{\boldsymbol{\delta}}\right)^{\mathbf{2}}\right)^{\mathbf{2}} * \boldsymbol{e}^{-\left(\frac{r}{\delta}\right)^2}, \wedge\, \boldsymbol{r} < \boldsymbol{\delta} \\ \mathbf{0}, \wedge\, \boldsymbol{r} \geq \boldsymbol{\delta} \end{cases}$$

It allowed us to force single refinement level across tripping region and avoid jumps. Figure 16 shows comparison of Gaussian profile and the function used in our simulations.

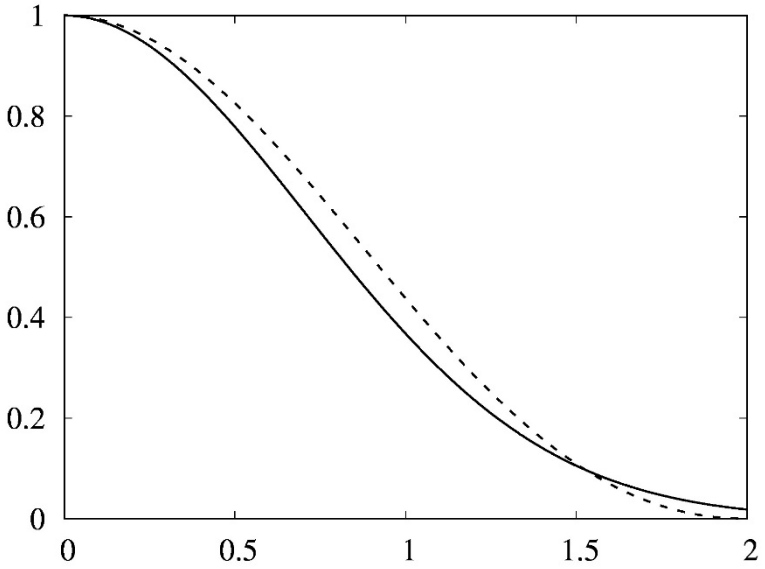


Figure 16: Gaussian profile (continuous) and a function used for localization of forcing in AMR runs (dotted).

2.2.7 Implementation of additional checks for refined/coarsened regions.

A fully dynamical mesh refinement can lead to very complex meshes that could potentially influence solver efficiency. By performing a careful investigation of such potential dependencies of our solver on certain meshes over a number of different test-cases and mesh structures we found no efficiency degradation in 2D runs and 3D laminar simulations. However, in the case of 3D turbulent runs, in which the flow is very complex and the code tends to follow very small structures constantly refining and coarsening given region, we found one mesh configuration that increased significantly pressure iteration count. In most studied cases this configuration was generated at the coarsening stage, where small number of elements with higher refinement level were replaced by the lower resolution ones creating narrow (single element width) gap in the high-resolution region and leaving at the same time single family (8 children of single parent) of high resolution element attached with the rest of high resolution region by edges only. This scenario is illustrated in Figure 17 and Figure 18.

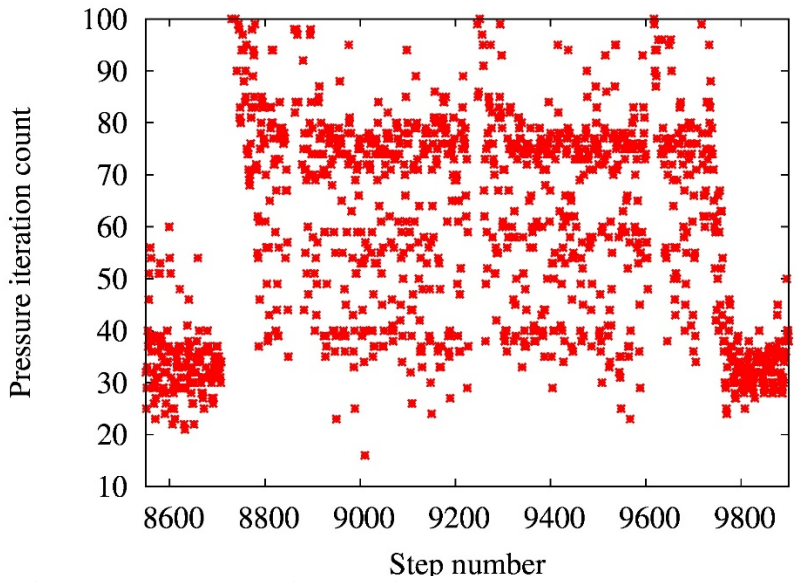


Figure 17: Number of pressure iterations as a function of time step number. The plot covers two refinement steps taking place at step 8700 and 9700.

Figure 17 presents the number of pressure iterations as a function of time step number and covers two refinement steps taking place at steps 8700 and 9700. The rapid increase in pressure iteration count between both refinement steps is clearly visible. Figure 18 shows the boundary of the highest resolution region at three stages: before (top) and after (middle) first refinement at step 8700 and the final mesh (bottom) after second refinement at step 9700. Although the relation between presence of the single high-resolution family of elements sticking out of the high resolution region and the jump in pressure iteration is clear, the mechanism is not fully understood yet. That is why we added set of additional global refinement and coarsening tests excluding this particular configuration.

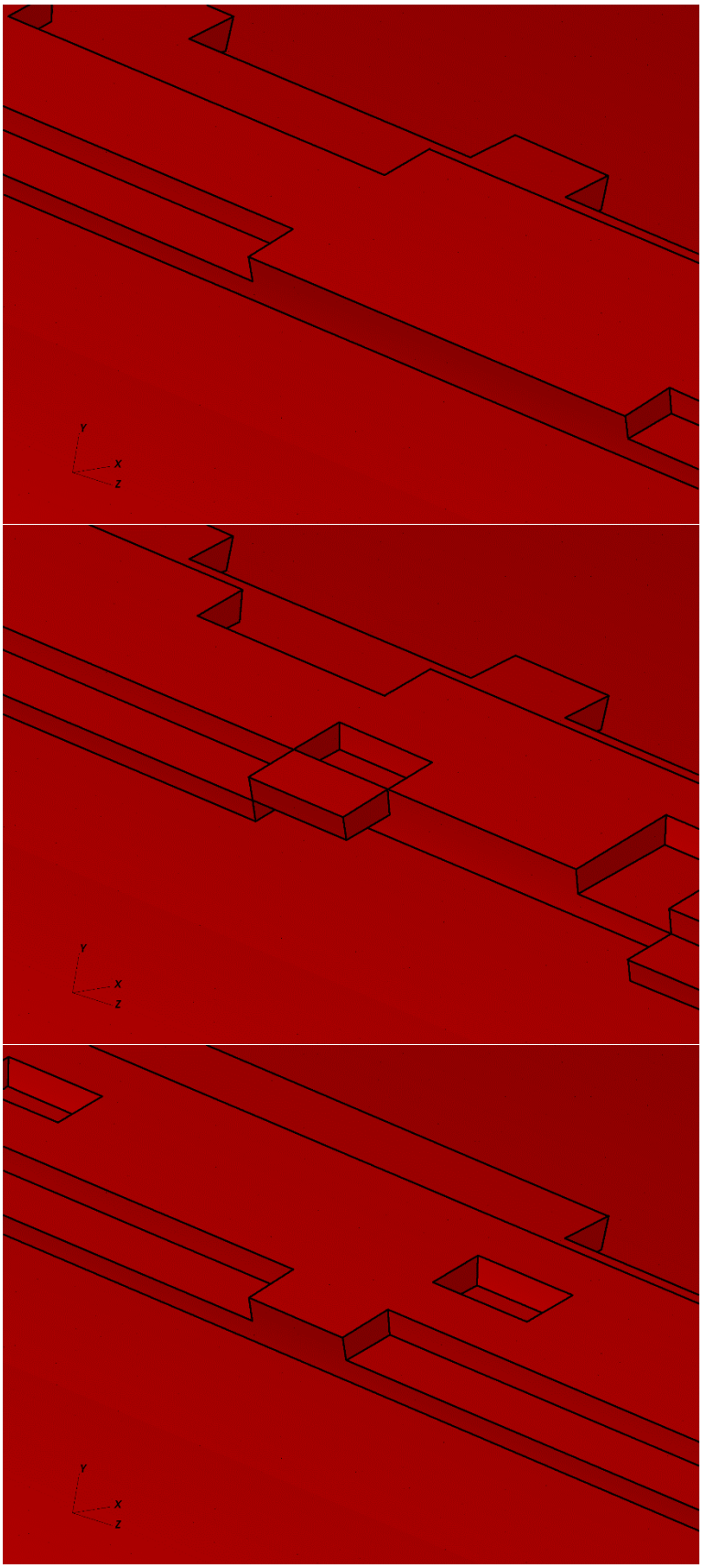


Figure 18: The boundary of the highest resolution region at three stages: before (top) and after (middle) first refinement at step 8700 and the final mesh (bottom) after second refinement at step 9700.

2.3 Impact of Preconditioners on Nektar++ Scaling: PETSc vs. Native Linear Solvers

2.3.1 Introduction

The purpose of this study is to evaluate how the scaling of Nektar++ changes when some components of the parallel linear algebra are replaced by third-party library. Nektar++ discretizes incompressible Navier-Stokes equations

$$\frac{\partial u}{\partial t} + (u \cdot \nabla)u = -\nabla p + \frac{1}{Re}\nabla^2 u,$$
$$\nabla \cdot u = 0,$$

by a velocity-correction high-order splitting scheme in which the pressure is first solved as a Poisson problem and the velocity then adjusted to enforce the incompressibility constraint through a series of Helmholtz problems. The discretization of the Poisson problem in massively parallel setting produces a linear system that can't be efficiently solved without appropriate preconditioner. We will therefore concentrate on evaluating the preconditioners for the prototype elliptic problem

$$\nabla^2 u(x) - \lambda \cdot u(x) = f(x),$$

equipped with appropriate initial and boundary conditions.

2.3.2 Preconditioners

Nektar++ implements a range of preconditioners suitable for the above problem. For large problems with complex flow physics, the full linear space preconditioner with low energy block is usually the default choice.

The preconditioner exploits the idea that writing the discrete solution $u^{\delta}(x)$ in terms of a suitable polynomial basis (which differs from the original expansion) can produce a discrete problem for which it is easier to find an efficient preconditioner. In the first step, static condensation (sub-structuring) is performed so that the task of solving the globally coupled linear system now requires a solution of a Schur complement system and the inverse of a number of relatively small, element-local matrices.

The Schur complement is moved into the new polynomial space by a matrix transformation that corresponds to a change of basis such that the coupling between elemental edge and node modes (in 3D) and face-interior modes is as weak as possible. The 'optimized' basis has edge modes such that their energy in the face interiors is the lowest possible for given polynomial space. Another effect of this transformation is that the Schur complement matrix can now be efficiently approximated by considering only its block-diagonal structure (in other words, the diagonal terms are much more dominant).

Finally, the low-energy Schur complement system is coupled with an additive Schwarz preconditioner that acts on the 'coarse space' comprising of vertex modes belonging to the modified basis. It is the coarse system that we attempted to solve with algebraic multigrid (BoomerAMG) available through the PETSc interface in Nektar++ as a drop-in replacement for XXT. A detailed description of the preconditioner can be found in [188].

For the purpose of the tests presented below, we considered the following setup in Nektar++ configuration files:

```
<PARAMETERS>
    <P> TimeStep       = 1e-6    </P>
    <P> NumSteps       = 520     </P>
    <P> IO_InfoSteps   = 1       </P>
    <P> wavefreq       = PI      </P>
    <P> epsilon        = 1.0     </P>
</PARAMETERS>

<SOLVERINFO>
    <I PROPERTY="EQTYPE"                VALUE="UnsteadyDiffusion"   />
    <I PROPERTY="Projection"            VALUE="Continuous"          />
    <I PROPERTY="AdvectionType"         VALUE="WeakDG"              />
    <I PROPERTY="DiffusionAdvancement"  VALUE="Implicit"            />
    <I PROPERTY="TimeIntegrationMethod" VALUE="DIRKOrder3"          />
    <I PROPERTY="GlobalSysSoln"         VALUE="IterativeStaticCond" />
    <I                                          PROPERTY="Preconditioner"
VALUE="FullLinearSpaceWithLowEnergyBlock"/>
    <I PROPERTY="LinearPreconSolver"    VALUE="Xxt"                 />
</SOLVERINFO>
```

When using algebraic multigrid, the last line

```
<I PROPERTY="LinearPreconSolver"    VALUE="Xxt" />
```

Would be replaced by

```
<I PROPERTY="LinearPreconSolver"    VALUE=”PETSc" />
```

And in addition to that, the following options were passed to PETSc:

```
-ksp_type    preonly    -pc_type    hypre    -pc_hypre_type    boomeramg    -
pc_hypre_boomeramg_relax_type_all Jacobi
```

2.3.3 Intercostal pair results

We solved the unsteady diffusion problem

$$\frac{\partial u(t,x)}{\partial t} + \lambda u(t,x) + \nabla^2 u(t,x) = f(x)$$

on an unstructured tessellation of an intercostal pair as shown in Figure 19. The initial condition was set to:

$$u(t=0,x) = \sin(\pi x) \cdot \sin(\pi y) \cdot \sin(\pi z),$$

and Dirichlet boundary condition *u=0* was imposed everywhere on the boundary.

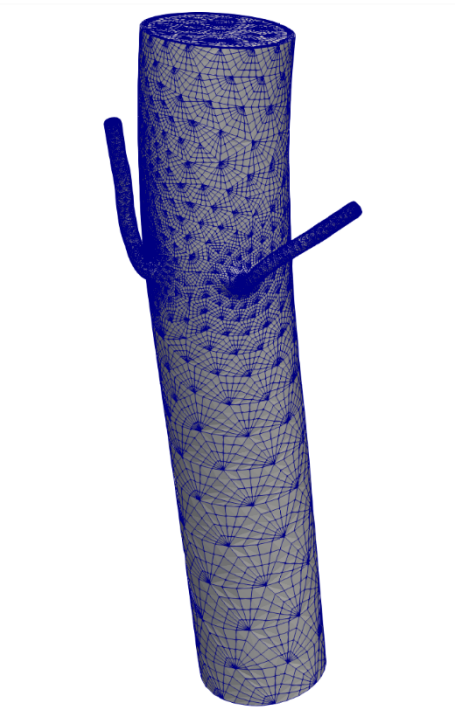


Figure 19: Intercostal pair geometry meshed with P5 tetrahedral elements.

A typical governing PDE for this kind of geometry would be incompressible Navier-Stokes equations. We intentionally study only the implicit component of the fractional step solver to eliminate the influence of explicit solvers for velocity components. The connectivity inside the grid, on the other hand, is typical for large unstructured meshes: one mesh node can be shared by several elements that varies in different parts of the mesh and we can't make a-priori assumptions about the communication patterns.

The simulation ran for 520 time-steps with $\Delta t = 10^{-6}$. We compared the CPU time to make one-time step when Nektar++ uses a parallel conjugate gradient solver to solve a statically condensed continuous Galerkin system in conjunction with

- A low-energy pre-conditioner
- An algebraic multigrid pre-conditioner as provided by the Hypre package through PETSc which is interfaced with the MultiRegions library of Nektar++

The solution was computed on a mesh consisting of high-order elements with a basis of polynomial degrees 5 and 10, respectively. The number of elements in both cases was 9,551.

The first few time-steps in Nektar++ are typically much more expensive than the rest of the computation due to additional setup costs. For this reason, the resulting CPU time per time-step was computed as an average for iterations 20-520.

Nektar++ with the low-energy pre-conditioner turned out to be more efficient in all computed cases. With increasing polynomial degree, the difference between Nektar++ and PETSc quickly decreases, see Figure 20 and Figure 21. Note that the

right-most point in both figures corresponds to case when the number of elements per core is approximately 9-10 and the cores are likely not sufficiently work-saturated.

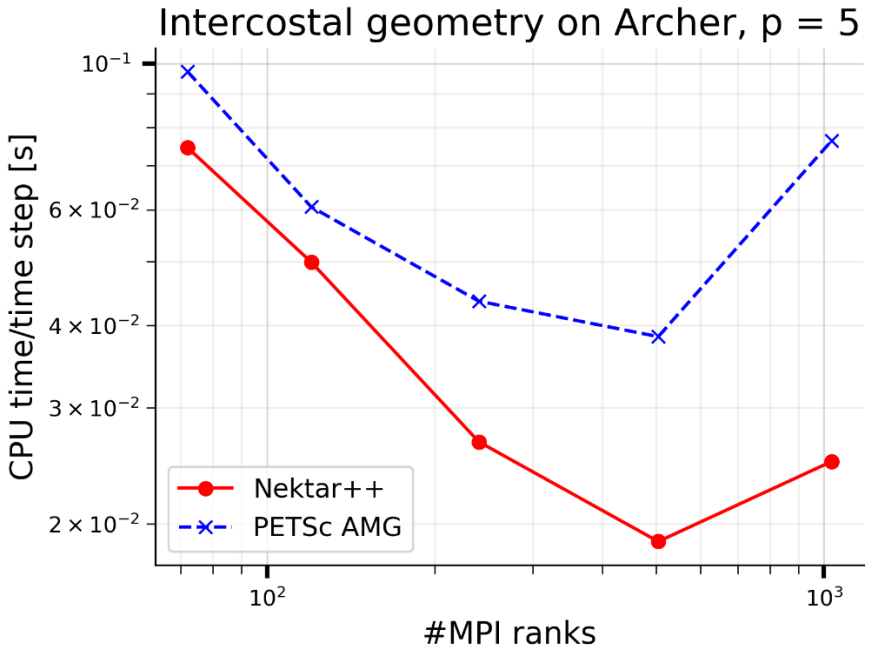


Figure 20: Impact of linear solvers on scaling - intercostal geometry on tetrahedra with deg = 5

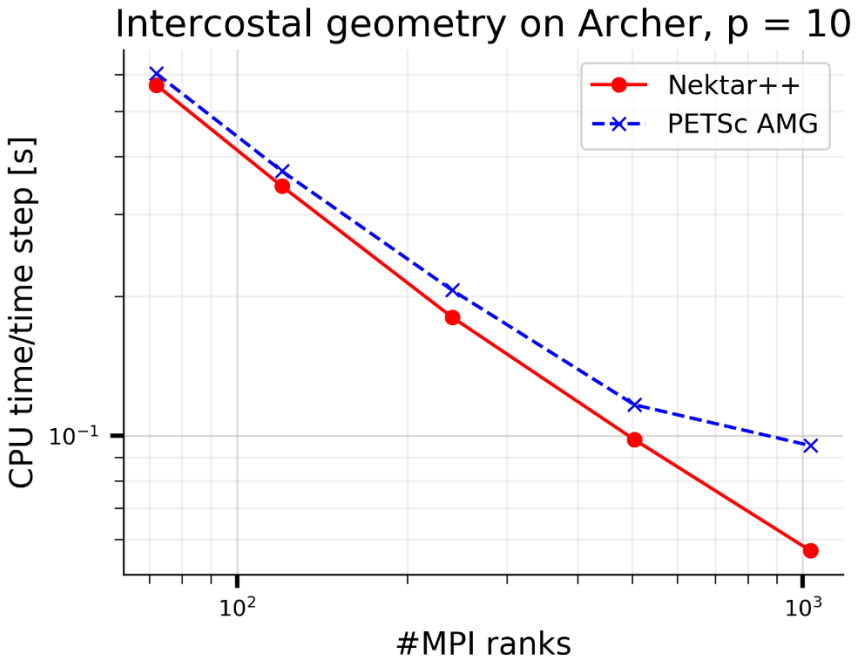


Figure 21: Impact of linear solvers on scaling - intercostal geometry on tetrahedra with basis of polynomial degree = 10.

2.3.4 Aorta geometry results

This test-case is a fully tetrahedral geometry with 150,302 elements. We solved the same problem as defined in the previous section and again studied the effect of replacing XXT in the coarse-solve stage in the preconditioner by algebraic multigrid solver available through PETSc.

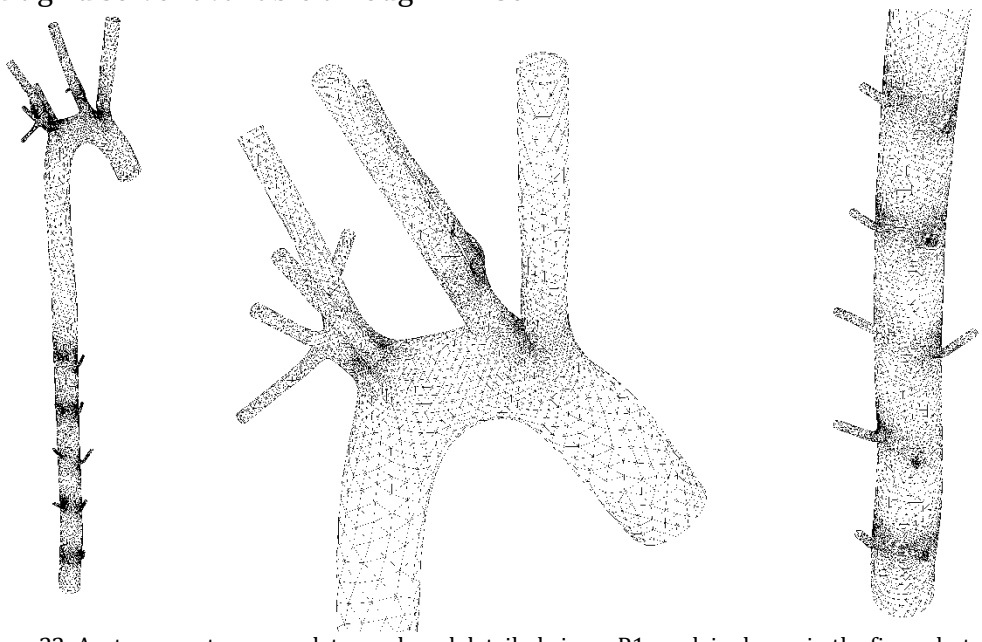


Figure 22: Aorta geometry - complete mesh and detailed views. P1 mesh is shown in the figure but elements with polynomial degrees 5 and 10 were considered for scaling tests.

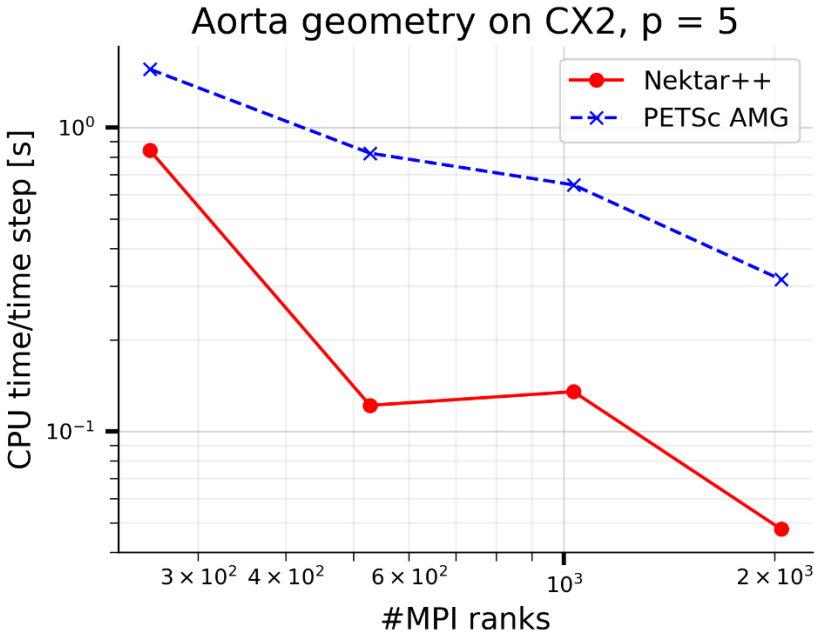


Figure 23: Impact of linear solvers on scaling - aorta geometry on tetrahedra with basis of polynomial degree = 5.

2.3.5 Conclusion

It was initially expected that the AMG solver would have a positive or at least neutral effect on performance, i.e. the timings should not be significantly worse, especially for larger core counts. After analysing the log files produced by PETSc and discussion with developers from Argonne National Laboratory [189] it was deduced that the performance of PETSc solvers suffers due to unsuitable partitioning of degrees of freedom among processors. The distributed sparse matrix storage format of PETSc (known as 'aij' in PETSc terminology) expects the solver to distribute complete rows of the matrix on different cores. This is unfortunately not compatible with the partitioning strategies of Nektar++ which are based on weighting the boundary (surface) degrees of freedom of each mesh element. As a result, the number of conjugate gradient iterations in the coarse preconditioner solve and residual magnitudes were consistent between Nektar++ and PETSc, with PETSc AMG timing being one order of magnitude worse than XXT. The following is an excerpt from log files generated when running on the aorta geometry with fifth-order tetrahedral elements:

XXT:

```
CG iterations made = 9 using tolerance of 1e-09 (error = 7.0867e-10, rhs_mag
= 291034)
CG iterations made = 8 using tolerance of 1e-09 (error = 8.56456e-10,
rhs_mag = 291028)
CG iterations made = 8 using tolerance of 1e-09 (error = 8.79147e-10,
rhs_mag = 291022)
Steps: 520      Time: 0.00052      CPU Time: 0.0432795s
```

BoomerAMG + PETSc:

```
CG iterations made = 10 using tolerance of 1e-09 (error = 8.79726e-10,
rhs_mag = 291034)
CG iterations made = 10 using tolerance of 1e-09 (error = 5.78063e-10,
rhs_mag = 291028)
CG iterations made = 9 using tolerance of 1e-09 (error = 9.80485e-10,
rhs_mag = 291022)
Steps: 520      Time: 0.00052      CPU Time: 0.284557s
```

Note that in the above listing, 'Steps' refers to the number of time iterations (we used 3-stage diagonally implicit Runge-Kutta scheme), 'Time' is physical time and 'CPU time' refers to the time spent in all three stages of the RK time stepper.

The pursuit of AMG as alternative coarse-level solver in the full block diagonal preconditioner as PETSc is in our opinion still worthwhile but making PETSc more competitive with the native linear algebra of Nektar++ will require additional design effort.

2.4 Algebraic Multi-Grid

As mentioned in D2.2, the key to the efficient resolution of the incompressible Navier-Stokes equations is the pre-conditioning of the pressure equation. In Nek5000, the pre-conditioning strategy is split into local subdomain operators and a global coarse grid solver. This coarse problem can currently be solved using either a sparse basis projection method, called XXT, or an in-house algebraic multigrid (AMG) solver, which is preferred for massively parallel (more than 1e4 processes) large simulations (more than 1e5 elements). Given the expected size of the simulations at exascale, the AMG option will be the only viable choice. However, the current in-house AMG solver, while highly parallel and efficient, requires a slow setup phase, done once for each mesh and performed by an off-line code. Two serial alternatives for the setup have been presented in D2.2. In the framework of AMR and mesh adaption, such a constraint is highly unpractical and only the XXT solver has been used so far as a consequence, despite lengthy setup times and lower performance for large runs. Therefore, recent developments aiming at integrating both the setup and the solver, in parallel, inside Nek5000 have been investigated and are presented.

The latest work done on the coarse grid solver was to fully include the Hypre library for linear algebra [20] in Nek5000. This external library is now used to perform both the AMG setup and solver, on-the-fly and in parallel. In practice, this means that the simulation should not be stopped and restarted after each mesh adaptation and that the Hypre AMG solver now replaces the original in-house one.

This is achieved with little modification to the existing code. First, the node numbering of the coarse grid problem needs to be modified to match Hypre's requirement. This is done by simply building a mapping between both numbering. Second, the construction of the global coarse grid operator requires the transfer of some local data between processes. This is readily done using MPI commands and the crystal router present in Nek5000. Following these two steps, the coarse grid operator is built in the form of a parallelly distributed Hypre matrix and the AMG setup is performed on that matrix. Several different methods are available within Hypre for the three parts of the setup: coarsening, interpolation and smoothing. While many of the options give acceptable or good results, the best combination determined so far is to use: PMIS-coarsening, an extended "classical" interpolation and a symmetric hybrid Gauss-Seidel relaxation scheme (see Hypre user's manual for more details [20]).

The resolution of the coarse grid problem using Hypre is straightforward given the setup data, the mapping between the two node numberings and a proper right-hand side to the problem.

The full use of Hypre for the coarse grid solver is first tested on a conforming mesh. The test case is the flow inside a turbulent straight pipe at $Re_\tau = 550$. The mesh is made of about 850,000 spectral elements and the simulation is run on 2048, 4096 and 8192 processes on Beskow, the Cray XC40 supercomputer at KTH. Results for the setup are presented in Table 2. As a comparison, the setup time for the XXT solver is shown for 2048 processors. It is observed that the setup, performed only once per mesh, does not scale. However, the AMG setup with Hypre is still faster

by more than two orders of magnitude compared to XXT. As a side note, the setup time for the original in-house AMG solver, done by a serial code is about 100 seconds.

Table 2: Setup time in seconds for the flow in a straight pipe.

PROC. NUMBER	HYPRE AMG	XXT
2048	1.7198	449.97
4096	2.1463	-
8192	2.8781	-

Solver times for the in-house AMG, Hypre AMG and XXT are shown for the same test case in Table 2. The results are an average over 50 timesteps and are obtained for the same node allocation for a given number of processors. Both AMG solvers perform equally well, while XXT is about one order of magnitude slower on 2048 processors. The performance of XXT being expected to degrade even more for larger simulations, the use of AMG is very much preferred in those cases. Furthermore, the results show that there is no penalty in switching from the in-house to the Hypre AMG.

Table 3: Time per timestep spent for the coarse grid solver, in seconds, for the flow in a straight pipe.

PROC. NUMBER	IN-HOUSE AMG	HYPRE AMG	XXT
2048	6.667×10^{-2}	6.679×10^{-2}	83.30×10^{-2}
4096	4.532×10^{-2}	4.010×10^{-2}	-
8192	8.107×10^{-2}	6.393×10^{-2}	-

Tests on 2D nonconforming meshes show a similar good behavior. Tests on large 3D nonconforming simulations are being investigated. As soon as the use of the Hypre AMG is validated for these simulations, it will become the default method for the coarse grid solver.

2.5 Performance of minimally intrusive fault tolerance mechanism

A prototype implementation of this algorithm, as detailed in section 5.1 of D1.3 has been developed in Nektar++. Nektar++ is heavily object-oriented and much of the static data is encapsulated within classes and other rich data structures.

Code was added to perform the message logging, message replay and exchange of static-and dynamic-data remote in-memory checkpoints to affect the resilience capabilities described in D1.3. Command-line parameters enable the user to specify the number of spare processes, S to be reserved dynamically at run-time from the total of N ranks requested and the rank offset k to be used for storing state preservation data. The last S ranks are, by default, assigned to be spares while the first $W = N - S$ ranks are allocated as workers. Each worker rank r sends state preservation data to rank $(r + k) \bmod W$, where $0 < k < W$ is the stride. The value of k can be set to the number of ranks on a node or chassis, to improve the resilience based on specific cluster configuration. Rank r also receives state preservation data from rank $(r + W - k) \bmod W$.

Performance is analysed using the solver for the incompressible Navier-Stokes equations. The specific test problem we consider is laminar flow in a rectangular duct of height D, streamwise length $20D$ and of width $10D$. This is an intentionally trivial problem designed to demonstrate the effectiveness and scalability of the resilience algorithm under a relatively non-intensive computational load.

The test system used is an SGI ICE XA system with 10,080 cores. Each of the 280 nodes are equipped with dual 2.1Ghz, 18-core Intel Xeon processors with hyper-threading enabled and 256GB RAM. All nodes are connected using a single Infiniband fabric and share a common Lustre parallel filesystem. Tests are performed across a range of working process counts, spanning two nodes, up to sixty-four nodes, or NP_{max}=2304 processes. One process was assigned per hardware core (up to 36 per node) and MPI bind-to-core was used to reduce the effects of inter-socket memory bandwidth contention and lower-level cache thrashing. For all fault tolerance tests, an additional node was allocated to ensure the number of working processes remained the same and the offset value was set at $k = 36$ to ensure state preservation data was stored on a different node.

Remote in-memory checkpoint time for the dynamic data was measured on all processes. Figure 24 (solid line) shows strong scaling results of measured in-memory checkpoint time, as a function of the number of processes. Checkpoint time monotonically decreases with increasing parallelism, due to reduced data volume per process, with little sign of saturation up to the limit of the available cores. In contrast, disk checkpointing (dashed line in Figure 24) saturates around 288 cores and increases for larger core counts. Above 2k cores, disk checkpoint time is approximately two orders of magnitude greater than remote in-memory check-pointing. As a point of comparison, we also show the execution time per time-step, measured as the wall-time of advancing the PDE by one time-step without any form of checkpointing.

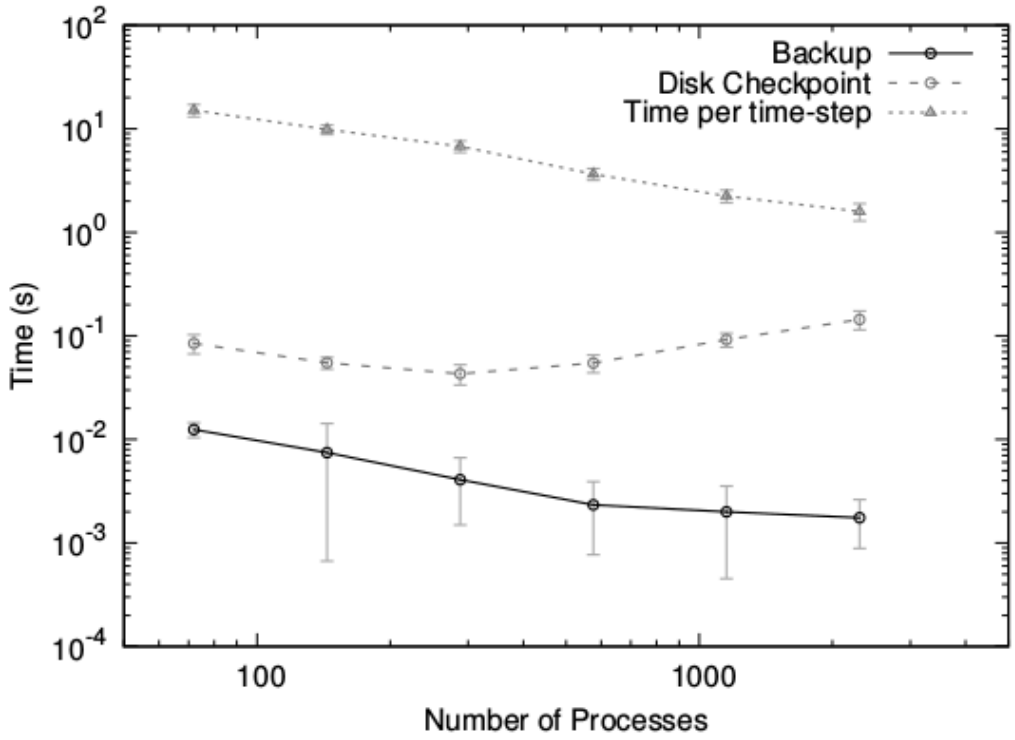


Figure 24: Remote in-memory dynamic data checkpointing time for test problem (solid line), compared with disk checkpointing (dashed line). For comparison the execution time per time-step, without any form of checkpointing, is shown (dotted line).

Figure 25 shows the time taken to recover following a process failure, as a function of the number of cores. Reconstruction (solid line) includes the communication of the state preservation data, reinitialisation of MPI communicators and execution of the initialisation phase of the application code to regenerate the static data. During initialisation, MPI operations use the result from the message log rather than performing actual communication and therefore the process recovers completely independently. This part of the algorithm scales well and takes less than half a second at NP_{max}. Since MPI communication during the initialization of the gather-scatter external library objects could not be logged by the prototype implementation in Nektar++, these are reinitialised exactly as for the normal start-up and shown separately (dashed line) in Figure 25. Until such communication can be included in the message log, this is the dominant cost of recovery at larger core counts.

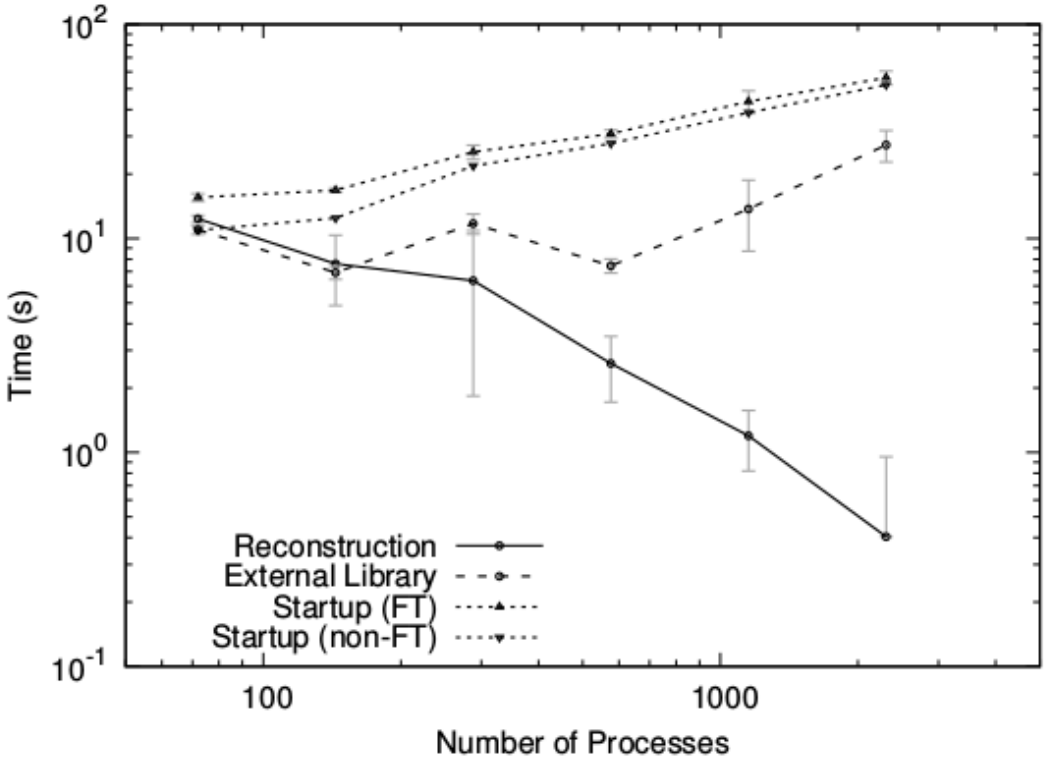


Figure 25: Recovery time as a function of number of processes, broken down into the time to repair communication and regenerate static data and, separately, the time taken to reinitialize the external gather-scatter library. Time for initial simulation start-up is shown for comparison, both for the original and fault-tolerant versions of the code.

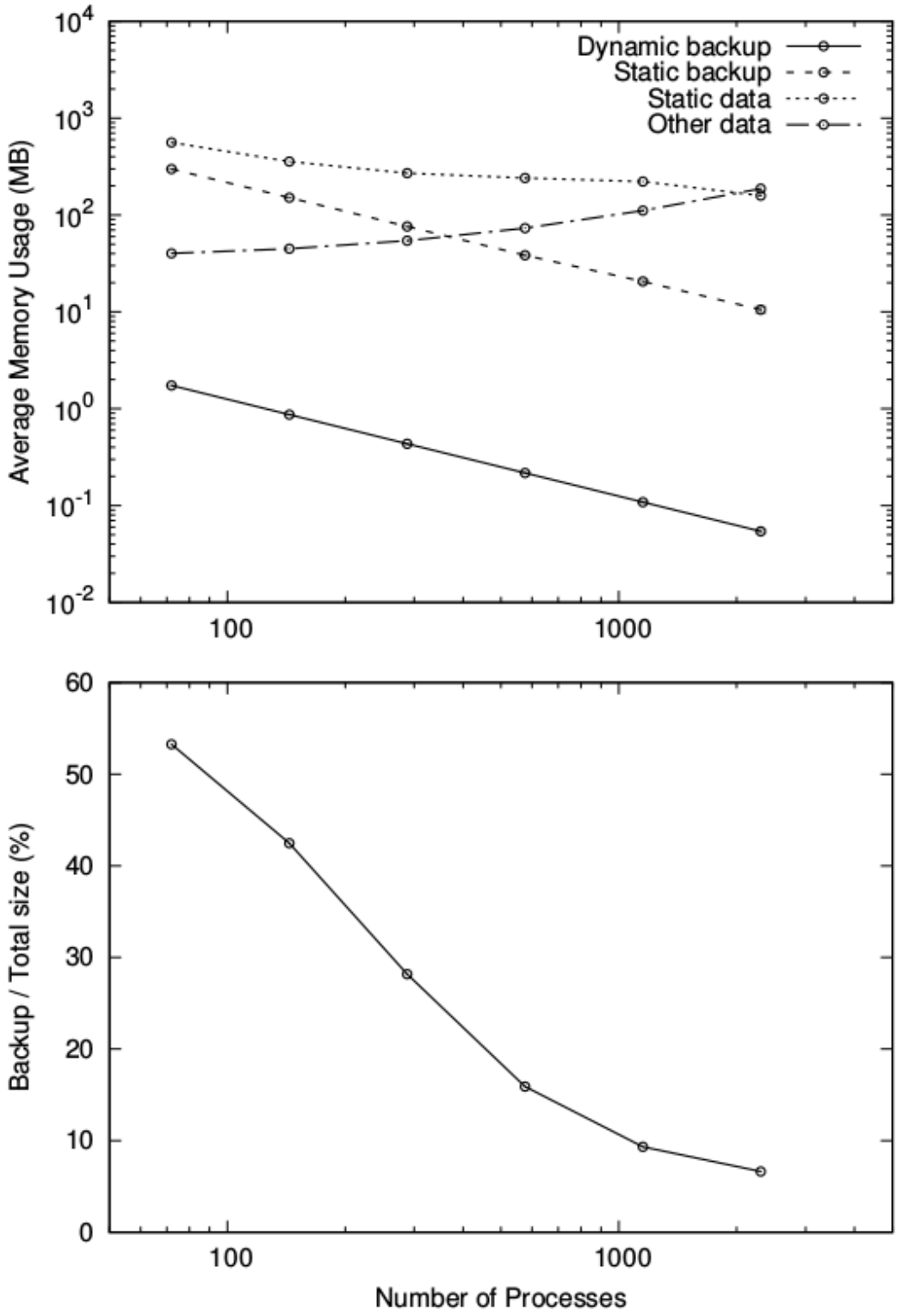


Figure 26: (a) Per-process memory usage as a function of number of processes. (b) Percentage size of static data backup vs full backup.

As a point of reference, we also show in Figure 25 the start-up time for the simulation using both the fault-tolerant implementation and the original unmodified version. The overhead of message-logging and the exchange of state preservation data can be seen to have limited impact on the start-up performance.

Figure 26(a) shows a breakdown of per-process memory usage for the major components of the fault-tolerant simulation code. Static data refers primarily to the matrix operators and element mappings which are constructed during initialisation of the solver and remain unchanged throughout the time-integration phase. This is the most significant component of the solver memory usage, particularly for lower core counts. At higher core counts, this quantity begins to saturate due to the emerging dominance of data structures whose size is independent of the sub-problem size being tackled by that process. The static backup component is the memory occupied (and subsequently transferred over the network) in providing resilience for the static data on a partner process. Specifically, this comprises the message logs from all MPI communications which were undertaken during the initialisation phase of the originating process. This decreases steadily with process count up to the limit of the number of available cores. In particular, for higher core counts, this quantity continues decreasing. The relative magnitude of the static backup size to the original static data size is presented in Figure 26(b) as a percentage. The difference in storage requirements drops considerably at higher core-counts and is approximately 6% at NP_{max}. The dynamic backup component represents the memory occupied by the copy of the solution vectors which are being advanced in time; therefore, this data changes at each timestep.

Since this is only vector data, the size is over two orders of magnitude smaller than the size of the static backup. The cost of storing the process's dynamic data and the backup of a partner process's dynamic data backup is approximately equal since, in the current implementation, no compression or erasure codes are applied to the checkpointing process and the work is equally distributed.

Finally, the other data component relates to the memory occupied by code and MPI initialisation. The latter increases with increasing core counts and becomes one of the dominant memory costs for the largest core counts.

2.6 Llama: A new library for Multilevel Checkpointing

Modern petascale machines experience node-failures on a regular, sometimes daily, basis [105][134]. Having to restart a job from scratch is a waste of energy, computational resources, and of the time of the domain scientists using numerical methods as part of their research.

In order to make numerical algorithms scale to the full size of large machines, the de facto standard for fault-tolerance is to have applications periodically write all essential data, required for a restart, to safe storage on a parallel file system [118]. In the event of failure of one or more nodes, the job may simply be rescheduled and restarted from the nearest checkpoint rather than from the beginning.

Periodically creating checkpoints on the parallel file system serves to mitigate the impact of failures on time and resources. The approach has served the community well for many years but is emerging as a bottleneck for the efficient scaling of algorithms on new machines. For many years, the speed of parallel file systems has been unable to keep up with the increase in compute power and memory of new generations of supercomputers. Time spent by compute nodes on writing data to safe storage is time not spent computing. The impact on parallel efficiency is application dependent, but as much as 10 to 20 percent overhead due to checkpointing is not uncommon [132]. In recent years, new approaches advocating, local recovery for local failures, has started to appear, challenging the notion that compute jobs, spanning hundreds or thousands of nodes, must terminate and restart in the case of failure of a single node. The new approach is necessary to facilitate scaling to big machines of upwards 50.000 nodes such as the planned exascale machine Aurora to be built at the Argonne Leadership Computing Facility[80]. Even if the mean time between failures (MTBF) of a node is a decade, the machine could experience 10+ node failures per day. Statistics compiled over years of supercomputing usage has shown that the majority of the failure incidents involve just a single node, or some small subset of nodes, depending on a shared set of resources. This knowledge may be used to design scalable methods to protect the data needed to rewind an application.

In the context of checkpointing, there are many possible ways to protect data locally, or near locally. A simple approach is to assign every rank with a buddy rank, the two may then be made to exchange critical data and keep it locally in-memory. Upon a failure, so long that no two buddies have failed collectively, data at the last point of exchange may be recovered and used for an application restart. Such lightweight local checkpoints may only recover from a subset of possible failure patterns, and it has therefore been proposed to use lightweight checkpoints as a complement to parallel file system checkpoints for increased efficiency and scalability. Protecting data locally is crucial for scalability as it allows to circumvent the bottleneck of the parallel file system. When compute nodes use local storage or memory, the bandwidth for protecting data increases linearly with the number of nodes which serves to avoid limiting network congestion. Several libraries for using lightweight checkpoints, both experimental and for production purposes, has been developed or is currently under development. Approaches of the different libraries under development is discussed in Section 2.6.1.

In this section, a new Library that has been developed, called Llama, is presented. In Switzerland, llamas are used as guards to protect flocks of sheep against predators, lonely or few in numbers. The Llama library uses layers of lightweight checkpoints to guard a cluster of nodes against failures, being particularly effective against local failures of limited scope. Hence the name.

Llama is built on top of ULFM-MPI, Jerasure and GF-Complete and allows for fast and scalable fault-tolerance in time-stepping MPI applications through multi-level checkpointing. A range of checkpoint types are supported, from strong to-disk checkpoints to light-weight in-memory checksum-checkpoints with group-local parallel encoding and decoding for maximum scalability. Light-weight checksum-checkpoints rely on parallel Reed-Solomon encoding and decoding. The library, unlike any other available, supports the usage of an arbitrary number of checkpoint types of arbitrary group and parity size so that checkpoints may be targeted to protect against single node or multiple node failures. It may also be used to provide protection of specific system-resources shared by multiple units, e.g. power supplies and network equipment or potentially entire racks.

The aim of the library is to provide a simple and easy-to-use interface for enabling applications to recover and rewind automatically, without restart, and continue in the face of single and multi-node failures. The library interface and basic usage, along with numerical experiments demonstrating the speed of the library components, is presented in this section. The goal is to eventually test the library for protecting applications written in Nektar++, an open-source software framework for high-performance scalable solvers for partial differential equations using the spectral/hp element method [100]. The material presented in this section has yet to be submitted for publication.

2.6.1 Checkpoint-Restart for HPC Fault-Tolerance

An introduction to HPC resilience using checkpointing is given in this section along with a review of libraries currently used, or under development. A brief introduction to User Level Failure Mitigation (ULFM) MPI is presented at the end. ULFM is a proposed extension to MPI, developed by the MPI Forum's Fault Tolerance Working Group. It is not a fault-tolerance library, rather it is an API that allows developers to implement fault-tolerant algorithms in MPI.

2.6.1.1 Checkpointing for Resilience

In current large-scale distributed memory applications, rudimentary fault tolerance is typically achieved by periodically synchronizing all nodes before writing a solution state to a checkpoint file as first demonstrated in [107]. These checkpoints are written to reliable storage, typically in the form of a parallel file system. Upon failure, an application may restart from a recent state by reading the checkpoint. Synchronization and writing of large files to the parallel file system introduces an overhead, limiting parallel efficiency of applications. Several techniques has been proposed to reduce the size of checkpoints so to minimize the impact. Data may be compressed before being written [109], or one may use an incremental checkpointing approach where only the difference between two checkpoints is stored [165][155], or combination of techniques including message logging[87][102]. It has also been proposed to mitigate the impact of to-disk

checkpointing by decreasing the frequency of which it is needed, through process replication. By running the same application on 2x nodes, the probability of both failing at the same time decreases substantially, which translates to a decrease in the optimal checkpointing frequency [97][106][90].

Diskless checkpointing, obtained by keeping all checkpoint data in-memory, has been recognized early on as one way of accelerating checkpointing [173][161]. Data may be distributed in such a way that if just a small set of nodes fail, it may be recovered from the nodes unaffiliated with the failure incident. A simple approach to realize this is by simply making many copies of the data to protect and distribute the copies. This approach, however fast, is problematic due to the large memory overhead it incurs. One approach for minimizing the memory footprint of both to-disk and in-memory checkpoint types, is to use RAID techniques or Reed-Solomon based encoding techniques [159][133]. Creating and storing parity codes has the advantage of requiring less storage, but comes at the cost of needing to encode data to protect it, and decode it upon recovery as well as added implementation complexity. Parity codes may be stored on additional nodes, or may be distributed equally amongst the nodes that created a particular set of parity codes.

In [182], the case was made for using two-level distributed recovery schemes, combining to-disk and in-memory type checkpoints. Their analysis suggested that a two-level system could be substantially more efficient as compared to the use of only a single level. It has been observed that most hard errors only effect a single node and when more nodes fail at the same time they typically do so in a predictable manor, i.e. a power supply unit, serving multiple nodes, fail. Ideally, a local failure should permit local recovery. Multi-level check-pointing addresses this problem by using different types of checkpoints, each of which have their own level of resilience and associated cost. Slower and more resilient levels, such as those made by writing data to the parallel file-system, may allow for recovery from many nodes failures. Cheaper and less resilient checkpoint levels may be utilizing node-local storage or memory[187].

The multi-level approach has received increasing attention from the community in recent years. Machines are now becoming so big, and the discrepancy between compute capacity and speed of parallel file systems so large, that the use of multiple levels of protection is no longer just a question of parallel efficiency and waste of resources. It is speculated that at exascale, the discrepancy may become so large that some applications will be unable to make computational progress when scaled to the full size of the machine, being trapped in a never-ending loop of failure-recovery-restart [175][118].

Recently, the use of node local solid-state drives and emerging nonvolatile memory technologies has been investigated for possible use in checkpointing[132][117]. Using local non-volatile storage allow for a fast-local approach to protect the data, as with standard random-access memory, but without the impact on memory consumption. In [103] it was suggested that checkpointing on node-local SSD drives might even be sufficient for exascale computation on certain applications.

2.6.1.2 Frequency of Checkpointing

A key thing to consider is the frequency with which to create checkpoints. Choosing an optimal frequency with which to checkpoint is about balancing the trade-off between loosing as little progress as possible in the event of a rollback, whilst not spending too much time on checkpointing too often. An estimate for the optimal length of the time-interval between checkpoints can be computed using the formula derived by Young [186], which assumes that failures occur as a Poisson process with failure rate λ

$$\tilde{\tau}_{opt} = \sqrt{2T_s T_f}$$

where $T_f = 1/\lambda$ is the mean time between failures and T_s the time to protect data. A higher order estimate for the optimal checkpoint interval was derived later by Daly [111] under the same assumptions

$$\tilde{\tau}_{opt} = \begin{cases} \sqrt{2T_s T_f}\left[1 + \frac{1}{3}\left(\frac{T_s}{2T_f}\right)^{\frac{1}{2}} + \frac{1}{9}\left(\frac{T_s}{2T_f}\right)\right] - T_s & , T_s < 2T_f \\ T_f & , T_s \geq 2T_f \end{cases}$$

With regards to multi-level checkpointing schemes, recent studies has investigated methods of optimal selection of the number of levels [91] and the frequency of check-pointing in the different levels[116][115]. In [176], a general theory for optimal checkpoint placement with arbitrary failure probability distribution was presented and [82] demonstrated how a checkpointing strategy could be modified if some form of online fault-prediction method is available.

Different checkpoint types may have different energy demands and the optimal frequency and number of levels in terms of runtime might therefore not be the same as the optimal frequency and the number of levels in terms of energy consumption, as has been demonstrated and explored in several papers [81][84][112]. In most clusters, nodes tend to share certain resources, the failure of which will result in the failure of all nodes. Different such components are likely to have different failure rates. Models to account for this was developed and presented in [83].

Extensive work has been done to develop different models, appropriate under different conditions, for choosing checkpoint levels and frequency of updating checkpoints in the context of optimizing both runtime and energy. One thing all models have in common is that they need estimates of mean time between failure of nodes to compute the optimal frequency of checkpointing. The sensitivity to poor estimates has not been examined as extensively. Studies presented in [144], running simulation using real workload data, suggested that the resulting parallel efficiency of an application is not particularly sensitive to the accuracy of the estimates used. This is perhaps not surprising. If one overestimates the failure rate, this will result in too much time spent checkpointing, but in turn one may save extra time upon recovery and restart. Conversely, if the failure rate is underestimated, an application will spend less time checkpointing. It appears that

in practice, the parallel efficiency of a given application is fairly robust with respect to choosing the checkpointing frequency.

2.6.1.3 Libraries for Automatic Checkpoint-Restart

Several libraries for fault-tolerance in HPC has been developed, or are in the process of being developed, for both experimental and production purposes.

One of the earliest examples of a library for multi-level checkpointing is from 2011, the fault tolerance interface (FTI). In FTI, global parallel-file-system checkpoints is combined with fast checkpoints written to node local storage in the form of SSDs[89]. Along with the checkpoint copy data, Reed-Solomon parity code blocks are computed and written so that an application may be restarted from the lightweight checkpoint even if some failures are non-transient, i.e., if a node experienced a hard failure and is no longer available. The authors demonstrated an eight percent checkpointing overhead on the TSUBAME2.0 supercomputer while running at over 0.1 petaflops and checkpointing every 6 minutes. FTI source code is still available in the authors git repository [88].

Scalable Checkpoint/Restart (SCR) is another library for checkpointing. It is well tested and has been used in early versions in production code at LLNL since 2009 and is supported with a comprehensive manual [154]. SCR uses a two-level checkpoint scheme. The more resilient level is a complete check-point to the parallel file system whereas the cheap, less resilient, checkpoint level is constructed using smaller groups of processors that save a check-point locally, either in-memory or on solid-state drives, whilst applying some redundancy scheme across the processors in the group. Two redundancy schemes are supported; a partner/buddy scheme where data copies are distributed, and an XOR model where parity blocks are coded in groups of 8 nodes and distributed evenly within nodes in a group as RAID5. The authors of SCR find that on the systems at LLNL, roughly 85 percent of all node failures may be recovered using the cheaper local checkpoint[153]. SCR may be used in conjunction with ULFM MPI trough the CRAFT library [171]. For a code to use the SCR Library, a list of 11 criteria must be satisfied. Most criteria are trivially satisfied by many applications, except for two:

- The code must take globally-coordinated checkpoints written primarily as a file per process.
- On some systems, checkpoints are cached in RAM disk. This restricts usage of SCR on those machines to applications whose memory footprint leaves sufficient room to store checkpoint file data in memory simultaneously with the running application.

The first requirement that a globally-coordinated checkpoint must be written as a file per MPI rank, this may be a limiting factor in some codes. If a code is MPI only, having a separate file for each process per checkpoint will create a very large number of files which may conflict with file system quotas. The second requirement is a potentially limiting factor for applications that are memory bound. Even if the XOR redundancy scheme is used, the total memory footprint of

the application will increase by more than a factor of two if solid-state drives are not available.

In addition to FTI and SCR, a library called Fenix has been developed for checkpointing purposes by researchers at Rutgers University and Sandia National Laboratory[129][130]. Fenix is based on ULFM MPI and provides an API to implement automatic rollback within the application code to avoid restarting jobs that have failed. In Fenix, data is protected by copying the checkpoint in-memory to a partner node's memory. The Fenix library was the first to demonstrate online multi-node recovery and automatic rollback at large scale when injecting actual failures. The library source code is available at a github repository along with rudimentary documentation on usage [178].

2.6.1.4 User Level Failure Mitigation (ULFM) MPI

The Fenix library achives online rollback, i.e. no restart, by using ULFM-MPI. User Level Failure Mitigation (ULFM) is a proposed extension to MPI developed by the MPI Forum's Fault Tolerance Working Group[95]. It is not a fault-tolerance library, but rather it is an API that allows developers to implement fault-tolerant algorithms in MPI. According to the authors, ULFM was designed to manage failures following three fundamental concepts: 1) simplicity, the API should be easy to understand and use in most common scenarios; 2) flexibility, the API should allow varied fault tolerant models to be built as external libraries and; 3) absence of deadlock, no MPI call (point-to-point or collective) can block indefinitely after a failure, but must either succeed or raise an MPI error. A prototype of ULFM is available to be used with the OpenMPI compiler[96]. As with Fenix, Llama is using ULFM MPI to implement the failure detection, communicator repair, and rollback methods.

2.6.2 The Llama Library

The Llama library is written in C++ and designed for use with MPI applications. It supports fault-tolerance trough checkpointing with automatic rollback without application restart through the use of ULFM MPI. To protect application data, the user may specify an arbitrary number of checkpoint levels. The levels may consist of both regular to-disk checkpoints to the parallel-file-system, and topology aware in-memory memory-conserving checksum checkpoints, each of which may be of arbitrary group size with an arbitrary number of checksums parity code blocks. In section 2.6.3, the design of the library is outlined, and in section 2.6.4 the usage of the core functionality is described.

2.6.3 Design

Llama is designed to be minimally invasive. The target code itself needs very little, if any, modification. Unlike other libraries, rather than modifying preexistent lines of code, code is added to indicate what data is essential to protect and to express how the application should proceed in the event of a rollback. The fundamental object that any application, using the library, must instantiate is the Llama Guard. The object must be created in the beginning of the application and is responsible for keeping track of arrays being protected, and checkpoints used, as well as initiating recovery and repair of the MPI communicator.

The functionality provided by the library is to be used in a loop within which the computational parallel work to protect takes place. Within the loop, the Llama Guard must perform a status check at the beginning each iteration. If the status check detects one or more failed ranks, the communicator will be repaired by the guard by injecting spare-ranks to take the place of failed ranks. The status check call will then return a flag, indicating that the application should perform a rollback and recovery of data. Llama provides an interface trough the Guard that expose data protected at a previous checkpoint to enable a rollback, however it is up to the user to write the functionality needed to do use the data to recover an earlier state as such an operation is application dependent. Instructions on rollback and data recovery is given in section 2.6.5.

Following declaration of the Guard, before entering loop, the library user may add as many levels of checkpoints as needed. Two different categories of checkpoints are supported. The to-disk checkpoint, which protects data by writing it to the parallel-file-system using MPI-IO, and in-memory checksum checkpoints that protects data by encoding parity blocks and distributing these in-memory across ranks. Upon failure, the Guard will find the most recently updated checkpoint for rollback to. If more than one checkpoint has been updated at the same index, the Guard will choose the checkpoint which it perceived as being faster to recover from.

The implementation of both checkpoint types is derived from a pure virtual class *checkpointinterface*. Other checkpoint types may be added to the library if needed, but to be compatible with the Llama Guard, they must inherit from *checkpointinterface*. Figure 27 contains a dependency graph of the Llama header that applications must include. The interface header contains the declaration of both the *Guard* class and the *checkpointinterface* as well as the definition of all templated methods of the two classes. The two methods of checkpointing are explained in greater detail in the two sections that follow.

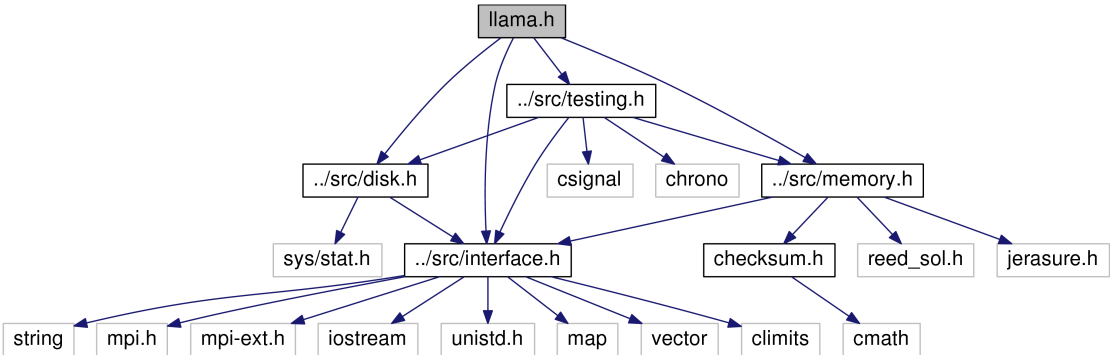


Figure 27: Dependency graph of the Llama header that must be included in applications using the library. The class declaration for the two checkpoint types supported are written disk.h and memory.h. They both derive from a pure virtual class checkpointinterface in the interface header. The interface header also includes the decleration of the Guard class. The testing header includes functionality used in the ctest package which includes 30 tests, testing to verify correct execution of the various components of Llama and their interactions.

2.6.3.1 Protecting data using the parallel file system

MPI-IO is used to write and read checkpoint data to the parallel file system. Distributed 1D arrays are written to a single binary file with each rank writing to it's own section of the file for optimal performance. 2D arrays are protected as arrays of 1D arrays, without closing and opening file objects in between row change. The bandwidth of writing and reading data may be slow for 2D arrays if the number of columns is very small due to the overhead of sending many small messages. This could be improved by implementing features to rearrange data before sending if the number of columns is small, but as of now this has not been implemented. 3D arrays are treated as arrays of 2D arrays, thus they are subject to the same limitation that the data transfer rate may decrease when the number of columns is small.

2.6.3.2 Protection data using in-memory checksum codes

Protecting data locally in-memory is fast. Transferring data to a nearby node over the network is in general much faster than writing it to a parallel file system. In addition, when compute nodes keep checkpoint data in nearby node-local storage or memory, the bandwidth for protecting data increases linearly with the number of nodes. This makes in-memory checkpointing potentially highly scalable. The disadvantage of protecting data in-memory on nearby ranks is the added consumption of memory which may be a limiting factor in some applications. When protecting data by simply copying it to nearby nodes, the overhead in terms of memory consumption is at least 200 percent.

One way of decreasing the memory footprint, required for a checkpoint, is to use some form of erasure code to encode checksum parity blocks rather than making extra copies of the data to protect. Generally speaking, an erasure code transforms a data block of k symbols into a larger data block of $n > k$ symbols in such a way that the original data block may be recovered using any subset of k symbols from the n symbol data block. Erasure codes are widely used in data-centers to protect data against disk failures.

In the context of HPC, several studies have demonstrated the usage of erasure codes for fault tolerance. In FTI, Reed-Solomon erasure code was used to encode/decode data saved locally on solid state disks[89] and in the SCR library XOR parity bits are computed and stored distributed in-memory to protect against single node failures [153]. In the Llama library, Reed-Solomon encoding is used to create in-memory memory conserving checksum checkpoints. Reed-Solomon codes have found use for a wide range of applications such as storage media, data transmission and data centers[164][184]. Reed-Solomon codes can be used to encode an arbitrary number of parity codes $p = n - k$ for a data block with an arbitrary number of symbols k thereby allowing for recovery of protected data for up to p failures.

The process of computing p parity code blocks $\{c_1, \cdots, c_p\}$ from k data blocks $\{c_1, \cdots, c_p\}$, can be seen as matrix-vector multiplication where all components are elements in a Galois field

$$\begin{bmatrix} c_1 \\ \vdots \\ c_p \end{bmatrix} = \begin{bmatrix} a_{1,1} & \cdots & a_{1,k} \\ \vdots & \ddots & \vdots \\ a_{p,1} & \cdots & a_{p,k} \end{bmatrix} \begin{bmatrix} d_1 \\ \vdots \\ \vdots \\ d_k \end{bmatrix}$$

and the matrix A a Vandermonde matrix. In the event that one or more data and/or code blocks are lost, they may be recovered trough another matrix-vector multiplication using the inverse of A, or the inverse of a subset of A along with the remaining code and data blocks.

In Llama, a library called Jerassure is used to create the matrix of Gallois field coefficients, needed for both the encoding and the decoding procedure. Jerasure is an open source library written in C that supports erasure coding in storage applications[162]. Another library, GF-Complete, is used for fast multiplication of coefficients with entire data blocks[160]. The multiplication of each coefficient can be done in parallel as the data blocks are stored on different nodes.

MPI reduce is used to compute the bitwise XOR reduction operation required, to compute the sum for each code-block. Through the parallel reduction operation, the sum is simultaneously computed and distributed. The in-memory checksum checkpoints are created in independent groups of size n, with the p parity codes distributed equally among the n nodes. Equal distribution is achieved trough block level stripping as is done in RAID5 and RAID6, but here implemented in-memory using MPI. With the distribution of parity code-blocks, the memory overhead associated with the checkpoint in addition to the copied data is

$$\sim \frac{p}{n-p}.$$

In the limit that the communication in the MPI bitwise reduction operation is the dominant cost of encoding the checkpoint, the time-to-encode will scale as

$$\sim p\lceil \log_2(n)$$

if all ranks in a group are involved in each reduction. In Llama, however, an array of communicators of size np is created when the checkpoint is declared. The array includes all possible patterns of reduction to avoid including group members not relevant in a reduction. The cost of updating the checkpoint therefore scales as

$$\sim p\lceil \log_2(n-p)$$

The improved scaling comes at the cost of an added overhead when creating the checkpoint object as well as when repairing it after a failure. The in-memory checkpoints may be made topology aware in the sense that the user may specify a critical distance that ranks within a group must be separated. If a checkpoint is to protect entire nodes, the user should provide the number of ranks running per node. If the checkpoint is to protect some other entity or multiple nodes, this should be specified in this distance.

2D arrays are protected by treating columns as blocks and each row as a new stripe, similar to RAID6. Due to this design, as with the to-disk checkpoints,

performance may suffer if the number of columns in an array is very small. This may be mitigated by repacking data before protection. 1D arrays are treated by creating a dummy array of pointers to transform it into a 2D array. 3D arrays are treated as many 2D arrays. As with the to-disk checkpoints, declaration is templated so that arrays of all fundamental types may be protected.

2.6.4 Usage

Llama seeks to provide a simple and easy-to-use interface for enabling applications to recover automatically and continue in the face of single and multi-node failures, or the failure of larger shared system resources such as power supplies or network equipment. In this section, library usage is briefly outline. The library uses components of ULFM-MPI, Jerasure and GF-Complete. The latter must therefore be installed first to build the library. After the installation of dependent libraries, Llama can be built from source by cloning the git repository available at [157] with a c4science account. As of now, the build system is rudimentary and the main cmake files needs to be modified manually with the local path to headers and library objects of ULFM-MPI, Jerasure and GF-Complete. Additional details for building the library, and associated examples, are available in the readme file in the repository.

2.6.4.1 Llama Guards

To make a time-stepping MPI application fault tolerant using Llama, the first step is to create a llama::guard object in the beginning of the application right after *MPI_Initialize*. The constructor for the llama::guard takes three arguments. The first is the address of the global communicator, the second is the number of ranks in the communicator that should be used as spares rather than active workers, and the third a boolean that indicates if the application should throw an error object if recovery is not possible.

```
llama::guard my_llama( MPI_Comm* my_world_comm, uint32_t my_num_spar
e_ranks, bool return_errors );
```

If *return_errors* is true, calls to llama::guard methods will throw an error object if recovery is not possible rather than exiting with failure and error message on the std::cerr stream. The situation of recovery not being possible may happen in one of two different situations: A failure pattern for which the checkpoints do not protect has occurred, or if the Guard has run out of spares to replace failed ranks. The Guard, when created, removes *my_num_spare_ranks* number of ranks from the *my_world_comm* communicator, and places an error-handler on *my_world_comm* so that if a rank in *my_world_comm* fails, MPI operation using *my_world_comm* will throw an error object and return rather than abort. In addition to splitting the main global communicator into two pools, a worker and a spare rank pool, and creating an error-handler for *my_world_comm*, the Guard is responsible for the following tasks:

- Keeping track of which arrays need protection
- Keeping track of the status of all checkpoints.
- Monitoring the status of the application and initiating recovery if a failure is detected.
- Choosing which checkpoint to recover from, when a failure is detected.

To notify the llama::guard as to what data needs to be protected, a member function called *ProtectArray* is used.

```
my_llama.ProtectArray<type T>(const std::string &key, T* p_array, ui
nt64_t dim1);
```

Each array to be protected must be supplied with a unique key in the form of an std::string. This key is used to identify which array to recover if recovery is needed. If one attempts to protect an array with a key that is already in use, ProtectArray will compare the new array and *dim1* input with the already existent. If they are identical, ProtectArray will ignore the call and if not, it will throw an error. *p_array* should be a pointer to the array that needs to be protected and *dim1* the number of elements in the array. Equivalent methods exist for 2D and 3D arrays, with the interface for these are as follows:

```
my_llama.ProtectArray<type T>(const std::string &key, T** p_array, u
int64_t dim1, uint64_t dim2);
my_llama.ProtectArray<type T>(const std::string &key, T*** p_array,
uint64_t dim1, uint64_t dim2, uint64_t dim3);
```

Arrays that contain vital information to allow for a quick restart of the application should all be marked for protection. The arrays may be marked for protection in the beginning of an application right after declaring the llama::guard, but may also be added dynamically during the time-stepping phase. Be aware, though, that ProtectArray is collective over all workers, i.e., all ranks in *my_world_comm* after creation of *my_llama*. Each rank must therefore specify an array to be protected, it is not required that the arrays are of the same size for all ranks. Though performance and memory efficiency is higher if all arrays belonging to the same key is of similar size. Similarly, there is an overhead involved in creating and updating arrays. The 1D arrays should ideally exceed 1000kb for the overhead involved in the synchronization between ranks to be mostly negligible. If a given array no longer needs to be protected, the llama::guard can be notified through the methods:

```
my_llama.ForgetArray(const std::string &key);
my_llama.ForgetAllArray();
```

2.6.4.2 Checkpoint Objects

Under the hood, arrays are protected by checkpoints. The user must specify which kind of checkpoints should be used. Llama supports two overall types of checkpoints; checkpoints to the parallel-file-system and checkpoints that are kept in-memory and distributed in groups across ranks. The constructor for creating checkpoints of the first type looks like this:

```
llama::checkpoint::disk my_disk_checkpoint(llama::guard* my_llama);
llama::checkpoint::disk my_disk_checkpoint(llama::guard* my_llama, s
td::string file_name );
llama::checkpoint::disk my_disk_checkpoint(llama::guard* my_llama, s
td::string file_name, std::string relative_path );
```

The first argument is a pointer to the llama::guard used, here simply *my_llama*. The second argument is a string, specifying the file name to use for the file written to the parallel-file-system to protect the data. If no name is supplied, a default name convention is followed. The third argument is a relative path in case the user wishes that the checkpoint files are placed somewhere else than in the same folder as the executable. Checkpoints written to the disk has the advantage of being completely resilient, i.e. all arrays marked for protection may be recovered no matter how many ranks are lost. The only limitation to recovery is the size of the pool of spare ranks. The disadvantage of checkpoints on the parallel-file-system is that they are slow to create for large jobs using hundreds of nodes as the bandwidth is limited. Llama supports light-weight in-memory checksum checkpoints. Checkpoints of this type are stored in-memory along with a small parity code that can be used to recompute the data lost if ranks have failed. Parity codes are computed locally in groups in parallel, which allows for the bandwidth of data encoded to scale linearly with the number of ranks in a job. Reed-Solomon codes are used for a limited memory footprint with high resilience. An in-memory checkpoint may be created like this:

```
llama::checkpoint::memory my_memory_checkpoint(llama::guard* my_llam
a, uint32_t connected_ranks, uint32_t num_ranks_per_group, uint32_t
num_parities_per_group);
```

As with the to-disk checkpoint, the first argument is a pointer to the llama::guard which serves to notify the Guard of the existence of the checkpoint and as to give the checkpoint access to information about which arrays must be protected. The second argument indicates the number of consecutive ranks that is deemed likely to fail together. This would most often be a node-entity, i.e. the number of ranks spawned on each node, but could in practice be anything. For instance, if multiple nodes share a power supply, then a checkpoint could be created with *connected_ranks* being the number of ranks which depend on the same power-supply. The third argument *num_ranks_per_group* is the size of the groups in which the parity data will be computed. The parity data is stored distributed across all ranks within a group. The fourth and final argument is the number of parity data blocks to compute within a group. In simpler terms, *num_parities_per_group* indicates how many failures are allowed to occur simultaneously within a group.

Say for example that *num_parities_per_group* = 1 and *num_ranks_per_group* = 8, then in each group of eight sets of *connected_ranks*, all protected arrays are recoverable so long as no more than one set of *connected_ranks* have failed. The group size *num_ranks_per_group* must be chosen so that *num_worker_ranks* / (*num_ranks_per_group* * *connected_ranks*) is a positive integer. The choice of *num_ranks_per_group* and *num_parities_per_group* will not only affect how resilient to failures the checkpoint is, but also how fast the checkpoint can encode and decode the parity data for the protected arrays as well as its memory footprint. In general, as *num_ranks_per_group* becomes larger, the memory footprint will become smaller but the encoding and decoding time will become larger. As *num_parities_per_group* increase, the checkpoint becomes more resilient since more sets of *connected_ranks* may fail without compromising the protected arrays, but the memory footprint increases and so does the encoding and decoding time. A simple constructor also exists to create a fast in-memory checkpoint layer

```
llama::checkpoint::memory my_memory_checkpoint(llama::guard* my_llam
a, uint32_t connected_ranks);
```

Here we only notify the checkpoint of the number of *connected_ranks* that are expected to be likely to fail together. The checkpoint sets *num_ranks_per_group* will then be set by the llama::guard to be the integer nearest to eight that satisfies *num_worker_ranks* / (*num_ranks_per_group* * *connected_ranks*) == integer. *num_parities_per_group* will be set to one. We consider a group size of eight with one parity block to be a good lightweight checkpoint for most purposes, thus it is the default option. With a group size of eight, the checkpoint is fairly memory conserving without being slow to encode or decode and one parity block will allow for recovery upon failure of up to one set of *connected_ranks* in every eighth set. When a checkpoint has been declared, it may be updated by the user using the *Update()* method:

```
my_disk_checkpoint.Update(uint64_t idx);
my_memory_checkpoint.Update(uint64_t idx);
```

The update method will loop over all arrays that has been marked in the llama::guard as critical and protect the data in each of them. The to-disk type checkpoints simply write the array to the parallel-file-system whereas the in-memory checkpoints will create an in-memory copy as well as encode parity data to be able to decode and recover data if ranks are lost. The provided index is essential. Upon recovery, the llama::guard will choose the checkpoint with the highest index among the checkpoints that are able to recover from a given failure pattern. Any checkpoint can be updated at anytime during the time-stepping procedure that the user deems necessary. If several different recoverable checkpoints with the same idx exist, the llama::guard will recover the data content using the checkpoint it deems to allow the fastest recovery.

2.6.4.3 Checking Application Status and Recovering Data

Here we outline how to recover the application in the event that failure on one or multiple ranks occur during a time-stepping procedure. A member method exists `my_llama.CheckStatus(uint64_t* idx)`

CheckStatus() plays an important role in the fault-tolerent time-stepping procedure and in the recovery. All spare ranks that were split from *my_world_comm* will proceed until they reach the *CheckStatus()* call and wait here until needed to replace a failed worker rank. In the beginning of each step in the time-stepping loop, the status of the application should be checked. The call to *CheckStatus()* will return one of four different statuses:

- LLAMA_STATUS_SUCCESS
- LLAMA_STATUS_REPAIRED
- LLAMA_STATUS_FAILED
- LLAMA_STATUS_COMPLETE

The first status flag, LLAMA_STATUS_SUCCESS indicates that no failed ranks could be detected in *my_world_comm* so the application can proceed safely. LLAMA_STATUS_REPAIRED indicates that one or more ranks were found to have

failed but they were successfully repaired, i.e., one or more ranks in *my_world_comm* has been replaced with one or more ranks from the pool of spare ranks and released from *CheckStatus()* together with all the other worker ranks. This status flag indicates that the user should take action to role back to a previous checkpoint idx at which a sufficiently resilient checkpoint was updated. LLAMA_STATUS_FAILED indicates that one or more ranks were found to have failed, but it wasn't possible to repair the communicator. This could happen if the llama::guard has run out of spare ranks or if the llama::guard detects a failure pattern for which none of its associated checkpoints may recover. This status flag will only be returned in the event that return_errors was set to true when creating the llama::guard, otherwise the application will write an error message on the std::cerr stream and exit with the EXIT_FAILED status. The final possible status flag LLAMA_STATUS_COMPLETE is only returned with a spare-rank and indicates that the application has completed and that the spare rank should proceed to clean-up.

In the event that a LLAMA_STATUS_REPAIRED status flag is returned, the user must specify how the application should be repaired. Llama simply exposes access to the content of all arrays that was marked as protected at the most recent checkpoint idx for which recovery is possible. To access the data content of an array at the most recent checkpoint idx, the method *RecoverArray()* is used.

```
my_llama.RecoverArray<class T>(const std::string &key, T* p_array, u
int64_t dim1)
```

key is the key that was previously used to identify the array and *T* is the type of the objects stored in the array. *p_array* indicates to where the recovered data should be copied. For ranks that had not failed, this might very well be the same pointer, whereas for a newly introduced worker, rank relevant data structures must be created. As before *dim1* indicates the number of elements in the array. The same interface is used for 2D and 3D arrays

```
my_llama.RecoverArray<class T>(const std::string &key, T** p_array,
uint64_t dim1, uint64_t dim2)
my_llama.RecoverArray<class T>(const std::string &key, T*** p_array,
 uint64_t dim1, uint64_t dim2, uint64_t dim3)
```

The example folder in the git repository contains several examples on how to make applications fault-tolerant.

2.6.4.4 Finalizing the Environment

Before finalizing the MPI environment, the Llama environment must be finalized using *my_llama.Finalize()* to clean up all checkpoints and to free all spares. If the environment is not finalized, unused data files may remain on the parallel file system.

2.6.4.5 Usage examples

A code examples are available in the Git repository. In the example in the appendix, a small code for finding a numerical approximation to the solution of the 2D Euler equation in parallel using MPI is made fault tolerant using the Llama library. Rank failures are forced by raising SIGKILL on a subset of nodes to demonstrate recovery in case of two different failure patterns. A detailed outline of the example is given in the code comments.

2.6.5 Numerical Experiments

Numerical experiments demonstrating the overhead initialization cost of creating the Llama Guard, adding checkpoints and marking arrays to protect are presented in sections 2.6.5.1, 2.6.5.2, and 2.6.5.3. In section 2.6.5.4, scaling tests on the walltime cost to periodically update checkpoints are presented. The cost of recovering data from a checkpoint is presented in section 2.6.5.5. All measurements are made for to-disk checkpoints, and for four different types of in-memory checksum checkpoints as outlined below:

To-Disk
Arrays to be protected are written to the GPFS filesystem using MPI-IO.

In-Memory G32P4
Arrays encoded in groups of 32 ranks. Four parity code blocks are created per code stripe. 14.3 percent memory overhead.

In-Memory G16P2
Arrays encoded in groups of 16 ranks. Two parity code blocks are created per code stripe. 14.3 percent memory overhead.

In-Memory G8P2
Arrays encoded in groups of eight ranks. Two parity code blocks are created per code stripe. 33.3 percent memory overhead.

In-Memory G8P1
Arrays encoded in groups of eight ranks. one parity code blocks are created per code stripe. 14.3 percent memory overhead.

The checkpoint types are listed in order of increasing resilience. The to-disk type checkpoints may recover lost data regardless of how many ranks have failed. In-memory checksum checkpoints may recover so long as the number of lost ranks within a group is smaller than the number of parity code blocks encoded and stored per stripe. Smaller groups with few parity code blocks may recover from fewer failure patterns and may thus be considered less resilient. Small groups with few code blocks are, however, expected to be faster to update as fewer operations and less communication between ranks in a group is needed to compute the checksum code blocks. Weak and strong scaling tests has been made from 112 to 7168 cores for all numerical experiments. The EPFL Fidis general purpose cluster was used to run the scaling tests. Each node in the cluster has two Intel Broadwell 2.6Ghz processors with 14 cores each, i.e. 28 cores per node. The cluster network fabric is Infiniband FDR fully-non-blocking, with a fat-tree topology.

2.6.5.1 Creating the Llama Guard

The first step in making an application fault tolerant by way of periodic checkpointing using Llama, is to declare a Guard object following the initialization of the MPI environment:

```
llama::guard::guard(MPI_Comm &input_comm, int num_spare_ranks, bool
return_errors);
```

The first argument is a pointer to a duplicate of the world communicator, containing all worker and spare ranks. The second argument tells the Guard how many ranks to put aside in a pool of spare worker ranks. The third argument, if true, tells the Guard to throw error objects rather than exit upon encountering failure patterns which are not recoverable. The constructor of the Guard initiates various structures needed, and it removes *num_spare_ranks* ranks from the *input_comm* communicator and places these ranks in a pool for later use in case of failures. A scaling test is presented in Figure 28. The walltime to create the Guard does not depend on checkpoints added later, and only increases weakly with the number of cores. The measured time varies substantially between measurements.

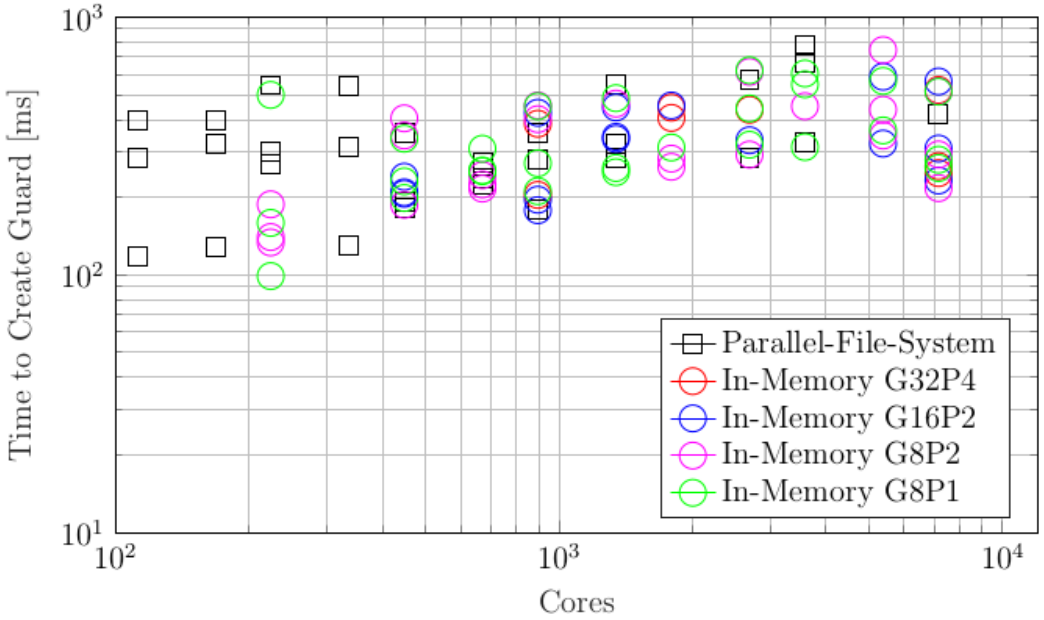


Figure 28: Scaling test, time to create Llama Guard as a function of the number of cores as tested on the EPFL Fidis cluster. The time to create the checkpoint increases slightly with an increasing number of cores, but varies mostly due to difference in network load and job placement on the shared general purpose cluster.

2.6.5.2 Adding a Checkpoint to the Guard

For the Guard to protect arrays, one or more checkpoint objects must be associated with the Guard. To-disk and in-memory type checkpoints may be added using their respective constructors and passing a pointer to the Guard.

```
// Declaration of the constructor for the to-disk type checkpoint
void llama::checkpoint::disk::disk(llama::guard *MyGuard)
// Declaration of the constructor for the in-memory type checkpoint
void llama::checkpoint::memory::memory(llama::guard *MyGuard, bool c
opy_protected_data, int connected_ranks, int local_group_size, int n
um_parities_per_group)
```

Within the constructor for declaring a to-disk type checkpoint, only a few local initialization operations are needed. For the in-memory checksum type checkpoints, global operations are needed to verify that all workers are creating in-memory type checkpoint of the same group size, using the same number of parity blocks. In addition, an array of local communicators are created within each group. The overhead of attaching an in-memory checkpoint to the Guard is

therefore higher than for the to-disk type checkpoint. As may be seen from the scaling test in Figure 29, the in-memory checksum checkpoint is an order of magnitude more expensive to initialize.

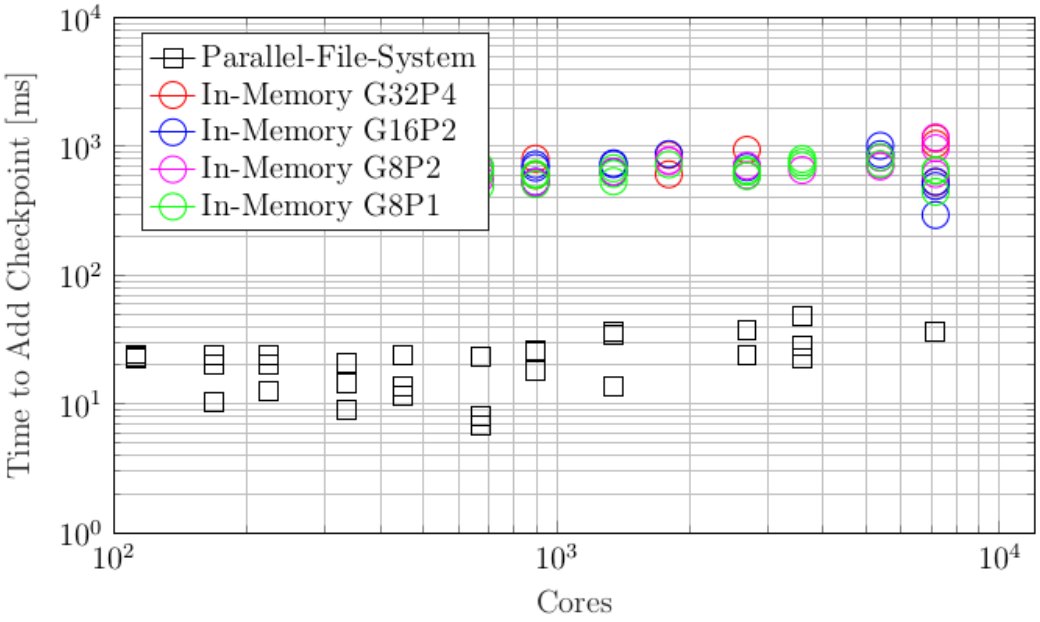


Figure 29: Scaling test, time to attach a checkpoint-type to the Llama Guard as a function of number of cores. The time increases slightly with an increasing number of cores, the overhead of checking parameters and creating a matrix of local communicators makes the in-memory type checkpoint slower to initialize.

2.6.5.3 Adding an Array for the Guard to Protect

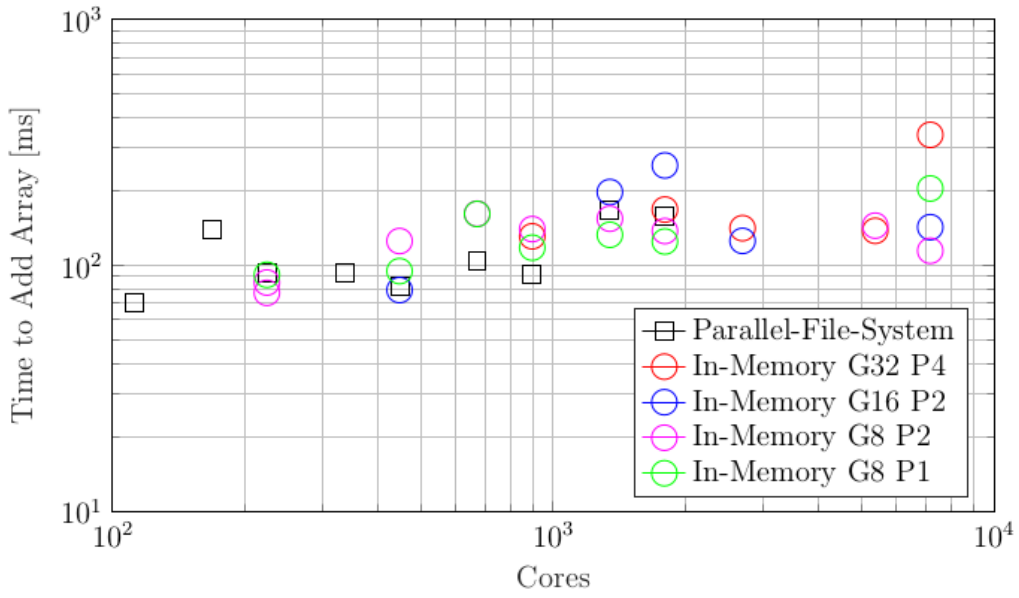
Any array, containing data needed upon recovery, must be protected by the Llama Guard created. To notify the Guard of the existence of the array, the *ProtectArray* method of the Llama Guard class is used. The method is a template on the data type of the array to be protected, and is overloaded by the number of arguments to distinguish between 1D, 2D, and 3D arrays.

```
// Method for 1D arrays
template<class T> void guard::ProtectArray(const std::string &key, T
 *p_array, int dim1)
// Method for 2D arrays
template<class T> void guard::ProtectArray(const std::string &key, T
 **p_array, int dim1, int dim2)
// Method for 3D arrays
template<class T> void guard::ProtectArray(const std::string &key, T
 ***p_array, int dim1, int dim2, int dim3)
```

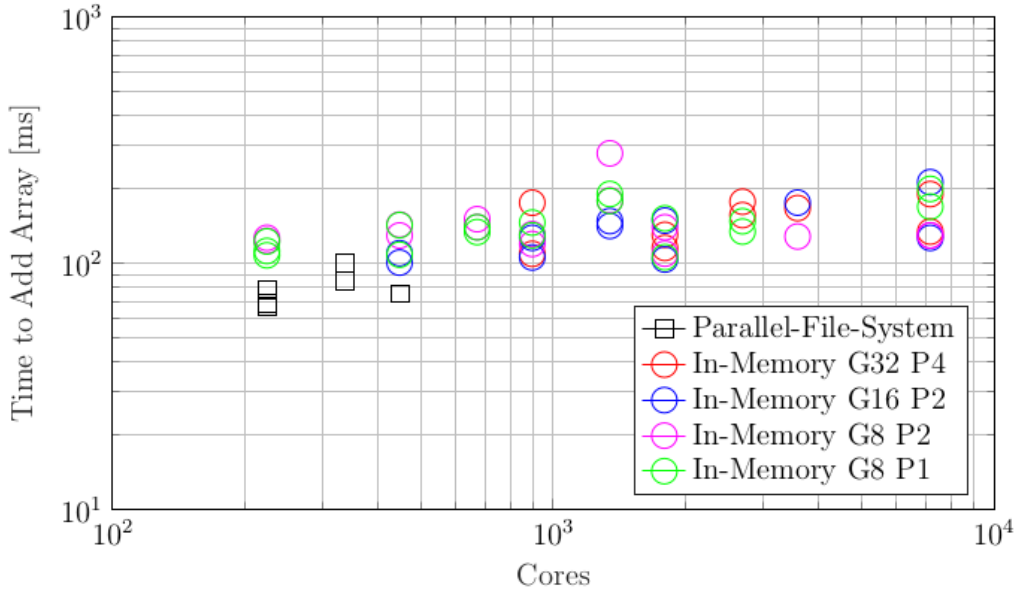
The method checks if the key supplied is unique- If it is not the application exists with an error code, or an error object is thrown, depending on how the Guard was initialized. If the key is unique, the array pointer and array dimensions are added together to a map connecting the key and the array information. Within the method, all checkpoints associated with the Guard is notified of the existence of a new array to be protected. Guard objects are friends of checkpoint objects, so checkpoints are notified by calling a protected method that initiates whatever structures that may be needed for the checkpoint type. Only two types of checkpoints exists, the to-disk that uses MPI-IO to protect an array on the parallel-

file-system, and the in-memory which encodes checksum parity codes to protect the data and distribute these parity codes in a local group. In the case of the to-disk checkpoint, no action is needed. For the in-memory checkpoint, memory is allocated for the parity code blocks that will be written. If other types of checkpoints are to be implemented, they must inherit from the pure virtual *checkpointinterface* class and implement all required functionality.

In either case, before notifying checkpoints of the addition of an array to protect, a check is performed across all workers, verifying that all workers are adding an array with the same unique key. The operation is thus global and must be accessed by all workers to avoid deadlock. Figure 30 contains a weak and a strong scaling plot of the overhead walltime associated with adding an array for the Llama Guard to protect. The numerical experiment indicates that there is little or no difference in the cost of adding an array between the to-disk and the in-memory type checkpoints. The principal cost of the *ProtectArray* method is therefore likely the addition of a new array to the map, followed by the global agreement operation in the Guard that is independent of the size of the array.



(a) Weak Scaling – 12MB per Array per Core



(b) Strong Scaling – 128GB Distributed Array

Figure 30: The wall-time cost of adding an array to the Llama Guard as measured on the EPFL Fidis general purpose cluster for 112 to 7168 cores. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made 3 times to illustrate the variance in timings due to job-placement and network load when using a shared cluster. The cost of adding an array is observed to be almost constant, only vaguely increasing with an increasing number of cores.

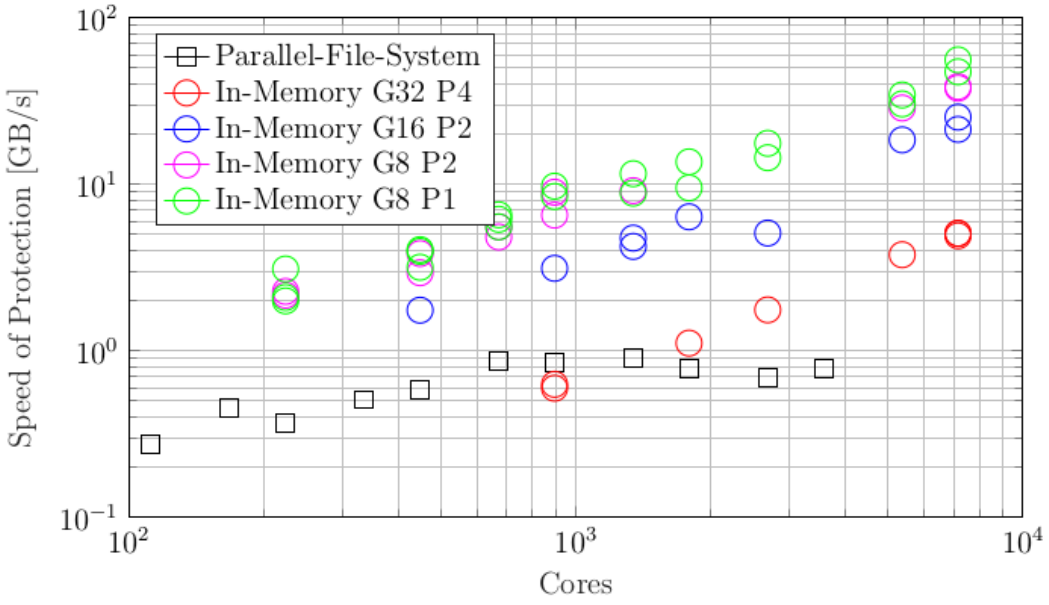
2.6.5.4 Cost of Updating Checkpoints

In the fault tolerant application, checkpoints protecting data will need to be periodically updated. This is done using the *Update* method. The method is a pure virtual function in the *checkpointinterface* class from which all checkpoints derive.

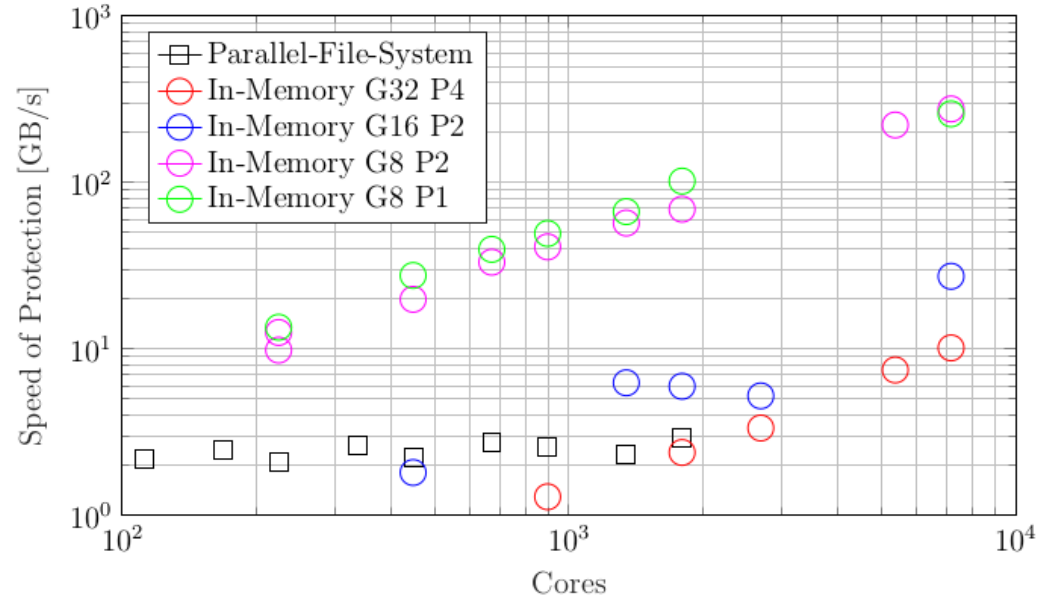
```
// Update all data arrays and encode all checksums in checkpoint
void llama::checkpoint::memory::Update(int new_checkpoint_idx)
void llama::checkpoint::disk::Update(int new_checkpoint_idx)
```

When calling the function, the checkpoint will update the protection of all arrays marked by protection. The variable *new_checkpoint_idx* indicates the index for the new checkpoint. The index provided must be greater than any previous index used.

The underlying action of *Update* depends on the checkpoint implementation. For the to-disk checkpoint, MPI-IO is used to write a copy of the array to be protected by the parallel file system. Once the array has been written, an agreement check is performed between all workers to verify that the new checkpoint is safe before deleting the file containing the previous checkpoint. For the in-memory checkpoint, all arrays are encoded using Reed-Solomon erasure code as outlined in section 2.6.3. The walltime cost of updating the checkpoint may be the most critical part of the Llama library as, unlike the creation of the Guard and adding arrays, the update will be performed periodically throughout the execution of the code to protect. Steps has therefore been taken to make the update as fast as possible. All essential checks on identifier keys and array dimensions has been moved to the *ProtectArray* method of the Guard class, used to notify the Guard of arrays to be protected. No checks are performed inside *Update*. The operation is global over all workers so deadlock will ensue if any workers are missing unless it being due to failure in which case it returns throwing an error object. Figure 31 contains weak scaling measurements on the EPFL Fidis cluster for 112 to 7168 cores for the protection of a 2D array. Measurements has been made for the to-disk type checkpoint as well as for four different types of in-memory checksum checkpoints. Each node on the Fidis cluster has 28 cores. In all four in-memory cases the number of adjacent nodes assumed likely to fail together is set to the size of a single node, 28, i.e., ranks 0,28,56,… will be placed in the zero'th group, rank 1,29,57,… in group 1 and so forth. At 12MB per core close to ideal scaling is observed when using a group size of eight with one or two parity codes per stripe. In the limit of using 7168 cores, Llama encodes data at a rate of 300GB/s. Which is more than two orders of magnitude faster than what may be achieved protecting the arrays using MPI-IO in the parallel-file-system checkpoint. Figure 32 contains strong scaling tests for a 16GB 2D array and a 128GB 2D array. In the strong-scaling test, the size of the 2D array per core decreases as the number of cores increases, making it more challenging to achieve perfect scaling. As in the weak scaling test, the in-memory checksum checkpoint is up to 200 times faster than the to-disk type checkpoint when using 7168 cores.

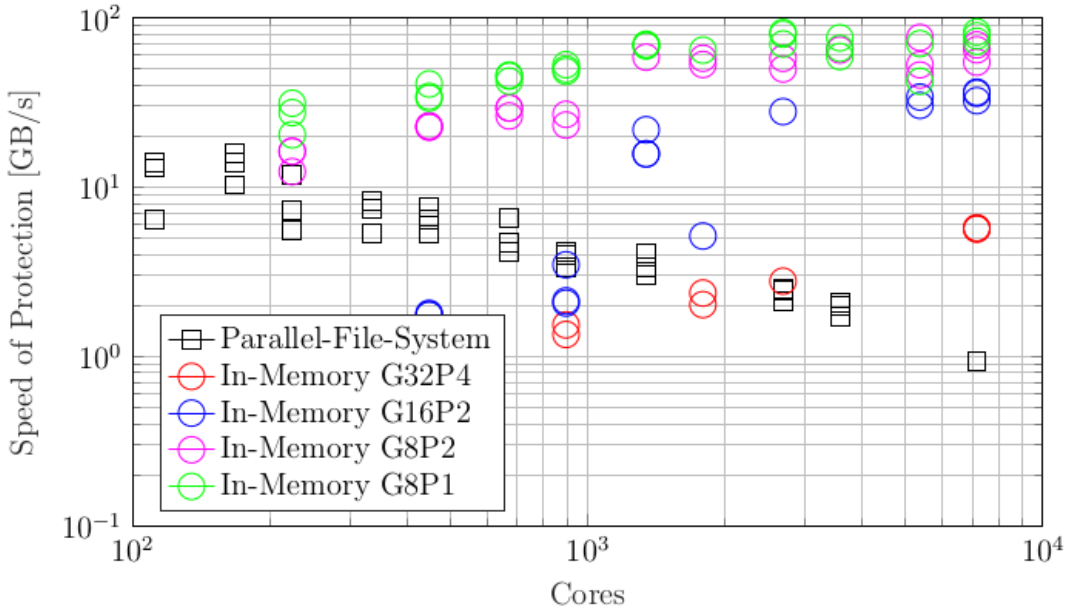


(a) Weak Scaling – 1.5MB per Array per Core

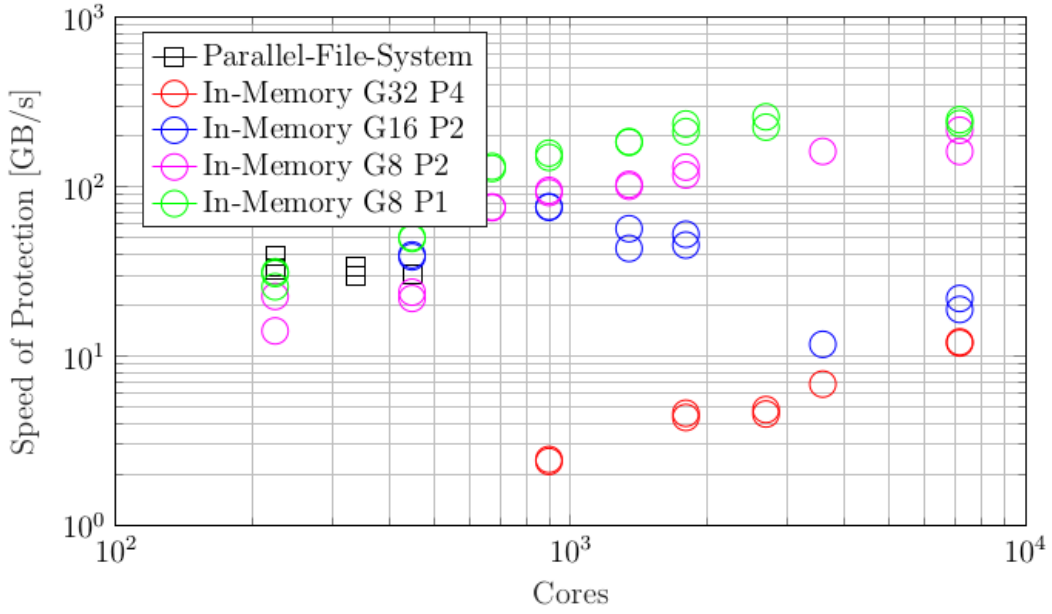


(b) Weak Scaling – 12MB per Array per Core

Figure 31: Weak scaling test on the cost of updating a checkpoint as measured on the EPFL Fidis cluster. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made three times to illustrate the variance in timings due to job-placement and network load when using a shared cluster. (a) 1.5MB per core. (b) 12MB per core. With as little as 1.5MB per core, good scaling is observed for the in-memory checkpoints.



(a) Strong Scaling – 16GB Distributed Array



(b) Strong Scaling – 128GB Distributed Array

Figure 32: Strong scaling test on the cost of updating a checkpoint as measured on the EPFL Fidis cluster. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made three times in order to illustrate the variance in timings due to job-placement and network load when using a shared cluster. (a) 12GB distributed array. (b) 128GB distributed array. Encoding four parity blocks over a large group containing 32 nodes is up to an order of magnitude faster than writing the data to protect to the parallel-file-system.

2.6.5.5 Cost of Data Recovery

If a status-check, using *CheckStatus* during the operation reveals that one or more ranks has been lost and replaced, the arrays at a previous *checkpoint_idx* may be recovered by the Llama Guard using the *RecoverArray* method of the Guard class. To identify the array to be recovered, one must use the key associated with the array, when the array was added to the Guard's list of arrays to protect using the method *ProtectArray* of the Guard class. The dimensions of the array, as well as a

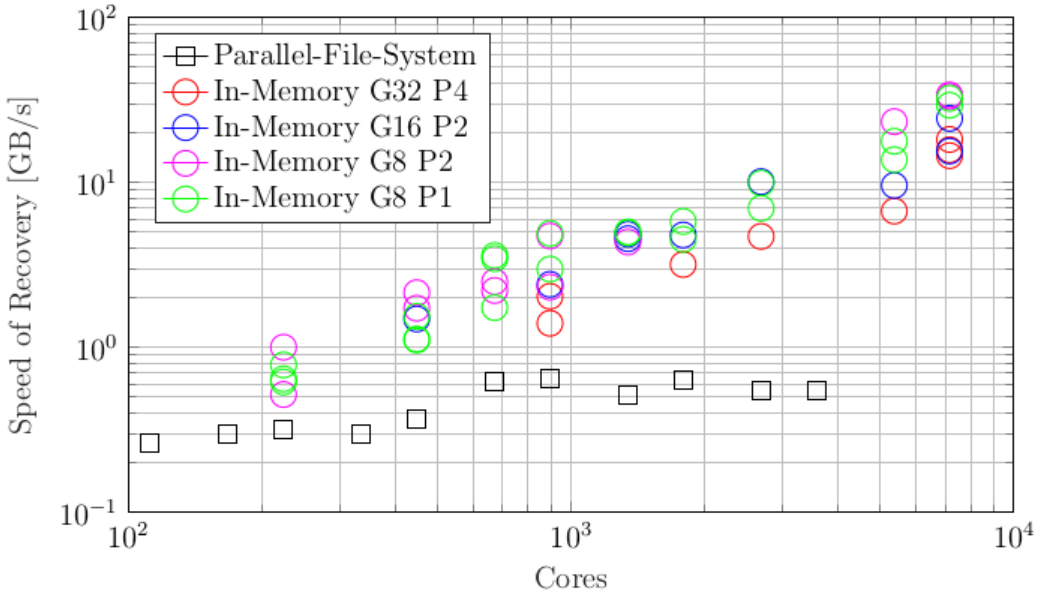
pointer *p_array* to where the newly recovered data from a previous checkpoint idx should be recovered, are part of the input arguments. This pointer can be a pointer to the same location as used when updating checkpoints associated with the Guard. The method is templated on the data type of the array to be recovered, and is overloaded by the number of arguments to distinguish between 1D, 2D, and 3D arrays.

```
// Initiate recovery of 1D array
template<class T> void RecoverArray(int checkpoint_idx, const std::s
tring &key, T *p_array, int dim1);
// Initiate recovery of 2D array
template<class T> void RecoverArray(int checkpoint_idx, const std::s
tring &key, T **p_array, int dim1, int dim2);
// Initiate recovery of 3D array
template<class T> void RecoverArray(int checkpoint_idx, const std::s
tring &key, T ***p_array, int dim1, int dim2, int dim3);
void llama::checkpoint::disk::Update(int new_checkpoint_idx)
```

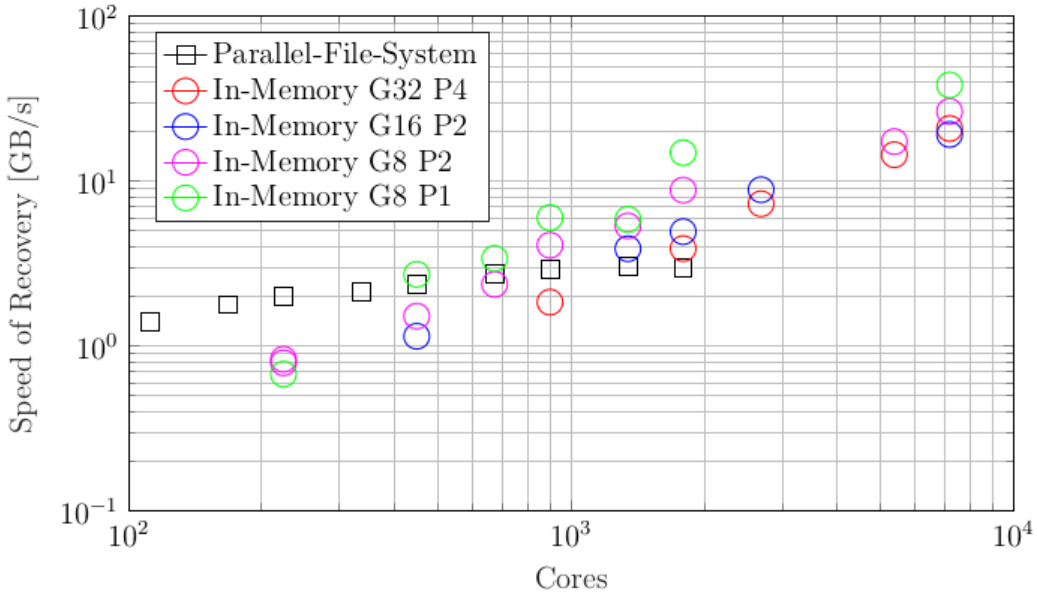
When a Guard is asked to recover the content of an array at some previous index, it first performs an agreement function, ensuring that all workers have been asked to recover the same array. This is followed by a few basic checks that the array exists and that the information on key, array data type, and dimensions is consistent with an array being protected. The Guard then cycles over all checkpoints type associated with it, and queries the checkpoint to see if it is able to recover all data at the checkpoint index specified. Among the candidates, it chooses the checkpoint which it reckons is the fastest to recover from. The chosen checkpoint is then instructed by the Guard to recover the data to the location of the pointer supplied. How recovery is performed depends on the checkpoint type. To-disk checkpoints will use MPI-IO to read from the parallel-file-system, while in-memory checksum checkpoints will perform a group-local decoding procedure.

Figure 33 contains weak scaling numerical experiment, demonstrating the rate at which a 2D array may be recovered using the different types of checkpoints. Tested for 112 cores to 7168 cores on the EPFL Fidis cluster. Two scaling tests are demonstrated, one with 1.5MB per core and another with 12MB per core. Each data point constitutes a job on the shared general-purpose cluster. Each experiment has been performed three times since job placement and cluster network load may substantially impact the speed of recovery. Figure 34 contains strong scaling measurements using a fixed global array size of 16GB in Figure 34(a) and 128GB in Figure 34(b). In all tests, a single rank in a single group is terminated by raising a SIGKILL locally on the rank to disappear. For recovery from in-memory checkpoints, all groups containing a broken rank must decode in parallel across the group to recover the lost data. All unbroken groups may simply copy the unbroken content of the local array. The time-to-recover is therefore limited by the time it takes the broken groups to decode all local data blocks. The location of ranks within a group on the cluster and the network may therefore substantially impact the speed of recovery. For to-disk checkpoints, all cores must read the array from a file on the parallel-file-system, regardless of whether or not they were involved in a failure or not. This makes the to-disk checkpoint time-to-recovery less sensitive to how ranks are distributed on the cluster. Instead the

speed of recovery is sensitive to the load on the parallel-file-system at the time of the job execution.

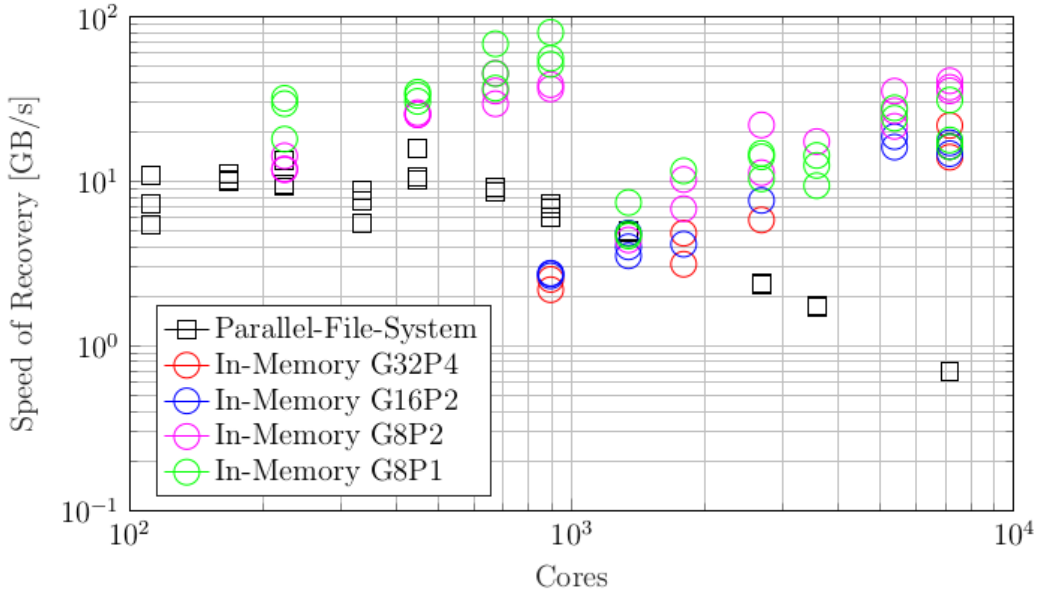


(a) Weak Scaling – 1.5MB per Array per Core

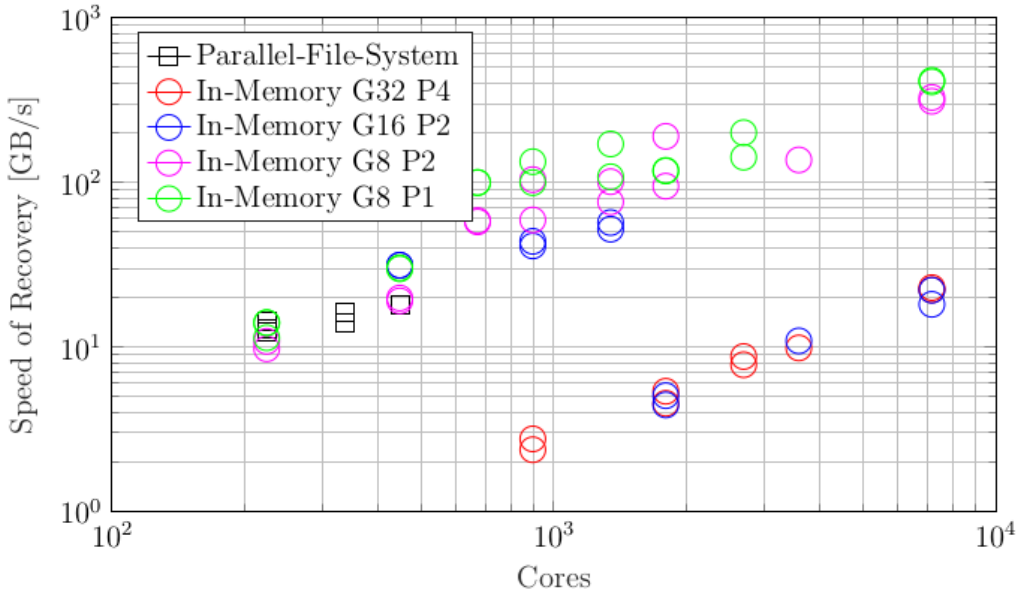


(b) Weak Scaling – 12MB per Array per Core

Figure 33: Weak scaling test on the cost of recovering data from a checkpoint as measured on the EPFL Fidis cluster. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made three times in order to illustrate the variance in timings due to job-placement and network load when using a shared cluster. (a) 1.5MB per core. (b) 12MB per core. With as little as 1.5MB per core, good scaling is observed for the in-memory checkpoints.



(a) Strong Scaling – 16GB Distributed Array



(b) Strong Scaling – 128GB Distributed Array

Figure 34: Strong scaling test on the cost of recovering data from a checkpoint as measured on the EPFL Fidis cluster. In the legend, G indicate group size and P indicate number of parity code blocks per stripe. Each measurement was made three times in order to illustrate the variance in timings due to job-placement and network load when using a shared cluster. (a) 12GB distributed array. (b) 128GB distributed array. Encoding 4 parity blocks over a large group containing 32 nodes is up to an order of magnitude faster than writing the data to protect to the parallel-file-system.

The recovery procedure always proceeds after a status check where one or more worker ranks are found to have failed. The status check is a method of the Llama Guard class. When worker ranks enter the *CheckStatus* method, an agreement operation is performed across all workers to check if any workers have failed. If one or more worker ranks are found to have failed, spare ranks are released and the worker communicator revoked and repaired. During the repair, spare ranks are injected to take the place of lost worker ranks. The process of detecting failures and repairing the worker communicator is essential for the automatic restart at

an earlier checkpoint index. The method is implemented using functionality of ULFM-MPI. Unfortunately we've found that when testing the Llama library using a large number of cores on the EPFL Fidis cluster, the detection of failed ranks and subsequent communicator repair sometimes experience either a deadlocks or fail completely. Following the simulation of rank failures by raising SIGKILL on a rank to terminate it, deadlock or failure of otherwise failure-free ranks often happen within *MPIX_Agree* or *MPIX_Comm_Replace*. The errors returned vary in type. Sometimes failure-unaffiliated ranks lose contact with the MPI helper daemon orted and exit, other times errors related to the operations on the InfiniBand network are returned. As described in section 2.6.1.4, ULFM-MPI is still at an early phase of development and is not part of the official MPI standard yet. According to the developers of ULFM-MPI, the possibility of buggy corner cases is high.

In our current implementation using ULFM-MPI 2.0, correct detection of failures and subsequent repair of the worker communicator happen only about half the time when running large jobs with 1000+ cores. When the detection and repair proceeds correctly, the time to repair ranges from less than one second to 80+ seconds. Due to the inconsistent behavior of the detection and communicator-repair operations, we're unable at the moment to perform meaningful large-scale scaling tests on the cost of this component of Llama.

2.6.6 Summary

The Llama library delivers fault tolerance in HPC trough checkpointing. It is the first library to support both automatic rollback without restart and the use of an arbitrary number of checkpoint levels with topology aware checksum checkpoints of arbitrary group size and number of parity code blocks. In section 2.6.5, it was demonstrated that the library can achieve in excess of 300GB/s encoding when running on 256 nodes, similar to what has been demonstrated using the SCR library by LLNL [153], with the major distinction that there are no limitation in Llama on the group size and the number of parities to use per group.

One issue not touched upon in outlining the recovery procedure is that for any application, newly joined worker ranks must create all structures that was created by the original worker ranks during initialization. This initialization phase may involve communication with other worker ranks and could potentially be very time consuming. A potential path around this issue was presented in [102] where it was demonstrated that this procedure need not be slow nor cumbersome to implement. By logging all MPI transactions during the initialization phase and protecting these, new workers can recover in complete isolation from other workers waiting to restart at the previous checkpoint.

The library is still under development and there are several points that needs to be addressed in future versions. At the moment, only the use of spare nodes to replace failed workers has been implemented, i.e., there is no support for spawning new ranks as needed. Another issue that remains is that upon injecting a large number of errors, the failure-detection and communicator repair may fail. Identifying the underlying cause of these issues is ongoing work.

Finally, the implementation of error handling within Llama functionality is not complete, so at the moment if worker ranks fail within Llama calls such as the update or recovery methods of the Guard, this will result in undefined behavior. To protect against failures during the process of updating in-memory checkpoints, the approach presented in [177] is to be used, maintaining an extra copy of the parity code blocks during checkpoint update.

3 Efficiency Improvements and infrastructure

This section shows the efficiency improvements made to the co-design applications which are not direct implementations of algorithms from WP1 but improvements to the codes in terms of performance.

3.1 IO Improvements

We measured the IO performance achieved by the Nektar++ [1] code [22] using three separate IO methods, XML, HDF5 (v1.8.14) and SIONlib (v1.6.2). HDF5 [24] is a hierarchical data format that allows parallel file access. It is necessary however to record explicitly which parts of a HDF5 dataset belong to which MPI process. The SIONlib library [25] on the other hand, can record such details automatically, removing the burden of having to manage file decomposition within the application code.

Our study involved the checkpoint files produced by two test cases. One we describe as small since it produces checkpoint files that contain a moderate amount of data, approximately 2.5 MB, derived from a mesh containing 150,302 elements. The other uses a much more detailed mesh, featuring 3.5 million elements, that generates a dataset roughly 5.5 GB in size. The first test case simulates the flow of blood through an aortic arch [26] using an advection-diffusion-reaction solver to simulate mass transport, whereas the second case employs an incompressible Navier-Stokes solver to model the flow of air around a racing car. We first executed both test cases for a range of node counts, 2^n, where n is in the range $5 - 8$, on the ARCHER platform [27], a Cray XC-30 machine: each node has 24 cores. This enabled us to generate the various checkpoint files that could then be used by a specially written IO benchmarker.

A performance measurement tool `FieldIOBenchmarker` [23] was written to better understand the performance costs associated with reading and writing checkpoint field files - it uses the Nektar++ `FieldIO` class, which was subclassed for each IO method discussed in this section. For example, the `FieldIOXml` class reads and writes data through the use of two routines called Import and Write. Each MPI process reads and writes to a unique checkpoint file, which stores the field definitions and field data that are handled by that process.

The `FieldIOHdf5` class supports the reading and writing of a single HDF5 checkpoint file that is accessed by all MPI processes. This file features a top-level groupcalled NEKTAR, which contains a sub-folder for every field type. The values of the parameters that define a field type are hashed so as to generate a unique sub-folder name — the parameter values are then added as attributes of the field type sub-folder. All other data are stored within several datasets that reside within the top-level folder. For example, every field that an MPI process might handle consists of elements, where each element has an ID and is also associated with a set of values: hence, there are datasets for element data and element IDs. The most important dataset however concerns the decomposition; this dataset allows every

MPI process to identify (via hash values) those field types it should be handling an[1]d secondly, to determine the correct offset for accessing the element datasets.

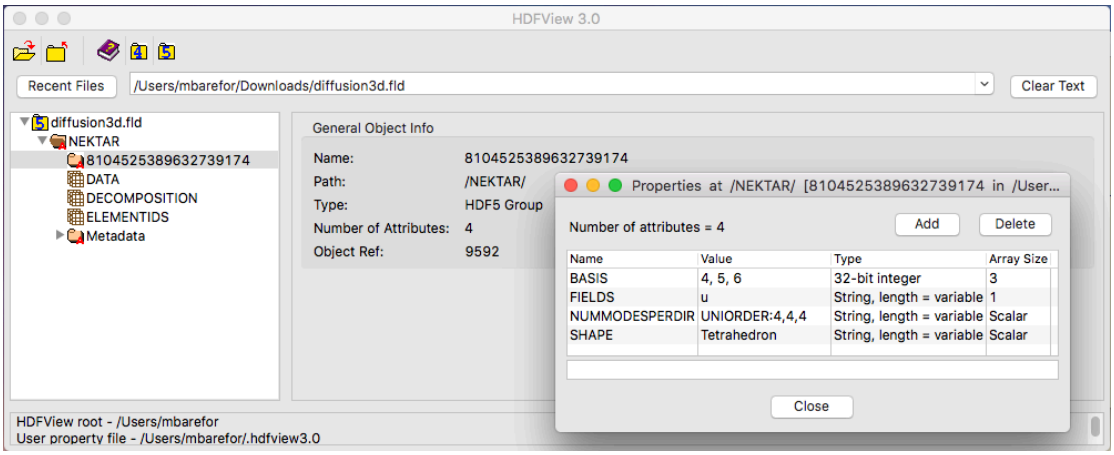


Figure 35: HDF5 checkpoint file format (Aortic Arch test case).

The job of creating the structure of the HDF5 file is delegated to a single *root* process that also writes the decomposition dataset having collected all the necessary metadata from the other ranks. All MPI processes then write their data to the appropriate locations within the element datasets. During the importing of HDF5 data, each MPI process uses the decomposition dataset to avoid reading parts of the element datasets that belong to other processes. The design of the `FieldIOHdf5::Import` routine incorporates a degree of flexibility that allows the user to redefine the mapping between processes and elements. This flexibility incurs a cost in the form of additional reads: specifically, the entire decomposition dataset has to be communicated to all processes.

Finally, the `FieldIOSIONlib` class does not require decompositional data to be recorded explicitly. The SIONlib library itself records which MPI processes have written which data to the checkpoint file: this permits each process to simply loop over the fields it handles, writing out or reading in the field definition, element IDs and element data in turn. It is also possible for an MPI process to impersonate another rank when opening a SIONlib file, permitting element redistribution.

3.1.1 Results

Sets of scaling runs were performed for each IO method, where each run recorded the time taken to read and write the data. Both IO operations were performed ten times for each core count, allowing an average value for execution time to be plotted. The `FieldIOBenchmarker` tool takes a parameter that specifies how many tests are to be performed. Setting this parameter to ten can lead to caching effects that result in fast file access times; to counter this unrealistic circumstance, the count parameter was set to one and `FieldIOBenchmarker` was instead called ten times from within the submission script. Further, the submission script was designed such that each iteration exercised all three IO methods. Having each IO method tested from *separate* scripts running at different times could make

[1] The variant of Nektar++ discussed in this report is currently maintained within the `ioext` branch stored within the GitLab Nektar++ repository [30]. The `ioext` branch was formed from Nektar++ the trunk as of 28 May 2018.

comparisons difficult since the overall file system load is expected to be varying continuously. All of this performance data, containing 240 time measurements (4 core counts × 3 IO methods × 2 IO operations × 10 tests), was then bundled into a single dataset. Similarly formatted datasets were then compiled on subsequent days in order to sample how Nektar++ IO performance could be affected by the different file system loads routinely experienced by ARCHER. Please note, the read and write times presented here are taken from the perspective of Nektar++, and therefore do not just cover the low-level IO operations, but also the necessary housekeeping required to initialise the data structures for all MPI ranks.

The Lustre file system on ARCHER controls access to a set of 48 object storage targets (OSTs) and has a theoretical peak performance of 30 GB/s [28]. By default, each file is split into 1 MiB stripes and then stored on one 1 OST. This is fine for codes using one file per process; whereas a single shared checkpoint file should in general have its stripe count set to -1, allowing use of all available OSTs. However, we found that when file sizes are sufficiently small the best IO performance is achieved with the default Lustre settings.

HDF5 nominates a set of MPI ranks to act as data aggregators: those processes collect the data from all the other ranks before writing to file. The number of HDF5 file writers was 48 for the core counts tested. SIONlib also funnels data to designated writers (or collectors) assuming one has opened the file in collective mode; although, we found that the number of SIONlib writers was fixed at the lower value of 32. Furthermore, using a *collective* SIONlib configuration would not be sufficient to yield the best performance, since each writer actually performs many writes instead of just one. As with HDF5, a SIONlib writer gathers data from other MPI ranks, but then it writes each rank's data to the part of the checkpoint file reserved for that rank. Thus, for high core counts, each SIONlib writer will perform thousands of writes across different locations. The solution is to add the `collectivemerge` tag to the mode string that is used to open the SIONlib checkpoint file. This forces each writer to output the collected data to one part of the file only; the part reserved for that writer's rank.

3.1.2 Results of Aortic Arch

Figures 3.1.2.1-2 present the results generated by the small checkpoint files. It shows that SIONlib achieved the fastest read/write times for all core counts, but, although the majority of the data points are close to the averages (enlarged data points) there are many outlying (i.e., slower) readings also. For example, the SIONlib results recorded for a core count of 6144 contain two *extreme* outliers that are pointed to by the vertical chevrons in Figure 36. A data point is judged to be an outlier if it is beyond a certain distance from the top of the third quartile. This distance is defined as twenty times the inter-quartile range (IQR) for the read data and 12 IQR for the write data. The two SIONlib write outliers are so extreme that the mean SIONlib write time for 6144 cores is in fact greater than the XML and HDF5 averages; this is despite the fact that the fastest writes were recorded by SIONlib. If we adjust the average read/write times by discounting these extreme values, we see that SIONlib is the best performing IO method, as shown in the line plots of Figure 36 to Figure 39.

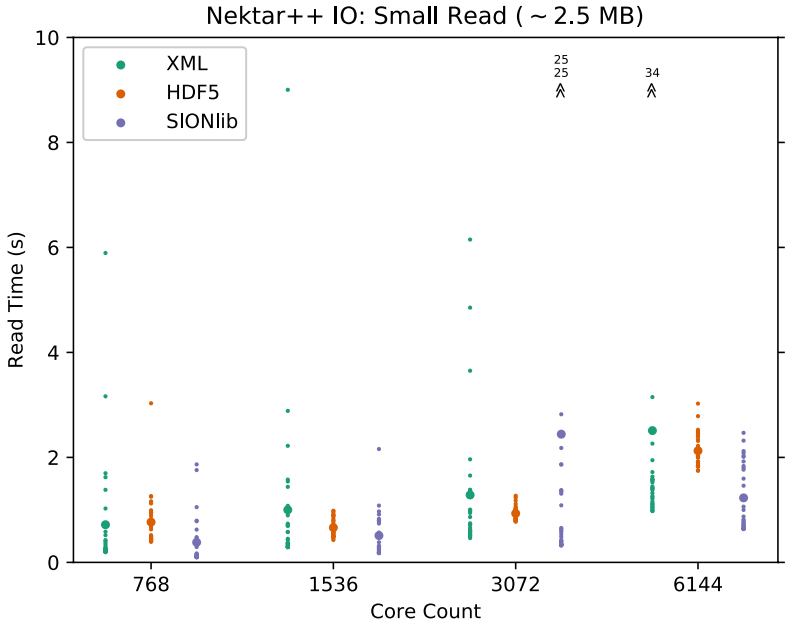


Figure 36: Read times for the small (2.5 MB) Nektar++ checkpoint files, presented as a scatter plot. The data are drawn from three datasets (720 timing measurements in total) compiled on separate days. Some of the datasets contained extreme outliers which are indicated by vertical chevron pairs with the actual outlier values printed immediately above.

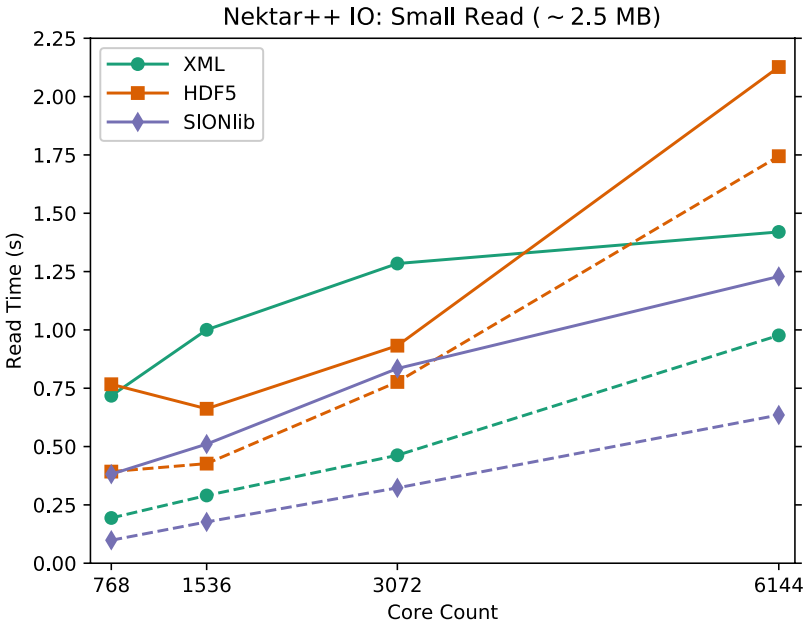


Figure 37: Read times for the small (2.5 MB) Nektar++ checkpoint files. The data are drawn from three datasets (720 timing measurements in total) compiled on separate days. Both the mean (solid) and minimum (dashed) times are shown. Note, the mean calculations exclude any outliers shown above.

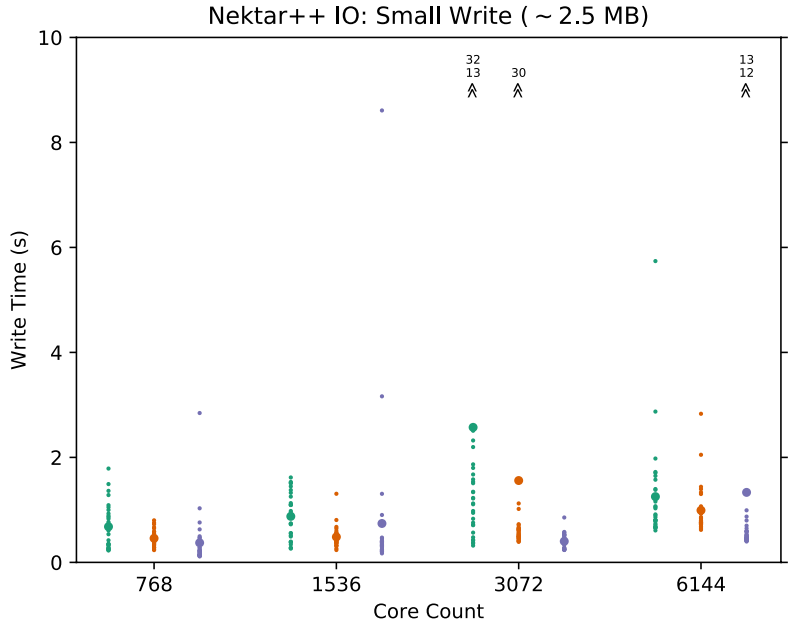


Figure 38: Write times for the small (2.5 MB) Nektar++ checkpoint files. The format follows that used for Figure 30.

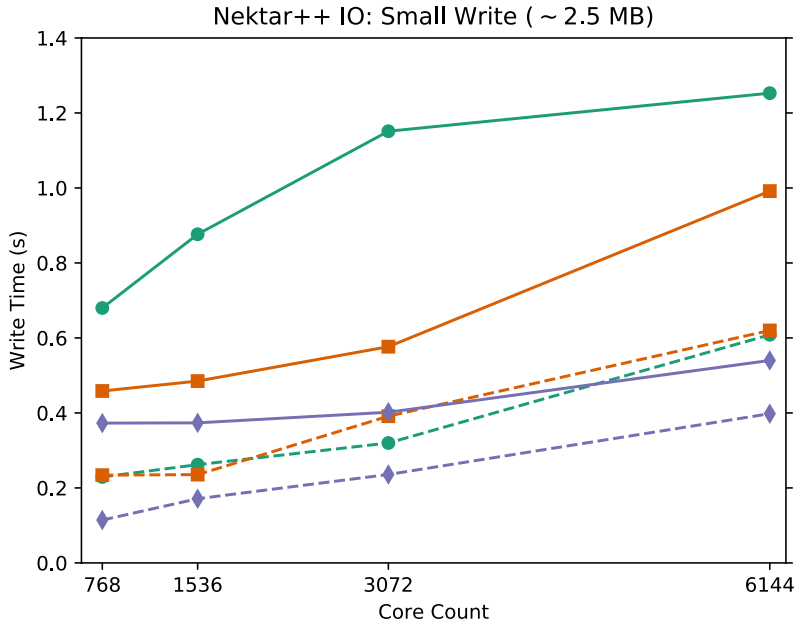


Figure 39: Write times for the small (2.5 MB) Nektar++ checkpoint files. The format follows that used for Figure 37.

3.1.3. Results of Racing Car

We repeated the measurements discussed above but this time we used a more industrial test case, one that produces checkpoint files approximately two thousand times larger than those generated by the aorta mesh.

We also investigated the impact of changing the Lustre settings for the checkpoint files. Just to recap, the default settings (stripe count/size = 1/1MiB) were used for

the small checkpoint files regardless of IO method. We then changed the Lustre stripe count to -1 for the single-shared file IO methods (HDF5 and SIONlib) for the large test case, enabling a checkpoint file to be striped across all available OSTs. As noted previously, the actual writing and reading of data, is handled by a (relatively) small group of processes: 48 in the case of HDF5 and 32 for SIONlib — all other processes simply send their data to the pool of designated writers. It seemed sensible then to enlarge the Lustre stripe size for the large checkpoint files to a value roughly equivalent to the amount of data handled by each writer, which turns out to be 256 MiB for SIONlib and 128 MiB for HDF5. In effect, each HDF5/SIONlib writer could now write all of their data within one stripe, allowing each writer to be assigned to one unique OST.

For the large test case, the SIONlib library continues to lead when it comes to reading data, but it is now the second fastest of the three IO methods for writing data: although, SIONlib manages to beat HDF5 at 6144 cores it is 1.5 times slower than the XML average. In fact, XML is now the fastest writer for all core counts, see Figure 43.

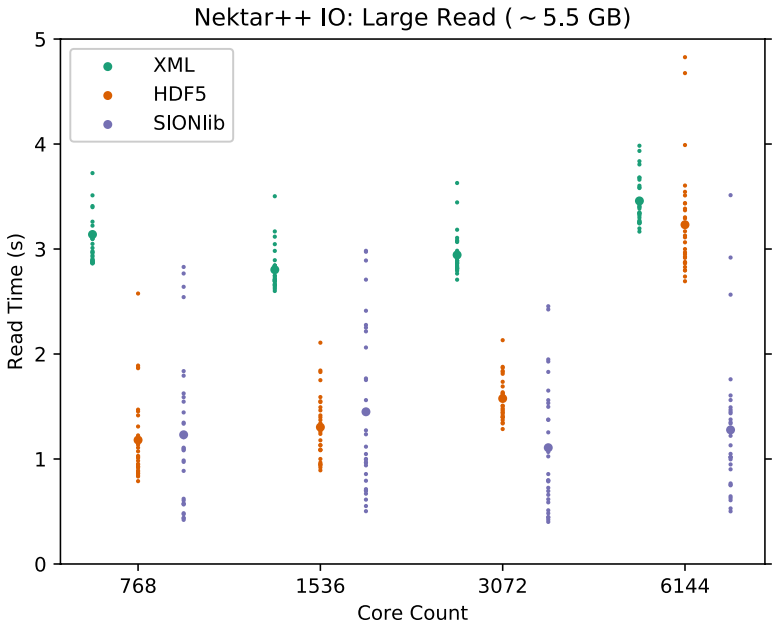


Figure 40: Read times for the small (5.5 GB) Nektar++ checkpoint files. The format follows that used for Figure 30.

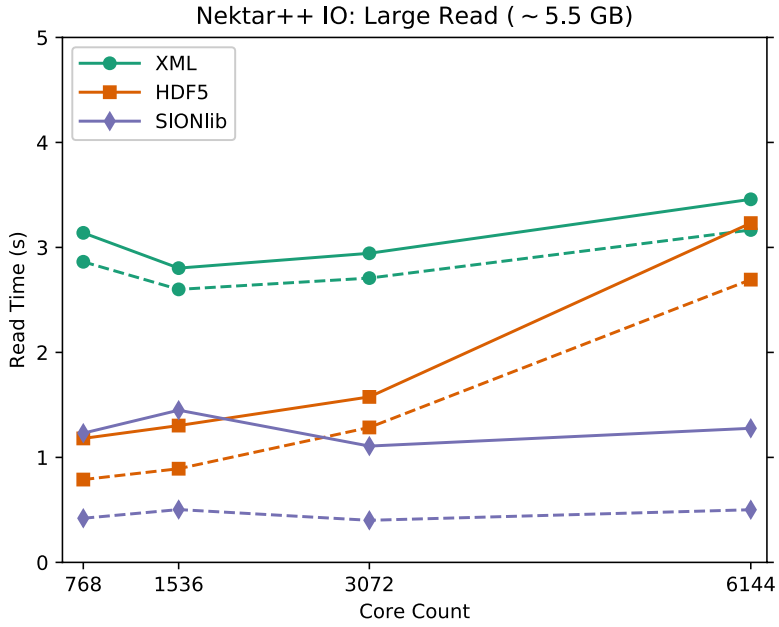


Figure 41: Read times for the small (5.5 GB) Nektar++ checkpoint files. The format follows that used for Figure 37. Both the mean (solid) and minimum (dashed) times are shown.

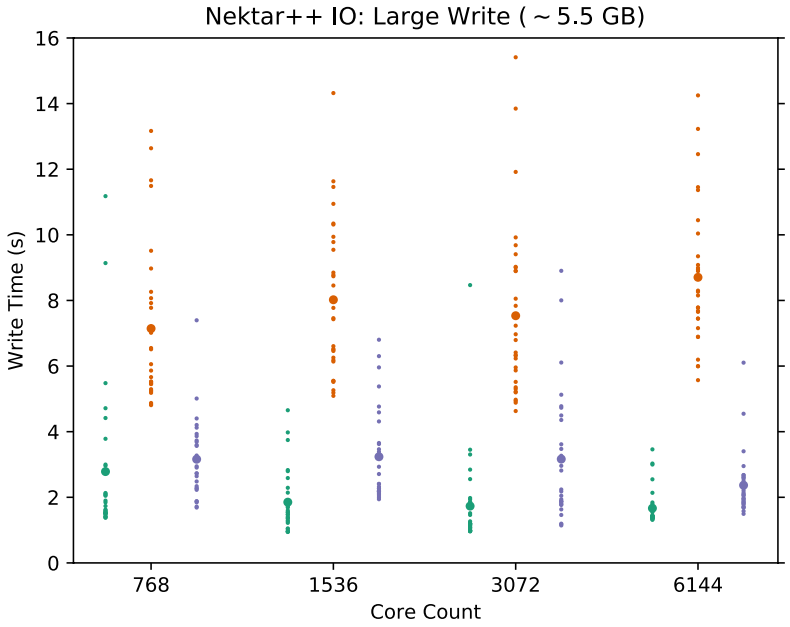


Figure 42: Write times for the small (5.5 GB) Nektar++ checkpoint files. The format follows that used for Figure 30.

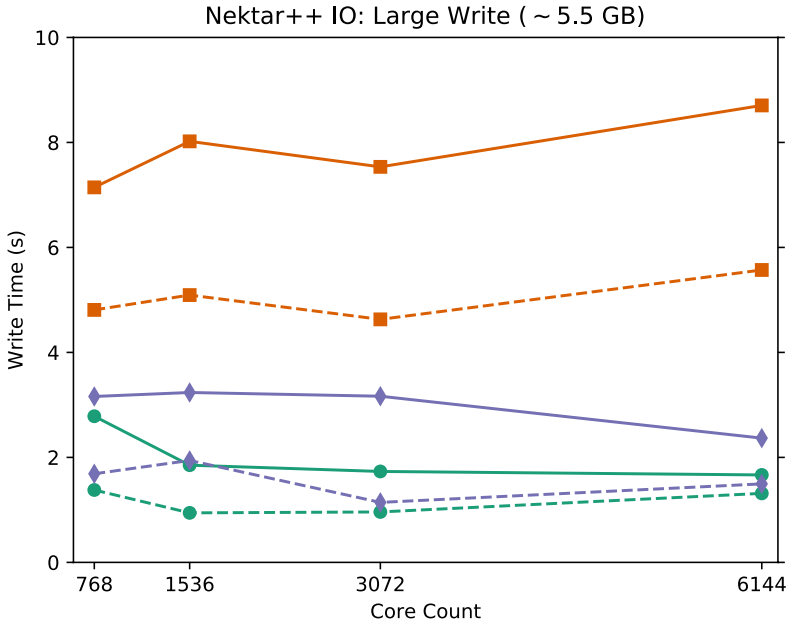


Figure 43: Write times for the small (5.5 GB) Nektar++ checkpoint files. The format follows that used for Figure 37.

3.1.4. Initial Summary

The `FieldIOBenchmarker` tool has provided sets of results that show the read and write times for Nektar++ checkpoint files using three IO methods, XML, HDF5 and SIONlib. The first of these methods (XML) maintains one checkpoint file per MPI process, whereas the other two use a single shared checkpoint file.

We find that the relative performance of the IO methods does depend on the size of the checkpoint file. The results for the small 2.5MB checkpoint file exhibited significant scatter: it was necessary to annotate the corresponding plots to indicate the presence of extreme outliers (Figure 40 to Figure 43). These extreme values (8 out of 720) were recorded for all IO methods; hence, we felt justified in recalculating the mean performance with the outliers removed. SIONlib achieved the fastest IO times for all core counts. XML was the slowest at writing (2.3 times the SIONlib result), but HDF5 showed the poorest read performance (1.7 times slower than SIONlib).

We should also mention that setting the stripe count to -1 for the single-shared file IO methods worsened the IO speeds for HDF5 and SIONlib, indicating, perhaps, that we had reached the core count limit for the partitioning of the aortic arch mesh [26]. This motivated further IO benchmarking but this time using *5.5GB* checkpoint files, which were sufficiently large to justify using a stripe count of -1, permitting the checkpoint data to be striped across all available OSTs. (Another benefit of handling greater data volumes was the disappearance of extreme outliers, which had complicated the aorta results). We also increased the stripe size for SIONlib and HDF5 to 256MiB and 128MiB. The intention here was to

create the possibility for each SIONlib/HDF5 aggregator to read/write their data to a dedicated OST using a single stripe.

The second set of results, Figure 40 to Figure 43, show that the SIONlib class is still able to read the checkpoint data in the fastest time. The HDF5 read performance is slower but not as slow as XML, despite the fact that there is a noticeable rise in HDF5 read times with core count. The HDF5 write performance has worsened considerably for the large test case, whereas XML has moved from third position to first.

3.1.5. Improving the HDF5 Checkpoint File Format

The HDF5 results shown above have revealed the cost of having every MPI process read in the entire decomposition dataset. It should be possible to redesign the format of the checkpoint file such that each process need only read the data pertinent to the mesh elements that it has been assigned.

Furthermore, the need to collate the decompositional data explicitly lengthens the execution time of the `FieldIOHdf5::Write` routine. This cost could easily be avoided if the mesh partitioning does not change during the simulation. Instead, the HDF5 write routine would simply have each MPI process determine the correct offset into the element dataset before updating the data contained therein.

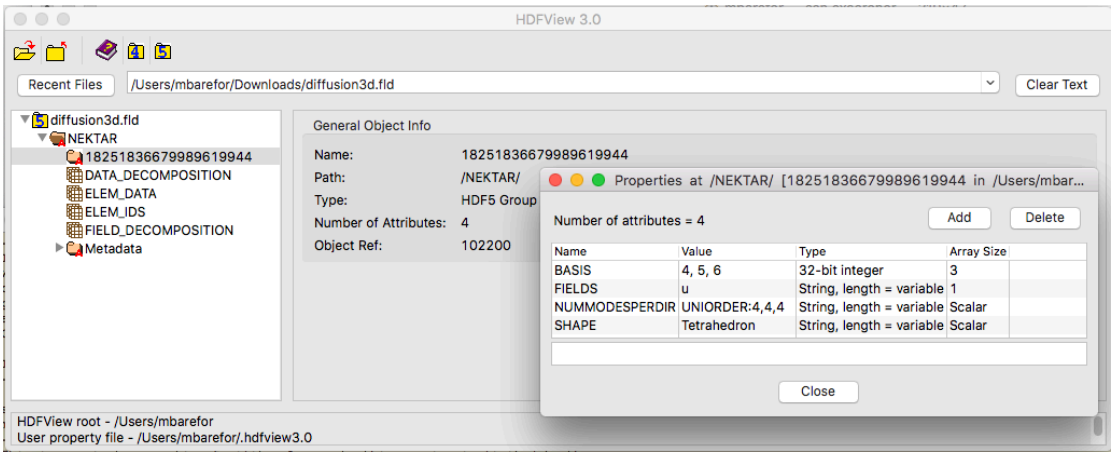


Figure 44: Revised HDF5 checkpoint file format (Aortic Arch test case).

The revised checkpoint file format now features two decomposition datasets, called `FIELD_DECOMPOSITION` and `DATA_DECOMPOSITION`. The first dataset allows each process to discover how many fields it is handling by only reading the part (just 128 bytes) of `FIELD_DECOMPOSITION` that is relevant to the process. The data obtained tells the process which offset to use when accessing the `DATA_DECOMPOSITION` dataset; the data read from this second dataset then tells the process where to access the `ELEM_IDS` and `ELEM_DATA` datasets.

Basically, this arrangement avoids having to have each process read in entire datasets. This is also useful when writing a checkpoint file: the HDF5 write method now uses a flag called reformatting that if set to false tells the code not to go through the process of collecting the decompositional data: those datasets should only be updated if the partitioning has changed also. And so, an MPI process will only need to update the `ELEM_DATA` dataset.

Figure 45 to Figure 48 presents the results from the improved `FieldIOHdf5` class. Please note, the tests for the other IO methods must also be run at the same time in order to account for the continually varying file system load. For the small test case we can see that the HDF5 performance for both read and write has improved and is now much less than the XML result. Moving to the large test case (Figure 3.1.5.3), we see that again the HDF5 read time is lower than the XML result; disappointingly however, the HDF5 write result has barely improved even when we restrict the writing to `ELEM_DATA` dataset (i.e., the reformatting flag is set to false).

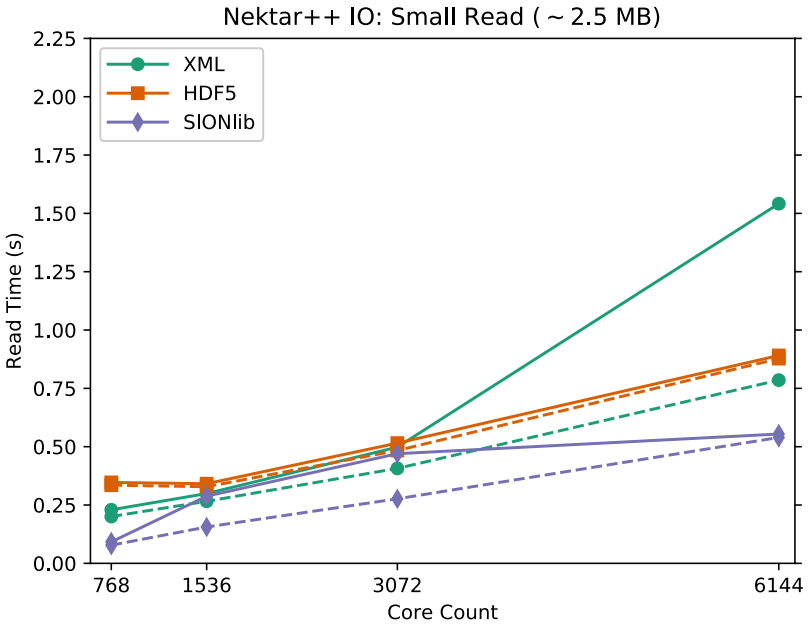


Figure 45: Read times for the small (2.5 MB) Nektar++ checkpoint files. The data are drawn from just one dataset (240 timing measurements in total). Both the mean (solid) and minimum (dashed) times are shown.

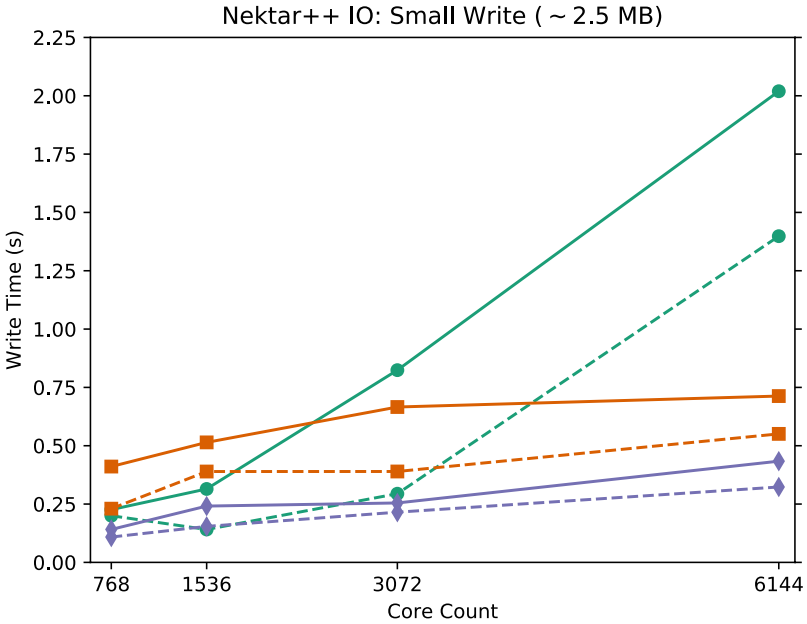


Figure 46: Write times for the small (2.5 MB) Nektar++ checkpoint files. The data are drawn from just one dataset (240 timing measurements in total). Both the mean (solid) and minimum (dashed) times are shown.

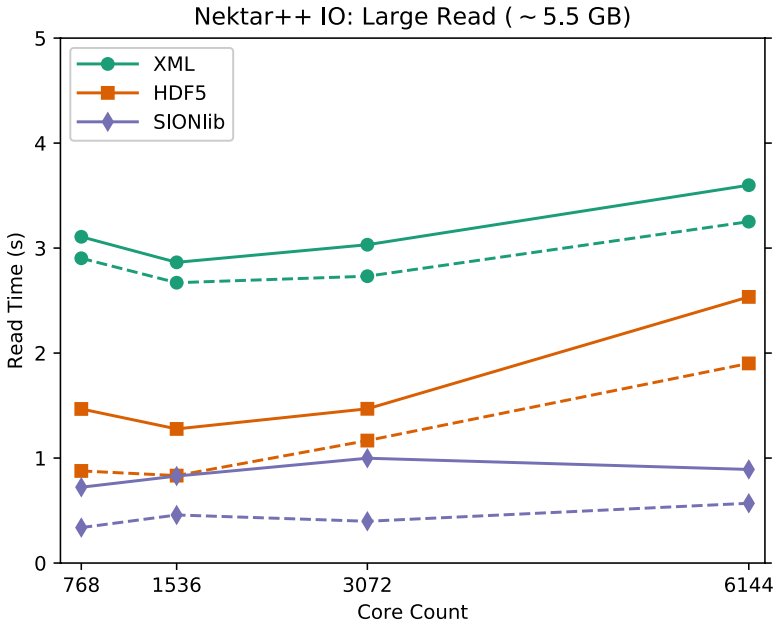


Figure 47: Read times for the large (5.5 GB) Nektar++ checkpoint files. The data are drawn from just one dataset (240 timing measurements in total). Both the mean (solid) and minimum (dashed) times are shown.

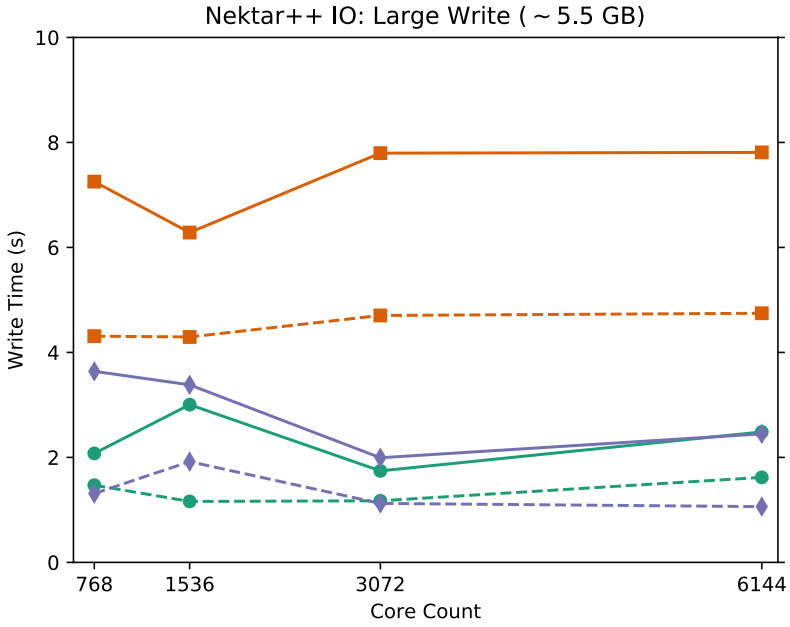


Figure 48: Write times for the large (5.5 GB) Nektar++ checkpoint files. The data are drawn from just one dataset (240 timing measurements in total). Both the mean (solid) and minimum (dashed) times are shown.

These results motivated us to profile the `FieldIOHdf5::Write` routine. We subsequently found that 80% of the time is spent closing the HDF5 file. Specifically, the `Write` routine calls a method called `WriteData` that is private to the `FieldIOHdf5` class; this method uses a C++ smart pointer to access the HDF5 checkpoint file and so when the method exits the smart pointer goes out of scope which activates its destructor causing the connection to the HDF5 file to be closed. The profiling results indicate that it is this HDF5 close operation that is taking most of the time whenever a checkpoint file is written.

The issue of a time-consuming HDF5 close has been noted in connection with the `cray-hdf5-parallel/1.8.14` module available on the ARCHER system. Indeed, the Cray team based at EPCC produced a short report [8] that documented this problem together with a workaround that involves performing a HDF5 flush operation immediately after opening the HDF5 file. Unfortunately, this workaround did not work for our Nektar++ racing car test case. We also tried using the latest `cray-hdf5-parallel` module available on ARCHER, v1.10.1.1: this also failed to reduce the HDF5 write time.

3.2 Communications Libraries

Gather-scatter operations are one of the most important communication kernels used in the spectral element codes Nek5000 and Nektar++ for fetching data dependencies (gather) and spreading back results (scatter). The current implementation in both Nektar++ and Nek5000 is based on the Gather-Scatter library, GS, which utilizes three different communication strategies: pairwise nearest neighbour exchange, message aggregation, and collectives, to efficiently perform communication on a given platform. GS is implemented using non-blocking, two-sided message passing via MPI and the library has proven to scale well to hundreds of thousands of cores.

However, the necessity to match sending and receiving messages in the two-sided communication abstraction can quickly increase latency and synchronization costs for very fine-grained parallelism, in particular for the complex communication patterns created by unstructured CFD problems. To address this issue and prepare both spectral element codes for extreme-scale computing, we have developed ExaGS, a reimplementation of the Gather-Scatter library, with the intent to use the best suitable programming model for a given architecture. Following the promising results from the EPiGRAM project[31], we have chosen to implement ExaGS using the one-sided programming model provided by the Partitioned Global Address Space (PGAS) abstraction, using Unified Parallel C (UPC).

There are two major issues for an efficient UPC implementation. First an efficient shared data layout is needed, which allows for different sized memory blocks on each thread, with easy access from any thread. Most implementations of distributed shared memory models, e.g. Coarray Fortran and UPC, allocate shared memory across threads in equal, fixed block sizes. Although this provides easy access from any thread, unless the problem size is divisible by the block size/number of threads one must allow for padded arrays, creating holes of unused memory. Furthermore, dynamically growing parts of the data will require a global reallocation of the entire shared memory object. To solve this problem, we have used the directory of objects approach, where each thread defines its own

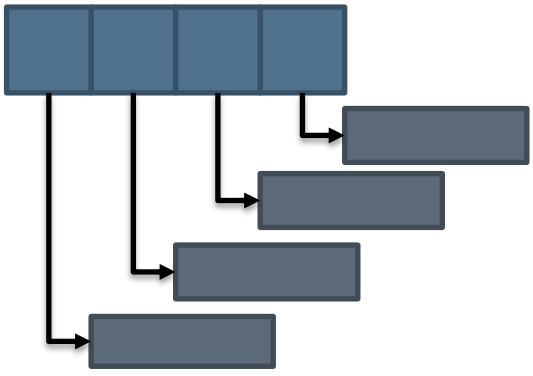


Figure 49: An illustration of the directory of object data structure on four threads.

shared memory region, which can grow and shrink independently of each other. Each region can be accessed by going through a directory, consisting of shared pointers to each region.

The second issue is to derive efficient point-to-point synchronization primitives in UPC, necessary for protecting communication buffers both for message aggregation and nearest-neighbour communication. For this we have investigated different strategies. First, by protecting each transfer with a locking mechanism. A lock is standard and provided natively by UPC, but requires at least two locks to protect both send and receive buffers, thus the communication becomes very similar to the two-sided model, with an explicit synchronization point for both send and receive operations, thus minimizing the gains from using a one-sided model. Our second approach is to synchronise by a shared variable acting as a semaphore. With either a strict access policy or using UPC's atomic update to ensure correctness. With a shared semaphore, a less restrictive communication abstraction can be chosen, where the semaphores implicitly enforce synchronization by protecting the shared memory buffers. This way, the communication pattern becomes very similar to multithreaded programming. However, care must be taken in order to avoid deadlocks, race conditions and network contention when polling or updating atomically the remote memory locations of the semaphores. Despite the slightly difficult implementation, the semaphore approach was chosen for ExaGS due to its more one sided, and lower latency nature.

By utilizing both the shared data structure and the point-to-point synchronisation strategies, we have successfully rewritten the entire Gather-Scatter library in UPC.

3.2.1 Crystal-Router and collective communication algorithms

The Crystal-Router and the collective communication algorithms in the library are all being based on message aggregation, whereas for a fixed number of steps, data is exchanged between a pair of threads in each step, processed and forwarded to another thread in the next step. For a hypercube of dimension *d*, and a *buffer* with data to be forwarded to another thread, the Crystal-Router kernel can be expressed as follows:

```
for i = 0 ... d – 1 do
  Exchange buffer with thread: mythread ⊕ 2^i
  for all received message msg do
    if dest(msg) == mythread then
      Keep msg
    else
      Add msg to buffer
    end if
  end for
end for
```

For a message passing based implementation, the exchange of buffers can simply be implemented by a blocking send and receive operation. In UPC we need to i) signal the receiver that data has arrived in his buffer and ii) make sure that the receiver has processed data in his buffer before any thread tries to write into it. We use an array of semaphores *flg* to achieve this, with the buffers accessible through a directory *dir*. The semaphores are flagged with the previous hypercube

dimension if the buffers are ready to accept data. Once a thread has written his data into the buffer, it changes the status of the semaphore to *-2* marking it as busy, until the receiving thread has processed everything and set it to the current dimension of the cube.

```
flgs[mythread] ← - 1
for i = 0 … d – 1 do
  dst ← mythread ⊕ 2^i
  Block while flgs[dst] ≠ (i – 1)
  dir[dst].buf ← buffer
  flgs[dst] ← -2
  Block while flgs[mythread] ≠ -2
  for all data in dir[mythread].buf do
    if destination(data) == mythread then
      Keep data
    else
      Add data to buffer
    end if
  end for
  flgs[mythread] ← i
end for
```

3.2.2 Pairwise nearest neighbour exchange

The pairwise point-to-point strategy is easier to realize and is often the fastest method on current HPC interconnects. Since the algorithm doesn't depend on message aggregation, a straightforward message passing based implementation can overlap all transfers by first posting a series of non-blocking receives, followed by all the non-blocking sends and later synchronise everything:

```
for i = 0 … n_neighbours do
  Post non-blocking recv(recv_buf[i])
end for
Fill send buffer send_buf
for i = 0 … n_neighbours do
  Post non-blocking send(send_buf[i])
end for
Sync all transfers
```

To implement a UPC version of the point-to-point exchange we used a similar approach as for the Crystal-Router with shared buffers for remote threads to write data into. Since the pairwise exchange doesn't have a fixed number of exchange steps, we needed to extend the shared buffers to a directory of directories of objects structure, such that each directory entry would point to a directory on each thread containing buffer space to receive data from any of its neighbouring threads in parallel. A similar structure was also used for the synchronisation primitives, whereas the sender signals on data movement by updating the

semaphores in the receiver's part of the synchronisation structure. Resulting in a synchronisation scheme where the receiver only needs to poll semaphores with local affinity to the thread, thus reducing traffic on the network. Furthermore, to increase concurrency we used non-blocking remote memory transfers on the sender side, injecting as much data as possible into the network, initiating a transfer as soon as any of the neighbouring threads receive buffers is flagged as ready. With *dir[...][...]* the directory of directories structure, *dir[...].flg[...]* our synchronisation structure and *nb_put* denoting non-blocking memory transfer, the synchronisation scheme for the sending part (similar for the receiver) of the kernel can be expressed as:

```
n ← n_neighbours, sent ← 0
while n > 0 do
  for i = 0... n_neighbours do
    dst ← receiver(i)
    if sent(i) ≠ 1 ∧ dir[dst].flg[mythread] ≠ 1 then
      nb_put(dir[dst][mythread], send_buf[i])
      nb_put(dir[dst].flg[mythread], 1)
      sent(i) ←1, n ← n - 1
    end if
  end for
end while
```

3.2.3 Performance evaluation

We have evaluated the performance of ExaGS on two different systems. First a regular InfiniBand cluster called Tegner (KTH), using the Berkely UPC runtime with the GNU UPC compilers, secondly the Cray XC40 Beskow (KTH) using Cray's own UPC compiler. As a benchmark problem we used Nek5000's own mini-application Nekbone, performing a weak scaling study on solving a linear system using the conjugate gradient method. We used 128 elements per core and ran Nekbone using both the Crystal-Router and the Pairwise communication strategies. For all tests we compared against the reference MPI implementation.

On Tegner we observed up to 10% better performance for our UPC version of the Crystal-Router algorithm as compared to the baseline MPI implementation. While the Pairwise strategy turned out to be slightly slower.

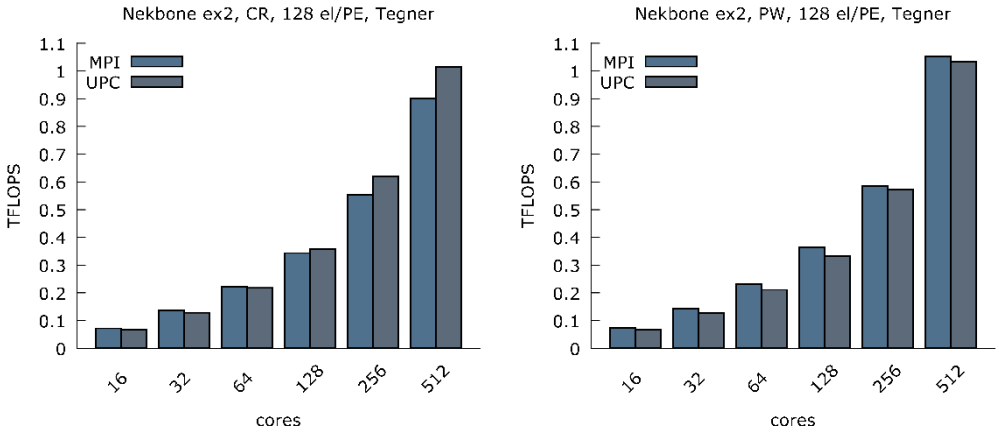


Figure 50: Nekbone performance on Tegner using the Crystal-Router (left) and the Pairwise (right) communication strategies.

At scale on Beskow, the results are different, and we start to see improvements for both communication algorithms when using our new UPC implementation. The Crystal-Router still achieves around 10% better performance for UPC compared to MPI, but on Beskow, the Pairwise strategy is also faster for UPC than MPI with up to 5% better flops rate at scale.

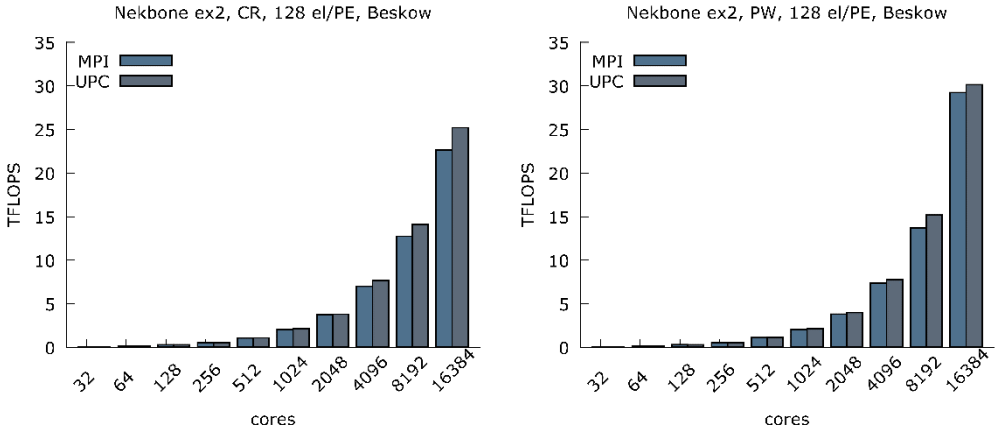


Figure 51: Nekbone performance on Beskow using the Crystal-Router (left) and the Pairwise (right) communication strategies.

3.3 Energy efficiency

3.3.1 Motivation

Subsequent to the results presented in D2.2, section 6.2, the decision was taken to verify those by performing additional measurements, in this case five runs of every setting. This disclosed a high variability when running identical jobs. As this variability is above the order of effects achieved by adapting the clock frequency (Dynamic Voltage and Frequency Scaling, DVFS), at least with respect to the energy delay product (EDP), it renders our prior results unreliable (Figure 52).

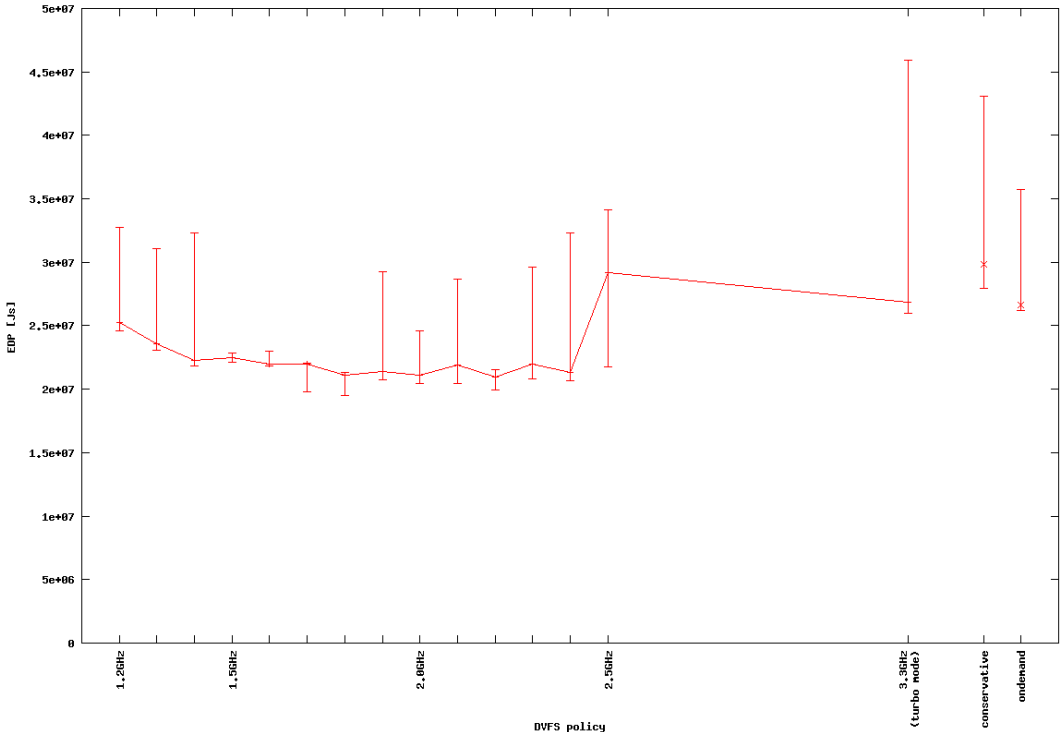


Figure 52: Variability of results presented in D2.2. Shown is EDP of actual computation on 48 nodes with HDF5 IO.

3.3.2 Enlarged use case

To achieve more reliable measurements by way of increasing the length of measured code segments, we supported project partner ASC(S in setting up an enlarged automotive use case - see D3.3 for further details. Using an enlarged use case allows for a higher number of samples per code segment when using a production-like amount of parallelism; this is due to our measurement frequency being fixed. Furthermore, it involves an increased memory footprint so that DRAM bandwidth might become the bottleneck and the code might hence benefit from a reduced clock frequency. The use-case previously examined allows for superlinear scaling, which we surmise is due to hot data fitting into caches. Since there is a strong correlation in between runtime and energy demand, adding parallelism in this case allows for an increase in potential energy saving while heavily increasing the power demand. The latter, however, is very likely to be a problem at the exascale. Moreover, superlinear scaling is unrealistic in typical HPC environments. An enlarged use case may also lower the ratio of communication to computation when deployed on amounts of parallelism that are typical in the

automotive industry. This should reduce the impact of runtime variability due to interconnect issues.

Supporting ASC(S in setting up an enlarged use case, however, required substantial effort since co-design application Nektar++ turned out to be not fully compatible with the automotive industry tool chain deployed by ASC(S. Furthermore, handling such a big use case disclosed some limitations in Nektar++'s own tool chain.

Unfortunately it was not possible to finalize the enlarged use case. Hence, it was decided to perform measurements with the previous use-case, but deploy it with 8 nodes only. The aim is to reduce the amount of communication and the amount of runtime variance. Furthermore, we measured the runtime of the entire time stepping loop instead of further breaking it down into computation, halo exchange and IO phases to improve the number of samples collected.

3.3.3 Measurement setup

The measurement setup hence was as follows:

- An industrially relevant automotive use case provided by ASC(S (cf. D3.4)
- 8 fully populated (i.e. 24 active cores) of Hazel Hen (CRAY XC40, 2 x E5-2680v3 / node, 128GB / node, Aries interconnect)
- 45GB of DRAM used per node
- HDF5 output
- Measured entire time stepping loop
- Git commit 7293b55be0ad7c3026bcab8916f3307f3ece1ed3 of Nektar++
- Built with GCC 5.3.0
- Energy and runtime demand measured by means of a CrayPAT trace experiment with the following parameters:
  - pat_build –w
  - export PAT_RT_PERFCTR=PACKAGE_ENERGY,DRAM_ENERGY,PM_POWER:NODE,PM_ENERGY:NODE
  - export PAT_RT_SUMMARY=1.
- Two identical runs of each setting

3.3.4 Results

The results achieved by means of the measurement setup mentioned above are depicted in Table 4. The values for CPU and DRAM are determined by means of RAPL counters, the node values are determined by measuring voltage and electric current of the node's power rail. For completeness, the results are shown for the entire application as well as the time-stepping loop. However, we restrict ourselves to the latter, since this is dominant in long running production jobs.

The variance appears low with respect to most of the settings which leads to the deduction that the reliability of the results presented here is higher than those of the results presented in D2.2. With respect to the runtime, the computation phase does not benefit from setting the DVFS governor to *performance* instead of *on-demand*. From this, it is possible to deduce that either the computation does not benefit from high CPU clock frequencies and that latencies incurred by switching

to such frequencies *on-demand* are not significant. Fixing the clock frequency to the base clock frequency of the CPU (2.5GHz) introduces a runtime penalty of 1.3%. Reducing this fixed frequency to 2GHz further increases the runtime by another 4%. With the respective energy saving potentials in mind (cf. below), such an increase might however be acceptable to the user, dependent on the urgency of their job.

With respect to energy consumed, we are able distinguish between the demands of the CPU, the DRAM and the entire node. However, DRAM results do not disclose a clear trend. This is probably due to DRAM being driven by another clock domain which is *not* varied here. The below focus is thus on the CPU and entire node, which do disclose the same trends. As seen with respect to runtime, the energy demand of governors *on-demand* and *performance* do not differ that much. Fixing the clock frequency at 2.5GHz allows for a saving of about 11% of energy if compared to governors *on-demand* and *performance*. This may be due to the kernel enabling the energy hungry Turbo Boost feature if temporary load spikes occur. Doing so might not be wise since the high load may disappear quickly but Turbo Boost (where the clock frequency in pushed before the normal maximum for a short period, if the thermal limit is not reached) requires much more power and remains active for a relatively long period. Reducing the fixed clock frequency to 2GHz allows for around 23% less energy demand if compared to governors *on-demand* and *performance*.

With respect to power, there are the same trends as already seen with respect to energy: there is little difference in between the governors *on-demand* and *performance*, a fixed clock frequency of 2.5GHz allows for about 12% less power demand, and 2GHz allows for about 27% less power demand.

3.3.5 Conclusions and Outlook

According to the runtime penalties and energy respectively power saving potentials depicted in section 3.3.4, it is beneficial to reduce the CPU clock frequency of a Nektar++ job if running on a small number of nodes. This might be true in industrial settings with local clusters of limited size, but also if there is a given (even temporary) power budget on exascale systems.

Future work will concentrate on investigating whether the large scale use case of project partner KTH can benefit from fixed and reduced clock frequencies.

DVFS setting	Run	max(Runtime) [s]		sum(Energy) [J]				sum(Power) [W]	
		ent.a.	t. intgr.	entire app.	time integration			ent. a.	t. intgr.
					CPU	DRAM	node		
2 GHz fixed	1	53.499	24.746	90.870.179	32.875.271	22.201.886	54.501.583	1.699	1.907
	2	53.751	24.733	90.998.070	33.320.188	24.613.757	54.849.598	1.693	1.900
2.5 GHz fixed	1	48.573	23.785	102.425.389	39.951.592	23.132.977	62.905.075	2.109	2.311
	2	48.696	23.785	101.036.722	38.188.258	22.912.158	62.149.313	2.075	2.266
governor ondemand	1	46.854	23.498	117.405.385	45.423.535	22.838.926	70.617.678	2.506	2.628
	2	48.325	23.473	121.111.984	45.336.026	22.707.272	70.945.107	2.506	2.623
governor performance	1	47.564	23.411	119.022.698	44.961.944	23.274.614	70.556.363	2.502	2.639
	2	47.596	23.498	119.411.087	44.962.236	20.824.626	70.864.858	2.509	2.607

Table 4: Runtime, energy and power demand of various DVFS settings. Shown are values for entire application and actual time stepping loop. Energy of the latter is further divided into CPU, DRAM and entire node.

3.4 Alternative architectures

Heterogeneous HPC architectures are increasingly prevalent in the Top500 list, with CPU-based nodes enhanced by accelerators or coprocessors optimized for floating-point calculations. This trend will increase moving towards exascale and it is vital that relevant HPC applications are able to exploit heterogeneity. Whilst accelerators offer a large boost in peak system speed, it is hard to translate into sustained application performance. For GPU accelerators, applications are typically rewritten in a bespoke, low-level language such as CUDA.

OpenACC [32] enables existing HPC application codes to run on accelerators with minimal source code changes. This is done using compiler directives and API calls, the compiler being responsible for generating optimized code with the user guiding performance only where necessary.

We have previously presented a serial case study of partially porting to parallel GPU-accelerated system for Nekbone/Nek5000. In this project, we expand our previously developed work and take advantage of the optimization results to port the full version of Nek5000 to multi-GPU accelerated systems, especially regarding the P_n-P_{n-2} algorithm [34]. The algorithm is a way to decouple the momentum from the pressure equations that does not lead to spurious pressure modes, see Figure 53. It is more efficient than other methods, but it involves different approximation spaces for velocity with polynomial order of N and pressure with N-2. The OpenACC version of Nek5000 is available at [33].

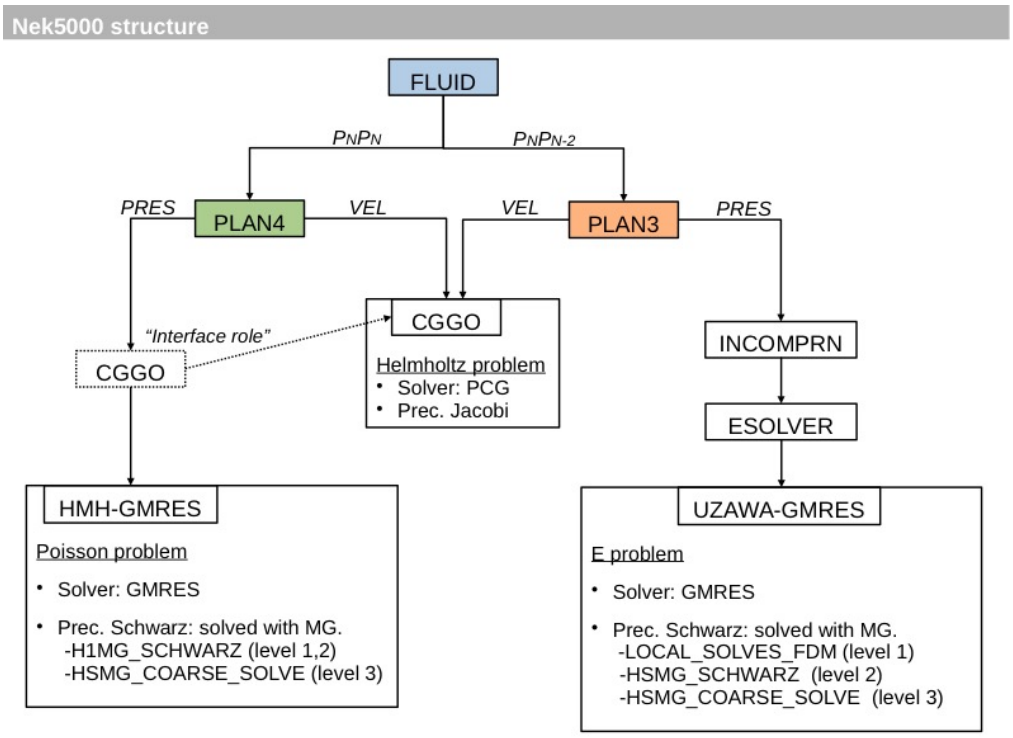


Figure 53: The structure of Nek5000 code for the incompressible flow simulations. The Pn-Pn-2 algorithm is implemented in the subroutine PLAN3.

In this implementation, we focused on the GMRES solver with multi-grid pre-conditioner for the pressure. All subroutines within the GMRES solver have been ported to GPUs via the OpenACC directives, see Figure 54. The three levels of Multigrid solver are employed as the preconditioner. The first two levels (LOCAL_SOLVERS_FDM and HSMG_SCHWARZ) of the multigrid preconditioner have been ported into GPU systems. However, the coarsest (third) level has to run on the CPU; Figure 55 shows the implementation. As a result, the multigrid data on the coarsest level needs to transfer between GPU and CPU, which reduces the parallel efficiency.

Each core exchanges MPI halos with at least 26 others. This is complicated by the separate memory spaces of the CPU and GPU, leading to higher communication overhead. With OpenACC v2.0, MPICH2 allows GPU memory buffers to be passed directly to MPI function calls, eliminating GPU data copy to the host before passing data to MPI. We use the normal *MPI_Isend/MPI_Irecv* functions for internode data transfer directly from/to the GPU memory using OpenACC directives *#pragma acc host data* to enable the GPUDirect communication.

```
do while(iconv.eq.0.and.iter.lt.100)
  if(iter.eq.0) then
    call col3_acc(r,ml,res,ntot2)            ! r = L  res
  else
    !update residual
    call copy_acc(r,res,ntot2)               ! r = res
    call cdabdtp_acc(w,x,h1,h2,h2inv,intype) ! w = A x
    call add2s2_acc(r,w,-1.,ntot2)           ! r = r - w
                                             !      -1
    call col2(r,ml,ntot2)                    ! r = L   r
  endif
  gamma(1) = sqrt(glsc2(r,r,ntot2))          ! gamma  = \/ (r,r)

  !check for convergence
  rnorm = 0.
  if(gamma(1) .eq. 0.) break !converged
  temp = 1./gamma(1)
  call cmult2_acc(v(1,1),r,temp,ntot2)       ! v  = r / gamma

  do j=1,m
    iter = iter+1                            !       -1
    call col3_acc(w,mu,v(1,j),ntot2)         ! w  = U   v
                                             !           j
    call hsmg_solve_acc(z(1,j),w)            ! z  = M   w
    call cdabdtp_acc(w,z(1,j),               ! w = A z
     $            h1,h2,h2inv,intype)       !        j
    call col2_acc(w,ml,ntot2)                ! w = L   w
    do i=1,j
       h(i,j)=vlsc2_acc(w,v(1,i),ntot2)      ! h    = (w,v )
    enddo                                    !  i,j       i
    call gop_acc(h(1,j),wk1,'+  ',j)         ! sum over P procs
    do i=1,j
      call add2s2_acc(w,v(1,i),-h(i,j),ntot2)! w = w - h    v
    enddo                                    !          i,j  i

    alpha = sqrt(glsc2_acc(w,w,ntot2))       ! alpha =  \/ (w,w)
    if(alpha.eq.0.) goto 900  !converged

    rnorm = abs(gamma(j+1))*norm_fac
    if (rnorm .lt. tolpss) goto 900  !converged
    temp = 1./alpha
    call cmult2_acc(v(1,j+1),w,temp,ntot2)   ! v    = w / alpha
                                             !  j+1
  enddo
900 iconv=1
  !sum up Arnoldi vectors
  do i=1,j
    call add2s2_acc(x,z(1,i),c(i),ntot2)     ! x = x + c  z
                                             !          i  i
  enddo
enddo
```

Figure 54: The pressure solver using the GMRES in subroutine ESOLV

```
do l = mg_lmax-1,2,-1
  call hsmg_rstr_acc(mg_solve_r(mg_solve_index(l,mg_fld)),mg_work2,l)
  ! w  := r
  !       l
  call copy_acc(mg_work2,mg_solve_r(mg_solve_index(l,mg_fld)),nt)
  ! e  := M        w
  !  l     Schwarz
  call hsmg_schwarz_acc(mg_solve_e(mg_solve_index(l,mg_fld)),mg_work2,l)
  ! e  := W e
  !  l       l
  call hsmg_schwarz_wt_acc(mg_solve_e(mg_solve_index(l,mg_fld)),l)
  ! w  := r  - w
  !       l
!\$acc parallel loop
  do i = 0,nt-1
     mg_rk2(i+1) = mg_solve_r(mg_solve_index(l,mg_fld)+i)-alpha*mg_work2(i+1)
  enddo
enddo

call hsmg_rstr_no_dssum_acc(mg_solve_r(mg_solve_index(1,mg_fld)),mg_work2,1)
call hsmg_do_wt_acc(mg_solve_r(mg_solve_index(1,mg_fld)),
      \$          mg_mask(mg_mask_index(1,mg_fld)),2,2,nzw)
!        -1
! e  := A   r
!  1         1
!\$acc update host(mg_solve_e(0:ntco-1),mg_solve_r(0:ntco-1))
      call hsmg_coarse_solve(mg_solve_e(mg_solve_index(1,mg_fld)),
     \$                      mg_solve_r(mg_solve_index(1,mg_fld)))
!\$acc update device(mg_solve_e(0:ntco-1))
      call hsmg_do_wt_acc(mg_solve_e(mg_solve_index(1,mg_fld)),
     \$                   mg_mask(mg_mask_index(1,mg_fld)),2,2,nzw)
do l = 2,mg_lmax-1
  ! w   :=  J e
  !            l-1
  call hsmg_intp_acc(mg_work2,mg_solve_e(mg_solve_index(l-1,mg_fld)),l-1)
  ! e   :=  e  + w
  !  l       l
!\$ACC PARALLEL LOOP
  do i = 0,nt-1
     mg_solve_e(mg_solve_index(l,mg_fld)+i) =
     \$        + mg_solve_e(mg_solve_index(l,mg_fld)+i) + mg_work2(i+1)
  endddo
enddo
! w   :=  J e
!            m-1
call hsmg_intp_acc(mg_work2,mg_solve_e(mg_solve_index(l-1,mg_fld)),l-1)
!\$acc parallel loop
do i = 1,nt
  e(i) = e(i) + copt2*mg_work2(i)
enddo
call ortho_acc(e)
```

Figure 55: The OpenACC implementation for the multigrid preconditioner.

Figure 56 shows the results of a 3D eddy simulation with a Dirichlet boundary condition running on this version of Nek5000, ported to the utilize GPU. The spectral convergence increases over the range N=4, 6, 8, 10, 12, 14, 16 with E=4x4x3 at time step 50. In Figure 57, a single-rod mesh with N=8 and E=2560 shows the pressure solver iteration counts for 1000 steps runs. Validated results agree up to 12-14 digits between CPU and GPU ports of the code.

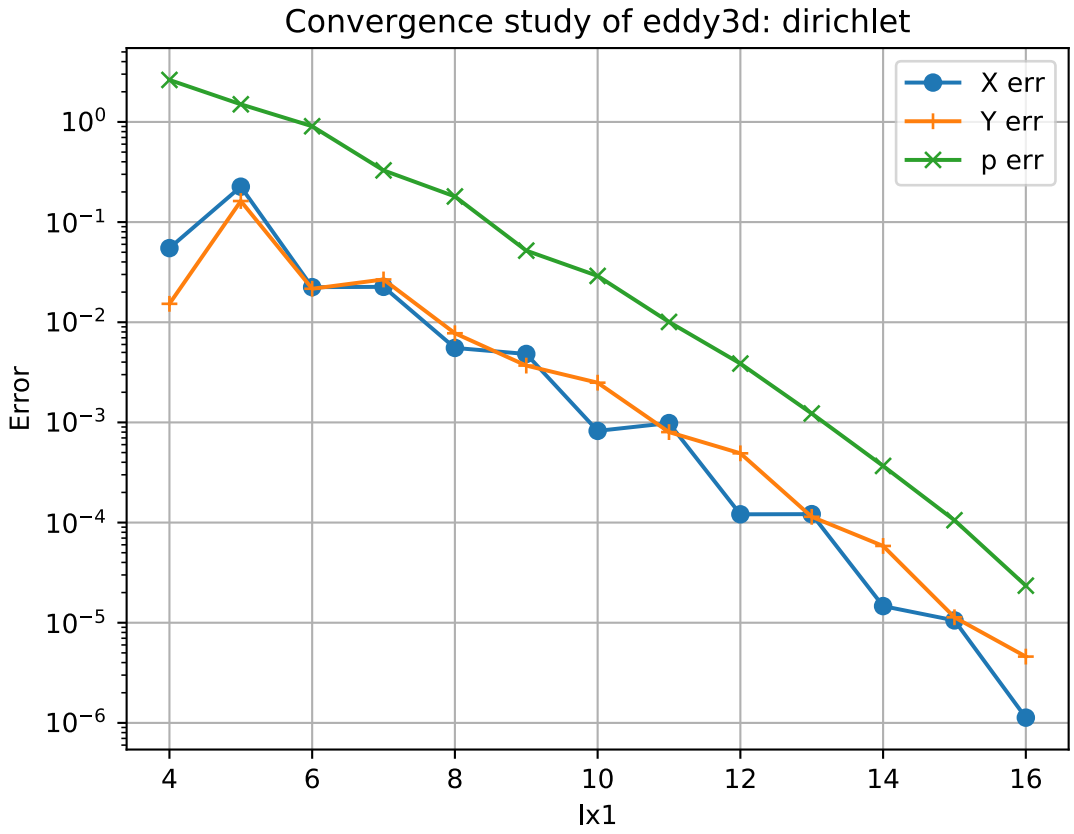


Figure 56: Convergence study of eddy 3D with the Dirichlet boundary condition. Polynomial degree is shown on the x-axis.

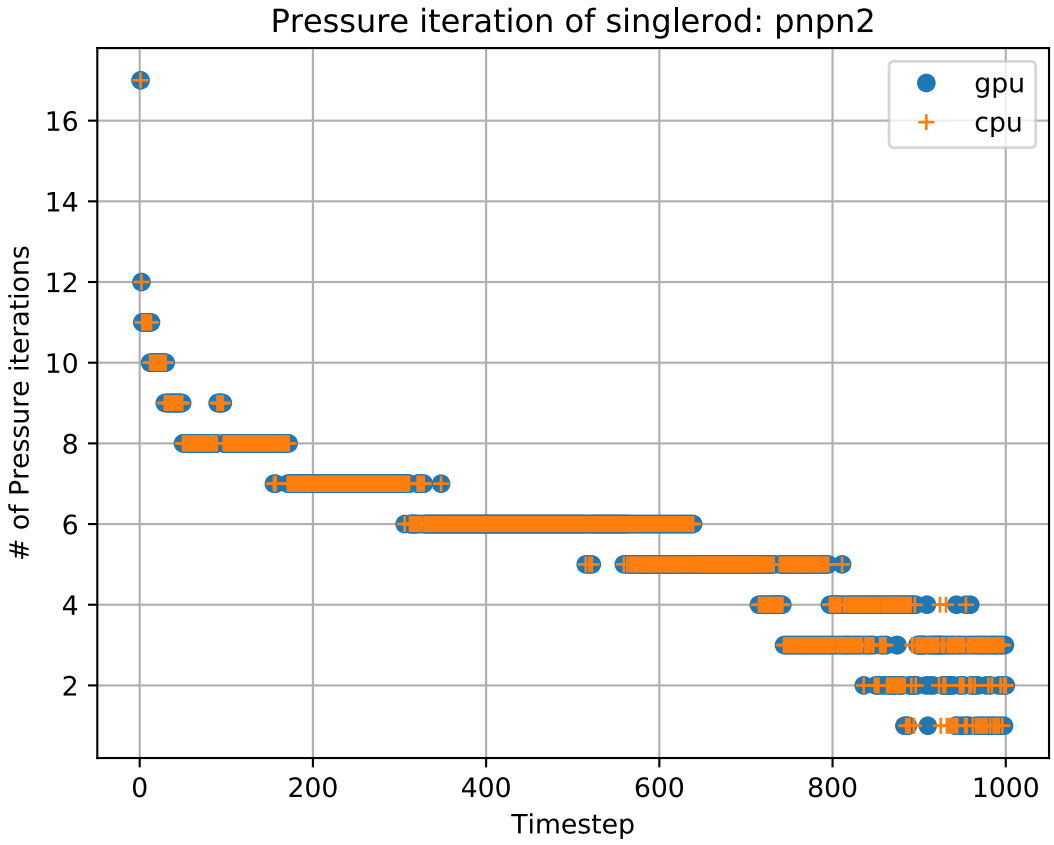


Figure 57: The iteration counts for the pressure by compared GPU and CPU.

The work for all operations in the iteration of the GMRES solver scale as the total number of grid points $n=EN^3$; the operation of matrix-matrix multiplications dominates the total execution times and can be up to 60%. In the full applications, the increased computational intensity leads to improved the performance on the GPU system, as can be seen in Figure 58.

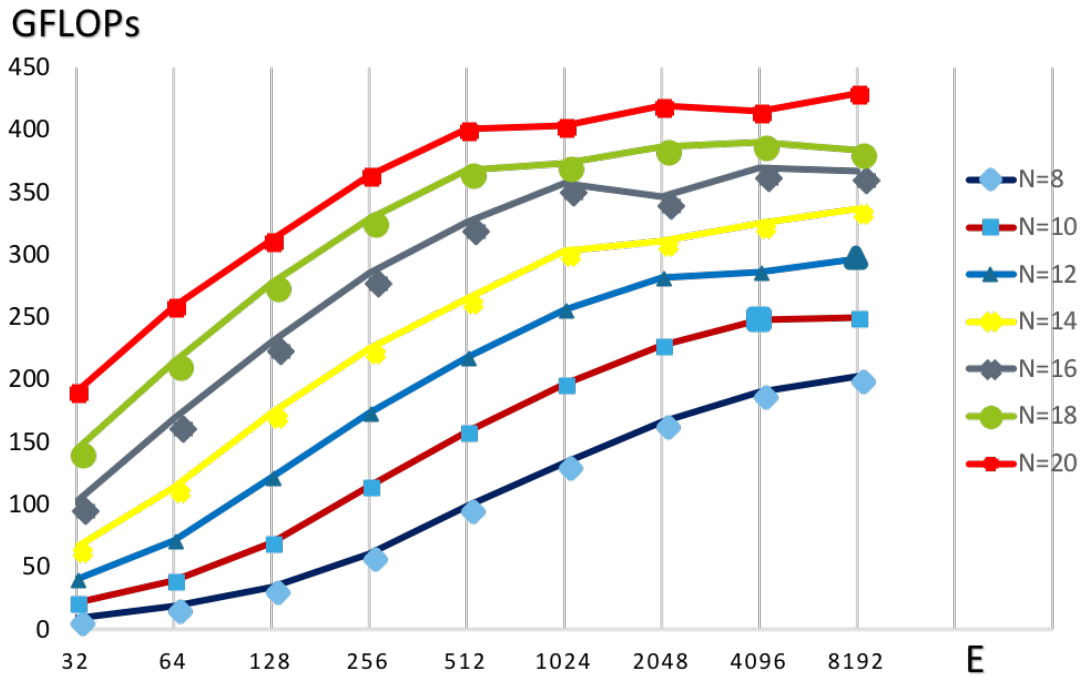


Figure 58: The performance of matrix matrix multiplication on a single NVidia P100 GPU. $N=EN^3$ for varying E and N.

The preliminary performance results of the OpenACC version of Nek5000 for the 3D eddy and stenosis pipe benchmarks are shown in Table 5 and Table 6.

E=48	Method	CPU (second/step)	GPU (second/step)	Speed-up
N=16	GMRES+MG	4.89	0.98	5.0
N=16	GMRES+Jacobi	16.93	2.02	8.4

Table 5: The performance of a 3D eddy simulation with E=48 elements on a single NVidia P100 GPU vs a fully populated CPU.

E=400	Method	4 CPU nodes (second/step)	4 GPUs (second/step)
N=16	GMRES+MG	1.35	2.42

Table 6: The performance of a Stenosis simulation with E=200 elements on 4 NVidia P100 GPUs.

3.5 Code generation

Version 2.0(beta) of the OpenSBLI code has been released as a branch on GitHub under the GNU General Public License: https://github.com/opensbli/opensbli/. This comprises of new capabilities that allows users to select different shock capturing schemes for individual terms of the governing equations at high-level. The current numerical methods supported are Weighted Essentially Non Oscillatory (WENO) schemes with different smoothness indicators (classic Jiang-Shu and new "z") and varying orders (3, 5 and 7) to detect and resolve shock-waves in the flow field. Numerous choices for the boundary conditions (Inflow, Outflow, Isothermal wall, Pressure inflow, extrapolation) are implemented compared to version 1 which supported only periodic or symmetric boundary conditions. Version 2.0 also supports higher order Carpenter boundary closures.

Verification and validation of OpenSBLI has been accomplished through a suite of test cases. The 3D compressible Taylor-Green vortex presented in D2.2 is run with different numerical schemes. Figure 59 shows the visualisation of the z-component of vorticity using 4th order central difference scheme with 16 Million grid points at two different times.

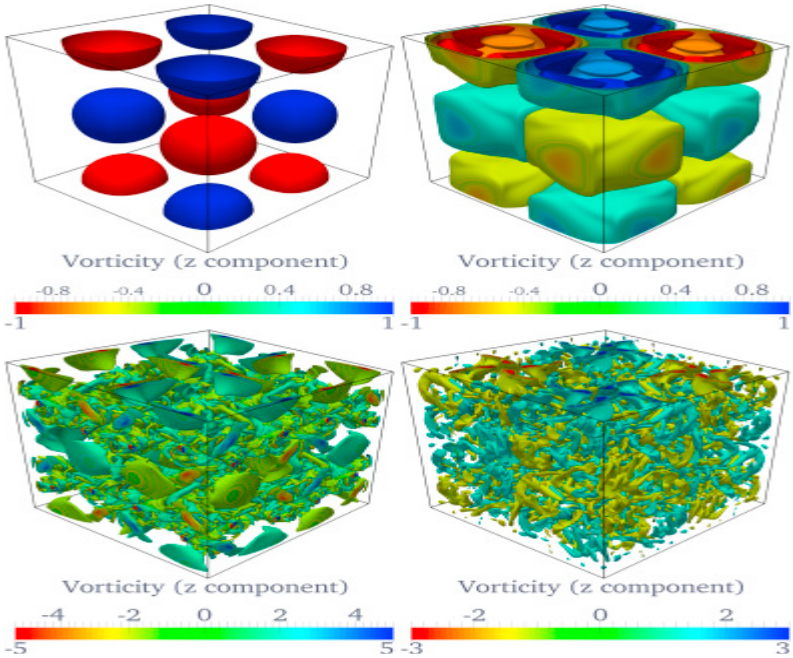


Figure 59: Visualisation of the solution to the Taylor-Green vortex problem in 3D.

The evolution of enstrophy using different schemes compared with the reference DNS data is shown in Figure 60. The central scheme is the least dissipative, but is not suitable for problems with shocks.

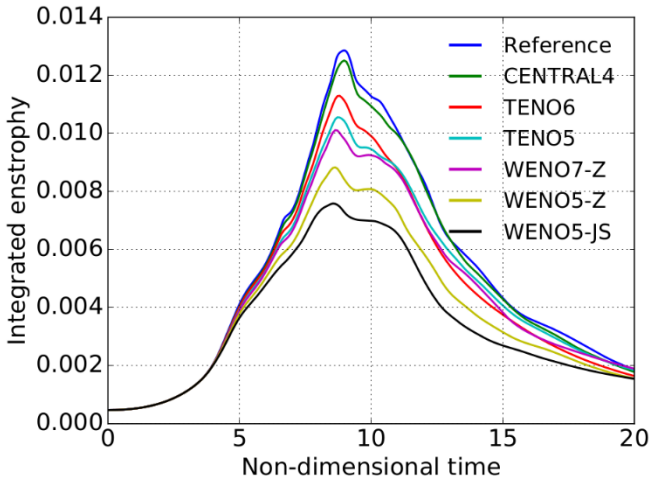


Figure 60: Temporal evolution of enstrophy for the 3D Taylor-Green vortex problem on 256^3 grids. The reference solution is the 512^3 spectral result from [43]

The shock-wave boundary layer interaction simulation is used for validating the implementation of shock capturing schemes in OpenSBLI. Figure 61 shows contours of Mach number after convergence, showing a well resolved shock-wave and the formation of a separation bubble caused by the thickening of the boundary layer at the shock impingement location.

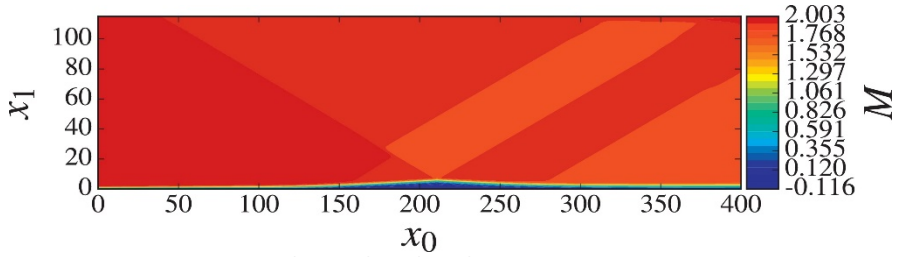


Figure 61: Mach number contours for the final flow field, a well resolved shock-wave impinges on the boundary-layer, causing thickening of the profile and the development of a separation bubble

The skin-friction distribution along the bottom wall is shown in Figure 62, showing good agreement with previously published results. The separation bubble is captured clearly by the numerical method. More details can be found in [41].

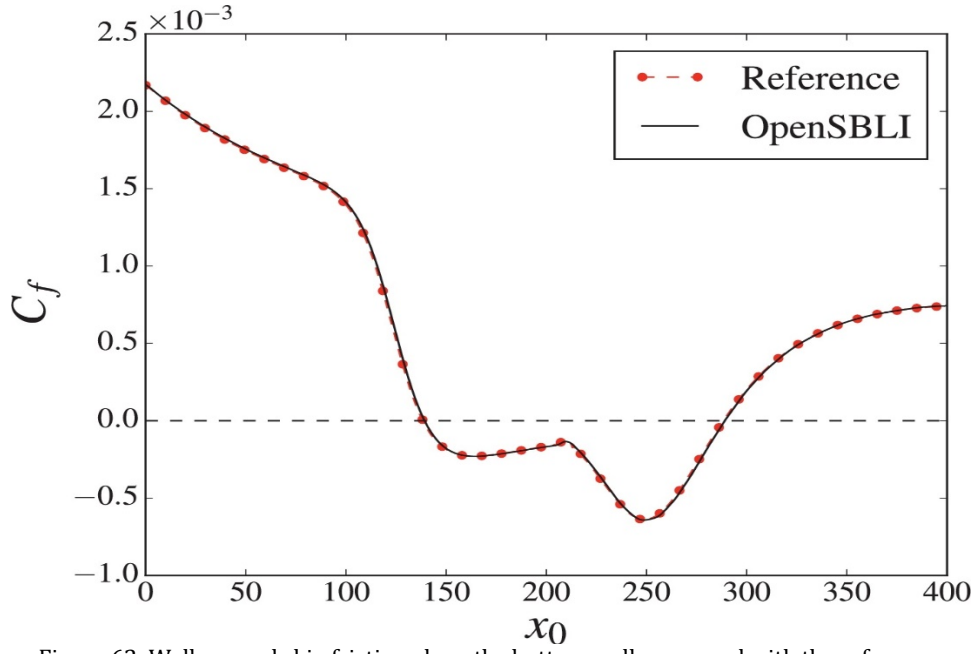


Figure 62: Wall-normal skin friction along the bottom wall compared with the reference.

3.5.1 Algorithms on Architectures and energy efficiency

Here we evalutate the speedup and energy efficiency of five different finite difference algorithms characterized by varying degrees of computational and memory intensiveness, brought about by storing derivatives in memory vs. recomputing them on the device. Their evaluation on multicore CPUs, has been reported in D2.2 and it has been shown that by storing the first derivatives of the velocity components as thread/process-local variables (rather than as global work arrays over the whole grid) a speed-up of ~2 relative to traditional CFD implementations (in which all field values are stored in global work arrays).

These algorithms are run on GPU's and a new low storage algorithm has been added which improves the energy-efficiency on GPU.

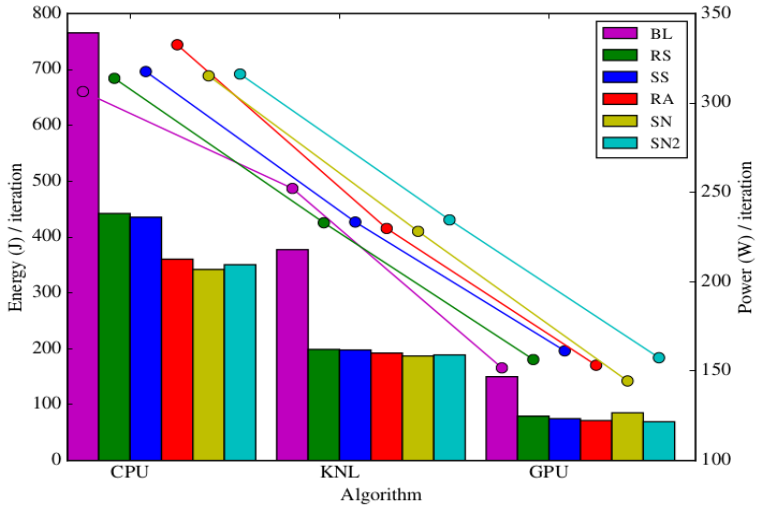


Figure 63: Energy per iteration (bars) and the power consumed per iteration (lines) on different architectures.

The findings are reported by Jammy et al. [42]. To summarise, Figure 63 shows the energy consumed per iteration and the average power consumption per iteration for each algorithm on the three architectures considered (CPU, GPU and KNL). The CPU node use more power per iteration than a KNL or GPU. GPUs use about 50% less power than a CPU node and the KNL systems are in-between. The trend is similar for energy consumption per iteration on three architectures.

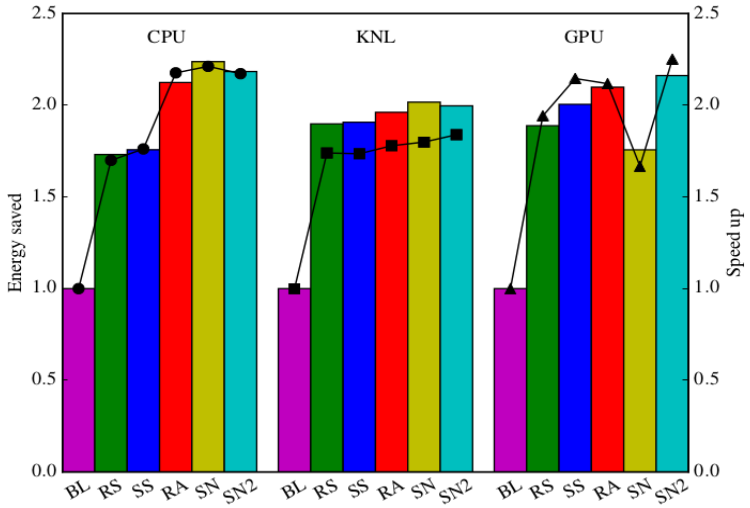


Figure 64: Energy saved (bars) and the speed-up (lines) of algorithms relative to the BL

Figure 64 shows a comparison of speed-up of the algorithms and the energy saved by the algorithms on different architectures. The energy saved is in direct proportion with speed-up relative to the baseline algorithm. The conclusions from the present work include: (a) Across all architectures, the low storage/ high compute intensity algorithms are more energy efficient and gives the best performance; (b) Runtime reduction reduces the energy used by the algorithm; (c) For all the algorithms considered CPUs are the least energy efficient and GPUs are the most energy efficient, with the KNL architecture in-between; (d) For the high compute intensity algorithms, the energy saving of KNL and GPU are ~2 and ~ 5 compared to the CPU node; (e) For optimizing the simulations on GPU, care should be taken to improve data locality

3.5.2 Scaling

OpenSBLI and its dependencies have been successfully installed on ARCHER at EPCC, Titan at Oakridge National Laboratories (ORNL) and Wilkes2 at the University of Cambridge supercomputing facilities for performance analysis and scaling. The description of the systems and the compilers used for the analysis is given in Table 7.

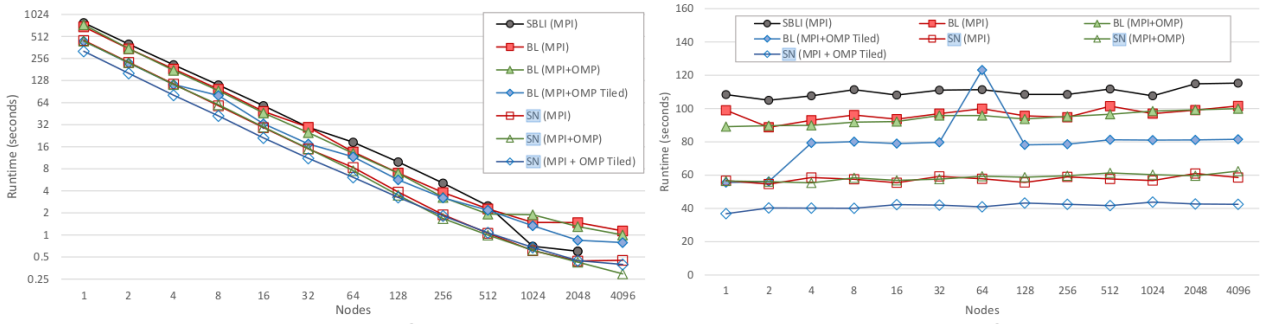
System	Archer (Cray XC30)	Titan (Cray XK7)	Wilkes2 (P100 Cluster)
Node Architecture	Intel Xeon E5-2697 v2 @ 2.7 GHz (Ivy Bridge)	AMD Opteron 6274 @ 2.2 GHz (Interlagos) + NVIDIA Tessla K20x GPU	Intel Xeon E5-2680 v4 @ 2.40GHz + 4 ×NVIDIA Tessla P100 GPUs
Procs × cores per node	2 ×12	1 ×16 + 1 GPU	2 ×12 + 4 ×GPUs
Memory/Node	64 GB	32 GB + 6 GB (K20x)	96GB + 16GB/P100 GPU
Total Nodes(cores), GPUs	4920 Nodes (118,080)	18688 Nodes, 18688 GPUs	90 Nodes, 360 GPUs
Interconnect	Cray Aries	Cray Gemini	Mellanox EDR Infiniband
OS	CLE 3.0.101	CLE 3.0.101	Scientific Linux release 7.5 (Nitrogen)
Compilers	PrgEnv-cray/5.2.82	PrgEnv-cray/5.2.82, CUDA 9.1	Intel Compilers 17.0.4, CUDA 9.0
Compiler flags	`-O3`	`-O3`	`-O3 -fno-alias -finline -inline-forceinline -xHost -parallel`

Table 7: Large-scale systems specifications

When a problem is described at a high-level with OpenSBLI and OPS, explicitly stating how and what data is accessed, it exposes opportunities that can be exploited to increase parallelism and reduce data movement. This allows the generated code to be tailored to different target hardware architectures.

Two such transformations that significantly alter the underlying parallel implementation to achieve higher performance through balancing computation intensity vs. data movement are considered. Of the five algorithms presented in D2.2, two algorithms are considered at the extreme end of computation and memory. (a) baseline (referred to as BL) code and (2) a code requiring minimal storage (store none, SN). The BL algorithm follows the conventional approach frequently used in large-scale finite difference codes such as the original SBLI code, and the SN algorithm uses minimum memory by increasing the computational intensity.

Figure 65 (a) shows the strong scaling of different backend implementations on ARCHER; the MPI, MPI+OpenMP, MPI+OpenMP with tiling versions of the algorithms are used. Note that these are the minimum run times from 5 runs to reduce impact of unusual runs. The strong scaling and runtime of the original SBLI code is also shown to compare the performance of the code-generation approach to that of a hand-written code (SBLI). As seen from the figure, the OpenSBLI code performs similar or better than a hand-written code. OpenSBLI shows very good strong scaling upto 4096 nodes (98304 cores, i.e 4/5th of the total machine).



(a) Strong Scaling - 384^3 mesh **(b)** Weak Scaling - 192^3 mesh per node

Figure 65: Scaling on ARCHER

The weak scaling upto $4/5^{th}$ of the machine size on ARCHER is shown in Figure 65 (b). As seen from the figure, OpenSBLI scales well for all the algorithms and backends considered.

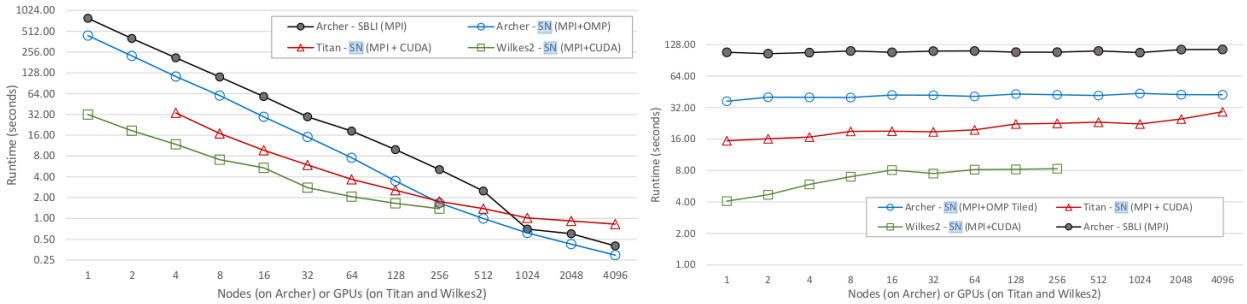


Figure 66: Scaling on Titan and Wilkes2 vs ARCHER.

Next, Figure 66 compares the best scaling runtime on Archer with that of the GPU clusters. The best runtime on Archer is given by SN, MPI+OMP for strong scaling and MPI+OMP Tiled for weak scaling, again these are the minimum run times from 5 different runs. The parallel version on the GPU clusters use MPI+CUDA. For comparison we also indicate the runtime from SBLI on Archer. The code scales well on Titan machine upto 4096 GPU's and 256 GPU's (64 nodes) on the Wilkies 2 machine. The NVIDIA P100 GPU's are ~4 times faster than the K20 series and ~8-9 times faster than Archer. More details can be found in [40].

3.6 Roofline analysis

In this section, we investigate a method for conducting a cache-aware roofline analysis for the Nek5000 code (v17.0.0) using freely available software tools. A code's performance can be plotted as arithmetic intensity versus FLOP rate. The arithmetic intensity is the number of FLOPs performed per byte of data accessed. A high-intensity code then can perform many floating-point operations for little data movement. Therefore, a code that exhibits a lower arithmetic intensity would have a longer runtime due to the fact that that code has to access the cache and main memory more frequently than the high-intensity code for the same number of FLOPs. The performance of a low-intensity code is likely to be memory bound, whereas the performance of a high-intensity code is compute bound. The roofline plots the theoretical maximum performance for any code running on a particular platform. At low arithmetic intensity, the roofline has a positive slope, here the performance is memory bound; but, as the intensity rises so does the peak performance level until the limiting factor changes from memory capacity to CPU frequency and the roofline becomes a straight horizontal line (Figure 67 and Figure 68).

A roofline analysis can be carried out quite straightforwardly using the Intel Advisor tool that comes with Intel Parallel Studio XE 2018 Cluster Edition. The command tool, `advixe-cl`, needs to be run twice with the research code; once to perform a general survey of the code and again to collect specific data such as loop iteration and FLOP counts. These runs will then write the results of the analysis to specific files which can then be viewed via the Intel Advisor GUI (`advixe-gui`). Figure 67 shows a roofline plot generated by an OpenMP code running on ARCHER.

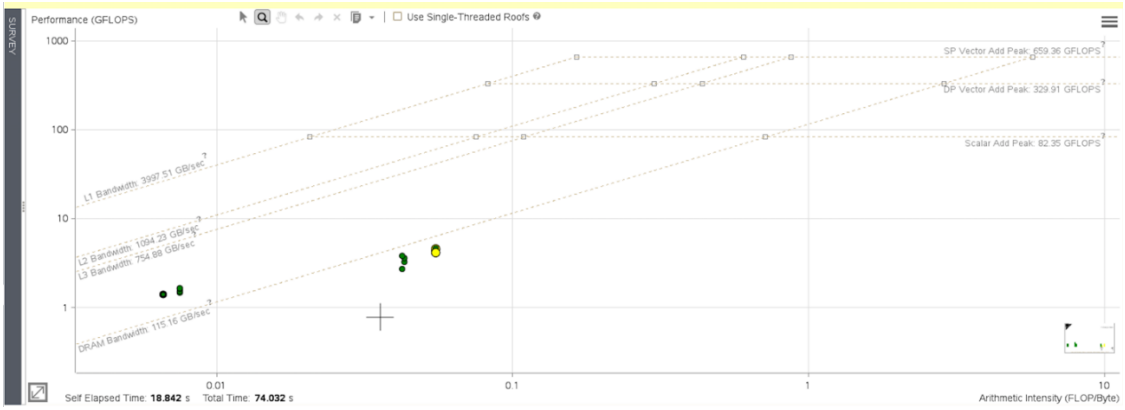


Figure 67: Cache-aware roofline plot generated by Intel Advisor for an OpenMP code running on one node of the ARCHER system.

Intel Advisor is not free to use however; indeed, for a system of the size of ARCHER, the licensing fee is approximately $10,000 per annum. For that reason, we decided to use a tool that has no licensing costs, namely, the Empirical Roofline Tool [35] (v1.1.0) maintained by Berkeley Lab.

The Empirical Roofline Tool (ERT) consists of three pieces of software, a python front-end and two pieces of C code, a driver and a kernel. Typically, the python

front-end, called `ert`, is invoked within the job submission script. The python module then generates the actual batch script (PBS in the case of ARCHER) that specifies the requirements for a driver-controlled kernel to run on the compute nodes. On execution, the kernel runs through a set of floating point operations for a range of data sizes. The completion time for these operations is then used to infer the cache limits. The resulting roofline is then generated by the `gnuplot` program during post-processing, see Figure 68.

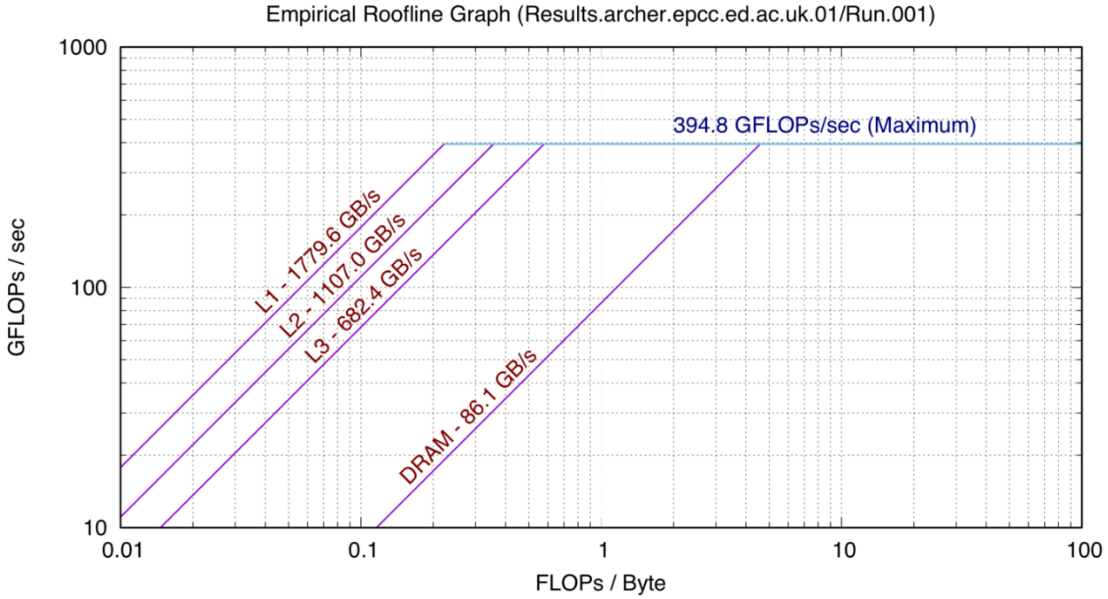


Figure 68: Cache-aware roofline plot generated by the Empirical Roofline Tool v1.1.0 using the inbuilt kernel running on one node of the ARCHER system.

Another advantage of the ERT (compared to Intel Advisor) is that it can be run on any HPC system: the tool does not rely on specific hardware counters since the data movement and FLOP counts are already known for the specially written kernel. Furthermore, ERT is open-source, which means it is possible for the kernel to be extended or replaced by one more suitable to a given hardware platform.

3.6.1 Hardware Counters

Calculating the arithmetic intensity of a code requires one to know the number of floating point operations (FLOPs) along with the size of any associated data movement described by the number of data accesses involving the cache and/or system memory.

The are many counters the could be used to infer the size of the data movement. Below is a list of four such counters revealed by running the `papi_avail` and `papi_native_avail` commands on an ARCHER compute node.

```
PERF_COUNT_HW_CACHE_L1D        L1 data cache (32 kB, per
core)
PAPI_L2_DCA                    L2 data cache (32 kB, per
core)
```

```
PERF_COUNT_HW_CACHE_LL          L3 data cache (30720 kB, per
node)
PERF_COUNT_HW_CACHE_NODE        Node memory (64 GB)
```

However, the counter value used to determine the data movement should not itself be dependent on factors that are extraneous to the algorithms implemented by the code under investigation, e.g., compiler, compiler optimisations and problem size. This was the consideration taken into account by Kwack et al [36]. In this work, the authors implemented a set of geometric series calculations performed whilst traversing a two-dimensional array in order to assess the variability of counter behaviour. The data movement associated with such a kernel is always two variables or 16 bytes (when operating at double precision) regardless of the number of terms in the geometric series. For a given compiler/problem setup, Kwack et al, ran their kernel over a range of series orders ($1+M+M^2+\ldots+M^n$), 1 to 29, recording the output from two counters, L1_DCA and DCA_GOOD.

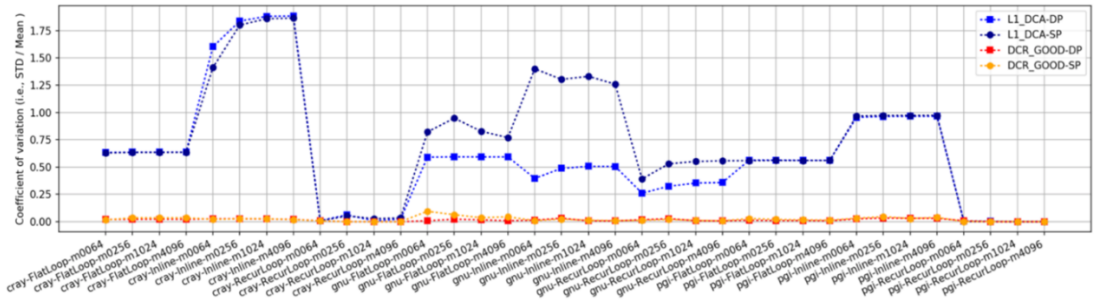


Figure 69: The coefficients of variation for two hardware counters at two levels of precision (single and double). The kernel was compiled with O3 optimisations.

We reproduce above Figure 2 from Kwack et al. (Figure 69). Each point represents a particular setup that identifies the compiler and describes the type of loop used to iterate over the terms in the series. The last part of the setup label (see the x-axis of Figure 69 gives the size of the enclosing array, i.e., the number of times the series calculation is done for each value of *n* (1-29). Hence, each point corresponds to a set of 29 counter values: the standard deviation of these values divided by the mean gives the position on the y-axis (the coefficient of variation).

This work, done on the Blue Waters Cray XE-6 system, revealed DCR_GOOD to be the most robust proxy for data movement. The DCR_GOOD counter records the number of data cache refills satisfied from the L2 cache and/or the system memory. Unfortunately, this counter is specific to the AMD Interlagos processor and so is not available on the ARCHER Intel Ivybridge platform.

Nevertheless, we developed a simple Fortran kernel to calculate a geometric series in the manner described above. In addition, we implemented a separate library as a wrapper to the low-level API part of the Performance Application Programming Interface (PAPI) v5.5.1.4 [37]. We designed these software components such that the hardware counters used to measure the data movement can be specified within the job submission script. The task of recording the change

in counter value due to each set of geometric series calculations is delegated to the PAPI wrapper, `papi_mpi_lib [76]`.

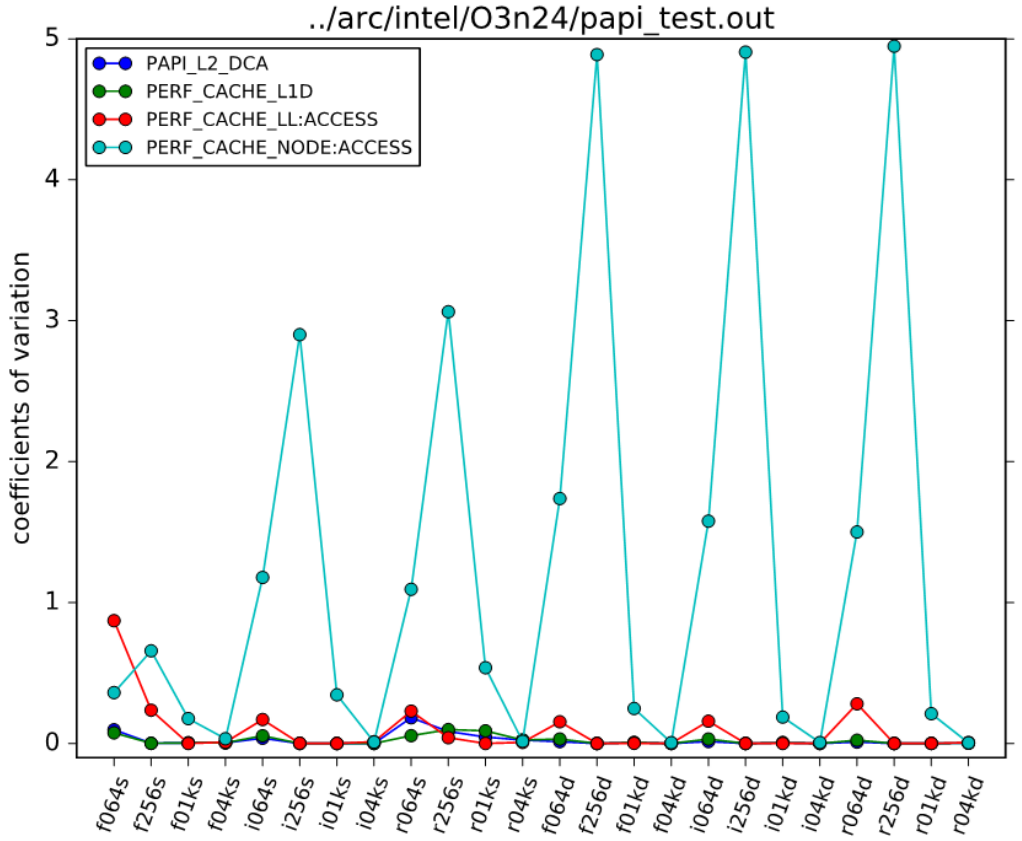


Figure 70: The coefficients of variation (standard deviation over the mean) for four hardware counters available on ARCHER. The labels indicate the environmental setup within which the kernel was run; the first part of the label is the precision (**s**ingle or **d**ouble), the second is the type of loop used to iterate over the series (**f**lat, **i**nline or **r**ecursive), and the third part is the size of the enclosing 2D array (64, 256, 1024 or 4096). The kernel was compiled with the Intel compiler v17.0.0.098 using O3 optimisations.

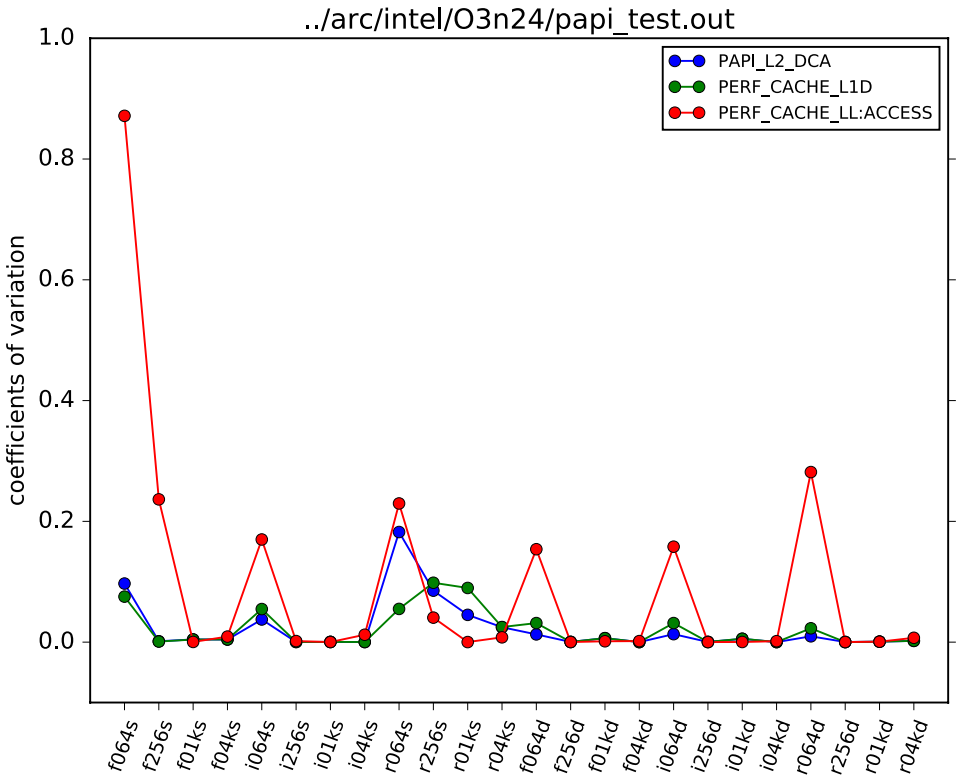


Figure 71: Repeat of Figure 70 but with the most variable counter, PERF_CACHE_NODE:ACCESS, removed.

Figure 70 and Figure 71 indicates that the `PERF_COUNT_HW_CACHE_L1D` counter is the most stable proxy for determining the data movement size, especially for double precision. We show results for the Intel-compiled geometric series kernel since that is the complier used for the Nek5000 code.

Now that we have chosen our counter, we can proceed with measuring the arithmetic intensity of Nek5000 and see where this result falls in relation to the ARCHER roofline (Figure 67).

3.6.2 Measuring the Arithmetic Intensity of Nek5000

We first obtain an average measure for the Nek5000 arithmetic intensity; this is done by inserting the hooks to the PAPI wrapper into the main time step loop implemented by the "`drive1.f`" source code file.

```
subroutine nek_solve
…
call papi_mpi_initialise(papi_out_fn)
do kstep=1,nsteps,msteps
    call papi_mpi_reset(1)
    call nek__multi_advance(kstep,msteps)
    call papi_mpi_record(kstep,1,1,0)

    call papi_mpi_reset(1)
    call check_ioinfo
    call papi_mpi_record(kstep,2,1,0)
```

```
    …
enddo
call papi_mpi_finalise()
```

Basically, each subroutine call within the loop is bookended by two PAPI wrapper calls, `papi_mpi_reset` and `papi_mpi_record`. The first call resets the counter to zero and the second reads the new counter value and writes it to the PAPI log file.

We then ran the Nek5000 code over four ARCHER nodes (96 MPI processes) using the jet-in-crossflow test case. This initial analysis showed that the `nek__multi_advance` subroutine accounts for over 97% of the runtime. The arithmetic intensity for that routine is approximately 0.02 FLOPs/byte with a FLOPs rate of 14 GFLOPs/second.

Figure 72 shows the levels of data movement according to the changing value of the `PERF_COUNT_HW_CACHE_L1D` counter --- the amount of data moved is plotted for each time step.

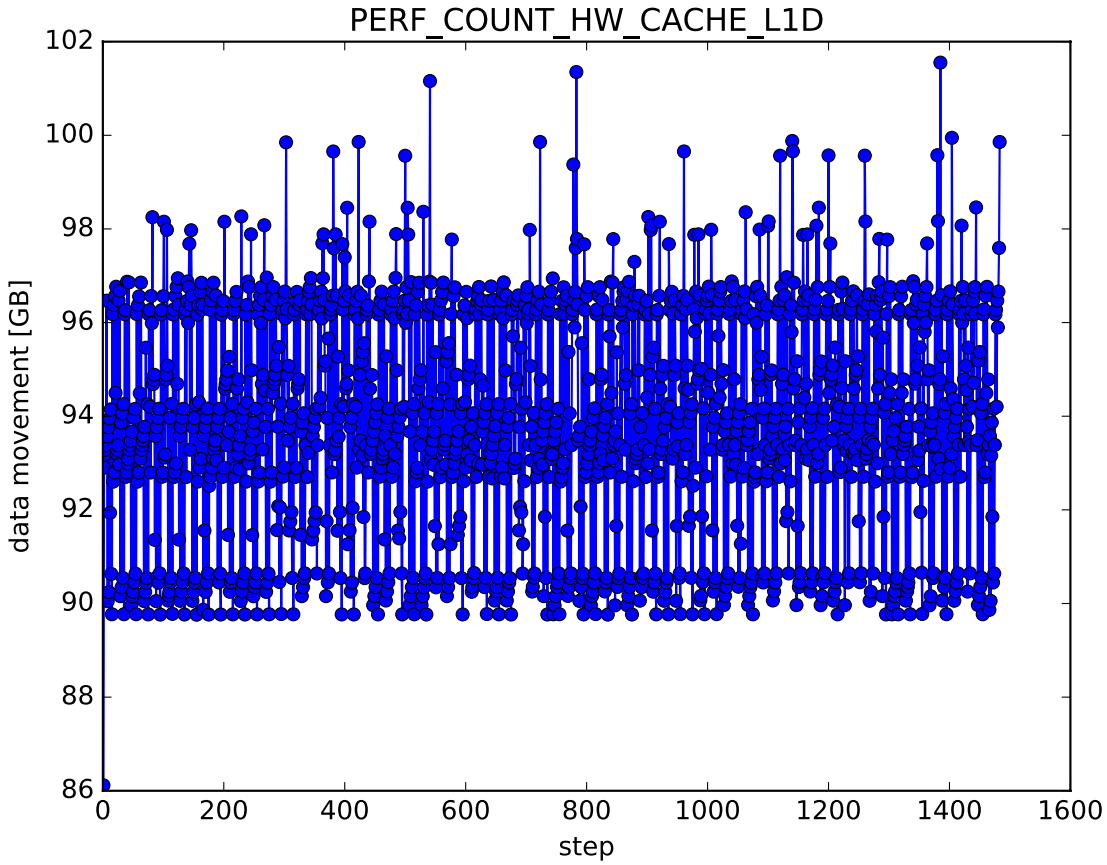


Figure 72: The change in the value of the `PERF_COUNT_HW_CACHE_L1D` counter for each timestep in the range 11 to 1493, i.e., the first ten and last ten steps have been excluded. The ARCHER L1 cache has a line size of 64 bytes, hence, each counter value is multiplied by 64 to get the number of bytes accessed.

On average 94 GB is accessed from the L1 cache every time step, Figure 73 plots the arithmetic intensity.

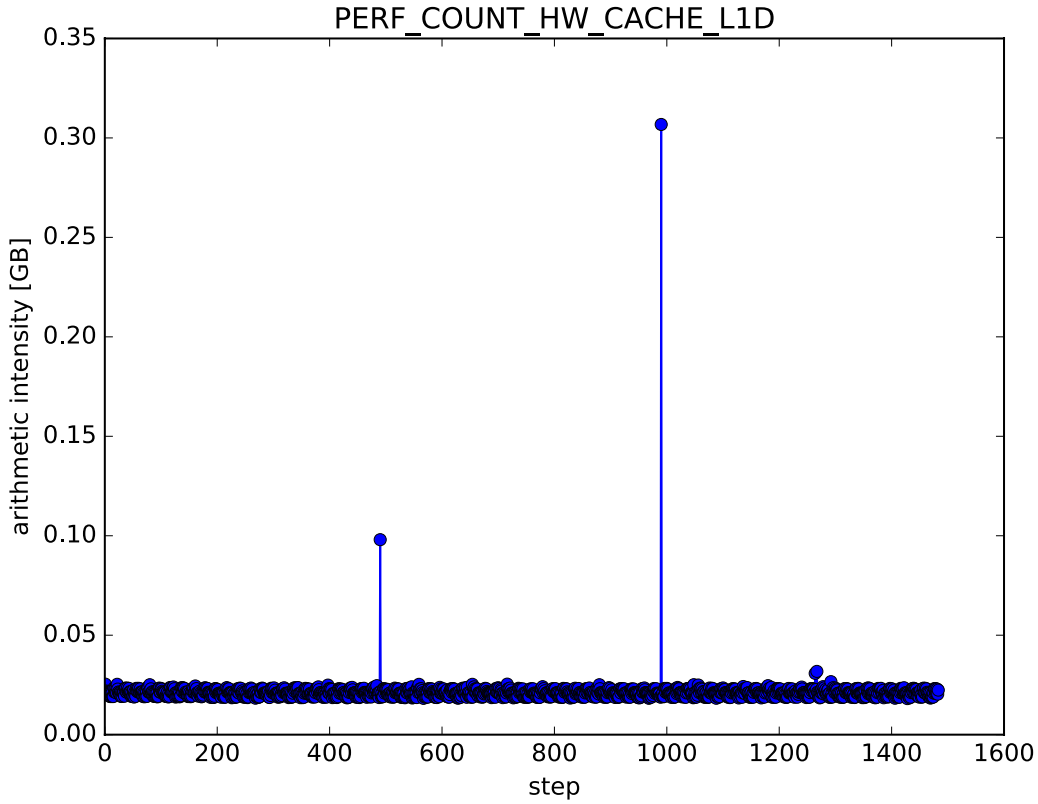


Figure 73: The arithmetic intensity for each timestep in the range 11 to 1493, i.e., the first ten and last ten steps have been excluded.

As mentioned previously, the average intensity is 0.02 FLOPs per byte. Notice the two spikes at steps 500 and 1000, those are the points in the simulation where a checkpoint file is written to disk: the dramatic rise in arithmetic intensity is due to an increased number of FLOPs being issued by the Nek5000 code responsible for checkpointing.

Further profiling inside the `nek__multi_advance` subroutine reveals that the majority of the runtime is spent within the `fluid` subroutine defined in "`drive2.f`", we therefore move our PAPI hooks to this part of the code. Within `fluid` the runtime is dominated by a subroutine called `plan3` ("`planx.f`"). The average intensity measured for `plan3` over the course of the jet-in-crossflow simulation is plotted as a black filled circle in Figure 74. This figure also shows the intensities for four subroutines called from within `plan3`. The names of these subroutines are given in the legend together with a runtime peer percentage.

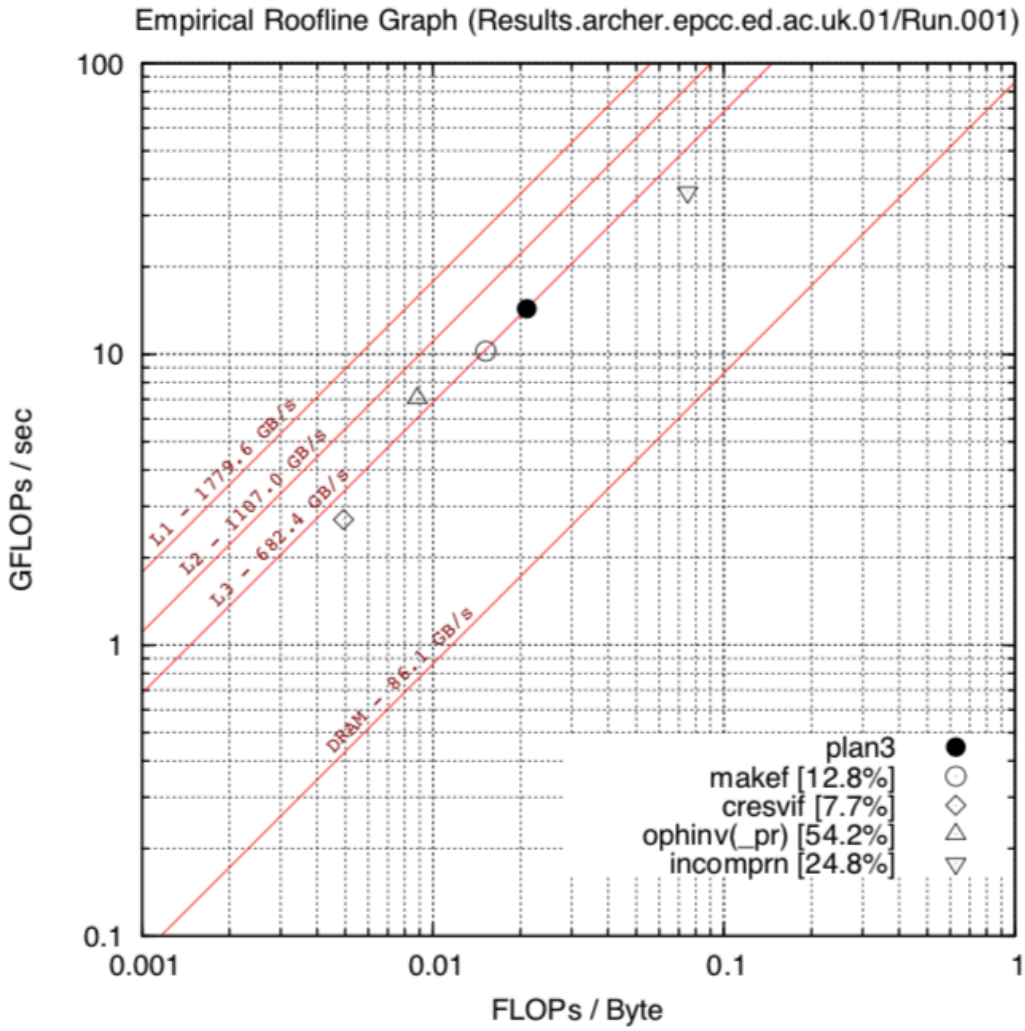


Figure 74: The ARCHER cache rooflines (red) from Figure 3.8.2, decorated with average arithmetic intensity points for the plan3 subroutine (black filled circle) and for four subroutines called by plan3.

Figure 74 suggests that the `plan3` subroutine is memory bound with respect to the L3 cache. The subroutines within `plan3` that account for the majority of the runtime are also close to the L3 cache line. One of these routines however, `ophinv`, is above L3, implying that the performance of that part of the code is likely to be limited by the size of L2 cache instead. The code within `plan3` would need to achieve an arithmetic intensity of 0.58 (L3) or 0.36 (L2) before the CPU speed becomes the limiting factor.

3.6.3 Summary

We have established a suite of software tools that allow us to plot the performance of any code against a roofline that indicates maximum theoretical performance for a specified hardware platform. The use of this software is not limited by licensing costs or by the HPC system.

The Empirical Roofline Tool (v1.1.0) is used to generate the roofline plots for the HPC target. We then employ our geometric series kernel (following the work of Kwack et al. [36]) to find the hardware counter that is the best proxy for data

movement. Having identified a suitable counter, we then link the code under investigation to the PAPI wrapper (`mpi_papi_lib`[76]). This wrapper will then report the change in counter value for specific parts of the code. The FLOPS is recorded at the same time allowing the code performance to be plotted under the roofline.

One aspect of this work that should be investigated however is the degree to which the recorded counter values agree with the expected number of FLOPs and bytes moved. If for example the number of bytes moved inferred from the counter data were overestimated more than the inferred FLOPs then the points plotted under the roofline would need be shifted in a south-east direction, i.e., in the direction of increasing intensity but decreasing FLOP rate.

3.7 A performance model of CG/HDG methods

3.7.1 Motivation for combining CG and HDG

High-order methods on unstructured grids are now increasingly being used to improve the accuracy of flow simulations since they simultaneously provide geometric flexibility and high fidelity. We are particularly interested in efficient algorithms for incompressible Navier-Stokes equations that employ high-order space discretization and a time splitting scheme. The cost of one step in time is largely determined by the amount of work needed to obtain the pressure field, which is defined as a solution to a scalar elliptic problem. Several Galerkin-type methods are available for this task and each has advantages and drawbacks.

The high-order continuous Galerkin (CG) method is the oldest. Compared to its discontinuous counterparts, it involves a smaller number of unknowns (Figure 75), especially in a low-order setting. The CG solution can be accelerated by means of static condensation, which produces a globally coupled system involving only those degrees of freedom on the mesh skeleton. The element interior unknowns are subsequently obtained from the mesh skeleton data by solving independent local problems that do not require any parallel communication.

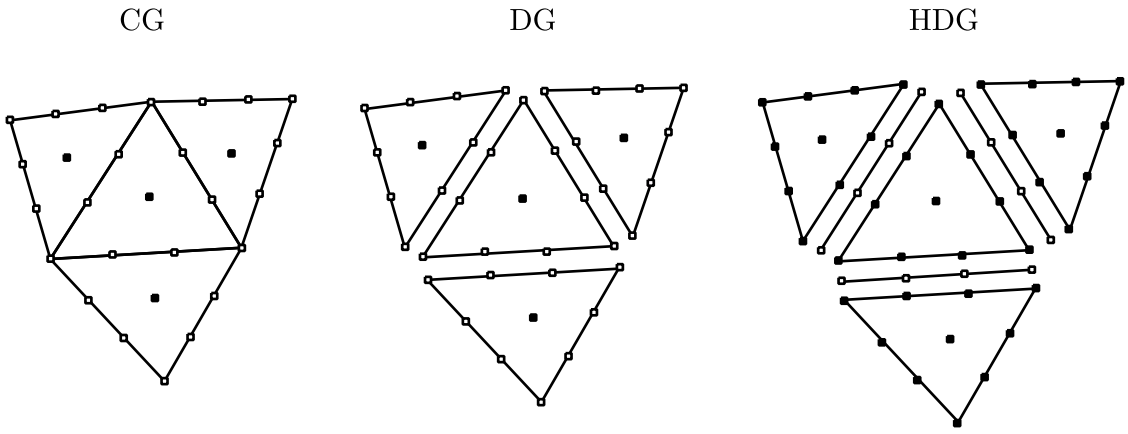


Figure 75: Distribution of unknowns for continuous and discontinuous Galerkin methods.

The amount of information interchanged while constructing and solving the statically condensed system, however, is determined by the topology of the underlying grid. Unstructured mesh generators often produce meshes with high vertex valency (number of elements incident to given vertex) and CG therefore has rather complex communication patterns in parallel runs, which has a negative impact on scaling [193].

Discontinuous Galerkin (DG) methods [191] duplicate discrete variables on element boundaries, thus decoupling mesh elements and requiring at most pairwise communication between them. This is at the expense of larger linear system and greater time spent in the linear solver. Discontinuous discretization is therefore expected to scale better on parallel computers, but the improved scaling is not necessarily reflected in significantly smaller runtimes when compared to a CG solver.

The hybrid discontinuous Galerkin (HDG) methods [191] address this problem by introducing an additional (hybrid) variable on the mesh skeleton. The hybrid degrees of freedom determine the rank of the global system matrix and HDG therefore produces a statically condensed system that is similar in size to the CG case. In contrast with CG, the static condensation in HDG takes place by construction rather than being an optional iterative technique. Similar to the classical DG method, HDG scales favorably in comparison with CG, but the work-to-communication ratio is once again improved due to increased amount of intra-node work rather than due to better overall efficiency.

To maximize the potential of each Galerkin variant in a unified setting, we study a finite element discretization that combines the continuous and discontinuous approach by considering a hybrid discontinuous Galerkin method applied to connected groups of elements supporting a globally continuous polynomial basis. This setting also naturally leads to a formulation of weak Dirichlet boundary conditions for the CG method.

The next section reviews the Hybridizable Discontinuous Galerkin Method which is then modified to obtain the mixed CG-HDG solver. The second part of this report addresses the expected efficiency of a CG-HDG method for elliptic PDEs.

3.7.2 Overview of the formulation of HDG method

We begin with a brief recap of the standard HDG formulation for a finite element mesh, following a similar approach to that taken in [192] and [193].

3.7.2.1 Continuous problem

We seek the solution of a Poisson equation as a representative elliptic problem

$$\begin{aligned} -\nabla^2 \mathrm{u}(\mathrm{x}) &= \mathrm{f}(\mathrm{x}) \qquad \mathrm{x} \in \Omega \\ u(x) &= g_D(x) \qquad x \in \partial\Omega_D \\ n \cdot \nabla u(x) &= g_N(x) \qquad x \in \partial\Omega_N, \end{aligned} \tag{1}$$

on a domain Ω with Dirichlet ($\partial\Omega_D$) and Neumann ($\partial\Omega_N$) boundary conditions, where $\partial\Omega_D \cap \partial\Omega_N = \partial\Omega$ and $\partial\Omega_D \cap \partial\Omega_N = \emptyset$. To formulate the HDG method, we consider a mixed form of (1) by introducing an auxiliary variable $q = \nabla u$:

$$-\nabla \cdot q = \mathrm{f}(x) \qquad x \in \Omega, \tag{2}$$

$$q = \nabla u(x) \qquad x \in \Omega, \tag{3}$$

$$u(x) = g_d(x) \qquad x \in \Omega_D \tag{4}$$

$$q \cdot n = g_N(x) \qquad x \in \partial\Omega_N \tag{5}$$

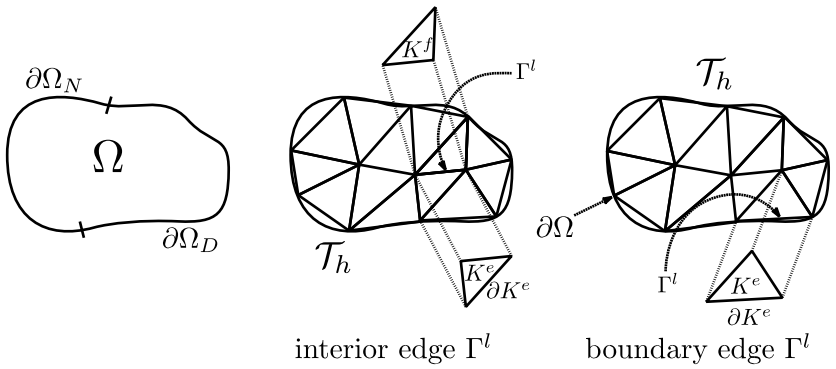


Figure 76: Computational domain and its tesselation demonstrating notation used in the text.

The gradient variable q is approximated together with the primal variable u, which contrasts with the CG method and other discontinuous methods for (1).

3.7.2.2 HDG discretization

We limit ourselves to two-dimensional problems for sake of simplicity, but the formal description remains unchanged in three dimensions. We assume that in the discrete setting, the computational domain Ω is approximated by its tesselation $\mathcal{T}_h$ consisting of non-overlapping and conformal elements K^e such that for each pair of distinct indices $e_i \neq e_j$, $K^{e_i} \cap K^{e_j} = \emptyset$. The symbol Γ^l denotes an interior edge of the tessellation $\mathcal{T}_h$, i.e. an edge $\Gamma^l = \overline{K^i} \cap \overline{K^j}$ where K^i and K^j are two distinct elements of the tesselation. We say that Γ^l is a boundary edge of the tesselation $\mathcal{T}_h$ if there exists an element K^e such that $\Gamma^l = K^e \cap \partial\Omega$ and the length of Γ^l is not zero, as shown in figure 2.

3.7.2.3 Global formulation for HDG problem

Given an element $K \in \mathcal{T}_h$ and two functions $u, v \in L^2(\mathcal{T}_h)$, we define their L^2 scalar product by

$$(u,v)_{\mathcal{T}_h} = \sum_{K\in\mathcal{T}_h}(u,v)_K, \quad \text{where} \quad (u,v)_K = \int_K uv \, dx.$$

Similarly, the L^2 product of functions u and v that are square-integrable on element traces are defined by:

$$\langle u,v\rangle_{\partial\mathcal{T}_h} = \sum_{K\in\mathcal{T}_h}\langle u,v\rangle_{\partial K}, \quad \text{where} \quad \langle u,v\rangle_{\partial K} = \int_{\partial K} uv \, ds$$

The HDG method seeks an approximation pair (u^{DG}, q^{DG}) to u and q, respectively, in the space $V_h \times \mathbf{\Sigma}_h$. The solution is required to satisfy the weak form of (2) and (3)

$$(\mathrm{q}^{\mathrm{DG}}, \nabla v)_{\mathcal{T}_h} = (f,v)_{\mathcal{T}_h} + \langle n^e \cdot \tilde{q}^{DG}, v\rangle_{\partial\mathcal{T}_h} \qquad (6)$$

$$(q^{DG}, w)_{\mathcal{T}_h} = -(u^{DG}, \nabla\cdot w)_{\mathcal{T}_h} + \left\langle \widetilde{u^{DG}}, w\cdot n^e\right\rangle_{\partial\mathcal{T}_h} \qquad (7)$$

for all (v,w) ∈ Vh(Ω)×Σh(Ω) where Vh(Ω) and Σh(Ω) are suitable polynomial spaces and the numerical traces ˜uDG and q˜DG have to be appropriately defined in terms of the approximate solution (uDG,qDG). For details, we refer the reader to [2]. The choice of trace variables allows us to construct the discrete HDG system involving only trace degrees of freedom $\tilde{u}^{DG}$. Once $\tilde{u}$ is known, the element-

interior degrees of freedom represented by both the primal variable u and gradient q can be reconstructed from element boundary values.

3.7.2.4 Local solvers in the HDG method

Assume that the function

$$\lambda := \tilde{u}^{DG} \in \mathcal{M}_h, \quad (8)$$

is given with $\mathcal{M}_h$ being a finite element space defined on element traces. Then the solution restricted to element K^e is a function $u^e, \boldsymbol{q}^e \text{ in } P(K^e) \times \boldsymbol{\Sigma}(K^e)$ that satisfies the following equations:

$$(q^e, \nabla v)_{K^e} = (f, v)_{K^e} + \langle n^e \cdot \tilde{q}^e, v \rangle_{\partial K^e} \quad (9)$$

$$(q^e, w)_{K^e} = -(u^e, \nabla \cdot w)_{K^e} + \langle \lambda, w \cdot n^e \rangle_{\partial K^e}, \quad (10)$$

for all $(v,w) \in P(K^e) \times \boldsymbol{\Sigma}(K^e)$. For a unique solution of the above equations to exist, the numerical trace of the flux must depend only on λ and on $(u^e, \boldsymbol{q}^e)$:

$$\tilde{q}^e(x) = q^e(x) - \tau\big(u^e(x) - \lambda(x)\big)n^e \quad \text{on} \quad \partial K^e \quad (11)$$

for some positive function τ. The analysis presented in [2] reveals that as long as $\tau > 0$, its value can be arbitrary without degrading the robustness of the solver. For the limiting value of $\tau \to \infty$, one obtains a statically condensed continuous Galerkin formulation. In this sense, τ plays the role of a method selector as opposed to traditional penalty parameter used in Nitsche's method, for example.

3.7.2.5 Global problem for trace variable

We denote by (U_λ, Q_λ) and by (U_f, Q_f) the solution to the local problem (9), (10) when $\lambda = 0$ and $f = 0$, respectively. Due to the linearity of the original problem (1) and its mixed form, the solution satisfies

$$(u^{HDG}, \boldsymbol{q}^{HDG}) = (U_\lambda, \boldsymbol{Q}_\lambda) + (U_f, \boldsymbol{Q}_f). \quad (12)$$

In order to uniquely determine λ, we require that the boundary conditions be weakly satisfied and the normal component of the numerical trace of the flux $\tilde{q}$ given by (11) is single valued, rendering the numerical trace conservative.
We say that λ is the element of trace space $\mathcal{M}_h$ such that

$$\lambda = \mathrm{P_h}(g_D) \quad \text{on} \quad \partial\Omega_D \quad (13)$$

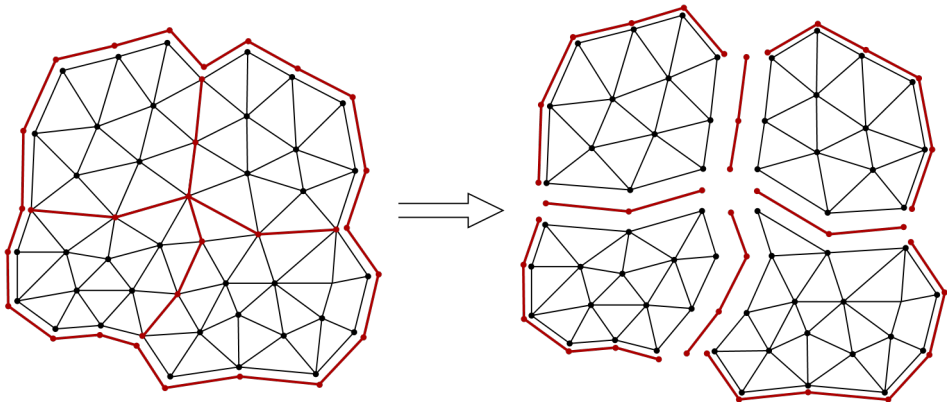
$$\langle \mu, \tilde{q} \cdot n \rangle_{\partial \mathcal{T}} = \langle \mu, g_N \rangle_{\partial \Omega_N}, \quad (14)$$

for all $\mu \in \mathcal{M}_h^0$ such that $\mu = 0$ on $\partial\Omega_D$. Here P_h denotes the L^2-projection into the space of restrictions to $\partial\Omega_D$ of functions of $\mathcal{M}_h$.

3.7.3 Combined Continuous-Discontinuous Formulation

To take advantage of the efficiency and lower memory requirements of continuous Galerkin method together with the flexibility and more favorable communication patterns of discontinuous Galerkin methods in domain-decomposition setting, we combine both as follows. Each mesh partition is seen as a 'macro-element', where the governing equation is discretized by continuous Galerkin solver, while the

patches are coupled together weakly as in HDG. This means that the scalar flux (hybrid variable) λ is only defined on inter-partition boundaries.



Given the hybrid CG-HDG setup, the HDG local solver defined previously for one element would now be applied to a group of elements supporting a piecewise-continuous basis. The motivation of this section is to take the matrix form of the HDG solver and apply it in such piecewise-continuous setting. We will show that the discrete weak form reduces to the 'standard' Laplace operator plus extra terms, which will be only applied on elements adjacent to partition boundaries, providing weak coupling between each partition and the global trace variable.

We start again from the weak mixed problem (9), (10), but integrate the second term in the flux equation (10) by parts once again. This modified flux form allows for a symmetric boundary contribution to the linear system as will be explained shortly. In order to distinguish between the standard HDG local solver within a single element and HDG applied to the whole domain tesselation T_h, the superscript '*e*' has been replaced by T_h where appropriate. The 'macro element' form yields a system

$$(\boldsymbol{q}^{\mathcal{T}_h}, \nabla v)_{\mathcal{T}_h} = (f, v)_{\mathcal{T}_h} + \langle \boldsymbol{n}^{\mathcal{T}_h} \cdot \tilde{\boldsymbol{q}}^{\mathcal{T}_h}, v \rangle_{\partial \mathcal{T}_h} \tag{15}$$

$$(\boldsymbol{q}^{\mathcal{T}_h}, \boldsymbol{w})_{\mathcal{T}_h} = (\nabla u^{\mathcal{T}_h}, \boldsymbol{w})_{\mathcal{T}_h} - \langle u^{\mathcal{T}_h}, \boldsymbol{w} \cdot \boldsymbol{n}^{\mathcal{T}_h} \rangle_{\partial \mathcal{T}_h} + \langle \lambda, \boldsymbol{w} \cdot \boldsymbol{n}^{\mathcal{T}_h} \rangle_{\partial \mathcal{T}_h} \tag{16}$$

The numerical approximation $u^{\mathcal{T}_h}$ belongs to the space $V_h^{\mathcal{T}_h}$ and $q^{\mathcal{T}_h}$ lies in $\boldsymbol{\Sigma}_h^{\mathcal{T}_h}$, which are defined as

$$V_h^{\mathcal{T}} := \{ v \in \mathcal{C}^0(\Omega) \quad : \quad v|_{K^e} \in P(K^e) \quad \forall K^e \in \mathcal{T}_h \},$$

$$\boldsymbol{\Sigma}_h^{\mathcal{T}} := \{ \boldsymbol{w} \in [L^2(\Omega)]^2 \quad : \quad \boldsymbol{w}|_{K^e} \in \boldsymbol{\Sigma}(K^e) \quad \forall K^e \in \mathcal{T}_h \}.$$

The trace flux is now only defined on *partition boundary* as

$$\tilde{\boldsymbol{q}}^{\mathcal{T}_h} = \boldsymbol{q}^{\mathcal{T}_h}(\boldsymbol{x}) - \tau \big(u^{\mathcal{T}_h}(\boldsymbol{x}) - \lambda(\boldsymbol{x}) \big) \boldsymbol{n}^{\mathcal{T}_h} \quad \text{on} \quad \partial \mathcal{T}_h \ .$$

A continuous Galerkin solver with Dirichlet data prescribed by the variable λ can be obtained by eliminating the flux variable from the system and reverting back to primal form for the unknown *u*. The matrix formulation written for a single element $K^e \in \mathrm{T}_h$ adjacent to Dirichlet boundary reads

$$\sum_{k=1,2}\left\{(\mathbf{D}_K^e)^T-\sum_{l=1}^{N_b^e}\tilde{\mathbf{E}}_{kl}^e\right\}\hat{\underline{q}}_k^e+\sum_{l=1}^{N_b^e}\tau^{e,l}\left[\mathbf{E}_l^e\hat{\underline{u}}^e-\mathbf{F}_l^e\hat{\underline{\lambda}}^{\sigma(e,l)}\right]=\underline{f}^e \tag{17}$$

$$\mathbf{M}^e\hat{\underline{q}}_k^e=\left\{(\mathbf{D}_K^e)-\sum_{l=1}^{N_b^e}\tilde{\mathbf{E}}_{kl}^e\right\}\hat{\underline{u}}^e+\sum_{l=1}^{N_b^e}\tilde{\mathbf{F}}_{kl}^e\hat{\underline{\lambda}}^{\sigma(e,l)},\qquad k=1,2 \tag{18}$$

For definitions of matrices in the above formulation and detailed derivation, please see deliverable D1.3. The discrete flux $\hat{\underline{q}}_k^e$ expressed from (18) and substituted in equation (17) yields element-wise contribution to the left- and right-hand side of the linear system which can be expressed as

$$\sum_{k=1,2}\Bigg\{\underbrace{(\mathbf{D}_k^e)^T(\mathbf{M}^e)^{-1}\mathbf{D}_k^e}_{\boxed{1}}-\underbrace{\left(\sum_{l=1}^{N_b^e}\tilde{\mathbf{E}}_{kl}^e\right)(\mathbf{M}^e)^{-1}\mathbf{D}_k^e}_{\boxed{2a}}\Bigg\}\hat{\underline{u}}^e$$

$$-\sum_{k=1,2}\Bigg\{\underbrace{(\mathbf{D}_k^e)^T(\mathbf{M}^e)^{-1}\left(\sum_{l=1}^{N_b^e}\tilde{\mathbf{E}}_{kl}^e\right)}_{\boxed{2b}}+\underbrace{\left(\sum_{l=1}^{N_b^e}\tilde{\mathbf{E}}_{kl}^e\right)(\mathbf{M}^e)^{-1}\left(\sum_{l=1}^{N_b^e}\tilde{\mathbf{E}}_{kl}^e\right)}_{\boxed{3}}\Bigg\}\hat{\underline{u}}^e+\underbrace{\sum_{l=1}^{N_b^e}\tau^{(e,l)}\mathbf{E}_l^e\hat{\underline{u}}^e}_{\boxed{4}}$$

$$=\underline{f}^e+\sum_{k=1,2}\left\{\left(\sum_{l=1}^{N_b^e}\tilde{\mathbf{E}}_{kl}^e-(\mathbf{D}_k^e)^T\right)(\mathbf{M}^e)^{-1}\left(\sum_{l=1}^{N_b^e}\tilde{\mathbf{F}}_{kl}^e\hat{\underline{\lambda}}^{\sigma(e,l)}\right)\right\}+\sum_{l=1}^{N_b^e}\tau^{(e,l)}\mathbf{F}_l^e\hat{\underline{\lambda}}^{\sigma(e,l)}$$

Term $\boxed{1}$ on the left-hand side is a discrete Laplacian that arises from the standard continuous Galerkin discretization, which would typically be accompanied by the forcing term f^e on the right hand side. This new expression therefore denotes a modification of the existing matrix system and right-hand side, which makes implementation relatively straightforward. The matrix expressions $\boxed{2a}$, $\boxed{2b}$, $\boxed{3}$ and $\boxed{4}$ appear in the formulation only for elements K^e containing at least one edge on Dirichlet boundary of Ω. In addition, expressions $\boxed{3}$ and $\boxed{4}$ are symmetric as a consequence of symmetry of $\tilde{E}_{kl}^e$, E_l^e and $(M^e)^{-1}$. The products $\boxed{2a}$ and $\boxed{2b}$ are transposes of each other, hence their sum is again symmetric. The modifications to the symmetric discrete Laplacian therefore preserve symmetry of the discrete weak form, meaning that efficient iterative solvers such as the conjugate gradient method can be used to obtain solutions.

3.7.4 Expected Performance

3.7.4.1 Cost in terms of FLOPs

We assume that the discrete Poisson problem is solved in two stages, both of which will significantly contribute to the overall CPU time spent in the solver. The stages are:

1. **Assembly and solution of statically condensed system.** This step involves processing unknowns on entity boundaries, where 'entity' would be each single element in the context of continuous and hybrid discontinuous Galerkin methods and one mesh patch (a group of elements spanned by a continuous polynomial basis) in the combined continuous-discontinuous Galerkin method.

The main difference between CG and HDG is that in the continuous case, trace variables are identical to variables located on element boundaries and are shared by neighboring elements. The HDG method, on the other hand, introduces an additional hybrid variable, thus requiring more memory storage. This variable is not globally continuous, hence degrees of freedom on face boundaries are duplicated. Therefore, the HDG interior solve on each element has to process a slightly larger local system.

2. **Interior solve.** Given solution on entity boundary, the solution in entity interior is reconstructed during this stage. Interior solve involves the inverse of a potentially large matrix.

Setup costs (for example precomputing and storing the matrix inverses needed in interior solve above) are not considered.

Domain Description

We assume a structured grid divided into $P \times P$ patches, each patch consisting of $N_e^{1D} \times N_e^{1D}$ elements (Figure 77). Each element has a polynomial basis of degree p, i.e. $(p + 1) \times (p + 1)$ degrees of freedom. These may or may not be shared with neighboring elements, depending on the setup (CG vs. DG vs. HDG) and global continuity of the polynomial bases. The number of inter-patch edges (red edges in Figure 77) is

$$N_{interpatch}^{edges} = 2P \cdot (P-1) \cdot N_e^{1D},$$

and each patch contains N_{patch}^{edges} interior edges, with

$$N_{patch}^{edges} = 2N_e^{1D} \cdot (N_e^{1D} - 1),$$

see figure 6.

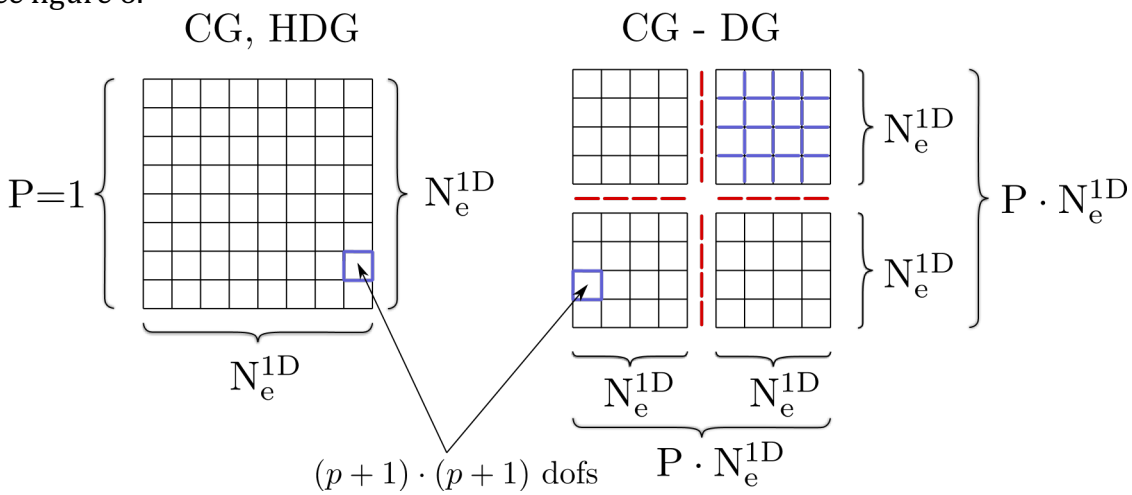


Figure 77: Idealized mesh divided into $P \times P$ patches, each patch containing $N_e^{1D} \times N_e^{1D}$ elements of order p.

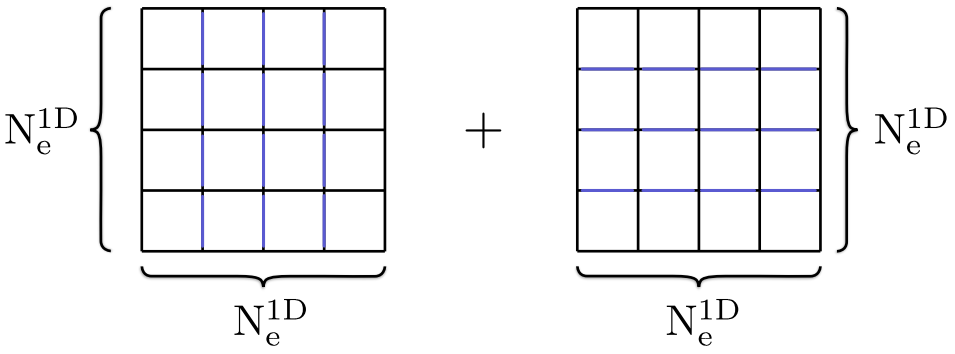


Figure 78: Interior edges (blue) within a patch.

3.7.4.2 Continuous Galerkin method

Since the total number of elements along each side of the mesh is $P \cdot N_e^{1D}$ in 2D, the total number of unknowns before static condensation (assuming Dirichlet boundary condition everywhere) is

$$N_{CG}^{dof} = \left(P \cdot N_e^{1\mathrm{D}} \cdot p - 1\right)^2.$$

This is also the rank of global system matrix. In case of one-level static condensation, the global system has the form

$$\begin{bmatrix} \mathbf{M}_b & \mathbf{M}_c \\ \mathbf{M}_c^T & \mathbf{M}_i \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_b \\ \boldsymbol{x}_i \end{bmatrix} = \begin{bmatrix} \boldsymbol{f}_b \\ \boldsymbol{f}_i \end{bmatrix}$$

and the rank of $\mathbf{M}_b$ is *approximately* (counting the boundary modes on the skeleton of the mesh) equal to

$$\begin{aligned} N_{CG}^{\lambda} &= \left(N_{interpatch}^{edges} + P \cdot N_{patch}^{edges}\right) \cdot p \\ &= \left(2P \cdot (P-1) \cdot N_e^{1D} + P \cdot 2N_e^{1D} \cdot (N_e^{1D} - 1)\right) \cdot p \\ &= 2\mathrm{P}N_e^{1D}(P + N_e^{1D} - 2)\mathrm{p} \end{aligned}$$

The remaining values of u in element-interior degrees of freedom can be obtained by inverting $(P \cdot N_e{}^{1\mathrm{D}})^2$ local matrices of rank $p - 1$. This means that the total cost of solving the CG problem is

$$C_{CG} = \mathcal{O}\left(cgsolve(PN_e^{1\mathrm{D}}(P + N_e^{1\mathrm{D}})p)\right) + (PN_e^{1\mathrm{D}})^2 \cdot \mathcal{O}\left((p-1)^3\right),$$

where *cgsolve*(n) is the cost function of solving a sparse system of rank n with conjugate gradients. The cost of the second term is small if the blocks of $\mathbf{M}_i$ are inverted and stored during setup phase. The second term in the estimate assumes that the inverse of each diagonal block of $\mathbf{M}_i$ costs as much as Gauss elimination/LU decomposition of a matrix of rank $p-1$, which has cubic time complexity.

3.7.4.3 HDG

The discrete transmission condition (14) generates a sparse system of rank

$$N_{HDG}^{\lambda} = \left(N_{interpatch}^{edges} + P \cdot N_{patch}^{edges}\right) \cdot (p+1)$$
$$= \left(2P(P-1)N_e^{1D} + P \cdot 2N_e^{1D}(N_e^{1D}-1)\right) \cdot (p+1)$$
$$= 2PN_e^{1D}(P + N_e^{1D} - 2)(p+1).$$

In addition, we need to invert $(PN_e^{1D})^2$ local systems $\in R^{(p+1)\times(p+1)}$ as in the CG case. The backsolve is more expensive however, because we have d additional mixed variables q_1 ..., q_d in d dimensions. The element-local inversion can be again precomputed and stored during setup.

The overall cost of solving for all unknowns scales as

$$C_{HDG} = \mathcal{O}\left(cgsolve(PN_e^{1D}(P + N_e^{1D})(p+1))\right) + (PN_e^{1D})^2 \cdot \mathcal{O}\left((p+1)^3\right).$$

3.7.4.4 Combined CG-HDG Solver

The number of hybrid degrees of freedom on interfaces between patches is

$$N_{CG-DG}^{\lambda} = N_{interpatch}^{edges}(p+1) = 2P(P-1)N_e^{1D}(p+1)$$

.

Each patch contains *approximately* $(N_e^{1D}p)^2$ interior degrees of freedom, hence the total cost is

$$C_{CG-DG} = \mathcal{O}\left(cgsolve(P^2N_e^{1D}(p+1))\right) + P^2 \cdot \mathcal{O}\left((N_e^{1D}p)^3\right).$$

In the limiting case where each patch coincides with one single element (i.e. $P := N_e^{1D}$ and $N_e^{1D} = 1$), the three estimates C_{CG}, C_{HDG} and C_{CG-DG} predict the same asymptotic cost.

3.7.4.5 Standard HDG Algorithm

The cost of linear solve in the PCG (preconditioned conjugate gradient) solver will mainly depend on the cost of evaluating matrix-vector multiplications. For a matrix of rank n, this cost is $O(n^2)$. Nektar++ solves the statically condensed system in matrix-free manner by performing the above matrix-vector multiplications *element-wise* and then summing them together. Suppose the (structured) mesh consists of quadrilaterals in 2D and hexahedra in 3D. Furthermore, we will assume that the triangular mesh is obtained by splitting each quadrilateral into 2 triangles and tetrahedral mesh is created by dividing each hexahedron into 6 tetrahedra.

The number of trace degrees of freedom of one element is

- $3 \cdot (p+1)$ for **triangles**
- $4 \cdot (p+1)$ for **quadrilaterals**
- $4 \cdot \frac{(p+1)(p+2)}{2}$ for **tetrahedra**
- $6 \cdot (p+1)^2$ for **hexahedra**

Under this assumption, *one matrix-vector multiplication* for the whole system (but performed on element-wise basis) will take

- $\mathcal{O}\left(2(N_e^{1\mathrm{D}})^2\left[3\cdot(p+1)\right]^2\right)=\mathcal{O}\left(18(N_e^{1\mathrm{D}})^2(p+1)^2\right)$ operations on **triangles**
- $\mathcal{O}\left((N_e^{1\mathrm{D}})^2\left[4\cdot(p+1)\right]^2\right)=\mathcal{O}\left(16(N_e^{1\mathrm{D}})^2(p+1)^2\right)$ operations on **quadrilaterals**
- $\mathcal{O}\left(6(N_e^{1\mathrm{D}})^3\left[2\cdot(p+1)(p+2)\right]^2\right)=\mathcal{O}\left(24(N_e^{1\mathrm{D}})^3(p+1)^2(p+2)^2\right)$ operations on **tetrahedra**
- $\mathcal{O}\left((N_e^{1\mathrm{D}})^3\left[6\cdot(p+1)^2\right]^2\right)=\mathcal{O}\left(36(N_e^{1\mathrm{D}})^3(p+1)^4\right)$ operations on **hexahedra**

3.7.4.6 HDG Algorithm Applied to Groups of Continuously Connected Elements (CG-HDG)

Now suppose that the trace system is built between patches and each patch has $N_e^{1\mathrm{D}} \times N_e^{1\mathrm{D}}$ quadrilaterals in 2D and $N_e^{1\mathrm{D}} \times N_e^{1\mathrm{D}} \times N_e^{1\mathrm{D}}$ hexahedra in 3D. The number of unknowns on the trace of one patch now becomes

- $4 \cdot N_e^{1D} \cdot p$ (**triangles** and **quadrilaterals**) and
- $6 \cdot (N_e^{1D})^2 \cdot p^2$ in 3D (**tetrahedra** and **hexahedra**),

which will require

- $\mathcal{O}\left(16(N_e^{1\mathrm{D}})^2 \cdot p^2\right)$ operations per matrix-vector multiplication in 2D and
- $\mathcal{O}\left(36(N_e^{1\mathrm{D}})^4 \cdot p^4\right)$ operations in 3D

This means that the PCG **algorithm in** CG-HDG **case scales one order worse when measured in terms of number of elements along patch face** $\left(\mathcal{O}\left((N_e^{1\mathrm{D}})^4\right)\right)$ **than the standard** HDG **algorithm** $\left(\mathcal{O}\left((N_e^{1\mathrm{D}})^3\right)\right)$.

Remark 1. *Note that the number on the surface of the patch is the same for triangles and quadrilaterals and for tetrahedra and hexahedra, respectively. For a continuous expansion, the number of DOFs on one quadrilateral face of a hexahedron is* $(p + 1)^2$, *and* $2 \cdot \frac{(p+1)(p+2)}{2} - (p + 1) = (p + 1)^2$ *for two triangles covering the same quadrilateral face.*

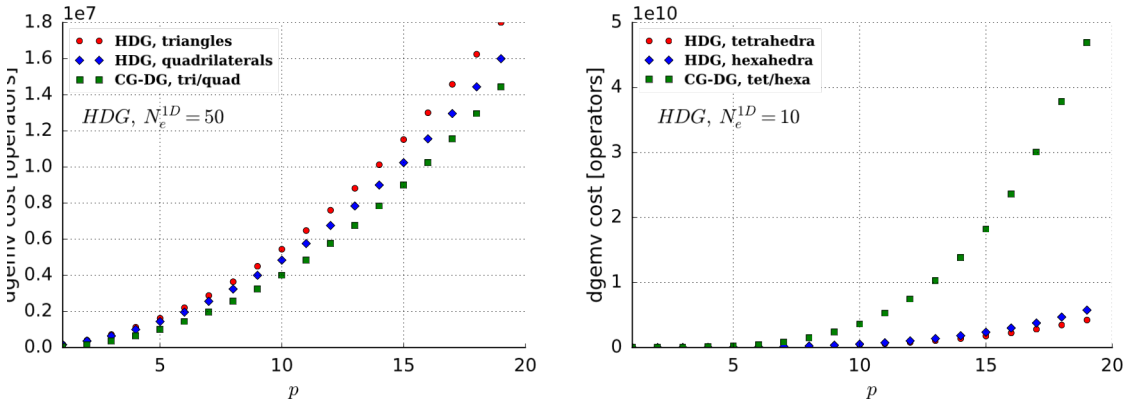


Figure 79: Asymptotic of matrix-vector multiplication measured by operation counts for HDG and combined CG-HDG methods.

The reconstruction of interior degrees of freedom involves the solution of a linear system with the matrix

$$\mathrm{A}^e = \begin{pmatrix} \sum_{l=1}^{N_b^e} \tau^{(e,l)} E_l^e & -D_1^e & -D_2^e \\ (D_1^e)^T & M^e & 0 \\ (D_2^e)^T & 0 & M^e \end{pmatrix}$$

The superscript ^{e} no longer refers to a single element as was the case of HDG. For CG-HDG method, all the blocks in A^e are a result of a continuous Galerkin discretization in the whole partition/patch. Note that the matrix E_l^e involves interaction of boundary and interior modes and for CG-HDG, it can be assembled by looping over all elements adjacent to partition boundary and adding corresponding elemental blocks to the matrix A^e. The sparse matrix A^e is potentially large and its explicit inverse will be dense, i.e. require significant storage.

The expensive interior solve together with increased operation count when inverting the statically condensed system in 3D indicates that the benefit of reduced communication pattern in continuous-discontinuous discretization might be outweighed by extra CPU cost and there is relatively little performance (if any) to be gained by combining the continuous and discontinuous Galerkin discretization into one hybrid solver.

We were of the partially block-diagonal structure of $\boldsymbol{A^e}$ and hoped that an efficient local solver for each mesh partition could be designed. The asymptotic costs of solving the globally coupled system as discussed in this section are too prohibitive, however.

3.7.4.7 Cost in terms of memory requirements

We again assume that the global system is solved by performing multiple PCG iterations, where global matrix-vector multiply is executed in matrix-free fashion.

For each elemental multiplication, the data containing the input matrix, the vector it operates on and a resulting vector must be loaded into processor cache. We discuss amount of data transferred for each method *during one PCG iteration* here.

3.7.4.8 Continuous Galerkin

On triangles, one element contains $3 \cdot p$ trace degrees of freedom (this is irrespective of global continuity of the solution, because we perform the multiplication element-by element and hence whether the DOFs are shared with neighbouring elements or not is irrelevant). The elemental statically condensed matrix has rank $(3 \cdot p)^2$ and the amount of data to move in and out of cache is therefore

$$(3 \cdot p)^2 + 2 \cdot (3 \cdot p) = 9p^2 + 6p$$

The term $2 \cdot (3 \cdot p)$ takes into account two vectors needed for elemental matrix-vector multiplication. This is repeated for each element, and therefore the total number of floating-point values transferred is

$$2 \cdot (N_e^{1D})^2 (9p^2 + 6p)$$

Repeating similar calculation for other element types, we arrive to the following estimates for continuous Galerkin method and different element types:

- **Triangles**: $N_{CG,tri} = 2 \cdot (N_e^{1D})^2 (9p^2 + 6p)$ floating point values
- **Quadrilaterals**: $N_{CG,quad} = (N_e^{1D})^2 (16p^2 + 8p)$ floating point values
- **Tetrahedra:** $\boldsymbol{N_{CG,tet} = 6 \cdot (N_e^{1D})^3 \left(\left(2p(p+1) \right)^2 + 4p(p+1) \right)}$ floating point values
- **Quadrilaterals:** $\boldsymbol{N_{CG,hex} = (N_e^{1D})^3 ((6p^2)^2 + 12p^2)}$ floating point values

3.7.4.9 Discontinuous Galerkin

Data for HDG are very similar with the exception that the trace values are discontinuous, which means that the elemental matrices are slightly bigger:

- **Triangles:** $\boldsymbol{N_{HDG,tri} = 2(N_e^{1D})^2 \left(9(p+1)^2 + 6(p+1) \right)}$ floating point values
- **Quadrilaterals** $\boldsymbol{N_{HDG,quad} = (N_e^{1D})^2 \left(16(p+1)^2 + 8(p+1) \right)}$ floating point values
- **Tetrahedra:** $\boldsymbol{N_{HDG,tet} = 6 \cdot (N_e^{1D})^3 \left(\left(2(p+1)(p+2) \right)^2 + 4(p+1)(p+2) \right)}$ floating point values
- **Quadrilaterals:** $\boldsymbol{N_{HDG,hex} = (N_e^{1D})^3 ((6(p+1)^2)^2 + 12(p+1)^2)}$ floating point values

3.7.4.10 CG-HDG

The hybrid CG-HDG method has a system matrix of rank $(4 \cdot N_e^{1D} \cdot p)$ in two dimensions, which means that the number of floating-point values involved in one matrix-vector multiply will be

$$N_{CG-HDG,2D} = (4 \cdot N_e^{1D} \cdot p)^2 + (8 \cdot N_e^{1D} \cdot p) = 16(N_e^{1D})^2 p^2 + 8N_e^{1D} p \text{ in 2D}$$

and similarly in 3D, where the number of DOFs on patch surface is $6 \cdot (N_e{}^{1\mathrm{D}})^2 \cdot p^2$:

$$N_{CG-HDG,3D} = (6 \cdot (N_e^{1\mathrm{D}})^2 \cdot p^2)^2 + (12 \cdot (N_e^{1\mathrm{D}})^2 \cdot p^2) .$$

3.7.4.11 Continuous Galerkin

The number of interior degrees of freedom in one high-order triangle is $(p+1)(p+2)/2-3p = (p-2)(p-1)/2$ and we suppose that this is the rank of elemental Schur complement which has to be inverted and stored. In the case of continuous Galerkin system, the element-interior matrix, left- and right- hand side vectors hold

$$N_{CG,tri} = ((p-2)(p-1)/2)^2 + 2 \cdot \big((p-2)(p-1)/2\big)$$

This cost has to be multiplied by number of elements present in the mesh. Cost for different element shapes is summarized below

- **Triangles**: $N_{CG,tri} = 2(N_e^{1D})^2 \left[\left((p-2)(p-1)/2\right)^2 + (p-2)(p-1)\right]$ floating point values
- **Quadrilaterals**: $N_{CG,quad} = (N_e^{1D})^2[(p-1)^2 + 2(p-1)]$ floating point values

3.7.4.12 Discontinuous Galerkin

In HDG, each element has its 'own' DOFs not shared with the hybrid variable, hence elemental matrices are again slightly bigger:

- **Triangles**: $N_{HDG,tri} = 2(N_e^{1D})^2 \left[\left((p+1)(p+2)/2\right)^2 + (p+1)(p+2)\right]$ floating point values
- **Quadrilaterals** $N_{HDG,quad} = (N_e^{1D})^2[(p+1)^2 + 2(p+1)]$ floating point values

3.7.4.13 CG-HDG

The 'interior matrix' is sparse, but involves all DOFs of the patch, whose count is approximately $(N_e^{1\mathrm{D}})^2 p^2$. The matrix and corresponding storage would then be

$$N_{CG-HDG,quad} = \left((N_e^{1\mathrm{D}})^2 p^2\right)^2 + 2(N_e^{1\mathrm{D}})^2 p^2,$$

Which is again orders of magnitude worse estimate than for CG and HDG.

4 Conclusion and Future Work

In this deliverable we have reported on the efficiency improvements to co-design applications that stem from both efficient algorithm implementations and system-specific tweaks. There is no silver bullet in making the most efficient use of hardware resources and this will continue to be the case at exascale. What can be done is to build an arsenal of methods for dealing with specific bottlenecks and limitations, and, understand the operating regimes in which to use them to best effect.

For example, we have seen that using the XML library method for Nektar++ to write the results from simulations or checkpoints is very efficient when the file-size is large, on the order of gigabytes, but performs poorly when dealing with small files, on the order of tens of megabytes.

Future work will continue the investigation of the performance of the methods and techniques detailed in this deliverable on emerging hardware, performing design space exploration to ascertain the best parameters settings. Further:

- Application using code generation, such as OpenSBLI, will require new back-ends written, or linking to updated libraries which are aware of new hardware. However, this work will benefit a community larger than that using a single application.

- IO will most likely change, particularly the ratio of IO bandwidth to compute cores will decrease as exascale systems increase core count beyond that seen today. Thus, further work will focus on improving the compression of data on the fly before writing, and on improving the choice of schema used for reading and writing in order operate in the most efficient regime for the given task.

- Energy efficiency will continue to be dominated by changes in system hardware and topology. However, there is a continued requirement to map the operating mode, clock freqneucy and other parameters to understand the best configuration for a particular code-phase, recognizing that the profile for IO differs from that of compute and post-process.

Bibliography

[1] Acharya, T., Tsai, P.: JPEG2000 Standard for Image Compression: Concepts, Algorithms and VLSI Architectures. John Wiley & Sons, New Jersey (2005)
[2] Bruylants, T., Munteanu, A., Schelkens, P.: Wavelet based volumetric medical image compression. Signal Processing: Image Communication 31, 112-133 (2015). doi:10.1016/j.image.2014.12.007
[3] Christopoulos, C., Skodras, A., Ebrahimi, T.: The JPEG2000 still image coding system: An overview. IEEE Transactions on Consumer Electronics 46(4), 1103-1127 (2000). doi:10.1109/30.920468
[4] Jacobs, T., Jammy, S.P., Sandham, N.D.: OpenSBLI: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures. Journal of Computational Science 18, 12-23 (2017). doi:10.1016/j.jocs.2016.11.001
[5] Lindstrom, P.: Fixed-Rate Compressed Floating-Point Arrays. IEEE Transactions on Visualization and Computer Graphics 20(12), 2674-2683 (2014). doi:10.1109/TVCG.2014.2346458
[6] Loddoch, A., Schmalzl, J.: Variable quality compression of fluid dynamical data sets using a 3-D DCT technique. Geochemistry, Geophysics, Geosystems 7(1), 1-13 (2006). doi:10.1029/2005GC001017
[7] Rabbani, M., Joshi, R.: An overview of the JPEG 2000 still image compression standard. Signal Processing: Image Communication 17(1), 3-48 (2002). doi:10.1016/S0923-5965(01)00024-8
[8] Schmalzl, J.: Using standard image compression algorithms to store data from computational fluid dynamics. Computers and Geosciences 29, 1021-1031 (2003). doi:10.1016/S0098-3004(03)00098-0
[9] Selent, B. and Rist, U.: Untersuchungen zur Dynamik und Strukturbildung in Jet-in-Crossflow-Konfigurationen. DFG Report RI 680/25-2 (2016)
[10] Taubman, D., Marcellin, M.: JPEG2000: Image Compression Fundamentals, Standards and Practice. Springer US, New York City (2002)
[11] Wenzel, C., Selent, B., Kloker, M., Rist, U.: DNS of compressible turbulent boundary layers and assessment of data/scaling-law quality. J. of Fluid Mechanics 842, 428-468 (2018)
[12] P. Fischer, N. Miller, and H. Tufo. An overlapping schwarz method for spectral element simulation of three-dimensional incompressible flow. In P. Bjorstad and M. Luskin, editors, Parallel Solution of Partial Differential Equations, volume 120 of The IMA Volumes in Mathematics and its Applications, pages 159–180. Springer New York, 2000.
[13] P. F. Fischer. An overlapping schwarz method for spectral element solution of the incompressible navier stokes equations. Journal of Computational Physics, 133:84–101, May 1997.
[14] P. F. Fisher and J. W. Lottes. Hybrid Schwarz-Multigrid Methods for the Spectral Element Method: Extension to Navier-Stokes, pages 35-49. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
[15] J. W. Lottes and P. F. Fisher. Hybrid multigrid/Schwarz algorithms for the spectral element method. Journal of Scientific Computing, 24(1):45-78, 2005.
[16] Paul F. Fisher. Projection techniques for iterative solution of Ax=b with successive right-hand sides. Computer Methods in Applied Mechanics and Engineering, 163:193-204, 1998.

[17] P. F. Fisher and G. W. Kruse and F. Loth. Spectral element methods for transitional flows in complex geometries. Journal of Scientific Computing, 17:81-98, 2002.
[18] P. Schlatter and S. Stolz and L. Kleiser. LES of transitional flows using the approximate deconvolution model. International Journal of Heat and Fluid Flow, 25:549-558, 2004
[19] P. Schlatter and R. Orlu. Turbulent boundary layers at moderate Reynolds numbers: inflow length and tripping effects. Journal of Fluid Mechanics, 710:5-34, 2012
[20] hypre: A Library of High Performance Preconditioners. Robert D. Falgout, Ulrike Meier Yang. s.l. : Springer Berlin Heidelberg, 2002. International Conference on Computational Science. pp. 632-641.
[21] Cantwell, C.D. & Nielsen, A.S. J Sci Comput (2018). https://doi.org/10.1007/s10915-018-0778-7
[22] C. Cantwell, D. Moxey, A. Comerford, A. Bolis, and G. R. et al., "Nektar++: An open-source spectral/hp element frame- work," Computer Physics Communications, vol. 192, pp. 205– 219, July 2015.
[23] R. Nash, S. Clifford, C. Cantwell, D. Moxey, and S. Sherwin, "Communication and i/0 masking for increasing the performance of nektar++," Edinburgh Parallel Computing Centre, Embedded CSE eCSE02-13, May 2016. [Online]. Available: https://www.archer.ac.uk/community/eCSE/eCSE02-13/eCSE02-13-TechnicalReport.pdf
[24] HDF5, HDF5 User's Guide, release 1.10.0 ed. University of Illinois, USA: The HDF Group, March 2016. [Online]. Available: https://support.hdfgroup.org/HDF5/doc/UG/HDF5_Users_Guide-Responsive%20HTML5/index.html#t=HDF5_Users_Guide%2FHDF5_UG_Title%2FHDF5_UG_Title.htm
[25] Julich Supercomputing Centre (JSC). (2016) Sionlib v1.6.2 - scalable i/o library for parallel access to task-local files. Web site. [Online]. Available: https://apps.fz-juelich.de/jsc/sionlib/docu/index.html
[26] P. E. Vincent, A. M. Plata, A. A. E. Hunt, P. D. Weinberg, and S. J. Sherwin, "Blood flow in the rabbit aortic arch and descending thoracic aorta," Journal of The Royal Society Interface, vol. 8, pp. 1708–1719, May 2011. [Online]. Available: http://rsif.royalsocietypublishing.org/content/8/65/1708
[27] EPCC. (2015) Archer user guide. Web site. [Online]. Available: http://www.archer.ac.uk/documentation/user-guide/
[28] EPCC. (2017) Parallel I/O Performance Benchmarking and Investigation on Multiple HPC Architectures. Web site. [Online]. Available: http://www.archer.ac.uk/documentation/white-papers/parallelIO-benchmarking/ARCHER-Parallel-IO-1.4.pdf
[29] H. Richardson, "benchio performance investigation", Cray EMEA Research Lab, ARCHER CoE.
[30] Nektar++ ioext branch. [Online]. Available: https://gitlab.nektar.info/nektar/nektar/tree/feature/ioext
[31] Ivanov, I., Gong, J., Akhmetova, D., Peng, I. B., Markidis, S., Laure, E., … Fischer, P. (2015). Evaluation of parallel communication models in Nekbone, a Nek5000 mini-application. Proceedings - IEEE International Conference on Cluster Computing, ICCC, 2015–Octob, 760–767.
[32] OpenACC https://openacc.org

[33] Nek5000's OpenACC version https://github.com/Nek5000/Nek5000/tree/openacc

[34] E. Otero, J. Gong, M. Min, P. Fischer, P. Schlatter, and E. Laure, "OpenACC accelerator for the Pn-Pn-2 algorithm in Nek5000", conference abstract in the Proceeding of EASC2018, April 2018, Edinburgh. Extend version has been submitted to a special issue of the Journal of Parallel and Distributed Computing.

[35] Berkeley Lab, CRD. (2018) Empirical Roofline Tool (ERT). Web site. [Online]. Available: https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/software/ert/

[36] J. Kwack, G. Arnold, C. Mendes, G. H. Bauer, 2018. "Roofline Analysis with Cray Performance Analysis Tools (CrayPat) and Roofline-based Performance Projections for a Future Architecture", Proceedings of the Cray User Group Conference 2018.

[37] Terpstra, D., Jagode, H., You, H., Dongarra, J. "Collecting Performance Data with PAPI-C", Tools for High Performance Computing 2009, Springer Berlin / Heidelberg, 3rd Parallel Tools Workshop, Dresden, Germany, pp. 157-173, 2010.

[38] Lusher, D. J., Jammy, S. P., and Sandham, N. D. (2018), Transitional shockwave/boundary-layer interactions in the automatic source-code generation framework

[39] OpenSBLI. In proceedings of the Tenth International Conference on Computational Fluid Dynamics (ICCFD10), Barcelona, Spain, July 9-13.

[40] Mudalige, G.R., Reguly, I.Z. , Jammy, S.P, Jacobs, C.T, Giles, M.B., Sandham, N.D. (submitted), Large-scale Performance of a DSL-based Multi-block Structured-mesh Application for Direct Numerical Simulation, submitted to Journal of Parallel and Distributed Computing

[41] Lusher, D. J., Jammy, S. P., and Sandham, N. D. (2018), Shock-wave/boundary-layer interactions in the automatic source-code generation framework OpenSBLI. Computers and Fluids. 173:17-21

[42] Jammy, S. P., Jacobs, C.T., Lusher, D. J., and Sandham, N. D (2018), Energy consumption of algorithms for solving the compressible Navier-Stokes equations on CPU's, GPU's and KNL's. In Proceedings of the 6th European Conference on Computational Mechanics (ECCM 6) and 7th European Conference on Computational Fluid Dynamics (ECFD 7), 11-15 June 2018, Glasgow, UK.

[43] James DeBonis. Solutions of the Taylor-Green Vortex Problem Using High-Resolution Explicit Finite Difference Methods. In 51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition, Aerospace Sciences Meetings. American Institute of Aeronautics and Astronautics, Jan 2013.

[44] Agullo, E., Giraud, L., Guermouche, A., Roman, J., Zounon, M.: Towards resilient parallel linear krylov solvers: recover-restart strategies. Technical Report RR-8324, INRIA (2013)

[45] Barker, A.D., Bernholdt, D.E., Bland, A.S., Hack, J.J., Hudson, D.L., Rogers, J.H., Straatsma, T.P., Thach, K.G., Vazhkudai, S.S., Wells, J.C., White, J.C.: High performance computing facility operational assessment 2013 oak ridge leadership computing facility. Technical report, OakRidge National Laboratory (2014)

[46] Bland, W., Bosilca, G., Bouteiller, A., Herault, T., Dongarra, J.: A proposal for user-level failure mitigation in the MPI-3 standard. Technical report (2012)

[47] Bland, W., Bouteiller, A., Herault, T., Bosilca, G., Dongarra, J.: Post-failure recovery of MPI communication capability. Int. J. High Perform. Comput. Appl. **27**(3), 244–254 (2013)

[48] Cantwell, C.D., Moxey, D., Comerford, A., Bolis, A., Rocco, G., Mengaldo, G., De Grazia, D., Yakovlev, S., Lombard, J.-E., Ekelschot, D., Jordi, B., Xu, H., Mohamied, Y., Eskilsson, C., Nelson, B., Vos, P., Biotto, C., Kirby, R.M., Sherwin, S.J.: Nektar++: an open-source spectral/hp element framework. Comput. Phys. Commun. **192**, 205–219 (2015)

[49] Cantwell, C.D., Sherwin, S.J., Kirby, R.M., Kelly, P.H.J.: From h to p efficiently: selecting the optimal spectral/hp discretisation in three dimensions. Math. Model. Nat. Phenom. **6**(3), 84–96 (2011)

[50] Cantwell, C.D., Sherwin, S.J., Kirby, R.M., Kelly, P.H.J.: From h to p efficiently: strategy selection for operator evaluation on hexahedral and tetrahedral elements. Comput. Fluids **43**(1), 23–28 (2011)

[51] Cappello, F.: Fault tolerance in petascale/exascale systems: current knowledge, challenges and research opportunities. Int. J. High Perform. Comput. Appl. **23**(3), 212–226 (2009)

[52] Cappello, F., Geist, A., Gropp, W., Kale, S., Kramer, B., Snir, M.: Toward exascale resilience: 2014 update. Supercomput. Front. Innova. **1**(1), 5–28 (2014)

[53] Chen, C., Du, Y., Zuo, K., Fang, J., Yang, C.: Toward fault-tolerant hybrid programming over large-scale heterogeneous clusters via checkpointing/restart optimization. J. Supercomput. (2017). https://doi.org/10.1007/s11227-017-2116-5

[54] Daly, J.: A model for predicting the optimum checkpoint interval for restart dumps. In: Computational Science ICCS 2003, volume 2660 of Lecture Notes in Computer Science, pp. 3–12 (2003)

[55] Di, S., Bouguerra, M.S., Bautista-Gomez, L., Cappello, F.: Optimization of multi-level checkpoint model for large scale HPC applications. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp. 1181–1190. IEEE (2014)

[56] Di Martino, C., Kramer, W., Kalbarczyk, Z., Iyer, R.: Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 HPC application runs. In: 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 25–36. IEEE (2015)

[57] Di Martino, C., Kramer, W., Kalbarczyk, Z., Iyer, R., Moody, A., Bronevetsky, G., Mohror, K., de Supinski, B.R.: Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–11. IEEE (2010)

[58] Elnozahy (Mootaz), E.N., Alvisi, L., Wang, Y.-M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. **34**(3), 375–408 (2002)

[59] Fagg, G.E., Dongarra, J.J.: FT-MPI: fault tolerant MPI, supporting dynamic applications in a dynamic world. In: Recent Advances in Parallel Virtual

Machine and Message Passing Interface, volume 1908 of Lecture Notes in Computer Science, pp. 346–353 (2000)

[60] Gamell, M., Katz, D.S., Teranishi, K., Heroux, M.A., Van der Wijngaart, R.F., Mattson, T.G., Parashar, M.: Evaluating online global recovery with fenix using application-aware in-memory checkpointing techniques. In: 2016 45th International Conference on Parallel Processing Workshops (ICPPW), pp. 346–355 (2016)

[61] Gamell, M., Van der Wijngaart, R.F., Teranishi, K., Parashar, M.: Specification of fenix MPI fault tolerance library version 1.0.1. Technical report (2016)

[62] Hassani, A., Skjellum, A., Brightwell, R.: Design and evaluation of FA-MPI, a transactional resilience scheme for non-blocking MPI. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 750–755. IEEE (2014)

[63] Kang, X., Gleich, D.F., Sameh, A., Grama, A.: Distributed fault tolerant linear system solvers based on erasure coding. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), pp. 2478–2485. IEEE (2017)

[64] Karniadakis, G.E., Israeli, M., Orszag, S.A.: High-order splitting methods for the incompressible Navier–Stokes equations. J. Comput. Phys. **97**(2), 414–443 (1991)

[65] Laguna, I., Richards, D.F., Gamblin, T., Schulz, M., de Supinski, B.R., Mohror, K., Pritchard, H.: Evaluating and extending user-level fault tolerance in MPI applications. Int. J. High Perform. Comput. Appl. **30**(3), 305–319 (2016)

[66] Meneses, E., Ni, X., Zheng, G., Mendes, C.L., Kalé, L.V.: Using migratable objects to enhance fault tolerance schemes in supercomputers. IEEE Trans. Parallel Distrib. Syst. **26**(7), 2061–2074 (2015)

[67] Moxey, D., Cantwell, C.D., Kirby, R.M., Sherwin, S.J.: Optimising the performance of the spectral/hp element method with collective linear algebra operations. Comput. Methods Appl. Mech. Eng. **310**, 628–645 (2016)

[68] Rajachandrasekar, R., Moody, A., Mohror, K., Panda, D.K.: A 1 PB/s file system to checkpoint three million MPI tasks. In: 22nd International Symposium on High-Performance Parallel and Distributed Computing, pp. 143–154. ACM, New York (2013)

[69] Sato, K., Maruyama, N., Mohror, K., de Supinski, B.R., Moody, A., Gamblin, T., Matsuoka, S.: Design, modeling, and evaluation of a non-blocking checkpointing system. In: SC '12 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 1–11. IEEE (2010)

[70] Schroeder, B., Gibson, G.A.: Understanding failures in petascale computers. J. Phys. Conf. Ser. **78**, 012022 (2007)

[71] Snir, M., Wisniewski, R.W., Abraham, J.A., Adve, S.V., Bagchi, S., Balaji, P., Belak, J., Bose, P., Cappello, F., Carlson, B., Chien, A.A., Coteus, P., DeBardeleben, N.A., Diniz, P.C., Engelmann, C., Erez, M., Fazzari, S., Geist, A., Gupta, R., Johnson, F., Krishnamoorthy, S., Leyffer, S., Liberty, D., Mitra, S., Munson, T., Schreiber, R., Stearley, J., Van Hensbergen, E.: Addressing failures in exascalecomputing. Int. J. High Perform. Comput. Appl. **28**(2), 129–173 (2014)

[72] Vogt, D., Giuffrida, C., Bos, H., Tanenbaum, A.S.: Techniques for efficient in-memory checkpointing. In: Proceedings of the 9th Workshop on Hot Topics in Dependable Systems—HotDep '13, pp. 1–5 (2013)
[73] Vos, P.E.J., Sherwin, S.J., Kirby, R.M.: From h to p efficiently: implementing finite and spectral/hp element methods to achieve optimal performance for low-and high-order discretisations. J. Comput. Phys. **229**(13), 5161–5181 (2010)
[74] Zheng, G., Ni, X., Kalé, L.V.: A scalable double in-memory checkpoint and restart scheme towards exascale. In: IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012), pp. 1–6. IEEE (2012)
[75] Zhu, Y., Gleich, D.F., Grama, A.: Erasure coding for fault-oblivious linear system solvers. SIAM J. Sci. Comput. **39**(1), C48–C64 (2017)
[76] Bareford, M. R., "The PAPI MPI Library", 2018, CRESTA (Collaborative Research into Exascale Systemware, Tools & Applications). Web site. [Online]. Available: https://github.com/cresta-eu/papi_mpi_lib
[77] Tinku Acharya and Ping-Sing Tsai. JPEG2000 Standard for Image Compression: Concepts, Algorithms and VLSI Architectures. John Wiley & Sons, Hoboken, New Jersey, 2005.
[78] Emmanuel Agullo, Luc Giraud, Abdou Guermouche, Jean Roman, and Mawussi Zounon. Towards resilient parallel linear krylov solvers: recover-restart strategies. Technical Report RR-8324, INRIA, 2013.
[79] E. Anderson. LAPACK users' guide, volume 9 of Software, environments, tools. Society for Industrial and Applied Mathematics (SIAM 3600 Market Street Floor 6 Philadelphia PA 19104), Philadelphia, Pa, 3rd ed. edition, 1999.
[80] Argonne Leadership Computing Facility (ALCF). Aurora 2021 Early Science Program: Data and Learning Call For Proposals, 2018.
[81] Guillaume Aupy, Anne Benoit, Thomas H_erault, Yves Robert, and Jack Dongarra. Optimal checkpointing period: Time vs. energy. In International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, pages 203{214. Springer, 2013.
[82] Guillaume Aupy, Yves Robert, Fr_ed_eric Vivien, and Dounia Zaidouni. Checkpointing algorithms and fault prediction. Journal of Parallel and Distributed Computing, 74(2):2048{2064, 2014.
[83] Noor Bajunaid and Daniel A. Menasce. Analytic models of checkpointing for concurrent component-based software systems. In Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE '17, pages 245{256, New York, NY, USA, 2017. ACM.
[84] Prasanna Balaprakash, Leonardo A. Bautista Gomez, Mohamed-Slim Bouguerra, Stefan M. Wild, Franck Cappello, and Paul D. Hovland. Analysis of the tradeoffs between energy and run time for multilevel checkpointing. In Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond, editors, High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation, pages 249-263, Cham, 2015. Springer International Publishing.
[85] W. Bangerth and R. Rannacher. Adaptive Finite Element Methods for Diffrential Equations. Birkhauser, Basel, 2002.
[86] Richard Barrett, Michael W Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and

Henk Van der Vorst. Templates for the solution of linear systems: building blocks for iterative methods, volume 43. Siam, 1994.
[87] Leonardo Bautista-Gomez, Thomas Ropars, Naoya Maruyama, Franck Cappello, and Satoshi Matsuoka. Hierarchical clustering strategies for fault tolerance in large scale hpc systems. In IEEE Cluster 2012, 2012.
[88] Leonardo Bautista-Gomez and Karol Sierocinski. FTI Repository, 2018.
[89] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. Fti: high performance fault tolerance interface for hybrid systems. In High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for, pages 1{12. IEEE, 2011.
[90] Anne Benoit, Aur_elien Cavelan, Valentin Le F_evre, and Yves Robert. Optimal checkpointing period with replicated execution on heterogeneous platforms. In Proceedings of the 2017 Workshop on Fault-Tolerance for HPC at Extreme Scale, FTXS '17, pages 9-16, New York, NY, USA, 2017. ACM.
[91] Anne Benoit, Aur_elien Cavelan, Valentin Le F_evre, Yves Robert, and Hongyang Sun. Towards optimal multi-level checkpointing. IEEE Transactions on Computers, 66(7):1212-1226, 2017.
[92] L. Blackford, J. Choi, A. Cleary, J. Demmel, Inderjit S. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK users' guide. Software, environments, tools. Soc. for Industrial and Applied Mathematics, Philadelphia, Pa., 1997.
[93] Wesley Bland, George Bosilca, Aurelien Bouteiller, Thomas Herault, and Jack Dongarra. A proposal for user-level failure mitigation in the MPI-3 standard. Technical report, 2 2012.
[94] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. Post-failure recovery of MPI communication capability. The International Journal of High Performance Computing Applications, 27(3):244{254, 1 2013.
[95] Wesley Bland, Aurelien Bouteiller, Thomas Herault, Joshua Hursey, George Bosilca, and Jack J Dongarra. An evaluation of user-level failure mitigation support in mpi. Computing, 95(12):1171{1184, 2013.
[96] George Bosilca and Aurelien Bouteiller. ULFM MPI Repository, 2018.
[97] Marin Bougeret, Henri Casanova, Yves Robert, Fr_ed_eric Vivien, and Dounia Zaidouni. Using group replication for resilience on exascale systems. The International Journal of High Performance Computing Applications, 28(2):210{224, 2014.
[98] A. Busse and N. D. Sandham. Parametric forcing approach to rough-wall turbulent channel flow. Journal of Fluid Mechanics, 712:169202, 2012.
[99] Xiao-Chuan Cai and Marcus Sarkis. A restricted additive schwarz preconditioner for general sparse linear systems. SIAM J. Sci. Comput., 21(2):792{797, September 1999.
[100] C.D. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D. De Grazia, S. Yakovlev, J.-E. Lombard, D. Ekelschot, B. Jordi, H. Xu, Y. Mohamied, C. Eskilsson, B. Nelson, P. Vos, C. Biotto, R.M. Kirby, and S.J. Sherwin. Nektar++: An open-source spectral/hp element framework. Computer Physics Communications, 192:205-219, 2015.
[101] Chris D. Cantwell, David Moxey, Andrew Comerford, Alessandro Bolis, Gabriele Rocco, Gianmarco Mengaldo, Daniele De Grazia, Sergey Yakovlev,

Jean-Eloi. Lombard, Dirk Ekelschot, Bastien Jordi, Hui Xu, Yumnah Mohamied, Claes Eskilsson, Blake Nelson, Peter Vos, Cristian Biotto, Robert M. Kirby, and Spencer J. Sherwin. Nektar++: An open-source spectral/ hp element framework. Computer Physics Communications, 192:205-219, 7 2015.

[102] Chris D Cantwell and Allan S Nielsen. A minimally intrusive low-memory approach to resilience for existing transient solvers. Journal of Scientific Computing, pages 1-17,2018.

[103] Jiajun Cao, Kapil Arya, Rohan Garg, Shawn Matott, Dhabaleswar K Panda, Hari Subramoni, Jerome Vienne, and Gene Cooperman. System-level scalable checkpointrestart for petascale computing. In Parallel and Distributed Systems (ICPADS), 2016 IEEE 22nd International Conference on, pages 932{941. IEEE, 2016.

[104] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. Supercomputing Frontiers and Innovations, 1(1), 1 2014.

[105] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. Supercomputing frontiers and innovations, 1(1):5-28, 2014.

[106] Henri Casanova, Yves Robert, Frederic Vivien, and Dounia Zaidouni. On the impact of process replication on executions of large-scale parallel applications with coordinated checkpointing. Future Generation Computer Systems, 51:7{19, 2015.

[107] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. ACM Transactions on Computer Systems (TOCS), 3(1):63-75, 1985.

[108] Zizhong Chen and Jack J Dongarra. Condition numbers of gaussian random matrices. SIAM Journal on Matrix Analysis and Applications, 27(3):603-620, 2005.

[109] Ivan Cores, Gabriel Rodriguez, Patricia Gonzalez, Roberto R Osorio, et al. Improving scalability of application-level checkpoint-recovery by reducing checkpoint sizes. New Generation Computing, 31(3):163-185, 2013.

[110] Zongduo Dai and Yufeng Zhang. Partition, construction, and enumeration of m-p invertible matrices over infinite fields. Finite Fields and Their Applications, 7(3):428-440, 2001.

[111] John T Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. Future generation computer systems, 22(3):303-312, 2006.

[112] D. Dauwe, R. Jhaveri, S. Pasricha, A. A. Maciejewski, and H. J. Siegel. Optimizing checkpoint intervals for reduced energy use in exascale systems. In 2017 Eighth International Green and Sustainable Computing Conference (IGSC), pages 1-8, Oct 2017.

[113] David David Taubman and Michael Marcellin. JPEG2000: Image Compression Fundamentals, Standards and Practice. Springer US, New York City, 2002.

[114] M. O. Deville, P. F. Fischer, and E. H. Mund. High-Order Methods for Incompressible Fluid Flow. Cambridge University Press, 2002.

[115] Sheng Di, Leonardo Bautista-Gomez, and Franck Cappello. Optimization of a multilevel checkpoint model with uncertain execution scales. In Proceedings of the International Conference for High Performance

Computing, Networking, Storage and Analysis, SC '14, pages 907-918, Piscataway, NJ, USA, 2014. IEEE Press.
[116] Sheng Di, Mohamed Slim Bouguerra, Leonardo Bautista-Gomez, and Franck Cappello. Optimization of multi-level checkpoint model for large scale hpc applications. In Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, pages 1181-1190. IEEE, 2014.
[117] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P. Jouppi. Hybrid checkpointing using emerging nonvolatile memories for future exascale systems. ACM Trans. Archit. Code Optim., 8(2):6:1-6:29, June 2011.
[118] Jack Dongarra, Thomas Herault, and Yves Robert. Fault tolerance techniques for high-performance computing. In Fault-Tolerance Techniques for High-Performance Computing, pages 3-85. Springer, 2015.
[119] Alan Edelman. Eigenvalues and condition numbers of random matrices. SIAM Journal on Matrix Analysis and Applications, 9(4):543-560, 1988.
[120] Evridiki Efstathiou and Martin J. Gander. Why restricted additive schwarz converges faster than additive schwarz. BIT Numerical Mathematics, 43(5):945-959, 2003.
[121] Lars Elden. Matrix Methods in Data Mining and Pattern Recognition (Fundamentals of Algorithms). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2007.
[122] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys (CSUR), 34(3):375-408, 2002.
[123] P. Fischer, N. Miller, and H. Tufo. An overlapping schwarz method for spectral element simulation of three-dimensional incompressible flow. In P. Bjorstad and M. Luskin, editors, Parallel Solution of Partial Differential Equations, volume 120 of The IMA Volumes in Mathematics and its Applications, pages 159-180. Springer New York, 2000.
[124] P. Fischer and J. Mullen. Filter-based stabilization of spectral element methods. Academie des Sciences Paris Comptes Rendus Serie Sciences Mathematiques, 332:265-270, February 2001.
[125] Paul F. Fischer. An overlapping schwarz method for spectral element solution of the incompressible navier stokes equations. Journal of Computational Physics, 133:84-101, May 1997.
[126] Paul F. Fischer, Gerald W. Kruse, and Francis Loth. Spectral element methods for transitional flows in complex geometries. J. Sci. Comput., 17(1-4):81{98, December 2002.
[127] Paul F. Fischer and James W. Lottes. Hybrid Schwarz-Multigrid Methods for the Spectral Element Method: Extensions to Navier-Stokes, pages 35{49. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
[128] John D Fulton. Generalized inverses of matrices over a _nite _eld. Discrete mathematics, 21(1):23-29, 1978.
[129] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar. Exploring automatic, online failure recovery for scientific applications at extreme scales. In SC14: International Conference for High Performance Computing, Networking, Storage and Analysis, pages 895-906, Nov 2014.
[130] Marc Gamell, Daniel S Katz, Keita Teranishi, Michael A Heroux, Rob F Van der Wijngaart, Timothy G Mattson, and Manish Parashar. Evaluating online global recovery with fenix using application-aware in-memory checkpointing

techniques. In 2016 45th International Conference on Parallel Processing Workshops (ICPPW), pages 346-355. IEEE, 2016.

[131] M. N. Gamito and M. Salles Dias. Lossless coding of oating point data with JPEG 2000 Part 10. In A. G. Tescher, editor, Applications of Digital Image Processing XXVII, volume 5558, pages 276-287, November 2004.

[132] Leonardo Arturo Bautista Gomez, Naoya Maruyama, Franck Cappello, and Satoshi Matsuoka. Distributed diskless checkpoint for large scale systems. In Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID '10, pages 63-72, Washington, DC, USA, 2010. IEEE Computer Society.

[133] Leonardo Bautista Gomez, Bogdan Nicolae, Naoya Maruyama, Franck Cappello, and Satoshi Matsuoka. Scalable reed-solomon-based reliable local storage for hpc applications on iaas clouds. In Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis, editors, Euro-Par 2012 Parallel Processing, pages 313{324, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[134] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. Failures in large scale systems: Long-term measurement, analysis, and implications. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17, pages 44:1-44:12, New York, NY, USA, 2017. ACM.

[135] Ralf Hartmann. Adjoint consistency analysis of discontinuous Galerkin discretizations. SIAM Journal on Numerical Analysis, 45(6):2671-2696, 2007.

[136] Henri Bruno Razandradina, Paul Auguste Randriamitantsoa, and Nicolas Raft Razandrakoto. Image compression with svd: A new quality metric based on energy ratio. CoRR, abs/1701.06183, 2016.

[137] Cheng Huang, Minghua Chen, and Jin Li. Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems. ACM Transactions on Storage (TOS), 9(1):3, 2013.

[138] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, Sergey Yekhanin, et al. Erasure coding in windows azure storage. 2012.

[139] C. T. Jacobs, S. P. Jammy, and N. D. Sandham. OpenSBLI: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures. Journal of Computational Science, 18:12-23, 2017.

[140] C. T. Jacobs, N. D. Sandham, and N. De Tullio. An error indicator for finite difference methods using spectral techniques with application to aerofoil simulation. In Abstracts of the Parallel CFD (ParCFD) 2017 conference, Glasgow, Scotland, 15-17 May 2017, 2017.

[141] Christian T. Jacobs, Satya P. Jammy, and Neil D. Sandham. Opensbli: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures. Journal of Computational Science, 18:12-23, 2017.

[142] Christian T. Jacobs, Markus Zauner, Nicola De Tullio, Satya P. Jammy, David J. Lusher, and Neil D. Sandham. An error indicator for finite difference methods using spectral techniques with application to aerofoil simulation. Computers and Fluids, 168:67-72, 2018.

[143] Claes Johnson and Peter Hansbo. Adaptive finite element methods in computational mechanics. Computer Methods in Applied Mechanics and Engineering, 101(1):143-181, 1992.
[144] William M. Jones, John T. Daly, and Nathan DeBardeleben. Application monitoring and checkpointing in hpc: Looking towards exascale systems. In Proceedings of the 50th Annual Southeast Regional Conference, ACM-SE '12, pages 262{267, New York, NY, USA, 2012. ACM.
[145] Gerald W. Kruse. Parallel Nonconforming Spectral Element Solution of the Incompressible Navier-Stokes Equations in Three Dimensions. PhD thesis, Providence, RI, USA, 1997. UMI Order No. GAX97-38573.
[146] Julien Langou, Zizhong Chen, Jack J Dongarra, and George Bosilca. Disaster survival guide in petascale computing. In Petascale Computing: Algorithms and Applications, pages 263-288. Chapman and Hall/CRC, 2007.
[147] Jurij Leskovec, Anand Rajaraman, and Je_rey D. Ullman. Mining of massive datasets. Cambridge University Press, Cambridge, second edition edition, 2014.
[148] P. Lindstrom. Fixed-rate compressed floating-point arrays. IEEE Transactions on Visualization and Computer Graphics, 20(12):2674-2683, Dec 2014.
[149] Alexander Loddoch and Jrg Schmalzl. Variable quality compression of uid dynamical data sets using a 3-d dct technique. Geochemistry, Geophysics, Geosystems.
[150] James W. Lottes and Paul F. Fischer. Hybrid multigrid/schwarz algorithms for the spectral element method. Journal of Scienti_c Computing, 24(1):45-78, 2005.
[151] M Manasse, C Thekkath, and A Silverberg. A reed-solomon code for disk storage, and efficient recovery computations for erasure-coded disk storage.
[152] Catherine Mavriplis. A posteriori error estimators for adaptive spectral element techniques. In Peter Wesseling, editor, Notes on Numerical Fluid Mechanics, pages 333-342, 1990.
[153] Kathryn Mohror, Adam Moody, Greg Bronevetsky, and Bronis R de Supinski. Detailed modeling and evaluation of a scalable multilevel checkpointing system. IEEE Transactions on Parallel and Distributed Systems, 25(9):2255-2263, 2014.
[154] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1-11. IEEE Computer Society, 2010.
[155] Nichamon Naksinehaboon, Yudan Liu, Chokchai Leangsuksun, Raja Nassar, Mihaela Paun, Stephen L Scott, et al. Reliability-aware approach: An incremental checkpoint/restart model in hpc environments. In 2008 8th International Symposium on Cluster Computing and the Grid (CCGRID, pages 783-788. IEEE, 2008.
[156] Prabal Negi, Philipp Schlatter, and Dan Henningson. A re-examination of filter-based stabilization for spectral-element methods. Technical report, KTH, Stability, Transition and Control, 2017. QC 20171121.
[157] Allan Svejstrup Nielsen and Chris Cantwell. Llama Repository, 2018.

[158] Dimitris S Papailiopoulos and Alexandros G Dimakis. Locally repairable codes. IEEE Transactions on Information Theory, 60(10):5843-5855, 2014.
[159] James S Plank. Improving the performance of coordinated checkpointers on networks of workstations using raid techniques. In srds, page 76. IEEE, 1996.
[160] James S Plank, KM Greenan, EL Miller, and WB Houston. Gf-complete: A comprehensive open source library for galois field arithmetic. University of Tennessee, Tech. Rep. UT-CS-13-703, 2013.
[161] James S Plank, Kai Li, and Michael A Puening. Diskless checkpointing. IEEE Transactions on Parallel and Distributed Systems, 9(10):972-986, 1998.
[162] James S Plank, Scott Simmerman, and Catherine D Schuman. Jerasure: A library in c/c++ facilitating erasure coding for storage applications-version 1.2. University of Tennessee, Tech. Rep. CS-08-627, 23, 2008.
[163] Majid Rabbani and Rajan Joshi. An overview of the fJPEGg 2000 still image compression standard. Signal Processing: Image Communication, 17(1):3 { 48, 2002. fJPEGg 2000.
[164] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. Journal of the society for industrial and applied mathematics, 8(2):300-304, 1960.
[165] Jose Carlos Sancho, Fabrizio Petrini, Greg Johnson, and Eitan Frachtenberg. On the feasibility of incremental checkpointing for scienti_c computing. In Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, page 58. IEEE, 2004.
[166] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. Xoring elephants: Novel erasure codes for big data. In Proceedings of the VLDB Endowment, volume 6, pages 325-336. VLDB Endowment, 2013.
[167] Philipp Schlatter, Stefen Stolz, and Leonhard Kleiser. Les of transitional flows using the approximate deconvolution model. International Journal of Heat and Fluid Flow, 25(3):549-558, 2004. Turbulence and Shear Flow Phenomena (TSFP-3).
[168] Bianca Schroeder and Garth A Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you? In FAST, volume 7, pages 1-16, 2007.
[169] Bianca Schroeder and Garth A. Gibson. Understanding failures in petascale computers. Journal of Physics: Conference Series, 78:012022, 7 2007.
[170] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash reliability in production: The expected and the unexpected. In Proceedings of the 14th USENIX Conference on File and Storage Technologies, 2016.
[171] Faisal Shahzad, Jonas Thies, Moritz Kreutzer, Thomas Zeiser, Georg Hager, and Gerhard Wellein. CRAFT: A library for easier application-level checkpoint/restart and automatic fault tolerance. CoRR, abs/1708.02030, 2017.
[172] Arman Shehabi, Sarah Josephine Smith, Dale A. Sartor, Richard E. Brown, Magnus Herrlin, Jonathan G. Koomey, Eric R. Masanet, Nathaniel Horner, In^es Lima Azevedo, and William Lintner. United states data center energy usage report. Technical report, 06/2016 2016.

[173] Luis Moura Silva and Joao Gabriel Silva. An experimental study about diskless checkpointing. In Euromicro Conference, 1998. Proceedings. 24th, volume 1, pages 395-402. IEEE, 1998.
[174] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A DeBardeleben, Pedro C Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Ley_er, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in exascale computing. The International Journal of High Performance Computing Applications, 28(2):129-173, 1, 2014.
[175] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, et al. Addressing failures in exascale computing. The International Journal of High Performance Computing Applications, 28(2):129-173, 2014.
[176] O. Subasi, G. Kestor, and S. Krishnamoorthy. Toward a general theory of optimal checkpoint placement. In 2017 IEEE International Conference on Cluster Computing (CLUSTER), pages 464-474, Sept 2017.
[177] Xiongchao Tang, Jidong Zhai, Bowen Yu, Wenguang Chen, and Weimin Zheng. Selfcheckpoint: An in-memory checkpoint method using less space and its practice on fault-tolerant hpl. In ACM SIGPLAN Notices, volume 52, pages 401-413. ACM, 2017.
[178] Keita Teranishi and Marc Gamell. Fenix Repository, 2018.
[179] The HDF Group. Hdf5 user's guide: Release 1.11.x. 2015.
[180] H.M Tufo and P.F Fischer. Fast parallel direct solvers for coarse grid problems. J. Parallel Distrib. Comput., 61(2):151-177, February 2001.
[181] Bryan E. Usevitch. Jpeg2000 compatible lossless coding of oating-point data. Eurasip Journal on Image and Video Processing, 2007:1-8, 3 2007.
[182] Nitin H Vaidya. A case for two-level distributed recovery schemes. In ACM SIGMETRICS Performance Evaluation Review, volume 23, pages 64-73. ACM, 1995.
[183] Allan Weber. The USC-SIPI Image Database, 2018.
[184] Stephen B Wicker and Vijay K Bhargava. An introduction to reed-solomon codes.
[185] Chuan-Kun Wu and Ed Dawson. Existence of generalized inverse of linear transformations over finite fields. Finite Fields and Their Applications, 4(4):307-315, 1998.
[186] John W Young. A first order approximation to the optimum checkpoint interval. Communications of the ACM, 17(9):530-531, 1974.
[187] Gengbin Zheng, Xiang Ni, and Laxmikant V Kal_e. A scalable double in-memory checkpoint and restart scheme towards exascale. In Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on, pages 1-6. IEEE, 2012.
[188] Sherwin, Spencer J., and Mario Casarin. "Low-Energy Basis Preconditioning for Elliptic Substructured Solvers Based on Unstructured Spectral/hp Element Discretization." Journal of Computational Physics 171, no. 1 (2001): 394-417.

[189] Balay, S., Abhyankar, S., Adams, M., Brown, J., Brune, P., Buschelman, K., Dalcin, L.D., Eijkhout, V., Gropp, W., Kaushik, D. and Knepley, M., 2017. Petsc users manual revision 3.8 (No. ANL-95/11 Rev 3.8). Argonne National Lab.(ANL), Argonne, IL (United States)

[190] Douglas N Arnold, Franco Brezzi, Bernardo Cockburn, and L Donatella Marini. Unified analysis of Discontinuous Galerkin methods for elliptic problems. *SIAM journal on numerical analysis*, 39(5):1749–1779, 2002.

[191] Bernardo Cockburn, Jayadeep Gopalakrishnan, and Raytcho Lazarov. Unified Hybridization of Discontinuous Galerkin, Mixed, and Continuous Galerkin Methods for Second Order Elliptic Problems. *SIAM Journal on Numerical Analysis*, 47(2):1319–1365, 2009.

[192] Robert M. Kirby, Spencer J. Sherwin, and Bernardo Cockburn. To CG or to HDG: A comparative study. *Journal of Scientific Computing*, 51(1):183–212, 2011.

[193] Sergey Yakovlev, David Moxey, Robert M. Kirby, and Spencer J. Sherwin. To CG or to HDG: A comparative study in 3D. *Journal of Scientific Computing*, 67(1):192–220, 2016.