European Commission

Horizon 2020
European Union funding
for Research & Innovation

E-Infrastructures
H2020-EINFRA-2015-1

EINFRA-5-2015: Centres of Excellence for computing applications

EoCoE

Energy oriented Center of Excellence for computing applications

Grant Agreement Number: EINFRA-676629

D1.12 - M36
Applied research activities

Project and Deliverable Information Sheet

EoCoE	Project Ref:	EINFRA-676629
	Project Title:	Energy oriented Centre of Excellence
	Project Web Site:	http://www.eocoe.eu
	Deliverable ID:	D1.12 - M36
	Lead Beneficiary:	Juelich JSC
	Contact:	Paul Gibbon
	Contact e-mail:	p.gibbon@fz-juelich.de
	Deliverable Nature:	Report
	Dissemination Level:	PU*
	Contractual Date of Delivery:	M36 30/09/2018
	Actual Date of Delivery:	30/09/2018
	EC Project Officer:	Carlos Morais-Pires

* - The dissemination level are indicated as follows: PU – Public, CO – Confidential, only for members of the consortium (including the Commission Services) CL – Classified, as referred to in Commission Decision 2991/844/EC.

Document Control Sheet

Document	Title:	Applied research activities
	ID:	D1.12 - M36
	Available at:	http://www.eocoe.eu
	Software tool:	LaTeX
Authorship	Written by:	Matthieu Haefele (MdlS), Stéphane Lanteri (Inria), Sebastian Lührs (Juelich), Pasqua D'Ambra (CNR), Corentin Roussel (MdlS), Alexis Gobé (Inria), Giorgio Giorgiani (CEA), Julien Bigot (MdlS)
	Contributors:	Wolfgang Frings (JUELICH), Agostino Funel (ENEA), Fiorenzo Ambrosino (ENEA), Guido Guarnieri (ENEA), Maciej Brzezniak (PSNC), Krzysztof Wadowka (PSNC), Karol Sierocinski (PSNC), Tomasz Paluszkiewicz (PSNC), Salvatore Filippone (Cranfield University), Daniela di Serafino (U. Campania), Leonardo Bautista (BSC), Kai Keller (BSC), Alexs Gobé (Inria), Jonathan Viquerat (Inria), Patrick Tamain (CEA)
	Reviewed by:	Haefele (MdlS), Gibbon (JSC), PEC members

Contents

1. Document release note

This document is the final report on software technology improvement. This document and contain the final status of all activities that have taken place in in the applied research context.

2. Motivation

From the outset, the EoCoE project was equipped with a diverse set of HPC expertise in WP1 designed to tackle a variety of possible performance bottlenecks in the applications from the four domain pillars. These range from state-of-the-art computer science tools for performance analysis, parallel IO etc. . . , to advanced linear algebra and other applied mathematics methods. This permits a layered approach to application tuning, starting from initial blind analysis to identify problematic code portions, then subsequently delving deeper to undertake complete refactoring of critical, compute-intensive routines. The key feature of EoCoE has been the close interaction between WP1 and the application domains WP2-WP5, enabling real-world energy applications to effectively exploit the existing European computing infrastructure and better equip them for future hardware advances. Ultimately we expect this work to expedite advances in simulations of low-carbon energy systems and technology.

This deliverable gathers the status of the long-term activities conducted within the project. By "long-term" we mean that EoCoE contributes to applied research whose results will likely have an impact well beyond the end of the project. This includes, for example, activities in advanced applied mathematics involving substantial personnel resources (up to 24PM), allowing more radical reworking of core numerical schemes in a given scientific application, together with comprehensive validation using real test cases. Another important aspect is the development of new generic software packages and libraries, which also consumes time and effort to ensure an impact within the HPC community beyond the immediate scope of the EoCoE.

3. Tokam3X

Contributors	Giorgio Giorgiani (CEA Cadarache), Patrick Tamain (CEA Cadarache)

The main goal of this research is to improve the numerical efficiency of the code TOKAM-3X. This task is divided in two main parts:

- implementation of scalable linear solvers for the inversion of the vorticity operator;
- investigation of advanced numerical schemes for plasma simulations based on non-aligned discretizations.

The methodology and results obtained for the two tasks are detailed next.

Scalable linear solvers in TOKAM-3X

Introduction

The vorticity operator describes the evolution of the electric potential in the machine. Due to the fast dynamic associated to the electric potential, the vorticity operator is treated implicitly, giving rise to a 3D Poisson-type equation with strongly anisotropic coefficients in the parallel and perpendicular directions with respect to the magnetic field. The spatial discretization is obtained with a second-order finite difference scheme specifically designed for anisotropic problems [GYKL05], and produces a linear system which solution represents the value of the electric potential in each point of the 3D plasma domain.

For physically interesting parameters and sufficiently fine meshes, the resulting linear system is typically characterized by high condition numbers ($> 10^{10}$). This is due to essentially three facts:

- strong anisotropy: the ratio between the parallel and the perpendicular diffusion in the vorticity operator is usually considered of the order 10^5;
- non alignment with the computational mesh: due to the shape of the magnetic field lines, the computational grid is aligned along the magnetic lines in the poloidal plane, but in the toroidal direction a small non-alignment is typically present (pitch angle);
- Bohm boundary condition: the plasma wall interaction is described by the so-called Bohm boundary condition for the electric potential, which is translated mathematically by a Robin-type condition where the ratio between the Neumann part and the Dirichlet part has the magnitude of the anisotropy ratio.

The approach used so far to invert the linear system has been to use a direct solver. In particular, the parallel library PastiX, developed by INRIA Bordeaux, has been used for this task. However, the scaling laws of performing a Gaussian elimination on a discretized 3D domain ($\mathcal{O}(n^{4/3})$ fill-in scaling and $\mathcal{O}(n^2)$ flop scaling), limits on the one hand the attainable grid resolution, and on the other obliges to make an isothermal hypothesis of the plasma (that allows to perform the LU factorization of the matrix just once at the beginning of the time iterations).

In order to overcome these limitations, the direct solver should be replaced by an iterative solver. Due to the high-condition number of the matrix, dedicated solution strategies are investigated.

Solution considered and state of the art

Three solutions were considered so far:

- **Preconditioned GMRES**: this solution relies on the implementation of a parallel GMRES iterative solver in the code TOKAM-3X. The key ingredient is the identification of a preconditioning technique, based on a physical insight of the problem, that allows to speed-up the convergence of the iterative solution. A reduced Matlab model was used to perform a parametric study of different preconditioners, that allowed to identify a suitable candidate for the introduction in the TOKAM-3X code, see Figure 1. In the following the proposed preconditioner will be referred as "$\mathcal{P}_9$ preconditioner".
  Two solutions were considered to introduce the parallel GMRES scheme in TOKAM3X:
  1. Via the "iterative refinement" procedure of PastiX. This solution was put on hold due to a minor interface problem, that will be solved in the next release of PastiX (due before the end of 2018).

  2. An "in house solution": a distributed GMRES scheme is directly implemented in TOKAM3X. At the courrent stage the implementation is not fully optimized, but preliminary results are encouraging (shown next).

- **AGMG solver**: AGMG implements an aggregation-based algebraic multi-grid method [NN15]. It is developed by the Université Libre de Bruxelles, which collaboration with the IRFM is brought in by the EoCoE network. After some preliminary tests performed on matrices saved from TOKAM-3X executions, the solver was implemented in the code and tested in a serial implementation. Results proved that AGMG could be a suitable candidate to replace PastiX for performing non-isothermal computations (hence, a new matrix is computed at each time iteration), and could also allow to attain finer grids than the one computed nowadays. However, the performance depends on the magnetic field geometry and needs to be further investigated. A distributed version of AGMG for TOKAM3X is still not available.

- **MaPHyS solver**: the EoCoE collaborative network allowed to test another iterative solver, MaPHyS, developed by INRIA Bordeaux [Nak15]. MaPHyS is a hybrid direct/iterative solver based on domain decomposition. Prelimiary results based on saved TOKAM-3X matrices showed interesting results, but so far less promising then the ones obtained with AGMG. The implementation in TOKAM-3X is in process.

Results

In the following are presented the performance of the *in house* GMRES solver with the proposed $\mathcal{P}_9$ preconditioner (in the following this solver will be referred simply as GMRES), and AGMG. The tolerance considered for the convergence is set to 10^{-8} in all the numerical tests. The performance of the iterative solvers are compared to PastiX, which is taken as reference. The number of iterations and time to solve are shown as functions of the number of unknowns in the mesh. A set of mesh sizes is considered, with numbero of points $8i \times 64i \times 8i$ for $i = 1, \ldots, i_{max}$ (respectively number of points in the r, θ and ϕ direction). The geometry considered so far is the circular geometry with an infinitely small limiter. No electron inertia was considered.

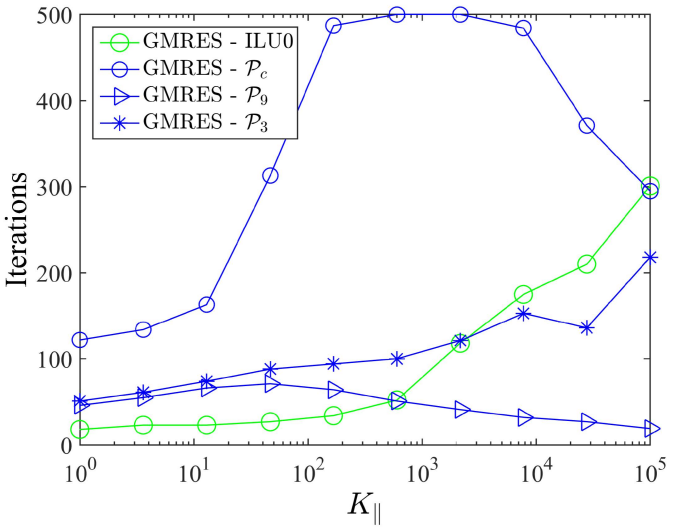


Figure 1: GMRES study: iterations to converge with different preconditioners.

In figure 2, the iterations to convergence are depticted as a function of the number of unkonws. Two different values of the total angle of the torus are considered, 90° and 180°. The total angle of the torus determines the curvature coefficients in the TOKAM3X code, and in turn it has an influence on the number of iterations to attain the desired convergence tolerance. Figure 2 shows that both solvers are able to find a solution with the specified tolerance, with a tendency of increasing the number of iterations with the mesh size. This fact is principally due to the increasing condition number of the linear system when the mesh size is increased. AGMG is more sensible to the variation of torus angle while GMRES seems to be less affected.

Since the definition of the residual is different in the two solvers, it is interesting to plot the error between the solution provided by the iterative solvers and the solution provided by PastiX. The error is shown in figure 3. AGMG provides a smaller error, but the tendency is similar for the two solvers.

In the following some timing tests are shown: all the computations are performed on the CCAMU cluster, consisting of Dell C6420 nodes (2 processors Intel Xeon Gold 6142, 192GB RAM per node, 16 cores @ 2.60GHz for each processor). In all the timing test, the parameter $Lphi = 180°$ is chosen. In figure 4 is depicted the time to solve for the various solvers, for a computation on a single node and full OpenMP parallelization. For PastiX, the time to obtain the solution is the sum of the time needed to perform the LU decomposition and the time for the actual computation of the solution once the LU decomposition is available. Hence, in figure 4, "time solve PastiX" refers to the latter. The equivalent *short cycling* times are also shown. These solve times are computed considering that the vorticity matrix is updated only once each 10 or 50 time steps, hence the *short cycling* time t_{SC} is computed as

$$t_{\mathrm{SC}} = t_{solve} + \frac{1}{N_{\mathrm{SC}}} t_{\mathrm{LU}},$$

where t_{LU} is the time to obtain the LU decomposition, t_{solve} refers to the time to compute the solution once the LU decomposition is available and N_{SC} is the number of time steps

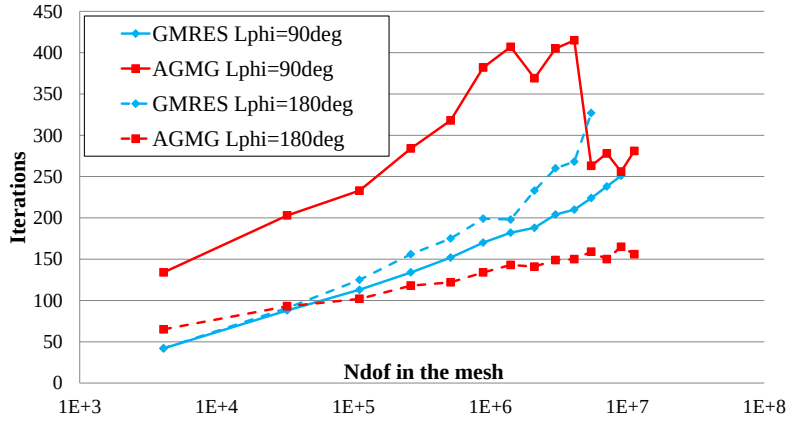


Figure 2: Solvers comparison: iterations to achieve convergence (with a tolerance of $10^{-}8$) for GMRES and AGMG.

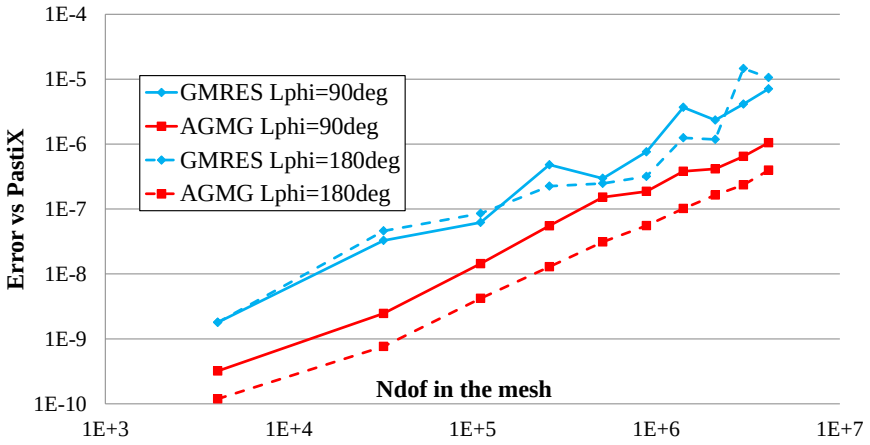


Figure 3: Solvers comparison: error in the final solution with respect to the reference solution (ie the solution obtained with PastiX), for GMRES and AGMG. The tolerance used for the convergence criterion is 10^{-8} in all the computations.

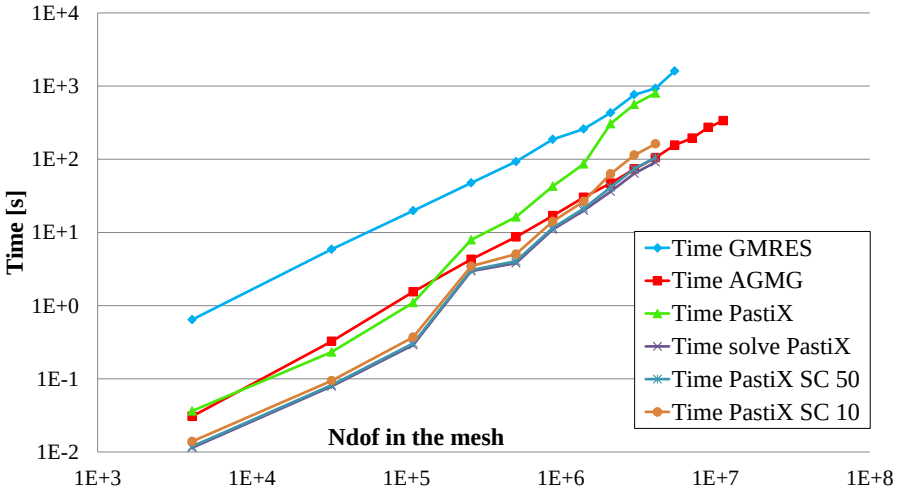


Figure 4: Solvers comparison on a single node on the CCAMU cluster with 32 cpus: solve time vs number of dof in the mesh for GMRES, AGMG, PastiX and PastiX "only solve". The equivalent time to solve with Pastix using short cycling with 50 and 10 iterations between matrix updates is also displayed.

between two LU decompositions.

The results show that AGMG outperforms PastiX even with no distributed computations. AGMG performs better than PastiX already for fairly small meshes (in this case from the computation with dimensions $32 \times 256 \times 32$), if an update of the mesh at each time iteration is considered. For the largest case for which a PastiX solve is possible on a single node (in this case $80 \times 640 \times 80$) AGMG is equivalent to a PastiX computation with 50 iterations between matrix update. Despite a number of iterations almost equivalent to AGMG, the GMRES solver is much slower, indicating that the implementation needs improvement.

In figure 5 is shown a strong scaling test with OpenMP parallelization on a single node. Two mesh size are considered, $32 \times 256 \times 32$ and $64 \times 512 \times 64$. The results point out that the only solver taking real advantage of the OpenMP parallelization is PastiX (mainly in the LU decomposition part), even if a small improvement of the GMRES performance is visible.

Some MPI scaling test are also performed for the GMRES and PastiX solvers (an AGMG distributed version for TOKAM3X is not available yet). It must be noticed that the fill-in of the factorized LU matrix increases when increasing the number of MPI processes for a given mesh. The reason for this behaviour is still not clear. The memory required to store the LU factorization for a given mesh increases with the number of MPI processes. In figure 6 is depicted a MPI strong scaling test on a single node for the two solvers, and using two different discretizations, $32 \times 256 \times 32$ and $64 \times 512 \times 64$. For PastiX, the largest mesh saturates the memory of the node starting from 8 MPI processes, therefore no results are available for 8 and 16 processes. The number of operations to perform the LU factorization also increases with the number of MPI processes for a given mesh, which affects the MPI scaling of PastiX as it can be notice from the figure. The scaling of the GMRES solver is fairly good even though a early saturation with the number of partitions can be noticed.

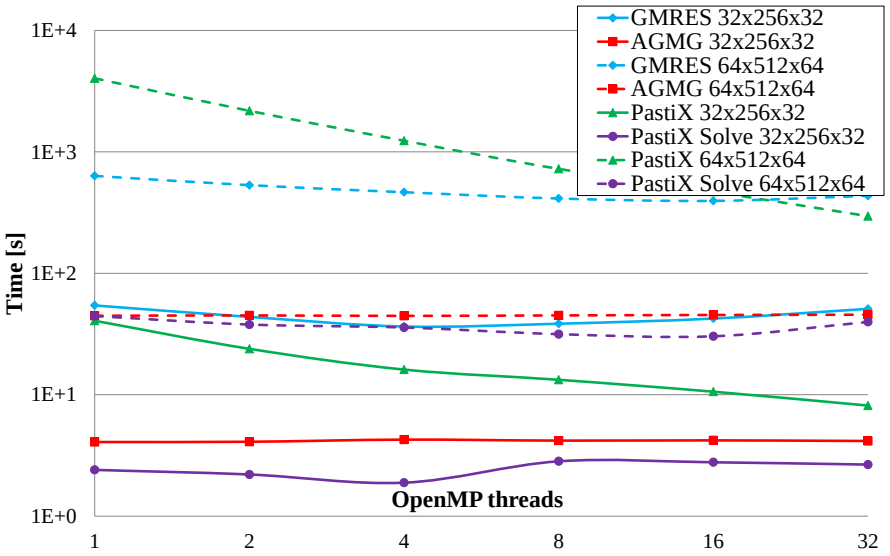


Figure 5: OpenMP strong scaling test for GMRES/AGMG/PastiX.

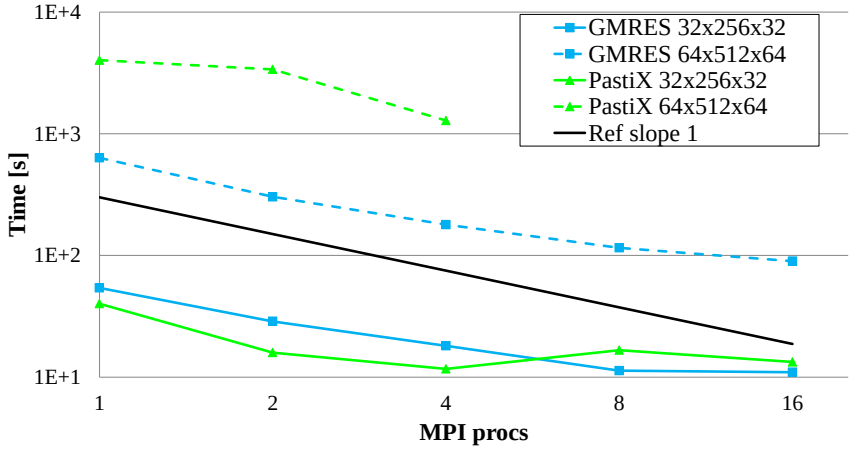


Figure 6: MPI strong scaling test for GMRES and PastiX solver on a single node.

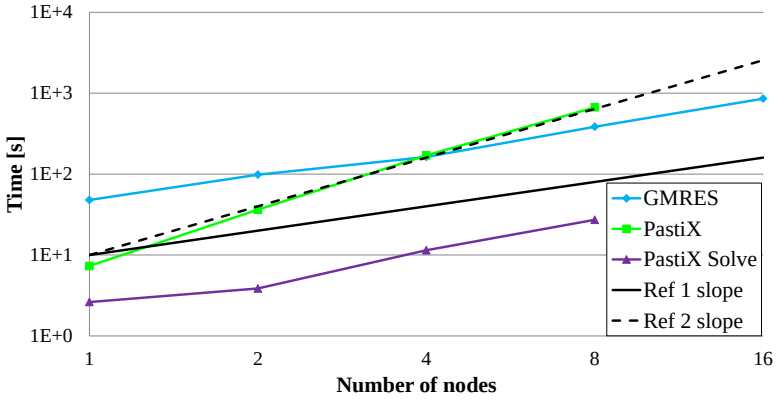


Figure 7: MPI weak scaling test for GMRES and PastiX solver. The number of MPI processes corresponds to the number of nodes used in the computations, with 32 OpenMP threads for each node in every case. The size of the problem is increased in each computation to mantain a fixed number of N_{dof} for each MPI process.

Finally, a weak scaling test is performed for GMRES and PastiX: the number of nodes used corresponds to the number of MPI processes, and a full OpenMP parallelization is performed in each node. The size of the problem is increased at each computation in order to have

$$\frac{N_{dof}}{N^0_{dof}} = N_{\text{MPI}procs} = N_{nodes}.$$

In this test, the workload for PastiX is expected to scale as N^2_{dof}, and therefore a linear increase in computing time is expected. For GMRES instead, the workload should scale as $N_{iter}N_{dof}$, and therefore the only increase in computing time should be only due to the increased number of interations to converge for each test. The results shown in figure 7 however deptict a different scenario: a quadratic increase in time is found for PastiX and a linear one for GMRES. This results need further investigastion.

Conclusions and future perspectives

Three iterative solvers are considered to speed up the inversion of the linear system derived by the discretization of the vorticity operator in TOKAM3X, task that so far was performed with the parallel direct solver PastiX. Interesting results, in terms of iterations to converge, are found with a preconditioned GMRES solver which take advantage of a preconditioner based on the physics of the problem, and also with the multigrid solver AGMG. The solver MaPHyS was found less satisfactory, so the investigation was focused on the other two solutions.

The configuration tested so far is the circular geometry with infinitely small limiter, and no electron inertia was considerd. Both GMRES and AGMG provide robust performances in this case, even if the metric coefficients seem to affect sensibly AGMG. Testing the divertor configuration remains a crucial test to be performed.

Timing tests on the CCAMU cluster pointed out some flaws of the GMRES implementation, which needs to be optimized to be competitive with PastiX or AGMG. The OpenMp parallelization of AGMG needs to be improved, and a distributed version is now fundamental to assess the possibility of using this solver in production cases.

Advanced numerical schemes for plasma simulations based on non-aligned discretizations

Introduction

Computational grids aligned with the magnetic field lines are of great interest in numerical modeling of tokamaks. From a numerical point of view, the use of aligned grids allows to reduce the pollution error in the perpendicular direction introduced by strong anisotropic equations.

However, non-aligned schemes open the path to new code capabilities that are nowadays very difficult to obtain with the aligned approach, for example

- very accurate description of the reactor chamber,
- computation extended up to the plasma center,
- possibility of computing the plasma transport in a moving equilibrium situation, for example, at start-up or during control operations.

Therefore, in order to introduce these new capabilities in the code TOKAM-3X, a non-aligned scheme is investigated. The interest was focused on a hybrid discontinuous Galerkin scheme (HDG), for its unique properties of stabilization, robustness and reduced degrees of freedom. In order to reduce the numerical diffusion introduced by the non-aligned approach, high-order polynomials are used for the interpolation of the solution.

HDG scheme and results

At first, a 2D isothermal model has been developed, with unknowns n, the plasma density, and Γ, the plasma parallel momentum. The code works on unstructured triangular grids with curved geometries, and employs polynomial interpolations of arbitrary order to approximate the solution. The code has been validated with manufactured solutions showing high-order convergence of the numerical solution and the recovery of the theoretical slopes, see Figure 8. A benchmark with the code SOLEDGE2D was then proposed, that showed a qualitative and quantitative agreement between the two codes, see Figures 9 and 10. Finally, a challenging problem with very low physical diffusivity and drift velocities was used to demonstrate the capabilities of the code to deal with very low diffusion values, and also the shock-capturing strategies used. This work allowed to publish a paper, see Ref. [GBC+18].

Taking advantage of the non-aligned grids, some computations with a moving equilibrium were performed. In the framework of the EoCoE collaboration with INRIA Sophia-Antipolis, the equilibrium code FEEQS.M was coupled to the HDG code to perform a simulation of a configuration transition from limiter to divertor. Figure 11 depicts the

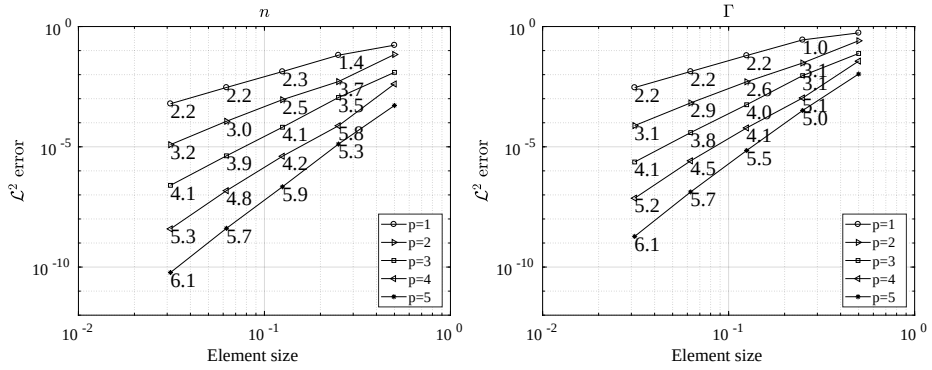


Figure 8: h-convergence tests showing the $p+1$ rate of convergence for the density (left) and parallel momentum (right). Evolution of the L^2-errors when refining meshes for 5 different polynomial degrees p.

energy deposit on the limiter and the divertor during a slow and a fast transitions. This study produced another publication, see Ref. [GCB+18].

Transport simulations including ions and electrons temperatures

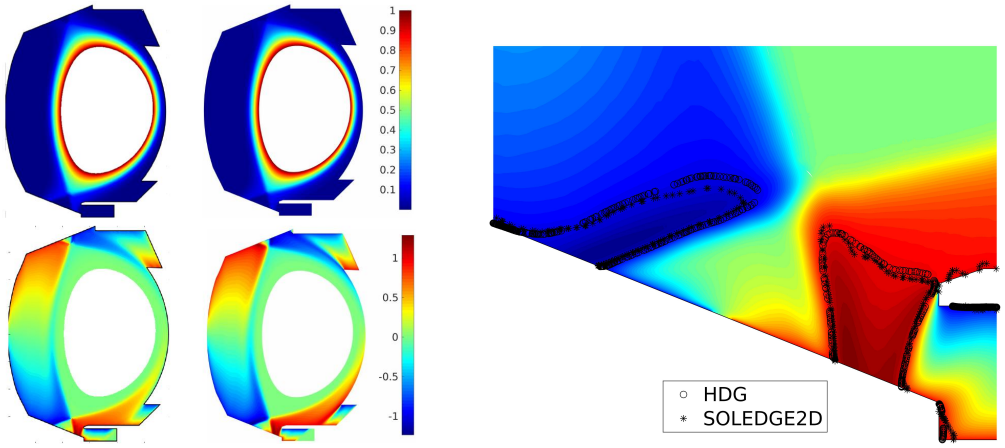
The code has been extended to include ions and electrons temperature equations. This is a critical step in evaluating the pertinence of using non-aligned grids for plasma-edge simulations: in fact, the temperature equations contain parallel diffusion terms of very large amplitude compared to cross field terms. It is well known that this anisotropy produces artificial diffusion when using non aligned low-order numerical schemes.

This new version of the code has been verified with manufactured solutions: in figure 12 are shown the convergence curves for the conservative variables, that is n, nu, nE_i, nE_e, where E_i and E_e are the ion and electron total energy, respectively.

Preliminary tests (not shown here) on a simple diffusion problem show that the use of high-order interpolations allow to reduce drastically the amount of artificial diffusion introduced by the non-aligned discretization.

Two tests of solving a set of plasma transport equations are shown here. In the first one, the geometry chosen consists in a cicular tokamak with an infintely small limiter, and two computational meshes are considered, see figure 13: a triangular mesh with $p = 8$ interpolation degree and a quadrilater aligned mesh with $p = 4$. Typical physical parameters are used to set the simulations (cross-field diffusion of 1 m^2/s and a maximum plasma temperature of 50 eV). Figures in 14 show the excellent agreement between the two computations for the density and ions and electrons temperatures.

The second test involves the WEST geometry. The computational mesh in this case is made up of 2289 $p = 6$ elements, and the same physical parameters used in the previous test are used in this computation. In figure 15 are shown the maps of density, ion and electron temperatures, and the Mach number. The results are consistent with the typical results of SOLEDGE2D for this kind of computation. The code verification however is still ongoing.



(a) (b)

Figure 9: Solutions benchmarking in WEST. In (a) the large scale flows predicted by SOLEDGE2D (left) and the new HDG solver (right) are shown. Maps of density n (top) and parallel Mach number $M_{||}$ (bottom). In (b) is depicted a zoom on the parallel Mach number $M_{||}$ around the separatrix within the divertor area, with black lines showing the transition to supersonic flows predicted by the two codes.

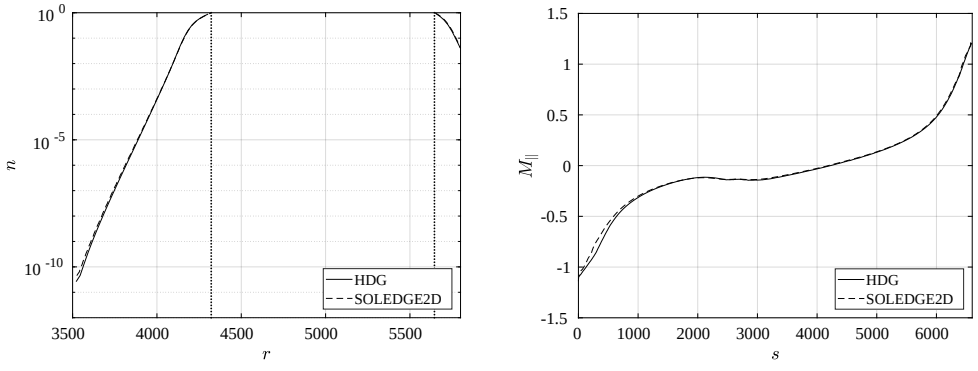


Figure 10: Solutions benchmarking in WEST. Radial density profiles at midplane (left), and parallel profiles of parallel Mach number along a magnetic field line within the SOL and close to the separatrix (right). The abscissa s defines the curvilinear coordinate along the magnetic field line. Computations are carried out with $D = \mu = 1m^2/s$.

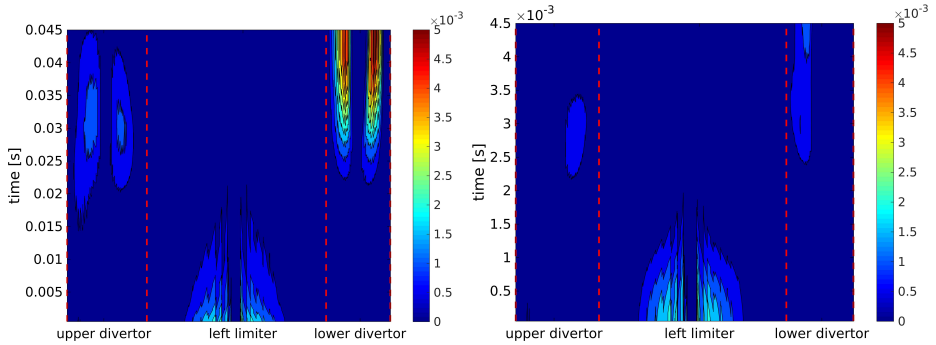


Figure 11: Evolving equilibrium: distribution of the outgoing flux of particles as a function of the time, in the slow transition simulation (left) and the fast transition simulation (right).

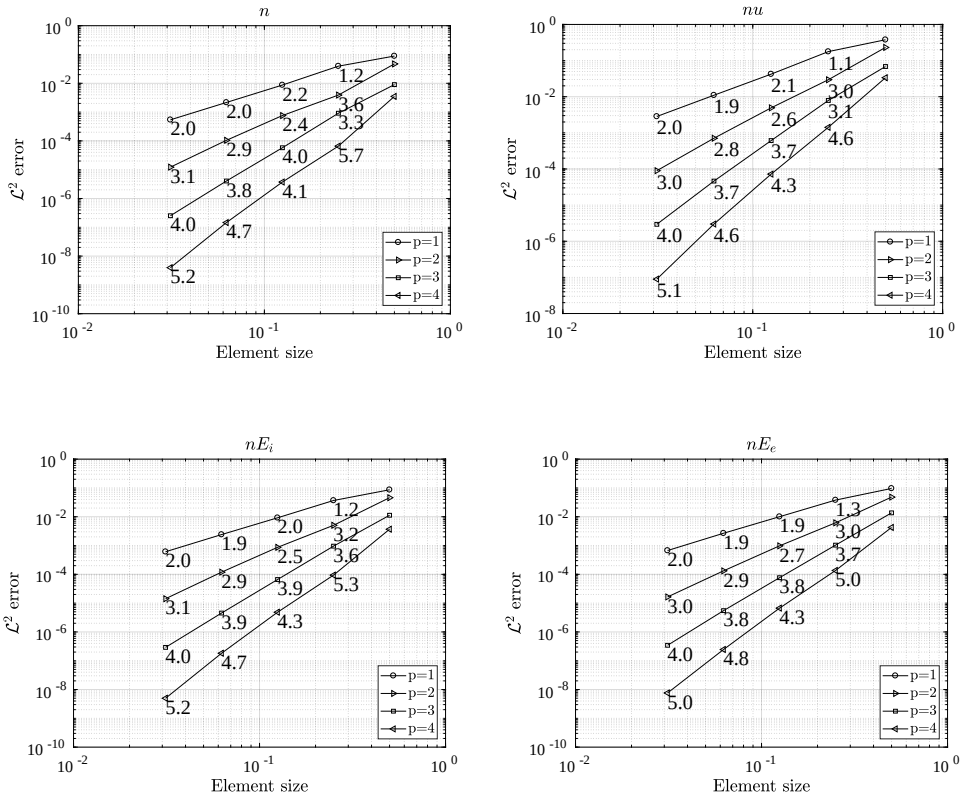


Figure 12: Convergence curves for the four conservative variables and various polynomial degrees. The slope of the curves is also shown.

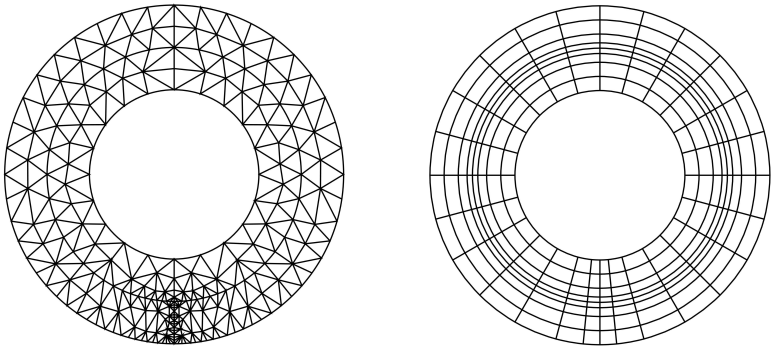


Figure 13: Circular configuration with infinitely small limiter: triangular mesh (left) and quadrangular mesh (right).

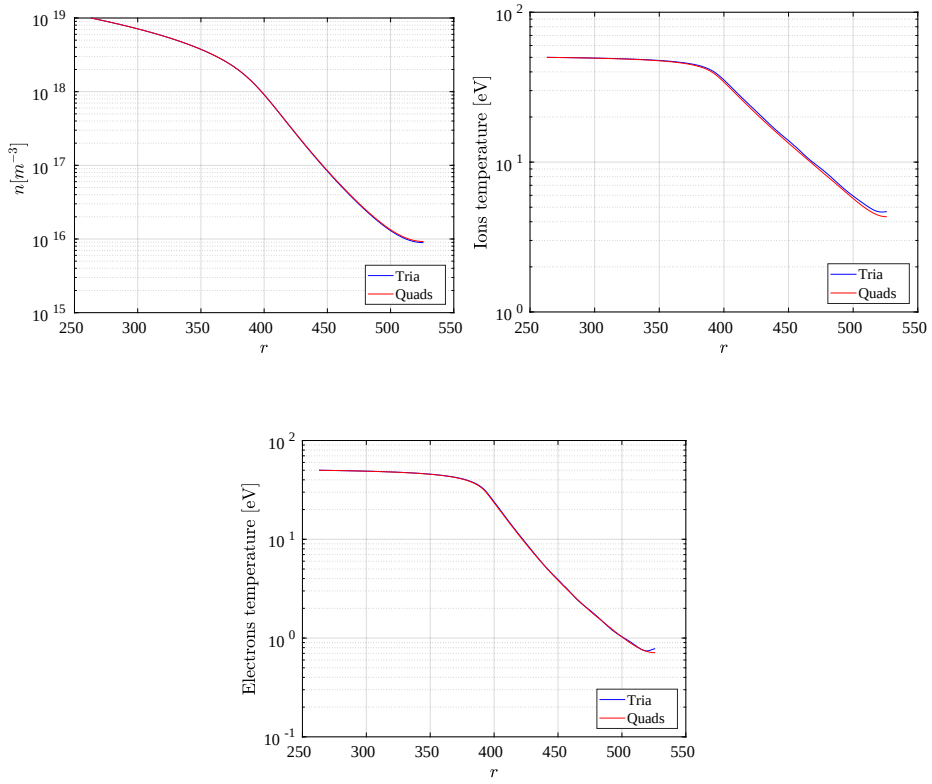
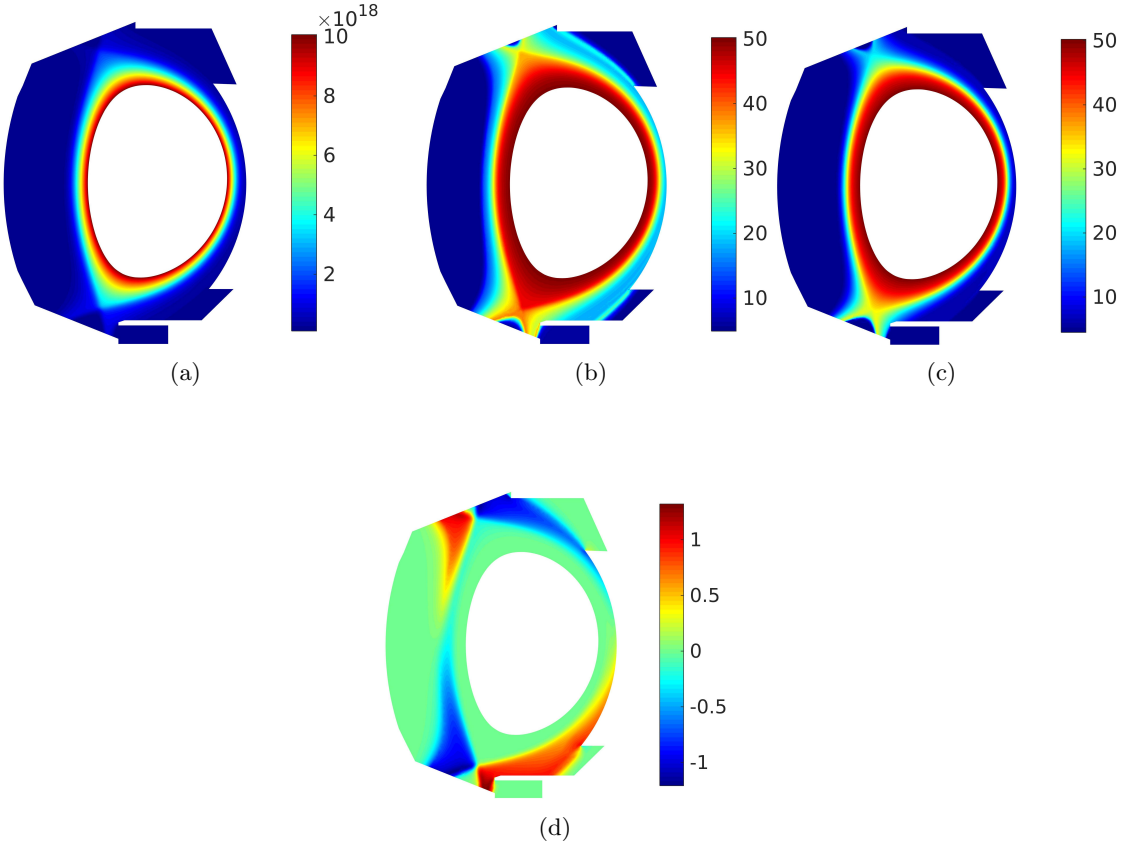


Figure 14: Circular configuration with infinitely small limiter: density, ions temperature and electrons temperature profiles for the triangular mesh and quadrangular mesh simulation.



(a) (b) (c)

(d)

Figure 15: WEST configuration: map of density (a), ion temperature (b), electron temperature (c), Mach number (d).

Conclusions and future perspectives

A new solver based on a hybrid discontinuous Galerkin scheme on a non-aligned discretization is investigated as an alternative to the commonly used finite-differences in the context of magnetised plasma simulations. The code is at present fully functional in a parallel implementation and capable of solving a transport model including density, momentum, ions and electron temperatures. The next step for the evaluation in terms of utilization in the code TOKAM3X will be the extension to the 3D to perform turbulent simulations.

References

[GBC+18] G. Giorgiani, H. Buffer, G. Ciraolo, P. Ghendrih, F. Schwander, E. Serre, and P.Tamain. A hybrid discontinuous galerkin method for tokamak edge plasma simulations in global realistic geometry. *J. Comput. Phys.*, 374:515–532, 2018.

[GCB+18] G. Giorgiani, T. Camminady, H. Bufferand, G. Ciraolo, P. Ghendrih, H. Guillard, H. Heumann, B. Nkonga, F. Schwander, E. Serre, and P. Tamain. A new high-order fluid solver for tokamak edge plasma transport simulations based on a magnetic-field independent discretization. *Contributions to Plasma Physics*, 2018.

[GYKL05] S. Günter, Q. Yu, J. Krüger, and K. Lackner. Modelling of heat transport in magnetised plasmas using non-aligned coordinates. *J. Comput. Phys.*, 209:354—-370, 2005.

[Nak15] S. Nakov. *On the design of sparse hybrid linear solvers for modern parallel architectures.* PhD thesis, Bordeaux University, 2015. https://tel.archives-ouvertes.fr/tel-01304315.

[NN15] Y. Notay and A. Napov. A massively parallel solver for discrete poisson-like problems. *J. Comput. Phys.*, 281:237—-250, 2015.

4. NanoPV

Activity type	WP1 support
Contributors	Alexis Gobé (Inria), Stéphane Lanteri (Inria) and Jonathan Viquerat (Inria)

Context

The goal of this work is to adapt and exploit a finite element type solver from the DIOGENeS software suite [1] developed at Inria Sophia Antipolis-Méditerranée for the simulation of light absorption in complex solar cells structures involving material layers with nanoscale textured surfaces. The considered electromagnetic wave propagation solver has been adpated in order to deal accuractely and efficiently with the multiscale features of the target problem. Some specific work has also been undertaken in order to optimize the scalability of the solver. This work has been realized in interaction with researchers from the group of Urs Aeberhard at IEK-5 Photovoltaic, Forschungszentrum Julich, in relation with the objectives of workpackage WP3 of the EoCoE project.

Introduction

The numerical modeling of light interaction with nanometer scale structures generally relies on the solution of the system of time-domain Maxwell equations, possibly taking into account an appropriate physical dispersion model, such as the Drude or Drude-Lorentz models, for characterizing the material properties of metallic nanostructures at optical frequencies [Mai07]. In the computational nanophotonics literature, a large number of studies are devoted to Finite Difference Time-Domain (FDTD) type discretization methods based on Yee's scheme [Yee66]. As a matter of fact, the FDTD [TH05] method is a widely used approach for solving the systems of partial differential equations modeling nanophotonic applications. In this method, the whole computational domain is discretized using a structured (cartesian) grid. However, in spite of its flexibility and second-order accuracy in a homogeneous medium, the Yee scheme suffers from serious accuracy degradation when used to model curved objects or when treating material interfaces. During the last twenty years, numerical methods formulated on unstructured meshes have drawn a lot of attention in computational electromagnetics with the aim of dealing with irregularly shaped structures and heterogeneous media. In particular, the Discontinuous-Galerkin Time-Domain (DGTD) method has met an increased interest because these methods somehow can be seen as a crossover between Finite Element Time-Domain (FETD) methods (their accuracy depends of the order of a chosen local polynomial basis upon which the solution is represented) and Finite Volume Time-Domain (FVTD) methods (the neighboring cells are connected by numerical fluxes). Thus, DGTD method offer a wide range of flexibility in terms of geometry (since the use of unstructured and non-conforming meshes is naturally permitted) as well as local approximation order refinement strategies, which are of useful practical interest.

The present study is concerned with the adaptation and application of a DGTD solver for the simulation of light trapping in a multilayer solar cell structured with textured interfaces. Our aim is to demonstrate the possibility and benefits (in terms of acuracy and computational efficiency) of exploiting topography conforming geometrical models based on non-uniform discretization meshes.

[1] http://diogenes.inria.fr/

DGTD solver for nanoscale light/matter interactions

The basic ingredient of our DGTD solver is a discretization method which relies on a compact stencil high order interpolation of the electromagnetic field components within each cell of an unstructured tetrahedral mesh. This piecewise polynomial numerical approximation is allowed to be discontinuous from one mesh cell to another, and the consistency of the global approximation is obtained thanks to the definition of appropriate numerical traces of the fields on a face shared by two neighboring cells. Time integration is achieved using an explicit scheme and no global mass matrix inversion is required to advance the solution at each time step. Moreover, the resulting time-domain solver is particularly well adapted to parallel computing. For the numerical treatment of dispersion models in metals, we have adopted an Auxiliary Differential Equation (ADE) technique that has already proven its effectiveness in the FDTD framework. From the mathematical point of view, this amounts to solve the time-domain Maxwell equations coupled to a system of *ordinary differential equations*. The resulting ADE-based DGTD method is detailed in [Viq15].

Mathematical modeling

Towards the general aim of being able to consider concrete physical situations relevant to nanophotonics, One of the most important features to take into account in the numerical treatment is physical dispersion. In the presence of an exterior electric field, the electrons of a given medium do not reach their equilibrium position instantaneously, giving rise to an electric polarization that itself influences the electric displacement. In the case of a linear homogeneous isotropic non-dispersive medium, there is a linear relation between the applied electric field and the polarization. However, for some range of frequencies (depending on the considered material), the dispersion phenomenon cannot be neglected, and the relation between the polarization and the applied electric field becomes complex. In practice, this is modeled by a frequency-dependent complex permittivity. Several such models for the characterization of the permittivity exist; they are established by considering the equation of motion of the electrons in the medium and making some simplifications. There are mainly two ways of handling the frequency dependent permittivity in the framework of time-domain simulations, both starting from models defined in the frequency domain. A first approach is to introduce the polarization vector as an unknown field through an auxiliary differential equation which is derived from the original model in the frequency domain by means of an inverse Fourier transform. This is called the Direct Method or Auxiliary Differential Equation (ADE) formulation. Let us note that while the new equations can be easily added to any time-domain Maxwell solver, the resulting set of differential equations is tied to the particular choice of dispersive model and will never act as a black box able to deal with other models. In the second approach, the electric field displacement is computed from the electric field through a time convolution integral and a given expression of the permittivity which formulation can be changed independently of the rest of the solver. This is called the Recursive Convolution Method (RCM).

In [Viq15], an ADE formulation has been adopted. We first considered the case of Drude and Drude-Lorentz models, and further extended the proposed ADE-based DGTD method to be able to deal with a generalized dispersion model in which we make use of a Padé approximant to fit an experimental permittivity function. The numerical treatment of such a generalized dispersion model is also presented in [Viq15]. We outline below the main characteristics of the proposed DGTD approach in the case of the Drude model.

The latter is associated to a particularly simple theory that successfully accounts for the optical and thermal properties of some metals. In this model, the metal is considered as a static lattice of positive ions immersed in a free electrons gas. In the case of the Drude model, the frequency dependent permittivity is given by $\varepsilon_r(\omega) = \varepsilon_\infty - \frac{\omega_d^2}{\omega^2 + i\omega\gamma}$, where ε_∞ represents the core electrons contribution to the relative permittivity ε_r, γ is a coefficient linked to the electron/ion collisions representing the friction experienced by the electrons, and $\omega_d = \sqrt{\frac{n_e e^2}{m_e \varepsilon_0}}$ (m_e is the electron mass, e the electronic charge and n_e the electronic density) is the plasma frequency of the electrons. Considering a constant permeability and a linear homogeneous and isotropic medium, one can write the Maxwell equations as

$$\begin{cases} \mathrm{rot}(\mathbf{H}) = \dfrac{\partial \mathbf{D}}{\partial t}, \\ \mathrm{rot}(\mathbf{E}) = -\dfrac{\partial \mathbf{B}}{\partial t}, \end{cases} \tag{1}$$

along with the constitutive relations $\mathbf{D} = \varepsilon_0\varepsilon_\infty\mathbf{E} + \mathbf{P}$ and $\mathbf{B} = \mu_0\mathbf{H}$, which can be combined to yield

$$\begin{cases} \mathrm{rot}(\mathbf{E}) = -\mu_0 \dfrac{\partial \mathbf{H}}{\partial t}, \\ \mathrm{rot}(\mathbf{H}) = \varepsilon_0\varepsilon_\infty \dfrac{\partial \mathbf{E}}{\partial t} + \dfrac{\partial \mathbf{P}}{\partial t}. \end{cases} \tag{2}$$

In the frequential domain the polarization $\mathbf{P}$ is linked to the electric field through the relation $\hat{\mathbf{P}} = -\frac{\varepsilon_0\omega_d^2}{\omega^2 + i\gamma_d\omega}\hat{\mathbf{E}}$, where $\hat{\cdot}$ denotes the Fourier transform of the time-domain field. An inverse Fourier transform gives

$$\frac{\partial^2 \mathbf{P}}{\partial t^2} + \gamma_d \frac{\partial \mathbf{P}}{\partial t} = \varepsilon_0 \omega_d^2 \mathbf{E}. \tag{3}$$

By defining the dipolar current vector $\mathbf{J}_p = \dfrac{\partial \mathbf{P}}{\partial t}$, (2)-(3) can be rewritten as

$$\begin{cases} \mu_0 \dfrac{\partial \mathbf{H}}{\partial t} = -\nabla \times \mathbf{E}, \qquad \varepsilon_0\varepsilon_\infty \dfrac{\partial \mathbf{E}}{\partial t} = \nabla \times \mathbf{H} - \mathbf{J}_p, \\ \dfrac{\partial \mathbf{J}_p}{\partial t} + \gamma_d \mathbf{J}_p = \varepsilon_0 \omega_d^2 \mathbf{E}. \end{cases} \tag{4}$$

Recalling the definitions of the impedance and light velocity in vacuum, $Z_0 = \sqrt{\mu_0/\varepsilon_0}$ and $c_0 = 1/\sqrt{\varepsilon_0\mu_0}$, and introducing the following substitutions, $\widetilde{\mathbf{H}} = Z_0\mathbf{H}$, $\widetilde{\mathbf{E}} = \mathbf{E}$, $\widetilde{\mathbf{J}}_p = Z_0\mathbf{J}_p$, $\widetilde{t} = c_0 t$, $\widetilde{\gamma}_d = \gamma_d/c_0$ and $\widetilde{\omega}_d^2 = \omega_d^2/c_0^2$, it can be shown that system (4) can be normalized to yield

$$\begin{cases} \dfrac{\partial \widetilde{\mathbf{H}}}{\partial t} = -\nabla \times \widetilde{\mathbf{E}}, \qquad \varepsilon_\infty \dfrac{\partial \widetilde{\mathbf{E}}}{\partial t} = \nabla \times \widetilde{\mathbf{H}} - \widetilde{\mathbf{J}}_p, \\ \dfrac{\partial \widetilde{\mathbf{J}}_p}{\partial t} + \gamma_d \widetilde{\mathbf{J}}_p = \widetilde{\omega}_d^2 \widetilde{\mathbf{E}}, \end{cases} \tag{5}$$

knowing that $\mu_0 c_0 / Z_0 = 1$ and $\varepsilon_0 c_0 Z_0 = 1$. From now on, we omit the $\widetilde{X}$ notation for the normalized variables.

DGTD method

The DGTD method can be considered as a finite element method where the continuity constraint at an element interface is released. While it keeps almost all the advantages of the finite element method (large spectrum of applications, complex geometries, etc.), the DGTD method has other nice properties:

- It is naturally adapted to a high order approximation of the unknown field. Moreover, one may increase the degree of the approximation in the whole mesh as easily as for spectral methods but, with a DGTD method, this can also be done locally i.e. at the mesh cell level.
- When the discretization in space is coupled to an explicit time integration method, the DG method leads to a block diagonal mass matrix independently of the form of the local approximation (e.g the type of polynomial interpolation). This is a striking difference with classical, continuous FETD formulations.
- It easily handles complex meshes. The grid may be a classical conforming finite element mesh, a non-conforming one or even a hybrid mesh made of various elements (tetrahedra, prisms, hexahedra, etc.). The DGTD method has been proven to work well with highly locally refined meshes. This property makes the DGTD method more suitable to the design of a *hp*-adaptive solution strategy (i.e. where the characteristic mesh size h and the interpolation degree p changes locally wherever it is needed).
- It is flexible with regards to the choice of the time stepping scheme. One may combine the discontinuous Galerkin spatial discretization with any global or local explicit time integration scheme, or even implicit, provided the resulting scheme is stable.
- It is naturally adapted to parallel computing. As long as an explicit time integration scheme is used, the DGTD method is easily parallelized. Moreover, the compact nature of method is in favor of high computation to communication ratio especially when the interpolation order is increased.

As in a classical finite element framework, a discontinuous Galerkin formulation relies on a weak form of the continuous problem at hand. However, due to the discontinuity of the global approximation, this variational formulation has to be defined at the element level. Then, a degree of freedom in the design of a discontinuous Galerkin scheme stems from the approximation of the boundary integral term resulting from the application of an integration by parts to the element-wise variational form. In the spirit of finite volume methods, the approximation of this boundary integral term calls for a numerical flux function which can be based on either a centered scheme or an upwind scheme, or a blend of these two schemes.

The DGTD method has already been considered as an alternative to the widely used FDTD method for simulating nanoscale light/matter interaction problems [NKSB09]-[BKN11]-[MNHB11]-[NDB12]. The main features of the DGTD method studied in [Viq15] for the

numerical solution of system (5) are the following:

- It is formulated on an unstructured tetrahedral mesh;
- It can deal with linear or curvilinear elements through a classical isoparametric mapping adapted to the DG framework [VS15];
- It relies on a high order nodal (Lagrange) interpolation of the components of $\mathbf{E}$, $\mathbf{H}$ and $\mathbf{J}_p$ within a tetrahedron;
- It offers the possibility of using a fully centered [FLLP05] or a fully upwind [HW02] scheme, as well as blend of the two schemes, for the evaluation of the numerical traces (also referred as numerical fluxes) of the $\mathbf{E}$ and $\mathbf{H}$ fields at inter-element boundaries;
- It can be coupled to either a second-order or fourth-order leap-frog (LF) time integration scheme [FL10], or to a fourth-order low-storage Runge-Kutta (LSRK) time integration scheme [CK94];
- It can rely on a Silver-Muller absorbing boundary condition or a CFS-PML technique for the artificial truncation of the computational domain.

Starting from the continuous Maxwell-Drude equations (5), the system of semi-discrete DG equations associated to an element τ_i of the tetrahedral mesh writes

$$\left\{\begin{array}{rcl} \mathbb{M}_i \dfrac{d\overline{\mathbf{H}}_i}{dt} & = & -\mathbb{K}_i \times \overline{\mathbf{E}}_i + \displaystyle\sum_{k\in\mathcal{V}_i} \mathbb{S}_{ik} \left(\overline{\mathbf{E}}_\star \times \mathbf{n}_{ik}\right), \\ \mathbb{M}_i^{\varepsilon\infty} \dfrac{d\overline{\mathbf{E}}_i}{dt} & = & \mathbb{K}_i \times \overline{\mathbf{H}}_i - \displaystyle\sum_{k\in\mathcal{V}_i} \mathbb{S}_{ik} \left(\overline{\mathbf{H}}_\star \times \mathbf{n}_{ik}\right) - \mathbb{M}_i \overline{\mathbf{J}}_i, \\ \dfrac{d\overline{\mathbf{J}}_i}{dt} & = & \omega_d^2 \overline{\mathbf{E}}_i - \gamma_d \overline{\mathbf{J}}_i. \end{array}\right. \tag{6}$$

In the above system of ODEs, $\overline{\mathbf{E}}_i$ is the vector of all the degrees of freedom of $\mathbf{E}$ in τ_i (with similar definitions for $\overline{\mathbf{H}}_i$ and $\overline{\mathbf{J}}_i$), $\mathbb{M}_i$ and $\mathbb{M}_i^{\varepsilon\infty}$ are local mass matrices, $\mathbb{K}_i$ is a local pseudo-stiffness matrix, and $\mathbb{S}_{ik}$ is a local interface matrix. Moreover, $\overline{\mathbf{E}}_\star$ and $\overline{\mathbf{H}}_\star$ are numerical traces computed using an appropriate centered or upwind scheme. All these quantities are detailed in [Viq15].

4.1 Application to photovoltaics

We study light trapping in a silicon-based thin-film solar cell setup that consists of several randomly textured layers. The focus is on amorphous and microcrystalline silicon (a-Si:H and μc-Si:H) which belong to the family of disordered semiconductors. The main characteristics of those materials is the structural disorder, which affect in an essential way the optical and electronic properties.

Dealing with measured optical properties

Given an experimental set of points describing a permittivity function of a material, a Padé type approximation is a convenient analytical coefficient-based function to approach experimental data. The fundamental theorem of algebra allows to expand this approxi-

mation as a sum of a constant, one zero-order pole (ZOP), a set of first-order generalized poles (FOGP), and a set of second-order generalized poles (SOGP), as

$$\varepsilon_{r,g}(\omega) = \varepsilon_\infty - \frac{\sigma}{i\omega} - \sum_{l\in L_1} \frac{a_l}{i\omega - b_l} - \sum_{l\in L_2} \frac{c_l - i\omega d_l}{\omega^2 - e_l + i\omega f_l}, \tag{7}$$

where $\varepsilon_\infty, \sigma, (a_l)_{l\in L_1}, (b_l)_{l\in L_1}, (c_l)_{l\in L_2}, (d_l)_{l\in L_2}, (e_l)_{l\in L_2}, (f_l)_{l\in L_2}$ are real constants, and L_1, L_2 are non-overlapping sets of indices. The constant ε_∞ represents the permittivity at infinite frequency, and σ the conductivity. This general writing allows an important flexibility for several reasons. First, it unifies most of the common dispersion models in a single formulation. Indeed, Debye (biological tissues in the MHz regime), Drude and Drude-Lorentz (noble metals in the THz regime), retarded Drude and Drude-Lorentz (transition metals in the THz regime), but also Sellmeier's law (glass in the THz regime), are naturally included. Second, as will be shown later, it permits to fit a large range of experimental data set in a limited number of poles (thus leading to reasonable memory and CPU overheads).

In order to fit the coefficients of (7) to experimental data, various techniques can be used, such as the well-known least square method. Here, a free existing algorithm from W.L. Goffe[2] was adapted for this study. In practice, for a given model, a set of experimental data is provided to the optimization algorithm. This method demonstrated good efficiency while fitting up to 17 parameters simultaneously.

Following similar steps as for the Drude model, one derives the system of PDEs, accounting for the generalized dispersive model in time-domain

$$\begin{cases} \dfrac{\partial \mathbf{H}}{\partial t} &= -\nabla \times \mathbf{E}, \\ \dfrac{\partial \mathbf{E}}{\partial t} &= \dfrac{1}{\varepsilon_\infty}\left(\nabla \times \mathbf{H} - \mathbf{J}_0 - \displaystyle\sum_{l\in L_1} \mathbf{J}_l - \sum_{l\in L_2} \mathbf{J}_l\right), \\ \mathbf{J}_0 &= (\sigma + \displaystyle\sum_{l\in L_2} d_l)\mathbf{E}, \\ \mathbf{J}_j &= a_l\mathbf{E} - b_l\mathbf{P}_l \quad \forall l \in L_1, \\ \dfrac{\partial \mathbf{P}_l}{\partial t} &= \mathbf{J}_l \quad \forall l \in L_1, \\ \dfrac{\partial \mathbf{J}_l}{\partial t} &= (c_l - d_l f_l)\mathbf{E} - f_l\mathbf{J}_l - e_l\mathbf{P}_l \quad \forall l \in L_2, \\ \dfrac{\partial \mathbf{P}_l}{\partial t} &= d_l\mathbf{E} + \mathbf{J}_l \quad \forall l \in L_2. \end{cases} \tag{8}$$

The numerical treatment of system (8) in the framework of a DGTD method is detailed in [Viq15].

Construction of geometrical models

The first task that we had to deal with aimed at developing a dedicated preprocessing tool for building geometrical models that can be used by the DGTD solver. Such a geometrical model consists in a fully unstructured tetrahedral mesh, which is obtained using an appropriate mesh generation tool. We use the tetrahedral mesh generator from the MeshGems suite[3].

[2] `http://ideas.repec.org/c/wpa/wuwppr/9406001.html`
[3] http://www.meshgems.com/

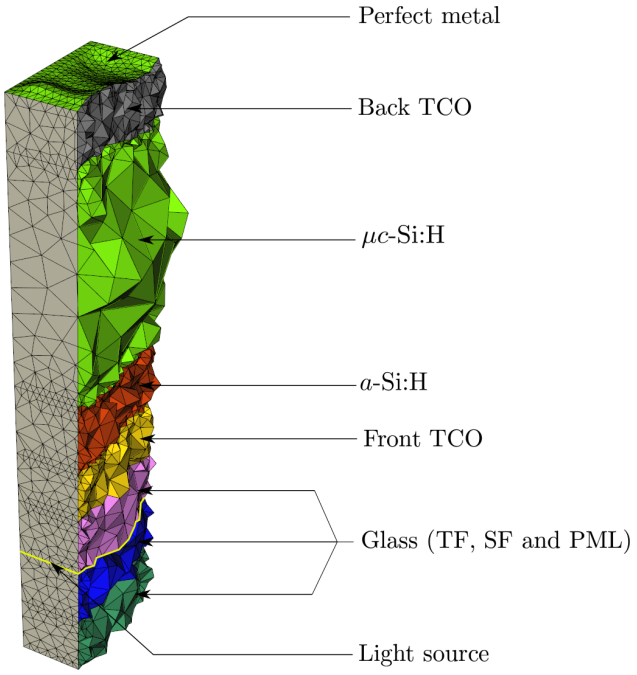


Figure 16: Geometrical model of the solar cell structure and composition of the different layers. Layer thicknesses are in the order of the wavelength of relevant sunlight.

Smoothing step

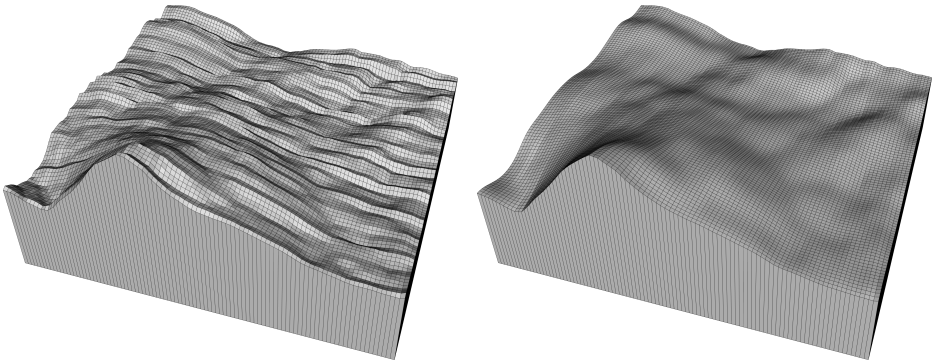
The initial layers data presented abrupt jumps in one direction of space. We started by smoothing the layers with an in-house tool (see a comparison in Fig. 17).

Building process

The building process of geoemtrical models of a cell structurr is the following:

1. Build an initial closed surface mesh made of quadrilaterals from the smoothed layers data, using a specifically developed tool;
2. Transform the quadrangular faces to triangular faces to obatin a highly refined trangular surface mesh;
3. Build a pseudo-CAD model from the triangular surface mesh;
4. Use of a surfacic meshing tool to create a new, optimized triangular mesh from the CAD model;
5. Build a tetrahedral mesh from the optimized surface mesh.

Obviously, all these steps introduce discrepancies between the ideal model and the obtained mesh. It would be important to check, as a future step, how we can control and minimize the impact of the aforementioned steps. However, we must note here that it is not an easy task to obtain an exploitable mesh. In particular, we have to impose Periodic Boundary Conditions (PBC) that allow to simulate artificially infinite mono-directional or



(a) Original surface. (b) Smoothed surface.

Figure 17: Smoothed and not-smoothed versions of the UcSI layer. This is a 100×100 subset of the full layer surface.

bi-directional arrays while considering only one elementary pattern. To do so, cells from a periodic boundary face are matched with their neighbors on the opposite boundary of the domain. We have two possibilities to obtained such boundaries. The first one consist of symmetrizing the mesh. The main drawback of this method is the multiplication of the domain size by 4. This is not a major issue for a small model, but for the full device this is not a feasible approach. The second option is to use a Tukey-window function on each layer to have the same 1D border and so the same lateral faces. (as done in [JLIZ15]). The disadvantage here is the loss of information on the edges of the structure as can be seen in Fig.18.

Obtained meshes

Partial views of generated meshes are shown in Fig. 18. These meshes corresponds to a $1 \times 1\,\mu\text{m}^2$ subset of the full model, which will be used for preliminary tests. The mesh in Fig. 19 is obtained for the full model using the Tukey-window approach.

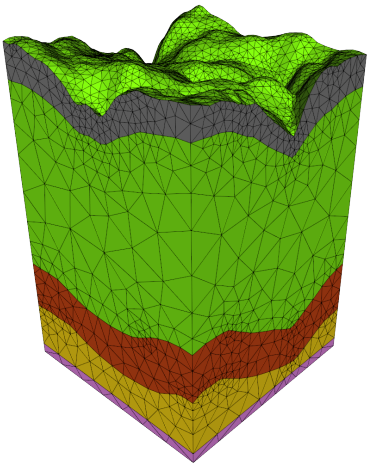
Material models

The optical properties of the different materials that constitute the considered solar cell structure have been fitted to the parameters of our generalized dispersion model, which was originally intended for metals. The obtained permittivity functions are plotted in Fig. 20. As can be seen, all materials are relatively well approximated on the range $\lambda = [300, 1300]$ nm. Regarding glass, a constant permittivity $\varepsilon_r = 2.25$ is used.

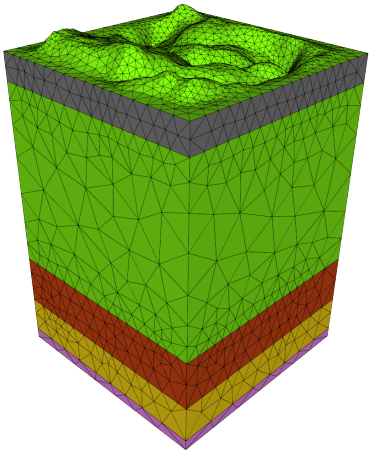
Parallelization aspects

The DGTD solver is parallelized using a classical SPMD strategy combining a partitioning of the underlying tetrahedral mesh, with a message passing programming model using the MPI standatd. The MeTiS[4] graph partitioning tool is exploited for decomposing the mesh into submeshes.

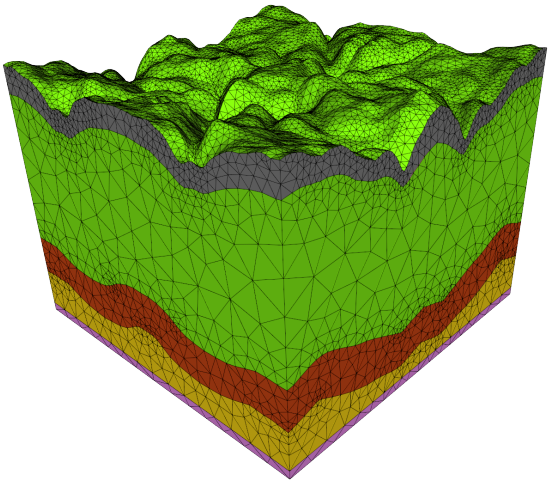
[4] http://glaros.dtc.umn.edu/gkhome/metis/metis/overview



(a) Original model.



(b) Mirrorized version of the original model.



(c) Modified model after applying Tukey-window.

Figure 18: Examples of the obtained meshes. SF and PML layers are not included here.

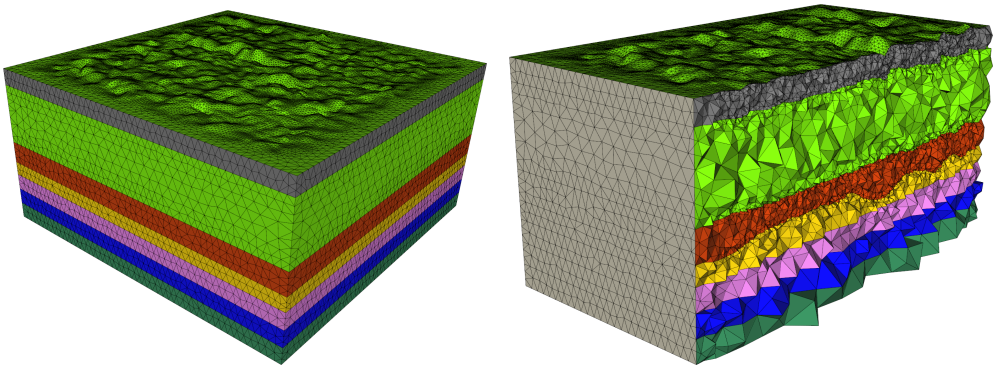
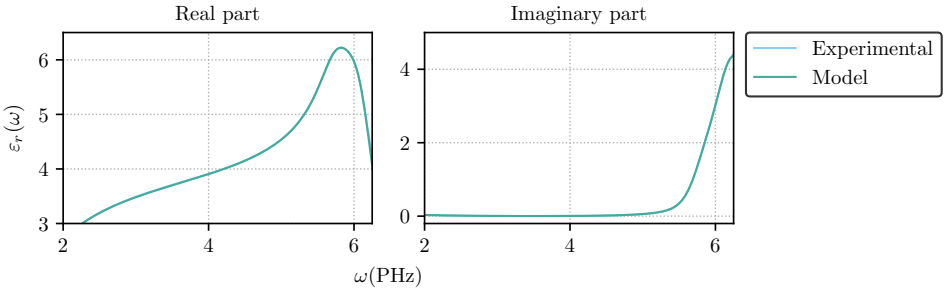
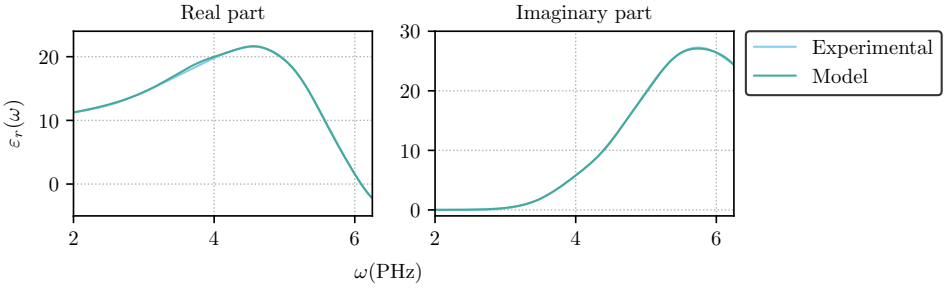


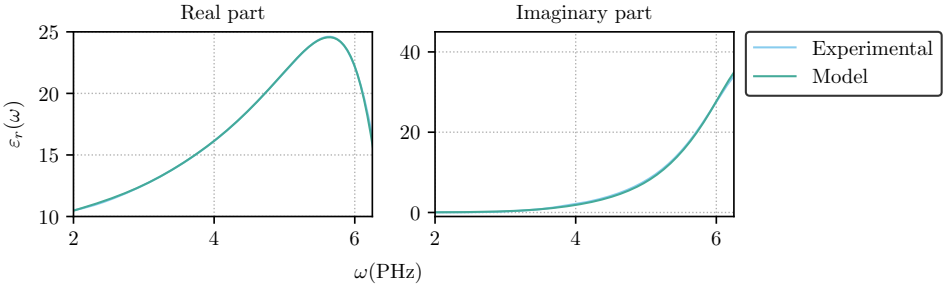
Figure 19: Examples of the obtained meshes.



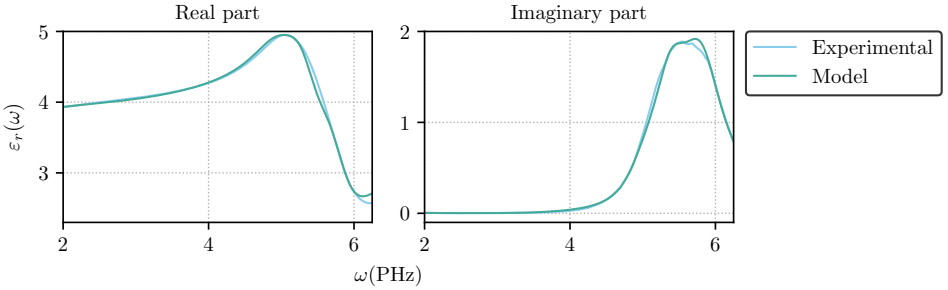
(a) Front TCO.



(b) a-Si:H.



(c) μc-Si:H.



(d) Back TCO.

Figure 20: Real and imaginary parts of the relative permittivity of front TCO, Asi-i, Ucsi-i and back TCO predicted by our dispersive model compared to experimental data.

Observable quantities

A physical quantity that is relevant for the study is the absorption in the silicon-based materials. It can be computed with a volumetric method [Viq15]. Indeed, it is possible to evaluate the ohmic losses directly inside the material. It can be shown that the power absorbed by a layer as ohmic losses is

$$\mathcal{A}_{\text{layer}}(\omega) = P_{Ohm}(\omega) = \frac{\varepsilon_0 \omega}{2} \int_{\Omega_S} \Im(\varepsilon_r(\omega)) |\widehat{\mathbf{E}}(\mathbf{r}, \omega)^2| \tag{9}$$

where Ω_S is the volume delimited by the layer. To allow for a straightforward comparison between experimental and simulation results, the external quantum efficiency (EQE) is used

$$\text{EQE}(\omega) = \frac{\mathcal{A}_{\text{layer}}(\omega)}{I_0 \cdot S_{\text{domain}}} \tag{10}$$

where I_0 is the intensity of the incident light and S_{domain} is the surface area of the domain perpendicular to the incident wave. This quantity represent the amount of the incident light which is absorbed in the layer.

Numerical and performance results

Convergence study

Numerical convergence has been assessed by considering $1 \times 1\,\mu\text{m}^2$ submodel of the solar cell structure on one hand, and the full size model on the other hand. Several tetrahedral meshes have been constructed using the procedure described in section 4.1. The characteristics of some of the meshes are summarized in Tab. 1. We plot in Fig. 21 to 22 the EQE obtained for polynomial orders ranging from 1 to 4 in the DGTD method. As can be seen, the DGTD-$\mathbb{P}_3$ and DGTD-$\mathbb{P}_4$ methods yield almost identical results when considering the submodels for the a-Si:H layer.

Mesh	Tetrahedron	$h_{\min}$ (nm)	$h_{\max}$ (nm)	$\frac{h_{\max}}{h_{\min}}$
$1 \times 1\,\mu\text{m}^2$	41387	9.9	482.6	48.7
$10 \times 10\,\mu\text{m}^2$	1151793	6.7	917.8	137.0
$10 \times 10\,\mu\text{m}^2$ homogeneous	603343	10.0	530.4	53.0

Table 1: Characteristics of the tetrahedral meshes for the $1 \times 1\,\mu\text{m}^2$ and $10 \times 10\,\mu\text{m}^2$ models.

Influence of layer surface texturing

Here we highlight the effects of texturing of the layers by comparing simulations made on the model with textured interfaces and a model with flat interfaces. In Fig. 24, we plot the EQE spectrum of the silicon-based layers for a $\mathbb{P}_4$ polynomial expansion. As can be seen, flat interfaces created cavity mode between layers which are responsible of the resonances.

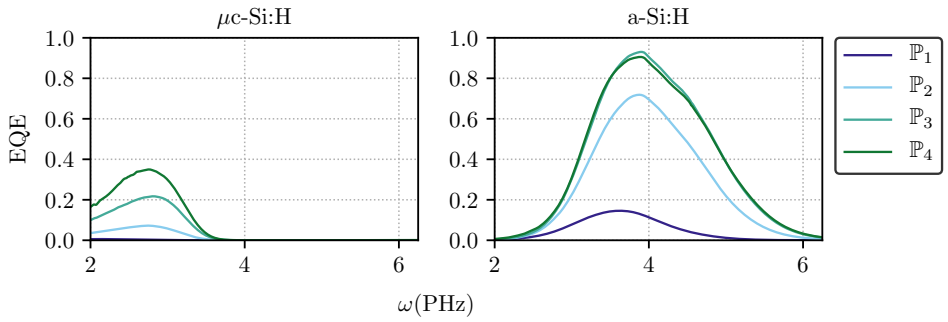


Figure 21: EQE for $1 \times 1\,\mu\text{m}^2$ submodel with polynomial orders from 1 to 4.

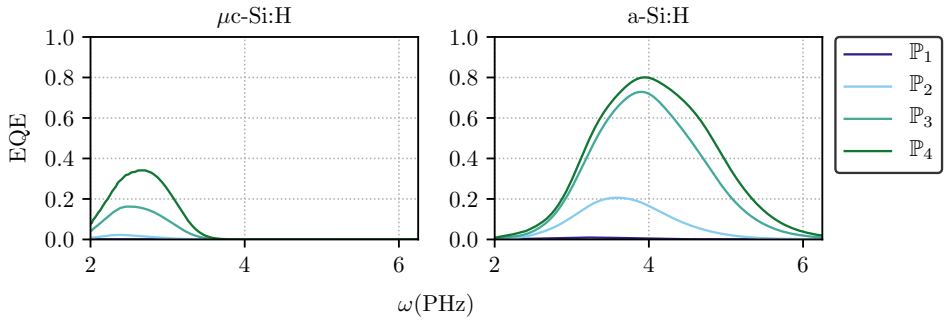


Figure 22: EQE for $10 \times 10\,\mu\text{m}^2$ model with polynomial orders from 1 to 4.

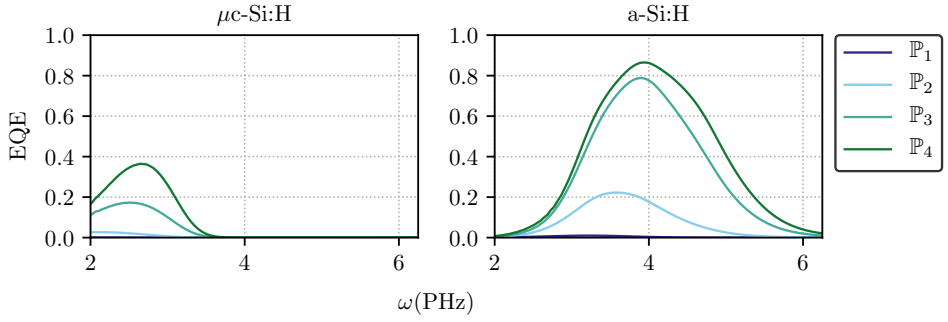


Figure 23: EQE for $10 \times 10\,\mu\text{m}^2$ homogeneous model with polynomial orders from 1 to 3.

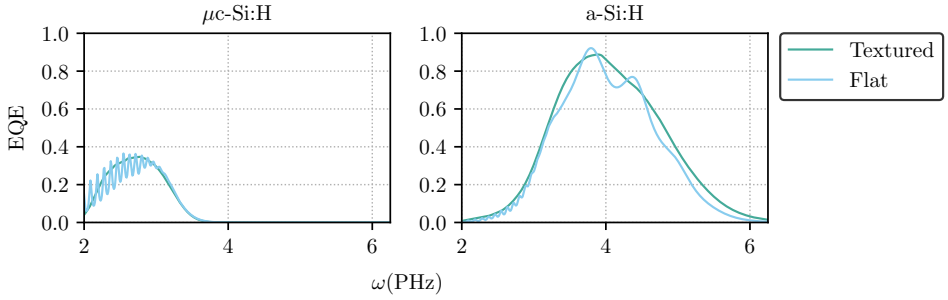


Figure 24: EQE of microcrystalline silicon (μc-Si:H) and amorphous silicon (a-Si:H) layers for textured and flat $1 \times 1\,\mu\text{m}^2$ submodel.

Comparison with FDTD simulations

A comparison with results from a FDTD simulations performed at IEK-5 is shown in Fig. 25. As can be seen, the results are in relatively good agreement.

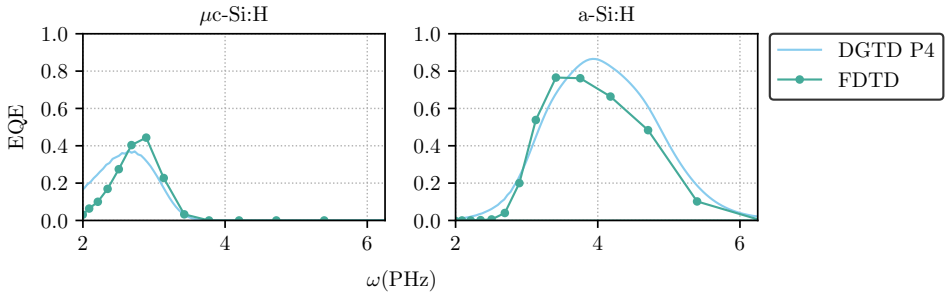


Figure 25: Comparison of EQE from simulations with the FDTD and DGTD solvers, for microcrystalline silicon (μc-Si:H) and amorphous silicon (a-Si:H) using the full model.

Shannon theorem for frequency acquisition

In order to reduce the computational overhead of computing the volumetric absorption we have exploited Shannon's theorem for sampling of frequency-dependent quantities. In fact, setting

$$\Delta t_{\text{obs}} = \frac{1}{2 f_{max}},$$

allows to perfectly sample the spectrum of the $\mathcal{A}_{\text{layer}}$ operator, by evaluating Eq. 10 at certain Δt_{obs} time steps. By doing so, we can reduce the CPU time up to 400% for the full model as one can see in Tab. 2. Simulations are run on a in-house cluster system with 128 cores (16 Intel E5-2670@2.60 GHz nodes each with 8 cores).

Mesh	Order	Improved time	Original time	Gain (%)
$1 \times 1\,\mu\text{m}^2$	$\mathbb{P}_1$	12m 38s	13m 58s	10.6
	$\mathbb{P}_2$	19m 01s	23m 39s	24.4
	$\mathbb{P}_3$	46m 58s	1h 01m 08s	30.2
	$\mathbb{P}_4$	3h 25m 34s	3h 45m 06s	9.5
$10 \times 10\,\mu\text{m}^2$	$\mathbb{P}_1$	3h 02m 16s	15h 54m 14s	423.5
	$\mathbb{P}_2$	9h 29m 52s	49h 46m 51s	424.1
	$\mathbb{P}_3$	32h 43m 57s	126h 12m 00s	285.5

Table 2: CPU times with and without applying Shannon theorem.

Scalability

We have also performed a strong scalability analysis by applying the proposed DGTD solver with a tetrahedral mesh of the full solar cell model consisting of 305,265 vertices

and 1,689,764 elements. This performance analysis is conducted on the Occigen PRACE system hosted by CINES in Montpellier. Each node of this system consists of two Intel Haswell E5-2690@2.6 GHz CPU each with 12 cores. The parallel speedup is evaluated for 1000 time iterations of the DGTD-$\mathbb{P}_k$ solver using a fourth-order low-storage Runge-Kutta time scheme. Here, $\mathbb{P}_k$ denotes the set of Lagrange polynomials of order less or equal to k. In other words, DGTD-$\mathbb{P}_k$ refers to the case where the interpolation of the components of the $(\mathbf{E}, \mathbf{H}, \mathbf{J}_p)$ fields relies on a k-order polynomial within each element of the mesh. For this preliminary study, the interpolation order is uniform (i.e. is the same for all the elements of the mesh) but the DG framework allows to easily adapt locally the interpolation order [VL16]. Performance results are presented in Tab. 3 and 4. where "Elapsed" is the elapsed time, which is used for the evalaution of the parallel speedup relatively to the first figure given for each configuration of the DGTD-$\mathbb{P}_k$ method.

In the first table, the timings include the calculation of an important observable quantity for the considered problem, which is the *volume absorption*. The computation of this quantity requires to sweep over frequencies in a target spectrum, and compute on the fly during the time evolution a discrete Fourier transform of the electrical field (i.e. for each degree of freedom of the DG approximation of the electrical field). The later computation can be performed in a fully parallel way for each element of the mesh. However, the evaluation of the volume absorption is limited to a subvolume (i.e. a layer) of the multilayer solar cell model; in the present case the aSi layer is selecetd. In our current method for partitioning the tetrahedral mesh for the SPMD parallelization of the DGTD solver, we do not take into account the computational load balancing issues raised by this localization of the computation of the volume absorption. Then, the obtained parallel speedup in Tab. 3 is suboptimal and degrades when the interpolation degree k is increased. On the contrary, when the evaluation of the volume absorption is deactivated, the parallel performances are quasi-optimal i.e. a linear speedud is observed (see Tab. 4).

Solver	# cores	Elapsed	Speedup
DGTD-$\mathbb{P}_1$	96	2681 sec	1.00 (1.0)
-	192	1365 sec	1.95 (2.0)
-	384	768 sec	3.50 (4.0)
DGTD-$\mathbb{P}_2$	96	4364 sec	1.00 (1.0)
-	192	2254 sec	1.95 (2.0)
-	384	1332 sec	3.30 (4.0)
DGTD-$\mathbb{P}_3$	192	3678 sec	1.00 (1.0)
-	384	2232 sec	1.65 (2.0)

Table 3: Strong scalability analysis of the DGTD-$\mathbb{P}_k$ solver on the Occigen system. Mesh M1 (full model) with 305,265 vertices and 1,689,764 elements. Timings for 1000 time iterations including the evaluation of the volume absorption in the aSi layer. Execution mode: 1 MPI process per core.

References

[BKN11] K. Busch, M. König, and J. Niegemann. Discontinuous Galerkin methods in nanophotonics. *Laser and Photonics Reviews*, 5:1–37, 2011.

[CK94] M.H. Carpenter and C.A. Kennedy. Fourth-order $2n$-storage Runge-Kutta

Solver	# cores	Elapsed	Speedup
DGTD-$\mathbb{P}_1$	96	584 sec	1.00 (1.0)
-	192	292 sec	2.00 (2.0)
-	384	146 sec	4.00 (4.0)
DGTD-$\mathbb{P}_2$	96	974 sec	1.00 (1.0)
-	192	490 sec	2.00 (2.0)
-	384	246 sec	3.95 (4.0)
DGTD-$\mathbb{P}_3$	192	808 sec	1.00 (1.0)
-	384	418 sec	1.95 (2.0)

Table 4: Strong scalability analysis of the DGTD-$\mathbb{P}_k$ solver on the Occigen system. Mesh M1 (full model) with 305,265 vertices and 1,689,764 elements. Timings for 1000 time iterations excluding the evaluation of the volume absorption in the aSi layer. Execution mode: 1 MPI process per core.

schemes. Technical report, NASA Technical Memorandum MM-109112, 1994.

[FL10] H. Fahs and S. Lanteri. A high-order non-conforming discontinuous Galerkin method for time-domain electromagnetics. *J. Comp. Appl. Math.*, 234:1088–1096, 2010.

[FLLP05] L. Fezoui, S. Lanteri, S. Lohrengel, and S. Piperno. Convergence and stability of a discontinuous Galerkin time-domain method for the 3D heterogeneous Maxwell equations on unstructured meshes. *ESAIM: Math. Model. Numer. Anal.*, 39(6):1149–1176, 2005.

[HW02] J.S. Hesthaven and T. Warburton. Nodal high-order methods on unstructured grids. I. Time-domain solution of Maxwell's equations. *J. Comput. Phys.*, 181(1):186–221, 2002.

[JLIZ15] K. Jäger, D.N.P. Linssen, O. Isabella, and M. Zeman. Ambiguities in optical simulations of nanotextured thin-film solar cells using the finite-element method. *Optics Express*, 23(19):A1060, 2015.

[Mai07] S.A. Maier. *Plasmonics - Fundamentals and applications.* Springer, 2007.

[MNHB11] C. Matysseka, J. Niegemann, W. Hergertb, and K. Busch. Computing electron energy loss spectra with the Discontinuous Galerkin Time-Domain method. *Photonics Nanostruct.*, 9(4):367–373, 2011.

[NDB12] J. Niegemann, R. Diehl, and K. Busch. Efficient low-storage Runge-Kutta schemes with optimized stability regions. *J. Comput. Phys.*, 231(2):364–372, 2012.

[NKSB09] J. Niegemann, M. König, K. Stannigel, and K. Busch. Higher-order time-domain methods for the analysis of nano-photonic systems. *Photonics Nanostruct.*, 7:2–11, 2009.

[TH05] A. Taflove and S.C. Hagness. *Computational electrodynamics: the finite-difference time-domain method - 3rd ed.* Artech House Publishers, 2005.

[Viq15] J. Viquerat. *Simulation of electromagnetic waves propagation in nano-optics with a high-order discontinuous Galerkin time-domain method.* PhD thesis, University of Nice-Sophia Antipolis, 2015. https://tel.archives-ouvertes.fr/tel-01272010.

[VL16] J. Viquerat and S. Lanteri. Simulation of near-field plasmonic interactions with a local approximation order discontinuous Galerkin time-domain method. *Photonics and Nanostructures - Fundamentals and Applications*, 18:43–58, 2016.

[VS15] J. Viquerat and C. Scheid. A 3D curvilinear discontinuous Galerkin time-domain solver for nanoscale light–matter interactions. *J. Comp. Appl. Math.*, 289:37–50, 2015.

[Yee66] K.S. Yee. Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. *IEEE Trans. Antennas and Propag.*, 14(3):302–307, 1966.

5. PSBLAS and MLD2P4

Contributors	Pasqua D'Ambra (National Research Council of Italy - CNR, Naples, Italy), Daniela di Serafino (University of Campania "L. Vanvitelli", Caserta, Italy), Salvatore Filippone (Cranfield University, Cranfield, UK)

5.1 Introduction

We summarize the long-term activities performed until the end of January 2018 by CNR, University of Campania "Luigi Vanvitelli" (formerly Second University of Naples) and University of Rome "Tor Vergata", within Task 2 (*Linear Algebra*) of Workpackage 1.

The activities described here were mainly motivated by the results obtained by applying the algebraic multilevel preconditioners implemented in MLD2P4 [DdSF10, P. 17], coupled with Krylov solvers from PSBLAS [FC00, FB12], to linear systems made available from WP2 and WP4. Although MLD2P4 was improved during the EoCoE project, and provided satisfactory results on some EoCoE test problems, its algebraic multilevel preconditioners (see Deliverable D 1.7 on Software Technology Improvement) lost their robustness and parallel efficiency when dealing with systems arising from highly anisotropic problems.

A crucial issue was the decoupled smoothed-aggregation implemented in MLD2P4 as coarsening algorithm [VMB96, RST00]. Therefore, we directed our interest toward different algebraic coarsening algorithms. The first results along this line are described in Sections 5.2-5.3 and were also discussed in [HDdSF18].

We also started some work for extending the current versions of PSBLAS and MLD2P4 with GPU plugins, specifically tailored for EoCoE applications, in order to efficiently run our solvers and preconditioners on current and future heterogeneous architectures toward exascale, including GPGPU accelerators.

5.2 Algebraic coarsening based on weighted matching

We began investigating the effectiveness, in the MLD2P4 framework, of a coarsening algorithm for symmetric positive definite (spd) matrices based on a graph matching approach. This algorithm, named *coarsening based on compatible weighted matching*, was recently proposed in [DV13, DV16] and implemented in the C package *BootCMatch: Bootstrap AMG based on Compatible Weighted Matching*. It defines a pairwise aggregation of unknowns where each pair is the result of a maximum weight matching in the matrix adjacency graph. Specifically, the aggregation scheme uses a maximum product matching in order to enhance the diagonal dominance of a matrix representing the hierarchical complement of the resulting coarse matrix, thereby improving the convergence properties of a corresponding compatible relaxation scheme. The matched nodes are aggregated to form coarse index spaces, and piecewise constant or smoothed interpolation operators are applied for the construction of a multigrid hierarchy. More aggressive coarsening can be obtained by combining multiple steps of the pairwise aggregation, which allows to reduce operator complexity of the final AMG preconditioners.

A parallel version of the maching-based coarsening algorithm was implemented in MLD2P4 by using a decoupled approach, where each parallel process performs coarsening on the part of the matrix owned by the process itself. The MLD2P4 software framework was extended in order to efficiently interface the BootCMatch functions implementing the

sequential coarsening algorithm, and to combine the new functionality with the other AMG components of MLD2P4. Details on the interfacing between MLD2P4 and BootCMatch are given in [HDdSF18].

Three algorithms for maximum product weighted matching were considered, all available to MLD2P4 through BootCMatch:

MC64: the algorithm implemented in the MC64 routine of the HSL library, which finds optimal matchings with a worst-case computational complexity $O(n(n+nnz)\log n)$, where n is the matrix dimension and nnz the number of its nonzero entries;

half-approximate: the greedy algorithm described in [Pre99], capable of finding, with complexity $O(nnz)$, a matching whose total weight is at least half the optimal weight;

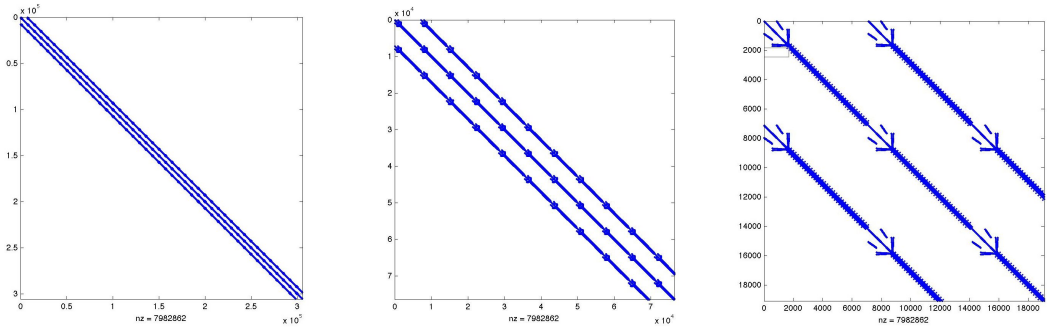
auction-type: a version of the near-optimal auction algorithm proposed in [Ber88], implemented in the SPRAL Library as described in [HS15]; note that this algorithm reduces the cost of the original auction one, producing a near-optimal matching at a much lower cost than that of the (optimal) MC64.

5.3 Results on EoCoE data sets

First experiments with multilevel preconditioners using maching-based coarsening were performed on two linear systems from WP2 (*Meteorology for Energy*) and on three linear systems from WP4 (*Water for Energy*), respectively. The experiments were carried out on the yoda linux cluster, operated by the Naples Branch of the CNR Institute for High-Performace Computing and Networking. Its compute nodes consist of 2 Intel Sandy Bridge E5-2670 8-core processors and 192 GB of RAM, connected via Infiniband. Given the size and the sparsity of the linear systems, at most 64 cores, running as many parallel processes, were used; 4 cores per node were considered, according to the memory bandwidth requirements of the linear systems. PSBLAS 3.4 and MLD2P4 2.2, installed on the top of MVAPICH 2.2, were used together with a development version of BootCMatch and the version of UMFPACK available in SuiteSparse 4.5.3. The codes were compiled with the GNU 4.9.1 compiler suite.

WP2 code: Alya

The systems come from computational fluid dynamics simulations for wind farm design and management, carried out at BSC by using the HPC multi-physics simulation code Alya [ea01]. Specifically, the systems arise from the numerical solution of Reynolds-Averaged Navier-Stokes equations coupled with a modified $k-\varepsilon$ model. The space discretization is obtained by using stabilized finite elements, while the time integration is performed by combining a backward Euler scheme with a fractional step method, which splits the computation of the velocity and pressure fields and thus requires the solution of two linear systems at each time step. The systems considered here concern the pressure field. They have symmetric spd matrices of size 790856 and 2224476, with 20905216 and 58897774 nonzeros, respectively, and are denoted by PRESS1 and PRESS2. Their sparsity pattern is shown in Figure 26. As mentioned in Section 5.1, we did not achieve satisfactory results on these matrices by using as coarsening algorithm the classical smoothed aggregation implemented in MLD2P4.



full zoom 1 zoom 2

Figure 26: Pressure matrices from wind farm simulations: sparsity pattern (full matrix and details).

PRESS1			
procs	iters	time	sp
1	8	51.22	1.0
2	39	25.39	2.0
4	40	11.69	4.4
8	50	5.96	8.6
16	57	2.89	17.7
32	84	2.18	23.5
64	58	1.20	42.8

PRESS2			
procs	iters	time	sp
1	8	76.00	1.0
2	40	81.90	0.9
4	44	39.26	1.9
8	43	19.24	3.9
16	48	10.84	7.0
32	52	5.25	14.5
64	48	2.60	29.2

Table 5: Test problems PRESS1 and PRESS2: number of iterations, execution time and speedup for weighted matching based on MC64.

PRESS1 and PRESS2 were preconditioned by using a K-cycle [Not10] with decoupled unsmoothed double-pairwise matching-based aggregation. One block-Jacobi sweep, with ILU(0) factorization of the blocks, was applied as pre/post-smoother, and UMFPACK (`http://faculty.cse.tamu.edu/davis/suitesparse.html`) was used, through the interface provided by MLD2P4, to solve the coarsest-level system, replicated in all the processes. The PSBLAS implementation of FCG(1) [Not00] was chosen to solve the systems, according to the variability introduced in the preconditioner by the K-cycle. The experiments were performed using all the three matching algorithms mentioned in Section 5.2. The zero vector was used as starting guess and the preconditioned FCG iterations were stopped when the 2-norm of the residual achieved a reduction by a factor of 10^{-6}. A generalized row-block distribution of the matrices, obtained by using the Metis graph partitioner (`http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`), was chosen. Among the matching algorithms, only MC64 was able to produce an effective coarsening, hence avoiding a significant increase of the number of iterations with the number of cores. Therefore, we discuss results for this case only.

In Table 5 we report the FCG iterations, the execution time (in seconds) and the speedup obtained with PRESS1 and PRESS2. The execution time includes the construction of the preconditioner and the solution of the preconditioned linear system. The preconditioned solver shows good algorithmic scalability in general; the number of iterations on a single

	MAT1			MAT2			MAT3		
np	MC64	half-app	auction	MC64	half-app	auction	MC64	half-app	auction
1	13	11	12	18	29	18	46	58	52
2	15	11	14	20	35	21	79	68	65
4	15	11	13	20	32	21	62	83	64
8	15	11	13	20	31	22	69	77	71
16	13	11	15	19	29	19	75	69	101
32	15	11	15	21	37	21	79	68	82
64	15	11	13	26	32	21	86	76	78

Table 6: Test problems MAT1, MAT2, and MAT3: number of FCG iterations for the three matching algorithms.

core is much smaller because in this case the smoother reduces to an ILU factorization. A speedup of 42.8 is achieved for PRESS1, which reduces to 29.3 for the larger matrix PRESS2; in our opinion this can be considered satisfactory, given the memory-bound nature of the computation.

WP4 code: ParFlow

The aggregation-based multilevel preconditioners were also tested on three linear systems coming from the numerical simulation of the filtration of 3D incompressible single-phase flows through anisotropic porous media, performed at the Jülich Supercomputing Centre (JSC) within WP4. The linear systems arise from the discretization of an elliptic equation with no-flow boundary conditions, modelling the pressure field, which is obtained by combining the continuity equation with Darcy's law [AGL07]. The discretization is performed by a cell-centered finite volume scheme (two-point flux approximation) on a Cartesian grid. The systems considered here have spd matrices with size 10^6 and a classical seven-diagonal sparsity pattern, with 6940000 nonzero entries. The anisotropic permeability tensor in the elliptic equation is randomly computed from a lognormal distribution with mean 1 and three standard deviation values, i.e., 1, 2 and 3, corresponding to three systems, denoted by MAT1, MAT2 and MAT3. The systems are generated by using a Matlab code implementing the basics of reservoir simulations and can be regarded as simplified samples of systems arising in ParFlow, an integrated parallel watershed computational model for simulating surface and subsurface fluid flow, currently used at JSC.

The preconditioner used in this case differs from the previous one for the choice of the hybrid backward and forward Gauss-Seidel methods as pre-smoother and post-smoother, respectively. The remaining testing details are the same as in the previous case. The number of FCG iterations obtained using the preconditioner variants corresponding to the three matching algorithms are reported in Table 6. The corresponding execution times and speedup values are shown in Table 7.

In general, the preconditioned FCG solver shows reasonable algorithmic scalability, i.e., for all systems, the number of iterations does not vary too much with the number of processes. A larger variability in the iterations can be observed with MAT3, due to the higher anisotropy of this problem and its interaction with the decoupled aggregation

MAT1						
MC64		half-app		auction		
np	time	sp	time	sp	time	sp
1	18.58	1.0	8.72	1.0	8.90	1.0
2	9.67	1.9	4.43	2.0	4.71	1.9
4	5.30	3.5	2.68	3.2	2.75	3.2
8	2.66	7.0	1.42	6.1	1.32	6.7
16	1.05	17.7	0.71	12.2	0.73	12.2
32	0.67	27.9	0.52	16.8	0.52	17.2
64	0.43	43.0	0.43	20.4	0.39	22.9

MAT2						
MC64		half-app		auction		
np	time	sp	time	sp	time	sp
1	19.54	1.0	12.46	1.0	9.73	1.0
2	11.16	1.8	6.61	1.9	5.58	1.7
4	5.55	3.5	4.05	3.1	3.14	3.1
8	2.79	7.0	2.02	6.2	1.63	6.0
16	1.18	16.6	1.07	11.7	0.77	12.6
32	0.75	26.2	0.82	15.3	0.60	16.1
64	0.51	38.5	0.60	20.7	0.40	24.1

MAT3						
MC64		half-app		auction		
np	time	sp	time	sp	time	sp
1	25.15	1.0	18.11	1.0	15.61	1.0
2	16.24	1.5	10.07	1.8	9.48	1.6
4	8.29	3.0	7.17	2.5	5.63	2.8
8	4.41	5.7	3.66	5.0	3.24	4.8
16	1.88	13.4	1.43	12.6	2.05	7.6
32	1.25	20.2	1.16	15.7	1.07	14.6
64	0.82	30.6	0.82	22.2	0.83	18.7

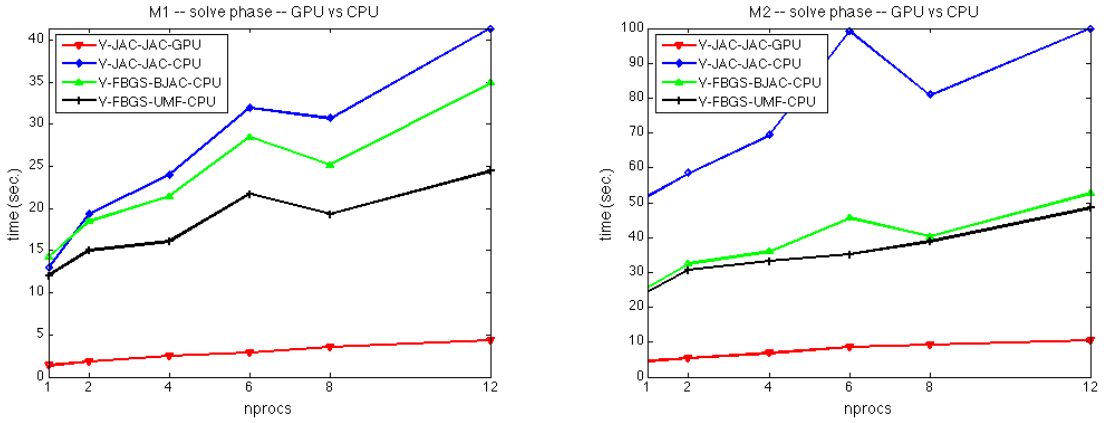
Table 7: Test problems MAT1, MAT2, and MAT3: execution time and speedup for the three matching algorithms.

strategy. None of the three matching algorithms yields preconditioners that are clearly superior in reducing the number of FCG iterations; indeed, for these systems there is no advantage in using the optimal matching algorithm implemented in MC64, since the non-optimal ones appear very competitive. The times corresponding to the half-approximation and auction algorithms are generally smaller, mainly because the time needed to build the corresponding AMG hierarchies is reduced. The speedup decreases as the anisotropy of the problem grows, because of the larger number of FCG iterations. The highest speedups are obtained with MC64, because of the larger time required by MC64 on a single core.

In conclusion, the results discussed so far show the potential of parallel matching-based aggregation and provide a basis for further work in this direction, such as the application of non-decoupled parallel matching algorithms.

5.4 PSBLAS and MLD2P4 GPU plugins

Two recently developed GPU plugins are now being tested and integrated. The PSBLAS plugin allows for transparent execution of linear algebra operators on NVIDIA GPGPUs, whilst the MLD2P4 plugin implements multiple approximate inverse algorithms for use in conjunction with the MLD2P4 smoothers and solvers. To use the GPU plugins it is only needed to declare and link the relevant data structures into the main application; neither the main application nor the library needs any other coding changes. Currently, the linear system solve phase only is implemented on GPUs.



MAT1 MAT2

Figure 27: GPU vs CPU execution times (secs.) on linear systems from ParFlow.

Preliminary tests to evaluate the performance on GPUs were carried out on the yoda cluster operated by ICAR-CNR, which is equipped with 2 NVIDIA K20M GPUs per node. The linear systems considered here were generated by using a parallel Fortran module, based on PSBLAS functionalities for matrix/vector data management, specifically developed to reproduce the same type of systems resulting from the aforementioned Matlab code for building test cases. This allowed us to build larger instances of the test problems. In particular, in our experiments we used instances of MAT1 and MAT2 such that number of matrix rows per process was kept equal to 4 million, achieving a system dimension of 48 million with 12 processes.

The matrices were preconditioned by using a V-cycle with decoupled smoothed aggregation. A simple point-wise Jacobi smoother and coarsest-level solver (1 and 10 sweeps, respectively) was used on GPUs. The same V-cycle with the hybrid forward/backward Gauss-Seidel smoother (1 sweep), coupled both with the distributed coarsest solver based on 10 sweeps of block-Jacobi (BJAC), with ILU(0) on the blocks, and UMFPACK on the replicated coarsest matrix, respectively, was also used on CPUs for comparison. The zero vector was used as starting guess and the preconditioned CG iterations were stopped when the 2-norm of the residual achieved a reduction by a factor of 10^{-6}. A pure row-block distribution of the matrices was applied. A strong reduction of the execution time was observed when the solve phase was run on GPUs, as shown Fig. 27.

We also began investigating the use of smoothers based on approximate inverses [BF16] within algebraic multilevel preconditioners on linear systems arising from ParFlow. Preconditioning by approximate inverses has a long history, and many algorithms and criteria have been proposed in the literature to build such approximations. On more traditional computing platforms, incomplete factorizations are more widely used, as they tend to be easier to build and provide better convergence properties; however they require an efficient implementation of the solution of a linear system with a sparse triangular coefficient matrix. These triangular linear systems are very difficult to solve efficiently on platforms such as GPGPUs; therefore the approximate inverses have an advantage, since they use the sparse matrix-vector product, for which very efficient kernels are available (see [FCBF16]). We run a set of tests on the matrices coming from the EoCoE applications, combining the algebraic multigrid framework we have discussed with approximate inverse smoothers, ob-

taining very promising performance data. The speed increase coming from the execution of the individual computational kernels on the GPU is such that a good speedup is obtained, despite an increase in the number of iterations to solution. A preliminary analysis was presented at the EoCoE workshop in the European HPC Summit 2017 in Barcelona and further experiments are in progress.

References

[AGL07] J. E. Aarnes, T. Gimse, and K.-A. Lie. An introduction to the numerics of flow in porous media using matlab. In E. Quak G. Hasle, K.-A Lie, editor, *Geometric Modelling, Numerical Simulation, and Optimization*, pages 265–306. Springer, 2007.

[Ber88] D. P. Bertsekas. The auction algorithm: a distributed relaxation method for the assignment problem. *Annals of Operations Research*, 14:105–123, 1988.

[BF16] D. Bertaccini and S. Filippone. Sparse approximate inverse preconditioners on high performance gpu platforms. *Comput. Math. Appl.*, 71:693–711, 2016.

[DdSF10] P. D'Ambra, D. di Serafino, and S. Filippone. Mld2p4: a package of parallel multilevel algebraic domain decomposition preconditioners in fortran 95. *ACM Trans. Math. Softw.*, 37:30:1–30:23, 2010.

[DV13] P. D'Ambra and P. S. Vassilevski. Adaptive amg with coarsening based on compatible weighted matching. *Comput. Visual Sci.*, 16:59–76, 2013.

[DV16] P. D'Ambra and P. S. Vassilevski. Adaptive amg based on weighted matching for systems of elliptic pdes arising from displacement and mixed methods. In Russo G. et al., editor, *Progress in Industrial Mathematics at ECMI 2014*, volume 22 of *Mathematics in Industry*, pages 1013–1020, Berlin/Heidelberg, Germany, 2016. Springer-Verlag.

[ea01] M. Vásquez et al. Alya: Multiphysics engineering simulation toward exascale. *J. Comput. Sci.*, 14:15–27, 201.

[FB12] S. Filippone and A. Buttari. Object-oriented techniques for sparse matrix computations in fortran 2003. *ACM Transactions on Mathematical Software*, 38:23:1–23:23, 2012.

[FC00] S. Filippone and M. Colajanni. Psblas: A library for parallel linear algebra computation on sparse matrices. *ACM Transactions on Mathematical Software*, 26:527—-550, 2000.

[FCBF16] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo. Sparse matrix-vector multiplication on gpgpus. *ACM Transactions on Mathematical Software*, 43:30:1–30:30, 2016.

[HDdSF18] A. Abdullahi Hassan, P. D'Ambra, D. di Serafino, and S. Filippone. Parallel aggregation based on compatible weighted matching for amg. In I. Lirkov and S. Margenov, editors, *Large Scale Scientific Computing. LSSC 2017.*, volume 10665 of *Lecture Notes in Computer Science*, pages 563—-571, Cham, Switzerland, 2018. Springer.

[HS15] J. Hogg and J. Scott. On the use of suboptimal matchings for scaling and ordering sparse symmetric matrices. *Numer. Linear Algebra Appl.*, 22:648–663, 2015.

[Not00] Y. Notay. Flexible conjugate gradients. *SIAM J. Sci. Comput.*, 22:1444–1460, 2000.

[Not10] Y. Notay. An aggregation-based algebraic multigrid method. *Electronic Transactions on Numerical Analysis*, 37:123–146, 2010.

[P. 17] P. D'Ambra and D. di Serafino and S. Filippone. Mld2p4 rel. 2.1, user's and reference guide, 2017. https://github.com/sfilippone/mld2p4-2/.

[Pre99] R. Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In J. Dongarra, K. Madsen, and J. Wasniewski, editors, *STACS'99*, volume 1563 of *Lecture Notes in Computer Science*, pages 259—-269, Berlin, Germany, 1999. Springer-Verlag.

[RST00] C. Tong R. S. Tuminaro. Parallel smoothed aggregation multigrid: Aggregation strategies on massively parallel machines. In J. Donnelley, editor, *SuperComputing 2000*, pages 267—-294, 2000.

[VMB96] P. Vaněk, J. Mandel, and M. Brezina. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. *Computing*, 56(3):179–196, 1996.

6. AMG for Stokes problems

Contributor	Yvan Notay (ULB)

6.1 Introduction

A method has been developed for solving stationary or time-dependent discrete Stokes equations. It uses one of the standard flavors of algebraic multigrid (AMG) for coupled partial differential equations, which, however, is not applied directly to the linear system stemming from discretization, but to an equivalent system obtained with a simple algebraic transformation (which may be seen as a form of pre-conditioning in the literal sense). Whereas the approach can in principle be combined with any type of algebraic multigrid scheme, an investigation of the properties of the coarse grid matrices reveals that plain aggregation has to be preferred to maintain nice two-grid convergence at coarser levels. Eventually, numerical experiments are reported showing that the resulting method is both robust and cost effective, being significantly faster than a state-of-the-art competitor which combines MINRES with optimal block diagonal preconditioning. More details, an extended bibliographic discussion, and a complete mathematical analysis can be found in [Not17].

6.2 Stokes equations and their discretization

We consider the following problem: find the velocity vector $\mathbf{u}$ and the pressure field p satisfying

$$\begin{aligned} \xi\mathbf{u} - \nu\Delta\mathbf{u} + \nabla p &= \mathbf{f}, \quad \text{in } \Omega\,, \\ \nabla\cdot\mathbf{u} &= 0, \quad \text{in } \Omega\,, \end{aligned} \tag{11}$$

and appropriate boundary conditions on $\partial\Omega$. In (11), Ω is a bounded domain of $\mathbb{R}^2$ or $\mathbb{R}^3$, $\mathbf{f}$ represents a prescribed force, and the parameters $\nu > 0$ (viscosity) and $\xi \geq 0$ are given. The latter is often a quantity proportional to the inverse of the time-step in an implicit time integration method applied to a nonstationary Stokes problem; $\xi = 0$ corresponds to the classical stationary Stokes problem.

In this work, we focus on standard finite difference and nodal finite elements discretizations, which lead to a linear system of the form

$$\begin{pmatrix} A & B^T \\ B & -C \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ \mathbf{p} \end{pmatrix} = \begin{pmatrix} \mathbf{b_u} \\ \mathbf{b_p} \end{pmatrix}. \tag{12}$$

In this system matrix, A is the discrete representation of the operator $\xi - \nu\Delta$; more precisely, A is block diagonal with one diagonal block per spatial dimension, being the discrete operator acting on one of the velocity components. It further follows that A is symmetric and positive definite (SPD). The matrix block B^T is the discrete gradient and $(-B)$ the discrete divergence; C is a stabilization term which is needed by some discretization schemes to avoid spurious solutions. Such spurious solutions arise when the discrete gradient admits more than the constant vector in its null space or near null space; i.e., when the discrete gradient is zero or near zero for some spurious pressure modes. The existence of such modes depends on which discretization scheme is used for velocities and pressure.

Test problems. Results are illustrated with experiments based on the following two particular discretizations of (11), the second of which being considered in both two- and

three-dimensional versions. (In the numerical results section, we additionally consider some examples of finite element discretizations with unstructured mesh.)

MAC scheme (2D). *In this problem,* Ω *is the unit square,* $\nu = 1$ *, and one imposes Dirichlet boundary conditions for all velocity components. One uses the marker and cell (MAC) finite difference discretization on a uniform staggered grid with mesh size* h *. As this scheme is naturally stable,* $C = 0$ *in this example.*

Collocated grid (2D/3D). *In this problem,* Ω *is the unit square/cube,* $\nu = 1$ *, and one imposes Dirichlet boundary conditions for all velocity components. One uses the standard finite difference discretization on a collocated uniform grid with mesh size* h *. With this scheme, all unknowns are located at the vertices of grid cells, which makes the discretization somewhat easier but induces the presence of spurious pressure modes with zero discrete divergence. Hence a form of stabilization is required and, according to the discussion in [LSS88],* C *is set equal to the five/seven point discretization of* $(16\,\nu)^{-1}h^2\Delta$ *(with Neumann boundary conditions).*

6.3 Algebraic transformation

As preliminary step, we first change the sign of the last block of rows in (12), yielding

$$\begin{pmatrix} A & B^T \\ -B & C \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ \mathbf{p} \end{pmatrix} = \begin{pmatrix} \mathbf{b_u} \\ -\mathbf{b_p} \end{pmatrix} . \tag{13}$$

Next, let $D_A = \mathrm{diag}(A)$, and perform the change of variables

$$\begin{pmatrix} \mathbf{u} \\ \mathbf{p} \end{pmatrix} = \begin{pmatrix} I & -D_A^{-1}B^T \\ & I \end{pmatrix} \begin{pmatrix} \widehat{\mathbf{u}} \\ \widehat{\mathbf{p}} \end{pmatrix} . \tag{14}$$

This leads to the transformed system

$$\widehat{\mathcal{A}} \begin{pmatrix} \widehat{\mathbf{u}} \\ \widehat{\mathbf{p}} \end{pmatrix} = \begin{pmatrix} \mathbf{b_u} \\ -\mathbf{b_p} \end{pmatrix} , \tag{15}$$

where

$$\widehat{\mathcal{A}} = \begin{pmatrix} A & \widehat{B}^T \\ -B & \widehat{C} \end{pmatrix} = \begin{pmatrix} A & B^T \\ -B & C \end{pmatrix} \begin{pmatrix} I & -D_A^{-1}B^T \\ & I \end{pmatrix} = \begin{pmatrix} A & (I - A\,D_A^{-1})B^T \\ -B & C + B\,D_A^{-1}B^T \end{pmatrix} . \tag{16}$$

The approach then solves the above system with a monolithic multigrid method using an unknown-based type coarsening [Cle05, RS87], in which the prolongation operator is set up by considering separately the different types of unknowns (in the present case, velocity components and pressure).

6.4 AMG methods with unknown-based coarsening

We first briefly describe multigrid methods for a general linear system

$$\mathcal{A}\,\mathbf{x} = \mathbf{b} \tag{17}$$

(without special structure). These methods are based on the recursive use of a two-grid method, which combines smoothing iterations with a coarse grid correction [TOS01].

Smoothing iterations are simple stationary iterations with a standard preconditioner. The coarse grid correction is based on solving a coarse representation of the problem with a reduced number of unknowns.

With the AMG methods, this correction is entirely determined by the prolongation matrix $\mathcal{P}$, of dimension $n \times n_c$, where n_c is the number of coarse unknowns. It extends to the fine grid a vector defined on the coarse space. The reverse operation is performed with the transpose of $\mathcal{P}$.

As indicated above, the two-grid method is rarely used as such, but rather constitutes a building block for developing a multigrid scheme. In fact, the application of the two-grid method requires solving systems with the coarse grid matrix. Within a multigrid algorithm, instead of the exact solution, one uses the approximation obtained by performing 1 or 2 iterations with the same two-grid method, but applied this time at the coarse level. This thus brings us to a coarser level, and the process is then repeated until the coarse system is sufficiently small so that an exact solution can be obtained at low cost. The chosen iterative scheme to solve the coarse systems defines the multigrid cycle: the V-cycle is obtained with one stationary iteration, the W-cycle with two stationary iterations, and the K-cycle [NV08] with two iterations accelerated by a Krylov subspace method.

With AMG schemes, the prolongation $\mathcal{P}$ is not fixed by the geometry, but obtained by applying appropriate algorithms to the system matrix. The corresponding solvers can then be used in black box mode. Note that these algorithms must also be used recursively: once $\mathcal{P}$ has been obtained for the fine grid on the basis of $\mathcal{A}$, the coarse grid matrix $\mathcal{A}_c$ is computed via $\mathcal{A}_c = \mathcal{P}^T \mathcal{A} \mathcal{P}$, and then one has to apply again the algorithm to $\mathcal{A}_c$ to obtain the prolongation at this coarse level, and so on.

These algorithms have been mainly developed for matrices corresponding to the discretization of scalar PDEs. For systems of PDEs, the unknown-based coarsening approach deals separately with the different types of unknowns, defining a prolongation for each type based on the corresponding diagonal block in the system matrix. Thus, for the transformed matrix $\widehat{A}$, we will let

$$\mathcal{P} = \begin{pmatrix} P_A & \\ & P_{\widehat{C}} \end{pmatrix}, \tag{18}$$

and define P_A based on the A block and $P_{\widehat{C}}$ based on the $\widehat{C}$ block. Note that the approach remains of black box type, but it is necessary to provide the solver with a properly ordered matrix and indicate the size of the different blocks.

As A is block diagonal with each diagonal block corresponding to a discrete negative Laplacian, the standard coarsening algorithms will work well. This is less clear for $\widehat{C} = C + B\,D_A^{-1}B^T$, since the structure of this term depends on the discretization scheme. However, the dominant term will be often $B\,D_A^{-1}B^T$ since C, when it is nonzero, is just a stabilization term, so in principle small. Furthermore, since $(-B)$ is a discrete representation of the divergence, and B^T a discrete representation of the gradient, their product is close to a negative Laplacian $(-\Delta_h)$. Therefore, usually, $\widehat{C}$ will also have a favorable structure for the application of AMG methods. This is the first expected benefit of the transformation: without it, the unknown-based coarsening must compute P_C on the basis of C. It is therefore not usable when $C = 0$ and hazardous otherwise, as C does not necessarily have a favorable structure.

To make the above discussion more concrete, consider for example the MAC scheme. Then

$\widehat{C} = B\,D_A^{-1}B^T$ corresponds, up to a scaling factor, to the standard finite difference formula for the discrete negative Laplacian (see below for more details). With finite element discretizations, it is harder to directly connect $\widehat{C}$ with a discrete Laplacian. However, for (stabilized) Q_1/P_0 and (stable) Q_2/Q_1 mixed finite elementson regular 2D grids, we checked that $\widehat{C}$ has nonnegative row-sum and nonpositive offdiagonal entries at least away from domain boundaries — such properties often suffice to ensure the proper functioning of AMG methods. On the other hand, more positive offdiagonal entries appear with (stabilized) P_1/P_1 and (stable) Crouzeix-Raviart elements [CR73] (as described in [ESW05], where they are denoted P_{2*}/P_{-1}). However, negative offdiagonal entries still dominate whereas below we show with two examples that the proposed approach may work well with these discretizations.

6.5 Two-grid method at coarser levels

Here we discuss the application of the two-grid method at coarser levels; that is, to solve systems with the coarse grid matrix obtained after one or several coarsening steps.

After one step, the coarse grid matrix is $\widehat{\mathcal{A}}_c = \mathcal{P}^T\widehat{\mathcal{A}}\mathcal{P}$; i.e., its structure depends upon the components P_A and $P_{\widehat{C}}$ of $\mathcal{P}$. When the prolongation is set up with an AMG algorithm, it is never fully structured even if the matrix stems from a constant coefficient PDE discretized on a uniform grid. Accordingly, a rigorous analysis of coarse grid matrices is generally untractable. However, in the present context, significant insight can be gained by considering the model geometric prolongations that AMG methods aim at imitating.

Thus, for constant coefficient isotropic problems, classical AMG methods along the lines of [BMR84, Stü01] tend to reproduce the standard $h - 2\,h$ coarsening and the associated geometric bilinear interpolation.[5] Using this latter on our test problems, we observed that, throughout coarsening steps, the nature of the different matrix blocks is preserved: away from the boundaries, one obtains regular stencils corresponding to Laplace operators for the diagonal blocks, and to first order derivatives for the offdiagonal blocks. However, the scaling of the blocks is not as in the original transformed matrix $\widehat{\mathcal{A}}$: the weight of the entries in the offdiagonal blocks relative to the entries in the diagonal blocks (e.g., the main diagonal) is increased by a factor of about two with each coarsening step. This stems from the type of discrete operators involved. The diagonal blocks correspond to scaled discrete Laplace operators. With finite differences, the entries are $\mathcal{O}(h^{-2})$, hence they are reduced by a factor of 4 with $h - 2\,h$ coarsening. On the other hand, offdiagonal blocks correspond to discrete first order differential operators, with $\mathcal{O}(h^{-1})$ entries that are reduced by a factor of 2 only. The picture is the same with finite element discretizations, where the entries are $\mathcal{O}(h^{d-2})$ for second order differential operators and $\mathcal{O}(h^{d-1})$ for first order ones, where d is the spatial dimension of the problem.

This relative increase of the weight of the offdiagonal blocks has a dramatic impact on the potentialities of the AMG method at coarser levels. To see this, observe that applying AMG to $\widehat{\mathcal{A}}$ would be just trivial if the offdiagonal blocks vanished. On the other hand, if the offdiagonal blocks dominate the diagonal ones, standard smoothers (as used with AMG methods) can just cease to converge (see Figure 28 below).

[5] For 5-point stencil in 2D, standard Ruge–Stüben AMG will produce a red-black coarsening of the fine mesh, and $h - 2\,h$ coarsening only at subsequent levels; however, for such small stencils, one often prefers aggressive coarsening, which imitates the $h - 2\,h$ coarsening right away; see, e.g., the discussion in [Stü01, Section A.7.1].

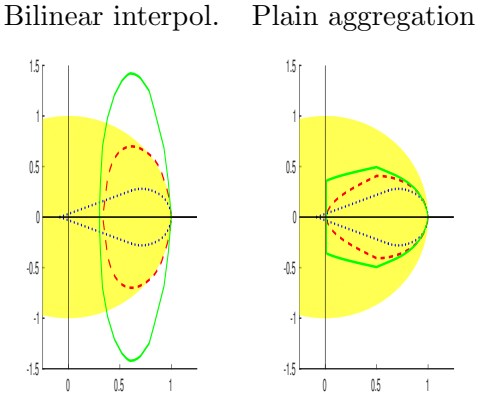


Figure 28: Convex hull of the eigenvalues of iteration matrices associated with damped Jacobi smoothing ($\omega = 0.5$) for the Collocated grid (2D) problem with $\xi = 0$; ····: fine grid level; - - -: first coarse level; ——: next coarse level; the yellow (shaded) region is a portion of the unit disk centered at the origin.

However, we observed that this phenomenon does not happen anymore when using coarsening by plain aggregation. The relative weights of the blocks remain as in the original matrix, which we explain by the well-known overweighting of coarse grid matrices associated with plain aggregation for second order differential operator; see, e.g., the discussion in [MN08], and the proposition in [Bra95] to overrelax coarse grid correction terms by a factor close to two, to compensate for this phenomenon.

Whereas this overweighting is sometimes seen as a weak point of aggregation-based AMG methods, here it comes as a good news. Thanks to it, the structure of the matrix remains roughly similar at the successive coarse levels, and, hence, one may expect that the recursive use of the two-grid method does not raise particular difficulty in this context.

These considerations are illustrated in Figure 28, where, considering the Collocated grid (2D) problem with $\xi = 0$, we depict the convex hull of the iteration matrices associated with damped Jacobi smoothing at several levels. Clearly, with (geometric) bilinear interpolation, there is a severe divergence from the second coarse level, whereas nothing particular happens with (geometric, boxwise) plain aggregation coarsening.

6.6 Multigrid strategy

For scalar PDEs, there are many valuable methods based either on classical AMG methods along the lines of [BMR84, Stü01], or on smoothed aggregation AMG [VMB96]. However, the developments in the preceding section show that their use is somehow uneasy as unknown-based coarsening strategy for the transformed matrices considered in this work.

It seems then sensible to prefer the alternative offered by plain aggregation-based AMG. Moreover, recent results about this approach show that it is already competitive for scalar, second-order, elliptic PDEs, at least when carefully used [NN14, Not10]; that is, when applying an aggregation algorithm that takes into account the matrix entries so as to build only high quality aggregates [NN12, Not12]. It is also important to use the K-cycle [NV08] to ensure that the optimal two-grid convergence [MN08, NN12, Not12] carries over the full multigrid scheme; i.e., to ensure that the convergence is independent of the number of levels.

This leads us to the following multigrid strategy: two-grid scheme obtained from the combination of symmetrized Gauss-Seidel smoothing with unknown-based coarsening by plain aggregation, using more specifically the algorithm in [Not00b]; multigrid scheme as in [Not10], obtained with the standard K-cycle for nonsymmetric matrices, in which all coarse systems are solved with two GCR iterations [EES83] using the two-grid preconditioner at the considered level (except at the coarsest level where a sparse direct solver is used).

In fact, this method requires only a slight modifications of the method in [Not00b], needed to implement the unknown-based coarsening. Furthermore, the corresponding code is available as the "block" version of the AGMG software [Not], which we therefore used for the numerical experiments reported in the next section.

6.7 Numerical results

In all cases, the multigrid method is used as a preconditioner for the GCR method restarted each 10 or each 30 iterations [EES83]. The right hand side is a vector with random velocity components and zero pressure components, the initial approximation is the zero vector, and iterations are stopped when the relative residual norm is below 10^{-6}. All results are reported for two different grid sizes: in the 2D examples, *Size 1* and *Size 2* refer to, respectively, $h^{-1} = 256$ and $h^{-1} = 1024$, whereas, in the 3D example, they refer to $h^{-1} = 48$ and $h^{-1} = 96$. Hence the number of unknowns ranges approximately from 2×10^5 to 3×10^6 in 2D, and from 4×10^5 to 3.5×10^6 in 3D (i.e., there is about one order of magnitude between Size 1 and Size 2).

In the side table, we report on *complexities*; that is, on the memory usage involved by the solution method. Two factors have to be taken into account. Firstly, the transformed matrices have more nonzero entries than the original system matrix; in the table, the ratio between the numbers of nonzero entries in the transformed and original matrices is reported in the columns labeled "rat_{Tr}". Next, the multigrid preconditioner involves some overhead, which is characterized by the algorithm complexity $\mathcal{C}_A$, defined as the sum of all nonzero entries in the matrices at all levels divided by the number of nonzero entries in the fine grid matrix (here, the transformed matrix). Finally, in the present context, we further define the global complexity $\mathcal{C}_G$ as the sum of all nonzero entries in the matrices at all levels divided by the number of nonzero entries in the *original* fine grid matrix; in other words: $\mathcal{C}_G = \mathcal{C}_A \cdot \text{rat}_{\text{Tr}}$. Despite this cumulative effects, the complexities are acceptable in all cases.

	Size 1			Size 2		
ξ	rat_{Tr}	$\mathcal{C}_A$	$\mathcal{C}_G$	rat_{Tr}	$\mathcal{C}_A$	$\mathcal{C}_G$
MAC scheme (2D)						
0	1.9	1.3	2.5	1.9	1.3	2.6
10	1.9	1.3	2.5	1.9	1.3	2.6
100	1.9	1.3	2.5	1.9	1.3	2.6
1000	1.9	1.3	2.5	1.9	1.3	2.6
Collocated grid (2D)						
0	1.5	1.4	2.1	1.5	1.4	2.1
10	1.5	1.4	2.1	1.5	1.4	2.1
100	1.5	1.4	2.1	1.5	1.4	2.1
1000	1.5	1.4	2.1	1.5	1.4	2.1
Collocated grid (3D)						
0	1.7	1.8	3.1	1.7	1.8	3.1
10	1.7	1.7	2.9	1.7	1.7	2.9
100	1.7	1.8	3.0	1.7	1.7	2.9
1000	1.7	1.6	2.8	1.7	1.7	2.9

The iteration counts are reported in Table 8. Here, for comparison purpose, we include a standard method ("Block Diag. Prec."), which solves the original system (12) by MINRES

	Monolithic AMG				Block Diag. Prec.			
	GCR(10)		GCR(30)		1 inner it.		2 inner it.	
Size	1	2	1	2	1	2	1	2
ξ:								
				MAC scheme (2D)				
0	17	17	16	17	57	64	41	44
10	17	17	16	17	55	62	37	41
100	15	17	15	16	49	57	33	37
1000	13	16	13	15	41	49	26	33
				Collocated grid (2D)				
0	20	27	20	26	61	66	46	50
10	20	27	19	26	58	66	44	48
100	18	24	17	23	53	61	38	42
1000	14	19	14	19	43	51	33	37
				Collocated grid (3D)				
0	15	17	15	17	68	71	48	51
10	15	17	15	17	63	69	46	49
100	14	16	14	16	53	59	38	43
1000	15	14	15	13	44	46	32	34

Table 8: Number of iteration to reduce the residual norm by a factor of 10^{-6}.

with state-of-the-art block diagonal preconditioner

$$\begin{pmatrix} \widetilde{A} & \\ & \widetilde{S} \end{pmatrix},$$

where $\widetilde{A}$ is an approximation of A, and $\widetilde{S}$ is an approximation of the Schur complement $C + B\,A^{-1}B^T$ [ESW05]. The needed inverse of $\widetilde{A}$ is obtained by applying the same AMG method as for the transformed matrices; we consider either one single application of the multigrid preconditioner ("1 inner it."), or two FCG iterations [Not00a] using this preconditioner to solve a system with matrix A ("2 inner it."). The latter option allows us to investigate whether it can be cost effective to use a more costly but more accurate approximation to A, as is the case when the the inverse of $\widetilde{A}$ obtained by applying a more standard AMG method, that provides a better approximation than the plain aggregation method considered here, but at a significantly higher cost (see the discussion in [Not14]).

Regarding the Schur complement approximation, in the stationary case ($\xi = 0$), it is standard to use $\widetilde{S} = \nu^{-1}I$ with finite difference discretizations. Time-dependent cases require more care, but the Cahouet–Chabard method [CC88] is both effective and optimal with respect to the mesh size and other problem parameters [OPR06]. It uses $\widetilde{S}^{-1} = \nu\,I + \xi\big(-\widetilde{\Delta}_h\big)^{-1}$, where $\widetilde{\Delta}_h$ is an approximation to a discrete Laplace operator with Neumann boundary conditions for the pressure variables; again, for $\big(-\widetilde{\Delta}_h\big)^{-1}$, we consider either one application of the AMG preconditioner or two inner iterations, in each case applied to solve a system with an exact discrete Laplace operator $-\Delta_h$. Note that the latter has to be supplied to the solver, hence this preconditioner is not fully algebraic.

	Monolithic AMG				Block Diag. Prec.			
	GCR(10)		GCR(30)		1 inner it.		2 inner it.	
Size	1	2	1	2	1	2	1	2
ξ:								
				MAC scheme (2D)				
0	2.8	2.7	2.5	2.8	6.2	5.5	6.8	7.0
10	2.5	2.7	2.5	2.8	6.5	6.9	8.8	9.1
100	2.3	2.7	2.4	2.7	5.7	6.4	7.9	8.3
1000	2.1	2.6	2.1	2.6	4.8	5.5	6.3	7.4
				Collocated grid (2D)				
0	2.9	3.9	3.0	4.2	5.5	5.8	7.9	7.9
10	2.9	3.9	2.8	4.2	6.6	7.2	10.6	10.5
100	2.7	3.6	2.6	3.8	6.1	6.8	9.3	9.3
1000	2.3	3.1	2.4	3.3	5.1	5.8	7.9	8.3
				Collocated grid (3D)				
0	9.4	5.1	8.5	4.8	6.6	7.2	8.8	9.7
10	7.4	4.4	7.4	4.6	7.4	8.4	11.0	11.9
100	5.2	4.3	5.2	4.4	6.2	7.2	9.0	10.5
1000	6.5	4.0	6.5	3.9	4.7	5.7	6.9	8.2

Table 9: Total solution time in microseconds per unknown.

One sees that the block diagonal preconditioner requires significantly more iterations than monolithic AMG preconditioners, even when using enhanced approximations $\widetilde{A}$ and $\widetilde{S}$ with two inner iterations. This explains the timing results reported in Table 9, [6] where one sees that the methods presented in this paper are significantly faster despite a higher cost per iteration. Thus, for the largest tested size, AMG applied to the transformed matrix is between 1.4 and 2.5 times faster than MINRES with block diagonal preconditioning (using the most cost effective variant which is finally the one with a single AMG application for the diagonal blocks).

Finally, we also tested two mixed finite element discretizations of stationary Stokes problems on unstructured grids. The first one is P_{2^*}/P_{-1} based on Crouzeix-Raviart elements [CR73], as described in [ESW05]; the domain is the unit square, and a zoom on the central part of the mesh is displayed on Figure 29 (left). The second discretization is P_1/P_1 stabilized according to the method in [BDG06]; the domain is the unit cube, and Figure 29 (right) offers a view on a cut of the mesh displaying one eighth of it.

For these matrices, the block diagonal preconditioning method requires the pressure mass matrix, which was not provided. Hence, our approach is tested here. Results are reported in Table 10, where we also reproduce the results obtained with finite difference discretizations (largest sizes). One sees that the method reaches similar efficiency for the mixed finite elements, except that the ratio between the numbers of nonzero entries in the trans-

[6] Timings are reported for a standard Linux workstation equipped with Intel i5-4570 @ 3.20GHz processor and 32 Gb DDR RAM memory; solvers were called from the Matlab environment, but all computationally intensive routines are written in Fortran and have been compiled with the GNU compiler (gfortran).

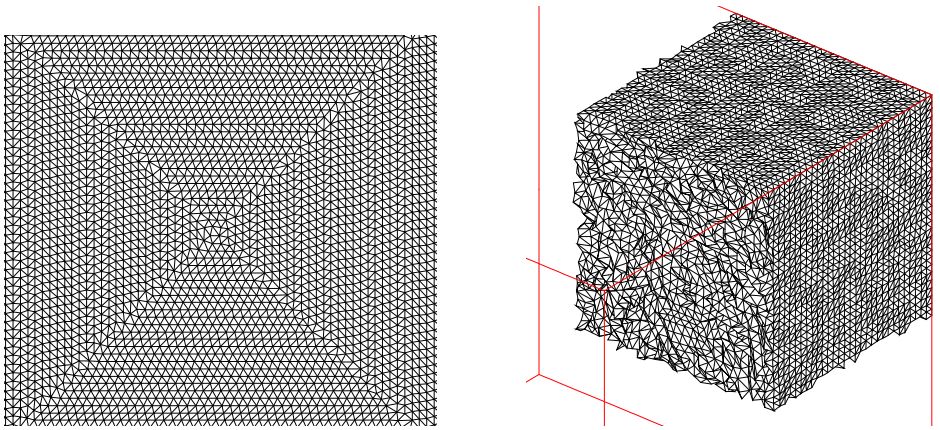


Figure 29: Zoom on the central part of the 2D unstructured mesh (left), and view of a cut of the unstructured 3D mesh, displaying one eighth of it.

						GCR(10)			GCR(30)		
	$\frac{n}{10^6}$	$\frac{n}{nnz}$	rat_{Tr}	$\mathcal{C}_A$	$\mathcal{C}_G$	it	$\frac{\text{tm}}{n}$	$\frac{\text{tm}}{nnz}$	it	$\frac{\text{tm}}{n}$	$\frac{\text{tm}}{nnz}$
MAC(2D)	3.1	6.0	1.9	2.3	2.5	17	2.7	0.44	17	2.8	0.47
Coll.(2D)	3.1	7.7	1.5	2.4	2.1	27	3.9	0.51	26	4.2	0.55
Coll.(3D)	3.5	9.9	1.7	1.8	3.1	17	5.1	0.51	17	4.8	0.49
P_{2^*}/P_{-1}(2D)	1.3	17.3	3.4	1.5	4.9	22	14.3	0.83	22	14.5	0.84
P_1/P_1(3D)	0.8	36.3	2.5	1.8	4.6	23	32.3	0.89	22	32.1	0.88

Table 10: Results with the transformation for the stationary problems on structured and unstructured meshes; "tm" refers to the total solution time and is reported in microsecond, either per unknown ($\frac{\text{tm}}{n}$) or per nonzero entry in the original matrix ($\frac{\text{tm}}{nnz}$).

formed and original matrices (rat_{Tr}) is here somehow larger. As a consequence, the time needed to solve the system per nonzero entry *in the original matrix* is slightly less than twice the time needed for finite difference discretizations (while the time per unknown is significantly increased, on account of the much larger number of nonzero entries per row).

6.8 Conclusions

We have shown that monolithic AMG methods can be successfully applied to solve discrete Stokes equations, using the standard unknown-based coarsening approach in which the prolongation operator is set up by considering separately the different types of unknowns. Two conditions, however, are to be satisfied. Firstly, the AMG method should not be applied to the linear system stemming from the discretization, but to an equivalent system obtained through a simple algebraic transformation. Secondly, when more than two levels are needed, plain aggregation-based AMG has to be preferred, because the induced coarse level matrices are better suited to the recursive application of the method. When both these requirements are met, monolithic AMG appears both robust and cost effective with respect to state-of-the-art block preconditioning.

References

[BDG06] Pavel B. Bochev, Clark R. Dohrmann, and Max D. Gunzburger. Stabilization of low-order mixed finite elements for the Stokes equations. *SIAM J. Numer.*

Anal., 44:82–101, 2006.

[BMR84] A. Brandt, S. F. McCormick, and J. W. Ruge. Algebraic multigrid (AMG) for sparse matrix equations. In D. J. Evans, editor, *Sparsity and Its Application*, pages 257–284. Cambridge University Press, Cambridge, 1984.

[Bra95] D. Braess. Towards algebraic multigrid for elliptic problems of second order. *Computing*, 55:379–393, 1995.

[CC88] J. Cahouet and J.-P. Chabard. Some fast 3D finite element solvers for the generalized Stokes problem. *Int. J. Numer. Meth. Fluids*, 8:869–895, 1988.

[Cle05] Tanja Clees. *AMG Strategies for PDE Systems with Applications in Industrial Semiconductor Simulation.* Dissertation, Mathematisch-Naturwissenschaftlichen Fakultät, Universität Köln, Germany, 2005.

[CR73] Michel Crouzeix and P-A Raviart. Conforming and nonconforming finite element methods for solving the stationary stokes equations I. *ESAIM Math. Model. Numer. Anal.*, 7:33–75, 1973.

[EES83] Stanley C. Eisenstat, Howard C. Elman, and Martin H. Schultz. Variational iterative methods for nonsymmetric systems of linear equations. *SIAM J. Numer. Anal.*, 20:345–357, 1983.

[ESW05] H. Elman, D. Silvester, and A.J. Wathen. *Finite Elements and Fast Iterative Solvers*. Oxford University Press, Oxford, 2005.

[LSS88] J Linden, B Steckel, and K. Stüben. Parallel multigrid solution of the Navier-Stokes equations on general 2D domains. *Parallel Computing*, 7:461–475, 1988.

[MN08] A. C. Muresan and Y. Notay. Analysis of aggregation-based multigrid. *SIAM J. Sci. Comput.*, 30:1082–1103, 2008.

[NN12] A. Napov and Y. Notay. An algebraic multigrid method with guaranteed convergence rate. *SIAM J. Sci. Comput.*, 34:A1079–A1109, 2012.

[NN14] A. Napov and Y. Notay. Algebraic multigrid for moderate order finite elements. *SIAM J. Sci. Comput.*, 36:A1678–A1707, 2014.

[Not] Y. Notay. AGMG software and documentation. `http://homepages.ulb.ac.be/~ynotay/AGMG`.

[Not00a] Y. Notay. Flexible conjugate gradients. *SIAM J. Sci. Comput.*, 22:1444–1460, 2000.

[Not00b] Y. Notay. A robust algebraic multilevel preconditioner for non symmetric *M*-matrices. *Numer. Linear Algebra Appl.*, 7:243–267, 2000.

[Not10] Y. Notay. An aggregation-based algebraic multigrid method. *Electron. Trans. Numer. Anal.*, 37:123–146, 2010.

[Not12] Y. Notay. Aggregation-based algebraic multigrid for convection-diffusion equations. *SIAM J. Sci. Comput.*, 34:A2288–A2316, 2012.

[Not14] Y. Notay. A new analysis of block preconditioners for saddle point problems. *SIAM J. Matrix Anal. Appl.*, 35:143–173, 2014.

[Not17] Y. Notay. Algebraic multigrid for Stokes equations. *SIAM J. Sci. Comput.*, 2017. To appear; see `http://homepages.ulb.ac.be/~ynotay/`.

[NV08] Y. Notay and P. S. Vassilevski. Recursive Krylov-based multigrid cycles. *Numer. Linear Algebra Appl.*, 15:473–487, 2008.

[OPR06] Maxim A. Olshanskii, Jörg Peters, and Arnold Reusken. Uniform preconditioners for a parameter dependent saddle point problem with application to generalized Stokes interface equations. *Numer. Math.*, 105:159–191, 2006.

[RS87] J. W. Ruge and K. Stüben. Algebraic multigrid (AMG). In S. F. McCormick, editor, *Multigrid Methods*, Frontiers in Appl. Math. 3, pages 73–130. SIAM, Philadelphia, 1987.

[Stü01] K. Stüben. *An introduction to algebraic multigrid*, pages 413–532. In Trottenberg et al. [TOS01], 2001. Appendix A.

[TOS01] U. Trottenberg, C. W. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, London, 2001.

[VMB96] P. Vaněk, J. Mandel, and M. Brezina. Algebraic multigrid based on smoothed aggregation for second and fourth order elliptic problems. *Computing*, 56:179–196, 1996.

7. I/O benchmarking

7.1 Overview

Contributors	F. Ambrosino (ENEA), M. Brzezniak (PSNC), W. Frings (JUELICH), A. Funel (ENEA), G. Guarnieri (ENEA), M. Haefele (CEA), F. Iannone (ENEA), S. Lührs (JUELICH), T. Paluszkiewicz (PSNC), K. Sierociński (PSNC)

Beside the computational scalability of an HPC application, its I/O behaviour can significantly influence the overall performance. The I/O behaviour is influenced by many aspects, e.g. by the implementation within the application, the file system, the network and data hardware and software as well as, since storage is usually a shared resource, other jobs running on the same cluster.

To evaluate the I/O behaviour and to facilitate the testing of I/O performance improvements, an I/O benchmarking activity was established as part of the EoCoE project. This task is based on three pillars:

1. Validation of the I/O behaviour of thespecific energy oriented applications in the EoCoE project to extract typical I/O patterns
2. Benchmarking of similar I/O pattern on different computing platforms using established I/O benchmarking codes
3. Interpretation and recommendations based on the benchmarking results

The following document describes the methodology and gathers the results of this activity. According to the targets of the task there are three main sections:

For validation and for getting a general overview of the I/O behaviour, we asked several of the EoCoE project related application owners to provide information concerning the specific I/O patterns of their codes. The questions and the result of this initial process are presented in section 7.2.

Based on the information provided by the EoCoE application owners, two benchmark applications were selected - IOR and SIONlib partest - and configured to represent different I/O behaviours. The setup and the configuration as well as a system overview is presented in section 7.3 and section 7.4.

Finally section 7.5 describes the overall results for the various benchmark executions and analyses the individual runs.

7.2 I/O behaviour of EoCoE codes

I/O questionnaire

To get an overview of the typical production run I/O behaviour of the various applications within the EoCoE project, a questionnaire was sent to the applicants of the EoCoE performance evaluation workshops.

The following questions were asked within this questionnaire:

1. I/O libraries used?

2. I/O strategy (master-slave, disjoint access to files, shared access to files)?
3. Typical I/O call behaviour (collective or individual I/O)?
4. Do you use additional pre-processing / post-processing steps or workflows in your production runs that are potentially I/O demanding?
5. Reading scheme (burst vs. continuous)?
6. Writing scheme (burst vs. continuous)?
7. Checkpointing strategy implemented?
8. Size of a single checkpoint (in MB)?
9. Typical number of checkpoints for production runs?
10. Number of files generated?
11. Total size of files (in MB)?
12. Is your I/O and, more generally your data management strategy, limited by the performance of I/O libraries and/or file system?

Questionnaire results

The following graphs visualize the results of the questionnaire and of the I/O behaviour of the applications used in context of EoCoE.

Figure 30 shows the overall API distribution. Typically multiple APIs are used within the same application. Here mainly the standard APIs of the preferred programming language are directly used, which typically use the POSIX I/O layer of the specific computing system. Readable ASCII files are mainly used for log-files, configuration files or smaller output files, while binary formats are used within a self-defined (or common) data format, either by using only a single process for reading and writing or by using a number of processes to create task local files.

In the list of high-level I/O APIs, HDF5 is the most among used application. Due to its platform independent file format the library, as well as the NetCDF library, provide significant benefits when moving files within a larger workflow between different applications.

In total 15 application developers send their feedback for this questionnaire.

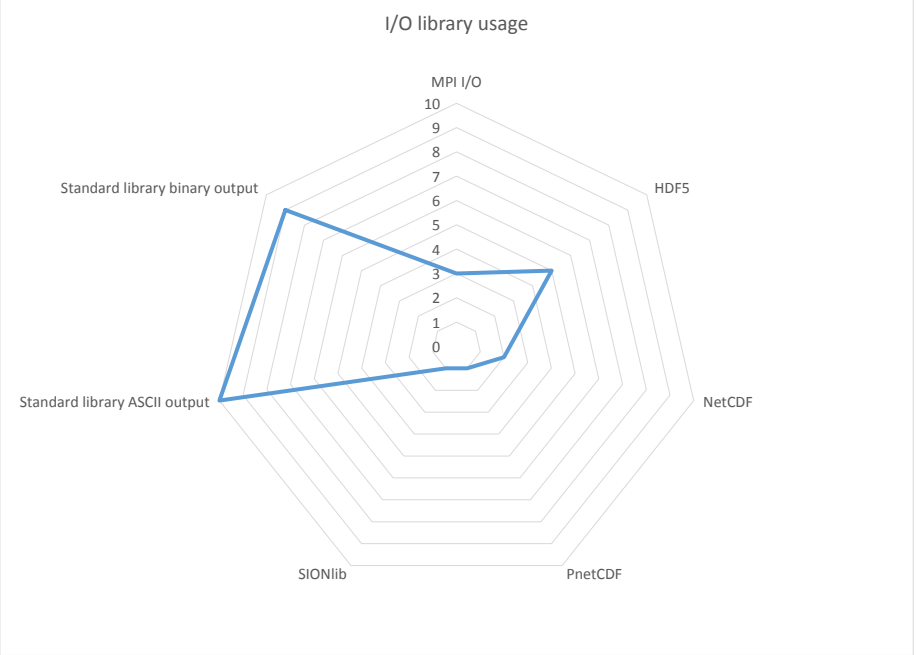


Figure 30: Distribution of I/O library usage

Figure 31 visualizes if an application uses multiple processes or a single process to access the output files. Some applications use different file access patterns for different parts of their I/O workflow. Here most of the applications use a Master-Slave behaviour e.g. to write log information or read configuration options with a single process. For the large amounts of simulation data, a parallel I/O approach either by using multiple files or by using a single shared file is used to take advantage of the parallel I/O hardware resources.

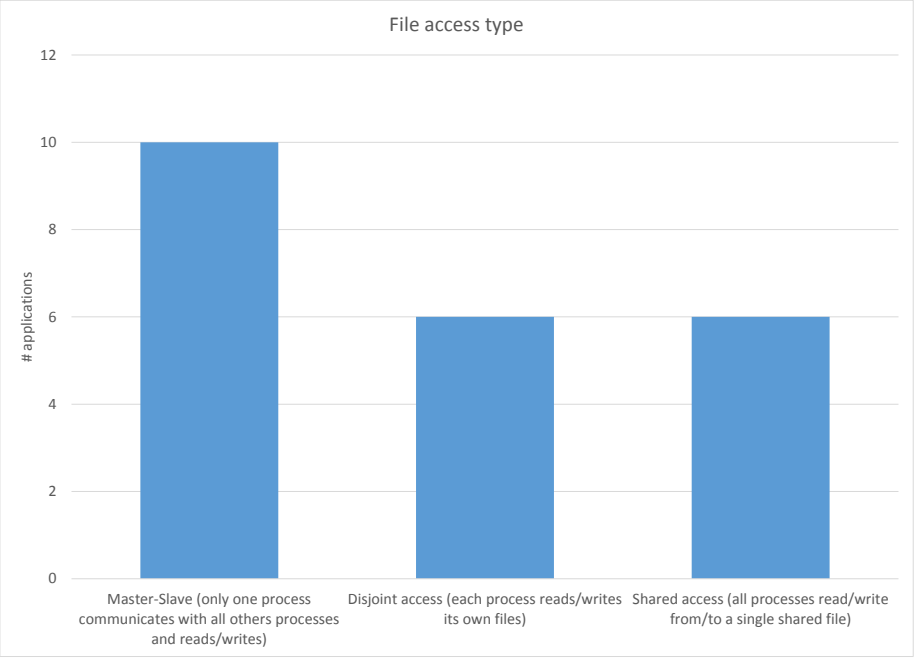


Figure 31: File access behaviour

Figure 32 shows the aggregate size of all output files, as well as the size of an individual checkpoint (if checkpointing is used within the application) for an average typical production run (here the output size can be smaller as compared to the data size needed to create an individual checkpoint). Of course the data size is highly influenced by the individual problem and should only represent a rough estimate of the individual I/O demands. Figure 33 shows the number of regular output files, which are generated within a typical run. From the data size point of view ParFlow and ESIAS (both using NetCDF) as well as SHEMAT-Suite, Gysela and OpenFOAM (using HDF5 and task local formats) have the largest I/O demands. For ParFlow and OpenFOAM also the number of files gets significantly important.

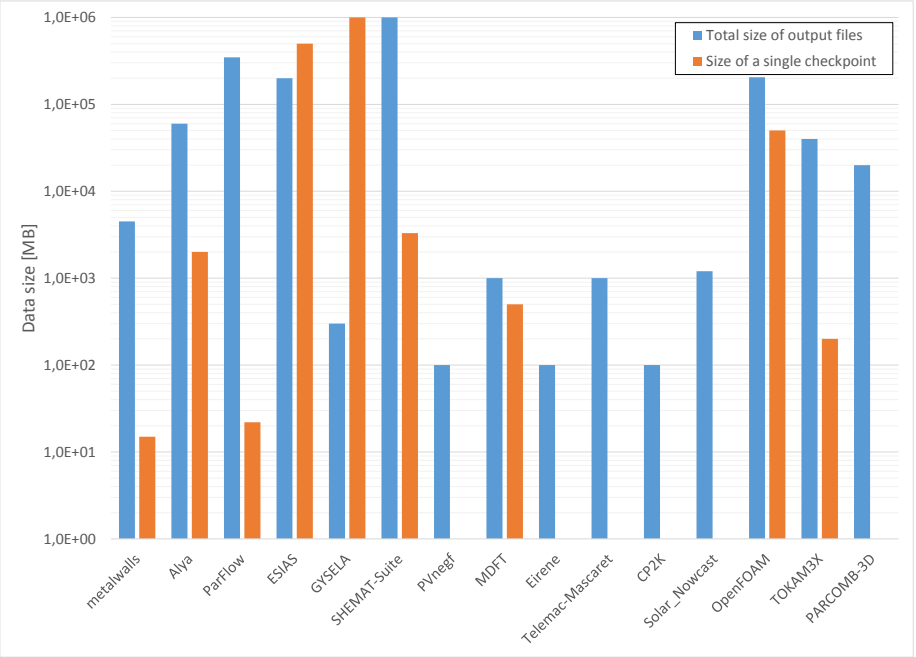


Figure 32: Data sizes overview

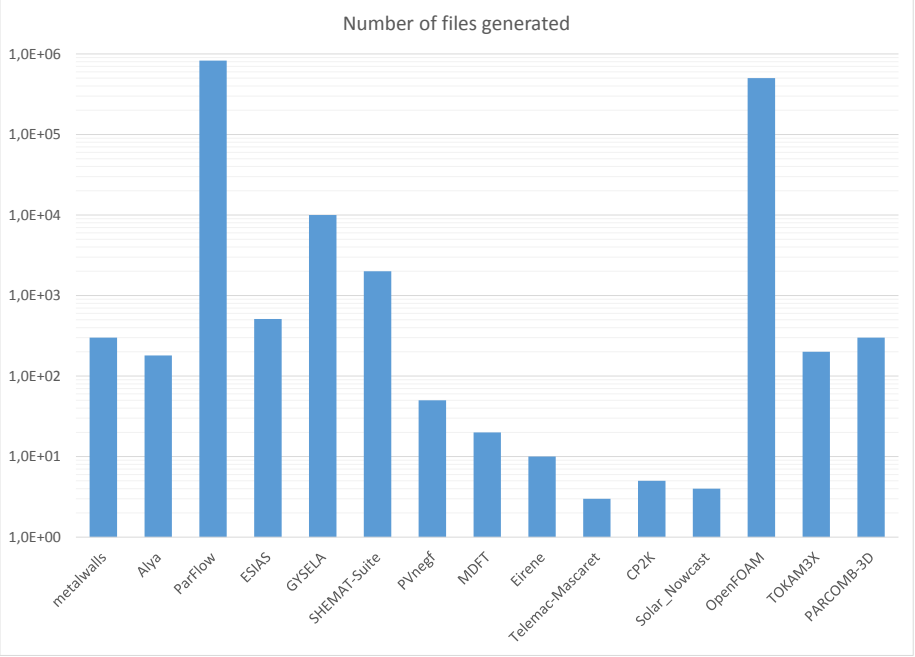


Figure 33: Number of files overview

Benchmark strategy

The questionnaire as well as the various application support activities within EoCoE show a wide variety of I/O patterns among all EoCoE applications. Using the applications directly as an I/O benchmark makes it difficult to change benchmark parameters, to investigate individual patterns and to test multiple HPC systems. Instead we used existing established generic benchmarks within this activity to reproduce different I/O patterns and to reproduce different file layouts, as the individual data distribution and file access patterns, which are hard to extract from an individual application, can have a strong influence on the overall I/O performance.

Within the benchmark configurations we tried to address different common parallel I/O topics like the difference between task local and shared file I/O or collective operations. The results should provide hints for the real applications and help to explain which I/O pattern can influence their overall performance.

7.3 Benchmark description

IOR

Overview

IOR (Interleaved Or Random) is a very popular parallel I/O benchmark program initially designed by Lawrence Livermore National Laboratory. Detailed information about IOR in its latest version can be found at: https://github.com/hpc/ior.

IOR can be used for testing the performance of parallel file systems using various interfaces (MPI-IO, HDF5, PnetCDF, POSIX) and different access patterns. IOR uses MPI for

process synchronization. An important feature of IOR is that it can simulate two basic parallel I/O strategies: shared file and one file per process. In the shared file case all processes read/write from/to a single common file, while in the one-file per process each process reads/writes from/to its own file. Depending on the selected interface, interface-specific configuration options are available. Beside the API specific parameters, IOR allows to specify the access and file layout using three major options:

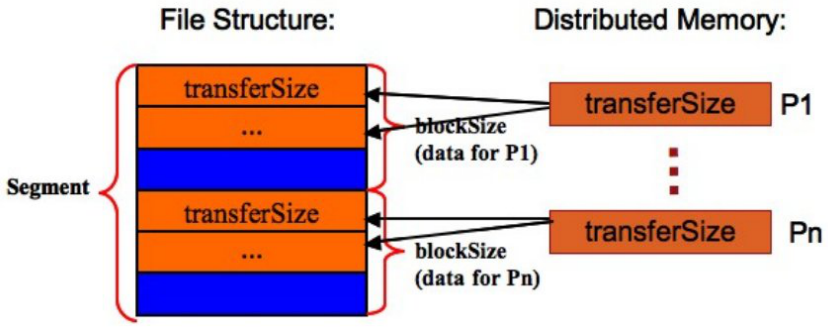


Figure 34: The IOR file structure.

Figure 34 illustrates the IOR file structure in the case of a write operation to a single shared file. An IOR file is as a sequence of segments which represent the application data (for example in an HDF5 or a PnetCDF dataset). Each segment holds the same data structure. This allows to increase the amount of benchmark data linearly with the number of used segments. Each processor holds a part of the segment called a "block". Each block, in turn, is divided into chunks called "transfer size". The transfer size is the amount of data transferred from the processor to the disk storage in a single I/O function call. IOR manages the blocks and combines them into "segments". The IOR file structure in the case of one file per process is the same except that each processor reads/writes from/to its own file.

The benchmark comes with a rich variety of options which can be passed as command line arguments to the executable. For example:

```
IOR -w -r -o outputfile
```

will perform a write and a read to the file `outputfile`. It is possible to setup a benchmark by preparing a configuration script and run it:

```
IOR -W -f script.
```

JUBE integration

To allow for a reproducible IOR execution, a comparable benchmark configuration as well as a comparable output format, the JUBE benchmarking environment (JUBE[7]) was used to run IOR on the different systems. JUBE includes the different configuration options in an IOR configuration file, creates the job script and finally submits the job to the supercomputer.

[7] http://www.fz-juelich.de/jube/

```
<parameterset name="iorParameter" init_with="ior_specs.xml">
    <parameter name="api">POSIX,MPIIO,HDF5,NCMPI</parameter>
    <parameter name="blockSize" type="int" mode="python">
        256*(1024**2)
    </parameter>
    <parameter name="transferSize" type="int" mode="python">
        ",".join(str(i) for i in [128*1024,1024**2,4*(1024**2),
                                  256*(1024**2)])
    </parameter>
    <parameter name="segmentCount" type="int">5</parameter>
    <parameter name="repetitions" type="int">3</parameter>
    <parameter name="verbose" type="int">2</parameter>
    <parameter name="filePerProc" type="int" mode="python">
        "0,1" if "$api" == "POSIX" else "0"
    </parameter>
    <parameter name="collective" type="int">0</parameter>
    <parameter name="memoryPerNode">0\%</parameter>
</parameterset>
```

Listing 1: Basic JUBE configuration for IOR

The configuration itself is shown in listing 19. JUBE supports the usage of templates (given by values separated with ,), which automatically forces the benchmark environment to test all possible different parameter combinations.

Benchmark cases

Two different benchmark cases were created to investigate different I/O behaviour.

basic: The first case (which is also shown in the example listing 19) uses a fixed block size and a fixed number of segments. The transfer size, the API and the number of processing elements are varied. This allows to have a direct performance comparison of different APIs and different transfer rates. This type of I/O is the most typical IOR configuration and represents an easy to handle I/O structure for almost all APIs. It is a good benchmark approach to measure the API overhead, metadata bottlenecks and overall bandwidth information. Figure 35 shows the benchmark behaviour, visualized using Darshan[8], for the POSIX API with one file per process, 48 processes, a transfer size of 4 MByte and 30 GByte total data. 7630 write and read accesses are needed to finalize the overall file. The pattern is nearly identical for all different APIs. All other APIs (in contrast to the POSIX task local setup) use a single file for the overall output.

[8] http://www.mcs.anl.gov/research/projects/darshan/

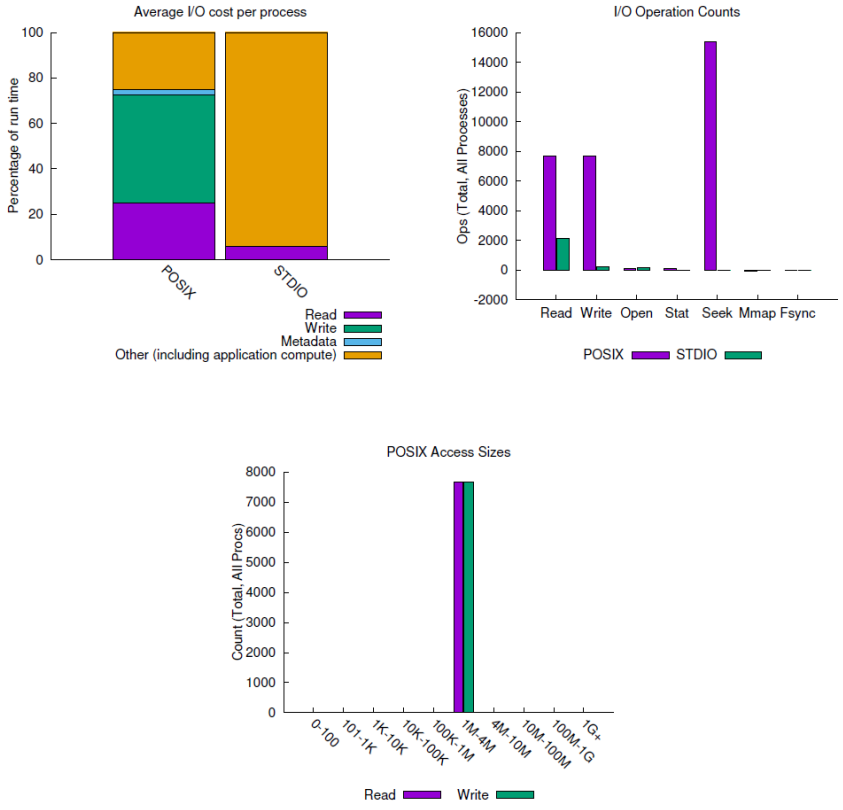
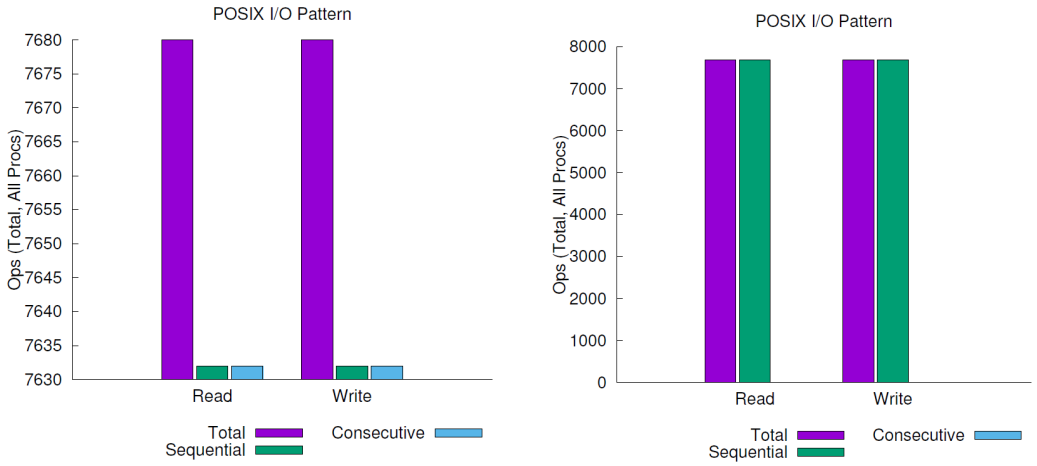


Figure 35: Darshan view of the IOR file access pattern using POSIX API and one file per process for the first benchmarking case (basic) on JURECA.

striping: The second case uses a block size equal to the size of the transfer size. The idea is to only have one transfer operation per segment shown in figure 34. To keep the overall data size stable, the number of segments is increased, corresponding to the selected transfer size. This benchmark creates a strided/striped file pattern to investigate the API behaviour as well as tuning mechanisms like collective I/O operations. Such a file layout is also present if chunks of a multi-dimensional array are written to disk. Figure 36 shows the main difference between the two benchmark cases for MPI-IO (HDF5 and PnetCDF show a similar behaviour due to their usage of MPI-IO underneath): Without the strided setup most of all I/O operations are consecutive operations (99%). Consecutive means, that the end of one I/O operation is directly followed by the begin of another I/O operation of the same process. In the strided setup the operations are still sequential but not consecutive anymore, due to the round robin setup within the file layout. Collective operations can rebuild the consecutive behaviour by combining the different I/O requests of different processes into a single request by one process. This is shown in figure 37 by allowing the benchmark to perform collective operations. Here for the read-operations 16 MByte of data (which is the default maximum MPI-IO aggregation size) are transferred at once, which reduces the number of I/O operations by a factor of 4 (because a transfer size of 4 MByte was selected, which means 4 processes can combine their data). Of course the data has to be moved between the processes afterwards by MPI-IO which needs additional time. This test was executed with an automatic collective buffering selection, that is why there is no effect on the write behaviour because the API decided to ignore the collective

approach for the writing part. More details on the collective behaviour, also for the write commands, are given in section 7.5.



(a) Number of required I/O operations for the first benchmark case (basic)

(b) Number of required I/O operations for the second benchmark case (striping)

Figure 36: Comparison of the MPI-IO file access behavior between the two IOR benchmark cases.

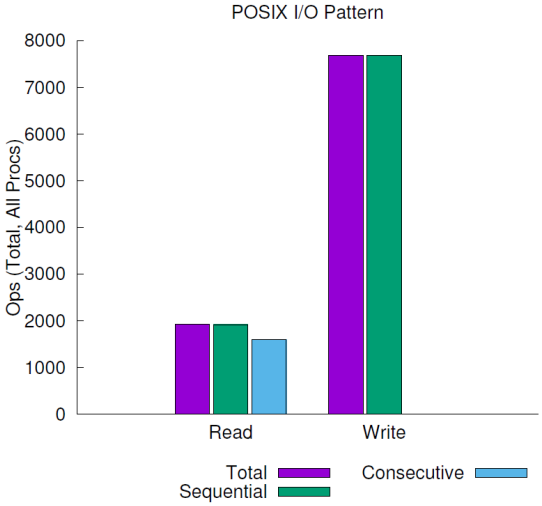


Figure 37: Implementing collective I/O operations for the strided I/O benchmark case

SIONlib - Partest

Overview

Partest is a benchmarking tool which is directly included in a SIONlib[9] library installation. In contrast to IOR the major focus of the benchmark is the analysis of large-scale I/O

[9] http://www.fz-juelich.de/jsc/sionlib

behavior especially in the context of checkpointing files. For this, Partest allows a direct comparison between the common way of creating task local checkpoint files and the usage of a shared file (or a small number of shared files) which are managed by the SIONlib API. The task local I/O behavior is very similar to the first IOR basic benchmark setup using the POSIX API and one file per process. Partest allows to directly measure the impact of having multiple files instead of a single files without common file system problems like the false sharing of file system blocks.

JUBE integration

JUBE was used to configure and run the individual benchmark configurations. The following Partest parameter configuration was created for this:

```
<parameterset name="partestParameter" init_with="partest_specs.xml">
      <parameter name="testtype" type="int" >0,3</parameter>
      <parameter name="bufsize">10KiB,4MiB,16MiB</parameter>
      <parameter name="localsize" mode="python">
        "{0}MiB".format(24*1024//${taskspernode})
      </parameter>
      <parameter name="numberoffiles" mode="python" type="int">
        1 if $testtype == 3 else $nodes
      </parameter>
      <parameter name="posix">0,1</parameter>
</parameterset>
```

Listing 2: Parttest JUBE configuration

Benchmark cases

Partest uses a fixed transfer size to write/read one big block of data into the output file. Either one file per process is used (POSIX case) or only one file or a small number of files is created (SIONlib case). In the benchmark setup the number of files for the SIONlib run are identical to the number of compute nodes, to allow all files of one node to share the same SIONlib file. Within the benchmark setup the transfer size (called "buffer size" in the benchmark configuration) as well as the number of processing elements are varied. Figure 38 shows the number of operations needed on the JURECA system using 48 processes, a transfer size of 4 MByte and 30 GByte total data. The number and size of I/O operations is identical to the IOR benchmark setup shown in figure 35.

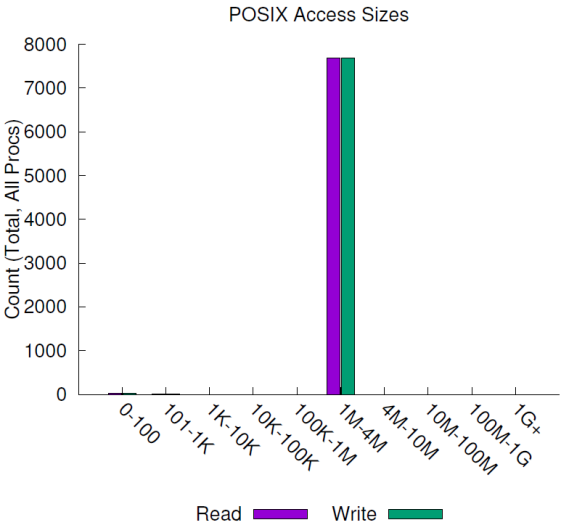


Figure 38: Darshan view of the SIONlib file access pattern.

7.4 System Overview

The following three systems at EoCoE partner sites ENEA, PSNC and JUELICH were used to perform the benchmark runs.

CRESCO

In the following we describe the CRESCO facilities at ENEA:

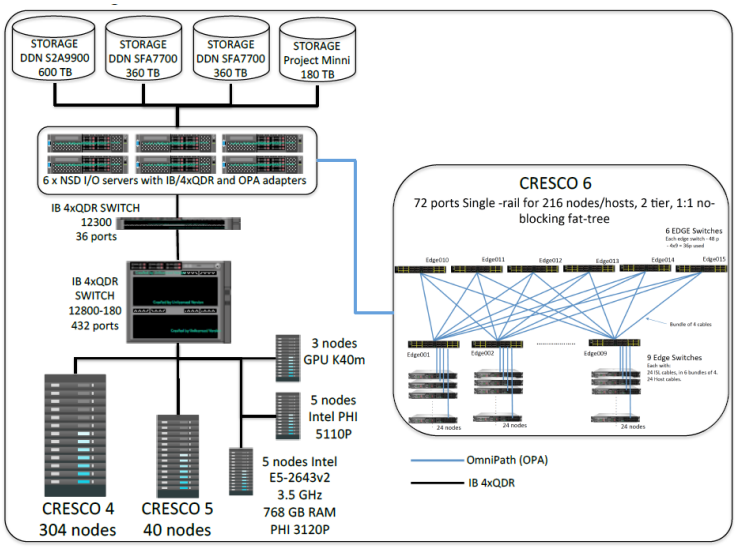


Figure 39: ENEA CRESCO systems architecture.

CRESCO4 section is a supercomputing facility with 4864 cores and a 101 TFlop/s peak made of 304 computing nodes, each of which has an 8-core dual socket (16 cores per node) Intel E5-2670 2.6GHz processor and 64 GB RAM DDR3.

CRESCO5 section is a supercomputing facility with 640 cores and a 25 TFlop/s peak made of 40 computing nodes, each of which has a 8-core dual socket (16 cores per node) Intel E5-2630 v3 2.4GHz processor and 64 GB RAM DDR3.

CRESCO4 and CRESCO5 have interconnection InfiniBand 4xQDR (40 Gb/s) based on QLogic switch 12800.

CRESCO6 section is a cluster with a peak performance of about 700 Tfp/s. The system is composed of 5 racks with 216 compute nodes with 10368 cores. Each node has two Intel Xeon 8160 CPUs, 3.22 TFlop/s peak, 192 GB RAM/node, 4 GB RAM/Core. The interconnect of CRESCO6 is Intel OmniPath 2 tier, 1:1 non-blocking fat-tree.

All CRESCO clusters share a 1 GB Ethernet network.

The storage of CRESCO systems dedicated to computing is composed of: 600 TB managed by a DDNS2A9900; $\sim$ 800 TB managed by two coupled DDNSFA7700, and 180 TB provided by a DotHill 3730.

EAGLE

Hardware characteristics:

Type:	PC cluster
Architecture:	Intel Xeon E5-2697
Network interfaces:	Infiniband FDR (56 Gb/s)
Number of CPU cores:	32984
Total computing power:	1372.13 TFlop/s
Size of system memory:	120.6 TB

Nodes characteristics:

Node name	CPU model	Nodes	CPU and cores	RAM	Node performance
HUAWEI CH121 V3	Intel Xeon E5-2697 v3	560	2x14	64 GB	1.1 TFlop/s
HUAWEI CH121 V3	Intel Xeon E5-2697 v3	530	2x14	128 GB	1.1 TFlop/s
HUAWEI CH121 V3	Intel Xeon E5-2697 v3	81	2x14	256 GB	1.1 TFlop/s
HUAWEI CH121 V4	Intel Xeon E5-2682 v4	55	2x16	128 GB	1.1 TFlop/s

Lustre characteristics:

- Metadata storage:
  - 1 SSD EMC ScaleIO 8TB matrix
  - 2 MDS servers:
    - 2x Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
    - 192GB RAM
    - InfiniBand FDR x8 / 56 Gb/s
- Data storage:

- 3 DDN SFA12kx40 disks matrices:
  - total number of disks: 420 (each 4TB)
  - storage: 1.2PB
- 16 OSS servers:
  - 2x Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
  - 124GB RAM
  - InfiniBand FDR x8 / 56 Gb/s
- Writing speed: 93.87 GB/s
- Reading speed: 101.49 GB/s

InfiniBand Topology: EAGLE InfiniBand (IB) topology consist of two layers: the island layer and the top of the cluster layer. Communication between islands is done through the top of the cluster. The island layer is composed of six islands. One island consists (in simplification) of three cabinets, four chassis each. Two of them additionally contain one switch. Each chassis has four InfiniBand links (two links per island layer switch). That gives 24 IB links for each switch. Each switch of the island layer is connected to two top of the cluster switches with 2 IB links per switch. Each island switch has 36 ports (24 + 4 used), blocking within the island is 4:1.

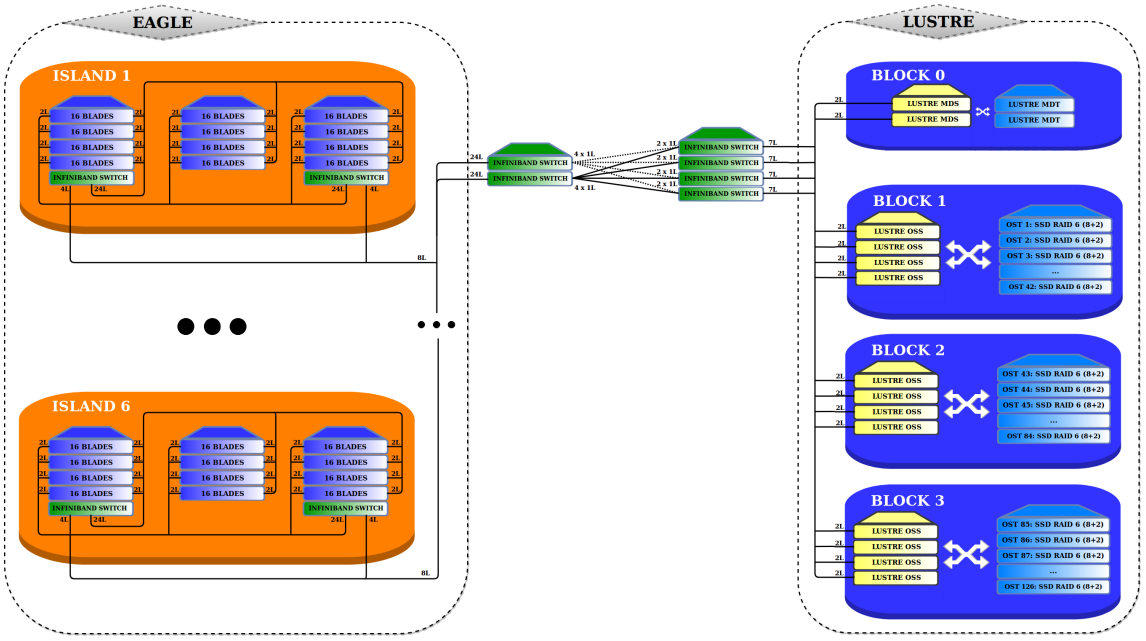


Figure 40: PSNC EAGLE network architecture.

JURECA

Hardware characteristics:

Type:	PC cluster
Architecture:	Intel Xeon E5-2680v3
Network interfaces:	Infiniband EDR (100 Gb/s)
Number of nodes:	1884
Number of CPU cores:	45216
Total computing power:	1800 TFlop/s + 430 TFlop/s (NVIDIA K40 and K80 GPUs)
Size of system memory:	128/256/512 GiB memory per node

JURECA uses a full fat tree network layout and Infiniband EDR between all nodes (100 Gb/s). The central GPFS based storage cluster JUST is connected by Infiniband FDR (56 Gb/s).

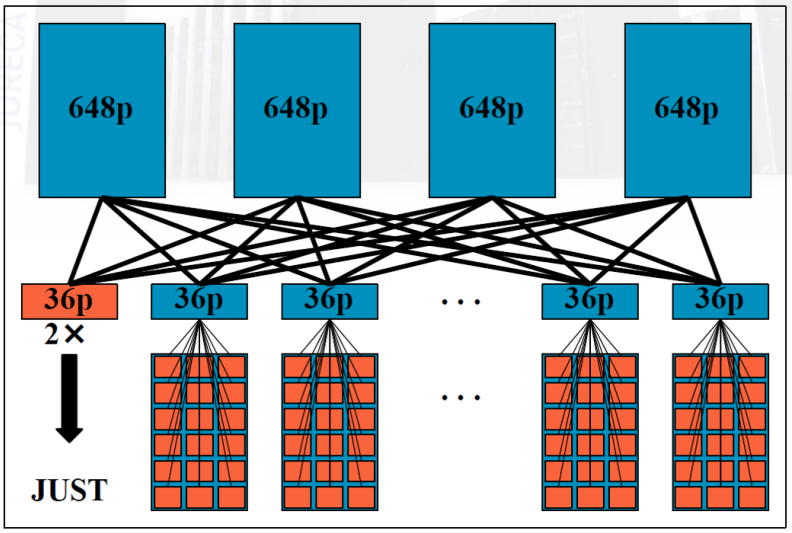


Figure 41: JUELICH JURECA full fat tree network layout.

Storage system JUST GPFS characteristics:

- 75 PB gross capacity
- JUST is capable of more than 400 GB/s
- JURECA bandwidth to JUST: 100 GB/s

7.5 Results

I/O API scaling behaviour

To scale the I/O of an application the overall bandwidth towards the file system must be increased. Typically this is achieved on a HPC system automatically by involving more network paths towards the file system when using a larger number of computing elements as each individual node usually has its own hardware link to the file system. In this case I/O and computing capabilities scale simultaneously. This allows the user to increase the overall theoretical I/O bandwidth while scaling up the computing part of the application. However, involving more I/O elements also implies additional synchronisation

and serialisation points, which have to be handled by the individual application.

Every I/O API uses a different approach to utilize the paths to the file system and to structure the I/O of the involved processes, but in general for the selected benchmark APIs three main groups can be defined:

- POSIX I/O on a shared file or a small number of shared files (e.g.when using the SIONlib API)
- POSIX I/O on task local files
- MPI I/O based shared file access (which is also used by HDF5 and PnetCDF)

On the JURECA system, which uses the GPFS file system, we see a good scaling behaviour on the reading side over all APIs which is shown in figure 42. The scaling is not entirely linear as certain runs for 12 or 16 nodes were slower than expected. This behaviour is mainly due to other applications running in parallel using the same I/O infrastructure as all tests were executed during regular production. Another aspect is a non ideal switch utilization, which depends on the node allocation provided by the resource manager. With regards to the writing part we see a lower bandwidth (roughly half of the reading bandwidth) and also a bandwidth drop at 20 nodes.

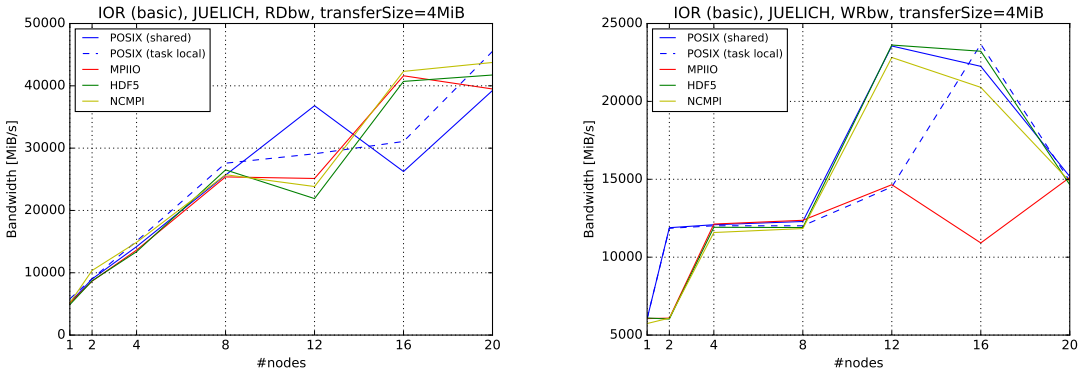


Figure 42: Scaling behaviour of the read and write bandwidth of the IOR benchmark using a fixed transfer size of 4 MiB and the basic benchmark setup on the JURECA system using 24 tasks per node.

The good scaling capabilities of all APIs for the "basic" benchmark configuration on GPFS is mainly due to the well structured file format and the regular access pattern as shown in figure 35. Within this pattern there is nearly no overlap between processes when accessing individual file system blocks. Switching to a non-consecutive pattern by using the second IOR benchmark case (striped pattern) together with a small strip size forces the processes to access data close to other processes which results in serialization on the file system level. Only the task local runs are unaffected by this pattern because each file receives its own file system block by default. All other shared APIs see a significant performance degradation as shown in figure 43.

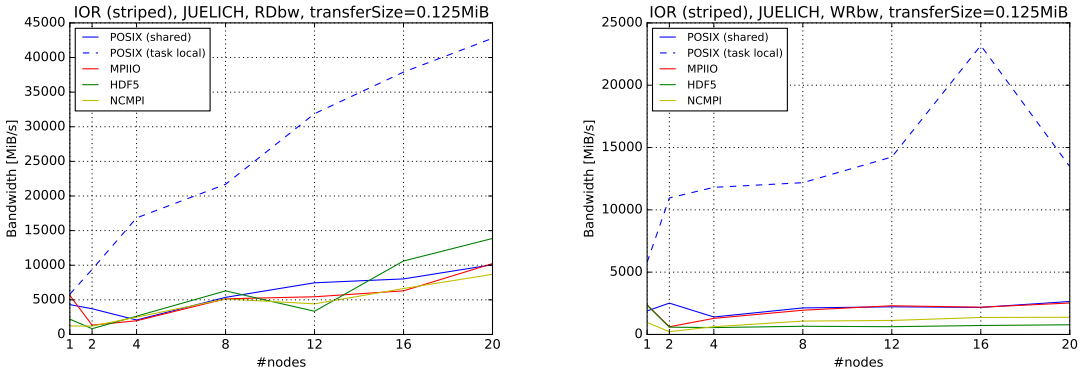


Figure 43: Scaling behaviour of the read and write bandwidth of the IOR benchmark using a fixed transfer size of 128 kiB and the striped benchmark setup on the JURECA system using 24 tasks per node.

For the EAGLE system, which uses the Lustre file system, we measured a large difference between task-local and shared file access as shown in figure 44 even for the "basic" benchmark case. The read and write bandwidth for task local files scales continuously with an increasing number of nodes and reaches up to 30 GiB/s, but all shared file accesses only scale slightly for the writing part and do not scale at all for the reading part. One reason for the performance degradation might be the utilization of all storage elements. Within Lustre a file is striped among a fixed number of so called Object Storage Targets (OST). Within the benchmark setup we used the default OST settings of the EAGLE system. The selection of OSTs is given by the file layout. In case of a highly structured file as in the basic benchmark case, only a subset of OSTs is involved in every parallel writing operation. In contrast to GPFS the user can adapt the OST settings by himself. We also tested this process on EAGLE as described in section 7.5.

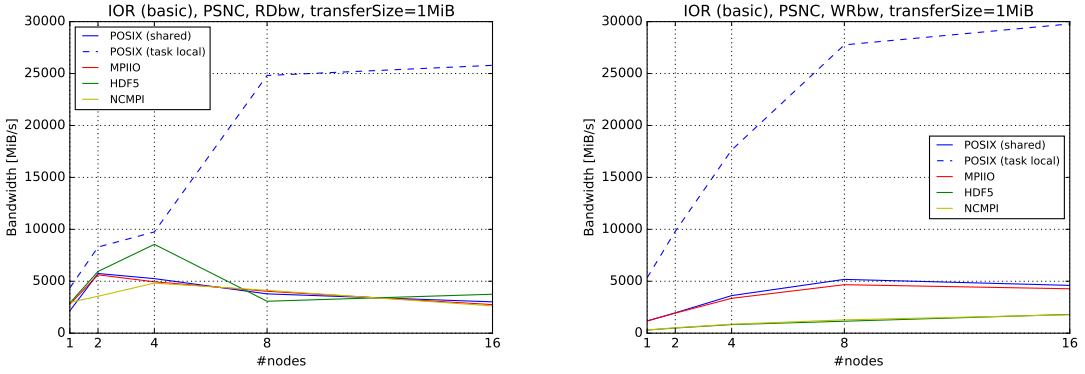


Figure 44: Scaling behaviour of the IOR benchmark using a fixed transfer size of 1MiB and the basic benchmark setup on the EAGLE system.

Task local vs. shared file I/O

As already shown in the scaling plots in section 7.5, having a single shared file in contrast to multiple task local files can influence the overall performance significantly. Typically the task local file access provides a better performance as many I/O pitfalls such a false sharing of file system blocks or metadata serialization can be avoided by using individual files for each process. However, dealing with a large number of files quickly becomes unmanageable and having large number of individual files is often also restricted by the computing site.

Using the Partest benchmark the difference between having multiple files as compared to having a single shared file or a small number of shared files can be measured with the help of the SIONlib library. SIONlib uses a file access scheme which avoids false sharing of filesystem blocks, which often leads to a reduced shared file performance. This allows SIONlib to achieve a similar bandwidth scaling as the task local approach without involving a large number of individual files.

For JURECA, in figure 45 and figure 46, we see a similar scaling behaviour between the task local and the SIONlib runs. The individual bandwidth shows fluctuation since all runs were performed during normal production. Within the benchmark configuration for SIONlib, one file per node was created, which allows to control the number of SIONlib files using a different number of tasks per node.

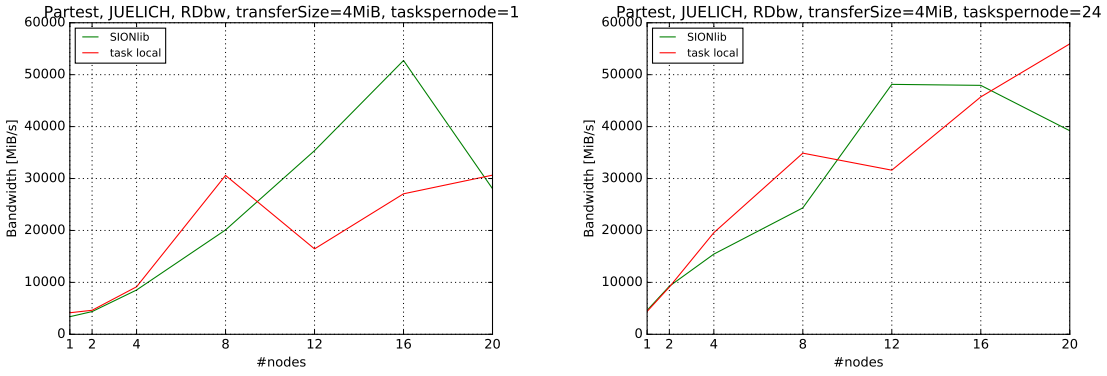


Figure 45: Read bandwidth performances on JURECA for 1 and 24 tasks per node using the partest benchmark

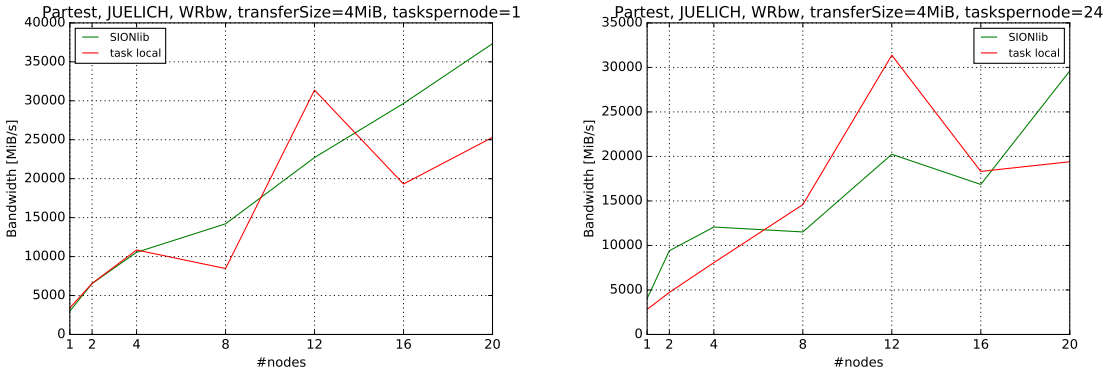


Figure 46: Write bandwidth performances on JURECA for 1 and 24 tasks per node using the partest benchmark

The same benchmark was also executed on the CRESCO4 system at ENEA using the GPFS 800 TB partition hosted on two coupled DDNSFA7700 (∼20 GB/s bandwidth) disk storage systems.

For analysing the SIONlib results we considered that data has been obtained during normal production which involved, on average, more than 300 computing nodes out of 380. To reduce the impact of user activities, measurements have been obtained by averaging on three different runs executed on different days.

The plots shown in figure 47 and figure 48 represent the general by observed behaviour: Similar to JURECA the task local and the SIONlib performance are close to each other. The I/O throughput reaches a plateau with 4 nodes; the reason is not clear but it might be due to the fact that even in the case of runs with more than one node, each of which has a network card capable of 4 GB/s, only one of them was involved in the I/O.

Figure 47: Read bandwidth performances on the CRESCO tmp filesystem for 1 and 16 tasks per node using the partest benchmark

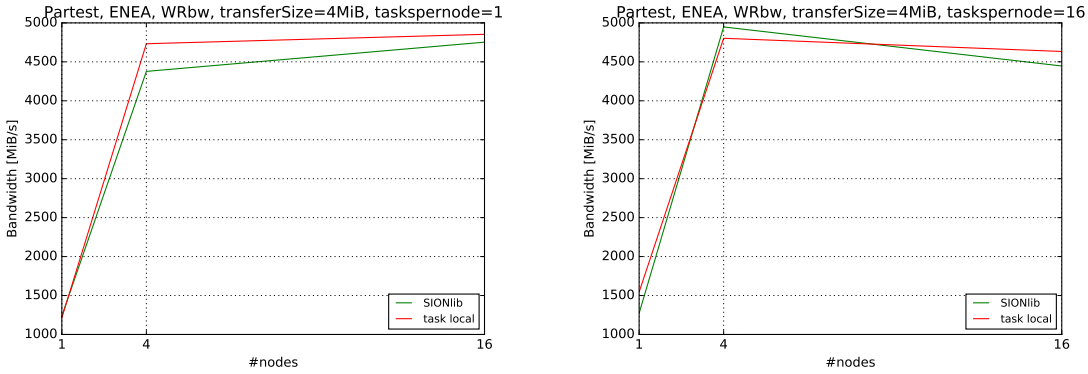


Figure 48: Write bandwidth performances on the CRESCO tmp filesystem for 1 and 16 tasks per node using the partest benchmark

On EAGLE at PSNC we tested the same approach using the Lustre file system. We see in figure 49 a better scaling for SIONlib if more tasks per node are involved. In contrast to this, the scaling of the writing part shows the opposite behaviour, as can be seen in figure 50. For the writing part the amount of files for $tasks per node = 1$ is the same for SIONlib as for the task local approach (here we see nearly the same bandwidth). For $tasks per node = 28$ the number of files for the task local approach is 28 times larger than the SIONlib run, here more files seem to provide a better utilization of the involved OSTs. The difference for the reading $tasks per node = 1$ run is unclear as the number of files is the same and the access scheme is handled in a similar way.

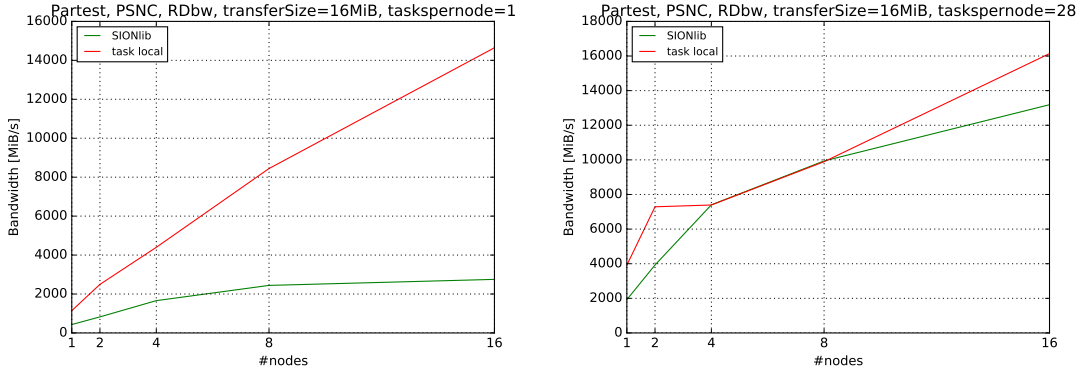


Figure 49: Read bandwidth performances on the EAGLE system for 1 and 28 tasks per node using the partest benchmark

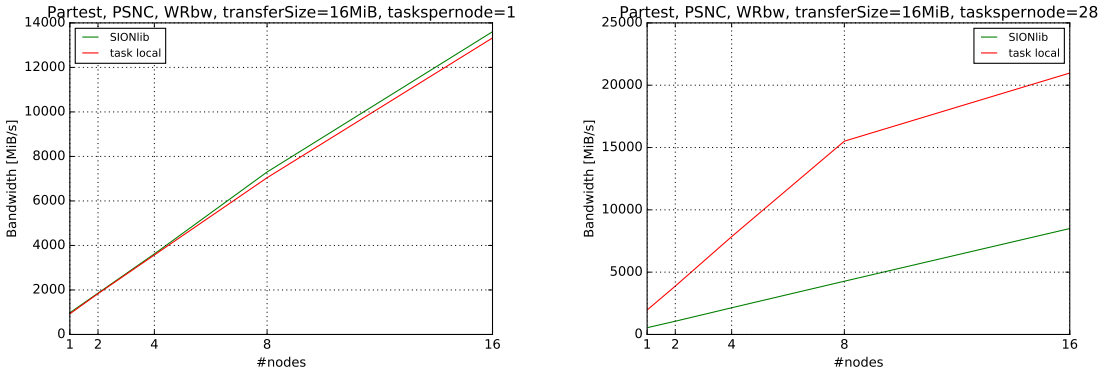


Figure 50: Write bandwidth performances on the EAGLE system for 1 and 28 tasks per node using the partest benchmark

I/O transfer size behaviour

Depending on the application, data can be transferred in different chunk sizes to the file system. Typically the overall local memory consumption forces an application to write smaller chunks multiple times instead of buffering and writing one larger chunk of data.

For the regular file pattern of the basic IOR benchmark setup the transfer size only has a small impact on the overall bandwidth as shown in figure 51. Here a transfer size of 128 kiB is only slightly slower as compared to a transfer size of 256 MiB.

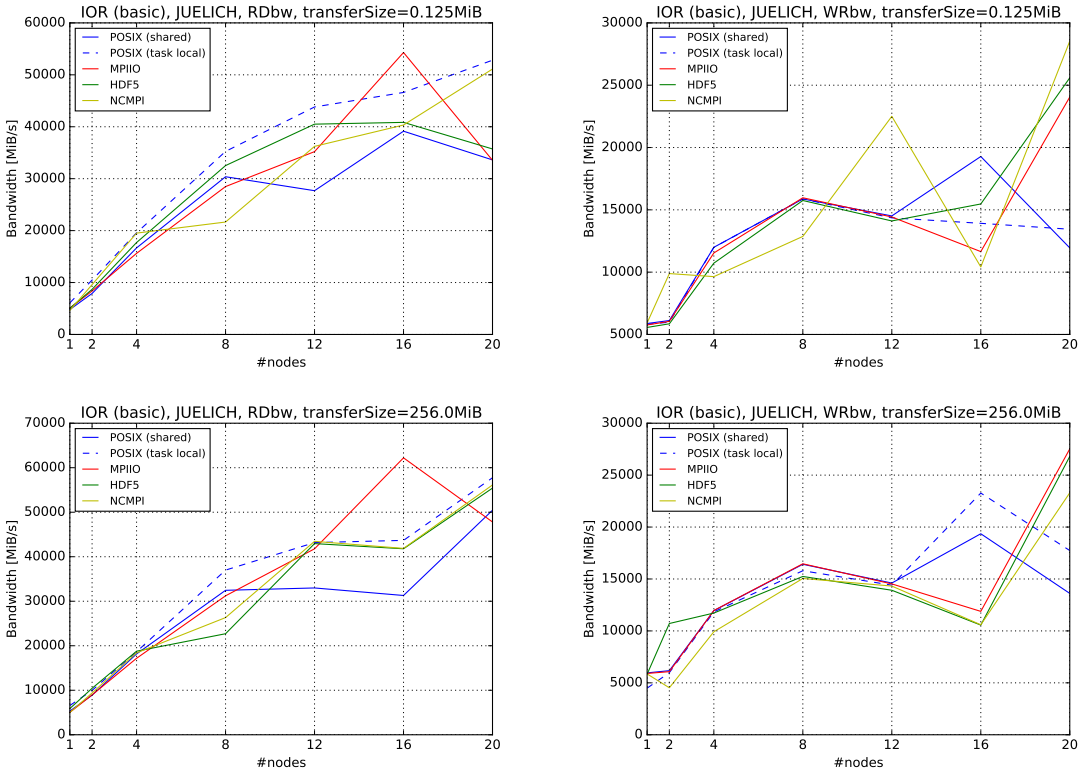


Figure 51: Scaling behaviour of the IOR (basic configuration) read and write performance by using 128 kiB and 256 MiB transfer sizes with 24 tasks per node on JURECA.

For the striping benchmark case, changing the transfer size also changes the internal file format by using larger continuous blocks per process. Increasing the transfer size creates a more regular file structure and can significantly increase the overall performance as shown on the CRESCO system for the writing part in figure 52. On the reading side, HDF5 is mostly affected by the larger transfer size, but all other APIs see no improvement or even a performance degradation for larger transfer sizes.

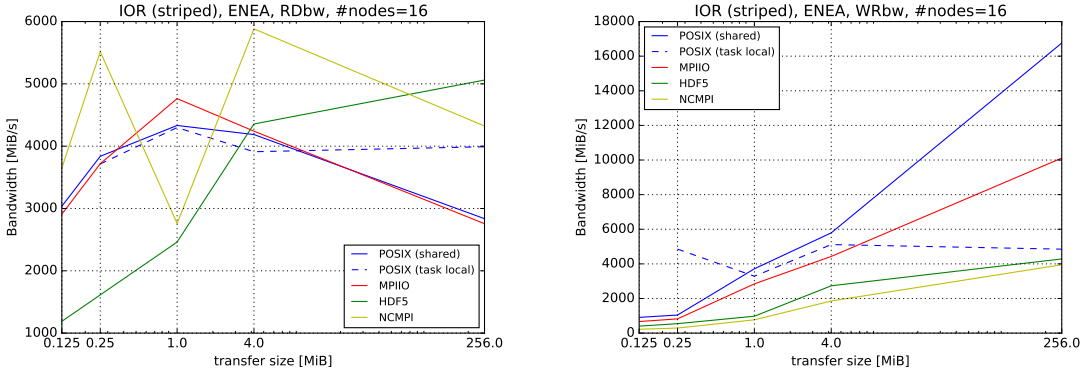


Figure 52: IOR (striping configuration) read and write performance by using different transfer sizes on 16 ENEA CRESCO4 nodes (256 cores).

For the EAGLE system in figure 53, the larger transfer sizes for the stripping benchmark example only have a larger impact on the writing behaviour, while for the reading behaviour the performance remains almost unchanged or is even reduced for the task local benchmark run when larger transfer sizes are used. The default EAGLE Lustre OST configuration was used for this runs.

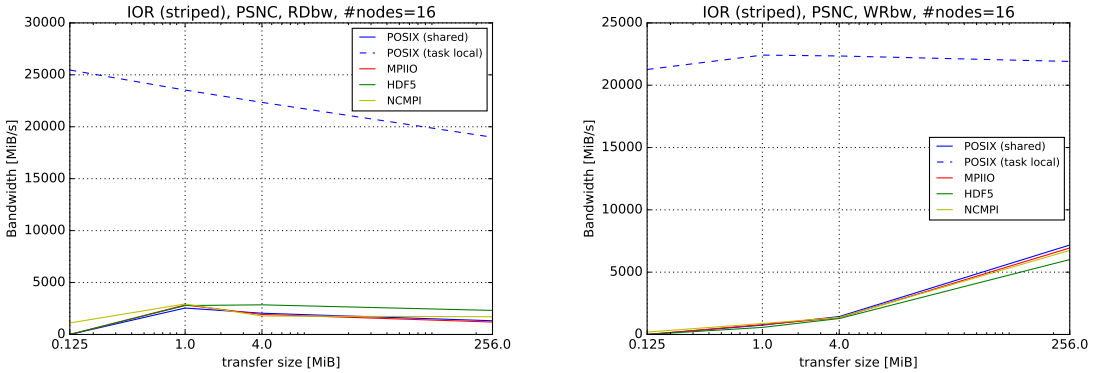


Figure 53: IOR (striping configuration) read and write performance by using different transfer sizes on 16 nodes of the EAGLE system.

Filesystem selection

Sometimes a computing center can provide different kinds of hardware disk storage systems. Depending on the needed I/O performance, it can be useful to dedicate some systems to intensive I/O workloads and others to low performance/long term storage. We present results concerning an experiment the purpose of which was to measure the efficiency of two different disk storage systems under the same heavy I/O workload by using the IOR benchmark. The efficiency of a system is obtained by measuring how much the I/O performance differs from its maximum peak. Efficiency gives an idea on how a storage technology is evolving. In figure 54 the experimental setup is shown.

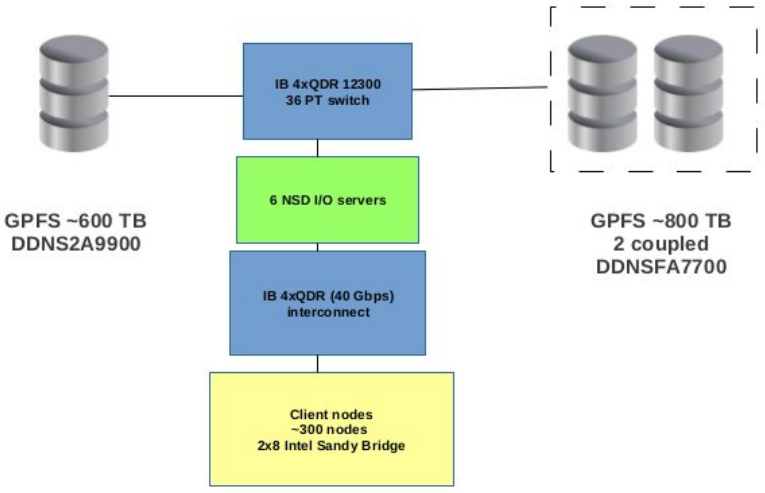


Figure 54: The two disk storage systems of which the efficiency has been measured by using IOR benchmark. Two different GPFS file systems with the same software configuration accessed a DDNS2A9900 of $\sim$ 600 TB (left) and two coupled DDNSFA7700 of $\sim$ 800 TB (right).

In this experiment two GPFS file systems with 1 MB blocksize and 6 I/O servers over an IB 4xQDR (40 Gbps) were used to access two storage systems: (A) a DDNS2A9900 of 600 TB and (B) two coupled DDNSFA7700 hosting 800 TB. The maximum available I/O throughput is 6 GB/s and 20 GB/s for (A) and (B) respectively. Each I/O client node has 16 cores Intel Sandy Bridge (E5- 2670 2.6 GHz). To simulate a heavy workload we run an IOR case with MPI-IO interface where each core executes an I/O task which reads/writes 1GB data. Figure 55 shows that in case of full load the efficiency is 8% for system (A) and 30% for (B).

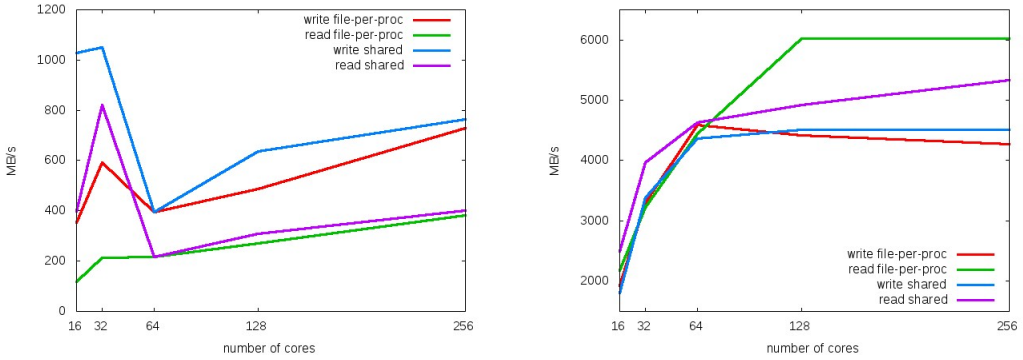


Figure 55: Measurement of I/O throughput for a (A) DDNS2A9900 (left) and (B) two coupled DDNSFA7700 (right) disk storage systems. Under full load the efficiency is 8% for system (A) and 30% for (B).

The IOR benchmark can be used to find optimal parameter settings of a file system. With the following experiment we aim at comparing the I/O performance of two GPFS file systems with 1 MB and 256 kB block size. We use a striping IOR configuration in which

`blockSize=transferSize`={128, 256} kB and {1, 4, 256} MB to simulate small, medium and large I/O. Runs are executed on a single computing node with 16 cores Intel Sandy Bridge (E5-2670 2.6 GHz) and 64 GB RAM. The aggregate amount of data for each run per core is 256 MB. For this setup (see figure 56) results show that with 1 MB block size the file system performs better than with 256 kB.

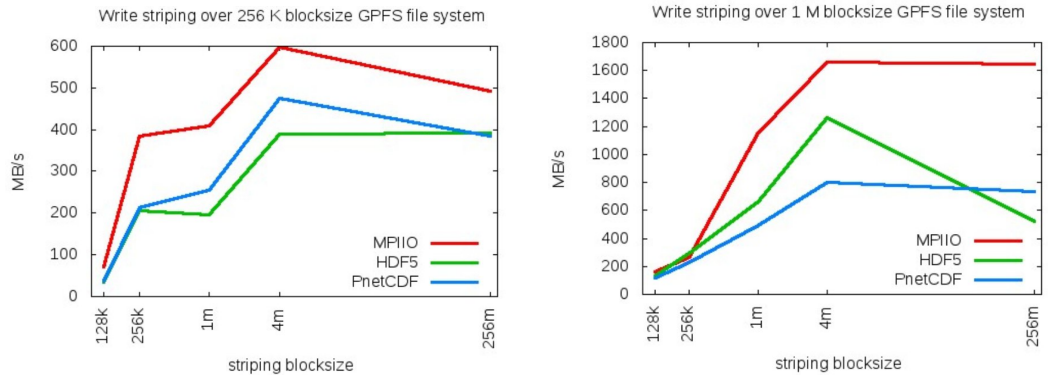


Figure 56: I/O performance of two GPFS file systems with 256 kB and 1 MB block size obtained by using IOR. The test is executed on a single node with 16 cores Intel Sandy Bridge (E5-2670 2.6 GHz) 64 GB RAM.

Caching

The gap between the CPU processing capabilities and the disk storage performance is a bottleneck that can sensibly reduce the overall productivity of jobs on large HPC systems. As shown in figure 57 the latency for accessing data is $\sim 10^{-12}$ s for CPU and $\sim 10^{-3}$ s for high quality HDD. Moreover, the number of I/O operations per second (IOPS) decreases by $\sim 10^{7}$ from CPU to disk drives. On the other hand fast SSD devices are very expensive and their exlusive usage for the storage might hence not be possible. In order to increase the I/O throughput with affordable cost, many storage disk systems provide fast cache buffers whit access times much smaller than that of spinning media. Often the cache buffer is made by a pool of SSD flash-based drives that actually acts as an extension of the DRAM cache.

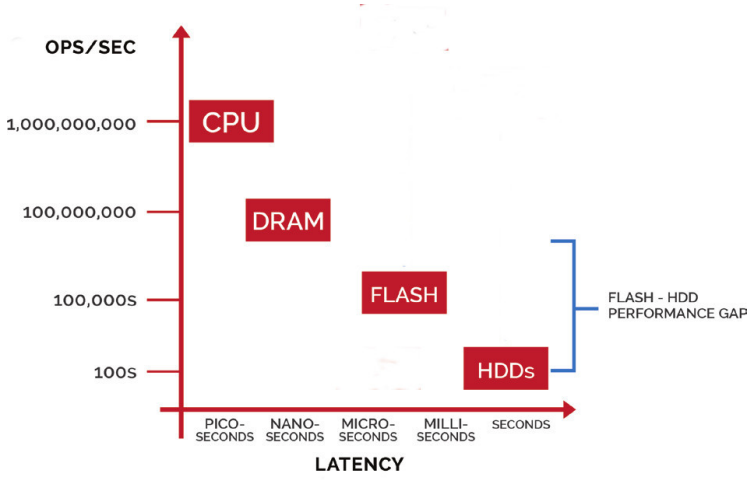


Figure 57: I/O data access latency and IOPS for different hardware.

IOR can be used to measure cache effects. We report the results of read and write IOR runs executed on CRESCO4 ENEA system by using two coupled DDN7700 disk storage systems each of which has a cache of 32 GB for reading and writing. The total available cache size for read/write operations is therefore 64 GB. The maximum bandwidth for accessing the disks is ∼20 GB/s.

Read case: In figure 58 the IOR striping results for a run with 16 nodes (256 cores) are shown. The runs were performed on a GPFS file system over an InfiniBand QDR 40 Gbps interconnect. The difference between these runs is the IOR reorderTasksConstant (-C) flag disabled (left) and enabled (right). Without the -C flag the data written by a node is (very likely) read-back by itself. The disk storage system has the read cache enabled and the node which wrote the data will read it from the cache reducing considerably the time spent to access metadata and blocks from disks. The resulting I/O throughput will increase. This is the case shown on the left where the peak of the read bandwidth exceeds the maximum I/O throughput of the storage disk system because data are stored in the cache whose access time is much smaller than that of the disks. By enabling the -C flag, IOR forces the data written by a node to be read by a different node. In this case the process which reads the data is different from the process which wrote it and cache effects are avoided. In fact the maximum bandwidth does not exceed the 20 GB/s.

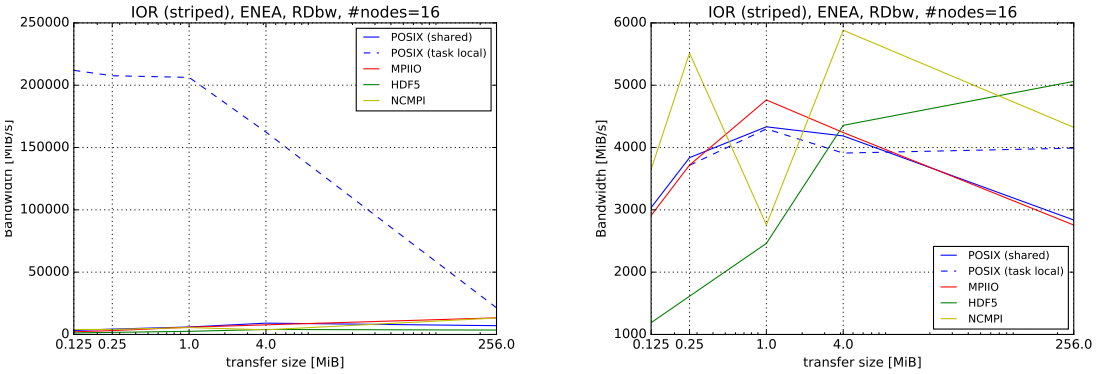


Figure 58: IOR (striping configuration) read performance with reorderTaskConstant (-C) flag disabled (left) and enabled (right). Runs are executed by using 16 ENEA CRESCO4 nodes (256 cores). The file system is GPFS and the network is IB QDR 40 Gbps. The storage disk system is composed of two coupled DDN7700. If the -C flag is disabled cache effects are present and the I/O throughput can exceed the maximum bandwidth towards the disks which is 20 GB/s. If the -C flag is enabled cache effects are avoided and the I/O throughput does not exceed 20 GB/s.

Write case: In figure59 the results of IOR striping runs with 16 nodes (256 cores) and reorderTasksConstant (-C) enabled (left) and disabled (right) on a GPFS file system over an InfiniBand QDR 40 Gbps interconnect are shown. The overall results are similar because the storage disk cache effect for write operations acts almost in the same way as for read operations. In fact, while in the read case the same data is taken from the fast cache, in the write case each block written is a piece of data stored in the cache independently of whether the -C flag is enabled or not. The write DDN7700 cache works as a buffer for incoming writes. When a large block of data is written by a host, it is streamed onto the cache at its maximum rate until it is filled. After that data are pulled and sent to the spinning disk. Because each core writes at most 256 MB of data, the maximum amount of data written to disks is 256 MB/core $\times$ 256 cores = 64 GB which is equal to the total available size of the write cache.

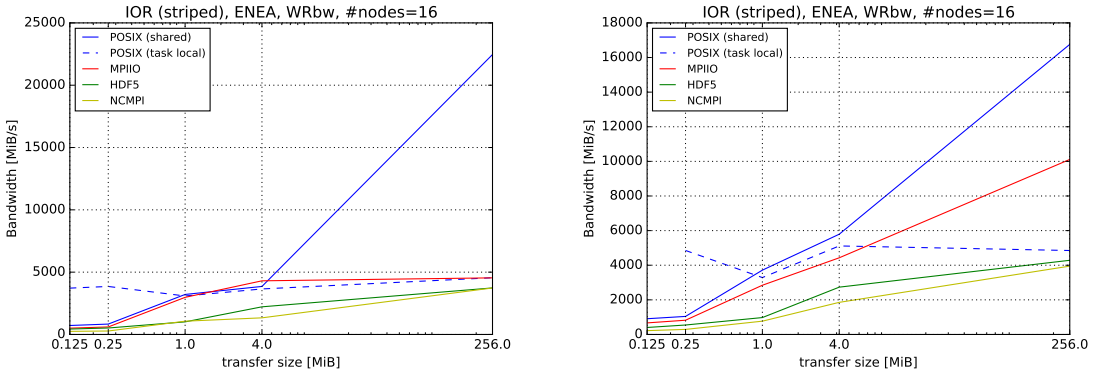


Figure 59: IOR (striping configuration) write performance with reorderTaskConstant (-C) flag disabled (left) and enabled (right). Runs are executed using 16 ENEA CRESCO4 nodes (256 cores). The file system is GPFS and the network is IB QDR 40 Gbps. The storage disk system is composed of two coupled DDN7700 with a total available write cache of 64 GB. Independently of whether the -C flag is enabled or not each block of new data written to disks is stored in the cache. The maximum amount of data written to disks is 64 GB and the write cache is continuously filled during the run and can absorb all new data. Once the cache is filled data are pushed to disk at the maximum available rate of 20 GB/s.

Collective I/O operations

Collective I/O operations can have a significant impact on the overall performance. Collective operations mean, that all involved processes read/write at the same time, which allows the I/O-API to optimise the performance by combining and redistributing multiple I/O accesses.

The most common collective I/O operation is collective buffering, which is performed within MPI-IO. Due to the dependency towards MPI-IO also HDF5 and NetCDF support collective buffering. The idea of collective buffering is to gather data for writing or distribute data for reading within a small number of aggregation processes. All I/O operations towards the file system are performed by those aggregation processes. This allows MPI-IO to perform larger structured filesystem requests, which can significantly reduce the overall I/O time towards the filesystem. On the other hand this mechanic adds additional time due to the aggregation and distribution mechanism, but also the file layout has a significant influence on the quality and the performance of collective I/O operations which we would like to demonstrate within our benchmark setup.

Figure 60 shows the collective I/O behaviour for the striping benchmark case on the JURECA system using MPI-IO. The performance in general for the striping benchmark case is quite low as shown in section 7.5. Collective buffering now tries to aggregate multiple requests to increase the overall performance. In the JURECA default configuration all tasks of one node are aggregated into the same buffer. For the given setup of having a 128 kiB transfer size for each process, the transfer size is increased to 3MiB towards the file system. Within a Darshan measurement this factor is directly visible and the performance improvement towards the file system is measured (see table 11). 24 MPI-IO data transfers

	access size [Byte]	count
MPI-IO	131,072	2,359,296
POSIX	3,145,728	98,304

Table 11: Darshan measurement of the most common I/O access sizes using collective reading and writing for the striping benchmark case.

are combined into a single one.

Having less operations towards the filesystem by aggregating multiple data blocks also avoids having an unstructured file access. However, faster POSIX I/O operations do not automatically mean that the overall I/O performs faster because collective buffering introduces an additional layer due to the need of transferring data between the tasks. In the writing case, as shown in figure 60, this overhead is much larger than the benefit provided by the faster POSIX access, so the overall runtime is even slower when using collective I/O operations. Here the individual data and transfer sizes and the number of collective aggregators might influence this behaviour, but it shows that collective buffering does not always provide a better performance.

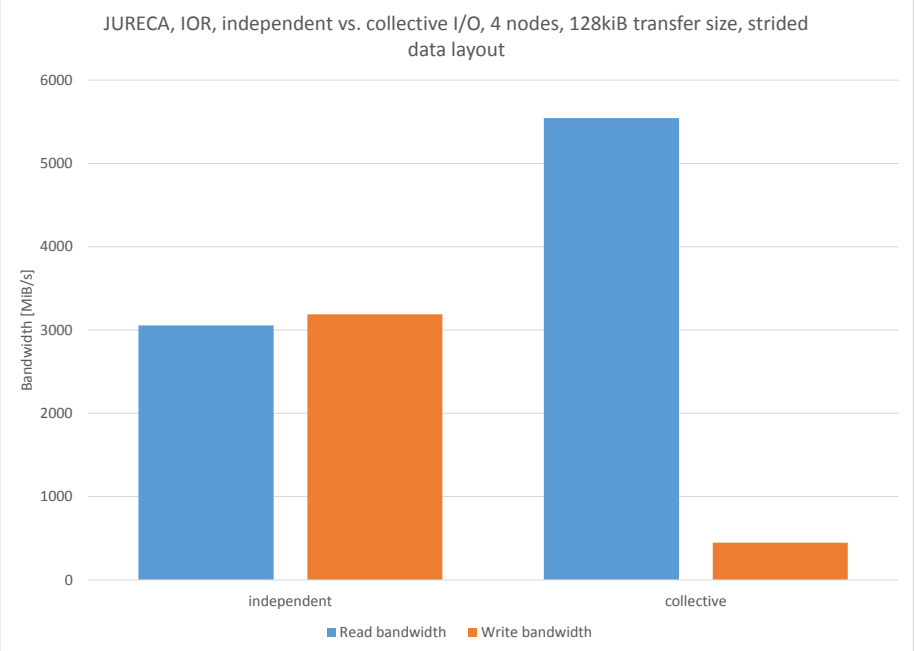


Figure 60: IOR striping benchmark run using 4 nodes and 24 tasks per nodes on JURECA and the MPI-IO API, comparing independent and collective I/O operations

The issue of collective buffering is even more significant if the file layout is changed. In case of the striping layout multiple processes are close to each other within the scope of the overall file, which is a perfect layout in context of collective buffering. In contrast, the basic file layout benchmark has large data blocks for each process. Using collective operations for such a file layout can reduce the overall bandwidth by a large factor as shown in figure 61.

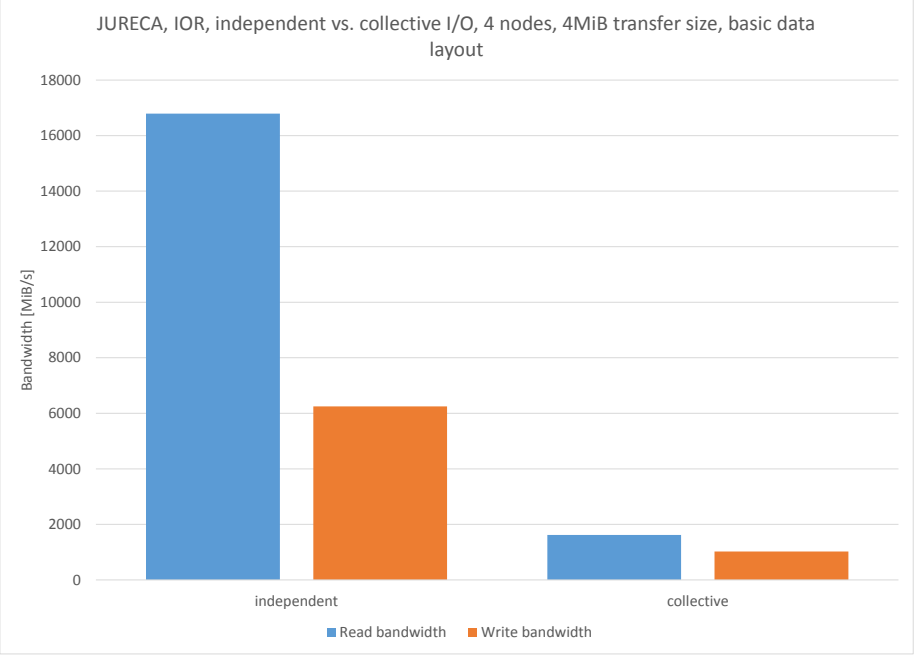


Figure 61: IOR basic benchmark run using 4 nodes and 24 tasks per nodes on JURECA and the MPI-IO API, comparing independent and collective I/O operations

There are two problems with this layout:

First, for the reading part the aggregator tries to read data for multiple processes at once. Beside the aggregation of 24 processes at the same time, there is an additional default limit of a maximum 16 MiB buffer size. For the given case, the buffer is filled with 16 MiB (due to the larger transfer size). The buffer always contains a continuous block of data. For the basic file layout all data only belongs to one process. So the 4 MiB which were initial requested are kept for this particular process, but all other data are ignored, because they are not needed by other processes. This mechanic is repeated until all processes read their 4 MiB data. As there are always 16 MiB transferred, the datasize is increased by a factor of 4, which finally reduces the overall bandwidth.

The writing part has the same problem as the reading part, but in addition when writing a 16 MiB buffer, only having 4 MiB of real process data creates the problem, that the unused buffer data can overwrite parts of the file. If MPI-IO sees this problem of having less data than buffer size (or having gaps within the buffer), it first fills the buffer by reading the original 16MiB from the file. This additional read operation is added on top. This is directly visible in the Darshan report in figure 62 and the access size overview in table 12. This buffering technique is similar to the MPI-IO data sieving approach, but it is also used in collective buffering when data sieving is disabled separately.

	access size [Byte]	count
MPI-IO	4,194,304	184,320
POSIX	16,777,216	264,574

Table 12: Darshan measurement of the most common I/O access sizes using collective reading and writing for the basic benchmark case.

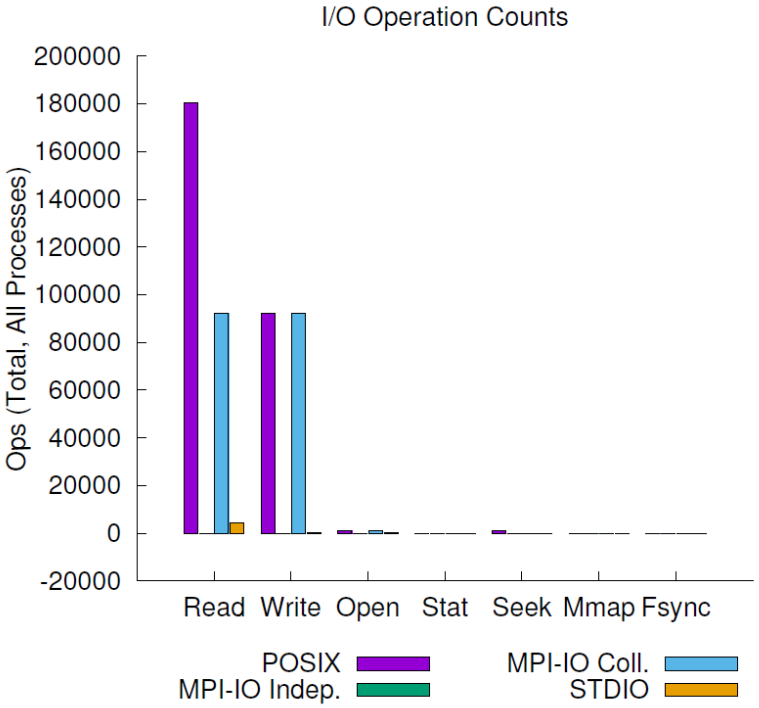


Figure 62: Darshan measurement for the basic benchmark setup on JURECA showing number of I/O operations.

This mainly points to the fact that having large continuous data blocks per process within a file can significantly decrease the collective operations bandwidth instead of providing any benefit and should hence not be used.

Lustre configuration options

In contrast to GPFS, the Lustre file system allows the user to change parameters, which may change the overall I/O behaviour. The most important values are:

- stripe count: Defines how many OSTs (Object Storage Targets) will be used to cover an individual file or a specific directory. OST is single storage volume on which Lustre reads and writes data.
- stripe size: Defines the size of an individual stripe on each OST. If the data is big enough Lustre can use multiple stripes on one OST in a round robin way.

In this section we will show how Lustre parameters influence the I/O performance. First

we test the default Lustre setting on Eagle and then try to optimise the configuration.

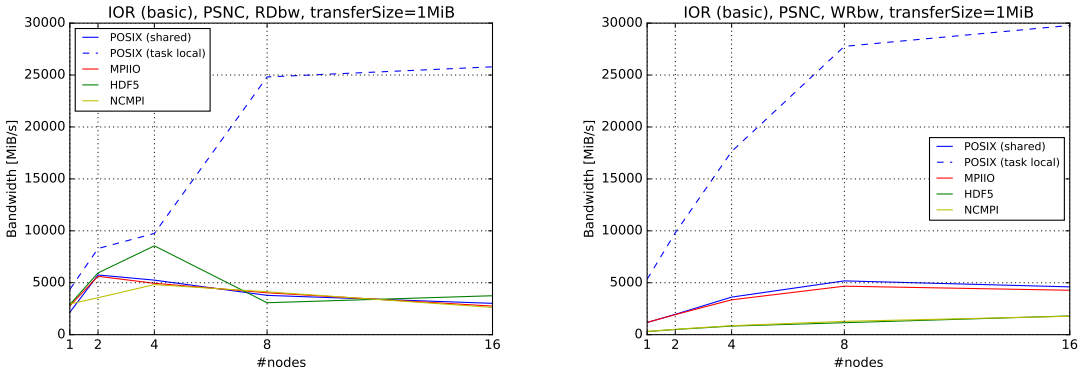


Figure 63: Scaling behavior of the IOR benchmark using a fixed transfer size of 1 MiB, the default stripe size and count setting and the basic benchmark setup on the EAGLE system.

As we already demonstrated in the scaling section 7.5 task local is the best approach for this configuration. This is due to small overhead of distributing data over OSTs and communication between processes. For a task local setup each individual file starts on another OST, which allows to utilize all OSTs in parallel and hence provides a good performance.

In figure 64 we manually increased the number of involved OSTs. This increased the overall bandwidth for the shared file approach, especially for reading data, but also slightly lowered the task local bandwidth.

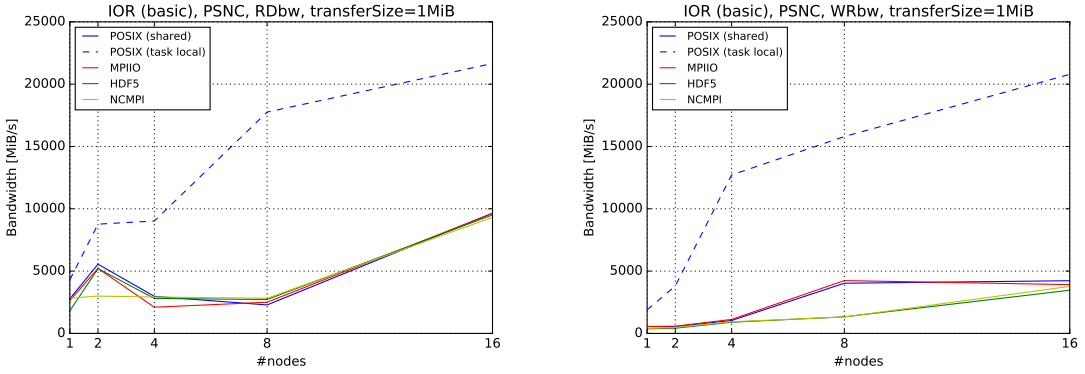


Figure 64: Scaling behavior of the IOR benchmark using a fixed transfer size of 1 MiB, the default stripe size, a stripe count of 126 and the basic benchmark setup on the EAGLE system.

Beside the number of involved OSTs, also the OST mapping of the file is very important. For the basic IOR benchmark setup, each process has 256 MiB of data as one combined chunk within the file layout. This means each process starts to write at position $rank * 256MiB$. The start points are distributed in a round robin way over the OSTs. For a

strip size of 1MiB this means byte position $rank * 256MiB$ will be on OST : $(rank * 256MiB) mod stripcount$. For 12 OST we end up with three different start OSTs for the first operation of all processes. Due to this behavior only three OSTs will be used in parallel, as all processes read and write the same amount of data. Using a stripe count of 126 allows the utilization of 63 OSTs. However the distribution of data is still not ideal. Instead it will be best to increase the stripe size to 256MiB. This allows to utilize all OSTs directly within the first and all other write operations.

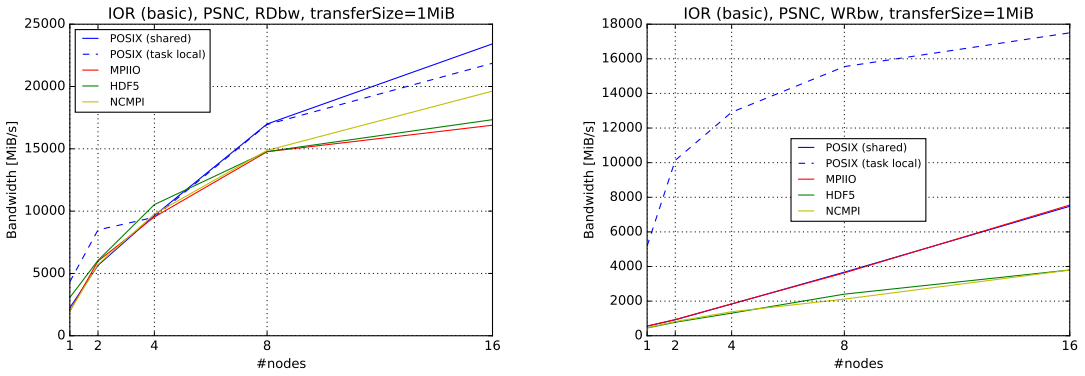


Figure 65: Scaling behavior of the IOR benchmark using a fixed transfer size of 1 MiB, a stripe size of 256 MiB, a stripe count of 126 and the basic benchmark setup on the EAGLE system.

Figure 65 shows that the read performance of the shared operations performs much better. The write performance is still not perfect but scales with the number of processes. The task local write performance is much lower in contrast to the smaller stripe size due to the fact that the small task local files now only utilize a small number of OSTs each.

7.6 Summary

Within this report we summarize the results of the I/O benchmarking activity within the EoCoE project. I/O behaviour is a general important aspect for each HPC application. In- and output sizes can significantly increase when increasing the overall application scalability, which can change an application bottleneck from the compute to the I/O side.

The I/O questionnaire for the EoCoE applications presents a wide variety of different I/O patterns and I/O behaviour. Using IOR and partest allows us to test IO scalability using different I/O patterns and APIs.

Using multiple benchmark runs on three different HPC systems we tested the I/O scalability using different APIs and transfer sizes, analysed the difference between task local and shared file I/O, investigated different file systems and caching effects and had a closer look on collective I/O operations.

Especially the chosen file layout and its access scheme has a huge influence on the overall performance. Within GPFS, large independent, continuous data blocks show a good performance and a good scalability for all APIs. Here the transfer size itself only has a minor impact on the overall bandwidth. Of course such a file layout is difficult to achieve if multidimensional, distributed data is handled. Approaches like data chunking in HDF5 or

collective I/O operations can help to restructure the file access patters. However, collective I/O operations can also introduce new performance problems if they are used without a closer look ot the file structure. So it is highly recommended to test the reading and writing routines of the applications on an individual basis to evaluate the benefit of collective I/O.

Task local file approaches often perform best, due to fact that problems like false sharing of file system blocks and metadata handling can be avoided automatically. On the other side task local files quickly become unmanageable and only represent the particular local data view of each processor. Using those for checkpointing is fine, but also shared file APIs like SIONlib can reach the same bandwidth while solving some of the issues related to task local files.

For the Lustre file system, it is important to adapt the OST handling to the given file format and access scheme. Otherwise even structured data is limited due to the data distribution and OST mapping scheme.

8. Parallel Data Interface (PDI)

Contributors	Julien Bigot (Maison de la Simulation, CEA), Corentin Roussel (Maison de la Simulation, CEA), Leonardo Bautista (BSC), Kai Keller (BSC), Karol Sierocinski (PSNC), Tomasz Paluszkiewicz (PSNC)

8.1 Context

High-performance computing (HPC) applications manipulate and store large datasets for scientific analysis, visualization purposes and/or resiliency. Multiple software libraries have been designed for interacting with the parallel file system and in some cases with intermediate storage levels. These libraries provide different level of abstraction and have been optimized for different purposes. The best I/O library for a given usage depends on multiple criteria including the purpose of the I/O, the computer architecture or the problem size. Therefore, to optimize their I/O strategies, scientists have to use multiple API's depending on the targeted execution. As a result, simulation codes contain intrusive and library dependent I/O instructions interwoven with domain instructions. We have designed a novel interface that transparently manage the I/O aspects and support multiple I/O libraries within the same execution.

8.2 Introduction

Scientific simulation codes consist of several components, such as one or several physical model implementations, post-processing code and multiple inputs and outputs. The first aspects found the basis of computational science. Inputs and outputs (I/O), on the other hand, are at the border between the concerns of computational scientists that know what data to write and for what purpose, and those who know how to write the data correctly and efficiently. I/O dedicated libraries have thus been designed to encode this technical knowledge and to provide it for the implementation in any kind of software.

The choice of a specific I/O library is however constrained by multiple aspects, such as the architecture of the supercomputer, the type and size of the data, the purpose of the write- or read-action and others. All these factors must be taken into account in order to optimize the I/O of a given application code. In order to optimize the I/O of a code, an efficient strategy is to select both the file format and the I/O library that suit best all the previous constraints. The intrusive I/O directives found inside the code affect clarity, decrease maintainability and increase development costs.

In this report, we describe the parallel data interface (PDI), a novel interface that enables users to access multiple I/O libraries through a single API. PDI is not an I/O library by itself; it only offers a unified way to access existing libraries. The API supports read- and write- operations using various I/O libraries within the same execution, and allows to switch and configure the I/O strategies without modifying the source (no re-compiling). However, it does not offer any I/O functionality on its own. It delegates the request to a dedicated library plug-in where the I/O strategy is interfaced. The range of functions and the performance of the underlying I/O libraries are not straitened

During the EoCoE project our work have focus on: **1)** Designing and implementing PDI,

a new interface for accessing multiple I/O libraries without the need of modifying or recompiling the source code of the application. **2)** Adapting the GYSELA code to use either FTI/HDF5/SIONlib plain or embedded via PDI for checkpointing. We have verified the correctness of the written datasets. **3)** Performing an evaluation at scale on different supercomputers with different architectures.

In the next section we summarize the main motivations, the design, and the implementation of PDI.

8.3 Motivations

GYSELA is a scientific code that models the electrostatic branch of the ion temperature gradient turbulence in tokamak efficiency [Gra15] excluding I/O and diagnostics. At the heart of GYSELA is a self-consistent coupling between a 3D Poisson solver and a 5D Vlasov solver. The main data manipulated in the code from which all other values are derived is a 5D particle distribution function in phase space.

Since the complete simulation state can be derived from the 5D particle distribution function and a few scalar values (time-step, etc.) only these data are written into checkpoints. This single field usually represents in the order of one quarter of the total memory consumed. Storing it too often would represent a large overhead both in term of time and storage space. Therefore, actual results exploited by physicists, take the form of *diagnostics*: smaller arrays computed from the 5D distribution function and written to permanent storage regularly. In term of I/O these two parts of the code have different requirements.

One can further distinguish two types of checkpoints with different requirements. Preventive checkpoints, which are written during execution for fault-tolerance purpose only and job segmentation checkpoints on the other hand, which are written at the end of a job. For preventive intermediate checkpoints, one can leverage burst-buffering strategies overlapping computation and I/O, use any kind of on-disk file format or even rely on temporary storage only available for the duration of the job. For final segmentation checkpoints, however, there is no more computation to overlap with I/O. The restart can happen on a different set of nodes or even on a different machine which requires to write them to permanent storage.

Given that the best I/O strategy depends on multiple criteria (e.g., purpose of the I/O, size of the problem, hardware platform) one would like to be able to independently select the strategy to use, for each case. Choosing the best I/O strategies requires deep technical knowledge in aspects that are not the main concern of domain scientists and is thus best handled by a different person. As a matter of fact, many parallel codes have no support by I/O specialists at all and any approach that would require such a role is likely to fail.

I/O libraries claim to offer a separation of concern between application developers that use the library and library developers that encode efficient I/O strategies in the library. Nonetheless, each of them has been created to account for a specific need and provides interesting features in a specialized context.

As no single library provides an optimal choice in all possible situation, a possible solution could be to rely of multiple distinct libraries to support different strategies in the code. The library choice can then either be made at compile-time or at run-time. Compile-time choice with approaches such as `#ifdef`s in the code, means that a single library is available for each run and does not enable to use multiple libraries for different purposes during the

same run (e.g., preventive versus segmentation checkpoints). The choice at run-time, with an approach such as "`switch/case`", can support mixing libraries. Without compile-time support however, this induces a hard dependency on the library that prevents compiling the code on a machine where a single of the options has not been ported yet. Thus, one would ideally have to implement both types of choices; a compile-time choice, to either depend on the library or not and a run-time choice, to select between all compiled libraries for each specific I/O.

A limitation of this approach is, however, that it increases the ratio of code, dedicated to I/O, with each new supported library or even each distinct strategy implemented using the same library. It requires cumbersome code to deal with both, run-time and compile-time choice of libraries. It leads to duplication of code, as the data to write has to be specified for each strategy implemented with distinct API's. As a result, this makes the code difficult to maintain, since multiple concerns are mixed at the same place. As the number of supported libraries grows, the maintenance cost may become unaffordable.

Since commonly none of the existing libraries can offer all the desired features at once, we propose a new interface, which permits users to enable distinct features of different libraries through one single API: the parallel data interface (PDI).

8.4 Design of the Parallel Data Interface

Acknowledging the huge amount of work that has already been done in existing libraries, we do not intend to re-develop the I/O strategies previously implemented but rather to build on top of them. PDI has therefore been designed to be a simple API that provides access to existing libraries and enables to combine them. The main goal of PDI is to separate the I/O aspects from the domain code and thus to improving the separation of concerns. PDI is a glue layer that sits in-between the implementation of these two aspects and interface both of them.

PDI has been designed in such a way that the separation of concern does not come at the expense of good properties of existing approaches. The implementation of a given I/O strategy through PDI should be as efficient and as simple (ideally less complex) than existing approaches, both from the user and I/O expert point of view. This should hold, whatever the level of complexity of the I/O strategy, from the simplest one where the implementation time is paramount, to the most complex one where the evaluation criterion is the performance on a specific hardware.

We therefore design PDI to act as a *lingua franca*, a thin layer of indirection that exposes a declarative API for the code to describe the information required for I/O and that offers the ability to implement the I/O behavior using these informations. In order to decouple both sides, we rely on a configuration file that correlates information exposed by the simulation code with that required by the I/O implementation and enables to easily select and mix the I/O strategies used for each execution. This approach brings important benefits as it improves the separation of concerns thanks to the two abstraction layers. It offers a simple API that allows a uniform code design while accessing and mixing the underlying I/O libraries.

The API limits itself to the transmission of information (required by the I/O implementation) that can only be provided during execution. Information that is known statically is expressed in the configuration file. The only elements that have to be described through

the API are therefore: **1)** the buffers that contain data with their address in memory and the layout of their content, **2)** the time period along execution when these buffers are accessible either to be read or written. In addition, the API handles the transmission of control flow from the code to the library through an event system. Events are either generated explicitly by the code or generated implicitly when a buffer is made available or just before it becomes unavailable.

The data layout is often at least partially fixed, only some of its parameters vary from one execution to the other (*e.g.* the size of an array). We therefore support the description of this layout in the configuration file so as not to uselessly clutter the application code. The value of parameters that are only known during the execution can be extracted from the content of buffers exposed by the code.

Implementing an I/O strategy is done by catching the control flow in reaction to a event emitted by the simulation code and using one ore more of the exposed buffers. The name of the events and buffers to use come from the configuration file, ensuring a weak coupling between both side. Two levels of API are offered. A low level API enables to react to any event and to access the internal PDI data structures where all currently exposed buffers are stored. A higher level API enables to call user-defined functions to which specific buffers are transmitted in reaction to well specified events.

When using the low-level API, it is the responsibility of the I/O code implementation to access the configuration file to determine the events and buffers to use. This API is well suited for the development of plugins that require a somewhat complex configuration because they are intended to be reused in multiple codes. This is typically the case when interfacing I/O libraries with declarative API's close to that of PDI where options in the configuration file are enough to match the API's.

User can implement their own I/O strategies that can be interfaced with PDI. When using user-defined functions, the name of the events and buffers passed to the function are specified in the configuration file in a generic way. The function itself does neither have access to the configuration file content nor to the list of shared buffers.

This approach is less flexible but much easier to implement. It is well suited when a specific code has to be written to use a given I/O library in a given simulation code as is often the case with libraries with imperative API's. It can be used to provide additional instructions that complement but are distinct from the library features.

In order to decouple this I/O implementation code both from PDI and from the simulation code, it is defined in dedicated object files that can either be loaded statically or dynamically (a plugin system). This means that PDI does not depend on any I/O library, only its plugins do. This also simplifies changing strategy from one execution to the other as the plugins to load are specified in the configuration file.

To summarize, PDI offers a declarative API for simulation codes to expose information required by the implementation of I/O strategies. The I/O strategies are encapsulated inside plugins that access the exposed information. A week coupling mechanism enables to connect both sides through a configuration file. This can be understood as an application of aspect oriented programing (AOP) to the domain of I/O in HPC. The locations in the simulation code where events are emitted are the *joint points* of AOP. The I/O behavior encapsulated in the plugins are the *advices* of AOP. The configuration file specifies which behavior to associate at which location and constitute the *pointcuts* of AOP.

```
enum PDI_inout_t { PDI_IN=1, PDI_OUT=2, PDI_INOUT=3 };

PDI_status_t PDI_init(PC_tree_t conf, MPI_Comm *world);
PDI_status_t PDI_finalize();

PDI_errhandler_t PDI_errhandler(PDI_errhandler_t handler);

PDI_status_t PDI_event(const char *event);

PDI_status_t PDI_share(const char *name, void *data, PDI_inout_t access);
PDI_status_t PDI_access(const char *name, void **data, PDI_inout_t access);
PDI_status_t PDI_release(const char *name);
PDI_status_t PDI_reclaim(const char *name);
```

Listing 3: The PDI public API

```
PDI_status_t PDI_export(const char *name, void *data);
PDI_status_t PDI_expose(const char *name, void *data);
PDI_status_t PDI_import(const char *name, void *data);
PDI_status_t PDI_exchange(const char *name, void *data);

PDI_status_t PDI_transaction_begin(const char *name);
PDI_status_t PDI_transaction_end();
```

Listing 4: Simplified PDI API for buffer exposing

8.5 PDI Implementation

PDI is freely and publicly available[10] under a BSD license. It is written in C and offers a C API with Fortran bindings to the simulation code. This covers uses from C, C++ and Fortran, the three most widespread languages in the HPC community. We present the C flavor in this section but the Fortran binding offers the exact same interface. The plugin API is currently limited to C but bindings for other languages (*e.g.* LUA, Python) are planned.

The simulation code API contain functions to initialize and finalize the library, change the error handling behavior, emit events and expose buffers as presented in Listings 3. The initialization function takes the library configuration (a reference to the content of a YAML [BKEN09] file) and the world MPI communicator that it can modify to exclude ranks underlying libraries reserve for I/O purpose. The error handling function enables to replace the callback invoked when an error occurs. The event function takes a character string as parameter that identifies the event to emit.

The most interesting functions of this API are however the buffer sharing functions. They support sharing a buffer with PDI identified by a `name` character string and with a specified `access` direction specifying that information flows either to PDI (`PDI_OUT`, read-only share), from PDI (`PDI_IN`, write-only share) or in both directions (`PDI_INOUT`.) The `PDI_share` and `PDI_access` function start a buffer sharing section while the `PDI_release` or `PDI_reclaim` function end it. `PDI_share` is used for a buffer whose memory was previously owned by the user code while `PDI_access` is used to access a buffer previously unknown to the user code. Reciprocally, `PDI_reclaim` returns the memory responsibility to the user code while `PDI_release` releases it to PDI.

In a typical code, the buffers are however typically shared for a brief period of time between

[10] `https://gitlab.maisondelasimulation.fr/jbigot/pdi`

```yaml
data: # data id and type
  my_array: { sizes: [$N,$N], type: double }
metadata:
  N: int
  it: int
plugins:
  declh5: # plug-in name
      write:
        my_array: # data to write
          dataset: array2D
          file: 'example_$it.h5'
          when: '($it>0) && ($it%10)' # condition to write
```

Listing 5: Example of PDI configuration file

two access by the code. The previously introduced API requires two lines of code to do that. The API presented in Listing 4 simplifies this case. Its four first functions define a buffer sharing section that lasts during the function execution only. The functions differ in terms of access mode for the shared buffer: `share(OUT)` + `release` for `PDI_export`; `share(OUT)` + `reclaim` for `PDI_expose`; `share(IN)` + `reclaim` for `PDI_import`; and, `share(INOUT)` + `release` for `PDI_exchange`.

This API has the disadvantage that it does not enable to access multiple buffers at a time in plugins. Each buffer sharing section ends before the next one starts. The two transaction functions solve this. All sharing sections enclosed between calls to these functions have their end delayed until the the transaction ends. This effectively supports sharing of multiple buffers together. The transaction functions also emit a named event after all buffers have been shared and before their sharing section ends.

At the heart of PDI is a list of currently shared buffers. Each shared buffer has a memory address, a name, an access and memory mode and a content data type. The access mode specifies whether the buffer is accessible for reading or writing and the memory mode specifies whose responsibility it is to deallocate the buffer memory. The content data type is specified using a type system very similar to that of MPI and is extracted from the YAML configuration file.

The `data` section of the configuration file (example in Listing 5) contains an entry for each buffer, specifying its type. The type can be a scalar, array or record type. Scalar types include all the native integer and floating point of Fortran and C (including boolean or character types.) Array types are specified by a content type, a number of dimensions and a size for each dimension. They support the situation where the array is embedded in a larger buffer with the buffer size and shift specified for each dimension. Record types are specified by a list of typed and named fields with specific memory displacement based on the record address.

The types can be fully described in the YAML file, but this makes them completely static and prevents the size of arrays to change at execution for example. Any value in a type specification can therefore also be extracted from the content of an exposed buffer using a dollar syntax similar to that of bash for example. The syntax supports array indexing and record field access. For the content of a buffer to be accessible this way, it does however needs to be specified in the `metadata` section of the YAML file instead of its `data` section. When a metadata buffer is exposed, its content is cached by PDI to ensure that it can be accessed at any time including outside its sharing section.

```
1 main_comm = MPI_COMM_WORLD
2 call PDI_init(PDI_subtree, main_comm)
3 call PDI_transaction_begin("checkpt")
4 ptr_int=> N; call PDI_expose("N" ,ptr_int)
5 ptr_int=> iter; call PDI_expose("it",ptr_int)
6 call PDI_expose("my_array",ptr_A)
7 call PDI_transaction_end()
8 call PDI_finalize()
```

Listing 6: Example of PDI API usage

The plugins to load are specified in the `plugins` section of the configuration file. Each plugin is loaded statically if linked with the application and dynamically otherwise. A plugin defines five function: an initialization function, a finalization function and three event handling functions. The event handling functions are called whenever one of the three types of PDI event occurs, just after a buffer becomes available, just before it becomes unavailable and when a named event is emitted.

The plugins can access the configuration content and the buffer repository. Configuration specific to a given plugin is typically specified under this plugin in the `plugins` section of the YAML file. The YAML file can however also contain configuration used by plugins in any section. It can for example contain additional information in a buffer description.

We currently have developed three plugins. The *FTI* plugin interfaces the declarative FTI library, the *decl'H5* plugin interfaces a declarative interface built on top of HDF5 and the *usercode* plugin supports user written code as specified in Section 8.4. A plugin interfacing a declarative version of SIONlib is also available in the repository, but most imperative libraries are best accessed through the *usercode* plugin.

Let us now present an example to show how PDI usage works in practice. Listing 6 shows the use of the Fortran API to expose to PDI two integers: `N` and `it`, and an array of dimension `N`×`N`, `my_array`. The configuration file for this example is the one presented in Listing 5.

When the `PDI_init` function is called, the configuration file is parsed and the *decl'H5* plugin is loaded. This plugin initialization function is called and analyzes its part of the configuration to identify the events to which it should react. No plugin modifies the provided MPI communicator that is therefore returned unchanged. A transaction is then started in which three buffers are exposed: `N`, `it` and `my_array`. The *decl'H5* plugin is notified of each of these events but reacts to none. The transaction is then closed that triggers a named event to which the *decl'H5* plugin does not react as well as three end of sharing section events, one for each buffer. The *decl'H5* reacts to the end of the `my_array` sharing since this buffer is identified in the configuration file. It evaluates the value of the `select` clause and if nonzero writes the buffer content in a dataset whose name is provided by the `var` value ("array2D") to a HDF5 file whose name is provided by the `file` value ("example$it.h5".)

8.6 Summary & Conclusions

The parallel Data interface, abbreviated PDI, is a novel interface that allows to separate most of the I/O concerns from the application code. PDI transparently manages the I/O aspects that are provided by external libraries or user codes. It does not impose any limitation on the underlying I/O aspects and decreases the programming effort required

to perform and adapt I/O operations for different machines. Moreover we have demonstrated[11] that, thanks to PDI, scientists can use multiple I/O libraries within the same execution by simply changing a configuration file and without the need of modifying or recompiling the source code of the application.

Currently, PDI supports FTI, HDF5 libraries and a SIONlib plug-in has been finalized. Other I/O libraries (XIOS for instance) and other use cases are considered, including but not restricted to, in-situ visualization, scientific workflows.

References

[BKEN09] Oren Ben-Kiki, Clark Evans, and Ingy dot Net. *YAML Ain't Markup Language (YAML) Version 1.2, 3rd edition.* No Starch Press, 2009.

[Gra15] Virginie Grandgirard. High-Q club: Highest scaling codes on JUQUEEN – GYSELA: GYrokinetic SEmi-LAgrangian code for plasma turbulence simulations. online, March 2015.

[11] A paper has been submitted to IEEE Cluster 2017 with this work.

9. Continuous Integration for HPC

Contributors	Kenny Blondy (Maison de la Simulation, CNRS), Julien Bigot (Maison de la Simulation, CEA)

9.1 Context

Continuous Integration is a development practice, based on the use of software to automate builds and tests on a shared project, to detect and correct errors quickly. The main interest is to define a set of tests that will be systematically achieved at each step of the development. In High-performance computing (HPC), the need to improve programs and simulations requires more development and tests. Compilation tests, unit tests, integration tests, performance tests, regression tests, etc. are necessary steps, repetitive, time consuming, but they are important to detect problems and to improve source code quality. Continuous Integration allows to dramatically reduce the time spent doing tests and speed up development.

The goal of the project is to enable users to automate tests during the development phase of research projects, using a relatively simple interface, standardized, while ensuring data security (integrity, confidentiality), and respecting the constraints related to computer systems and security.

Continuous Integration implies:

- to share source code, with git for instance,
- to commit as often as possible,
- to develop tests.

For Continuous Integration, a lot of enterprises are using an open source automation server, Jenkins. A single instance used to be open to the whole developers, but the need to scale Jenkins horizontally more than vertically has led to isolation of multiples instances. To ensure the isolation between teams, they containerize the instances of Jenkins, with Docker.

Jenkins

Jenkins is an open source automation server written in Java. It helps to automate the non-human part of the software development process, with continuous integration and facilitating technical aspects of continuous delivery. It is based on plugins that can be added to get new possibilities, for instance to connect to a git repository or to use SSH instead of the native java client/server implementation.

Docker

Docker is a software for create and manage containers from images. A container consists of an entire runtime environment: an application, plus all its dependencies, libraries and other binaries, and configuration files needed to run it, bundled into one package. By containerizing the application platform and its dependencies, differences in OS dis-

tributions and underlying infrastructure are abstracted away. Images are built from a descriptive file called a Docker file. This file contains the parent image to use and a list of actions like download and install packages, copy files. Docker embeds an orchestrator, the swarm mode, ensuring clustering, failover, redundance and other functionalities.

The isolation provided by Docker allows Jenkins' instances to access remote servers, however the reverse is not. To allow users to connect to the web interface, admins need to link a port of the host machine for each container, or use a reverse proxy having a port open and dispatching on instances. Authentication can be made through Jenkins web interface, with internal databases or using LDAP, or using a module to let reverse proxy ensuring authentication.

9.2 Implementation

The implementation is a set of Bash and Python scripts to manage Docker and services running in containers, multiple Dockerfile to build images, and configurations files of the services. Requirements to set up the infrastructure are to have Python 2.7 and Docker 1.12 or higher. Set up consists of building Docker images and starting servers: a reverse proxy to access the Jenkins, a database for user authentication, a web server to navigate between projects and manage authorizations, and an http proxy to log outgoing connections. Containers are distributed in three networks:

- one for web services (web servers and databases)
- one for access servers (reverse proxy and http proxy)
- one for the jenkins' instances.

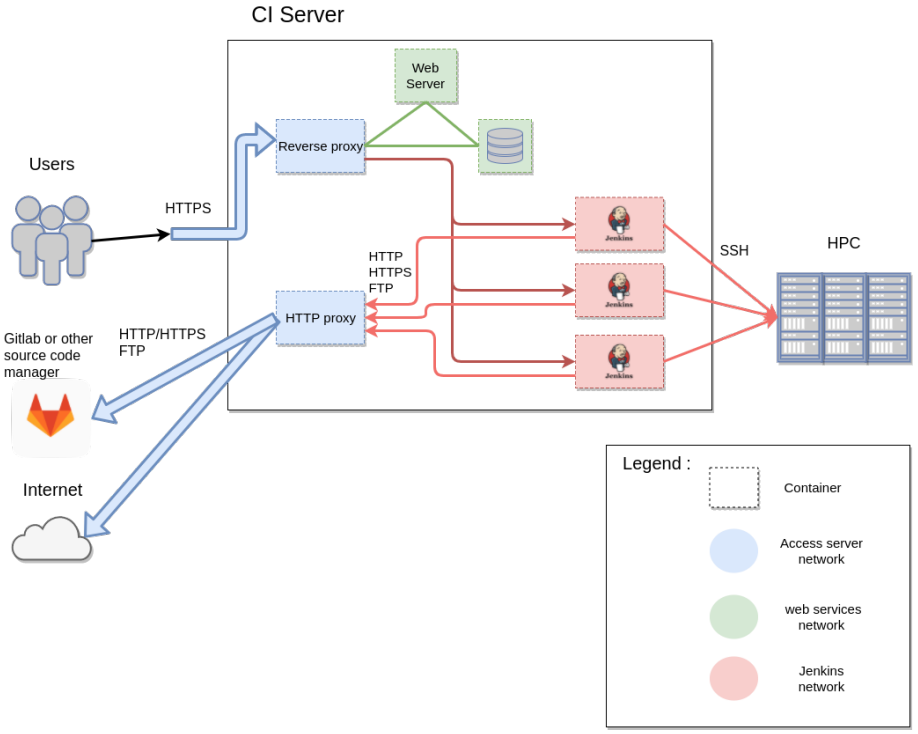


Figure 66: The infrastructure

Every connections initiate from the Jenkins instances are logged, as well as each connection to a Jenkins instances. This ensures non-repudiation of actions taken by users. This is an essential prerequisite in HPC centers. Each project has its own Jenkins isolated in a container. Accesses are done HTTPS, the two previous points ensure confidentiality of user data. Moreover, in each Jenkins a project manager acts as administrator. He can install plugins, manage credentials to connect with SSH, manage slaves, etc. Other users are developers and can create, manage and run jobs.

9.3 Summary & Conclusions

Continuous integration for HPC is a problem divided in three main issues: choose softwares to use, design an infrastructure, and implement a solution answering user needs and respecting the security rules of HPC centers. The first one is based on the software's community to ensure development and support (Docker and Jenkins have large communities). The second is more dependent of system and security constraints. And, the last is putting practices of reflection and research to allow users to have continuous integration in HPC center. As a development practice, CI is not only about setting up an infrastructure with software but also about to train users and administrators.

The future of the project is to start a first CI platform, and from there, move it forward with more functionalities. Its future is also to write set up instructions to deploy this kind of solution in most HPC centers, administration instructions to keep it operational, and users guides to let the scientific community do the best uses of implementations.