Data driven Computational Mechanics at EXascale

Data driven Computational Mechanics at EXascale

Work program topic: EuroHPC-01-2019
Type of action: Research and Innovation Action (RIA)

AI-SOLVE PERFORMANCE EVALUATION REPORT

DELIVERABLE D3.3

Version No 1

http://www.dcomex.eu/
This project has received funding from the European High-Performance Computing Joint Undertaking Joint Undertaking ('the JU'), under Grant Agreement No 956201

DOCUMENT SUMMARY INFORMATION

Project Title	**Data driven Computational Mechanics at EXascale**
Project Acronym	DCoMEX
Project No:	956201
Call Identifier:	EuroHPC-01-2019
Project Start Date	01/04/2021
Related work package	WP 3
Related task(s)	Task 3.1, 3.2, 3.3, 3.7
Lead Organisation	NTUA
Submission date	11/07/2024
Re-submission date	
Dissemination Level	PU

Quality Control:

	Who	**Affiliation**	**Date**
Checked by internal reviewer	George Stavroulakis	NTUA	29/06/2024
Checked by WP Leader	Vissarion Papadopoulos	NTUA	02/07/2024
Checked by Project Coordinator	Vissarion Papadopoulos	NTUA	02/07/2024

Document Change History:

Version	**Date**	**Author (s)**	**Affiliation**	**Comment**
1.0	28.06.2024	Ioannis Kalogeris	NTUA	

Contents

1. Description

The objective of Deliverable 3.3 is to assess the performance of the set of algorithms developed as part of the AI-Solve library, which were introduced in Deliverable 3.2. While the theory behind these algorithms is presented in D3.2, this deliverable focuses on a comprehensive evaluation report detailing the effectiveness, efficiency, and robustness of these algorithms across a diverse range of complex applications in computational mechanics.

The AI-Solve algorithms can be classified into two main categories:

- **ML-assisted solution schemes** that aim to accelerate solutions to large-scale parameterized problems by:
    - Utilizing surrogate models built from a small set of high-fidelity system solutions to derive highly accurate initial guesses to iterative solvers.
    - Employing ML-algorithms that leverage information from previous system solutions to develop customized preconditioners for Preconditioned Conjugate Gradient (PCG) Methods, resulting in faster convergence rates than traditional solvers.
- **Solvers for parallel computing environments**:
    - These solvers are designed to accelerate solution to large-scale problems by optimizing performance in parallel computing environments.

The remainder of this deliverable presents a series of applications that demonstrate the performance of the proposed algorithms compared to conventional solution techniques. This evaluation includes:

- Effectiveness: Measuring the accuracy and reliability of the algorithms in various scenarios.
- Efficiency: Assessing the computational time and resources required to reach solutions.
- Robustness: Evaluating the algorithms' ability to handle a wide range of problem complexities and variations.

The applications highlighted in subsections 2.1, 2.3, and 3.1 of this deliverable have already been included in deliverable 3.2 to demonstrate the computational merits of the proposed solution approaches. However, we believe it is important to include them here as well to provide a more comprehensive overview of the different problem types that the AI-Solve library can handle.

2. ML-Assisted solution schemes

2.1 POD-2G method – Application to the parameterized Biot steady-state problem

According to Task 3.2 of the DCoMEX project "*Surrogate models for preconditioning and coarse problem solution*", a series of data-driven techniques were developed to accelerate the solution process of large-scale parameterized linear systems. In this section, we present the data-driven solution framework, termed POD-2G, that was described in section 3.1 of deliverable 3.2. This algorithm has been tested a parameterized version of the Biot problem (deformable porous medium) based on the $u - p$ formulation [1] and the results of this investigation are presented herein.

The problem under investigation involves a 3D solid cube under prescribed displacement and pressure boundary conditions, as shown in figure 1.

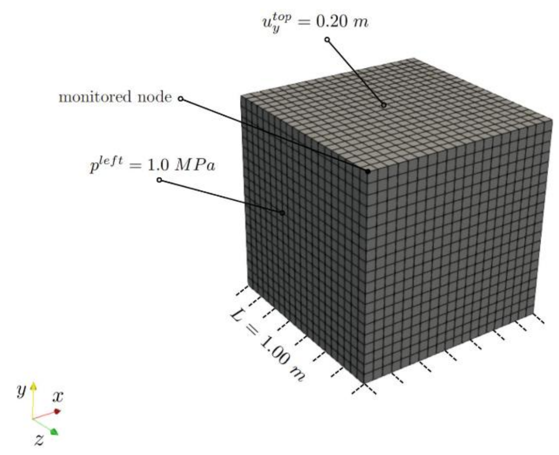


Figure 1: Geometry, boundary conditions and FE discretization of the Biot problem

We assumed for this problem that the Lame coefficients μ and λ are random variables following the distributions given in table 1. As a first step, the Latin Hybercube sampling method was utilized to generate $N_{train} = 300$ parameter samples $\{\mu_i, \lambda_i\}_{i=1}^{N_{train}}$. The surrogate's architecture is presented in Figure 2. The CAE is trained for 100 epochs with a batch size of 10 and a learning rate of 0.001, while the FFNN is trained for 5000 epochs with a batch size of 20 and a learning rate of 10-4. The average normalized *l*2 norm error of the surrogate model in the test data set is 0.68%.

Parameter	Distribution	Mean	Standard deviation
$\mu\ (MPa)$	Lognormal	0.30	0.09
$\lambda\ (MPa)$	Lognormal	1.70	0.51

Table 1: Random parameters of the Biot problem

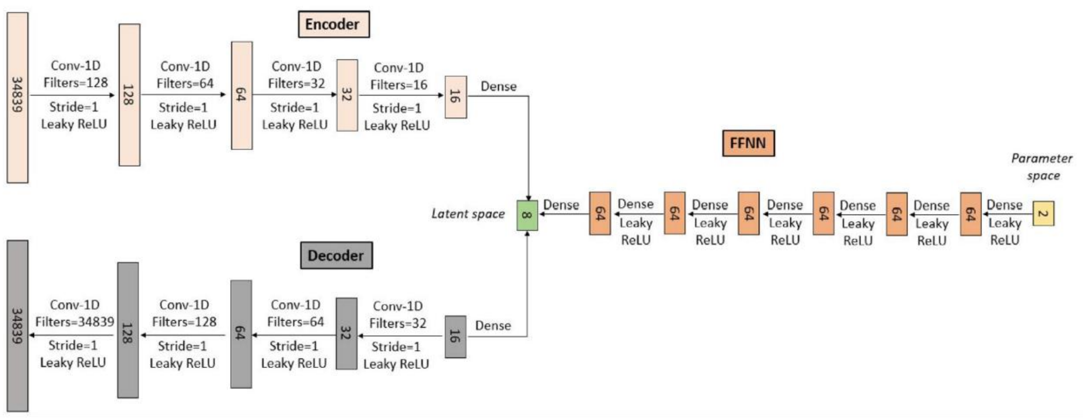

Encoder
34839
Conv-1D Filters=128 Stride=1 Leaky ReLU
128
Conv-1D Filters=64 Stride=1 Leaky ReLU
64
Conv-1D Filters=32 Stride=1 Leaky ReLU
32
Conv-1D Filters=16 Stride=1 Leaky ReLU
16
Dense
FFNN
Parameter space
2
Dense Leaky ReLU
64
Dense Leaky ReLU
64
Dense Leaky ReLU
64
Dense Leaky ReLU
64
Dense Leaky ReLU
64
Dense Leaky ReLU
64
Dense
Latent space
8
Decoder
Dense
16
Conv-1D Filters=32 Stride=1 Leaky ReLU
32
Conv-1D Filters=64 Stride=1 Leaky ReLU
64
Conv-1D Filters=128 Stride=1 Leaky ReLU
128
Conv-1D Filters=34839 Stride=1 Leaky ReLU
34839

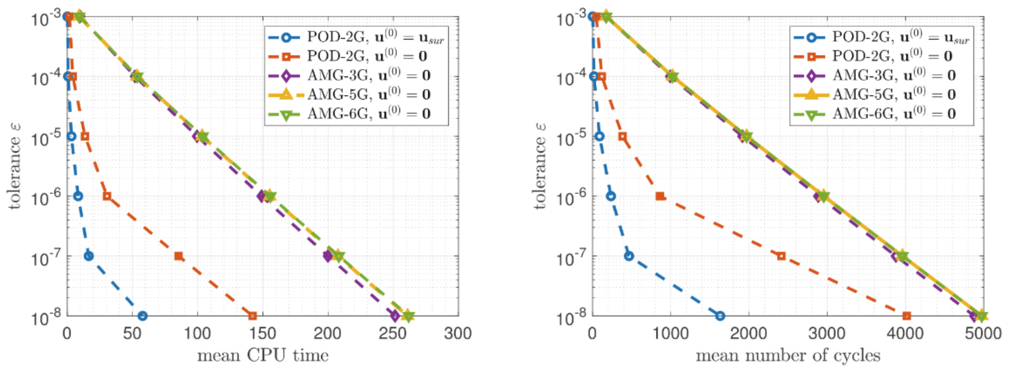
Figure 2: Surrogate model architecture

Subsequently, a number of N_{test} parameter vectors $\{\{\mu_i, \lambda_i\}_{i=1}^{N_{test}}$ were generated according to their distribution and the corresponding problems were solved with the proposed POD-based solver and different Ruge-Stüben AMG solvers, with the number of grids ranging from 2 to 6. The size of the system of equations at the coarsest level for each of these solvers is presented in Table 2. For this example, eight eigenvectors were retained in the POD expansion, as these were sufficient for capturing 99.99% of the dataset's variance.

	System size
Initial problem	34839×34839
AMG-2G	8625×8625
AMG-3G	1421×1421
AMG-4G	229×229
AMG-5G	47×47
AMG-6G	9×9
POD-2G	8×8

Table 2: Size of the problem at the coarsest grid for the different solvers

The mean value of the CPU time and the number of cycles required for convergence to the desired value of tolerance are displayed in Figure 3 and Table 3. The results are very promising in terms of computational cost. For instance, for $\varepsilon = 10^{-5}$ and $\boldsymbol{u}^{(0)} = \mathbf{0}$, a reduction of computational cost of ×7.32 is achieved when comparing the proposed solver with the 3-grid AMG solver. Furthermore, obtaining an accurate initial solution $\boldsymbol{u}^{(0)}$ is again a very important component of the proposed framework. Specifically, by considering $\boldsymbol{u}^{(0)} = \boldsymbol{u}_{sur}$ instead of $\boldsymbol{u}^{(0)} = \mathbf{0}$ for $\varepsilon = 10^{-5}$, an additional decrease in CPU time of ×4.31 can be achieved.



(A) Comparison of mean CPU time

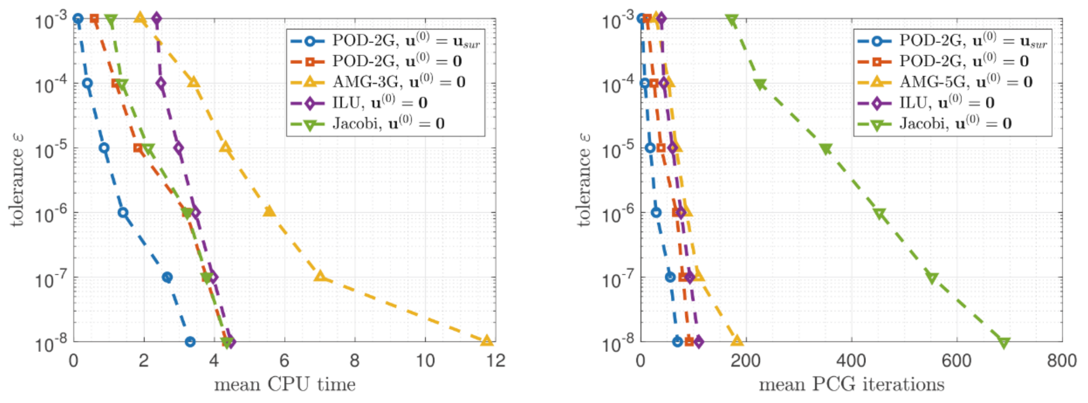
(B) Comparison of mean number of cycles

Figure 3: Comparison of mean CPU time and mean number of cycles over 500 analyses for different multigrid solvers

	$\varepsilon = 10^{-4}$	$\varepsilon = 10^{-5}$	$\varepsilon = 10^{-6}$	$\varepsilon = 10^{-7}$	$\varepsilon = 10^{-8}$
AMG-3G ($\boldsymbol{u}^{(0)} = \mathbf{0}$)	×1.00	×1.00	×1.00	×1.00	×1.00
AMG-5G ($\boldsymbol{u}^{(0)} = \mathbf{0}$)	×0.97	×0.96	×0.96	×0.96	×0.96
AMG-6G ($\boldsymbol{u}^{(0)} = \mathbf{0}$)	×0.97	×0.96	×0.96	×0.96	×0.96
POD-2G ($\boldsymbol{u}^{(0)} = \mathbf{0}$)	×12.31	×7.32	×4.89	×2.34	×1.77
POD-2G ($\boldsymbol{u}^{(0)} = \boldsymbol{u}_{sur}$)	×76.89	×31.54	×17.90	×12.12	×4.35

Table 3: Computational speedup of solvers compared to AMG-3G

The convergence behavior of the proposed method when used as a preconditioner in the context of the PCG method is presented in Figure 4. Again, the results delivered by the proposed methodology showed its superior performance not only over AMG preconditioners but also over ILU and Jacobi preconditioners. In this case, for $\varepsilon = 10^{-5}$ and $\boldsymbol{u}^{(0)} = \mathbf{0}$, a reduction of computational cost of ×2.37 is observed between the proposed method and the 3-grid AMG, of ×1.63 with the ILU and of ×1.16 with the Jacobi. Finally, the initial solution delivered by the surrogate model, $\boldsymbol{u}^{(0)} = \boldsymbol{u}_{sur}$, managed to further reduce the computational time by ×2.12 when compared to POD-2G with $\boldsymbol{u}^{(0)} = \mathbf{0}$ (See Table 4).



(A) Comparison of mean CPU time

(B) Comparison of mean PCG iterations

Figure 4: Comparison of mean CPU time and mean number of PCG iterations over 500 analyses for different preconditioners

	$\varepsilon = 10^{-4}$	$\varepsilon = 10^{-5}$	$\varepsilon = 10^{-6}$	$\varepsilon = 10^{-7}$	$\varepsilon = 10^{-8}$
AMG-3G ($\boldsymbol{u}^{(0)} = \mathbf{0}$)	×1.00	×1.00	×1.00	×1.00	×1.00
ILU ($\boldsymbol{u}^{(0)} = \mathbf{0}$)	×1.38	×1.45	×1.61	×1.77	×2.63
Jacobi ($\boldsymbol{u}^{(0)} = \mathbf{0}$)	×2.50	×2.04	×1.73	×1.85	×2.70
POD-2G ($\boldsymbol{u}^{(0)} = \mathbf{0}$)	×2.86	×2.37	×1.74	×1.86	×2.71
POD-2G ($\boldsymbol{u}^{(0)} = \boldsymbol{u}_{sur}$)	×8.88	×5.02	×3.98	×2.64	×3.55

Table 4: Computational speedup of different preconditioners compared to the AMG-3G preconditioner

2.2 POD-2G method: Application to parameterized dynamic problems

As evidenced from the results of the previous section, the ML-enhanced preconditioners that were developed in the context of POD-2G methodology showed significant computational gains for parameterized static problems. In this section we present an extension of this framework, which was developed as part of the works in Task 3.2 “*Surrogate models for preconditioning and coarse problem solution*” in order to handle dynamic systems of high-dimensionality. The idea is to leverage a set of high-fidelity system solutions to construct a reduced basis for the system’s response. In this case, the data set consists of system responses for different parameter values that are collected at multiple time instances of the system’s evolution (system snapshots). Then, these reduced basis vectors will be subsequently used for the development of efficient preconditioners that will accelerate the convergence of the preconditioned Conjugate Gradient method for a system with temporal dependency.

The test case presented in this deliverable involves a 2D plane stress problem of a high-rise building subjected to a dynamic load at its top, as shown in figure 5. This problem involves ~12000 unknowns (degrees of freedom) and 100 time steps, while it is assumed that the Young modulus of the material is a lognormal stochastic field that varies at each position (x, y) of the structure. Therefore, the aim in this application is to obtain the probabilistic characteristics of the response via a brute force Monte Carlo simulation that requires a large number of model simulations. Each model simulation is solved using a PCG solver with (a) the standard Jacobi diagonal preconditioner and (b) the POD-2G preconditioner from the AI-Solve library, which is formed from the most ‘energetic’ eigenvectors in the solution data set.

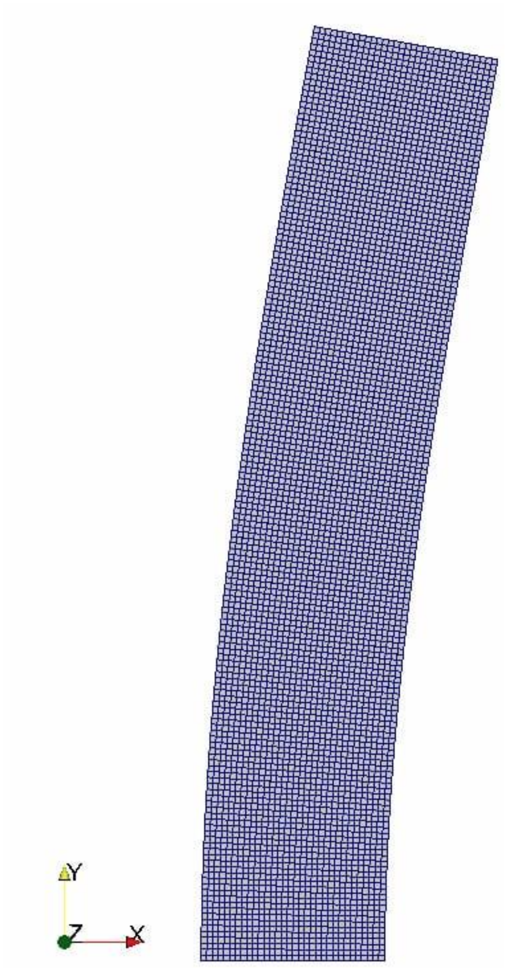


Figure 5: High-rise building under dynamic load

The numerical results of this investigation are presented in the table below, which compares the average wall-clock time (per 1000 Monte Carlo simulations) and the average number of iterations per time step needed to achieve convergence between the two approaches. In particular, this table investigates the influence of the number of eigenvectors kept to construct the POD-2G preconditioner. It should be mentioned that for this case, we performed 100 simulations for different realizations of the Young's modulus field and we stored every solution snapshot. We notice the optimal number of eigenvectors is 10 which achieves a speedup of 5.24x, compared to solving the problem with the standard Jacobi-preconditioned CG.

Eigenvectors kept	Average number (per time step) of PCG iterations with Jacobi preconditioner	Average number (per time step) of PCG iterations with AI-Solve	Average time (per simulation) for PCG with Jacobi preconditioner	Average time (per simulation) with AI-Solve's version of PCG	Speedup per analysis
1	754	142	23255 (ms)	9720 (ms)	2.39x
5	754	68	22170 (ms)	5040 (ms)	4.40x
10	**754**	**54**	**22045 (ms)**	**4205 (ms)**	**5.24x**
15	754	52	22025 (ms)	4295 (ms)	5.12x
20	754	50	22365 (ms)	4560 (ms)	4.90x

Table 5: Speedup of AI-Solve compared to the Jacobi-preconditioned CG for different numbers of eigenvectors kept.

Next, we investigate the optimal strategy to construct the solution data set. By retaining only a part of the solution snapshots instead of all of them would reduce the training (offline) cost needed to form the POD-2G preconditioner. However, this will impact the online solution cost as a coarser training data set will inevitably contain less information. In this direction, tables 6-8 present the speedup when the solution snapshots are stored per 5, 10 and 15 time steps, respectively.

Eigenvectors kept	Average number (per time step) of PCG iterations with Jacobi preconditioner	Average number (per time step) of PCG iterations with AI-Solve	Average time (per simulation) for PCG with Jacobi preconditioner	Average time (per simulation) with AI-Solve's version of PCG	Speedup per analysis
1	754	142	21270 (ms)	9705 (ms)	2.19x
5	754	68	23030 (ms)	5305 (ms)	4.34x
10	**754**	**54**	**23040 (ms)**	**4375 (ms)**	**5.27x**
15	754	52	22985 (ms)	4495 (ms)	5.11x
20	754	50	23065 (ms)	4760 (ms)	4.85x

Table 6: Speedup of AI-Solve compared to the Jacobi-preconditioned CG for different numbers of eigenvectors kept. The solution snapshots in the data set were stored per 5 time steps.

Eigenvectors kept	Average number (per time step) of PCG iterations with Jacobi preconditioner	Average number (per time step) of PCG iterations with AI-Solve	Average time (per simulation) for PCG with Jacobi preconditioner	Average time (per simulation) with AI-Solve's version of PCG	Speedup per analysis
1	754	142	22230 (ms)	9825 (ms)	2.26x
5	754	68	22230 (ms)	5055 (ms)	4.40x
10	**754**	**54**	**22250 (ms)**	**4215 (ms)**	**5.28x**
15	754	52	22200 (ms)	4300 (ms)	5.16x
20	754	50	23340 (ms)	4560 (ms)	4.90x

Table 7: Speedup of AI-Solve compared to the Jacobi-preconditioned CG for different numbers of eigenvectors kept. The solution snapshots in the data set were stored per 10 time steps.

Eigenvectors kept	Average number (per time step) of PCG iterations with Jacobi preconditioner	Average number (per time step) of PCG iterations with AI-Solve	Average time (per simulation) for PCG with Jacobi preconditioner	Average time (per simulation) with AI-Solve's version of PCG	Speedup per analysis
1	754	142	22190 (ms)	9830 (ms)	2.26x
5	754	68	22260 (ms)	5065 (ms)	4.40x
10	**754**	**54**	**22590 (ms)**	**4245 (ms)**	**5.32x**
15	754	52	22265 (ms)	4360 (ms)	5.11x
20	754	50	22290 (ms)	4625 (ms)	4.82x

Table 8: Speedup of AI-Solve compared to the Jacobi-preconditioned CG for different numbers of eigenvectors kept. The solution snapshots in the data set were stored per 15 time steps.

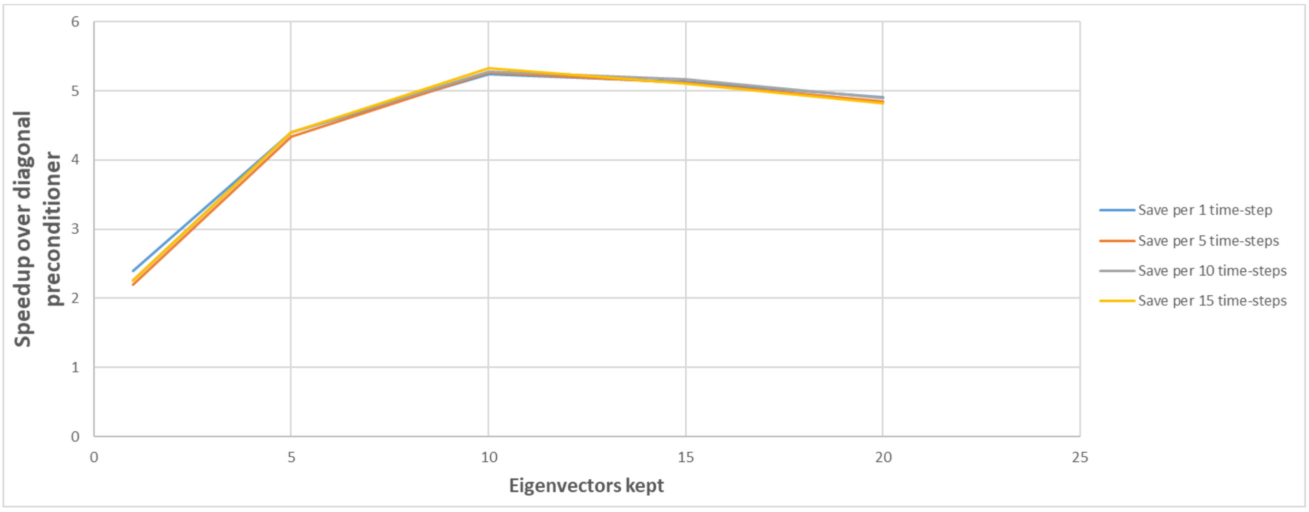


Figure 6: Summary of results from tables 5-8.

Based on the results of tables 5-8, also summarized in figure 6, the conclusion is drawn that all strategies to store solution snapshots provide similar results, but the cost for obtaining the eigenvectors for the solution matrices drastically reduces from 400 seconds, when storing each snapshot, to 0.3 seconds, when storing 1 snapshot per 15 time steps. Evidently, the most important aspect in the proposed framework is the number of eigenvectors kept and a remarkable 5x speedup is attained when constructing the AI-Solve's ML-enhanced preconditioner using 10 of them.

2.3 Transformer based-surrogate models – Application to the nonlinear transient analysis of a water tower under random base excitation

Another surrogate modeling technique that was developed within the premise of Task 3.2 "*Surrogate models for preconditioning and coarse problem solution*" of the project, leverages the novel Transformer neural network architecture to construct surrogate models for nonlinear dynamic problems. This data-driven solution framework has been already presented in Section 3.2 of deliverable 3.2 and was recently published in [2]. It provides a novel methodology to produce accurate initial guesses to nonlinear iterative solvers by employing a state-of-the-art variant of Transformers, namely the Temporal Fusion Transformers (TFTs). The goal is to reduce the total number of Newton-Raphson nonlinear iterations by using the surrogate's predictions, thus effectively decreasing the solution time for nonlinear transient problems under random loading conditions.

In this example, the water tower shown in figure 7 is initially subjected to a series of monochromatic seismic ground accelerations of the form $\boldsymbol{p}_{ext} = -\boldsymbol{M} \cdot \boldsymbol{1}_d \cdot A \cdot sin(\theta \cdot t)$, where $\boldsymbol{M}$ is the mass matrix, $\boldsymbol{1}_d$ is the excitation's directivity vector, A is an amplitude modifier and θ is the angular frequency of the excitation, which is considered a system parameter ranging from 0 to 0.5 . The structure's model consists of 4104 tetrahedral elements and 1767 nodes corresponding to d = 5241 degrees of freedom.

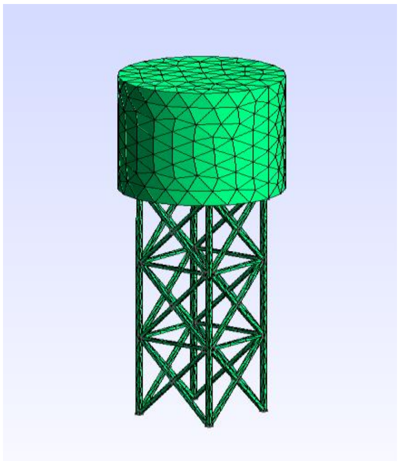


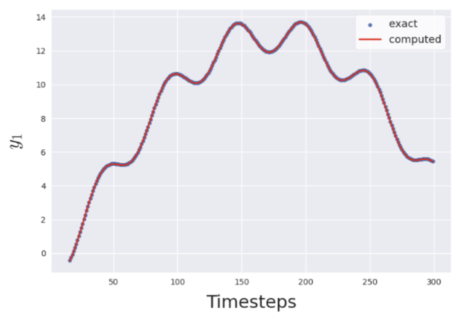
Figure 7: Water tower geometry and finite element discretization

Initially, the response of the structure is calculated for N_{train} separate angular frequency values considering zero initial conditions for each performed analysis. The second step of the proposed framework is to start by conducting a PCA analysis and evaluate how the captured variance changes as the number of principal components increases. After careful observation, we opted to choose the first $d_{PCA} = 9$ components as they effectively retained 99.98% of the original variance for both the solution and the related force vector space.

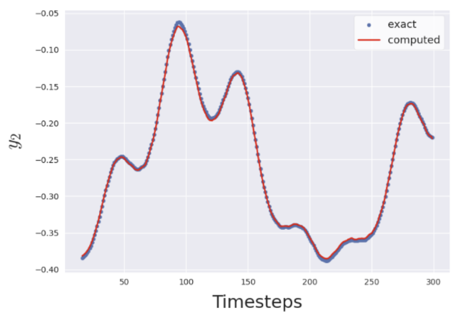
Following that, we proceeded to train nine distinct scalar-valued TFTs for a total of 2000 epochs. Each TFT corresponded to one of the latent space dimensions that were derived. The models were trained to generate one-step-ahead forecasts, where the value k of the length of the look-back window was set to 16. Their performance on the training dataset is summarized in Table 9. Figure 8 illustrates the performance of the TFTs' prediction on the temporal evolution of the 9 PCA components over time for the sample θ in the training data set with the median test percentage error.

Performance	% training error
Overall % loss	0.2813
Best sample	0.1804
Worst sample	0.7579

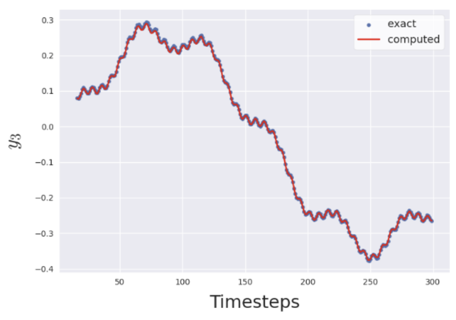
Table 9: Percentage training error



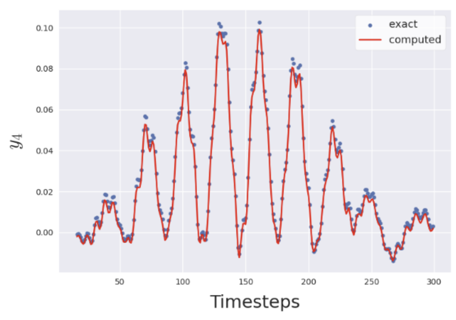
(a) TFT_1



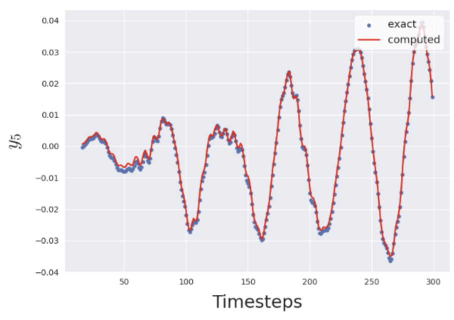
(b) TFT_2



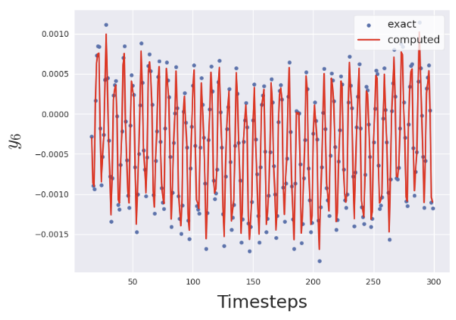
(c) TFT_3



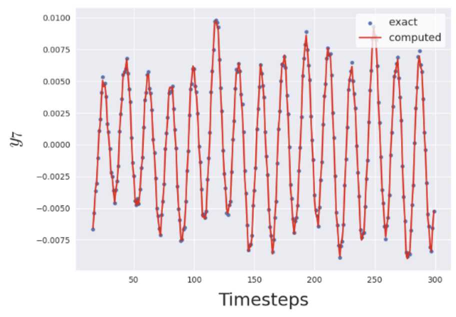
(d) TFT_4



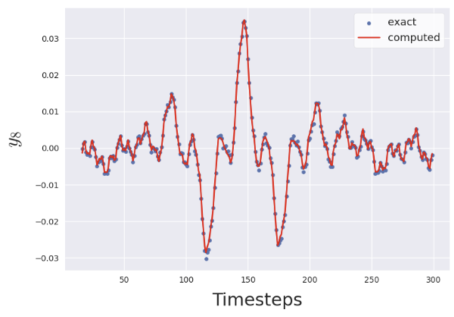
(e) TFT_5



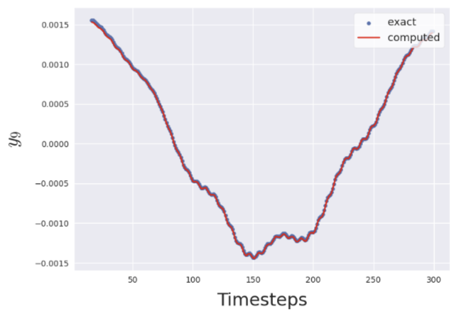
(f) TFT_6



(g) TFT_7



(h) TFT_8



(i) TFT_9

Figure 8: Performance of the TFTs' prediction on the temporal evolution of the 9 PCA components over time for the sample θ in the training data set with the median test percentage error.

The next step was to generate artificial ground motions that resemble actual earthquakes using the Spectral Representation method [3]. Then, to test the performance of the previously trained surrogate model on this more challenging scenario, we conducted 1200 simulations of the FE model for different ground motions using the surrogate's predictions as input to the FE nonlinear solver. We observed a noticeable reduction in the average values of NR iterations needed, as shown in table 10, even though the surrogate wasn't trained on this type of excitations.

Iterations without surrogate	**Iteration with surrogate**
1	1.000
2	1.121
3	1.374
4	1.513
5	2.419
6	3.471
7	3.857
8	2.2
9	3.5

Table 10: Extrapolation Performance on the earthquake time series: The right column illustrates the average iteration count for our solver to converge, utilizing surrogate's predictions as initial solutions. These averages are calculated across time series where the number of iterations needed prior to employing our approach corresponds to the values in the left column, highlighting our approach's effectiveness in handling multiple-frequency time series.

In addition, figure 9 shows the average number of iterations per time step for 1200 simulations with and without the use of the surrogate. In this test case, the average number of iterations per simulation without the use of the surrogate was 1246.88 and the total number of iterations was 1496256 for the 1200 simulations. However, using the surrogate's predictions as initial solutions, we managed to drop these numbers to 576,13 and 691356, respectively, as shown in figure 10. These results provide a strong indication that the TFT models are able to learn the system's dynamics in the latent space and can extrapolate beyond the training data set.

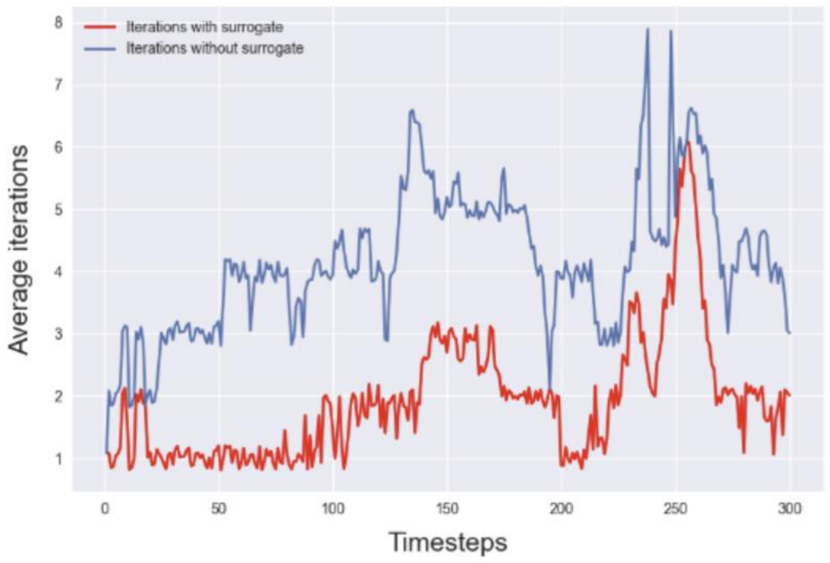


Figure 9: Per timestep comparison on the earthquake timeseries

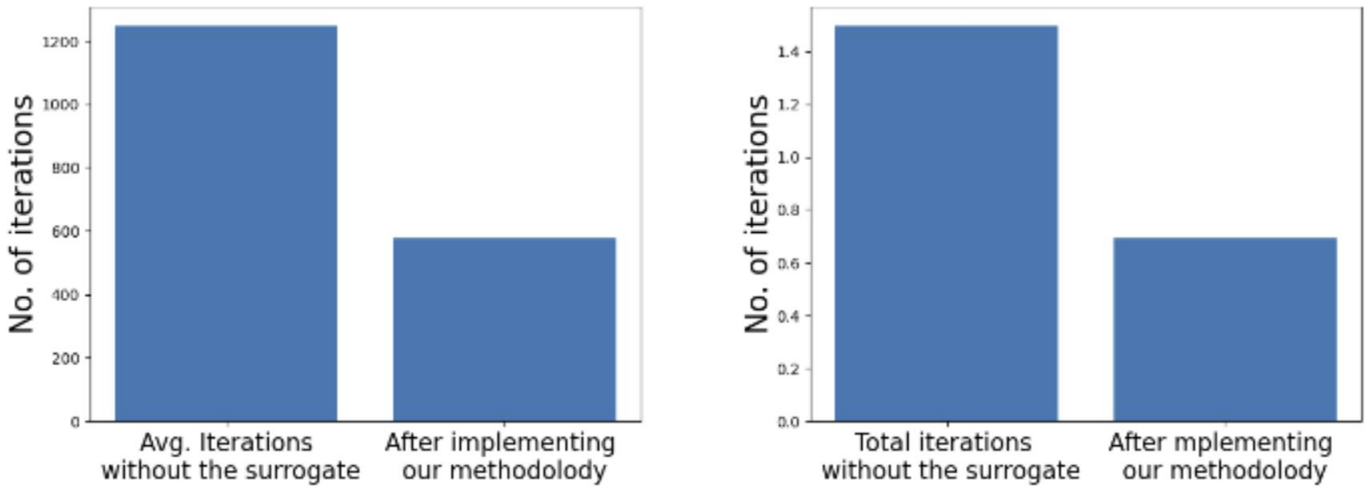


Figure 10: Extrapolation Results on the earthquake time series: This figure compares convergence metrics before and after applying our methodology, showcasing the contrast in average iterations for individual time series on the left and the total iterations for the entire dataset on the right

3. Solvers for Parallel Computing Environments

3.1 Parallel DDM method – Application to crack propagation problems

The extended finite element method (XFEM) is one of the most popular methods to simulate fracture phenomena. However, this approach requires the solution of large-scale systems that are often ill-conditioned and thus require specialized iterative solving techniques. To tackle such problems, in [4] several variants of Domain Decomposition Methods (DDM) were implemented according to task 3.1 "*DDM preconditioners for exascale systems*" of the DCoMEX project. The family of DDM algorithms constitutes an integral part of the AI-Solve library due to the computational advantages these offer in terms of accuracy and scalability, as well as their ability to handle complex problems such as crack propagation.

The theory and algorithmic implementation of DDM algorithms has been presented in section 2.5 of D3.2,
while this report presents the results of the numerical investigation of a crack propagation problem in a beam supported at three points and loaded at a fourth point, as illustrated in figure 11. This problem was solved with several DDM variants and their scalability was assessed.

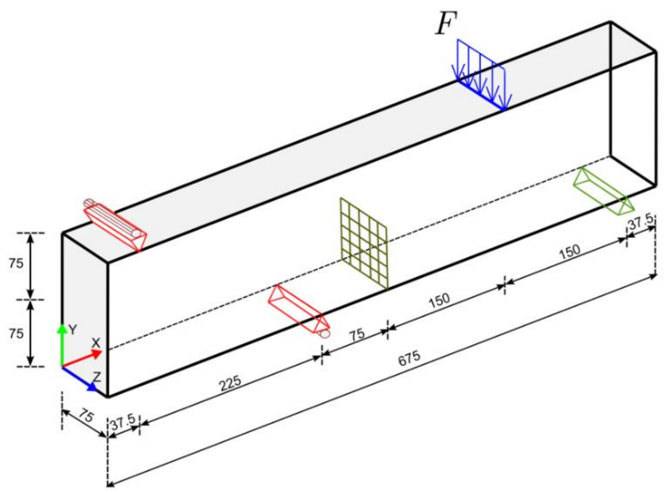


Figure 11: 4-point bending beam test case. Geometry, boundary conditions and initial configuration of the crack surface (dimensions in mm)

The material properties are $E = 3 \cdot 10^7 \frac{N}{mm^2}$, $v = 0.3$ and the applied load is $F = 1000\ N$. The dimensions of the beam, the placement of supports and load and the initial configuration of the crack surface are given in the figure. It is assumed that the crack propagates in a quasi-static manner with a constant increment, until it reaches the boundary of the domain and collapse occurs after 13 propagation steps. The whole analysis is repeated for various mesh densities, with each mesh consisting of 8-node hexahedral elements. The initial and final number of dof for each mesh are listed in the table below:

Mesh	dof at first step	dof at last step
45 x 10 x 5	9,492	9,816
90 x 20 x 10	64,130	65,384
135 x 30 x 15	204,336	206,880
180 x 40 x 20	470,820	475,419
225 x 50 x 25	903,812	910,832
270 x 60 x 30	1,537,228	1,548,295
315 x 70 x 35	2,431,872	2,444,940

Table 11: Number of dof per mesh. The difference in dof between the first and last step of the crack propagation analysis is due to the mesh enrichment required by XFEM to capture the crack's advancing front

To test the performance of the DDM methods, the following cases were considered: Dirichlet preconditioned FETI-DP (abbreviated as FETI-DP-D), lumped-preconditioned FET-DP (abbreviated as FETI-DP-L), and the P-FETI-DP. The improved versions of these methods using re-initializations techniques for crack propagation problems are denoted as FETI-DP-D-I, FETI-DP-L-I and P-FETI-DP-I. The speedup each of these methods achieved is compared to supernodal sparse Cholesky factorization direct solver [5]. The results of these analyses are presented in figure 12.

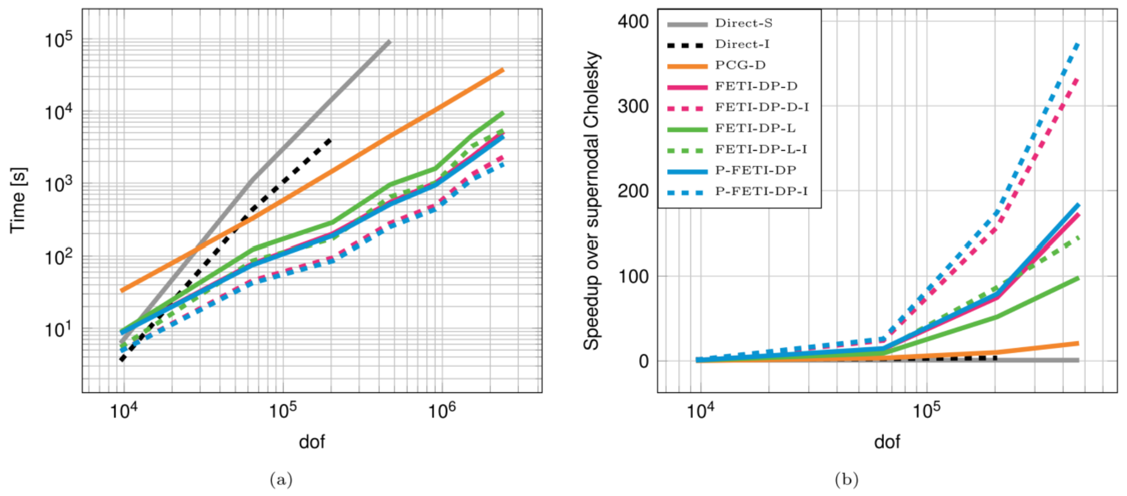


Figure 12: Performance comparison of the DDM solvers for the 4-point bending beam problem: (a) Computing time (in seconds), (b) speedup of the solvers relative to the supernodal Cholesky solver.

As evidenced by the results in figure 12, all DDM methods exhibit superior computational performance in comparison with the direct solver and the PCG method. Also, the P-FETI-DP and its improved version P-FETI-DP-I were the most efficient solution techniques from the family of DDM methods.

3.2 Parallel block-CG method - Application to a 3D elasticity problem

Besides parallel DDM methods, the AI-Solve library contains another strategy, the block CG method, which leverages the parallel computing capabilities offered by modern systems to solve extremely large-scale linear systems. The theory behind block CG has been presented in section 2.3 of D3.2 and the aim of this report is to demonstrate the scaling properties of the algorithm. Also, the synergy of DDM and block-CG algorithms has been investigated as part of the work in Tasks 3.1 "*DDM preconditioners for exascale systems*" and 3.5 *"Communication optimization"* of the project, and the findings are presented in this section.

In our numerical investigation we study a 3D elastictity problem of a cantilever subjected to a concentrated force on its free end. The structure is discretized into $144 \times 48 \times 48$ hexahedral elements leading to a total of 1.03×10^6 degrees of freedom (see figure 13). The main advantage of block PCG over regular PCG is the significant reduction in I/O communication costs it achieves when used in distributed memory environments. With block PCG, the speedup in I/O communication costs is equal to the block size s.

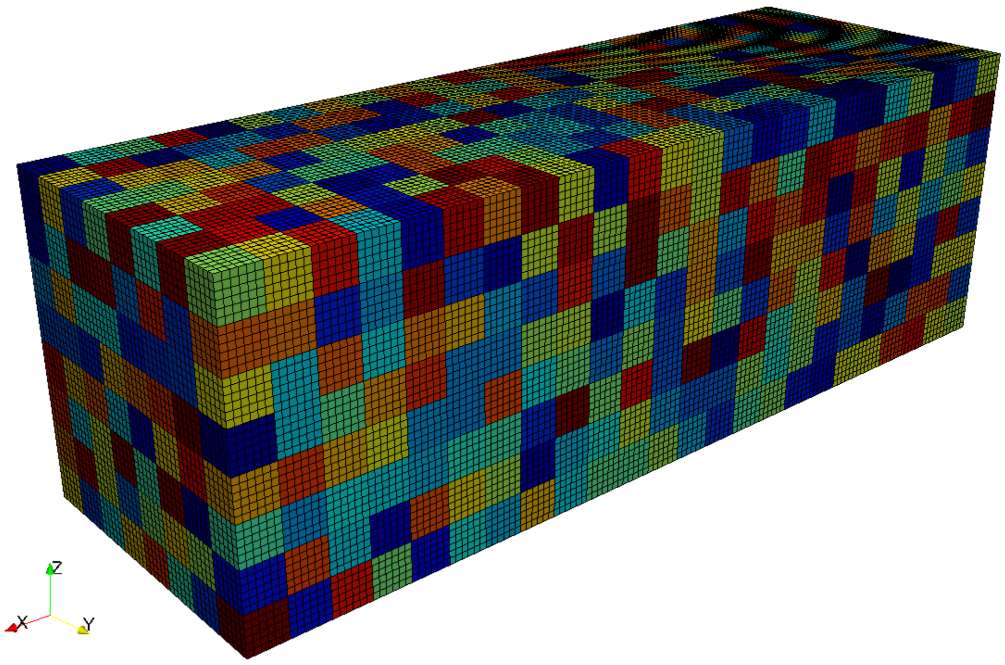


Figure 13: Discretized cantilever into $144 \times 48 \times 48$ hexahedral elements and $24 \times 8 \times 8$ subdomains.

		Tolerance 10^{-8}	
		Iterations	**Time**
	PCG	818	4509.3 sec
Block PCG	s=1	818	4537.9 sec
	s=2	818	4543.1 sec
	s=3	818	4548.8 sec
	s=4	818	4555.5 sec
	s=5	966	5362.4 sec
	s=6	1170	6498.5 sec
	s=7	1204	6684,4 sec
	s=8	1762	9786,1 sec

Table 12: Effect of block size in Jacobi-preconditioned block PCG

As evidenced by the results of table 12, block sizes $s < 5$ the PCG iterations remain constant. For larger block sizes, an increase of block PCG iterations is observed, as expected. To circumvent this problem, we combined block PCG with a domain decomposition method. When combining P-FETI-DP with block PCG, the iterations required for convergence remain constant until a larger block size of $s = 8$ (see table 13). In turn, this allows further reduction of I/O operations making block PCG a more attractive choice. It is worth mentioning that the interface problem of P-FETI-DP is solved using the block PCG method, whereas the coarse problem of P-FETI-DP is small enough to be handled by direct solver (Cholesky).

		Tolerance 10^{-8}	
		Iterations	**Time**
	PCG	100	913.2 sec
Block PCG	s=1	100	916.5 sec
	s=2	100	917.1 sec
	s=3	100	917.9 sec
	s=4	100	919.6 sec
	s=5	100	920.4 sec
	s=6	100	921.8 sec
	s=7	100	921.3 sec
	s=8	112	1028.9 sec

Table 13: Effect of block size in block PCG combined with P-FETI-DP

3.3 Inexact block-iterative solvers - Application to a 2D elasticity problem

Another promising research direction to speed up the solution process of large-scale problems is by employing inexact arithmetic techniques. This approach has been studied as part of task 3.3 "*Inexact block-iterative solvers for scalability and error resilience*" of the project. A mixed precision PCG implementation is proposed in which all computations are performed in single precision, except for a double precision computation of the matrix vector multiplication occurring during the recursive evaluation of the residual vector in the PCG algorithms. This implementation is a robust and reliable solution procedure, even for handling large and ill-conditioned problems, while it is also computer storage effective.

To test this approach, we study the cantilever shown in previous figure 5 with a static load imposed at its free tip and we perform a Monte Carlo simulation for different realizations of the modulus of elasticity. In this example, we compare the direct Skyline Solver, the incomplete Cholesky preconditioned PCG and the Neumann CG algorithms with double precision, (Direct, PCG-DP and NCG-DP, respectively) against the single precision PCG and NCG. The comparison is performed in terms of storage requirements, CG iterations needed for convergence and CPU time.

Figure 14 compares the storage requirements between the direct solver and the four iterative algorithms. We notice a reduction of almost 50% in the Mbytes required to store the system matrices for the single precision preconditioners over their double precision counterparts. In terms of CG iterations, figure 15 illustrates their average number per MC simulation, where we observe that the single precision preconditioners require slightly more iterations to achieve convergence. However, the total CPU time after 1000 MC simulations (fig. 16) is significantly smaller when using the single precision preconditioners, achieving a speedup of almost 35% with respect to the double precision ones. Based on these findings, the conclusion is drawn that inexact arithmetic techniques are very efficient in reducing the computational requirements for large-scale problems.

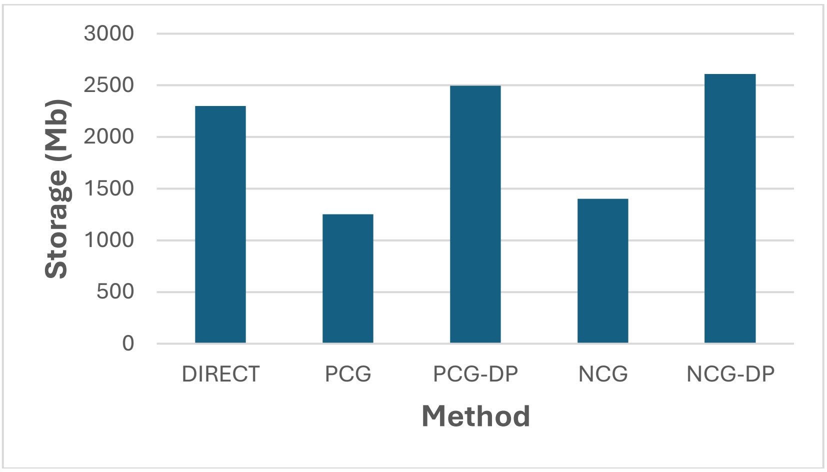


Figure 14: Storage requirements for the preconditioners with double and single precision

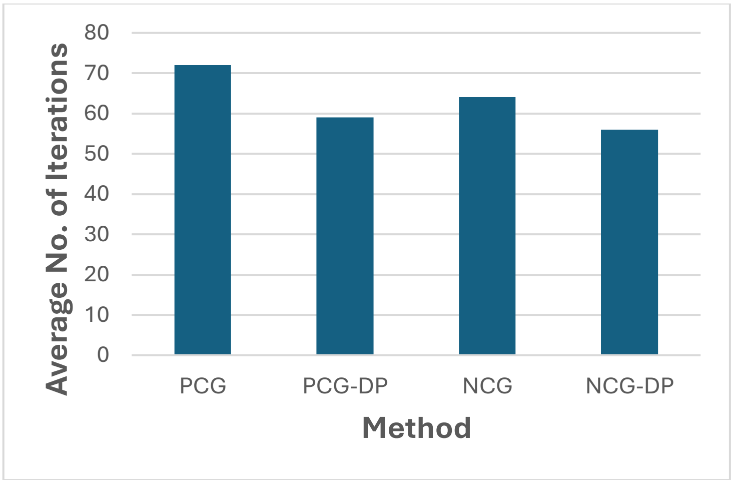


Figure 15: Average number of CG iterations per MC simulation

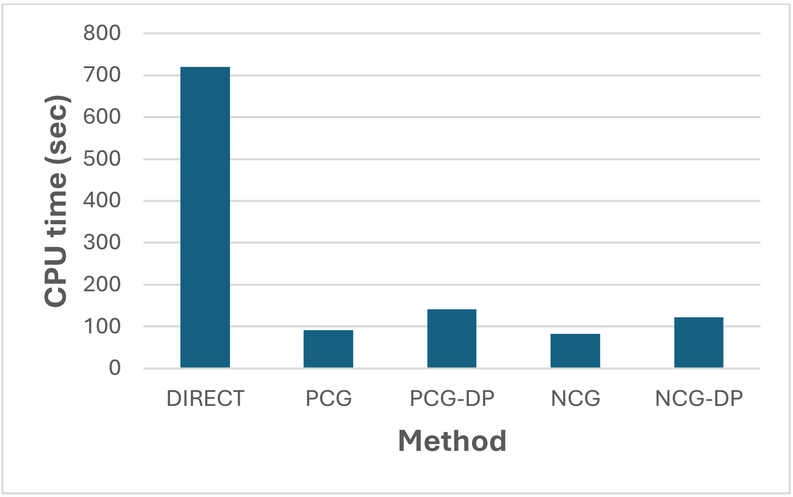


Figure 16: Total CPU time after 1000 MC simulations

3.4 Sparse Matrix-Vector Multiplication

The last section of this deliverable is devoted to the works done in Task 3.4 “*Scalable sparse computations*” of the project. Sparse Matrix-Vector (SpMV) multiplication is encountered in many scientific fields including scientific computing and machine learning. It performs the product of a sparse matrix with a dense input vector and returns a dense output vector. SpMV is one of the lowest-performing kernels, a fact that can be attributed to many potential performance issues, like its extremely low operational intensity (memory bandwidth bound), irregular memory accesses to the input vector (memory latency bound), load imbalance and low instruction-level parallelism ILP). While a lot of work has been done to mitigate these problems, as matrices grow larger and the gap between computational power

and memory speed widens, the memory bandwidth bottleneck remains the most important obstacle to high performance.

Data compression emerges as a natural solution to address the memory bandwidth bottleneck of SpMV. Compression sacrifices unused processing power to reduce the memory footprint and, consequently, the required bandwidth. Most of the research effort so far has focused on compressing the redundancy in the structure of the matrix, i.e., reducing the indexing footprint of sparse matrices. In the generic sparse representation format CSR, indexing metadata include approximately (n + nnz) 32-bit integers, where n and nnz are the number of rows and nonzero elements of the matrix respectively. Many applications however, require double precision arithmetic, which means that the memory footprint of the actual values of the matrix is nnz 64-bit floats, clearly comprising almost 2/3 of the total matrix footprint.

As part of the DCoMEX project, we have implemented an approach that aggressively compresses the memory footprint of a sparse matrix and thus we target both its metadata (indexing structure) and actual data. The scheme applies combined index and value compression. Our approach is currently applied to CPUs and is designed to be effective for large-scale matrices that exceed the size of the Last Level Cache (LLC), where the memory bandwidth limitation is much more severe. The potential adaptation of the method to other architectures like GPUs and FPGAs, where memory bandwidth limitations are also present, is left for future work. Our scheme utilizes delta encoding techniques, effectively reducing the matrix memory footprint, and is capable of drastically improving the SpMV performance of large matrices. We achieve lossless matrix compression by leveraging integer arithmetic to calculate the deltas of floating point values. The compression process has a rather low overhead, in the order of a few dozens of SpMV operations on average, making it effective both for offline and online processing, when the SpMV is applied in solvers with multiple iterations.

In Figure 17 we present a performance evaluation on a 16-core Intel Xeon Silver 4314 platform. The dataset used is presented in Table 14. We compare against several other double-precision formats, namely a) two utilizing dictionary based value compression, **CSR&RV** [6] and a custom implementation, b) **LCM** [7], a format that searches for strided and partially strided substructures in the sparse matrix and generates an optimized and vectorized codelet for each one, c) **SparseX** [8], an academic format that aggressively tries to reduce the size of the structural metadata of sparse matrices (i.e., indices) and d) the Intel **MKL** library’s Inspector-Executor format. We also include a float variant of the MKL format for comparison. Our format (**DIV** / **DIV_RF**) achieves 1.44x speedup over the median performance of MKL double-precision and manages to approach the MKL single-precision performance.

Matrix Name	Symmetry	NNZ	CSR Memory Footprint	Unique Values Fraction	Memory Ratio (%)
cage15	G	99M	1154	0.00054	33.77
ML_Geer	G	111M	1274	99	69.62
Flan_1565	S	117M	1349	0.65	30.39
Cube_Coup_dt0	S	127M	1464	15	42.58
Cube_Coup_dt6	S	127M	1464	15	42.65
Bump_2911	S	128M	1472	51	57.58
vas_stokes_4M	G	132M	1522	3.8	44.93
nlpkkt160	S	230M	2658	0.0016	33.07
HV15R	G	283M	3247	84	83.67
Queen_4147	S	329M	3786	48	62.92
stokes	G	349M	4041	3.1	46.92
nlpkkt200	S	448M	5191	0.0012	33.07

Table 14: Characteristics of datasets

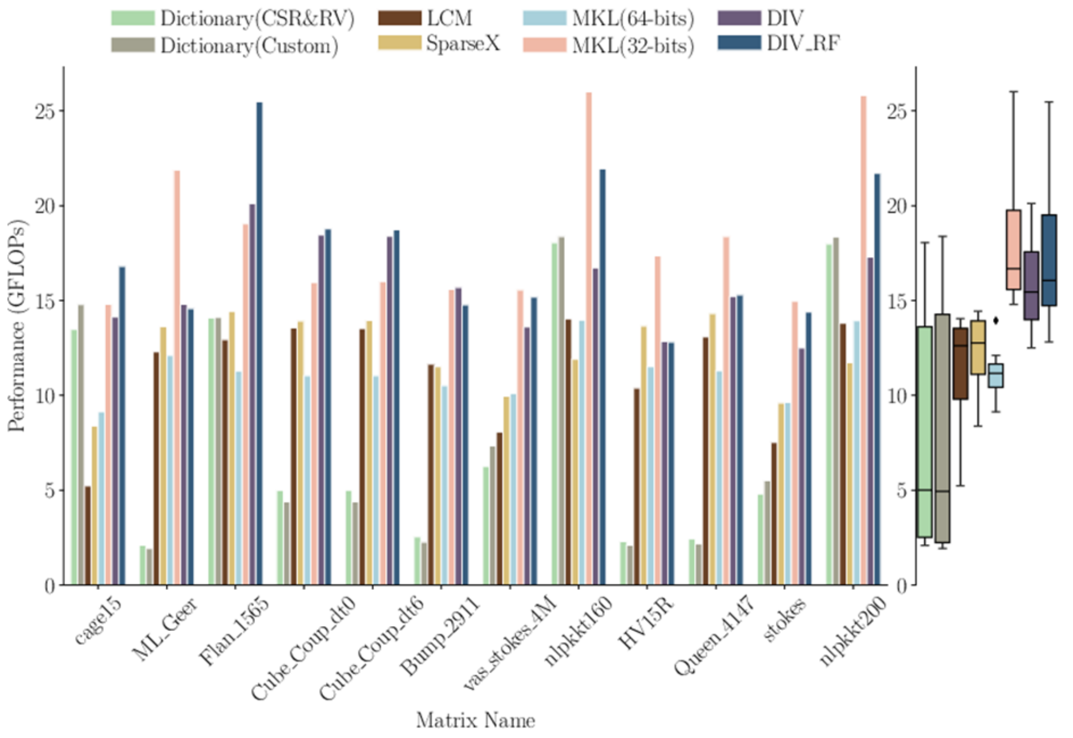


Figure 17: Performance benchmark on a 16-core Intel Xeon Silver 4314

4. Summary

In this report, we demonstrated the performance of the algorithms in the AI-Solve library through a series of diverse computational mechanics problems. Specifically:

- The POD-2G data-driven solution framework was tested on parameterized linear static and dynamic problems, showing significant computational gains over traditional solution approaches.

- The Transformer-based surrogate modeling strategy was applied to nonlinear transient problems, resulting in a substantial reduction in the number of nonlinear Newton-Raphson iterations needed for convergence at each time step.

- Various Domain Decomposition methods were implemented within the AI-Solve library, and their scalability was assessed. Benchmark analyses revealed that the P-FETI-DP and its improved version, P-FETI-DP-I, were the most efficient methods for solving large-scale systems in parallel environments.

- The block-PCG method was implemented and tested, demonstrating a notable reduction in I/O communication costs compared to the standard PCG method.

- A significant improvement in storage requirements and time-to-solution was reported using mixed precision arithmetics.

- A compression method for SPMV on large matrices has been developed and tested that achieves 1.44x speedup over the median performance of MKL double-precision and manages to approach the MKL single-precision performance.

References

1. S. Nikolopoulos, I.Kalogeris, G. Stavroulakis, V. Papadopoulos, "*AI-Enhanced iterative solvers for accelerating the solution of large-scale parametrized systems*", International Journal of Numerical Methods in Engineering, 2023

2. L. Papadopoulos, K. Atzarakis, G. Sotiropoulos, I. Kalogeris, V. Papadopoulos, "Fusing nonlinear solvers with transformers for accelerating the solution of parametric transient problems", Computer Methods in Applied Mechanics and Engineering, 2024

3. M. Shinozuka, G. Deodatis, "Simulation of Stochastic Process by Spectral Representation", Applied Mechanics Reviews, 1991

4. S. Bakalakos, M. Georgioudakis, M. Papadrakakis, "*Domain decomposition methods for 3D crack propagation problems using XFEM",* Computer Methods in Applied Mechanics and Engineering*,* 2022.

5. Y. Chen, T.A. Davis, W.W. Hager, S. Rajamanickam, "*Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate"*, ACM Trans. Math. Software, 2008.

6. J. Yan, X. Chen, J. Liu. "*Csr&rv: An efficient value compression format for sparse matrix-vector multiplication*", Network and Parallel Computing, 2022

7. K. Cheshmi, Z. Cetinic and M.M. Dehnavi "Vectorizing sparse matrix computations with partially-strided codelets", Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2022

8. A. Elafrou et al. "SparseX: A library for high-performance sparse matrix-vector multiplication on multicore platforms", ACM Transactions on Mathematical Software, 2018