Communication-aware Sparse Patterns for the Factorized Approximate Inverse Preconditioner

Sergi Laut
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
Barcelona, Spain
sergi.lautturon@bsc.es

Marc Casas
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
Barcelona, Spain
marc.casas@bsc.es

Ricard Borrell
Barcelona Supercomputing Center
Barcelona, Spain
ricard.borrell@bsc.es

ABSTRACT

The Conjugate Gradient (CG) method is an iterative solver targeting linear systems of equations $Ax = b$ where A is a symmetric and positive definite matrix. CG convergence properties improve when preconditioning is applied to reduce the condition number of matrix A. While many different options can be found in the literature, the Factorized Sparse Approximate Inverse (FSAI) preconditioner constitutes a highly parallel option based on approximating A^{-1}. This paper proposes the *Communication-aware Factorized Sparse Approximate Inverse preconditioner (FSAIE-Comm)*, a method to generate extensions of the FSAI sparse pattern that are not only cache friendly, but also avoid increasing communication costs in distributed memory systems. We also propose a filtering strategy to reduce inter-process imbalance. We evaluate FSAIE-Comm on a heterogeneous set of 39 matrices achieving an average solution time decrease of 17.98%, 26.44% and 16.74% on three different architectures, respectively, Intel Skylake, Fujitsu A64FX and AMD Zen 2 with respect to FSAI. In addition, we consider a set of 8 large matrices running on up to 32,768 CPU cores, and we achieve an average solution time decrease of 12.59%.

CCS CONCEPTS

• **Mathematics of computing → Mathematical software performance**; *Solvers*; • **Computing methodologies → Distributed algorithms**; Linear algebra algorithms.

KEYWORDS

Conjugate Gradient; FSAI; SpMV

ACM Reference Format:
Sergi Laut, Marc Casas, and Ricard Borrell. 2022. Communication-aware Sparse Patterns for the Factorized Approximate Inverse Preconditioner. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '22), June 27-July 1, 2022, Minneapolis, MN, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3502181.3531472

The final publication is available at ACM via
http://dx.doi.org/10.1145/3502181.3531472

1 INTRODUCTION

Many scientific simulation workloads involve linear systems of equations. These systems usually appear on numerical methods to solve Partial Differential Equations (PDE) such as finite elements or finite differences. Iterative methods are commonly used to solve such systems, often defined by large sparse matrices, since they generally require much less memory and computation than direct methods like LU. In particular, Krylov methods solve a linear system $Ax = b$ by projecting the solution into a Krylov subspace that is created following an iterative process that considers powers of matrix A multiplied by the right-hand side vector b, that is, $\{b, Ab, A^2b, ..., A^mb\}$. When considering systems with symmetric and positive definite matrices, the popular Conjugate Gradient (CG) method [35], is typically applied.

The implementation of the CG algorithm involves three linear algebra kernels: the Sparse Matrix-Vector (SpMV) product $y = Ax$, the dot-product, and the linear combination of two vectors. These three kernels are memory bound and determine the performance of the CG solver. However, while the two latter kernels show regular memory access patterns, SpMV memory accesses on x are irregular as they depend on the sparsity pattern of A.

From the numerical point of view, preconditioning techniques are used to improve the conditioning of the system matrix A, and ultimately reduce the iterations required to converge to the solution. Among many types of preconditioners, such as Block-Jacobi [35] or Multi-Grid techniques [19], the Sparse Approximate Inverse (SAI) preconditioners [10, 11] offer an easy-to-apply and parallelizable method that only requires to add an extra SpMV product in the solver stage of the Krylov method. SAI consists in evaluating an approximation of the inverse $M \approx A^{-1}$ constrained to a sparse pattern. The preconditioned system is then $MAx = Mb$. When the system matrix is symmetric and positive definite the factorized version of the algorithm (FSAI) is used and A^{-1} is approximated by a factorization G^TG instead of a single matrix, M. The definition of the sparse pattern is a very important aspect that determines FSAI performance. State-of-the-art approaches define this pattern by considering numerical aspects and cache-aware algorithms [31], but they do not take into account any consideration regarding the communication pattern that the sparse matrix defines in a distributed memory scenario.

This paper proposes the *Communication-aware Factorized Sparse Approximate Inverse preconditioner (FSAIE-Comm)*, an approach to extend FSAI sparse patterns in distributed memory scenarios. This approach successfully increases the efficacy of the FSAI preconditioner without introducing any significant communication overhead between the different parallel processes of the distributed

memory execution. FSAIE-Comm achieves this low-overhead extension by just adding matrix entries that either correspond to the local data of each process, or involve communications between two processes for which the initial sparse pattern requires some degree of data exchange. Additionally, we propose a method to eliminate the load imbalance that our sparse pattern extension may introduce. This paper makes the following contributions:

- We propose FSAIE-Comm, a new method to extend the sparse pattern of FSAI in distributed memory scenarios. Our method does not increase communication overhead and introduces minimal memory traffic.
- We propose a dynamic filtering-out strategy to mitigate the inter-process load imbalance that FSAIE-Comm pattern extensions may introduce.
- We present an exhaustive evaluation campaign considering three high-end systems based on Intel Skylake, Fujitsu A64FX and AMD Zen 2 nodes. The average gains obtained in terms of time-to-solution are 17.98% and 26.35% and 16.74%, respectively, and executions with up to 32768 CPU cores have been considered.

2 BACKGROUND

2.1 Conjugate Gradient

The Conjugate Gradient (CG) method [35] is an iterative method to solve linear systems $Ax = b$, where A is symmetric and positive definite (SPD). In the ith iteration, the CG algorithm obtains the best solution approximation, x_i, with respect to the A-norm, that belongs to the subspace $x_0 + D^i$; where $D^i = \text{span}\{r_0, Ar_0, ..., A^{i-1}r_0\}$, r_0 is the initial residual, and the A-norm is defined as $\| v \|_A = \sqrt{v^T Av}$.

To find the approximation x_{i+1}, an A-orthogonal basis $\{d_0, d_1, ..., d_i\}$ of D^{i+1} is built by using the conjugate Gram-Schmidt method. In practice, this is done by adding a new element, d_i, to the basis $d_0, ...d_{i-1}$ previously derived for D^i:

$$\beta_i = \frac{r_i^T r_i}{r_{i-1}^T r_{i-1}},\ d_i = r_i + \beta_i d_{i-1},$$

where r_i refers to the ith residual and β_i is the Gram-Schmidt conjugation coefficient. In the new orthogonal basis, x_{i+1} can be expressed as

$$x_{i+1} = \alpha_0\, d_0 + \ldots + \alpha_{i-1}\, d_{i-1} + \alpha_i\, d_i, \tag{1}$$

where the coefficients $\alpha_0, ..., \alpha_{i-1}$ are evaluated in previous iterations. Hence, in the $i+1$th iteration it is only necessary to evaluate the component α_i associated to d_i to obtain x_{i+1}:

$$\alpha_i = \frac{r_i^T r_i}{d_i^T A d_i},\ x_{i+1} = x_i + \alpha_i d_i \tag{2}$$

Besides scalar operations, note that only three basic linear algebra operations are used through the steps of the algorithm: Sparse Matrix-Vector product (SpMV), linear combination of two vectors, referred as AXPY in the BLAS terminology, and dot-product.

2.2 Sparse Approximate Inverse Preconditioner

Sparse Approximate Inverse (SAI) preconditioners are based on the assumption that the inverse of the system matrix contains many relatively small entries that can be ignored to build a sparse approximation of the inverse to be used as preconditioner. In the setup process of the SAI method, an approximation of the inverse $M \approx A^{-1}$ constrained to a fixed sparse pattern $\mathcal{S}$ is found. Then, the preconditioned system $MAx = Mb$ is considered.

The preconditioned version of the CG algorithm [35] requires the inverse of the preconditioner at each time step. For SAI, since the approximation of the inverse is computed explicitly, the preconditioning process consists in an SpMV operation, which makes the algorithm attractive from a computational point of view due to the parallelizable nature of the SpMV kernel. The SAI preconditioner has been extensively used for solving linear systems coming from different application areas [1, 3, 12, 34]. The parallel nature of the SpMV kernel has also fostered the application of SAI to high-end architectures such as GPUs [8, 20, 33].

When dealing with symmetric and positive definite problems, to preserve the symmetry required by the CG algorithm, the Factorized Sparse Approximate Inverse (FSAI) preconditioner is applied and A^{-1} is approximated by a factorization $G^T G$ instead of a single matrix M, what means that two SpMV products are performed instead of one. G is a sparse lower triangular matrix approximating the inverse of the Cholesky factor, L, of A in terms of the equation below:

$$min_{G \in \mathcal{S}} \|I - GL\|_F^2, \tag{3}$$

where $\|.\|_F$ is the Frobenius norm and $\mathcal{S}$ is a given lower triangular sparse pattern. This minimization problem can be solved independently for each row i of G, by solving the local system:

$$A_{\mathcal{S}_i \mathcal{S}_i} g_i = e_i, \tag{4}$$

where $A_{\mathcal{S}_i \mathcal{S}_i}$ is the restriction of A to the coefficients of the ith row of the sparse pattern $\mathcal{S}$ and e_i is the ith column of the identity matrix restricted to the same space [10, 28].

We apply state-of-the-art techniques [10] to find G without explicitly evaluating L, i. e., only using the initial matrix A. Moreover, the sparse pattern $\mathcal{S}$ is defined a priori as the pattern of a power N of $\tilde{A}$, where $\tilde{A}$ is obtained from A by dropping small entries. The power used to fix the sparse pattern is referred as the sparse level of the preconditioner. In Algorithm 1, we show the method proposed by Chow [10] to find G.

Algorithm 1 FSAI, $G^T G \approx A^{-1}$

1: Threshold A to produce $\tilde{A}$.
2: Compute the pattern $\tilde{A}^N$, and let the pattern of G be the lower triangular part of the pattern of $\tilde{A}^N$.
3: Compute the nonzero entries in G by solving the Frobenius minimization problem.
4: Drop small entries in new G and rescale.

2.3 Factorized Sparse Approximate Inverse with Pattern Extension

Previous work [31] takes into account computer architecture considerations, complementary to numerical ones, to define the sparse pattern where the FSAI inverse approximation is computed. Particularly, cache-friendly pattern extensions are proposed to add new entries to the inverse approximation without incurring any significant

performance penalty. This algorithm is referred as the *Factorized Sparse Approximate Inverse with pattern Extension (FSAIE)*. Numerical experiments demonstrate how FSAIE consistently outperforms FSAI on several high-end architectures. Algorithm 2 represents the FSAIE preconditioner. It shows its modifications with respect to FSAI algorithm in bold letters. Step 3 carries out the pattern extension. Step 4 computes an inverse approximation G_{ext}. Finally, Step 5 drops small entries of G_{ext} and recomputes its nonzero values.

Algorithm 2 FSAIE, $G_{ext}^T G_{ext} \approx A^{-1}$

1: Threshold A to produce $\tilde{A}$.
2: Compute the pattern $\tilde{A}^N$ and let the pattern $\mathcal{S}$ of G be the lower triangular part of the pattern of $\tilde{A}^N$.
3: **Compute cache-friendly extension of the pattern of G, $\mathcal{S}_{ext}$.**
4: **Precalculate an approximation G_{ext} of the preconditioner and filter out entries of $\mathcal{S}_{ext}$ according to its values.**
5: Calculate G_{ext} on the sparse pattern obtained from the previous step.

The FSAIE approach exploits cache locality in the context of multi-core shared-memory architectures [31], but it does not consider the communication cost across several nodes of distributed-memory executions. Obtaining communication-aware sparse pattern extensions able to reduce the number of iterations required for CG to converge requires considering communication costs and keeping load balance when extending the $\mathcal{S}$ sparse pattern (Line 3 of Algorithm 2). Section 3 proposes a new communication-aware sparse pattern extension method that avoids increasing communication costs. Section 4 considers a new filtering strategy to correct inter-process imbalance. Finally, Section 5 evaluates the resulting algorithm on three high-end systems. It considers executions on up to 32,768 CPU cores.

3 FSAIE-COMM: EXTENDING SPARSE PATTERNS WITHOUT INCREASING COMMUNICATION OVERHEAD

This section describes *FSAIE-Comm*, a method to extend the sparse patterns of FSAI preconditioners and make it possible to reduce the iteration count of the CG method without increasing neither internode communication nor intranode memory traffic.

We consider a distributed memory parallelization of CG based on the standard Message Passing Interface (MPI) [16], where the system matrix is distributed into subsets of rows. The distribution is performed applying the METIS [26] partitioner to the adjacency graph of the system matrix. The same distribution is applied to the unknown (x) and right-hand side (b) vectors of the $Ax = b$ system. The unknowns assigned to each MPI process are considered their *local unknowns*. The unknowns of other processes linked to these local unknowns are referred as *halo unknowns*. The matrix entries stored by each MPI process can be divided into the *local entries*, representing couplings between local unknowns, and *halo entries*, representing couplings between local unknowns and halo unknowns. To compute the SpMV kernel in distributed memory scenarios, communications are required between processes to obtain the values of the halo unknowns. This operation is referred as the *halo update*.

Algorithm 3 FSAIE-Comm: Cache-Friendly Fill-In with halo extension

1: $\mathcal{S} \rightarrow$ Local sparse pattern to extend
2: $x \rightarrow$ Local multiplying vector in the SpMV
3: **procedure** ExtendPattern
4: Loop over every row i of pattern $\mathcal{S}$
5: Loop over every entry j in row i
6: If j is in an already considered column block
7: Go to Line 5 and move to next j
8: Endif
9: Identify the cache line of its corresponding x_j coefficient
10: Compute the initial and final columns of the block of entries matching the cache line of x_j
11: Loop over every entry k of the block
12: If k is local $\rightarrow$ Add to pattern $\mathcal{S}_{ext}$
13: Else $\rightarrow$ Add to pattern $\mathcal{S}_{ext}$ if i is sent to the process where j is local
14: Endif
15: Endloop
16: Endloop
17: Endloop
18: Return $\mathcal{S}_{ext}$

Algorithm 3 displays the FSAIE-Comm extension, which is applied in the Step 3 of Algorithm 2. FSAIE-Comm avoids additional cache misses by only considering new entries corresponding to components of the multiplying vector contained in memory blocks already fetched by the original sparse pattern $\mathcal{S}$ of the FSAI preconditioner. Step 10 of Algorithm 3 computes all potential new entries fulfilling this restriction. Local entries of G belonging to a specific process also share the same process in the context of G^T, which means that all additional entries that are local can be added to the pattern extension since they do not incur any additional inter-process communication (Step 12 of Algorithm 3). However, halo entries of G belonging to a specific process may not share the same process in G^T. Therefore, from all cache-friendly entries in the halo we can only consider those that keep the communication scheme unvaried (Step 13 of Algorithm 3). While added halo entries do not produce any additional cache miss when computing the SpMV product involving G, they may incur additional misses in the context of G^T. However, partitions typically minimize the amount of communication and, therefore, reduce the number of halo entries as much as possible. Consequently, FSAIE-Comm extensions of the halo are much smaller than those of local part in terms of the number of additional matrix coefficients.

Figure 1 provides an example of a sparse pattern extension in a distributed memory scenario composed of 2 MPI processes. We focus this example on describing the area where the halo pattern extension can take place, since the halo extension is more complex than the extension of the local part. Rows belonging to the top half of the matrix are owned by one process, and bottom half rows by

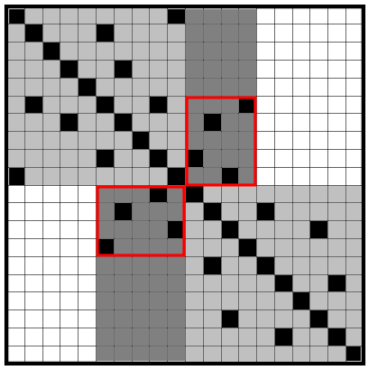


Figure 1: FSAIE-Comm. Graphical explanation of halo region where entries can also be added in a cache-friendly communication-aware extension in a sample 20x20 matrix. Black squares correspond to initial entries.

the other one. The light grey area depicts the two local regions, one per process. These regions represent couplings between local unknowns. The dark grey area represents the halo regions containing couplings between local and halo unknowns. Initial non-zero entries are represented by black squares. FSAIE-Comm exploits the structure of the halo area by adding additional non-zero entries corresponding to columns where there is already a non-zero halo entry, which does not increase communication costs since the corresponding x_i coefficient has to be exchanged when computing the SpMV product Ax with the initial sparse pattern $\mathcal{S}$. For a symmetric and positive-definite matrix $A = GG^T$, the preconditioning step involves two SpMV products with matrices G and G^T. Therefore, the potential halo extensions of FSAIE-Comm for matrix G are halo coefficients belonging to columns where there is already a non-zero halo entry, to avoid increasing communications when computing Gx, and halo coefficients belonging to rows where there is already a non-zero halo entry, to avoid increasing communications when computing $G^T x$. Red rectangles of Figure 1 represent halo regions where adding new entries does not increase communication costs.

4 DYNAMIC FILTERING-OUT

Extending the preconditioning system on each MPI process independently might lead to workload imbalance in the SpMV products by G and G^T. This is undesirable since work imbalance derives in idle processes at synchronization points. A way to overcome this problem could be to transfer some of the workload from the most loaded processes to the least ones. However, this solution is costly, as it requires additional data transfers and could interfere with the cache-friendliness of the sparse pattern extensions.

We propose a dynamic filtering strategy to avoid extension imbalance in distributed memory systems, opposed to the common one, which we will call static. In static filtering, the same *Filter* value is used for all processes and small entries are removed from the inverse approximation according to it in a scale-independent comparison with diagonal entries [10]. In FSAIE, the filtering-out is applied when computing G in the extended pattern, as seen in Step 4 of Algorithm 2. After, the final G_{ext} is obtained over the extended and filtered sparse pattern.

In this novel approach the filtering-out of Step 4 of Algorithm 2 is performed using a dynamic value. This value is adjusted to filter-out additional entries on the highest loaded processes.

Algorithm 4 Dynamic filtering-out for FSAIE and FSAIE-Comm

```
1:  New_filter = Filter
2:  Prev_filter = New_filter
3:  Obtain total S_ext entries using Filter.
4:  Compute process imbalance, imb.
5:  If imb > 1.05
6:    While imb > 1.05 AND imb < 0.95
7:      If imb > 1
8:        Prev_filter = New_filter; New_filter *= 2;
9:      Else
10:       New_filter = (New_filter + Prev_filter) / 2
11:     Endif
12:     Compute process entries using New_Filter
13:     Compute new process imbalance, imb.
14:   Endwhile
15: Endif
16: Return New_Filter
```

Algorithm 4 shows how to compute the dynamic filter in each process. The algorithm requires an input *Filter* value, which serves as a starting point for the computation of the *New_Filter*. It requires each process to compute the amount of entries of its part of $\mathcal{S}_{ext}$ using the initial *Filter* and after, using MPI_Allreduce, to communicate and evaluate the total amount of entries. With these two values we obtain the relative load, *imb*, in each process by dividing its number of entries by the average. If the value is larger than 1 it means the process $\mathcal{S}_{ext}$ has more entries than the average and some entries have to be filtered to avoid imbalance. The *New_Filter* is obtained using a bisection method. Oftentimes the *New_Filter* may require several steps to achieve a desired tolerated imbalance. Setting a maximum amount of iterations leads to *New_Filter* values that overcome the imbalance on the SpMV operation.

5 EVALUATION

This section evaluates the performance of the communication-aware extensions of FSAIE-Comm. It compares FSAIE-Comm with the FSAI algorithm as well as sparsity pattern extensions obtained via applying FSAIE.

5.1 Experimental Setup

We consider some of the largest symmetric and positive definite (SPD) matrices from the SuiteSparse Matrix Collection [13]. We include experiments for SPD matrices with the number of non-zero entries ranging from 1M to 40M. Table 1 lists the complete matrix test set and shows some key matrix properties. Table 2 shows a larger test set used in Section 5.5.1.

We perform experiments on three high-end systems: the MareNostrum supercomputer from the Barcelona Supercomputer Center (BSC), which is composed of nodes with two Skylake 24-core Intel

Table 1: Test matrices along with key properties and results for Skylake. Results are provided as the solving times (in seconds) and iterations-to-convergence for the basic FSAI and for FSAIE and FSAIE-Comm with a 0.01 dynamic *Filter*. For the cases of FSAIE and FSAIE-Comm we also provide the percentage of lower triangular pattern entries increase with respect to FSAI pattern after the extensions (% NNZ).

							FSAI		FSAIE			FSAIE-Comm		
ID	**Matrix**	**#rows**	**NNZ**	**Type**	**#CPU cores**	**#Nodes**	**Solver**	**Iter**	**Solver**	**Iter**	**% NNZ**	**Solver**	**Iter**	**% NNZ**
1	PFlow_742	742793	37138461	2D/3D Problem	1152(1152)	24(9)	1.43e+00	2775	7.67e-01	1458	17.44	7.06e-01	1340	19.3
2	nd24k	72000	28715634	2D/3D Problem	432(512)	9(4)	6.52e-01	553	5.51e-01	490	7.14	5.48e-01	435	14.26
3	Fault_639	638802	27245944	Structural Problem	864(896)	18(7)	1.16e+00	1923	5.71e-01	939	24.5	5.28e-01	856	27.69
4	msdoor	415863	19173163	Structural Problem	576(640)	12(5)	1.74e+00	3599	1.46e+00	2833	42.5	1.39e+00	2748	43.63
5	af_shell7	504855	17579155	Subsequent Structural Problem	1104(1152)	23(9)	5.36e-01	1800	4.87e-01	1541	47.86	4.79e-01	1528	50.2
6	af_shell8	504855	17579155	Subsequent Structural Problem	1104(1152)	23(9)	5.29e-01	1800	4.79e-01	1541	47.86	4.76e-01	1528	50.2
7	af_shell4	504855	17562051	Subsequent Structural Problem	1104(1152)	23(9)	5.18e-01	1800	4.81e-01	1542	47.89	4.68e-01	1530	50.26
8	af_shell3	504855	17562051	Subsequent Structural Problem	1104(1152)	23(9)	5.24e-01	1800	5.22e-01	1542	47.89	4.81e-01	1530	50.26
9	nd12k	36000	14220946	2D/3D Problem	240(256)	5(2)	4.91e-01	516	4.30e-01	452	7.19	3.87e-01	403	14.59
10	crankseg_2	63838	14148858	Structural Problem	240(256)	5(2)	1.77e-01	215	1.44e-01	171	17.65	1.35e-01	160	22.04
11	bmwcra_1	148770	10641602	Structural Problem	336(384)	7(3)	1.09e+00	2325	8.91e-01	1850	36.02	8.85e-01	1800	40.16
12	crankseg_1	52804	10614210	Structural Problem	336(384)	7(3)	1.19e-01	216	9.95e-02	177	14.65	9.11e-02	161	20.05
13	hood	220542	9895422	Structural Problem	624(640)	13(5)	1.11e-01	397	9.14e-02	312	43.07	9.27e-02	315	44.76
14	thermal2	1228045	8580313	Thermal Problem	528(512)	11(4)	1.07e+00	2799	9.41e-01	2117	165.76	9.60e-01	2113	166.53
15	G3_circuit	1585478	7660826	Circuit Simulation Problem	480(512)	10(4)	6.22e-01	1715	5.92e-01	1286	218.45	5.52e-01	1283	219.14
16	nd6k	18000	6897316	2D/3D Problem	96(128)	2(1)	4.79e-01	476	4.19e-01	413	9.84	3.74e-01	364	17.58
17	consph	83334	6010480	2D/3D Problem	192(128)	4(1)	3.13e-01	634	2.95e-01	575	37.99	2.94e-01	562	46.19
18	boneS01	127224	5516602	Model Reduction Problem	192(128)	4(1)	3.62e-01	847	3.51e-01	783	47.78	3.51e-01	779	51.92
19	tmt_sym	726713	5080961	Electromagnetics Problem	336(256)	7(2)	7.76e-01	2319	6.93e-01	1888	193.84	7.08e-01	1883	195.69
20	ecology2	999999	4995991	2D/3D Problem	336(256)	7(2)	9.89e-01	3428	8.44e-01	2510	276.44	8.53e-01	2502	278.05
21	shipsec5	179860	4598604	Structural Problem	288(256)	6(2)	4.73e-01	1618	4.26e-01	1427	25.86	4.29e-01	1424	29.05
22	offshore	259789	4242673	Electromagnetics Problem	144(128)	3(1)	3.96e-01	794	3.36e-01	641	54.06	3.34e-01	635	56.89
23	smt	25710	3749582	Structural Problem	240(256)	5(2)	3.09e-01	882	2.03e-01	551	24.19	1.82e-01	485	31.15
24	parabolic_fem	525825	3674625	Computational Fluid Dynamics Problem	240(256)	5(2)	4.04e-01	1481	3.49e-01	1077	116.57	3.50e-01	1076	116.87
25	Dubcova3	146689	3636643	2D/3D Problem	240(256)	5(2)	3.85e-02	152	3.35e-02	120	97.31	3.28e-02	117	99.67
26	shipsec1	140874	3568176	Structural Problem	240(256)	5(2)	5.92e-01	1987	5.68e-01	1874	27.56	5.70e-01	1878	30.99
27	nd3k	9000	3279690	2D/3D Problem	48(128)	1(1)	3.57e-01	406	3.06e-01	342	11.38	2.84e-01	316	17.55
28	cfd2	123440	3085406	Computational Fluid Dynamics Problem	192(256)	4(2)	6.59e-01	2590	5.22e-01	1847	106.42	5.30e-01	1853	115.1
29	nasasrb	54870	2677324	Structural Problem	144(128)	3(1)	7.15e-01	2765	7.03e-01	2653	15.96	6.98e-01	2629	17.6
30	oilpan	73752	2148558	Structural Problem	144(128)	3(1)	4.04e-01	1554	3.39e-01	1301	20.65	3.37e-01	1285	22.28
31	cfd1	70656	1825580	Computational Fluid Dynamics Problem	48(128)	1(1)	4.01e-01	933	3.81e-01	753	101.18	3.77e-01	750	104.75
32	qa8fm	66127	1660579	Acoustics Problem	48(128)	1(1)	5.35e-03	13	4.68e-03	11	27.33	4.76e-03	11	29.27
33	2cubes_sphere	101492	1647264	Electromagnetics Problem	48(128)	1(1)	6.01e-03	12	5.58e-03	11	12.84	5.59e-03	11	13.37
34	thermomech_dM	204316	1423116	Thermal Problem	96(128)	2(1)	2.92e-03	9	2.98e-03	9	6.09	2.98e-03	9	6.21
35	msc10848	10848	1229776	Structural Problem	48(128)	1(1)	2.51e-01	711	1.86e-01	489	27.11	1.84e-01	482	28.72
36	Dubcova2	65025	1030225	2D/3D Problem	48(128)	1(1)	4.26e-02	155	3.77e-02	113	158.66	3.76e-02	112	160.15
37	gyro_k	17361	1021159	Duplicate Model Reduction Problem	48(128)	1(1)	1.23e+00	4363	9.34e-01	3101	38.46	9.27e-01	3116	39.28
38	gyro	17361	1021159	Model Reduction Problem	48(128)	1(1)	1.25e+00	4382	9.30e-01	3106	38.46	9.26e-01	3071	39.28
39	olafu	16146	1015156	Structural Problem	48(128)	1(1)	4.76e-01	1768	3.65e-01	1330	20.57	3.64e-01	1324	21.45

Table 2: Large test matrices along with key properties and results for Zen 2. Results are provided as the solving times (in seconds) and iterations-to-convergence for the basic FSAI, and for FSAIE and FSAIE-Comm with a 0.01 *Filter*. For the cases of FSAIE and FSAIE-Comm we also provide the percentage of lower triangular pattern entries increase with respect to FSAI pattern after the extensions (% NNZ).

							FSAI		FSAIE			FSAIE-Comm		
ID	**Matrix**	**#rows**	**NNZ**	**Type**	**#CPU cores**	**#Nodes**	**Solver**	**Iter**	**Solver**	**Iter**	**% NNZ**	**Solver**	**Iter**	**% NNZ**
1	Queen_4147	4147110	316548962	2D/3D Problem	32768	256	1.09e+00	5735	9.40e-01	4958	9.38	9.00e-01	4755	13.54
1*	Queen_4147	4147110	316548962	2D/3D Problem	16384	128	1.23e+00	5734	1.07e+00	4997	10.2	1.06e+00	4909	13.85
2	Bump_2911	2911419	127729899	2D/3D Problem	7936	62	4.70e-01	2297	4.50e-01	2206	7.35	4.50e-01	2206	9.14
3	Flan_1565	1564794	114165372	Structural Problem	7168	56	8.70e-01	5299	7.90e-01	4751	14.9	7.70e-01	4578	17.9
4	audikw_1	943695	77651847	Structural Problem	4864	38	2.80e-01	1453	2.40e-01	1212	48.2	2.20e-01	1114	62.56
5	Geo_1438	1437960	60236322	Structural Problem	3712	29	1.30e-01	715	1.20e-01	656	21.26	1.20e-01	654	25.07
6	Hook_1498	1498023	59374451	Structural Problem	3712	29	4.00e-01	2186	4.30e-01	1907	51.41	3.60e-01	1877	58.64
7	bone010	986703	47851783	Model Reduction Problem	2944	23	1.39e+00	7980	1.22e+00	6792	37.93	1.21e+00	6688	46.9
8	ldoor	952203	42493817	Structural Problem	2688	21	1.50e-01	1064	1.40e-01	939	36.37	1.30e-01	860	37.9

Xeon® Platinum 8160 processors at 2.1GHz; the CTE-ARM cluster at BSC composed of nodes with an ARM 48-core A64FX Fujitsu processor at 2.2GHz; and the Hawk supercomputer from the High-Performance Computing Center Stuttgart (HLRS) composed of nodes with two Zen 2 64-core AMD EPYC™ 7742 processors at 2.25GHz. An important hardware parameter for the pattern extensions generated by FSAIE and FSAIE-Comm is the size of the cache line. This parameter is 64Bytes for the Intel and AMD CPUs and 256Bytes for the Fujitsu CPU.

Our code has been written in C language and compiled using GCC 10.1.0 on Skylake, Fujitsu compiler 1.2.26b on A64FX and GCC 9.2.0 on Zen 2. The code is fully hybrid and uses OpenMP 5.0 constructs. We have used the library METIS [26] to perform matrix partitions and MKL library in Skylake and Zen 2 to solve the linear systems to compute the final inverse approximation. For A64FX we have used OPENBLAS 0.3.10. For all the experiments the convergence criterion is the reduction of the initial residual by eight orders of magnitude. The initial guess is always zero. For each matrix a random right-hand side is generated normalized to the matrix max norm. For the time measurements we always consider the minimum time among 50 repetitions.

Our evaluation considers the following methods:

- FSAI - *Factorized Sparse Approximate Inverse preconditioner* [10]. The baseline FSAI preconditioner using the sparse pattern of the lower triangular part of A, without thresholding and filtering only null entries. It is described in Algorithm 1.
- FSAIE - *Factorized Sparse Approximate Inverse preconditioner with Pattern Extension* [31]. A method to extend the sparse pattern in shared-memory scenarios. We apply it to distributed memory systems by extending only the local entries of the input matrix in each process. We consider previously proposed static filtering approaches [31] and the dynamic methods that Section 4 describes. We consider filtering values of 0.01, 0.05, 0.1 and 0.2.
- FSAIE-Comm - *Communication-aware Factorized Sparse Approximate Inverse preconditioner*. The method to extend spare patterns including both the local and the halo entries of the system matrix in distributed memory scenarios without increasing communication cost that we describe in Section 3. We consider previously proposed static filtering approaches [31] and the dynamic methods that Section 4 describes. We consider filtering values of 0.01, 0.05, 0.1 and 0.2.

5.2 Configuration of Parallel Executions

Our data set contains a wide range of matrices featuring different characteristics. Depending on parameters like size, density, and pattern, they will perform differently when increasing the amount of MPI processes used to solve the associated linear system. For this reason, we apply the same criteria to each matrix to select the number of cores to use on its resolution. Considering 8 threads per MPI process, we start with a workload of 256K entries per thread (i.e. 2M per MPI process) and we keep doubling the core count until the parallel efficiency at doubling is smaller than 75%. Using this method, we determine the core and node counts we show in columns 6 and 7 of Table 1, respectively. The numbers in parentheses apply only to Zen 2.

5.3 Evaluation Intel Xeon® Platinum 8160 (Skylake)

This section describes our experimental campaign on the Skylake cluster we describe in Section 5.1. We evaluate the performance of FSAI, FSAIE, and FSAIE-Comm FSAIE-Comm. For each matrix in the set we evaluate several *Filter* values and filtering strategies. Table 1 shows results for the three techniques using the dynamic filtering strategy with an initial *Filter* value of 0.01. Columns 6-7 report number of CPU cores and nodes used for the executions. Columns 8-9 report solver time and iterations-to-convergence for the basic FSAI. Columns 10-12 report solver time, iterations-to-convergence and percentage of added entries to the pattern, respectively, for FSAIE. Finally, columns 13-15 report the same metrics for FSAIE-Comm.

The average results for all matrices in terms of iteration and time-to-solution are shown in Table 3. We compare the FSAIE and FSAIE-Comm with both static and dynamic filtering strategies with respect to FSAI. Table 3 provides average results together with the highest improvement and degradation over all the matrices. We show data for the *Filter* values 0.01, 0.05, 0.1 and 0.5, and for a case where the best *Filter* is picked for each matrix. The best overall configuration is using FSAIE-Comm with dynamic filtering. With respect to FSAI, the average iterations decrease is 20.98%, and the time decrease is 17.98%. In the best case the time-to-solution decrease achieves 55.58%, and in the worst case the time is augmented by only 0.27%. This results shows that FSAIE-Comm can provide performance boosts of up to 2× with a low risk of degrading the performance of FSAI. FSAIE-Comm outperforms FSAIE in average by 2 percentile points.

Note that the dynamic filtering strategy does not have a substantial effect on the average data values as it is a feature that may only be applied to unbalanced matrices. Dynamic filter is further analysed in Section 4.

Finally, in Figure 2 we illustrate the time-to-solution improvements of FSAIE-Comm with dynamic *Filter* with respect to FSAI. In all the experiments the communication cost is unvaried as the same communication scheme is used for all extension methods, being the halo extensions in FSAIE-Comm communication-aware. Most of the matrices show significant improvements and only for one the performance is slightly degraded.

Table 3: Average percentages of iterations-to-solution improvement, time-to-solution improvement, highest time-to-solution improvement and lowest time-to-solution degradation when using FSAIE and FSAIE-Comm with static and dynamic filters. Several *Filter* values are used and best *Filter* is considered for all matrices on a Skylake system.

FSAIE - Static Filter				
Filter value	**Avg. iterations**	**Avg. time**	**Highest imp.**	**Highest deg.**
0.01	20.17	14.66	51.35	-1.96
0.05	16.46	14.93	51.09	-4.46
0.1	12.12	11.38	51.20	-5.56
0.2	7.72	7.69	49.75	-3.99
Best Filter	**18.93**	**16.17**	**51.35**	**0.00**
FSAIE - Dynamic Filter				
Filter value	**Avg. iterations**	**Avg. time**	**Highest imp.**	**Highest deg.**
0.01	19.95	14.8	50.86	-2.33
0.05	15.36	13.92	50.07	-0.34
0.1	9.64	9.03	50.10	-6.85
0.2	5.26	5.25	44.44	-6.62
Best Filter	**18.87**	**16.06**	**50.86**	**-0.24**
FSAIE-Comm - Static Filter				
Filter value	**Avg. iterations**	**Avg. time**	**Highest imp.**	**Highest deg.**
0.01	22.22	16.24	54.75	-1.48
0.05	18.22	16.6	54.97	-0.69
0.1	13.42	12.58	55.09	-0.65
0.2	8.69	8.53	53.23	-6.41
Best Filter	**21.33**	**17.90**	**55.09**	**-0.34**
FSAIE-Comm - Dynamic Filter				
Filter value	**Avg. iterations**	**Avg. time**	**Highest imp.**	**Highest deg.**
0.01	22.04	16.64	54.58	-2.3
0.05	16.83	14.8	55.58	-4.35
0.1	10.68	9.59	53.23	-6.91
0.2	6.0	5.65	44.91	-6.24
Best Filter	**20.98**	**17.98**	**55.58**	**-0.27**

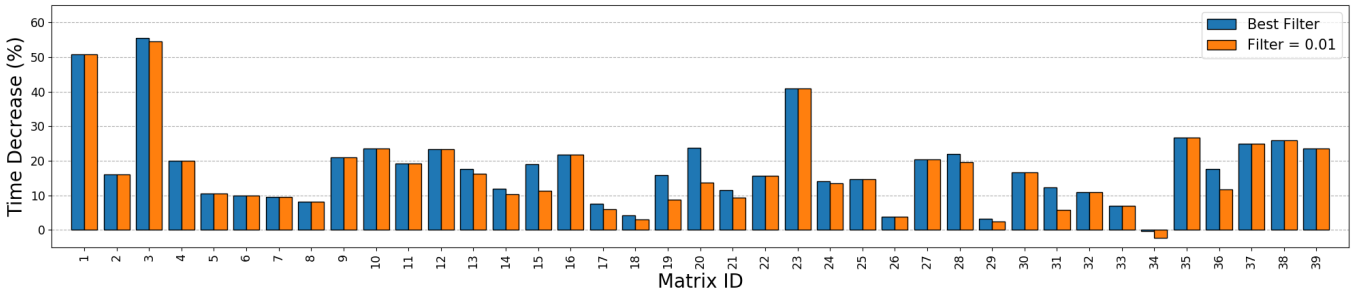
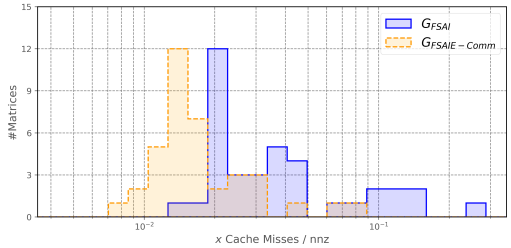
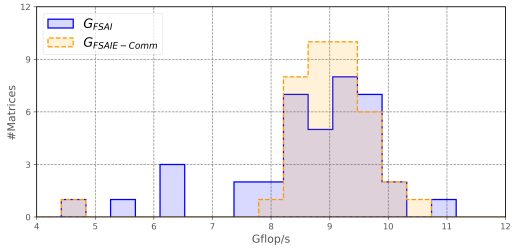


Figure 2: Time decrease of the FSAIE-Comm vs FSAI for the best *Filter* value (blue columns) and for the 0.01 *Filter* value (orange columns) on the Skylake architecture.



(a) L1 DCM of accesses to multiplying vector x in G^TGx normalized to the number of G matrix non-zero entries.



(b) GFLOP/s per process in the preconditioning SpMV operations G^TGx.

Figure 3: Histograms of the preconditioning operation G^TGx in Skylake. Blue columns correspond to baseline FSAI and orange columns to cache-friendly extended matrices using FSAIE-Comm without filtering.

5.3.1 Effects on Data Cache Misses and FLOP/s per process. Cache-aware pattern extensions used in FSAIE and FSAIE-Comm aim to generate new entries on the inverse approximation with a low computational cost. In this section we demonstrate the performance boost obtained in the SpMV product with the communication-aware extensions generated by the FSAIE-comm algorithm when fully extending a pattern. Particularly, we analyze the cache misses and FLOPs metrics.

Figure 3a shows two histograms comparing FSAI (in blue) and FSAIE-Comm (in orange) for the 39 matrices of Table 1 and using 8 OpenMP threads per MPI process. Average L1 data cache misses per process on accesses to vector x when computing the preconditioning operation G^TGx are displayed. We normalize the metric to the number of entries of the G matrices. FSAIE-Comm clearly reduces the cache misses per nonzero element when computing the SpMV operations, which demonstrates that sparse pattern extensions generated by FSAIE-Comm do not incur significant memory overhead. Figure 3b shows a histogram containing the average GFLOP/s per process when performing the preconditioning operation G^TGx for the matrices of Table 1. FSAIE-Comm does not hurt the GFLOP/s rate of FSAI, and slightly improves it for some matrices, which demonstrates it incurs minimal communication and memory overhead. FSAIE-Comm allows to increase FLOPs obtained in the SpMV products by an average 6%.

Table 4: Percentages of average iteration-to-solution decrease, time-to-solution decrease and FLOPs increase in preconditioning SpMV operations when using FSAIE/FSAIE-Comm for different hybrid configurations in Skylake.

CPU/Process	Iter. dec.	Time dec.	FLOPs inc.
1	13.76/19.80	10.59/16.43	-1.55/-2.68
2	16.31/20.91	13.39/17.38	0.05/-0.29
4	17.44/20.88	15.02/18.21	2.23/0.45
8	17.87/20.65	14.56/17.86	8.55/7.30
48	19.54/20.93	17.83/19.29	32.90/33.08

5.3.2 Hybrid configuration. In this section we analyse the influence of the hybrid parallelization set up on the performance of the FSAIE and FSAIE-Comm versus FSAI.

Table 4 shows average results of iterations-to-solution, time-to-solution and SpMV FLOP/s improvement percentages of FSAIE and FSAIE-Comm with respect to FSAI using the best *Filter* with a dynamic strategy. We have considered a workload of 16k non-zero entries per CPU in all cases of the test set. SpMV FLOP/s measurements are obtained without filtering. We perform executions with 1, 2, 4, 8 and 48 CPU threads/cores per MPI process. Increasing the amount of cores per process also increases available L1 cache storage for the process.

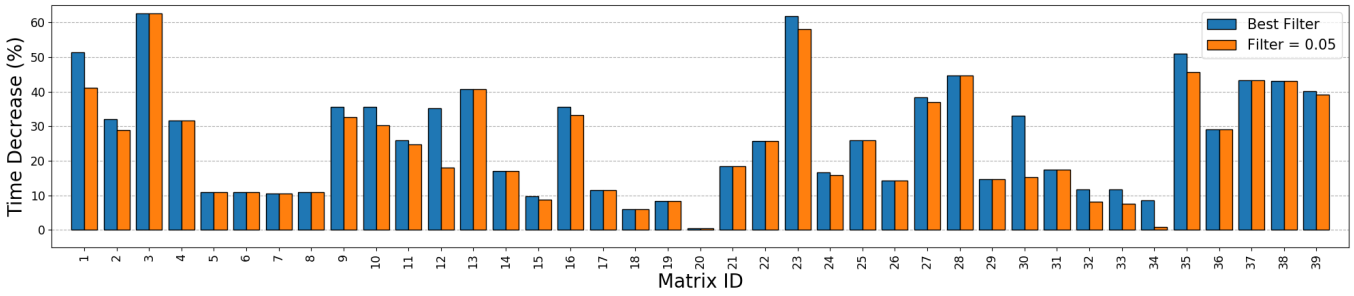
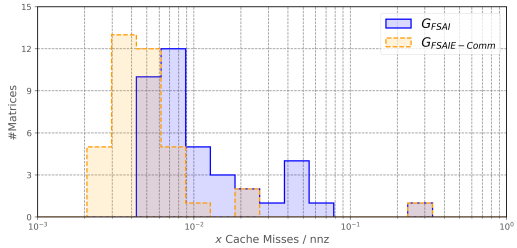
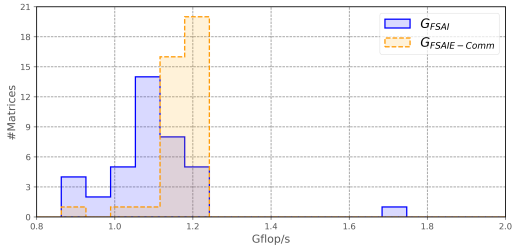


Figure 4: Time decrease of the FSAIE-Comm vs FSAI for the best *Filter* value (blue columns) and for the 0.05 *Filter* value (orange columns) on the A64FX architecture.



(a) L1 DCM of accesses to multiplying vector x in G^TGx normalized to the number of G matrix non-zero entries.



(b) GFLOP/s per process in the preconditioning SpMV operations G^TGx.

Figure 5: Histograms of the preconditioning operation G^TGx in A64FX. Blue columns correspond to baseline FSAI and orange columns to cache-friendly extended matrices using FSAIE-Comm without filtering.

Cache-aware extensions take advantage from larger L1 cache size, for this reason the configuration of 48 threads per MPI process is the one where FSAIE and FSAIE-Comm outperform more FSAI, being the gain obtained with FSAIE-Comm 2 percentile points higher. On the other extreme, using one thread per process maximizes the number of matrix halo entries and makes it possible for FSAIE-Comm to deliver much larger average performance improvements (16.43%) than FSAIE (10.59%) with respect to FSAI. Different problems may require different hybrid parallelization configurations to optimize their solving time. Our experiments indicate that FSAIE-Comm outperforms FSAIE for all of them.

5.3.3 Impact of Dynamic Filtering-out in Load Balance. The dynamic filtering-out strategy eliminates unbalanced matrix extensions , i. e., it avoids some processes being overloaded in terms of nonzero matrix entries assigned to them. Whether the FSAIE-Comm matrix extensions incur load imbalance issues or not depends on the sparse pattern of the input matrix. While load imbalance is not an issue when extending a large portion of the matrices we display in Table 1, it becomes a very important problem for several of them. In this section we show in detail a case of a matrix in the set that benefits from dynamic filtering.

We compute the imbalance index as the ratio between average process entries and maximum amount of entries in a process. A value equal to 1 means all processes are balanced, while a value lower than 1 means that there is at least one process that displays an amount of entries larger than the average. Focusing on matrix 17 of Table 1, we have a case where the partition of matrix A is not well balanced. The imbalance index for the partitions of G and G^T is 0.88, and applying the extension of FSAIE-Comm with $Filter = 0.01$ drops the average imbalance index of their partitions to 0.75. In this case, when comparing FSAIE-Comm with FSAI, iterations-to-solution are reduced by 11.83%, and time-to-solution only by **2%**. Applying a dynamic *Filter* improves the imbalance index of the factors partition to 0.82, what makes the decrease in time-to-solution rise to **6%**. In summary, the dynamic filtering is an auto-tuning capability added to the algorithm which increases its robustness with respect to imbalance issues.

5.4 Evaluation on Fujitsu A64FX (A64FX)

We run experiments considering the CTE-ARM cluster composed of nodes with an ARM 48-core Fujitsu A64FX CPU. In this case, the CPUs have 256Bytes cache lines, which allows larger cache-aware extensions.

The average results for all matrices in terms of iteration and time-to-solution are shown in Table 5. We compare the FSAIE-Comm with dynamic filtering with respect to FSAI. The table provides average results together with the highest improvement and degradation over all the matrices. We show results considering the dynamic filtering-out strategy with initial *Filter* values 0.01, 0.05, 0.1 and 0.5, and a case where the best *Filter* is picked for each matrix.

Table 5: Average percentages of iterations-to-solution improvement, time-to-solution improvement, highest time-to-solution improvement and lowest time-to-solution degradation when using FSAIE-Comm with dynamic filters. Several *Filter* values are used and best *Filter* is considered for all matrices on the A64FX system.

FSAIE-Comm - Dynamic Filter				
Filter value	**Avg. iterations**	**Avg. time**	**Highest imp.**	**Highest deg.**
0.01	34.39	22.69	61.75	-20.81
0.05	27.72	24.16	62.63	0.49
0.1	18.98	17.26	59.51	-1.5
0.2	10.51	9.94	55.68	-7.32
Best Filter	**31.32**	**26.44**	**62.63**	**0.49**

The larger cache lines of the A64FX CPU allow to achieve large time-to-solution improvements than the Skylake scenario due to the larger iteration decrease brought by extra added entries. On average, FSAIE-Comm reduces iterations-to-convergence by 31.32% and time-to-solution by 26.44%.

Figure 4 displays the time-to-solution improvement for each matrix of the data set when using the best *Filter* and when using a *Filter* value of 0.05. In general the performance boost achieved is notably higher for most matrices compared to Intel Skylake. Figures 5a and 5b show the effects on the SpMV operations of FSAIE-Comm for A64FX. Cache misses on accesses to x are reduced and FLOP/s, consequently, improved. In this case, FLOP/s increase on average by 7.5%.

5.5 Evaluation on AMD EPYC™ 7742 (Zen 2)

We run numerical experiments considering the Hawk supercomputer, which is composed of nodes with two Zen 2 64-core AMD EPYC 7742 processors. In this case, the CPUs have 64Bytes cache lines. The core counts we consider in this architecture are specified in Table 1 in parentheses. The average results for all matrices are shown in Table 6. The results for the Zen 2 case lead to an average iteration-to-solution decrease of 20.64% and time-to-solution decrease of 16.74%. These values are close to Skylake results since both systems feature the same cache line size.

Figure 6 displays the time-to-solution improvement for each matrix of the data set when using the best *Filter* and when using a *Filter* value of 0.05. Figure 7 shows FLOP/s improvement in the preconditioning SpMV operations. Note that in this architecture the FLOP/s achieved with FSAI and FSAIE-Comm are much higher than for Skylake and A64FX. On average, FSAIE-Comm achieves FLOPs improvements of 19% on AMD.

5.5.1 Large-scale experiments. We run large-scale experiments on the Zen 2 system. We consider the largest matrices in the SuiteSparse collection [13] for these experiments. These matrices along with some of its key characteristics are shown in Table 2.

For all matrices we consider a workload of 16k non-zero entries per CPU. For the case of matrix 1 we consider node counts of 128 and 256 due to the cluster topology requirements and provide results for both cases. We perform executions up to 32768 CPUs.

Table 6: Average percentages of iterations-to-solution improvement, time-to-solution improvement, highest time-to-solution improvement and lowest time-to-solution degradation when using FSAIE-Comm with dynamic filters. Several *Filter* values are used and best *Filter* is considered for all matrices on the Zen 2 system.

FSAIE-Comm - Dynamic Filter				
Filter value	**Avg. iterations**	**Avg. time**	**Highest imp.**	**Highest deg.**
0.01	22.18	14.36	56.75	-13.82
0.05	16.73	14.28	57.52	-1.13
0.1	10.07	8.73	52.09	-1.85
0.2	5.43	5.09	40.92	-7.27
Best Filter	**20.64**	**16.74**	**57.52**	**-1.05**

Table 7: Average iterations-to-solution improvement percentage, average time-to-solution improvement percentage, highest time-to-solution improvement percentage and lowest time-to-solution degradation when FSAIE-Comm dynamic filter. Several *Filter* values are used and best *Filter* is considered for large matrices on a Zen 2 system.

FSAIE-Comm - Dynamic Filter				
Filter value	**Avg. iterations**	**Avg. time**	**Highest imp.**	**Highest deg.**
0.01	14.49	12.28	19.09	3.64
0.05	10.82	10.83	16.93	3.94
0.1	7.04	7.38	13.43	1.82
0.2	3.55	4.03	9.77	-3.63
Best Filter	**13.89**	**12.59**	**19.09**	**3.94**

Table 2 shows FSAIE-Comm always improves at least the result of FSAIE, outperforming it on average by 3 percentile points.

Figure 8 displays the time-to-solution improvement for each matrix when using the best *Filter* and when using a *Filter* value of 0.01. The average results for these cases can be seen in Table 7 and lead to an average iteration-to-solution decrease of 13.89% and time-to-solution decrease of 12.59% when using the best *Filter* value per matrix. Note for this case the *BestFilter* results are close to the $Filter = 0.01$ scenario and that no matrix degrades its performance except the $Filter = 0.2$ case. For the large data set FSAIE-Comm achieves average FLOP/s improvements of 22% when running the preconditioning operation $G^T Gx$.

6 RELATED WORK

The Sparse Approximate Inverse (SAI) preconditioner is commonly used due to its parallel nature and easy applicability. Research efforts on SAI focus on optimizing the existing variants and finding new alternatives to improve solvers. SAI methods require to evaluate an approximation to the inverse of a matrix on a pattern. When these patterns are defined a priori they are called static patterns. When they are defined at the same time as the approximation to the inverse is being computed they are called dynamic patterns [14].

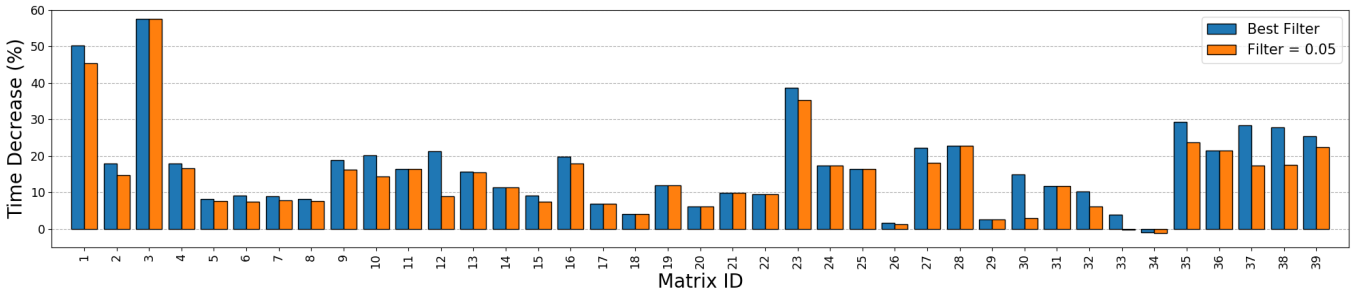


Figure 6: Time decrease of the FSAIE-Comm vs FSAI for the best *Filter* value (blue columns) and for the 0.05 *Filter* value (orange columns) on the Zen 2 architecture.

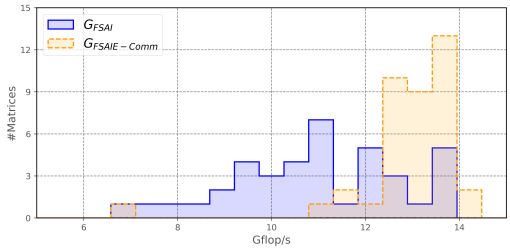


Figure 7: Histogram of the GFLOP/s per process the preconditioning operation $G^T Gx$ normalized to the number of G matrix non-zero entries. Blue columns correspond to baseline FSAI and orange columns to cache-friendly extended matrices using FSAIE-Comm without filtering on Zen 2.

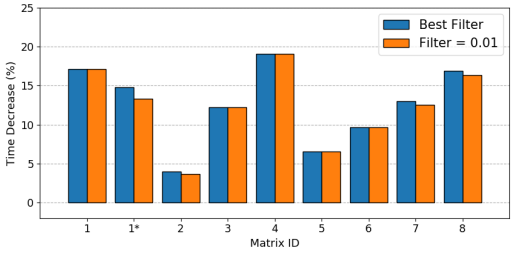


Figure 8: Time decrease of the FSAIE-Comm vs FSAI for the best *Filter* value (blue columns) and for the 0.01 *Filter* value (orange columns) on the Zen 2 architecture.

In this paper we consider static patterns. We chose as initial pattern the lower triangular part of the system matrix A to compute the factor G for FSAI. Powers of A, usually A^2 or A^3 [9, 17, 21] are commonly used to define static patterns. Dense patterns may be sparsified using thresholding and filtering techniques [4, 5, 10, 15, 27, 29].

Dynamic patterns are created through adaptive procedures that compute the approximation to the inverse with iterative methods. Usually, they begin with an initial pattern that is grown following some strategy until some criteria is fulfilled. At the same time the inverse approximation is computed. Dynamic methods are usually more powerful than static ones, however, they are difficult to parallelize and implement efficiently, and usually are computationally costlier. SPAI [18] is the direct dynamic counterpart to SAI and FSPAI [22] to FSAI. Other methods such as BSAI, PSAI and RSAI [23–25] have been developed recently. SAI preconditioners for GPUs have been proposed for both static [2, 6, 30, 36] and dynamic strategies [7].

Preconditioners are usually designed based on numerical criteria. Some methods propose preconditioner storage formats with reordering to achieve better locality and improve cache hit ratios [32]. Previous work [31] paved the way to develop preconditioner extensions with cache-awareness, taking into account architectural criteria to extend any pattern and improve its efficiency. FSAIE-Comm brings a communication-aware approach to distributed memory systems that improves any method by extending it in a cache-friendly manner that does not increase communication costs.

7 CONCLUSIONS

This paper demonstrates FSAIE-Comm is an efficient preconditioning method for distributed memory systems that outperforms FSAIE in most cases. Essentially, cache-friendly pattern extensions can be applied to every process in an MPI implementation to achieve iteration count reductions with low overhead that are translated into time-to-solution improvements. Our new method for distributed memory systems also takes advantage of halo entries and extends patterns without increasing communication costs. We also propose a new dynamic filtering strategy that mitigates inter-process imbalance when extending patterns. For our evaluation we have considered three different high-end systems: a Skylake system with 64Byte cache lines, an A64FX system with 256Byte cache lines and a Zen 2 system with 64Byte cache lines. Overall, on a 39 matrices data set with a wide range of non-zero entries we achieve 17.98%, 26.44% and 16.74% average time-to-solution improvements on Skylake, A64FX and Zen 2, respectively. For all the systems there are matrices that achieve time-to-solution improvements above 50%. We have also tested FSAIE-Comm on a smaller set of the largest available matrices on Zen 2 with executions of up to 32768 CPUs achieving an average time-to-solution improvement of 12.59%.

ACKNOWLEDGMENTS

Marc Casas is supported by Grant RYC-2017-23269 funded by MCIN/AEI/10.13039/501100011033 and by "ESF Investing in your future". This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 955606. This work has been supported by the Computación de Altas Prestaciones VIII (BSC-HPC8) project. It has also been partially supported by the EXCELLERAT project funded by the European Commission's ICT activity of the H2020 Programme under grant agreement number: 823691 and by the Spanish Ministry of Science and Innovation (Nucleate, Project PID2020-117001GB-I00).

REFERENCES

[1] Guillaume Alléon, Michele Benzi, and Luc Giraud. 1997. Sparse approximate inverse preconditioning for dense linear systems arising in computational electromagnetics. *Numerical Algorithms* 16 (02 1997), 1–15. https://doi.org/10.1023/A:1019170609950

[2] Hartwig Anzt, Edmond Chow, Thomas Huckle, and Jack Dongarra. 2016. Batched Generation of Incomplete Sparse Approximate Inverses on GPUs. 49–56. https://doi.org/10.1109/ScalA.2016.011

[3] Michele Benzi, Carl D. Meyer, and Miroslav Tůma. 1996. A Sparse Approximate Inverse Preconditioner for the Conjugate Gradient Method. *SIAM Journal on Scientific Computing* 17, 5 (1996), 1135–1149. https://doi.org/10.1137/S1064827594271421 arXiv:https://doi.org/10.1137/S1064827594271421

[4] Luca Bergamaschi, Giuseppe Gambolati, and Giorgio Pini. 2007. A numerical experimental study of inverse preconditioning for the parallel iterative solution to 3D finite element flow equations. *J. Comput. Appl. Math.* 210 (12 2007), 64–70. https://doi.org/10.1016/j.cam.2006.10.056

[5] Luca Bergamaschi, Ángeles Martínez, and Giorgio Pini. 2006. Parallel preconditioned conjugate gradient optimization of the Rayleigh quotient for the solution of sparse eigenproblems. *Appl. Math. Comput.* 175, 2 (2006), 1694 – 1715. https://doi.org/10.1016/j.amc.2005.09.015

[6] Massimo Bernaschi, Mauro Bisson, Carlo Fantozzi, and Carlo Janna. 2016. A Factored Sparse Approximate Inverse Preconditioned Conjugate Gradient Solver on Graphics Processing Units. *SIAM Journal on Scientific Computing* 38, 1 (2016), C53–C72. https://doi.org/10.1137/15M1027826 arXiv:https://doi.org/10.1137/15M1027826

[7] Massimo Bernaschi, Mauro Carrozzo, Andrea Franceschini, and Carlo Janna. 2019. A Dynamic Pattern Factored Sparse Approximate Inverse Preconditioner on Graphics Processing Units. *SIAM Journal on Scientific Computing* 41, 3 (2019), C139–C160. https://doi.org/10.1137/18M1197461 arXiv:https://doi.org/10.1137/18M1197461

[8] Daniele Bertaccini and Salvatore Filippone. 2016. Sparse approximate inverse preconditioners on high performance GPU platforms. *Computers & Mathematics with Applications* 71, 3 (2016), 693 – 711. https://doi.org/10.1016/j.camwa.2015.12.008

[9] Edmond Chow. 2000. A Priori Sparsity Patterns for Parallel Sparse Approximate Inverse Preconditioners. *SIAM Journal on Scientific Computing* 21, 5 (2000), 1804–1822. https://doi.org/10.1137/S106482759833913X arXiv:https://doi.org/10.1137/S106482759833913X

[10] Edmond Chow. 2001. Parallel Implementation and Practical Use of Sparse Approximate Inverse Preconditioners With a Priori Sparsity Patterns. *International Journal of High Performance Computing Applications* 15 (05 2001). https://doi.org/10.1177/109434200101500106

[11] Edmond Chow and Yousef Saad. 1998. Approximate Inverse Preconditioners via Sparse-Sparse Iterations. *SIAM Journal on Scientific Computing* 19, 3 (1998), 995–1023.

[12] Mark D. Kremenetsky, John Richardson, and Horst D. Simon. 1995. - Parallel preconditioning for CFD problems on the CM-5. In *Parallel Computational Fluid Dynamics 1993*, A. Ecer, J. Hauser, P. Leca, and J. Periaux (Eds.). North-Holland, Amsterdam, 401–410. https://doi.org/10.1016/B978-044481999-4/50173-0

[13] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[14] Massimiliano Ferronato. 2012. Preconditioning for Sparse Linear Systems at the Dawn of the 21st Century: History, Current Developments, and Future Perspectives. *ISRN Applied Mathematics* 2012 (12 2012). https://doi.org/10.5402/2012/127647

[15] Massimiliano Ferronato, Carlo Janna, and Giorgio Pini. 2012. Shifted FSAI preconditioners for the efficient parallel solution of non-linear groundwater flow models. *Internat. J. Numer. Methods Engrg.* 89 (03 2012), 1707–1719. https://doi.org/10.1002/nme.3309

[16] Message P Forum. 1994. *MPI: A Message-Passing Interface Standard.* Technical Report. USA.

[17] John R. Gilbert. 1994. Predicting Structure in Sparse Matrix Computations. *SIAM J. Matrix Anal. Appl.* 15, 1 (1994), 62–79. https://doi.org/10.1137/S0895479887139455 arXiv:https://doi.org/10.1137/S0895479887139455

[18] Marcus J. Grote and Thomas Huckle. 1997. Parallel Preconditioning with Sparse Approximate Inverses. *SIAM Journal on Scientific Computing* 18, 3 (1997), 838–853. https://doi.org/10.1137/S1064827594276552

[19] Wolfgang Hackbusch. 1985. *Multi-Grid Methods and Applications.* Vol. 4. https://doi.org/10.1007/978-3-662-02427-0

[20] Guixia He, Renjie Yin, and Jiaquan Gao. 2019. An efficient sparse approximate inverse preconditioning algorithm on GPU. *Concurrency and Computation: Practice and Experience* 32 (12 2019). https://doi.org/10.1002/cpe.5598

[21] Thomas Huckle. 1999. Approximate sparsity patterns for the inverse of a matrix and preconditioning. *Applied Numerical Mathematics* 30, 2 (1999), 291 – 303. https://doi.org/10.1016/S0168-9274(98)00117-2

[22] Thomas Huckle. 2003. Factorized Sparse Approximate Inverses for Preconditioning. *Journal of Supercomputing* 25, 2 (2003), 109–117.

[23] Carlo Janna and Massimiliano Ferronato. 2011. Adaptive Pattern Research for Block FSAI Preconditioning. *SIAM Journal on Scientific Computing* 33, 6 (2011), 3357–3380. https://doi.org/10.1137/100810368 arXiv:https://doi.org/10.1137/100810368

[24] Zhongxiao Jia and Wenjie Kang. 2017. A residual based sparse approximate inverse preconditioning procedure for large sparse linear systems. *Numerical Linear Algebra with Applications* 24, 2 (2017), e2080. https://doi.org/10.1002/nla.2080 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/nla.2080 e2080 nla.2080.

[25] Zhongxiao Jia and Baochen Zhu. 2009. A Power Sparse Approximate Inverse Preconditioning Procedure for Large Sparse Linear Systems. *Numerical Linear Algebra with Applications* 16 (04 2009), 259 – 299. https://doi.org/10.1002/nla.614

[26] George Karypis and Vipin Kumar. 1995. METIS – Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0. (01 1995).

[27] Liliya Yu. Kolotilina, Andy A. Nikishin, and Alex Yu. Yeremin. 1999. Factorized Sparse Approximate Inverse Preconditionings. IV: Simple Approaches to Rising Efficiency. *Numerical Linear Algebra With Applications - NUMER LINEAR ALGEBR APPL* 6 (10 1999), 515–531. https://doi.org/10.1002/(SICI)1099-1506(199910/11)6:73.0.CO;2-0

[28] Liliya Yu. Kolotilina and Alex Yu. Yeremin. 1993. Factorized Sparse Approximate Inverse Preconditionings I. Theory. *SIAM J. Matrix Anal. Appl.* 14, 1 (1993), 45–58. https://doi.org/10.1137/0614004 arXiv:https://doi.org/10.1137/0614004

[29] Jiří Kopal, Miroslav Rozložník, and Miroslav Tůma. 2015. Approximate inverse preconditioners with adaptive dropping. *Advances in Engineering Software* 84 (2015), 13 – 20. https://doi.org/10.1016/j.advengsoft.2015.01.006 CIVIL-COMP.

[30] I.B. Labutin and I.V. Surodina. 2013. Algorithm for sparse approximate inverse preconditioners in the conjugate gradient method. 19 (01 2013), 120–126.

[31] Sergi Laut, Ricard Borrell, and Marc Casas. 2020. Cache-Aware Sparse Patterns for the Factorized Sparse Approximate Inverse Preconditioner. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing* (Virtual Event, Sweden) *(HPDC '21)*. Association for Computing Machinery, New York, NY, USA, 81–93. https://doi.org/10.1145/3431379.3460642

[32] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. 2014. Cache-aware sparse matrix formats for Kepler GPU. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. 281–288. https://doi.org/10.1109/PADSW.2014.7097819

[33] G. Oyarzun, R. Borrell, A. Gorobets, and A. Oliva. 2014. MPI-CUDA sparse matrix–vector multiplication for the conjugate gradient method with an approximate inverse preconditioner. *Computers & Fluids* 92 (2014), 244 – 252. https://doi.org/10.1016/j.compfluid.2013.10.035

[34] Yousef Saad. 2002. Preconditioned Krylov Subspace Methods for CFD Applications. *Proceedings of the international workshop on solution techniques for large-scale CFD problems* (02 2002).

[35] Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems* (2nd ed.). Society for Industrial and Applied Mathematics, USA.

[36] K. Xu, D.Z. Ding, Z.H. Fan, and R.S. Chen. 2011. FSAI preconditioned CG algorithm combined with GPU technique for the finite element analysis of electromagnetic scattering problems. *Finite Elements in Analysis and Design* 47, 4 (2011), 387 – 393. https://doi.org/10.1016/j.finel.2010.11.005