A scalable adaptive-matrix SpMV for heterogeneous architectures

Han D. Tran
University of Utah, USA
hantran@cs.utah.edu

Milinda Fernando
University of Texas at Austin, USA
milinda@oden.utexas.edu

Kumar Saurabh
Iowa State University, USA
maksbh@iastate.edu

Baskar Ganapathysubramanian
Iowa State University, USA
baskarg@iastate.edu

Robert M. Kirby
University of Utah, USA
kirby@cs.utah.edu

Hari Sundar
University of Utah, USA
hari@cs.utah.edu

2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-8106-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/IPDPS53621.2022.00011

***Abstract*—**
In most computational codes, the core computational kernel is the Sparse Matrix-Vector product (SpMV) that enables specialized linear algebra libraries like PETSc to be used, especially in the distributed memory setting. However, optimizing SpMV performance and scalability at all levels of a modern heterogeneous architecture can be challenging as it is characterized by irregular memory access. This work presents a hybrid approach (HyMV) for evaluating SpMV for matrices arising from PDE discretization schemes such as the finite element method (FEM). The approach enables localized structured memory access that provides improved performance and scalability. Additionally, it simplifies the programmability and portability on different architectures. The developed HyMV approach enables efficient parallelization using MPI, SIMD, OpenMP, and CUDA with minimum programming effort. We present a detailed comparison of HyMV with the two traditional approaches in computational code, matrix-assembled and matrix-free approaches, for structured and unstructured meshes. Our results demonstrate that the HyMV approach achieves excellent scalability and outperforms both approaches, e.g., achieving average speedups of 11x for matrix setup, 1.7x for SpMV with structured meshes, 3.6x for SpMV with unstructured meshes, and 7.5x for GPU SpMV.

***Index Terms*—adaptive-matrix, matrix-assembled, matrix-free, element-by-element, FEM, parallel computing, heterogeneous architectures**

I. Introduction

For domain-based numerical methods, such as the finite element method (FEM), the approximation of the weak form results in a system of discretized equations $\boldsymbol{Ku} = \boldsymbol{f}$ of the unknown $\boldsymbol{u}$. A traditional matrix-assembled approach is often used to solve this system, in which elemental matrices are assembled to form the global matrix $\boldsymbol{K}$, and a sparse matrix/linear algebra library (e.g., PETSc [1]) is employed. An alternative is to use the so-called matrix-free approach where the global matrix $\boldsymbol{K}$ is not explicitly assembled but is computed implicitly during the matrix-vector multiplication when solved by an iterative method. The matrix-assembled approach is attractive from being efficient (less computation). However, it has higher storage and communication overhead, and irregular memory access. Additionally, optimizing sparse linear algebra is complex, especially for modern heterogeneous architectures, and is an area of active research [2]. The matrix-free approach is attractive for large-scale distributed computing, as we can limit communication (in the weak-scaling sense). Memory footprint is also much lower as the matrix is not stored and, more importantly, can be made structured. However, it can be expensive when the elemental assemblies are complex. The computational cost is compounded when multiple matrix-vector products are required (e.g., implicit solve). In most cases, optimizing the elemental assembly for modern architectures can also be challenging, similar to the Sparse Matrix-Vector product (SpMV).

For these reasons, we develop an SpMV that uses a 'hybrid' approach called HyMV to overcome the high costs associated with the above methods. The theoretical background of our approach is the element-by-element (EbE) technique, a well-known method in FEM (e.g., [3]–[5]), in which the global operation $\boldsymbol{Kv}$ (for an intermediate vector $\boldsymbol{v}$ during the iterative solving) is performed via the sum of elemental matrix-vector products (eMV). Within HyMV, we store the element matrices provided by users and perform the eMV so that the global (and therefore distributed memory) computations are handled in a matrix-free fashion. Such an approach ensures lower computations and communication, while storage (memory footprint) can still be high. While node-local storage and computations are higher than the matrix-assembled approach, it is essential to note that these involve structured memory accesses (dense eMV) instead of irregular access for SpMV. Additionally, it is far easier to write efficient vectorized linear algebra kernels within HyMV. Furthermore, adding GPU support in HyMV is straightforward, while it can be challenging for both matrix-assembled and matrix-free approaches. HyMV is particularly effective for problems requiring frequent refinements or enrichments, e.g., crack modeling using the extended finite element method (XFEM). In XFEM, when a crack occurs, additional unknowns are enriched in the cracked element. This enrichment changes the stiffness matrix of few (cracked) elements while most (uncracked) elements are intact. HyMV handles this issue efficiently since only the cracked elements are recomputed (in contrast, if a matrix-assembled approach is used, the entire

1530-2075/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPS53621.2022.00011

global matrix must be reassembled). The primary purpose of this work is to develop an SPMV operation combining the advantages of matrix-assembled and matrix-free methods for efficiently solving large-scale sparse systems on modern heterogeneous architectures.

The main contributions of this work include:

- HYMV converts global sparse linear algebra to local dense linear algebra, reducing the data-movement cost and increasing the throughput when solving the discretized system $\boldsymbol{Ku} = \boldsymbol{f}$ on modern heterogeneous architectures. It is particularly beneficial for problems that require frequent refinements or enrichments.
- HYMV demonstrates scalability for many different real-life problems.
- In HYMV, it is far easier to implement efficient vectorized linear algebra kernels being optimized for various architectures, and it is also straightforward to add GPU support.
- HYMV is a standalone library that is easily incorporated into existing codes of any domain-based numerical method. HYMV is an open source written in C++, OpenMP, MPI, and supports GPUs. It also supports block preconditioners.

Organization: Section II summarizes the preliminaries of our development, including related works. The underlying ideas of HYMV are in section III. Section IV presents the HYMV's key components for solving the discretized system. Section V presents the numerical experiments conducted using our developed HYMV for the problems of various computational complexities, illustrating the superiority of HYMV compared with the matrix-assembled and matrix-free approaches. The conclusion is in section VI.

II. PRELIMINARIES

In this section, we briefly explain the motivation for our development and the past works that relate to the problem. We take a simple example of solving the Poisson's equation by FEM for the mere purpose of illustration. The weak form of a Poisson's equation defined on the domain Ω for the field variable $u(\boldsymbol{x})$, subjected to body force $\bar{b}(\boldsymbol{x})$, boundary conditions $u(\boldsymbol{x}) = \bar{u}(\boldsymbol{x})$ for $\boldsymbol{x} \in \partial\Omega_{\mathrm{u}}$ and $\partial u(\boldsymbol{x})/\partial n = \bar{t}(\boldsymbol{x})$ for $\boldsymbol{x} \in \partial\Omega_t$ (where $\boldsymbol{n}$ is the unit vector normal to $\partial\Omega_t$), is written as

$$\int_\Omega (\nabla u \cdot \nabla v)\, dV = \int_{\partial\Omega_t} \bar{t} v dA + \int_\Omega \bar{b} v dV \tag{1}$$

where $v(\boldsymbol{x})$ is an arbitrary test function such that $v(\boldsymbol{x}) = 0$ for $\boldsymbol{x} \in \partial\Omega_{\mathrm{u}}$. Invoking Galerkin approximation $u(\boldsymbol{x}) = \sum_{i=1}^{N} u_i \phi_i(\boldsymbol{x})$ and $v(\boldsymbol{x}) = \sum_{j=1}^{N} v_j \phi_j(\boldsymbol{x})$, and arbitrariness of test function $v(\boldsymbol{x})$, equation (1) is re-written as, for $j = 1, 2, ..., N$,

$$\sum_{i=1}^{N} \int_\Omega (\nabla\phi_i \cdot \nabla\phi_j)\, dV u_i = \int_{\partial\Omega_t} \bar{t}\phi_j dA + \int_\Omega \bar{b}\phi_j dV, \quad \text{or} \quad \boldsymbol{Ku} = \boldsymbol{f} \tag{2}$$

where the components of the global matrix $\boldsymbol{K}$ are defined as $K_{ij} = \int_\Omega (\nabla\phi_i \cdot \nabla\phi_j)\, dV$, and the right-hand-side vector $f_j = \int_{\partial\Omega_t} \bar{t}\phi_j dA + \int_\Omega \bar{b}\phi_j dV$. In equation (2), $\phi_i(\boldsymbol{x})$ is the basis function associated with the grid point $\boldsymbol{x}_i$ (hereafter we call 'node' which is commonly used in the FEM community), $u_i = u(\boldsymbol{x}_i)$, and N is the total number of nodes. Following the decomposition of domain Ω into N_e subdomains (i.e. elements), the global matrix $\boldsymbol{K}$ is constructed from the contribution of N_e element matrices $\boldsymbol{K}_e$, i.e.,

$$K_{ij} = \sum_{e=1}^{N_e} \int_e (\nabla\phi_i \cdot \nabla\phi_j)\, dV = \sum_{e=1}^{N_e} (K_e)_{ij}\,, \tag{3}$$

where $(K_e)_{ij} = \int_e (\nabla\phi_i \cdot \nabla\phi_j)\, dV$ is the component of element matrix associated with element e. The basis function ϕ_i has compact support, i.e., $K_{ij} \neq 0$ only when both nodes $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are on the same element. For this reason, the global matrix $\boldsymbol{K}$ is sparse.

Matrix-assembled approach: The system (2) is often solved using a *matrix-assembled approach*, i.e., the global matrix $\boldsymbol{K}$ is fully assembled, as shown in (3), before invoking a solver to obtain the solution of $\boldsymbol{u}$. In distributed memory, the domain Ω is partitioned into p partitions, i.e., $\Omega = \{\omega_i\}_{i=1}^{p}$. Each partition ω_i is discretized into a finite set of elements, i.e., $\omega_i = \{e_m^i\}_{m=1}^{|\omega_i|}$. Each partition needs to loop over its owned elements during the global assembly and accumulate to the global matrix using the element connectivity (i.e., mapping from local-element indices to global indices of the nodes). Even though the above communication is peer-to-peer, for increasing p values, the global assembly becomes expensive. This is primarily due to irregular memory access that is difficult and expensive for an SPMV to optimize for.

Matrix-free approach: An alternative for solving the system (2) is to use a *matrix-free approach*, in which the global SPMV (i.e., $\boldsymbol{Kv}$ where $\boldsymbol{v}$ is an intermediate vector during the iterative solving) is performed without explicitly forming the global matrix $\boldsymbol{K}$. In this approach, the EMV (i.e., $\boldsymbol{K}_e\boldsymbol{v}_e$) is computed in every iteration. The above requires either explicit or implicit computation of element matrices $\boldsymbol{K}_e$ in every iteration. Algorithm 4, shown in the appendix, is an example of using explicit calculation of $\boldsymbol{K}_e$. The calculation of elemental matrices is operator-specific, i.e., it depends on the application and can vary significantly. For example, the computation of $\boldsymbol{K}_e$ for the Laplacian operator is substantially cheaper than the operator in linear elasticity or Navier-Stokes equations. For complicated operators, matrix-free approaches might be expensive since, for each SPMV, we need to perform the indirect computation of elemental matrices.

Hybrid approach: As seen in (3), the global matrix $\boldsymbol{K}$ is accumulated by element contributions. Based on the element connectivity, the global SPMV $\boldsymbol{Kv}$ can be decomposed into a series of independent elemental EMV $\boldsymbol{K}_e\boldsymbol{v}_e$. This keeps the local computation structured, enabling good performance on modern architectures and keeping distributed memory computations efficient as in the matrix-free case. The same idea has been used by the so-called Element-by-element (EBE) methods [3], [4], [6]–[8], but largely for cases where the architecture did not have sufficient memory to perform

full global assembly, and not motivated by performance or programmatic aspects.

The challenges with matrix-free and matrix-assembled methods to FEM discussed above have received significant attention from the broader computational sciences community. We briefly summarize some of these approaches. The cost of assembling FEM matrices is well acknowledged, and several approaches have been proposed to make them more efficient. These include approaches that identify redundant computations [9] and perform cross-loop optimizations [10]. Optimizations have also been proposed for improving parallel performance using one-sided MPI communications [11] and approaches for speeding up matrix-free implementations [12]. Several recent works also focus on GPU and many-core implementations of matrix-assembled [13]–[15] and matrix-free [16]–[19] approaches to FEM, highlighting the difficulty in achieving good performance and scalability on modern architectures. Finally, efficient sparse matrix-vector (SpMV) is an active area of research with several works focusing on scaling them on different architectures [20]–[23].

III. Underlying ideas

To overcome the matrix assembly cost, we propose for the HYMV approach that element matrices are computed and stored locally. The global SPMV $\boldsymbol{Kv}$ is performed in a way similar to the matrix-free approach, i.e., elemental EMV is computed using locally stored pre-computed element matrices. Communication between processes is carried out for the elements sharing the nodes on partition boundaries.

No global assembly: Like matrix-free approaches, HYMV does not assemble the global matrix, reducing significantly the setup time when the number of processes p is large. Indeed, the setup time of HYMV does not depend on p provided that the granularity is kept constant. Furthermore, the storing of element matrices is extremely useful for applications with adaptive multiresolution (AMR) or frequent enrichments (e.g., XFEM for crack modeling), where only a minor subset of elements needs to be updated, while the global assembly is completely avoided during the simulation.

No re-computation of $\boldsymbol{K}_e$: Unlike matrix-free approaches, HYMV does not recompute the element matrices in each SPMV. This property is significantly helpful for applications having expensive elemental operators.

Global sparse linear algebra $\rightarrow$ local dense linear algebra: Achieving machine peak performance with sparse computations is challenging due to the arbitrary data movement patterns. The above is an additional drawback of the matrix-assembled approaches. HYMV converts globally sparse linear algebra to locally dense linear algebra that is well-suited for modern heterogeneous architectures.

Ease of programming: HYMV has intrinsic support to parallelism and optimizations enabling efficient SPMV computations on heterogeneous machines. Additionally, it is far easier to implement efficient vectorized linear algebra kernels for the elemental EMV in HYMV.

IV. Methodology

In this section, we present the key components of the HYMV approach for solving the sparse system $\boldsymbol{Ku} = \boldsymbol{f}$, which enable HYMV to be scalable on large-scale heterogeneous architectures.

A. HYMV's data structures

HYMV can perform the SPMV operation (in iterative solution) on arbitrary meshes using the distributed data structures and abstractions as described in the following section. Figure 1 shows a simple example to illustrate the maps used in HYMV. In distributed-memory FEM computation, the computational domain Ω is partitioned into subdomains with non-overlapping interiors, i.e., $\Omega = \{\omega_i\}_{i=1}^{p}$. Each subdomain is discretized into a finite set of elements, i.e., $\omega_i = \{e_m^i\}_{m=1}^{|\omega_i|}$. Each partition consists of locally-owned nodes used to define the basis functions (as explained in section II). The elemental and nodal indices can be *local* (i.e., well-defined only for a specific partition and globally ill-defined) or *global* (i.e. well-defined across all partitions). HYMV is agnostic to the underlying mesh structure provided the following information for each partition.

- Number of local elements present, i.e., $|\omega_i|$;
- E2G map: For partition i, $\texttt{E2G}^i$ acts as a mapping from local-element index of a node to its corresponding global index;
- Range of global indices of the nodes owned by ω_i, i.e., $[N^i_{begin}, N^i_{end}]$.

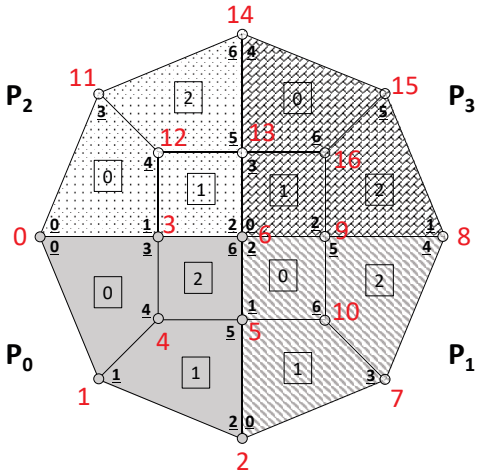


Fig. 1: Example to illustrate the E2G, E2L, LNSM, and GNGM maps used in HYMV. There are 4 partitions $P_0 \rightarrow P_3$. The mesh composes of 12 elements and 17 nodes. Global node indices (red color) are $0 \rightarrow 16$. The node's pattern matches with its owner's pattern. Local node indices (underlined) are $0 \rightarrow 6$ for every partition. Local element indices are framed. Taking P_2 as an example: $N_{begin} = 11$, $N_{end} = 14$, $N_{local} = 4$, $N_{total} = 7$, $G_{pre} = \{0, 3, 6\}$, $G_{post} = \{\emptyset\}$, element 0 has E2G = $[0, 3, 12, 11]$ and E2L = $[0, 1, 4, 3]$, LNSM = $\{5, 6\}$, and GNGM = $\{0, 1, 2\}$.

Based on this information, the following local maps are computed during the HYMV setup phase.

- E2L map: For partition i, $\texttt{E2L}^i$ acts as a mapping from the local-element index of a node (including ghost/halo node) to its corresponding local index. Algorithm 1 shows the construction of the E2L map;
- Local node scatter map (LNSM): For partition i, $\texttt{LNSM}^i$ denotes the local node indices of partition i that need to be scattered to neighboring partitions;
- Ghost node gather map (GNGM): For partition i, $\texttt{GNGM}^i$ denotes the ghost node indices of partition i that need to be accumulated to neighboring partitions for SPMV.

B. Connectivity maps

As introduced in section II, the global SPMV is performed via a series of elemental EMV based on the element connectivity. The algorithm of HYMV SPMV is listed in Algorithm 2. The extraction and accumulation of the element vectors shown in Algorithm 2 require the local map (i.e., $\texttt{E2L}^i$). As shown in Fig. 1 for given partition i, there exists nodal information that should be communicated from neighboring partitions. These are referred to as ghost/halo nodes. The ghost nodes are further categorized into pre-ghost (if they are communicated from neighboring partitions of lower ranks) and post-ghost (if they are communicated from neighboring partitions of higher ranks). The $\texttt{E2L}^i$ map is constructed based on the provided ω_i, $\texttt{E2G}^i$ map, and the range $[N^i_{begin}, N^i_{end}]$ (see Algorithm 1). The ghost nodes are identified based on the comparison of $\texttt{E2G}^i$ and $[N^i_{begin}, N^i_{end}]$. The $\texttt{E2L}^i$ map is merely an offset and reordering of the $\texttt{E2G}^i$ map to accommodate the ghost nodes and distributed computations on the mesh.

Algorithm 1 E2L map construction, processor i

Require: ω_i, $\texttt{E2G}^i$, and $[N^i_{begin}, N^i_{end}]$
Ensure: $\texttt{E2L}^i$, N_{local} number of locally owned nodes, N_{total} number of total nodes including ghost nodes

$N_{local} \leftarrow N^i_{end} - N^i_{begin} + 1$
$[G_{pre}, G_{post}] \leftarrow ComputeGhost(E2G^i, N^i_{begin}, N^i_{end})$
$N_{total} \leftarrow N_{local} + |G_{pre}| + |G_{post}|$
for $e \leftarrow 1$ to $|\omega_i|$ **do**
  $\texttt{E2L}[e] \leftarrow \{\emptyset\}$
  for $m \in \texttt{E2G}^i[e]$ **do**
    if $m < N^i_{begin}$ **then**
      $\texttt{E2L}[e] \leftarrow \texttt{E2L}[e] \cup G_{pre}.index(m)$
    else if $m > N^i_{end}$ **then**
      $\texttt{E2L}[e] \leftarrow \texttt{E2L}[e] \cup (|G_{pre}| + N_{local} + G_{post}.index(m))$
    else
      $\texttt{E2L}[e] \leftarrow \texttt{E2L}[e] \cup (|G_{pre}| + v - N^i_{begin})$

C. Distributed array

The distributed array (DA) is the underlying data structure in HYMV to represent partitioned vectors defined over a mesh. As shown in Fig. 2, the DA consists of pre-ghost, owned, and post-ghost nodal information. The ghost regions consist of the data received from neighboring partitions to perform the SPMV operation. The locally owned elements of partition ω_i can be further classified as independent and dependent elements based on the E2L map.

- Independent elements $I(\omega_i) = e \in \omega_i$ and $\forall m \in E2L^i[e]$ is owned by partition ω_i.
- Dependent elements $D(\omega_i) = \omega_i \setminus I(\omega_i)$

The above distinction between independent and dependent elements is used to overlap communication with computation (either on host or device if using GPU as presented at the end of this section).

Algorithm 2 SPMV in HYMV, processor i

Require: $\texttt{E2L}^i$, $\texttt{LNSM}^i$, $\texttt{GNGM}^i$ maps, $\{K_e\}_{e\in\omega^i}$ (HYMV setup), partitioned $\boldsymbol{u}^i, \boldsymbol{v}^i$ vectors
Ensure: $\boldsymbol{v}^i = (\boldsymbol{K}\boldsymbol{u})^i$

$\boldsymbol{v}^i \leftarrow 0$
$local\_node\_scatter\_begin(\boldsymbol{u}^i, LNSM^i)$
for $e \leftarrow 1$ to $|I(\omega_i)|$ **do** ▷ independent elements
  $u_e \leftarrow u^i(E2L^i[e])$ ▷ extract element vector $\boldsymbol{u}_e$
  $\boldsymbol{v}_e \leftarrow \boldsymbol{K}_e\boldsymbol{u}_e$
  $v^i(E2L^i[e]) \leftarrow v^i(E2L^i[e]) + v_e$ ▷ accumulate element vector $\boldsymbol{v}_e$
$local\_node\_scatter\_end(\boldsymbol{u}^i, LNSM^i)$
for $e \leftarrow 1$ to $|D(\omega_i)|$ **do** ▷ dependent elements
  $u_e \leftarrow u^i(E2L^i[e])$ ▷ extract element vector $\boldsymbol{u}_e$
  $\boldsymbol{v}_e \leftarrow \boldsymbol{K}_e\boldsymbol{u}_e$
  $v^i(E2L^i[e]) \leftarrow v^i(E2L^i[e]) + v_e$ ▷ accumulate element vector $\boldsymbol{v}_e$
$ghost\_node\_gather\_begin(\boldsymbol{v}^i, GNGM^i)$
$ghost\_node\_gather\_end(\boldsymbol{v}^i, GNGM^i)$

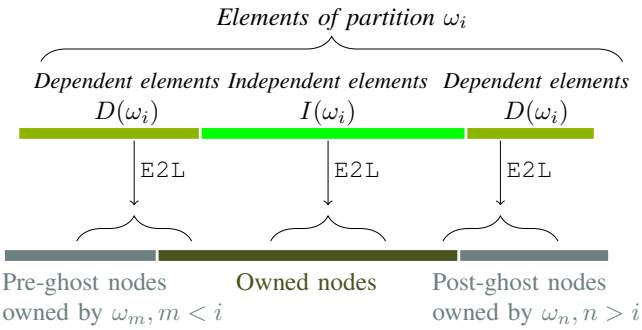


Fig. 2: Memory layout for the distributed array (i.e., partitioned vector). Partition ω_i has local nodes composing pre-ghost, post-ghost, and owned nodes. It is further logically decomposed into independent $I(\omega_i)$, and dependent $D(\omega_i)$ disjoint element sets such that $\omega_i = I(\omega_i) \cup D(\omega_i)$, which are used in overlapping the computation with the communication.

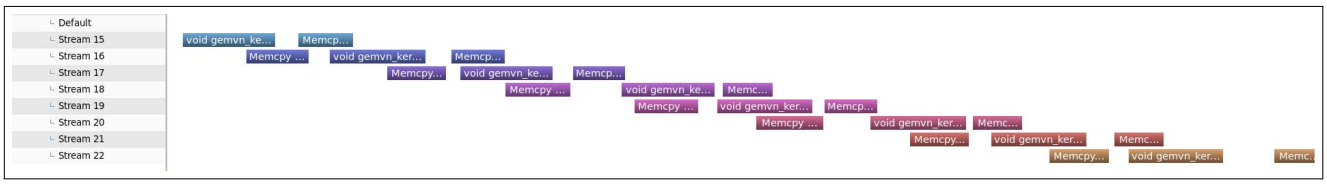


Fig. 3: Snapshot of profiling shows the overlapping between data transfers and kernel execution when using eight streams for the elasticity example defined in section V-B.

D. Communication maps (LNSM and GNGM)

The communications shown in Algorithm 2 require additional data structures to enable inter-process communication. To perform inter-process communication, we construct two maps: the local node scatter map (LNSM) and the ghost node gather map (GNGM). For a given partition ω_i, $\texttt{LNSM}^i$ stores the information on the subset of the locally owned nodes in partition ω_i, that needed to be scattered to neighboring partitions. Similarly, the inverse communication pattern is computed and stored in the ghost node gather map (GNGM). For partition ω_i, the $\texttt{GNGM}^i$ is used to accumulate (i.e., gather) the elemental EMV contribution from the neighboring partitions to the local nodes of the partition ω_i. The above maps are constructed once based on the provided E2G, and the range $[N^i_{begin}, N^i_{end}]$ during the HYMV setup phase.

E. Hybrid parallelism and vectorization

To achieve hybrid parallelism for HYMV SPMV, we deploy OpenMP shared-memory parallelism by distributing the local elemental EMV among OpenMP threads. Additionally, the elemental EMV is vectorized with either AVX512 intrinsics or OpenMP SIMD. For this, the element matrix $\boldsymbol{K}_e$ should be stored in column-major format (to minimize cache misses), and elemental EMV is computed as a sum of vector multiplication operations, i.e.,

$$\boldsymbol{v}_e = \sum_{j=1}^{n_e} [\boldsymbol{K}_e]_j \cdot [\boldsymbol{u}_e]_j, \tag{4}$$

where vector $[\boldsymbol{K}_e]_j$ is the j^{th}-column of $\boldsymbol{K}_e$, vector $[\boldsymbol{u}_e]_j$ contains only the j^{th}-component of $\boldsymbol{u}_e$, and n_e is the number of columns of $\boldsymbol{K}_e$.

F. GPU parallelism

As a prominent advantage of the HYMV approach, the acceleration of elemental EMV using GPU can be implemented straightforwardly. Since element matrices are stored locally, they are transferred from host to device once at the setup time and only updated if necessary (e.g., when local enrichment or refinement occurs). The algorithm of SPMV with GPU is shown in Algorithm 3, in which the implementation of data transfers and kernel execution is carried out using the MAGMA library [24]. Algorithm 3 presents the non-overlapping (i.e., pure GPU) SPMV. The overlapping of MPI communication and independent element computation can be easily included (as shown in Algorithm 2 of CPU-based SPMV) since HYMV provides the modular non-blocking communication and the dependent/independent element data structure (see Fig. 2). Since the number of elemental EMV is usually large, the batched matrix-vector multiplication [25], [26] is employed for the kernel execution. Furthermore, the kernel execution is set to overlap the data transferring between host and device to optimize the GPU performance. For this, we break up the array (of element matrices) $\boldsymbol{b}_{ke}$ and the arrays (of element vectors) $\boldsymbol{b}_{ue}$ and $\boldsymbol{b}_{ve}$ into N_s chunks. The data transfers and kernel execution of each chunk occur in N_s different streams which are overlapped. Figure 3 shows the overlapping obtained by using eight streams for the elasticity example described in section V-B. We employ OpenMP parallelism to further improve the performance when setting the array $\boldsymbol{b}_{ue}$ (pinned memory) and accumulating the array $\boldsymbol{b}_{ve}$ as shown in Algorithm 3.

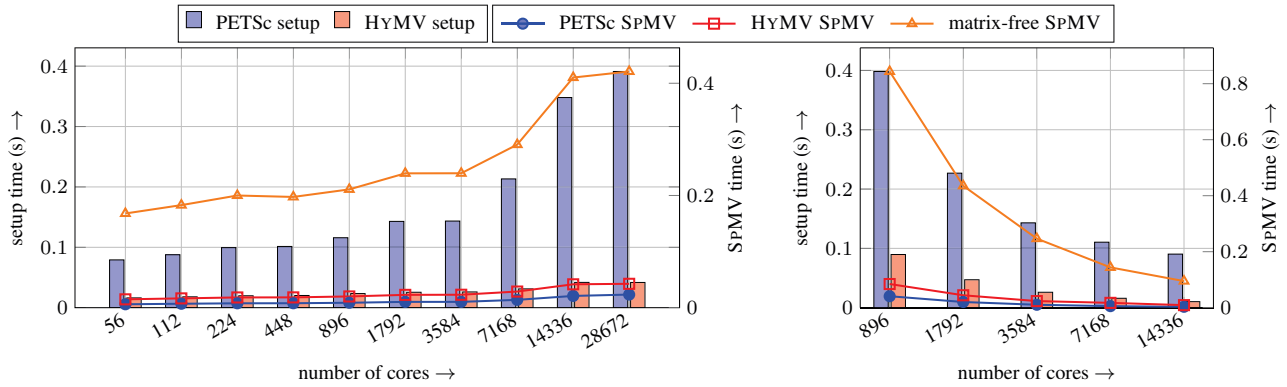
Algorithm 3 GPU SPMV in HYMV, processor i

1: $local\_node\_scatter\_begin(\boldsymbol{u}^i, LNSM^i)$
2: $local\_node\_scatter\_end(\boldsymbol{u}^i, LNSM^i)$
3: set element vectors $\boldsymbol{b}_{ue}$ from values of $\boldsymbol{u}^i$ ▷ parallelized by OpenMP
4: **for** stream $s \leftarrow 1$ to N_s **do**
5: transfer $\boldsymbol{b}^s_{ue}$ from host to device
6: batched EMV $\boldsymbol{b}^s_{ve} = \boldsymbol{b}^s_{ke}\boldsymbol{b}^s_{ue}$ on device
7: transfer $\boldsymbol{b}^s_{ve}$ from device to host
8: accumulate element vectors $\boldsymbol{b}_{ve}$ to $\boldsymbol{v}^i$ ▷ parallelized by OpenMP
9: $ghost\_node\_gather\_begin(\boldsymbol{v}^i, GNGM^i)$
10: $ghost\_node\_gather\_end(\boldsymbol{v}^i, GNGM^i)$

As mentioned above, independent elements (see Fig. 2) do not need to communicate with neighboring partitions. Thus their EMV can overlap with the communication followed by the EMV of dependent elements. We implemented two options for the overlapping: 1). the host performs the EMV of dependent elements; 2). the device performs the EMV of dependent elements. The performance of these options is discussed in section V-D.

V. RESULTS

In this section, we first summarize the tests used to verify the implementation's correctness. Next, we present the numerical experiments showing the scalability of the HYMV approach in comparison with the matrix-assembled (i.e., PETSc [1]) and the matrix-free methods.



(a) Weak scalability: the problem size is 11.3K DoFs per process. The largest problem has 331M DoFs. HYMV setup is 10× faster than PETSc setup.

(b) Strong scalability: the problem size is 42M DoFs. HYMV setup is 9× faster than PETSc setup.

Fig. 4: Scalability for Poisson's problem using structured meshes with 8-node linear elements on TACC's Frontera. The bars show matrix setup time (left y-axis). The lines show the time for ten SPMV operations (right y-axis).

A. Experimental setup

The scalability experiments reported in this paper were performed on TACC's Frontera supercomputer. Frontera is an Intel supercomputer at Texas Advanced Computing Center (TACC) with a total of 8,008 nodes, each consisting of a Xeon Platinum 8280 ("Cascade Lake") processor, with a total of 448,448 cores. Each node has 192 GB of memory. The interconnect is based on Mellanox HDR technology with full HDR (200 Gb/s) connectivity between the switches and HDR100 (100 Gb/s) connectivity to the compute nodes. GPU SPMV experiments are performed on Frontera nodes having four NVIDIA Quadro RTX 5000 (Turing) accelerators, with GPU memory of 16 GB.

B. Correctness verification

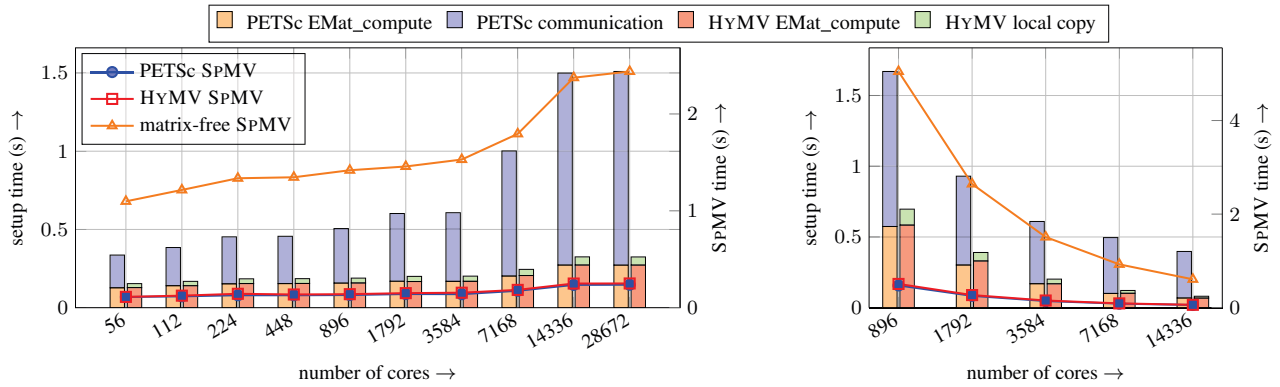
The key component of HYMV implementation is the SPMV function, which is incorporated into PETSc matrix-free solvers [1]. To verify the implementation's correctness, we examined Poisson's problem defined as $\nabla^2 u + \sin(2\pi x)\sin(2\pi y)\sin(2\pi z) = 0 \; \forall \{x, y, z\} \in \Omega = [0, 1]^3; u = 0$ on $\partial\Omega$. We used linear hex elements for the analysis. The domain is partitioned in z-direction into four partitions owning equal numbers of elements. We started with a mesh of $10 \times 10 \times 10$ elements, and subsequently doubled the elements in all directions up to $160 \times 160 \times 160$ elements. The numerical result of u is compared with the exact solution of $u = \frac{1}{12\pi^2}\sin(2\pi x)\sin(2\pi y)\sin(2\pi z)$ by computing $err = \|\boldsymbol{u} - \boldsymbol{u}_{exact}\|_\infty$. All errors are between 23.4×10^{-5} (the coarsest mesh) and 0.1×10^{-5} (the finest mesh).

Next, we examined the problem of an elastic prismatic bar stretched by its weight [27]. The bar, of dimension $\{L_x, L_y, L_z\}$, is hung from its top face center, with gravity acceleration g. The material has Young's modulus E, Poisson's ratio ν, and density ρ. A uniform traction $t_z = \rho g z$ is applied on the top face. The coordinate system's origin is at the bottom face center. For the analysis, uniform meshes of either linear hex elements or quadratic hex elements are employed. Three meshes are used for the verification: $4 \times 4 \times 4$, $8 \times 8 \times 8$, and $16 \times 16 \times 16$ elements. These meshes are partitioned in z-direction into 2, 4, 8 partitions, respectively. The numerical displacements in the bar are compared with the exact solution of $u_x = -\frac{\nu\rho g}{E}xz$; $u_y = -\frac{\nu\rho g}{E}yz$; $u_z = \frac{\rho g}{2E}(z^2 - L_z^2) + \frac{\nu\rho g}{2E}(x^2+y^2)$ by computing $err = \|\boldsymbol{u} - \boldsymbol{u}_{exact}\|_\infty$. All meshes give $err < 10^{-8}$.

C. Parallel scalability

For the experiments of HYMV parallel scalability, we repeat the problems described in section V-B with different meshes. The performance of HYMV is compared with the matrix-assembled approach (i.e., PETSc) and matrix-free approach.

*1) **Example of Poisson's equation**:* Figure 4a shows the weak scalability with a problem size of approximately 11.3K degrees of freedom (DoFs) per process. In matrix-assembled and HYMV approaches, we measure the setup time that includes the element matrix computation, followed by the global matrix assembly (matrix-assembled) or local copy operation (HYMV). There is no setup time in the matrix-free method. Following the matrix setup, we perform ten SPMV operations. In the matrix-assembled approach, we use PETSc `MatMult` function for the SPMV operation. The HYMV local assembly is significantly faster than the global assembly of the matrix-assembled approach. This is due to the high communication overhead associated with the global assembly. Note that the cost of HYMV SPMV is comparable with the matrix-assembled approach and significantly lower than the matrix-free approach. The matrix-free approach is much more expensive than matrix-assembled and HYMV approaches, mainly due to the indirect re-computation of element matrices. In this experiment, the largest problem has 330M DoFs, where the HYMV local assembly is 10× faster than the global matrix assembly. Figure 4b shows the strong scalability of this Poisson problem for a fixed problem size of 42M DoFs, in which the HYMV setup is 9× faster than the matrix-assembled setup.



(a) Weak scalability: the problem size is 33.5K DoFs per process. The largest problem has 918M DoFs. HYMV setup is 5× faster than PETSc setup.

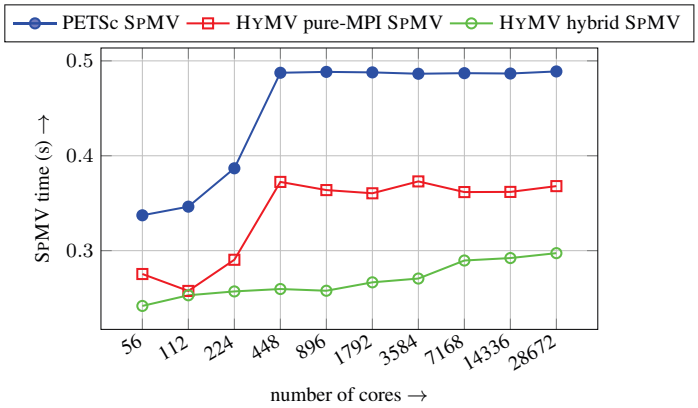
(b) Strong scalability: the problem size is 117M DoFs. HYMV setup is 5× faster than PETSc setup.

Fig. 5: Scalability for the elasticity problem using structured meshes with 8-node linear elements on TACC's Frontera. The bars show matrix setup time (left y-axis), in which `EMat_compute` denotes the element matrix computation. The lines show the time of ten SPMV operations (right y-axis).

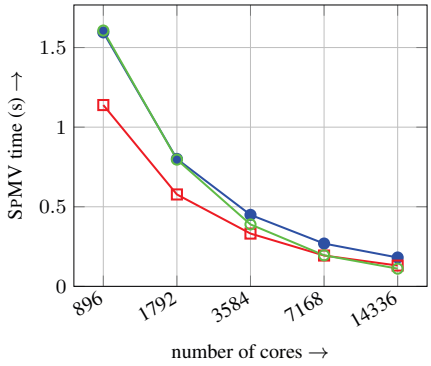
*2) **Example of elasticity**:* Figure 5a shows the weak scalability for this problem using linear elements. The problem size is approximately 33.5K DoFs per process. Also, in this figure, we present the overall cost breakdown between elemental matrix computation and the overhead associated with local and global setup operations in the HYMV and matrix-assembled approaches. The largest problem has 918M DoFs, where the HYMV setup is 5× faster than the matrix-assembled setup. Figure 5b shows the strong scaling results across 14K cores for a fixed problem size of 117M DoFs. It is seen that the matrix-free approach is significantly expensive compared to matrix-assembled and HYMV approaches, mainly due to the indirect re-computation of the element matrices. The HYMV setup is 5× faster than the matrix-assembled setup.

Figure 6a shows the weak scalability for this problem using quadratic elements, with a problem size of approximately 33.5K DoFs per process. In this experiment, we also compare the performance of the HYMV approach between pure MPI parallelism and hybrid (MPI + OpenMP) parallelism. To minimize network communication, in hybrid parallelism, the number of OpenMP threads is set to the number of physical cores per socket, and the number of MPI processes is set to the number of sockets over the nodes. It is seen that the SPMV time in both parallelism schemes of the HYMV approach is less than the matrix-assembled approach. Especially, the SPMV time of hybrid parallelism is better than pure MPI parallelism for this elasticity example when using quadratic elements (i.e., the complexity of elemental matrix computation is higher compared with the case of linear elements). As mentioned earlier, the HYMV approach enables efficient element-level shared-memory parallelism. Figure 6b shows the strong scalability of this example with a fixed problem size of 174M DoFs.

*3) **Example of unstructured meshes**:* Uniformly structured meshes are trivial to partitioning and achieving work and communication balance. Moving to unstructured meshes



(a) Weak scalability: the problem size is 33.5K DoFs per process. In average, HYMV hybrid SPMV is 1.7× faster than PETSc SPMV.



(b) Strong scalability: the problem size is 174M DoFs. In average, HYMV hybrid SPMV is 1.2× faster then PETSc SPMV.

Fig. 6: Scalability for the elasticity problem using structured meshes with 20-node quadratic elements on TACC's Frontera. The measured time is for ten SPMV operations.

creates additional partitioning challenges, leading to complex communication patterns where the matrix-assembled approach becomes expensive. This experiment presents a comparison of HYMV and the matrix-assembled approach on unstructured meshes with quadratic tetrahedral elements for the Poisson problem described in section V-B. We used Gmsh [28] to generate unstructured meshes and partitioned the meshes using METIS [29] library. A strong scalability study on HYMV and matrix-assembled approach for unstructured meshes is presented in Fig. 7. We used a fixed mesh size of 8.5M DoFs (6.3M elements). As shown in Fig. 7, HYMV obtains much better parallel scalability than the matrix-assembled approach. The HYMV average setup is 11× faster than the matrix-assembled setup. The sparsity pattern of the matrix-assembled and the partitioning boundaries becomes irregular with unstructured grids, making the matrix-assembled setup and the SPMV inefficient with the increasing number of nodes. In contrast, HYMV converts the global sparse computation (i.e., irregular) to dense local computation (i.e., regular), with the efficient overlapping between communication and computation to achieve efficient parallel scalability.

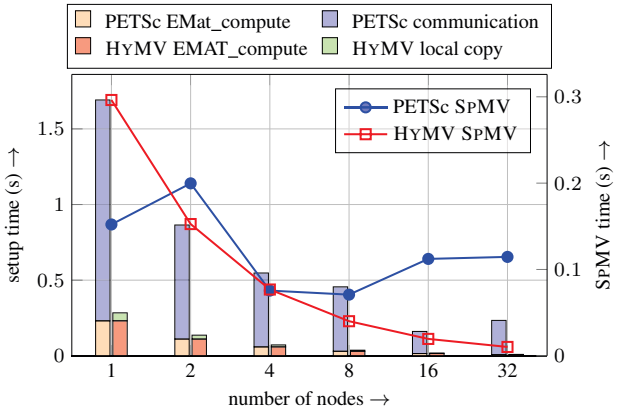
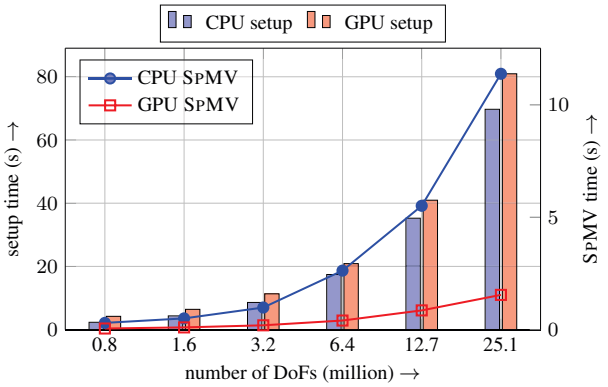


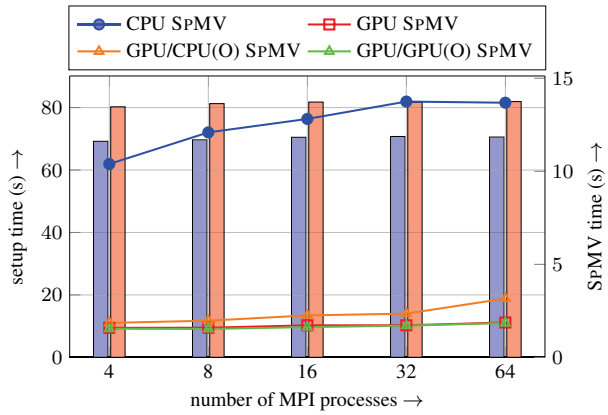
Fig. 7: Strong scalability for Poisson's problem using an unstructured mesh with quadratic tetrahedron elements on TACC's Frontera across 1792 cores (32 nodes). The bars show matrix setup time (left y-axis). The lines show the time for ten SPMV operations (right y-axis). The problem size is 8.5M DoFs. In average, HYMV setup is 11× faster than PETSc setup, and HYMV SPMV is 3.6× faster than PETSc SPMV.

D. HYMV-*GPU* SPMV

We examined the performance of GPU SPMV for the elasticity problem described in section V-B. Firstly, we ran the experiment with 25M DoFs on a single Frontera node (with 2 MPI processes and 14 OpenMP threads) with increasing streams and measured the time of ten SPMV. The results show that using eight streams gives the best performance for this problem. For this, we used eight streams for the following experiments. Next, we compare the performance of GPU SPMV with CPU SPMV. We run the experiment on a single Frontera node (with 2 MPI processes and 14 OpenMP threads) with increasing DoFs. As shown in Fig. 8a, the speedup of GPU SPMV is approximately constant (e.g., the speedup is 7.4 when DoFs = 25.1M). Even though the data transfer overhead increases, the GPU's parallel performance is significantly better than the CPU for larger problems. The GPU matrix setup time is slightly larger than the CPU setup time due to the overhead associated with element matrix transfer from the host to the device. Note that the HYMV setup is a one-time cost to pay to achieve significant speedup in the SPMV operation, which is essential for fast iterative solving of a linear system.



(a) Single Frontera node with increasing DoFs: the speedup of GPU SPMV is 7.4× for the problem size of 25.1M DoFs.



(b) Weak scalability: the problem size is 6.3M DoFs per process. In average, GPU SPMV is 7.5× faster than CPU SPMV.

Fig. 8: Performance of HYMV-GPU SPMV compared with HYMV-CPU SPMV for the elasticity problem using structured meshes with 20-node quadratic elements on TACC's Frontera. The bars show matrix setup time (left y-axis), and the lines indicate the time of ten SPMV operations (right y-axis).

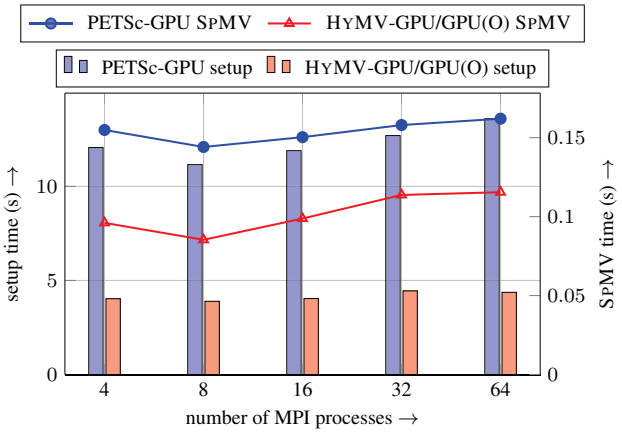
Next, we performed weak scalability for this elasticity problem across 16 Frontera nodes, as shown in Fig. 8b. Each GPU Frontera node has 16 cores and 4 GPU accelerators. We run the experiments using 4 OpenMP threads and 4 MPI tasks. Similar to the previous single-node experiment, GPU SPMV is about 7.5× faster than CPU SPMV, and GPU's matrix setup is a bit higher than CPU. Three different schemes of overlapping

are used: 1). GPU SPMV (i.e., using blocking MPI communication followed by GPU computation of all elements), 2). GPU/CPU (overlapped) SPMV (i.e., using non-blocking MPI communication overlapped by GPU computation of independent elements and CPU computation of dependent elements), 3). GPU/GPU (overlapped) SPMV (i.e., using non-blocking MPI communication overlapped by GPU computations of both independent and dependent elements). There is no notable difference between GPU and GPU/GPU (overlapped) for the distributed GPU experiment. The above is expected because of the problem scale (i.e., 16 nodes), while overlapped communication performs better in large-scale runs (see figures 4a, 5a, and 6a). We also see that the GPU/CPU (overlapped) is slower with increasing nodes due to the larger ratio of dependent elements over independent elements. Finally, we compared HYMV-GPU with PETSc-GPU (using cuSPARSE) as shown in Fig. 9a (weak scaling) and Fig. 9b (strong scaling). It is seen that using 27-node quadratic elements, HYMV-GPU is faster than PETSc-GPU in both setup time and SPMV time.

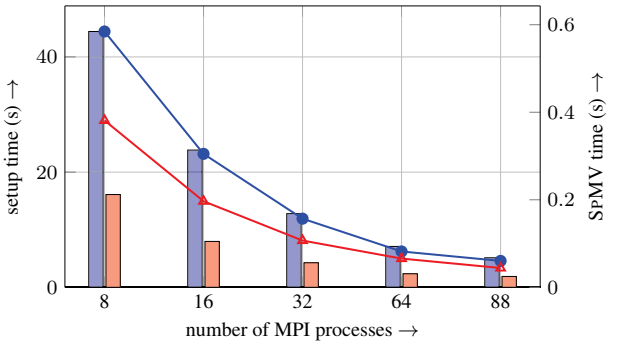
E. Throughput comparison

As an additional comparison on the throughput between HYMV, matrix-assembled, and matrix-free methods, we computed the floating-point operations (FLOP) per second in the SPMV function. Firstly, we analyzed the SPMV function performance for the elasticity problem using quadratic elements. Figure 10 is the roofline plot generated using Intel Advisor [30], where the arithmetic intensity (AI) and work (in GFLOP/s) of each method are presented. We ran the experiment on a single core of the Frontera node. The experimental results show the work done by HYMV lies in between matrix-assembled and matrix-free methods. Specifically, the HYMV SPMV archives AI = 0.079 FLOP/Byte and 1.614 GFLOP/s, while the matrix-assembled approach has AI=0.161 FLOP/Byte and 1.062 GFLOP/s, and the matrix-free approach has AI=0.083 FLOP/Byte and 5.053 GFLOP/s. The lower AI of HYMV SPMV (compared with the matrix-assembled method) is due to the loading of element matrices and vectors (to perform EMV) and the accumulation of element vectors back to the global vector. In contrast, the matrix-assembled SPMV only loads the global matrix (accumulated in the setup phase) and global vector. On the other hand, the matrix-free SPMV loads more data (e.g., mesh information, material properties) and performs additional computations (e.g., interpolation, numerical integration). Hence, even though the HYMV approach uses dense formats, the achieved AI is different.

Next, we computed the FLOP per second (FLOP rate) of ten SPMV function calls for the same elasticity problem with results presented in Table I. This experiment is conducted on Frontera with one and four nodes (56 cores per node). For the same problem size, HYMV reports a higher FLOP rate than the matrix-assembled method. Even though matrix-assembled SPMV has higher AI (as shown in Fig. 10) and lower computations, it fails to achieve a higher FLOP rate due to irregular memory access. HYMV achieved a higher FLOP rate due to dense local computations leading to memory-friendly structured memory accesses. These experiments show that the matrix-free approach performs the most work, achieving the highest FLOPS among the three methods due to minimum memory accesses. But we argue that both these traditional metrics, i.e., higher AI and higher FLOP rate, are not the only considerations, as HYMV achieves a lower time-to-solution than both other approaches.



(a) Weak scalability: the problem size is ∼488K DoFs per MPI process. In average, HYMV-GPU is 3.0× faster in setup time and 1.5× faster in SPMV time than PETSc-GPU.



(b) Strong scalability: the problem size is 15.8M DoFs. In average, HYMV-GPU is 2.9× faster in setup time and 1.4× faster in SPMV time than PETSc-GPU.

Fig. 9: Performance of HYMV-GPU SPMV compared with PETSc-GPU SPMV for the elasticity problem using unstructured meshes with 27-node quadratic elements on TACC's Frontera. The bars show matrix setup time (left y-axis), and the lines indicate the time of ten SPMV operations (right y-axis).

F. Preconditioning and total solve time

For most applications, the total solve time is the metric of interest, which depends on the number of iterations for convergence, the setup, and the SPMV time. Preconditioning can reduce the number of iterations, thereby the total solve time. Therefore, this section presents experiments comparing total solve time (i.e., from matrix setup to solution convergence) with and without preconditioning. HYMV SPMV is

Method	granularity = 0.1M DoFs per MPI process						granularity = 0.2M DoFs per MPI process					
	one node, $N_p = 56$			four nodes, $N_p = 224$			one node, $N_p = 56$			four nodes, $N_p = 224$		
	GFLOP	Time	GFLOP/s	GFLOP	Time	GFLOP/s	GFLOP	Time	GFLOP/s	GFLOP	Time	GFLOP/s
Matrix-assembled	19.2	0.80	24.1	76.8	0.78	98.7	38.2	1.55	24.7	152.8	1.55	98.4
HYMV	32.3	0.72	44.7	129.0	0.58	221.3	64.5	1.17	55.0	258.0	1.21	213.7
HYMV GPU	32.3	0.31	103.7	129.0	0.36	361.3	64.5	0.61	106.2	258.0	0.65	396.7
Matrix-free	2,264.0	7.46	303.4	9,056.1	7.47	1,211.9	4,528.0	14.96	302.7	18,112.1	15.05	1,203.6

TABLE I: Number of flops (GFLOP), time (s) and flop rate (GFLOP/s) of ten SPMV performed by matrix-assembled, HYMV, and matrix-free approaches for the elasticity problem using 20-node quadratic elements on TACC's Frontera across 224 cores. The weak scaling is set up to run on one and four nodes of Frontera (56 cores per node). Two values of granularity 0.1M DoFs and 0.2M DoFs per MPI task are examined.

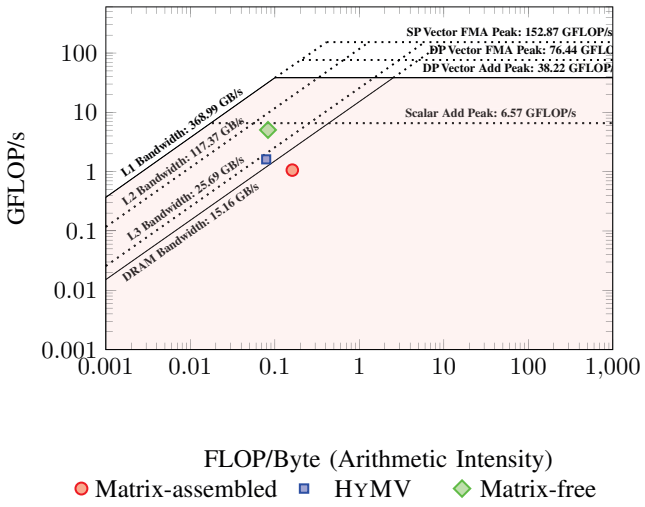


Fig. 10: Roofline plot for the SPMV function of HYMV compared with matrix-assembled and matrix-free approaches for the elasticity problem using 20-node quadratic elements on a single core of Frontera. The plot was generated using Intel Advisor [30].

incorporated with PETSc solvers via the `MatShell` interface. We employ the conjugate gradient (CG) method in all experiments. We examine the problem of an elastic bar presented in section V-B. Firstly, we take a bar of dimensions $L_x = L_y = L_z = 100$ and use an unstructured mesh with 8-node linear elements. The problem is solved using CG with no preconditioning and CG with Jacobi preconditioner, and the results are shown in Fig. 11a. Secondly, we use structured meshes with 20-node quadratic elements and partition the bar in the z-direction (both L_z and the number of elements in the z-direction are increased proportionally with the number of partitions, while L_x and L_y are fixed). Figure 11b presents the total solve time using Jacobi and block Jacobi preconditioners for this experiment. It can be seen that HYMV is faster than PETSc in all cases. We remark that, for block Jacobi preconditioner, HYMV needs to assemble the diagonal block matrix (i.e., the block matrix corresponding to owned DoFs). Finally, we re-examine the first experiment using unstructured meshes of 27-node quadratic elements and compare HYMV-GPU with PETSc's CUDA backend, as shown in Fig. 11c. It is seen that HYMV-GPU is also faster than PETSc-GPU in total solve time. It is noted that HYMV only uses the GPU for accelerating SPMV but not for other operations part of the CG solve (handled by PETSc). Additionally, we incorporated HYMV SPMV with select preconditioners supported by PETSc to demonstrate the feasibility of using HYMV with preconditioning. However, these are not heavily optimized, and this will be addressed in future work.

VI. Conclusion

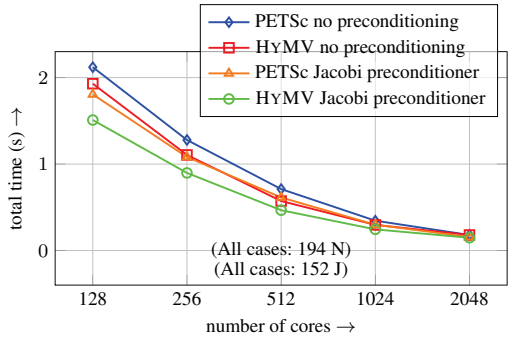
This paper presents an adaptive-matrix approach for the SPMV when solving linear systems arising from PDE discretizations using iterative solvers. The proposed HYMV approach uses local assembly, avoids global communication, and performs SPMV using dense linear algebra. HYMV is well-suited for modern heterogeneous architectures, including GPUs. HYMV enables efficient parallelization with MPI, SIMD, OpenMP, and CUDA, achieving excellent scalability and speedup in heterogeneous architectures. The developed HYMV is a standalone library that can easily integrate with existing computational codes and linear solver packages such as PETSc.

Acknowledgment

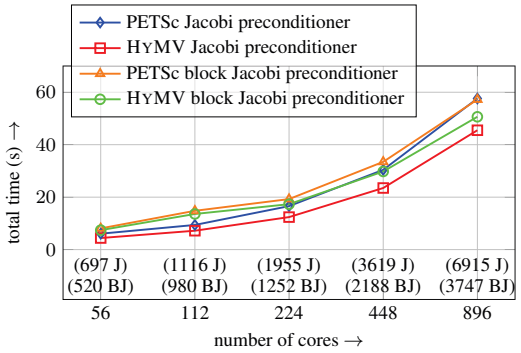
This work was funded in part by National Science Foundation grant OAC-1808652. The first, fifth, and sixth authors acknowledge support from the Air Force Research Laboratory (under FA8650-19-C-5212 and FA8650-17-C-5269). The computing resources on Frontera were through an allocation by the Texas Advanced Computing Center PHY20033.

Appendix

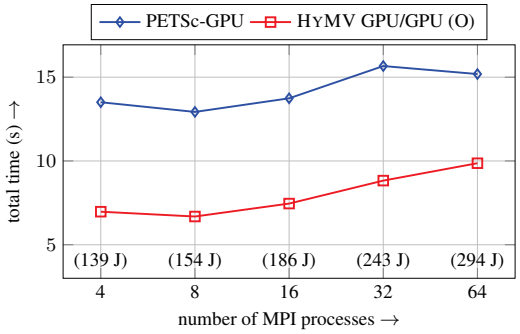
Algorithm 4 presents the SPMV of the matrix-free approach [31] that we used for the comparisons with HYMV. The overlapping of communication and independent element computation is implemented similarly to Algorithm 2. The critical difference from HYMV SPMV is the re-computation of elemental matrices $\boldsymbol{K}_e$ (based on the nodal coordinates $\boldsymbol{x}_e$ and material properties) during every solution iteration instead of loading from memory.



(a) Strong scalability: an unstructured mesh with 8-node linear elements is employed. The problem size is 3.4M DoFs. The infinity norm of error (compared with analytic solution) is $\mathcal{O}(10^{-4})$ for no preconditioning, and $\mathcal{O}(10^{-5})$ for Jacobi preconditioner. In average, HYMV is $1.1\times$ (no preconditioning) and $1.2\times$ (Jacobi preconditioner) faster than PETSc.



(b) Weak scalability: structured meshes of 20-node quadratic elements are employed. The problem size is 10.3K DoFs per process. The infinity norm of error (compared with analytic solution) is $\mathcal{O}(10^{-8})$ for Jacobi preconditioner, and $\mathcal{O}(10^{-5})$ for block Jacobi preconditioner. In average, HYMV is $1.3\times$ (Jacobi preconditioner) and $1.1\times$ (block Jacobi preconditioner) faster than PETSc.



(c) Weak scalability: unstructured meshes of 27-node quadratic elements are employed. The problem size is ∼488K DoFs per MPI process. Jacobi preconditioner is used. The infinity norm of error (compared with analytic solution) is $\mathcal{O}(10^{-5})$. In average, HYMV is $1.8\times$ faster than PETSc.

Fig. 11: Total solve time for the elasticity problem using different mesh types, elements, and preconditioners. The solution converges with relative tolerance $\epsilon = 10^{-3}$ after the corresponding number of iterations shown in the parentheses, in which {N, J, BJ} represents for {No, Jacobi, block Jacobi} preconditioner, respectively. In all cases, HYMV is faster than PETSc in total solve time.

Algorithm 4 SPMV in matrix-free approach, processor i

Require: $\texttt{E2L}^i$, $\texttt{LNSM}^i$, $\texttt{GNGM}^i$ maps, partitioned $\boldsymbol{u}^i, \boldsymbol{v}^i$ vectors

Ensure: $\boldsymbol{v}^i = (\boldsymbol{K}\boldsymbol{u})^i$

$\boldsymbol{v}^i \leftarrow 0$
$local\_node\_scatter\_begin(\boldsymbol{u}^i, LNSM^i)$
$local\_node\_scatter\_end(\boldsymbol{u}^i, LNSM^i)$
for $e \leftarrow 1$ to $|\omega_i|$ **do**
  $u_e \leftarrow u^i(E2L^i[e])$ ▷ extract element vector $\boldsymbol{u}_e$
  compute $\boldsymbol{K}_e$
  $\boldsymbol{v}_e \leftarrow \boldsymbol{K}_e\boldsymbol{u}_e$
  $v^i(E2L^i[e]) \leftarrow v^i(E2L^i[e]) + v_e$ ▷ accumulate element vector $\boldsymbol{v}_e$
$ghost\_node\_gather\_begin(\boldsymbol{v}^i, GNGM^i)$
$ghost\_node\_gather\_end(\boldsymbol{v}^i, GNGM^i)$

REFERENCES

[1] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, "PETSc Web page," https://www.mcs.anl.gov/petsc, 2019. [Online]. Available: https://www.mcs.anl.gov/petsc

[2] M. Grossman, C. Thiele, M. Araya-Polo, F. Frank, F. O. Alpak, and V. Sarkar, "A survey of sparse matrix-vector multiplication performance on large matrices," 2016.

[3] T. J. Hughes, I. Levit, and J. Winget, "An element-by-element solution algorithm for problems of structural and solid mechanics," *Computer Methods in Applied Mechanics and Engineering*, vol. 36, pp. 241–254, 1983.

[4] G. F. Carey and B. N. Jiang, "Element-by-element linear and nonlinear solution schemes," *Communications in Applied Numerical Methods*, vol. 2, no. 2, pp. 145–153, 1986.

[5] G. F. Carey, "Parallelism in finite element modeling," *Communications in Applied Numerical Methods*, vol. 2, no. 3, pp. 281–288, 1986.

[6] J. M. Winget and T. J. Hughes, "Solution algorithms for nonlinear transient heat conduction analysis employing element-by-element iterative strategies," *Computer Methods in Applied Mechanics and Engineering*, vol. 52, pp. 711–815, 1985.

[7] G. F. Carey, E. Barragy, R. McLay, and M. Sharma, "Element-by-element vector and parallel computations," *Communications in Applied Numerical Methods*, vol. 4, pp. 299–307, 1988.

[8] E. Barragy and G. F. Carey, "Parallel-vector computation with high-p element-by-element methods," *International Journal of Computer Mathematics*, vol. 44, pp. 329–339, 1992.

[9] R. C. Kirby, M. Knepley, A. Logg, and L. R. Scott, "Optimizing the evaluation of finite element matrices," *SIAM Journal on Scientific Computing*, vol. 27, no. 3, pp. 741–758, 2005.

[10] F. Luporini, A. L. Varbanescu, F. Rathgeber, G.-T. Bercea, J. Ramanujam, D. A. Ham, and P. H. Kelly, "Cross-loop optimization of arithmetic intensity for finite element local assembly," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 4, pp. 1–25, 2015.

[11] N. Jansson, "Optimizing sparse matrix assembly in finite element solvers with one-sided communication," in *International Conference on High Performance Computing for Computational Science*. Springer, 2012, pp. 128–139.

[12] S. Bauer, D. Drzisga, M. Mohr, U. Rude, C. Waluga, and B. Wohlmuth, "A stencil scaling approach for accelerating matrix-free finite element implementations," *SIAM Journal on Scientific Computing*, vol. 40, no. 6, pp. C748–C778, 2018.

[13] C. Cecka, A. J. Lew, and E. Darve, "Assembly of finite element methods on graphics processors," *International journal for numerical methods in engineering*, vol. 85, no. 5, pp. 640–669, 2011.

[14] A. Dziekonski, P. Sypek, A. Lamecki, and M. Mrozowski, "Accuracy, memory, and speed strategies in gpu-based finite-element matrix-generation," *IEEE Antennas and Wireless Propagation Letters*, vol. 11, pp. 1346–1349, 2012.
[15] M. Kronbichler and K. Ljungkvist, "Multigrid for matrix-free high-order finite element computations on graphics processors," *ACM Transactions on Parallel Computing*, vol. 6, no. 1, 2019.
[16] K. Ljungkvist, "Matrix-free finite-element computations on graphics processors with adaptively refined unstructured meshes." in *SpringSim (HPC)*, 2017, pp. 1–1.
[17] J. Martínez-Frutos and D. Herrero-Pérez, "Efficient matrix-free gpu implementation of fixed grid finite element analysis," *Finite Elements in Analysis and Design*, vol. 104, pp. 61–71, 2015.
[18] J. Martínez-Frutos, P. J. Martínez-Castejón, and D. Herrero-Pérez, "Fine-grained gpu implementation of assembly-free iterative solver for finite element problems," *Computers & Structures*, vol. 157, pp. 9–18, 2015.
[19] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. Cerveny, V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, W. Pazner, M. Stowell, V. Tomov, I. Akkerman, J. Dahm, D. Medina, and S. Zampini, "Mfem: A modular finite element methods library," *Computers and Mathematics with Applications*, vol. 81, pp. 42–74, 2021.
[20] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," Nvidia Technical Report NVR-2008-004, Nvidia Corporation, Tech. Rep., 2008.
[21] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, "Cvr: Efficient vectorization of spmv on x86 processors," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018, pp. 149–162.
[22] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, "Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 347–358.
[23] C. Liu, B. Xie, X. Liu, W. Xue, H. Yang, and X. Liu, "Towards efficient spmv on sunway manycore architectures," in *Proceedings of the 2018 International Conference on Supercomputing*, 2018, pp. 363–373.
[24] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5-6, pp. 232–240, Jun. 2010.
[25] A. Haidar, T. Dong, S. Tomov, P. Luszczek, and J. Dongarra, "Framework for batched and gpu-resident factorization algorithms to block householder transformations," in *ISC High Performance*, Springer. Frankfurt, Germany: Springer, 07-2015 2015.
[26] A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, T. Kolev, I. Masliah, and S. Tomov, "High-Performance Tensor Contractions for GPUs," Tech. Rep. UT-EECS-16-738, 01-2016 2016.
[27] S. Timoshenko and J. N. Goodier, *Theory of elasticity*. McGraw-Hill Book Company, 1951.
[28] C. Geuzaine and J.-F. Remacle, "Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities," *International journal for numerical methods in engineering*, vol. 79, no. 11, pp. 1309–1331, 2009.
[29] G. Karypis and V. Kumar, "Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," 1997.
[30] Intel, "Intel advisor." [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/advisor.html#gs.2pbhqy
[31] M. O. Deville, P. F. Fischer, and E. H. Mund, *High-Order Methods for Incompressible Fluid Flow*, ser. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2002.