KDRSolvers: Scalable, Flexible, Task-Oriented Krylov Solvers

David Kai Zhang
Stanford University
Stanford, CA, USA
dkzhang@stanford.edu

Rohan Yadav
Stanford University
Stanford, CA, USA
rohany@cs.stanford.edu

Alex Aiken
Stanford University
Stanford, CA, USA
aaiken@stanford.edu

Fredrik Kjolstad
Stanford University
Stanford, CA, USA
kjolstad@stanford.edu

Sean Treichler
NVIDIA Corporation
Santa Clara, CA, USA
sean@nvidia.com

Abstract

We present KDRSolvers, a novel framework for representing sparse linear systems and implementing Krylov subspace methods on modern heterogeneous supercomputers. KDRSolvers uses *dependent partitioning* to uniformly represent sparse matrix storage formats as abstract maps between a matrix's domain, range, and set of nonzero entries. This abstraction enables KDRSolvers to define universal co-partitioning operators for matrices and vectors independent of underlying storage formats, allowing changes in data partitioning strategies to automatically propagate through an application with no code modification. KDRSolvers also introduces *multi-operator systems* in which matrix and vector data can be ingested and processed in multiple non-contiguous pieces without data movement. Our implementation of KDRSolvers, targeting the Legion runtime system, achieves greater flexibility and competitive performance compared to PETSc and Trilinos. In experiments with up to 1,024 GPUs on the Lassen supercomputer, our implementation achieves up to a 9.6% reduction in execution time per iteration.

CCS Concepts

• **Computing methodologies** → **Parallel computing methodologies**; **Linear algebra algorithms**; **Representation of mathematical objects**; • **Software and its engineering** → **Distributed systems organizing principles**; • **Mathematics of computing** → **Solvers**; **Mathematical software performance**.

Keywords

Krylov Subspace Methods, Sparse Iterative Solvers, Task-Oriented Runtime Systems, Dependent Partitioning, Legion, PETSc, Trilinos

ACM Reference Format:

David Kai Zhang, Rohan Yadav, Alex Aiken, Fredrik Kjolstad, and Sean Treichler. 2025. KDRSolvers: Scalable, Flexible, Task-Oriented Krylov Solvers. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3731599.3767501

SC Workshops '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1871-7/25/11
https://doi.org/10.1145/3731599.3767501

1 Introduction

Solving large, sparse systems of linear equations is one of the most important workloads in computational science and engineering. The numerical analysis and scientific computing communities have spent decades developing algorithms and tuning high-performance software packages for this task. This effort has produced several large, mature libraries for solving linear systems at scale on modern supercomputers, including PETSc [5], Trilinos [19], and HYPRE [10].

While these libraries have seen wide adoption across a variety of application domains, their interfaces present challenges for developers who wish to integrate them into complex, distributed scientific applications.

P1: Efficient composition. Existing solver libraries are designed for a bulk-synchronous programming model (MPI) in which the main application pauses its own work while the library performs a solve. This makes it difficult to interleave application work with the execution of the solver, potentially leaving performance on the table. Distributed linear solvers are usually bottlenecked by inter-node communication, so processor cycles are wasted waiting for data to arrive.

P2: Format specificity. Existing solver libraries expect users to supply matrices and vectors using a small collection of predefined storage formats. This often requires relocation of matrix and vector data into library-specific data structures, and it is challenging to implement user-defined storage formats without invasive changes to library code.

P3: Partitioning-awareness. Existing solver libraries place the burden of compatibly partitioning matrix and vector data on the user. Changes to data layout and partitioning strategies often require the user to write or modify their own data movement routines, which impedes prototyping and performance-tuning of complex distributed applications.

P4: Layout flexibility. The matrix and vector data of a linear system may not be naturally co-located in a complex scientific application. For example, a boundary-value problem may involve an interaction between 2D boundary data and 3D interior data from different sources (say, generated by different subroutines). Traditional solver libraries require their users to explicitly reindex and reassemble this data into a single contiguous structure,

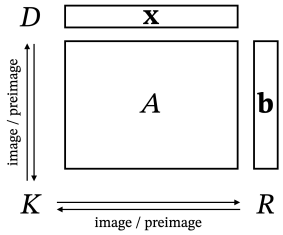

Figure 1: Schematic view of a linear system $A\mathbf{x} = \mathbf{b}$ and its three fundamental index spaces in KDRSolvers: the *kernel space* K, which indexes the nonzero entries of the matrix A, the *domain space* D, which indexes the solution vector $\mathbf{x}$, and *range space* R, which indexes the right-hand side vector $\mathbf{b}$. The structure of the maps $K \rightleftarrows D$ and $K \rightleftarrows R$ is determined by the storage format of the sparse matrix A.

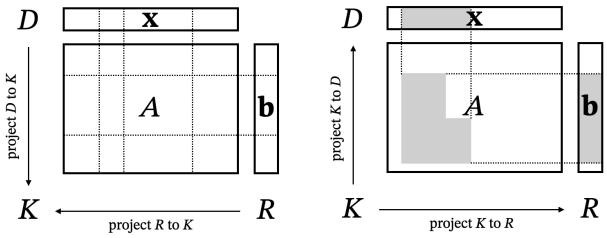

Figure 2: Given a partition of $\mathbf{x}$ or $\mathbf{b}$, KDRSolvers performs a projection along the corresponding map $D \to K$ or $R \to K$ to obtain a compatible partition of K. Conversely, given a partition of K, projection along the maps $K \to D$ and $K \to R$ yields compatible partitions of D or R.

which necessitates expensive data movement and often creates serial bottlenecks [2, 13].

These limitations are well-known to practitioners. Indeed, the US Exascale Computing Project has specifically identified "interleaving of communication and computation," "data-dependent communication patterns," and "assembling matrix and vector objects to define a linear system" as key challenges in the design of exascale sparse solver libraries [2].

We address these challenges by introducing KDRSolvers, a novel framework for representing sparse linear systems and executing Krylov subspace methods on modern heterogeneous (e.g., CPU + GPU) supercomputers. In KDRSolvers, matrix storage formats are represented as a pair of mathematical relations between a *kernel space* K (which indexes the non-zero entries of a sparse matrix), a *domain space* D (the coordinates of the solution vector), and a *range space* R (the coordinates of the right-hand side vector), as shown in Figure 1. KDRSolvers relates these spaces using *dependent partitioning*, a language that facilitates efficient co-partitioning of distributed data structures [17, 18]. This unified abstraction allows KDRSolvers to use *projections* to implement universal co-partitioning operators that are independent of particular matrix storage formats, as illustrated in Figure 2.

KDRSolvers also extends the notion of a linear system by introducing *multi-operator systems* in which a single logical matrix or vector can be composed of multiple non-contiguous, distributed, and possibly aliasing subarrays. To adapt Krylov subspace methods to this more general setting, KDRSolvers decomposes each logical matrix-vector operation into optimized computational kernels that process subarrays in-place. This architecture enables matrix and vector data from multiple sources to be efficiently ingested and processed without amalgamation into a single data structure.

KDRSolvers combines these abstractions to directly address the aforementioned challenges to performance, flexibility, and extensibility that existing solver libraries face.

P1: Efficient composition. KDRSolvers provides a natural mapping of Krylov subspace methods onto a task-oriented runtime system, enabling automatic overlap of communication and computation, in addition to interleaving application and library workloads. The overhead of a task-oriented runtime is hidden in this context by the availability of spare processor cycles during latency-bound solver steps.

P2: Format specificity. Logical regions [6] and dependent partitioning [17, 18] allow KDRSolvers to define universal array co-partitioning operations that automatically extend to user-defined storage formats without any custom data movement subroutines or modification to library code.

P3: Partitioning-awareness. By connecting the kernel, domain, and range spaces of a linear system, KDRSolvers allows a compatible partition of any one of these spaces to be automatically derived from a given partition of any other. Dependent partitioning enables KDRSolvers to automatically propagate these partitions through both user and library code, enabling developers to change partitioning strategies without modifying their code.

P4: Layout flexibility. Multi-operator systems enable KDRSolvers to ingest matrices and vectors in-place, even in multiple non-contiguous pieces, without relocating data into library-specific data structures. This eliminates the need for explicit matrix and vector reassembly in complex scientific applications involving data from multiple sources.

We have implemented the KDRSolvers methodology in a prototype library called LegionSolvers, which targets the Legion [6] distributed task-based runtime system. In experiments using up to 256 nodes and 1,024 GPUs on the Lassen supercomputer, we find that LegionSolvers exhibits performance that is competitive with, and in most cases better than PETSc and Trilinos. We also demonstrate the flexibility of the KDRSolvers framework by using LegionSolvers to implement advanced functionality that is difficult to achieve using existing solver libraries, such as efficiently solving systems with non-contiguous data and dynamic load-balancing.

In summary, the contributions of this paper are:

- A unified description of sparse matrix storage formats in the language of dependent partitioning, allowing matrix-vector co-partitioning operators to be defined and implemented independently of the underlying storage format (Section 3).
- The notion of a multi-operator system, which enables Krylov subspace methods to be executed in-place on non-contiguous distributed matrix and vector data without explicit reassembly (Section 4).
- An implementation of the preceding ideas targeting the Legion runtime system (Section 5) which exhibits competitive

performance and greater flexibility compared to existing solver libraries (Section 6).

2 Background

2.1 Krylov Subspace Methods

Krylov subspace methods (KSMs) are algorithms for solving systems of linear equations of the form $A\mathbf{x} = \mathbf{b}$ that take an initial guess $\mathbf{x}_0$ and produce a sequence of approximations $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots$ that converge to an exact solution. Well-known examples of KSMs include the conjugate gradient method (CG) [11], the biconjugate gradient method (BiCG) and its stabilized form (BiCGStab) [20], the minimum residual method (MINRES) [14], and its generalized form (GMRES) [15].

KSMs are distinguished from other algorithms for solving linear systems by only accessing the matrix A through a black-box subroutine that, given a vector $\mathbf{v}$, computes the matrix-vector product $A\mathbf{v}$. This abstraction makes KSMs especially suitable for solving *sparse* linear systems, since $A\mathbf{v}$ can be computed in time proportional to the number of nonzero entries in A.

In scientific applications, KSMs are rarely run to full convergence. Instead, computation is terminated as soon as the approximate solution $\mathbf{x}_i$ fulfills some application-specific measure of accuracy. However, convergence of KSMs is very difficult to predict *a priori* from the matrix A. There is usually no principled approach besides trial and error to know which KSM will perform best on a given linear system. Thus, libraries of interchangeable KSMs are important for prototyping scientific applications that perform sparse solves.

2.2 Sparse Distributed Solver Libraries

Traditional distributed solver libraries, including PETSc and Trilinos, expect users to assemble library-specific data structures (e.g., `MatMPIAIJ` in PETSc or `Tpetra::CrsMatrix` in Trilinos) through insertion of non-zero coordinates. These data structures have predefined partitioning schemes specific to the library. For example, PETSc only supports disjoint row-based sparse matrix partitions [4], while Trilinos supports disjoint row- and column-based partitions but does not support more general partitioning strategies that may alias across rows and columns. These pre-defined partitioning strategies limit the flexibility of these libraries when interfacing with external applications, as repartitioning may be needed to match the library.

PETSc and Trilinos operate in the bulk-synchronous MPI programming model. Thus, when an iterative solve is issued to these libraries, they assume exclusive control over a set of computing resources. This property is not ideal, since iterative linear solvers are often bound by communication costs, so an application could potentially overlap independent work with the solve to maximize performance.

3 Sparse Matrix Storage Formats

A sparse matrix consists of a list of numbers augmented with metadata that specifies the placement of each number in a rectangular grid. The first key observation that underlies KDRSolvers is that in any sparse matrix storage format, the fundamental role of this metadata is to specify a relation between positions in a list of numbers, which we call the *kernel space* K, and positions in a rectangular grid, whose axes we call the *domain space* D and *range space* R.

To formalize this notion, let D and R be arbitrary index spaces. (An *index space* is a finite set of identifiers.) A *dense* $R \times D$ *matrix* is a collection of numbers $A = \{A_{ij}\}_{i \in R, j \in D}$ indexed by the Cartesian product $R \times D$ that collectively define a linear transformation $\mathbf{v} \in \mathbb{R}^D \mapsto \mathbf{w} \in \mathbb{R}^R$ as follows:

$$w_i = \sum_{j \in D} A_{ij} v_j \qquad \text{for all } i \in R \tag{1}$$

We call D and R the *domain space* and *range space* of (the linear transformation defined by) the matrix A. A *sparse* $R \times D$ *matrix* consists of a collection of numbers $A = \{A_k\}_{k \in K}$ over a different index space K, called the *kernel space*, together with two binary relations $\text{col} \subseteq K \times D$ and $\text{row} \subseteq K \times R$, called the *column relation* and *row relation*. These collectively define a linear transformation $\mathbf{v} \in \mathbb{R}^D \mapsto \mathbf{w} \in \mathbb{R}^R$ as follows:

$$w_i = \sum_{\substack{k \in K \\ (k,i) \in \text{row}}} \sum_{\substack{j \in D \\ (k,j) \in \text{col}}} A_k v_j \qquad \text{for all } i \in R \tag{2}$$

This definition generalizes equation (1) by including a double sum over related points in the kernel space K and domain space D. In conventional sparse matrix representations, each kernel point $k \in K$ is related to at most one grid position $(i, j) \in R \times D$, which causes this double sum to collapse to a single sum. However, in KDRSolvers, we explicitly allow many-to-many column and row relations, enabling a single stored number to be *aliased* into multiple matrix entries. This more general approach enables data storage and movement optimizations that will be discussed in Section 4.

In Table 3, we show how a variety of storage formats can be reformulated as instances of this definition. For example, the COO matrix format represents a sparse matrix as a collection of triples $\{(A_k, i_k, j_k)\}_{k \in K}$ each consisting of a matrix entry and its row and column coordinates. Since an indexed collection $\{x_k\}_{k \in K}$ is equivalent to a function $K \to X$, we can equivalently think of the matrix entries $\{A_k\}_{k \in K}$, row indices $\{i_k\}_{k \in K}$, and column indices $\{j_k\}_{k \in K}$ as specifying three functions: $\text{entry} : K \to \mathbb{R}$, $\text{row} : K \to R$, and $\text{col} : K \to D$. These constitute the entry function, row relation, and column relation of the COO storage format.

Some storage formats impose *structural assumptions* on the form of the index spaces (K, D, R). For example, the CSR format requires its kernel space K to be totally ordered so that its row relation can relate points in R to contiguous intervals in K. As another example, dense matrices carry the structural assumption that K is the full Cartesian product $R \times D$. The row and column relations are simply the canonical projection functions $\pi_1 : R \times D \to R$ and $\pi_2 : R \times D \to D$, so no additional stored data is required to describe them. Thus, dense matrices in KDRSolvers consist of a structural assumption paired with an empty data structure.

Note that our abstract notion of a sparse matrix storage format does not necessarily specify a physical data layout. For example, a COO matrix consists of an indexed collection of records with three fields $\{\text{entry} : K \to \mathbb{R}, \text{col} : K \to D, \text{row} : K \to R\}$, which can be laid out as an array-of-structures or a structure-of-arrays. Indeed, our implementation (which we describe in Section 5) handles both of these layouts.

Format	Structural Assumptions	Column Relation	Row Relation
Dense	$K = R \times D$	$\pi_2 : R \times D \to D$ (implicit)	$\pi_1 : R \times D \to R$ (implicit)
COO	(none)	$\mathsf{col} : K \to D$	$\mathsf{row} : K \to R$
CSR	K is totally ordered	$\mathsf{col} : K \to D$	$\mathsf{rowptr} : R \to [K, K]$
CSC	K is totally ordered	$\mathsf{colptr} : D \to [K, K]$	$\mathsf{row} : K \to R$
ELL	$K = R \times K_0$	$\mathsf{col} : K \to D$	$\pi_1 : R \times K_0 \to R$ (implicit)
ELL′	$K = D \times K_0$	$\pi_1 : D \times K_0 \to D$ (implicit)	$\mathsf{row} : K \to R$
DIA	$D = \{1, \ldots, d\}$ $R = \{1, \ldots, r\}$ $K = K_0 \times \{1, \ldots, d\}$ $\mathsf{offset} : K_0 \to \mathbb{Z}$	$\mathsf{col} : (k_0, i) \mapsto i$ (implicit)	$\mathsf{row} : (k_0, i) \mapsto i - \mathsf{offset}(k_0)$ (implicit)
BCSR	$K = K_0 \times B_R \times B_D$ $D = D_0 \times B_D$ $R = R_0 \times B_R$ K_0 is totally ordered	$\mathsf{col} : K_0 \to D_0$	$\mathsf{rowptr} : R_0 \to [K_0, K_0]$
BCSC	$K = K_0 \times B_R \times B_D$ $D = D_0 \times B_D$ $R = R_0 \times B_R$ K_0 is totally ordered	$\mathsf{colptr} : D_0 \to [K_0, K_0]$	$\mathsf{row} : K_0 \to R_0$

Figure 3: A variety of sparse matrix storage formats are realized in KDRSolvers as particular forms of column and row relations combined with structural assumptions on the index spaces (K, D, R). Relations marked as (implicit) can be specified without additional metadata as a consequence of structural assumptions. Here, π_i denotes the function that maps a Cartesian product onto its ith coordinate. For a totally ordered index space K, we denote by $[K, K]$ the set of contiguous intervals in K.

3.1 Partitioning Sparse Matrices

Having formalized matrix storage formats as pairs of row and column relations, we are now prepared to develop a general notion of partitioning sparse matrix data. A *partition* of an index space I is a function $P : C \to 2^I$ that maps elements of a finite set C to subsets of I. We call C the *color space* of the partition P, and we say that a point $i \in I$ is assigned a color $c \in C$ by P if $i \in P(c)$. We say P is *complete* if it assigns every point in I at least one color; it is *disjoint* if no point in I is assigned multiple colors.

The notion of partitioning in KDRSolvers is sufficiently general to developed in any parallel programming environment that provides a distributed indexed collection (e.g., a distributed array). In particular, we assume the following capability: given a collection $X = \{x_i\}_{i \in I}$ over an index space I and a partition $P : C \to 2^I$, the collection X can be split into a family of disjoint data structures $\{X_c\}_{c \in C}$ where each member X_c represents the collection $\{x_i : i \in P(c)\}$. Various parallel programming systems, e.g., Legion (`Legion::IndexPartition`) and Chapel (`Chapel::Distribution`), provide data structures for this purpose.

This capability prescribes a natural way to partition the entries of a sparse matrix: given a partition of the kernel space K, the indexed collection $\{A_k\}_{k \in K}$ of nonzero matrix entries can be split accordingly. The remaining question is how to split the metadata that specify the column and row relations. To do this, we introduce *images* and *preimages*, dependent partitioning operatoions that allow us to derive partitions of D and R from partitions of K, and vice versa.

Given a relation $R \subseteq I \times J$ between two index spaces I and J and a partition $P : C \to 2^I$, the *image* of P along R is the partition $Q : C \to 2^J$ defined by:

$$Q(c) \coloneqq \{j \in J \mid \exists i \in I : i \in P(c) \wedge (i, j) \in R\} \tag{3}$$

Conversely, given a partition $Q : C \to 2^J$, the *preimage* of Q along R is the partition $P : C \to 2^I$ defined by:

$$P(c) \coloneqq \{i \in I \mid \exists j \in J : j \in Q(c) \wedge (i, j) \in R\} \tag{4}$$

We collectively refer to these two operations as *projection* along the relation R. When R is a function $f : I \to J$, these notions reduce to the familiar concepts of *images* $f[P(c)]$ and *preimages* $f^{-1}[Q(c)]$ of the sets $P(c)$ and $Q(c)$ under f.

These projection operations simplify dependence analysis in task-oriented runtime systems and can be efficiently implemented without any structural assumptions about the underlying index spaces [17, 18]. Thus, they are applicable to any storage format. In particular, there are four important projections furnished by the row and column relations:

- $\mathsf{col}_{K \to D}[P]$: project a partition P of the kernel space K along col to yield a partition of D.
- $\mathsf{row}_{K \to R}[P]$: project a partition P of the kernel space K along row to yield a partition of R.
- $\mathsf{col}_{D \to K}[Q]$: project a partition Q of the domain space D along col to yield a partition of K.
- $\mathsf{row}_{R \to K}[Q]$: project a partition Q of the range space R along row to yield a partition of K.

These projection operators compute the set of elements needed to compute a particular piece of a matrix-vector product. For example, suppose we are given a partition $P : C \to 2^R$ of the range space R. For each color $c \in C$, the piece $\mathbf{y}_c$ of the matrix-vector product $\mathbf{y} \coloneqq A\mathbf{x}$ depends only on the matrix piece A_c in

$\text{row}_{R\to K}[P]$ and the vector piece $\mathbf{x}_c$ in $\text{col}_{K\to D}[\text{row}_{R\to K}[P]]$. Indeed, $\text{col}_{K\to D}[\text{row}_{R\to K}[P]]$, is the finest partition from which the pieces $\mathbf{y}_c$ of $\mathbf{y}$ can be independently computed. As another example,

$$\text{col}_{K\to D}[\text{row}_{R\to K}[\text{col}_{K\to D}[\text{row}_{R\to K}[P]]]] \tag{5}$$

yields the finest partition of D needed to compute $A^2\mathbf{x}$.

In parallel programming environments such as Legion [6] that implement projections over arbitrary index spaces, these co-partitioning operators can be uniformly applied to a variety of sparse matrix storage formats. Thus, by explicitly formalizing row and column relations as part of the library interface, the KDRSolvers framework enables data-dependent communication and partitioning operations to share a universal implementation for all storage formats, including user-defined formats, with computable row and column relations.

4 Multi-Operator Systems

The second key idea of the KDRSolvers framework is the notion of a *multi-operator system*, which allows multiple distributed matrices and vectors in a mixture of storage formats to form one logical linear system without physically amalgamating their underlying data into a single structure.

A linear system $A\mathbf{x} = \mathbf{b}$ involves a matrix A that relates a solution vector $\mathbf{x} \in \mathbb{R}^D$, indexed by the domain space D of A, to a right-hand-side vector $\mathbf{b} \in \mathbb{R}^R$, indexed by its range space R. In KDRSolvers, we generalize this notion to allow a *multi-component vector* $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ indexed by a sequence of domain spaces $(D_1, \ldots, D_n)$, whose components collectively form a single logical solution vector $\mathbf{x}_{\text{total}} \in \mathbb{R}^{D_{\text{total}}}$ indexed by the *total domain space* $D_{\text{total}} = D_1 \sqcup \cdots \sqcup D_n$. We extend the same treatment to right-hand-side vectors, allowing a sequence $(\mathbf{b}_1, \ldots, \mathbf{b}_m)$ indexed by range spaces $(R_1, \ldots, R_m)$ whose disjoint union is the *total range space* R_{total}.

To define linear maps between multi-component vectors, we introduce the notion of a *multi-component operator*

$$\{(K_1, A_1, i_1, j_1), \ldots, (K_N, A_N, i_N, j_N)\} \tag{6}$$

where each A_ℓ is a sparse matrix over $(K_\ell, D_{i_\ell}, R_{j_\ell})$ that relates one domain space D_{i_ℓ} to one range space R_{j_ℓ}. This shares some similarities with the familiar notion of a block linear system, which contains one matrix relating each pair of domain and range spaces:

$$\begin{bmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{m1} & \cdots & A_{mn} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_n \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \vdots \\ \mathbf{b}_m \end{bmatrix} \tag{7}$$

However, the KDRSolvers notion of a multi-component operator is more general in that it allows any number of matrices to relate a single pair (D_i, R_j) of domain and range spaces. These matrices may overlap and alias each other in arbitrary patterns, as shown in Figure 4, allowing memory to be reused in repeated submatrices.

Formally, we define the linear transformation $A_{\text{total}} : \mathbb{R}^{D_{\text{total}}} \to \mathbb{R}^{R_{\text{total}}}$ associated to $\{(K_1, A_1, i_1, j_1), \ldots, (K_N, A_N, i_N, j_N)\}$ as follows: for each $j \in \{1, \ldots, m\}$, the component $\mathbf{w}_j \in \mathbb{R}^{R_j}$ of the output vector $\mathbf{w} = A\mathbf{v}$ is defined as

$$\mathbf{w}_j = \sum_{\ell : j_\ell = j} A_\ell \mathbf{v}_{i_\ell} \tag{8}$$

with an empty sum treated as the zero vector $\mathbf{0} \in \mathbb{R}^{R_j}$.

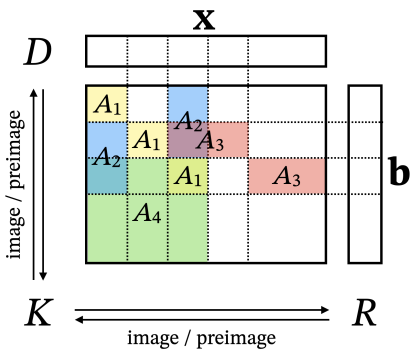

Figure 4: Operators in a multi-operator system can arbitrarily alias and overlap each other, with sums implicitly taken in overlapping areas. Repeated submatrices, such as A_1, are transmitted and stored only once.

4.1 Executing KSMs on Multi-Operator Systems

All Krylov subspace methods (KSMs) take a matrix A, an initial solution vector $\mathbf{x}$, and a right-hand-side vector $\mathbf{b}$, and execute a sequence of the following operations:

- arithmetic operations on scalars, $\alpha \in \mathbb{R}$;
- scalar-vector multiplication, $\mathbf{w} \leftarrow \alpha\mathbf{v}$;
- vector-vector addition, $\mathbf{w} \leftarrow \mathbf{u} + \mathbf{v}$;
- vector-vector inner products, $\alpha \leftarrow \mathbf{v} \cdot \mathbf{w}$;
- matrix-vector multiplication, $\mathbf{w} \leftarrow A\mathbf{v}$;
- adjoint matrix-vector multiplication, $\mathbf{w} \leftarrow A^*\mathbf{v}$;

KSMs produce vectors belonging to the *Krylov subspaces* generated by repeatedly applying A and its adjoint A^* to the initial vectors $\mathbf{x}$ and $\mathbf{b}$. Note that all vectors produced in this fashion lie in $\mathbb{R}^{D_{\text{total}}}$ or $\mathbb{R}^{R_{\text{total}}}$.

To execute a KSM on a multi-operator system, we must perform these operations on multi-component vectors and operators without explicitly assembling their components. Scalar arithmetic operations need no modification, while scalar-vector multiplication and vector-vector addition can be executed independently on each component. Thus, to perform $\mathbf{w} \leftarrow \mathbf{u}+\mathbf{v}$, we perform n independent addition operations $\mathbf{w}_i \leftarrow \mathbf{u}_i + \mathbf{v}_i$. To compute an inner product, we execute a sum-reduction across the components $\mathbf{v}_i \cdot \mathbf{w}_i$. To perform matrix-vector multiplication $\mathbf{y} \leftarrow A_{\text{total}}\mathbf{x}$, we first set $\mathbf{y} \leftarrow \mathbf{0}$. Then, for each quadruple $(K_\ell, A_\ell, i_\ell, j_\ell)$, we perform a matrix multiply-add of the form $\mathbf{y}_{i_\ell} \leftarrow A_\ell \mathbf{x}_{j_\ell} + \mathbf{y}_{i_\ell}$.

Decomposing a single logical operation into a collection of single-component operations is especially suitable for task-oriented programming systems, since an optimized computational kernel can be dispatched for every combination of matrix and vector storage formats. Formulating a problem as a multi-operator system also reveals implicit parallelism as data from one component can be processed while waiting on other components to arrive. Interference analysis must be performed to ensure that simultaneous multiply-add operations $\mathbf{y}_{i_\ell} \leftarrow A_\ell \mathbf{x}_{j_\ell} + \mathbf{y}_{i_\ell}$ do not write into the same component $\mathbf{y}_i$, but the results of this analysis can be cached and reused in future iterations to reduce overhead [12].

4.2 Applications of Multi-Operator Systems

Multi-operator systems provide a natural generalization of several advanced features provided by other solver libraries for solving multiple similar linear systems at once. These are called "application-aware solvers" in Trilinos [7] and are unsupported in PETSc (see Section 2.3.2 of [4]).

Multiple right-hand sides. Some applications require the solution of a collection of linear systems of the form

$$A\mathbf{x}_1 = \mathbf{b}_1 \quad A\mathbf{x}_2 = \mathbf{b}_2 \quad \cdots \quad A\mathbf{x}_n = \mathbf{b}_n \tag{9}$$

where each equation involves the same matrix A but different right-hand side vectors $\mathbf{b}_i$. This can be expressed as a multi-operator system of the following form:

$$\{(K, A, 1, 1), (K, A, 2, 2), \ldots, (K, A, n, n)\} \tag{10}$$

Here, the aliasing capability of multi-operator systems is crucial to avoid needless n-fold duplication of the matrix A. Instead of explicitly forming a large block diagonal matrix containing n independent copies of A, the level of indirection furnished by multi-operator systems allows a solver library to reuse the physical memory backing a single instance of A.

Related systems. Some applications require the solution of a collection of linear systems of the form

$$(A_0 + \Delta A_1)\mathbf{x}_1 = \mathbf{b}_1 \quad \cdots \quad (A_0 + \Delta A_n)\mathbf{x}_n = \mathbf{b}_n \tag{11}$$

where each equation involves a matrix that differs from a common base matrix A_0 by small perturbation ΔA_i that only modifies a small number of entries. This can be expressed as a multi-operator system of the following form:

$$\begin{gathered}\{(K_0, A_0, 1, 1), (K_1, \Delta A_1, 1, 1),\\ \vdots \\ (K_0, A_0, n, n), (K_n, \Delta A_n, n, n)\}\end{gathered} \tag{12}$$

As before, the multi-operator system abstraction requires only a single copy of the base matrix A_0 to be stored.

5 Implementation

We develop an implementation of the KDRSolvers methodology that targets the Legion runtime system [6, 8], which we call *LegionSolvers*. Legion is a natural fit for KDRSolvers because it provides a distributed task-oriented runtime with native support for multi-dimensional index spaces and highly efficient implementations of projection operations [17]. As a natural consequence, LegionSolvers inherits the benefits of a task-oriented runtime environment, including automated data movement, overlapping communication with computation, and interleaving solver tasks with other work. The task abstraction allows LegionSolvers to support custom computational kernels for user-defined storage formats and matrix-free operations with no modification to library code.

LegionSolvers is implemented as a C++ library with Legion as its only necessary dependency. It uses C++ templates to implement KSMs generically with respect to numeric data types (`float`, `double`, etc.) and index types (signed/unsigned, 32/64-bit, etc.). Optional dependencies on cuBLAS, cuSPARSE, and Kokkos [9] provide GPU acceleration.

LegionSolvers provides two user-facing components: a *planner*, which is used to set up a multi-operator system together with a data partitioning strategy, and a *solver*, which implements a KSM in terms of mathematical operations provided by the planner. This separation between planner and solver allows solver implementations to be written with no awareness of storage formats, multiple operators, or data movement, enabling users of LegionSolvers to independently prototype solver algorithms and data partitioning strategies. Our implementation provides a variety of solvers that implement commonly-used KSMs, including CG, BiCGStab, and GMRES, all with a common interface that allows drop-in replacement. LegionSolvers also exposes all necessary facilities for users to implement their own solvers.

The tables in Figures 5 and 6 describe the planner APIs for problem setup and executing matrix-vector operations. The user is first expected to call `add_sol_vector` and `add_rhs_vector` in order to supply pieces of the initial solution vector $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ and the right-hand-side vectors $(\mathbf{b}_1, \ldots, \mathbf{b}_m)$. The domain and range spaces $D_{\text{total}} = D_1 \sqcup \cdots \sqcup D_n$ and $R_{\text{total}} = R_1 \sqcup \cdots \sqcup R_m$ are then inferred from the underlying index spaces of $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ and $(\mathbf{b}_1, \ldots, \mathbf{b}_m)$.

In addition to the implicit parallelism furnished by multi-operator systems, LegionSolvers also allows the user to explicitly specify additional parallelism by optionally providing a *canonical partition* for each domain space D_i and range space R_j when calling `add_sol_vector` and `add_rhs_vector`. Each canonical partition is required to be complete and disjoint, and is used to further subdivide each linear algebra task into subtasks using Legion's index task launch mechanism [16]. The experiments presented in Section 6 demonstrate the efficiency and scalability of both types of parallelism, used independently and in concert.

After constructing the total domain and range spaces, the user then calls `add_operator` to supply components $(K_\ell, A_\ell, i_\ell, j_\ell)$ of the matrix A_{total}, as defined in Section 4. They may optionally call `add_preconditioner` to supply components $(K'_\ell, P_\ell, i_\ell, j_\ell)$ of a *preconditioner*, i.e., a matrix P_{total} such that $P_{\text{total}}A_{\text{total}}$ is approximately the identity matrix. If supplied, a precondtioner can be used to accelerate the convergence of many KSMs.

The code listing in Figure 7 illustrates the use of the planner to implement the conjugate gradient method (CG). In LegionSolvers, a solver is any C++ object that can be constructed from (a mutable reference to) a planner and exposes a `step()` method. Solvers can optionally expose a `get_convergence_measure()` method that returns a scalar that describes the progress of the solver, such as the norm of the residual $\|A\mathbf{x} - \mathbf{b}\|$. If provided, this is used to repeatedly call `step()` until it falls below a user-provided threshold.

6 Experiments

6.1 Library Performance Comparisons

To evaluate the performance of LegionSolvers, we ran benchmarks against two mature distributed solver libraries[1] optimized for high-performance scientific computing: PETSc [5] and Trilinos [19]. We compare the current development version of LegionSolvers

[1]We do not compare to HYPRE [10], which exclusively focuses on algebraic multigrid methods rather than general Krylov methods.

```
sol_id planner.add_sol_vector(region, field, [partition])
rhs_id planner.add_rhs_vector(region, field, [partition])
  void planner.add_operator(linear_operator, sol_id, rhs_id)
  void planner.add_preconditioner(linear_operator, sol_id, rhs_id)
```

Figure 5: User-facing planner operations for describing a multi-operator system. Square brackets denote optional arguments.

`bool planner.is_square()`	return $D_i = R_i \; \forall i$
`bool planner.has_preconditioner()`	return $P_{\text{total}} \neq 0$
`vec_id planner.allocate_workspace_vector([SOL | RHS])`	
`void planner.copy(dst : vec_id, src : vec_id)`	dst $\leftarrow$ src
`void planner.scal(dst : vec_id, alpha : scalar)`	dst $\leftarrow \alpha \cdot$ dst
`void planner.axpy(dst : vec_id, alpha : scalar, src : vec_id)`	dst $\leftarrow$ dst $+ \alpha \cdot$ src
`void planner.xpay(dst : vec_id, alpha : scalar, src : vec_id)`	dst $\leftarrow$ src $+ \alpha \cdot$ dst
`future planner.dot_product(v : vec_id, w : vec_id)`	return $\mathbf{v} \cdot \mathbf{w}$
`void planner.matmul(dst : vec_id, src : vec_id)`	dst $\leftarrow A($src$)$
`void planner.psolve(dst : vec_id, src : vec_id)`	dst $\leftarrow P($src$)$

Figure 6: Solver-facing planner operations for implementing KSMs. Square brackets denote optional arguments.

```cpp
template <typename ENTRY_T>
class CGSolver {
    LegionSolvers::Planner<ENTRY_T> &planner;
    static constexpr vec_id SOL = 0;
    static constexpr vec_id RHS = 1;
    vec_id P, Q, R;
    Scalar<ENTRY_T> res; // squared residual
public:
    explicit CGSolver(Planner<ENTRY_T> &planner)
        : planner(planner) {
        assert(planner.is_square());
        assert(!planner.has_preconditioner());
        P = planner.allocate_workspace_vector();
        Q = planner.allocate_workspace_vector();
        R = planner.allocate_workspace_vector();
        planner.copy(P, RHS);
        planner.copy(R, RHS);
        res = planner.dot(R, R);
    }
    void step() {
        planner.matmul(Q, P);
        Scalar<ENTRY_T> p_norm = planner.dot(P, Q);
        planner.axpy(SOL, res / p_norm, P);
        planner.axpy(R, -res / p_norm, Q);
        Scalar<ENTRY_T> new_res = planner.dot(R, R);
        planner.xpay(P, new_res / res, R);
        res = new_res;
    }
    Scalar<ENTRY_T> get_convergence_measure() const {
        return res;
    }
}; // class CGSolver<ENTRY_T>
```

Figure 7: C++ implementation of CG in LegionSolvers.

to PETSc 3.18.1.1 and Trilinos (Tpetra/Belos) 14.0 on the Lassen supercomputer, which provides 40 POWER9 CPU cores and 4 NVIDIA V100 GPUs per node.

We selected three widely-used GPU-compatible KSMs from each library: CG, BiCGStab, and GMRES. Each KSM was run on the following families of double-precision linear systems $A\mathbf{x} = \mathbf{b}$ constructed via finite-element analysis of Poisson's equation $\Delta u = f$ on Cartesian meshes:

- 3-point stencil for the 1D Laplacian operator
- 5-point stencil for the 2D Laplacian operator
- 7-point stencil for the 3D Laplacian operator
- 27-point stencil for the 3D Laplacian operator

We constructed each problem family in sizes ranging from 2^{24} unknowns to $2^{32} \approx 4$ billion unknowns, stepping in powers of two. To ensure fair comparison, all matrices were stored in CSR format using an identical row-based partitioning strategy, as this is the only GPU-accelerated sparse matrix storage format supported by PETSc (and one of two formats supported by Trilinos). We note that GPUs increasingly provide the vast majority of computational horsepower in modern supercomputers. For example, Lassen provides 22,192 peak GPU TFLOPs compared to only 850 CPU TFLOPs.

The results of this experiment are reported in Figure 8, where each subplot in the 4×3 grid corresponds to a choice of stencil matrix (3-point 1D, 5-point 2D, 7-point 3D, 27-point 3D) and KSM (CG, BiCGStab, GMRES). In each subplot, we report average execution time per iteration as a function of problem size, measured as the minimum of 3 consecutive runs for each system, where each run consists of 20 warmup iterations followed by 200 timed iterations. In all cases, we see that LegionSolvers performs comparably to PETSc and Trilinos, with notable performance leadership in many runs of CG and GMRES[2].

Note that the logarithmic horizontal axis implies that the three rightmost points in each subplot span 75% of all problem sizes that fit in memory. In other words, on problem sizes that utilize any significant fraction of the machine, the performance of LegionSolvers is characterized by the last few points in each graph. The execution time of LegionSolvers on small problems is dominated by fixed overheads; at these scales, there is not enough work to hide the the scheduling costs of a dynamic runtime system such as Legion. However, we note that the penalty is not large and the absolute running times for these problem sizes are small. On larger problem sizes, LegionSolvers generally pulls ahead, achieving a geometric mean 9.6% improvement over Trilinos and a 5.4% improvement over PETSc on the three largest problem sizes shown in each subplot.

6.2 Multi-Operator System Performance

To evaluate the runtime cost of the multi-operator system abstraction, we performed an experiment in which we used LegionSolvers to solve the same sequence of problems twice (5-point Laplacian

[2]PETSc is not benchmarked on GMRES because LegionSolvers and Trilinos implement a static GMRES(10) restart schedule, while PETSc offers only a dynamic restart schedule that short-circuits some GMRES iterations.

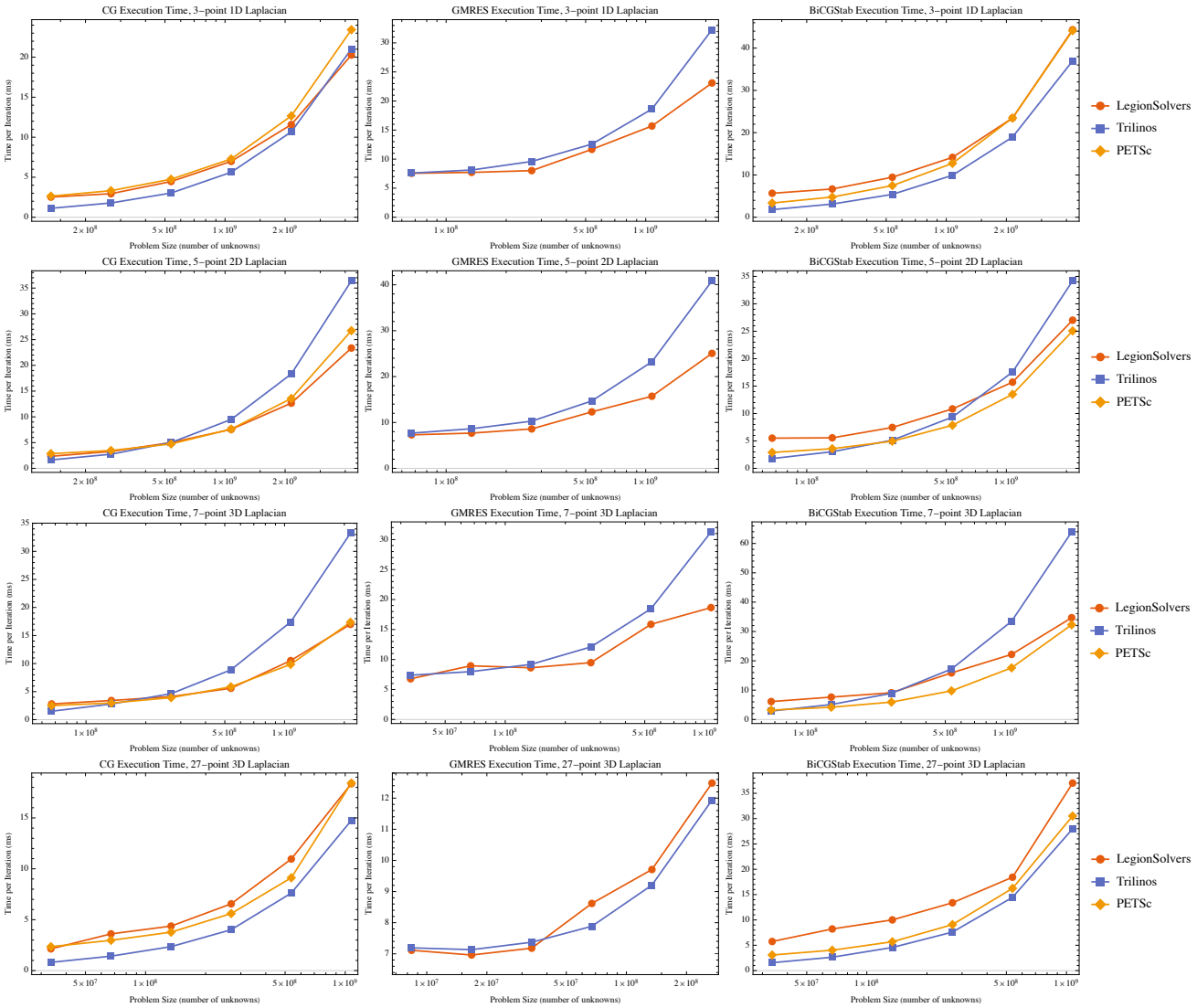

Figure 8: Performance of KSMs (CG, BiCGStab, GMRES) applied to various stencil matrices (3-point 1D, 5-point 2D, 7-point 3D, 27-point 3D) on 16 nodes of the Lassen supercomputer (64 GPUs) as implemented in LegionSolvers, PETSc, and Trilinos. Note that PETSc is not benchmarked on GMRES because it implements a different restart policy than LegionSolvers and Trilinos.

stencil on $2^n \times 2^n$ grid), with each problem formulated in two different ways:

- first, as a single-operator system using only one domain space D to represent the entire grid;
- second, as a multi-operator system using two domain spaces, D_1 and D_2, each representing half of the grid.

Note that this is a square system, so in each case, the range spaces $\{R_i\}$ are identical to the domain spaces $\{D_i\}$. In the first case, we formulate the problem using a single sparse matrix $A : \mathbb{R}^D \to \mathbb{R}^D$ stored in CSR format, while in the second case, we use four CSR matrices:

- two self-interaction matrices $A_{11} : \mathbb{R}^{D_1} \to \mathbb{R}^{D_1}$ and $A_{22} : \mathbb{R}^{D_2} \to \mathbb{R}^{D_2}$ that compute the Laplacian within each domain;
- two boundary-interaction matrices $A_{12} : \mathbb{R}^{D_1} \to \mathbb{R}^{D_2}$ and $A_{21} : \mathbb{R}^{D_2} \to \mathbb{R}^{D_1}$ that compute the Laplacian across the boundary.

In Figure 9, we plot the average execution time per iteration of LegionSolvers's implementation of BiCGStab running on both the single-operator and multi-operator formulations of this problem. For small problem sizes under 10^9 unknowns, the multi-operator system is slower due to fixed task launch overhead costs present in Legion. However, at larger problem sizes, the multi-operator system becomes faster as computation of the self-interaction terms can be overlapped with the communication of boundary terms.

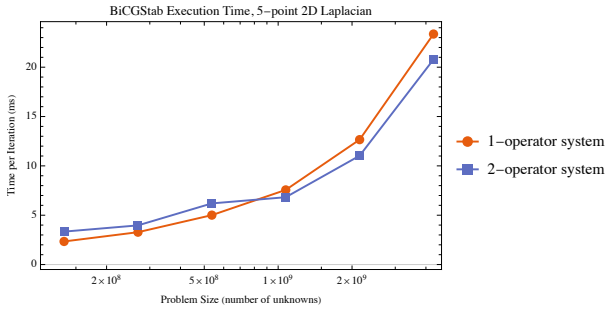

Figure 9: Execution time per iteration of BiCGStab using a 5-point Laplacian stencil on a $2^n \times 2^n$ grid, measured as a function of n, formulated as either a single-operator system (red) or a multi-operator system (blue).

6.3 Dynamic Load Balancing

Finally, we perform an experiment that demonstrates two novel capabilities of LegionSolvers: interleaving solver work with other application work, and dynamically rebalancing the task mapping of a KSM in response to changing external loads. These capabilities are difficult to achieve in MPI-based libraries, such as PETSc and Trilinos.

In this experiment, we run CG using a 5-point 2D Laplacian stencil on 32 Lassen nodes, each equipped with 40 POWER9 CPU cores. Each CPU also simultaneously runs a background task that occupies some of its cores. After every 100^{th} CG iteration, the number of cores occupied by each background task is uniformly randomized in the range $[0, 39]$. This stochastic background load is a simple proxy for a multiphysics application that performs local and global work in lockstep (e.g., a global pressure solve and local chemical reaction simulation).

We construct a multi-operator system by subdividing a $2^{16} \times 2^{16}$ grid into 64 domain pieces $\{D_1, \ldots, D_{64}\}$ and cutting the stencil matrix into 64×64 tiles $\{A_{1,1}, \ldots, A_{64,64}\}$. We assign each of our 32 nodes two domain pieces $\{D_{2i-1}, D_{2i}\}$ and 2×64 matrix tiles $\{A_{2i-1,1}, \ldots, A_{2i-1,64}, A_{2i,1}, \ldots, A_{2i,64}\}$. We then compare the performance of LegionSolvers on this multi-operator system using two different Legion mappers: one that maintains this static assignment of matrix tiles to nodes, and one that dynamically rebalances the assignment of matrix tiles to nodes after every 10^{th} CG iteration. In both cases, each matrix tile $A_{i,j}$ is always physically located on either the node that owns the input domain piece D_j or the output domain piece D_i, and each matrix-vector multiplication task $\mathbf{y}_i \leftarrow A_{i,j}\mathbf{x}_j$ is executed by the node that currently owns $A_{i,j}$.

We adopt the following thermodynamic load-balancing strategy: after every 10^{th} CG iteration, each node i compares its own CG execution time T_i to a precomputed reference value T_0 representing execution time with an average background load (i.e., with 20 cores occupied). If $T_i > T_0$, node i gives away each matrix tile it currently owns with probability $\min(e^{\beta(T_i - T_0)}, 1)$, where the parameter $\beta = 10^{-3}\ \text{ms}^{-1}$ controls the rate of adaptation. Each matrix tile $A_{i,j}$ only has two potential owners in this setup, so the target node of each giveaway is uniquely determined, and no global communication is involved.

The results of this experiment are presented in Figure 10, which plots the execution time of each CG iteration in a single run on a $2^{16} \times 2^{16}$ grid. The red and blue series show execution time with and without dynamic load-balancing, respectively. The red points are occasionally higher than the blue points, indicating that the dynamic load-balancing strategy sometimes produces worse task mappings, but this condition never persists for more than 10 iterations. In total, using dynamic load-balancing results in a 66% reduction of total execution time.

Note that the goal of this experiment is not to measure the absolute performance of LegionSolvers, but to demonstrate capabilities of LegionSolvers that are difficult to achieve using other libraries. It uses CPU matrix multiplication kernels and does not use Legion optimizations, such as control replication and dynamic tracing, that are employed in other experiments. We make no claim that this particular load-balancing strategy is optimal; in general, different background loads will require different load-balancing strategies and rates of adaptation.

7 Related and Future Work

Iterative Linear Solvers in Task-Based Systems. Agullo et al. [1] discuss the process of developing and optimizing conjugate gradient solvers in StarPU [3], a task-based runtime system similar to Legion. StarPU does not currently support dependent partitioning projection operations, so the optimization efforts of Agullo et al. involve manually computing an image-like operation to limit the amount of communication performed by the runtime. KDRSolvers uses dependent partitioning to represent a variety of sparse matrix formats and implement many KSMs beyond CG.

Dependent Partitioning in Distributed Tensor Computations. SpDISTAL [21] is a compiler for distributed sparse tensor arithmetic that also applies the ideas of dependent partitioning to sparse arrays in a more limited fashion than KDRSolvers, supporting only the CSR and CSC formats (and higher-dimensional variants). To our knowledge, the use of row and column relations in their full generality, particularly as a library interface applicable to user-defined storage formats, is unique to KDRSolvers. Systems like SpDISTAL could be integrated into KDRSolvers to synthesize optimized computational kernels for operations required by new solvers.

Preconditioning multi-operator systems. Although LegionSolvers is capable of working with a user-provided preconditioning matrix (or matrix-free task), it currently does not implement any algorithms for automatically deriving a preconditioner P from a given matrix A. Extending classical preconditioning algorithms, such as Jacobi preconditioning and successive over-relaxation, to the context of multi-operator systems is an important research problem that we intend to consider in future work.

Mixing and composing sparse array storage formats. Recent work on the SparseTIR tensor compiler [22] shows that some sparse array operations arising in deep learning can be accelerated by decomposing a sparse array into a mixture of storage formats adapted to its local structure. To our knowledge, optimizations of this type have not yet been applied in the context of scientific computing, where solver libraries have traditionally been designed to

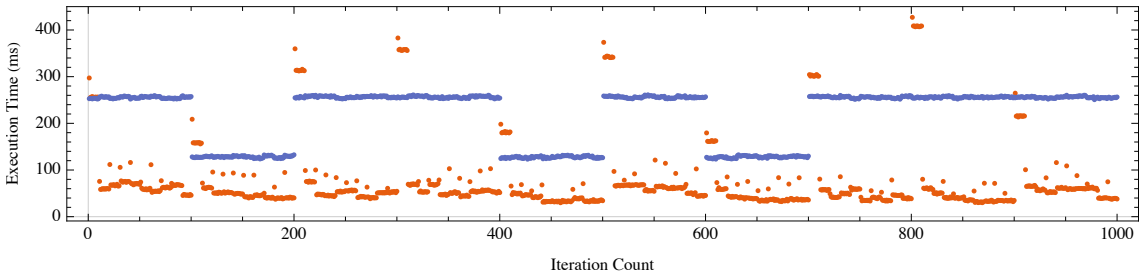

Figure 10: Execution time per iteration of BiCGStab using a 5-point Laplacian stencil on a $2^{16} \times 2^{16}$ grid with (red) and without (blue) dynamic load-balancing.

work with a single sparse array in a single format. However, multi-operator systems allow KDRSolvers to process pieces of a matrix stored in multiple formats within a single linear system, enabling an exciting possibility for future optimization that we intend to explore in future work.

8 Conclusion

We have presented KDRSolvers, a novel methodology for describing sparse linear systems and implementing KSMs on heterogeneous distributed-memory parallel computers. We have described two primary contributions of the KDRSolvers framework: first, a uniform representation of sparse matrix storage formats in terms of abstract maps between index spaces, allowing for the construction of universal co-partitioning operators in the language of dependent partitioning. We have also defined multi-operator systems that increase the flexibility problem setup, enabling the direct consumption of non-contiguous distributed data in addition to reuse and aliasing optimizations. These features directly address significant challenges in the design of exascale-ready sparse solver libraries identified by computational scientists. Finally, we have presented an implementation of the KDRSolvers methodology in the Legion parallel programming system, called LegionSolvers, which exhibits comparable performance to Trilinos and PETSc while offering novel features, such as interleaved execution and dynamic load-balancing against a changing background workload.

Acknowledgments

This work is based upon research supported by the U.S. Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0003968 within the PSAAP III (INSIEME) Program at Stanford University.

References

[1] E Agullo, L Giraud, A Guermouche, S Nakov, and Jean Roman. 2016. *Task-based Conjugate Gradient: from multi-GPU towards heterogeneous architectures.* Research Report RR-8912. Inria. https://hal.inria.fr/hal-01316982

[2] Hartwig Anzt, Erik Boman, Rob Falgout, Pieter Ghysels, Michael Heroux, Xiaoye Li, Lois Curfman McInnes, Richard Tran Mills, Sivasankaran Rajamanickam, Karl Rupp, Barry Smith, Ichitaro Yamazaki, and Ulrike Meier Yang. 2020. Preparing sparse solvers for exascale computing. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378, 2166 (2020), 20190053. arXiv:https://royalsocietypublishing.org/doi/pdf/10.1098/rsta.2019.0053 doi:10.1098/rsta.2019.0053

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198. doi:10.1002/cpe.1631

[4] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, Jacob Faibussowitsch, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, and Junchao Zhang. 2022. *PETSc/TAO Users Manual.* Technical Report ANL-21/39 - Revision 3.18. Argonne National Laboratory.

[5] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil M. Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, Jacob Faibussowitsch, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, and Junchao Zhang. 2022. PETSc Web page. https://petsc.org/. https://petsc.org/

[6] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* IEEE, 1–11. doi:10.1109/SC.2012.71

[7] Eric Bavier, Mark Hoemmen, Sivasankaran Rajamanickam, and Heidi Thornquist. 2012. Amesos2 and Belos: Direct and iterative solvers for large sparse linear systems. *Scientific Programming* 20, 3 (2012), 241–255.

[8] Dan Bonachea and Paul H. Hargrove. 2018. GASNet-EX: A High-Performance, Portable Communication Library for Exascale. In *Proceedings of Languages and Compilers for Parallel Computing (LCPC'18) (Lecture Notes in Computer Science, Vol. 11882).* Springer International Publishing. doi:10.25344/S4QP4W Lawrence Berkeley National Laboratory Technical Report (LBNL-2001174).

[9] H. Carter Edwards and Christian R. Trott. 2013. Kokkos: Enabling Performance Portability Across Manycore Architectures. In *2013 Extreme Scaling Workshop (xsw 2013).* 18–24. doi:10.1109/XSW.2013.7

[10] Robert D. Falgout and Ulrike Meier Yang. 2002. hypre: A Library of High Performance Preconditioners. In *Computational Science — ICCS 2002*, Peter M. A. Sloot, Alfons G. Hoekstra, C. J. Kenneth Tan, and Jack J. Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 632–641.

[11] Magnus R. Hestenes and Eduard Stiefel. 1952. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards* 49 (1952), 409–435. https://api.semanticscholar.org/CorpusID:2207234

[12] Wonchan Lee, Elliott Slaughter, Michael Bauer, Sean Treichler, Todd Warszawski, Michael Garland, and Alex Aiken. 2018. Dynamic tracing: memoization of task graphs for dynamic task-based runtimes. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18).* IEEE Press, Article 34, 13 pages.

[13] Paul T. Lin, Michael A. Heroux, Richard F. Barrett, and Alan B. Williams. 2015. Assessing a mini-application as a performance proxy for a finite element method engineering application. *Concurrency and Computation: Practice and Experience* 27, 17 (2015), 5374–5389. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.3587 doi:10.1002/cpe.3587

[14] C. C. Paige and M. A. Saunders. 1975. Solution of Sparse Indefinite Systems of Linear Equations. *SIAM J. Numer. Anal.* 12, 4 (1975), 617–629. arXiv:https://doi.org/10.1137/0712047 doi:10.1137/0712047

[15] Youcef Saad and Martin H. Schultz. 1986. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM J. Sci. Statist. Comput.* 7, 3 (1986), 856–869. arXiv:https://doi.org/10.1137/0907058 doi:10.1137/0907058

[16] R. Soi, M. Bauer, S. Treichler, M. Papadakis, W. Lee, P. McCormick, A. Aiken, and E. Slaughter. 2021. Index Launches: Scalable, Flexible Representation of Parallel Task Groups. In *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, Los Alamitos, CA, USA, 1–14. doi:10.1145/3458817.3476175

[17] Sean Treichler, Michael Bauer, Rahul Sharma, Elliott Slaughter, and Alex Aiken. 2016. Dependent partitioning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 344–358. doi:10.1145/2983990.2984016

[18] Sean Treichler, Michael Bauer, Rahul Sharma, Elliott Slaughter, and Alex Aiken. 2016. Dependent partitioning. *SIGPLAN Not.* 51, 10 (oct 2016), 344–358. doi:10.1145/3022671.2984016

[19] The Trilinos Project Team. 2020 (acccessed May 22, 2020). *The Trilinos Project Website*. https://trilinos.github.io

[20] H. A. van der Vorst. 1992. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Statist. Comput.* 13, 2 (1992), 631–644. arXiv:https://doi.org/10.1137/0913035 doi:10.1137/0913035

[21] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. SpDISTAL: Compiling Distributed Sparse Tensor Computations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 59, 15 pages.

[22] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 660–678. doi:10.1145/3582016.3582047

Artifact Description (AD)

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

The main software contribution of our paper is a C++ library, called LegionSolvers, that implements Krylov subspace methods (KSMs), i.e., iterative methods for solving sparse linear systems, on top of the Legion parallel programming system. We also present a collection of benchmarking programs that evaluate the performance of LegionSolvers, Trilinos, and PETSc in executing KSMs on modern supercomputers.

- C_1 LegionSolvers: A C++ software library that implements Krylov subspace methods on top of the Legion parallel programming system.
- C_2 A collection of benchmarking programs that compare the rate of execution of Krylov subspace methods in LegionSolvers, Trilinos, and PETSc.

A.2 Computational Artifacts

LegionSolvers is available in the following GitHub repository. We also provide links to subdirectories of this repository that contain the aforementioned Trilinos and PETSc benchmark programs.

- A_1 https://github.com/dzhang314/LegionSolvers
- A_2 https://github.com/dzhang314/LegionSolvers/tree/main/benchmarks/trilinos
- A_3 https://github.com/dzhang314/LegionSolvers/tree/main/benchmarks/petsc

A permanent archive of this repository is also available on Zenodo: https://doi.org/10.5281/zenodo.17088316

These artifacts relate to our contributions as follows:

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Figures 8–10
A_2	C_2	Figure 8
A_3	C_2	Figure 8

B Artifact Identification

B.1 Computational Artifact A_1

Relation To Contributions

Computational artifact A_1 is the LegionSolvers library itself, which implements all aspects of the KDRSolvers framework as described in the main paper.

Expected Results

Expected results from running the LegionSolvers benchmark programs are described in Figures 8–10 in the main paper. It should be found that LegionSolvers performs similarly to or better than PETSc and Trilinos when executing KSMs on problem sizes that occupy a significant fraction of (main or GPU) memory.

Expected Reproduction Time (in Minutes)

- Artifact Setup: Downloading and building LegionSolvers and all of its dependencies can be expected to take 10–20 minutes, depending on the compilers used and the speeds of network and disk accesses. If tests are also built and run for LegionSolvers and all of its dependencies, this estimate increases to roughly 60 minutes.
- Artifact Execution: Each benchmarking run for LegionSolvers takes roughly 10 minutes to scale from the smallest problem size (4096 unknowns) to the maximum problem size that fits into four NVIDIA V100 GPUs (see next section on Artifact Setup for details). Multiplied by 4 different test matrices and 3 different KSMs, this means one benchmarking run takes roughly 120 minutes. This must be repeated for each node count (in our case, scaling from 1 to 256 in powers of two), increasing the total execution time to roughly 960 minutes.
- Artifact Analysis: Execution time is directly measured by the benchmark program `BenchmarkStencil`, and the only analysis step applied to these execution times is to divide by the iteration count and then take the minimum across three independent runs. This trivial arithmetic, together with generating plots of execution time as a function of problem size, can be performed in less than one minute.

These values are all hardware-dependent and should be expected to change on other supercomputers.

Artifact Setup (incl. Inputs)

LegionSolvers supports a wide range of supercomputer architectures by using GASNet-EX to abstract over networks (InfiniBand, OFI, UCX, etc.) and Kokkos to abstract over accelerators (OpenMP, CUDA, HIP, etc.). In principle, LegionSolvers can be run with any network interconnect supported by GASNet 2023.9.0 and any accelerator architecture supported by Kokkos 4.2.1.

In this section, we describe the specific hardware and software configuration present on the Lassen supercomputer at Lawrence Livermore National Laboratory, which was used to produce the results presented in the main paper.

Hardware. Each Lassen node provides 44 POWER9 cores, clocked at 3.45GHz, in a dual-socket configuration. Two cores per socket are reserved for system use, so a total of 40 cores per node are user-accessible. Each node is also equipped with 256GB of main memory and four NVIDIA V100 GPUs, each with 16GB of GPU memory. Nodes are connected with an InfiniBand EDR network. Our experiments use up to 256 nodes, but partial results can be reproduced using a smaller number of nodes.

Software. LegionSolvers depends on Legion, which itself depends on GASNet-EX, CUDA, and Kokkos. We use a specific distribution of GASNet-EX 2023.9.0 provided by the Legion development team.

- GASNet-EX 2023.9.0: https://github.com/StanfordLegion/gasnet/blob/master/GASNet-2023.9.0.tar.gz
- CUDA 11.8.0: https://developer.nvidia.com/cuda-11-8-0-download-archive
- Kokkos 4.2.1: https://github.com/kokkos/kokkos/releases/tag/4.2.01

The experiments described in the main paper were performed using a development version of Legion.

- Legion: development version on `control_replication` branch, commit hash `28db23b4`.
  https://gitlab.com/StanfordLegion/legion/-/commit/28db23b4bc140992f384eb27427dbf597850eb34
- LegionSolvers: development version, `main` branch.
  https://github.com/dzhang314/LegionSolvers/tree/main

Note: As of April 2024, the `control_replication` branch has been merged into `legion/master`. The first release version of Legion that includes the necessary features to support LegionSolvers is Legion 24.03.0.

Datasets / Inputs. No external datasets are needed to run the experiments described in the main paper. All of the sparse matrices used are automatically generated by the benchmark programs at runtime.

Installation and Deployment. To build LegionSolvers on Lassen, we compile host code with GCC 11.2.1 and GPU code with NVCC, as provided in CUDA Toolkit 11.8.0.

LegionSolvers builds are automated using a combination of CMake and Python scripts (CMake 3.23.1, Python 3.7.2). To build GASNet-EX and Kokkos, run `build_dependencies.py`; to build Legion, run `build_legion.py`; and to build LegionSolvers, run `refresh_cmake.py` followed by `make_all.py`. All of these scripts are provided at the root of the LegionSolvers repository. These scripts set the following build flags:

- GASNet-EX:
  `make CONDUIT=ibv`
- Kokkos:
  `CMAKE_CXX_STANDARD=17`
  `Kokkos_ENABLE_SERIAL=ON`
  `Kokkos_ENABLE_OPENMP=ON`
  `Kokkos_ENABLE_CUDA=ON`
  `Kokkos_ENABLE_CUDA_LAMBDA=ON`
  `Kokkos_ENABLE_CUDA_CONSTEXPR=ON`
  `Kokkos_ENABLE_CUDA_LDG_INTRINSIC=ON`
  `Kokkos_ARCH_POWER9=ON`
  `Kokkos_ARCH_VOLTA70=ON`
- Legion:
  `CMAKE_CXX_STANDARD=17`
  `Legion_MAX_NUM_NODES=4096`
  `Legion_MAX_NUM_PROCS=256`
  `Legion_USE_OpenMP=ON`
  `Legion_USE_CUDA=ON`
  `Legion_USE_Kokkos=ON`
  `Legion_NETWORKS=gasnetex`
  `CMAKE_CXX_FLAGS=`
  `  -DREALM_TIMERS_USE_RDTSC=0`
  `CMAKE_CUDA_FLAGS=`
  `  -DREALM_TIMERS_USE_RDTSC=0`

Note that these scripts expect the environment variables `LEGION_SOLVERS_SCRATCH_DIR` and `LEGION_SOLVERS_LIB_PREFIX` to be set to valid directories.

Artifact Execution

Once LegionSolvers and all of its dependencies are built, output binaries will be placed in the directory `${LEGION_SOLVERS_SCRATCH_DIR}/LegionSolversBuild`. The `BenchmarkStencil` executable is the main program used to collect the performance results reported in Figures 8–10.

The details of running `BenchmarkStencil` depend on the job management system used on a particular supercomputer. On Lassen, we run `BenchmarkStencil` using `jsrun` as follows:

```
jsrun --bind none --rs_per_host 1
    --cpu_per_rs 40 --gpu_per_rs 4
    BenchmarkStencil -ll:util 4 -ll:gpu 4
    -ll:csize 240G -ll:fsize 12G
    -lg:eager_alloc_percentage 20
    -dim <dim> -solver <solver>
    -nx <nx> -ny <ny> -nz <nz>
    -it 500 -pt 1 -vp <vp>
```

The parameters `<dim>` and `<solver>` specify the matrix and KSM to be used.

- `dim=1`: 3-point 1D Laplacian stencil
- `dim=2`: 5-point 2D Laplacian stencil
- `dim=3`: 7-point 3D Laplacian stencil
- `dim=4`: 27-point 3D Laplacian stencil
- `solver=1`: Conjugate Gradient (CG)
- `solver=2`: Biconjugate Gradient Stabilized (BiCGStab)
- `solver=3`: Generalized Minimum Residual (GMRES)

The parameters `<nx>`, `<ny>`, and `<nz>` specify the dimensions of the grid, and `<vp>` specifies the number of pieces each matrix and vector should be partitioned into. On Lassen, where each node has four GPUs, `<vp>` should be set equal to four times the node count.

Note that the memory sizes `-ll:csize 240G` and `-ll:fsize 12G` are set slightly smaller than the true memory sizes (256GB and 16GB for core and GPU memory, respectively) because Legion reserves some memory for asynchronous communication. The number of utility processors `-ll:util 4` and the size of the eager allocation

pool `-lg:eager_alloc_percentage 20` were tuned manually for Lassen; other values may be appropriate on other supercomputers.

Running the above `jsrun` command will execute 500 iterations of the specified KSM on the specified matrix, using a fixed right-hand side vector containing entries in the interval $[0, 1]$, and then print a measurement of total execution time to the console. This process is then repeated for each KSM, each matrix, each problem size, and each node count.

Artifact Analysis (incl. Outputs)

For each combination of KSM, matrix, problem size, and node count, we performed three independent runs of the LegionSolvers test program `BenchmarkStencil` and reported the minimum among the three execution times. Then, we plotted execution time per iteration as a function of problem size for each KSM, matrix, and node count.

B.2 Computational Artifact A_2

Relation To Contributions

Computational artifact A_2 is a Trilinos port of the `BenchmarkStencil` test program which was used to produce the comparative benchmarks reported in Figure 8.

Expected Results

The results of running the Trilinos benchmark program are described in Figure 8 in the main paper.

Expected Reproduction Time (in Minutes)

- Artifact Setup: Downloading and building Trilinos can be expected to take roughly 30 minutes, depending on the compilers used and the speeds of network and disk accesses.
- Artifact Execution: Each benchmarking run for Trilinos takes roughly 10 minutes to scale from the smallest problem size (4096 unknowns) to the maximum problem size that fits into four NVIDIA V100 GPUs (see next section on Artifact Setup for details). Multiplied by 4 different test matrices and 3 different KSMs, this means one benchmarking run takes roughly 120 minutes. This must be repeated for each node count (in our case, scaling from 1 to 256 in powers of two), increasing the total execution time to roughly 960 minutes.
- Artifact Analysis: Execution time is directly measured by the benchmark program, and the only analysis step applied to these execution times is to divide by the iteration count and then take the minimum across three independent runs. This trivial arithmetic, together with generating plots of execution time as a function of problem size, can be performed in less than one minute.

These values are all hardware-dependent and should be expected to change on other supercomputers.

Artifact Setup (incl. Inputs)

The hardware and software environments used to execute this artifact were identical to those used for artifact A_1. We used Trilinos release version 14.0.0, available at:

https://github.com/trilinos/Trilinos/releases/tag/trilinos-release-14-0-0

Datasets / Inputs. No external datasets are needed to run the experiments described in the main paper. All of the sparse matrices used are automatically generated by the benchmark programs at runtime.

Installation and Deployment. As with artifact A_1, both Trilinos and our benchmark program were compiled with GCC 11.2.1 and NVCC from CUDA Toolkit 11.8.0. We used CMake scripts provided by Trilinos with the following build options:

```
export TRILINOS_DIR=<...>
export OMPI_CXX=$TRILINOS_DIR/
    packages/kokkos/bin/nvcc_wrapper
export LLNL_USE_OMPI_VARS=Y
export CUDA_LAUNCH_BLOCKING=1
export CUDA_MANAGED_FORCE_DEVICE_ALLOC=1

cmake \
    -DTPL_ENABLE_MPI=ON \
    -DTPL_ENABLE_CUDA=ON \
    -DTrilinos_CUDA_NUM_GPUS=4 \
    -DTrilinos_ENABLE_Tpetra=ON \
    -DTrilinos_ENABLE_Belos=ON \
    -DTrilinos_ENABLE_Ifpack2=ON \
    -DTrilinos_ENABLE_Teuchos=ON \
    -DTrilinos_ENABLE_Kokkos=ON \
    -DTrilinos_ENABLE_FLOAT=ON \
    -DTrilinos_ENABLE_COMPLEX=OFF \
    -DTrilinos_ENABLE_Fortran=OFF \
    -DTrilinos_ENABLE_OpenMP=ON \
    -DTrilinos_ENABLE_CUDA=ON \
    -DKokkos_ENABLE_CUDA=ON \
    -DTPetra_ENABLE_CUDA=ON \
    -DKokkos_ENABLE_Cuda_UVM=ON \
    -DKokkos_ENABLE_Cuda_Lambda=ON \
    -DTpetra_INST_CUDA=ON \
    -DTpetra_INST_OPENMP=ON \
    -DTPL_ENABLE_Matio=OFF \
    ..
```

Here, `TRILINOS_DIR` should be set to the directory where the Trilinos 14.0.0 release archive was decompressed.

After the Trilinos library is built and installed, CMake should also be run in the `benchmarks/trilinos` directory to build the Trilinos benchmark program.

Artifact Execution

Once the Trilinos benchmark program is built, it can be run using `jsrun` as follows:

```
jsrun -M "-gpu" --rs_per_host 4 \
    --cpu_per_rs 10 --gpu_per_rs 1 \
    TrilinosSolverBenchmark \
    <matrix> <ksm> <nx> <ny> <nz>
```

Note that the command-line options differ from the LegionSolvers benchmark program; instead of using numeric codes, the KSM and matrix are directly specified using the strings `1D`, `2D`, `3D`,

3D27, and CG, BiCGStab, and GMRES. As before, <nx>, <ny>, and <nz> specify the grid size.

Artifact Analysis (incl. Outputs)

For each combination of KSM, matrix, problem size, and node count, we performed three independent runs of the Trilinos test program and reported the minimum among the three execution times. Then, we plotted execution time per iteration as a function of problem size for each KSM, matrix, and node count.

B.3 Computational Artifact A_3

Relation To Contributions

Computation artifact A_3 is a PETSc port of the BenchmarkStencil test program which was used to produce the comparative benchmarks reported in Figure 8.

Expected Results

The results of running the PETSc benchmark program are described in Figure 8 in the main paper.

Expected Reproduction Time (in Minutes)

- Artifact Setup: Downloading and building PETSc can be expected to take roughly 30 minutes, depending on the compilers used and the speeds of network and disk accesses.
- Artifact Execution: Each benchmarking run for PETSc takes roughly 10 minutes to scale from the smallest problem size (4096 unknowns) to the maximum problem size that fits into four NVIDIA V100 GPUs (see next section on Artifact Setup for details). Multiplied by 4 different test matrices and 3 different KSMs, this means one benchmarking run takes roughly 120 minutes. This must be repeated for each node count (in our case, scaling from 1 to 256 in powers of two), increasing the total execution time to roughly 960 minutes.
- Artifact Analysis: Execution time is directly measured by the benchmark program, and the only analysis step applied to these execution times is to divide by the iteration count and then take the minimum across three independent runs. This trivial arithmetic, together with generating plots of execution time as a function of problem size, can be performed in less than one minute.

These values are all hardware-dependent and should be expected to change on other supercomputers.

Artifact Setup (incl. Inputs)

The hardware and software environments used to execute this artifact were identical to those used for artifact A_1. We used PETSc release version 3.18.1, available at: https://gitlab.com/petsc/petsc/-/tree/v3.18.1

Datasets / Inputs. No external datasets are needed to run the experiments described in the main paper. All of the sparse matrices used are automatically generated by the benchmark programs at runtime.

Installation and Deployment. As with artifact A_1, both PETSc and our benchmark program were compiled with GCC 11.2.1 and NVCC from CUDA Toolkit 11.8.0. We used the Python build scripts provided by PETSc with the following configuration options:

```
configure_options = [
  '--download-c2html=0',
  '--download-hwloc=0',
  '--download-sowing=0',
  '--with-64-bit-indices=0',
  '--with-clanguage=C',
  '--with-cxx-dialect=C++17',
  '--with-cuda=1',
  '--with-debugging=0',
  '--with-fftw=0',
  '--with-hdf5=1',
  '--with-mumps=0',
  '--with-precision=double',
  '--with-scalapack=0',
  '--with-scalar-type=real',
  '--with-shared-libraries=1',
  '--with-ssl=0',
  '--with-suitesparse=0',
  '--with-trilinos=0',
  '--with-valgrind=0',
  '--with-x=0',
  '--with-zlib=1',
  'CFLAGS=-g -DNoChange',
  'COPTFLAGS=\"-O3\"',
  'CXXFLAGS=\"-O3\"',
  'CXXOPTFLAGS=\"-O3\"',
  'FFLAGS=-g',
  'CUDAFLAGS=-std=c++17',
  'FOPTFLAGS=',
  'PETSC_ARCH=arch-linux-c-opt',
]
configure.petsc_configure(configure_options)
```

After the PETSc library is built and installed, the Makefile in the benchmarks/petsc directory can be used to build the PETSc benchmark program.

Artifact Execution

Once the PETSc benchmark program is built, it can be run using jsrun as follows:

```
jsrun -M "-gpu" --rs_per_host 4 \
    --cpu_per_rs 10 --gpu_per_rs 1 \
    PETScSolverBenchmark -dim <dim> \
    -nx <nx> -ny <ny> -nz <nz> \
    -ksp_type <ksm> -ksp_max_it 500 \
    -pc_type none -ksp_atol 1.0e-100 \
    -ksp_rtol 1.0e-100 \
    -ksp_divtol 1.0e+100 \
    -vec_type cuda -mat_type aijcusparse
```

Note that the command-line options differ from the Legion-Solvers benchmark program; instead of using numeric codes, the matrix is directly specified using the strings 1D, 2D, 3D, 3D27, while the KSM is specified using the -ksp_type flag as one of cg, bcgs, or gmres. The tolerances -ksp_atol 1.0e-100, -ksp_rtol 1.0e-100, and -ksp_divtol 1.0e+100 are set to extreme values to prevent

early exit due to approximate convergence, and `-vec_type cuda -mat_type aijcusparse` is used to request GPU operation using CSR matrices.

Artifact Analysis (incl. Outputs)

For each combination of KSM, matrix, problem size, and node count, we performed three independent runs of the Trilinos test program and reported the minimum among the three execution times. Then, we plotted execution time per iteration as a function of problem size for each KSM, matrix, and node count.