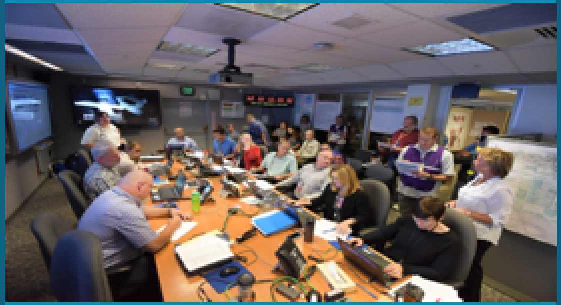
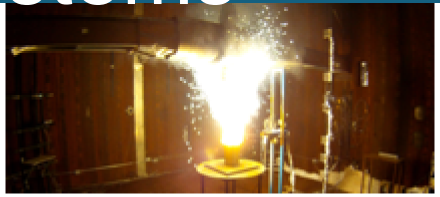
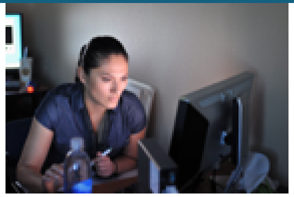
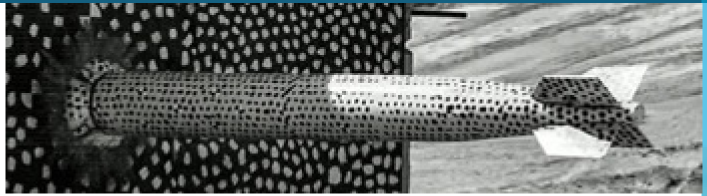
SAND2020-12737C

ADELUS: A Performance-Portable Dense LU Solver for Distributed-Memory Hardware-Accelerated Systems

By

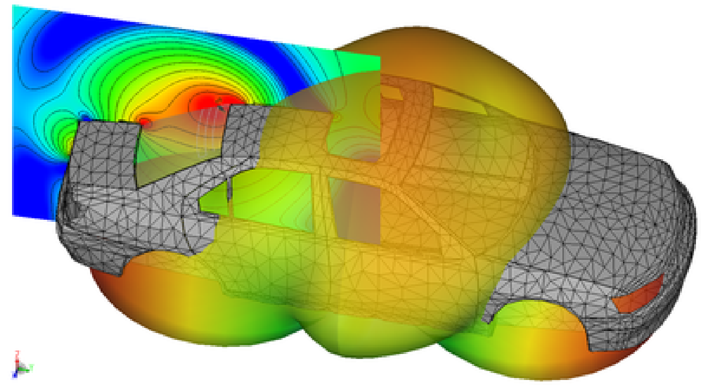
Vinh Dang, Joseph Kotulski, and Sivasankaran Rajamanickam

WACCPD@SC20 – Nov 13, 2020

Agenda

Introduction

- Solving a dense linear equations system **A*X=B** is one of the most fundamental problems in numerous applications: physics, mathematics, and engineering
  - Our application of interest: ***method of moments*** in electromagnetics
- Despite its high computational complexity, a direct solver (LU factorization) often provides more robust results than iterative solvers for extremely ill-conditioned system matrices
- A distributed-memory, dense LU solver capable of utilizing hardware accelerators available on top supercomputers is in need
- Performance-portability is important since future generation exascale HPC architectures are continuously evolving with significantly different architectures and programming models



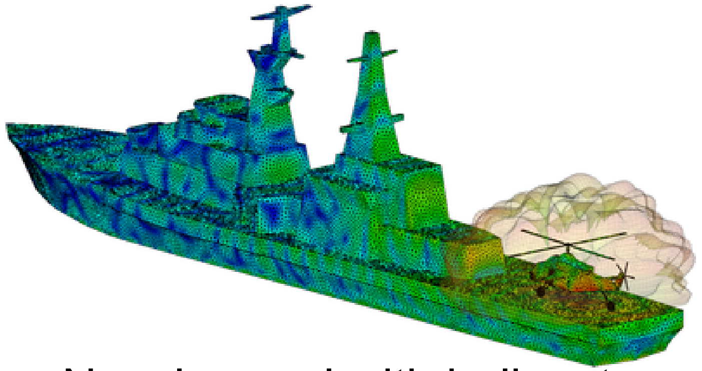
Near-field and radiation pattern analysis of integrated windscreen antenna
(Source: FEKO)

Argonne NATIONAL LABORATORY
U.S. DEPARTMENT OF ENERGY
intel
Hewlett Packard Enterprise
Aurora

The Aurora supercomputer
(Source: Intel)

ADELUS's Objectives

- A performance-portable dense LU solver for current and next generation distributed-memory hardware-accelerated HPC platforms
- Using LU factorization with partial pivoting for real/complex dense linear systems in distributed-memory using MPI
- Using torus-wrap mapping scheme for workload distribution
- Leveraging Kokkos and Kokkos Kernels to provide performance portability
- Integrating with a real-world application production code and achieving PFLOPS performance


Naval vessel with helicopter on deck (Source: FEKO)

U.S. DEPARTMENT OF ENERGY
NNSA
Lawrence Livermore National Laboratory
AMD
CRAY
EL CAPITAN
The El Capitan supercomputer (Source: Hewlett Packard Enterprise)

Kokkos and Kokkos Kernels

Kokkos Overview

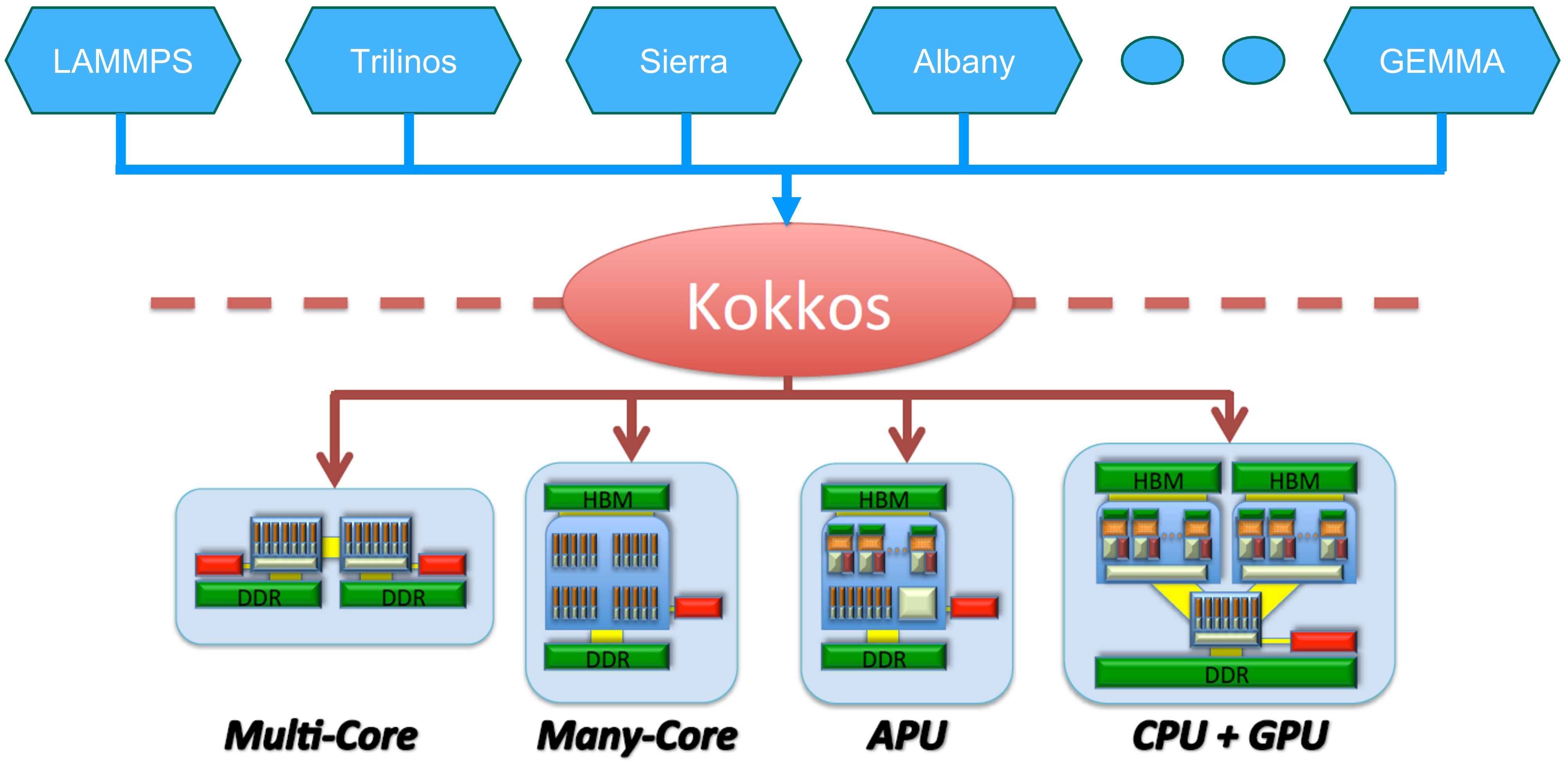
Kokkos is a **productive**, **portable**, **performant**, shared-memory programming model.

- ❑ is a C++ **library**, not a new language or language extension.
- ❑ supports **clear**, **concise**, **thread-scalable** parallel patterns.
- ❑ lets you write algorithms once and run on **many architectures**

  e.g. multi-core CPU, NVidia GPU, Xeon Phi, ...
- ❑ **minimizes** the amount of **architecture-specific implementation details** users must know.
- ❑ **solves the data layout problem** by using multi-dimensional arrays with architecture-dependent **layouts**

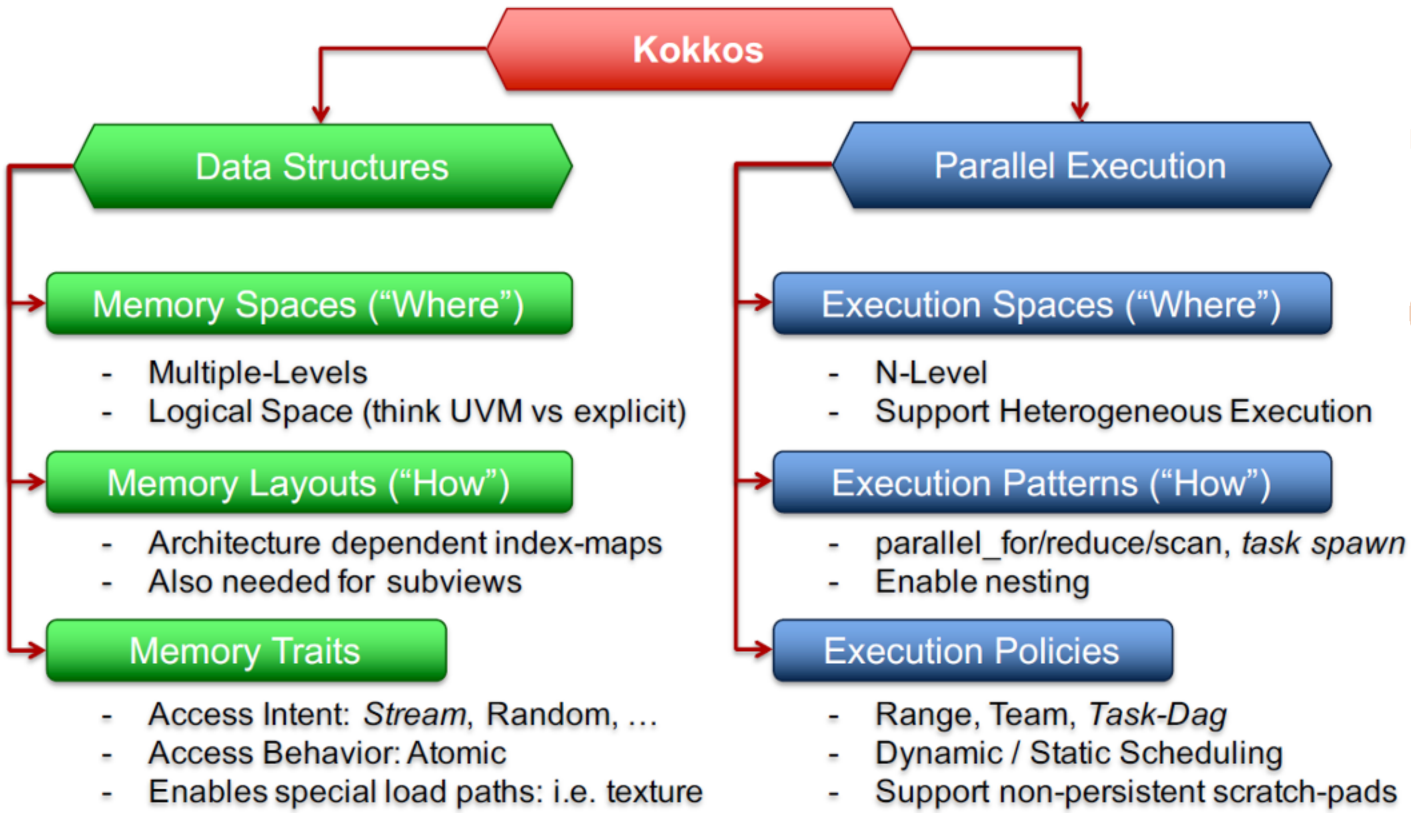
https://github.com/kokkos/kokkos-tutorials/blob/master/Intro-Short/Slides/KokkosTutorial_ATPESC18.pdf

An Abstraction Layer to Prevent Rewriting an Entire Code



Christian Trott, “Kokkos: Capabilities Overview”. https://github.com/kokkos/kokkos-tutorials/blob/master/KokkosCapabilities.pdf

Kokkos Data Management and Execution



Kokkos

Data Structures

Memory Spaces (“Where”)
- Multiple-Levels
- Logical Space (think UVM vs explicit)

Memory Layouts (“How”)
- Architecture dependent index-maps
- Also needed for subviews

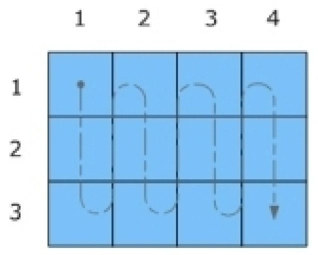
Memory Traits
- Access Intent: *Stream*, Random, …
- Access Behavior: Atomic
- Enables special load paths: i.e. texture

Parallel Execution

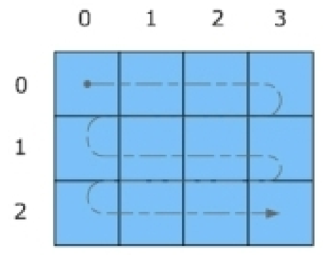
Execution Spaces (“Where”)
- N-Level
- Support Heterogeneous Execution

Execution Patterns (“How”)
- parallel_for/reduce/scan, *task spawn*
- Enable nesting

Execution Policies
- Range, Team, *Task-Dag*
- Dynamic / Static Scheduling
- Support non-persistent scratch-pads



A: Column-major order (Fortran-style)



B: Row-major order (C-style)

Execution Policies Patterns

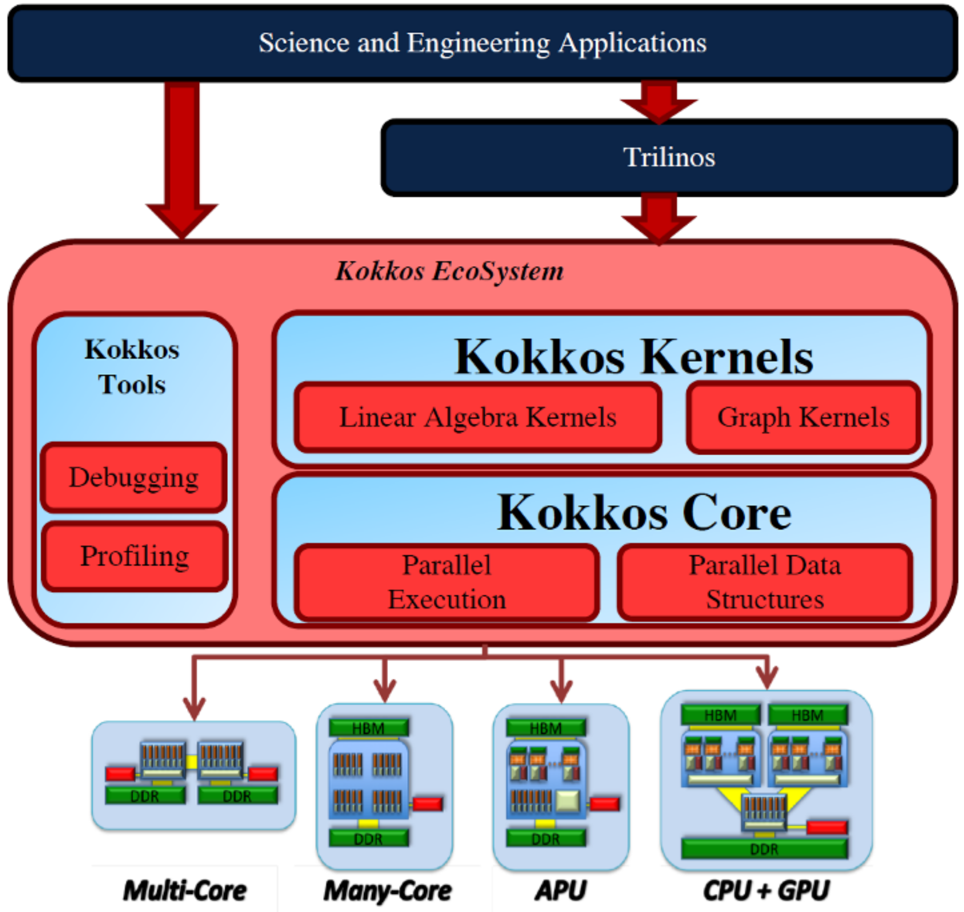
```
parallel_for(N, [=] (const size_t i) {
  /* loop body */
});

double totalIntegral = 0;
parallel_reduce(numberOfIntervals,
  [=] (const size_t i, double & valueToUpdate) {
    valueToUpdate += function(...);
  },
  totalIntegral);


parallel_outer(
  TeamPolicy<>(numberOfTeams, teamSize, vectorLength),
  KOKKOS_LAMBDA (const member_type & teamMember[, ...]) {
    /* beginning of outer body */
    parallel_middle(
      TeamThreadRange(teamMember, thisTeamsRangeSize),
      [=] (const int indexWithinBatch[, ...]) {
        /* begin middle body */
        parallel_inner(
          ThreadVectorRange(teamMember, thisVectorRangeSize),
          [=] (const int indexVectorRange[, ...]) {
            /* inner body */
          }[, ....);
        /* end middle body */
      }[, ...]);
    /* end of outer body */
  }[, ...]);
```

Christian Trott, “Kokkos: Capabilities Overview”. https://github.com/kokkos/kokkos-tutorials/blob/master/KokkosCapabilities.pdf
Intel. *Developer Guide for Intel Math Kernel Library for Linux*. https://software.intel.com/en-us/node/528573

Kokkos Kernels

KokkosKernels is a library for *node-level*, performance-portable, computational kernels for sparse/dense linear algebra and graph operations, using the Kokkos shared-memory parallel programming model.

- KK is available publicly both as part of Trilinos and as part of the **Kokkos ecosystem**
- Building block of a solver, linear algebra library that uses MPI and threads for parallelism, or it can be used stand-alone in an application.
- Generic implementations for various scalar types and data layouts
- Interfaces to Intel, NVIDIA and other vendor provided kernels available in order to leverage their high-performance libraries
- Several new kernels are being added as needed by the applications
  - E.g.: Distance-2 Coloring, Deterministic coloring, **dense linear system solver**, …
- Expand the scope of BLAS to hierarchical implementations.



Method of Moments for Linear Electromagnetics

Maxwell's Equations in the Frequency Domain

Maxwell's Equations:

$$\text{Faraday}: \quad \nabla \times \mathbf{E} = -j\omega \mathbf{B}$$
$$\text{Ampere} - \text{Maxwell}: \quad \nabla \times \mathbf{H} = \mathbf{J} + j\omega \mathbf{D}$$
$$\text{Electric Gauss}: \quad \nabla \cdot \mathbf{D} = \rho$$
$$\text{Magnetic Gauss}: \quad \nabla \cdot \mathbf{B} = 0$$

Vector and Scalar Potentials:

$$\mathbf{E} = -j\omega \mathbf{A} - \nabla \Phi$$
$$\mathbf{B} = \nabla \times \mathbf{A}$$

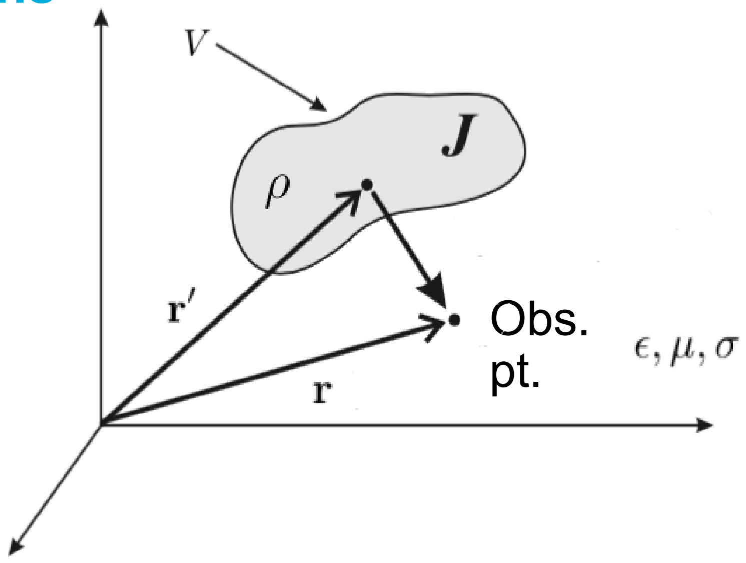
Lorenz gauge condition:

$$\nabla \cdot \mathbf{A} = -j\omega\epsilon\mu\Phi$$

Wave Equations:

$$\nabla^2 \mathbf{A} + \omega^2 \mu\epsilon \mathbf{A} = -\mu \mathbf{J}$$
$$\nabla^2 \Phi + \omega^2 \mu\epsilon \Phi = \rho/\epsilon$$

Instead of solving Maxwell's equations in 3D space via the wave equations, we solve them on the boundary between regions.



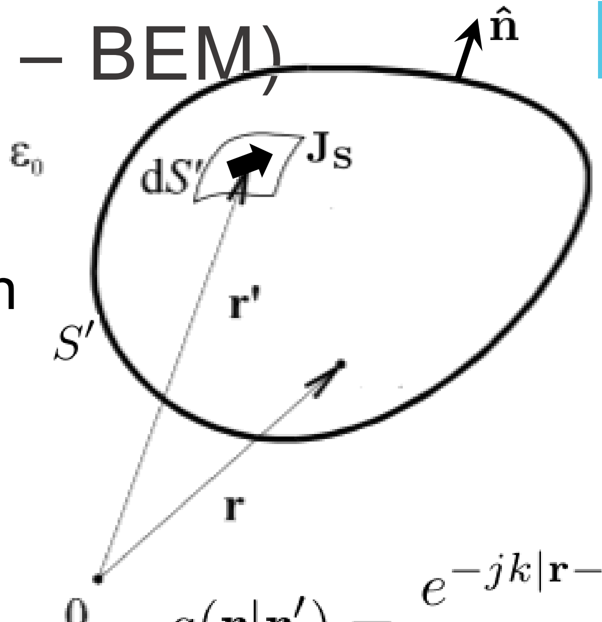
For a linear homogeneous, unbounded medium:

$$\mathbf{A} = \int_V \mu \mathbf{J}(\mathbf{r}') g(\mathbf{r}|\mathbf{r}') dv'$$
$$\Phi = -\int_V \frac{\rho(\mathbf{r}')}{\epsilon} g(\mathbf{r}|\mathbf{r}') dv'$$

Free-Space Green's Function:

$$g(\mathbf{r}|\mathbf{r}') = \frac{e^{-jk|\mathbf{r}-\mathbf{r}'|}}{4\pi|\mathbf{r}-\mathbf{r}'|}$$

Integral Equations (Boundary Element Method – BEM)



Example of an electric field integral equation (EFIE) for metallic scatterer:

Through the equivalence principle, we consider the current on an objects boundary instead of the field around and inside the object. Enforcing the boundary condition at the surface:

$$\hat{\mathbf{n}} \times (\mathbf{E}_{\mathbf{inc}} + \mathbf{E}_{\mathbf{scat}}) = \mathbf{0}$$

$$g(\mathbf{r}|\mathbf{r}') = \frac{e^{-jk|\mathbf{r}-\mathbf{r}'|}}{4\pi|\mathbf{r}-\mathbf{r}'|}$$

where,

$$\mathbf{E}_{\mathbf{scat}} = -j\omega\mu \int_{S'} \left(\boxed{\mathbf{J}_{\mathbf{S}}(\mathbf{r}')g(\mathbf{r}|\mathbf{r}')} + \boxed{\frac{1}{\omega^2\mu\epsilon}\nabla' \cdot \mathbf{J}_{\mathbf{S}}(\mathbf{r}')\nabla g(\mathbf{r}|\mathbf{r}')} \right) ds'$$

results in the following integral equation:

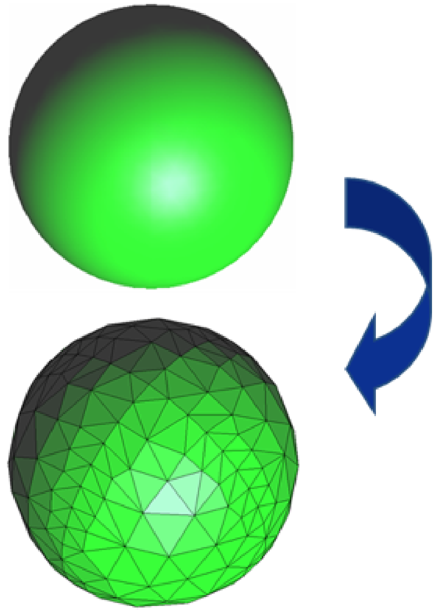
$$\boxed{\begin{gathered} \int_{S'} \hat{\mathbf{n}} \times \left(\mathbf{J}_{\mathbf{S}}(\mathbf{r}')g(\mathbf{r}|\mathbf{r}') + \frac{1}{\omega^2\mu\epsilon}\nabla' \cdot \mathbf{J}_{\mathbf{S}}(\mathbf{r}')\nabla g(\mathbf{r}|\mathbf{r}') \right) ds' = \frac{1}{j\omega\mu}\hat{\mathbf{n}} \times \mathbf{E}_{\mathbf{inc}} \\ L\{\mathbf{J}_{\mathbf{S}}\} = \frac{1}{j\omega\mu}\hat{\mathbf{n}} \times \mathbf{E}_{\mathbf{inc}} \end{gathered}}$$

Method of Moments (MoM)

Numerical solution of integral equation:

$$L\left\{\mathbf{J_S}\right\} = \frac{1}{j\omega\mu}\hat{\mathbf{n}} \times \mathbf{E_{inc}}$$

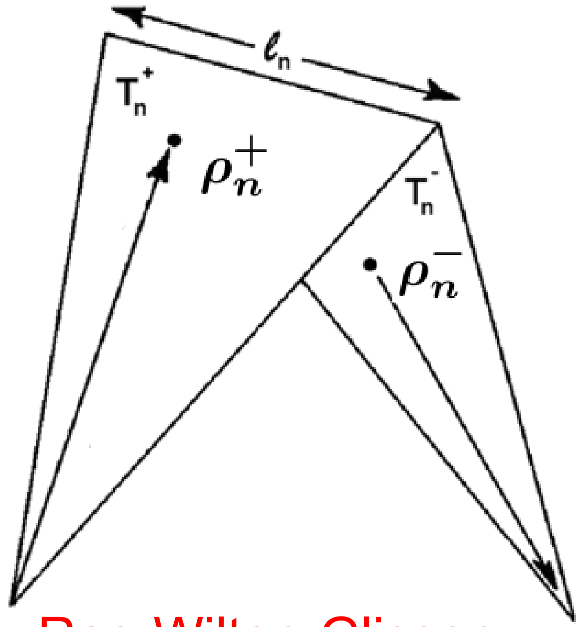
Discretize the scatterer



Expand unknown in a set of basis functions:

$$\mathbf{J_S}(\mathbf{r}) \approx \sum_n I_n \mathbf{f_n}(\mathbf{r})$$

$$\mathbf{f_n}(\mathbf{r}) = \begin{cases} \frac{\ell_n}{2A_n^+}\boldsymbol{\rho}_n^+ & \mathbf{r} \in T_n^+ \\ \frac{\ell_n}{2A_n^-}\boldsymbol{\rho}_n^- & \mathbf{r} \in T_n^- \\ \mathbf{0} & \text{otherwise} \end{cases}$$



Rao-Wilton-Glisson (RWG) basis functions

Test integral equation with basis functions:

$$\boxed{\begin{gathered} \int_S \mathbf{f_m} \cdot L\left\{\mathbf{J_S}\right\} ds = \frac{1}{j\omega\mu}\int_S \mathbf{f_m} \cdot \left(\hat{\mathbf{n}} \times \mathbf{E_{inc}}\right) ds \\ \overline{\mathbf{Z}}\mathbf{I} = \mathbf{V} \\ Z_{m,n} = \int_{f_m}\int_{f_n}\left[j\omega\mu \boldsymbol{f}_m \cdot \boldsymbol{f}_n - \frac{j}{\omega\epsilon}\nabla \cdot \boldsymbol{f}_m \nabla' \cdot \boldsymbol{f}_n\right]\frac{e^{-ikr}}{4\pi r} \end{gathered}}$$

Parallel LU Solver Implementation

ADELUS Interface and Storage

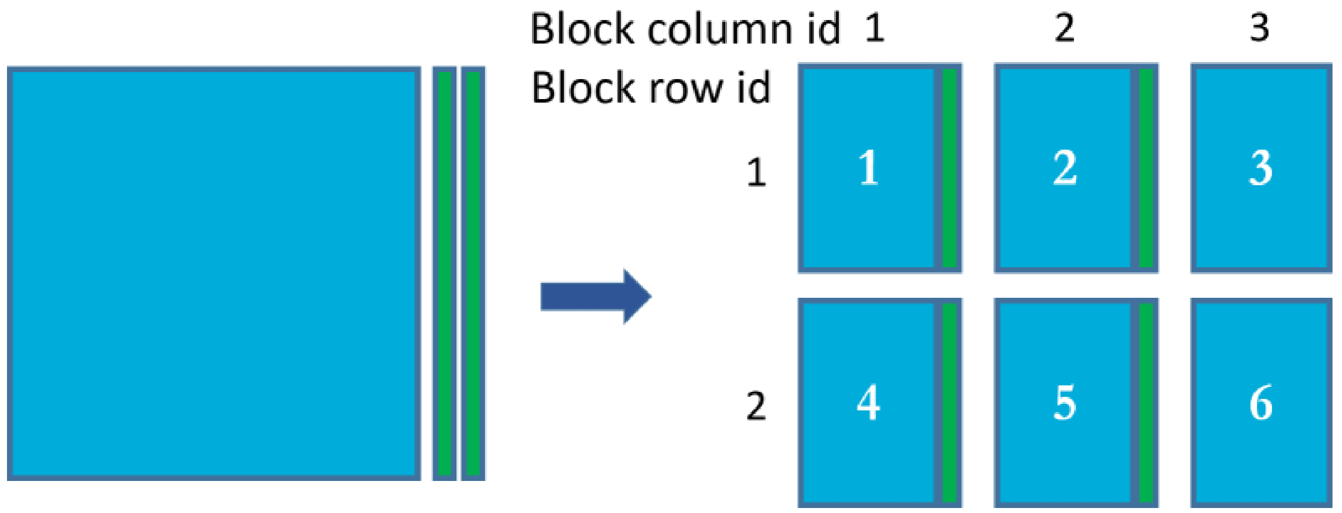
- Dense matrix and RHS vectors that are block-mapped to the MPI processes
- ADELUS is called by MPI processes with the matrix portions packed with RHS vectors (column-major order) as their inputs
- ADELUS data container is implemented by the Kokkos View for portability

In the host memory:

```
Kokkos::View<Kokkos::complex<double>**,Kokkos::LayoutLeft,Kokkos::HostSpace>
                                       A("A",my_rows,my_cols+my_rhs);
```

In the CUDA device memory:

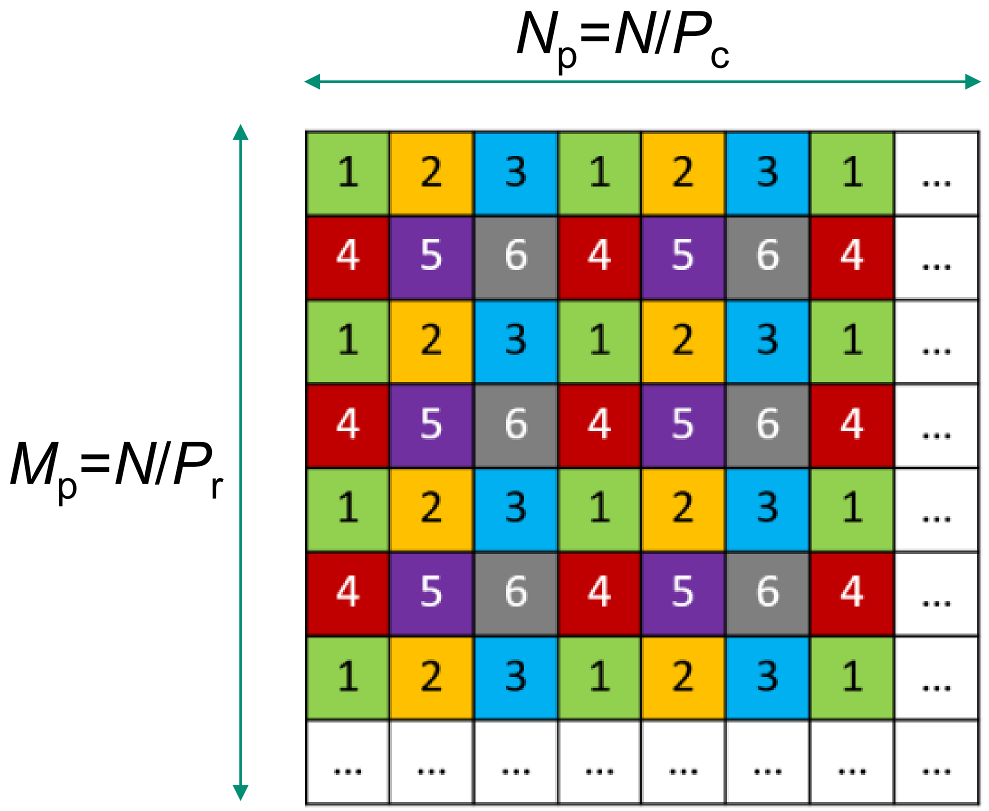
```
Kokkos::View<Kokkos::complex<double>**,Kokkos::LayoutLeft,Kokkos::CudaSpace>
                                       A("A",my_rows,my_cols+my_rhs);
```



- Total number of MPI processes = 6
- Number of processes for a row = 3
- Number of right-hand sides = 2

Torus-Wrap Mapping

- Advantage: each process has nearly the same workload and the process idle time is minimized
- Column indices assigned to a MPI process constitute a linear sequence with step size P_c
- Row indices are in a sequence separated by P_r
- No need to redistribute the block-mapped matrix for torus-wrapped solver
  - A block-mapped system can be solved by a solver assuming a torus-wrapped system.
- Solution vectors are corrected afterwards by straightforward permutations



1	2	3	1	2	3	1	...
4	5	6	4	5	6	4	...
1	2	3	1	2	3	1	...
4	5	6	4	5	6	4	...
1	2	3	1	2	3	1	...
4	5	6	4	5	6	4	...
1	2	3	1	2	3	1	...
...	...	...	...	...	...	...	...

$P=P_c \times P_r$

- Total number of MPI processes: 6 (P=6)
- Number of processes for a row: 3 (P_c=3)
- Number of right-hand sides: 2

LU Factorization and Forward Solve

- Right-looking variant of the LU factorization with partial pivoting
- Kokkos Kernels BLAS interfaces are used for local matrices in each MPI process
  - Calls to optimized vendor library BLAS routines: multi-threaded CPU (IBM's ESSL BLAS), massively parallel GPU architectures (cuBLAS)
- CUDA-aware MPI: simple communication patterns: point-to-point communication and collective communication
- 4 steps per iteration:

1. Find the pivot: each process finds its own local maximum entry in the current column and then exchanges for the global pivot value.
2. Scale the current column with the pivot value and generate and communicate column update vector from the column
3. Exchange pivot row and diagonal row
4. Update the current column, and if saving enough columns, to update Z via gemm

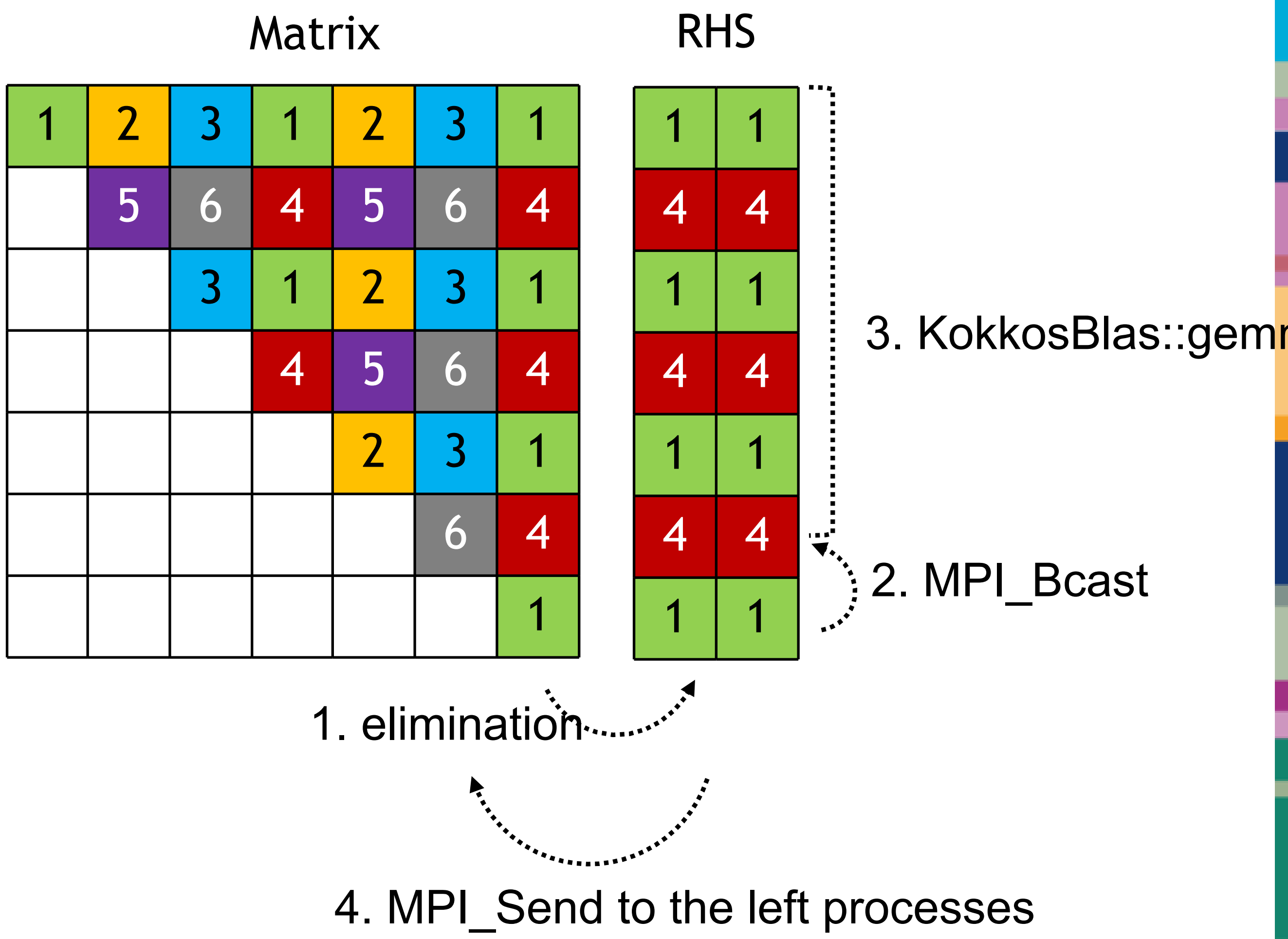
```
KokkosBlas::iamax()
KokkosBlas::scal()
KokkosBlas::copy()
KokkosBlas::gemm()
```

```
MPI_Send()
MPI_Recv()
MPI_Irecv()
MPI_Allreduce()
MPI_Bcast()
```

Backward Solve

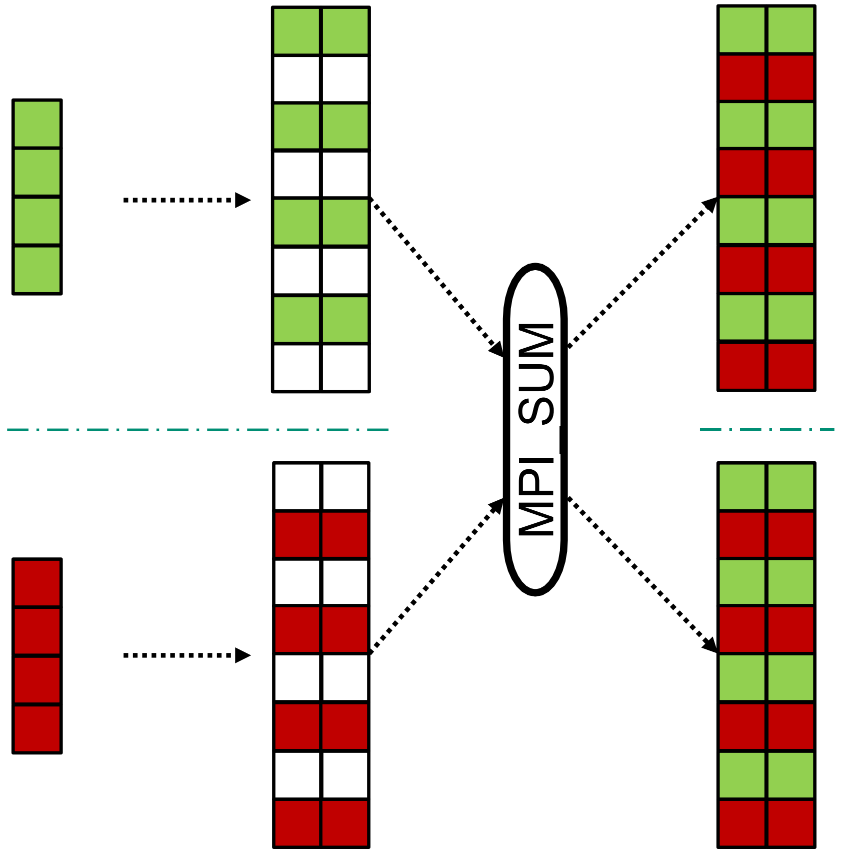
Backward Solve

1. The elimination of the RHS is performed by the process owning the current column using the Kokkos `parallel_for`
2. The results from the elimination step are broadcasted to all the processes within the MPI column sub-communicator
3. The `KokkosBlas::gemm` is then called to update the RHS
4. To prepare for the next iteration, the newly-computed RHS vectors are sent to the processes to the left



Permutation

- Permutation: to "unwrap the results"
  - Solver assumes the torus-wrap mapping scheme while the input matrix is not torus -wrapped
  - A temporary buffer for global solution vectors created
  - Kokkos `parallel_for` to fill the correct locations in the global vectors
  - MPI Allreduce to collectively update the change



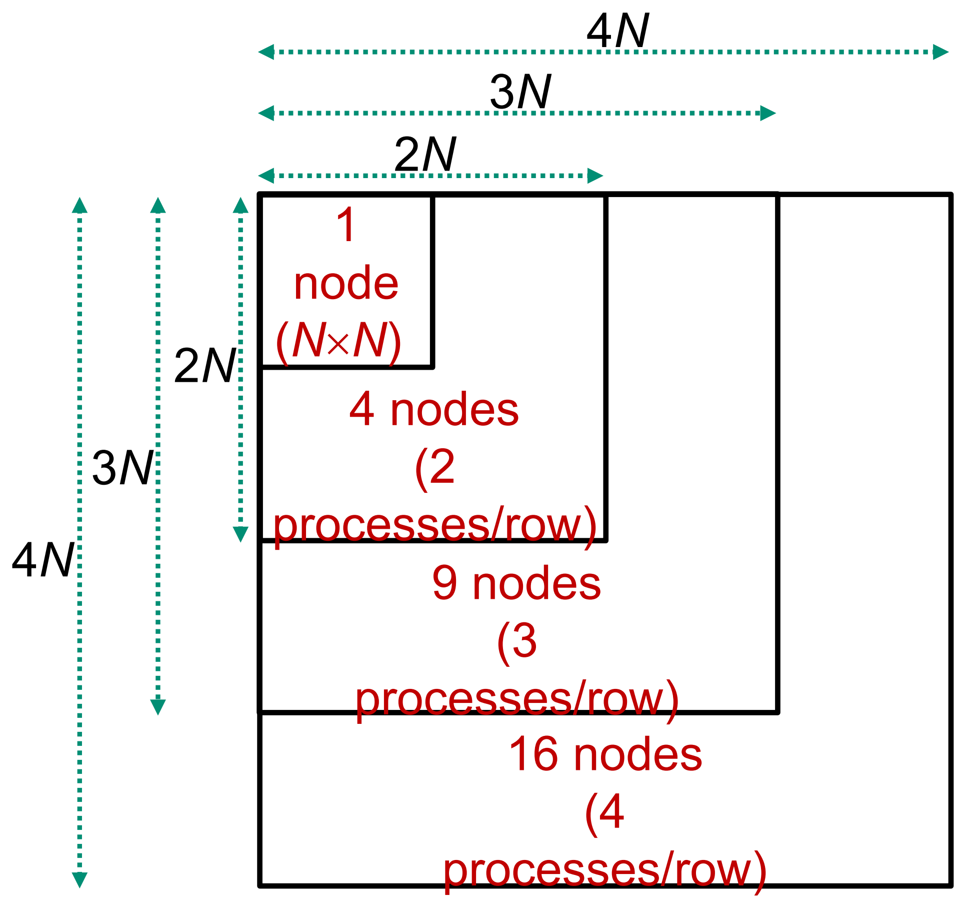
Experimental Results

Experimental Setup

- Summit system at the ORNL (4608 nodes): evaluating performance of ADELUS with randomly-generated matrices
  - Hardware (per node): 2 POWER9 CPUs (22 cores/each), **6** V100 GPUs (16GB memory/GPU)
    - Intra-node connection: NVIDIA's NVLink 2.0
    - Inter-node connection: Mellanox dual-rail enhanced data rate (EDR) InfiniBand network
  - Software environment: GCC 7.4.0, CUDA 10.1.243, ESSL 6.2.0, Spectrum MPI 10.3.1
  - DPLASMA: IBM XL C/C++ Compiler 16.1.1 instead of GCC 7.4.0
  - SLATE: we use GCC 6.4.0 and ESSL 6.1.0, Netlib SCALAPACK 2.0.2
- Sierra system at the LLNL (4320 nodes): demonstrating performance of ADELUS when integrated into a production electromagnetic application code, EIGER
  - Hardware (per node): 2 POWER9 CPUs (22 cores/each), **4** V100 GPUs (16GB memory/GPU)
    - Intra-node connection: NVIDIA's NVLink 2.0
    - Inter-node connection: Mellanox dual-rail enhanced data rate (EDR) InfiniBand network
  - Software environment: GCC 7.2.1, CUDA 10.1.243, ESSL 6.2.0, Spectrum MPI 10.3.0

Randomly-Generated Matrices (1/6)

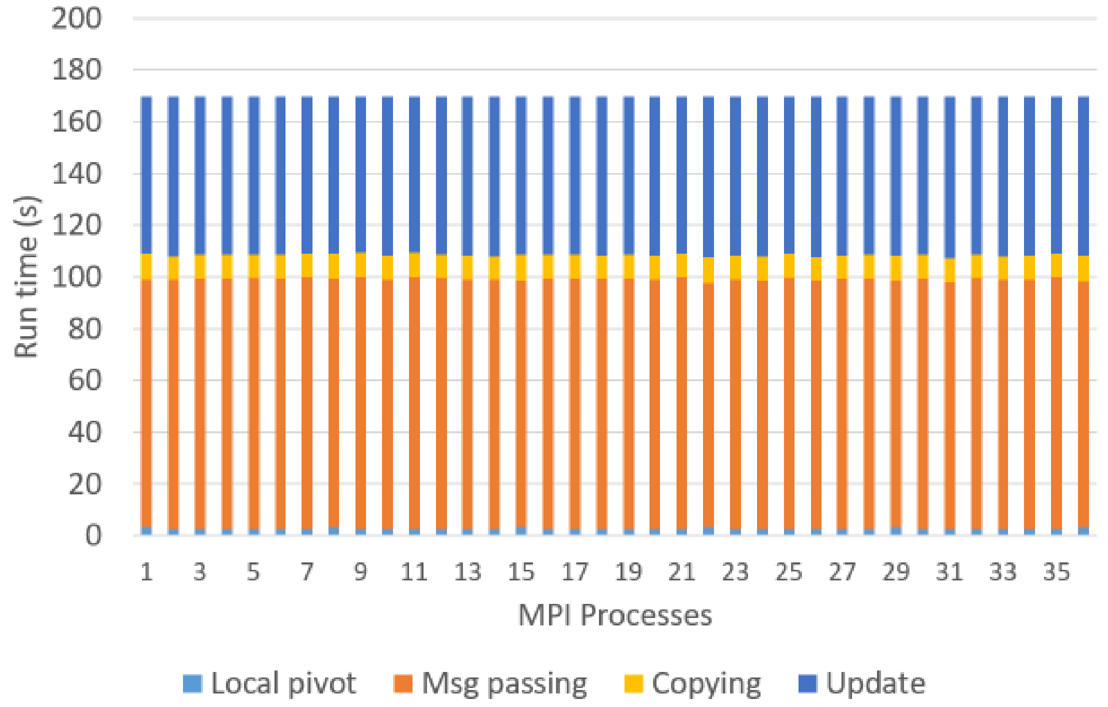
- Single RHS vector and the matrix size is increased as we increase the hardware resource
- GPU backend: ADELUS runs with one MPI rank per GPU.
- CPU backend: ADELUS runs with one MPI rank per node (42 cores)


4N
3N
2N
2N
3N
4N
1 node (N×N)
4 nodes (2 processes/row)
9 nodes (3 processes/row)
16 nodes (4 processes/row)

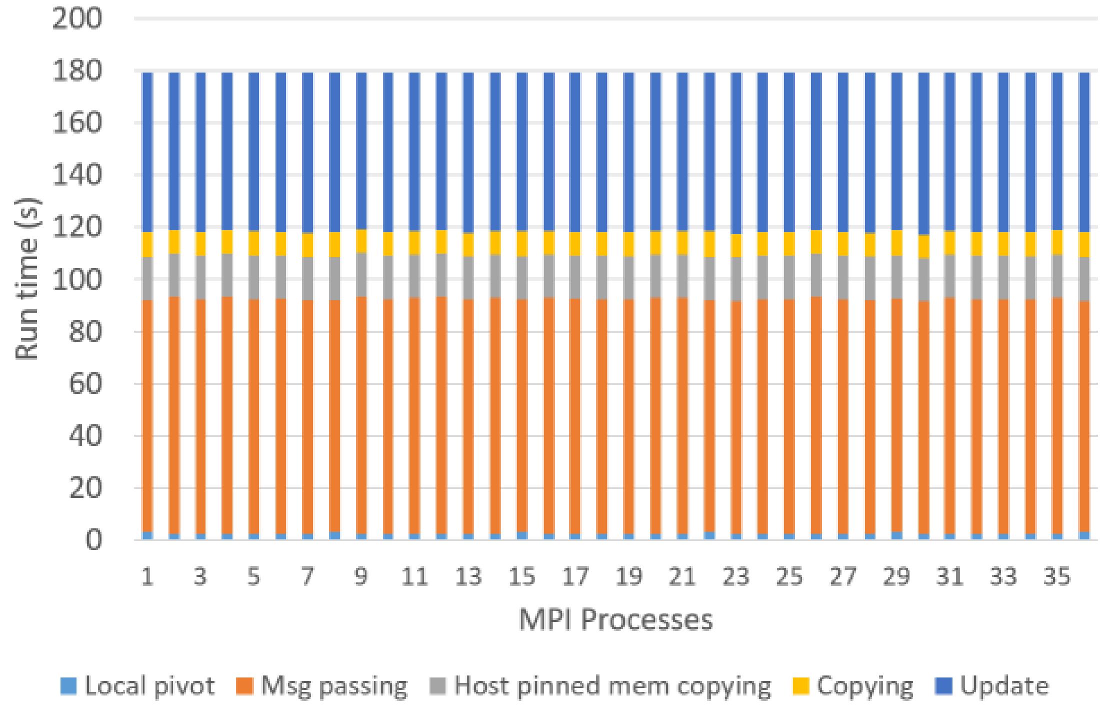
Randomly-Generated Matrices (2/6)

❑ Load Balancing Verification

- Factorization time on 36 MPI processes (36 GPUs) with the matrix size of 6N6N (167,292×167,292)
- Communication and the update contribute the most to the total time
- Communication time is 1.47x-1.6x the update time



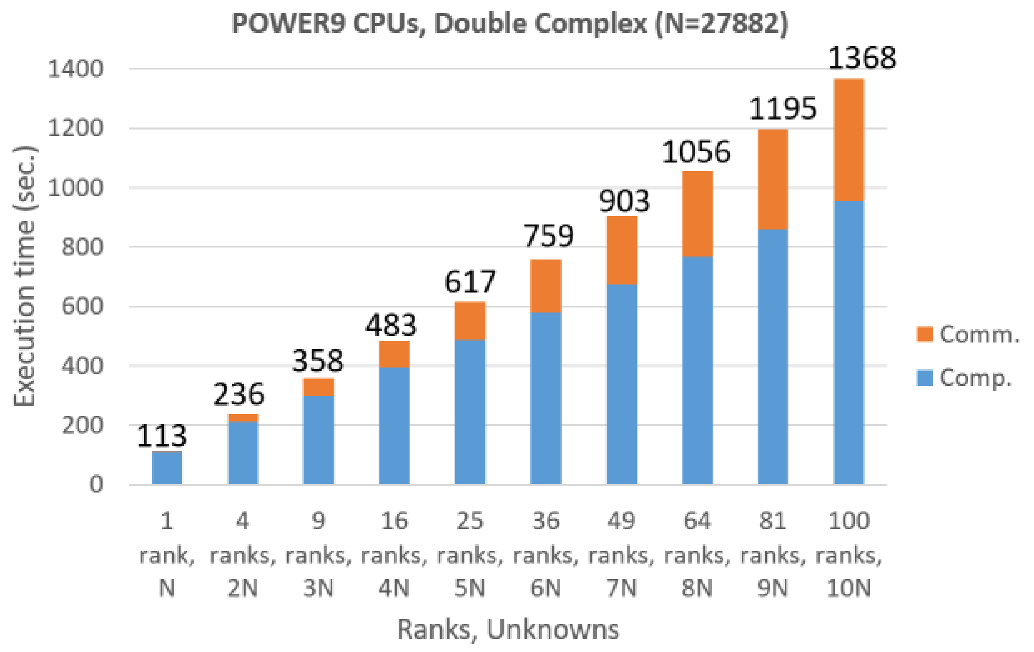
Cuda-aware MPI



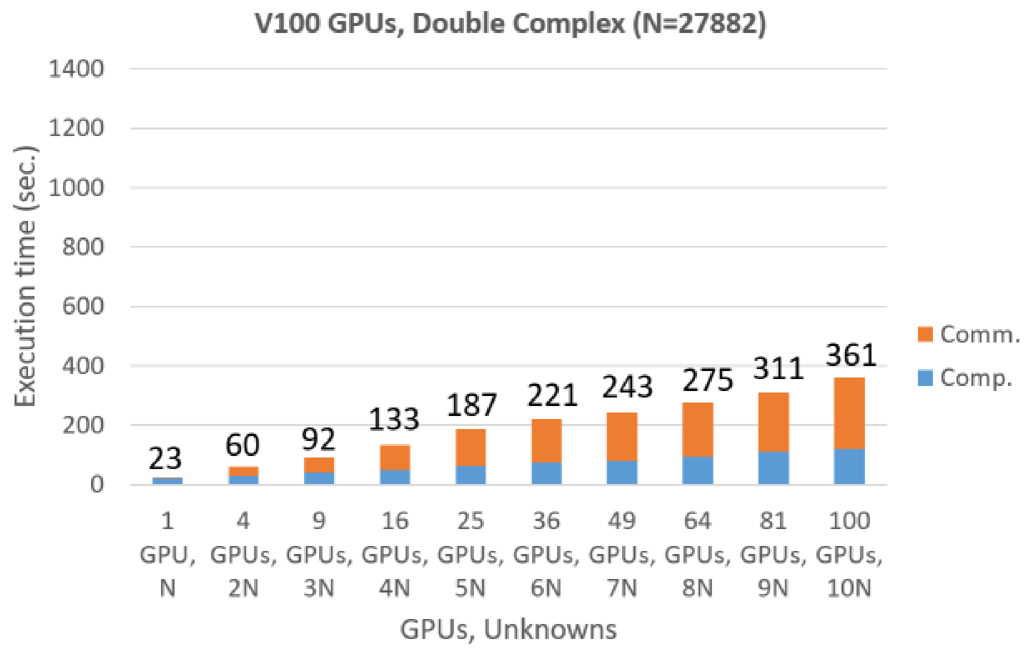
Host pinned memory for MPI

Randomly-Generated Matrices (3/6)

- CPU vs. GPU
  - A single GPU is 4.9x faster than a 42-core CPU
  - 100 GPUs is 3.8x faster than 100 42-core CPUs
  - Communication overhead increases as processing larger problems (broadcasting pivot rows and exchanging rhs vectors among column processes)
  - CPU computation is still the dominant component in the total CPU time
  - GPU computation is fast that makes the computation overhead the bottleneck



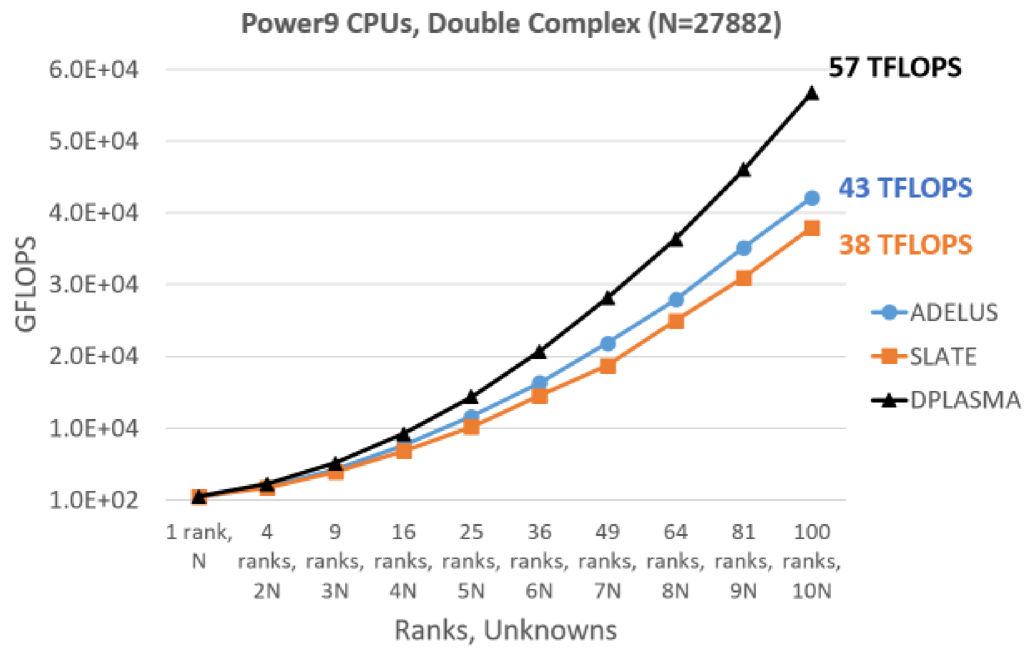
CPU execution time



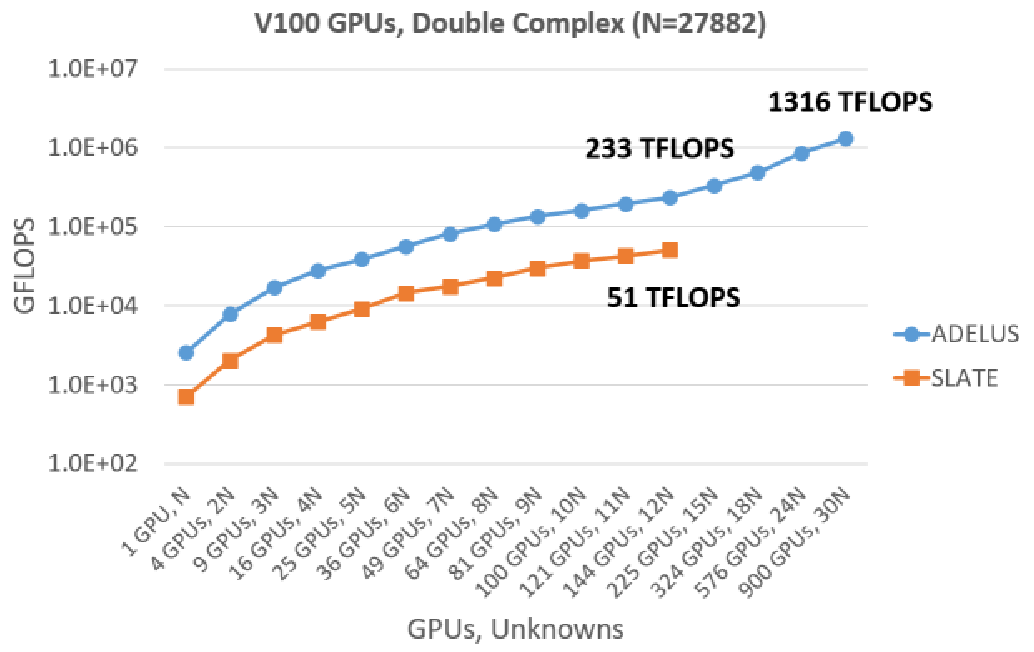
GPU execution time with host pinned memory

Randomly-Generated Matrices (4/6)

- ADELUS vs. DPLASMA and SLATE
  - Tuning DPLASMA and SLATE for their best performance
  - ADELUS (43 TFLOPS) outperforms SLATE (38 TFLOPS) while is slower than DPLASMA (57 TFLOPS) on 100 CPUs
  - ADELUS is 4.57x faster than SLATE on 144 GPUs
  - ADELUS can achieve 1,316 TFLOPS (1.3 PFLOPS) with 900 GPUs (the first complex, dense LU solver reachs PFLOPS performance)



CPU performance

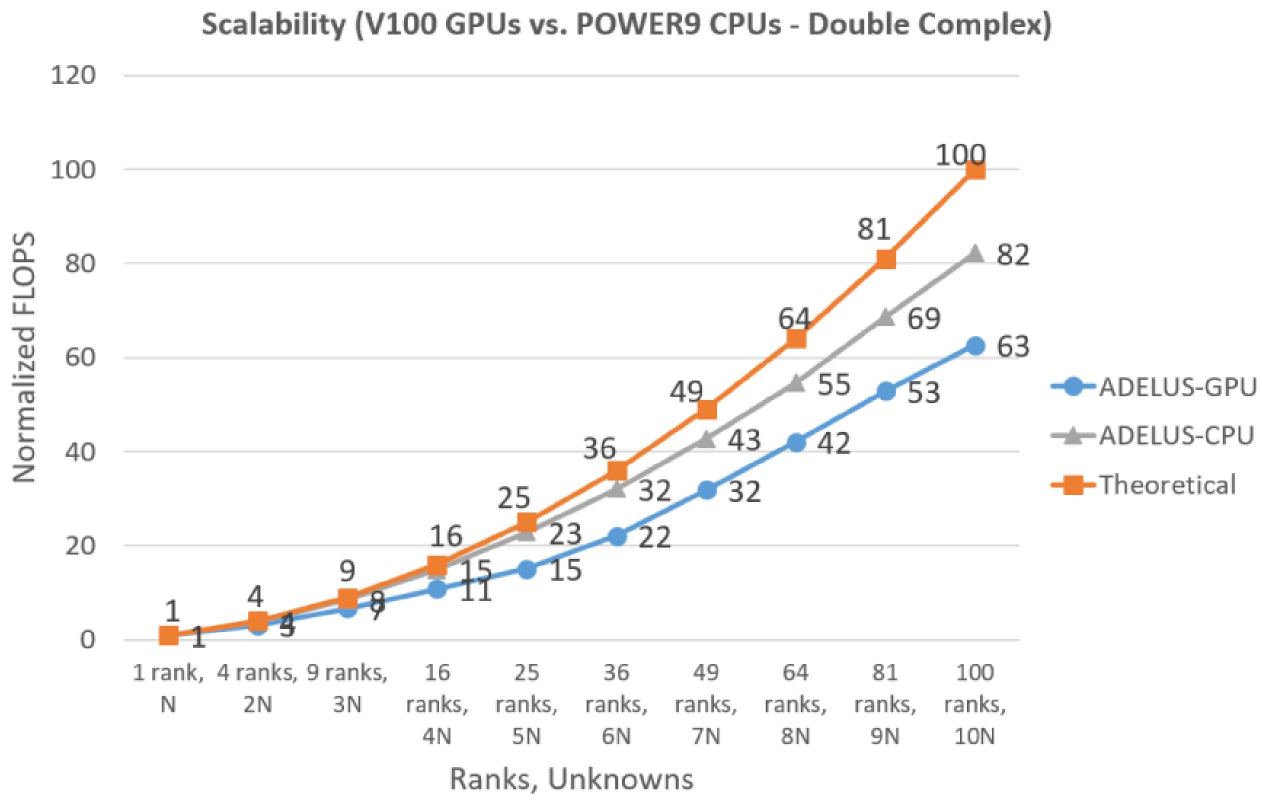


GPU performance

Randomly-Generated Matrices (5/6)

❑ Scalability Analysis

- Scalability is defined as the normalized FLOPS of multiple MPI processes with respect to FLOPS of a single MPI process
- The increase of communication overhead results in less than ideal scalability in both CPU and GPU runs
- ADELUS on CPUs scales more closely to the theoretical ideal scalability than ADELUS on GPUs
- GPU performance is MPI bound due to the increase in the communication cost and its high FLOPS
- Scalability needs further improvement



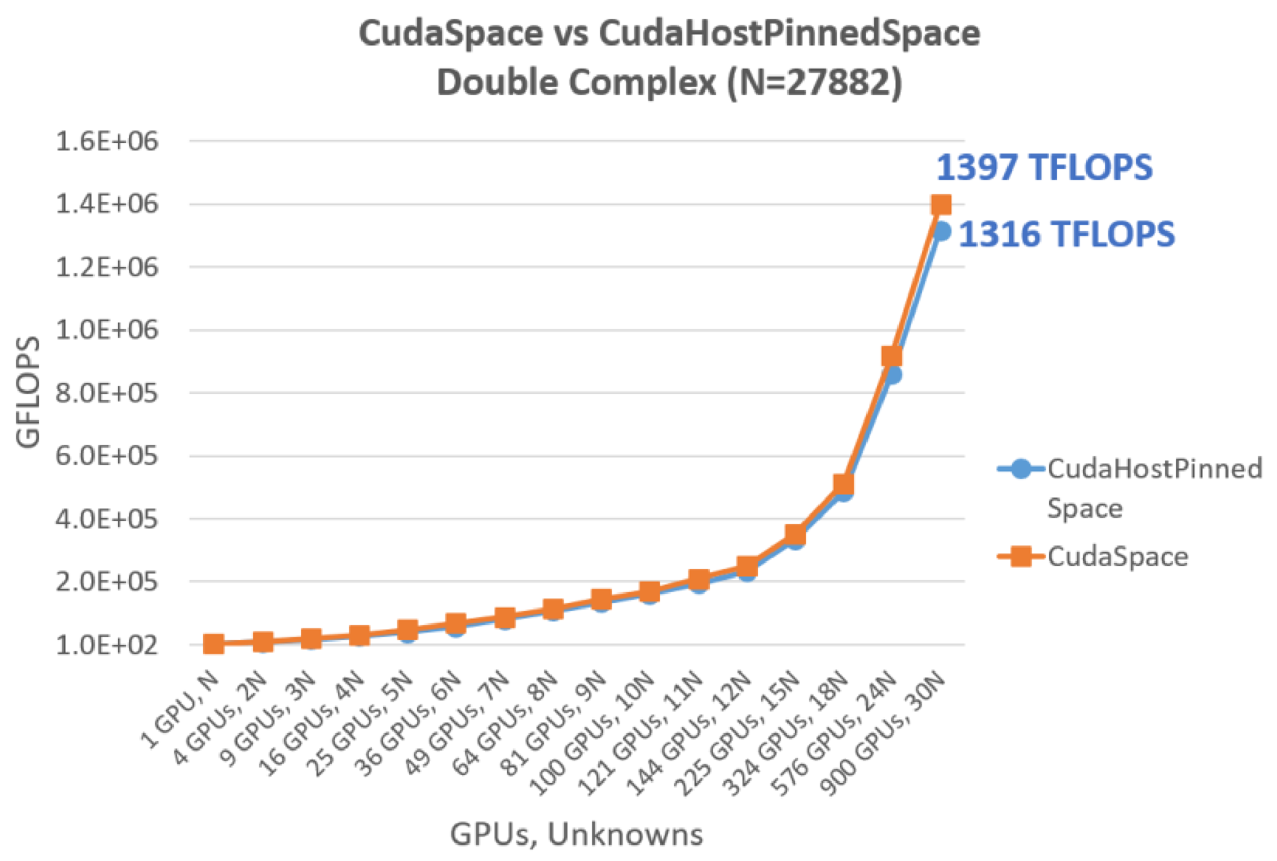
$$S=\frac{\text{FLOPS}(\mathbf{m}\text{ ranks/GPUs, }\mathbf{n}*\text{N unknowns})}{\text{FLOPS}(\mathbf{1}\text{ rank/GPU, }\mathbf{1}*\text{N unknowns})}$$

where ranks/GPUs = 1, 4, 9, 16, 25, 36, 49, 64, 81, 100
unknowns = 1N, 2N, 3N, 4N, 5N, 6N, 7N, 8N, 9N, 10N

Randomly-Generated Matrices (6/6)

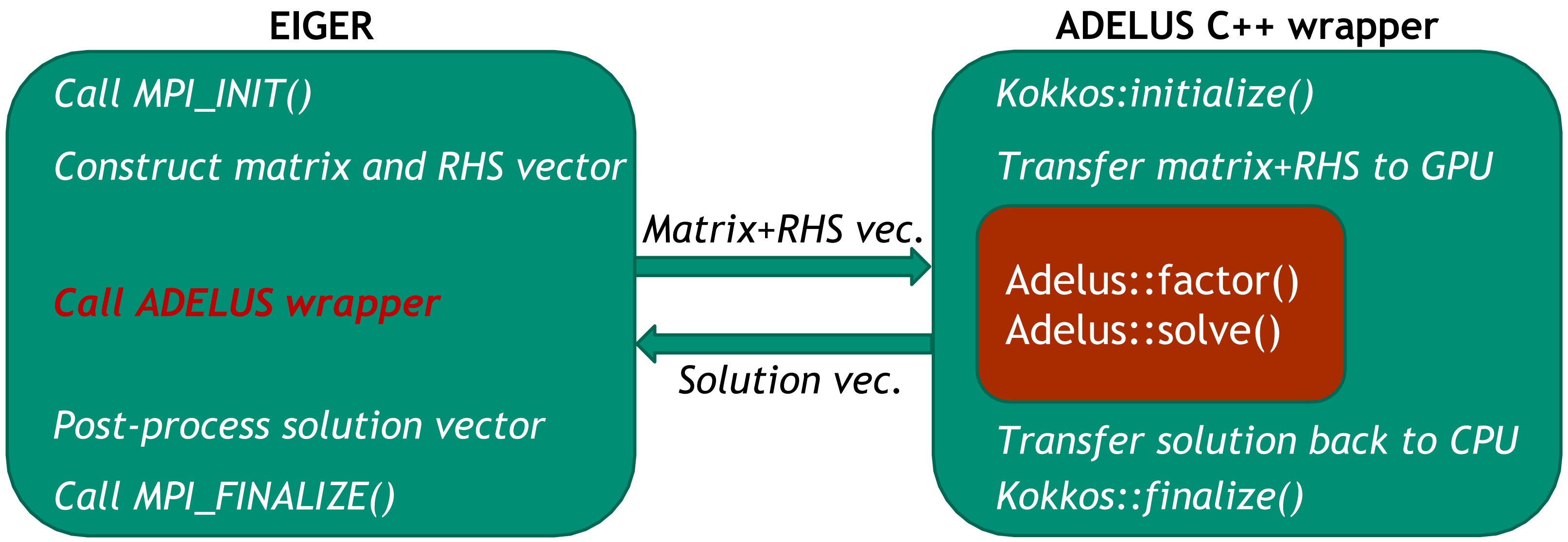
❑ MPI Buffers on Different Memory Spaces

- Both CudaSpace and CudaHostPinnedSpace can attain performance above 1000 TFLOPs
- Using CUDA-aware MPI can improve the performance by 6% since we do not need to explicitly buffer data on host memory before or after calling the MPI function



Large-Scale EM Simulation with EIGER

- Couple EIGER with ADELUS to perform large-scale electromagnetic simulations on the LLNL's Sierra platform
- First time Petaflops performance with a complex, dense LU solver: 7.72 Petaflops (16.9% efficiency) when using 7,600 GPUs on 1,900 nodes on a 2,564,487-unknown problem
- ADELUS's performance is affected by the distribution of the matrix on the MPI processes
  - Assigning more processes per row yields higher performance



N	Nodes (GPUs)	Solve time (sec.)	TFLOPS	Procs/row
226,647	25 (100)	240.5	1291.0	10
1,065,761	310 (1240)	1905.1	1694.5	31
1,322,920	500 (2,000)	6443.9	958.1	20
1,322,920	500 (2,000)	2300.2	2684.1	50
1,322,920	500 (2,000)	2063.6	2991.9	100
2,002,566	1,200 (4,800)	3544.1	6042.6	100
2,564,487	1,900 (7,600)	5825.2	7720.7	80

Conclusions and Future Work

- A parallel, dense, performance-portable, LU solver based on torus-wrap mapping and LU factorization algorithm
- Obtaining portability through Kokkos and Kokkos Kernels
- ADELUS's performance on Summit: 1.397 PFLOPS on 900 GPUs
- The GPU execution is 3.8x faster than the CPU execution
- ADELUS integrated into an electromagnetic application (EIGER) achieves 7.720 PFLOPS on 7600 GPUs (a problem of 2.5M unknowns) on Sierra
- Future work:
  - Using computation-communication overlapping to improve ADELUS scalability on GPUs
  - A hybrid implementation where both CPU and GPU resources are fully utilized to overcome the limitation of the GPU memory

Availability

https://github.com/trilinos/Trilinos/tree/master/packages/adelus

- ❑The driver code used for our ADELUS experiments can be found in https://github.com/trilinos/Trilinos/tree/master/packages/adelus/example

Acknowledgment
Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.