Speeding Up Matrix-Vector Operations via a 2D Domain Decomposition using MPI

Thomas Gorham
Department Computer Science and Engineering
The University of Tennessee, Chattanooga, TN 37403

***Abstract*—Matrix-vector operations such as $y = Ax$ can be sped up by partitioning two-dimensional logical process grids with MPI to reduce the number of operations for distributed-memory processes to carry out. This paper describes how MPI communicators facilitate the partitioning of matrix A(M,N) on (P,Q) logical MPI process grids to establish A(m,n) sub-regions. Vector x is then built on the sub-region of size n and broadcasted to the other processes with the MPI column communicator to the sub-regions where m != 0 (e.g., node 1). Each MPI process then performs its local/respective A(m,n)*x(n) computations. In order to finish deriving the global solution to A(M,N) * x(N), the MPI_Allreduce collective is called in placed on Darray y, effectively storing the solution Vector y(M) in each process row. Next, the solution Vector y(M) is redistributed into Vector X(N) by multiplying each MPI domains'respective y(n) by an identity matrix(m,n). This final computation effectively stores the same solution to the computation derived from A(M,N)*y(N) in each process column via vector X. Lastly, we port the solution to Kokkos to enable immediate GPU offload as needed, along with the potential to enable on-node shared memory parallelism with OpenMP.**

***Index Terms*—Domain Decomposition, GPU, HPC, Kokkos, Matrix-Vector Operations, MPI,**

I. Introduction

As the High Performance Computing (HPC) community prepares for exascale, modern state-of-the-art systems are increasingly making use of GPU accelerators in-tandem with multi-core CPUs [1]. So called "hybrid" (CPU+GPU) architectures (e.g., Summit, Sierra) are also referred to as "heterogeneous memory systems," in that they contain different types of computational units [2]. Heterogeneous memory systems provide the potential benefit of significantly increasing system performance and reducing power consumption at the cost of complexity in their architectures [3]. In today's high-performance computers, there is a wide range of diversity in hybrid system architectures that make it particularly challenging to exploit these machines fully for computational science in a broad range of HPC applications [4].

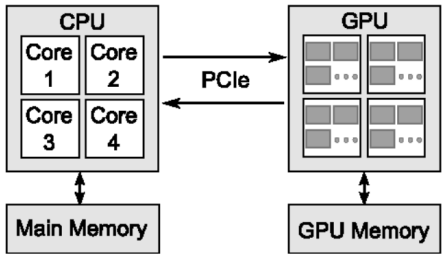


Figure 1. Hybrid CPU GPU Architecture [5]

Thanks to standardization in Application Programming Interface's (APIs), such as Message Passing Interface (MPI), issues regarding *portability* are "not as serious as in years past" [6]. Portability can be defined as a parallel programs' ability to run efficiently on different HPC systems, by making compile-time changes [6], [7]. With the intricacy in today's modern systems, portability issues are again a challenge, as these systems have ultimately surpassed standardized programming abstractions [1], [3], [4], [8]. For example, substantial code refactoring is often required in order to run the same program across HPC machines with diversified heterogeneous architectures, hardware often requires substantial code refactoring [7]. This usually means that a CPU+GPU program that is very efficient on one type of architecture, has to be rewritten before it is able to efficiently run on a different system with other types GPUs (AMD/Nvidia).

Rewriting code for large-scale scientific applications can become very time consuming, as typical HPC programs are 300K-600k lines of code [7] [8]. One way to address this issue, as noted by the 2020 Exascale Computing Project (ECP) Software Technology Capability Assessment Report, is to have compilers generate optimized code for numerous architectures [8]. HPC programs that can run on modern heterogeneous architectures without losing notable performance are considered *performance portable* [7] [8].

This paper will discuss the use of Kokkos, which is a C++ performance portability framework that provides performance portability for HPC applications on modern system architectures. Porting an application to Kokkos enables the program to make use of advanced metaprogramming techniques via Kokkos parallel abstractions in order to produce hardware-specific code at compile time [7]. Syntactically, programs that utilize *Kokkos* consist of the same code for different archi-

tectures, including those that are heterogeneous. The desired outcome of having parallel HPC applications use Kokkos, is to allow the application to effectively utilize different parallelization implementations for the specific architecture, without rewriting thousands of lines of code.

The purpose of this paper is to test the performance portability of Kokkos + MPI on a heterogeneous system with multiple NVIDIA GPUs. This is carried out by solving two common algorithms in HPC (matrix-vector operations and two-dimensional Domain Decomposition). The work described in this study begins with writing a unique MPI-only solution with user-defined MPI communicators, and then ports to Kokkos to compare and contrast implementation techniques and performance portability. Though heterogeneous architectures can comprise of multiple types of accelerators, the scope of this study is focused on heterogeneous (CPU + GPU) architectures. The long term goal of is to create a framework to improve the performance and portability of HPC applications on modern systems with heterogeneous cluster architecture. Future work will take another step towards this goal and consist of further optimizations.

A. Research Questions and Objectives

1) Can the implementation of the MPI Solution remain the same when adding Kokkos parallel execution bodies?
2) How is data communicated to and from the CPU/GPU when using Kokkos shared memory in combination with MPI distributed memory?
3) Under what condition can we accelerate an existing MPI application with GPUs when performing matrix-vector operations on square matrices decomposed to square process grids?
4) What key features from the Kokkos Ecosystem should be emphasized to perform GPU operations on heterogeneous compute clusters?

B. Outline of the Paper

The remainder of this paper is organized as follows: Section II provides context for the study by briefly describing key topics necessary for understanding the experiment and information reviewing similar work; Section III presents the systematic approach of designing the algorithms and compiling for Kokkos+GPUs; Section IV reviews the performance results of square matrix sizes and process domains; Section V discusses the findings and discusses the results; Section VI provides the summary and conclusions of the study, and then outlines future work.

II. Background

This sections briefly outlines key integrates necessary for interpretation of the following sections.

A. Matrix Composition

In High Performance Computing, the objective is to aggregate computational power to solve large-scale scientific problems [9]. The underlying basis for many of these scientific problems is a two-dimensional array of numbers with m rows and n columns, or a *matrix* [10]. The rows and columns (m, n) can also be referred to as a the *dimensions* of a matrix. Thus, a matrix with the dimensions $(3, 3)$ can be seen as the following.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{2.1}$$

B. Matrix-Vector Multiplication

Matrices and matrix operations are key to many important HPC applications, as they provide a means of reducing computationally taxing problems, to operations on matrices [10]. Matrix-vector multiplication, in particular, can be used to solve systems of linear equations [10]. For example, multiplying a Matrix A by a vector x results in a new vector y, as seen below.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_{11} \\ x_{21} \\ x_{31} \end{pmatrix} = \begin{pmatrix} y_{11} \\ y_{21} \\ y_{31} \end{pmatrix} \tag{2.2}$$

The following block of pseudoocode represents serial matrix-vector multiplication to be executed sequentially.

```
    for (i = 0; i < M; ++i) {
        c[i] = 0;
        for (j = 0; j < N; ++j) {
            y[i] = c[i] + A[i][j]*x[j];
    }
}
```

C. Parallel Matrix-Vector Multiplication

Compute performance can be enhanced in [2 eq. (2)] by breaking down the problem into separate parts and solving each part concurrently (parallel computing) [6]. In greater detail, multiple compute resources assume a smaller subset of instructions to be executed individually. After this execution, we can solve the computational problem in its entirety by reconstructing the individual solutions that were computed by each subset of instructions. In the context of parallel computing, a domain decomposition refers to partitioning of computational work among multiple processors by distributing the computational domain (space) of a problem [11]. Figure 2 builds on [2 eq. (2)] by changing it to a row-wise block-striped domain decomposition.

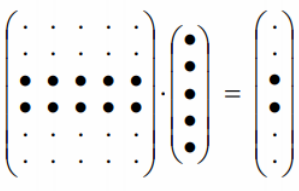


Figure 2. Row-wise block-striped decomposition [12]

Here, data is distributed amongst P processors, each process is responsible for a contiguous part of about N/P rows of A.

Each process gets some rows of the matrix A, and the entire multiplying vector x to compute that processes' components of the product vector y. The program designed in III does something similar, to figure 2, and is described in the next section.

D. Shared Memory vs Distributed Memory

In shared memory systems, all individual processors have access to the same main memory via a system bus. Theoretically, this means the solution to the problem above could end here, since all processes are able to access the memory address of solution vector y. However, in distributed memory systems each processor stores memory from computations locally. Thus, in order to acquire the result of computations performed on other CPUs, processors must communicate with each other via a network [6]. MPI is the standard interface for this interprocess communication on distributed memory networks [13]. In the example illustrated in this paper, MPI is used to communicate smaller subsets of the problem to each individual processor. Furthermore, we demonstrate interprocess communication with MPI allows the programmer to determine which processors store the final memory address of solution vector y.

To facilitate the development of appropriate solutions on numerous system architectures, parallel programming models exist as an abstraction for expressing algorithms with respect to the hardware [6]. This work reflects what is considered to be a " hybrid " parallel programming model, as it combines both shared and distributed memory programming on a heterogeneous (CPU-GPU) architecture. In essence, the combination of utilizing Kokkos shared memory abstractions and MPI distributed memory practices establishes a hybrid parallel programming model. We exemplify the nature of this programming model by offloading computationally intensive kernels to the GPU(s) on-node. After executing the required operations on the GPU to achieve speedup, MPI communicates the pertinent data to reflect the solution across all processors. The approach that has just been defined is realized in the hybrid programming model visualization below. [6, Fig(3)].

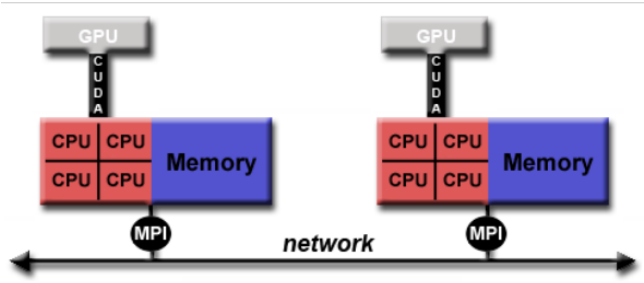


Figure 3. Hybrid Programming model [6]

Note that the program developed for this study is written using the Kokkos performance portability Ecosystem and therefore is not confined to CUDA in the above figure

E. Related Work

1) ***Second-generation polyalgorithms for parallel dense-matrix multiplication*** [14]
The matrix-vector operations presented in this paper were designed to reflect selected linear algebra algorithms presented by Nansamba in Nansamba's graduate thesis publication. In this publication, Nansamba reviews, re-tunes, re-validates, and extends these parallel linear algebra algorithms with MPI. In doing so, Nansamba effectively makes use of the numerous compute resources that distributed memory systems inherently possess, in order to attain strong scalability. We follow Nansamba's approach of establishing two-dimensional logical process grids of MPI processes in our two-dimensional domain decomposition. This strategy enables us to map individual subsets of matrix-vector multiplication algorithms onto confined process grid meshes using the Message Passing Interface (MPI) to reduce the problem size. The design of Nansamba's experiments contributed greatly to the implementation of this work, and we agree with Nansamba's conclusion that redistributing data within matrices and vectors can deliberately minimize the innate costs of computation.
2) ***An MPI-Based 2D Data Redistribution Library for dense matrices on 2D Logical Grid Topologies with MPI*** [15] In following this chain of scientific progress, the work in this paper additionally references Namugwanya's graduate thesis publication. The scope of Namugwanya's work presented in this publication primarily involves the design of a modern C++ library that is able to optimally restructure MPI process meshes in an efficient manner. With particular regards to this research, we reference Namugwanya's use of utilizing MPI Communicators, in addition to the mathematical procedure Namugwanya proposed for storing two-dimensional data in the presence of a load imbalance. These strategies, along with the proposed data maneuvers that Namugwanya outlined, assisted our approach of accurately solving parallel matrix-vector computations in this work. From Namugwanya's experimentation, Namugwanya further analyzes when is it deemed appropriate to reorder the storage in memory of these data structures, depending on the computational cost to re-establish them. We agree with Namugwanya's findings that distributed memory redistribution costs vary greatly, in comparison to prior implementations. For this reason, we are sure to include time measurements of initialization and redistribution of the data. Defining these execution times help us to investigate the portability of this programming model, and ideally should be re-assessed and re-visited in future work. In section III, we expand on Namugwanya's definitions of: a Data Grid, a Process Grid, and Data Mapping in greater detail. At last, Section IV, Figure 6 illustrates the handling of distributed matrix data in the presence of a load imbalance.

III. Experimental Design

A. 2D Domain Decomposition with MPI

This section presents the design methodology for the program. As explained in [14], [15], partitioning data to MPI processes can be illustrated with three key concepts: a global data grid; a local process grid, and the mapping of data on multiple MPI processes. In both the MPI implementation and the Kokkos GPU implementation discussed in section III, two-dimensional data is mapped from a global data grid onto a local process grid such that each process allocates a piece of the global data.

Data Grid: The global data grid represents a matrix described in [2 (eq(1)]. To illustrate a two-dimensional decomposition of this grid, let matrix A represent a global data grid where M denotes the number of rows in the grid, and N refers to the total number of columns. Coordinates of a particular element (i, j) are then used to reference some data at the location $(i, j) \in A$, where i denotes the row of a given element, and j denotes the column. For example, the coordinates that correspond to the location Element 13 in this data grid are $i = 2;\ j = 3$. In other words, to access element 13 in Matrix A, the notation $A(2, 3)$ is used [15].

0	1	1	2	3
2	3	4	5	6
2	3	3	4	5
4	5	6	7	8
6	7	9	10	11

Figure 4. Example Data Grid $M = 5, N = 5$

Process Grid: The process grid depicts the local mesh of processes assigned to two dimensions P Q, also representing a matrix. Here we extend the abstraction with the variables P, Q, where P is the total number or rows for a given MPI process, and Q is the number of columns for the MPI process. Further, $P * Q$ should be the total number of processes that are at executed at runtime. Elements that exist within the process grid can also be referenced by the notation (i, j). For transparency, we will call the i, j for the process grid $(local\_row,\ local\_col)$. Here, $local\_row$ delineates the dimension P and $local\_col$ represents dimension Q. Thus, for a given process grid shape P x Q, the grid has indexes $local\_row$ such that $0 \leq local\_row < P$ and $local\_col$ such that $0 \leq j < Q$, respectively [15].

Below, figure 5 simply depicts the distribution of 4 MPI processes $2x2$ process grid. Here, each color represents the dimension for a unique MPI process to compute. Accessing elements now rely on the newly defined dimensions, such that the green dimension has the following mesh coordinates $local\_row = 1, local\_col = 1$.

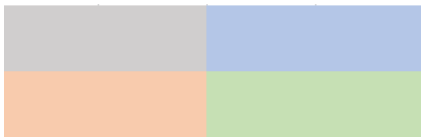


Figure 5. Example Process grid $P = 2; Q = 2$

Data Allocation: Partitioning the data refers to distributing the global matrix data onto smaller, localized, two-dimensional MPI process grid meshes. This can be done by dividing

$$rows\_per\_process = M/P$$

$$columns\_per\_process = N/Q$$

In doing so, each MPI process initializes a piece of the global matrix. However, in the event the data grid dimensions MN are not evenly divisible by grid dimensions PQ, the modulo operator can be used to add remaining rows or columns.

$$remaining\_rows = M\%P$$

$$remaining\_cols = N\%Q$$

Given the data grid from Figure 4, its mapping on a $2x2$ process process grid is illustrated in Figure 7.

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Figure 6. An example of a 5 x 5 data grid mapped to a 2 x 2 process grid

In order to analyze the *performance portability* of the Kokkos framework, in addition to performance comparisons of the CPU vs the GPU, the MPI implementation is developed first. After the MPI implementation was tested for correctness, it was ported to kokkos. Thus, both the MPI implementation and the Kokkos+MPI+GPU implementation are detailed in turn.

B. MPI Implementation

$$y_m = A_{mn}x_n \quad eq(1)$$

$$x_m = I_{mn}y_n \quad eq(2)$$

The design methodology for solving equation (1) and (2) begins by constructing local process' matrix from a global data grid as described in Figure 6. x and y are vectors of either the dimension m or n, and A is a matrix of the dimensions m, n. These dimensions derive from global grid sizes set at run-time, and enable each MPI process to divide the global domain into local subdomains. The variables M, N, P, Q are read in, the data for matrix A is mapped to a process grid PQ. We will continue with Figure 6 to illustrate,

such that $M = 5$ $N = 5$ $P = 2$ $Q = 2$

Calculations for each process's chunk of matrix A are illustrated in the code below

```
int local_row = world_rank / Q;
int local_col = world_rank % Q;
 int m, n; // shape of matrix in process
     grid
if (local_row < P - 1) {
  m = MdivP;
} // if everything divides evenly
else {
  m = MdivP + MmodP;
} // add whats left
if (local_col < Q - 1) {
  n = NdivQ;
} // same logic here
else {
  n = NdivQ + NmodQ;
}
```

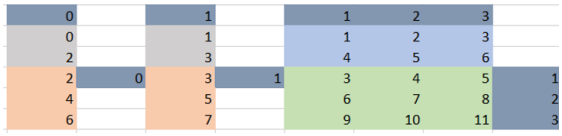
Listing 1. Process Chunk Calculation

Each process gets its own chunk of the global data in two dimensions and attempts to split the work evenly amongst the number of processes. When work is split evenly, speed up is achieved when the available processors are partitioned as efficiently as possible. Initializing A this way is is to avoid the cost of having one MPI process construct the entire matrix A. Instead, the work is partitioned according to the number of processes to a process grid. Mapping MPI processes in this manner helps us to achieve scalable performance with larger matrix sizes [14], [15].

In the equation

$$y_m = A_{mn}x_n \quad eq(1)$$

Vector x is built on process row 0, and then broadcasted with the column communicator.



Rank 0	Rank 1	Rank 2	Rank 3
0	1	0	0
1	2	1	1
	3		2

Figure 7. Ax

After this, each process performs the dot operations, and MPI Reduce is called so store the product of Ax as y.

$$x_m = I_{mn}y_n \quad eq(2)$$

In [3, eq. (2)], the process is very similar, but the global coordinates to make $I(mn)$ are needed. This is done with the following block of code.

```
int i_increment = M / P;
int j_increment = N / Q;
for (int local_i = 0; local_i < m; ++
    local_i) {
    for (int local_j = 0; local_j < n; ++
        local_j) {
        int global_i = (i_increment *
            local_row + local_i);
        int global_j = (j_increment *
            local_col + local_j);
        if (global_i == global_j) {
        id_A(local_i, local_j) = 1;
        } else
        id_A(local_i, local_j) = 0;
    }
}
```

Listing 2. Local to Global Index

MPI_AllReduce is then used to store vector x in every process column.

C. Port To Kokkos

Once the matrix-vector multiplication equations have been solved, we port to Kokkos. The first thing to note about this process is that class definitions are no longer needed (though could potentially be useful). For instance, in the MPI implementation, matrix A and vector y were defined with the following classes.

```
/* Filename: densematrix.h */
template <typename T> class DenseMatrix {
public:
  /* constructor*/
  DenseMatrix(size_t _m, size_t _n)
   : m(_m), n(_n), matrixData(new T[_m * _n
       ]) {}

  /* destructor */
  virtual ~DenseMatrix() { delete[]
      matrixData; }

  /* copy assignment */
  DenseMatrix &operator=(const DenseMatrix<
      T> &rhs);

  /* row major assignment */
  T &operator()(int i, int j) {
    return matrixData[i * n + j];
  }
  /* return data row major */
  T operator()(int i, int j) const {
    return matrixData[i * n + j];
  }

  /* for mpi calls*/
  T *get_Matrix() { return &matrixData[0];
      }
  /* accessors */
  size_t Rows() const { return m; }
  size_t Cols() const { return n; }

private:
  /* need size values: M rows, N columns */
  size_t m, n;
```

```
  T *matrixData;
};
```

Listing 3. Matrix Definition used in MPI Implementation

```
/* Filename: densevector.h */
template <typename T> class DenseVector {
public:
  /* constructor */
  DenseVector(size_t _length);

  /* destructor */
  virtual ~DenseVector();

   /* assign data */
  T &operator()(size_t index);

  /* return data */
  const T &operator()(size_t index) const;

  /* accessor */
  size_t size() const { return length; }

   /* for mpi calls */
  T *get_Vector() { return &vectordata[0];
      }
  }
*/

private:
  size_t length; // size of vector
  T *vectordata; // pointer to vector
};
```

Listing 4. Vector Definition used in MPI Implementation

These classes were not used in the Kokkos version, in order to utilize Kokkos' own data structures that provide performance portability features. The features used are discussed below.

Kokkos Execution Spaces: Kokkos uses the term execution spaces to describe a logical grouping of computation units which share an identical set of performance properties [7].Aninstanceof an execution space is a specific instantiation of an execution space to which a programmer can target parallel work. Essentially, this provides a homogeneous set of execution resources in addition to the mechanism used to perform the execution. In layman's terms, an execution space defines the place and the way to run a particular block of code. These blocks of code are determined by calling *Kokkos::parallelfor()*, which is a Kokkos core construct that computes in parallel according to the execution policy. One important note is that everything outside of kokkos parallel constructs is run by the host process. Typically, the *host* process does things like MPI calls or I/O. Parallel code is run by one of kokkos's execution spaces which is specified at compile time, and if not specified, its set to default execution space. The programmer can control or change the default execution space simply by passing the *RangePolicy* parameter. Below is an example of a Kokkos::parallel_for() construct with a default execution space, followed by a custom execution space.

```
---DEFAULT Execution Space---
parallel_for("Label",
     numberofIntervals,
     [=] (const int64_t i) {
    body
    });
 ---CUSTOM Execution Space---
parallel_for("Label",
     RangePolicy<ExecutionSpace>
    (0,numberofIntervals),
    [=] (const int64_t i) {
    body
    });
```

2) Kokkos Memory spaces: Are explicitly manageable memory resources that give the programmer a means to control or determine where data resides. Some available memory spaces that Kokkos provides are: Host Space, Cuda space, CudaUVM space. Each execution space has a default memory space associated with it, and should be the memory space that the execution space can access the best. In code, we use typedefs to decide what memory spaces to use depending on what we compile for.

3) Kokkos Views: One of the most critical ways to manage data in a Kokkos Program is a Kokkos *View*. A particular benefit of using the view abstraction in a heterogeneous programming environment derives from how the view stores data in a memory space. This memory space is set at compile time and can be used to uniquely identify and determine which data belongs to the CPU, and which data belongs to the GPU, amongst other storage targets. In essence, *views* are simply a lightweight C++ class that of which accommodate a templated pointer to array data [7]. In practice, views provide a procedure to effectively move data back and forth amongst different memory spaces within the program. This is done by creating a *mirror view*, which are simply copies of templated arrays that are capable of residing in different memory spaces. Fundamentally, views behave just like C++11 smart pointers. For example, if a programmer makes two host mirror views from a view that is allocated on CudaSpace, they will be two different allocations in that these two new views are either copy constructed, or, allocated in the defined memory space. Adversely, if you create the second view from the first host mirror view, they'll point to the same data. One important note is that views should be copied by value directly into functors.

In the program designed for this study, the code for the performance portability features above were defined as in the following block of code.

```
int main(){
    using ExecSpace = Kokkos::Cuda
    using MemSpace = Kokkos::CudaSpace;
    using Layout = Kokkos::LayoutRight;
    using range_policy = Kokkos::
        RangePolicy<ExecSpace>;
    using ViewVectorType = Kokkos::View
    <double *, Layout, MemSpace>;
```

```
    using ViewMatrixType = Kokkos::View
    <double **, Layout, MemSpace>;
}
```

Listing 5. Kokkos View Initialization

Kokkos Mirror View: Views can have a HostMirror typedef which is a view type with compatible layout inside the HostSpace. Additionally there is a create_mirror and create_mirror_view function which allocate views of the HostMirror type of a view [7]. In effect, this abstraction allows programs to use execute code on different computation units such as the GPU. The difference between the two is that create_mirror will always allocate a new view, while create_mirror_view will only create a new view if the original one is not in the HostSpace [7]. Pertaining to this topic, its particularly useful to utilize mirror views if not using GPU aware MPI.

MPI in a Kokkos Environment The MPI Interface uses raw pointers [7]. This means that data needs to be stored contiguously, and if the MPI is not GPU Aware, the program will need to copy to the host with *create mirror view*. Normal MPI functions can be used, but MPI must be initialized before Kokkos [7]. For example:

```
int main ( int argc , char * argv [])
{
    MPI_Init (& argc ,& argv );
    Kokkos :: initialize (argc , argv );
    [magic]
    Kokkos :: finalize ();
    MPI_Finalize ();
}
```

Listing 6. Kokkos+MPI Skeleton code

Lastly, we change the multiplication functions in our initial design to Kokkos::parallel_reduce() for performance portability.

```
/*   eq(1)    */
template <typename T>
void yAx(const DenseMatrix<T> &A, const
    DenseVector<T> &x, DenseVector<T> &y) {
  for (size_t i = 0; i < A.Rows(); ++i) {
    double tempval = 0;
    for (size_t j = 0; j < A.Cols(); ++j) {
      tempval += A(i, j) * x(j);
    }
    y(i) = tempval;
  }

/*   eq(2)    */
template <typename T>
void xIDy(const DenseMatrix<T> &id_A, const
     DenseVector<T> &y,
          DenseVector<T> &x) {
  for (size_t i = 0; i < id_A.Rows(); ++i)
      {
    double temp = 0.0;
    for (size_t j = 0; j < id_A.Cols(); ++j
        ) {
      x(i) += id_A(i, j) * y(i);
    }
  }
}
```

Listing 7. Solving Matrix-Vector Operations Before Kokkos

```
/*   eq(1)    */
double result = 0;
    Kokkos::parallel_reduce(
        "yAx", m,
        KOKKOS_LAMBDA(int i,
                      double &update) {
          double temp2 = 0;
          for (int j = 0; j < n; ++j) {
           update += A(i, j) * x(j);
          }
         y(i) = update;
},
        result);
     Kokkos::fence(); //synchronize devices
    Kokkos::deep_copy(h_y, y); // copy back
         to host fom device

/*   eq(2)    */
double result = 0;
    Kokkos::parallel_reduce(
        "yAx", m,
        KOKKOS_LAMBDA(int i,
                      double &update) {
          double temp2 = 0;
          for (int j = 0; j < n; ++j) {
           update += A(i, j) * x(j);
          }
         y(i) = update;
},
        result);
     Kokkos::fence(); //synchronize devices
    Kokkos::deep_copy(h_y, y); // copy back
         to host fom device
```

Listing 8. Solving Matrix-Vector Operations After Kokkos

REFERENCES

[1] V. Artigues, K. Kormann, M. Rampp, and K. Reuter, "Evaluation of performance portability frameworks for the implementation of a particle-in-cell code," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 11, p. e5640, 2020.

[2] "Heterogeneous computing system," 2013. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/heterogeneous-computing-system

[3] "Heterogeneous – the benefits to performance and power consumption: Ieee computer society." [Online]. Available: https://www.computer.org/publications/tech-news/heterogeneous-system-architecture/heterogeneous%E2%80%93the-benefits-to-performance-and-power-consumption

[4] L. G. Jack, Dongarra and N. J. Higham, "Numerical algorithms for high-performance computational science," Jan 2020. [Online]. Available: https://royalsocietypublishing.org/doi/abs/10.1098/rsta.2019.0066

[5] E. L. Padoin, L. Pilla, F. Z. Boito, R. Kassick, P. Velho, and P. Navaux, "Evaluating application performance and energy consumption on hybrid cpu+gpu architecture," *Cluster Computing*, vol. 16, pp. 511–525, 2012.

[6] "Introduction to parallel computing tutorial." [Online]. Available: https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial#HybridMemory

[7] Kokkos, "kokkos/kokkos." [Online]. Available: https://github.com/kokkos/kokkos/wiki

[8] M. Heroux, L. McInnes, and R. Thakur, "Ecp software technology capability assessment report," Nov 2020. [Online]. Available: https://www.osti.gov/servlets/purl/1463232/

[9] "What is high performance computing?" [Online]. Available: https://www.usgs.gov/core-science-systems/sas/arc/about/what-high-performance-computing

[10] D. of Mathematics and C. Science, *Matrix Multiplication*. Gordon College, 2020. [Online]. Available: http://www.math-cs.gordon.edu/courses/cps343/presentations/Matrix_Mult.pdf

[11] G. Thomas, *Domain Decomposition*. Boston, MA: Springer US, 2011, pp. 78–87. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_291

[12] D. R.-P. Mundani, "Pdf," 2008.

[13] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 2014.

[14] G. Nansamba, "Second-generation polyalgorithms for parallel dense-matrix multiplication. masters thesis, the university of tennessee at chattanooga," Master's thesis, University of Tennessee at Chattanooga, 2020.

[15] E. Namugwanya, "An mpi-based 2d data redistribution library for dense arrays," Aug 2021. [Online]. Available: https://scholar.utc.edu/theses/726/