Implementation of Mesh-Free Algorithms for Field Transfer Applications in GPUs

Abhiyan Paudel[*]
Rensselaer Polytechnic Institute
Troy, USA
paudea@rpi.edu

Sinanzwayinkosi Professor Ndhlovu[*]
Rensselaer Polytechnic Institute
Troy, USA
ndhlos@rpi.edu

ABSTRACT

This project deals with the implementation of mesh-free algorithms on GPUs, focusing on the moving least squares (MLS) algorithm for field mapping between source and target domains. Mesh-free methods are popular for their simplicity and ease of implementation as they don't require mesh topology information, making them well-suited for field data transfer tasks. Leveraging the computational power of NVIDIA's Tesla V100 GPUs, the project achieved substantial speedups. An impressive speedup factor of 875 was attained with 24 MPI ranks/GPUs compared to a single GPU/MPI rank setup. Furthermore, weak scaling experiments revealed an efficiency close to 1 for 24 GPUs, indicating the successful utilization of additional computational resources to improve performance. The project aimed to assess the performance of mesh-free methods in heterogeneous computing platforms by devising parallelization strategies employing MPI, MPI/O, and CUDA. This includes optimizing algorithms for efficient data transfer and processing, particularly focusing on the MLS reconstruction. The findings underscore the remarkable performance potential of parallelized MLS reconstruction across both strong and weak scalability scenarios, particularly in situations with balanced workloads.

KEYWORDS

Interpolation, Mesh Free Algorithm, Moving Least Squares

ACM Reference Format:
Abhiyan Paudel and Sinanzwayinkosi Professor Ndhlovu. 2024. Implementation of Mesh-Free Algorithms for Field Transfer Applications in GPUs. In *Proceedings of* . ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In multiphysics simulations, the data transfer operation is inevitable when we use the partitioned approach in which an individual solver with its numerical discretization methods is employed. The methods used for data transfer should be scalable on high-performance computing hardware. In addition to scalability, these approaches should be accurate and conservative. There are mainly two ways to carry out field transfer between two physical domains: mesh-based and mesh-free approaches.

Several research groups have developed methods to handle interfaces where one side is discretized with a mesh. These methods aim to maintain both conservation and accuracy properties. Typically, these methods involve solving a weighted residual problem, where the accuracy and conservation properties depend on the numerical integration method used to assemble the problem and the accuracy of the functional support in the underlying physics discretization [1, 3]. Parallelism in these methods has been approached by leveraging tools and techniques from parallel finite element assembly. This means the problem is divided into smaller tasks that can be solved concurrently, utilizing multiple processing units to speed up computation. By effectively distributing the workload across multiple processors, parallel finite element assembly allows for efficient handling of large-scale simulations while preserving accuracy and conservation properties. Jiao and Heath [5] have explored common-refinement-based data transfer between non-matching meshes in multiphysics simulations.

While the mesh-based method is known for its conservative and accurate nature, it does come with a drawback: the requirement for topological information. Dealing with such information can sometimes be cumbersome. In contrast, mesh-free methods offer a topology-independent approach to transferring data between two spaces represented by simple point clouds. This eliminates the need to adhere to a specific numerical solution scheme. This simplicity extends to computational usability. Mesh-free methods excel in scenarios where data transfer occurs between particle-based and continuum-based methods, such as kinetic Boltzmann or Navier-Stokes equations.

Among different mesh-free methods, the Spline interpolation method is quite popular. This method is has been widely employed for function reconstruction through either global or local approximations. Extensive discussion on this method is found in [7]. In recent multiphysics analyses, Slattery [9], Zhang et al. [12] and Lancaster et al. [6] investigated conservative global spline methods, based on the linear combination of RBF and polynomial terms. These studies consistently demonstrate high accuracy and conservation while allowing for straightforward parallel computational implementation. Slattery [9] delves into the realm of mesh-free data transfer algorithms for partitioned grids, shedding light on the significance of accuracy and conservation in numerical methods for data transfer. The study compares three algorithms, including the weighted residual technique with the common-refinement method and two mesh-free methods with compact support. The study notably expands the application of mesh-free methods to three-dimensional problems to determine the most efficient approach across different scenarios. The author has implemented mesh-free methods utilizing a distributed system. Similarly, another mesh-free method, the Moving Least

[*]Both authors contributed equally to this research.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Squares Method (MLS), initially proposed by Lancaster and Salkauskas [6], performs local approximation by combining polynomial terms conservatively, minimizing the global squared error norm. As illustrated in Quaranta et al. [8], the residual-weighted approach employs RBFs as weight functions. Both methods share the characteristic of defining a compact support for the RBFs. Moreover, RBF finds application as a primary interpolation technique, as demonstrated by Fontberg and Flyer [4], who utilized the Finite Difference Method with RBFs to solve geoscience problems like wave propagation, diffusion, and convection. It's worth noting that the accuracy and conservation of RBFs heavily depend on the support domain size. Recent studies, such as Zhang et al. [12], aim to enhance adaptability by predicting local error norms and adjusting the compact support domain size accordingly.

Our project aims to enhance the mesh-free algorithm by integrating it into a heterogeneous system. This setup involves the CPU handling data reading and decomposition, while the majority of computations are delegated to GPUs. Specifically, we have employed the Moving Least Squares (MLS) method to modern GPUs using Kokkos library in NVIDIA GPUs. The primary objective is to showcase the GPU implementation of the MLS method and perform the performance analyses. We have used simple 2D scenarios with two different mesh topologies to illustrate the computational efficiency of the parallel MLS algorithm on GPUs for field transfer applications. Input data consist of nodal data from a distributed mesh, with data mapping executed on the GPUs.

2 RBF BASED MOVING LEAST SQUARES

In this project, we've integrated Radial Basis Function (RBF)-based moving least squares techniques to map function values from a source field to a target field. This approach employs radial basis functions within the local supports of target points derived from the source field. The subsequent section provides a comprehensive overview of the radial basis functions tailored for this particular method.

2.1 Locally supported radial basis functions

Contrary to the weighted residual technique, mesh-free methods for data transfer don't rely on discretizing fields to construct the interpolant. Instead, they represent the domain using arbitrary point clouds with functional support that's independent of topology, defined solely for interpolation purposes. Radial basis functions enable the creation of this functional support within arbitrary point clouds by utilizing the Euclidean distance, denoted as r, between a point i and any of its supporting points j.

$$r = \left\|\mathbf{x}_i - \mathbf{x}_j\right\|_2 \tag{1}$$

where $\mathbf{x}_i$ and $\mathbf{x}_j$ are the physical coordinates of the points i and j respectively. In our work, we have used Wu's C^4 function as a radial basis function,

$$\phi(r) = \left(5\left(\frac{r}{\rho}\right)^5 + 30\left(\frac{r}{\rho}\right)^4 + 72\left(\frac{r}{\rho}\right)^3 + 82\left(\frac{r}{\rho}\right)^2 + 36\left(\frac{r}{\rho}\right) + 6\right)\left(1 - \frac{r}{\rho}\right)_+^6 \tag{2}$$

Here, ρ represents the physical support radius of the function, and the notation $(\cdot)_+$ on the second term indicates that the term evaluates to zero if $(1 - r/\rho) < 0$, effectively truncating support at the boundary of the physical support radius. The choice of ρ determines the number of control points used for support, with a higher number typically resulting in better interpolation. However, for cases where the grids in the data transfer have non-uniform spacing, such as those with local refinement, a fixed support radius may not be ideal. In such scenarios, locations with little refinement might lack sufficient support, while those with heavy support can be computationally intensive. Instead, one can fix the size of the local support and compute the local radius from the supporting points. Throughout this paper, we maintain a fixed radius of support.

2.2 Moving least square method

Quaranta et al. [8] introduced a function reconstruction approach for arbitrary point clouds using a moving least squares discretization. This technique establishes support and, thus, the data transfer operator through solutions to local least squares kernels defined by compactly supported radial basis functions. The solution at each target point is initially expressed as a polynomial sum defined by,

$$\mathbf{g}_i = \sum_{i=1}^{m} p_i\left(\mathbf{t}_i\right) a_i\left(\mathbf{t}_i\right) \tag{3}$$

where the a_i are the polynomial coefficients and the $p_i\left(\mathbf{t}_i\right)$ are quadratic polynomials in terms of spatial coordinates In vector form for an arbitrary three-dimensional point, q, these polynomials are:

$$\mathbf{p}(q) = \begin{bmatrix} 1 & x_q & y_q & x_q^2 & x_q y_q & y_q^2 \end{bmatrix}^T \tag{4}$$

Then the interpolant is crated by minimizing the data transfer residual through a weighted least square procedure at each individual target point, $\mathbf{t}_i$:

$$\text{Minimize } \frac{\partial}{\partial a_i\left(\mathbf{t}_i\right)} \int_{\Omega} \phi\left(\mathbf{t}_i - \mathbf{s}\right)(\mathbf{g} - \mathbf{f})^2 d\Omega(\mathbf{s}) \tag{5}$$

where the $a_i\left(\mathbf{t}_i\right)$ is free parameters in the minimization, $\Omega(\mathbf{s})$ is the domain of interest specifically defined by the supporting source control points, and compactly supported radial basis functions defined at the target control point and supported by the source control points, $\phi\left(\mathbf{t}_i - \mathbf{s}\right)$, serve as the weights in the minimization. To close the problem, Eq.3 provides a natural constraint for the minimization procedure. Solving the minimization problem gives the following kernel to be evaluated at each target control point:

$$\xi\left(t_i\right) = \mathbf{p}\left(t_i\right)^T \left[\mathbf{P}\left(s_{t_i}\right)^T \mathbf{\Phi}\left(s_{t_i}\right) \mathbf{P}\left(s_{t_i}\right)\right]^{-1} \mathbf{P}\left(s_{t_i}\right)^T \mathbf{\Phi}\left(s_{t_i}\right) \tag{6}$$

with s_{t_i} the set of n source points within the support radius of the i th target point, $\mathbf{t_i}$. The polynomial component is defined as:

$$\mathbf{P}\left(s_{t_i}\right) = \begin{bmatrix} \mathbf{p}\left(s_{t_1}\right) \\ \mathbf{p}\left(s_{t_2}\right) \\ \cdots \\ \mathbf{p}\left(s_{t_n}\right) \end{bmatrix} \tag{7}$$

and the basis component:

$$\mathbf{\Phi}\left(s_{t_i}\right) = \begin{bmatrix} \phi_{s_{t_1} t_i} & 0 & \cdots & 0 \\ 0 & \phi_{s_{t_2} t_i} & \cdots & 0 \\ \cdots & \cdots & \ddots & \cdots \\ 0 & 0 & \cdots & \phi_{s_{t_n} t_i} \end{bmatrix}$$

We can then reconstruct the target function through independent evaluations of the moving least square kernel in Eq. (38) at each target control point. Alternatively, we can store the solutions to the kernels as a matrix to form the data transfer operator:

$$H_{ij} = \xi(\mathbf{t}_i)_j, \iff \mathbf{s}_j \in s_{t_i} \tag{8}$$

where i is the index of the target control point and j the index of the source control point providing support within the radius of $\mathbf{t}_i$. Unlike spline interpolation technique, this method has no coupling amongst target control points. In addition to this, it also has a naturally local formulation without the need for a global linear solve or a partition of unity method to enable locality. In other words, the structure of the resulting linear operator, $\mathbf{H}$, in Eq. 8 is sparse because radial basis support only links target control centers to neighboring source control centers rather than a linking to both neighboring source and target control points. This property renders it well-suited for GPU platforms, allowing localized target data computation using a team policy in Kokkos.

3 KOKKOS

Kokkos is a C++ programming model to achieve performance portability, as highlighted in [11]. It offers a shared-memory parallel programming paradigm, allowing code to be written once and executed on diverse architectures, including multi-core CPUs, NVIDIA GPUs, Xeon Phi, etc. The primary objective of Kokkos is to maintain performance consistency across different platforms while preserving efficiency. It stays aligned with advancements in the C++ standard, ensuring adaptability to evolving requirements in scientific and engineering codes. Kokkos provides abstractions for data structures, facilitating efficient memory access patterns.

Within Kokkos, memory spaces define the location of data (e.g., High Bandwidth Memory, DDR, Non-Volatile, Scratch), while execution spaces determine where computations occur (CPU, GPU, or custom executors) [10]. Memory layouts (row/column-major, tiled, or strided) and memory traits (like streaming or atomic operations) influence execution strategies. Kokkos supports parallel constructs such as `parallel_for`, `parallel_reduce`, and task-spawning. It employs Policies to express parallelism, including options for tightly and non-tightly nested loops. Kokkos supports both hierarchical and flat parallelism.

In Kokkos, users can specify different memory spaces for data allocations. For instance, Kokkos offers a dedicated memory space called CudaUVMSpace, allowing allocations to be accessible from both the host and the device. Developers can configure Kokkos to utilize this space as the default for CUDA allocations. In CUDA, memory spaces require explicit management using constructs like cudaMalloc and cudaMemcpy, necessitating developers to explicitly handle memory transfers between the host and the device.

3.1 Kokkos Support for CUDA Programming

Kokkos provides robust support for CUDA programming by offering a high-level abstraction layer that simplifies the development of parallel code targeting CUDA-enabled GPUs [2]. Here's how Kokkos supports CUDA programming with its features and syntax:

3.1.1 *Execution Policies.*

Kokkos introduces execution policies, which define where and how parallel code should be executed. Users can specify execution policies to control how parallel loops are mapped to CUDA threads and blocks.

```
Kokkos::RangePolicy<> policy(0, size);
Kokkos::parallel_for(policy, KOKKOS_LAMBDA(const int i)
       {
    // Parallel code block
});
```

3.1.2 *Memory Spaces.*

Kokkos provides memory spaces such as `Kokkos::CudaSpace`, which represent memory resources on CUDA devices. Developers can allocate and manage memory in these spaces, enabling efficient data transfer and access between the host and CUDA device.

```
Kokkos::View<double*, Kokkos::CudaSpace> array("
     array_name", size);
```

3.1.3 *Parallel Constructs.*

Kokkos offers parallel constructs like `parallel_for`, `parallel_reduce`, and `parallel_scan`, which are similar to CUDA's kernel launches but provide a higher-level abstraction. These constructs enable developers to express parallelism in a platform-independent manner.

```
Kokkos::parallel_for(size, KOKKOS_LAMBDA(const int i) {
    // work body
});
```

3.1.4 *Memory Management.*

Kokkos provides memory management functionalities similar to CUDA, allowing developers to allocate and deallocate memory on CUDA devices. Kokkos abstracts away the details of memory management, making it easier to write portable code.

```
Kokkos::View<double*, Kokkos::CudaSpace> array("
     array_name", size);
```

3.1.5 *Functors.*

Kokkos utilizes functors, which are objects that behave like functions, to encapsulate parallel algorithms. Functors enable code reuse and flexibility in expressing parallelism on CUDA devices.

```
struct Functor {
    KOKKOS_INLINE_FUNCTION
    void operator()(const int i) const {
        // Parallel code block
    }
};
Kokkos::parallel_for(size, Functor());
```

3.1.6 *Team Parallel Patterns.*

Kokkos offers team-level parallel patterns such as `TeamPolicy` and `TeamThreadRange`, which allow for efficient use of shared memory and synchronization within CUDA blocks.

```
Kokkos::TeamPolicy policy(num_teams, team_size);
Kokkos::parallel_for(policy, [=] (const Kokkos::
     TeamPolicy::member_type& team_member) {
    // Team-level parallel computation
});
```

By providing these features, Kokkos simplifies CUDA programming, making it more accessible to developers while ensuring high performance and portability across different architectures.

3.2 Algorithm implementation

3.3 Problem definition

We first generated data for a domain of 100×100 with dimensions of 3000 units in the x-direction and 3000 units in the y-direction, resulting in a set of source points. Additionally, one million target points are generated within this domain. The objective is to evaluate the scalability of MLS algorithm in GPUs using a test function. Specifically, the task is to determine each target point's corresponding source supports. This involves identifying which source points contribute to the computation for each target point. In other words, for each target point, we need to ascertain the nearby source points that influence its computation. This step is crucial for various applications such as interpolation, data mapping, or function approximation, where understanding the influence of neighboring points is essential for accurate results. The challenge lies in efficiently finding these source supports for each target point, especially considering the large number of target points (one million) and potentially numerous source points within the domain. The efficiency of the methodology will be evaluated based on factors such as computational speed, memory usage, and scalability with increasing problem sizes.

3.4 Computational Challenges

Initially, we encountered memory issues when attempting to store the local supports for each target point and their corresponding ids in an array, particularly when dealing with larger problem sizes. Consequently, we opted to modify our approach. Recognizing that computation is less resource-intensive than storage in GPUs, we devised a strategy to evaluate the supports for each target point locally on the fly while calculating the kernel. To implement this strategy, we leveraged the team policy offered by Kokkos. Under this policy, each target point was assigned to a team for each execution of work, with the computation carried out using the threads within each team. This approach allowed us to efficiently perform computations without incurring the memory overhead of storing extensive arrays.

3.5 Kokkos policy usages

Despite the availability of more efficient search algorithms, we opted for a simple $O(n^2)$ approach. We calculated the distance between each target point and all source points within each team to identify the nearest neighbors. We implemented atomic operations to ensure thread safety and prevent race conditions during this operation. Furthermore, the Cholesky algorithm, utilized for finding the inverse, also employed blocking operations such as `Kokkos::single(Kokkos::PerTeam(team))`. This approach was essential as the computation of values depended on previous threads. By employing blocking operations, we mitigated the risk of race conditions and ensured the correctness of the algorithm's results.

Algorithm 1 MatrixInverse()

```
Require: Matrix $A$
Ensure: Inverse of matrix $A$
 1: Compute Cholesky decomposition of $A$: $A = LL^T$
 2: Initialize matrix $X$ as identity matrix
 3: Initialize matrix $A_{\text{inv}}$ as zeros matrix
 4: for $i$ from 1 to $n$ do
 5:     for $j$ from 1 to $n$ do
 6:         if $i = j$ then
 7:             sum = 0
 8:             for $k$ from 1 to $i - 1$ do
 9:                 sum = sum + $L[i,k]^2$
10:             end for
11:             $L[i,i] = \sqrt{A[i,i] - \text{sum}}$
12:         else
13:             sum = 0
14:             for $k$ from 1 to $i - 1$ do
15:                 sum = sum + $L[j,k] \times L[i,k]$
16:             end for
17:             $L[j,i] = \frac{A[j,i] - \text{sum}}{L[i,i]}$
18:         end if
19:     end for
20: end for
21: for $i$ from 1 to $n$ do
22:     for $j$ from 1 to $i - 1$ do
23:         sum = 0
24:         for $k$ from 1 to $j - 1$ do
25:             sum = sum + $L[i,k] \times X[j,k]$
26:         end for
27:         $X[i,j] = \frac{I[i,j] - \text{sum}}{L[i,i]}$
28:     end for
29: end for
30: for $i$ from $n$ to 1 do
31:     for $j$ from $n$ to 1 do
32:         sum = 0
33:         for $k$ from $j + 1$ to $n$ do
34:             sum = sum + $L[j,k] \times A_{\text{inv}}[k,i]$
35:         end for
36:         $A_{\text{inv}}[j,i] = \frac{X[j,i] - \text{sum}}{L[j,j]}$
37:     end for
38: end for
39: return $A_{\text{inv}}$
```

4 PERFORMANCE ANALYSIS

In this study, we employed heterogeneous systems, utilizing CPUs for data reading and GPUs for intensive computation tasks. The number of MPI ranks corresponds to the number of GPUs, with each GPU assigned to a single core. Data is organized in a structured layout, and the domain is partitioned among the MPI ranks. MPI I/O is utilized to read data from files, after which the data is transferred to the GPU for computation.

Given that the computation involves associating target points with source supports, MPI communication becomes necessary. This communication occurs among MPI ranks situated at the boundaries. Specifically, data exchange involves the transfer of five rows of data between neighboring MPI ranks, as illustrated in Figure 1. This exchange facilitates the sharing of information required for computing the target points' source supports.

Performance Analysis entails evaluating how their performance alters with the inclusion of additional processors or cores. This is achieved by executing identical tasks across multiple processors or cores and gauging the total time required for task completion. By comparing this against the performance of the system using just one processor or core, one can gauge the overall efficiency of the system with varying processor or core counts.

4.1 Strong Scaling

Strong scaling refers to how efficiently a program's execution time decreases as the number of processors (or compute resources) increases while keeping the problem size constant

Table 1 gives the MPI communication and MPI I/O overheads across different number of cores. It can be better understood from the figures 2 and 3.

It can be seen from the figure 2 that as the number of cores increases from 1 to around 5 initially, the communication time spikes dramatically. This sudden increase is likely due to the overhead of managing multiple cores and coordinating their communication. However, as the number of cores continues to increase beyond this point. The communication time decreases sharply. Eventually, it levels off and remains relatively constant at a low

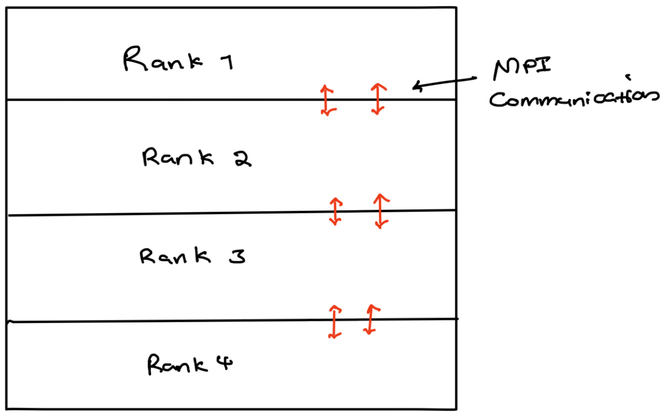

Figure 1: MPI Communication

Num of Cores	MPI Comm Time	MPI Reading Time
1	0.000007	0.200807
2	0.000229	0.070388
3	0.000196	0.067433
4	0.00097	0.058389
5	0.000177	0.036828
6	0.000218	0.036111
12	0.000115	0.024481
24	0.000205	0.015695

Table 1: MPI Communication and MPI I/O overheads

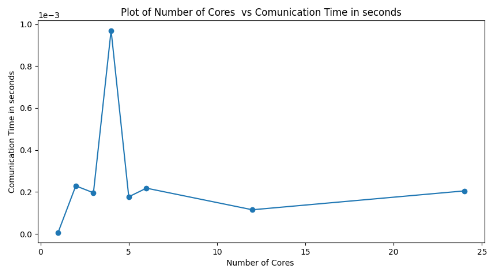

Figure 2: MPI Communication vs Number of Cores

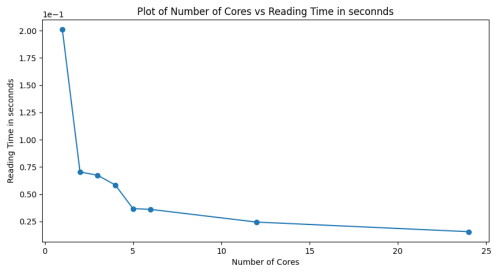

Figure 3: MPI I/O overhead

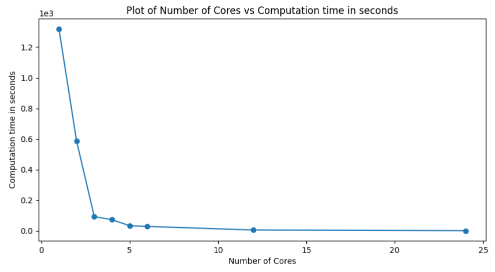

Figure 4: Computation Time vs Number of Cores

value. The leveling-off suggests that adding more cores beyond a certain point does not significantly improve communication time. The graph indicates an optimal core count (around 5) for minimizing communication time. Beyond this point, adding more cores doesn't yield substantial benefits regarding communication efficiency. While parallelizing tasks across multiple cores can improve performance, excessive parallelism can lead to increased communication overhead. Factors affecting communication time include network latency, synchronization, and data transfer rates. Balancing parallelism and communication is crucial for optimizing performance.

In figure 3, it is observed that with fewer cores (around 1), the reading time is at its peak, slightly above 1.2 seconds. As the number of cores increases, the reading time decreases significantly. There's a rapid decline in reading time up to around 5 cores. Beyond five cores, there is only a slight decrease in reading time as more cores are added. The line becomes almost flat from around ten cores onwards, indicating diminishing returns regarding reading time reduction. The graph suggests an optimal core count (around 5) for minimizing reading time. Beyond this point, adding more cores doesn't yield substantial benefits in terms of reducing reading time. While parallelizing data retrieval across multiple cores can improve performance, excessive parallelism can lead to increased overhead (e.g., synchronization, memory access). Reading time factors include data size, storage access speed, and parallel I/O efficiency.

Num of Cores	MPI Compu Time	MPI Writing Time
1	1320.346866	0.026077
2	586.1045	0.008246
3	93.386869	0.045399
4	73.470859	0.009102
5	33.63173	0.002622
6	29.207346	0.047467
12	5.891505	0.003469
24	1.508535	0.004431

Table 2: Computation and MPI writing overheads

In the figure 4, initially, with fewer cores , the computation time is at its peak, slightly above 1.2e3 seconds (approximately 1.2 seconds). There is not much wait as it is just fewer cores. As the number of cores increases, computation time decreases significantly. There's a rapid decline in computation time up to around 5 cores. Beyond this point, additional cores result in only marginal reductions in computation time.The line becomes almost flat from around ten cores onwards, indicating diminishing returns in terms of computation speedup. The graph suggests an optimal core count (around 5) for minimizing computation time. Beyond this point, adding more cores doesn't yield substantial benefits in terms of reducing computation time. Parallelizing tasks across multiple cores can improve performance,

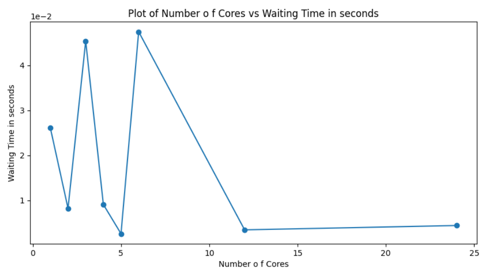

Figure 5

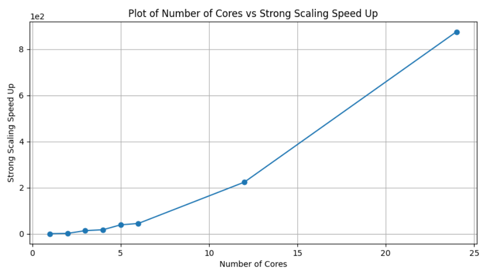

Figure 6: Speed up for strong scaling

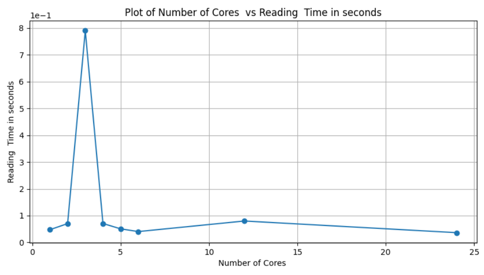

Figure 7: MPI I/O overhead vs number of Ranks/GPUs

but excessive parallelism can lead to increased overhead (e.g., communication, synchronization). Various factors that can determine the computation time include algorithm efficiency, load balancing, and memory access patterns. Properly tuning MPI parameters and workload distribution is crucial for achieving optimal performance.

In figure 5, for cores (around 1), the waiting time is at its peak, slightly above 4 seconds. As the number of cores increases, the waiting time decreases significantly. There's a rapid decline in waiting time up to around 5 cores.Beyond five cores, there is a noticeable fluctuation in waiting time when the core count is between approximately 5 and 15. After about 15 cores, there's a steady decline in waiting time. The graph suggests an optimal core count (around 5) for minimizing waiting time during data writing. Beyond this point, adding more cores doesn't necessarily yield substantial benefits in reducing waiting time. There will be under utilization of resource. It is possible that excessive parallelism can lead to increased overhead (e.g., synchronization, memory access). Possibilities influencing waiting time include data size, storage access speed, and parallel I/O efficiency.

Num of Cores	MPI Compu Time
1	1
2	2.25274992087588
3	14.1384637919492
4	17.9710280235052
5	39.2589636631836
6	45.2059857133202
12	224.110285232721
24	875.25106543766

Table 3: Computation Time in Various MPI Ranks/GPUs

$$\text{Speed Up} = t(1)/t(N) \quad (9)$$

In figure 6, as the number of cores increases, there is a linear increase in strong scaling speed up. In other words, adding more cores results in a proportional improvement in performance. When the number of cores is low (on the left side of the fig 4), the strong scaling speed up is also low. As you increase the number of cores (moving to the right), the strong scaling speed up improves significantly. However, beyond a certain point, adding more cores may not lead to a significant speedup (diminishing returns).

4.2 Weak Scaling

Weak scaling refers to the ability of a parallel algorithm or system to maintain a constant workload per processor or compute unit as the problem size increases. In other words, when the size of the input data or workload increases proportionally to the number of processors or compute units, the execution time remains constant or grows slowly.

For example, in a weak scaling scenario, if we double the number of processors, we also double the size of the problem or workload. If the algorithm exhibits weak scaling, the execution time should ideally remain constant or increase only slightly.

Weak scaling is particularly important in parallel computing environments where the problem size varies or needs to be dynamically adjusted based on available computational resources. It provides a measure of how efficiently a parallel algorithm or system can handle larger problem sizes without sacrificing performance.

Assessing weak scaling involves measuring the execution time of a parallel application while varying both the problem size and the number of processors. A good weak scaling behavior indicates that the parallel application can effectively utilize additional processors to handle larger problems without significantly increasing the overall execution time per processor.

Num of Cores	MPI Reading Time	MPI Writing Time
1	0.047545	0.001559
2	0.070211	0.021096
3	0.789862	0.019691
4	0.070623	0.004236
5	0.050682	0.081689
6	0.040528	0.002288
12	0.079925	0.004719
24	0.036414	0.036696

Table 4: Weak Scaling Efficiency and computation time

In the figure 7, the graph shows reading time (y-axis) against the number of cores (x-axis). Initially, as the number of cores

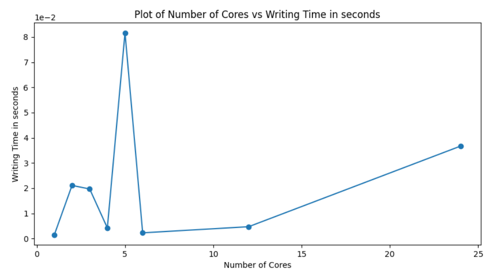

Figure 8: Number of Cores vs Writing

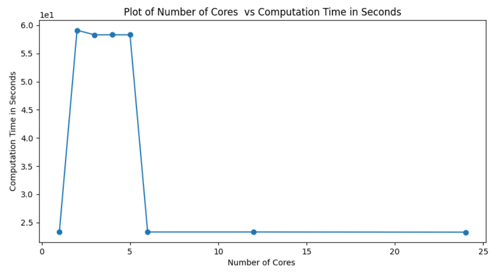

Figure 9: Execution time vs number of cores

increases from 0 to around 5, there is a significant spike in communication time, peaking at approximately 8 seconds. Beyond this point, as the number of cores continues to increase, the communication time drastically decreases and stabilizes close to zero. The initial spike (around 5 cores) suggests that communication overhead dominates when too many processes try to exchange data simultaneously. After this spike, the parallelization becomes more efficient, resulting in faster communication times.The flattening near zero indicates that communication overhead is minimal when using a larger number of cores.

Num of Cores	MPI Compu	Weak Scaling Efficiency
1	23.314489	1
2	59.112542	0.3944085
3	58.290733	0.399969048
4	58.305303	0.399869099
5	58.296536	0.399929234
6	23.318465	0.999829491
12	23.320223	0.999754119
24	23.290386	1.001034891

Table 5: Weak Scaling

Up to around 5 cores, the computation time remains relatively constant at approximately 6 seconds. Beyond 5 cores, there's a significant reduction in computation time. The computation time stabilizes at around 2.5 seconds from approximately 10 cores onwards. The initial constant region (up to 5 cores) suggests that parallelization isn't fully effective. After 5 cores, the parallelization becomes more efficient, resulting in faster computation times. However, there might be diminishing returns beyond a certain core count due to communication overhead or other factors.

$$\text{Efficiency} = t(1)/t(N) \tag{10}$$

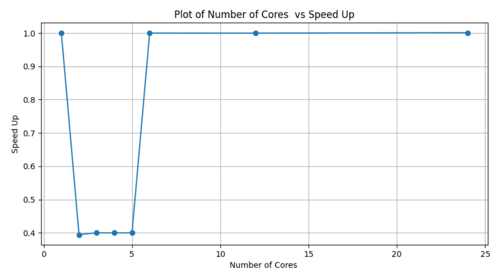

Figure 10: Weak Scaling Efficiency

In figure 10, with one core (far left on the graph), the speed-up is at its maximum value of 1.0. This means that using one core achieves the same performance as the sequential (single-core) computation. As the number of cores increases, there's a drastic drop in speed-up to approximately 0.4 when reaching around 5 cores. This decline suggests that adding more cores initially reduces efficiency due to factors like communication overhead or load imbalance. Surprisingly, after this initial drop, the speed-up rises sharply back to nearly 1.0 around 10 cores. This indicates that MPI adapts and scales well with more cores involved. Beyond 10 cores, the speed-up remains relatively constant, maintaining high efficiency. The graph suggests an optimal core count (around 10) for achieving near-perfect efficiency (speed-up close to 1.0). Beyond this point, adding more cores doesn't significantly improve efficiency. While parallelizing tasks across multiple cores can improve performance, careful consideration of factors like communication and load balancing is essential. Despite the initial drop, MPI efficiently scales with more cores, demonstrating its suitability for parallel computing.

5 CONCLUSION

The implementation of mesh-free algorithms, specifically focusing on the moving least squares (MLS) algorithm for field mapping, has showcased remarkable performance enhancements through parallelization on NVIDIA's Tesla V100 GPUs. By leveraging the substantial computational power offered by these GPUs, the project achieved significant speedups, with an impressive speedup factor of 875 observed when utilizing 24 MPI ranks/GPUs compared to a single GPU/MPI rank configuration.

Moreover, the project's exploration of weak scaling experiments demonstrated an efficiency close to 1 for 24 GPUs, indicating the effective utilization of additional computational resources to enhance overall performance. These findings underscore the successful deployment of parallelization strategies employing MPI, MPI/O, and CUDA, aimed at optimizing MLS reconstruction and other mesh-free methods for efficient data transfer and processing.

The study not only reaffirms the efficacy of mesh-free methods in field data transfer tasks but also highlights the potential of heterogeneous computing platforms in maximizing computational performance. By addressing synchronization, communication, and computation overhead, the project effectively balanced parallelism to achieve optimal performance gains. Additionally, the research underscores the importance of proper parameter tuning and workload distribution for realizing the full potential of parallel computing.

In conclusion, the project's results emphasize the significant performance improvements attainable through parallelization strategies, particularly in scenarios with balanced workloads. The findings provide valuable insights into the potential of mesh-free algorithms in heterogeneous computing environments, laying a solid foundation for future advancements in parallel MLS reconstruction and related fields.

ACKNOWLEDGMENTS

We thank Prof. Carothers for teaching the parallel computing course and making it enjoyable. We also thank CCI for providing computational resources for this course.

REFERENCES

[1] J. R. Cebral and R. Lohner. 1997. Conservative load projection and tracking for fluid–structure problems. *AIAA Journal* 35, 4 (1997), 687–692.
[2] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202 – 3216. https://doi.org/10.1016/j.jpdc.2014.07.003 Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.
[3] P. Farrell, M. Piggott, C. Pain, G. Gorman, and C. Wilson. 2009. Conservative interpolation between unstructured meshes via supermesh construction. *Computer Methods in Applied Mechanics and Engineering* 198 (2009), 2632–2642.
[4] Bengt Fornberg and Natasha Flyer. 2015. *A primer on radial basis functions with applications to the geosciences.* SIAM.
[5] X. Jiao and M.T. Heath. 2004. Common-refinement-based data transfer between non-matching mesh in multiphysics simulation. *International Journal of Numerical Methods in Biomedical Engineering* 61 (2004), 2402–2427.
[6] Peter Lancaster and Kes Salkauskas. 1981. Surfaces generated by moving least squares methods. *Mathematics of computation* 37, 155 (1981), 141–158.
[7] L.F. Paullo Muñoz, C.A.T. Mendes, and D. Roehl. 2024. Comparative evaluation of meshfree data transfer methods in hydromechanical problems. *Applied Mathematical Modelling* 128 (2024), 539–557. https://doi.org/10.1016/j.apm.2024.01.024
[8] Giuseppe Quaranta, Pierangelo Masarati, Paolo Mantegazza, et al. 2005. A conservative mesh-free approach for fluid-structure interface problems. In *International Conference for Coupled Problems in Science and Engineering, Greece.*
[9] Stuart R. Slattery. 2016. Mesh-free data transfer algorithms for partitioned multiphysics problems: Conservation, accuracy, and parallelism. *J. Comput. Phys.* 307 (2016), 164–188. https://doi.org/10.1016/j.jcp.2015.11.055
[10] Christian Trott, Luc Berger-Vergiat, David Poliakoff, Sivasankaran Rajamanickam, Damien Lebrun-Grandie, Jonathan Madsen, Nader Al Awar, Milos Gligoric, Galen Shipman, and Geoff Womeldorff. 2021. The Kokkos EcoSystem: Comprehensive Performance Portability for High Performance Computing. *Computing in Science Engineering* 23, 5 (2021), 10–18. https://doi.org/10.1109/MCSE.2021.3098509
[11] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 805–817. https://doi.org/10.1109/TPDS.2021.3097283
[12] Q. Zhang, Y. Zhao, and J. Levesley. 2017. Adaptive radial basis function interpolation using an error indicator. *Numerical Algorithms* 76 (2017), 441–471. https://doi.org/10.1007/s11075-017-0265-5

A LIST OF CODES

```makefile
Home=/gpfs/u/home/PCPE/PCPEpdlb
KOKKOS_PATH = ${HOME}/scratch/Kokkos/kokkos
KOKKOS_DEVICES = "Cuda"
EXE_NAME = "MPI_IO_kokkos"

SRC = $(wildcard *.cpp)

default: build
  echo "Start Build"

CXX = mpicxx

ifneq (,$(findstring Cuda,$(KOKKOS_DEVICES)))
EXE = ${EXE_NAME}.cuda
KOKKOS_ARCH = "Volta70"
KOKKOS_CUDA_OPTIONS = "enable_lambda,rdc"
else
EXE = ${EXE_NAME}.host
KOKKOS_ARCH = "BDW"
endif

CXXFLAGS = -O3
LINK = ${CXX}
LINKFLAGS =

DEPFLAGS = -M

OBJ = $(SRC:.cpp=.o)
LIB =

include $(KOKKOS_PATH)/Makefile.kokkos

build: $(EXE)

$(EXE): $(OBJ) $(KOKKOS_LINK_DEPENDS)
  $(LINK) $(KOKKOS_LDFLAGS) $(LINKFLAGS) $(EXTRA_PATH)
     $(OBJ) $(KOKKOS_LIBS) $(LIB) -o $(EXE)

clean: kokkos-clean
  rm -f *.o *.cuda *.host *.tmp core.* *.out *.err

# Compilation rules

%.o:%.cpp $(KOKKOS_CPP_DEPENDS)
  $(CXX) $(KOKKOS_CPPFLAGS) $(KOKKOS_CXXFLAGS) $(
     CXXFLAGS) $(EXTRA_INC) -c $<

test: $(EXE)
  ./$(EXE)
```

Listing 1: Makefile for building executable

```
Time taken for reading: 0.015695 seconds
Time taken for communication: 0.000205 seconds
Time taken for computation: 1.508535 seconds
Time taken for writing: 0.004431 seconds
```

Listing 2: Outputfile

```python
import pandas as pd
import matplotlib.pyplot as plt

# Load the Excel file into a pandas DataFrame
df = pd.read_excel('/Users/sina/Downloads/
     weak_scaling_data/Excel_Files/Reading.xlsx')

# Get the column names from the DataFrame
column_names = df.columns.tolist()

# Plot the data using the column names for labels
plt.figure(figsize=(10, 5))
plt.plot(df[column_names[0]], df[column_names[1]],
     marker='o')

# Set the y-axis to scientific notation
plt.ticklabel_format(style='sci', axis='y', scilimits
     =(0,0))

# Add labels and title using the column names
plt.xlabel(column_names[0])
plt.ylabel(column_names[1])
plt.title(f'Plot of {column_names[0]} vs {column_names
     [1]}')

# Add grid to the plot
plt.grid(True)

# Show the plot
plt.show()
```

Listing 3: Python for plotting

```bash
#!/bin/bash
#SBATCH --job-name=kokkos_mpi
#SBATCH --output=kokkos_mpi.out
#SBATCH --error=kokkos_mpi.err
```

```
#SBATCH --partition=el8
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=6
#SBATCH --time=00:20:00
#SBATCH --gres=gpu:24
#SBATCH --gres=nvme
#SBATCH --gpus-per-node=6




module load xl_r spectrum-mpi cuda/11.2

export OMPI_CXX=/gpfs/u/home/PCPE/PCPEpdlb/scratch/
    Kokkos/kokkos/bin/nvcc_wrapper
export LD_LIBRARY_PATH=/gpfs/u/home/PCPE/PCPEpdlb/
    scratch/Kokkos/kokkosInstall/lib64:
    $LD_LIBRARY_PATH

cd /gpfs/u/home/PCPE/PCPEpdlb/scratch/
    parallel_assignments/class_project/
    strong_scaling_analysis

mpirun --bind-to core --report-bindings -np
    $SLURM_NPROCS ./MPI_IO_kokkos.cuda --kokkos-num-
    devices=24 > output8.txt
```

Listing 4: Batch script for job submission