Data movement framework for dynamic load-balanced massively parallel distributed algorithms

Submission #265

ABSTRACT

Efficient parallel computational geometry algorithms are becoming crucial to carry out increasingly complex high-fidelity numerical simulations on meshes with dynamic geometry and topology. Meanwhile, the development of such algorithms on distributed-memory architectures requires a substantial amount of work to set up the required data communications. This paper aims to identify different key data distribution frames imposed on computational geometry algorithms by the numerical simulation workflow into which they are integrated. Data sets are distributed in either *partitioned* or *block-distributed* frames. Numerical simulation solvers rely on an optimized partitioning that provides the required data locality, while keeping communications to a minimum. However, this partitioning may be ill-suited to geometric computations as the entities of interest are usually unevenly distributed across the processes. Dynamic load balancing is therefore essential for maintaining good performance, and can be achieved by dividing the workload into unitary independent tasks that are redistributed equitably. The block-distributed frame makes this possible, while guaranteeing parallelism-independent results through the use of parallel sorting algorithms. This framework extends beyond computational geometry, owing to its high degree of genericity. Scaling studies using the framework implementation in the open-source *Blinded Reference* library show satisfactory performance and demonstrate that this framework can be effectively integrated into a massively parallel simulation workflow.

CCS CONCEPTS

• **Computing methodologies → Massively parallel algorithms**.

KEYWORDS

High Performance Computing (HPC), Communication graph, Message Passing Interface (MPI), Distributed-memory computer, Dynamic load balancing

ACM Reference Format:
Submission #265. 2024. Data movement framework for dynamic load-balanced massively parallel distributed algorithms. In *Proceedings of ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Solving partial differential equations using numerical simulation codes on a partitioned mesh performs well on supercomputers if the sub-domains are evenly distributed across all the compute cores. Historically, geometric operations on meshes were carried out using specific sequential pre- and post-processing tools. The increasing need for high-fidelity simulations requires the consideration of fine geometric details which are crucial for a better understanding of physical phenomena. Thus, bigger meshes are required and more geometric operations during the simulation (co-processing) need to be carried out. Performance is thus paramount, as these algorithms must be executed repeatedly in the iterative cycle of the simulation. Traditional numerical methods operate on compact stencils which induce only data exchange in a close neighbourhood. Generating one good partitioning for all iterations is optimal in this context since the same method is carried out at each iteration. This will be referred to as a static parallel context. However, geometric problems such as distance calculations, generally require the inspection of much larger regions of space. To achieve high performance a dynamic parallel context with load and memory balancing at several steps in the algorithm is mandatory. Indeed, these computational geometry problems are by nature unbalanced. Data locality is necessary for carrying out the geometric computations. As the data in question is distributed, communications are required to achieve locality. Copying all data into distributed memory or collecting it on a single node is not a viable option due to memory constraints. These calculations take place in a simulation workflow with multiple components with significant memory footprints. Furthermore, industrial applications reach such large scales that the complete mesh can not fit in the memory of a single node.

This aspect is underlined in the 2020 update of the CFD Vision 2030 Roadmap [8]. Indeed, one of the technology demonstrations which was aimed for 2020 was "on-demand analysis/visualization of a 10 billion point mesh, unsteady CFD simulation". Considering a tetrahedral mesh, this leads to roughly 60 billion cells. A rough estimation shows that at least 1680 GB of memory is required to store minimal information for such a mesh:

- cell global identifiers : 480 GB (8 bytes per cell global identifier)
- cell → vertex connectivity : 960 GB (4 bytes per vertex local identifier)
- vertex coordinates : 240 GB (8 bytes per vertex coordinate)

Each node of the TOPAZE supercomputer of the CEA-CRRT [9] (ranked 238 in the top 500 in November 2023) has a 256 GB RAM capacity. Thus, working on distributed-memory architectures is a crucial challenge to face in order to compute on such large meshes. On top of that, using all the available computational resources induces a major speed-up when the sequential part is reduced to a maximum as stated in Amdahl's law [5].

The International Exascale Software Project roadmap [14] warns against the danger of developing frameworks for one-of-a-kind applications. It points out the harmfulness of the proliferation of tools of this type because they lead to redundancy. The tools developed cannot be shared, as they are incompatible or sub-optimal for any other application. The difficulty of maintaining and improving this multitude of tools is ultimately limiting the number of exascale applications. Originally designed for computational geometry applications with sets of mesh entities, the framework presented in this paper extends to any set of abstract items. Intended to serve as a common kernel for various applications, methods implementing this framework are available in the open-source *Blinded Reference* (Blinded Acronym) library [1]. The library provides data movement tools wrapping the Message Passing Interface (MPI) [23] to facilitate the development of algorithms relying on dynamic redistribution of unitary independent tasks. Many of the concepts addressed in these tools are crucial topics in the development of high-performance distributed parallel algorithms. For instance, pooling the parallel sort algorithms outside of the geometric algorithms in an open-source library makes them accessible to be used in other kinds of applications.

After an overview of the related work in Sec. 2, the data movement framework concept's are presented at the beginning of Sec. 3. Afterwards in sub-section 3.2 the implementation in *Blinded Reference* for data movement from the *block-distributed* frame to the *partitioned* one is detailed. The reverse exchange which is key for dynamic load balancing is introduced in sub-section 3.3. Last, performance results of these features for several communication graph scenarios are presented.

2 BACKGROUND AND RELATED WORK

Many scientific computation applications run on distributed-memory architectures using Message Passage Interface (MPI) programming for data communications. In this paper, we identify several works that address the data movement between different distributions in a dynamic parallel context : ZOLTAN [11], SCOREC-PCU [18], DIY [21], PetscSF [25], PLE [24] and PaMPA [19].

ZOLTAN [11] features data management services for adaptive and dynamic applications based on unstructured meshes in a parallel context. It offers capabilities for developing load-balanced algorithms: mainly a communication graph construction helper and a data migration tool. These are generic features which users specify by setting up mandatory callback functions. Locating off-process data is done using a distributed hash table, referred to as the "distributed directory". This is used to distribute data evenly across processes in ZOLTAN based on collective communications. Moreover, an array of unsigned integers is used to uniquely identify mesh entities, which is key as explained in Sec. 3. All features require to create a `Zoltan_Struct` data structure specific to ZOLTAN, which makes it intrusive in the calling code.

Similar tools are offered in SCOREC-PCU [18], which serves as a toolbox for parallel communication in SCOREC, with additional support for hybrid MPI/thread environments. This library allows to exploit the shared memory per node of supercomputers through multi-threading. No information is available on the technological choice (e.g. OpenMP, MPI-3) to do this. SCOREC-PCU is said to work using the Bulk Synchronous Parallel model for their communications. In this model, exchanges are divided in supersteps composed of a phase of computation, communication and finally synchronization. In our experience, massive parallelism benefits from as little synchronization barriers as possible.

Similarly, DIY [21] provides communications tools aimed as building blocks in the development of parallel algorithms in distributed and shared memory environments. Instead of process-to-process communications, data are decomposed in blocks which are assigned to processing elements (processes or threads) leading to block-to-block exchanges. Between these blocks local synchronous, remote, asynchronous and global (reductions) communication functions are provided.

PETSc features a communication component called PetscSF. Compared to other state-of-the-art tools, it stands out for taking into account GPUs and other accelerators. Communication patterns in this library by Zhang et al. [25] are managed by using a star-forest graph representation.

In the context of an algorithm for locating point clouds in meshes, Fournier [15] provides the `ple_locator_exchange_point_var` function in the PLE [24] library, which is a tool to exchange data between two partitioned sets. In the context of data transfer based on point location inside a mesh, the result of the interpolation computation of a field from the source mesh onto the target point cloud has to be transferred from the source mesh entities set to the point cloud entities set. A mapping for point-to-point communications between the source entities processes and the target point cloud processes is created. A reverse function is provided so that the target points can provide extra information to the donor entities. To avoid collective communications, separate send and receive exchanges are scheduled but this can lead in the worst-case scenario to serialized exchanges.

PaMPA [19] is a parallel distributed library for redistributing and remeshing unstructured meshes based on a sequential remesher using PT-SCOTCH [10] for mesh partitioning operations. The `PAMPA_dmeshHaloValue` routine is used to spread information held by local vertices to the associated ghost vertices on neighboring processes. Collective communications are used except when the number of neighbouring processes involved in data exchanges is less than a percentage of the total number of processes. This service is available in a synchronous and asynchronous version. Naturally, this data communication routine is application-specific.

The work described above reveals how essential it is to develop communication frameworks capable of adapting to hardware-specific standards. Still it remains difficult to tackle all at once the communication pattern and hardware genericity allowing performance in a non-intrusive tool which is easy to maintain and upgrade. The framework presented in this paper focuses on communication pattern genericity. It is implemented in *Blinded Reference* in such way as to minimize the impact on the calling code, only using simple contiguous arrays rather then complex data structures. Wrapping MPI function calls makes it easier to adapt to future hardware.

3 DATA MOVEMENT FRAMEWORK

Before introducing the data movement framework in sub-section 3.2 and sub-section 3.3, necessary key concepts and definitions are presented.

3.1 Definitions

Global identifiers. Dynamic load balancing involves moving data items between different distributions. In order to keep track of these moving items, each one of them must be identified unambiguously. To make this possible, the data movement framework presented in this paper relies on unique (across the processes) integers assigned to each data item. We refer to these integers as *global identifiers*. Note that the use of global identifiers allows to develop algorithms independent of parallelism. When a choice has to be made between candidate items in an algorithm, it is done using a heuristic based on the global identifiers of the items. Consequently, the same decisions are made independent of the run of the algorithm.

Block-distributed frame. Let us consider a set of items sorted using their global identifiers with a single occurrence of each one, like the set in Tab. 1. A simple method for distributing this set of globally identified items consists in assigning each process an equal portion of the set. In order to quickly determine which process an item belongs to, let each process retrieve a portion of the set in ascending order of process identifier, while maintaining item order. The items global identifiers allow every pair of the set to be comparable. The set is therefore partially ordered. Knowing the global identifier range of each process enables efficient location of each item by dichotomy. We refer to this frame by the term *block-distributed* (summarized in Def. 3.1).

Table 1: Global identifiers associated to a set of items distributed across processes identified by $\{p_i\}$ for $i \in \{0, 1, 2\}$ separated by ||

$\{p_1\}$			$\{p_2\}$		$\{p_3\}$	
1	2	3	4	5	6	7

Definition 3.1. Let any block-distribution $\mathcal{B} := \{\mathcal{B}_p\}_{0 \le p < P}$ where $\mathcal{B}_p$ is the set of items held by process p out of P. $D := (b_0, b_1, \ldots, b_P)$ is the *block distribution index* array that gives the range of global identifiers of each process and satisfies $b_0 \le b_1 \le \ldots \le b_P$. In other words, a process p always holds items with smaller global identifiers than process $p+1$. Each block $\mathcal{B}_p$ holds $b_{p+1} - b_p$ entities with contiguous global identifiers. Note that b_P is the total number of items in the distribution. D is thus sufficient to describe completely the block-distributed frame.

Partitioned frame. Let us consider a set of items distributed across the processes to suit the resolution of a given problem. We refer to this data distribution as the *partitioned* frame (summarized in Def. 3.2). Tab. 2 illustrates a set of equally distributed items without any particular ordering. There are other possible combinations. For instance, Tab. 3 depicts a situation in which a process has an empty partition. Tab. 4 underlines the fact that in the partitioned frame the uniqueness of items of the set across all processes is not guaranteed. Moreover, the item with the global identifier 5 does not appear in this distribution while 6 and 3 are duplicated.

Table 2: Global identifiers associated to a set of partitioned items

$\{p_1\}$		$\{p_2\}$			$\{p_3\}$	
2	6	7	1	5	3	4

Table 3: Global identifiers associated to a set of partitioned items with a process without any

$\{p_1\}$					$\{p_2\}$	$\{p_3\}$	
2	6	7	1	5		3	4

Table 4: Global identifiers associated to a set of partitioned items with duplicates and one missing

$\{p_1\}$		$\{p_2\}$			$\{p_3\}$	
2	6	3	1	6	3	4

Definition 3.2. Let any partition $\mathcal{P} := \{\mathcal{P}_p\}_{0 \le p < P}$, where the partition $\mathcal{P}_p$ held by process p is an arbitrary subset of size n_p of the total item set. In particular, $\mathcal{P}_p$ may be empty or contain multiple occurrences of the same item.

Mesh connectivity. In the context of computational geometry, the items of interest are mesh entities. A minimal mesh representation stores the mesh entities global identifiers, the entity $\longrightarrow$ vertex adjacency and the vertex coordinates. In *Blinded Reference*, the adjacency list is stored as in the Distributed CSR Format by ParMeTiS presented in [17]. These adjacency relations can be represented in matrix form. With e_i the i-th edge out of m and v_j the j-th vertex out of the n, an example of edge-vertex adjacency matrix is shown in Fig. 1.

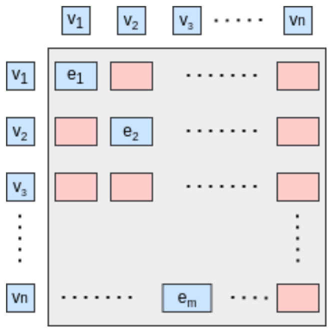

Figure 1: Sparse edge-vertex adjacency matrix with the red boxes underlining the edge absence between a pair of vertices

This adjacency matrix is usually sparse when describing a mesh. For cache efficiency reasons it is better suited to store the adjacencies in the form of a list (see Fig. 2). A mesh can be composed of entities with a different amount of vertices (e.g. 4 for tetrahedra, 8 for hexahedra). In addition to the adjacency list to define the mesh an index array is created. It provides the range of vertices associated to each entity. In this case the index array is trivial because each edge is defined by only two vertices.

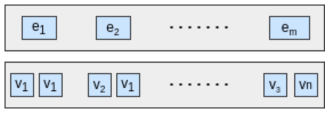

Figure 2: Compact edge-vertex adjacency storage

This method stores data more compactly. Since limited memory is allocated to each process, the adjacency list cannot contain globally indexed entities. Consequently, a local and compact numbering is used to index the entities in local memory. Each process holds the mapping from the local process identification into the global identification.

The mesh structure is usually stored in a mesh file (e.g. CGNS [2], VTK [4] format). These files contain entity headers followed by the amount of data and the data itself. For instance, the amount of cells in a mesh and the associated cell → vertex adjacency list. A global read provides the total amount of data to read in order to initialize a numerical simulation. Then, in a parallel mesh read, each process retrieves data as in the block-distributed frame (see Fig. 3). Mesh partitioning tools like ParMeTiS [17] and PT-Scotch [10] operate the graph partitioning on the block-distributed entities after a mesh read. Numerical simulation codes typically require such a partitioned mesh as input. During the mesh read, physical fields defined on the mesh entities are loaded into memory. These are not transferred when partitioning the mesh. Keeping a correspondence between block-distributed entities and partitioned entities is therefore essential.

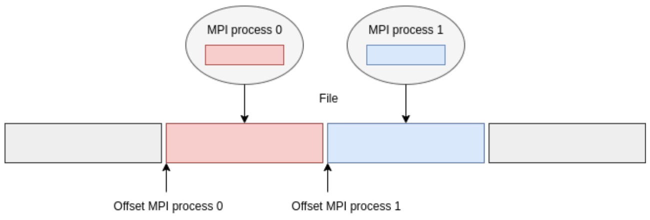

Figure 3: Parallel file read by two MPI processes

Binary search gap algorithm. To create the communication graph between those two frames a parallel search algorithm is used on the global identifiers. For each item $e_i \in \mathcal{P}_p$, the *unique* MPI process q that holds e_i in $\mathcal{B}$ has to be found. Therefore the global identifier of e_i, noted g, has to satisfy $b_q \leq g < b_{q+1}$. Because the block distribution index D is a sorted array, this search can be performed efficiently in $O(\log P)$ for each partitioned entity using a binary search algorithm. Thus, each process performs this step is in $O(n_p \log P)$ and requires no communication or synchronization, as D is shared by all processes.

To generate the communication graph, collective communications are used by each process to send the identifiers of the items it needs to get data from to the processes which hold them in the block-distributed frame. This algorithm is essential to move data without any user knowledge about the array distribution into memory. Thus, the user does not make any direct MPI communications. The framework presented in this paper acts as an intermediary to exchange data between $\mathcal{B}$ and $\mathcal{P}$. In a sense, data in $\mathcal{B}$ is for the user in a virtual global array from which to take or put back data from $\mathcal{P}$ given item global identifiers.

3.2 Block-to-Part

Sub-section 3.1 defined the block-distributed (*Block* in short) and partitioned (*Part* in short) frames. The aim of *Block-to-Part* is to abstract exchanges from a block-distributed set to a partitioned set. The example of a set of mesh entities in Fig. 4 shows that they are not necessarily on the same process depending on the frame. Consequently, a mapping for required inter-process exchanges has to be established using entities global identifiers. Indeed, as highlighted in Fig. 4, process local numbering does not allow to uniquely identify mesh entities independently from the frame.

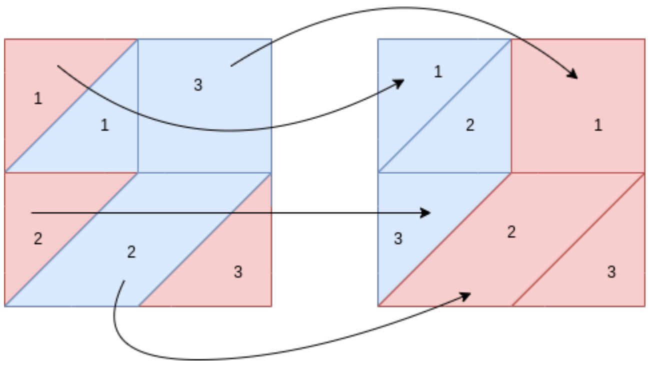

Figure 4: Inter-process exchanges (arrows) from a mesh in the block-distributed frame (left) to the partitioned frame (right) with local to process number associated to mesh entities

From the user point-of-view, a *Block-to-Part* structure is created by mainly providing the global identifiers of the entities for which data needs to be exchanged, i.e. the global identifiers of the entities in the partitioned frame. Then the exchange is performed by providing the block-distributed data array and the pointer to retrieve the partitioned array of data. Note that this *Block-to-Part* instance can be reused as many times as necessary for that set of entities since it holds the communication graph between the entities in block-distributed frame and the partitioned entities.

Behind the scenes a communication graph between $\mathcal{B}$ and $\mathcal{P}$ is established. For each entity in $\mathcal{P}$, the MPI process which holds it in $\mathcal{B}$ has to be found. Indeed, this MPI process has the data to transfer. Here the uniqueness across the MPI processes and the partial order of the entities set is key. Entities being ordered in the block-distributed frame, search heuristics can be used to establish the communication graph. This is done in *Blinded Reference* using the binary search gap algorithm defined above. For each entity in

$\mathcal{P}$, the range $[b_p, b_{p+1})$ in the block distribution index in which the associated global identifier falls is found. Hence, data associate to the entity will be retrieved from $\mathcal{B}_p$ on process p. Each MPI process now holds the information from which other MPI processes it needs to receive data. Thus each MPI process can communicate to the MPI processes from which it wishes to receive data associated to given entities. This is done using collective communications. `MPI_Alltoall` is used to exchange the amount of data to receive from the other MPI processes. Then, `MPI_Alltoallv` exchanges the local number for each MPI process of the entity from which it needs to receive data. This is summarized Alg. 1.

Algorithm 1: *Block-to-Part* creation

```
Data:
Block entity distribution
Partition entity global identifiers
Result:
Communication graph
On each process do:
for entities in the partition do
    Map the entity to the process that holds its associated
      block-distributed data using binary search gap
end
Count the amount of data to send to each process
Build an array of the requested entities from other processes.
  For all process $p$, $\mathcal{B}_p$ is locally sorted. Consequently, the
  requested entities can unambiguously be referred to by
  their local identifier (determined in $O(1)$ from the global
  identifier).
Use collective communications to exchange the count array
Use collective communications to exchange the requested
  entities array
```

Collective communications such as `MPI_Alltoall` are costly in bandwidth. Still, it is a commonly used solution as seen in the related work overview. Indeed, bypasses are inconclusive in critical cases as underlined by Fournier [15] who faces sequential exchanges in the worst-case scenario. Developing complementary *Block-to-Part* implementations based on point-to-point communications could nonetheless facilitate the handling of large messages on supercomputers with less efficient interconnection networks.

3.3 Part-to-Block

This last sub-section presents the reverse communication operation of the one described in sub-section 3.2. *Part-to-Block* is used to balance unitary independent tasks identified in a parallel distributed algorithm. In this context the block-distribution is usually not imposed by the user but automatically generated using the following *parallel bucket sampling* algorithm.

Parallel bucket sampling algorithm. Derived from the partitioning algorithm based on the Morton space-filing curve in the FVM library [16], this algorithm aims to equitably distribute a set of N *weighted* items, initially distributed in the partitioned frame $\mathcal{P}$. If each item represents a unitary independent task, its associated weight is typically an estimate of the workload. Given P partitions $\mathcal{P}_p$ containing each n_p items with global identifiers (g_i) and associated weights (w_i), the goal is to find an optimal distribution index $(b_0, b_1, \ldots, b_p)$ such that each block $\mathcal{B}_p$ has an equal weight. Since a given item with global identifier g can exist in multiple partitions, the *global* weight $\hat{w}_g$ of this item needs to be considered, which is simply the sum of the weights of all its occurrences in $\mathcal{P}$. From Def. 3.1 it can be deduced that the weight of block $\mathcal{B}_p$ is $W_p = \sum_{g=b_p}^{b_{p+1}-1} \hat{w}_g$. First, a uniform distribution, i.e. $b_p = \frac{pN}{P}$, is associated to each process p. This step is performed in $O(n_p \log P)$ time using the binary search gap algorithm defined in sub-section 3.1. Then, a collective reduction (`MPI_Allreduce`) is used to determine the weight W_p of $\mathcal{B}_p$ for the given initial uniform distribution. The quality of this distribution is evaluated by a *load imbalance factor* expressed as $f = \frac{1}{W_{\text{opt}}} \left(\max_p W_p - \min_p W_p\right)$, where $W_{\text{opt}} = \frac{\sum_p W_p}{P}$ is the optimal, i.e. average weight. If this factor is greater than a fixed tolerance ϵ, the block splitters b_p are shifted in order to create a new distribution with higher quality. As long as the distribution quality is not satisfactory, this procedure is repeated. Let $s(g) = \sum_{i=0}^{g-1} \hat{w}_g$ denote the cumulative frequency associated to this distribution. Ideally, it should satisfy $s(b_p) = pW_{\text{opt}}$ for all $0 \leq p \leq P$. Illustrated by the blue line in Fig. 5, a coarse-grained piece-wise linear approximation $\tilde{s}$ of the cumulative frequency in red is obtained, under the assumption that global item weights are uniform within each block. Each splitter b_p is then moved to $b'_p = \lfloor (1-\lambda) b_q + \lambda b_{q+1} \rfloor$, where q is such that $\tilde{s}(b_q) \leq pW_{\text{opt}} < \tilde{s}(b_{q+1})$, and $\lambda = \frac{pW_{\text{opt}} - \tilde{s}(b_q)}{\tilde{s}(b_{q+1}) - \tilde{s}(b_q)} \in [0, 1]$. The approximation $\tilde{s}$ may be refined by using more than one *bucket* per block. This oversampling helps accelerating convergence of the iterative algorithm in situations where weight values are highly dispersed. The load-balance improvement, thanks to the splitter relocation, can be observed on the y-axis of Fig. 6 compared to the initial distribution Fig. 5. On the x-axis it can be seen that the splitters at equal distance induce load imbalance. In the *Blinded Reference* implementation, using $4P$ buckets reveals usually sufficient. The tolerance ϵ is set to 0.1, since it is often preferable to cope with a mild imbalance rather than carry out additional iterations with diminishing returns. Finally, the maximum number of iterations is set to 5. These are purely heuristic results in the search for a compromise between efficiency and load balance quality.

The block-distribution $\mathcal{B}$ is determined using the parallel bucket sampling algorithm described above. Indeed, each item is associated to only one process. Multiple partitions can hold the same item, with possibly distinct associated data. Each process will receive data from each instance of that item across the processes. This data will be received in the `MPI_AlltoAllv` frame. Then a quicksort algorithm, which relies on determining the position of the array elements relative to a pivot element, is used to generate an indirection between this frame and the locally sorted block-distributed one. This step, performed in an embarrassingly parallel fashion, is referenced as "post-processing" in Alg. 2. One could choose to retain only data received from the process with the lowest process identifier. On the other hand, all the received data can be retained, either to carry out a reduction operation, or to gather partial information calculated by each process.

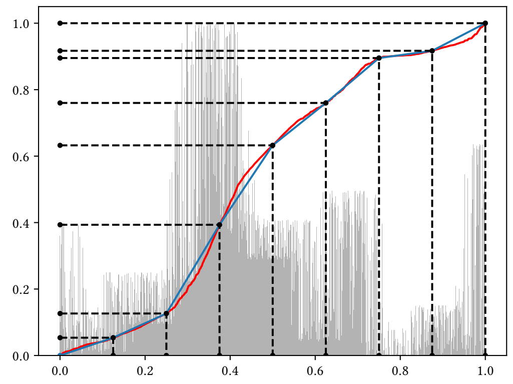

Figure 5: Normalized weights associated to each item (grey) with associated cumulative frequency (red) and interpolation based on a naive sampling (blue)

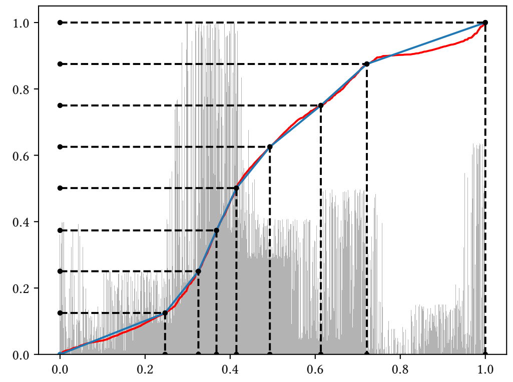

Figure 6: Normalized weights associated to each item (grey) with associated cumulative frequency (red) and interpolation based on an optimized sampling (blue)

Application example. The *Part-to-Block* framework expends algorithm development horizons by simplifying the implementation of dynamic load balancing. To illustrate this, let us focus on the design of an isosurface extraction algorithm on distributed-memory architectures. Given a distributed volume mesh on which a discrete scalar field is evaluated, this method involves generating a surface mesh where the interpolated field is constant for a given specified value. The bi-color vertex-centered field in Fig. 7 indicates whether the field is lower than or greater than the specific value at these vertices. Many serial isosurface extraction algorithms have been proposed in literature [13, 20]. Applying these algorithms on each process in a multi-sequential fashion can lead to severe load imbalance since the isosurface is in general not equally distributed in the input volume mesh partitions, as can be seen on Fig. 8. To overcome this issue, the workload needs to be broken down into a set of unitary independent tasks that can be evenly redistributed. Let us consider the following suggested algorithm. First, each process identifies the cells with different colors, meaning where the

Algorithm 2: *Part-to-Block* creation

```
Data:
Block entity distribution (optional)
Partition entity global identifiers
Weights (optional)
Result:
Communication graph
On each process do:
if No block entity distribution then
    Generate a balanced block-distribution using parallel bucket sampling
end
if Weights then
    Compute global weights to take into account duplicates
end
for entities in the partition do
    Map the entity to the process that holds its associated partitioned data using binary search gap
end
Count the amount of data to send to each process
Build an array of global identifiers associated to the requested entities from other processes
Use collective communications to exchange the count array
Use collective communications to exchange the requested entities array
Generate the data post-processing indirection
```

field encloses the disered isovalue. Then, the selected cells are redistributed using *Part-to-Block* leading to the setting in Fig. 9. A serial isourface extraction algorithm is finally executed on the well balanced block-distributed cells. In addition to the efficiency of this method, the resulting isosurface is evenly distributed, which is advantageous if further computations need to be carried out. This isosurface extraction example shows that usually the data distribution is initially ill-conditioned for the geometric computation to perform. Indeed, either the data distribution suits the needs of a computational code with a static parallel context or it was adapted to the previous step of the computational geometry algorithm.

4 PERFORMANCE RESULTS

Sec. 3 presented an innovative framework and associated tools to dynamically load-balance parallel problems. A performance study has been led on those two data movement tools in *Blinded Reference* isolated from the computational geometry context. The different settings have been designed to be representative of the final problems targeted by the library. The aim is to have a quantitative analysis of these tools for the suitability of their use in a computational geometry algorithms to be called within a numerical simulation workflow.

This study was initiated on the assumption that better performance is achieved when the communication graphs are sparse and concentrated around the main diagonal because it limits the amount of large messages passing through the interconnection network. In addition, Brandfass et al. [7] argue that intra-node communications are usually much faster than inter-node communications.

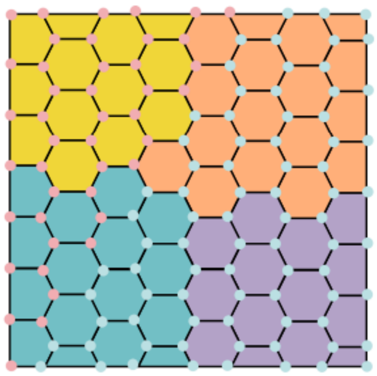

Figure 7: Bi-color field indicating whether the vertex-centered field is lower than or greater than a specific value on a partitioned mesh

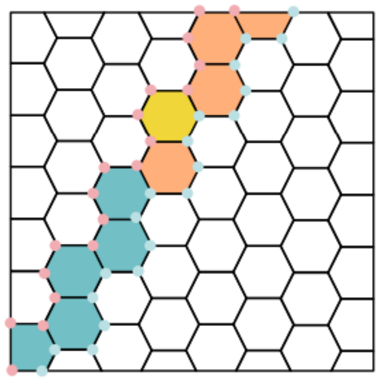

Figure 8: Extracted area of field color change for the application of a multi-sequential isosurface algorithm

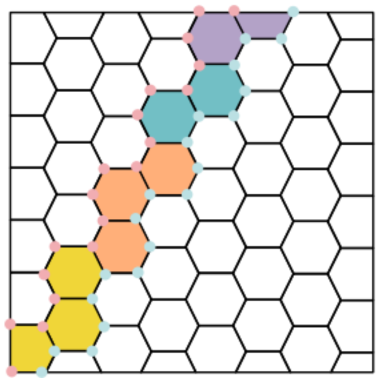

Figure 9: Extracted and load-balanced area of field color change for the application of a isosurface algorithm

The optimized communication graph using their rank reordering method for MPI backs our assumption by being essentially diagonal. Thus, to investigate our hypothesis the two data movement tools are studied on communication graphs ranging from *diagonal* via *quasi-diagonal* to *random*.

At the a priori most optimal end of the studied range is the *diagonal* communication graph for which the block-distributed frame and partitioned frame are equal ($\mathcal{P}_p = \mathcal{B}_p$). No data is exchanged between processes in this setting. Indeed, the required data is already present on the receiver MPI process. This means no data will pass through the interconnection network.

Then, there is the *quasi-diagonal* one for which $\mathcal{P}_p$ holds items from $\mathcal{B}_p$ and from the subsets on neighbouring MPI processes from p. This induces a thick diagonal on the resulting communication graph as can be seen on Fig. 10. The settings for this scenario are generated by adding exchanges to the *diagonal* scenario with a percentage of all processes on both sides of the main diagonal. To fit real usage scenarios, this paper presents results for the 10% and 25% cases. Here, most of the exchanges with other MPI processes will be intra-node as shown on the distance graph in Fig. 11.

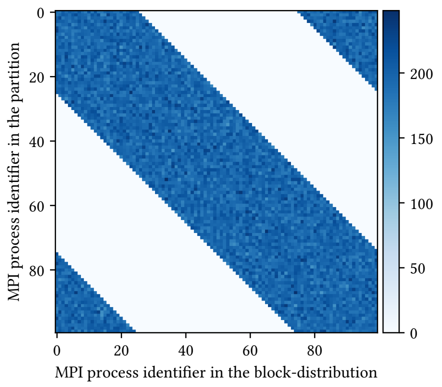

Figure 10: Quasi-diagonal communication graph with 25% shift on both sides of the main diagonal with the amount of integers per MPI process

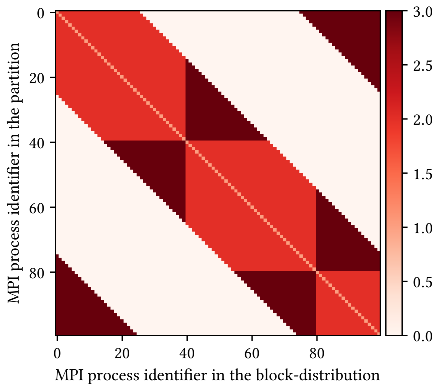

Figure 11: Quasi-diagonal distance graph (0: no data exchanged, 1: data required from current MPI process, 2: intra-node communication and 3: inter-node communication)

Last, the a priori worst-case scenario consists in drawing uniformly *random* partitions $\mathcal{P}_p$. This leads to a dense communication graph. Each MPI process will communicate with all the other MPI processes in the communicator (see Fig. 12). Exchanges through this communication graph are more heavily impacted by the interconnection network latency. Fig. 13 illustrates this well with the majority of exchanges being dark red (value: 3), signifying inter-node exchanges.

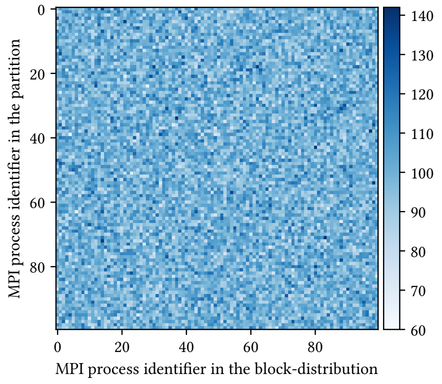

Figure 12: Random communication graph with the amount of integers per MPI process

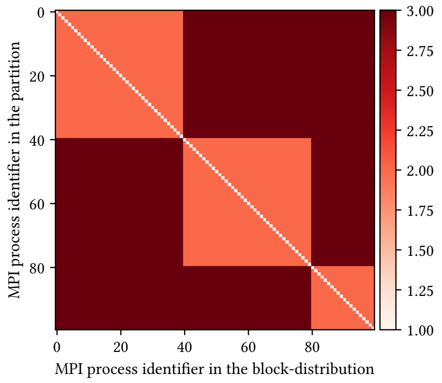

Figure 13: Random distance graph (0: no data exchanged, 1: data required from current MPI process, 2: intra-node communication and 3: inter-node communication)

4.1 Hardware description

The tests presented in this section were carried out on the same supercomputer. It has 400 compute nodes each composed of 2 Intel Xeon « Cascade Lake - 6240R » processes running at 2.4 GHz. Each node is composed of 48 compute cores. 192 GB of memory is available per node. Exchanges are operated through an Intel Omnipath interconnection network at 100 GB/s.

4.2 Block-to-Part

Performance results for *Block-to-Part* in the different scenarios previously discussed are examined in this section. An array of integers is exchanged between $\mathcal{B}$ and $\mathcal{P}$.

To assess the performance in a realistic use-case, a weak scaling study is conducted. Indeed, when the size of the numerical simulation grows, it is standard to increase the amount of used processes as well. A distributed array of 600 000 integers is exchanged for Fig. 14 from a block-distributed to a partitioned frame. For each scenario, the total elapsed time for the creation of the communication graph followed by one exchange is reported. A first interesting observation, comparing the *diagonal* and *quasi-diagonal* scenarios, reveals that the density of the communication graph, particularly the concentration around the main diagonal, quickly strongly impacts performance. Given the initial hypothesis, a key result is that the difference between the *quasi-diagonal* and full *random* scenarios is not significant for a shift above 10%.

Let us compare these results with their context of use. The efficiency of numerical simulation codes can be assessed by the time taken to perform one iteration on one cell of the mesh, which is of the order of $1\mu s$ for state-of-the art codes [22]. For a mesh with 600 000 cells per process, one iteration thus accounts for 6s. Taking a closer look it can be seen that the *random* scenario takes a bit less than 3s on 4800 processes. In the breakdown in Tab. 5 it can be seen that the *random* communication graph creation takes about 1.8s for set $\mathcal{B}$ composed of 2.88 billion items. Once the communication graph established, it is reused to exchange as much data as needed, taking about 1s per exchange for an array of integers. The order of magnitude of the execution time of *Block-to-Part* is consistent for a use in a co-processing context because geometric computations are not required at each simulation iteration. Last, this perspective has been done for a computational geometry algorithm handling the entire volume mesh, whereas these computations are usually restricted to surface entities.

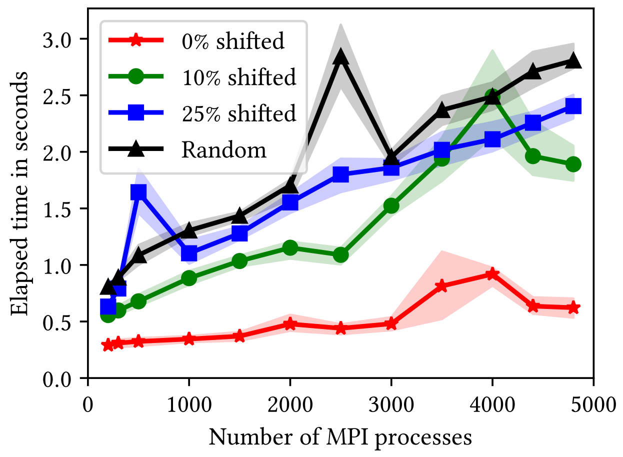

Figure 14: Mean elapsed time (in s) (with the range from minimum to maximum in shaded color) of a *Block-to-Part* communication for increasing number of MPI processes for 600 000 integers sent per process

4.3 Part-to-Block

As described in sub-section 3.3, operating the reverse communication of *Block-to-Part* is more computationally intensive. To focus on the influence of communication graph density on performance, this study assumes that entities are uniformly weighted. The main observation of the weak scaling results in Fig. 15 is that the same trend as in the previous section can be observed. As expected due to the extra steps in the *Part-to-Block* algorithm, the *random* scenario takes about 5s on 4800 processes. The OSU benchmark [3] latency measurements on the same supercomputer with the same MPI library version are used to quantify the consistency of these results compared to a `MPI_AlltoAllv` alone. The exchange of 512 bytes to each process on 4800 processes takes 0.06s in the OSU benchmark. About 500 bytes are communicated with each other process for the integer exchange through the communication graph established in the *random* scenario. The `MPI_AlltoAllv` communication in this exchange step takes as well 0.06s. Meanwhile, the breakdown in Tab. 6 shows 1.6s for this exchange step. Since the MPI function signature constrains order of the received data arrays, costly copies are required to return data in a sorted manner to the user. Further analysis of the breakdown shows that the collective communications used to create the communication graph are the most time-consuming step. Because of the uniqueness property of $\mathcal{B}$, in the communication graph creation of *Block-to-Part* integers can be exchanged instead of global identifiers of entities. Global identifiers are represented as 64-bit integers. This doubles size of the exchanged messages at this step. Moreover, the data post-processing using a quick sort algorithm induces an additional overhead. Still, as mentioned above for *Block-to-Part*, these results are reasonable taking into account the context of use.

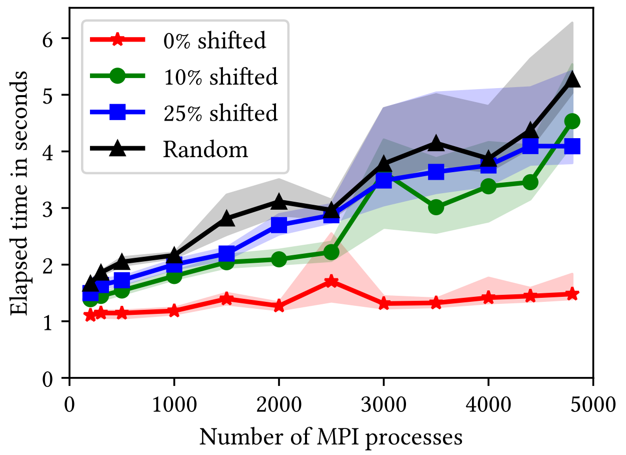

Figure 15: Mean elapsed time (in s) (with the range from minimum to maximum in shaded color) of a *Part-to-Block* communication for increasing number of MPI processes for 600 000 integers sent per process

This study revealed that the density of the communication graph is a crucial factor for the performance of this framework. For computational geometry applications, a way of inducing a sparse communication graph is to use geometric partitioning, which is, as Devine et al. [12] underlines, an overlooked technique allowing to quickly generate a partitioning using only geometric coordinates of the mesh points. In *Blinded Reference*, such a partitioning method is implemented, based on the Hilbert space filling curve. As explained by Bader [6], space-filing curves can be used on a variety of multi-dimensional items to create a mapping which will ensure that close-by items will be mapped to neighbouring items in the target set. Partitioning using space filing curves does not guarantee the output to be connected but this is not mandatory for the geometric problem we aim to solve. Note that this is a good solution for our applications since only geometric locality is important. By doing this we make sure that the communication graph is as concentrated around the main diagonal as possible.

5 CONCLUSION

In this paper an innovative data movement framework to easily load-balance unitary independent tasks on distributed-memory architectures has been presented. Abstracting communications and encapsulating the MPI sublayer makes it easier to manipulate complex structures with a satisfactory efficiency. Consequently, difficult implementations are avoided to get back to an embarrassingly parallel setting by using the tools from the *Blinded Reference* library. The concepts and implementation being compatible with any set of items, this framework is suitable for load-balancing requirements in a wide range of applications.

As things stand, strongly concentrating exchanges around the main diagonal of the communication graph reveals interesting. Several avenues can be explored to gain sparsity and adapt communications to it. First, switching from a graph-based partitioning to a geometric partitioning is relevant for computational geometry applications. An optimal global numbering generated using a space-filing curve is an effective way to make the distributions $\mathcal{B}$ and $\mathcal{P}$ similar, inducing little process-to-process communications. Then, ongoing work is carried out to take advantage of MPI-3's shared-memory and neighbourhood functionalities to specialize *Block-to-Part* and *Part-to-Block* for sparse communication patterns.

The increasing use and need of heterogeneous computing architectures requires adapting the presented tools. Ongoing work focuses on CPU-GPGPU hybridization. The load-balancing problem remains open because different architectures require different frameworks. Moreover, adapting collective communications to the network structure increases efficiency. The aim is to further extend the presented framework to take into account the unprecedented parallelism required for peta/exascale computation.

REFERENCES

[1] [n. d.]. Blinded Reference.
[2] 2024. CFD General Notation System Standard Interface Data Structures. https://cgns.github.io/CGNS_docs_current/sids/index.html.
[3] 2024. OSU Micro-Benchmarks 7.3. https://mvapich.cse.ohio-state.edu/benchmarks/.
[4] 2024. Visualization Toolkit. https://vtk.org/.
[5] Gene M Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilitie. In *AFIPS spring joint computer conference*.
[6] Michael Bader. 2012. *Space-Filling Curves*. Springer Berlin, Heidelberg.

Table 5: Breakdown of execution times per MPI process (in s) of *Block-to-Part* exchange on 4800 MPI processes with 600 000 entities per MPI process on a random communication graph

Block-to-Part exchange steps	minimum	mean	maximum
Parallel binary search gap	0.69169	0.69608	0.75033
Exchanges for communication graph creation	1.09230	1.14167	1.20343
Data exchange	0.95447	0.97010	1.00124

Table 6: Breakdown of execution times per MPI process (in s) of *Part-to-Block* exchange on 4800 MPI processes with 600 000 entities per MPI process on a random communication graph without weights

Part-to-Block exchange steps	minimum	mean	maximum
Generate distribution	0.03597	0.04551	0.05122
Parallel binary search gap	0.62758	0.62924	0.68985
Exchanges for communication graph creation	2.05819	2.23897	3.04367
Data post-processing	0.71671	0.73666	0.83285
Data exchange	1.59580	1.62053	1.65560

[7] B. Brandfass, T. Alrutz, and T. Gerhold. 2013. Rank reordering for MPI communication optimization. *Computers Fluids* 80 (2013), 372–380. Selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011.

[8] Andrew W. Cary, John Chawner, Earl P. Duque, William Gropp, William L. Kleb, Raymond M. Kolonay, Eric Nielsen, and Brian Smith. [n. d.]. *CFD Vision 2030 Road Map: Progress and Perspectives.*

[9] CCRT-CEA. 2024. TOPAZE. https://www-ccrt.cea.fr/fr/moyen_de_calcul/index.htm.

[10] François Pellegrini Cédric Chevalier. 2008. PT-SCOTCH: A tool for efficient parallel graph ordering. *Parallel Comput.* 34 (2008), 318–331.

[11] Karen Devine, Erik G. Boman, Robert T. Heaphy, Bruce Hendrickson, and Courtenay T. Vaughan. 2002. Zoltan data management services for parallel dynamic applications. *Computing in Science & Engineering* 4, 2 (2002), 90–96.

[12] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Bruce A. Hendrickson, James D. Teresco, Jamal Faik, Joseph E. Flaherty, and Luis G. Gervasio. 2005. New challenges in dynamic load balancing. *Applied Numerical Mathematics* 52, 2 (2005), 133–152. ADAPT '03: Conference on Adaptive Methods for Partial Differential Equations and Large-Scale Computation.

[13] Akio Doi and Akio Koide. 1991. An efficient method of triangulating equi-valued surfaces by using tetrahedral cells. *IEICE TRANSACTIONS on Information and Systems* 74, 1 (1991), 214–224.

[14] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, Franck Cappello, Barbara Chapman, Xuebin Chi, Alok Choudhary, Sudip Dosanjh, Thom Dunning, Sandro Fiore, Al Geist, Bill Gropp, Robert Harrison, Mark Hereld, Michael Heroux, Adolfy Hoisie, Koh Hotta, Zhong Jin, Yutaka Ishikawa, Fred Johnson, Sanjay Kale, Richard Kenway, David Keyes, Bill Kramer, Jesus Labarta, Alain Lichnewsky, Thomas Lippert, Bob Lucas, Barney Maccabe, Satoshi Matsuoka, Paul Messina, Peter Michielse, Bernd Mohr, Matthias S. Mueller, Wolfgang E. Nagel, Hiroshi Nakashima, Michael E Papka, Dan Reed, Mitsuhisa Sato, Ed Seidel, John Shalf, David Skinner, Marc Snir, Thomas Sterling, Rick Stevens, Fred Streitz, Bob Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne Trefethen, Mateo Valero, Aad van der Steen, Jeffrey Vetter, Peg Williams, Robert Wisniewski, and Kathy Yelick. 2011. The International Exascale Software Project roadmap. *The International Journal of High Performance Computing Applications* 25, 1 (2011), 3–60.

[15] Yvan Fournier. 2020. *Massively Parallel location and exchange tools for unstructured mesh-based CFD.* Technical Report. EDF R&D.

[16] Y Fournier, J Bonelle, C Moulinec, Z Shang, AG Sunderland, and JC Uribe. 2011. Optimizing Code_Saturne computations on Petascale systems. *Computers & Fluids* 45, 1 (2011), 103–108.

[17] Kirk Schloegel George Karypis and Vipin Kumar. 1997. *PARMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library.* Technical Report. University of Minnesota.

[18] Daniel A. Ibanez, E. Seegyoung Seol, Cameron W. Smith, and Mark S. Shephard. 2016. PUMI: Parallel Unstructured Mesh Infrastructure. *ACM Trans. Math. Softw.* 42, 3, Article 17 (2016), 28 pages.

[19] Cédric Lachat. 2013. *Conception et validation d'algorithmes de remaillage parallèles à mémoire distribuée basés sur un remailleur séquentiel.* Theses.

[20] William E. Lorensen and Harvey E. Cline. 1987. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '87).* Association for Computing Machinery, New York, NY, USA, 163–169.

[21] Tom Peterka, Robert Ross, Wesley Kendall, Attila Gyulassy, Valerio Pascucci, Han-Wei Shen, Teng-Yok Lee, and Abon Chaudhuri. 2011. Scalable Parallel Building Blocks for Custom Data Analysis. In *Proceedings of Large Data Analysis and Visualization Symposium LDAV'11.*

[22] Alexandre Suss, Ivan Mary, Thomas Le Garrec, and Simon Marié. 2023. Comprehensive comparison between the lattice Boltzmann and Navier–Stokes methods for aerodynamic and aeroacoustic applications. *Computers Fluids* 257 (2023), 105881.

[23] CORPORATE The MPI Forum. 1993. MPI: a message passing interface. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing.* 878–883.

[24] Martin Ferrand Yvan Fournier. 2024. Parallel Location and Exchange. https://github.com/code-saturne/libple.

[25] J. Zhang, J. Brown, S. Balay, J. Faibussowitsch, M. Knepley, O. Marin, R. Mills, T. Munson, B. F. Smith, and S. Zampini. 2022. The PetscSF Scalable Communication Layer. *IEEE Transactions on Parallel & Distributed Systems* 33, 04 (2022), 842–853.

Received 25 January 2024