GAPAR: A GROUP-THEORETIC FRAMEWORK FOR SYMMETRY-AWARE PARALLEL COMPUTING

Di Zhang
School of Advanced Technology
Xi'an Jiaotong-Liverpool University
Suzhou, Jiangsu, China
di.zhang@xjtlu.edu.cn

October 18, 2025

ABSTRACT

Modern parallel computing frameworks, such as the Message Passing Interface (MPI), provide low-level primitives for communication but lack high-level abstractions for exploiting the inherent symmetries present in many scientific and machine learning problems. This forces programmers to manually map algorithmic symmetry onto process topologies, leading to complex, error-prone code and suboptimal performance, especially when symmetry is broken or system faults occur. This paper introduces GAPAR, a novel parallel computing framework grounded in computational group theory. GAPAR allows programmers to declaratively specify the algebraic symmetry of their problem. The framework then automates key aspects of parallelization, including domain decomposition, communication scheduling, and load balancing, by leveraging the structure of the specified group and its action on the computational domain. We present the design philosophy and a concrete implementation blueprint for GAPAR, contrast it with topology-based approaches like MPI, and demonstrate its expressiveness and potential for performance optimization through two concrete examples: a symmetric heat equation solver and a Graph Neural Network (GNN) operating on highly symmetric graphs. Our conceptual analysis suggests that GAPAR can significantly reduce development complexity while enabling sophisticated, dynamic optimizations that are infeasible with current paradigms.

1 Introduction

Parallel computing is the cornerstone of high-performance scientific simulation and large-scale machine learning. The dominant programming model for distributed-memory systems, the Message Passing Interface (MPI) [1], provides a portable and powerful set of primitives. While MPI offers constructs like CARTESIAN topologies to map processes to geometric grids, its abstraction level remains low. Programmers are responsible for explicitly managing data decomposition, inter-process communication, and synchronization, which often results in code that is verbose, difficult to maintain, and brittle to changes in problem parameters or system state.

A significant class of computational problems—from molecular dynamics and computational fluid dynamics to graph processing on structured networks—exhibits profound mathematical *symmetry*. These symmetries can be described formally by *groups* and their *actions* on the computational domain. For instance, a square computational mesh is invariant under the dihedral group D_4, and a graph may possess a non-trivial automorphism group. Current parallel frameworks are *symmetry-agnostic*; any exploitation of this structure is ad-hoc, manual, and hard-coded by the programmer. This leads to several shortcomings:

- **Development Complexity**: Manual implementation of ghost layer exchanges and collective operations in MPI is tedious and error-prone.
- **Suboptimal Performance**: Static decompositions may not adapt to runtime symmetry breaking (e.g., localized phenomena).

- **Lack of Resilience**: A single process failure typically crashes the entire application, as the framework has no structural understanding of task redundancy introduced by symmetry.

This paper proposes a paradigm shift: a parallel computing framework where the programmer declaratively specifies the problem's symmetry group, and the runtime automatically derives an efficient parallel execution plan. We present the design and a concrete implementation blueprint for GAPAR (Group-based Adaptive Parallelism), a framework that uses computational group theory as its foundational abstraction. By elevating symmetry from an implicit property to a first-class citizen, GAPAR aims to simplify parallel programming, enhance performance portability, and introduce novel capabilities for adaptive load balancing and fault tolerance.

2 Background and Related Work

2.1 Topology-Based Parallelism with MPI

MPI's `MPI_Cart_create` allows for the logical arrangement of processes in a grid. This is useful for structuring communication in problems like finite difference stencils. However, an MPI topology is merely a *labeling* of processes; it has no semantic understanding of the geometry or symmetry it is meant to represent. Communication patterns (e.g., `MPI_Sendrecv` for ghost cell exchanges) must be explicitly coded by the developer. Frameworks like PETSc [2] provide higher-level abstractions for linear algebra and meshes but still rely on the programmer to define the decomposition and do not formally reason about symmetry.

2.2 Computational Group Theory

Computational Group Theory (CGT) is a well-established field concerned with developing algorithms for solving problems in group theory [3]. Software systems like GAP [4] and Magma are capable of computing properties of large finite groups, such as subgroups, conjugacy classes, and automorphisms. The application of CGT to parallel computing has been mostly confined to specialized domains, such as enumerating combinatorial objects or solving specific problems like graph isomorphism [5]. GAPAR seeks to generalize this connection, using CGT as a core runtime component for general-purpose parallel computing.

2.3 Symmetry in Computing

The use of symmetry to reduce computational cost is a classic technique, often called "symmetry breaking" in constraint satisfaction or "exploiting fundamental domains" in physics. In high-performance computing, it is typically done manually. Recent work in Machine Learning, specifically on Equivariant Neural Networks [6], has formalized the use of group theory in model architecture design to guarantee equivariance to input transformations. GAPAR draws inspiration from this philosophy but applies it to the parallelization process itself, aiming for *parallelization-equivariance* to the problem's symmetry group.

3 The GAPar Design

The core thesis of GAPAR is that by expressing a problem's symmetry through a group G and its action on a domain X, the framework can automate decomposition, communication, and optimization.

3.1 Core Abstractions

Definition 1 (Symmetry Group Declaration)**.** *The programmer specifies the problem's symmetry group G (e.g., a cyclic group C_n, dihedral group D_n, or a permutation group). This can be done by name for common groups or via generators.*

Definition 2 (Domain and Group Action)**.** *The computational domain X (e.g., a mesh, a graph) is defined. The programmer specifies how G acts on X (i.e., how elements $g \in G$ transform points $x \in X$). The field data (e.g., temperature, node features) is tagged with its transformation type (scalar, vector, etc.) under this action.*

3.2 System Architecture

GAPAR's runtime consists of several key components:

1. **Symmetry Analyzer**: Accepts the user's declarative specification of G and its action on X. It uses an integrated CGT library (e.g., a GAP interface) to compute group properties like generators, subgroups, and orbits.
2. **Decomposition Engine**: Partitions the domain X into regions corresponding to the *orbits* of a chosen subgroup of G. A standard choice is to partition X into a set of *fundamental domains*, where each process computes the solution for one representative from each orbit. This guarantees load balance for the symmetric case.
3. **Communication Manager**: Derives necessary communication from the group's *Cayley graph* defined by its generators. For example, a ghost cell exchange corresponds to communicating with processes associated with the action of group generators on the local domain. It can employ optimized, persistent communication channels based on this graph.
4. **Dynamic Optimizer**: Monitors application state. If symmetry is broken (e.g., load imbalance is detected), it can recompute the decomposition based on a smaller *subgroup* of G that remains relevant, dynamically redistributing work.

3.3 Contrast with MPI's Topology Approach

The difference is fundamental. In MPI, a 2D grid topology is a static map. In GAPAR, a dihedral group D_4 action is a *dynamic specification of allowable transformations*.

- **Decomposition**: MPI: Manual choice of `dims[2]`. GAPAR: Automatic derivation from the orbit structure of G.
- **Communication**: MPI: Manual `Sendrecv` calls for neighbors. GAPAR: Automatic derivation from group generators.
- **Optimization**: MPI: Static. GAPAR: Can dynamically reconfigure based on algebraic structure (e.g., falling back to a cyclic subgroup C_2 if reflection symmetry is broken).

4 Implementation Blueprint

Translating the GAPAR design into a functional library requires a layered architecture that bridges abstract algebra and high-performance computing. We propose a concrete implementation strategy based on four layers.

4.1 Layer 1: Core Abstraction and CGT Binding

This layer provides the foundational C++ classes (or Python equivalents with C++ bindings for performance) and interfaces with a CGT backend.

- **Class** `Group`: An abstract base class representing a finite group. Concrete subclasses (`FiniteGroup`, `PermutationGroup`) would wrap an embedded CGT library like **GAP** via its C library interface (libgap) or a system call. Key methods include:

```
class Group {
 std::vector<GroupElement> generators();
 std::vector<GroupElement> elements();
 Subgroup lattice();
 GroupElement compose(GroupElement a, GroupElement b);
 // ...
};
```

- **Class** `GroupAction`: Encapsulates the action of a `Group` on a `Domain`. For a grid, this would apply rotations/reflections; for a graph, it would permute nodes.

```
class GroupAction {
 DomainElement act(GroupElement g, DomainElement x);
 std::vector<Orbit> compute_orbits();
 bool is_fundamental_domain(Subset S);
};
```

- **Initialization**: For common groups (C_n, D_n), predefined specifications can be loaded. For graphs, an initial, potentially expensive, automorphism group computation using a library like **Bliss** or **Nauty** is performed during setup.

4.2 Layer 2: Decomposition and Metadata Management

This layer uses the algebraic structures from Layer 1 to partition the problem.

- **Class** `DecompositionStrategy`: An interface for different partitioning schemes (e.g., `OrbitDecomposition`, `FundamentalDomainDecomposition`).
- **Process-Orbit Mapping**: The primary strategy maps each orbit (or a union of orbits) to an MPI process. This creates a `std::map<ProcessRank, std::vector<Orbit>>`. The framework ensures that the image of any local data under a group generator is sent to the correct process.
- **Metadata Storage**: Each process stores:
  1. Its *local* set of domain elements (e.g., grid points, graph nodes).
  2. The *ghost* elements, which are the images of its local boundary under the group's generating set.
  3. A `std::map<DomainElement, ProcessRank>` to locate the owner of any ghost element.

4.3 Layer 3: Communication Engine

This is the performance-critical core that replaces manual `Sendrecv` logic.

- **Cayley Graph Derivation**: The framework constructs the Cayley graph of the group G with respect to its generating set S. Each process is associated with a vertex in this graph (or a coset, depending on decomposition).
- **Persistent Communication Channels**: For stencil-based computations (Example 5.1), the framework pre-computes and establishes `MPI_Send_init` and `MPI_Recv_init` for each generator in S. This minimizes communication overhead per iteration.
- **Collective Operations**: Operations like REDUCE can be optimized. A reduction over the entire domain can be performed by first reducing within orbits (which are structurally identical) and then combining the results, potentially reducing communication volume.
- **API**: The user-facing API is simple:

```
Decomposition decomp = ...;
auto comm_handle = decomp.create_communicator();
comm_handle.start_ghost_exchange(local_field); // Non-blocking
comm_handle.finish_ghost_exchange();
```

  The framework handles all rank tagging and synchronization.

4.4 Layer 4: Runtime and Dynamic Optimizer

This layer enables GAPAR's adaptive behavior.

- **Performance Monitoring**: The runtime periodically samples computation time per process. A significant deviation indicates symmetry breaking.
- **Subgroup Detection**: Upon detecting imbalance, the optimizer analyzes the load distribution. It then queries the `Group` object from Layer 1 for a maximal subgroup H that is consistent with the current load pattern. For instance, if a hotspot breaks reflection symmetry but preserves 180° rotation, it identifies $H = C_2$.
- **Dynamic Recomposition**: The framework triggers a recomputation of the decomposition using H instead of G. This involves:
  1. Computing the new orbits under H (which are larger than those under G).
  2. Redistributing data according to the new orbit-process map using MPI collective communication.
  3. Rebuilding the communication channels in Layer 3 based on the generators of H.

  This process is costly but is amortized over long-running simulations where performance degradation would otherwise be permanent.

4.5 Prototyping Path

A viable path is to first implement a prototype in Python, leveraging `mpi4py` and the GAP system through a pipe or library call. This allows for rapid validation of the core concepts (decomposition, communication derivation) on small problems. The performance-critical components (Communication Engine, Layer 3) can then be incrementally rewritten in C++/MPI for production use.

5 Conceptual Examples

5.1 Example 1: 2D Heat Equation on a Square Domain

Problem: Solve the 2D heat equation on a square domain with initial central hotspot. The domain has D_4 symmetry.

- **MPI+Topology Approach**: The programmer creates a 2D Cartesian topology, manually divides the global grid into $P \times P$ subgrids, and writes explicit `Sendrecv` logic for North, South, East, West neighbors. Code is long and rigid.
- **GAPAR Approach**:

```
// Declarative Specification
Group G = DihedralGroup(4);
Domain X = SquareGrid(1000, 1000);
Field temperature = ScalarField(X, G);

// Computational Kernel (User focuses on physics)
@kernel void stencil(Field t_new, Field t_old) { ... }

// Runtime automates the rest
Decomposition decomp = GAPar.decompose(X, G);
decomp.execute(stencil, temperature);
```

Advantages:

1. **Simplicity**: Code is reduced to its essentials. No communication logic.
2. **Correctness**: Ghost exchanges are handled correctly by the framework.
3. **Adaptivity**: If the hotspot moves off-center, breaking D_4 symmetry, GAPAR can detect load imbalance and dynamically reconfigure the decomposition based on the remaining C_2 (180° rotation) symmetry, redistributing work without programmer intervention.

5.2 Example 2: Graph Neural Network on a Symmetric Graph

Problem: Train a GNN on a graph Γ with a large automorphism group $\mathrm{Aut}(\Gamma)$ (e.g., a circular ladder graph).

- **MPI+Topology Approach**: The programmer uses a generic graph partitioning library (e.g., Metis) to partition the nodes, minimizing edge cuts. The resulting partition is arbitrary with respect to the graph's symmetry, leading to potentially unnecessary communication. Managing node/edge feature aggregation across partitions requires complex book-keeping.
- **GAPAR Approach**:

```
Graph Γ = load_graph("symmetric_graph.gml");
Group G = GAPar.compute_automorphism_group(Γ); // Uses Nauty/Traces
Domain X = Γ; // The domain is the graph itself
Field node_features = GraphField(X, G); // Features are permuted by G

@kernel void gnn_layer(Field h_new, Field h_old) {
 // For each node, aggregate messages from neighbors
 for_node n in local_nodes {
  h_new[n] = AGGREGATE( h_old[neighbors(n)] );
 }
}
```

```
Decomposition decomp = GAPar.decompose(Γ, G);
decomp.execute(gnn_layer, node_features);
```

Advantages:

1. **Exploited Redundancy**: GAPAR partitions the graph into orbits of $\mathrm{Aut}(\Gamma)$. Nodes in the same orbit are structurally identical and can be processed by the same process, or their computations can be efficiently replicated without storage overhead for features.
2. **Optimal Communication**: The communication pattern for message passing is derived from the group action, ensuring that only non-redundant, necessary data is exchanged.
3. **Fault Tolerance**: If a process managing one orbit fails, its work can be reassigned to the process managing a symmetric orbit, as the computation is identical, enabling graceful degradation.

6 Conclusion and Future Work

We have presented the vision and a concrete implementation blueprint for GAPAR, a parallel computing framework that uses group theory as a first-class abstraction. By moving from an imperative, topology-based model to a declarative, symmetry-based one, GAPAR has the potential to dramatically simplify the development of parallel applications for a vast class of symmetric problems. The implementation strategy, layered around a CGT core, demonstrates the feasibility of this approach. The examples in Section 5 illustrate the framework's promise in reducing code complexity, enabling dynamic optimizations, and providing inherent fault tolerance.

The path forward involves several challenging but exciting research directions. A full-fledged prototype of GAPAR must be implemented, integrating a CGT library with a parallel runtime. The performance overhead of online symmetry analysis must be evaluated against its benefits. The framework needs to be extended to handle more complex group actions and hybrid scenarios with broken symmetry. Finally, formalizing the concept of "parallelization-equivariance" and exploring its integration with equivariant machine learning models presents a compelling long-term research agenda. GAPAR is not merely an incremental improvement but a step towards a more intelligent and algebraic foundation for parallel computing.

References

[1] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 4.0, 2021. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[2] Satish Balay and others. PETSc Users Manual. Technical Report ANL-95/11 - Revision 3.19, Argonne National Laboratory, 2022.

[3] Derek F. Holt, Bettina Eick, and Eamonn A. O'Brien. *Handbook of Computational Group Theory*. Chapman and Hall/CRC, 2005.

[4] GAP – Groups, Algorithms, and Programming, Version 4.12.2. The GAP Group, 2022. https://www.gap-system.org.

[5] Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, 25(1):42–65, 1982.

[6] Taco S. Cohen and Max Welling. Group Equivariant Convolutional Networks. In *Proceedings of the 33rd International Conference on Machine Learning*, 2016.