UNIVERSITY OF CALIFORNIA
SANTA CRUZ

ACTOR GRAPH LIBRARY

A project submitted in partial satisfaction of the
requirements for the degree of

MASTERS OF SCIENCE

in

COMPUTER SCIENCE

by

Tanuj Gupta

June 2023

The Master's Project of Tanuj Gupta
is approved:

Professor Scott Beamer, Chair

Professor Andrew R. Quinn

Alexander Wolf
Dean and Distinguished Professor of Computer Science

Table of Contents

List of Figures

Abstract

Actor Graph Library

by

Tanuj Gupta

Graph processing is fundamental in various domains, including social network analysis, recommendation systems, and biological network analysis. The Actor Graph Library (AGL) aims to provide researchers and developers with a user-friendly interface and efficient execution for distributed graph processing. AGL leverages the capabilities of the HClib Actor framework, addressing the limitations of existing graph processing libraries. It offers a high-level interface and a set of functionalities specifically designed to streamline distributed graph processing. With AGL, users can generate graphs and build them using various approaches.

In this project, we present AGL's design, implementation, and evaluation. The evaluation examines factors such as build time, load imbalance, and kernel performance using breadth-first search (BFS) as a benchmark. Our evaluation demonstrates that AGL is a promising starting point for a graph processing library built on top of HClib Actor. However, our findings also highlight the need to improve build time and address out-of-memory issues. We propose future directions for extending AGL, such as incorporating additional generators and builders, further studying and benchmarking BFS performance, and comparing AGL against other distributed graph libraries.

Acknowledgments

I would like to express my sincere gratitude to all those who have supported and contributed to the successful completion of this research project.

I would like to express my deepest gratitude to my advisor, Scott Beamer, whose guidance and support have been pivotal to the success of this research project. Throughout the course of this study, Scott has consistently provided valuable insights and ideas on how to approach various problems and challenges. His extensive knowledge of the field and his ability to reference relevant papers and resources have greatly enriched the research process. Moreover, Scott's keen eye for detail has helped identify potential pitfalls in my approach and provided invaluable suggestions for improvement. I am truly grateful for his mentorship and the countless hours he has dedicated to reviewing and refining my work.

I am grateful to my reader and co-advisor, Andrew Quinn, for his valuable insights and expertise in reviewing my project. His constructive feedback has immensely contributed to improving the quality and clarity of my work.

Special thanks go to Amogh Lonkar, a PhD student in my lab, for his guidance, collaboration and assistance throughout this project. His willingness to share ideas have greatly influenced the outcomes of my project. Additionally, I would like to express my gratitude to all the researchers in my lab who have created a stimulating academic environment and provided me with invaluable opportunities for learning and growth.

Chapter 1

Introduction

In this paper, we present the Actor Graph Library (AGL), a robust and scalable graph processing library developed on top of the HClib Actor framework. HClib Actor is a powerful framework that provides support for actor-based programming, enabling efficient and concurrent execution of tasks [14]. By leveraging the capabilities of HClib Actor and addressing the limitations of existing graph processing libraries, AGL offers a user-friendly interface and efficient execution for graph analytics tasks. Our aim is to provide researchers and developers with a robust tool for distributed graph processing that simplifies the development of graph-based applications by abstracting away standard functions and complexities associated with graph processing while leveraging the scalability and performance advantages offered by HClib Actor.

AGL distinguishes itself by adhering to design principles that promote extensibility and flexibility. It offers users various approaches to generate and build graphs, enabling them to represent and manipulate graph data in a way that suits their specific

needs. The library provides a range of graph processing algorithms and operations, empowering developers to analyze and process graph structures efficiently. This extensibility allows users to explore and experiment with different load balancing techniques. For instance, in our work, we have integrated five different hash-based load balancing techniques into AGL, enabling us to thoroughly test and compare their effectiveness in balancing the load for RMAT graphs. This ability to easily add and test multiple load balancing techniques is made possible by the design of AGL, which fosters a playground for exploring and optimizing graph processing performance.

AGL encapsulates the complexities of distributed graph processing, allowing users to concentrate on the core aspects of their research or application development. Furthermore, AGL serves as a platform for advancing load balancing techniques and exploring innovative strategies within the context of actor based programming, contributing to the broader field of distributed graph processing.

Overall, AGL presents a valuable tool that combines the benefits of HClib Actor's design principles, extensibility, and load balancing capabilities. It offers a user-friendly interface, efficient execution, and a playground for exploring and optimizing graph processing techniques. By providing researchers and practitioners with a powerful and flexible graph processing library, AGL aims to advance the field of distributed graph processing and facilitate the development of scalable and efficient graph-based applications.

1.1 Actor Graph Library - Benefits and Features

By leveraging HClib Actor's capabilities, AGL enables large-scale graphs processing in a distributed environment while offering high performance, flexibility, ease of use, and interoperability with other graph processing frameworks and tools. AGL provides following benefits and features:

- **Scalability:** AGL enables efficient processing of large-scale distributed graphs by leveraging distributed algorithms, parallel processing, and HClib Actor for optimized resource utilization and scalability.

- **Performance:** AGL delivers high-performance graph processing through optimized data structures, algorithms, and parallel processing. It maximizes efficiency by optimizing data access patterns, leveraging parallelism, and minimizing communication overhead using HClib Actor's message aggregation and Partitioned Global Mailbox.

- **Flexibility:** AGL offers customizable options for graph representation. AGL also provides flexibility in graph shuffling techniques to improve data locality and algorithm performance.

- **Ease of Use:** AGL features a well-documented API that abstracts complexities, offering high-level abstractions for graph operations. The paper includes examples showcasing AGL's functionality.

- **AGL is Extendable:** AGL supports custom generator and builder implementa-

tions, empowering users to tailor the library to their specific needs and incorporate domain-specific optimizations.

In the following chapters of this research paper, we will explore various aspects of the Actor Graph Library (AGL). In Chapter 1, we provide a background on HClib Actor and an overview of existing graph processing libraries. We also discuss the limitations of these libraries, leading to the introduction of AGL and its benefits and features. In Chapter 3 and 4, we dive into the implementation details of AGL, focusing on the Graph Generator Module (Chapter 3) and the Graph Builder Module (Chapter 4). In Chapter 5 we evaluate the performance of AGL generators and builders. Finally, we conclude (Chapter 6) by summarizing the findings and contributions of AGL and propose future work (Chapter 7)

Chapter 2

Background

In this background section, we provide a brief overview of the key concepts and technologies that underpin AGL's design and functionality.

2.1 Overview

To provide a solid foundation for understanding AGL, it is essential to delve into the HClib Actor framework [14]. HClib Actor is a scalable actor-based programming system specifically designed to accelerate irregularly distributed applications. It combines the productive and asynchronous message-passing approach of the actor model with the efficiency and scalability of the HClib runtime.

The actor model [1], popular in the enterprise and cloud computing platforms [2], provides a productive approach for handling distributed objects [13]. Languages such as Erlang, Scala, and Rust have successfully implemented the actor model, enabling asynchronous message-passing communication [3] [12] [11]. However, the actor model's

application in the context of Partitioned Global Address Space (PGAS) applications on clusters has been limited.

HClib Actor addresses the limitations of PGAS applications by offering a new programming system. To address this, HClib Actor offers a new programming system that leverages the concept of "selectors" - actors with multiple mailboxes [10]. In the HClib Actor framework, each actor maintains its own logical mailbox, allowing both actors and non-actors to send messages asynchronously [14]. This asynchrony enables flexible and efficient message processing within the mailboxes. HClib Actor also introduces the Partitioned Global Mailbox, a high-throughput communication mechanism for efficient message passing in distributed applications. With HClib Actor, programmers can express point-to-point remote operations as fine-grained asynchronous actor messages, simplifying programming complexities related to message aggregation and termination detection.

By extending the classical Bulk Synchronous Parallelism model with fine-grained asynchronous communications within a phase or superstep, HClib Actor strikes a desirable balance between productivity and performance for PGAS applications. It enables developers to express all communications at a fine-grained granularity natural to the application while achieving scalable performance. In experiments using seven irregular mini-applications from the Bale benchmark suite executed on the NERSC Cori system with 2048 cores, HClib Actor demonstrates geometric mean performance improvements of 20X compared to standard PGAS versions (UPC [5] and OpenSHMEM [7]) while maintaining comparable productivity [14].

2.2 Overview of Existing Graph Processing Libraries

We highlight several of the most related graph processing libraries that have been developed to address the challenges of distributed graph analytics. These libraries provide a range of graph algorithms and data structures optimized for high-performance computing environments.

1. **CombBLAS:** CombBLAS is a comprehensive graph processing library designed for efficient and scalable computations on distributed memory systems [4]. It provides a rich set of graph algorithms and optimizations, allowing users to perform various graph analytics tasks with high performance. The library's design and implementation focus on leveraging parallelism and distributed memory to handle large-scale graphs effectively. CombBLAS has been widely used in the research community for its capability to handle complex graph computations and its ability to scale to massive datasets.

2. **X10 Graph Library:** X10, a high-performance and high-productivity programming language [15], features a graph library designed explicitly for PGAS programming models [18]. The X10 Graph Library leverages the PGAS paradigm to provide scalable and efficient graph processing capabilities [16]. It offers a range of graph algorithms and optimizations, enabling developers to harness the power of distributed memory systems.

Chapter 3

AGL - Graph Generator

The AGL library, along with HClib Actor, are implemented in C++, which was a deliberate choice to leverage the existing capabilities of HClib Actor and its underlying libraries. HClib Actor itself utilizes C++ libraries such as the Habanero C/C++ library (HClib) and libgetput, which offers multiple backends (UPC and OpenSHMEM) for users to choose from during the library build process. By building AGL in C++, we benefit from the high-performance and widespread adoption of the language, enabling efficient graph processing. Furthermore, this choice of implementation language facilitates seamless integration with other components of graph analytics workflows, taking advantage of the performance optimizations and compatibility with existing C++ libraries. Graph generation is crucial in various graph analytics tasks, providing the initial set of vertices and edges for subsequent analysis and processing.

The Graph Generator AGL generates graphs structures in a distributed manner, and it offers different implementations to cater to various graph generation require-

ments.

The Graph Generator offers different implementations of a Uniform Random Graph Generator [8] and a RMAT Graph Generator [6]. These implementations leverage the capabilities of actors to efficiently distribute the graph generation process, enabling users to generate large-scale graphs.

3.1 Uniform Random Graph Generator:

The Uniform Random Graph Generator produces graphs with a uniform random distribution of edges across the vertices. It suits scenarios where a low diameter and uniform degree distribution are desired. The HClib Actor selector class associated with the Uniform Random Graph Generator is the `UniformGeneratorSelector`.

The `UniformGeneratorSelector` class extends the HClib Actor Selector class and provides the necessary logic to generate a uniform random graph using the HClib Actor framework. Its **constructor** takes parameters such as *SCALE* and *edge-factor*, which control the size and density of the generated graph. Additionally, it accepts a pointer to an object of the `ResultFromGenerator` class, which is responsible for storing the generated graph's details.

Note: In the actor model, computation primarily occurs within the actor's local memory. However, in this case, the generated result must be accessed outside the actor before termination. Therefore, the actor saves the result in the memory location specified by the pointer. This approach is adopted because the selector runtime in HClib

Actor currently does not support returning values upon termination.

3.2 RMAT Graph Generator

The RMAT Graph Generator implementation utilizes a recursive matrix algorithm to generate graphs that replicate the characteristics of real-world networks, such as power-law degree distributions, small-world properties, and community structures. The HClib Actor selector class associated with the RMAT Graph Generator is the `RmatGeneratorSelector`.

The `RmatGeneratorSelector` class, extending the HClib Actor Selector class, incorporates the recursive matrix algorithm and efficiently generates RMAT graphs in a distributed manner using the HClib Actor framework. Similar to the Uniform Random Graph Generator, the `RmatGeneratorSelector` **constructor** also takes parameters such as *SCALE*, *edge-factor*, and a pointer to an object of the `ResultFromGenerator` class.

It is important to note that the work performed by the `UniformGeneratorSelector` and `RmatGeneratorSelector` is done locally within the actor's memory space to adhere to the principles of actors and selectors. Only after all the graph generation work is completed the relevant details are saved in the generatorResult object, which resides outside of the actor's memory space. Additionally, the generator includes an `enum class` called `GraphGeneratorTypeEnum`, which is used as a pointer to determine which specific implementation to be used.

3.3 Main Class and Constructors

The generator's main class is called `Generator`. It serves as the primary entry point for graph generation and provides two constructors to initialize the object:

- `Generator`(*int edgefactor*, *int SCALE*, *GraphGeneratorTypeEnum type*): This constructor allows the user to specify the *edgefactor*, *SCALE*, and which graph generator to use in the `GraphGeneratorTypeEnum`.
- `Generator`(*int edgefactor*, *int SCALE*) : `Generator`(*edgefactor*, *SCALE*, *UniformGenerator*): This **constructor** is convenient to use when the user wants to generate a graph without worrying about the type of generation

, and it uses the `UniformGenerator` as a default.

`GenerateEL` is the main method of the Generator class, and it is responsible for initiating the graph generation process. Internally it invokes the appropriate selector class based on the graph generator type specified in the **constructor**.

3.4 Structure of the `ResultFromGenerator` Class

The `ResultFromGenerator` class is responsible for holding the results generated by the graph generator module. It provides various properties and data structures to store information about the generated graph. Here is the structure of the class:

```
class ResultFromGenerator {
public:
```

```
int total_pe;
int SCALE;
int edgefactor;
int64_t local_edges;
int64_t global_vertices;
int64_t global_edges;
EdgeList edgeList;
WEdgeList wEdgeList;
int64_t edgelist_size;
int64_t max_edgelist_size;
double make_graph_time;

ResultFromGenerator() {
    // Default constructor
}

ResultFromGenerator(const ResultFromGenerator& other) {
    // Copy constructor
    Implementation
}
```

```
    ResultFromGenerator operator=(const ResultFromGenerator& other) {
        // Assignment operator
        Implementation
    }
};
```

The `ResultFromGenerator` class includes properties such as *total_pe* (total number of processing elements), *SCALE* (graph scale), *edgefactor* (edge factor), *local_edges* (number of local edges), *global_vertices* (total number of vertices in the graph), and *global_edges* (total number of edges in the graph). These properties provide general information about the generated graph. The *make_graph_time* property records the time taken to generate the graph

Additionally, the class contains *edgeList* which represents the distributed edge list. The *edgelist_size* property specifies the local size of the edge list, while the *max_edgelist_size* property represents the maximum size of the edge list among all the edge lists. The `ResultFromGenerator` class serves as the container for the generated graph data, allowing it to be accessed and utilized by other parts of the library or by users of the AGL API.

Note: The `ResultFromGenerator` class currently includes a *wEdgeList* field. However, it is not currently utilized, nor is there any parameter available for users to create weighted graphs. However, in future iterations, the generator and its result could be extended to allow users to create weighted graphs.

3.5 Usage Instructions

To generate a graph using AGL, follow these steps:

1. Instantiate an object of the `Generator` class. The `Generator` class requires three parameters: *edgefactor*, *SCALE*, and *type*. The *edgefactor* parameter determines the number of edges per vertex, the *SCALE* parameter determines the size of the graph, and the *type* parameter specifies the type of graph generator to use. The *type* parameter should be chosen from the *GraphGeneratorTypeEnum enum* class, such as *RmatGenerator* or *UniformGenerator*. Here's an example of creating a generator object:

   ```
   Generator generator(edgefactor, SCALE, RmatGenerator);
   ```

2. Call the main method **GenerateEL()** on the *generator* object to generate the edge list. This method will initiate the graph generation process and return the result in an object of the `ResultFromGenerator` class. Here's an example of generating the edge list:

   ```
   ResultFromGenerator generatorResult = generator.GenerateEL();
   ```

3. Utilize the `ResultFromGenerator` object and its properties to access the generated graph data. The `ResultFromGenerator` class contains various properties that provide information about the generated graph, such as *total_pe*, *SCALE*, *edgefactor*, *local_edges*, *global_vertices*, *global_edges*, *edgeList*, *wEdgeList*, *edgelist_size*,

max_edgelist_size, and *make_graph_time*. Here's an example of accessing the generation time:

```
fprintf("time: %f", generatorResult.make_graph_time);
```

Note: It's important to manage the memory allocated for the generator and result objects appropriately by deleting them when they are no longer needed to avoid memory leaks.

3.6 Extending Generator with a Custom Implementation

AGL allows you to extend its functionality by adding your own custom implementations. This enables you to tailor the graph generation and building process according to your specific requirements. Adding custom implementations involves the following steps:

1. **Create an HClib Actor Selector Class:** To implement a new graph generator implementation, you need to create a new HClib Actor selector class inside the selector directory. This class should inherit from `HClib::Selector` and define the necessary behavior for your custom implementation. You can refer to the existing selector classes, such as `UniformGeneratorSelector` and `RmatGeneratorSelector`, as examples.

2. **Update the Enum Class:** Next, update the enum class `GraphGeneratorTypeEnum` to include a new type that represents your custom generator.

3. **Update the Main Class:** Update the main class `Generator` to include the

necessary logic for invoking your custom implementation. Modify the constructor or add new constructors to accept the additional parameters required for your custom implementation. Within the main method of the class **GenerateEL()**, incorporate conditional statements or switch cases to call the appropriate HClib Actor selector class based on the provided type.

Following these steps, you can seamlessly integrate your custom implementations into the AGL framework. This flexibility allows you to extend AGL's capabilities and adapt them to suit your graph processing needs. **Note:** When adding custom implementations, it's essential to ensure compatibility with the overall AGL design and adhere to the principles of actor-based programming to maintain the desired performance and scalability benefits.

Chapter 4

AGL - Graph Builder

The Graph Builder in AGL focuses on constructing the Compressed Sparse Row (CSR) representation of a graph in a distributed manner. It provides efficient methods to build CSR graphs using different vertex distribution schemes. Distributions are based on hash functions and include 1. Cyclic, 2. Range, 3. Snake, 4. Rotation, 4. Snake with Rotation. Each of these implementations has unique characteristics and advantages, allowing you to choose the one that best suits the graph building requirements.

4.1 CSRGraph: Compressed Sparse Row Graph Representation

The `CSRGraph` class in AGL's Graph Builder Module implements the Compressed Sparse Row (CSR) graph data structure, known for its efficiency in graph processing algorithms. It utilizes two important data structures: the index array and the neighbor's array.

```
class CSRGraph {
    private:
        std::vector<int> index;
        std::vector<int64_t> neighbors;
}
```

The index array represents the offset index for all local vertices assigned to a particular processing element (PE). Each component of the index array points to the starting position of the neighbors of a vertex. Specifically, the index[i] entry corresponds to the first neighbor index of the ith vertex on the PE, and index[i+1] represents the starting position of the neighbors of the (i+1)th vertex. The neighbor's array stores the actual neighboring vertices of each vertex. The ith vertex's neighbors are stored from index index[i] to index[i+1]-1 in the neighbor's array.

In addition to these structures, the `CSRGraph` class provides three key functions to handle vertex IDs:

1. int64_t local_to_vertex (int64_t pe, int64_t localv): Given a local vertex ID (*localv*) on a specific processing element (*pe*), this function maps it to the corresponding global vertex ID (v) in the entire graph.

2. int64_t vertex_to_local (int64_t v): Given a global vertex ID (v), this function determines the local vertex ID on the processing element where the vertex resides.

3. int64_t vertex_owner(int64_t v): Given a global vertex ID (v), this function identifies the processing element that owns the vertex.

The use of both local and global vertex IDs serves a crucial purpose in distributed graph processing. The local vertex IDs correspond to vertex indices within each processing element (PE)'s assigned graph portion. These local IDs enable efficient local operations and traversal without requiring global communication between PEs. On the other hand, global IDs allow PEs to identify and refer to vertices across the distributed graph, enabling seamless communication and coordination between different parts of the graph.

The `CSRGraph` class serves as the base class for various implementations, each tailored to a specific builder type. These implementations extend the `CSRGraph` class and provide their own implementations of the three virtual functions mentioned above. By customizing these functions, each `CSRGraph` implementation changes how vertices are allocated across PEs, which optimizes the graph representation.

4.2 CSRGraph Implementations and HClib Actor Selectors

The Graph Builder Module in the Actor Graph Library (AGL) provides different implementations for graph building, each tailored to meet specific requirements. These implementations are realized through a specific implementation of the `CSRGraph` class. Each of these implementations is hash function-based and utilizes the same HClib Actor selector class, `BuildGraphDeterministicHashSelector`. However, each implementation may require a different HClib Actor class for the graph-building process. Each implementation aims to balance vertices across Processing Elements (PE), but their

hash functions perform differently based on the input graph topology. Let's explore the different builder implementations along with their corresponding HClib Actor selector classes and the hash functions employed in each implementation. Figure 4.1 demonstrates how the five builders will allocate 16 vertices (labeled 0 to 15) and 4 PEs (labeled 0 to 3).

1. **Cyclic Builder**: `CyclicCSRGraph` assigns vertices to PEs in a cyclic manner.

2. **Range Builder**: `RangeCSRGraph` partitions the vertex range equally among the processing elements (PEs). This implementation assigns a contiguous range of vertices to each PE. This partitioning scheme guarantees that vertices within each PE's range are stored consecutively in memory, facilitating efficient local access and minimizing data movement between PEs during graph processing.

3. **Snake Builder**: `SnakeHashCSRGraph`, constructs the Compressed Sparse Row (CSR) graph using a snake-like pattern by alternating the cyclic order. This approach aims to achieve a more evenly distributed vertex assignment across the processing elements (PEs) when there is a bias towards lower or higher vertex IDs.

   It follows a snake-like pattern, where the vertices are allocated in a zigzag manner across the PEs. Additionally, the flipping of the assignment direction after each PE ensures a more uniform distribution of vertices and reduces the potential for bottlenecks in specific PE ranges.

4. **Rotation Builder**: `RotationHashCSRGraph` constructs the Compressed Sparse Row (CSR) graph by rotating the cyclic order. Instead of flipping the assign-

ment direction like the snake builder, the Rotation Builder introduces rotation to further enhance load balancing and reduce potential bottlenecks. This distribution enhances load balancing and reduces the likelihood of hotspots during graph processing.

5. **Snake with Rotation Builder**: `SnakeRotationCSRGraph` class combines the snake-like pattern with rotation to construct the Compressed Sparse Row (CSR) graph using both alternation (Snake) and Rotation. This approach aims to counter biases.

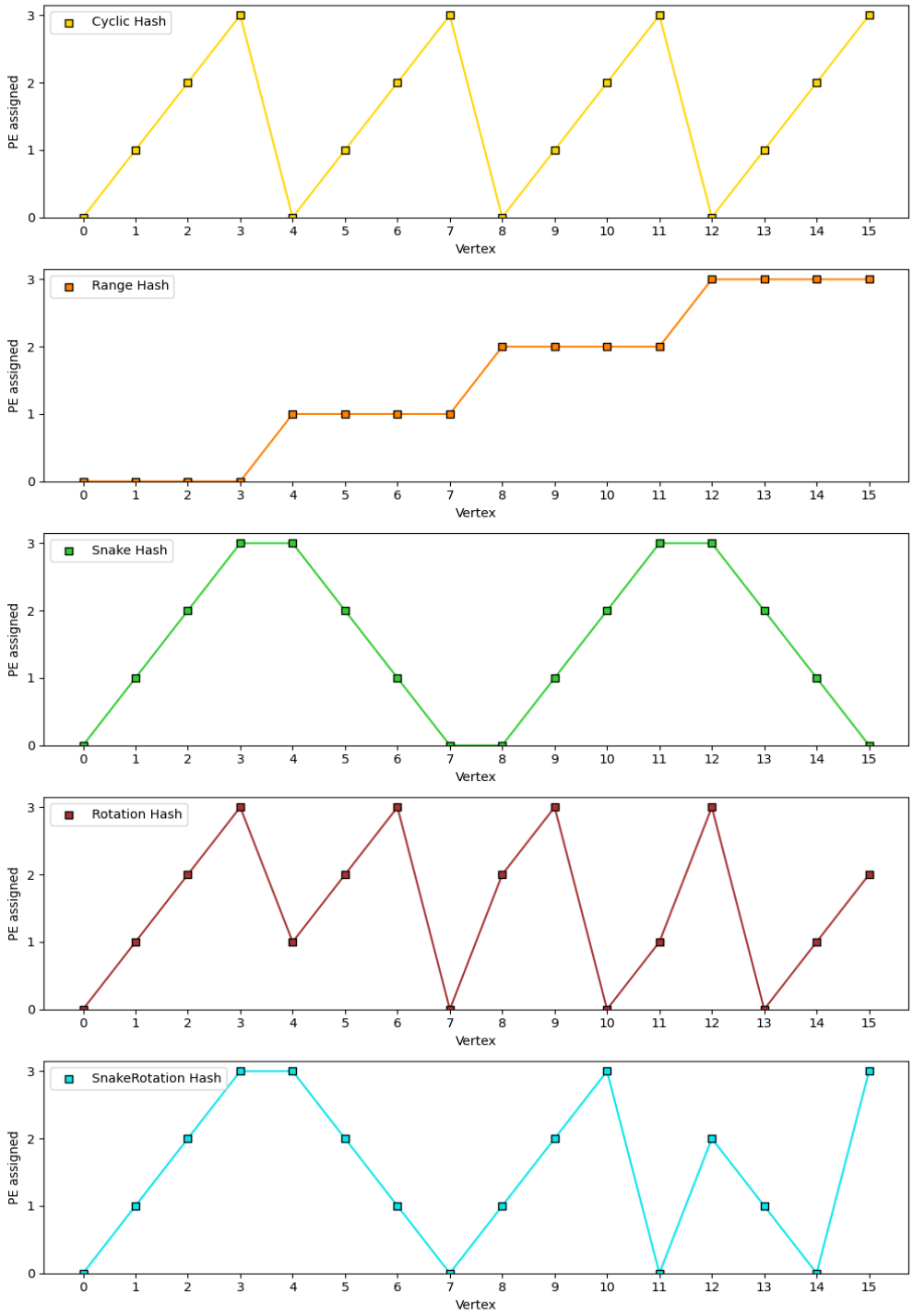

Figure 4.1: Vertex-to-PE Allocation of 16 vertices on 4 PEs (labeled 0 to 3) using 5 Hashing Techniques in AGL

4.3 Main Class and Constructors

The main class in the Graph Builder is `BuilderBase`. It serves as the central class responsible for coordinating the graph-building process and providing necessary functionality. The `BuilderBase` class includes constructors and methods to facilitate graph construction. The constructors of the `BuilderBase` class allow initializing the builder with different parameters, including the type of builder implementation to use:

- `BuilderBase`(*ResultFromGenerator generatorResult*): The first constructor is a shorthand that defaults to the Cyclic builder implementation.
- `BuilderBase`(*ResultFromGenerator generatorResult*, *GraphBuilderTypeEnum type*): The second constructor allows you to specify the type of builder implementation explicitly.

In the constructor of the `BuilderBase` class, depending on the builder type passed, the appropriate implementation of the CSR graph is instantiated. For example, if the builder type is Cyclic, the constructor creates a `CyclicCSRGraph` object and assigns it to the *csrGraph* property of the `ResultFromBuilder` object.

The `BuilderBase` class also includes a main method called **buildCSRGraph**, which performs the actual graph construction. This method internally invokes the appropriate selector class (in our case, the `BuildGraphDeterministicHashSelector`) to build the CSR graph based on the chosen builder implementation. The resulting CSR graph is encapsulated within a `ResultFromBuilder` object, which holds various graph properties and statistics.

This flexible design allows for easy extensibility and customization by supporting multiple builder implementations and dynamically selecting the corresponding CSR graph implementation based on the chosen builder type.

4.4 Structure of the ResultFromBuilder class

The `ResultFromBuilder` class encapsulates the result of the graph-building process and provides access to the constructed CSR graph and other relevant information. This class serves as a container for the output of the builder module.

The `ResultFromBuilder` class includes the following properties:

1. *CSRGraph* csrGraph*: This property stores the constructed CSR graph object. It represents the compressed sparse row representation of the graph and provides efficient access to the graph structure and properties.

2. *double build_graph_time*: This property records the time taken for the graph-building process, providing insights into the efficiency of the builder implementation.

3. *int total_pe*: This property denotes the total number of processing elements (PEs) involved in the graph-building process.

4. *int SCALE*: This property represents the scale parameter of the generated graph.

5. *int edgefactor*: This property represents the edge factor of the generated graph.

6. *int64_t local_edges*: This property stores the number of local edges in the constructed CSR graph.

7. *int64_t global_vertices*: This property represents the total number of vertices in the global graph.

8. *int64_t global_edges*: This property represents the total number of edges in the global graph.

9. *ShufflingEfficiencyStats shufflingEfficiencyStats*: This property provides statistical information about the shuffling efficiency during the graph-building process, including the lowest and highest load on PEs, average load, variance, and highest deviation from the average.

The `ResultFromBuilder` class also includes utility methods, such as **printShufflingEfficiencyStats()**, which allows printing the shuffling efficiency statistics. Additionally, the `ResultFromBuilder` class provides convenient access to the constructed CSR graph by exposing three virtual functions: *int64_t* **local_to_vertex**(*int64_t pe*, *int64_t localv*), *int64_t* **vertex_to_local**(*int64_t v*), and *int64_t* **vertex_owner**(*int64_t v*). By using the ResultFromBuilder class, users can easily retrieve the constructed CSR graph, analyze the graph properties, and perform subsequent graph processing operations.

4.5 Usage Instructions

To utilize the Builder in AGL, follow these steps outlined below:

1. Instantiate an object of the BuilderBase class:

```
BuilderBase builder(generatorResult, Cyclic);
```

Here, generatorResult is an instance of the ResultFromGenerator class obtained from the Graph Generator module, and Cyclic is the desired builder type.

2. Call the method buildCSRGraph() on the builder object to build the CSR graph:

```
ResultFromBuilder builderResult = builder.buildCSRGraph();
```

This method executes the graph building process using the specified builder implementation and returns a ResultFromBuilder object.

3. Utilize the properties and methods provided by the ResultFromBuilder class to access information about the graph construction process:

```
fprintf("time:%f", builderResult.build_graph_time);
builderResult.printShufflingEfficiencyStats();
```

In the example above, the construction time of the graph is accessed through the *build_graph_time* property of the `ResultFromBuilder` object. Additionally, the **printShufflingEfficiencyStats()** method can be used to display shuffling efficiency statistics.

4.6 Extending Builder with a Custom Implementation

The Builder in AGL allows for the addition of custom implementations to cater to specific graph construction needs by extending the `CSRGraph` class and implementing

the necessary functions. To add a custom builder implementation, follow these steps:

1. Add a new type to the GraphBuilderTypeEnum enum

```
enum GraphBuilderTypeEnum {
    Cyclic,
    ...
    CustomBuilder, // Add your custom builder type here
};
```

2. Create a new class that extends the `CSRGraph` class. Implement the virtual functions **local_to_vertex()**, **vertex_to_local()**, and **vertex_owner()** within your custom class:

```
class CustomBuilderCSRGraph : public CSRGraph {
    // Implement necessary functions and add custom functionality
    int64_t local_to_vertex(int64_t pe, int64_t localv) override {
        // Custom implementation for local_to_vertex function
    }
    int64_t vertex_to_local(int64_t v) override {
        // Custom implementation for vertex_to_local function
    }
    int64_t vertex_owner(int64_t v) override {
        // Custom implementation for vertex_owner function
    }
```

```
};
```

3. Update the BuilderBase class constructor to instantiate the appropriate CSRGraph object based on the builder type:

```
BuilderBase(ResultFromGenerator generatorResult, GraphBuilderTypeEnum t
    //Instantiate your custom CSRGraph object
    if (type == Cyclic)
        builderResult.csrGraph = new CustomBuilderGraph(generatorResult
    else
        builderResult.csrGraph = new CSRGraph(generatorResult);
}
```

4. If your custom builder implementation requires a new HClib Actor class, update the **buildCSRGraph()** method in the `BuilderBase` class to call the appropriate HClib Actor class based on the builder type.

With these updates, you can include custom implementations in the Builder module by extending the available builder types, instantiating the correct `CSRGraph` object, and incorporating any necessary HClib Actor classes. This allows you to tailor the graph-building process to your specific needs and leverage the flexibility of the Builder module in AGL.

Chapter 5

Evaluation

In this section, we evaluate the performance of different implementations within the AGL (Asynchronous Graph Library) framework. Through these evaluations, we gain insights into the strengths and performance characteristics of different implementations in various scenarios.

To ensure a comprehensive assessment, we conduct both strong scaling and weak scaling experiments for our evaluation. We perform our experiments on different clusters ranging from 1 Node (16 Cores) to 32 Nodes (512 Cores).

Strong scaling experiments involve fixing the problem size while increasing the number of computational resources, such as nodes (cores). This setup allows us to observe how well the system scales and utilizes additional resources. By measuring the performance metrics under strong scaling, we can analyze the efficiency of the components and identify any potential bottlenecks or limitations. For the strong scaling experiments, we use a graph scale of 24, i.e., 2^{24} vertex graph.

Weak scaling experiments involve maintaining a constant workload per computational resource while increasing both the problem size and the number of resources. This type of scaling evaluates the system's ability to handle larger problem sizes as the resources scale proportionally. By measuring the performance metrics under weak scaling, we can determine if the system can effectively maintain a consistent level of performance as the workload increases. For the weak scaling experiments, we used a graph scale ranging from 20 to 25.

In the following subsections, we delve into the details of our evaluations, focusing on comparing graph generators (5.1) and graph builders (5.2). We examine various factors such as load balance, performance, and BFS (Breadth-First Search) performance to understand how different implementations perform.

5.1 Evaluation of Graph Generators

In this section, we evaluate the performance of different graph generators. Our evaluation includes comparisons among AGL generators as well as a comparison of AGL generators with external baselines.

5.1.1 Comparison Among AGL Generators

In this subsection, we focus on comparing different generators within AGL, specifically the RMAT generator and the Uniform Random generator. These experiments only time generating the distributed edge list, which will later need to be passed to the builder to become a graph.

For strong scaling experiments (Figure 5.1), the generation time for the RMAT and the Uniform Random generators decreases as the number of cores increases (expected). However, when comparing the two AGL generators, we observe that the RMAT generator generally requires more time than the Uniform Random generator. This difference can be attributed to the inherent complexity of generating RMAT graphs compared to uniform random graphs.

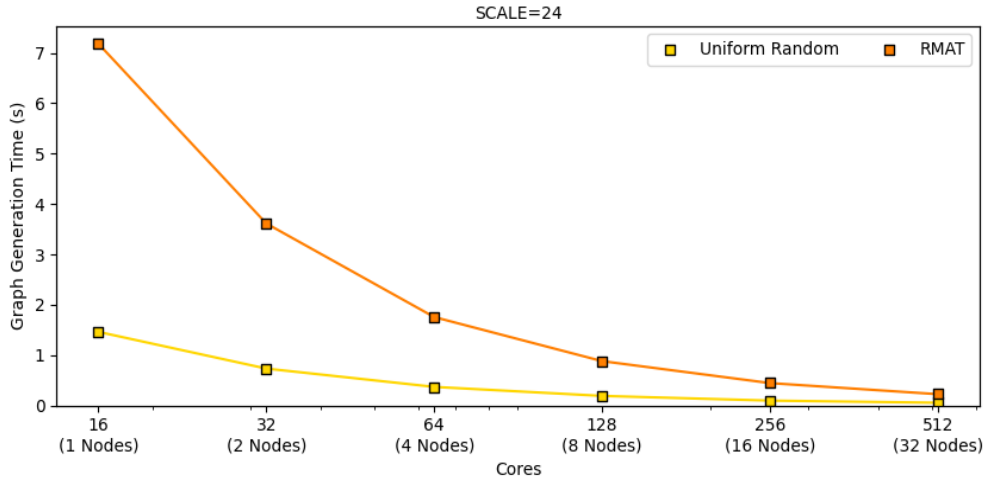

Figure 5.1: Strong Scaling of Graph Generation Time (AGL Generators) with SCALE=24

Moving on to the weak scaling experiments (Figure 5.2), we expect relatively constant generation times for both AGL generators since the workload per core remains constant. While this expectation holds true, we observe a slight increase in generation time as the number of cores increases for both the RMAT generator and the Uniform Random generator.

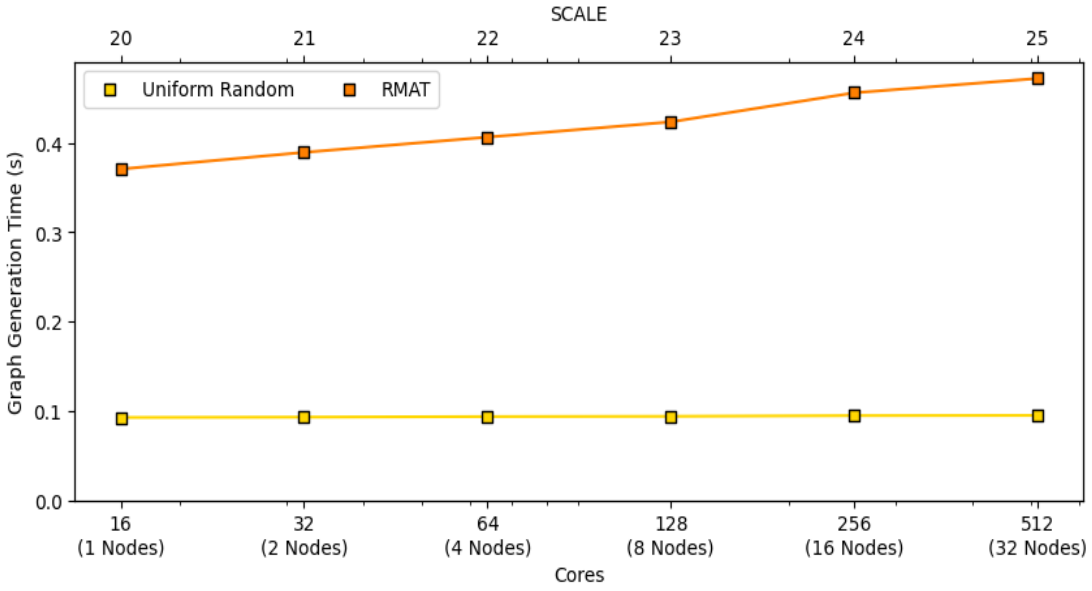

Figure 5.2: Weak Scaling of Graph Generation Time (AGL Generators)

Reasoning: The observe increase in execution time for larger scales may be influenced by factors such as potential inefficiencies in the coordination and synchronization of AGL actors on larger scales. While the generator itself doesn't involve any communication overhead, the increases scale might introduce coordination challenges, leading to slightly longer execution times. Further analysis and optimization of the coordination mechanisms within the AGL graph generator could potentially address these inefficiencies and improve the overall performance.

5.1.2 Comparison of AGL to External Alternatives

In this subsection, we compare the AGL generators to an alternative, specifically the Graph500 reference code's RMAT generator [6].

During the strong scaling experiments (Figure 5.3), we measure the generation time of AGL's RMAT generator and Uniform generator, and the Graph500 reference code's RMAT generator. As expected, the generation time for all generators decreases as the number of cores increases. However, the Graph500 reference code's RMAT generator is the slowest. Thus AGL generators offer better performance for the graph generation.

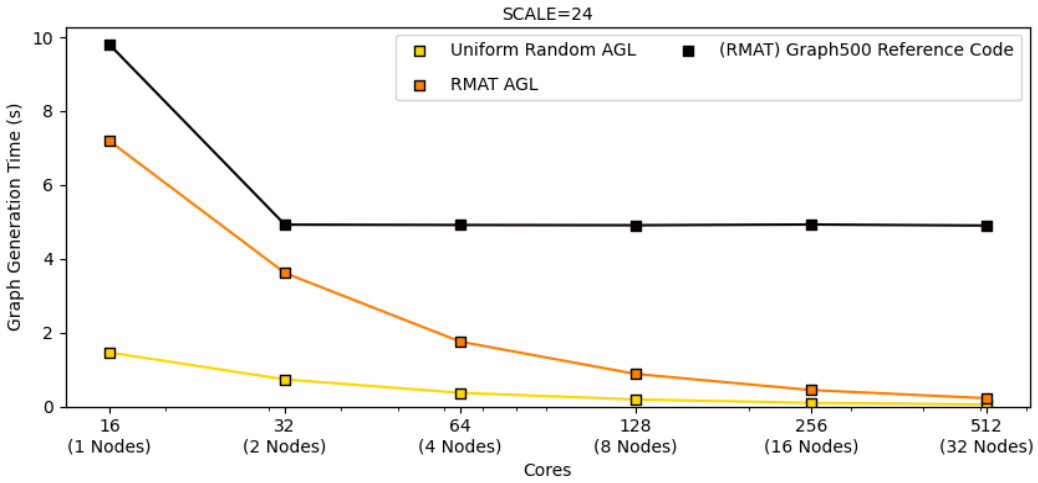

Figure 5.3: Strong Scaling of Graph Generation Time (AGL Generators VS Graph500 Reference Code RMAT Generator) with SCALE=24

Furthermore, in the weak scaling experiments (Figure 5.4), where the workload per core remains constant, we observe relatively constant generation times for both the AGL generators and the Graph500 Reference Code RMAT generator. While all generators experience slight increases in generation time as the number of cores increases, the rate of increase was higher for the Graph500 generator compared to the AGL generators, i.e. the AGL generators are the fastest.

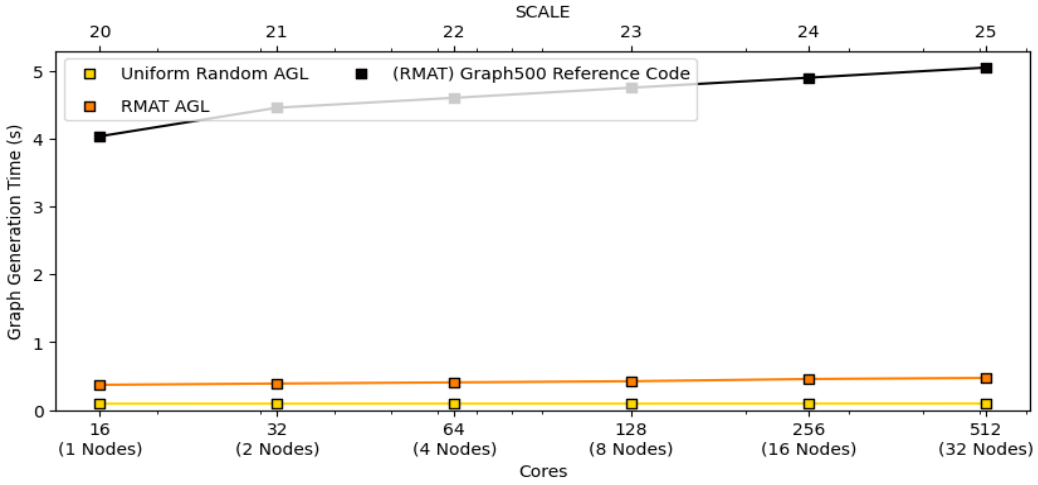

Figure 5.4: Weak Scaling of Graph Generation Time (AGL Generators VS Reference Code RMAT Generator)

Reasoning: A possible reason for the performance differences observed between the AGL's and the Graph500 reference code's RMAT generator is the involvement of communication. While the AGL generators generate a local set of edges on each processing element (PE) without requiring explicit communication, the Graph500 RMAT code involves MPI Cart communications. These communication operations can introduce additional overhead and contribute to longer generation times in the Graph500 generator.

5.2 Evaluation of Graph Builders

In this section, we evaluate the performance of different graph builders within the AGL framework. Our evaluation includes comparisons among AGL builders as well as a comparison of AGL builders with an external alternative.

5.2.1 Comparison Among AGL Builders

In this subsection, we compare the performance of different builders within the AGL framework. For each builder, we analyze the build time, load imbalance, and BFS time using the built graph.

1. **Load Imbalance**:

   Surprisingly, the load imbalance results for both weak (Figure 5.5) and strong (Figure 5.6) scaling experiments are nearly the same. This indicates that load imbalance is not dependent on the graph scale but rather on the number of cores and the chosen algorithm. As the number of cores increases, the load imbalance also tends to increase. Among the builders, we observe that the Range builder has the highest load imbalance, followed by Cyclic, SnakeHash, RotationHash, and SnakeRotation. SnakeRotation and RotationHash builders demonstrate similar performance, while the Snake builder performs worse. This suggests that the flipping operation in SnakeRotation may not be necessary, and that the rotation operation is the key factor for achieving efficient load balancing.

   The load balancing equation used in this study is defined as:

$$load\_imbalance = \frac{\text{maximum deviation of work assigned to one PE from average}}{\text{average work assigned to PEs}}$$

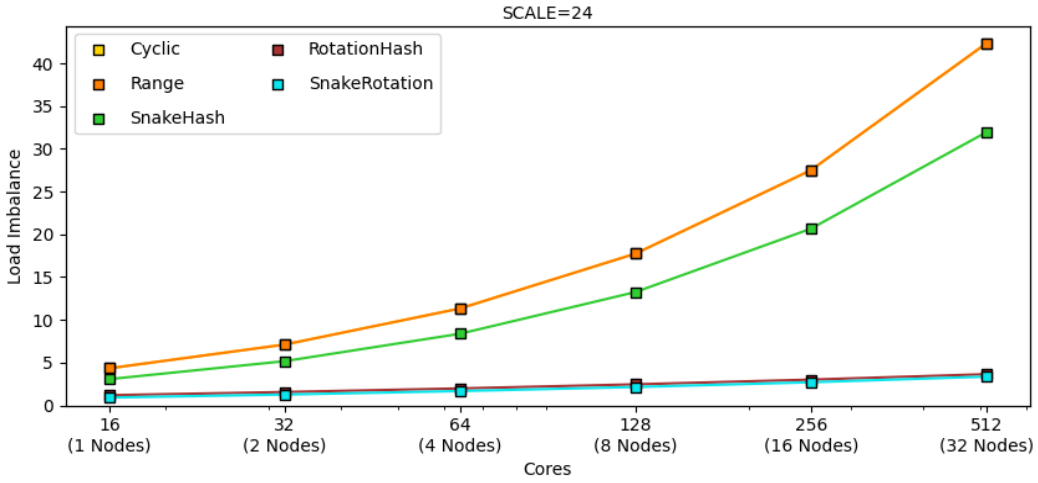

Figure 5.5: Strong Scaling of Graph Load Imbalance on Graphs built by AGL builders. SCALE used for evaluation = 24

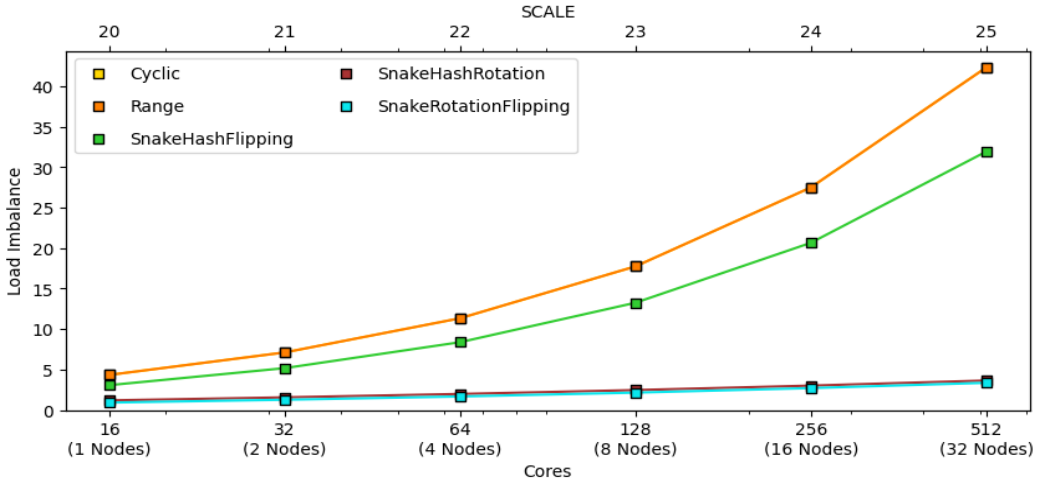

Figure 5.6: Weak Scaling of Graph Load Imbalance on Graphs built by AGL builders

Reasoning: The RotationHash and SnakeRotation builders demonstrated better load balancing compared to the other implementations. This can be attributed to the additional randomization introduced by the snake functions, which helps

balance the workload by distributing the sum of degrees more evenly across the processing elements. Furthermore, the superior load balancing performance of the Cyclic builder compared to the Range builder can be attributed to the utilization of bitshift operations, which aid in achieving a more balanced vertex-to-processing element allocation.

2. **Build Time**: For strong scaling (Figure 5.7), we observe that the build time reduces as the number of cores increases. Among the AGL builders, we find that the Range builder has the highest build time, followed by Cyclic, SnakeHash, RotationHash, and SnakeRotation. Notably, using bit shift operators in the Cyclic builder makes it more efficient. However, we believe that the SnakeHash builders could also perform even better by replacing the modulo and divide operators with bitshift operations.

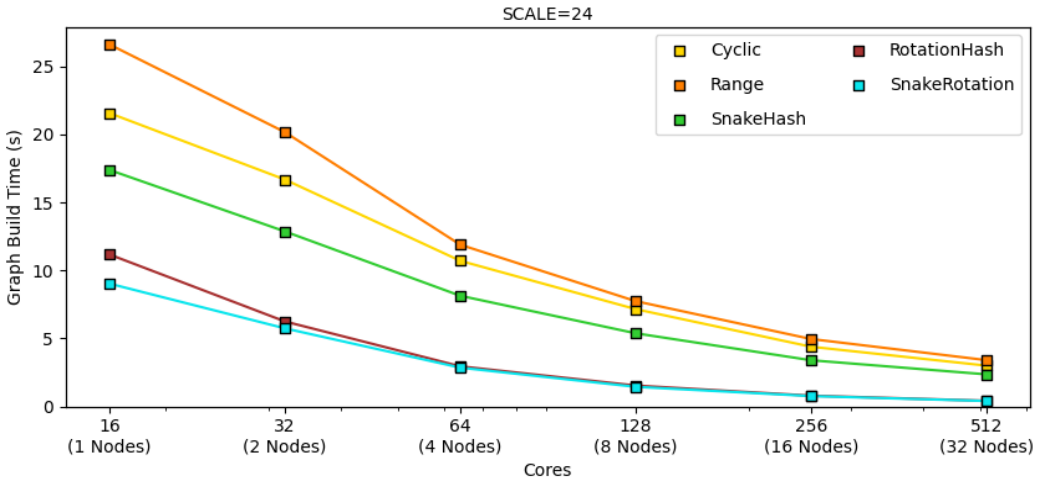

Figure 5.7: Strong Scaling of Graph Build Time on Graphs built by AGL builders. SCALE used for evaluation = 24

In the case of weak scaling (Figure 5.8), we observe that the build time increases as the scale and number of cores increase. Similar to the strong scaling results, the Range builder has the highest build time, followed by Cyclic, SnakeHash, RotationHash, and SnakeRotation. We also notice that the rate of increase in build time is the highest for the Range builder and the lowest for the SnakeRotation builder. Thus, the SnakeRotation builder performs better as the problem size increases.

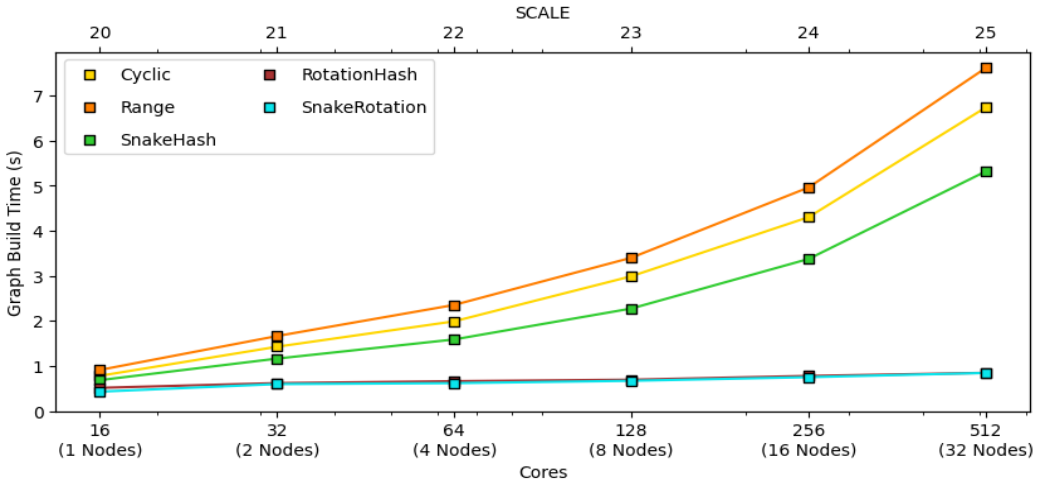

Figure 5.8: Weak Scaling of Graph Build Time on Graphs built by AGL builders

Reasoning: These findings highlight the importance of load balancing in graph processing tasks, particularly in relation to build time. Builders that achieve better load balance optimize edge locality, minimizing communication overhead. By reducing the need for extensive communication, these builders can achieve more efficient build times.

3. **BFS Time**:

   For strong scaling (Figure 5.9), we observe that the BFS time reduces as the number of cores increases, similar to the build time analysis. The difference in BFS time between the builders also decreases as the number of cores increases. Among the AGL builders, the Range builder has the slowest build time, followed by Cyclic, SnakeHash, RotationHash, and SnakeRotation.

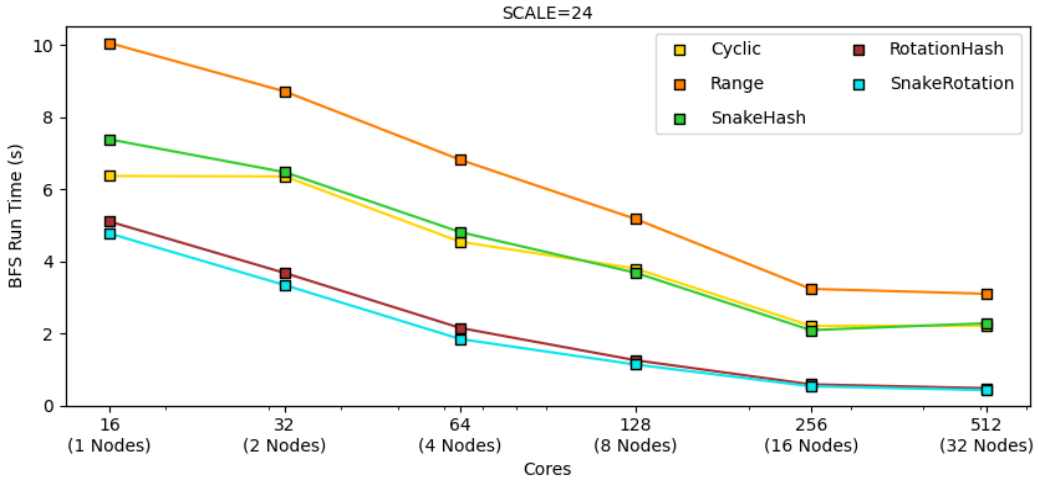

Figure 5.9: Strong Scaling of Graph BFS Time (Top-Down BFS) on Graphs built by AGL builders. SCALE used for evaluation = 24

   In terms of weak scaling (Figure 5.10), as the scale and number of cores increase, the BFS time also increases. The rate of increase in BFS time was as follows: $Range > Cyclic > SnakeHash > RotationHash > SnakeRotation$. Once again, the SnakeRotation builder performs better for all scales, especially as the problem size increases. Note: Rotation seems to be the key aspect for optimized load balancing and better BFS Times.

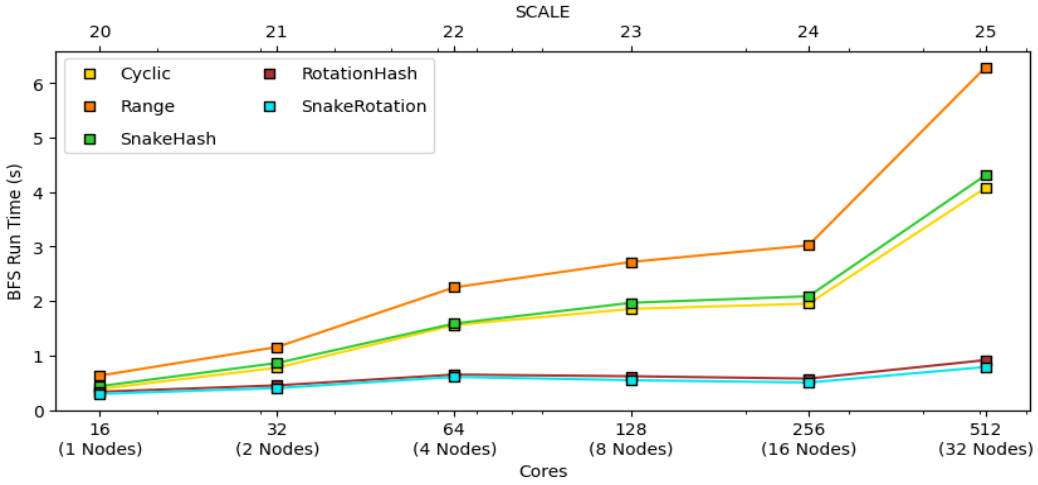

Figure 5.10: Weak Scaling of Graph BFS Time (Top-Down BFS) on Graphs built by AGL builders)

Reasoning: These results emphasize the impact of load imbalance on build time in graph processing tasks. The builders that achieved better load balancing, such as SnakeRotation, demonstrated improved performance in terms of build time. This aligns with the notion that load imbalance can adversely affect the efficiency of graph processing algorithms, and addressing load imbalance can lead to significant improvements in runtime performance.

In summary, the Range builder consistently has the slowest build time, but the RotationHash and SnakeRotation builders demonstrate better load balance. When it comes to BFS, the SnakeRotation builder outperforms the others. These results suggest that better load balancing contributes to the superior performance of the RotationHash and SnakeRotation builders.

5.2.2 Comparison of AGL Builders to External Alternatives

In this subsection, we compare the performance of the AGL builders to a non-AGL alternative, specifically the Graph500 reference code's builder, which is Cyclic. We focus on the build time and observe the differences between the two approaches, i.e., AGL Builders vs. the Graph500 reference code Cyclic Builder.

For strong scaling (Figure 5.11), we find that the Graph500 reference code Cyclic builder performs better than all AGL builders. However, it is worth noting that as the number of cores increases, the performance of the AGL builders improves faster than that of Graph500 reference code Cyclic. Notably, the SnakeHash builders outperform Graph500 reference code Cyclic for 16 and 32 nodes used in the experiments.

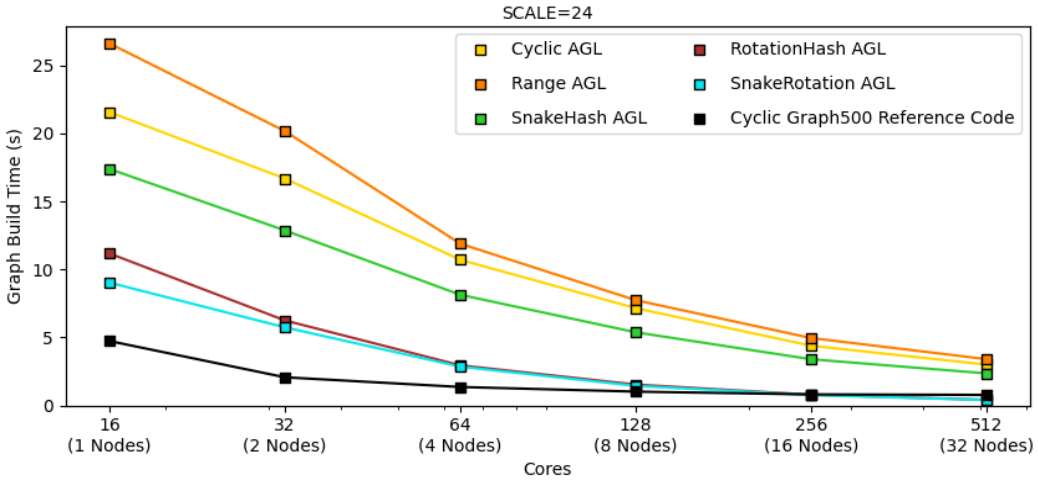

Figure 5.11: Strong Scaling of Graph Build Time on Graphs built by AGL builders VS Reference code Cyclic Builder. SCALE used for evaluation = 24

For weak scaling (Figure 5.12), we observe that Graph500 reference code Cyclic performs better than most AGL builders, except for the RotationHash and SnakeRota-

tion builders. Overall, these results indicate that the SnakeHash builders show promising performance. If we can improve the AGL Cyclic builder to perform equally or better than Graph500 Cyclic, the SnakeHash builders could significantly outperform the reference implementation. The effectiveness of snake hashing can be attributed to the added randomness in the shuffling process.

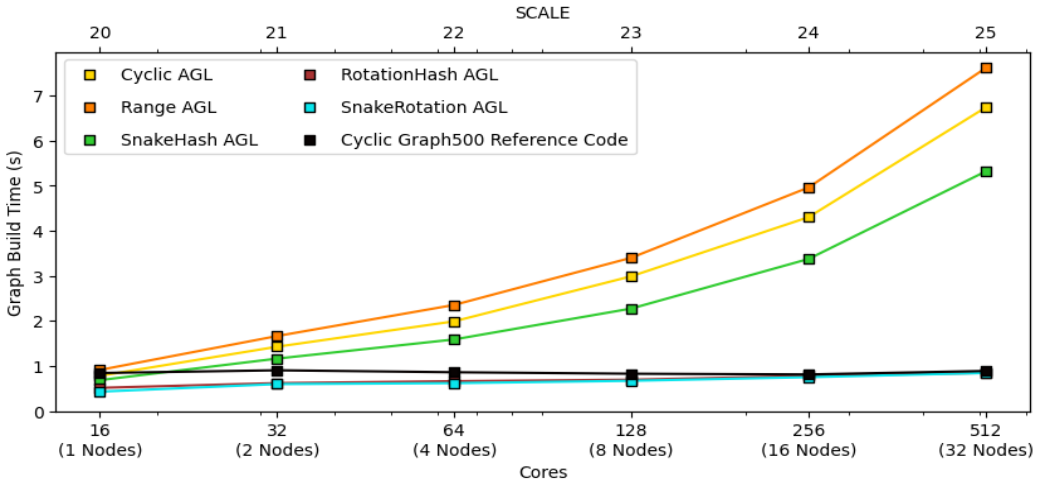

Figure 5.12: Weak Scaling of Graph Build Time on Graphs built by AGL builders VS Reference code Cyclic Builder.

Reasoning: Overall, the slower performance of AGL builders compared to Graph500 reference code Cyclic can be attributed to potential factors such as the effectiveness of the graph generator used in Graph500, which might employ random shuffling and relabeling techniques to enhance performance. Additionally, the HClib Actor library, while providing powerful actor-based programming capabilities, may introduce overhead in communication-heavy operations. Further investigation and testing are required to analyze and optimize the AGL builders for improved build time performance.

5.3 Challenges Faced

During our evaluation of the Actor Graph Library (AGL) and the underlying HClib Actor library, we encountered certain challenges that are worth highlighting. These challenges shed light on areas that require further investigation and debugging to improve the performance and usability of AGL.

One notable challenge we faced was out-of-memory errors encountered by AGL builders in specific scenarios. The team working on the HClib Actor library informed us that the out-of-memory issue occurs because memory is not being appropriately deallocated in the send calls. Consequently, an increases number of send calls results in a higher occurrence of dangling memory, leading to out-of-memory errors. Although efforts have been made to address this issue in the HClib Actor library, AGL builders still encounter this challenge.

To mitigate this issue, we used smaller-scale workloads in our evaluation, such as scale 24 for strong scaling and scale 20-25 for weak scaling. This was necessary as higher-scale workloads on certain configurations encountered out-of-memory issues faster. Limiting the scale allowed us to focus on evaluating the performance and behavior of AGL within feasible memory limits. Furthermore, we did not extend our evaluation beyond 32 nodes or 512 cores due to the increases communication and send calls associated with a larger number of cores. The higher number of send calls created a strain on the underlying HClib Actor library, leading to out-of-memory errors. This limitation highlights the need for further optimization and memory management

strategies within HClib Actor to handle larger core counts efficiently.

Addressing these challenges requires a thorough understanding of the memory deallocation processes within the HClib Actor library. By investigating the causes of out-of-memory errors, profiling the memory usage, and optimizing the communication patterns, future enhancements can be made to overcome these limitations and improve the reliability and scalability of AGL.

Chapter 6

Conclusion

The evaluation results highlight the strengths and potential of AGL for graph generation and traversal. However, it is important to acknowledge that AGL's performance in terms of build time falls short when compared to the Graph500 reference code builder. Further investigation and profiling are needed to identify the exact reasons behind this performance difference.

During our evaluation, we encountered certain challenges related to the usage of AGL and the underlying HClib Actor library. Specifically, AGL builders encountered out-of-memory errors in two scenarios: when dealing with high-scale graphs on fewer machines and when utilizing many cores. These out-of-memory issues are likely related to the high number of send calls, which need further investigation and debugging to pinpoint the exact causes.

In conclusion, the Actor Graph Library (AGL) project has successfully developed a robust and scalable graph processing library based on the HClib Actor frame-

work. AGL offers a user-friendly interface and efficient execution for graph analytics tasks, simplifying the development of graph-based applications and leveraging the scalability advantages of distributed computing. Through our evaluation of AGL's graph generators and builders, we gained valuable insights into their performance characteristics and compared them with the Graph500 reference code.

Chapter 7

Future Work

While the Actor Graph Library (AGL) demonstrates promising capabilities and performance in distributed graph processing, there are several areas for future exploration and improvement. The following directions can be pursued to enhance the functionality and effectiveness of AGL:

1. **Extending AGL with More Graph Generators and Builders**: Currently, AGL includes graph generators and builders based on hash-based partitioning and equal balancing of vertices. In the future, additional graph generators and builders can be incorporated to provide more flexibility and options for representing and manipulating graph data. This includes builders that balance edges instead of vertices, non-hash-based builders, and builders that allow for assigning larger vertices to multiple nodes, enabling more complex graph processing scenarios.

2. **Further Study and Benchmarking of BFS Performance**: AGL has provided valuable insights into the performance characteristics of different builders in the

context of breadth-first search (BFS). Future work can focus on expanding the range of builders available in AGL and conducting comprehensive benchmarking to identify the most suitable builder for BFS. This investigation will deepen our understanding of how different builder strategies impact BFS performance and guide the development of optimized graph processing solutions.

3. **Debugging Out of Memory Issues and Optimizing Build Times**: The out of memory issues encountered during the evaluation of AGL builders highlight the need for further debugging and optimization efforts. By thoroughly investigating the underlying causes of these issues and profiling the memory usage, the AGL library can be fine-tuned to handle larger-scale graphs and a higher number of cores without encountering memory limitations. Additionally, efforts can be made to identify and address any performance bottlenecks that contribute to longer build times, enhancing the overall efficiency of the library.

4. **Comparison of AGL Performance against Other Distributed Graph Processing Libraries**: To gain a comprehensive understanding of AGL's performance and capabilities, it would be valuable to compare it against other distributed graph processing libraries. This comparative analysis can shed light on the strengths and weaknesses of AGL in different scenarios and provide insights into areas for further improvement. By benchmarking AGL against established graph processing frameworks, researchers and practitioners can make informed decisions about the most suitable tool for their specific graph analytics tasks.

These future directions will contribute to the continued development and refinement of AGL as a powerful and versatile graph-processing library. By addressing the identified limitations, expanding the functionality, and conducting thorough benchmarking, AGL can further establish itself as a valuable tool in the field of distributed graph processing.

Bibliography

[1] Gul Agha. Actors: A model of concurrent computation in distributed systems. 10 2004.

[2] Gul Agha. Actors programming for the mobile cloud. In *2014 IEEE 13th International Symposium on Parallel and Distributed Computing*, pages 3–9, 2014.

[3] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[4] Aydın Buluç and John R Gilbert. The combinatorial blas: Design, implementation, and applications. *Int. J. High Perform. Comput. Appl.*, 25(4):496–509, nov 2011.

[5] William Carlson, Jesse Draper, David Culler, Katherine Yelick, Eugene Brooks, Karen Warren, and Lawrence Livermore. Introduction to upc and language specification. 04 1999.

[6] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, 2004.

[7] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn,

Chuck Koelbel, and Lauren Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, New York, NY, USA, 2010. Association for Computing Machinery.

[8] P Erdös and A Rényi. On random graphs i. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

[9] Max Grossman, Vivek Kumar, Nick Vrvilo, Zoran Budimlic, and Vivek Sarkar. A pluggable framework for composable hpc scheduling libraries. pages 723–732, 2017.

[10] Shams M. Imam and Vivek Sarkar. Selectors: Actors with multiple guarded mailboxes. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents amp; Decentralized Control*, AGERE! '14, page 1–14, New York, NY, USA, 2014. Association for Computing Machinery.

[11] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, USA, 2018.

[12] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition*. Artima Incorporation, Sunnyvale, CA, USA, 2nd edition, 2011.

[13] Sri Raj Paul, Akihiro Hayashi, Kun Chen, and Vivek Sarkar. A productive andnbsp;scalable actor-based programming system fornbsp;pgas applications. In *Computational Science – ICCS 2022: 22nd International Conference, London, UK,*

June 21–23, 2022, Proceedings, Part I, page 233–247, Berlin, Heidelberg, 2022. Springer-Verlag.

[14] Sri Raj Paul, Akihiro Hayashi, Kun Chen, and Vivek Sarkar. A scalable actor-based programming system for pgas runtimes, 2022.

[15] Mhd Saeed Sharif, Maysam Abbod, and Abbes Amira. Neuro-fuzzy based approach for analysing 3d pet volume. In *2011 Developments in E-systems Engineering*, pages 158–163, 2011.

[16] Olivier Tardieu, Benjamin Herta, David Cunningham, David Grove, Prabhanjan Kambadur, Vijay Saraswat, Avraham Shinnar, Mikio Takeuchi, and Mandana Vaziri. X10 and apgas at petascale. *SIGPLAN Not.*, 49(8):53–66, feb 2014.

[17] Olivier Tardieu, Benjamin Herta, David Cunningham, David Grove, Prabhanjan Kambadur, Vijay Saraswat, Avraham Shinnar, Mikio Takeuchi, and Mandana Vaziri. X10 and apgas at petascale. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, page 53–66, New York, NY, USA, 2014. Association for Computing Machinery.

[18] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 International Workshop on Parallel Symbolic*

Computation, PASCO '07, page 24–32, New York, NY, USA, 2007. Association for Computing Machinery.