ANALYZING COMMUNICATION IMPACTS OF GRAPH-REFINEMENT LOAD-BALANCING IN STENCIL3D

BY

SOWMYA YELLAPRAGADA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

Adviser:

Professor Emeritus Laxmikant V Kale

ABSTRACT

Load balancing is a critical aspect of parallel and distributed computing systems, especially in environments characterized by dynamic workloads and high communication volumes. In this thesis, we delve into various load-balancing strategies for Charm++ applications, with a specific focus on communication-aware methods. Our goal is to maximize internal communication and minimize external communication across the nodes. We analyze two primary load-balancing algorithms - GreedyRefineLB and GraphRefineLB. Graph-based partitioning strategies aim to optimize load distribution by considering the communication topology. These algorithms leverage graph partitioning techniques to achieve better load balance. However, they also face challenges related to scalability and adaptability to changing workloads.

Our major contribution lies in analyzing the communication impacts of a new algorithm: Communication-aware Diffusion LB. This innovative approach seeks to strike a balance between two critical objectives - reducing object migrations and enhancing communication locality. To assess the efficacy of these strategies, we employ the Stencil3d benchmark—a well-established synthetic standard for evaluating parallel and distributed systems. Our evaluation reveals the trade-offs associated with different load-balancing approaches. Notably, Communication-aware Diffusion LB demonstrates superior performance in managing dynamic workloads with substantial communication requirements.

In summary, our research underscores the importance of communication-aware load balancing in parallel computing environments. As applications continue to evolve, finding the right balance between communication optimization and efficient system utilization remains crucial. The Communication-aware Diffusion LB offers a promising avenue for achieving this balance and enhancing the overall performance of Charm++ applications.

To my parents, sister, family, and friends, for their love and support.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to the people who supported me throughout my journey to completing this thesis.

Firstly, I am deeply grateful to Professor Kale, my adviser, for his invaluable guidance and support. He has been instrumental in my learning and development, and I appreciate his willingness to explain even the most fundamental concepts patiently and without judgment. I would also like to thank Kavitha for her generosity in allowing me to experiment with her simulator, which formed the foundation of my thesis research. Her support has been crucial to its success. I extend my gratitude to my fellow PPL labmates, especially Maya, for always being willing to lend a helping hand whenever needed.

To my amma, nanna, grandparents, and mamayya, I extend my heartfelt thanks for their unwavering encouragement during challenging times. Their belief in me has been a source of strength throughout my academic pursuits.

A special thank you goes to my little big sister, Sushma, who flew across the country to support me on this important occasion. Your presence truly means the world to me.

I am grateful to my roommate, Ananthi, for her delicious meals and all the fun. I also want to thank Raj, my dearest friend, for being a pillar of support. His words of encouragement and consolation during moments of stress were invaluable.

Finally, thanks to Isha for the enjoyable Friday nights and dinners, to Aravind for sharing his spiritual wisdom, and to my many other cherished friends, though I cannot name you all here, your friendship has enriched my life in countless ways.

To all of you, thank you for your contributions to this thesis.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

The ever-increasing complexity of scientific and engineering problems demands computing power beyond the capabilities of single-processor machines. Parallel programming offers a solution by harnessing the combined power of multiple processors to tackle these challenges. However, effectively writing parallel applications requires a programming model that simplifies the complexities of managing concurrency, communication, and data locality. This introduction delves into Charm++, a powerful parallel programming model designed to facilitate the development of high-performance parallel applications.

1.1 INTRODUCTION TO CHARM++

Charm++ is not a standalone programming language; rather, it extends the capabilities of C++ to enable the creation of parallel applications [1, 2]. It builds upon the familiar syntax and features of C++ while introducing additional constructs and functionalities specifically aimed at parallel programming. This approach allows programmers with existing C++ expertise to quickly learn and leverage Charm++ for developing parallel applications.

1.2 THE CHARM++ PROGRAMMING MODEL

A fundamental concept in Charm++ is the Processing Element (PE) [3]. A PE essentially represents a unit of computation within the system. These PEs can either be operating system threads or processes, depending on how Charm++ is configured. Each PE has its own private scheduler that manages a pool of messages. Throughout the rest of this document, we will refer to Processing Elements simply as PEs.

Programming in Charm++ revolves around lightweight, migratable objects called "chares" which is a fundamental unit of parallelism. These chares encapsulate both data and code that operates on that data. The Charm++ runtime system manages the creation, execution, and communication of chares across a network of processors. A brief introduction of various features of Charm++ are listed below:

1.2.1 CHARES: BUILDING BLOCKS OF CHARM++ PROGRAMS

A chare in Charm++ is essentially a parallel object that encapsulates data and a set of methods [3]. The key components of a chare (as depicted in Figure 1.1, taken from the

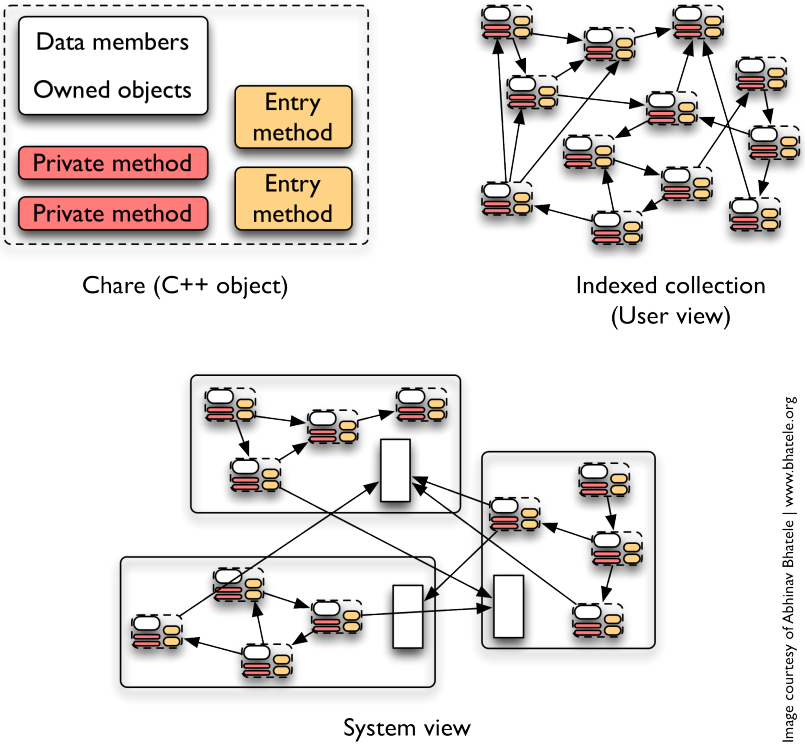


Figure 1.1: User View and System View in Charm++

Charm++ website [4]) are:

- Data Members: These variables store the private data specific to a chare.
- Owned Objects: These are instances of other classes that are managed by the chare.
- Entry Methods: These are special methods designed to be invoked remotely by other chares. They facilitate communication and collaboration between chares. Entry methods are the mechanism by which chares communicate with each other asynchronously.
- Private Methods: These are local methods that can only be called by other methods within the same chare. They handle internal computations specific to the chare.

1.2.2 INDEXED COLLECTIONS: ORGANIZING CHARES

Charm++ provides a way to organize chares into collections, ensuring all members have the same type. These collections are indexed using chare arrays and messages may be sent to individual chares within a chare array, meaning each chare within the collection has a unique identifier [4]. This indexing scheme is crucial for communication between chares. A chare can send messages to specific chares within a collection using their indices.

1.2.3 KEY ASPECTS OF CHARM++

- Object-Oriented Design: Charm++ leverages familiar object-oriented concepts to structure the program using chares. This makes it easier to reason about parallel programs and potentially reuse existing code.

- Globally Addressable Objects: Certain classes can be designated as globally visible, allowing any chare to interact with them. This is useful for coordinating communication or sharing data across the entire parallel program.

- Asynchronous Methods: Communication between chares happens through asynchronous method invocations. This means the invoking chare doesn't wait for the receiving chare's method to finish before continuing its execution. This improves the overall efficiency and scalability of parallel programs, as chares are not blocked while waiting for responses.

- Migratable Objects: Chares can be migrated between processors for better load balancing and performance optimization. This allows the Charm++ runtime system to dynamically distribute the workload across available processing cores or nodes in a parallel computing environment.

In summary, the Charm++ programming model centers around chares as parallel objects and indexed collections for organizing them. This structure, along with features like asynchronous communication and object migration, empowers us to write efficient parallel applications in C++.

1.3 OBJECT MIGRATION IN CHARM++

Object migration allows chares to be dynamically moved between processors during program execution [5]. This capability offers several benefits:

- Load Balancing: Charm++ can automatically migrate chares to processors with lower workloads, ensuring a more even distribution of computation across the available resources. This leads to improved application performance. The Charm++ runtime system uses various techniques to monitor workload and determine optimal migration targets. These techniques can be user-defined or rely on built-in algorithms within the runtime system, some of which will be discussed in Chapter 2.

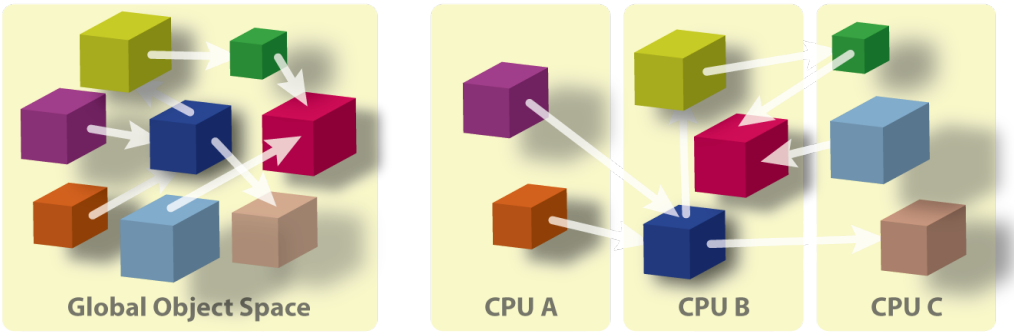


Figure 1.2: Application decomposed into Charm++ objects and partitioned across nodes

- Data Co-locality: Chares can be migrated to processors that hold the data they require most frequently (Figure 1.2, taken from the PPL website [3]). This reduces communication overhead and improves performance by bringing computation closer to the data. Charm++ provides mechanisms for programmers to specify data access patterns and guide the runtime system in placing chares for optimal data locality.

- Fault Tolerance: In case of processor failure, Charm++ can migrate chares away from the failing processor, ensuring application resilience and continued execution. This ability contributes to the robustness of parallel applications developed using Charm++.

However, object migration also introduces a layer of complexity. Programmers need to consider the costs associated with migration. These costs include the overhead of transferring the state of a chare (data and associated code) and potential disruption to ongoing computations on the source and destination processors. Careful design and optimization of the object migration strategy is necessary to ensure that the benefits of migration outweigh the associated costs.

1.4 COMMUNICATION COST OF OBJECT MIGRATION

Excessive object decomposition can lead to significant communication overhead among the PEs [6]. When objects are broken down into smaller components, PEs need to exchange information more frequently to complete tasks. This communication becomes particularly expensive if it occurs across different nodes in the system.

Hence, our primary concern lies in maximizing internal communication while minimizing external communication between nodes. By minimizing inter-node communication, we can significantly reduce overall communication costs.

Several techniques exist to ensure communication primarily happens within a single node, some of the relevant ones are listed below:

- Block Mapping: This method involves grouping objects and assigning them to the same PE. This strategy will be explained in detail in Chapter 4.
- Orthogonal Recursive Bisection (ORB): A non-uniform partitioning strategy which facilitates communication between objects regardless of their location ignoring the asymmetry between internal and external communication costs [7, 8]. However, for performance reasons, local communication within a node is typically preferred.
- Recursive Coordination Bisection (RCB): Similar to ORB, RCB enables communication between objects [9], but its design might prioritize local communication as well.
- Metis Load Balancer (MetisLB): A graph partitioning library used in Charm++ to achieve load balancing across processors. It aims to distribute the workload evenly among processors, minimizing idle time and maximizing computational efficiency [3]. This load-balancing technique helps ensure that communication primarily occurs within a single node by balancing the computational workload across processors. However, Block Mapping can result in more predictable communication patterns and reduced overhead associated with dynamic load balancing algorithms as control over object placement and communication locality is desired.

Note that, block mapping and related strategies that keep objects on the same PE favor low communication costs. This is because communication within a single node is typically much faster than communication across different nodes. On the other hand, strategies like ORB that allow objects to be scattered across nodes can incur higher communication costs, especially if objects need to communicate frequently.

Charm++ offers dynamic load-balancing capabilities. These strategies manage workload distribution among PEs centrally.

As computational tasks become more intricate, the workload naturally changes over time. Earlier load-balancing approaches, like MetisLB and ORB, faced difficulties in such scenarios. Frequent load shifts rendered initial assignments to PEs obsolete, requiring constant re-evaluation and new assignments.

Distributed load-balancing strategies emerged to address the aforementioned issue. One such method, the communication graph, will be discussed in detail in Chapter 3. This approach leverages communication patterns to optimize object placement for efficient communication within a node.

Charm++ offers several advantages over traditional parallel programming models:

- Ease of Use: By leveraging the familiar C++ syntax, Charm++ provides a relatively easy learning curve for programmers already comfortable with C++. This makes it easier for scientists and engineers to develop parallel applications without needing to learn a completely new programming language.

- Portability: Charm++ applications can be executed on a variety of parallel architectures, including clusters, multi-core machines, and cloud platforms. This promotes code reusability across diverse computing environments. The Charm++ runtime system is designed to be adaptable and can be configured to work on different hardware platforms with minimal code modifications.

- Scalability: Charm++ applications can scale efficiently to large numbers of processors, making it a suitable choice for tackling highly parallel problems. The message-passing architecture and object migration capabilities allow applications to maintain good performance even when distributed across a large number of processing units.

- Performance: The combination of asynchronous communication, object migration, and automatic load balancing allows Charm++ applications to achieve high performance on parallel computing systems [3]. Charm++ provides a framework for efficient communication and data movement, which are critical factors for achieving good parallel performance.

This thesis explores communication-aware load balancer and its impact on performance. We begin by establishing a foundation in Chapter 2 with an introduction to load balancers and their various strategies. This understanding is essential for the performance comparison conducted later. Chapter 3 then dives deep into communication-aware load balancers, the core of our study. We'll explore their inner workings and advantages in detail. To evaluate the performance of our chosen load balancer, Chapter 4 introduces Stencil3d, a synthetic benchmark designed for this purpose. In Chapter 5, we will detail the experiments conducted using Stencil3d and analyze the results to gain insights into the load balancer's performance. Finally, Chapter 6 summarizes the key takeaways of the thesis and explores potential areas for future research.

CHAPTER 2: LOAD BALANCERS

Charm++ is a parallel programming language specifically designed for high-performance scientific computing on large-scale parallel systems. One of its key features is automatic load balancing, which helps distribute computational workloads evenly across available processors, leading to improved application performance. This chapter explores four different load-balancing strategies in Charm++: GreedyLB, GreedyRefineLB, MetisLB, and Graph-Refinement LB.

The Charm++ runtime system manages the placement of chares on processors and automatically migrates them during execution to balance the workload. Load-balancing decisions are triggered by synchronization points within the application code. Charm++ provides a framework with a suite of load-balancing strategies which include various centralized, distributed, and hierarchical schemes for balancing computation load or communication [6]:

2.1 GREEDY LOAD BALANCING

Greedy Load Balancing (GreedyLB) is a lightweight load-balancing approach. It prioritizes basic workload distribution by using a greedy algorithm. This algorithm simply assigns the heaviest object to the processor currently handling the least work [10]. It operates in the following steps:

- Load Measurement: The runtime system gathers workload information from each processor. This could involve metrics like CPU utilization, memory usage, or application-specific measurements provided by the programmer.
- Greedy Assignment: Processors are sorted based on their measured load, creating a min-heap of processors. Objects or chares are sorted based on their load, creating a max heap of objects. Chares are then iterated through, and the heaviest chare from the object heap is assigned to the least loaded processor, then processor load is recomputed and it is pushed back into the processor heap. This is repeated till all objects have been assigned to a processor.
- Migration: Chares assigned to different processors are migrated to their new locations.

GreedyLB typically computes a good load balance solution due to its straightforward approach. However, it does not take the initial object-to-PE mapping into consideration and

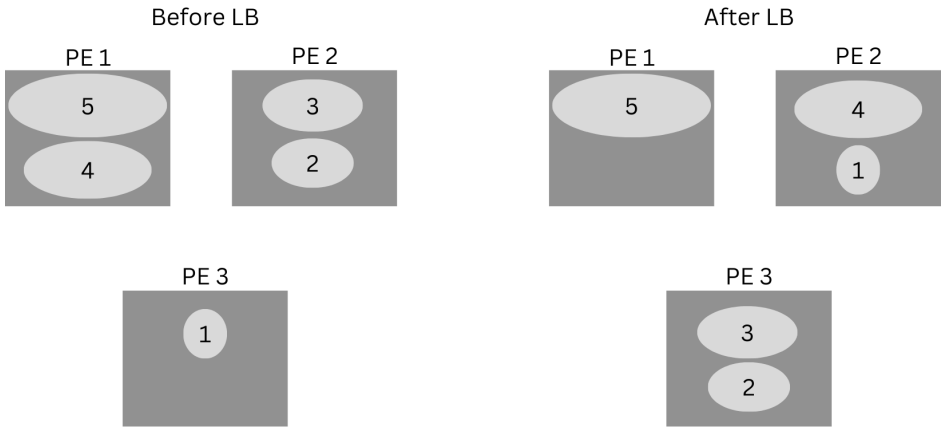


Figure 2.1: This figure exemplifies GreedyLB before and after load balancing in Charm++, where each load (marked with a number indicating its size) is assigned to a PE. Tasks with varying loads are then migrated over the three PEs using a greedy strategy.

can result in a large number of migrations. Additionally, it might not always achieve the optimal distribution, especially for complex application workloads or heterogeneous computing environments.

The task distribution using GreedyLB is illustrated in Figure 2.1 with an example.

Usage: `+balancer GreedyLB`

2.2 GREEDYREFINE LOAD BALANCING

GreedyRefine Load Balancing (GreedyRefineLB) enhances the basic GreedyLB strategy by introducing nuanced decision-making to minimize migrations while improving load balance. This sophisticated approach optimizes load distribution by selectively migrating objects to achieve better overall performance.

Operating on a similar premise as GreedyLB, GreedyRefineLB begins by measuring the workload on each processor, gathering pertinent metrics such as CPU utilization and memory usage. However, its refinement process incorporates additional considerations:

- Greedy Assignment: Like GreedyLB, GreedyRefineLB initially assigns the heaviest remaining object to the least loaded processor. This step ensures efficient utilization of resources.

- Migration Optimization: GreedyRefineLB goes further by evaluating the current processor's load status. If the object is currently assigned to a heavily loaded processor, and if it reaches above the tolerance level, it is migrated to the least loaded proces-

sor [10]. However, if the object is already on a lightly loaded processor, it remains there to minimize unnecessary migrations and overhead.

- Tolerance Adjustment: GreedyRefineLB offers flexibility through an optional tolerance parameter specified via the configuration file. This parameter defines the acceptable load tolerance above the maximum load produced by the basic Greedy algorithm. Adjusting this tolerance allows users to balance between load optimization and migration overhead according to application requirements.

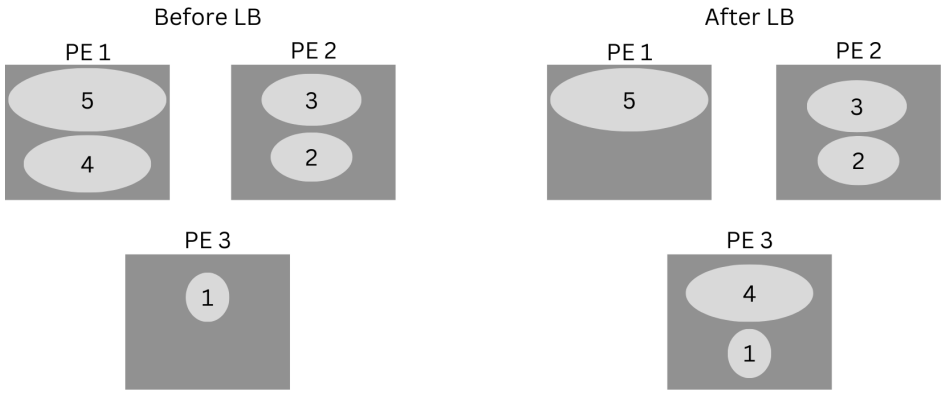


Figure 2.2: This example shows GreedyRefineLB before and after load balancing in Charm++, where each load (marked with a number indicating its size) is assigned to a PE. It shows how this strategy migrates the heaviest object to the least loaded processor, bringing the max load per PE from 9 to 5.

By intelligently balancing load distribution and minimizing migrations, GreedyRefineLB achieves improved performance without excessively disrupting the system. This makes it particularly suitable for scenarios where minimizing overhead is crucial while maintaining optimal load balance. The task distribution using GreedyRefineLB is illustrated in Figure 2.2.

Usage: `+balancer GreedyRefineLB`

GreedyRefineLB represents a refined approach to load balancing, leveraging the strengths of greedy algorithms while introducing intelligent migration strategies. Its adaptive nature and configurable parameters make it a versatile tool for optimizing performance in Charm++ applications.

2.3 METIS LOAD BALANCING

Graph partitioning has great significance for work distribution in parallel applications and locality maximization. It helps in balancing the computations and minimizing the amount of

communication. Metis Load Balancing (MetisLB) leverages the capabilities of the external Metis library, a high-performance graph partitioning tool [11]. It involves the following steps:

- Graph Construction: Charm++ constructs a communication graph representing the application. Nodes in the graph represent processors, and edges represent communication between chares residing on different processors. Edge weights can be assigned based on communication volume or frequency.
- Metis Partitioning: The communication graph is passed to the Metis library. Metis utilizes sophisticated algorithms to partition the graph into balanced subgraphs, aiming to minimize inter-processor communication while keeping the workload distribution fair.
- Chare Placement: Chares are assigned to processors based on the partitioning results obtained from Metis.
- Migration: Chares requiring relocation are migrated to their designated processors.

MetisLB often achieves superior load balancing compared to GreedyLB and RefineLB, especially for applications with complex communication patterns. However, it introduces the overhead of graph construction and the dependency on an external library. The task assignment to PEs using MetisLB is shown in Figure 2.3 with an example.

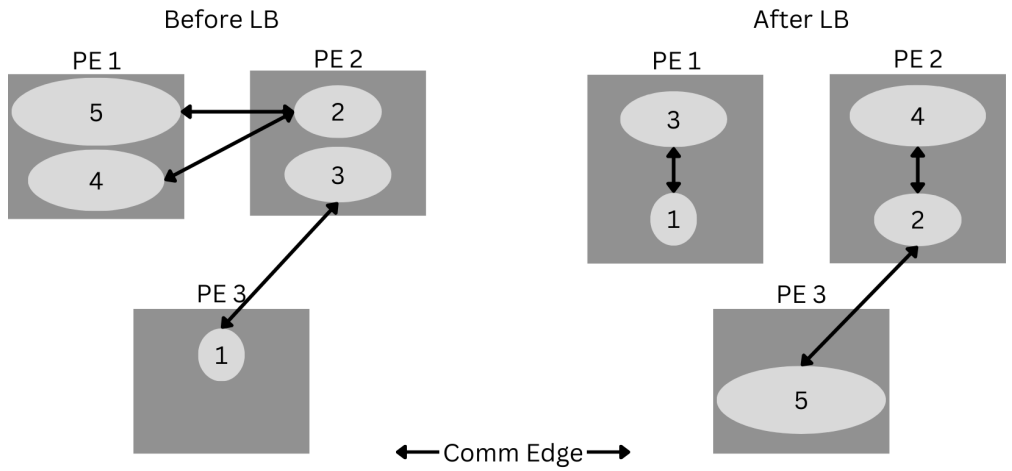


Figure 2.3: This example shows MetisLB before and after load balancing in Charm++, where each load (marked with a number indicating its size) is assigned to a PE. The arrows indicate communication between objects. MetisLB places closer indices in the same PE and hence reduces communication overhead.

2.4 GRAPH-REFINEMENT LOAD BALANCER

Graph-Refinement LB, embedded within Charm++, serves as a pivotal component for facilitating dynamic load balancing in parallel computing environments. It does so by providing a fast distributed strategy which also reduces object migration by only moving objects from overloaded PEs to underloaded PEs.

Load balancing is imperative in parallel computing systems to ensure optimal resource utilization and prevent performance bottlenecks. Mechanisms like process migration for load balancing have also been investigated [12], whereas we focus on object migration. In essence, Graph-Refinement LB operates by iteratively redistributing computational tasks among PEs based on load information exchanged among them. The exchange of information and communication between PEs is depicted with an example in Figure 2.4. A more detailed explanation is described in Chapter 3. But first, let's delve into the inner workings of Graph-Refinement LB to gain an overview of its functionality.

- Initialization: The process begins with the initialization phase, where each PE aggregates load information for objects on the PE to compute the initial computational workload. A graph of neighbor nodes is computed on each PE. This ensures that all PEs are aware of their respective starting points in terms of workload distribution.

- Exchange of Load Information: Central to Graph-Refinement LB is the exchange of load information among neighboring PEs. Through inter-PE communication channels, each PE shares its load information with its adjacent PEs. This iterative process of exchange enables PEs to gain insights into the overall workload distribution within the system, laying the groundwork for subsequent load-balancing decisions.

- Computation of Load Changes: Following the iterative process of exchange of load information, each PE computes the change in workload relative to its neighboring PEs. This computation is instrumental in assessing the dynamics of workload distribution within the system. PEs analyze the disparity between their workload and that of neighboring PEs to gauge the necessity for load redistribution. This redistribution is again, an iterative process.

- Determination of Load Migration: Based on the computed load changes, Graph-Refinement LB makes informed decisions regarding load migration. PEs with excessive workloads relative to their neighbors may initiate load migration processes to offload tasks to underutilized PEs. Conversely, PEs with spare computational capacity may

accept migrated tasks to aid in load redistribution. The goal is to achieve a more equitable distribution of computational burden across all PEs.

- Migration of Computational Tasks: The migration of computational tasks occurs in this phase, wherein tasks are transferred between PEs according to the load migration decisions made in the previous step. Task migration entails the movement of computational workloads from overloaded PEs to less burdened ones, thereby alleviating imbalances in workload distribution.

- Convergence Criterion: The convergence criterion serves as a termination condition for the load-balancing process. Typically, convergence is achieved when the system attains a predefined threshold of load balance across all PEs. Currently, we use a fixed number of iterations. A convergence criterion that ensures the load-balancing process terminates once an acceptable level of equilibrium is reached will be implemented in the future.

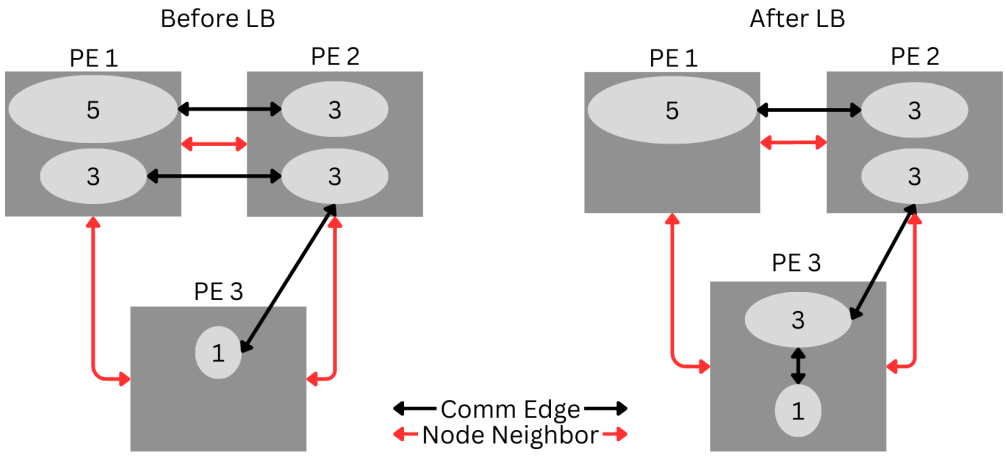


Figure 2.4: This example shows Graph-Refinement LB before and after load balancing in Charm++, where each load (marked with a number indicating its size) is assigned to a PE. The black arrows indicate communication between objects and the red arrows depict the communication with the neighboring nodes.

2.5 CHOOSING THE RIGHT LOAD BALANCER

As highlighted in [13], selecting the optimal load-balancing strategy for a Charm++ application is a nuanced decision. Several factors play a role, including the application's inherent characteristics (workload distribution, communication patterns, communication overhead),

system features (number of processors, heterogeneity, network topology), and desired performance outcomes. The optimal load-balancing strategy for a Charm++ application depends on several factors:

- Application Characteristics: Factors like workload distribution within chares, communication patterns, and the importance of communication overhead should be considered.
- System Characteristics: The number of processors, their heterogeneity, and network topology can influence the effectiveness of different strategies.
- Performance Requirements: For scenarios where fast execution is crucial, GreedyLB might suffice. For applications demanding optimal load distribution, MetisLB could be the choice. RefineLB offers a balance between speed and quality.

For scenarios prioritizing rapid execution, GreedyLB, with its focus on basic distribution, might appear sufficient. However, when the application demands both speed and an optimal level of load balance, more sophisticated strategies like the graph refinement approach come into play. While GreedyLB offers a quick initial distribution by assigning the "heaviest object" to the least loaded processor, the graph refinement strategy can further analyze and refine this placement, potentially leading to superior communication efficiency and load balancing in the long run.

Charm++ also allows the users to dynamically switch between load-balancing strategies during application execution. Experimentation with different strategies helps identify the most suitable option for a specific application and computing environment.

CHAPTER 3: COMMUNICATION-AWARE LOAD BALANCING

3.1 INTRODUCTION

It is a well-known fact that parallel computing unlocks significant processing power by distributing tasks across multiple processing units. However, achieving optimal performance requires efficient load balancing, ensuring each unit is neither overloaded nor underutilized. This task becomes particularly challenging when dealing with applications featuring:

- **High Communication Volume:** Frequent data exchange between processing units (often called "chunks") is essential for the program's execution.
- **Dynamic Workload:** The distribution of work across processing units changes frequently throughout the program's execution.

Traditional load-balancing algorithms struggle in these scenarios. Greedy algorithms, known for their simplicity, might trigger a large number of unnecessary object migrations, increasing overhead. Conversely, refinement algorithms, focused on minimizing migrations, can lead to objects being scattered away from their frequent communication partners, hindering performance.

Communication-aware diffusion load balancing emerges as a solution tailored for high-communication, dynamic workloads [14]. It strives to find an equilibrium between minimizing object migrations and maintaining good communication locality. Essentially, the algorithm aims to keep frequently communicating objects physically close together, either on the same processing unit (node) or on neighboring nodes.

3.2 ALGORITHM

A novel variation of the graph-refinement diffusion algorithm has been recently developed by Kavitha Chandrasekar [15], and Monika Gangapuram, for both real machines and as a simulator. The work in this thesis focuses on analyzing the performance of the strategy as implemented in the simulator for specific load patterns using a synthetic benchmark developed in her work. Other graph partitioning methods have been studied in [16].

The Communication-aware Diffusion Load Balancing algorithm operates in a multi-level manner, addressing load balance both at the inter-node and within-node levels. At each level, the algorithm aims to distribute objects effectively while minimizing migration overhead and preserving communication patterns [17].

3.2.1 MULTI-LEVEL APPROACH

The inter-node load balancing is performed at multiple levels, allowing for refinement, optimization, and utilizing various load balancing strategies at each stage. This enables the algorithm to achieve a balanced distribution of objects across the entire system.

3.2.2 INTER-NODE LOAD BALANCING

Each node selects a subset of its neighboring nodes to exchange objects with. The algorithm chooses n highest communicating neighbors initially and iteratively adds neighbors till K neighbors are selected, ensuring symmetry is maintained in the communication network.

3.2.3 WITHIN-NODE LOAD BALANCING

- **Object Distribution within Nodes**: Within each node, objects are managed using a refinement strategy to ensure optimal load balance. Objects are kept within the node, and tokens are used to represent incoming objects.
- **Refinement Strategy**: Objects and object tokens are dynamically moved within the node using a refinement strategy. This strategy aims to minimize load imbalances while avoiding excessive migrations.

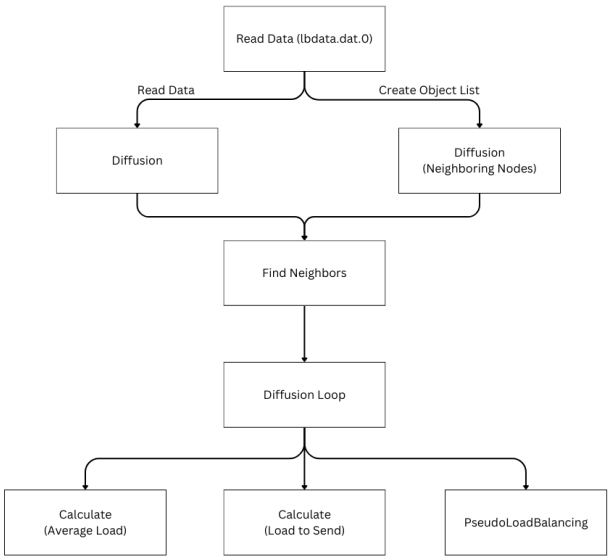


Figure 3.1: Collaboration diagram for Graph-Refinement Load Balancer

By employing a combination of inter-node and within-node load balancing techniques, Communication-aware Diffusion Load Balancing achieves efficient load distribution with minimal migration overhead. The algorithm's focus on maintaining communication proximity between objects ensures that communication-intensive applications can operate with optimal performance and scalability. The collaboration diagram for graph refinement LB can be found in Figure 3.1.

Since our main focus is on inter-node communications (increasing internal communication and decreasing external communication over LB iterations), the pseudocode for `Diffusion.C` which implements this is seen in Listing 3.1.

```
// Node initialization
Diffusion(int num_nodes) {
  // ... (read data, create object list, find neighbors)
}

// Diffusion loop (one iteration)
void diffuse() {
  avg_load = average_neighbor_load();
  to_send = calculate_load_to_send(avg_load);
  PseudoLoadBalancing(to_send);
}

// Functions for specific tasks
int average_neighbor_load() {
  // ... (calculate the average load of neighbors)
}

int calculate_load_to_send(int avg) {
  // ... (calculate load difference with neighbors)
}

void PseudoLoadBalancing(int to_send) {
  // ... (implement pseudo-balancing strategy)
}

// Final load balancing
void perform_load_balancing() {
  // ... (transfer objects based on communication cost)
}
```

Listing 3.1: Pseudocode of Diffusion Load Balancer Algorithm

CHAPTER 4: BENCHMARK APPLICATION

This chapter introduces a benchmarking application to evaluate various load-balancing strategies discussed previously.

4.1 STENCIL APPLICATION

4.1.1 PURPOSE

Stencil3d is a well-established synthetic benchmark used to evaluate the performance of parallel and distributed computing systems. It simulates a three-dimensional (3D) heat diffusion process, making it a valuable tool for understanding how these systems handle real-world scientific computations.

It divides a 3D grid (array) into sub-blocks and assigns them to worker processes for computation. This distribution allows the application to leverage the processing power of multiple machines or cores to solve the simulation faster compared to a single machine.

4.1.2 FUNCTIONALITY

The application performs the following steps:

- Setup:
  - Reads command-line arguments to determine the dimensions of the 3D array (global grid) and the sub-blocks (assigned to each worker process).
  - Calculates the number of worker processes needed in each dimension (x, y, z) based on the array and block sizes.
  - Creates worker processes using a communication library.
- Data Distribution:
  - The global 3D array is conceptually divided into sub-blocks, and each worker process is responsible for calculations on its assigned sub-block.
  - Worker processes communicate ghost cell data (values from neighboring sub-blocks) at the boundaries to perform accurate calculations on the edges.
- Stencil Computation:

  - Each worker process iterates through a loop for a specified number of times (iterations).
  - Within each iteration, a stencil update rule is applied to the temperature values within the sub-block. This rule considers the current temperature value and the temperature values of its six nearest neighbors (in the 3D grid).
  - An artificial workload change is introduced for a specific range of processes at certain iterations to simulate uneven workloads.

- Communication and Synchronization:
  - Worker processes exchange ghost cell data with their neighbors at the beginning of each iteration to ensure they have up-to-date information for boundary calculations.
  - The application might use synchronization points (`AtSync`) at specific intervals to balance the workload among processes if the workload becomes uneven due to artificial changes.

- Completion:
  - After all iterations are complete, worker processes signal their completion to the main process.
  - The main process might calculate and report the execution time.

4.1.3 CODE BREAKDOWN

The Stencil3d code [18] showcases the following implementation:

- Global Variables: These variables define the dimensions of the global array, sub-blocks, and the worker process grid.
- Class 'Main': This class handles the main program logic. It parses command-line arguments, sets up worker processes, and initiates the computation.
- Class 'Stencil': This class represents a worker process. It holds the sub-block data (temperature), and ghost cell data, and performs computations and communication with neighboring processes.

4.1.4 BENEFITS OF PARALLEL STENCIL COMPUTATION

- Scalability: The application can be easily scaled to a larger number of worker processes to handle larger 3D grids or more complex simulations.
- Performance: Stencil3d leverages parallel processing to achieve significant performance improvements compared to a single-threaded implementation. The workload is strategically distributed among multiple processes, enabling concurrent computations and accelerating the overall simulation. We employ a synthetic load pattern to rigorously evaluate the application's scalability and efficiency under parallel execution.

This Stencil3d application demonstrates a parallel approach to solving a 3D heat diffusion problem. It leverages multiple worker processes to improve the computational efficiency of the simulation.

4.2 MODIFIED MODEL: BLOCK MAPPING

Block mapping groups data elements of the 3D grid in Stencil3d together and assigns them to the same PE within a node. This approach aims to minimize communication across different PEs and maximize communication happening within a single PE's local memory.

By grouping related data objects and assigning them to the same PE, block mapping increases the chance that these objects will reside in the same local memory space. This allows for faster communication and data exchange between them compared to a scenario where they might be scattered across different PEs. The C code for the same can be found in Listing 4.1.

```
class BlockMap2 : public CkArrayMap {
public:
    int ratiox, ratioy, ratioz;
    int penum;
    double procDim;

    BlockMap2(int numx, int numy, int numz, double proc_numxyz) {
        procDim = int(cbrt(proc_numxyz));
        if (procDim*procDim*procDim != proc_numxyz) {
            CkAbort("Block map requires a power of three number of PEs\n")
   ;
        }
        else {
            ratiox = numx / procDim;
            ratioy = numy / procDim;
            ratioz = numz / procDim;
        }
    }
    int registerArray(CkArrayIndex& numElements,CkArrayID aid) {
        return 0;
    }
    // Call that returns the homePE of element with index idx
    int procNum(int arrayHdl, const CkArrayIndex& idx) {
    // Sum up the number of ints in the index (not the same as dimension)
        int x, y, z;
        x = ((int*)idx.data())[0];
        y = ((int*)idx.data())[1];
        z = ((int*)idx.data())[2];

        int procx, procy, procz;

        procx = x / ratiox;
        procy = y / ratioy;
        procz = z / ratioz;

        penum = procx*procDim*procDim + procy*procDim + procz;

        // Mask the sum to a PE number
        return penum;
    }
};
```

Listing 4.1: BlockMap in Charm++

CHAPTER 5: EXPERIMENTS AND EVALUATION

5.1 ENVIRONMENT SETUP

We used a computing instance from Delta [19] to conduct experiments, study performance, and evaluate. Delta is an open-production environment with a site at the University of Illinois at Urbana-Champaign. It is a dedicated, NSF's ACCESS [20] allocated resource designed by HPE and NCSA, delivering a highly capable GPU-focused compute environment for GPU and CPU workloads.

This experimental setup utilizes the Delta computing instance to leverage its high-performance GPU and CPU capabilities and user-friendly access for studying the performance of Stencil3d in addressing the communication costs of the application. A computing instance on Delta with CPUs installed was created. The detailed instance specification is shown in Table 5.1. The experiment will evaluate factors like effects on maximum loads per PE and external communication for average load per PE and total communication respectively to assess the effectiveness of the chosen configuration.

OS	RedHat 8.8
Architecture	x86_64
CPU	2 AMD EPYC 7763 64-Core Processor @ 2.45GHz
L1d cache	32K
L1i cache	32K
L2 cache	512K
L3 cache	32768K
RAM	252 GB
Virtualization	AMD-V

Table 5.1: CPU Compute Node Specifications

5.2 LOAD PATTERNS

5.2.1 PE BASED LOAD PATTERN

This strategy creates load imbalance by assigning a higher workload (3.5) to PEs divisible by 3, while all others receive a lower load (1.0). This creates an uneven distribution of work, where specific PEs are significantly more burdened, potentially leading to underutilization of resources on lightly loaded PEs. The pseudocode for this load pattern is given in Listing 5.1.

```
Start
 |
For each Processing Element (PE):
 |   If PE is divisible by 3:
 |   |   Assign load = 3.5
 |   Else:
 |   |   Assign load = 1.0
End
```

Listing 5.1: Pseudocode for PE Based Load Pattern

5.2.2 LINEAR LOAD PATTERN

This pattern gradually increases the load assigned to each PE as the PE number increases. Starting from 0 for the first PE, the load progressively rises until it reaches the highest value for the last PE `numNodes`. This creates a gradient of workload across the PEs. The pseudocode for this load pattern is given in Listing 5.2.

```
Start
 |
For each PE from 0 to numNodes:
 |   Calculate load using linear distribution
 |   Set wallTime for the corresponding object to this load
End
```

Listing 5.2: Pseudocode for Linear Load Pattern

5.2.3 TRIANGULAR LOAD PATTERN

The workload distribution resembles a triangle. PEs in the middle have the highest workload, and it gradually decreases towards both ends (PEs with lower or higher numbers). The pseudocode for this load pattern is given in Listing 5.3.

```
Start
 |
For each PE from 0 to numNodes:
 |   Calculate load using a triangular distribution
 |   Set wallTime for the corresponding object to this load
End
```

Listing 5.3: Pseudocode for Triangular Load Pattern

5.2.4 DYNAMIC-SPIKE LOAD PATTERN

This pattern introduces temporary workload spikes during the simulation. The specific details of how these spikes are created can vary. It might involve random bursts, time-based triggers, or other mechanisms, but the core idea is to introduce unpredictable increases in workload on top of a base load distribution.

In our case, we induced spikes by setting a probability of a random spike in any loop iteration and a spike factor by which it increases the load. The pseudocode for this load pattern is given in Listing 5.4.

```
Start
 |
For each Object:
 |   Calculate load by creating a dynamic spike:
    |   if random_number is within spike_probability and PE is even:
        |   load *= spike_factor
 |  Set wallTime for the corresponding object to this load
End
```

Listing 5.4: Pseudocode for Dynamic-Spike Load Pattern

5.2.5 HISTORY-BASED LOAD PATTERN

We employed an object ID-based load distribution pattern for multiple load balancing steps. This initial assignment assigned higher workloads to PEs divisible by 3. To simulate real-world workload fluctuations, the workload for each object was then adjusted every 100 iterations by a random factor, either increasing by 50% or decreasing by 30%. This approach allowed us to observe how this dynamic load pattern affected performance across the different balancing steps. The pseudocode for this load pattern is given in Listing 5.5.

```
Start
 |
For each Object:
 |   Calculate load based on history
 |   Assign load to Processing Element
End
```

Listing 5.5: Pseudocode for History-Based Load Pattern

5.3 PERFORMANCE EVALUATION

5.3.1 NUMBER OF PES AND ITS EFFECTS ON THE LOAD DISTRIBUTION

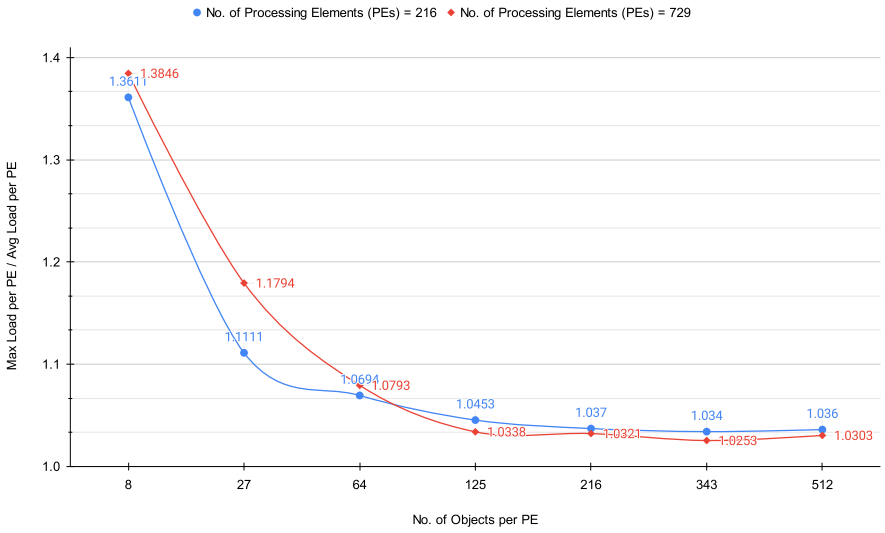


Figure 5.1: No. of Objects vs Ratio of Max-to-Avg Load per PE for 216 and 729 PEs

Figure 5.1 shows the ratio of maximum PEs per load to the average PEs per load. A higher ratio indicates more overhead. In the graph, the y-axis represents the ratio and the x-axis represents the number of objects per PE.

We performed the experiments with 216 PEs and 729 PEs and studied how the ratio changed with the number of objects per load for a load pattern based on PE, where even-numbered PE is assigned a load of 2.0 and odd-numbered with 1.0.

As the number of objects per load increases, the ratio of maximum PEs to average PEs decreases. This means that the overhead decreases. For instance, with 8 objects per load which translates to 1728 objects (216 * 8), the ratio is around 1.35, which suggests 35% overhead. On the other hand, with 125 objects per load which is around 27K objects, the ratio goes down to around 1.04, which translates to roughly 4% overhead.

Therefore, it is learned from this graph that increasing the number of objects per load helps reduce overhead in PEs.

Similarly, if we observe the curve for 729 PEs in Figure 5.1, we can see an overhead reduction from 40% to 4%. This significant improvement can be attributed to a more efficient utilization of PEs. When overhead is high, a larger portion of PEs is dedicated to managing tasks rather than performing core computations. Reducing overhead frees up PEs

to focus on computations, leading to faster execution times and improved throughput.

Building upon load imbalances from Section 5.2, we evaluated the performance of our graph refinement strategy against a greedy refinement approach using simulators. For our graph refinement simulator, we set the fixed iteration count of 80 for diffusing load values to neighbors and set K, the number of neighbors per PE, as 8. Our key metrics were load distribution and communication patterns, intending to maximize internal communication and minimize external communication (as discussed in Chapter 4). This translates to a desired ratio of external communication to total communication close to 0. It is also important to consider the impact of the initial load distribution on the effectiveness of the refinement strategy.

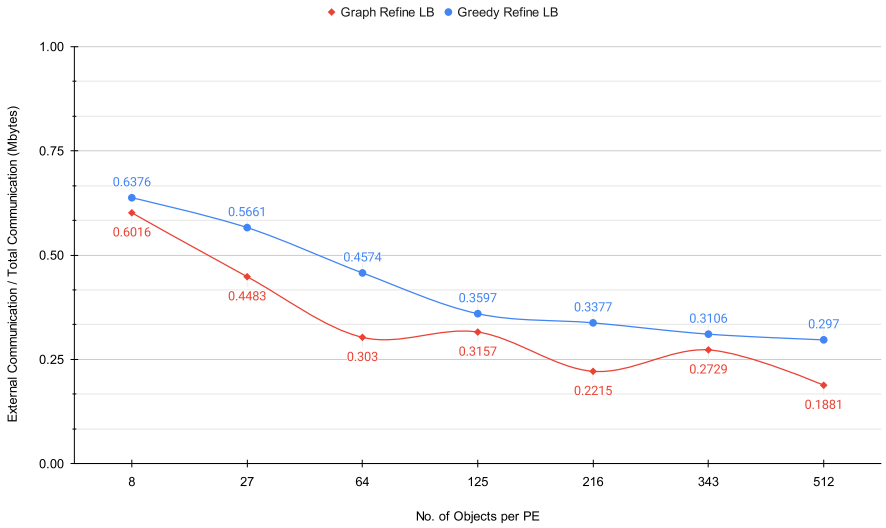


Figure 5.2: No. of Objects vs Ratio of External-to-Total Communication - PE-based Load Pattern

5.3.2 APPROACH 1: PE-BASED LOAD PATTERN

In this experiment, we used a hard-coded load pattern where PEs divisible by 3 received a higher load (3.5) compared to others (1.0). While the graph refinement strategy can potentially improve communication patterns, this static load distribution might create inefficiencies.

Higher communication: PEs with a higher load might need to communicate more frequently to offload work, potentially increasing external communication.

Underutilized resources: PEs with a lower load might become underutilized, reducing overall efficiency.

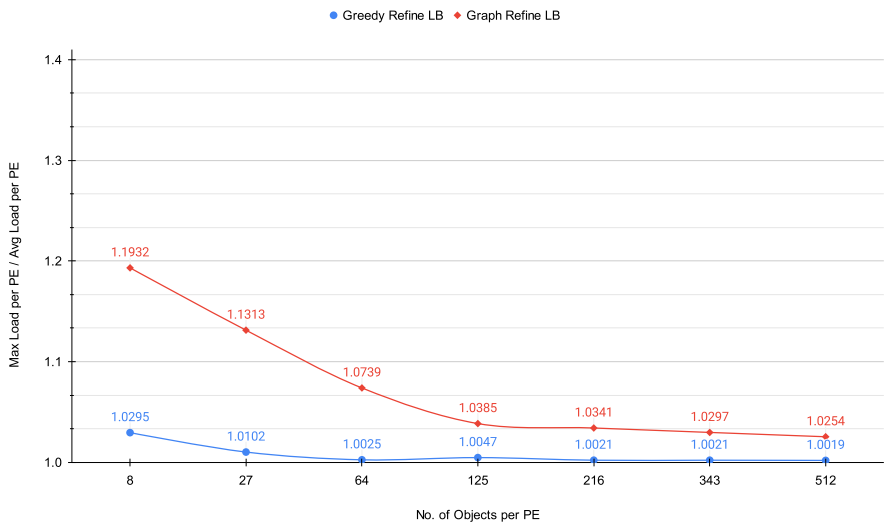


Figure 5.3: No. of Objects vs Ratio of Max-to-Avg Load per PE - PE-based Load Pattern

Figure 5.2 showcases this external-to-total communication ratio for both strategies across different object-per-PE configurations, using 729 PEs for the test. As evident from the figure, the graph refinement strategy demonstrates a significant improvement in communication efficiency compared to the greedy algorithm, achieving a closer ratio to the ideal 0 value. This indicates better utilization of internal communication channels. However, as a trade-off, Figure 5.3 reveals a slight deviation in load distribution for the graph refinement strategy compared to the greedy approach. This figure depicts the ratio of maximum load to average load per PE for both strategies across object-per-PE configurations.

5.3.3 APPROACH 2: LINEAR LOAD PATTERN

A linear load pattern, where the workload assigned to each PE gradually increases with its PE number, offers a more balanced starting point compared to static patterns. This approach aligns better with the goal of maximizing internal communication, as PEs with closer PE numbers are likely to have more data dependencies. The graph refinement strategy can potentially leverage this locality by grouping PEs with similar workloads together, minimizing the need for communication across distant PEs (which becomes external communication). This could lead to a lower external-to-total communication ratio, as shown in Figure 5.4, signifying improved communication efficiency.

However, limitations exist. The fixed linear increase might not perfectly match the actual workload requirements of the graph data, leading to slight imbalances at the edges. The first

few PEs might be underutilized, while the last few could be overloaded. Additionally, the graph refinement strategy might not be able to fully compensate for any inherent imbalances within the graph structure itself. This can be observed in Figure 5.5.

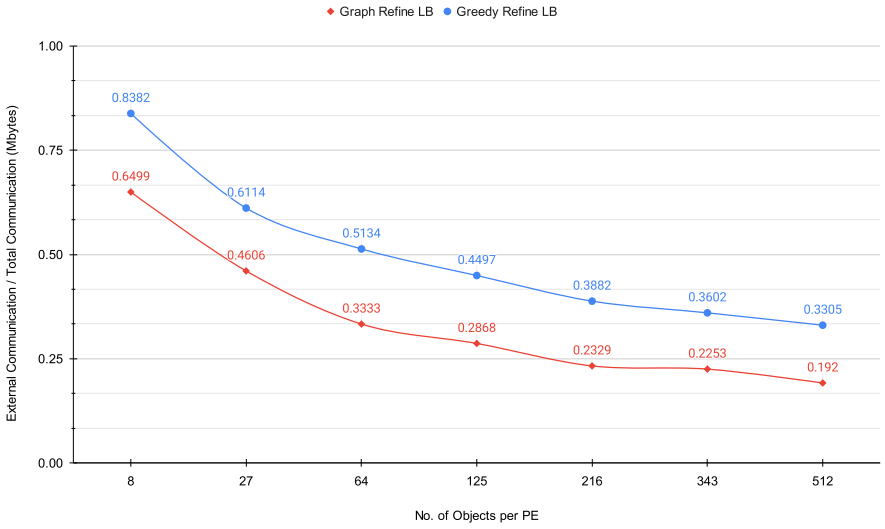


Figure 5.4: No. of Objects per PE vs Ratio of External-to-Total Communication - Linear Load Pattern

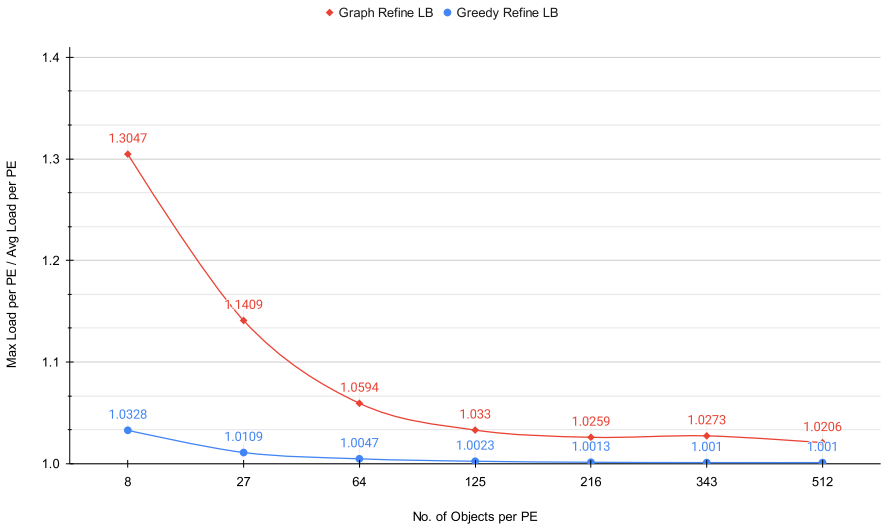


Figure 5.5: No. of Objects vs Ratio of Max-to-Avg Load per PE - Linear Load Pattern

5.3.4 APPROACH 3: TRIANGULAR LOAD PATTERN

A triangular load pattern, where the workload assigned to PEs resembles a triangle with the highest load in the middle and tapering towards both ends, presents an interesting scenario for the graph refinement strategy. Compared to a purely linear increase, the triangular pattern distributes the workload more evenly across most PEs, potentially reducing underutilized resources at the beginning and end. The external-total communication can be seen in Figure 5.6. However, the high concentration of workload in the middle PEs might lead to increased communication overhead as they exchange data. The graph refinement strategy might need to carefully manage communication patterns to avoid bottlenecks in this central region. The effects on the distribution of the workload in Figure 5.7 also capture this trade-off.

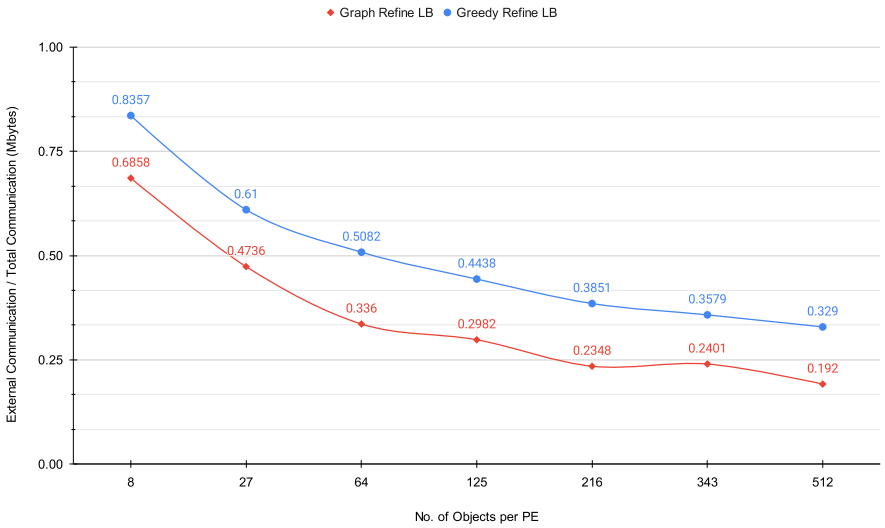


Figure 5.6: No. of Objects per PE vs Ratio of External-to-Total Communication - Triangular Load Pattern

5.3.5 APPROACH 4: DYNAMIC-SPIKE LOAD PATTERN

The dynamic spike load pattern introduces temporary surges in workload on top of a base load distribution. While this approach can simulate real-world workload variations, random or time-based workload spikes can disrupt the communication patterns established by the graph refinement strategy. PEs that were previously grouped for efficient communication might experience sudden spikes, requiring them to communicate with different PEs. This can lead to increased communication overhead and reduced efficiency. The graph refinement strategy might not be able to fully anticipate and adapt to dynamic spikes. If a spike

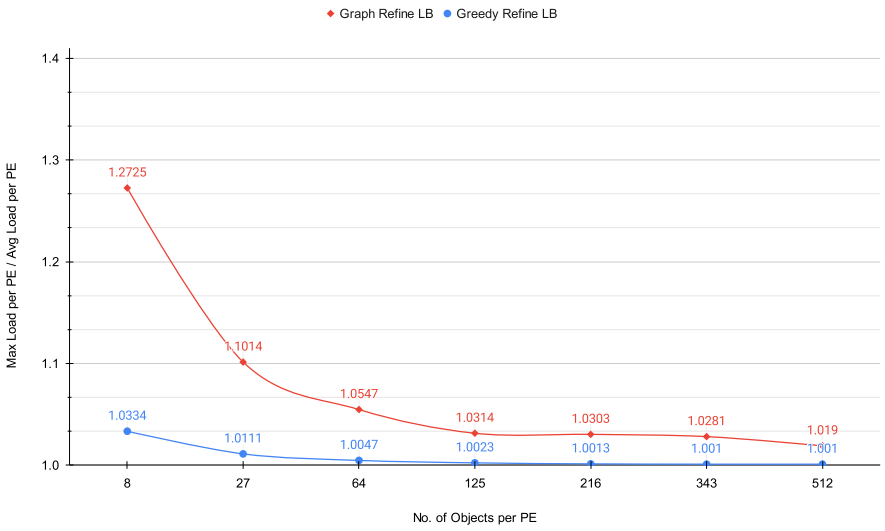


Figure 5.7: No. of Objects vs Ratio of Max-to-Avg Load per PE - Based on Triangular Load Pattern

hits an under-resourced PE, it could lead to processing delays and bottlenecks. Conversely, spikes on already heavily loaded PEs might overwhelm their capacity, hindering overall progress. The graph refinement strategy exhibited a clear advantage in the external-to-total communication ratio when the number of objects per PE increased, as shown in Figure 5.8. However, the greedy refinement approach maintained a slightly better max-to-avg load ratio per PE (Figure 5.9).

5.3.6 APPROACH 5: HISTORY-BASED LOAD PATTERN

This experiment aimed to assess load-balancing strategies for Stencil3d, a program capable of adjusting workloads dynamically. To simulate real-world applications, we implemented random changes in object workloads every 100 iterations over 10 simulated load balancing steps (for a total of 1000 iterations). Two object configurations were tested: one with 64 objects per processing element (PE), resulting in a total of 36^3 objects, and another with 216 objects per PE, resulting in a total of 54^3 objects.

The findings revealed an intriguing trade-off between two load-balancing approaches: graph refinement and greedy refinement. While graph refinement exhibited better performance than the greedy approach across multiple load balancing steps in both configurations, its communication efficiency showed a slight decline over time (as seen in Figure 5.10a). This indicates a potential increase in external communication compared to total communication

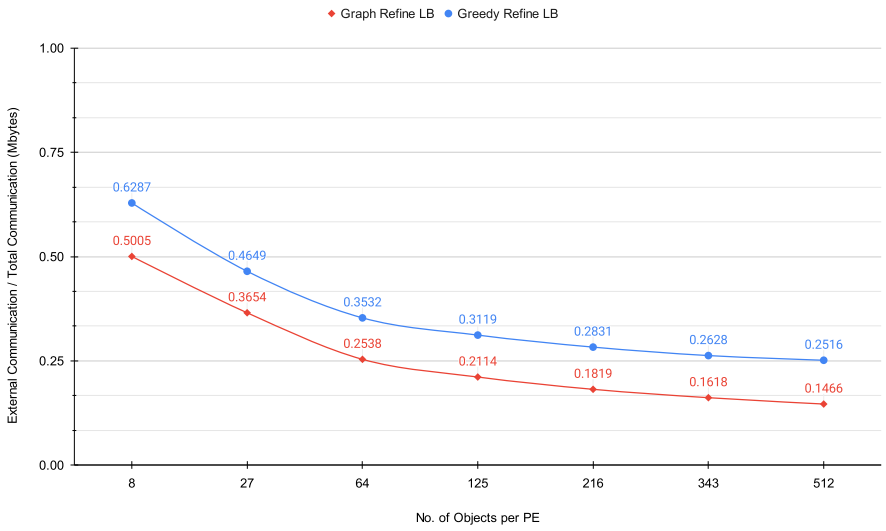


Figure 5.8: No. of Objects per PE vs Ratio of External-to-Total Communication - Dynamic Spike Load Pattern

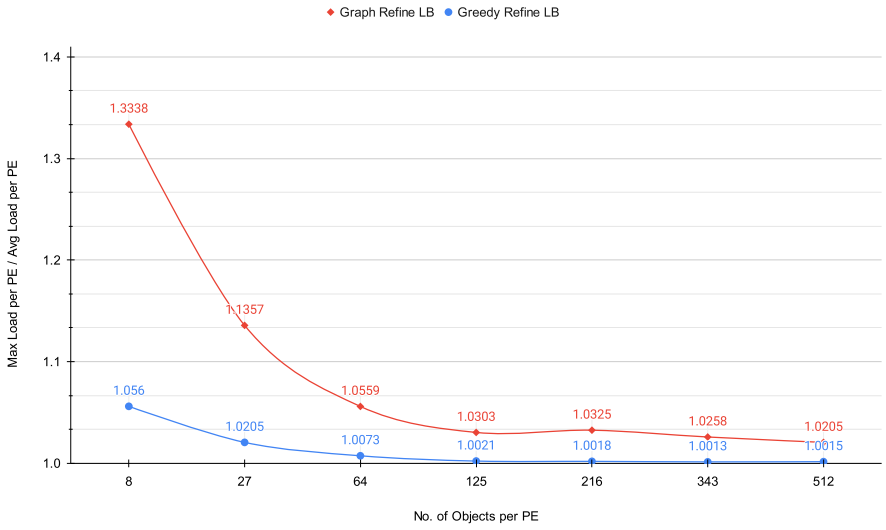
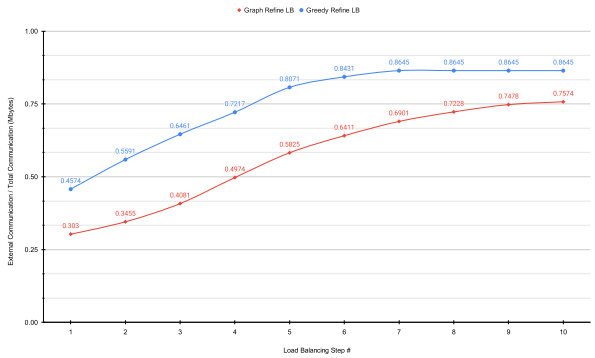


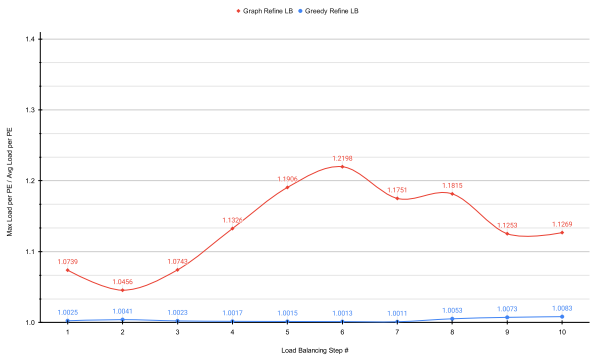
Figure 5.9: No. of Objects vs Ratio of Max-to-Avg Load per PE - Dynamic Spike Load Pattern

as the iterations progressed, suggesting room for improvement through internal optimization or increased parallelization within the load balancer itself. Interestingly, the performance improved slightly when the number of objects per PE was increased to 216 (Figure 5.11a).

In contrast, the greedy approach maintained a more balanced workload distribution throughout the experiment for both object configurations (demonstrated by a lower ratio of

maximum load to average load in Figures 5.10b and 5.11b, respectively). However, due to its tendency to maintain a more even distribution of work (reflected in a lower external-to-total communication ratio), the graph refinement strategy might be a more suitable choice for scenarios requiring long-term load balancing.

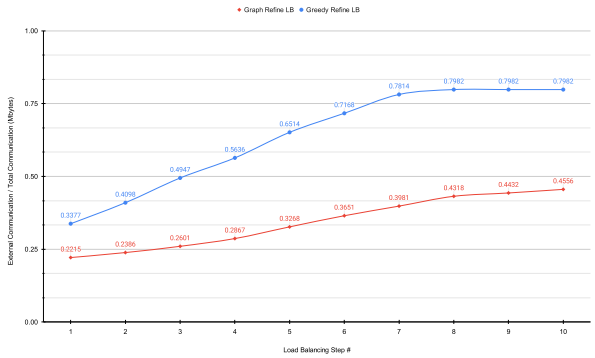


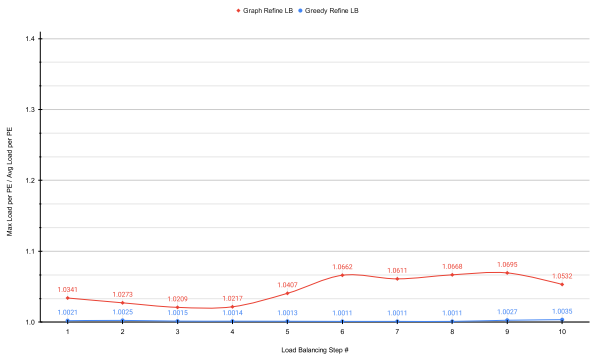


(a) No. of Objects per PE vs Ratio of External-to-Total Communication

(b) No. of Objects vs Ratio of Max-to-Avg Load per PE

Figure 5.10: Performance for 64 Objects per PE for History-based Load Pattern





(a) No. of Objects per PE vs Ratio of External-to-Total Communication

(b) No. of Objects vs Ratio of Max-to-Avg Load per PE

Figure 5.11: Performance for 216 Objects per PE for History-based Load Pattern

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

In this thesis, we have emphasized the critical role of load balancing in enhancing the performance of Charm++ applications. By analyzing existing load-balancing strategies and the novel Communication-aware Diffusion LB algorithm, we address the challenges posed by intense communication demands in dynamic workloads.

This work validates the effectiveness of the Communication-aware Diffusion LB algorithm. By striking a delicate balance between fairly even load distribution and enhancing communication locality, this algorithm significantly improves the efficiency of Charm++ applications.

Future work could involve further refinement of the algorithm to adapt to diverse application characteristics and network topologies. Additionally, exploring integration with other Charm++ functionalities and investigating its performance with a broader range of parallel applications would be valuable areas for further research.

Evaluating the algorithm across a wider spectrum [4] of parallel applications beyond the Stencil3d benchmark will provide deeper insights. Understanding its behavior in diverse scenarios will help us generalize its applicability.

In summary, our work highlights the importance of communication-aware load balancing in parallel computing environments. As applications evolve, finding the right balance between local optimization and global efficiency remains crucial. The Communication-aware Diffusion LB algorithm holds promise as a key enabler for achieving this balance and enhancing system performance.

REFERENCES

[1] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[2] N. Wu, I. Gonidelis, S. Liu, Z. Fink, N. Gupta, K. Mohammadiporshokooh, P. Diehl, H. Kaiser, and L. V. Kale, *Quantifying Overheads in Charm++ and HPX Using Task Bench*. Springer Nature Switzerland, 2023, p. 5–16. [Online]. Available: http://dx.doi.org/10.1007/978-3-031-31209-0_1

[3] "Parallel programming laboratory." [Online]. Available: https://charm.cs.uiuc.edu/

[4] "The charmplusplus website." [Online]. Available: https://charmplusplus.org/

[5] B. Xu, W. Lian, and Q. Gao, "Migration of active objects in proactive," *Information and Software Technology*, vol. 45, no. 9, pp. 611–618, 2003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095058490300048X

[6] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel Programming with Migratable Objects: Charm++ in Practice," ser. SC, 2014.

[7] "Orthogonal recursive bisection." [Online]. Available: https://cseweb.ucsd.edu/~baden/classes/cse262_wi02/lectures/Lec07/Lec07.html

[8] H. D. Simon and S.-H. Teng, "How good is recursive bisection?" *SIAM J. Sci. Comput.*, vol. 18, pp. 1436–1445, 1997. [Online]. Available: https://api.semanticscholar.org/CorpusID:3212422

[9] "Recursive coordinate bisection." [Online]. Available: https://sandialabs.github.io/Zoltan/ug_html/ug_alg_rcb.html

[10] "Available load balancing strategies." [Online]. Available: https://charm.readthedocs.io/en/latest/charm++/manual.html#available-load-balancing-strategies

[11] G. Karypis, "Multi-constraint mesh partitioning for contact/impact computations," in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, ser. SC '03. New York, NY, USA: Association for Computing Machinery, 2003. [Online]. Available: https://doi.org/10.1145/1048935.1050206 p. 56.

[12] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "The limited performance benefits of migrating active processes for load sharing," *SIGMETRICS Perform. Eval. Rev.*, vol. 16, no. 1, p. 63–72, may 1988. [Online]. Available: https://doi.org/10.1145/1007771.55604

[13] H. Menon, K. Chandrasekar, and L. V. Kale, "Poster: Automated load balancer selection based on application characteristics," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3018743.3019033 p. 447–448.

[14] M. Lieber, K. Gößner, and W. E. Nagel, "The potential of diffusive load balancing at large scale," in *Proceedings of the 23rd European MPI Users' Group Meeting*, ser. EuroMPI '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2966884.2966887 p. 154–157.

[15] K. Chandrasekar, "Adaptive runtime techniques for node-aware resource optimization."

[16] A. Bhatele, S. Fourestier, H. Menon, L. V. Kale, and F. Pellegrini, "Applying graph partitioning methods in measurement-based dynamic load balancing." [Online]. Available: https://www.osti.gov/biblio/1114706

[17] "Diffusionlb simulator." [Online]. Available: https://github.com/kavithachandrasekar/pgms/tree/main/charm%2B%2B/diffusion_only

[18] "Stencil3d code." [Online]. Available: https://github.com/UIUC-PPL/charm/tree/main/examples/charm%2B%2B/load_balancing/stencil3d

[19] "Delta user documentation." [Online]. Available: https://docs.ncsa.illinois.edu/systems/delta/en/latest/index.html

[20] "Access allocations." [Online]. Available: https://access-ci.org/