PERFORMANCE ANALYSIS OF PARALLEL SOLVER IMPLEMENTED USING FLECSI FOR APPLICATION IN STRUCTURAL DYNAMIC PROBLEMS

by

Chuanqiu He

A Project Report

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the

Requirements for the Degree of

Master of Science in Computer Science

Major: The Department of Computer Science and Engineering

University of Louisiana State University

November 2023

Acknowledgments

I would like to take this opportunity to thank my committee members, Drs. Hartmut Kaiser, Golden G. Richard III, and Patrick Diehl, for serving on my committee and providing helpful feedback and suggestions.

Especially, I would like to express my deep appreciation to my esteemed advisor, Dr. Hartmut Kaiser, who has been a patient and knowledgeable mentor throughout my Master's in Computer Science at Louisiana State University. When I joined the Ste||ar group, I had the opportunity to explore the world of C++ under Dr. Kaiser's invaluable mentorship. During the past years, Dr. Kaiser was instrumental in boosting my C++ skills and navigating the world of high-performance computing. His dedication to my growth as a computer scientist and commitment to knowledge was genuinely inspiring.

I extend my heartfelt gratitude to my co-advisor, Dr. Patrick Diehl, who has been instrumental in my academic journey. Dr. Patrick Diehl guided me through learning Matlab, C++, and structural dynamics, opening doors to the fascinating world of physics simulations. His encouragement and support gave me valuable opportunities to explore and grow in this field.

I deeply thank the FleCSI group at Los Alamos National Lab for their resources and support. I extend my gratitude to Drs. Scott Pakin (LANL), Li-Ta Lo (LANL), Davis Herring (LANL), and Berger (LANL) for their thoughtful research suggestions and help on this project. I would like to express my sincerest gratitude to Ollie for his timely and invaluable support during my graduate research intern program at LANL.

I would like to thank my very nice friends and colleagues working together at STE||AR group, Ali, Bita, Giannis, Karame, Katie, Dr. Nanmiao Wu, Dr. Noujoud, Dr. Rod, Dr.Shahrzad, Srinivas, Dr. Steven R. Brandt, and Dr.Weile Wei, who are always willing to help other people. Finally, A special heartfelt thanks to my parents and my family members for their unwavering support and encouragement. Their constant presence by my side during challenging times has been my greatest source of strength and motivation.

Abstract

In structural dynamics, the importance of parallel computing cannot be overstated. Structural dynamic simulations often involve vast amounts of data and intricate mathematical models that demand substantial computational power. To tackle these challenges effectively, the parallel implementation of such simulations is imperative. Parallel computing enables engineers and scientists to expedite their analyses, thereby providing quicker insights into the behavior of complex structural systems. Moreover, it allows for investigating more detailed and accurate models that would be practically infeasible with sequential computing.

This project delves into the core of this need for parallelism in structural dynamics, focusing on a simple yet illustrative spring-mass system. Employing Euler's solver iteratively, the project leverages the power of parallel computing to simulate this system. The choice of the FleCSI library for this task exemplifies the significance of utilizing modern parallel computing frameworks.

The study not only focuses on the parallel implementation using the FleCSI library but also conducts a comprehensive performance analysis of MPI and HPX backends of FleCSI. The results unearth insights into the behavior of parallel solvers, revealing that, with sufficient problem size, the MPI backend exhibits great scalability across a range of ranks from 1 to 40. It can achieve a maximum of 17 times speedups with 32 ranks(input size = 2 million). However, for smaller problems, performance gains are limited due to considerable communication overhead dominated.

The HPX backend, designed to excel in task-based parallelism, falls short in this specific simulation. There is no performance scaling at all across the cores. Its performance is worse than the MPI backend across various cores because of the application does not expose task parallelism. The application doesn't add parallelism on the flecsi task level because the tasks are strongly sequential. Therefore, the problem's simplicity and limited task dependencies result in a task tree too narrow for HPX to showcase its parallel processing capabilities, highlighting the importance of tailored parallelism for specific problems. In essence, the outcomes from the HPX backend confirm that inefficiency arises from a lack of task dependency parallelism, particularly the width

of the task tree and the magnitude of work within each dependency task.

These findings offer valuable guidance for enhancing performance with the HPX backend. This may involve optimizing algorithms to promote local parallelism and ensuring that tasks contain substantial workloads.

To address this issue, we introduced a parallel for-loop known as **forall()** and replaced the conventional for-loop within the application's solver. This optimization allows for more efficient workload distribution, fully harnessing the capabilities of the HPX backend. Our solution yields favorable performance scaling across multiple cores. Notably, as the input size increases, the performance of the HPX backend becomes comparable to, and in some cases even surpasses, that of the MPI backend.

This 1-dimensional structural dynamic system simulation project serves as an introductory case study for newcomers to the FleCSI library. It also marks the first time primary attempt to assess the performance of the HPX backend of FleCSI, offering a foundation for future enhancements and optimizations in parallel computing for structural dynamic simulations.

Keywords: Flecsi library, parallel computing, Euler's method, optimization, performance analysis, MPI backend, HPX backend.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Structural dynamics lies at the heart of numerous engineering applications, from earthquake engineering to aerospace design and mechanical systems analysis. These systems are often governed by second-order ordinary differential equations (ODEs), and the numerical integration methods used to solve them have played a crucial role in simulating their behavior. However, as computational demands grow and simulations become increasingly complex, there is a pressing need to explore parallel computing techniques to enhance solver efficiency and reduce computational time.

To address this challenge, high-performance computing techniques have become essential for efficiently addressing complex simulations. This project comprehensively examines the FleCSI (Flexible Computational Science Infrastructure) library and its role in implementing simulations for structural dynamic problems.

The FleCSI Library

The FleCSI library, developed by the Los Alamos National Laboratory (LANL), is a powerful tool designed to provide flexibility and adaptability to various computational science simulations. FleCSI offers a compelling framework for constructing highly parallelized applications, specifically focusing on managing data dependencies and orchestrating complex simulations.

FleCSI's data-driven design philosophy is instrumental in managing complex simulation data dependencies. This design allows for the explicit definition of data relationships between different simulation components, streamlining the parallel execution of tasks. By abstracting complex data-task interactions, FleCSI simplifies the expression and comprehension of parallelism, helping mitigate issues like data races and ensuring accurate and efficient execution within parallel and distributed computing environments.

Motivation

The motivation behind exploring FleCSI lies in the existence of challenging problems that demand innovative solutions. For example, consider using the Euler solver to simulate an arbitrary number of spring-mass structural dynamic systems. This scenario encompasses engineering, physics, and computational science, involving the simulation and analysis of complex systems' vibrational and dynamic behavior. FleCSI's adaptability and multi-backend support make it a valuable tool for addressing such intricate problems efficiently.

Our investigation will employ a 1D spring-mass system problem as a representative case study. This problem, while conceptually simple, introduces data dependency challenges that are emblematic of more complex structural dynamic simulations. By applying FleCSI and numerical integration methods like Euler to this specific problem, we aim to comprehensively understand the FleCSI library and how it can be leveraged to advance computational science. We will introduce key FleCSI concepts, architecture, and backend support, emphasizing its role in addressing data dependencies and orchestrating the parallel execution of tasks. Furthermore, we will delve into performance and scalability aspects to gain a holistic perspective on FleCSI's potential in the realm of computational science.

Primary Exploration: FleCSI and Its Backend Workings

This thesis embarks on a primary exploration of the FleCSI library, focusing on the role of different backends in enhancing its capabilities. We will elucidate the core concepts of FleCSI, its architecture, and the intricacies of its backend support, shedding light on how it enables the development of simulations across diverse scientific domains.

Performance Testing with Different Backends

In addition to understanding the fundamental concepts of FleCSI, this project also aims to assess the performance of simulations using different backends, including HPX and MPI.

Performance metrics such as execution time, speedup, and scalability will be carefully evaluated to provide insights into each backend's comparative advantages and trade-offs. This performance testing will be a critical component of our exploration, guiding researchers and practitioners in selecting the most suitable backend for their specific computational needs.

Through this research, we aspire to empower readers with a profound appreciation for the value of FleCSI and its ability to drive advancements in computational science, ultimately facilitating solutions to complex, cross-disciplinary challenges.

Report Structure

This report is structured as follows: In Chapter 2, we delve into the literature on structural dynamics systems and provide an overview of the backends of the FleCSI library. Chapter 3 introduces the programming model and the main components of FleCSI, offering insights into how FleCSI is implemented. Chapter 4 outlines the methodology, including the problem statement for our structural dynamic system. Chapter 5 presents the implementation code for simulating this system using FleCSI. Lastly, in Section 6, we detail our experimental setup and benchmarks, share our key findings, and suggest avenues for future research.

Chapter 2

Literature Review

In this chapter, we present a comprehensive literature review that lays the foundation for our project's objectives. We delve into three critical areas of study: the 1D mass-spring system, the FleCSI library, and its backends HPX and MPI.

2.1 1D Mass-Spring System

2.1.1 Mechanical Dynamics of Mass-Spring Systems

The 1D mass-spring system represents a fundamental mechanical system widely studied in the realm of classical mechanics and engineering. One of the cornerstones of this system is Hooke's law, formulated by Robert Hooke in 1678. Hooke's law describes the linear relationship between the force applied to a spring and the resulting displacement [1]. This law serves as the basis for modeling the behavior of linear springs in mass-spring systems.

2.1.2 Numerical Simulation of Mass-Spring Systems

In the realm of computational mechanics, researchers have extensively explored numerical methods to simulate the dynamics of mass-spring systems. These methods include Euler's method, Runge-Kutta methods, and finite element methods [2]. For instance, Euler's method discretizes time and approximates the solution to the equations of motion, making it a valuable tool for modeling and analyzing the behavior of mass-spring systems. Such numerical techniques are pivotal for achieving accurate predictions of mass-spring system behavior, particularly in practical engineering applications.

2.2 FleCSI Library overviews

2.2.1 Introduction to FleCSI

The FleCSI (Flexible Computational Science Infrastructure) library has gained prominence as a versatile computational framework tailored to meet the multifaceted challenges of scientific simulations. FleCSI's modular and flexible architecture empowers researchers to compose simulations by assembling reusable components [3]. Its adaptability makes it a compelling choice across various scientific domains. The applicability of the FleCSI library extends across a spectrum of scientific domains.

2.3 FleCSI's Backends: HPX and MPI

2.3.1 HPX (High-Performance ParalleX)

HPX is a C++-integrated runtime system that excels as an Asynchronous Many Task (AMT) solution. HPX offers a C++ standard API, complying with standard C++. Its commitment to aligning with C++ standardization proposals ensures a uniform interface, especially in parallelism and concurrency, and extends support for distributed and heterogeneous computing.

Many facilities empower HPX parallelism. Such as Threads (thread), Futures (hpx::future), Asynchronous Tasks (hpx::async), Synchronization primitives(whenall), and so on.

HPX lightweight thread: HPX's lightweight threading system quickly switches user-level threads, reducing overhead. Very small overheads empower the program to be divided into smaller tasks. The minimized overhead, including thread creation, scheduling, and execution time, supports the scheduling of numerous tasks with minimal performance impact [4]. HPX's lightweight threading system, when coupled with upcoming features, offers a significant advantage: it makes auto-parallelization highly efficient. This means that it allows us to directly express the dependency graph as a runtime-generated execution tree, streamlining the entire parallel processing process.

HPX future: serves as a result that has not been computed yet. The benefit from 'future'

[5, 6]:

- Enables transparent synchronization with the producer
- effectively concealing the intricacies of thread management
- By encapsulating data dependencies(future represents a data dependency), it renders asynchrony more manageable.
- Enabling the coordination of asynchronous execution across multiple tasks.
- Turns concurrency into parallelism

Task Continuation: HPX offers additional support for task-based programming alongside its lightweight thread and hpx::future features, including task continuation. By employing hpx::future, one can attach continuations to asynchronous operations instead of waiting for the result. This approach eradicates the need for blocking waits (such as .get()) and saves resources that would otherwise be spent on thread polling. Consequently, it greatly enhances program responsiveness. Additionally, one can create chains of hpx::future instances by linking one to another, establishing an implicit dependency graph through continuations. This not only optimizes program performance but also streamlines asynchronous task handling.

Therefore, with advanced parallelism facilities. HPX is an essential backend of FleCSI library, stands as a task-based parallel programming framework designed for scalability and efficiency [7]. It facilitates asynchronous task execution, distributed computing, and high-performance parallelism, rendering it apt for simulations characterized by irregularity and dynamism.

2.3.2 How HPX backend of flecsi works

When invoking the magic parallel function of flecsi:

```
1    flecsi::execute< yourtask >
```

, flecsi undertakes a sequence of actions to manage task execution and dependencies. What flecsi does: it has some code that iterates all the arguments and invokes the backend for each of the arguments. at that point, the backend can decide what to do with each argument.

The HPX backend is responsible for examining annotations associated with these arguments and making decisions based on their characteristics. For instance, if an argument is labeled as “read-only,” the HPX backend establishes a read-only dependency for that argument. These arguments typically reference FLECSI fields.

The HPX backend introduces ”futures” to this process. Each field is associated with a future, which becomes “ready” when the field is updated. When dependencies are established, the HPX backend attaches continuations to these futures. Subsequently, when the task runs and makes the associated ‘hpx::future’ ready, it triggers the execution of continuations for tasks that depend on it. This mechanism enables the proper sequencing of tasks, as the backend keeps track of implicit dependencies, such as when one task writes to a field and another reads from it.

To illustrate, when the task is executed, it first writes to the relevant field. Once this operation is completed, it signals the associated future to become ready, triggering the execution of continuations for dependent tasks. The task execution order is meticulously orchestrated by examining annotations and defining the order in which tasks must run. Tasks dependent on a specific field run as continuations on the ‘hpx::future’ that become ready when the task updates the field. In essence, the HPX backend reconstructs the dependency graph using the annotations associated with the parameters of tasks. It leverages the concept of ”futures” to create a precise task execution sequence and maintain the correct order.

Therefore, all the HPX backend does is launch tasks and run them in parallel whenever possible. It constructs the dependency tree of these tasks by analyzing the attributes of the arguments in the FLECSI task. Specifically, it examines the annotations, such as ‘read-only’ or ‘write-only,’ to determine the nature of the dependency. For instance, tasks that involve writing

must wait for the write to complete before reading.

This dependency information is expressed through continuations on futures. Each field is associated with its own future, and when the field is updated, the future becomes 'ready.' This readiness triggers all the associated dependencies automatically. In essence, the HPX backend utilizes these mechanisms to coordinate and optimize the execution of tasks, ensuring they run in the correct sequence.

In sum, when one uses flecsi::execute$< \ yourtask \ >$, FLECSI manages task execution and dependencies. The HPX backend examines argument annotations, such as 'read-only' or 'write-only,' to establish dependencies. It uses 'futures' associated with fields to coordinate tasks, triggering continuations when the tasks are ready. This mechanism ensures the correct sequence of tasks. The HPX backend optimizes task execution by running tasks in parallel whenever possible and reconstructs the task dependency tree based on argument attributes. Implicit dependencies, like write-then-read, are automatically managed. In essence, it automates task scheduling and execution while maintaining proper sequencing.

2.2.3 MPI (Message Passing Interface)

MPI, or the Message Passing Interface, serves as a ubiquitous standard for message-passing parallelism in scientific computing [8]. Renowned for its adeptness in distributed memory parallel computing, MPI finds extensive usage in high-performance computing (HPC) systems and clusters.

2.2.4 Parallelization in MPI and HPX backend of FleCSI

FleCSI is empowered with different backends for high-performance computing.

MPI bakend: FleCSI employs the MPI backend with a one-rank-per-core approach. In this configuration, parallelization happens across ranks instead of within a single rank. To illustrate, FLECSI first split the array into partitions with workload balance, distributing them across the various ranks so that each rank has its own amount of work and communicates behind the scenes to do the boundary exchanges.

HPX bakend:

The HPX backend employs a parallelization strategy by running multiple threads within a single rank. Usually, HPX is designed to facilitate parallel execution, encompassing both across ranks(distributing) and inside the locality.

Parallelization within a locality benefits applications that inherently expose parallelism. Because FleCSI constructs a task dependency tree, while HPX schedules tasks based on these dependencies. However, parallelism is limited if the task tree lacks width and is linear, leading to increased overhead without significant gains. The linear tasks means don't get any parallelization because the dependencies do not allow that. That means, on the one hand, applications don't get any parallelization from that but pay for the overhead that the hpx backend introduced. Additionally, hpx tasks have a certain overhead associated with them, approximately in the order of milliseconds per task. That means if there is less work inside the task, then applications have to pay more overhead cost than gain benefit from parallelism.

Therefore, two key factors for HPX to be efficient:

- **Abundant Local Parallelism**:

  HPX performs most efficiently when the application inherently exposes substantial local parallelism. This means having a wide dependency tree where multiple tasks depend on a single task. In this scenario, HPX can spawn and run all the dependent tasks concurrently. Hence, the first requirement for efficiency is a wealth of local parallelism.

- **Sufficiently Workload Within Tasks**:

  HPX tasks come with some inherent overhead. For instance, a simple addition operation executed as a FLECSI task generates an HPX thread with just that operation. This results in paying a tremendous amount of overhead, which is not there in the MPI case, because MPI directly executes that function without spawning a new thread.

  In contrast, in the MPI context, FleCSI tasks are executed in line without any additional scheduling. So don't pay any overhead for small fleCSI tasks, but pay a lot of overhead for

small fleCSI tasks in the HPX back end. Therefore, the second condition for efficiency is to ensure tasks are substantial enough to offset the overhead introduced by HPX.

In summary, HPX operates most efficiently when your application exhibits ample local parallelism and contains tasks with workloads large enough to compensate for the inherent overhead. However, in situations with trivial applications, the MPI backend may outperform HPX due to the absence of this overhead.

Therefore, for performance analysis of the application, focus on two key criteria:

1. How much parallelism does it expose: What is the dependency of those tasks, and how wide is it. Is it just one task wide, or is it ten tasks wide or 100 tasks wide
2. And the second thing is how much work is inside each of the dependency tasks.

This assessment will help determine how effectively HPX performs within the application based on these criteria.

2.4 Gap Analysis: Identifying Research Objectives

While existing literature provides invaluable insights into simulation, the FleCSI library, HPX, and MPI, a notable gap emerges in the context of harmoniously integrating these components.

In this section, we outline the specific research objectives and gaps that motivate our project. These objectives are driven by the need to integrate the HPX backend into FleCSI, enhance its performance, and address the lack of a 1D simulation problem application within the FleCSI framework.

2.4.1 Integration of HPX Backend

Objective 1: The integration of the HPX backend into the FleCSI library represents a crucial step in expanding the library's capabilities. Historically, FleCSI lacked a dedicated HPX backend,

which limited its potential for high-performance parallelism and distributed computing. Our objective is to seamlessly integrate the HPX backend into FleCSI, enabling researchers to leverage its features for enhanced parallelism and improved performance.

Rationale: By integrating HPX into FleCSI, we empower users with a powerful backend that excels in asynchronous task execution and distributed computing. This integration fills a critical gap in the FleCSI ecosystem, opening the door to a broader range of applications and computational challenges.

2.4.2 Bridging the Gap: 1D Spring-Mass Simulation

Objective 2: Another significant gap in the FleCSI library is the absence of a 1D spring-mass simulation problem application. While FleCSI is a versatile framework, it currently lacks a straightforward example that introduces new users to the world of 1D simulations. Our objective is to develop a comprehensive 1D spring-mass simulation using FleCSI, serving as a valuable entry point for newcomers to the library.

Rationale: Providing a 1D spring-mass simulation within the FleCSI framework not only addresses the lack of a basic example but also offers new users a practical and intuitive introduction to the library's capabilities. This example will serve as a valuable resource for researchers and students looking to embark on simulation work using FleCSI.

2.4.3 Advancing the FleCSI Ecosystem

By achieving these objectives, we contribute to the advancement of the FleCSI ecosystem. The integration of the HPX backend enhances the library's parallel computing capabilities, making it a more competitive choice for high-performance scientific simulations. Simultaneously, the development of a 1D spring-mass simulation within FleCSI expands its applicability and accessibility to a broader user base, fostering growth and innovation within the community.

In the following chapters, we will detail our approach to accomplishing these objectives and demonstrate how they align with our project's overarching goals.

Chapter 3

Programming model and main components of FleCSI library

In this chapter, we embark on a comprehensive journey into the core of the FleCSI (Flexible Computational Science Infrastructure) library. At the heart of FleCSI lies a modular architecture that reshapes the landscape of scientific software development. This architecture promotes code reusability, organization, and flexibility, allowing simulations to comprise a collection of interchangeable components. We explore the FleCSI library's programming model architecture, delve into the key components(data, execution, and control models) and some important concepts(ghost copy, boundary layers), and explore how they work together within the context of scientific simulations.

3.1 Runtime Model

The FleCSI runtime model offers a simple interface to inquire about the system's status. This includes details on runtime initialization like the total threads, number of processes, and threads per process. It also provides task-specific information, such as the number of colors and the color of the current point task when executing.

3.2 Control Model: Orchestrating Simulation Execution

Among the key components of the FleCSI library, the Control Model plays a central role in orchestrating the execution of simulations. It governs task sequencing, coordination, and synchronization, ensuring coherent simulation flow. The control model enables users to define application structure with a control-flow graph (CFG). Each control point hosts a directed acyclic graph (DAG) of actions, where actions often trigger tasks, each operating on a specific data subset or "color". Task launches produce concurrent point task instances for each color.

In particular, control Points are static markers, defining a control-flow graph (CFG) that may contain cycles. Actions, which are C++ functions executed within control points, have

developer-defined dependencies, shaping a directed acyclic graph (DAG). Additionally, tasks are C++ functions launched by actions for distributed data interaction. FleCSI dynamically performs topological sorting of these actions within each control point during runtime, thereby establishing a sequential program order.

This model substitutes the conventional hardcoded execution structure of an application with a clearly defined and extensible mechanism, offering easy verification and visualization.

3.2.1 Visualization of Control model

FleCSI provides the command-line option "–control-model," which enables users to generate dot files for visualizing both the control model and the sequential ordering of registered actions, as shown in Figure 3.1.

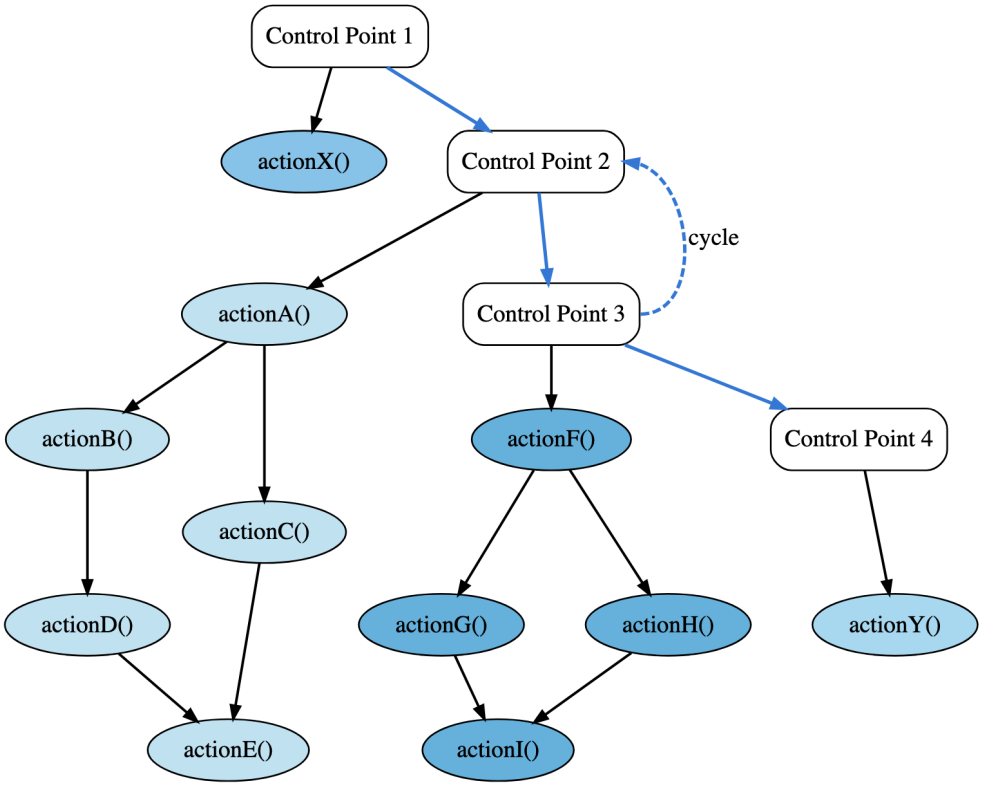

Figure 3.1: Flecsi Control Model [9]

The benefit of this model is that it enables users to easily add new actions with dependencies to the model's DAGs without altering upstream code. Figure 3.2 demonstrates the integration of

a new node, action N(), under Control Point 2. FleCSI's data model ensures seamless integration, promoting extensibility.

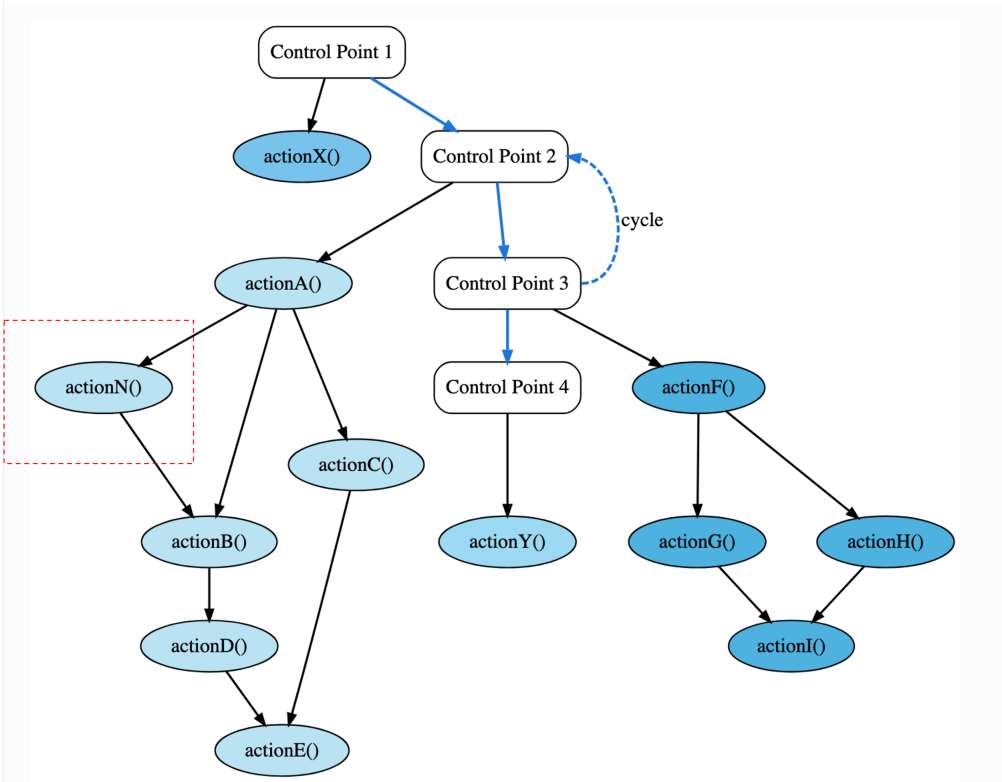

Figure 3.2: Flecsi Control Model After adding extra action

3.2.2 Advantages of Control model

- Control Flow Management

The Control Model within FleCSI encapsulates the control flow of a simulation. It defines the sequence of control points, actions, and tasks required to advance the simulation in a coherent manner. This capability is essential for simulations with complex dependencies or those that involve multiple interacting components.

- Task Scheduling and Coordination

Enables asynchronous task execution, optimizing performance.

Underpinning the Control Model is the ability to schedule and coordinate tasks. FleCSI's

task-based parallelism relies on asynchronous execution, allowing tasks to be scheduled and executed independently. The Control Model ensures that tasks are executed in a logical sequence while taking advantage of parallelism to enhance performance.

- Synchronization and Data Dependencies

Manages data dependencies and maintains consistency.
In simulations, synchronization is often necessary to satisfy data dependencies. The Control Model manages synchronization points, ensuring that tasks that rely on shared data or resources are executed in the correct order. This helps maintain data consistency and accuracy throughout the simulation.

- Integration with Other Components

The Control Model seamlessly integrates with other key components of FleCSI, such as the Mesh Topology, Data Management, and Specialization components. This integration ensures that simulation execution is tightly aligned with data access and manipulation, enhancing overall efficiency.

- Facilitating Complex Simulations

Abstracts task scheduling intricacies, facilitating focus on scientific aspects.
In scientific simulations, complexity is often inherent. The Control Model empowers researchers to manage and navigate this complexity efficiently. It abstracts the intricacies of task scheduling and synchronization, allowing users to focus on the scientific aspects of their simulations.

3.3 Data Model

Defining topology instances and fields:

FleCSI offers a data model that seamlessly integrates with task and kernel abstractions, facilitating straightforward registration and access to diverse data types while automatically

tracking dependencies. Since tasks can run in various memory spaces, the runtime handles data management, ensuring copies are available in the relevant memory space and changes are propagated to subsequent tasks that require it [9].

The FleCSI library's data model is a fundamental aspect of its design, providing a structured way to manage and interact with data within FleCSI-based applications. It serves as a framework for organizing and accessing various data types while also automatically tracking dependencies.

Key components and concepts of the FleCSI library's data model include:

- Fields: Fields are the primary data containers in FleCSI, representing various data types such as scalars, arrays, or complex data structures. These fields can be declared and accessed within the program to store and manipulate data.

- Accessors: Accessors are used to interact with fields, providing a way to read from or write to them. Accessors are equipped with privileges, such as read-only or read-write, which dictate how data consistency is maintained.

- Memory Management: As FleCSI allows tasks to execute in various memory spaces, it manages data efficiently by copying it to the relevant memory space when needed. This ensures that data is accessible and up-to-date for tasks executing in different contexts.

- Header Declarations: When using fields in header files, it's essential to declare them as inline const to adhere to the One-Definition Rule (ODR), preventing potential issues with multiple definitions of the same data.

```
1 // field definition
2 inline const field<double>::definition<topology, topology::entities> myVar;
```

Listing 3.1: field define

Overall, the FleCSI library's data model is designed to simplify the management of data within parallel and distributed applications, offering a structured and efficient way to work with data while ensuring data consistency and proper parallel execution.

3.4 Execution Model

Launching tasks and kernels:

FleCSI employs “Tasks” and “Kernels” mechanisms to express work: Tasks operate on distributed data, ensuring memory consistency via data privileges. Kernels work on data within a single address space and require explicit barriers for consistency (relaxed-consistency memory model).

Task Execution example :

Task-based parallelism lies at the heart of FleCSI’s execution model. Here, tasks are created, scheduled, and executed asynchronously. This approach optimizes performance, promotes load balancing, and adapts to various computing architectures.

List 3.1 demonstrates utilizing the execute method to trigger a task(line 13). **Triggering tasks using the execute function is where the real FleCSI magic happens to implement parallelism.**

```
1  // field definition
2  const field<double>::definition<topology, topology::entities> myVar;
3  // task definition
4  void
5  project_namespace::
6      task::initdata(topology::accessor<ro> tp,
7                     field<double>::accessor<rw>f){
8          forall(data, tp.entities()) {
9          f(data) = 0.0; // dummy initialization
10     }
11 // task execution
12 void
13 exec_init() {
14     execute<task::initdata, /*accelerator*/ >(topo, values(topo));
15 }
```

Listing 3.2: Task execution example

For this example, several notable points can be learned:

- Using the **execute** function to trigger the tasks.
- Passing accessors' privileges to the task:

  Accessors' privileges, like "rw" in line 7, dictate the memory consistency operations executed on the data when the next task is invoked. Available privileges include "na" (no access), "ro" (read-only), "wo" (write-only), and "rw" (read-write). "na" privilege can be employed to delay consistency updates.

3.5 Topologies of FleCSI

The following sections provide important concepts and also offer a starting point for simulation in flecsi specialization.

A topology is a distributed-memory entity that holds user-registered fields across one or more index spaces. It can also retain structural details needed to interpret those fields in a context relevant to the topology category, like a structured mesh. Users can create multiple instances of any topology as needed.

3.5.1 Specialization

Customize data structure(mesh, narray), interface, coloring method, or runtime behaviors to align with the specific demands of the application. FleCSI's specialization involves defining each topology type as a customized version of the core topology types provided by FleCSI. These customizations specify various properties, including the mesh's dimensionality and which types of connectivity information should be explicitly stored.

3.5.2 Index Space

Define the index Space in specialization:

In FleCSI, index spaces are crucial in defining field arrays that store user data. To simplify, consider an index space as a way to list an array's elements. Indeed, within FleCSI, index spaces represent the logical components or entities across all of our topology types.

3.5.3 Coloring

Define the coloring method in specialization: A coloring defines how the indices within an index space should be divided or partitioned. In FleCSI, colorings define how to partition indices within an index space, without implying size or being linked to any specific execution space. This differs from the static rank-to-process mapping in MPI.

3.5.4 Domain

The domain enumeration classifies the types of partition entities that can be requested from a topology specialization created using this particular type.

These domains in Figure 3.3 are crucial in various interface methods, offering details about an axis, including size, extents, and offsets.

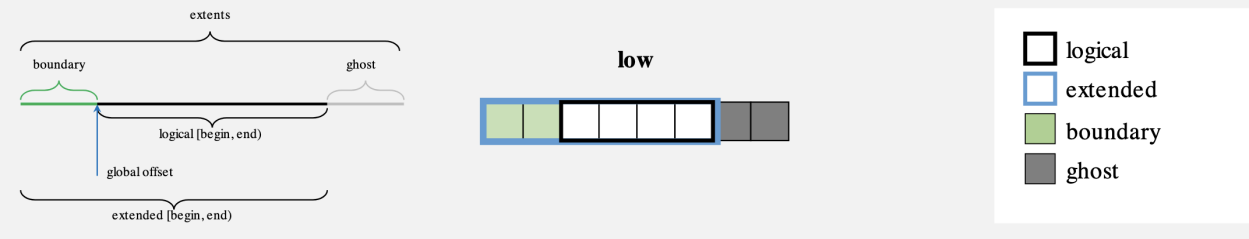

Figure 3.3: Layouts for one possible orientation

Table 1 shows the details for each domain of FleCSI library.

As determined by the coloring algorithm, the domain enumeration in a mesh part represents particular characteristics. And specify the ghost layer and boundary layer in the coloring method. When the boundary layer is integrated into the coloring function, the zero-based logical index space transitions to an n-based index space, with 'n' indicating the depth of your customized boundary.

Table 1: Annotations for each domain

Enumerator	
logical	the logical, i.e., the owned part of the axis
extended	the boundary padding along with the logical part
all	the ghost padding along with the logical part
boundary_low	the boundary padding on the lower bound of the axis
boundary_high	the boundary padding on the upper bound of the axis
ghost_low	the ghost padding on the lower bound of the axis
ghost_high	the ghost padding on the upper bound of the axis
global	global info about the mesh, the meaning depends on what is being queried

3.6 Ghost Layer and Boundary Layer in FleCSI

In the context of the FleCSI library and parallel computing, the terms ”ghost layer” and ”boundary layer” refer to specialized layers of data that are used to manage communication and data exchange between adjacent domains or partitions in a parallel simulation. These layers play a crucial role in ensuring that data dependencies and interactions are correctly handled across distributed memory systems.

Ghost Layer:

- Definition: The ghost layer, also known as the ghost zone or halo region, represents a layer of data that extends beyond the boundary of a local domain or partition in a parallel simulation.
- Purpose: The ghost layer is used to store data associated with mesh elements or other computational entities that are not entirely contained within a single domain. These entities may have interactions with neighboring domains, and the ghost layer facilitates communication and synchronization between adjacent domains.
- Communication: During simulation, data in the ghost layer is exchanged or ”ghosted” between neighboring domains to ensure that each domain has access to the data it requires for computation. This communication is typically performed using message-passing libraries like MPI (Message Passing Interface) or HPX (High-Performance ParalleX).

- Efficiency: Careful management of the ghost layer is essential for minimizing communication overhead in parallel simulations. Techniques like ghost layer aggregation and optimizing data transfer can enhance simulation efficiency.

Boundary Layer:

- Definition: The boundary layer is a term that is sometimes used interchangeably with the ghost layer. It refers to the portion of the ghost layer that is closest to the boundary of a domain.

- Purpose: The boundary layer specifically focuses on the part of the ghost layer that is adjacent to the domain's true boundary. It contains data points or entities that are near the domain boundary and are involved in interactions with entities from neighboring domains.

- Local Computations: The boundary layer is typically involved in local computations, and its data is used in conjunction with data from the local domain to ensure consistency in the simulation results.

Overall, ghost layers and boundary layers are essential elements in parallel simulations to facilitate inter-domain communication, enforce boundary conditions, and enable accurate modeling of physical phenomena near domain boundaries.

Chapter 4

Methodology

In this chapter, we outline the problem statement for our project, which revolves around simulating a 1D spring-mass system and addressing the challenges associated with it. We will introduce the structure of the 1D Spring-Mass system, discuss the governing equations, explain our choice of the numerical integration method, and highlight the data dependency issues that need to be considered during implementation in parallel later. In order to improve simulation efficiency, we implement it using flecsi library and will compare the performance of different backends(MPI, HPX).

4.1 Theoretical Backgrounds

A second-order hyperbolic system of ordinary differential equations typically characterizes structural dynamics problems. In our study, we choose to use a simplified 1D spring-mass system as our experimental case. This choice allows us to streamline our focus, bypassing the complexities of intricate physics. Instead, we can concentrate primarily on implementing parallel simulation techniques for solving structural dynamics problems.

4.1.1 Spring-Mass Systems

1D spring-mass systems are fundamental mechanical systems that consist of an arbitrary number of masses attached to springs. The behavior of these systems has been extensively studied in classical mechanics. Hooke's law describes the relationship between the force applied to a spring and its resulting displacement. Key equations governing the motion of such systems, including Newton's second law and the equation of motion, are well-established [Newton, 1687]. To solve this problem, discretizes a complex problem into smaller, simpler elements, approximating solutions over the entire domain.

4.1.2 Structure of the 1D Spring-Mass System

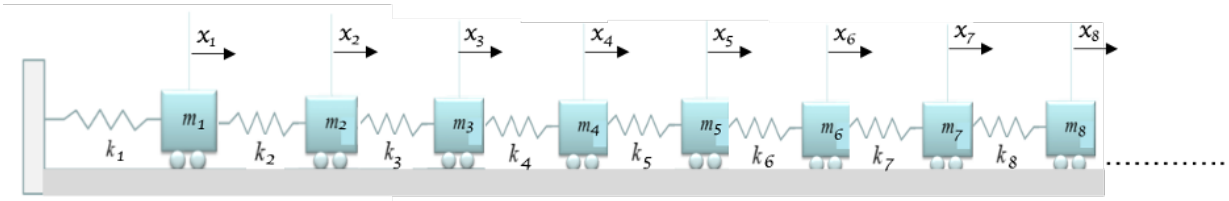

Figure 4.1: 1D Spring-Mass System

The structure of the 1D spring-mass system can be visualized in **Figure 4.1**: The components and their interactions within the system are clearly depicted in the diagram. It illustrates how the mass, spring, and external forces are interconnected.

4.1.3 Governing Second-Order Differential Equations (ODEs)

mathematical model:

The dynamics of a 1D undamped spring-mass system are described by second-order differential equations (ODEs) that model the behavior of the mass-spring system. The general form of the governing ODEs is:

$$\boldsymbol{m \cdot x''(t) + c \cdot x'(t) + k \cdot d(t) = F(t)} \tag{4.0.1}$$

Where:

- m represents the mass of the object.
- $x''(t)$ is the acceleration of the mass.
- c represents damping (optional).
- $x'(t)$ is the velocity of the mass.
- k represents the spring constant.
- $x(t)$ is the displacement of the mass.

- $F(t)$ represents any external forces applied.

These equations govern the motion of the mass and are central to our simulation. In our case, we actually simulate an undamped spring-mass system, which means $c = 0$. Therefore, the governing equation of the undamped spring-mass system should be:

$$\boldsymbol{m \cdot x''(t) + k \cdot d(t) = F(t)} \tag{4.0.2}$$

4.2 Numerical Integration Method: Euler's Method

Numerical integration, also known as numerical methods for ordinary differential equations (ODEs), is a fundamental technique in computational science and engineering. It plays a crucial role in simulating dynamic systems, including the behavior of 1D spring-mass systems. The primary purpose of numerical integration is to approximate the solution of ODEs when analytical solutions are not readily available or practical. In our project, we will explore using Euler's method for its simplicity and clarity.

4.2.1 Background of Numerical Integration

The need for numerical integration arises from the fact that many real-world problems involve complex differential equations that cannot be solved analytically. These differential equations describe the behavior of dynamic systems over time. In engineering, physics, biology, and various scientific disciplines, numerical integration methods are employed to obtain numerical solutions to these equations.

One of the earliest and simplest numerical integration methods is Euler's method, named after the Swiss mathematician Leonhard Euler. Euler's method is based on the idea of approximating the solution to an ODE by taking discrete time steps and updating the state of the system incrementally. This method is conceptually straightforward and provides a clear way to understand the dynamics of a system over time.

4.2.2 Euler's Method

Euler's method involves two primary steps for updating the state variables (e.g., position and velocity) of a dynamic system:

Displacement Update:

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \Delta t \cdot \boldsymbol{x'}_n \tag{4.0.3}$$

Velocity Update:

$$\boldsymbol{x'}_{n+1} = \boldsymbol{x'}_n + \Delta t \cdot \boldsymbol{x''}_n \tag{4.0.4}$$

Where:

- x_n and $x_n^{'}$ represent the displacement and velocity at the time step n;
- Δt is the time step size.

Euler's method is computationally straightforward and provides an explicit update scheme. Euler's method allows us to approximate the system's state at each time step, which is crucial for simulating dynamic systems.

Here are the steps for applying Euler's Method in a structural dynamic system:

1. Define the problem:

   Clearly define the structural dynamic system you want to analyze. This includes specifying the physical properties of the structure, such as mass, stiffness, and damping, as well as the external forces or loads acting on it.

2. Formulate the governing ODEs:

   Write down the governing differential equations that describe the motion of the structure. These equations typically involve second-order ODEs, which relate the displacements, velocities, and accelerations of the structural components.

3. Discretize the time domain:

   Choose a time step (Δt) for the numerical integration. This step size will determine the accuracy and computational efficiency of the analysis. Smaller time steps provide more accurate results but require more computational resources.

4. Initialize the system: Set the initial conditions for the system. This includes specifying the initial displacements and velocities of the structural components.

5. Time-stepping loop:

   Implement a loop that iterates through the time steps from the initial time (t = 0) to the final time (t = T), where T is the total duration of the analysis.

6. Update velocities and displacements:

   Within each iteration of the time-stepping loop, calculate the velocities and displacements of the structural components at the current time step based on the governing ODEs. Use Euler's Method to perform the updates:

7. Update the displacement and velocity at time t using equations 4.0.3, 4.0.4.

8. Store or analyze the results:

Recording the time history of displacements, velocities, and accelerations for further analysis. This data can be used to study the dynamic response of the structure to different loading scenarios or to extract key performance indicators, such as natural frequencies and mode shapes.

Euler's method is computationally straightforward and provides an explicit update scheme. Euler's method allows us to approximate the system's state at each time step, which is crucial for simulating dynamic systems.

However, it has limitations, especially regarding accuracy and stability. It may introduce errors in simulations with highly oscillatory behavior or stiff systems.

For more accurate simulations, higher-order numerical integration methods like the Runge-Kutta methods [10] or implicit methods such as the Backward Euler method are often preferred. These methods can mitigate some of the issues associated with Euler's method, but they are computationally more demanding.

In our project, we will explore the use of Euler's method for its simplicity and clarity, but we will also consider the potential need for more sophisticated numerical integration techniques, especially when dealing with structural dynamics problems that demand high accuracy and stability. The choice of integration method will depend on the specific requirements of our simulation and the trade-offs between accuracy and computational efficiency.

4.3 Data Dependency Issues

In the context of simulating dynamic systems, data dependencies arise due to the interdependence of variables at different time steps. The solution at time step n+1 depends on the values at time step n. Also, in the 1D spring-mass system, at each time step, we need to update one element using its left and right neighbors' values. This complicated data dependency makes parallelization and efficient computation challenging.

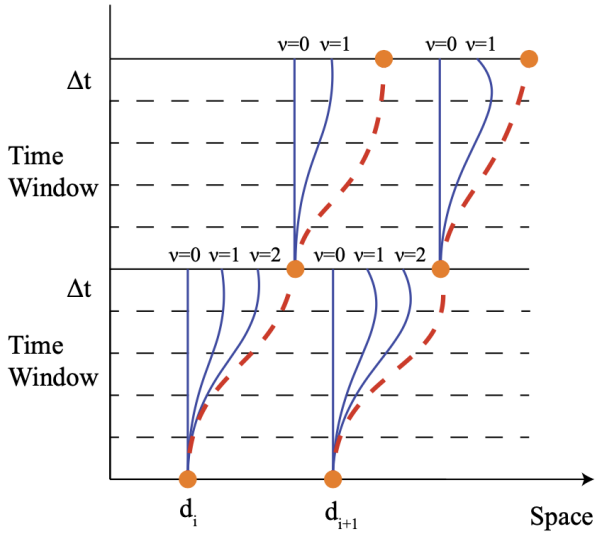

Figure 4.2: Space decomposition and iterations over the time windows

Pasetto and et.al research illustrated the dependency in Figure 4.2 [11]: the Space decomposition and iterations over the time windows. The continuous lines represent the solutions obtained for each iteration while the dashed lines are the converged solutions; the dot points represent the values at the end of each time window, taken as initial conditions for the subsequent one.

4.4 Solution

To address data dependency issues, we need a computational framework that can manage dependencies and efficiently execute simulations in parallel. In the next chapter, we will introduce the FleCSI library and how to implement this simulation using flecsi, and compare the performance of its associated backends, HPX and MPI.

In summary, our project focuses on simulating a 1D spring-mass system with a clear understanding of its theoretical background, structure, governing equations, the numerical integration method (Euler's method), and the data dependency issues that need to be resolved. And learn how to implement this simulation using flecsi and compare the performance of different backends of flecsi library. The following chapter will introduce the FleCSI library and its backends as key components of our solution.

Chapter 5

Implementations

In this chapter, we have presented the implementation details of our 1D spring-mass system simulation using the FleCSI library. We harnessed FleCSI's modular components for efficient data management, and employed Euler's method for time integration. Challenges related to load balancing, data consistency, and performance optimization were addressed effectively.

5.1 Structure of Code

This section provides an overview of the structure of our simulation code, which is designed to simulate 1D spring-mass systems using the FleCSI library. Understanding the code organization to present how different components interact and contribute to the simulation.

As previously discussed in the Control Model section, a FleCSI application's control flow is structured around a hierarchy consisting of control points - actions - tasks. Figure 5.1 provides a visual representation of Springmass's overall control flow.

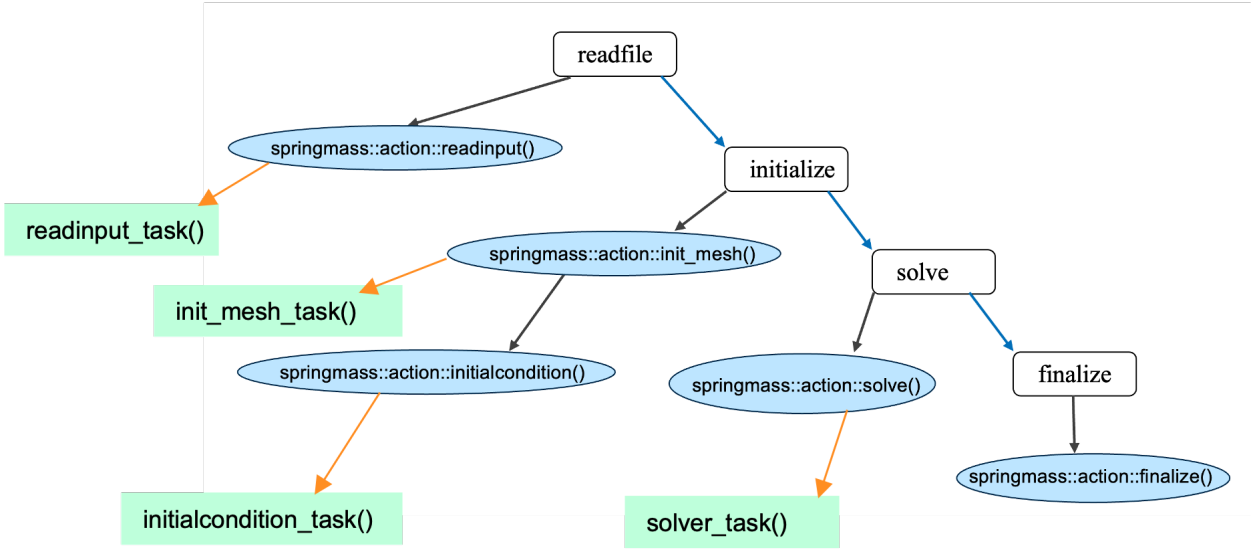

Figure 5.1: 1D spring-mass simulation control model

There are four control points: readfile, initialize, solve, and finalize, which will be executed in serial.

- The 'readfile' component aims to enhance the code's extensibility by enabling it to read an arbitrary number of elements, such as springs and masses, along with their associated variables from external files

- Within the 'initialize' section, two critical actions, 'init_mesh()' and 'initialcondition()'. Notably, 'init_mesh()' precedes 'initialcondition()' in execution order, as illustrated in Listings 5.1 and 5.2. These listings demonstrate how to design actions(L4 in Listings 5.1) and manage their dependencies effectively(L8 in Listings 5.2).

```
1 namespace springmass {
2 namespace action {
3     void init_mesh(control_policy &);
4     inline control::action<init_mesh, cp::initialize> init_mesh_action;
5     } // namespace action
6 } // namespace springmass
```

Listing 5.1: action design

```
1 // action dependencies
2 namespace springmass {
3 namespace action {
4
5 void initialcondition(control_policy &);
6
7 inline control::action<initialcondition, cp::initialize> initialcondition_act;
8 inline auto const initial_cond_dep = initialcondition_act.add(init_mesh_action);
9 // initial dependency, first init mesh, then initial_cond action.
10
11 } // namespace action
12 } // namespace springmass
```

Listing 5.2: action dependencies design

- solve section implements Euler's method, real distribution, and parallel task.
- In the end, finalize the whole code.

5.2 Specialization implementation

Before writing real Flecsi code, define the data structure(narray), index_space, dimension, coloring method, and interface in the specialization model to align with the specific demands of the spring-mass system application.

First, look at the simplified spring-mass system after discretization in Figure 5.2.

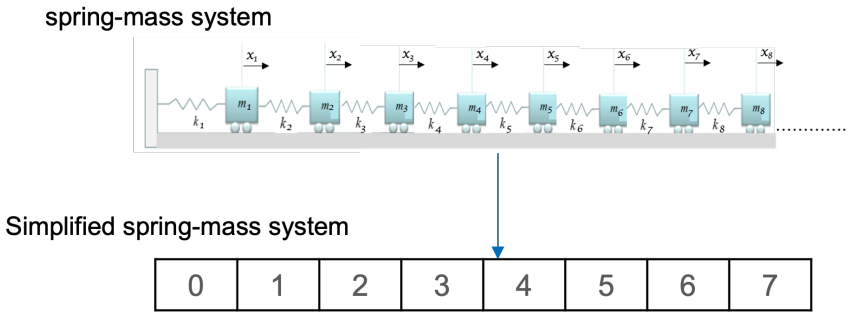

Figure 5.2: discretization 1D spring-mass system

Data structure: In our case, we employ the 'narray' data structure to represent the discretization of our 1D mesh topology. In this representation, smaller elements are connected at specific points referred to as 'vertices,' which, in our 1D scenario, are aligned exclusively along the x-axis. We employ four logical domains: 'all,' 'global,' 'boundary_low,' and 'boundary_high' to manage our simulation effectively.

coloring: Our partitioning method, called the 'coloring method,' defines how the system is divided into partitions. It returns a vector of axis definitions for each partition, enabling efficient data distribution.

See Listing 5.3: In line 36, we establish a 'layer ghost layer' for each vertex, and in Line 37, we introduce an additional 'boundary layer' to simplify the system and improve its manageability. In Figure 5.3, the grey blocks represent ghost copies for adjacent partitions, while the green blocks denote the boundary layers at both ends.

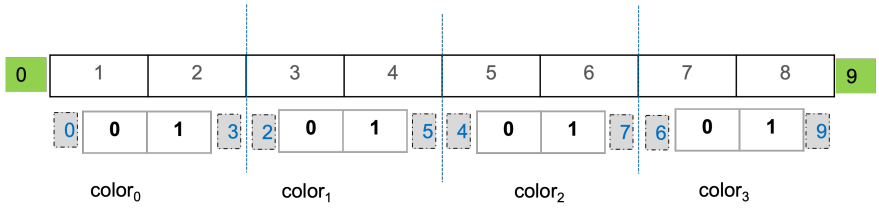

Figure 5.3: ghost copy and boundary layer

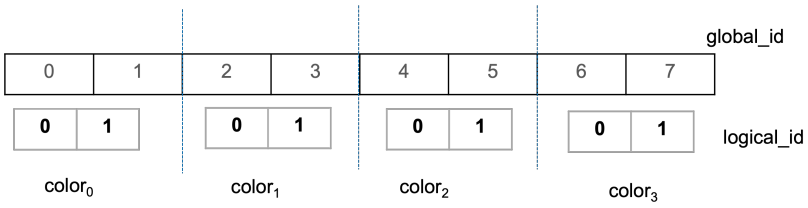

Figure 5.4: map partitions to ranks

Interface: In the interface, the global_id function is designed to provide the original global index of vertices across various domains.

On the other hand, the **vertices()** function returns distinct index ranges based on different domains. For instance, in Line 58 from Listing 5.3, when the domain is set to 'logical,' this function will provide the local index for each color or partition. For a visual understanding of this mapping process, please refer to Figure 5.4.

```
namespace springmass {
// Define a specialization for a 1D mesh
struct mesh : flecsi::topo::specialization<flecsi::topo::narray, mesh> {
  // Define the index space for the nodes
  enum index_space { vertices };
  using index_spaces = has<vertices>;

  // Define the domain and axis
  enum domain {logical,global, boundary_low, boundary_high};
  enum axis { x_axis };
  using axes = has<x_axis>;
```

```
12
13  // define the dimension
14  static constexpr std::size_t dimension = 1;
15  };
16
17  /*--------------------------------------------------------------------*
18    Color Method.
19   *--------------------------------------------------------------------*/
20  // Define a distributed color type
21   /*
22    Create a vector of axis definitions for the given extents and number of colors.
23    The method takes as input the distribution of colors over axes.
24    @param num_colors distribution of colors per axis
25    @param axis_extents indices number of entities
26
27    \return vector of axis definitions for each partition
28   */
29  static coloring color(std::size_t num_colors, gcoord axis_extents) {
30    index_definition idef;
31    idef.axes = flecsi::topo::narray_utils::make_axes(num_colors, axis_extents);
32    idef.axes[0].hdepth = 1; // add one layer of ghost
33    return {{idef}};
34  }
35
36  /*--------------------------------------------------------------------*
37    Interface
38   *--------------------------------------------------------------------*/
39
40   template<axis A>
41    FLECSI_INLINE_TARGET std::size_t global_id(std::size_t i) const {
42      return B::template global_id<mesh::vertices, A>(i);
43    }
```

```
44
45   template <axis A, domain DM = interior>
46    FLECSI_INLINE_TARGET auto vertices() const {
47      if constexpr (DM == interior) {
48        // The outermost layer is either ghosts or fixed boundaries:
49        return flecsi::topo::make_ids<mesh::vertices>(
50            flecsi::util::iota_view<flecsi::util::id>(
51                1,B::template size<mesh::vertices, A, base::domain::all>() - 1));
52      } else if constexpr (DM == logical) {
53        return B::template range<mesh::vertices, A, base::domain::logical>();
54      } else if constexpr (DM == all) {
55        return B::template range<mesh::vertices, A, base::domain::all>();
56      }else if constexpr (DM == boundary_low) {
57        return B::template range<mesh::vertices, A, base::domain::boundary_low>();
58      }else if constexpr (DM == boundary_high) {
59        return B::template range<mesh::vertices, A, base::domain::boundary_high>();
60      }
61    }
```

Listing 5.3: specialization for a 1D spring-mass system

5.3 Data variables

In our case, we work with four variables: displacement, velocity, mass, and stiffness, each associated with every vertex.

To represent these variables comprehensively, we utilize the fields ‘displacementsd’, ‘velocities’, ‘massesd’, and ‘stiffnessesd’. These fields encompass all displacement, velocity, mass, and stiffness values across all vertices.

5.4 Solver implementation and optimizations

5.4.1 initialcondition

At the outset, data read from the file is distributed and assigned to each field. It's crucial to note that we perform field initialization for each color or partition.

In Listing 5.4, Line 3 signifies the traversal of the logical domain of each partition. The function vertices() is employed to iterate through all the vertices in the local rank. Utilizing the global_id function, we correctly assign global values to their respective local counterparts. Furthermore, as part of the initialization process, boundary conditions are set to 0.0.

```
1  namespace springmass {
2      // initialize real elements on ranks
3      for(auto i: m.vertices<mesh::x_axis, mesh::logical>()) {
4        d[i] = displacements[m.global_id<mesh::x_axis>(i)];
5        v[i] = velocities[m.global_id<mesh::x_axis>(i)];
6        mas[i] = masses[m.global_id<mesh::x_axis>(i)];
7        k[i] = stiffnesses[m.global_id<mesh::x_axis>(i)];
8      } // for
9
10     // initialize boundary_left
11     for(auto l: m.vertices<mesh::x_axis, mesh::boundary_low>()) {
12       d[l] = 0.0 ;
13       v[l] = 0.0 ;
14       mas[l] = 0.0 ;
15       k[l] = 0.0 ;
16     } // for
17
18     // initialize boundary_right
19     for(auto h: m.vertices<mesh::x_axis, mesh::boundary_high>()) {
20       d[h] = 0.0 ;
21       v[h] = 0.0 ;
22       mas[h] = 0.0 ;
```

```
23        k[h] = 0.0 ;
24      } // for
25  } // initialize condition
```

Listing 5.4: initial condition

5.4.2 optimization: simplify Euler's solver

For original implement Euler's solver: In Figure 4.1, we illustrate the original 1D spring-mass system. The corresponding motion equations (ODEs) for each vertex are listed in Listing 5.5.

In the case of the leftmost mass-spring point, no left neighbor is involved, and the equation only pertains to the right neighbor (as indicated in Line 4 of Listing 5.5). Conversely, for the rightmost mass-spring point, there is no right neighbor involved, and the equation solely relates to the left neighbor (as specified in Line 7 of Listing 5.5). However, for the middle section, each vertex has both left and right neighbors involved in its equation.

We note that the middle, left, and right boundary vertices exhibit different data dependencies, resulting in distinct equations.

```
1  for (auto i : m.vertices<mesh::x_axis, mesh::logical>()) {
2      if (i == 0) { // leftmost mass
3  d[i] = d[i] + dt * v[i];
4  v[i] = v[i] + dt * (-((k[i] + k[i + 1]) * d[i] - k[i + 1] * d[i + 1])/ mas[i]);
5      } else if (i == size - 1) { // rightmost mass
6  d[i] = d[i] + dt * v[i];
7  v[i] = v[i] + dt * ((-k[i] * (d[i] - d[i-1])) /mas[i]);
8      } else { // middle masses
9  d[i] = d[i] + dt * v[i];
10 v[i] = v[i] + dt * ((-k[i] * (d[i] - d[i-1]) + k[i+1] * (d[i+1] - d[i])) /mas[i]);
11     }
12 }
13
```

Listing 5.5: Euler's solver

To simplify and generalize the spring-mass system model, Flecsi introduces a boundary layer. By adding boundary layers on both the left and right sides, which means each vertex has a left and right neighbor involved, we achieve consistency in the equations across all vertices.

```
for (auto i : m.vertices<mesh::x_axis, mesh::logical>()) {
    d[i] = d[i] + dt * v[i];
    v[i] = v[i] + dt * ((-k[i] * (d[i] - d[i-1]) + k[i+1] * (d[i+1] - d[i]))
    /mas[i]);
}
```

Listing 5.6: generalized Euler's solver

5.4.3 optimize Euler's solver equations

Let's analyze the generalized formula in Listing 5.6, assuming it runs on two ranks. The partition pattern is depicted in Figure 5.5:

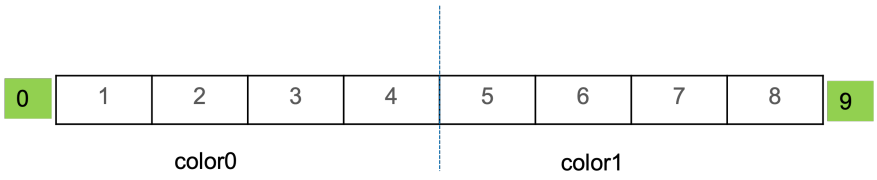

Figure 5.5: partition pattern

When no GPU is involved, we update the values of each vertex individually in a serial manner. As Listing 5.6 equations show, at time 't ', we update 'd[i]' using old displacement and old velocity. Then, update the velocities for each vertice. Since v[i] depends on d[i-1], d[i], and d[i+1], a ghost copy of (d[i+1])is needed before launching the kernel. On each rank, update 'v[i]' using the new displacement of the left neighbor, the new current displacement, and the old value of the right neighbor.

Regarding across ranks, for boundary vertices, such as 'update[5] ', we use a ghost copy of vertex 4(left neighbor), which could be updated and back to global that can be used, and an old value of vertex 6(right neighbor). This generally leads to a fixed mixed pattern of old and new data calculation patterns.

However, when GPUs are introduced, the execution on each rank is no longer serial. For instance, the GPU may compute vertex 2 first, we can get new displacement d2; Then it comes to velocities.

We update v[2] using the old displacement of the left neighbor (vertex 2) and the old displacement of the right neighbor(vertex 3);

Assume then update vertex 4 using the old value of vertex 3 and the old value of (ghost copy of) vertex 5, and subsequently update vertex 3 using the new values of vertex 2 and vertex 4. This scenario results in a different mixed pattern of old and new data, potentially leading to unpredictable behavior and results.

To address these issues and ensure consistency in data patterns, the solution is to split the displacement and velocity equations, as listing 5.6 shows, into two separate for loops. The final optimized Euler's solver in FleCSI is presented in Listing 5.7. In this manner, update the displacements of all vertices using their old displacements and velocities. Then, update the velocities using all the new displacements, regardless of the scenario.

```
// update displacement using old_displacement and old velocity
for (auto i : m.vertices<mesh::x_axis, mesh::logical>()) {
    d[i] = d[i] + dt * v[i];
}
// update velocity using new displacement;
// velocity depends on d[i-1], d[i] and d[i+1], a ghost copy
// is needed before launching the kernel
// (done by FleCSI before the invocation of this function).
for (auto i : m.vertices<mesh::x_axis, mesh::logical>()) {
    v[i] = v[i] + dt * ((-k[i] * (d[i] - d[i-1]) + k[i+1] * (d[i+1] - d[i]))
    /mas[i]);
```

```
12 }
```

Listing 5.7: optimized Euler's solver

5.4.4 optimize algorithms

To achieve high efficiency with the HPX backend of FleCSI, two key criteria can be considered:

Abundant Local Parallelism (Width of Task Dependency Tree): This criterion is determined by the width of the task dependency tree. In our case, the application does not naturally expose task parallelism. The tasks are strongly sequential, making it challenging to expand the task tree to improve parallelism.

One solution is to add more parallel workload inside the task by using parallel forall() algorithm replace the conventional for loops.

```
1 // update displacement using old_displacement and old velocity
2 forall (i , (m.vertices<mesh::x_axis, mesh::logical>()), "integrate_d") {
3     d[i] = d[i] + dt * v[i];
4 }
5 // update velocity using new displacement;
6 forll (i , (m.vertices<mesh::x_axis, mesh::logical>()), "integrate_v") {
7     v[i] = v[i] + dt * ((-k[i] * (d[i] - d[i-1]) + k[i+1] * (d[i+1] - d[i]))
8     /mas[i]);
9 }
```

Listing 5.8: optimized Euler's solver

5.5 Execute solver in parallel and distributed

After implementing the solver task, initiate the parallel and distributed computation at each time step by invoking the flecsi::execute$< task >$() parallelization function (Line2).

```
1 // Euler solver
```

```
void springmass::action::solve(control_policy &cp) {
    for (int t = 0; t < ndt; ++t) {
        flecsi::execute<task::solver> (cp.m, displacementsd(cp.m),
                                            velocitiesd(cp.m));
    }
  }

} // solver run
```

Listing 5.9: parallel and distributed execute Euler's solver

Chapter 6

Results and conclusions

In this section, verify the correctness of the simulation and then conduct a performance analysis between MPI and HPX backend.

6.1 Correctness verification

Consider an 8-mass-8-spring system as an example. Figure 6.1 illustrates the displacement changes in the rightmost mass. Where time step =0.01s and number of time steps = 8000.

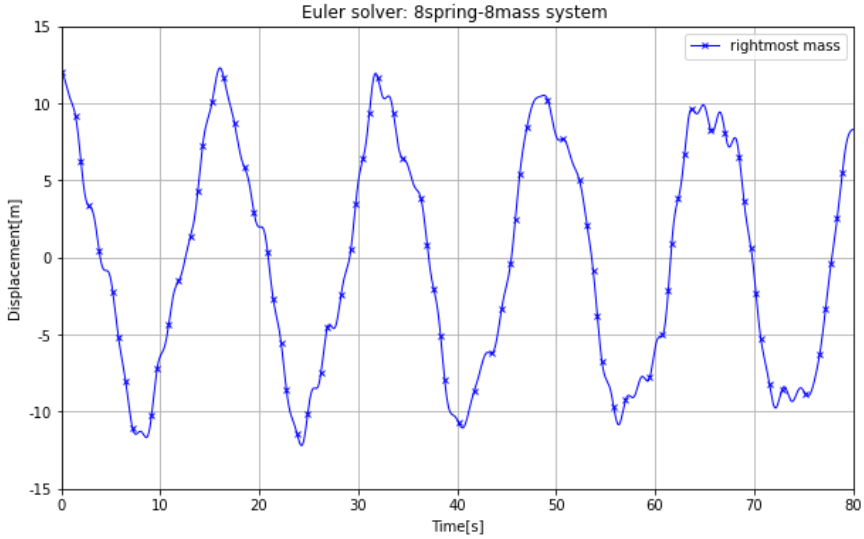

Figure 6.1: The vibration(displacement changes over time) of rightmost vertex

1. Simplified system

To verify the correctness of the simulation code, one approach is to simplify the complex system as much as possible.

In this process, we can transform, for example, an 8-mass-8-spring system into its simplest form, which is a 1-mass-1-spring system. To achieve this, we fix the positions of the first seven vertices by giving the greater value of mass and a very small stiffness value, effectively reducing the 8-spring-mass system to a single mass-spring system.

2. Verify the correctness of the simplified system-1mass-1spring system

The vibration shape of a 1D mass-spring system, which is undamped and consists of only one mass and one spring, will exhibit a simple harmonic motion. This motion involves the mass oscillating back and forth around its equilibrium position.

The equation describing the displacement 'x' of the mass as a function of time 't' in this system is typically given by:

$$\boldsymbol{x}(t) = A \cdot sin(2\pi ft + \phi) \tag{6.0.1}$$

Where:

- A is the amplitude of the oscillation.
- f is the frequency of the oscillation.
- ϕ is the phase angle, depending on the initial conditions of the system.

As a result, as depicted in Figure 6.2, our simplified system (which closely approximates a one-mass and one-spring system) also exhibits a perfect harmonic motion pattern.

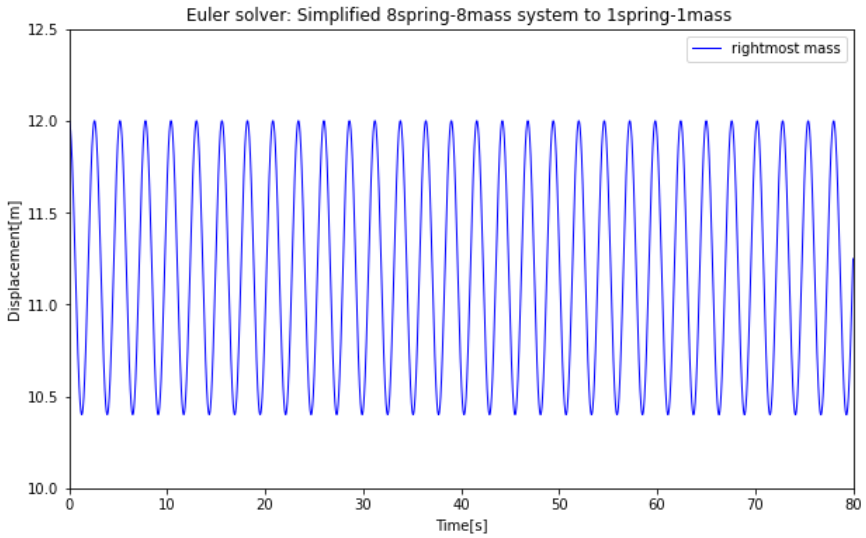

Figure 6.2: The vibration(displacement changes over time) of rightmost vertex after simplifying

3. Verify the correctness of the 2mass-2spring system

Assuming read data: m1=6.0; m2=3.0; d1_0= 6.0; d2_0= 12.0; k1=k2=1/6.0; v1_0=v2_0 = 0.0; from file.

The motion equation for this system:

$$\begin{cases} m1 \cdot d1''(t) + (k1 + k2) \cdot d1(t) - k2 \cdot d2(t) = & 0 \\ m2 \cdot d2''(t) + k2 \cdot d2(t) - k1 \cdot d1(t) = & 0 \end{cases} \tag{6.0.2}$$

The vibration shape of a 1D mass-spring system, which is undamped and consists of 2 masses and 2 springs. We get the vibration of mass2 in Figure 6.3.

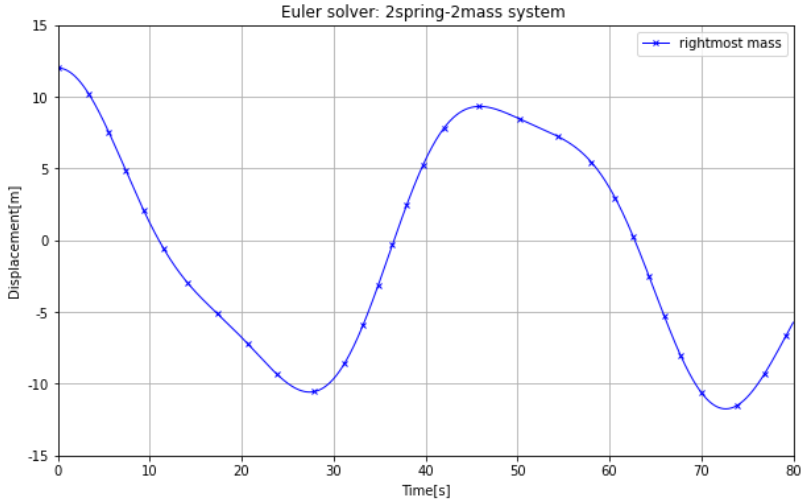

Figure 6.3: Undamped two-mass system dynamic displacement result of the m2 mass over time

Following the motion equation governing the 2mass-2spring system (as given in Equation 6.0.2), we have derived an analytical solution (as detailed in Appendix A from Pasetto et al research [11]). When we compare this analytical solution with the results obtained from our Euler's solver, as illustrated in Figure 6.4, it becomes evident that our Euler's solver closely matches the exact analytical solution.

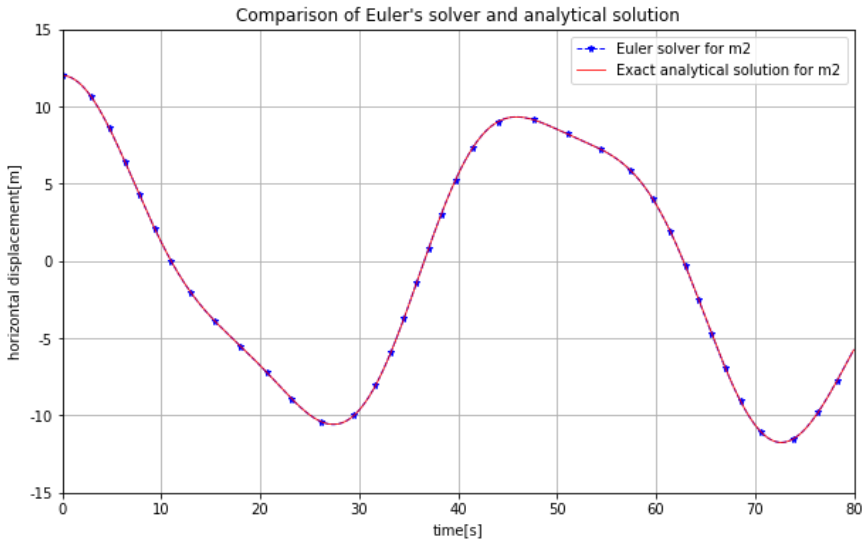

Figure 6.4: Undamped two-mass system dynamic displacement result of the m2 mass. Comparison of displacement of Euler's methods with the analytical solution, time step = 0.01s

6.2 Performance analysis

In this project, we utilize various benchmarks with different problem sizes to evaluate the performance of our implementation. The problem sizes encompass a range from 8 to 10,000,000. All data was gathered from the Medusa node, which features an x86/64 architecture with 40 CPU cores, located on the Rostam cluster at CCT, LSU.

Experiment setups:

The hardware specifications include:

- L1 cache: 32 KB (32,768 bytes)
- L2 cache: 1,024 KB (1,048,576 bytes)
- L3 cache: 28,160 KB (28,856,320 bytes)
- NUMA nodes: 2
- Cores per socket : 20

The application:

- time step = 0.01s,
- number of steps = 64000

6.2.1 Performance of MPI backend

We measure the execution time of the MPI backend with a single rank as the reference point for speedup calculation across different problem sizes. Figure 6.5 visualizes the speedup ratios across various numbers of ranks, demonstrating that significant scalability improvements can be achieved with sufficiently large problem sizes.

To closely examine the scalability of problem sizes within different ranges, refer to Figure 6.6.

In Figure 6.6(a), for smaller problem sizes below 50,000, increasing the number of ranks does not enhance performance; instead, it leads to diminished performance. The cause of this lack of scalability lies in the substantial communication overhead between ranks, which outweighs any performance gains. With more ranks, the communication overheads increase.

Consequently, employing more ranks results in inferior performance compared to using just one rank. Running on a single rank allows the entire array to reside in memory, eliminating communication and associated overhead. In contrast, when using two or more ranks, communication overhead is introduced while diminishing the work allocated to each rank. This addition of communication overhead is the primary factor behind the declining performance, and it requires a certain array size to amortize the communication costs.

In Figure 6.6(b), In the realm of intermediate problem sizes, such as 400,000 or 500,000, the MPI backend exhibits no substantial speedup, but it still maintains scalability across all ranks.

In Figure 6.6(c), for large problem sizes exceeding 1,000,000, the speedup curve experiences a rapid ascent as the number of ranks increases from 2 to 24, with no significant additional speedup realized when using more than 24 ranks in the implementation. It shows flattening after 24 ranks,

which is due to Inter Process Communication(IPC), and the more cores added, the more IPC overheads.

And when problem sizes reach 4,000,000 the speedup value decreases as the problem size increases due to the more data(big size), the more IPC overheads because of more data transfer work.

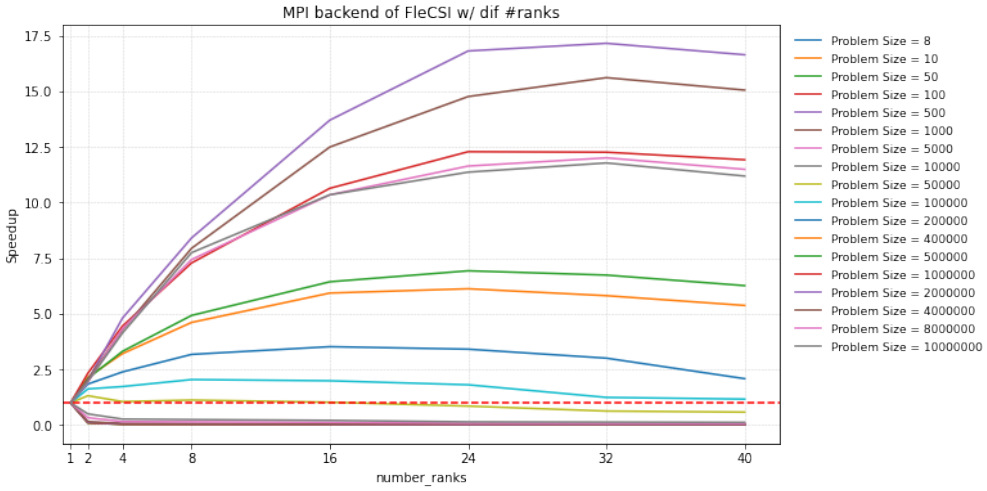

Figure 6.5: Scaling plot for 1D spring-mass system benchmark running with different problem sizes. The graphs demonstrate the relationship between speedup and the number of ranks when using MPI backend of flecsi. A higher rank number means that larger partitions are created for tasks. The speedup is calculated by scaling the execution time of a run by the execution time of the single-thread run. A larger speedup factor means a smaller execution time for the sample.

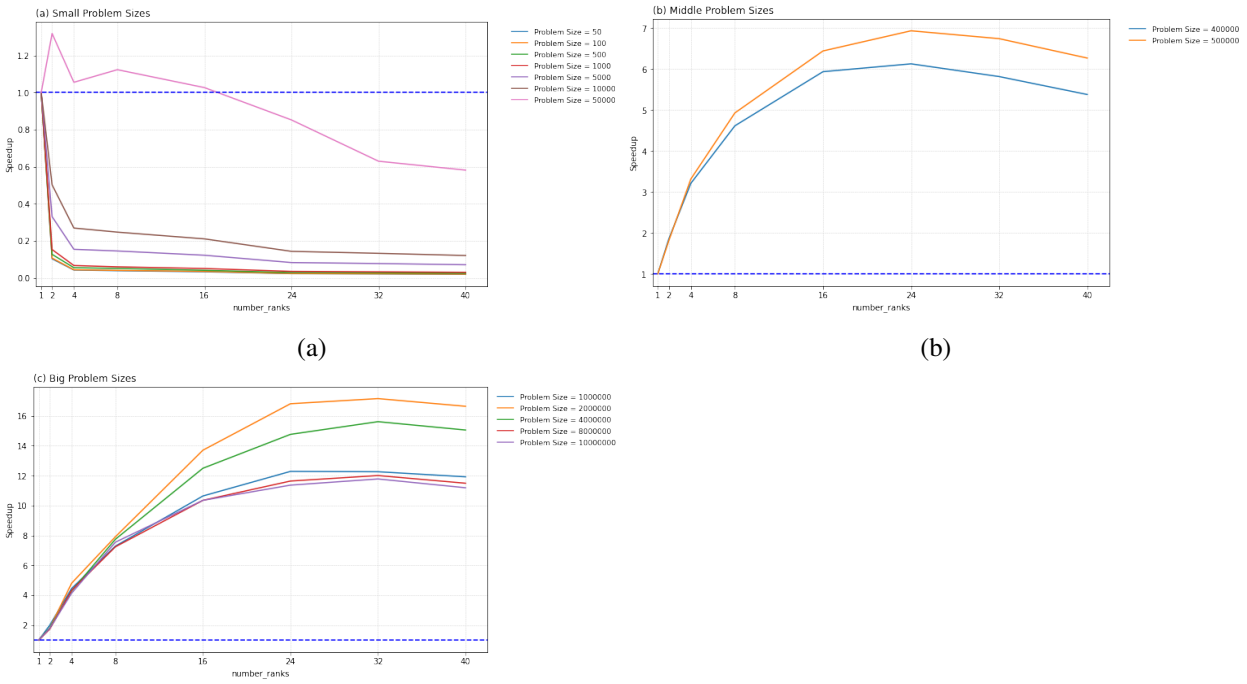

Figure 6.6: Scaling plots when MPI backend of FleCSI ran with different problem sizes: (a) small problem size, (b) middle problem size, and (c) big problem size.

6.2.2 Performance Analysis of HPX backend with optimized solver using parallel forall() algorithm

With parallel forall() algorithms added inside of the dependency task, we finally measure the execution time of the application using HPX backend with a single thread as the reference point for speedup calculation across different problem sizes. Figure 6.7 shows the speedup ratio with different numbers of threads. The optimized solver with parallel forall() provides sufficient workload inside the task. Therefore we gain good scaling performance. It shows bad performance for small sizes due to the overheads dominating over performance gain as threads increase, and It shows significant performance scaling for larger sizes due to the sufficient parallel work.

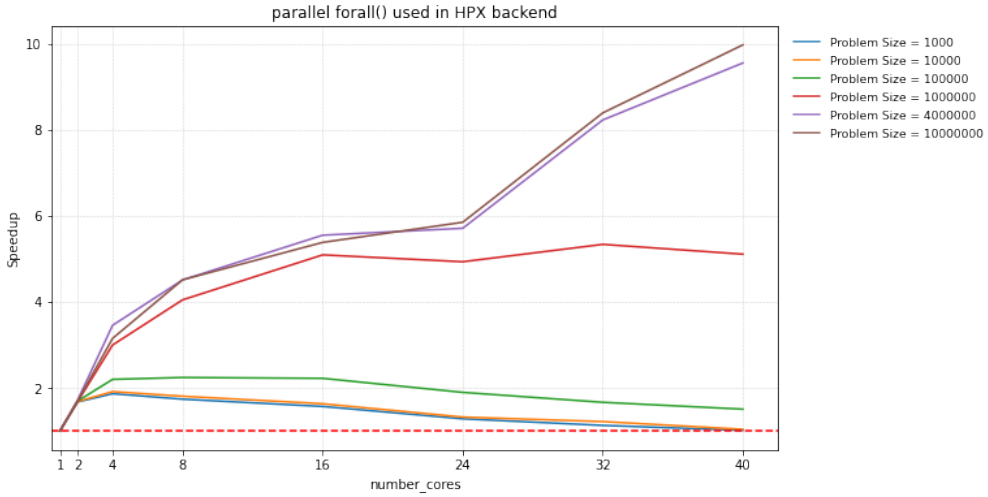

Figure 6.7: Scaling plot of HPX backends of FleCSI running with different problem sizes. The speedup is calculated by scaling the execution time of a run by the execution time of the single-rank run. A larger speedup factor means a smaller execution time for the sample.

6.2.3 Compare HPX backend with optimized solver using parallel forall() algorithm and MPI backend

In summary, our study compared the performance of the application utilizing both the HPX backend and the MPI backend. To establish the speedup metric, we utilized the execution time of the application without the addition of parallel forall() when using the HPX backend as the baseline.

Figure 6.8 illustrates the speedup ratio across varying core counts.

In Figure 6.8(a), let's first consider the baseline. The green line represents the application running without the inclusion of the parallel for() loop (forall()) using the HPX backend. Notably, the performance remains consistent as the application exhibits no inherent parallelism. With only one task within the application, there are no task dependencies or parallel operations within the task.

Next, the red line depicts the application running with an optimized solver (forall()) using the HPX backend. We've employed a parallel loop forall() instead of a traditional for loop () to update displacement or velocity. It's essential to note that in HPX, we execute it on a single rank while

increasing the number of cores. The observed speedup with the red line corresponds to the increase in core count. This arises due to the internal parallelization of the flecsi task itself, facilitated by HPX's parallelization using the parallel for loop. Consequently, this parallel task runs iteratively within the time domain, resulting in the observed speedup.

Furthermore, the blue line displays a speedup in MPI. This is because the application operates not on a single core but across N ranks, each having one core. The parallelism arises from flecsi's design, which distributes large arrays across the ranks, enabling parallelism across segmented pieces of the array.

Throughout our experimentation with different problem sizes(Figure 6.8 (a-d)), we observed that as the problem size increased, the performance of the HPX backend became comparable to MPI and, notably, even outperformed MPI with larger problem sizes.

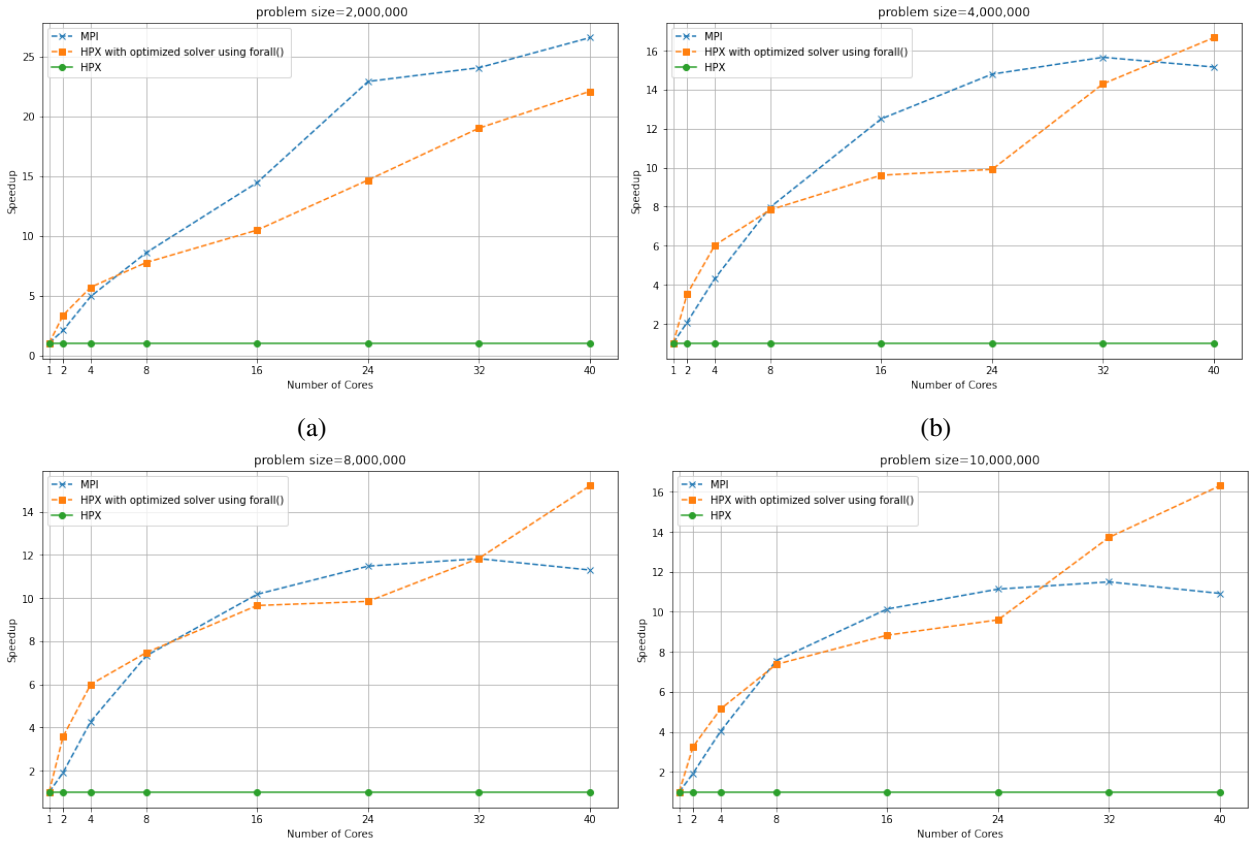

Figure 6.8: Scaling plots for 1D spring-mass system benchmark running with different problem sizes: (a) 2 million, (b) 4 million, (c) 8 million, and (d) 10 million. The graphs demonstrate the relationship between speedup and the number of cores when using HPX backend, HPX with optimized solver using parallel forall() algorithm, and MPI backend.

6.3 Conclusions

In this project, we delve into the intricacies of the 1-dimensional spring-mass system, employing iterative solutions based on Euler's method. The primary focus is implementing this system using the FleCSI library, a framework for parallel computing. We further conduct a comprehensive performance analysis using the MPI and HPX backends of FleCSI.

The results reveal intriguing insights into the parallel solver's behavior. The MPI backend demonstrates commendable scalability when applied to sufficiently large problem sizes, exhibiting a significant speedup across a range of ranks. In contrast, for smaller problems, the MPI backend shows limited performance gains due to substantial communication overhead among the ranks.

The HPX backend is designed for task-based parallelism. The simplicity of the simulation problem, featuring a limited number of tasks and minimal task dependencies, results in a task tree that is insufficiently wide for HPX to demonstrate its parallel processing capabilities. In essence, the problem is a linear task with no parallel work inside the task, leaving little room for HPX to exhibit parallel performance. Our solution is to add more workload by replacing the conventional for-loop inside of each solver task with a parallelized for-loop called forall(). We then gain good performance scaling across cores. In this case, HPX backend presents comparable and even better performance over MPI backend as the input size increases.

Implementing a simulation of a 1-dimensional dynamic system serves as a valuable introductory case study for newcomers to the FleCSI library, offering a unique exploration of the library's capabilities. Additionally, it marks the first attempt to assess the performance of the HPX backend of FleCSI, providing a foundation for further improvement and optimization in this area.

In the future, we have exciting plans in store. We aim to explore and implement complex system problems using flecsi, such as 2D structural dynamics, and investigate more accurate and reliable numerical methods (such as the new mark family). Additionally, add some parallelism on the task level, for even more speedup. Furthermore, we are eager to optimize our HPX backend by implementing and incorporating more commonly used parallel algorithms.

References

[1] J. Rychlewski, On hooke's law, Journal of Applied Mathematics and Mechanics 48 (3) (1984) 303–314.

[2] M. F. Fathoni, A. I. Wuryandari, Comparison between euler, heun, runge-kutta and adams-bashforth-moulton integration methods in the particle dynamic simulation, in: 2015 4th International Conference on Interactive Digital Media (ICIDM), 2015, pp. 1–7. doi:10.1109/IDM.2015.7516314.

[3] B. Bergen, N. Moss, M. R. J. Charest, Flexible computer science infrastructure (flecsi), Tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States) (2016).

[4] B. Wagle, S. Kellar, A. Serio, H. Kaiser, Methodology for adaptive active message coalescing in task based runtime systems, in: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, 2018, pp. 1133–1140.

[5] P. Diehl, M. Seshadri, T. Heller, H. Kaiser, Integration of cuda processing within the c++ library for parallelism and concurrency (hpx), in: 2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2), IEEE, 2018, pp. 19–28.

[6] T. Heller, Extending the C++ Asynchronous Programming Model with the HPX Runtime System for Distributed Memory Computing, Friedrich-Alexander-Universitaet Erlangen-Nuernberg (Germany), 2019.

[7] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, D. Fey, Hpx: A task based programming model in a global address spacedoi:10.1145/2676870.2676883.
URL https://doi.org/10.1145/2676870.2676883

[8] W. Gropp, E. Lusk, N. Doss, A. Skjellum, A high-performance, portable implementation of the mpi message passing interface standard, Parallel computing 22 (6) (1996) 789–828.

[9] B. Bergen, I. Demeshko, C. Ferenbaugh, D. Herring, L.-T. Lo, J. Loiseau, N. Ray, A. Reisner, Flecsi 2.0: The flexible computational science infrastructure project (2022) 480–495.

[10] E. Hairer, C. Lubich, M. Roche, The numerical solution of differential-algebraic systems by Runge-Kutta methods, Vol. 1409, Springer, 2006.

[11] M. Pasetto, H. Waisman, J. Chen, A waveform relaxation newmark method for structural dynamics problems, Computational Mechanics 63 (2019) 1223–1242.