BERGISCHE UNIVERSITÄT WUPPERTAL

Distributed memory parallelization of CubeLib library using MPI

submitted by
Aleksandar Mitić

A master thesis presented for the degree of
M.Sc. Computer Simulation in Science

Date:	11th March 2024
Supervisor:	Prof. Dr. Francesco Knechtli
External Supervisor:	Dr. Pavel Saviankou
Second Supervisor:	Dr. Roman Höllwieser

Bergische Universität Wuppertal
in cooperation with
Forschungszentrum Jülich

Erklärung
gem. § 20 Abs. 9 PO (Allgemeine Bestimmungen)

Hiermit erkläre ich, dass ich die von mir eingereichte Abschlussarbeit (Master-Thesis) selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Stellen der Abschlussarbeit, die anderen Werken dem Wortlaut oder Sinn nach entnommen wurden, in jedem Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Datum und Ort,
11.03.2024, Wuppertal
Aleksandar Mitic

Acknowledgements

I extend my heartfelt gratitude to Dr. Pavel Saviankou, my mentor, for his unwavering guidance and support throughout the entire process of my master thesis. His expertise, encouragement, and valuable insights have been instrumental in shaping this research, and without his dedicated mentorship, this work would not have been possible.

I am also deeply thankful to Dr. Roman Höllwieser, my supervisor, for his consistent and constructive feedback during the development of this thesis. His commitment to excellence and willingness to share his expertise have been invaluable. Additionally, I extend my appreciation to Prof. Dr. Francesco Knechtli for supervising my thesis.

Special thanks are due to FZ Juelich for their support, and I express my gratitude to Christian Feld for his assistance in setting up the *Score-P* measurements.

Table of contents

List of Figures

List of Tables

1 Introduction

In the contemporary landscape of scientific and computational research, the pursuit of computational power has driven the adoption of supercomputers and parallel programming paradigms. As researchers and engineers strive to solve increasingly complex problems, the need for efficient utilization of high-performance computing (HPC) resources has become paramount. The development and utilization of sophisticated performance analysis tools became a necessity to ensure optimal program execution.

The existing Cube Framework, an integral component of the performance analysis toolkit, has been pivotal in providing insights into application behavior. However, a workload such as performance measurement analysis of large applications can be limited by the computation load. Large memory footprint can put a limitation on the size of feasible performance measurement configuration.

This thesis undertakes the task of rewriting and enhancing the *CubeLib* library, which is part of Cube Framework, with Message Passing Interface (MPI) capabilities. With a new parallel approach, workload and memory footprint is distributed among the processes. This not only makes the computations faster but also decreases the amount of memory each process needs. This change addresses and helps to avoid the previous issues, making the library more efficient and effective.

The structure of this thesis reflects a comprehensive exploration of challenges and solutions encountered as follows: In Chapter 2, we give introduction to HPC which explores the need for supercomputers and the emergence of parallel programming as a fundamental paradigm in HPC. Also, it discusses the significance of tools like *Score-P* or *CubeLib* in measuring an application's performance, and how it forms a three-dimensional performance space. Chapter 3 presents the point-to-point communication structure in the MPI environment including an essential approach of asynchronous communication. Providing an in-depth examination of the Cube Framework, Chapter 4 outlines its role in performance analysis and its relevance in the HPC domain. It identifies the limitations and opportunities for enhancement within the Cube library, which is the main topic of this thesis. Chapter 5 describes the design and implementation of a new MPI prototype to address the challenges identified in the existing *CubeLib* library. Exploring the process of integrating the MPI prototype into the Cube library, it details the improvements that are made. In Chapter 6, we present the outcomes of the new *CubeLib* library, emphasizing the performance of new approach focusing on the artificial benchmark profiles and testing the performance of the enhanced algorithm on the real cubes of different sizes. In Chapter 7, we summarize the key findings and contributions of the thesis providing insights into the broader implications of the parallelized Cube library and leaving the space for future research and development.

2 HPC environment

2.1 Evolution of supercomputers

High performance computing (HPC) is essential for tackling complex computational tasks across diverse fields, from scientific research to engineering and healthcare. Their ability to handle massive datasets and perform simulations accelerates innovation and drives breakthrough discoveries. By providing the computational power needed for innovation, supercomputers play a crucial role in advancing knowledge and driving economic growth.

Transitioning from the evolution of supercomputers, it's evident that these advancements paved a long way. In 1977, the CRAY-1 was the sole computer that fulfils the computer capability requirement of processing from 20 to 60 million floating point operations per second [1]. By employing a method known as "chaining," the CRAY-1 was capable of enhancing the computational efficiency. This method involves linking the CRAY-1's vector functional units with scalar and vector registers to produce intermediate outputs [2]. These results are then reused immediately without additional memory references, thus preventing slowdowns commonly observed in other contemporary computer systems.

Following the CRAY-1, one significant development on the evolution of the supercomputers was the emergence of parallel processing architectures: The Connection Machine [3], introduced in the mid-1980s, offered a new solution for fast symbolic processing tasks. The Connection Machine architecture stands out as one of the most innovative among recent parallel systems. It's considered a "logic-in-memory" design, where processors are not physically distinct from the main memory. Since this architecture has a large number of floating-point processors, it becomes a very useful tool for performing large-scale numerical computations effectively [4].

Another significant stride in HPC was seen with the advent of Beowulf clusters [5], which revolutionized parallel computing. Beowulf clusters exploit together all three main components such as commodity personal computers (can be called as workstations), cost friendly Ethernet networks and the open source Linux operating system [6]. They often utilize parallel processing libraries such as MPI and Parallel Virtual Machine (PVM) [7]. These libraries enable programmers to split tasks among a network of computers and gather the processed results. Examples of MPI software include Open MPI or MPICH, with other implementations also being available.

The CRAY T3D is a machine employing "Multiple Instruction Multiple Data" architecture, featuring a rapid interconnect network facilitating the exchange of control information and data. While its memory is physically distributed, it is logically shared [8, 9]. Meanwhile, The Computer Machine-5 (CM-5) is introduced with a memory arrangement where each processor has its own local memory for quick access [10]. Additionally, processors can access data stored in the remote memories of other processors through the network.

When the early 2000s came, researchers recognized the potential of using programmable graphics hardware (GPUs) for general-purpose computing tasks, offering significantly faster solutions to compute-intensive problems compared to conventional CPUs [11]. GPU computing with CUDA - CUDA is a parallel computing platform and programming model developed by NVIDIA [12] - presented a novel approach, utilizing hundreds of on-chip processor cores that collaborate to address complex computing challenges, effectively transforming GPUs into massively parallel processors.

Supercomputers are known for their extraordinary processing capabilities [13]. Supercomputers have come a long way from their early models like the CRAY-1 to the advanced systems we have today. They're

incredibly powerful and can handle huge amounts of data, making them crucial tools in fields like science and engineering. As technology progresses, supercomputers are expected to become even more powerful, leading to exciting possibilities for solving complex problems and driving innovation in the digital era.

However, the impact of supercomputers goes beyond just their computational power. These HPC systems play an important role across different human activities, from weather forecasting and scientific simulations to financial modeling and more on. Two key components are critical for the evolution of supercomputers: the increasing complexity of problems and the demand for faster results. This has led to the continuous development of these machines in both size and capability.

Given their consumption of power, time, and financial resources, optimizing the use of supercomputers is needed. Making them as efficient as possible not only saves important resources but also makes the computer tasks work better. This need for optimization sets the stage for the next chapter of this thesis, which explores parallelization and optimization strategies.

2.2 Motivation for parallelization

In a traditional computer, a single processor executes the actions specified in a program. To increase computational speed, multiple processors can be employed to tackle a single problem simultaneously. This approach involves splitting the overall problem into separate parts, each executed by a distinct processor in parallel [14]. Programming for this type of computation is called parallel programming, where the goal is to efficiently distribute tasks across multiple processors to maximize computational efficiency. While this approach offers potential for increased computational power, it requires rewriting serial programs into parallel ones or developing translation programs to automatically convert them [15].

Parallelism has proven effective across a multitude of domains, spanning from HPC and server environments to graphics accelerators and various embedded systems [16]. Its utilization extends to diverse applications, including scientific simulations, data analysis, image processing, machine learning, and real-time embedded systems. This versatility highlights parallelism's significance in improving computational efficiency and meeting the demands of modern computing tasks across different domains.

2.3 Parallel systems and architectures

In parallel programming, memory architecture refers to the organization and access methods of memory within a parallel computing setup. It encompasses different types like shared memory architecture, where all processors access a single memory space, and distributed memory architecture, where each processor has its own local memory. Each computer includes a processor and local memory, but this memory isn't accessible by other processors [14]. The interconnection network facilitates communication between processors according to the program's requirements. These types of multiprocessor systems are often referred to as message passing multiprocessors. Open Multi-Processing (OpenMP), is an Application Programming Interface (API) designed to use shared-memory multiprocessing programming across platforms in C, C++, and Fortran languages [17]. OpenMP gives a framework that offers developers an accessible and adaptable interface for parallel applications suitable for a variety of setups, from standard desktop computers to supercomputers [18]. Achieving a standard for message passing systems, which offers library routines and associated operations, is made possible

through the message passing interface (MPI). Applications developed utilizing a hybrid parallel programming model can operate on computer clusters by integrating both OpenMP and the MPI. In this model, OpenMP uses parallelism within a multi-core node, whereas MPI manages parallelism among nodes. In this thesis, the focus is on the distributed memory parallelization approach. Distributed memory in MPI refers to a computing architecture where each processor has its own private memory, and processors communicate by sending and receiving messages through a network. Such configuration can be seen in Figure 1.

The MPI enables the coding of parallel programs that exchange data through sending and receiving messages between participating processes where the relevant data is being exchanged through these messages [19]. MPI provides a comprehensive set of communication for parallel computing. This includes point-to-point communication, in which messages are exchanged between specific pairs of processes, as well as collective communication, where messages are broadcasted or gathered among groups of processes. [15]. These functionalities are essential for coordinating computation and data exchange in parallel programs, enabling efficient collaboration among distributed computing resources.

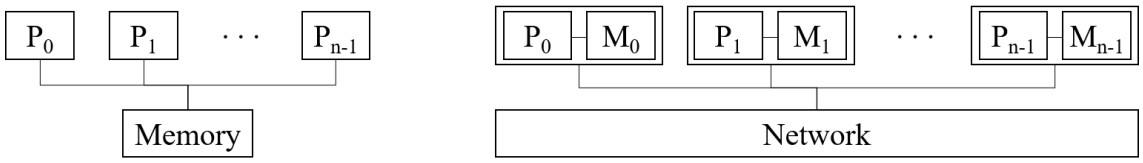

Figure 1: Types of parallel systems

On the left, the configuration represents a shared memory system, where each 'P' block signifies a single processing unit. On the right, the arrangement depicts a distributed memory system, maintaining the same notation for the processing unit and each 'M' block denotes a memory unit**.**

2.4 Performance analysis tools

In addition to the typical bugs encountered in software engineering, parallel applications introduce a unique set of challenges, primarily due to the simultaneous execution of parallel algorithms, e.g., increasing parallelization leads to algorithmic and performance pitfalls, which are hard to impossible to resolve in manual approach at small scale. Many appear only at large scale and we cannot detect them without using specialized performance analysis tools. Tools that assist programmers with this purpose are crucial for conducting HPC research. Each tool comes with its unique set of strengths and weaknesses.

HPCtoolkit is a collection of tools tailored for the analysis of high-performance computing applications [20]. It assists in profiling, tracing and visualizing performance metrics to pinpoint and lighten bottlenecks, thereby streamlining application performance. Accommodating parallel programming frameworks like MPI and OpenMP, *HPCtoolkit* is a favored choice within the HPC sector for its robust analysis capabilities.

Barcelona Supercomputing Center's *Paraver* tool specializes in the examination of parallel and distributed systems [21]. It brings to the table functionalities for performance measurement, visualization, and analysis. Its adaptability to a variety of parallel programming frameworks, such as MPI and OpenMP, renders it a flexible solution for enhancing application performance.

The *Score-P* project contributes an instrumentation and measurement interface that it offers tracing, profiling, and visualization capabilities that aid in the detection and optimization of performance bottlenecks, supporting MPI, OpenMP, and hybrid programming models [22].

Developed at the Jülich Supercomputing Centre, *Scalasca* stands as a suite of tools designed for the performance analysis of parallel and distributed applications [23]. It offers automatic trace analysis supporting MPI, OpenMP, and hybrid programming models.

Vampir, a product of Technische Universität Dresden, is acclaimed in the HPC community for its detailed analysis of parallel and distributed application behaviors [24]. It provides extensive features for visualization of data tracing collected by *Score-P*, delivering in-depth understanding of execution flows, communication patterns, and resource usage.

2.4.1 Score-P profiling and tracing

Score-P plays a vital role for performance tracing for parallel codes. By providing a unified infrastructure for instrumentation and measurement, *Score-P* simplifies the process of collecting and analyzing trace data, which is crucial for understanding the behavior of parallel applications [25]. Tracing allows developers to visualize the execution flow, identify performance bottlenecks, and optimize code behavior.

With *Score-P*, users can instrument their codes to capture detailed trace data, including function calls, communication events, and synchronization points. This rich set of trace information enables deeper insights into the runtime behavior of parallel applications, helping developers pinpoint inefficiencies and optimize performance.

The redundancies in the current tools landscape pose challenges for both developers and users. By integrating similar components and interfaces into a unified infrastructure, *Score-P* reduces redundant effort for development, maintenance, and support across multiple groups. This streamlined approach not only enhances interoperability but also frees up resources for enhancing analysis functionality in individual tools. From a user perspective, *Score-P* addresses the discomfort caused by multiple learning curves, incompatible configurations, and redundant installations, thereby improving usability and reproducibility in performance analysis tasks. *Score-P*'s role as a joint measurement infrastructure aligns with the goal of overcoming the challenges posed by the fragmented tools landscape and promoting efficient and effective performance analysis in parallel computing environments. In this thesis, *Score-P* is used to obtain profiles from the performance measurement of the parallelized *CubeLib* library. In Chapter 4, more details about the *CubeLib* library are given.

2.5 Performance space

Each tool records its performance measurement outcomes in varied formats. Nonetheless, for effective performance analysis and particularly for comparing results, a standardized and more broadly applicable interpretation of this data is essential. Consequently, discussions about this data often refer to it within a concept known as the "performance space," which allows for a unified understanding and analysis of the performance metrics.

The performance space is characterized by three dimensions: performance metrics, call tree, and system description [26]. Each of them answers the following questions, during the execution, respectively:

1-) What is kind of performance metric 2-) When is it measured during execution in the place of source code? 3-) Where is it in the machine or system?

Metrics identify performance issues, call trees locates them in the application's execution, and system description provides information about the component of the HPC system. Each coordinate in this space is associated with a numeric value representing issue severity, enabling quantitative performance assessment. In the following section, the dimensions of performance metrics, call tree, and system description, along with their associated severity values, are discussed in detail.

2.5.1 Metrics

Metrics provide valuable insights into various aspects of the application's behavior during execution. For example, one commonly tracked metric is $Execution\ Time$ [27], which measures the time taken by the application to complete its execution. This metric offers a high-level overview of the application's overall performance and efficiency.

$MPI\ Time$ [28] represents the duration spent within instrumented MPI function calls during program execution. It offers insights into the overall time dedicated to MPI operations, including message passing and synchronization. It's important to note that depending on the configuration, certain classes of MPI calls may be excluded from measurement, impacting the analysis report's completeness.

On the other hand, MPI Communication Time measures the time allocated specifically to MPI communication operations. This encompasses various types of communication, such as point-to-point, collective, and one-sided communication. Understanding MPI Communication Time helps in assessing the efficiency of data exchange and synchronization among MPI processes, providing valuable information for performance optimization, and identifying potential bottlenecks in parallel applications.

Addition to the time metrics, another metric important for this thesis is the *Visits* metric. The *Visits* metric counts how often a specific function is used in the application. Functions that are used a lot often have a bigger effect on the app's performance.

Another important metric is $Memory\ Usage$, which includes measurements such as memory footprint, allocation rates, and memory accesses. These metrics help evaluate how effectively the application manages memory resources and can highlight potential memory-related issues. Within performance analysis frameworks for parallel applications, numerous other metrics play a vital role in assessing various aspects of program execution and evaluating the behavior and efficiency of parallel applications, including those utilizing MPI. We will denote total number of metrics as $N_{metrics}$.

2.5.2 Call paths

The call path represents a specific sequence of function calls within a parallel application, tracing the execution flow from the root function down to leaf functions [29]. It provides a detailed view of how computational tasks are nested and interconnected, offering insights into the program's control flow and execution behavior. The call path is crucial for understanding the fine-grained dynamics of code execution, identifying performance bottlenecks, and diagnosing issues related to function call patterns, recursion, or nested loops. By analyzing the call path, developers can identify critical paths within the application, optimize algorithmic efficiency, and address performance anomalies at a granular level. Moreover, the call path contributes to the performance space by associating performance metrics with specific call paths, enabling developers to evaluate the impact of function calls on overall application performance. Also, the

call path serves as a fundamental concept in Cube's performance analysis toolkit, facilitating comprehensive assessment and optimization of parallel applications.

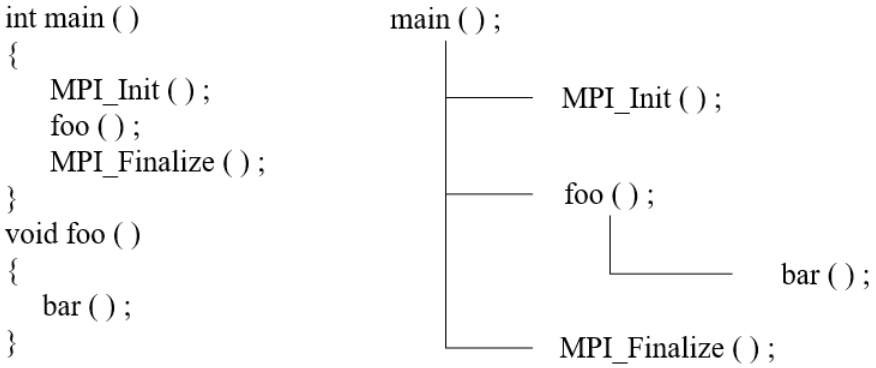

Figure 2: Call tree

We observe an illustration demonstrating how the code's organization forms a call tree. On the left, there's a pseudocode of a basic program, while on the right, there's its corresponding call tree. Every node in the call tree symbolizes a single call path. For instance, the `main()` node has three branches, `foo()` has one, while `MPI_Init()`, `bar()` and `MPI_Finalize()` are leaves.

Therefore, the extent of the second dimension is linked to the code's size and its complexity. Functions that are highly recursive can influence the depth of the call tree. Let's designate the total count of call paths as N_{cnodes} and a corresponding object call tree node from Cube codebase as *cnode*.

2.5.3 System locations

In parallel computing, especially when working with MPI applications, understanding the complex interactions between processes requires looking at performance from multiple angles. One key aspect is the system location, which tells us where the code is actually running, whether on a specific thread or process on some specific part of the HPC system also known as rank 0 or nodeplane 2, etc. This detail is crucial for identifying and fixing performance issues because different parts of the application may run on different processors.

The system location serves as a map to the execution environment. For applications that use the MPI interface alone, the system locations match with MPI processes. If an application relies solely on the OpenMP interface, then the system locations are the OpenMP threads [30]. However, many modern applications use a hybrid approach, combining MPI and OpenMP. In such cases, each MPI process might start several OpenMP threads, and the number of threads can vary across MPI processes. This setup introduces a more layered understanding of system locations, as it now includes threads initiated by different MPI processes.

Recognizing the specific system locations in hybrid applications is essential for a detailed performance analysis. It enables us to zoom in on how different parts of the application interact with the computing environment. This precise approach helps in identifying exactly where performance may lag and how to optimize it. By tailoring our strategies to the unique setup of each system location—considering the mix of MPI processes and OpenMP threads—we can enhance the application's performance more effectively. This method not only helps in pinpointing the root causes of performance bottlenecks but also ensures that solutions are well-suited to the application's specific parallel structure,

leading to improved efficiency and scalability. The total number of locations corresponds to the aggregate of all threads:

$$N_{locations} = \sum_{i=1}^{N_p} N_{t_i}$$

Where N_p represents the number of MPI processes, and N_{t_i} denotes the number of OpenMP threads initiated by the i-th process. In cases where the application does not utilize OpenMP, the formula should be considered as:

$$N_{locations} = N_p$$

In Figure 3, a machine composed of one compute node as an example of simple setup is presented. The application operates with two MPI processes, where each of these MPI processes spawns two OpenMP threads. This structure results in a total of four distinct locations for computation: two MPI processes each associated with two OpenMP threads. This configuration is a common structure in hybrid parallel applications that uses both distributed memory parallelism (via MPI) and shared memory parallelism (via OpenMP) to optimize performance and resource utilization.

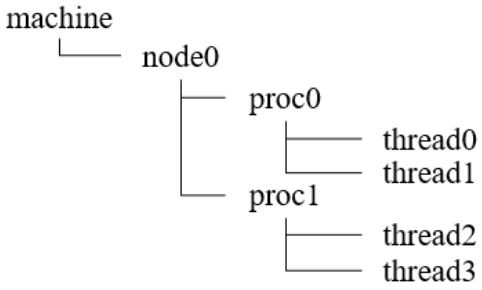

Figure 3: System tree

We have an instance of a system tree, of a setup with one compute node within the machine. The application is executed using 2 MPI processes, and each process generating 2 OpenMP threads. In total, there are 4 locations identified within this configuration.

In the Cube framework, the system dimension is not bounded by depth as in the standard description of performance space with no limitations to the specified four levels. Rather, it permits an unrestricted depth in system description. Unlike the rigid machine and node components, Cube introduces a versatile system tree node, capable of representing anything from the entire machine to a socket [26]. Each system tree node has the ability to define location groups, typically denoting processes, as well as being the branch of itself. These location groups further define individual locations, often synonymous with execution threads. Each value within the Cube data model is associated with a distinct location. From here we will denote total number of locations as $N_{locations}$, total number of system tree nodes as N_{stn}, and total number of location groups as N_{lg}. We can observe such a description in example in Figure 4.

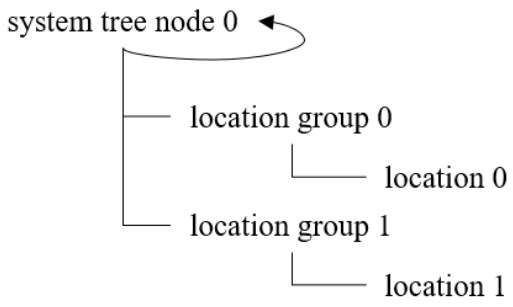

Figure 4: Cube definition of system tree

2.5.4 Severity values

Severity in Cube performance space is the next key concept, representing the actual measurement for a specific metric at a particular point in this multidimensional space. Each dimension of the Cube performance space is structured hierarchically. The metric dimension is organized in an inclusion hierarchy [31], meaning metrics lower in the hierarchy are part of their parent metrics. For instance, 'communication time' is a subset of 'execution time' (see Chapter: Metrics). The system dimension is laid out in a multi-level hierarchy, ranging from the machine level down to individual threads.

Cube also encompasses a library, such as *CubeLib* library, that reads and writes of this data model into *.cubex* or Cube4 files, which is divided into metadata describing structure of the dimensions and the data containing severity numbers. The severity value, thus, is not merely a number; it is a vital link that connects the performance of a particular metric with its occurrence in the program's workflow and its execution context within the system. This connection is essential for pinpointing performance issues and optimizing the application, as it provides a clear and quantifiable way to understand and navigate the complexity of parallel computing performance within Cube's 3D performance space. Within this framework, any point within the performance space, denoted as (m, c, s), is associated with a specific metric (m) while a particular process or thread (s) executes a given call path (c). This association is quantified by a value that represents the actual measurement of the metric m, known as the severity of the performance space.

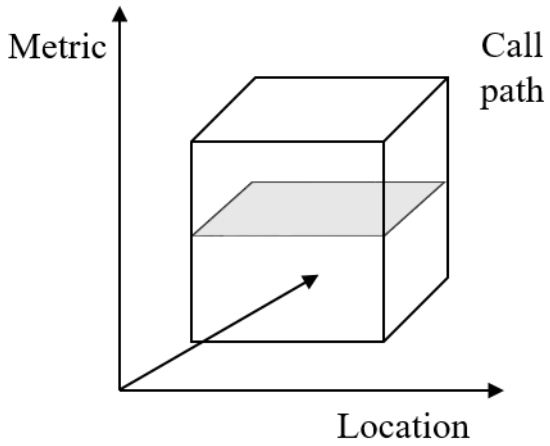

Figure 5: Representation of severity matrix

Figure above shows a cube structure and its three dimensions in hierarchical order: 1-Metrics or performance property, 2-Call path or program location, 3-System location [32].

3 MPI Point-to-Point communication

When choosing to parallelize an algorithm using the distributed memory paradigm, communication between different components of an HPC system becomes an inevitable necessity, with exception for only a few trivial instances of parallelization. At the heart of this system is point-to-point communication, a key operation that requires precise coordination of send and receive operations at runtime. Communication in MPI is crucial for data exchange between processes, involving distinct roles for sending and receiving data [33]. Although this dynamic coordination is fundamental for parallel programming [34] and offers significant benefits, it can sometimes impact performance due to the necessity of matching operations accurately to prevent issues such as deadlock.

Point-to-point communication in MPI not only allows for the seamless exchange of messages between two different processes [35] but also ensures the integrity and error-free transmission of every message. This method equates sending and receiving with the operations of storing and loading, considering factors like the rank of the involved processes, message tags, and communicators. Additionally, it plays a role in local synchronization by safeguarding against the premature overwriting of the receive buffer, thereby ensuring that data remains valid upon the completion of the receive operation.

However, any mismatch from communication can result in a deadlock, where both sending and receiving processes are idle, awaiting actions from each other that cannot be completed due to the operational misalignment. This scenario requires management of MPI operations to prevent communication disruptions and maintain smooth execution across processes.

3.1 Blocking and nonblocking communication

Building on the basic principles of MPI point-to-point communication, this section explores the distinctions between blocking and non-blocking communication modes. These modes are very important in shaping the efficiency and dynamics of data exchanges between processes in parallel applications. By examining the features and uses of `MPI_Send()` and `MPI_Recv()` as blocking operations, as well as the asynchronous alternatives `MPI_Isend()` and `MPI_Irecv()`, we look into the communication patterns for better performance and adaptability in parallel computing settings.

`MPI_Send()` is a blocking (also called synchronous) send operation [36]. It sends a message from one process to another and only returns once the message data has been copied out of the send buffer. The receiver must post a corresponding receive operation for the send to complete successfully. During the execution of `MPI_Send()`, the sending process is blocked from proceeding until the message transfer is acknowledged, ensuring that the send buffer can be safely reused or modified after the call returns. On the other hand, `MPI_Recv()` is the blocking receive operation counterpart to `MPI_Send()`. It waits for a message to arrive from a specified sender, matching a specific tag and communicator. `MPI_Recv()` blocks the receiving process until the expected message has arrived and has been copied into the receive buffer, ensuring that the data is fully received and available for the process to use.

To manage the communication pattern in parallel application, non-blocking operations (also called asynchronous) such as MPI_Isend() and MPI_Irecv() play important roles. `MPI_Isend()` initiates the sending of a message but returns immediately, allowing the sending process to perform other operations while the message transfer is still in progress. The completion of the send operation can be verified later

using test or wait functions like `MPI_Test()` or `MPI_Wait()`. This approach is useful for overlapping communication with computation or for managing communication in a more flexible manner. Similar to `MPI_Isend()`, `MPI_Irecv()` is a non-blocking receive operation. It begins the reception of a message but does not wait for the message to be fully received before returning control to the calling process. This allows the receiving process to continue executing other code while the message is being delivered.

3.2 Communication and computation overlap

In HPC systems, communication delays significantly impact overall performance. To mitigate this, programmers increasingly rely on asynchronous communication methods, enabling simultaneous communication and computation [37]. This strategy is crucial for improving performance, especially as HPC applications evolve towards exascale computing. However, it introduces complexity into the code, necessitating more development effort and resulting in programs that are less intuitive to understand.

Addressing the scalability challenges of future exascale machines, the MPI standard provides essential support through non-blocking routines as described in chapter above [38]. This approach underscores the balance between achieving operational efficiency and managing code complexity in the development of scalable HPC applications.

For our prototype, we will use such a strategy and try to achieve communication and computation overlap, as the way of parallelization will be explained in the following chapters.

4 Cube framework

Cube framework provides libraries for writing and reading measurement profiles and it includes a set of tools to manipulate profiles, to export and visualize data via graphical user interface for the manual performance analyses. The performance reports come in Cube4 file format, which also are produced by *Score-P* or *Scalasca* profiling and tracing methods. It is continuously being developed by Forschungszentrum Jülich team [23].

4.1 Overview of Cube libraries

Cube consists of a set of libraries and tools which can explore and save performance data. The most recent version of Cube is 4.82, and it is utilized by *Score-P* for storing performance measurement data in the Cube4 file format. *CubeLib* reproduces or replicates data models across every metric, while *CubeW* is used to write performance profiles. Additionally, Cube features a graphical user interface called *CubeGUI* for visually presenting the measured performance data, along with a Java reader library known as *jCubeR*.

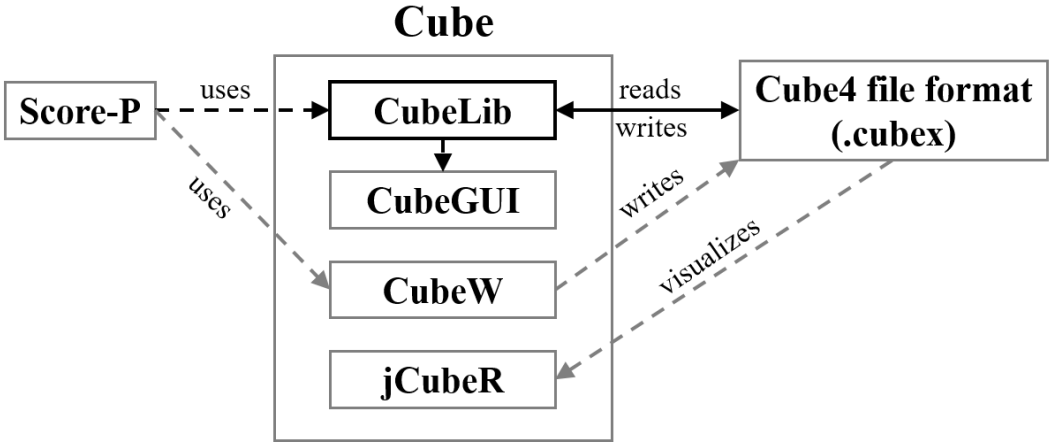

Figure 6: Cube libraries

The figure above shows a relation between Cube libraries. In the spectrum of this thesis is the *CubeLib* library, which can read and write data in the Cube4 file format [30].

The data is loaded into the cache memory through the utilization of *CubeLib*, subsequently presented in a structured format with all dimensions arranged within three columns and initial root nodes collapsed. In *CubeGUI*, users have the flexibility to collapse or expand non-leaf nodes within the metrics, call tree, or system tree, allowing for adjustable levels of detail and granularity. This feature simplifies the process of pinpointing problem sources. Moreover, severity values are visually represented by colored squares, with the specific color indicating the value, which makes easy detection of noteworthy nodes. *CubeGUI* also offers a flat call tree option, presenting call paths as a sequential list, which is particularly useful for analyzing the severities of specific methods. The software is designed to be customizable through a variety of predefined plugins. The interface of *CubeGUI* is presented in Figure 7, showcasing a three-panel display that mirrors the three dimensions.

Figure 7: CubeGUI

The image above displays a snapshot of *CubeGUI*, where the left panel exhibits metrics, the middle panel showcases a call tree, and the right panel features a system tree. Each node in the trees has the option to collapse or expand. In this instance, the nodes for the call tree (*cnodes*) `main` and `THREADS` are expended under the 'p1' as well as all nodes under the machine in system tree. All metrics are leaf's and they cannot be expanded.

However, the loading of extensive Cube4 files into *CubeGUI [39]* introduces delays and occasionally proves unattainable, causing memory overhead, performance degradation, potential data loss, scalability issues, etc. This consequently, results in slow user experience. The *CubeLib* library is used by *CubeGUI* to read or write files, and handling such issues requires the library to be restructured. First, we take a look into its architecture and methods.

4.2 CubeLib library

The *CubeLib* library, which is the main focus of this thesis, is a general purpose library made in C++ that finds application in reading and writing Cube4 file formats and allows customization of measured profiles.

4.2.1 Usage of the library

An overview of the user's interaction with the *CubeLib* library is presented. Before actual calculation of severity values takes place, the *CubeLib* library needs to be initialized. This includes the creation of the main `Cube` struct which is created with a call to: `Cube cube`. After that, the user must provide definitions of all metrics, regions, call paths, locations, etc. The *CubeLib* library provides all necessary methods for their creation: `cube.cube_def_metric(…)`, `cube.cube_def_region(…)`, `cube.cube_def_cnode(…)`, etc. Next step for the user is to provide severity values for each metric by calling the method `cube.set_sev(Metric* met, Cnode* cnode, Location* loc, double value)`.

Arguments `met`, `cnode`, `loc` are pointers to the structures we previously defined. Number `value` is a type of double severity value for a particular `cnode` of a `metric`. This is a way to write values in the

Cube4 file format. Another way of getting the data is calling `cube.openCubeReport(string _cubename)` where `_cubename` stands for input file of Cube4 type. In order to perform computations in such set up configuration, user should make a call to:

```
cube.get_sev( Metric* metric, CalculationFlavour mf, Cnode* cnode, CalculationFlavour cnf, Sysres* sysres, CalculationFlavour sf )
```

Arguments `metric, cnode, sysres` are pointer positions which are in the Cube4 file for dimensions respectively. `CalculationFlavour` is the type of `enum` and calculation flavors `mf`, `cnf` and `sf` relate to the type of severity we want to calculate.

```
CalculationFlavour  {
    CUBE_CALCULATE_INCLUSIVE = 0,    ///< Value includes children
    CUBE_CALCULATE_EXCLUSIVE = 1,    ///< Value excludes children
    CUBE_CALCULATE_SAME      = 2,    ///< Value depends on type
    CUBE_CALCULATE_NONE      = 3,  ///< Used to identify "empty" call as a default value
};
```

Output value from the call `cube.get_sev(…)` can be seen in the user's terminal if it is part of some command line tool such as `cube_dump` or in the *CubeGUI*, depending on the usage of the library. A key feature for this master thesis is that it replicates the performance space and provides calculation routines to perform different aggregations across the data in order to obtain desired value.

4.2.2 Structure of CubeLib library

In this section of the master's thesis, we introduce the core components of the *CubeLib*, which being implemented in C++, uses the capabilities of object-oriented programming, offering a rich framework that utilizes classes and inheritance. Being a reader and writer of performance measurement files, *CubeLib* consists of classes and methods relevant to such measurements. `Cube` as a main class has access to all the dimension classes such as :

`Metric` instances of the class provide a means to quantify and evaluate the performance characteristics of the software or system.

`Region` objects allow users to group performance data based on different code segments, enabling fine-grained analysis and optimization.

`Cnode` or call node represents a specific call path or execution context within the application. Its objects are typically used to represent call stacks or execution paths through the application's code.

`SystemTreeNode, LocationGroup` and `Location`, as explained in Chapter 2.5.3, objects are nested within one another and structuring the representation of system hierarchy. Here a brief information is given as following:

`SystemTreeNode`: Cube v4 introduces a generic system tree node in place of fixed machine and node elements, representing any level from the entire machine down to a socket. It can have a child such as another System Node Tree or Location Group and such example is given in Figure 4.

`LocationGroup`: Each system tree node can specify location groups which typically signify processes. Its parent is always a SystemTreeNode and children is Location.

`Location`: Every location group establishes locations, which are often equivalent to execution threads. Actual coordinates of data are Id's of locations.

It is essential to highlight that this library encompasses a multitude of functionalities beyond the scope of this thesis. A simplified visual representation of its structure and components that are relevant to us, as basic UML class diagrams are presented in Figure 8.

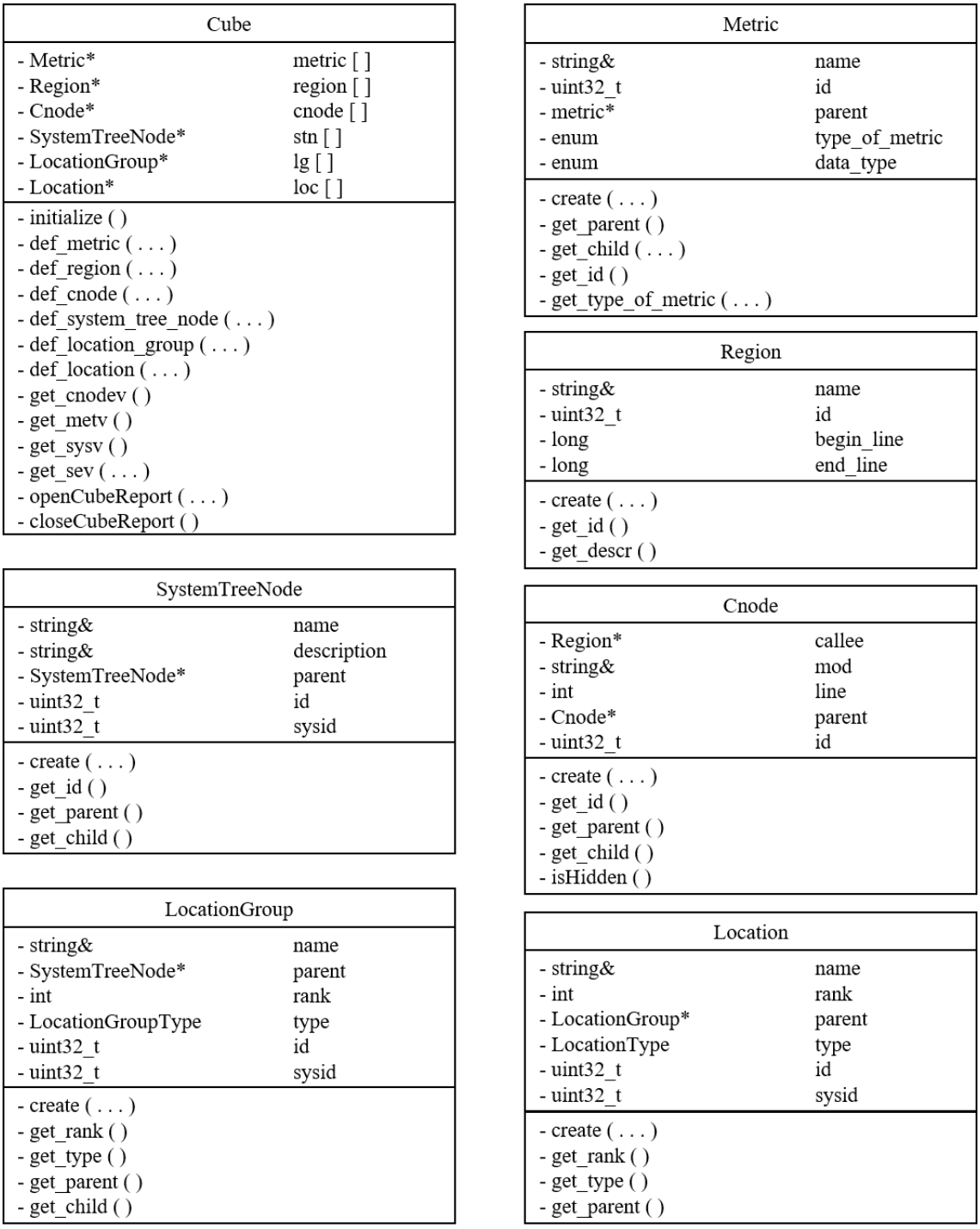

Figure 8: Shorten UML class diagram of CubeLib

The figure shows a shortened UML class diagram of the *CubeLib* library with the most important attributes and methods for the focus of this thesis.

When the struct cube is being initialized via `Cube cube` and process opens a Cube4 file via function call `cube.openCubeReport(…)`, it loads all the necessary dimension information into memory such as, metric tree, call tree and system tree at once, but data can be loaded at once, or portion by portion. If done by portion, the process can later load data only on demand. The *CubeLib* library offers a variety of functions, such as `cube.get_sev(…)`, which enable different types of aggregations to be performed on the loaded information.

When the `cube.get_sev(…)` is being called depending on the input parameters, the severity type of value is being returned as an output. This method goes inside the cube library and checks based on the given input parameters what type of a metric, which *cnode* and which system node is being asked for. For example, if the input was `cube.get_sev(0,0,0,0,-1,-1)`, this would relate to calculating the first metric with type of *Inclusive* in the metric tree, the first *cnode* with a type of *Inclusive* in the call tree and the entire system tree values. If the metric was already *Exclusive* type, it would sum up all the values of the children of the desired node and return a `double` value. We will look at this kind of measurement because it is the most data and computation intensive part of calculation, as it requires calculation of the root *cnode* and all its children.

The current *CubeLib* library computes the severity values entirely sequentially with no parallelization. In the next chapter of the thesis, we will see how we can improve its usage, but we have to omit the advanced calculations of this method and only focus when the input is for the build-in-type *Exclusive* metric for its *Inclusive* value.

4.2.3 Cube metrics

In Cube, the metric format refers to the structure or syntax used to define and represent performance metrics within the Cube performance analysis tool. This format includes specifications for metric names, data types, aggregation methods, and any additional properties or expressions used to compute and analyze performance data. On the other side, the metric type refers to the classification or category of a performance metric based on its behavior and characteristics. This classification determines how the metric is aggregated, visualized, and analyzed within the Cube performance analysis tool.

Derived metrics in Cube differ from regular metrics in that they don't store data directly within the cube report. Instead, they calculate their value based on an expression formulated using CubePL [40]. Here the type of the metrics is given:

- *Inclusive* metrics: These metrics accumulate values from the start of a call path to its end, including the time spent in all called functions. They provide a holistic view of resource usage or time spent within a given context in the call tree.
- *Exclusive* metrics: *Exclusive* metrics, in contrast to *Inclusive* ones, measure only the time or resources used directly in a function, excluding any calls to other functions. This allows for pinpointing specific areas of code for optimization.
- *Derived* metrics: Derived metrics indeed do not store data directly but calculate their values based on expressions using CubePL. This makes them highly flexible and powerful for custom analyses.
- *Post-derived* metric: Evaluated after all other metric aggregations, this metric type allows for complex analyses that can adapt based on the structure of either the system tree or the call tree. The flexibility in its calculation, depending on whether it references other metrics or stands alone, underscores Cube's adaptability in performance analysis.

We already explained the *Inclusive* and the *Exclusive* type of metrics. Therefore, for the performance measurement, we focus on including all of the *cnodes* and the system tree values when giving an input to the `cube.get_sev(…)` function.

Figure 9 illustrates the execution times of functions within an MPI program, displaying both *Inclusive* and *Exclusive* time measurements. In the *Inclusive* time view, `main()` encompasses the total time, including the initialization and finalization of MPI with `MPI_Init(…)` and `MPI_Finalize()`, as well as the execution times of `foo()` and its nested function `bar()`. The total *Inclusive* time for main() is the sum of all the times of the functions it calls, plus the time spent in `main()` itself.

In the *Exclusive* time view, `main()` only accounts for the time spent in the main() function itself, excluding the time spent in functions it calls, like `foo()`. The *Exclusive* time is calculated by subtracting the time spent in `foo()`, and any other called functions or MPI operations from the total time `main()` is active.

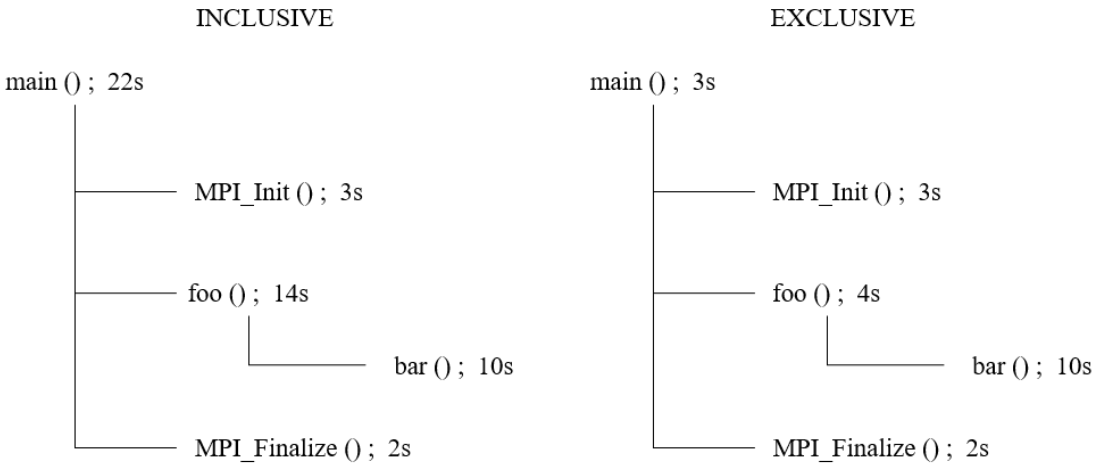

Figure 9: Inclusive and Exclusive values

In a performance measurement system, time is typically captured as the duration between the entry and exit points of a function, inherently resulting in an *Inclusive* measurement. This *Inclusive* time metric encompasses the total time spent within the function as well as the time spent in all the functions it calls. For instance, the *Time* metric in Cube is inherently *Inclusive* because it accounts for the entire duration a function is active, including the execution times of its child functions.

On the other hand, an *Exclusive* metric, such as the *Visits* metric, counts the occurrences of function calls without regard to the calls made to other functions by these functions. This metric is exclusively concerned with the frequency of a particular function's invocation and remains unaffected by the children's values.

In performance analysis tools like *CubeGUI*, the way call path severity values are recorded in a file is influenced by whether the metric is *Inclusive* or *Exclusive* [41]. This distinction also informs the user interface behavior: when a call node is expanded in *CubeGUI's* tree view, it shows *Exclusive* values, whereas a collapsed node indicates *Inclusive* values. The underlying library, therefore, must be capable of converting between *Exclusive* and *Inclusive* values to accommodate user interactions and provide a complete picture of the application's performance. By the given following formulas bellow [30], we can calculate the values.

$$t_{excl} = t_{incl} - \sum_{children} t_{incl} \tag{4.1}$$

and

$$t_{incl} = t_{excl} + \sum_{children} t_{incl} \tag{4.2}$$

Formula (4.2) can be extended:

$$t_{inlc} = t_{excl}^{own} + \sum_{\substack{direct \\ children}} t_{excl} + \sum_{\substack{grand \\ children}} t_{incl} \tag{4.3}$$

The distinction between the formulas for converting *Inclusive* to *Exclusive* times, and vice versa, is significant. To obtain an *Exclusive* time from an *Inclusive* time, we only require the *Inclusive* times of a node's immediate children. In contrast, to transform *Exclusive* value into *Inclusive*, it's essential to first determine the *Inclusive* value for all children of the node, with a recursive calculation that extends through the entire subtree. The sequence diagram in Figure 10 presents the flow of a function called `get_sev_aggregated(…)`, which checks if a metric is *Exclusive* or cached and if not, recursively aggregates values from child nodes before returning the final result. This is where the modifications will be implemented within the *CubeLib* codebase.

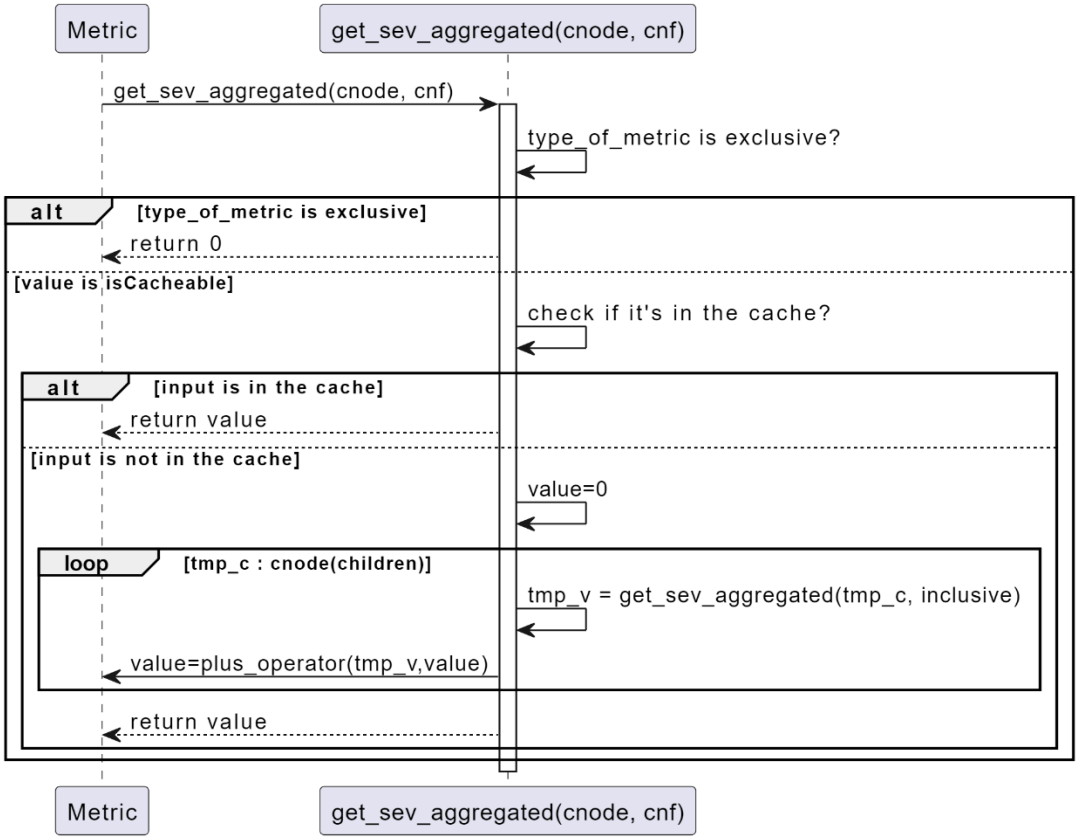

Figure 10: Current internal steps of the value calculation

This recursive nature of computing *Inclusive* times aligns with principles of data access efficiency in computing, notably influenced by data locality. Data locality refers to the speed advantage gained when data is stored contiguously, allowing for faster access due to fewer data transfers between storage and memory. This concept is leveraged when *Inclusive* values are used, optimizing data storage by placing child nodes' data adjacent to each other. Such an arrangement follows a *Breadth-first-search* [42] pattern in tree traversal, where a node is followed sequentially by its children, then by its grandchildren, and so on.

In the case of *Exclusive* values, the preferred order is based on a *Depth-first-search [43]*, where the traversal dives recursively into each child node, proceeding to the leaf nodes before backtracking. This method of ordering ensures that nodes are processed and stored in a way that reflects their hierarchical relationship in the call tree (Figure 11).

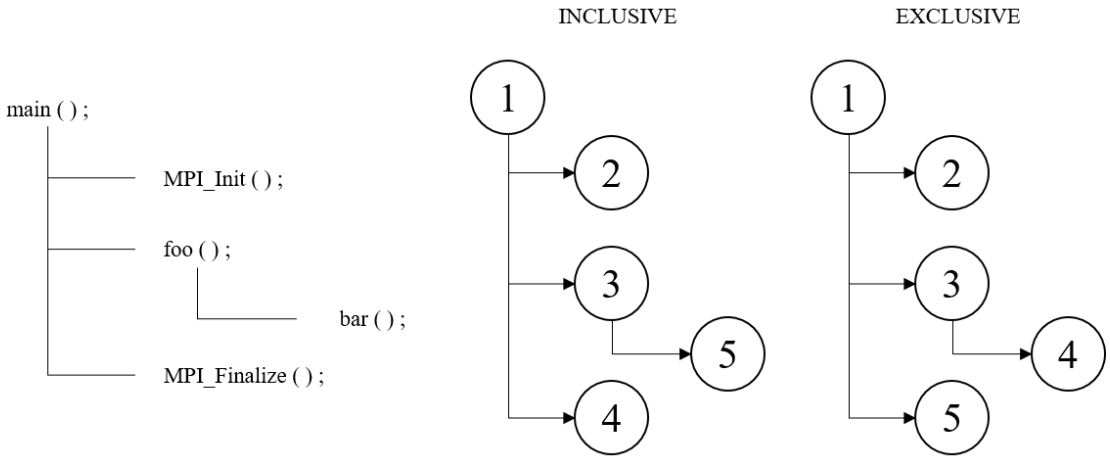

Figure 11: Tree ordering
The figure shows Inclusive and Exclusive order for a simple call tree.

Order in the file	Inclusive	Exclusive
1	`main()`	`main()`
2	`MPI_Init()`	`MPI_Init()`
3	`foo()`	`foo()`
4	`MPI_Finalize()`	`bar()`
5	`bar()`	`MPI_Finalize()`

Table 1: Call path order

In the enumeration process, whether a *Round-Robin* system or other approach is applied, we repeat the same detailed step-by-step examination of tree structure as we go through the *cnode* list. When we focus on an *Exclusive* metric, which is the focus of this thesis, the vector is examined using a *Depth-first-search* method. This means we explore each branch before moving to the next. As a result, the *Round-Robin,* the *Deepest chain* or similar become almost identical in practice. What sets them apart is how they each trace the pathway through the nodes in the call tree. Calculating the *Inclusive* value for the root node, especially when dealing with an *Exclusive* metric, is the most computationally demanding aspect of our algorithm. These details about the algorithms are explained in the following chapter.

4.3 CubeLib tools

CubeLib is equipped with different command-line tools for data export, manipulation, and analysis. Among the most important tools are `cube_diff` and `cube_merge`, which enable users to perform algebraic operations on Cube4 files to generate new, aggregated results. Similarly, tools like `cube_dump` and `cube_calltree` are useful for presenting data in different formats. Another important tool in this suite is the `cube_server`, designed specifically to apply the remote analysis of performance data. `cube_server` tool benefits significantly from the optimizations presented in this thesis.

During performance measurement, the obtained files result in a very large call tree and this leads to huge Cube files. Therefore, the user needs to filter the Cube files, and repeat that process until a desired profile is achieved. In the initial stages of performance measurement, the data is raw and unrefined, leading to the creation of these large Cube files. Due to their size and the difficulty of transferring such data to a local machine for analysis, it is more efficient to initiate the Cube server on the HPC system itself. By taking advantage of the client-server features of *CubeGUI* with the Cube server, the user can directly access and analyze the results on the HPC system. This approach not only avoids the process of transferring large files but also utilizes the HPC system to enable the Cube server's operations.

The current `cube_server` does not utilize the task distribution, therefore if we were to connect to the Cube server, we would only connect to the log-in node and then the configuration would only depend on the log-in node itself. Also, if the user could score large Cube4 files, instead of copying them to the local machine, he could keep them in the HPC cluster and filtering those files would be easier. In this case, we offer clients another setup. It goes with two parts as client and server. The server is part of the *CubeLib* library, and the client is a typical GUI.

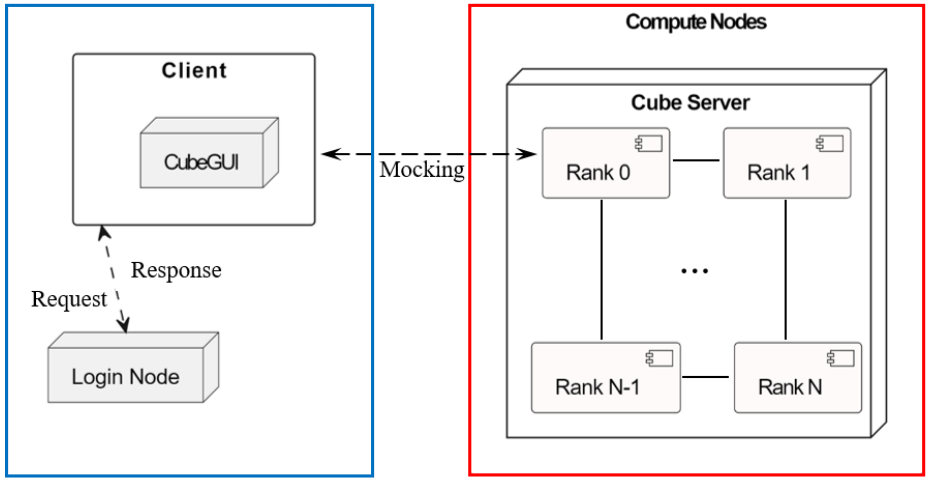

Figure 12:Proposed setup of Cube server on HPC

We do not focus on the details of the server in this thesis, but we will simulate client-server behavior during our library performance measurement and leave space for future work to integrate with the proposed prototype. Proposed setup can be seen in Figure 12, where the blue color frame represents current setup and the red color frame on how future setup can be established for CubeGUI to communicate directly with `cube_server`. This approach is important for minimizing the impact on

system performance by applying filters, selective recording, and small-scale measurements from the outset.

In this study, our focus is on examining the calculation algorithm itself. However, the execution of the `cube_server` on compute nodes introduces an additional complexity, as it requires the capability to connect to the `cube_server` while it operates on these nodes. This aspect represents an important challenge, and it falls outside the scope of our current analysis. Therefore, rather than directly tackling this connection issue, we choose a simulation approach. We simulate a mock task that performs the behavior of a request coming from the *CubeGUI* client to evaluate the calculation performance. This mock task allows us to measure the performance characteristics of the `cube_server` precisely in a controlled environment without the need for an actual connection to the server running on compute nodes. This method allows us to concentrate on studying the main algorithm while preparing for future research on how the server works with compute nodes. The main contribution of this thesis is that the described improvement allows the cube server to operate across multiple compute nodes, enhancing reaction times and minimizing the memory footprint.

5 New MPI CubeLib library

From previous chapters we have seen how the calculation methods work in the current library and its tools. If one requests *Inclusive* value, the library gathers one list of children and recursively calculates their *Inclusive* values and then it sums them up. We introduce a new parallelized approach for library methods and how `cube_server` could adapt to it.

5.1 Main idea

In the new version, the user will start the `cube_server` on compute nodes, on N ranks, where every rank opens the same cube file, enumerate the call tree identically and wait for requests. Requests are coming from the mock-task on rank 0, while the processing of the request, rank X loads data and calculates values only from the *own cnodes* and sends other subsequent requests to remote ranks for the value of *not own cnodes*. All initialization routines remain the same, hence the memory footprint for the initialization and opening workload is the same for all ranks. But as every rank calculates values from only its *own cnodes* as shown in formula (4.2), it loads only data from the call paths marked as its *own*, which shows the reason for improving the memory footprint. Here *own cnodes* stands if a process´s enumeration number matches its MPI rank and *not own cnodes* indicates the case if process's enumeration does not match its MPI rank.

In the new calculation method, we create two lists of *cnodes*: 1-) *local*-marked rank is equal to own rank, 2-) *remote*-marked rank is not the same as own rank. The working rank spawns tasks for every remote *cnode* and while they are being processed, it calculates values from the *local* list. Once the *local* calculation is done the library waits for the results from the *remote* task. This approach proposes overlapping of communication and computation in the general sense as mentioned in chapter 3.2, which is the reason for the computation performance improvement.

5.2 Asynchronous communication in CubeLib

As we split children list into two lists of *remote* and *local* values based on their association with the current MPI rank , formula (4.2) is now transformed into:

$$t_{incl} = t_{excl}^{own} + \sum_{children} t_{incl} = t_{excl}^{own} + \sum_{\substack{local \\ children}} t_{incl} + \sum_{\substack{calculted\ remote \\ children}} t_{incl} \qquad (5.1)$$

Each value from the *remote* children is now calculated asynchronously while simultaneously calculating values for *local* children *cnodes*. One waits for the asynchronous *remote* computations to finish, and it combines the results from *local* and *remote* contributions to calculate the total severity value. The resulting value is then sent back to the requesting process. The corresponding sequence diagram is presented in Figure 13.

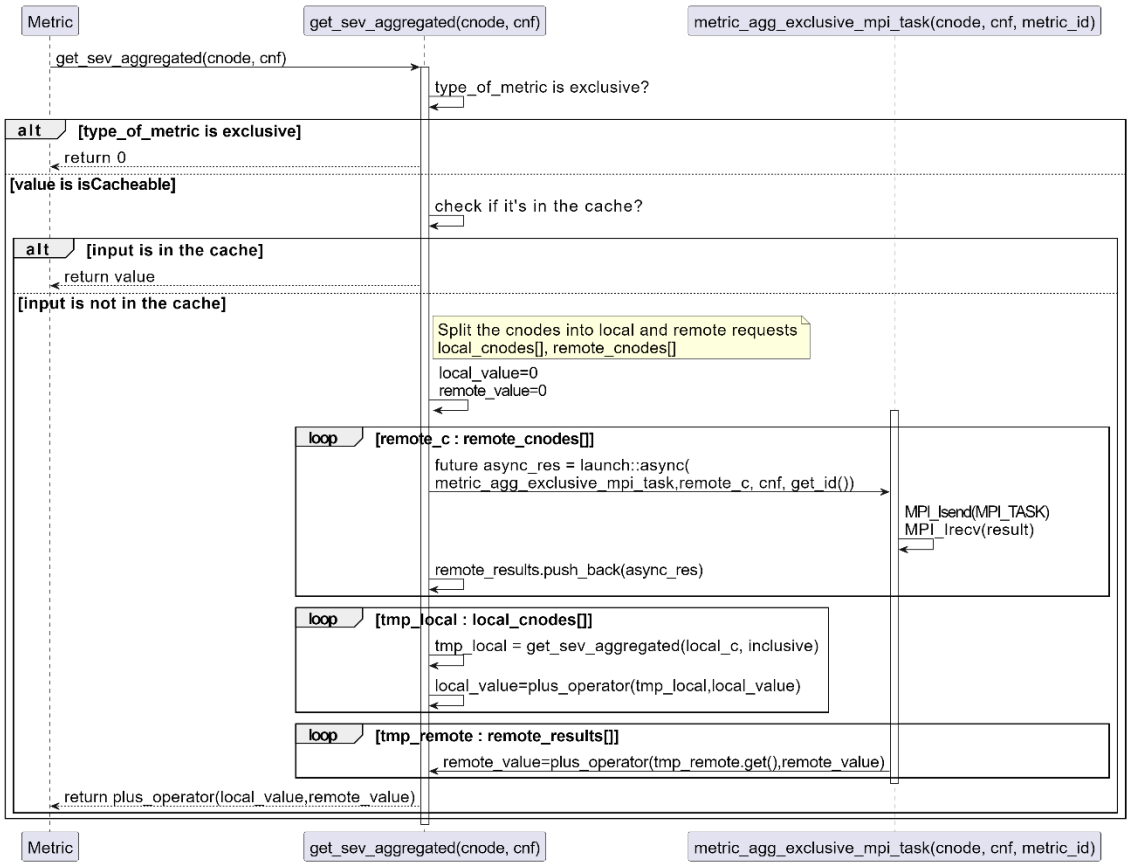

Figure 13: Suggested parallelization for internal steps of value calculation using C++ and MPI

5.3 Task distribution

The organization of user requests within a parallel processing system requires a sophisticated approach to task management. When the request comes from the user, process 0 will be handling input and output to the user as well as the mock task distribution. We use a combination of MPI and `<future>` library to achieve asynchronous task execution. A simple example of a flow diagram of task execution within a parallel processing system is explained in Figure 14. It shows different stages of task management and execution, indicating the order and manner in which tasks are distributed and processed.

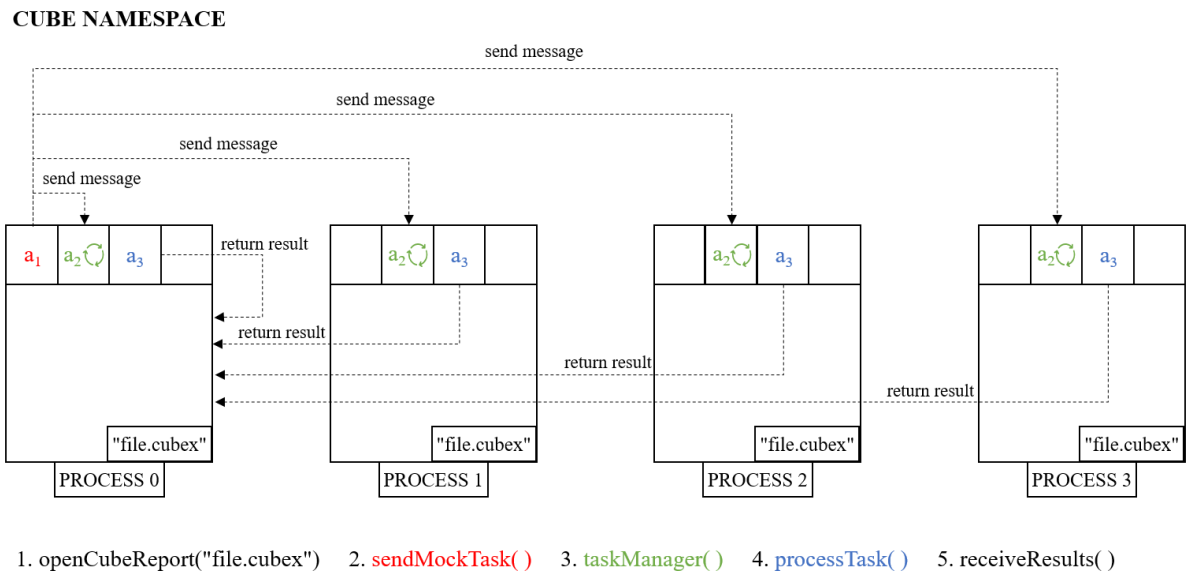

Figure 14: Workflow of Task execution

Figure shows the workflow diagram where squares labeled as *a1*, *a2*, *a3* represent threads and function calls that are being executed at the moment of Task spawning on 4 MPI Ranks. Process 0 is responsible for sending mock tasks and receiving the results.

The diagram describes a multi-stage process in a parallel computing environment, where tasks are managed and executed concurrently. Initially, tasks are generated by the `sendMockTask(…)` function, indicating the simulation of task creation for the purpose of the workflow demonstration as described in chapter 4.3. These tasks are then passed to a `taskManager(…)`, a thread which runs asynchronously, responsible for receiving and initializing the task execution . Thread is in a constant `while` loop, waiting for a killing signal to be sent by process 0, where it will break its execution. Every process continues to listen for incoming messages using `MPI_Iprobe(…)`. Then, `processTask(…)` is being invoked with `std::async` function and launch policy is set to `std::launch::async` which signifies the actual execution of the tasks by the receiving process. When task computation is finished, the result is sent back to `process 0`, which uses `receiveResults(…)` method. The workflow is indicative of a typical parallel processing pattern where the task generation, management, and execution phases are distinct yet function in an integrated asynchronous communication.

5.3.1 Call tree parallelization and tree structure benchmarks

In selecting the optimal dimension for parallelization among the three available options, namely metric, call tree, and system tree, parallelization along the call tree dimension emerges as the most suitable choice. The reason for this can be broken down into cases of why not to choose the other two in the current configuration:

1. *Metrics* tree parallelization - *Cube4* file contains only 11 metrics, in remapped version more than 100, so we would only utilize up to 100 MPI ranks. Additionally, the size of the *Cube4* file originates from the call or system tree. In *CubeGUI* the majority of operations happen to one selected metric, and other metrics are idle, hence parallelization along the metric tree would

result in consistently selecting different ranks for what would otherwise be serial computations. This results in data parallelization but not parallel computation.

2. *System* tree parallelization - it is too homogeneous, resulting in no computations being performed within the system tree pane since the entire tree is displayed. Furthermore, for the computation of values in the call tree, it would be communication between almost all participating ranks, as the call tree pane displays aggregated values over the entire system tree.

Therefore, it seems natural to implement parallelization along the call tree, leaving on the back of the head, that other options are possible but not necessarily better.

The maximum number of ranks that can be used for parallelization is less than or equal to the total number of *cnodes* (N_{cnodes}), as one process can hold a minimum of one *cnode*. Our data set, performance profile in form of a Cube4 file, structured as a tree, requires us to distribute the workload among the ranks (processes) effectively. These trees, as shown in Figure 15, can exhibit either balanced or non-balanced structures, thereby influencing the approach to parallelization. We examine three degenerated cases: 1-) linear tree, 2-) single-level tree 3-) binary tree. These *cubes* will serve as artificial benchmarks, allowing us to pin-point the influence of the tree structure on the performance.

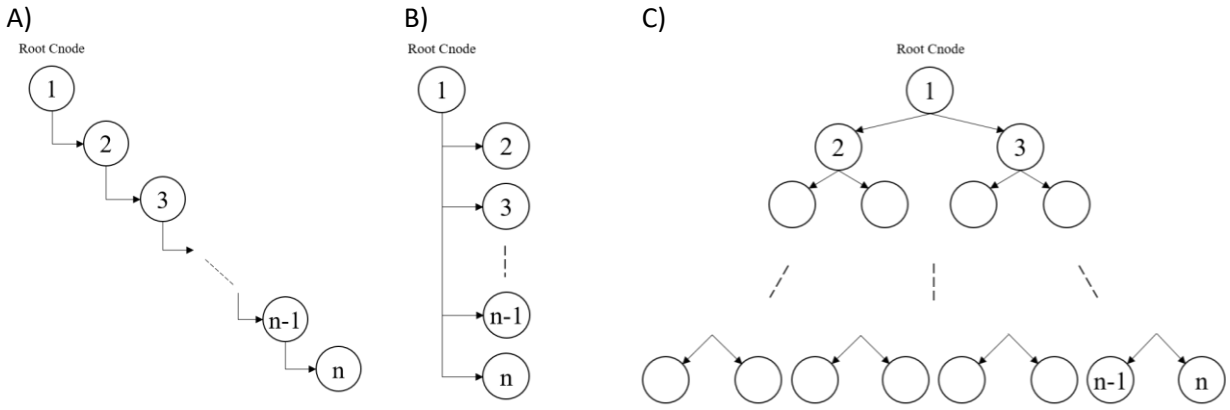

Figure 15: Different Tree structures

A) Linear tree, B) Single level tree and C) Binary tree structure types of Call Paths (*cnodes*) respectively shown in figure above.

The distribution methods ensures that each MPI rank gets almost equal number of nodes for the call tree, meaning that if $N_{ranks} \leq N_{cnodes}$, we will distribute N_{cnodes}/N_{ranks} per process.

5.3.2 Enumeration methods

Tools like *Score-P* and *Scalasca* generate the call tree in a *Depth-first-search* (DFS) order, although this is not always the case. However, benchmarks are provided with a BFS order of Ids, specifically for these performance studies. This enumeration is derived from the method used to create the call tree during the development of a Cube4 file. It's important to consider this difference when analyzing the results.

Within the scope of this thesis, we study different enumeration strategies which are: *Plane enumeration, Round-Robin*, *Random shuffle [44]* and little bit modified *Depth-first-search* as we name it *Deepest chain*.

Breadth-first-search (BFS) is a method used to search for a specific node within a tree data structure that meets a predetermined condition. The process begins from the root of the tree and examines all

nodes at the current depth before advancing to those at the next depth level. Because we calculate an *Inclusive* value, the call path tree is already sorted in BFS fashion as we described in chapter 4.2.3. Therefore, we simplify the enumeration by doing a plain distribution of *cnodes* as shown in Figure 16. The figure uses red color numbers to denote the MPI ranks (0, 1, 2, 3), and the blue color numbers to indicate the position Id of each call path (node) in a list of call paths. This enumeration is derived from the method used to create the call tree during the development of a Cube4 file.

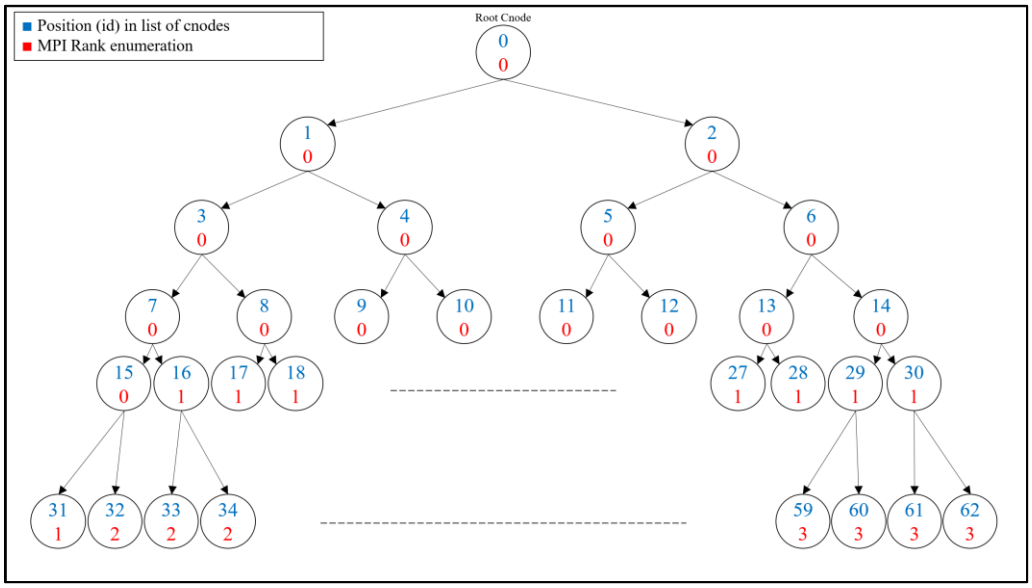

Figure 16: Plain enumeration (BFS) of MPI Ranks

The above figure shows a Binary tree with 63 Call Paths (*cnodes*) and *Plain* enumeration method of 4 MPI Ranks.

Round-Robin distribution is a method where processes, numbered from zero up to $N_{ranks} - 1$, are sequentially assigned to each node within a call tree. For example, let's assume tree has N_{cnodes} and N_{ranks} ranks, where $N_{ranks} \leq N_{cnodes}$. Assignment of ranks to nodes starting from 0 to $N_{ranks} - 1$ is in a cyclic manner. This means after reaching the $N_{ranks} - 1$th rank, the rank starts again from 0. The formula to calculate the rank of each node would be

$$rank(i) = i \ mod \ N_{ranks} \tag{5.2}$$

where *i* is the index of the node, ranging from 0 to $N_{cnodes} - 1$.

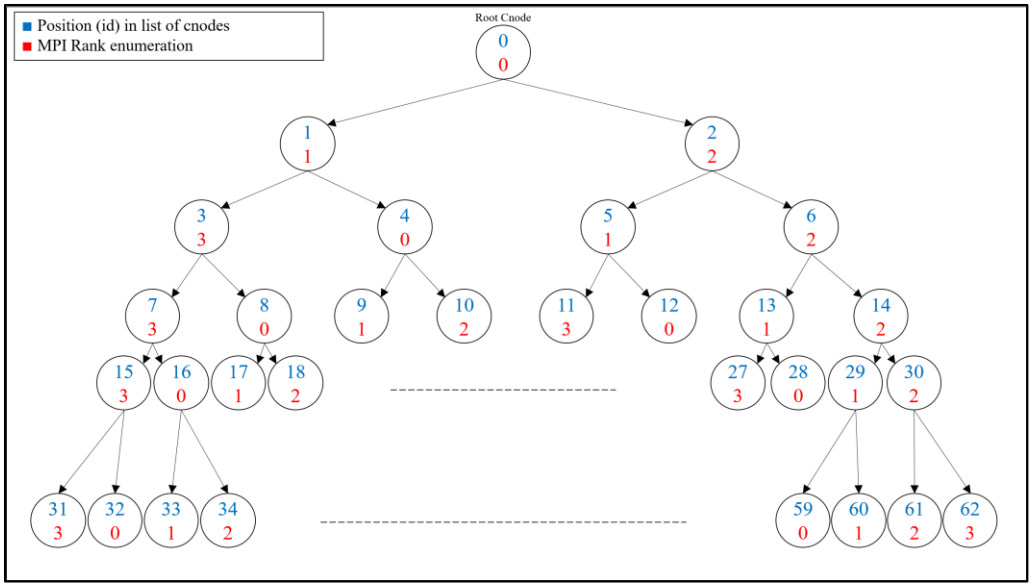

Figure 17: Round-Robin enumeration of MPI Ranks

The above figure shows a Binary tree with 63 Call Paths (*cnodes*) and *Round-Robin* enumeration method of 4 MPI Ranks.

Depth-first search (DFS) is an approach for exploring or locating elements within tree or graph data structures. It begins at the root node and enumerates each branch to the fullest extent before reversing direction. It will be called the *Deepest chain* method after this point, as it is modified compared to DFS in terms of additional parameter which is leverage. Users can select in cases of non-balanced structures how many extra nodes can be assigned to a single rank, instead of trying to uniformly distribute them. This leverage can sometimes bring more deeper chain of *cnodes* that are assigned to a single rank.

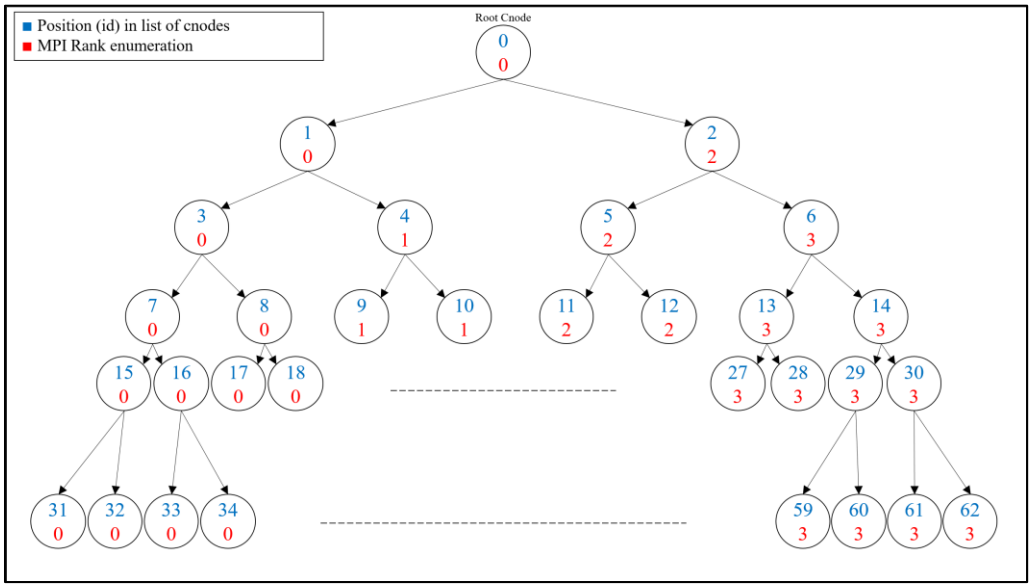

Figure 18: Deepest chain enumeration of MPI ranks

The *Deepest chain* method is believed to outperform the other algorithms up to a certain threshold. This is attributed to its strategy of segmenting the tree into subtrees where such a distribution promotes an early and efficient overlap of computation and communication, optimizing the overall processing time.

The tree structure given in Figure 18 presents the 64 call paths distributed across 4 MPI ranks in a *Deepest chain* method. The position Id is a way to index or reference each call path in this list. This means that the distribution of tasks is prioritized to assign the longest sequence of dependent tasks to the same rank, which can help in maintaining data locality and potentially optimizing communication overhead. The root of the tree, at the top, is the starting point of the computation and has the MPI rank of 0. From there, the computation branches out, with each level of the tree representing a further subdivision of tasks.

Random shuffle, as the name says, assigns processes randomly across the call tree but every process is assigned almost the same amount to the *cnodes*.

Furthermore, the *Random shuffle* algorithm in this thesis is designed to evenly distribute nodes across ranks. The number of nodes assigned to each rank is determined by dividing the total number of *cnodes* in the vector by the number of processes involved. When this distribution coincides with a system characterized by low communication overhead, the *Random shuffle* algorithm stands to match the performance of the *Deepest chain* method, possibly even surpassing it under optimal conditions. The inherent randomness can lead to a natural balance in the distribution of computational workloads, which is particularly advantageous when the tasks vary significantly in their computational demands. This balance ensures that no single rank is overburdened, promoting an efficient use of resources, and potentially optimizing overall communication overhead.

Of course, many other tree searching algorithms exist. We investigate the influence of these methods on parallelization performance and possible implications. Therefore, the focus is placed on the evaluation of their performance in terms of execution speed and memory usage within the given computational framework.

5.4 Integration

The current *CubeLib* library is written without MPI or OpenMP integration; however, this is now changing as all processes will utilize the library simultaneously. We have reworked the calculation components and the main `Cube` initialization constructor. As a result, all methods that were public remain unchanged. They are just reconfigured based on the new calculation and tasking methods. This approach also works on one process without the MPI which results in the original library. The source code can be seen in A Appendix - source code.

In the revised setup, each process initiates the cube object and the struct has input parameters such as MPI rank and the total number of processes. The new initialization looks like this `Cube cube(cube_mpi_rank, cube_mpi_processes).`

When `cube.cubeOpenReport(…)` is invoked, all data is read by the processes and all ranks enumerate the *cnodes* in the very same manner, depending on the user's choice of enumeration methods. A notable distinction arises during the calculation phase, where specific processes execute their designated tasks and then engage in communications with other processes to implement a new calculation algorithm. Consequently, each process is equipped with a `taskManager(…)` method that operates within a while loop, actively probing for incoming mock and internal tasks. These tasks, characterized as MPI datatype objects, consist of seven parameters: `taskId`, `metricId`, `metricCalculation`, `cnodeId`, `cnodeCalculation`, `systemId`, and

`systemCalculation`. Upon detection of a task, the corresponding process receives it and begins processing with the `get_sev(…)` method while remaining alert to new tasks.

Depending on input parameters, Internal calculations can be initiated. Communication between processes may occur, depending on how many processes are involved as well as the distributing methods of the *cnodes*. The `get_sev_aggregated(…)` method can be invoked by any process, with the *cnode* and the desired calculation flavor as its inputs. This method is optimized by dividing the *cnode* vector of children into two vectors representing local and remote values. As a first step, an asynchronous request for remote values is made, followed by the execution of local calculations. After receiving all the results from remote computations, the calculation concludes with the application of either a plus or minus operator, depending on the metric's type. The process that holds the final result sends it to process 0, which initiates the mock request. The cube operation concludes with a termination signal from process 0, indicating the end of the while loop after the final results are received. With the implementation of this MPI-integrated *CubeLib* library, the traditional configuration of the library has been updated, requiring users to specify MPI compilers.

6 Results and discussion

6.1 Experimental setup and configuration

The new library underwent testing on the JURECA [45] supercomputer located at Forschungszentrum Julich. JURECA uses a combination of Slurm (Simple Linux Utility for Resource Management) [46] and Parastation [47] for workload management on the available resources.

Figure 19: Jülich Research on Exascale Cluster Architectures (JURECA) at Jülich Supercomputing Centre
(Copyright: Forschungszentrum Jülich / Ralf-Uwe Limbach)

Nodes	CPU	RAM	GPU
480 standard compute nodes	AMD EPYC 7742, 2×64 cores, 2.25 GHz clock speed	512 (16× 32) GiB DDR4, 3200 MHz	-
96 large-memory compute nodes	2× AMD EPYC 7742, 2× 64 cores, 2.25 GHz clock speed	1024 (16× 64) GiB DDR4, 3200 MHz	-
192 accelerated compute nodes	2× AMD EPYC 7742, 2× 64 cores, 2.25 GHz clock speed	512 (16× 32) GiB DDR4, 3200 MHz	4× NVIDIA A100 GPU, 4× 40 GB
12 login nodes	2× AMD EPYC 7742, 2× 64 cores, 2.25 GHz clock speed	1024 (16× 64) GiB DDR4, 3200 MHz	2× NVIDIA Quadro RTX8000

Table 2: JURECA system information

In total, JURECA is configured with 780 compute nodes. and 98,304 CPU cores with a peak performance of 3.54 petaflops. Users are required to submit batch applications (shell scripts) to dispatch jobs to compute nodes. An illustrative example of a simple batch script is provided below:

```
#!/bin/bash
#SBATCH --account=<budget>
#SBATCH --nodes=2
#SBATCH --ntasks=128
#SBATCH --ntasks-per-node=64
#SBATCH --output=mpi-out.%j
#SBATCH --error=mpi-err.%j
#SBATCH --time=00:15:00
#SBATCH --partition=dc-cpu

srun ./mpi_cube_server_prototype --cpu-bind=threads
```

In this illustration, the script runs the `mpi_cube_server_prototype` file on 2 nodes with a total of 128 MPI tasks (64 tasks per node). Output and error streams are saved with names containing the job ID (%j). The execution is assigned to the dc-cpu partition and is restricted to a maximum time of 15 minutes. Additionally, the script ensures optimal performance by binding CPU threads during execution. JURECA is linked to the Juelich Storage Cluster (JUST) [48] with a storage bandwidth of 350 GiB per second. JUST functions as a central GPFS fileserver, supporting supercomputing systems like JURECA, JUWELS [49], JUSUF [50].

6.2 Score-P Instrumentation

In order to do performance measurement such as timings and memory allocations of the new library, we used *Score-P* profiling tool [51]. It is important to mention that both *CubeLib* library and *Score-P* are configured with the same compiler on the system, otherwise measurement is not possible. For this setup, we are using Intel compilers, and its MPI version. To get a good profile, we filter out all the unnecessary files and methods from *CubeLib* library during the run time. This is actually the step after the "initial measurement", which is used to reduce measurement overhead. Such a filtering script looks like this:

```
SCOREP_REGION_NAMES_BEGIN
  EXCLUDE *
  INCLUDE cube::openCubeReport*
  INCLUDE cube::Cube::get_sev*
  INCLUDE cube::Cube::task_manager*
  INCLUDE cube::process_task*
  INCLUDE main*
  INCLUDE pmpi*
  INCLUDE *Exclusive_sev_aggregated
  INCLUDE *Inclusive_sev_aggregated
  INCLUDE *Mock_task
  INCLUDE *Internal_agg_metric_calculation
  INCLUDE *Kill_Signal
  INCLUDE cube::metric_agg_inclusive_mpi_task*
  INCLUDE cube::metric_agg_exclusive_mpi_task*
  EXCLUDE *cube::Cube::get_sev_adv
SCOREP_REGION_NAMES_END
```

We first exclude all of the files and methods from runtime measurement and include only methods that are relevant to us such as aggregation calculation methods, `openCubeReport(…)`, `get_sev(…)`, `task_manager(…)`, `processTask(…)`, etc.

6.3 Performance measurement of new library

To have a better understanding of the results and to determine an optimal number range of MPI ranks for future use of the library, we test performance of the implemented algorithm on benchmark cubes that have linear, single-level and balanced tree (or special case - binary tree) structure of the call paths. To simulate communication with the client, we mock the connection to it and pretend that connection is established, since the current `cube_server` is not adapted to the new parallel library. A script where the user decides the input parameters for MPI Task is being used as simulation for client-server communication.

We initialize the MPI environment with the highest instance of thread safety `MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided)`, and define a `MPI_Task` with its parameters. Parameters will always be set up to calculate the root *cnode Inclusive* value for an *Exclusive* type of metric, as it is the most compute intensive operation and further calculations are "given away" afterwards due to internal caching. In the case of an *Exclusive* value for Inclusive metric, only children of the root node in the call path tree are included in task computation as given in formula (4.2).

6.3.1 Prototype timings

To study the influence of the different aspects of the structure of the call tree on the parallelization efficiency, we do measurements on the artificial benchmark cubes. Then in the following chapter we measure the timings of the real-life cubes. First, we investigate the timings of every rank enumeration algorithm and its scaling capabilities. All the figures are represented on a log-log scale, and the timings value in seconds, are taken from *Score-P* measurements profile. In the profile, we look for MPI rank onto which root *cnode* was distributed and check the timing for its first thread that initiated the `process_task(…)` method. That is our mock task coming from the user and corresponds to "time to result" timing, which corresponds with the reaction time of the client. Hence, as short this time is, as more responsive the client (*CubeGUI*) is, which leads to comfortable and efficient performance experience within the *CubeGUI*.

Important to notice is that every method starts at the same point as we have only 1 MPI Rank available, therefore the timing of 1 Task doesn't depend on the enumeration method rather the total number of Call paths and Locations. All methods end in the same point, since $N_{ranks} = N_{cnodes}$ therefore the enumeration methods don't play a role. If $N_{ranks} > N_{cnodes}$, then $N_{ranks} - N_{cnodes}$ ranks become idle, and we don't observe such scenarios.

Result 1.1: In the three results presented, we explore the timing outcomes for prototypes and how different call tree structures affect it, keeping the total number of *cnodes* and locations fixed. Figure 20 begins the analysis with a linear tree structure, indicating that parallelization might not be required. This is due to a steady increase in timing, which is related to the tree structure. Both computation and communication become expensive, leading to longer processing times for each method.

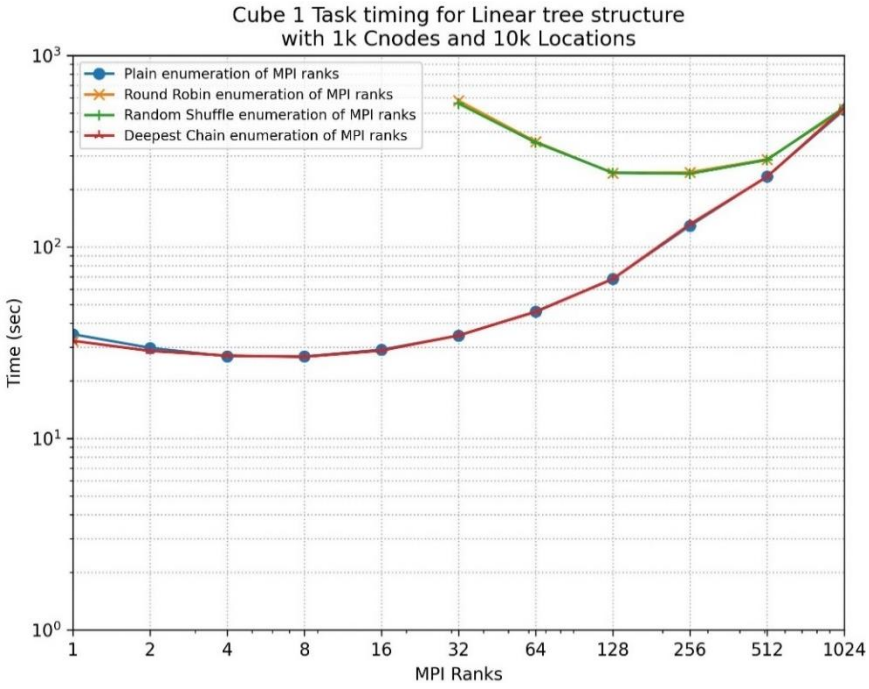

Figure 20: Task timing of enumeration methods on linear tree

What we can observe is that the Plain enumeration (or BFS) and *Deepest chain* are equivalent, but due to communication overhead timings increase, as only one process is executing the task. In this scenario, we will have that the total number of communications is $N_{comm} = N_{ranks} - 1$ but, due to tree structure, we result in serial computing. The performance of the other two methods is less effective, primarily because the communication overhead significantly increases. This is because one process is responsible for computing one *cnode* at a time, and $N_{comm} = N_{cnodes} - 1$. This setup leads to serialization and overhead in both aspects. While other *cnodes* are calculated, previous calculation tasks are "on hold", occupying memory and resources. which leads to the crash of Score-P (version 8.1). Unfortunately, their analysis graph could not be completed due to time constraints on JURECA and the memory limitations of Score-P.

Let's take a close look, why it happens. In Figure 21, we have an example of 16 *cnodes* and 4 MPI ranks, and 2 different enumeration methods-Plain and Round-Robin enumeration. Computation of values is done in a recursive manner as shown in formula (5.1), with a split of local and remote values. In case of plain enumeration, rank 0 starts calculating, but has to wait for rank 1 to finish its part. But again rank 1 cannot finish until rank 2 does, and this goes on until rank 3, which holds the last value that is necessary for calculation. Then the communication goes backwards, resulting in only 1 process working at the given time, no parallelization whatsoever. In the second picture, with Round-Robin enumeration, every rank needs to communicate to the next one at each calculating step, making them all wait for each other. This is an even worse scenario as communication time gives larger overhead. Such results lead to never having the full split of local and remote contributions at the same time for a given *cnode*, as it only has one direct child.

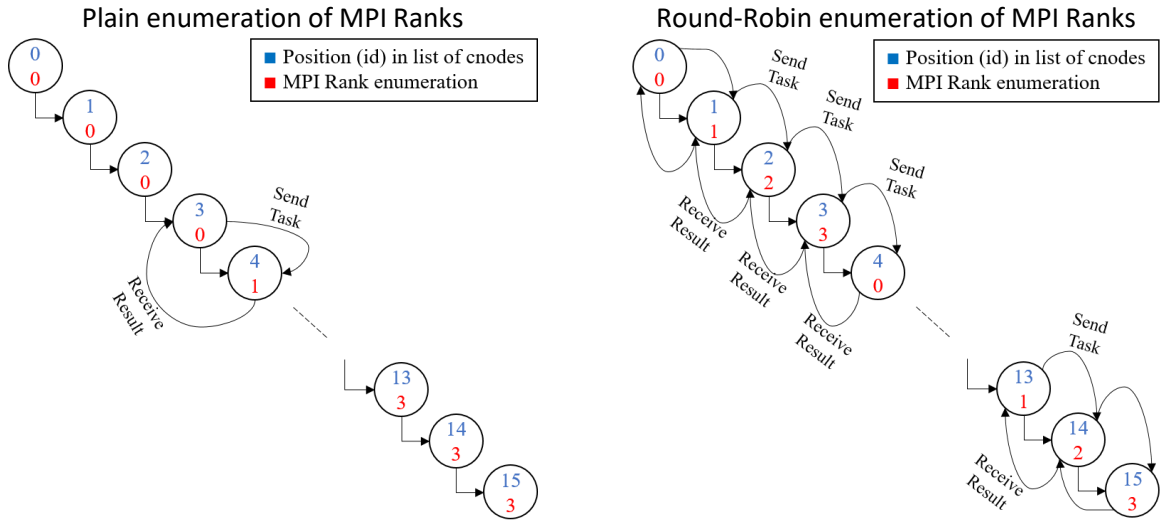

Figure 21: Example of Plain and Round-Robin rank enumerations on linear tree

Result 1.2: How the proposed parallelization algorithm performs for another degenerate case, for the case of a single level tree, which is quite opposite than the linear tree. Single level tree has one root *cnode*, and the rest *cnodes* are direct children of it. The rank enumeration doesn't play a role as all *cnodes* , except the root *cnode*, are leaves. Meaning, that the timings for all enumeration methods in this setup are very similar, and the total number of communications is $N_{comm} = N_{cnodes} - \frac{N_{cnodes}}{N_{ranks}} - 1$. All the additions are handled by the root process.

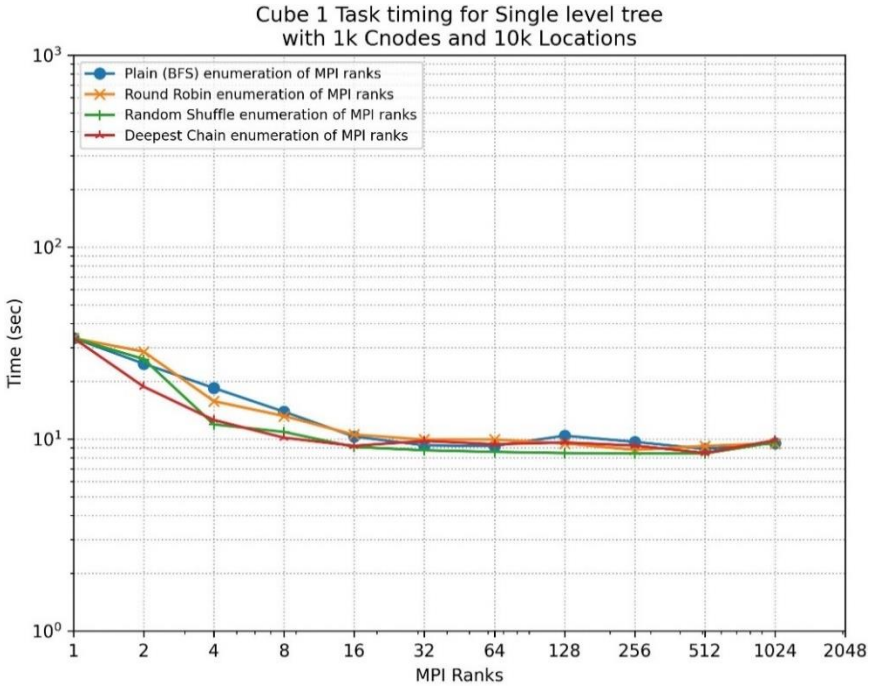

Figure 22: Task timing of enumeration methods on single-level tree

We notice that as the number of ranks increases, the timings become flattened, but parallelization still results in faster execution. This outcome is the result of spreading the workload among various processes. Yet, reducing the workload for each process to an even smaller amount doesn't impact the timing anymore. This occurs because the root process has already summed up its local contributions, and then proceeds to request remote values at a specific moment. The performance gain due to parallelization reaches minimum early on, and it is sufficient to utilize `cube_server` with a relatively small amount of MPI ranks, in this case would be 16 MPI ranks.

Result 1.3: The timings for the binary tree are measured five times, and the graph includes error bars representing the standard deviation for each data point. By repeating measurements multiple times and averaging the results, we can better isolate the actual performance characteristics of the parallelized application from the background noise inherent in the system. We can see similar performance of the *Round Robin*, *Random shuffle* and *Deepest chain* methods which indicates that the communication cost is not the primary bottleneck in this context, and that the computational workload is more significant as when it happens during execution. Communication and computational overlap is being visible, as different *cnodes* that are located on the same rank are being computed simultaneously.

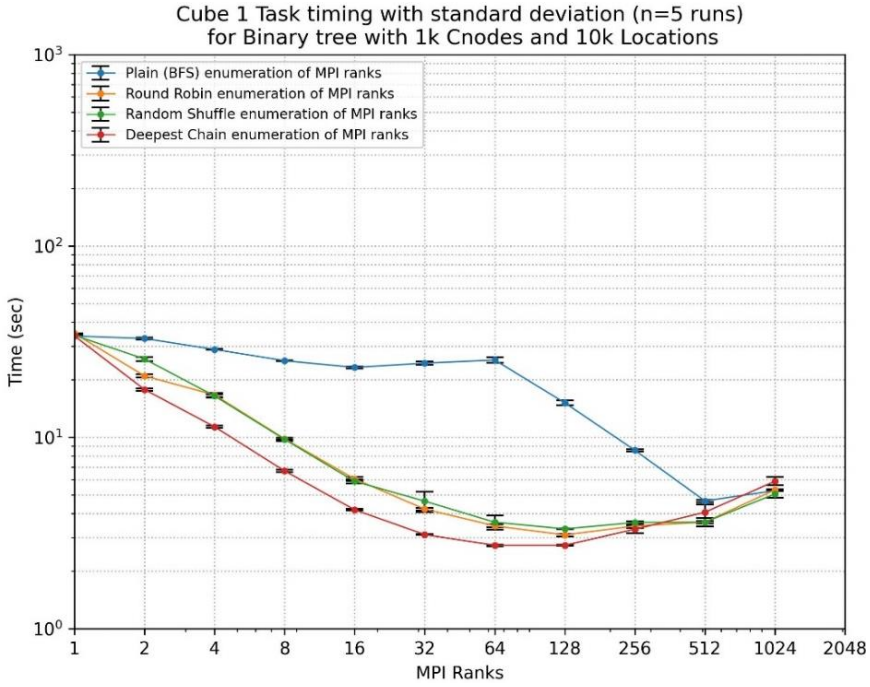

Figure 23: Task timing of enumeration methods on binary tree

The chart displays four enumeration methods: BFS (Plain distribution), Round Robin, Random Shuffle and Deepest chain with their timing's values for different number of processes. Cube file has 1024 Call paths and 10000 locations. The points are represented with standard deviation as every method is measured five times.

The reason for the *Plain* enumeration method to be the worst in this scenario, is that with a small number of MPI ranks, the top of the tree is being processed by the rank 0. Hence it is serialization by rank 0 with small parallelization when calculation is deep in the call tree, and only with sufficient number of ranks depending on depth of call tree and total workload, rank 1 appears high enough in three to enable parallelization of the computation. After increment of ranks it starts to speedup due to less computation,

and then it merges with other algorithms as workload and communication start to overlap in the early stage of distribution.

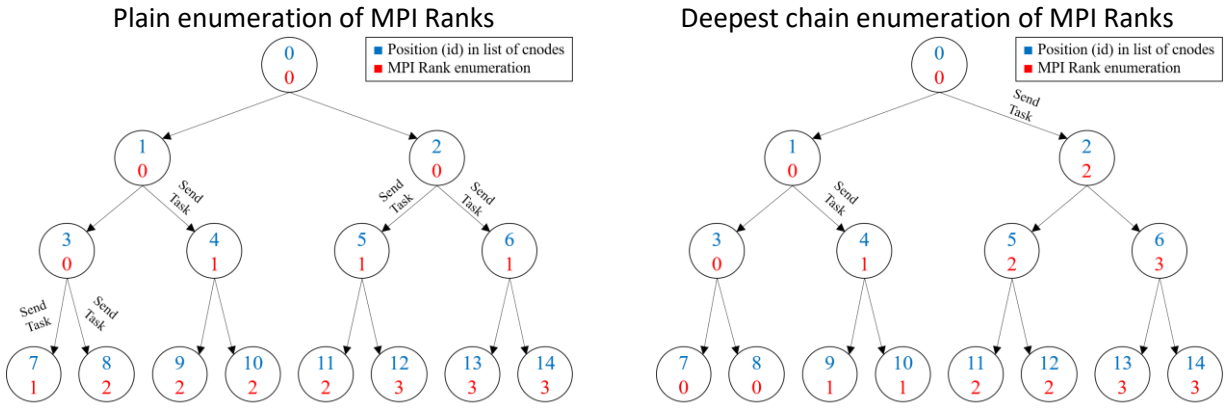

Figure 24: Example of Plain and Deepest chain rank enumerations on binary tree

The *Random shuffle* can result in a communication pattern that is sufficiently efficient, such that the additional optimization provided by the *Deepest chain* method does not translate into a substantial performance improvement. In general, parallelization results in timing speedup up to a certain point, where it starts to vary depending on the number of ranks involved. The interval where minimum is reached is between 64 and 256 MPI ranks.

To visualize the parallel computation, an example in Figure 25 can be seen. For 4 MPI ranks, and 15 *cnodes*, rank 1 starts computing two tasks, very early in computation, but also it computes two more tasks at the very bottom asynchronously. Therefore, such task distribution leads to speed up even for random rank enumeration.

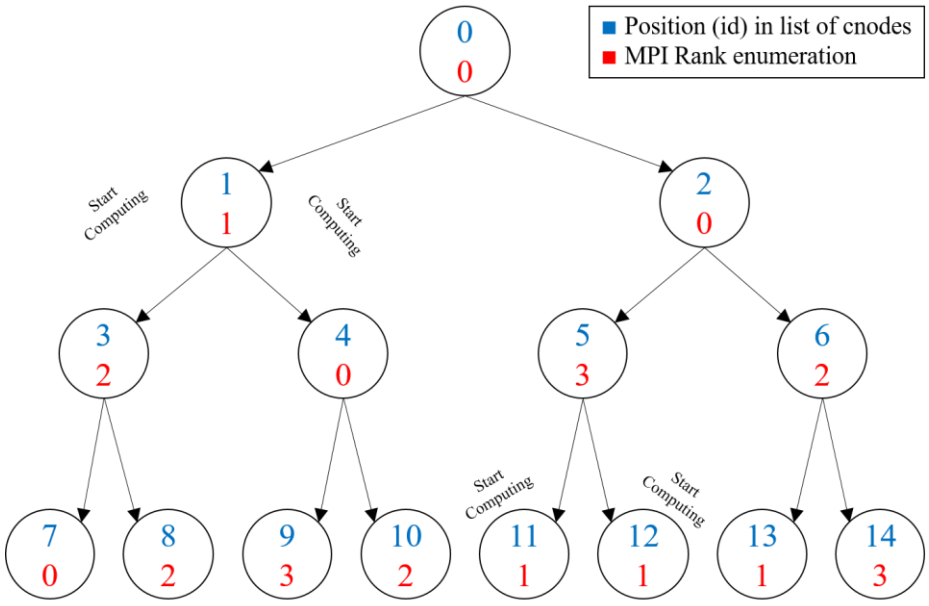

Figure 25: Example of Random shuffle rank enumeration method on binary tree

Results also suggest that as the number of MPI ranks increases, the execution time between the various methods diminish, indicating that the impact of the task distribution strategy decreases with higher levels of parallelism. This is because the communication overhead becomes expensive, since we have less workload per process. We can filter our selection of enumeration methods based on our findings: *Deepest chain* emerges as the most promising, while *Random shuffle* stands out as a reliable option capable of adapting to tree structures.

Result 2: We measure timing on different balanced and non-balanced tree cube types, with increment on children per *cnode* and fixed number of total *cnodes*, using *Deepest chain* enumeration method. With the start from the binary tree, and as the depth of the tree reduces, timings intend to converge to the single level tree.

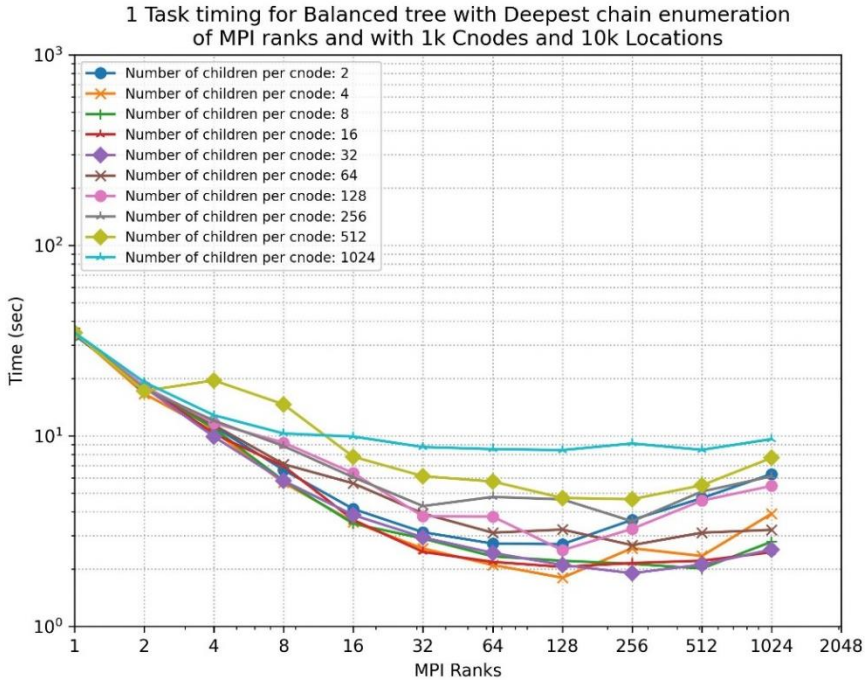

Figure 26: Task timing with Deepest chain and balanced tree

In this setup, for $N_{cnodes} = N_{ranks}$, results don't end in the same point, since communication patterns change, as we have more *cnodes* on one side of the tree. A detailed example of such a tree and communication pattern, with 4 MPI ranks, 32 *cnodes* and 4 children per *cnode*, can be seen in Figure 27. Rank 0 has the most levels in terms of deepness of the tree, therefore his remote calculations are going to be faster computed, resulting in overlap of its local contributions.

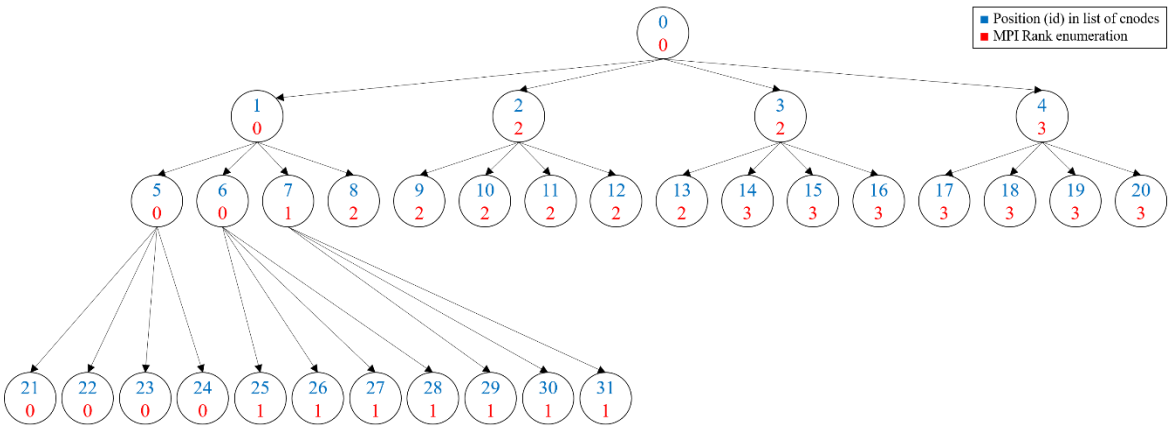

Figure 27: Example of Non-Balanced tree with Deepest chain enumeration method

The more trees get flatter, the ranks that are on leaves finish their own calculation, and parent ranks need to handle all the summations, making the total value computation workload more significant. Communication becomes expensive, resulting in increased timings. Despite this, we still observe some overlap, indicating that parallelization continues to be beneficial, especially on intervals between 16 and 128 ranks.

Result 3.1: We continue studying the timing of task within a binary tree structure using *Deepest Chain* enumeration of MPI ranks with a fixed system tree size of 10k locations and varying numbers of *cnodes* (1024, 2048, 4096, and 8192). The timings measured on 1 Rank now start from different points, as the workload increases, and the curves don't meet at the end.

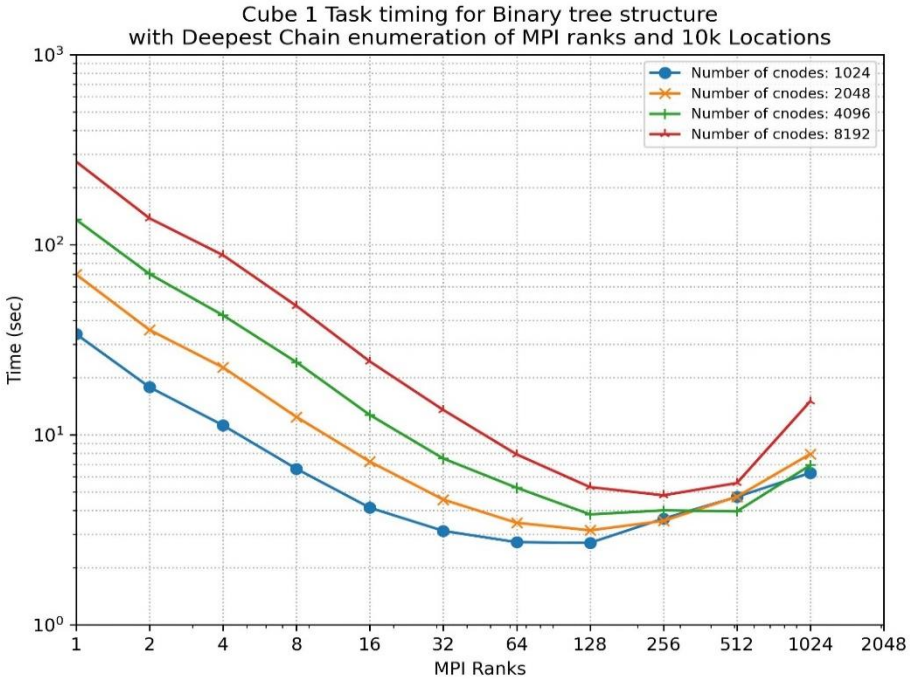

Figure 28: Task timing with Deepest chain and binary tree for different number of cnodes

This increment of timing is expected due to the large number of computations per rank, as we have more *cnodes* to compute, but the number of communications remains unchanged. Hence, the method keeps the behavior of the curve throughout different cube structures with a change to reach the minimum, where points usually shift up right as computations become heavier. Interval in which the minimum is located is usually between 64 and 256 processes. This observation will help us to determine the optimal interval of processes for future setups. Although it is possible to extend measurements on more than 1024 MPI ranks, as there are more than 1024 *cnodes*, we simply stop there since the timings start increasing again at the end points.

Result 3.2: In Figure 29, we study the impact of how different number of locations in system tree and fixed number of total *cnodes* affect the timings for binary tree structure using Deepest chain enumeration method of MPI ranks. In this setup, the timings measured on 1 Rank will start from different points, since the computation per *cnode* increases.

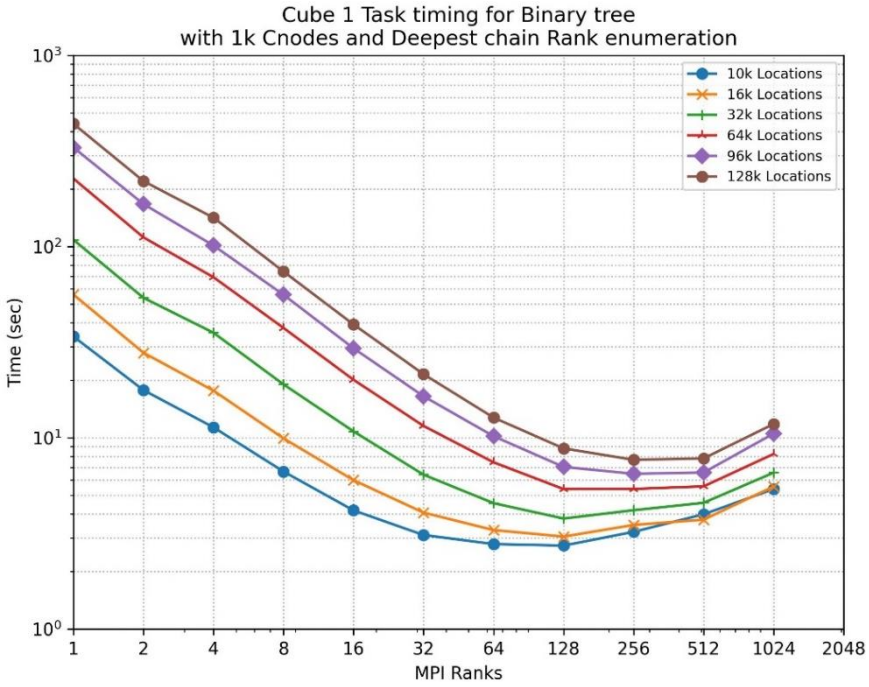

Figure 29: Task timing with Deepest chain and binary tree for different number of locations

We see the same curve behavior throughout different cubes, and we observe that with increasing number of locations individual and therefore total workload grows, hence the curve timings are increased. The lowest point moves upwards and to the right as the system tree grows bigger. Parameters in mock task are defined so that we always compute the call tree node in all locations. We find that the time it takes to complete the task is related not only to the size of the call tree but also to the size of the system tree.

Result 3.3: Using the same cube file structure, we perform test measurements but switch the enumeration algorithm to *Random shuffle*. Based on our earlier findings from result 1.3 with binary trees, we expect the *Random shuffle* method in this setup to be quite similar to the *Deepest chain* method.

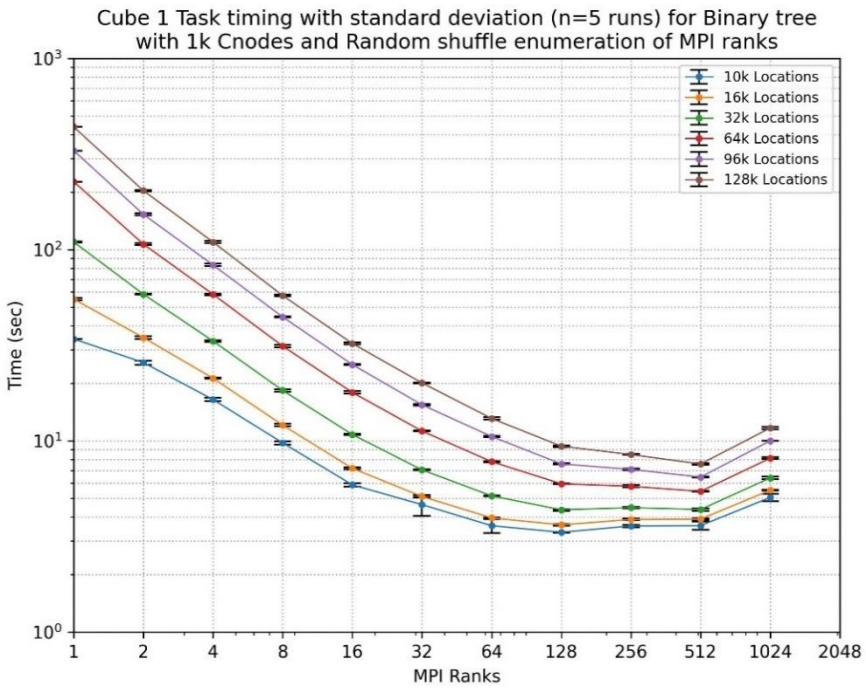

Figure 30: Task timing with Random Shuffle and binary tree for different number of locations

Timings have a minor increment compared to the previous measurement with *Deepest chain* method, but the curve behavior remains. Communication and computational overlap are a reason for this speedup of timings, as one rank is computing tasks in parallel. The measurements are repeated five times, as we want to isolate background noise inherent in the system. In the case of 10 000 locations, for 32 and 64 ranks, a bit larger deviation is seen, but it doesn't affect the stability of the method. Minimum shifts up right starting from 64 to 512 ranks, as the number of locations increase.

Result 4.1: We have seen from large cubes how the timings are measured, and in all cases, we see that finding minimum points is possible. We want to study the impact of relatively small, micro cube files on the task timings. Cube files in Figure 31 have only 64 *cnodes* and different number of locations in the system tree, and ranks are enumerated with the *Deepest chain* method. The measurements don't start at the same point, as we have less workload per process, but they all end up at the same point. This is because for the smallest case, communication becomes expensive and as we increase the computation on each node, the task timing becomes dominated by the computation rather than the communication or the structure of the enumeration method. Therefore, the difference in initial task distribution becomes less relevant, and the overall task time converges because the majority of time is spent on node-level computation.

For such small cubes, we see that parallelization is almost not necessary, as communication becomes overhead, due to fast computations of tasks. With the increment of the total number of locations from 8000, the minimum point becomes visible on 8 processes as computational workload becomes more dominant.

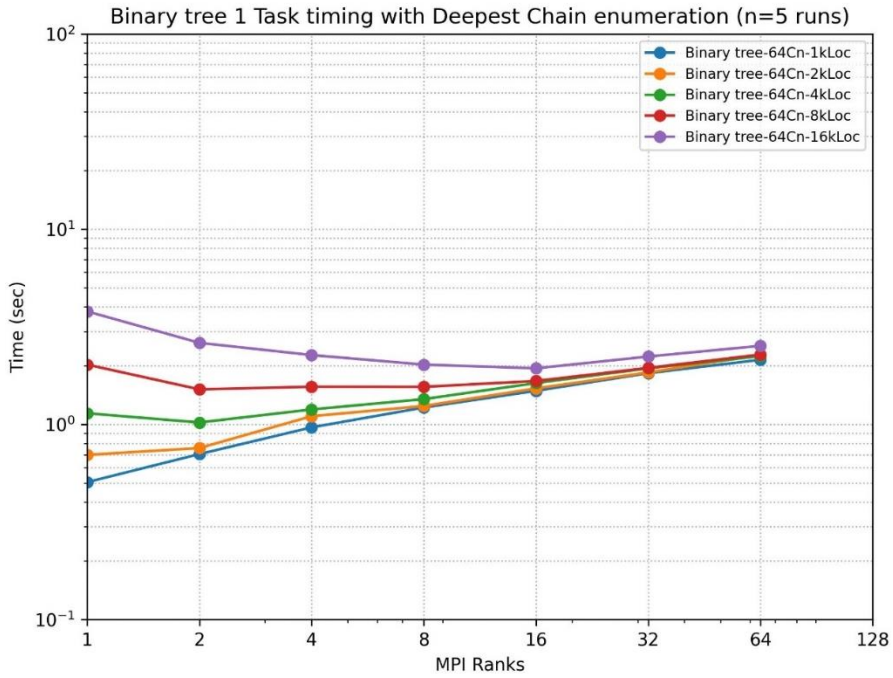

Figure 31: Task timing with Deepest chain and small binary tree for different number of locations

Result 4.2: In Figure 32, the same setup was repeated as in result 4.1 but the enumeration method of MPI ranks is *Random shuffle*. We see similar results as in the previous setup, therefore the influence of enumeration methods does not play a role for micro cubes. Once the workload becomes significantly large, the speedup can be seen on 8 MPI ranks.

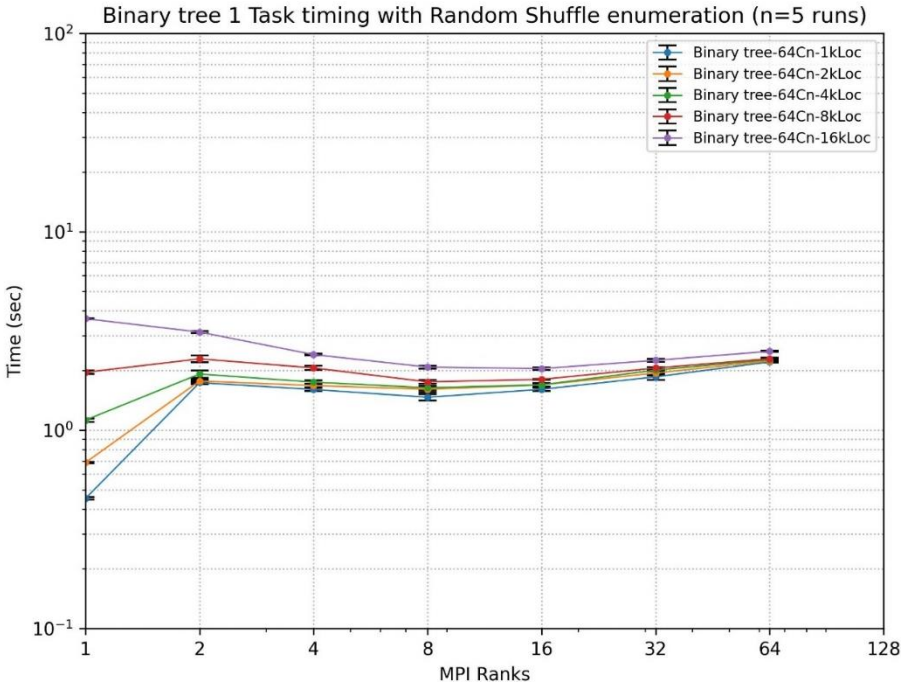

Figure 32: Task timing with Random Shuffle and small binary tree for different number of locations

Result 4.3: For the final measurement of task timing on the benchmark micro cubes, we study the impact of different call tree sizes with binary structure, on a total number of 1000 locations with *Random shuffle* enumeration method.

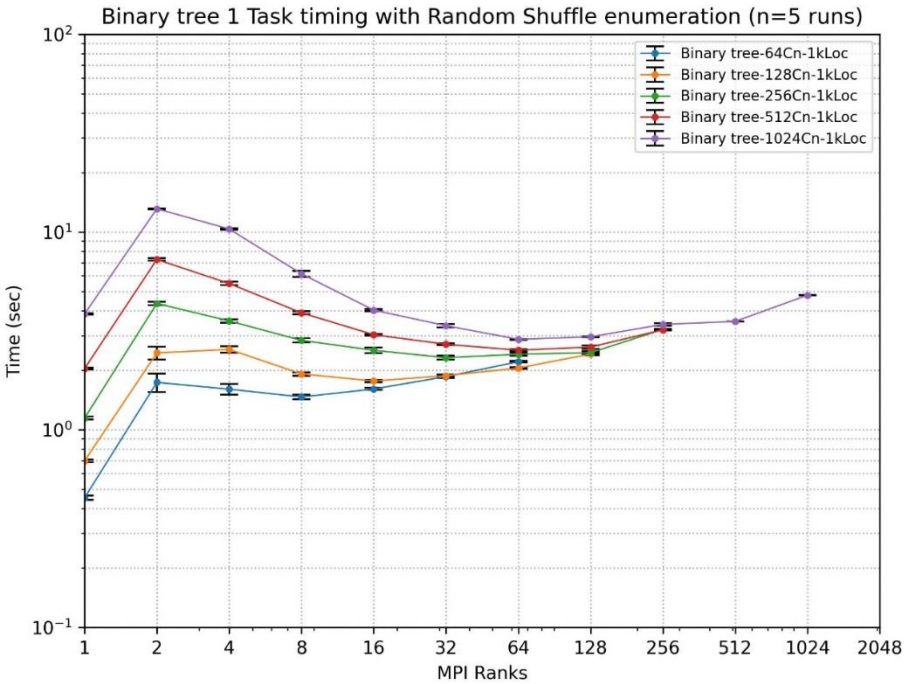

Figure 33: Task timing with Random Shuffle and small binary tree for different number of cnodes

The workload per *cnode* is now fixed, but for relatively small call trees timings go up. This result is a direct consequence of communication overhead, and it is persistent for all measurements on a small number of MPI ranks. Starting from 1024 *cnodes*, a minimum point is reached on 64 ranks. Resulting that if we have significantly small call and system trees, parallelization is not needed.

Result 5: Strong scaling plot is a common way to illustrate the effectiveness of parallelization in computational tasks and the results indicate that there is a limit to the benefits gained using the *Deepest chain* and *Random shuffle* methods. In Figure 34 such a scaling plot is presented. Initially, as the number of processors increases, the speedup also increases, but it does not keep pace with the ideal (linear) speedup. The plot shows that the actual speedup stops increasing for both methods and this is typical behavior in parallel computing where the overhead of communication starts to negate the benefits of adding more processors. Ideally, efficiency should be close to 1, indicating that additional processors are being utilized effectively and contributing proportionally to performance gains.

p	1	2	4	8	16	32	64	128	256	512	1024
$T_{parallel}$	33.85	17.79	11.21	6.62	4.13	3.12	2.72	2.7	3.62	4.71	6.3
$S = T_{serial} / T_{parallel}$	1	1.902754	3.02	5.11	8.19	10.85	12.44	12.4	9.35	7.18	5.37
$E = S / p$	1	0.951	0.724	0.638	0.512	0.339	0.194	0.096	0.036	0.014	0.005

Table 3: Speedups and Efficiencies

Speedups (S) and Efficiencies (E) of a Cube's 1 task on a binary tree with $N_{cnodes} = 1024$ and $N_{locations} = 10000$ with *Deepest chain* enumeration of MPI Ranks. $T_{parallel}$ denotes time spent in parallel execution of the task, and T_{serial} denotes time spent in executing the task on one process.

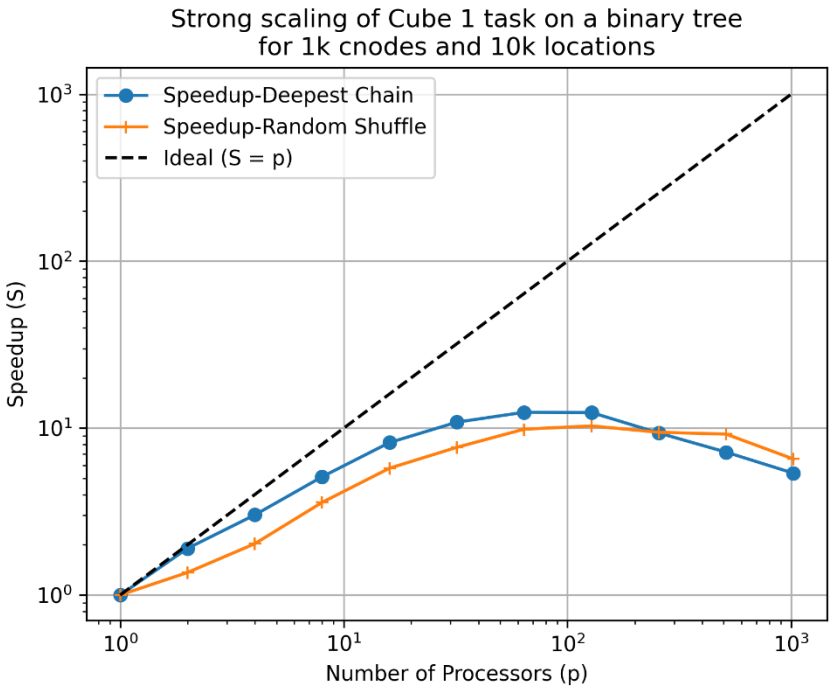

Figure 34: Scaling plot of Deepest chain enumeration method

6.3.2 Heuristic rules for estimating MPI rank interval

Now we can summarize previous results and give a proposal on what the optimal interval of ranks is, based on the Cube4 file structure and its size. Here are the following rules:

1. Baseline interval:
    - Start with a baseline MPI rank interval of 32 to 256 ranks as derived from previous performance analyses charts.
2. *Cnode* consideration:
    - If $N_{cnodes} \leq 1000$: Use the lower end of the baseline interval (16-64 ranks).
    - If $1000 < N_{cnodes} \leq 8000$: Use an interval that captures the plateau of performance before diminishing returns (32-128 ranks).
    - If $N_{cnodes} > 8000$: Consider extending the interval to 256-512 ranks, monitoring for diminishing returns due to overhead.
3. Location consideration:
    - If $N_{locations} \leq 10000$: Use the lower end of the baseline interval (16-64 ranks).
    - If $10000 < N_{locations} \leq 50000$: Use the mid-range of the baseline interval (32-128 ranks).
    - If $N_{locations} > 50000$: Use the upper end of the baseline interval (192-256 ranks) or higher if N_{cnodes} is also large, monitoring for diminishing returns due to overhead.
4. File size:
    - If file size is in GiB range: Consider the upper end of the baseline interval or beyond (64-256 ranks), especially if N_{cnodes} and $N_{locations}$ are also high.
    - If file size is in MiB range: Stick to the baseline interval unless N_{cnodes} or $N_{locations}$ suggest otherwise.
5. Average number of children per *cnode*:
    - If average children per *cnode* is ≤ 3: Stick to the lower end of the baseline interval due to likely shallower tree depth (16-64).
    - If average children per *cnode* is > 3: Consider the upper end of the baseline interval to account for potentially deeper tree structures that could benefit from more ranks.

An example of how to apply these rules would be the following. Let's take a cube file with 200 *cnodes* and 30,000 locations with average children per *cnode* to be of 2. We start with a baseline interval which is 32-200 ranks, then since $N_{cnodes} \leq 1000$ and $10000 < N_{locations} \leq 50000$, we switch first to a bit lower interval of 16-64 ranks and then based on the overhead, we do mid-range interval of it and get 16-96, as we have less *cnodes*. Since the cube file size will be obviously in GiB and average number of children per *cnode* is 2, we decided to reduce the upper limit as we have less *cnodes* and we stick to the lower end of the interval, which is 16-64. Such trees and predicted intervals can be seen in the next section.

6.3.3 Real life cubes

A clear improvement is visible on benchmark cubes, in which they had a structured order of nodes in the call tree, and now with more insights into the algorithms we are ready to test the prototype's performance on real-life cube files. We split the cube files into groups of two: large-file size is in GiB, small-file size is in MiB. It's important to note that not all of the data was used in the calculation, but rather only a part of it, as calculation includes only one metric. All the cube files are gathered throughout various performance measurements, and the details of them can be found in Table 4 and Table 5. In the tables is also a column with the predicted interval of MPI ranks for which we believe it is going to be the ideal measurement setup, as one example shown in chapter 6.3.2.

Cube_Files	N_m	N_Cn	N_loc	File Size (GiB)	Av_Children_Cn	Pr_Interval
nekbone/1m	14	221	1048576	10.65	2(2.4555)	32 - 128
nekbone/262k	14	193	262144	2.36	2	16 - 64
LargeCubes/64k	187	222	65536	9.17	2(2.4667)	32 - 128
LargeCubes/256k	188	38	262144	6.19	3(2.6428)	4 - 32
LargeCubes/32k	187	222	32768	4.58	2(2.4667)	16 - 64
LargeCubes/16k	190	222	16384	3.29	2(2.4667)	16 - 64

Table 4: Large cube file information

N_m: Total number of metrics, N_Cn: Total number of *cnodes*, N_loc: Total number of locations, Av_ Children_Cn: Average number of children per *Cnode*, Pr_Interval: Predicted interval

In Figure 35, we measure the timings of cube's task execution using the Random Shuffle method, while repeating the runs for five times. *Random shuffle* is chosen, as it doesn't depend on the tree structure as well the scaling of the method is somewhat similar when compared to *Deepest chain*.

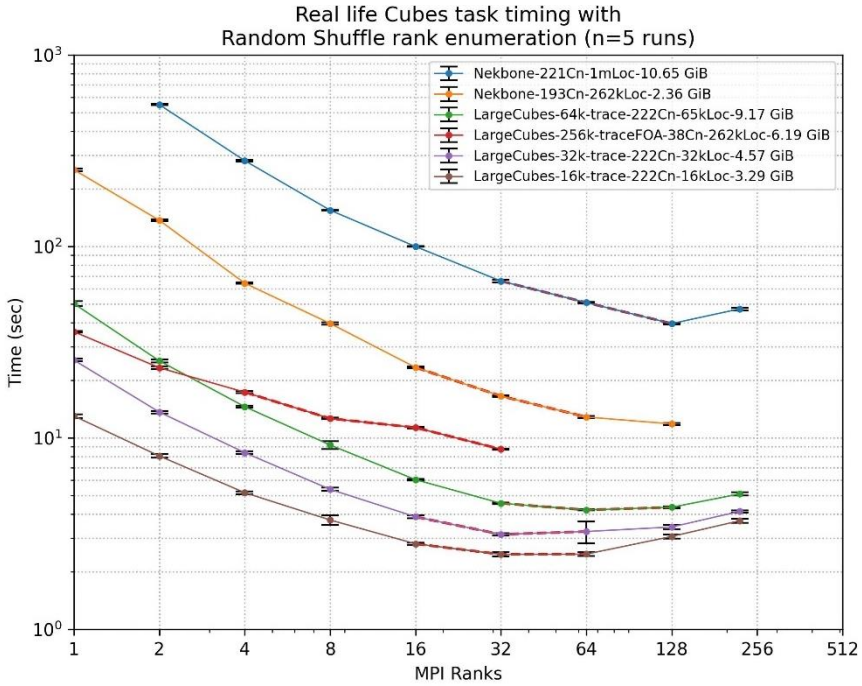

Figure 35: Real life big cubes

The red colored dotted intervals in the line charts represent our expectations of the minimum time we aim to reach, based on the rules we set in the previous section. Predicted interval aligns with the results, and using parallelization on the real-life large cubes improves the time execution of the task, when compared to the sequential execution.

In all cases, the more MPI ranks used, the better the performance we see of the new library, especially for the largest cube. Call tree sizes are not large, as it only goes up to 222, but each *cnode* carries a heavy computational load as we have large system trees. Especially for the red line, it is extreme case with large system tree, and small call tree, hence it is best to assign $N_{ranks} = N_{cnodes}$. In the graph, the curve's minimum wasn't reached, and it has a slightly higher number of children per *cnode*, therefore the reason why it is slightly flatter, similar to the result 2And the more MPI ranks are used, which means assigning fewer nodes per rank, leads to a speedup in performance. This is because each rank can focus on processing its smaller, more manageable set of *cnodes* more efficiently, resulting in improved overall performance. Every rank takes enough time for computation of *cnode* value, which implies that the communication pattern between ranks doesn't play a role.

We test the prototype on smaller real-life cubes with the same enumeration method *Random shuffle,* as it has proved to be stable from the previous measurement.

Cube_Files	N_m	N_Cn	N_loc	File Size (MiB)	Av_Children_Cn	Pr_Interval
LargeCubes/scout	65	221	1048576	894.2	2(2.4555)	16 - 64
Hemelb	15	41	239615	812.24	6(5.66667)	2 - 16
Rohed	6	33	524288	656.38	5(4.57143)	2 - 8
sum_papi	17	832	6144	296.48	2(2.3994)	4 - 16
bnbpbn	12	4126	1280	182.98	6)	32 - 128
ICON-OMP+HWC	15	5621	9	6	3(2.5111)	32 - 128
mini_lpt960	15	68	960	4.1	24	2-16

Table 5: Smaller cube file information

From Figure 36, the predicted interval is not precise as in the previous measurement. This can be due to too many non-balanced call tree structures. But even in mid-range size cubes, parallelization leads to speedup of task timing, except for micro cubes. This is expected as we showed for similar micro cube files that the communication patterns don't play a role until a certain workload is reached per rank.

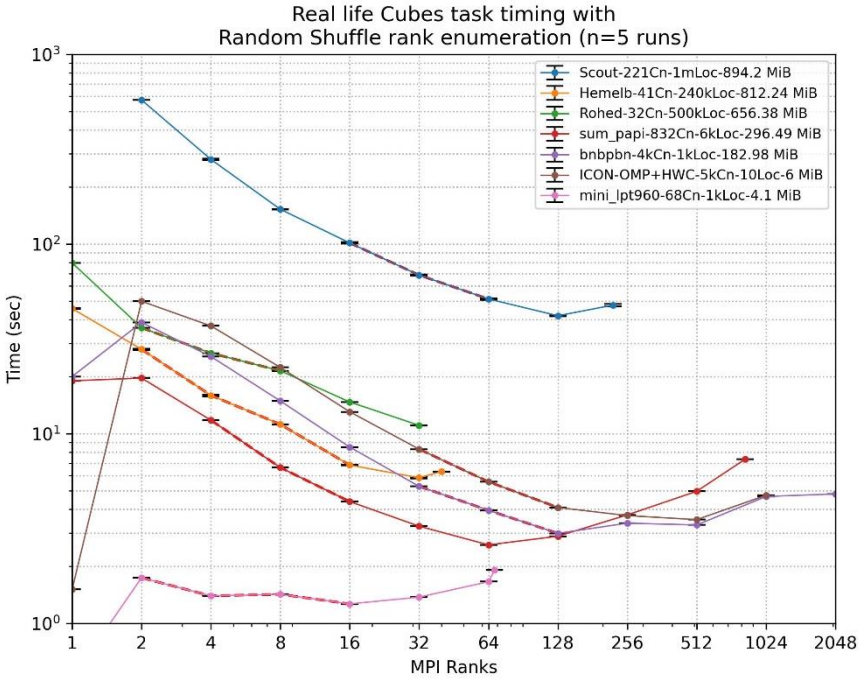

Figure 36: Real life small cubes

6.3.4 Memory recordings

With the growing size of the cube files, which one wants to open and analyze, memory might be a limiting factor.

We want to study first the impact of different cube files, where we vary the number of nodes in call and system trees, on memory footprint when information about the cube file is being loaded or read. A general formula for calculating the memory footprint can be given as:

$$M_{footprint} = \underbrace{O\left(M_{cube\_description\_data}\right)}_{A} + \underbrace{O\left(M_{metrics\_data}\right)}_{B} \tag{6.1}$$

In the current approach we cannot change the first term in formula, hence with a large cube one can hit the upper limit. However, we can check how much memory it takes when we load the cube file.

Obtaining this measurement is done again via Score-P profiling. We set the variable `SCOERP_MEMORY_RECORDING` to true and look for maximum memory heap allocation of `openCubeReport(…)` in profile obtained from the performance run. Such measurement can be seen in Figure 37.

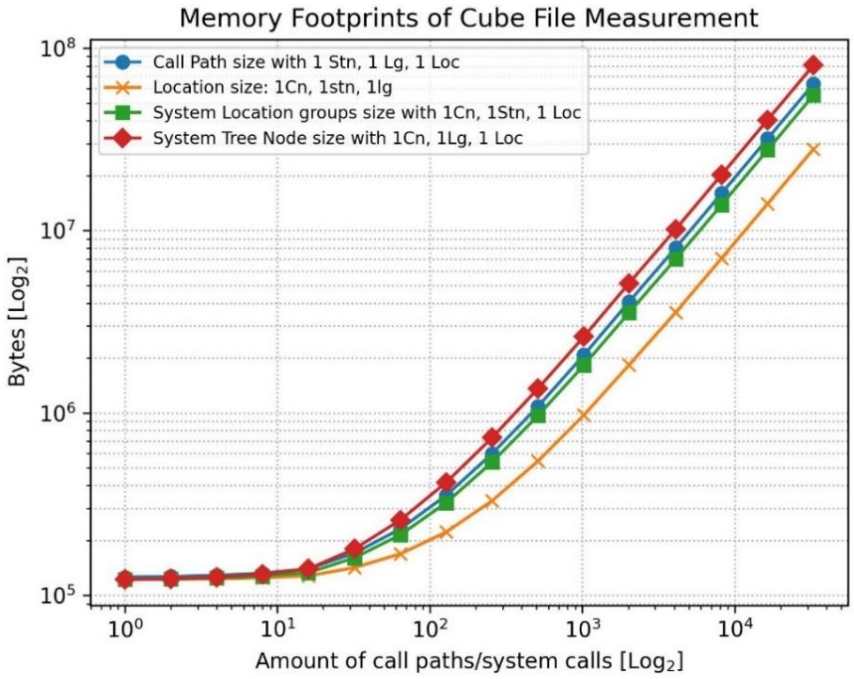

Figure 37: Memory footprint

Figure shows a memory measurement of the cube file. Configuration names: Cn - Call path Nodes, Stn - System Tree Nodes, Lg - Location groups, Loc - Locations

A linear growth starts appearing from 100 call or system nodes, and on average 2 kilobytes per *cnode*/system tree node is the memory watermark. Therefore for example, if one would have limited memory of 512 MiB of RAM on their computer setup, it can not even open a big cube file that has more tehn 100000 *cnodes* and 100000 locations. Nevertheless, even if one manages to open, exploring such a cube file can result in a bad user experience and crashes of software due to memory limitations. With the new MPI library we can also measure the amount of memory per process when handling such requests coming from the user while exploring the file.

If we extend the formula (6.1) for both terms, we get

$$M_{footprint} = \underbrace{\mathcal{O}(N_{metrics}) + \mathcal{O}(N_{cnodes}) + \mathcal{O}(N_{stn})}_{A} + \underbrace{\mathcal{O}(N_{metrics} \cdot N_{cnodes} \cdot N_{stn})}_{B} \tag{6.2}$$

It can be seen that the largest contributor in this equation comes from the last term. If we distribute the compute nodes to different ranks, it will result in a lower memory footprint per process.

In order to prove that data distribution if properly done, we see in Figure 38, a maximum memory allocation footprint of rank 0, when executing the mock task for binary call tree using *Deepest chain* and *Random shuffle* enumeration methods. Cube file with the total memory size of 13 GiB, has 8192 cndoes and 100000 number of locations in the system tree, was used for this test.

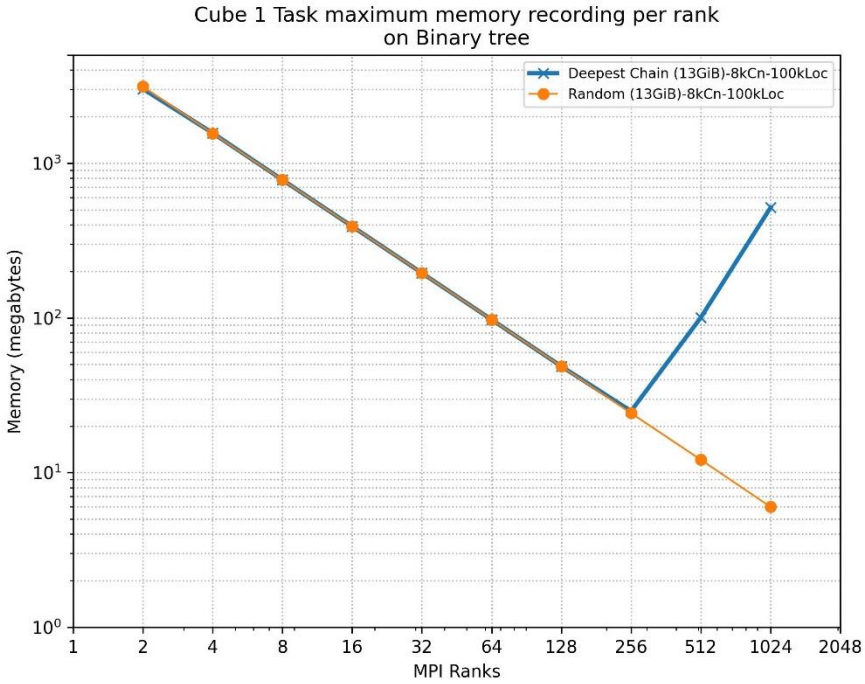

Figure 38: Memory footprint of task execution from one process

The result shows a linear behavior of reducing the memory footprint on rank 0, while increasing the number of processes with *Random shuffle* enumeration. This is expected, as we distributed the workload uniformly to all ranks, where every rank computes its *own* data and only sends one result per task to its parent rank.

A similar behavior can be observed for *Deepest chain* enumeration, but starting from 256 ranks, the memory starts to increase. This can be due to clustering of *cnodes* in the tree, resulting in more memory resources to be used as local contributions are higher, as the method is designed to preserve parent-child relations as much as possible. Therefore, as for overall measurements *Random shuffle* showed to be a solid and efficient method when compared to other enumeration methods of MPI ranks.

In conclusion we can say that as long as we can load the metadata into the RAM, we can give so many MPI ranks to the `cube_server`, that individual memory footprint drops under the limit of available memory, and one would be able to process such large cubes.

7 Conclusion and future work

7.1 Conclusion

During the performance analysis, the generation and reading of the desired cube files occur in three steps: First, the initial measurement; second, refining the cube files based on previous or inadequate files; and third, iterating the first two steps until achieving satisfactory measurements. Initial measurement is usually the least useful and largest one, but perhaps most important one as one uses it to refine the filtering, selective recording in order to minimize impact of the measurement overhead on the quality of the measurement. This thesis aims to improve this step. The objective was to rewrite the *CubeLib* library, with the future potential of its Cube server tool, which is used to produce and read data models. We improved the current calculation methods and task distribution, enabling parallel data computation by different processes.

The findings of this study demonstrate that parallelization is beneficial for processing cube files of any size, as it consistently achieves optimal timings for a given number of processes. Compared to traditional sequential processing methods, parallel approaches were found to be more efficient and quicker, a conclusion supported by tests conducted on various datasets. This increased efficiency is due to the equal distribution of data across processes and its asynchronous management, which is largely a result of the overlap between computation and communication processes. Notably, the library, when parallelized, is enough to be effective even in single-process configurations, thus serving different user requirements.

In the context of MPI rank enumeration as we have seen in 5.3.2, *Random shuffle* enumeration exhibits a surprisingly solid performance under the condition of a truly uniform distribution and *Deepest chain* enumeration can be a second choice. However, the primary advantage of random enumeration is that it doesn't depend on the structure of the call tree, hence it is easier to implement. Downsize is that in narrow deep call trees, like in linear trees, it performed weaker compared to the Deepest chain as it can be seen in Figure 20.

When analyzing real-world Cube4 files, it is evident that even small to moderately sized cubes benefit from the utilization of MPI-parallelized library. Only micro-cubes do not derive significant advantages from this parallelization as seen in figures Figure 32 and Figure 33 (see chapter 6.3.1. results from 4.1-4.3). However, in the case of "deep narrow" cubes, which are typical for initial measurement as one performs on a small scale and without any filtering or selective recording, and have larger call tree, indicate noticeable improvements in the computation time.

Heuristic rules are formulated which allow users to choose parallelization parameters, such as number of MPI ranks, by examining the cube without loading the data. Hence one can develop a small examination tool, which gives advice or even generates Slurm script for `cube_server`.

Individual memory footprint drops linearly as expected, hence, if the description of dimensions "fits" into the memory, one can always utilize enough MPI ranks to the server, so that data also fits in the memory. This enables processing of enormous cube files, as in call tree size no filtering is needed and as in location size can be used from exascale HPC systems. For such large cube files manual exploration is possible, as maximal speedup from 10 to 25 times has been observed throughout the results.

7.2 Future work

Looking ahead, future efforts should aim at using OpenMP or tasks parallelization strategy to compute individual load (value of *own cnodes*), where one can press down timings further and enable even further comfortable manual performance analysis. Using additional resources on individual MPI processes, such as GPU cards, will allow from one side 1-) more data load, which means larger individual load, larger number of locations, 2-) speedup of the calculation value of own cnode. This opens even more the possibility to process larger cubes, coming perhaps even from the exascale HPC system.

So far, we have studied only "data" metrics, but cube also supports derived metrics. One needs to study how compatible this parallelization pattern is with the derived metrics engine. First tests showed that if the CubePL expression is local (access to the same cnode as the request parameters) - then it works without further modification. However, the CubePL expression might be "non-local" and this situation requires further deeper study.

In this thesis we have been studying only `cube_server`, which only delivers data to the client. Hence `get_sev(…)` has been modified. It is expected that any tool, which only reads cube files will benefit from this parallelization, however, this needs to be studied, if `cube_dump`, `cube_calltree`, and other tools actually benefit.

Parallelization is only applied on the call tree and it showed promising results. However, it is still possible to make parallelization along the system tree. It is expected that `cube_remap2` tool will profit from the configuration, where the number of MPI ranks is the same as it was during the measurement. Meaning that, one should include remapping steps into the measurement script as a post processing step.

Many tools do produce another cube as an output. This requires study, how `set_sev(…)` needs to work in an MPI environment and if it is possible to profit from it. Keyword should be on the “locality” of the data. For reading it has to be "close" to each other, and for writing as separate as possible.

To get to the production stage one needs to remove "mocking" and replace it with the actual connection to the client. As every HPC system has individual safety protocols, this has to be worked out. The current implementation of *CubeLib* has few build dependencies and therefore can be built on nearly every platform. With MPI parallelization it gets one. Therefore, it is a separate task to implement MPI communication in such a way that one still can build a *CubeLib* library in a non-MPI environment, using preprocessor macros, or wrappers of any other techniques.

8 References

[1] Bell, G., Cady, R., McFarland, H., Delagi, B., O'Laughlin, J., "A new architecture for minicomputers-," *AFIPS Press*, vol. 36, pp. 657–675.

[2] Richard M. Russell, "The CRAY- 1 Computer System," *Computer Systems*, vol. 21, 1978.

[3] William Daniel Hillis, "The Connection Machine," Massachusetts Institute of Technology, Cambridge/USA, June/1985.

[4] Erol Gelenbe, "Performance analysis of the connection machine," *ACM*, 1990.

[5] D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawak, and C. V. Packer, Eds., *Beowulf: A Parallel Workstation For Scientific Computation*, 1995.

[6] Jarett Cohen, "Computing with Beowulf," *R and D Magazine*, 1999.

[7] Wikipedia, *Beowulf cluster.* [Online]. Available: https://en.wikipedia.org/w/index.php?title=Beowulf_cluster&oldid=1189158357 (accessed: 13-Feb-24).

[8] Michael W. Berry,Charles Grassl,Vijay K. Krishna, "Blocked Data Distribution for the Conjugate Gradient Algorithm on the CRAY T3D," University of Tennessee, 01-August-1994.

[9] *Cray Research Inc | Encyclopedia.com.* [Online]. Available: https://www.encyclopedia.com/social-sciences-and-law/economics-business-and-labor/businesses-and-occupations/cray-research-inc (accessed: 13-Feb-24).

[10] W. Daniel Hillis, Lewis W. Tucker, "The CM-5 Connection Machine," *Communication of the ACM*, vol. 36, 1993.

[11] I. Buck, "GPU Computing: Programming a Massively Parallel Processor," in *GPU Computing: Programming a Massively Parallel Processor*, San Jose, CA, 2007, p. 17.

[12] NVIDIA Developer, *CUDA Zone - Library of Resources.* [Online]. Available: https://developer.nvidia.com/cuda-zone (accessed: 13-Feb-24).

[13] M. Ginsberg, Ed., *Supercomputers: Challenges to designers and users*, 1982.

[14] B. Wilkinson and C. M. Allen, *Parallel programming: Techniques and applications using networked workstations and parallel computers,* 2nd ed. Upper Saddle River NJ: Pearson/Prentice Hall, 2005.

[15] P. S. Pacheco, *An introduction to parallel programming*. Amsterdam, Boston: Morgan Kaufmann, 2011.

[16] Sarita V. Adve et. al, *Parallel Computing Research at Illinois: The UPCRC Agenda*, 2008.

[17] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts*. Hoboken NJ: Wiley, 2013.

[18] Wikipedia, *OpenMP.* [Online]. Available: https://en.wikipedia.org/w/index.php?title=OpenMP&oldid=1199889263 (accessed: 01-Mar-24).

[19] F. Nielsen, *Introduction to HPC with MPI for Data Science*. Cham: Springer International Publishing, 2016.

[20] *HPCToolkit Home.* [Online]. Available: http://hpctoolkit.org/ (accessed: 14-Feb-24).

[21] *Paraver: a flexible performance analysis tool | BSC-Tools.* [Online]. Available: https://tools.bsc.es/paraver (accessed: 14-Feb-24).

[22] C. Roessel, *VI-HPS :: Projects :: Score-P.* [Online]. Available: https://www.vi-hps.org/projects/score-p/ (accessed: 14-Feb-24).

[23] D. Becker, *Scalasca.* [Online]. Available: https://www.scalasca.org/ (accessed: 14-Feb-24).

[24] GWT GmbH, *Vampir 10.4.* [Online]. Available: https://vampir.eu/ (accessed: 14-Feb-24).

[25] H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, *Tools for High Performance Computing 2011*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[26] P. Saviankou, M. Knobloch, A. Visser, and B. Mohr, "Cube v4: From Performance Report Explorer to Performance Analysis Tool," *Procedia Computer Science*, vol. 51, pp. 1343–1352, 2015, doi: 10.1016/j.procs.2015.05.320.
[27] The Scalasca Development Team, "Scalasca 2.3 - User Guide: Scalable Automatic Performance Analysis," Jülich Forschungszentrum, 2016.
[28] Forschungszentrum Jülich, *Performance properties: Scalasca Patterns.* [Online]. Available: https://apps.fz-juelich.de/scalasca/releases/scalasca/2.5/help/scalasca_patterns.html (accessed: 14-Feb-24).
[29] The Scalasca Development Team, "CUBE 4.3.4 – User Guide: Generic Display for Application Performance Data," Forschungszentrum Jülich, 2016.
[30] Bine Brank, "Parallel MPI IO in Cube: Design & Implementation," Bergische Universität Wuppertal, Forschungszentrum Jülich, Germany, 2018.
[31] F. W. Fengguang Song, "CUBE - User Manual: Generic Display for Application Performance Data," University of Tennessee, 2005.
[32] Brian Wylie, "Analysis report examination with Cube," Jülich Supercomputing Centre, Tennessee, USA, Apr. 2019. Accessed: 01-March-2024. [Online]. Available: https://www.vi-hps.org/cms/upload/material/tw31/Cube.pdf
[33] University of South Florida, *Point-to-Point Communication.* [Online]. Available: http://www.rc.usf.edu/tutorials/classes/tutorial/mpi/chapter4.html (accessed: 15-Feb-24).
[34] O. Tatebe, Y. Kodama, S. Sekiguchi, and Y. Yamaguchi, "Highly efficient implementation of MPI point-to-point communication using remote memory operations," in *Proceedings of the 12th international conference on Supercomputing*, Melbourne Australia, 1998, pp. 267–273.
[35] R. Farber, *CUDA application design and development*. Waltham MA: Morgan Kaufmann, 2011.
[36] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard: Version 3.0," 2012. [Online]. Available: https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf
[37] E. Castillo *et al.,* "Optimizing computation-communication overlap in asynchronous task-based programs," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, Washington District of Columbia, 2019, pp. 415–416.
[38] M. Sergent, M. Dagrada, P. Carribault, J. Jaeger, M. Pérache, and G. Papauré, "Efficient Communication/Computation Overlap with MPI+OpenMP Runtimes Collaboration," in *Lecture Notes in Computer Science, Euro-Par 2018: Parallel Processing*, M. Aldinucci, L. Padovani, and M. Torquati, Eds., Cham: Springer International Publishing, 2018, pp. 560–572.
[39] The Scalasca Development Team, "CubeGUI User 4.8 - User Guide: Introduction in Cube GUI and its usage," Forschungszentrum Jülich, 2023.
[40] The Scalasca Development Team, "CubeLib4.5 - Derived Metrics: Intoduction in CubePL and Cube's derived metrics," Jülich Forschungszentrum, 2020.
[41] Geimer, M., Saviankou, P., Strube, A., Szebenyi, Z., Wolf F., J. N. Wylie, B., Ed., *Further improving the scalability of the scalasca toolset*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.
[42] Wikipedia, *Breadth-first search.* [Online]. Available: https://en.wikipedia.org/w/index.php?title=Breadth-first_search&oldid=1197517460 (accessed: 01-Mar-24).
[43] Wikipedia, *Depth-first search.* [Online]. Available: https://en.wikipedia.org/w/index.php?title=Depth-first_search&oldid=1199710840 (accessed: 01-Mar-24).
[44] *std::random_shuffle, std::shuffle - cppreference.com.* [Online]. Available: https://en.cppreference.com/w/cpp/algorithm/random_shuffle (accessed: 01-Mar-24).

[45] P. Thörnig, "JURECA: Data Centric and Booster Modules implementing the Modular Supercomputing Architecture at Jülich Supercomputing Centre," *JLSRF*, vol. 7, A182, 2021, doi: 10.17815/jlsrf-7-182.
[46] *Slurm Workload Manager - Documentation.* [Online]. Available: https://slurm.schedmd.com/ (accessed: 01-Mar-24).
[47] *ParTec – Modular Supercomputing.* [Online]. Available: https://par-tec.com/ (accessed: 01-Mar-24).
[48] *JUST.* [Online]. Available: https://www.fz-juelich.de/en/ias/jsc/systems/storage-systems/just (accessed: 01-Mar-24).
[49] *JUWELS.* [Online]. Available: https://www.fz-juelich.de/en/ias/jsc/systems/supercomputers/juwels (accessed: 01-Mar-24).
[50] *JUSUF.* [Online]. Available: https://www.fz-juelich.de/en/ias/jsc/systems/supercomputers/jusuf (accessed: 01-Mar-24).
[51] "Score-P User Manuel: Scalable performance measurement infrastructure for parallel codes," 2019. Accessed: 01-March-2024. [Online]. Available: https://scorepci.pages.jsc.fz-juelich.de/score-p-pipelines/docs/scorep-6.0/pdf/scorep.pdf

A Appendix - source code

In this section of the thesis, a rewritten source code can be found. Only the modified files including the .cpp and header .h files, as well as files that need to be added such as benchmark cube script and enumeration methods, are shown. For easier navigation through and methods shown in chapter 5, all related codes are presented here. The *CubeLib* library is an open source library, therefore it can be downloaded for free (see [23]).

A.1 Cube.cpp

```
/****************************************************************************
**  CUBE        http://www.scalasca.org/                                    **
*****************************************************************************
**  Copyright (c) 1998-2023                                                 **
**  Forschungszentrum Juelich GmbH, Juelich Supercomputing Centre           **
**                                                                          **
**  Copyright (c) 2009-2015                                                 **
**  German Research School for Simulation Sciences GmbH,                    **
**  Laboratory for Parallel Programming                                     **
**                                                                          **
**  This software may be modified and distributed under the terms of        **
**  a BSD-style license.  See the COPYING file in the package base          **
**  directory for details.                                                  **
****************************************************************************/

namespace cube
{
    MPI_Datatype MPI_TASK;

    void cnode_plain_distribution(int p, Cube& cube)
    {
        const std::vector<Cnode *> cnodes = cube.get_cnodev();
        int Cnodes_total = cnodes.size();
        int cnodesPerProcess = Cnodes_total / p;
        int startIdx;
        int endIdx;
        int i;
        for (int my_rnk = 0; my_rnk < p; my_rnk++)
        {
            startIdx = my_rnk * cnodesPerProcess;
            endIdx = (my_rnk + 1) * cnodesPerProcess;
            if (my_rnk == (p - 1))
            {
                endIdx = Cnodes_total;
            }
            for (i = startIdx; i < endIdx; i++)
            {
                cnodes[i]->set_calculating_rank(my_rnk);
            }
        }
    }

    double metric_agg_exclusive_mpi_task(const Cnode* tmp_c, const CalculationFlavour cnf,
uint32_t tmp_id)
```

```
        {
            int msg_receive_tag=10003+tmp_c->get_id();
            double res;

            Task* task=new Task();
            task->taskId = tmp_c->get_id();
            task->metric_id = tmp_id;
            task->metric_calc = CUBE_METRIC_EXCLUSIVE;
            task->cnode_id = tmp_c->get_id();
            task->cnode_calc =cnf;
            task->sys_id = -1;
            task->sys_cal= -1;

            MPI_Send(task,  1,  MPI_TASK,  tmp_c->get_calculating_rank(),  tmp_c->get_id(),
MPI_COMM_WORLD);
            MPI_Recv(&res,  1,  MPI_DOUBLE,  tmp_c->get_calculating_rank(),  msg_receive_tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            delete task;
            return static_cast<double>(res);
        }

        double metric_agg_inclusive_mpi_task(const Cnode* tmp_c, const CalculationFlavour cnf,
uint32_t tmp_id)
        {
            int msg_receive_tag=10003+tmp_c->get_id();
            double res;

            Task* task=new Task();
            task->taskId = tmp_c->get_id();
            task->metric_id = tmp_id;
            task->metric_calc = CUBE_METRIC_INCLUSIVE;
            task->cnode_id = tmp_c->get_id();
            task->cnode_calc =cnf;
            task->sys_id = -1;
            task->sys_cal= -1;

            MPI_Send(task,  1,  MPI_TASK,  tmp_c->get_calculating_rank(),  tmp_c->get_id(),
MPI_COMM_WORLD);
            MPI_Recv(&res,  1,  MPI_DOUBLE,  tmp_c->get_calculating_rank(),  msg_receive_tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            delete task;
            return static_cast<double>(res);
        }

        void process_task(Task* task, int sender_rank, int tag, Cube &cube)
        {
            MPI_Request request=MPI_REQUEST_NULL;
            int message_tag;
            if (tag==10001)
            {
                message_tag=tag*3;
            }
            else
            {
                message_tag = 10003+tag;
            }
            double tmp_val;
```

```
            const std::vector<Cnode *> &cnodes = cube.get_cnodev();
            const std::vector<Metric *> &metrics = cube.get_metv();
            const std::vector<Sysres *> &syss = cube.get_sysv();

            Metric *metric = metrics[task->metric_id];
            CalculationFlavour    metric_calc    =    static_cast<CalculationFlavour>(task-
>metric_calc);
            Cnode *cnode = cnodes[task->cnode_id];
            CalculationFlavour    cnode_calc    =    static_cast<CalculationFlavour>(task-
>cnode_calc);

            if (task->sys_id == -1)
            {
                tmp_val = cube.get_sev(metric, metric_calc, cnode, cnode_calc);
            }
            else
            {
                Sysres *sys = syss[task->sys_id];
                CalculationFlavour sys_cal = static_cast<CalculationFlavour>(task->sys_cal);
                tmp_val = cube.get_sev(metric, metric_calc, cnode, cnode_calc, sys, sys_cal);
            }
            MPI_Isend(&tmp_val,  1,  MPI_DOUBLE,  sender_rank,  message_tag,  MPI_COMM_WORLD,
&request);
            int send_complete = 0;
            while (!send_complete)
            {
                MPI_Test(&request, &send_complete, MPI_STATUS_IGNORE);
                std::this_thread::yield();
            }
            delete task;
        }

        void task_manager(Cube &cube)
        {
            // SCOREP_USER_REGION_DEFINE( my_region_handle )
            // SCOREP_USER_REGION_DEFINE( my_region_handle2 )
            // // SCOREP_USER_REGION_DEFINE( my_region_handle3 )
            // SCOREP_USER_REGION_DEFINE( my_region_handle4 )

            int message_tag;
            int sender_rank;
            int kill_signal;
            std::vector<std::future<void>> futures;
            // std::vector<MPI_Request> recv_requests;
            const std::vector<Cnode *> &cnodes = cube.get_cnodev();
            int blocklen[7] = {1, 1, 1, 1, 1, 1, 1};
            // array of displacements
            MPI_Aint displacements[7] = {
                offsetof(Task, taskId),
                offsetof(Task, metric_id),
                offsetof(Task, metric_calc),
                offsetof(Task, cnode_id),
                offsetof(Task, cnode_calc),
                offsetof(Task, sys_id),
                offsetof(Task, sys_cal)};
            // array of types
```

```
            MPI_Datatype types[7] = {MPI_UINT64_T, MPI_UINT64_T, MPI_INT32_T, MPI_UINT64_T,
MPI_INT32_T, MPI_INT64_T, MPI_INT32_T};
            MPI_Type_create_struct(7, blocklen, displacements, types, &MPI_TASK);
            MPI_Type_commit(&MPI_TASK);

            std::this_thread::sleep_for(std::chrono::seconds(1));

            while (true)
            {
                MPI_Status status;
                // Check if there is a message
                int flag = 0;
                MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &flag, &status);
                // MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
                if (flag)
                {
                    // Receive the message
                    sender_rank = status.MPI_SOURCE;
                    message_tag = status.MPI_TAG;

                    if (message_tag <= 10001 && message_tag>=0)
                    {
                        //            SCOREP_USER_REGION_BEGIN(            my_region_handle2,
"Internal_agg_metric_calculation",SCOREP_USER_REGION_TYPE_COMMON )
                        Task* task=new Task();
                        MPI_Recv(task, 1, MPI_TASK, sender_rank, message_tag, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                        futures.push_back(std::async(std::launch::async, process_task, task,
sender_rank, message_tag, std::ref(cube)));
                        // SCOREP_USER_REGION_END( my_region_handle2 )
                    }
                    else if (message_tag==10002)
                    {
                        //            SCOREP_USER_REGION_BEGIN(            my_region_handle4,
"Kill_Signal",SCOREP_USER_REGION_TYPE_COMMON )
                        for (std::future<void> &future : futures)
                        {
                            future.wait(); // Wait for all async tasks to complete
                        }
                        MPI_Recv(&kill_signal,   1,   MPI_INT,   sender_rank,   message_tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        // SCOREP_USER_REGION_END( my_region_handle4 )
                        break;
                    }
                }
            }
        }
    }

    int& Cube::getCubeIdCounter() {
        static int cubeIdCounter = 0;
        return cubeIdCounter;
    }
    std::unordered_map<Cube*, int>& Cube::getCubeIds() {
        static std::unordered_map<Cube*, int> cubeIds;
        return cubeIds;
    }
```

```
Cube::Cube(CubeEnforceSaving _enforce_saving,
           int _cube_mpi_rank,
           int _cube_mpi_processes)
    : cur_cnode_id(0),
      cur_metric_id(0),
      cur_region_id(0),
      cur_stn_id(0),
      cur_location_group_id(0),
      cur_location_id(0)
{
    cubepl_memory_manager = new CubePL2MemoryManager();
    //…
    initialized = false;
    set_post_initialization(true);

    enforce_saving = _enforce_saving;
    cube_mpi_rank = _cube_mpi_rank;
    cube_mpi_processes = _cube_mpi_processes;
    //…
    cubeId = getCubeIdCounter();
    getCubeIds()[this] = cubeId;
    getCubeIdCounter()++;
}

int Cube::getCubeId() const {
    return cubeId;
}

void Cube::openCubeReport(std::string _cubename,
                          bool disable_tasks_tree,
                          bool _disable_clustering)
{
    closeCubeReport();
    //…
    initialize();
    if (!disable_tasks_tree)
    {
        process_tasks();
    }
    // DISTRIBUTION of cnodes
    cnode_plain_distribution(cube_mpi_processes, *this);
}

void Cube::initialize()
{
    //…
    // STARTING TASK MANAGER
    a2 = std::async(std::launch::async, task_manager, std::ref(*this));
}
```

A.2 CubeMetricBuildInType.h

```
template <class T>
class BuildInTypeMetric : public Metric
{
protected:
    SimpleCache<T>* t_cache;

public:
    int metric_mpi_rank;
        BuildInTypeMetric( const std::string& disp_name,
                       const std::string& uniq_name,
                       const std::string& dtype,
                       const std::string& uom,
                       const std::string& val,
                       const std::string& url,
                       const std::string& descr,
                       FileFinder*        ffinder,
                       Metric*            parent,
                       uint32_t           id                              = 0,
                       const std::string& cubepl_expression               = "",
                       const std::string& cubepl_init_expression          = "",
                       const std::string& cubepl_aggr_plus_expression     = "",
                       const std::string& cubepl_aggr_minus_expression    = "",
                       const std::string& cubepl_aggr_aggr_expression     = "",
                       bool               row_wise                        = true,
                       VizTypeOfMetric    is_ghost                     = CUBE_METRIC_NORMAL
                       )
        :
        Metric( disp_name,
                uniq_name,
                dtype,
                uom,
                val,
                url,
                descr,
                ffinder,
                parent,
                id,
                cubepl_expression,
                cubepl_init_expression,
                cubepl_aggr_plus_expression,
                cubepl_aggr_minus_expression,
                cubepl_aggr_aggr_expression,
                row_wise,
                is_ghost
                )
    {
        t_cache = NULL;
        MPI_Comm_rank(MPI_COMM_WORLD,&metric_mpi_rank);

    }
// Many more functionialites that are out of scope of this thesis
// …
};
```

A.3 CubeInclusiveMetricBuildInType.h

```
    template <class T>
    T
    InclusiveBuildInTypeMetric<T>::get_sev_aggregated(    const    Cnode*    cnode,    const
CalculationFlavour cnf )
    {
        // SCOREP_USER_REGION_DEFINE( my_region_handle7 )
        if ( !( this->active ) )
        {
            return 0.;
        }

        if ( this->adv_sev_mat == NULL && get_type_of_metric() == CUBE_METRIC_INCLUSIVE )
        {
            return 0.;
        }
        T v = static_cast<T>( 0 );
        if ( this->isCacheable() && ( ( this->t_cache )->testAndGetTCachedValue( v, cnode, cnf
)  ) )
        {
            return v;
        }

        size_t sysv_size = this->sysv.size();
        for ( size_t i = 0; i < sysv_size; i++ )
        {
            const Location* _loc = this->sysv[ i ];
            T               tmp  = this->get_sev_elementary( cnode, _loc );
            ( v ) = this->aggr_operator( v,  tmp );
        }

        //                    SCOREP_USER_REGION_BEGIN(                    my_region_handle7,
"Inclusive_sev_aggregated",SCOREP_USER_REGION_TYPE_COMMON )
        if ( cnf == CUBE_CALCULATE_EXCLUSIVE &&  ( cnode->num_children() > 0 ) )
        {
            T _cv = static_cast<T>(0);
            // Split the the children vector of given cnode depending on the rank distribution
            std::vector<cube::Cnode*> local_cnodes;
            std::vector<cube::Cnode*> remote_cnodes;
            for (cnode_id_t cid = 0; cid < cnode->num_children(); cid++)
            {
                cube::Cnode* tmp_c = cnode->get_child(cid);
                if (!tmp_c->isHidden())
                {
                    if (tmp_c->get_calculating_rank() == this->metric_mpi_rank)
                    {
                        local_cnodes.push_back(tmp_c);
                    }
                    else
                    {
                        remote_cnodes.push_back(tmp_c);
                    }
                }
            }
```

```
            // Calculate remote contributions asynchronously
            std::vector<std::future<double>> remote_results;
            uint32_t tmp_id=this->get_id();
            for (cube::Cnode* tmp_c : remote_cnodes)
            {
                if (!tmp_c->isHidden())
                {
                    std::future<double>       a4       =       std::async(std::launch::async,
metric_agg_inclusive_mpi_task, tmp_c,cnf, tmp_id);
                    remote_results.push_back(std::move(a4));
                }
            }

            // Calculate local contributions
            T _cv_local = static_cast<T>(0);
            for (cube::Cnode* tmp_c : local_cnodes)
            {
                if (!tmp_c->isHidden())
                {
                    T           tmp_t           =           this->get_sev_aggregated(tmp_c,
cube::CUBE_CALCULATE_INCLUSIVE);
                    _cv_local = this->plus_operator(_cv_local, tmp_t);
                }
            }

            // Wait for remote result
            T _cv_remote = static_cast<T>(0);
            for (std::future<double>& a4 : remote_results)
            {
                T remote_result = a4.get();
                _cv_remote = this->plus_operator(_cv_remote, remote_result);
            }

            // v = this->minus_operator(v, this->plus_operator(_cv_remote, _cv_local));
            _cv=this->plus_operator(_cv_remote, _cv_local);
            v = this->minus_operator( v, _cv );
        }
        if ( this->isCacheable() )
        {
            this->t_cache->setTCachedValue( v,  cnode, cnf );
        }
        // SCOREP_USER_REGION_END( my_region_handle7 )
        return static_cast<double>( v );
    }
```

A.4 CubeExclusiveMetricBuildInType.h

```
    template <class T>
    T
    ExclusiveBuildInTypeMetric<T>::get_sev_aggregated(    const    Cnode*    cnode,    const
CalculationFlavour cnf )
    {
        // SCOREP_USER_REGION_DEFINE( my_region_handle5 )
        if ( !( this->active ) )
        {
            return 0.;
        }
        if ( this->adv_sev_mat == NULL && get_type_of_metric() == CUBE_METRIC_EXCLUSIVE )
        {
            return 0.;
        }

        T v = static_cast<T>( 0 );
        if ( this->isCacheable() && ( ( this->t_cache )->testAndGetTCachedValue( v,  cnode,
cnf ) ) )
        {
            return v;
        }

        size_t sysv_size = this->sysv.size();
        for ( size_t i = 0; i < sysv_size; i++ )
        {
            Location* _loc = this->sysv[ i ];
            T         tmp  = this->get_sev_elementary( cnode, _loc );
            v = this->aggr_operator( v,  tmp );
        }
        //                    SCOREP_USER_REGION_BEGIN(                    my_region_handle5,
"Exclusive_sev_aggregated",SCOREP_USER_REGION_TYPE_COMMON )
        // Split the the children vector of given cnode depending on the rank distribution
        T _cv = static_cast<T>(0);
        std::vector<cube::Cnode*> local_cnodes;
        std::vector<cube::Cnode*> remote_cnodes;
        for ( cnode_id_t cid = 0; cid < cnode->num_children() ; cid++  )
        {
            cube::Cnode* tmp_c = cnode->get_child(cid);
            if ( cnf == cube::CUBE_CALCULATE_INCLUSIVE || tmp_c->isHidden() )
            {
                if (tmp_c->get_calculating_rank() == this->metric_mpi_rank)
                {
                    local_cnodes.push_back(tmp_c);
                }
                else
                {
                    remote_cnodes.push_back(tmp_c);
                }
            }
        }

        // Calculate remote contributions asynchronously
        std::vector<std::future<double>> remote_results;
        uint32_t tmp_id=this->get_id();
```

```
        for (cube::Cnode* tmp_c : remote_cnodes)
        {
            std::future<double>          a3          =          std::async(std::launch::async,
metric_agg_exclusive_mpi_task, tmp_c, cnf, tmp_id);
            remote_results.push_back(std::move(a3));
        }

        // Calculate local contributions
        T _cv_local = static_cast<T>(0);
        for (cube::Cnode* tmp_c : local_cnodes)
        {
            T tmp_t = this->get_sev_aggregated( tmp_c, cube::CUBE_CALCULATE_INCLUSIVE);
            _cv_local = this->plus_operator(_cv_local, tmp_t);
            // v = this->plus_operator( v, tmp_t );
        }

        // Wait for remote result
        T _cv_remote = static_cast<T>(0);
        for (std::future<double>& a3 : remote_results)
        {
            T remote_result = a3.get();
            _cv_remote = this->plus_operator(_cv_remote, remote_result);
        }
        _cv = this->plus_operator(_cv_remote, _cv_local);
        v = this->plus_operator(v, _cv);
        // v = this->plus_operator(v, this->plus_operator(_cv_remote, _cv_local));

        if ( this->isCacheable()  )
        {
            ( this->t_cache )->setTCachedValue( v,  cnode, cnf );
        }
        // SCOREP_USER_REGION_END( my_region_handle5 )
        return v;
    }
```

A.5 regioninfo_calls.h

```
/**
 * Accumulate usr, mpi, com and total values in md.
 */
template<class T>
void
acc_with_type( metric_data<T>& md, CRegionInfo& reginfo, CBlacklist* blacklist )
{
    /*
       accumulate usr, mpi, com and total values in md
     */
    typename std::map<Region*, double>::const_iterator it;
    for ( it = md.excl.begin(); it != md.excl.end(); ++it )
    {
        T             nv( ( T )it->second );
        const Region* region( it->first );

        md.total += nv;
        switch ( reginfo[ region->get_id() ] )
        {
            case cube::MPI:
                md.mpi += nv;
                break;

            case COM:
                md.com += nv;
                break;

            case USR:
                md.usr += nv;
                break;
            default: // another regions are not observed at all and will be ignored
                break;
        }

        if ( blacklist != 0 )
        {
            if ( ( *blacklist )( region->get_id() ) )
            {
                md.bl += nv;
            }
            else
            {
                switch ( reginfo[ region->get_id() ] )
                {
                    case cube::MPI:
                        md.mpi_bl += nv;
                        break;

                    case COM:
                        md.com_bl += nv;
                        break;

                    case USR:
                        md.usr_bl += nv;
```

```
                        break;
                    default: // another regions are not observed at all and will be ignored
                        break;
                }
            }
        }
    }


    /**
     * Calculate a cost ( given as a severity value mulpilied wih a factor ) for
       every region for every thread. Calculate a cost for every kind of regions
       (usr, mpi, omp, and so on) over all threads. And get a maximum value at the end.
     */
    //…
                    const unsigned long long newval = static_cast<unsigned long long>( input-
>get_sev( visits, cnode, thread ) );
                    const unsigned long long costs  = newval * d;
                    buffer[ ThreadId ]               += costs;
                    tbcosts.pt_all[ ThreadId ]       += costs;
                    tbcosts.acc_costs_by_region[ i ] += costs;
                    if ( blacklist != 0 )
                    {
                        if ( ( *blacklist )( ( Region* )region ) )
                        {
                            tbcosts.pt_bl[ ThreadId ] += costs;
                        }
                        else
                        {
                            switch ( reginfo[ i ] )
                            {
                                case cube::MPI:
                                    tbcosts.pt_mpi_bl[ ThreadId ] += costs;
                                    break;

                                case USR:
                                    tbcosts.pt_usr_bl[ ThreadId ] += costs;
                                    break;

                                case COM:
                                    tbcosts.pt_com_bl[ ThreadId ] += costs;
                                    break;
                                default: // another regions are not observed at all and will
be ignored
                                    break;
                            }
                            tbcosts.pt_wbl[ ThreadId ] += costs;
                        }
                    }

                    switch ( reginfo[ i ] )
                    {
                        case cube::MPI:
                            tbcosts.pt_mpi[ ThreadId ] += costs;
                            break;
```

```
                    case USR:
                        tbcosts.pt_usr[ ThreadId ] += costs;
                        break;

                    case COM:
                        tbcosts.pt_com[ ThreadId ] += costs;
                        break;
                    default: // another regions are not observed at all and will be ignored
                        break;
                }
            }
        }

        unsigned long long max_costs = find_max( buffer ).first;
        if ( tbcosts.max_costs_by_region[ i ] < max_costs )
        {
            tbcosts.max_costs_by_region[ i ] = max_costs;
        }
    }

    /*
       BEGIN: calculate total costs split by category (mpi, com, usr, blacklist) using
get_met_tree(...)
     */

    for ( size_t regionId = 0; regionId < regions.size(); regionId++ )
    {
        Region*               region = regions[ regionId ];
        const vector<Cnode*>& cnodev( region->get_cnodev() );
        unsigned long long    d = TypeFactor( region->get_name() );
        for ( size_t cnodeId = 0; cnodeId < cnodev.size(); cnodeId++ )
        {
            map<Metric*, double> excl_metrics;
            map<Metric*, double> incl_metrics;
            Cnode*               cnode = cnodev[ cnodeId ];
            input->get_met_tree( excl_metrics, incl_metrics, EXCL, INCL, cnode, 0 );
            Metric* metric = input->get_met( "visits" );

            unsigned long long visits( ( unsigned long long )excl_metrics[ metric ] );
            unsigned long long nc = d * visits;

            tbcosts.acc_all += nc;

            if ( blacklist != 0 )
            {
                if ( ( *blacklist )( ( Region* )region ) )
                {
                    tbcosts.acc_bl += nc;
                }
                else
                {
                    switch ( reginfo[ regionId ] )
                    {
                        case cube::MPI:
                            tbcosts.acc_mpi_bl += nc;
                            break;
```

```
                            case USR:
                                tbcosts.acc_usr_bl += nc;
                                break;
                            case COM:
                                tbcosts.acc_com_bl += nc;
                                break;
                            default: // another regions are not observed at all and will be
ignored
                                break;
                        }
                    }
                }

                switch ( reginfo[ regionId ] )
                {
                    case cube::MPI:
                        tbcosts.acc_mpi += nc;
                        break;
                    case USR:
                        tbcosts.acc_usr += nc;
                        break;
                    case COM:
                        tbcosts.acc_com += nc;
                        break;
                    default: // another regions are not observed at all and will be ignored
                        break;
                }
            }
        }

        /*…
         */

        return tbcosts;
    }
```

A.6 Enumeration_Methods.cpp

```
void
cnode_plain_distribution(int p, Cube& cube)
{
    const std::vector<Cnode *> cnodes = cube.get_cnodev();
    int Cnodes_total = cnodes.size();
    int cnodesPerProcess = Cnodes_total / p;
    int startIdx;
    int endIdx;
    int i;
    for (int my_rnk = 0; my_rnk < p; my_rnk++)
    {
        startIdx = my_rnk * cnodesPerProcess;
        endIdx = (my_rnk + 1) * cnodesPerProcess;
        if (my_rnk == (p - 1))
        {
            endIdx = Cnodes_total;
        }
        for (i = startIdx; i < endIdx; i++)
        {
            cnodes[i]->set_calculating_rank(my_rnk);
        }
    }
}
void
round_robin_distribution(int p, Cube& cube)
{
    const std::vector<Cnode *> cnodes = cube.get_cnodev();
    int my_rnk=0;
    for (int i = 0; i <cnodes.size() ; i++)
    {
        if (my_rnk>=p)
        {
            my_rnk=0;
        }
        cnodes[i]->set_calculating_rank(my_rnk);
        my_rnk++;
    }
}

void cnode_random_distribution(int p, Cube& cube) {
    const std::vector<Cnode *> cnodes = cube.get_cnodev();
    int Cnodes_total = cnodes.size();
    int my_rnk=0;
    std::vector<int> shuffled_ranks(Cnodes_total - 1);
    for (int i = 0; i < Cnodes_total - 1; i++) {
        if (my_rnk>=p)
        {
            my_rnk=0;
        }
        shuffled_ranks[i] = my_rnk;
        my_rnk++;

    }
```

```
        std::shuffle(shuffled_ranks.begin(),                                shuffled_ranks.end(),
std::default_random_engine());

        cnodes[0]->set_calculating_rank(0);

        for (int i = 1; i < Cnodes_total; i++) {
            cnodes[i]->set_calculating_rank(shuffled_ranks[i - 1]);
        }
    }

    void assignValue(int totalNodes, int p, int leverage, int& nodeDivision, std::vector<int>&
childAllocated)
    {
        if ((totalNodes%p)==0){
            nodeDivision = (totalNodes / p);
        }
        else{
            nodeDivision = (totalNodes / p) + 1;
        }

        for (int i = 0; i < p; i++)
        {
            childAllocated[i]=0;
        }

    }

    int caculateTotallength(cube::Cnode *node)
        {
            if (node == nullptr)
            {
                return 0;
            }
            int totalLength = 0;
            for ( cnode_id_t cid = 0; cid < node->num_children(); cid++  )
            {
                Cnode* tmp_c = node->get_child( cid );
                totalLength += caculateTotallength(tmp_c);
            }
            return totalLength + 1;
        }

    void assignSameRank(Cnode *cnode, int rank,std::vector<int>& childAllocated)
    {
        if (cnode == nullptr)
        {
            return;
        }
        cnode->set_calculating_rank(rank);
        childAllocated[rank] += 1;
        for ( cnode_id_t cid = 0; cid < cnode->num_children(); cid++  )
        {
            Cnode* tmp_c = cnode->get_child( cid );
            assignSameRank(tmp_c, rank, childAllocated);
        }
    }
```

```
    void assignRootRank(Cnode* cnode, int p, int laverage, int& nodeDivision,
std::vector<int>& childAllocated)

    {
        std::vector<int> rankCount;
        std::vector<int> rankCountIndex;
        for(int i=0; i<p; i++){
            rankCount.push_back(0);
            rankCountIndex.push_back(i);
        }
        // count the rank of children
        for ( cnode_id_t cid = 0; cid < cnode->num_children(); cid++  )
        {
            Cnode* tmp_c = cnode->get_child( cid );
            rankCount[tmp_c->get_calculating_rank()] += 1;
        }

        // sort the rankCount
        for(int i=0; i<rankCount.size(); i++){
            for(int j=i+1; j<rankCount.size(); j++){
                if(rankCount[i] < rankCount[j]){
                    int temp = rankCount[i];
                    rankCount[i] = rankCount[j];
                    rankCount[j] = temp;

                    temp = rankCountIndex[i];
                    rankCountIndex[i] = rankCountIndex[j];
                    rankCountIndex[j] = temp;
                }
            }
        }

        int notAssign= -1;
        // assign the rank to the node
        for(int i=0; i<rankCount.size(); i++){
            if(childAllocated[rankCountIndex[i]] < (nodeDivision+laverage)){
                cnode->set_calculating_rank(rankCountIndex[i]);
                childAllocated[rankCountIndex[i]] +=1;
                notAssign= 0;
                break;
            }
        }
    }

    void distributeRank(Cnode *cnode, int nodeDivision, std::vector<int>& childAllocated, int
p, int laverage)
    {
        if (cnode == nullptr)
        {
            return;
        }
        // Calculate the length of the current node
        int length = caculateTotallength(cnode);

        if (length <= nodeDivision)
        {
```

```
        int rank=-1;
        for(int i=0; i<childAllocated.size(); i++){
            int childAppendable = nodeDivision - childAllocated[i];
            if(childAppendable >= length ){
                rank = i;
                break;
            }
        }

        if(rank != -1){
            // assigning longest chain to single process
            assignSameRank(cnode, rank, childAllocated);
        }else{
            // drop the call down to all children
            for ( cnode_id_t cid = 0; cid < cnode->num_children(); cid++  )
            {
                Cnode* tmp_c = cnode->get_child( cid );
                distributeRank(tmp_c,nodeDivision,childAllocated, p, laverage);
            }
            // call the assign root base on majority rule out
            assignRootRank(cnode,p,laverage, nodeDivision, childAllocated);
        }
    }
    else
    {
        // drop the call down to all children
        for ( cnode_id_t cid = 0; cid < cnode->num_children(); cid++  )
        {
            Cnode* tmp_c = cnode->get_child( cid );
            distributeRank(tmp_c ,nodeDivision,childAllocated,p, laverage);
        }
        // call the assign root base on majority rule out
        assignRootRank(cnode,p,laverage, nodeDivision, childAllocated);
    }

}

void deepest_bfs(int p, Cube& cube )
{
    int nodeDivision;
    int laverage = 0;
    std::vector<int> childAllocated(p);
    const std::vector<Cnode *> cnodes = cube.get_cnodev();
    Cnode *root;
    for (Cnode* cnode : cnodes)
    {
        if (cnode->get_parent() == nullptr)
        {
            root = cnode;
            break;
        }
    }
    assignValue(cnodes.size(), p, laverage, nodeDivision, childAllocated);
    distributeRank(root, nodeDivision, childAllocated, p, laverage);
}
```

A.7 mpi_prototype.cpp

```
/****************************************************************************
**  CUBE        http://www.scalasca.org/                                    **
****************************************************************************
**  Copyright (c) 1998-2023                                                 **
**  Forschungszentrum Juelich GmbH, Juelich Supercomputing Centre           **
**                                                                          **
**  Copyright (c) 2009-2015                                                 **
**  German Research School for Simulation Sciences GmbH,                    **
**  Laboratory for Parallel Programming                                     **
**                                                                          **
**  This software may be modified and distributed under the terms of        **
**  a BSD-style license.  See the COPYING file in the package base          **
**  directory for details.                                                  **
****************************************************************************/


#include "Cube.h"
#include "CubeMetric.h"
#include "CubeCnode.h"
#include "CubeThread.h"
#include <mpi.h>
// #include <scorep/SCOREP_User.h>

namespace pmpi
{
    void task_send_mock(Cube& cube)
    {
        MPI_Status status;
        std::vector<Metric*> metrics = cube.get_metv();
        std::vector<Cnode*>  cnodes = cube.get_cnodev();
        std::vector<Task> task(cnodes.size());

        // std::vector<MPI_Request> request(cnodes.size(),MPI_REQUEST_NULL);
        MPI_Request request=MPI_REQUEST_NULL;
        // for (int i = 0; i < cnodes.size(); i++)
        // {
        int i=0;
        int processId = cnodes[i]->get_calculating_rank();
        task[i].taskId = i;
        task[i].metric_id= metrics[0]->get_id();
        task[i].metric_calc= 0;
        task[i].cnode_id= cnodes[i]->get_id();
        task[i].cnode_calc = 0;
        task[i].sys_id = -1;
        task[i].sys_cal=-1;
        // send the task to the process
        MPI_Isend(&task[i], 1, MPI_TASK, processId, 10001, MPI_COMM_WORLD, &request);
        //}
        // MPI_Waitall(request.size(),request.data(), MPI_STATUSES_IGNORE);
    }

    void Received_Results(int p)
    {
        // MPI_Request request;
```

```
            MPI_Status recv_status;
            int resultCount = 0;
            int flag;
            int kill_signal = -1;
            int total_count=1;
            std::vector<double> result(total_count);
            std::vector<MPI_Request> request2(total_count+p,MPI_REQUEST_NULL);
            // receiving the results
            while(resultCount < total_count)
            {
                flag = 0;
                MPI_Iprobe(MPI_ANY_SOURCE, 30003, MPI_COMM_WORLD, &flag, &recv_status);
                // MPI_Probe(MPI_ANY_SOURCE, 1, MPI_COMM_WORLD, &recv_status);
                if (flag)
                {
                    int sender_rank = recv_status.MPI_SOURCE;
                    int message_tag = recv_status.MPI_TAG;

                    //   MPI_Recv(&result,   1,   MPI_DOUBLE,   sender_rank,   message_tag,
MPI_COMM_WORLD, &recv_status);
                    MPI_Irecv(&result[resultCount], 1, MPI_DOUBLE, sender_rank, message_tag,
MPI_COMM_WORLD, &request2[resultCount]);
                    std::cout<< "Result is: "<< result[resultCount] <<" received from process
"<< sender_rank<<std::endl;
                    resultCount++;
                }
            }
            // sending a kill signal Here
            for(int i = 0; i < p; i++)
            {
                // MPI_Send(&kill_signal, 1, MPI_INT, i, 6, MPI_COMM_WORLD);
                MPI_Isend(&kill_signal,    1,    MPI_INT,    i,    10002,    MPI_COMM_WORLD,
&request2[total_count+i]);
            }
            MPI_Waitall(request2.size(),request2.data(), MPI_STATUS_IGNORE);
        }
    }

    int main(int argc, char **argv)
    {
        // SCOREP_USER_REGION_DEFINE( my_region_handle10 )
        int my_rank, p, provided;
        std::future<void> a1;
        std::future<void> a5;
        // intialize with thread support
        MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
        MPI_Comm_size(MPI_COMM_WORLD, &p);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
        //                   SCOREP_USER_REGION_BEGIN(                   my_region_handle10,
"Cube_opened",SCOREP_USER_REGION_TYPE_COMMON )
        Cube cube(CUBE_ENFORCE_ZERO,my_rank,p);
        cube.openCubeReport("children-example.cubex");
        // SCOREP_USER_REGION_END( my_region_handle10 )

        if (my_rank==0)
        {
            a1 = std::async(std::launch::async, pmpi::task_send_mock, std::ref(cube));
```

```
        // int totalCount=cube.get_cnodev().size();
        a5 = std::async(std::launch::deferred,pmpi::Received_Results,p);
        a1.wait();
        a5.wait();
    }
    // cube.closeCubeReport();
    cube.a2.wait();
    MPI_Type_free(&MPI_TASK);
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}
```

A.8 cube_bt.cpp

```
    // Function to generate random severity values for balanced tree
    double generateRandomSeverity(const Cnode *tmp_cnode, int i)
    {
        return static_cast<double>(0.0001 * i * tmp_cnode->get_id());
    }

    int main(int argc, char* argv[])
    {
        int total_cn=65;
        int total_stn=1;
        int total_lg=1;
        int total_loc=100;
        int childCount = 2; // Number of children for each cnode
        string cube_name="tree-example-65Cn3";
        Metric* met0, * met1;

        try
        {
            Cube cube;

            // Build metric tree
            met0 = cube.def_met("Time", "time", "DOUBLE", "sec", "", "", "root node", NULL,
CUBE_METRIC_EXCLUSIVE); // without using mirror
            met1 = cube.def_met("User time", "usertime", "DOUBLE", "sec", "", "", "root node",
NULL, CUBE_METRIC_INCLUSIVE); // without using mirror

            // Build call tree
            string mod = "/ICL/CUBE/example.c";

            // Define the root cnode
            Region* rootRegion = cube.def_region("Cnode 0 (Root)", "cnode_0", "mpi",
"barrier", 1, 1, "", "1st level", mod);
            Cnode* rootCnode = cube.def_cnode(rootRegion, mod, 1, NULL);


            // Build child cnodes recursively
            vector<Cnode*> cnodes;
            cnodes.push_back(rootCnode);

            int level = 1;
            int cnodesCount = 1;

            while (cnodesCount < total_cn) {
                int currentLevelCnodesCount = pow(childCount, level);
                int remainingCnodesCount = total_cn - cnodesCount;
                int levelCnodesCount = min(currentLevelCnodesCount, remainingCnodesCount);

                for (int i = 0; i < levelCnodesCount; i++) {
                    int currentIndex = cnodesCount + i;
                    string cnodeName = "Cnode " + to_string(currentIndex);
                    Region*     region     =     cube.def_region(cnodeName,     "cnode_"     +
to_string(currentIndex),  "mpi",  "barrier",  currentIndex  +  1,  currentIndex  +  1,  "",
to_string(level) + " level", mod);
```

```
                // Calculate the parent index correctly to create a tree structure
                int parentIndex = (currentIndex - 1) / childCount;
                Cnode* parentCnode = cnodes[parentIndex];

                Cnode* cnode = cube.def_cnode(region, mod, currentIndex + 1, parentCnode);
                cnodes.push_back(cnode);
            }
            level++;
            cnodesCount += levelCnodesCount;
        }

        // Build location trees
        vector<LocationGroup*> locationGroups(total_lg); // Vector to store location
groups
        vector<SystemTreeNode*> systemTreeNodes(total_stn); // Vector to store system tree
nodes
        vector<vector<vector<vector<Location*>>>> locations(total_stn,
vector<vector<vector<Location*>>>(1, vector<vector<Location*>>(total_lg,
vector<Location*>(total_loc))));


        for (int i = 0; i < total_stn; i++) {
            systemTreeNodes[i] = cube.def_system_tree_node("Machine " + to_string(i), "",
"", nullptr);
            cube::LocationGroupType groupType = cube::CUBE_LOCATION_GROUP_TYPE_PROCESS;
            for (int k=0; k<total_lg; k++)
            {
                locationGroups[k] = cube.def_location_group("Location Group " +
to_string(k), k, groupType, systemTreeNodes[i]);
                for (int j = 0; j < total_loc; j++) {
                    Location* location = cube.def_location("Thread " + to_string(i) + "_"
+ to_string(j), i * 10 + k, cube::CUBE_LOCATION_TYPE_CPU_THREAD, locationGroups[k]);
                    locations[i][0][k][j] = location;
                }
            }
        }

        cube.initialize();

        // Set random severity values for metrics, cnodes, and threads
        for (Cnode* cnode : cnodes) {
            for (int i = 0; i < total_stn; i++) {
                for (int j = 0; j < 1; j++) {
                    for (int k = 0; k < total_lg; k++) {
                        for (int l = 0; l < total_loc; l++) {
                            Location* location = locations[i][j][k][l];
                            cube.set_sev(met0, cnode, location,
generateRandomSeverity(cnode, l));
                            cube.set_sev(met1, cnode, location,
generateRandomSeverity(cnode, l));
                        }
                    }
                }
            }
        }

        // Output file
```

```
        cube.writeCubeReport(cube_name);
    }
    catch (const RuntimeError& error)
    {
        cout << error.what() << endl;
    }
    return 0;
}
```