Concurrency and Computation: Practice and Experience

WILEY

RESEARCH ARTICLE OPEN ACCESS

Mpisee: Communicator-Centric Profiling of MPI Applications

Ioannis Vardas[1] | Jesper Larsson Träff[1] | Ruben Laso[2] | Sascha Hunold[1]

[1]Research Unit Parallel Computing, Faculty of Informatics, TU Wien, Vienna, Austria | [2]Research Group Scientific Computing, Faculty of Computer Science, University of Vienna, Vienna, Austria

Correspondence: Ioannis Vardas (vardas@par.tuwien.ac.at)

Received: 3 September 2024 | **Revised:** 15 April 2025 | **Accepted:** 30 May 2025

Funding: Austrian Science Fund (FWF) in whole or in part [10.55776/P31763, 10.55776/P33884].

Keywords: High-Perfomance Computing | HPC | Message Passing Interface | MPI | parallel computing | performance analysis | performance profiling

ABSTRACT

`mpisee` is a lightweight profiling tool designed to track MPI communication operations per communicator, providing fine-grained insights into MPI applications that use communicators to partition MPI communication. While existing profiling tools offer valuable information, they may limit detailed analysis and optimization for such MPI applications, as they do not associate MPI communication with their communicator. Additionally, `mpisee` categorizes MPI communication operations based on message size, offering more granular information. It uses an SQLite database to efficiently store the profiling data, enabling users to analyze the application's profile from various perspectives, focusing on specific MPI ranks, operations, and more. Our analysis shows that `mpisee` incurs less than 5% overhead, performing on par with other state-of-the-art profilers. We demonstrate `mpisee` 's effectiveness by profiling and analyzing an FFT application, revealing potential performance bottlenecks related to the `MPI_Alltoallv` collective operation on small communicators and insights not available by other profilers. Leveraging this detailed information, we improved the application's overall performance by selecting different algorithms for `MPI_Alltoallv` and measuring their performance on different communicators with `mpisee`. This study illustrates `mpisee` 's utility and highlights the significant advantages of a communicator-centric approach in MPI profiling.

1 | Introduction

Optimizing the performance of applications is a key challenge in High-Perfomance Computing (HPC) where even minor inefficiencies can lead to significant slowdowns, especially as applications scale across thousands of processing nodes. Achieving high performance in these complex environments requires detailed profiling, analysis, and fine-tuning of communication patterns between processes. The Message Passing Interface (MPI) is the dominant programming model for enabling parallelism across multiple compute nodes, making it indispensable in HPC [1]. MPI provides a robust set of communication operations that facilitate data exchange and synchronization among parallel processes. This work focuses on the profiling of MPI applications as a less intrusive and more practical approach to performance analysis, especially at a large scale.

An essential abstraction of MPI is the communicator, which is a distributed object that comprises an ordered set of processes and defines a communication context in which the participating processes can exchange messages [2]. All communication operations

Abbreviations: ECP, Exascale Computing Project; FFT, Fast Fourier Transform; HPC, High-Perfomance Computing ; MPI, Message Passing Interface.

in MPI, whether point-to-point, collective, or one-sided, rely on communicators.

MPI communicators can also control the process-to-core mapping via the virtual topology mechanism. A virtual topology allows the communicator to store a logical arrangement of processes determined by the algorithm and the geometry of the problem. For instance, in stencil computations on structured grids, where each element's value depends on its immediate neighbors, a Cartesian virtual topology models the grid structure within MPI, assigning each process a position within the virtual grid. The MPI library can reorder MPI ranks within the Cartesian communicator to improve process-to-core mapping, minimizing communication overhead and improving application performance [3–5].

A particular use case of communicators is problem decomposition, a fundamental technique where a computational problem is divided into smaller subproblems that can be solved concurrently by different processes in a distributed system. This approach distributes work across multiple processors, exploiting parallelism, and can enhance the efficiency of large-scale computations. Communicators structure communication within these subproblems by creating process groups linked to specific parts of the problem. Molecular dynamics applications, including GROMACS [6], use such techniques to structure communication, as do computational fluid dynamics simulators (e.g., OpenFOAM [7]) and parallel FFT libraries [8].

Figure 1 illustrates the concept of problem decomposition, showing an example of an MPI application with eight processes that are grouped into smaller groups using communicators. Figure 1a shows that all eight processes are grouped into two communicators, and Figure 1b shows a different communication structure with four smaller communicators, each containing two processes. These communicators define distinct communication groups, enabling structured communication within each group, such as performing specific collective communication calls like `MPI_Allreduce`, `MPI_Bcast`, and others, reducing unnecessary message exchange between unrelated processes.

A similar approach is followed by Träff and Hunold [9], who use communicators to decompose MPI collectives and structure them to exploit multi-lane communication. Their method groups processes into multiple small communicators to parallelize communication tasks and use multiple communication paths. In this way, they maximize the network bandwidth and reduce the communication time by reducing contention.

Despite the central role of communicators, state-of-the-art MPI profiling tools, to the best of our knowledge, are communicator-oblivious. They maintain a global view and profile MPI applications on a per-process basis, disregarding the applications' communicators for performing MPI operations. This means that they do not track MPI communicators and do not associate MPI communication operations with them. Tracking communicators is challenging for external profiling tools because while communicators themselves are exposed to the profiler, the identification mechanism that the MPI library uses is not. Therefore, profiling tools aiming to keep track of MPI communicators must address this problem outside the MPI library.

1.1 | Motivation

To illustrate the limitations of communicator-oblivious profiling, we use a sample application, shown in Listing 1. This application creates two subcommunicators by splitting `MPI_COMM_WORLD`, each containing half of the processes from `MPI_COMM_WORLD`. The application performs 30 calls to `MPI_Allreduce` in `MPI_COMM_WORLD` and 100 additional calls in each subcommunicator. We executed this application with 1024 processes and profiled it using Score-P [10] and mpiP [11], which are well-established communicator-oblivious profiling tools. The profiles generated by these tools are shown in Figure 2.

LISTING 1 | Sample MPI application that creates two communicators by splitting MPI_COMM_WORLD and performing MPI_Allreduce in them.

```
#define WORLD_ALLREDUCES 30
#define COMM_ALLREDUCES 100
#define ALLREDUCECOUNT 1000

MPI_Comm rootcomm;
int rank, size;
// Perform allreduce on MPI_COMM_WORLD
for (int i = 0; i < WORLD_ALLREDUCES; i++) {
      MPI_Allreduce(send_buffer, recv_buffer,
      ALLREDUCECOUNT, MPI_INT, MPI_SUM,
      MPI_COMM_WORLD);
}
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_split(MPI_COMM_WORLD, rank > (size/2 -
    1), rank, &rootcomm);
// Perform allreduce on the new communicator
for (int i = 0; i < COMM_ALLREDUCES; i++) {
      MPI_Allreduce(send_buffer, recv_buffer,
      ALLREDUCECOUNT, MPI_INT, MPI_SUM, rootcomm
    );
}
```

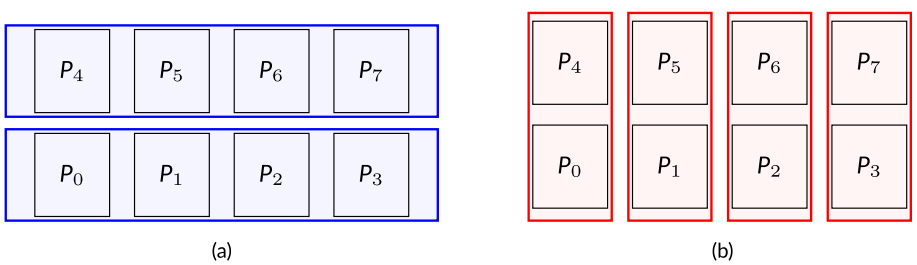

FIGURE 1 | An example of organizing eight processes of an MPI application into different communication groups using communicators. (a) Two communicators, each defining a group of four processes, (b) Four communicators, each defining a group of two processes.

```
Estimated aggregate size of event trace:                   9MB
Estimated requirements for largest trace buffer (max_buf): 9kB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       4097kB
(hint: When tracing set SCOREP_TOTAL_MEMORY=4097kB to avoid intermediate flushes
 or reduce requirements using USR regions filters.)

flt     type max_buf[B]  visits  time[s] time[%]   time/visit[us]  region
         ALL      8,717 138,240 11704.05   100.0         84664.72  ALL
         MPI      8,676 137,216 11192.23    95.6         81566.47  MPI
      SCOREP         41   1,024   511.83     4.4        499829.80  SCOREP

         MPI      8,580 133,120  9213.82    78.7         69214.42  MPI_Allreduce
      SCOREP         41   1,024   511.83     4.4        499829.80  scorep_profiler_bench
         MPI         24   1,024     0.00     0.0            4.25  MPI_Comm_rank
         MPI         24   1,024    92.77     0.8        90591.39  MPI_Comm_split
         MPI         24   1,024   326.02     2.8       318382.03  MPI_Finalize
         MPI         24   1,024  1559.61    13.3      1523055.55  MPI_Init
```

(a)

```
---------------------------------------------------------------------------
@--- Aggregate Time (top twenty, descending, milliseconds) ----------------
---------------------------------------------------------------------------
Call                 Site       Time     App%    MPI%      Count    COV
Allreduce               2   7.73e+06    76.03   79.66     102400   0.06
Allreduce               1   1.88e+06    18.53   19.41      30720   0.12
Comm_split              3   8.98e+04     0.88    0.93       1024   0.03
---------------------------------------------------------------------------
@--- Aggregate Collective Time (top twenty, descending) -------------------
---------------------------------------------------------------------------
Call                 MPI Time %             Comm Size              Data Size
Allreduce                  4.22        512 -     1023       2048 -     4095
Allreduce                  1.27       1024 -     2047       2048 -     4095
---------------------------------------------------------------------------
```

(b)

FIGURE 2 | Communicator-oblivious profiles of the sample MPI application by Score-P and mpiP profilers. (a) The Score-P profile showing aggregated data for MPI operations, (b) A subset of the mpiP profile categorizing `MPI_Allreduce` calls by callsite.

The profile generated by Score-P, as depicted in Figure 2a, indicates a total of 133,120 calls to `MPI_Allreduce`, with a cumulative execution time of 9213 s. This result occurs because Score-P aggregates the number of calls across all processes and communicators, yielding the breakdown: 30 calls/process × 1024 processes/communicator × 1 communicator + 100 calls/process × 512 processes/communicator × 2 communicators = 133,120 calls. However, this profile completely overlooks the existence of the subcommunicators created during execution.

In contrast, the subset of the mpiP profile, shown in Figure 2b, associates `MPI_Allreduce` with two communicator sizes (512–1023 and 1024–2047), as seen in the *Aggregate Collective Time* section. However, this profile is still limited, as it fails to differentiate between the two distinct communicators of size 512, created by `MPI_Comm_split`. The *Aggregate Time* section offers slightly more detail, categorizing the `MPI_Allreduce` calls by call site, which helps us understand that there are two distinct calls to `MPI_Allreduce`. Specifically, it shows that `MPI_Allreduce` at Site 2 takes significantly longer than the one at Site 1. Summing the time of `MPI_Allreduce` at these two call sites, we get 7730 s + 1880 s = 9610 s, which is close to the `MPI_Allreduce` time from Score-P. While knowledge of the source code allows us to infer that the calls at Site 2 occur in split communicators, the profile does not reveal in which of the two split communicators these calls took place. The mpiP profile shows per-process information, which is extensive and therefore, it is not included in its entirety here.

Such communicator-oblivious profiles do not allow us to distinguish the performance of `MPI_Allreduce` across different communicators, making it difficult to accurately assess its specific impact on each communicator. Without this distinction, we cannot clearly understand the impact of targeted optimizations for specific communicators or communicator sizes. For example, while we can select different algorithms for `MPI_Allreduce` based on the size of the communicator, we cannot evaluate how these choices affect the performance of individual communicators. We can observe the overall result in the total execution time, but we cannot determine the specific contribution of each communicator to that result. This limits our ability to fine-tune performance and effectively address potential bottlenecks.

1.2 | Contributions

To address these limitations, we introduce `mpisee`, a lightweight MPI profiler that implements a communicator-centric paradigm in MPI profiling.[1] Unlike traditional approaches that aggregate global data, `mpisee` tracks MPI calls per communicator, providing a detailed characterization of MPI communication, including time spent, number of calls, and the volume of data transferred for each MPI function within individual communicators. Additionally, `mpisee` categorizes MPI communication calls based on their buffer size. To efficiently store and manage profiling data, we leverage the SQLite library [12], enabling flexible querying and analysis of results.

Our evaluation of mpisee provides a comprehensive overhead analysis, rigorously measuring and characterizing the overhead introduced by mpisee compared to state-of-the-art profiling tools rather than just baseline runs. Additionally, we demonstrate mpisee 's capabilities by analyzing the performance of an MPI application that uses a real-world FFT library, revealing performance bottlenecks that traditional profiling methods overlook. We show that insights from mpisee 's profiles can guide targeted optimizations, enhancing application performance. Finally, this paper goes beyond introducing mpisee ; it showcases the implementation of a communicator-centric paradigm in MPI profiling. While mpisee embodies our approach, this paradigm can be integrated into existing MPI profilers.

2 | Related Work

Our research focuses on tools for performance analysis of MPI-based applications, which we categorize into profiling and tracing tools. Profiling tools aggregate data per event to provide a summarized view of application behavior. In contrast, tracing tools provide a more fine-grained perspective by recording each event's start and end timestamps. While tracing tools offer a plethora of features, including the ability to replay execution traces, and can be applied to a broader range of applications beyond MPI, they typically incur higher overheads than profiling tools.

mpiP [11] and IPM [13] are MPI profilers that share many similarities with mpisee. Both tools rely on PMPI, the profiling interface of MPI [2, Section 15.2], and they report the accumulated time spent in different MPI communication operations. mpiP presents profiling statistics per *call site*, a point in the program where the MPI call is performed. In this way, it distinguishes the MPI calls per call site. IPM categorizes the MPI calls per buffer size, e.g., it distinguishes between a call to MPI_Bcast with 30 bytes to another call with 500 bytes. IPM does not distinguish the MPI calls per communicator, whereas mpiP maintains information on the communicator size but does not distinguish between different communicators. For example, two calls of the same MPI operation on different communicators of the same size are aggregated. Finally, both mpiP and IPM use a text file as output, which can be inefficient for large profiles.

Score-P [10] is a versatile tool that supports various programming models beyond MPI, such as OpenMP and CUDA. It employs compile-time instrumentation and has two modes of operation: Profiling and tracing. In profiling mode, it uses the PMPI interface to record timing statistics of MPI operations, similar to mpisee. In tracing mode, it records the start and end times of MPI events and functions, offering a more fine-grained analysis. Additionally, it supports automatic instrumentation and can track events using a sampling approach. It supports output in OTF2, CUBE4, and TAU formats for interfacing with performance analysis tools such as Scalasca [14] and TAU [15].

Vampir [16] is a commercial performance visualization tool that can process and analyze traces, particularly those recorded in OTF and OTF2 formats (such as the ones generated by Score-P). Vampir provides extensive filtering capabilities, including filtering operations by communicator, which enables a form of communicator-centric performance analysis. However, this approach differs from mpisee in that it operates on full traces rather than aggregated profiles, requiring potentially larger storage overhead and post-processing time.

HPCToolkit [17], like Score-P, supports various programming models and can do profiling and tracing. It uses a sampling-based approach to create program traces and record performance data. HPCToolkit does not depend on the PMPI interface. Instead, it relies on call stack unwinding. It provides a graphical user interface (GUI) to present the performance data, primarily offering code-centric views and enabling users to assess the performance variability across processes and threads.

Scalatrace [18] and Pilgrim [19] are MPI tracers focusing on reducing the size of recorded traces, as traces tend to become exceedingly large. ScalaTrace uses a two-step compression scheme, distinguishing between intra- and inter-node compression. Intra-node compression is performed during MPI program execution, where each process compresses its local trace data. Subsequently, these compressed local traces are merged into a global profile, leveraging pattern recognition to exploit repetitive sequences for further inter-node compression. Pilgrim uses lossless compression by building a context-free grammar during the program's execution. It stores the MPI calls and their parameters in a signature table to be used as terminal symbols in the grammar.

In most MPI libraries, communicators are internally identified by a unique integer context ID that is consistent across all processes in the communicator's group. More specifically, the context ID is the same on all processes that are part of the communicator and is unique among all communicators on a given process [20]. Context IDs ensure that communication operations match only within the same communicator.

The work of Gropp and Thakur [20] addresses the problem of identifying the MPI communicators within the MPI library. More specifically, their work focuses on the generation of context IDs to identify communicators in multithreaded MPI environments. The challenge in multithreaded MPI environments is to avoid deadlocks and race conditions when multiple threads simultaneously attempt to generate new context IDs. The authors address this by developing an algorithm that coordinates the generation process across threads. Their algorithm uses a shared bitmask and thread locks to ensure thread-safe and efficient context ID generation, preventing deadlocks and unnecessary repetitions.

Profiling tools that aim to track MPI communicators require an external tracking or naming scheme, as the MPI library does not expose its internal context IDs of communicators to profilers. Solving this problem was a key focus in the development of mpisee and other profiling tools.

Geimer et al. [21] address the communicator tracking problem outside MPI. In their scheme, when a communicator is created, the process 0 in a communicator maintains a state variable to count the number of communicator calls up to this point. Then, it broadcasts this count along with the global MPI rank of the process ranked 0 in the newly created communicator and increases its counter value after the broadcast. This strategy

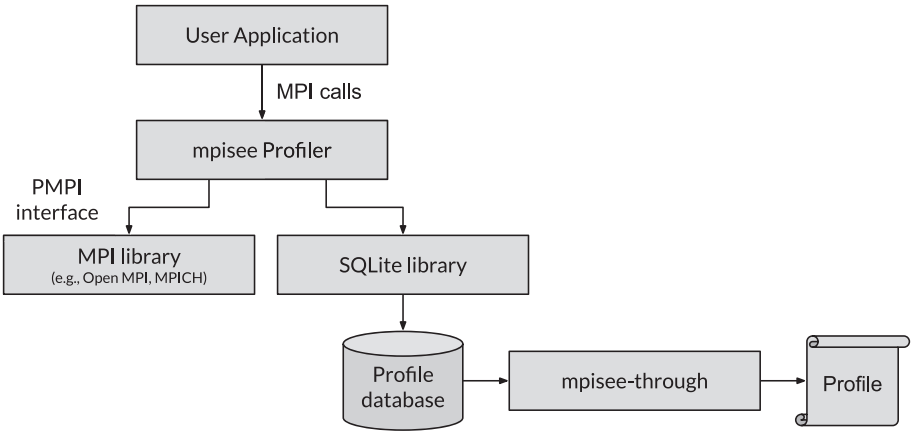

FIGURE 3 | An overview of the architecture of `mpisee`, depicting how `mpisee` interacts with the user application and the MPI library and the SQLite library to create the profile database.

ensures that all processes within the communicator have the necessary information for later unification. At the end of the measurement, these two numbers are used to create a global identifier for each communicator. Each communicator is assigned a global identifier sequentially, starting with those defined by rank 0. An exclusive prefix reduction calculates the total number of communicators defined by all preceding ranks, which is then shared across processes to adjust local identifiers into globally unique ones. The total number of communicators and their participant ranks are determined, ensuring that each communicator has a unique, globally consistent identifier.

This work significantly extends our previous implementation [22], which had fundamental limitations: First, it required collective calls within newly created communicators, which were incompatible with the semantics of `MPI_Comm_idup` and `MPI_Comm_create_group`. Second, it represented communicator names as strings, leading to memory usage proportional to the depth of nested communicator creation calls. The current implementation resolves these issues by adopting a scheme that maintains partial communicator information, ensuring semantic correctness and constant memory usage regardless of the nesting depth. Furthermore, we added support for MPI one-sided operations and neighborhood collectives, and extended support for Fortran programs that use `MPI_IN_PLACE`, `MPI_BOTTOM`, `MPI_STATUSES_IGNORE`, `MPI_STATUS_IGNORE`, and `MPI_UNWEIGHTED`. In addition, we introduce several new features, such as the categorization of MPI communication by buffer size and SQLite database integration for efficient storage and analysis (replacing the previous CSV format).

In this paper, we explain the new features, present a comprehensive overhead analysis, including a comparison to two other state-of-the-art profiling tools, and demonstrate `mpisee`'s practical utility by profiling an FFT application, revealing performance bottlenecks that traditional profiling methods could overlook.

3 | Methods and Implementation

We use the standard profiling interface of MPI called PMPI to implement `mpisee`. PMPI provides an alternative entry point to each MPI function using the `PMPI_` prefix [2, Section 15.2]. This allows `mpisee` to intercept and wrap MPI calls from the user program, collecting necessary information before and after executing the underlying PMPI call. Notably, `mpisee` does not require compiler instrumentation and can be loaded at runtime using the `LD_PRELOAD` environment variable without recompilation.

Figure 3 shows an overview of the architecture of `mpisee`. The `mpisee` profiling infrastructure resides between the user application and the MPI library. Therefore, when the user application calls MPI functions, `mpisee` intercepts these function calls via its function wrappers. This allows `mpisee` to gather profiling information before and after invoking the corresponding PMPI function to execute the actual MPI operation. Upon intercepting the `MPI_Finalize`, it outputs the profiling information into an SQLite database file on disk. The SQLite file database can be further analyzed by our external tool called `mpisee-through`.

This section discusses the implementation details of `mpisee` in the following order: Initialization, communicator creation, MPI communication operations profiling, finalization, and SQLite library integration. Finally, we present `mpisee-through`, our data analysis tool. Nevertheless, before going into the implementation details, we first discuss how `mpisee` keeps track of the MPI communicators using a communicator naming scheme.

3.1 | Naming Scheme for MPI Communicators

The unique feature of `mpisee` is that it associates MPI communication operations with communicators while providing the user with a global view of the MPI application. The fundamental challenge in achieving this is the lack of a native, globally consistent way to identify or name communicators across all processes. While each process can identify a communicator using its local handle, these handles are process-specific and cannot be directly compared or correlated across processes. A unified representation is essential to accurately attribute MPI communication events to each communicator, requiring processes to agree upon this unified representation. However, introducing synchronization within communicator creation

TABLE 1 | Unique characters used by `mpisee` to identify different MPI communicator creation functions.

Ch.	Communicator creator	Ch.	Communicator creator	Ch.	Communicator creator
a	MPI_Cart_create	g	MPI_Dist_graph_create	r	MPI_Graph_create
b	MPI_Cart_sub	j	MPI_Dist_graph_create_adjacent	s	MPI_Comm_split
c	MPI_Comm_create	i	MPI_Comm_idup	t	MPI_Comm_split_type
d	MPI_Comm_dup	o	MPI_Comm_idup_with_info	u	MPI_Comm_create_group

calls would violate their semantics for some MPI functions, e.g., `MPI_Comm_idup` is not a blocking call. The dynamic creation and destruction of communicators during execution further complicates maintaining a consistent global view. To address these challenges, `mpisee` generates consistent communicator names across processes, merging partial data into a unified global view during `MPI_Finalize`.

Since MPI processes can participate in multiple communicators, we uniquely identify each process-communicator pair with a two-tuple key:

1. The global MPI rank of the process ranked 0 in the newly created communicator and
2. A local counter of communicator creation calls made by this process.

The names are not globally consistent before `MPI_Finalize`, because each process only maintains the local count of communicator creation calls for each communicator it participates in. The names will be made globally consistent by synchronizing the local two-tuple keys for every communicator during `MPI_Finalize`. To achieve this, in each communicator, process 0 broadcasts its local two-tuple key to the other members of this communicator. Synchronization is important because the process with global rank 0 gathers all profiling data from each process at the end to create the profile database. Therefore, communicators must be consistently named before being received by process 0. For readability, the names also include a character identifying the communicator's creation call, which is shown in Table 1.

Figure 4 illustrates this scheme in the context of the `MPI_Comm_split` function, which can create multiple new communicators based on the color value provided. This example uses four MPI processes ranked from 0 to 3. The left side of the Figure shows how each process manages its two-tuple keys using the local communicator creation counters and the global MPI rank. The right side depicts how each process maintains the profile data of MPI for each communicator it participates in.

When `MPI_Init` is called, it creates the `MPI_COMM_WORLD` communicator, and each process creates a tuple named `W`, as shown on the left side of Figure 4. Then, process 0 in the communicator records the value of the local communicator creation call counter for this communicator, which is 0, in tuple `W`. The counter is then incremented to 1. On the right side, we notice that each process creates a local profile, denoted as *P*W0*, for this communicator to record MPI operations.

When `MPI_Comm_split` is called, it creates two new communicators, grouping rank 0 with rank 1 and rank 2 with rank 3. Each process records both the character `S` for `MPI_Comm_split` and the counter of the local communicator creation calls, the latter being 1 at this point. Then, as shown on the right side, each process creates another local profile *P*S1* (separate from *P*W0*) for this communicator. Notice that, initially, the two new communicators created by `MPI_Comm_split` are temporarily named the same way by all processes. This naming is resolved in the next step, when the keys are synchronized and unified.

Processes synchronize and unify their two-tuple keys after `MPI_Finalize`. For each communicator, the process 0 broadcasts its two-tuple key to all other processes within the communicator. Once all processes in a communicator receive the two-tuple key, they rename the local profiling data accordingly. For example, processes 0 and 1, which participate in the same sub-communicator from `MPI_Comm_split`, rename their local profile from *P*S1* to *P*S0.1*, while processes 2 and 3 rename *P*S1* to *P*S2.1*. Finally, process 0 in `MPI_COMM_WORLD` gathers all the local profiles and combines them communicator-wise (e.g., all *P*S2.1* are combined to form *PS2.1*) to create a global view of the application.

The above naming scheme enables `mpisee` to accurately track the MPI communicators and categorize MPI communication per communicator. The following sections describe how we implement this naming scheme in `mpisee`.

3.2 | Initialization

`mpisee` initializes its internal data structures and state by intercepting `MPI_Init`. When intercepting the `MPI_Init` function, `mpisee` calls the corresponding PMPI function to initialize MPI and then performs its initialization as shown in Listing 2. First, it creates two attribute keys: One key to store (and retrieve) the communicator data and another for the operation data map within the communicator object, which is explained in more detail in the following section. We create these attribute keys using the `MPI_Comm_create_keyval`. In this way, they are managed by the MPI library, ensuring the appropriate application of user-defined copy and delete functions during communicator lifecycle events such as duplication and destruction.

We use different keys for communicator and operation data because the operation data is accessed much more frequently than the communicator data. The communicator data are accessed only during the creation and deallocation of the communicator and once during `MPI_Finalize`. In contrast, the

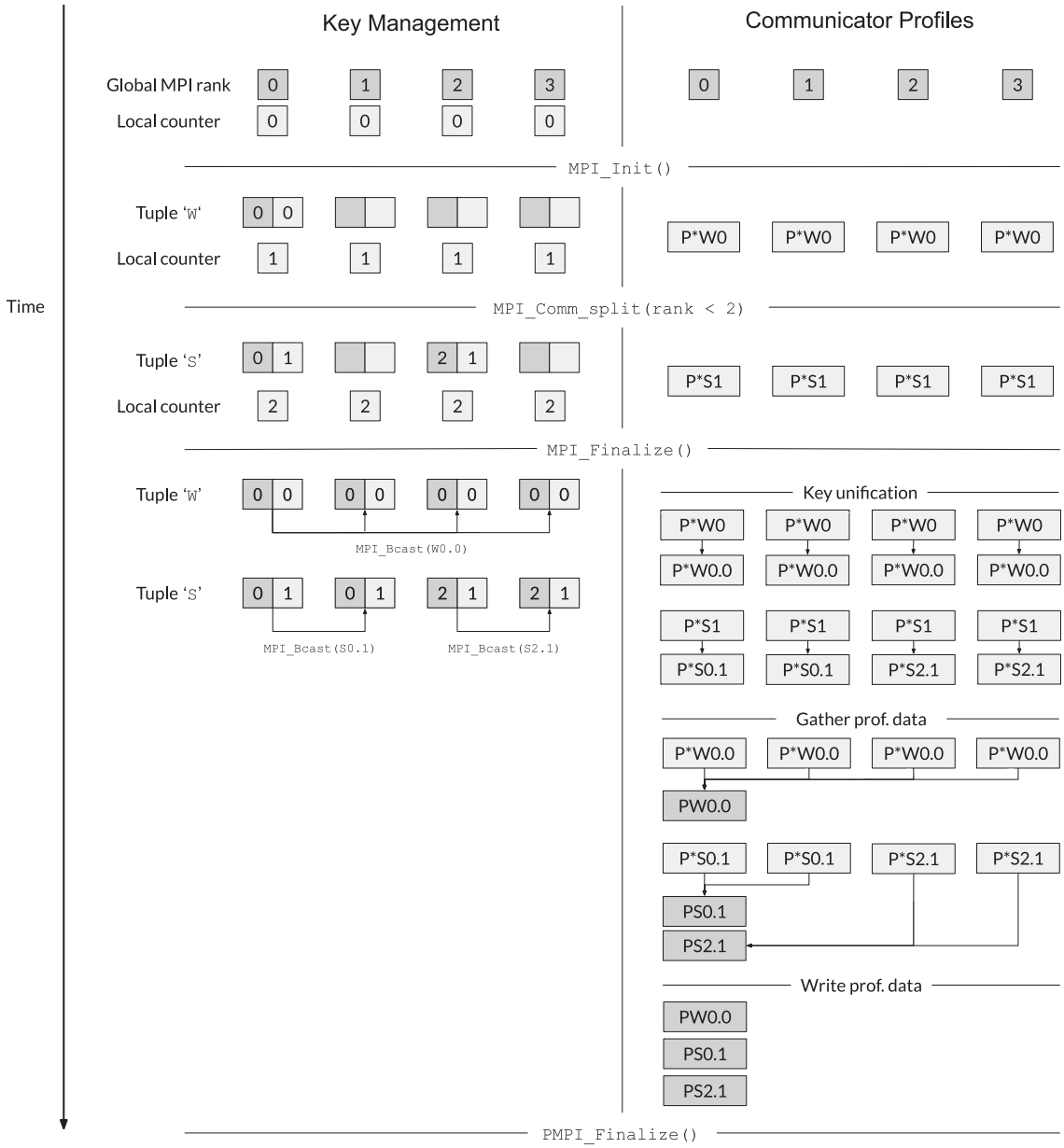

FIGURE 4 | An example of the `mpisee` communicator naming scheme using `MPI_Comm_split` with four processes. The figure depicts how processes manage their two-tuple keys on the left and the communicator profiling data on the right. We use different shades of grey to denote global (dark) and local (light) information.

operation data are accessed every time an operation occurs in the communicator. By separating these keys, we avoid unnecessary data retrieval each time the operation data is accessed. These keys are stored in a global table, shown in Listing 3, Line 23.

After creating the keys, the `MPI_Init` wrapper function calls the `comm_prof_init` function, which allocates and attaches the profiling objects to the communicator object of `MPI_COMM_WORLD`. We explain this function in the following section, which discusses the creation and deallocation of the communicator.

3.3 | Communicator Creation

With the initialization complete, `mpisee` can now start tracking the MPI communicators that are created throughout the application's execution. During communicator creation, `mpisee` allocates and initializes the necessary profiling objects to maintain the profiling information within each communicator. It leverages the MPI attribute caching mechanism [2, Section 7.4] to attach the profiling objects to a communicator object. This enables fast storage and retrieval of profiling data directly linked to these communicators [23].

LISTING 2 | The MPI_Init wrapper: Creates two keys for communicator and operation data and calls the communicator profiling function.

```
int MPI_Init(int *argc, char ***argv) {
    PMPI_Init(argc, argv); // Call the original
    MPI_Init

    // Create the key for communicator data
    MPI_Comm_create_keyval(MPI_COMM_DUP_FN,
    MPI_COMM_NULL_DELETE_FN, &keys[0],NULL);

    // Create the key for the operation data map
    MPI_Comm_create_keyval(MPI_COMM_DUP_FN,
    MPI_COMM_NULL_DELETE_FN, &keys[1],NULL);

    comm_prof_init(MPI_COMM_WORLD,'W'); //
    Initialize the communicator profiling
    objects
    // Rest of the MPI_Init code
}
```

LISTING 3 | The necessary data types and global data structures maintained by mpisee.

```
1  // Communicator data stored in communicators and
       used for naming in the end
2  typedef struct communicator_data {
3      int size;
4      int comms;
5      char id;
6  } comm_data;
7
8  // MPI operation data maintained within the map
9  typedef struct operation_data {
10     double time;
11     int num_messages;
12     uint64_t volume;
13 }op_data;
14
15 // Operation data map stored in communicators
16 typedef struct operation_data_map {
17     std::unordered_map<int, op_data> map;
18 }op_data_map;
19
20 int local_cid = 0; // Local communicator call
       counter
21
22 // Global table to store and retrieve keys for
       communicator and operation data
23 int keys[2]; // keys[0] for communicator data
24              // keys[1] for MPI operation data
25
26 // Global table to reference communicators
27 std::vector<MPI_Comm> comms_table;
28
29 // Global table to associate request objects
       with communicators
30 std::unordered_map<MPI_Request, MPI_Comm>
       requests_map;
```

`mpisee` maintains two types of objects within each communicator: The communicator data and the operation data map. We show these data types in Listing 3. The communicator data contains the communicator size, the number of communicator calls by the time this communicator was created, and a character denoting the type of communicator creation call. This information is crucial for our communicator tracking scheme to name the communicators during `MPI_Finalize`. The operation data map (shown in Listing 3, Line 16) maintains the actual profiling data for MPI operations within this communicator; the time, number of calls, and volume (Line 9). The key to the operation data map is created by combining the enumerated type of the MPI operation with its message size. We implemented the operation data map using the `std::unordered_map` from the C++ Standard Library because it is a dynamic data structure that provides flexibility in efficiently handling varying amounts of profiling data. It also offers an average $\mathcal{O}(1)$ time complexity, ensuring efficient insertion and retrieval operations while avoiding dependence on external libraries.

LISTING 4 | Communicator profiling initialization function: Creates the data objects for profiling MPI calls within the communicator.

```
void comm_prof_init(MPI_Comm comm,char id){
    comm_data *commdata;
    op_data_map *opdata_map;

    commdata = new comm_data();  // Allocate the
     communicator data object
    opdata_map = new op_data_map(); // Allocate
    the operation data map object

    // Assign the communicator data
    commdata->comms = local_cid++; // Record and
     increase the communicator call counter
    commdata->id = id;
    PMPI_Comm_size(comm, &commdata->size); //
    Record the communicator size

    PMPI_Comm_set_attr(comm, keys[0], commdata);
     // Attach the communicator data attribute
    to communicator using keys[0]
    PMPI_Comm_set_attr(comm, keys[1], opdata_map
    ); // Attach the operation data attribute to
     communicator using keys[1]

    comms_table.push_back(comm); // Record the
    communicator in the table
}
```

Communicator creation is handled by `comm_prof_init`, shown in Listing 4. This function takes the newly created communicator object and a character ID as input and allocates two profiling objects: One for communicator data and another that contains a map of the operation data. It assigns the values to the necessary communicator data: Communicator size, the number of communicator calls by the time this communicator was created, and a character denoting the type of communicator creation call. It attaches the communicator data to the communicator object using the attribute caching function `PMPI_Comm_set_attr` and the key with index 0 from the keys table (see Listing 3, Line 23). The operation data is attached using the key with index 1. Finally, it adds a reference of the communicator to the `comms_table`. We chose an `std::vector` to implement the `comms_table` since it offers amortized $\mathcal{O}(1)$ complexity for appending elements, which can be frequent as they occur during communicator creation. Lookup operations for data in this table, with searches of complexity $\mathcal{O}(n)$, are infrequent since they are performed during `MPI_Comm_free` and `MPI_Finalize`.

3.4 | Profiling MPI Communication Operations

Once communicators are created and properly tracked, `mpisee` can start profiling the MPI communication events that occur within these communicators. MPI communication operations enable point-to-point, collective, and one-sided communication. These are the functions that `mpisee` primarily profiles. To profile these operations, `mpisee` employs function wrappers that perform the following actions:

1. *PMPI*: Call the corresponding PMPI function.
2. *Measurement*: Measure the elapsed time of the PMPI function.
3. *Data retrieval*: Retrieve the operation data map associated with the communicator by calling `PMPI_Comm_get_attr`.
4. *Key generation*: Generate the key for the map from the MPI function and the send buffer size.
5. *Data update*: Update the data for volume, number of calls, and time spent in this MPI function on the map.

`mpisee` categorizes and presents statistics of MPI calls based on send buffer size ranges, referred to as buckets. Each bucket represents a specific range of buffer sizes, for example, 0–100 B, 101–500 B, etc. For each MPI communication operation, `mpisee` collects statistics within each bucket and stores them within the operation data map (see Listing 3). Those MPI operation data include the time spent, the number of calls that occurred, and the total volume within that buffer range, as shown in Listing 3, Line 9. The volume is represented as $\sum_{i=0}^{n} S(m_i)$, where $S(m_i)$ is the size of the send buffer of the ith call of this MPI function. The user can configure the number of buckets and their corresponding ranges when building `mpisee`. The key to the map is generated by combining the enumerated type corresponding to the MPI function and the size of the send buffer. The data values are initialized if the key does not exist in the map.

Vector operations: The first bucket is always chosen for MPI [v,w] operations. The [v,w] notation in MPI refers to vector operations designed to handle non-contiguous and varying data distributions across different processes. For instance, vector operations such as `MPI_Scatterv` and `MPI_Allgatherv` allow sending or receiving variable data per process, accommodating applications with non-uniform data distribution. Similarly, operations like `MPI_Alltoallw` manage scenarios where data types and distributions vary.

Categorizing all MPI [v,w] operations in a single bucket, regardless of buffer size, addresses their inherently variable nature. This approach ensures consistency in profiling by preventing discrepancies that could arise if the same function call is categorized differently when called with varying buffer sizes by different processes. As these operations are not categorized by buffer range, measuring their total volume is essential to accurately capture the overall generated traffic, thereby providing a clearer insight into network load and potential bottlenecks.

Communicator-less operations: Certain MPI functions, such as `MPI_Wait` and `MPI_Test`, operate on `MPI_Request` objects rather than communicators. This presents a challenge to our implementation since MPI does not enable attribute caching in `MPI_Request` objects, a feature we have advocated for in our previous work [23]. To address this limitation, we associate the `MPI_Request` with the communicator before they are called. We establish this association in the function wrappers of the non-blocking MPI communication functions such as `MPI_Isend`, `MPI_Irecv`, etc. We implement this by using an `std::unordered_map` data structure from the C++ Standard Library, which maps the `MPI_Request` (key) to the `MPI_Comm` (value). Subsequently, when a function such as `MPI_Wait` is called, the wrapper uses the request as a key to find the communicator and call the profiling function, which will update the number of calls, the time spent in those calls, but will not record additional volume as it is already recorded in the respective non-blocking call (e.g., `MPI_Isend`, `MPI_Irecv`, etc.) that created the request. MPI operations that use arrays of requests (e.g., `MPI_Waitall` or `MPI_Testall`) report a single completion time regardless of the communicators involved with each request. This provides no information about individual request completion times, making it impossible to attribute the total measured time to the respective communicators in a precise manner. To work around this limitation, we consider two scenarios: If all requests belong to the same communicator, `mpisee` assigns this completion time to that communicator. Otherwise, `mpisee` assigns the completion time to a special "dummy" communicator marked with an asterisk, namely `*0.0`. The total time accumulated by this dummy communicator is then reported separately in the final profiling summary. It is important to note that the applications benchmarked in this study do not use mixed-communicator request arrays in these MPI calls.

One-sided operations: These pose a similar challenge as the `MPI_Wait` and `MPI_Test` for our implementation, as they operate on `MPI_Win` instead of communicators. We associate the `MPI_Win` with the `MPI_Comm` using the window caching mechanism, similar to the communicator creation. More specifically, we wrap the window creation functions, such as `MPI_Win_create`, and call `MPI_Win_set_attr` to assign the communicator object as an attribute to the newly created window. Therefore, when one-sided communication operations such as `MPI_Put` or synchronization such as `MPI_Win_fence` are called, we retrieve the communicator from the window using `MPI_Win_get_attr` and call the profiling function.

3.5 | Communicator Deallocation

When communicators are deallocated, `mpisee` ensures that the associated profiling data is preserved. `MPI_Comm_free` marks the communicator object for deallocation. This poses a problem for `mpisee` as it stores its profiling data as attributes to the communicator object. To address this, before deallocating the communicator, we create a pair consisting of its profiling data and its associated `MPI_Group`. We store these pairs in a global freelist, which we implement using `std::vector` for the same reasons as the `comms_table`. Finally, we use these pairs of `MPI_Group` and profiling data to re-create the communicators with their corresponding data during `MPI_Finalize`. In this way, we do not perform a collective operation to name the communicator within `MPI_Comm_free`. Therefore, we preserve its semantics as a local operation as defined in the MPI standard [2, Section 7.4.3].

3.6 | Finalization

After capturing detailed profiling data during the application's execution, `mpisee` must properly finalize this data and store it on disk so that it is accessible for post-execution analysis. The

function wrapper of `MPI_Finalize` performs the following operations:

1. *Re-create the deallocated Communicators*: Ensures that the communicators marked for deallocation by `MPI_Comm_free` are available for the subsequent naming process.
2. *Create communicator names*: Assigns globally consistent and unique names to all communicators across processes.
3. *Gather profiling data*: From all processes to process with rank 0 in `MPI_COMM_WORLD`.
4. *SQLite database file creation*: Write profile data into a file for further analysis.

The profiler re-creates communicators deallocated by `MPI_Comm_free` using their corresponding `MPI_Group`. Each process will call `MPI_Comm_create_group` using the `MPI_Group` from the freelist (see communicator deallocation) to create the communicator. It then stores the profiling data to the corresponding communicator and a reference to the newly created communicator in the `comms_table`, as shown in Listing 4. This simplifies the communicator naming process, which requires traversing the `comms_table` as we explain in the next step.

Listing 5 illustrates the communicator naming process during `MPI_Finalize`. This loop traverses the `comms_table`, which contains references to all created communicators. The process ranked 0 in each communicator broadcasts the two integers necessary for creating the name to all other communicator processes. Each process uses this information to generate a consistent communicator name by calling `write_comm_name`. Finally, the root process in `MPI_COMM_WORLD` gathers all profiling data from the other processes and creates the SQLite database file on disk.

`mpisee` stores the profiling data in an SQLite database with tables for metadata, mapping information, and detailed profiling metrics. The metadata table contains the application's input arguments, the profiling date, the MPI library and `mpisee` version,

LISTING 5 | Communicator name creation during MPI_Finalize. All processes in each communicator agree on a common name.

```
for(i = 0; i < comms_table.size(); ++i) {
    // Retrieve the communicator's profiling
    data
    PMPI_Comm_get_attr(comms_table[i],
     namekey(),
                       &com_info, &flag);
    buf[0] = rank; // Global rank in
    MPI_COMM_WORLD
    buf[1] = com_info->comms; // Local
    communicator counter
    /*
     * Process that is ranked 0 in each
    communicator
     * broadcasts its two identifiers in the
    communicator.
     * Therefore, all processes in the
    communicator are synchronized.
     */
    PMPI_Bcast(buf, 2, MPI_INT, 0, comms_table[i
    ]); // Broadcast the two-tuple key in the
    communicator
    write_comm_name(&comm_info, buf[0], buf[1]);
}
```

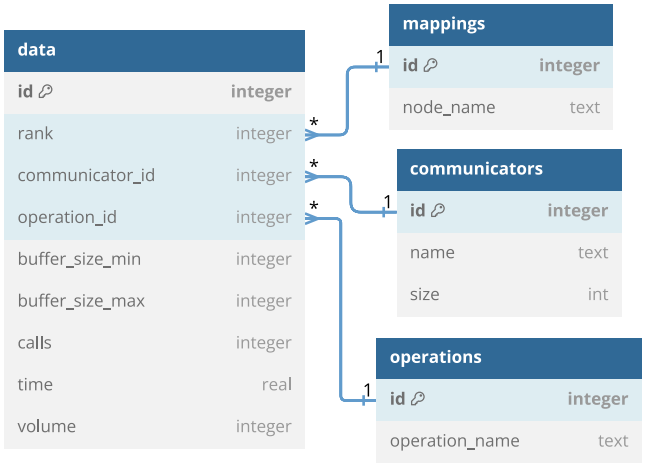

FIGURE 5 | The relation database schema of `mpisee` depicting the main tables of the SQLite database.

and the build date. We show the relational database schema in Figure 5. The SQLite database maintains four tables:

- *Communicators*: Stores the name and size of communicators, providing a central reference for profiling data.
- *Mappings*: Stores the MPI ranks as keys and the corresponding compute node names they are mapped into.
- *Operations*: Stores the names of the MPI operations that were executed by the application.
- *Data*: The core table storing individual profiling records. It links communication metrics to communicators, operations, and mappings.

To reduce redundancy, MPI operation names are stored in a separate table, each assigned a unique ID. The comms table reduces redundancy in the main data table by storing communicator names and sizes once, allowing data records to reference them by ID. These tables are created after process 0 has gathered all profiling data from other processes.

We use a single transaction for multiple `INSERT` statements to insert data into each table efficiently. Thus, instead of inserting rows individually, we batch these operations, simultaneously writing larger chunks of data to the database. This approach significantly reduces disk writes and journaling operations performed by the SQLite library, which are essential for maintaining data integrity and consistency during transactions.

3.7 | The Data Analysis Tool: `mpisee-through`

To analyze the profiling data captured by `mpisee` in the SQLite database, we developed a portable data analysis tool called `mpisee-through`. It uses the SQLite3 Python API and can perform several queries on the output SQLite database to provide multiple perspectives of the application's profile. `mpisee-through` provides pre-built queries that users can customize through simple command-line arguments. With `mpisee-through`, the users can do the following: Filter and sort results based on MPI operations, communicators, buffer

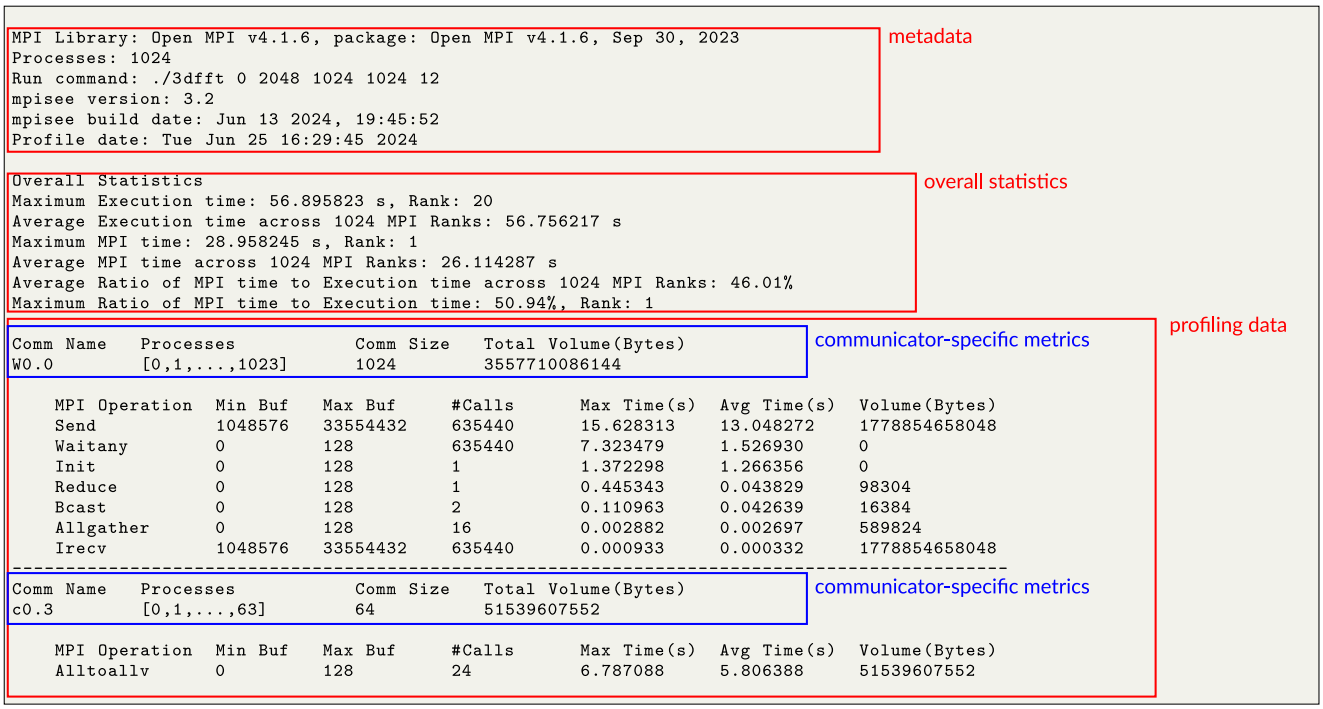

FIGURE 6 | A subset of the default view that `mpisee-through` produces by analyzing an `mpisee` profile.

sizes, time ranges, or MPI rank; choose between terminal output or CSV export for further processing; and run analyses locally or remotely, requiring only a Python environment.

Figure 6 shows the default view of `mpisee-through` of an `mpisee` profile that contains three main sections: The first section is the metadata, which includes information on the MPI library, arguments used to run the application, `mpisee` version and build, and profile date. The second section is the overall statistics, providing an overview of the following: The (net) execution time, which is the total time spent executing instructions after initializing `mpisee` (excluding the time in `MPI_Finalize`). The MPI time, which is the total time spent purely in MPI functions (excluding the time `MPI_Finalize`), and their corresponding ratio. The third is the profiling data section, which displays data per communicator for each combination of MPI operation and buffer size range. For each communicator, `mpisee-through` displays two types of aggregated data: Communicator-specific metrics and MPI operation statistics.

- *Communicator-Specific Metrics*: (1) Name and Size: The communicator's name and the number of participating processes. (2) Global Ranks of Processes: The global MPI ranks of processes that participate in the communicator. (3) Total Volume: The summary of the size of all messages sent by all operations performed by all processes in this communicator.
- *MPI Operation Statistics*: (1) Operation Name: Identify the specific MPI call. (2) Buffer Size Range: Indicates the range of message sizes for which these statistics apply. (3) Number of Calls: The number of calls of this MPI operation within the specified range of buffer size. (4) Maximum and Average Time: The maximum and average time of calls of this MPI operation within the specified buffer size range among all processes within this communicator. (5) Total Volume Sent: The summary of message sizes sent by all processes calling this MPI operation within the specified buffer size range and this communicator (see Section 3.4).

The default view of `mpisee-through` aggregates data among all processes within each communicator. The number of calls for point-to-point operations (such as `MPI_Send`) is the summary of calls among all processes within the specified buffer range. This summary is divided by the number of processes participating in collective operations. Therefore, Figure 6 shows that there is 1 call to `MPI_Init` and 16 calls to `MPI_Allgather` with 1024 processes. The volume summarizes the sizes of all messages sent by all processes when calling this MPI operation within the specified buffer range: $\sum_{j=0}^{p}\sum_{i=0}^{n} S(m_i^j)$, where $S(m_i^j)$ is the size of the send buffer of the ith call of jth process. `mpisee-through` offers more options for data analysis, such as showing the data of user-specific MPI ranks instead of aggregating. It can also filter point-to-point or collective MPI operations and specific buffer ranges.

Per-communicator and per-bucket analysis allow users to see how different buffer sizes impact the performance of their MPI application within specific communicators. This insight helps users identify trends, patterns, and potential performance bottlenecks related to buffer sizes, which is valuable for code optimization.

4 | Evaluation

After detailing the design and implementation of `mpisee`, we now evaluate its performance and demonstrate the value of the insights it provides. The evaluation consists of three parts: First, we analyze the overhead caused by `mpisee` on a set of

MPI applications and compare it to other profiling tools. Second, we demonstrate the advantages of communicator-centric over a communicator-oblivious profiling approach by using the sample MPI application presented in Listing 1. Third, we demonstrate the usefulness of communicator-centric information by profiling an FFT MPI application, identifying potential performance bottlenecks and then using this information to improve its overall performance by selecting different algorithms for `MPI_Alltoallv`.

We conducted our experiments on the Discoverer Supercomputer,[2], which is an HPC system with a total of 1128 compute nodes. Each compute node in our partition is equipped with two AMD EPYC 7H12 64-Core processors and 256 GB of DDR4 RAM. The nodes are connected through an NVIDIA ConnectX-6 InfiniBand with up to 200 Gbit/s throughput. The machines run Red Hat Enterprise Linux 8 OS, and we used the Open MPI 4.1.6 library.

4.1 | Overhead Analysis

To assess `mpisee` 's impact on application performance, we conducted an overhead analysis using a set of benchmark MPI applications from the ECP Proxy Applications Suite.[3] Proxy applications are relatively small, simplified codes designed to capture fundamental aspects of larger, real-world applications [24]. We focused on eight applications in total, seven from release 5.0 of the ECP Proxy Applications Suite and one application (SNAP) from release 6.0, which we show in Table 2.

Following Sultana et al. [25], we excluded Ember, miniQMC, XSBench, and MACSio due to minimal MPI usage or I/O focus. XSBench performs only a barrier and a reduction at the end to aggregate results, and miniQMC performs only `MPI_Reduce`. Ember is a structural simulation toolkit that is intentionally simplified without any calculations, control flow, etc. We also excluded MACSio as in our previous study since it focuses on I/O operations [26]. This means that these applications would not frequently invoke the profilers' code, making them uninformative for demonstrating the overhead of profiling tools. We also include SNAP from the release 6.0 as it creates Cartesian communicators, making it suitable to assess the overhead associated with `mpisee` 's handling of communicators.

We compare the overhead of `mpisee` to Score-P v8.4 and mpiP v3.5. We chose these two profilers out of three (Score-P, mpiP, and IPM) we discussed in Section 2 for the following reasons: Score-P is one of the most well-maintained and popular profiling tools. mpiP provides callsite information and also information on MPI communicator sizes, which can be compared to `mpisee`. We did not include IPM because it is not actively maintained and therefore is not suitable for our study. We use the MPI profiling functionality of Score-P and instrument the applications with `--nocompiler` and `--noopenmp`.

To accurately measure the overhead introduced by profiling an MPI application, we adopt the method established in our previous study [26]. This method characterizes three types of overhead:

- *Wrap*: Time before entering `MPI_Init` and after exiting `MPI_Finalize`.
- *Hook*: Time spent in `MPI_Init` and `MPI_Finalize` functions.
- *Net*: Time after exiting `MPI_Init` and before entering `MPI_Finalize`.

Net MPI overhead is the most critical as it can distort the application's profile. Hook and wrap MPI overheads are not distorting and, therefore, less critical. The sum of these times equals the application's overall execution time. To measure these overheads, we use the same method as in our previous work [26]. We wrap calls to `MPI_Init` and `MPI_Finalize` with custom functions, `TIME_MPI_Init` and `TIME_MPI_Finalize`. This requires a minor source code modification, replacing the original MPI init and finalize functions. The new functions record the timestamps before entering and after exiting `MPI_Init` and `MPI_Finalize` using the POSIX function `clock_gettime`. We adopt this method because `MPI_Wtime` is unavailable before the MPI environment is initialized. This way, we can measure the hook and net times. To measure the wrap time, we record the time spent in an `srun` or `mpirun` call, which reflects the actual execution time of the application.

Figure 7 shows the three overheads (net, wrap, and hook) introduced by `mpisee`, Score-P, and mpiP for the eight ECP Proxy Applications. We record the times among five runs using 1024 processes distributed across 8 nodes with 128 processes per node.

We notice that all three profilers exhibit negligible wrap overhead. Score-P introduces the highest overhead among all profilers because it writes its output during the wrap phase, after `MPI_Finalize`. The maximum wrap overhead introduced by Score-P is for SNAP (5.3%). In the cases of mpiP and `mpisee`, the maximum wrap times are 0.3% and 0.1%, respectively, also in the SNAP application. Generally, the highest type of overhead is the hook time. Specifically, we see that mpiP has the highest hook time due to writing the output during `MPI_Finalize` in plain text format, which leads to increased write times. The hook overhead is particularly noticeable in SWFFT (32.9%), miniAMR (18.6%), miniVite (36.9%), and SNAP (35.1%). Although `mpisee` also writes the output during this phase, the hook overhead is generally negligible except for Nekbone (4.6%), which creates more than a hundred communicators, increasing the amount of data to be written. Finally, the net overhead across all applications is very low, usually under 1%, with some exceptions like MiniAMR (up to 3.2% with mpiP) and Nekbone (up to 2.7% with Score-P).

TABLE 2 | The ECP Proxy Applications are used for the overhead analysis of `mpisee`, Score-P, and mpiP profilers.

Application name	sw4lite	SWFFT	AMG	miniAMR	ExaMiniMD	Nekbone	miniVite	SNAP
Commit Hash	06b888c	203c595f	3ada8a1	ff07856	3264e29	8b0cdf1	2324e20	e7ab43d

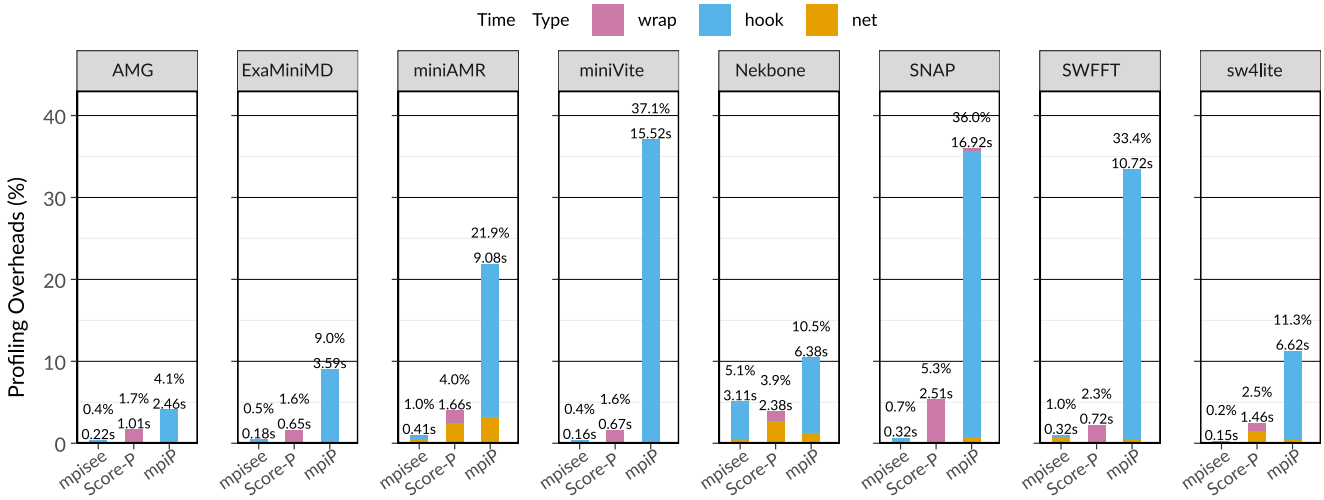

FIGURE 7 | Overheads (wrap/hook/net) when profiling with `mpisee`, Score-P, and mpiP across 8 ECP Proxy Applications. On top of each bar, we show the total overhead in percentage and absolute values. We use averages from five runs, each with 8×128 MPI processes on Discoverer. Color coding indicates overhead type.

TABLE 3 | A comparison of the profile sizes (in kB) of the eight ECP Proxy Applications produced by the `mpisee`, Score-P, and mpiP profilers. The number of communicators additional to `MPI_COMM_WORLD` (if any) is shown in parentheses. Run were performed with 1024 processes on Discoverer.

	ECP proxy application (number of communicators in addition to `MPI_COMM_WORLD`)							
Profiler	**sw4lite (1)**	**SWFFT (5)**	**AMG (1)**	**miniAMR (2046)**	**ExaMiniMD**	**Nekbone (102)**	**miniVite**	**SNAP (66)**
`mpisee`	264	328	876	500	688	34,000	588	448
Score-P	936	1200	1400	852	788	996	1000	1016
mpiP	7500	14,000	4500	13,000	7400	5800	17,000	4400

This analysis shows that `mpisee` introduces negligible overhead for all the three overhead types we defined (wrap, hook, net) compared to two state-of-the-art profilers. More importantly, it introduces minimal net overhead, meaning it does not distort the application's execution, while providing communicator-centric information.

Besides the overhead analysis, we also compare the sizes of the profiles produced by `mpisee`, Score-P, and mpiP. We recognize that the profilers maintain different data, making a direct comparison difficult. However, profile size impacts storage and analysis time, making it essential to estimate.

Table 3 presents the profile sizes produced by `mpisee`, Score-P, and mpiP for the ECP Proxy Applications running with 1024 processes. We show the number of communicators created by each application in parentheses next to its name. Overall, mpiP generates the largest profiles, whereas `mpisee` produces the smallest profiles, except for Nekbone, which is the largest at 34 MB. This is attributed to three reasons: First, we execute Nekbone with 1024 processes. Second, each process creates 102 communicators (in addition to `MPI_COMM_WORLD`) by calling `MPI_Comm_dup` on `MPI_COMM_WORLD`. Third, each communicator is involved in 4 to 5 different MPI functions, thus, 4.5 events on average. We can estimate the size of each event from the data table in Figure 5 to be around 72 B. Therefore, the size of the Nekbone profile from `mpisee`, as shown in Table 3 is approximately: $1024 \text{ processes} \times 103 \text{ communicators/process} \times 4.5 \text{ events/communicator} \times 72 \text{ B/event} = 34,172,928 \text{ B} \approx 34 \text{ MB}$. Nekbone demonstrates a significant profile size, highlighting how applications that frequently create new communicators and perform numerous MPI operations within them can cause the memory and storage requirements of `mpisee` to grow linearly.

In contrast, miniAMR, which creates even more communicators, has a small database because it creates smaller communicators (of sizes 1, 2, 8, and 16) and performs only a single MPI call within them. The same applies to SNAP, which creates 66 communicators of size 32 and performs mainly three operations in them (`MPI_Isend`, `MPI_Recv`, and `MPI_Waitall`), with no MPI communication on `MPI_COMM_WORLD`. Therefore, the approximate size of the output database of SNAP is: $32 \times 66 \times 3 \times 72 \text{ B} = 456,192 \text{ B} \approx 450 \text{ kB}$. In summary, the `mpisee` profile size increases proportionally to the number of communicators, their size, and the number of MPI communication events within each communicator.

Despite Nekbone's large profile size, `mpisee` writes it faster than mpiP, as shown in Figure 7. This demonstrates the efficiency of the SQLite library used by `mpisee` compared to the plain text format used by mpiP. Additionally, `mpisee` maintains detailed communicator data and buffer size information for MPI calls while keeping the profile size relatively small.

```
Comm Name   Processes           Comm Size   Total Volume(Bytes)
s0.1        [0,1,...,511]       512         204800000

    MPI Operation  Min Buf   Max Buf     #Calls      Max Time(s)  Avg Time(s)  Volume(Bytes)
    Allreduce      1024      8192        100         7.300658     7.285067     204800000
-------------------------------------------------------------------------------------------
Comm Name   Processes           Comm Size   Total Volume(Bytes)
s512.1      [512,513,...,1023]  512         204800000

    MPI Operation  Min Buf   Max Buf     #Calls      Max Time(s)  Avg Time(s)  Volume(Bytes)
    Allreduce      1024      8192        100         7.182865     6.852134     204800000
-------------------------------------------------------------------------------------------
Comm Name   Processes           Comm Size   Total Volume(Bytes)
W0.0        [0,1,...,1023]      1024        122880000

    MPI Operation  Min Buf   Max Buf     #Calls      Max Time(s)  Avg Time(s)  Volume(Bytes)
    Allreduce      1024      8192        30          2.040856     1.799128     122880000
    Init           0         128         1           1.440564     1.353389     0
-------------------------------------------------------------------------------------------
```

FIGURE 8 | The communicator-centric profile of the sample MPI application by `mpisee`, offering statistics in different communicators.

These results highlight that `mpisee` incurs a minor net overhead, thereby not distorting the application's performance. `mpisee` 's overhead is on par with other state-of-the-art profiling tools. It maintains a relatively small profile size while providing additional information on MPI communicators, which is unavailable from the other tools.

4.2 | Advantages of Communicator-Centric Profiling

Having established mpisee's minimal overhead, we now turn to its unique advantages in analyzing MPI applications through its communicator-centric approach. Figure 8 shows the `mpisee` profile of the sample MPI application Listing 1, which we previously profiled with Score-P and mpiP in Figure 2. In contrast to the previous profiles, the `mpisee` profile reveals that these calls occur in three distinct communicators. Specifically, the profile by `mpisee` shows that there are 30 calls to `MPI_Allreduce` in `MPI_COMM_WORLD` (`W0.0`) and 100 in each of the other two communicators named `s0.1` and `s1.1`. `W0.0` is of size 1024 and the other two communicators are of size 512 each. In the `mpisee` profile, we see the range of the message sizes and the maximum and average time of each of the `MPI_Allreduce` calls in each communicator. This demonstrates that `mpisee` offers more detailed data as it separates communication per communicator.

More importantly, `mpisee` offers additional insights into the application. The `mpisee` profile shows that the calls to `MPI_Allreduce` in the smaller communicators have an average duration of around 7 s, which is significantly higher than `MPI_Allreduce` calls in `MPI_COMM_WORLD`. This distinction can be crucial for understanding performance bottlenecks and identifying optimization opportunities, especially where multiple communicators exhibit such varied behavior. This information can guide us to focus on optimizing the calls to `MPI_Allreduce` (e.g., via algorithm selection) in the smaller communicators, which can have a much higher impact on the execution time of the application than those in `MPI_COMM_WORLD`. We demonstrate such a use case in the following sections.

4.3 | Analyzing and Tuning an FFT Application With `Mpisee`

We conducted a case study on an FFT application to illustrate these advantages further. The following section demonstrates how `mpisee` provided crucial insights for optimizing the application's performance.

More specifically, the FFT application utilizes the fftMPI library[4] developed by Sandia National Laboratories. The fftMPI library computes 3D and 2D FFTs in parallel as sets of 1D FFTs in each dimension of the FFT grid, interleaved with MPI communication to move data between processors. The FFT application uses the fftMPI library to compute 3D FFTs of size $2048 \times 1024 \times 1024$, which we run on the Discoverer Supercomputer using 8 compute nodes and 128 processes per node for a total of 1024 processes.

Figure 9 shows the maximum time spent by all processes on the MPI operations in the top 20 communicators (out of the 100 the application creates) in terms of time spent. The x-axis shows the communicators with their corresponding sizes in parentheses, while different colors depict different MPI operations.

We observe that the application spends most of its time performing `MPI_Send` in the `MPI_COMM_WORLD` communicator (`W0.0(1024)`), along with other operations such as `MPI_Allreduce`, `MPI_Waitany`, and `MPI_Reduce`. Also, it spends significant time performing `MPI_Alltoallv` in many sub-communicators of size 16. We do not categorize the `MPI_Alltoallv` by buffer size, hence the buffer size range 0 to 128. However, `mpisee` records the total volume of the send buffers for all MPI operations, which can help us estimate the buffer sizes.

To estimate the average buffer size of these `MPI_Alltoallv` calls, we focus on communicators of size 16 and 64. Figure 10 presents detailed statistics by `mpisee` for communicators `c36.4`, `c9.4`, `c128.3`, and `192.3`, which are of size 16 and 64, respectively. For the communicators of size 16, the *Volume* column shows that all 16 processes in each of these two communicators send 1.2 GB of data through 24 calls to `MPI_Alltoallv`. This results in an average buffer size of

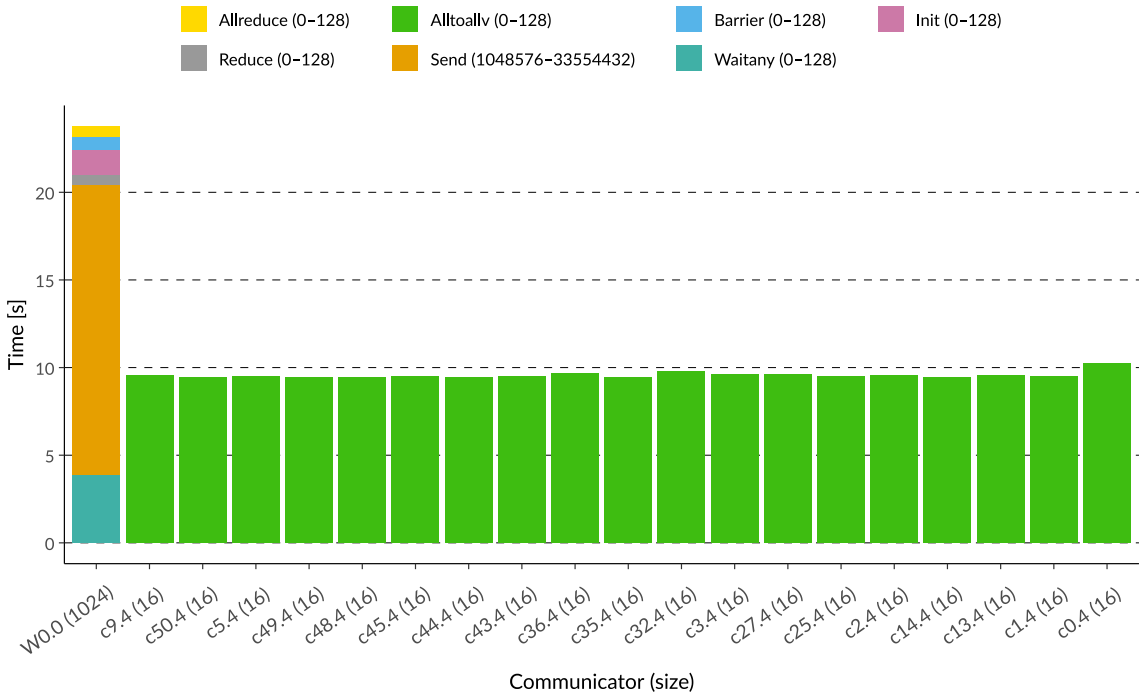

FIGURE 9 | An `mpisee` profile of the FFT application depicting the maximum time spent on MPI calls of in 20 communicators; 8 × 128 processes on Discoverer Supercomputer using Open MPI 4.1.6.

```
Comm Name   Processes            Comm Size   Total Volume(Bytes)
c9.4        [9,73,...,969]       16          12884901888

   MPI Operation  Min Buf   Max Buf   #Calls   Max Time(s)   Avg Time(s)   Volume(Bytes)
   Alltoallv      0         128       24       9.553356      7.458819      12884901888
-------------------------------------------------------------------------------------------
Comm Name   Processes            Comm Size   Total Volume(Bytes)
c36.4       [36,100,...,996]     16          12884901888

   MPI Operation  Min Buf   Max Buf   #Calls   Max Time(s)   Avg Time(s)   Volume(Bytes)
   Alltoallv      0         128       24       9.668800      7.383124      12884901888
-------------------------------------------------------------------------------------------
Comm Name   Processes            Comm Size   Total Volume(Bytes)
c128.3      [128,129,...,191]    64          51539607552

   MPI Operation  Min Buf   Max Buf   #Calls   Max Time(s)   Avg Time(s)   Volume(Bytes)
   Alltoallv      0         128       24       3.776787      2.960950      51539607552
-------------------------------------------------------------------------------------------
Comm Name   Processes            Comm Size   Total Volume(Bytes)
c192.3      [192,193,...,255]    64          51539607552

   MPI Operation  Min Buf   Max Buf   #Calls   Max Time(s)   Avg Time(s)   Volume(Bytes)
   Alltoallv      0         128       24       3.765231      3.002782      51539607552
```

FIGURE 10 | `mpisee` profile statistics of communicators of sizes 16 and 64 of the FFT application as produced by `mpisee-through`. The specific communicators `c9.4`, `c36.4`, `c128.3`, and `c192.3` are shown. We use different colors to highlight data of different communicator sizes.

32 MB per `MPI_Alltoallv` call for each process. Notably, the average buffer size per `MPI_Alltoallv` call for each process in communicators of size 64 is also 32 MB.

The *Max Time* column reveals that the maximum time spent by processes in communicators of size 16 is approximately 9.5 s, whereas the time spent in communicators of size 64 (`c128.3` and `192.3`) is significantly lower, at around 3.7 s. This discrepancy indicates a potential bottleneck, as `MPI_Alltoallv` appears to be slower in the smaller communicators of size 16, despite the same number of calls and identical buffer sizes. The longer times in smaller communicators may stem from inefficiencies in data transfer management, which requires further analysis.

4.3.1 | Tuning by Selecting Different Algorithms for `MPI_Alltoallv`

Based on the findings from our analysis, we implemented targeted optimizations to enhance performance. Our analysis so far has indicated a potential bottleneck in the performance of `MPI_Alltoallv`. We now focus on communicators of size 16 and 64, selecting different algorithms for `MPI_Alltoallv` to improve its performance within these communicators.

By default, Open MPI employs an internal logic to select algorithms for collective operations based on parameters such as the message size and the number of processes. It also allows

TABLE 4 | Execution and MPI time of the FFT application by `mpisee`, combining the pairwise and linear algorithms for `MPI_Alltoallv` for communicators of size 16 and 64; using the average of ten runs, 8 × 128 processes on Discoverer, Open MPI 4.1.6.

Algorithm (com. size 16)	Algorithm (com. size 64)	Notation	Execution time (s)	MPI time (s)
Linear	Pairwise	Linear16+Pairwise64	60.67	31.93
Linear	Linear	Linear16+Linear64	60.01	31.33
Pairwise	Linear	Pairwise16+Linear64	58.89	30.17
Pairwise	Pairwise	Pairwise16+Pairwise64	57.32	28.66

the user to override this logic and set the algorithm for specific collectives at runtime by setting a specific environment variables through the Modular Component Architecture (MCA).[5] Open MPI 4.1.6 offers two algorithms for `MPI_Alltoallv`, *pairwise* and *linear*. For `MPI_Alltoallv`, the decision between these two algorithms is based only on the communicator size.

In this experiment, we tune the FFT application by combining the linear and pairwise algorithms for `MPI_Alltoallv` with the two communicator sizes, 16 and 64. Therefore, there are four possible combinations of these two algorithms and two communicator sizes, which we show in Table 4. The first combination follows the internal selection logic of Open MPI 4.1.6 ; it selects the linear algorithm for communicators of size 16 and the pairwise algorithm for communicators of size 64. For convenience, we use the notation shown in Table 4 to denote each combination.

We conducted ten runs for each algorithm combination and used `mpisee` to obtain the average execution and MPI times. We present these results in Table 4. By comparing the MPI and execution times, we notice that reducing the MPI time directly influences the application's execution time. The lowest execution and MPI times are attributed to the combination Pairwise16+Pairwise64 showing a 6% reduction in execution time and a 11.4% reduction in MPI time compared to the Linear16+Pairwise64, which corresponds to the default logic of Open MPI 4.1.6. We notice that Pairwise16+Linear64 also shows reduced execution and MPI time, whereas Linear16+Linear64 does not show notable differences compared to Linear16+Pairwise64. Therefore, we might infer that, for this application, the pairwise algorithm is beneficial for the performance of `MPI_Alltoallv` when used in communicators of size 16. While the differences in execution times are generally small, they were consistent across multiple runs and not attributable to system fluctuations.

In this experiment, we targeted `MPI_Alltoallv`, which was identified as a potential bottleneck from our prior analysis with `mpisee`. We used four combinations of linear and pairwise algorithms for `MPI_Alltoallv` to improve its performance. The lowest execution and MPI times are attributed to Pairwise16+Pairwise64, with Pairwise16+Linear64 exhibiting similar results but to a lesser degree. From these results, we might infer that the pairwise algorithm benefits communicators of size 16. However, we require a communicator-centric analysis, showing the performance per communicator of `MPI_Alltoallv` to confirm this and demonstrate the exact impact on its performance. A more detailed and communicator-centric analysis is also required to explain the performance of Linear16+Linear64, which appears to have almost no difference compared to Linear16+Pairwise64, although it uses a different algorithm for communicators of size 64.

4.3.2 | Analyzing the Performance of Different `MPI_Alltoallv` Algorithms

In our previous experiment, we improved the performance of the FFT application using different algorithms for `MPI_Alltoallv`. Now, we analyze these results with `mpisee` in a communicator-centric way. Our goal is to pinpoint the effects of the algorithm combinations on the performance of `MPI_Alltoallv` for communicators of size 16 and 64. Finally, we compare our communicator-centric analysis to a communicator-oblivious analysis with mpiP.

Communicator-centric analysis: Figure 11 shows the results of this analysis in two parts. The left part of Figure 11 shows the time of `MPI_Alltoallv` in communicators of size 16, while the right part shows the time `MPI_Alltoallv` in communicators of size 64. The *x*-axis shows the four algorithm combinations, and the *y*-axis represents the time of `MPI_Alltoallv`. For each of the ten runs, we recorded the maximum time of `MPI_Alltoallv` across all processes.

First, we observe that the time spent in communicators of size 16 is double that of communicators of size 64. Therefore, reducing the time of `MPI_Alltoallv` in communicators of size 16 will significantly impact the MPI time.

Focusing on the right part of Figure 11, we observe that the pairwise algorithm combinations outperform those that use the linear for communicators of size 64. Here, Pairwise16+Pairwise64 and Linear16+Pairwise64 have similar performance, with Pairwise16+Pairwise64 being more consistent. On the left part, we notice that Pairwise16+Pairwise64 outperforms all other combinations. Specifically, it reduces the time of `MPI_Alltoallv` by 2.5 s compared to Linear16+Pairwise64. In general, the combinations that use the pairwise algorithm in communicators of 64 perform better than those that use the linear one. However, there are differences between those that use the same algorithm: Linear16+Linear64 performs better than Linear16+Pairwise64, although both use the linear algorithm in communicators of size 16. We observe the same effect for Pairwise16+Pairwise64 and Pairwise16+Linear64.

This effect can be explained by the influence that the algorithm used in one communicator has on the performance of

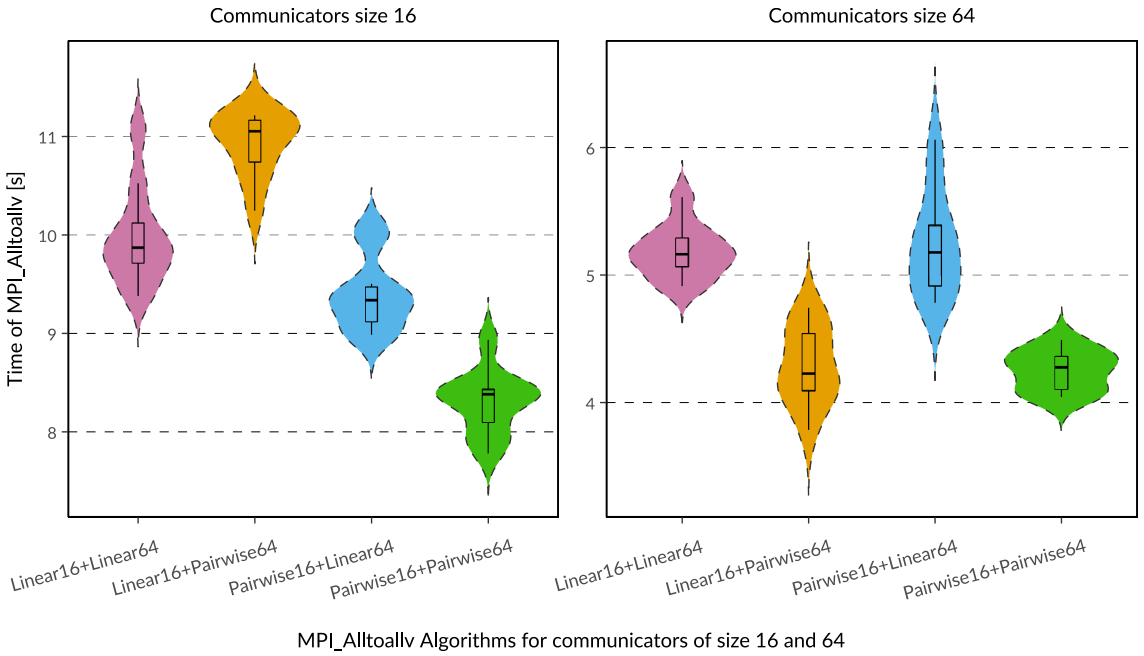

FIGURE 11 | Time spent in `MPI_Alltoallv` in communicators of size 16 (on the left), and size 64 (on the right) of the FFT application using different algorithms of `MPI_Alltoallv` as profiled by `mpisee`. Showing the maximum time of `MPI_Alltoallv` across all processes for each of the ten runs; 8×128 processes on Discoverer, Open MPI 4.1.6.

`MPI_Alltoallv` in other communicators. Specifically, the algorithm applied to `MPI_Alltoallv` in communicators of size 16 can impact the *process arrival patterns* in subsequent calls to `MPI_Alltoallv` in communicators of size 64. The term *process arrival pattern* refers to the timing when different processes arrive at an MPI collective operation [27]. A balanced process arrival pattern, where processes arrive at the collective operation site roughly simultaneously, leads to better performance. In contrast, an imbalanced arrival pattern, where processes arrive at different times, can degrade performance. Thus, the choice of algorithm for communicators of size 16 can create either balanced or imbalanced arrival patterns, thereby influencing the efficiency of subsequent `MPI_Alltoallv` calls in communicators of size 64. A detailed analysis of the arrival patterns is an extensive and complex subject that is beyond the scope of this article.

These results provide us with new insights into these algorithms' performance. We can clearly see that Pairwise16+Pairwise64 outperforms all others for both types of communicators. We can now pinpoint the improvement in MPI time of Pairwise16+Pairwise64 over Linear16+Pairwise64: Our previous results showed that Pairwise16+Pairwise64 improves the MPI time over Linear16+Pairwise64 by 3.2 s. This analysis shows that Pairwise16+Pairwise64 improves the `MPI_Alltoallv` time over Linear16+Pairwise64 in communicators of size 16 by 2.5 s, while they perform similarly on communicators of size 64. Therefore, we can attribute this improvement in MPI to the performance of communicators of size 16. Moreover, as we saw earlier, Linear16+Linear64 and Linear16+Pairwise64 performed similarly. This analysis provides more insights into these results, showing that Linear16+Linear64 performs better in communicators of 16 but worse on communicators of size 64 than Linear16+Pairwise64. Therefore, although they both use the same algorithm in communicators of size 16, using a different algorithm in communicators of size 64 may also influence those of size 16. This shows that for an accurate analysis, it is essential to consider both communicator sizes.

Communicator-oblivious analysis with mpiP : Figure 12 shows a subset of the profile produced by mpiP. The *Aggregate Time* section shows the most time-consuming MPI operations, which are similar to what Figure 9 shows. However, the MPI operations are not associated with any communicators. The *Aggregate Collective Time* section includes only data from the `MPI_COMM_WORLD`, omitting information for other communicators or information on `MPI_Alltoallv`. The *Callsite Time statistics* section is more detailed, showing the number of calls and the maximum, mean, and minimum time for each process. The *App%* and *MPI%* columns show the time ratio for this call to the overall application time for each MPI rank. Here, we focus on the statistics for `MPI_Alltoallv`. Similarly, the *MPI%* shows the ratio of time for this call to the overall MPI time for each rank. The final line aggregates the statistics for `MPI_Alltoallv` across all processes.

Since `MPI_Alltoallv` information is not associated with any communicators, it obscures the performance variations observed across different communicators, which our communicator-centric analysis has shown to be critical for algorithm selection. For example, we cannot determine if the default algorithm performs worse on communicators of size 16 than the linear algorithm, but better on communicators of size 64. Knowing the performance differences at the communicator level is crucial for optimizing MPI applications because different parts of an application might use communicators of varying sizes and characteristics. By understanding which algorithms perform best for specific communicators, we can make decisions about

```
---------------------------------------------------------------------------
@--- Aggregate Time (top twenty, descending, milliseconds) -----------------
---------------------------------------------------------------------------
Call                 Site       Time    App%    MPI%      Count    COV
Send                    8   1.33e+07   22.01   48.68     635440   0.09
Alltoallv              26   1.16e+07   19.15   42.36      49152   0.08
Waitany                 1   1.53e+06    2.53    5.59     635440   0.46
Barrier                 6   4.33e+05    0.71    1.58       1024   0.28
...
---------------------------------------------------------------------------
@--- Aggregate Collective Time (top twenty, descending) --------------------
---------------------------------------------------------------------------
Call                 MPI Time %             Comm Size             Data Size
Reduce                 0.000359        1024 -     2047          64 -     127
Allreduce               0.00015        1024 -     2047           0 -       7
Allreduce              8.97e-05        1024 -     2047           8 -      15
Bcast                  5.98e-05        1024 -     2047           8 -      15
---------------------------------------------------------------------------
@--- Callsite Time statistics (all, milliseconds): 28672 -------------------
---------------------------------------------------------------------------
Name              Site Rank  Count      Max      Mean       Min   App%   MPI%
Alltoallv           26    0     48    1e+03       273      48.3  23.97  47.81
Alltoallv           26    1     48 1.15e+03       268      56.6  23.58  49.84
...
Alltoallv           26    *  49152 1.15e+03       218      39.3  19.15  42.36
```

FIGURE 12 | A subset of the profile of the FFT application by mpiP that was used for our analysis.

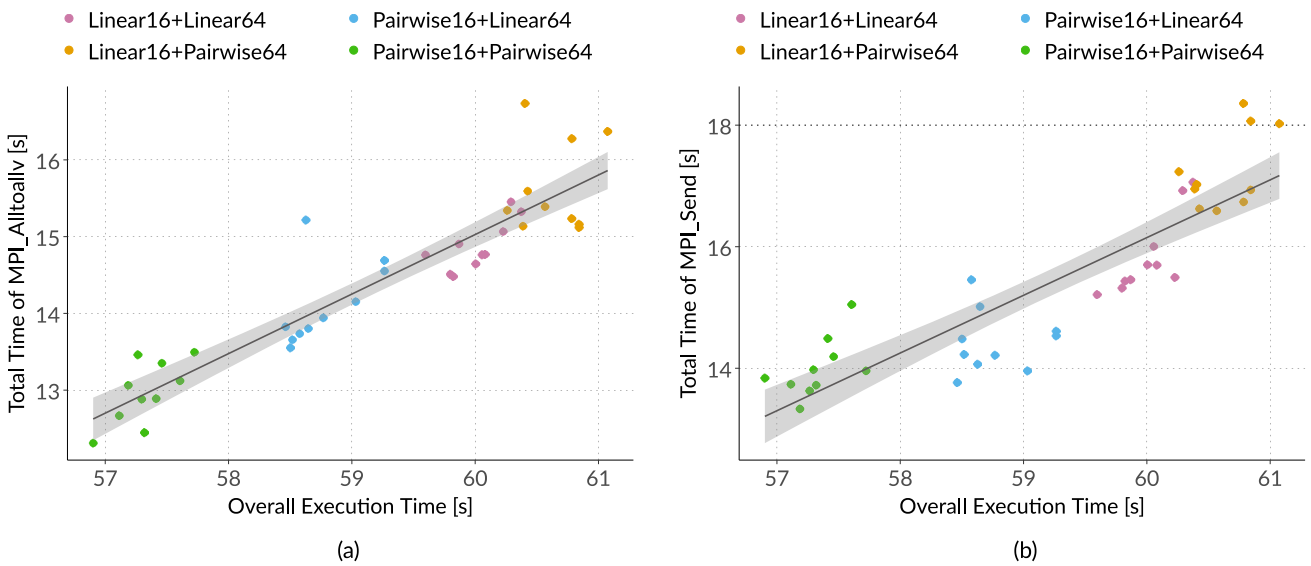

FIGURE 13 | Correlations between `MPI_Alltoallv` in different communicators (left) and `MPI_Send` (right) to the overall execution time of the FFT application. Ten runs for each of the four configurations, each dot represents a different run, and we use different colors to denote different configurations. We use the Interquartile Range (IQR) method for detecting outliers; 8×128 processes on Discoverer, Open MPI 4.1.6. (a) Time spent in `MPI_Alltoallv` in communicators size 16 and 64, (b) Time spent in `MPI_Send` in `MPI_COMM_WORLD`.

which algorithms to use in different parts of their application, guiding targeted optimization efforts. Such a level of detail is not available in communicator-oblivious profiling.

Correlating the time of `MPI_Alltoallv` *to the execution time of FFT*: Based on these observations, we hypothesize that the maximum time spent in `MPI_Alltoallv` in sub-communicators of sizes 16 and 64 directly correlates to the overall execution time. To test this hypothesis, we use the results of our previous experiment (10 runs for each of the four configurations) and calculate the Pearson correlation coefficient between execution time and the maximum time spent in communicators of sizes 16 and 64 performing `MPI_Alltoallv`. We used the Interquartile Range (IQR) method to detect outliers. We compare it to the maximum time spent performing `MPI_Send` on `MPI_COMM_WORLD`, the most time-consuming operation, as seen in Figure 9. To calculate the time, we get the maximum time spent on each communicator across all processes for each run and then calculate the average of all runs. In Figure 13a, the correlation coefficient of 0.92 between the overall execution time and the time of `MPI_Alltoallv` in small communicators. Similarly, there is an apparent correlation between the overall execution time and the time spent in `MPI_Send` in `MPI_COMM_WORLD`, as shown in Figure 13b.

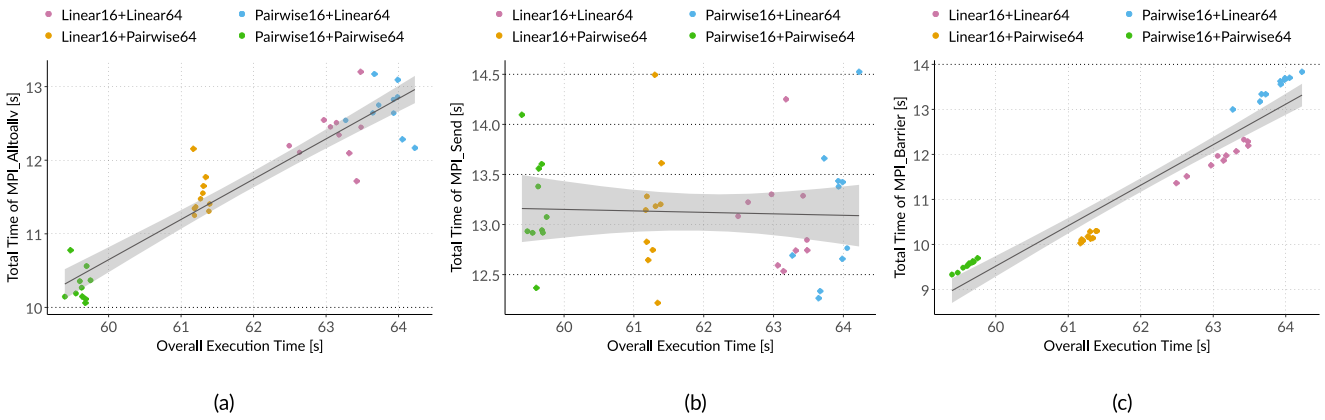

FIGURE 14 | Correlations between `MPI_Alltoallv` (left), `MPI_Send` (middle), and `MPI_Barrier` (right) to the overall execution time of the FFT application when using an `MPI_Barrier` before `MPI_Send`. Ten runs for each of the four configurations, each dot represents a different run, and we use different colors to denote different configurations. We use the Interquartile Range (IQR) method for detecting outliers; 8×128 processes on Discoverer, Open MPI 4.1.6. (a) Time of `MPI_Alltoallv`, (b) Time of `MPI_Send`, and (c) Time of `MPI_Barrier`.

To further investigate the nature of this relationship, we added an `MPI_Barrier` before `MPI_Send` operations, and show the correlation of `MPI_Alltoallv`, `MPI_Send`, and the newly added `MPI_Barrier` in Figure 14. We observe two important differences: First, the overall execution time increases in Figure 14 compared to Figure 13, especially for those that use the Linear algorithm in communicators of size 64. This results in Linear16+Pairwise64 outperforming both Linear16+Linear64 and Pairwise16+Linear64. We also observe the same for the performance of `MPI_Alltoallv`. The Linear algorithm seems to perform better for larger communicators only when processes are skewed. When forced to synchronize via the barrier, its performance advantage disappears due to network contention. Second, the most important observation in Figure 14 is that `MPI_Send` time and execution time are uncorrelated. In Figure 13b, the time of `MPI_Send` —as reported by the profiler—consisted of both the waiting time for the send operation to start and the actual time for sending data. In constrast, in Figure 14, the barrier synchronizes the processes, removing the process skew, and reducing the waiting time within `MPI_Send`. This indicates that the choice of the `MPI_Alltoallv` algorithm does not improve data transfer speeds of `MPI_Send` but instead reduces the waiting time within `MPI_Send`. Additionally, Figure 14a shows that the high correlation between `MPI_Alltoallv` and the execution time persists, confirming that `MPI_Alltoallv` 's performance directly impacts execution time. However, Figure 14b,c indicate that `MPI_Send` 's apparent correlation was a consequence of process skew caused by `MPI_Alltoallv`.

As a closing remark, this analysis of `MPI_Alltoallv` is only possible because `mpisee` is able to report the accumulated times per communicator, which in turn allows us to tune the algorithm selection for specific communicator sizes.

4.4 | Other Use Cases of Mpisee

Beyond the use cases of `mpisee` presented here, `mpisee` has been utilized in other research as well, offering insights on different MPI applications. In our previous work [22] we used `mpisee` to examine the traffic and time spent in different communicators of `SPLATT` and `GROMACS` applications. We also used the profiling information provided by `mpisee` to improve the performance of `MPI_Alltoallv` in different communicators of `SPLATT`. More specifically, we altered the decision logic of Open MPI for `MPI_Alltoallv` for communicators of size 256 resulting in significant performance improvements for the application, especially when running with 1024 processes.

Swartvagher et al. [28], employed `mpisee` to profile the traffic and time spent by different MPI operations in MPI communicators. They implement a mixed-radix decomposition to rename MPI ranks, creating multiple communicators. The authors used microbenchmarks and the `SPLATT` application to assess the performance of their method. Through `mpisee`, they analyzed communicator sizes and time spent within them, confirming that the performance improvements in `SPLATT` were directly attributed to the reduced communication time caused by the improved process mapping.

5 | Conclusions

Existing MPI profiling tools offer valuable insights into application performance, but they do not associate MPI communication operations with their communicators. This can hinder the detailed analysis and optimization efforts of MPI applications that utilize multiple communicators. To address this issue, we developed `mpisee`, a profiling tool that implements a communicator-centric approach, which records the MPI communicators and associates MPI communication operations with their respective communicators. This approach provides the users with deeper insights into the communication patterns of their MPI applications, facilitating targeted optimizations and ultimately improving the performance of MPI applications in HPC environments.

In our detailed analysis of `mpisee` 's overhead, we used eight MPI applications commonly from the ECP Proxy Apps Suite and compared `mpisee` 's overhead to two other well-known profiling

tools. We characterized three different types of overhead. Our results show that mpisee incurs less than 1% net MPI overhead, which is the lowest compared to the other two profilers, ensuring minimal disruption to the application's runtime behavior. The total overhead added by mpisee remains less than 5%. Additionally, thanks to the use of the SQLite library, the size of the profile database of mpisee, except Nekbone, is smaller compared to the other profilers for these applications while maintaining communicator information. The large profile size observed for Nekbone exemplifies a scenario where applications that create communicators repeatedly and perform numerous MPI operations within them can lead to linearly growing memory and storage requirements in mpisee. Future optimizations could include mechanisms to identify and compress repetitive communicator patterns to mitigate such issues.

We used mpisee to analyze and tune an MPI application which uses a real-world parallel FFT library. Our initial analysis revealed hidden performance issues related to MPI_Alltoallv for specific communicator sizes. Therefore, we targeted MPI_Alltoallv for optimization by using different algorithms to improve its performance. Through our communicator-centric analysis, we observed the performance of MPI_Alltoallv for each communicator size. This detailed view allowed us to understand the various algorithms' performance impact, which was not possible with communicator-oblivious profiling. By understanding which algorithms perform best for specific communicator sizes, we can make informed decisions and tune the application, thereby speeding up specific parts.

While mpisee provides detailed per-communicator profiling, there are opportunities for further enhancements. One promising direction is to profile MPI datatypes, analyzing their usage patterns and potential for optimization. Additionally, incorporating the profiling of persistent communication, while challenging due to its complex semantics, could offer further insights into application behavior.

In conclusion, mpisee is a valuable addition to the MPI profiling toolkit, offering a communicator-centric approach and providing information that is not available through other profiling tools. Furthermore, this paper introduces more than mpisee as a tool; it showcases the implementation of a communicator-centric paradigm in MPI profiling. While mpisee implements this idea, this paradigm can also be incorporated into existing MPI profilers, demonstrating its broader applicability and potential to enhance MPI profiling and analysis across different profiling tools in HPC.

Acknowledgments

This research was funded in whole or in part by the Austrian Science Fund (FWF) [10.55776/P31763, 10.55776/P33884]. We acknowledge the EuroHPC Joint Undertaking for awarding us access to Discoverer at SofiaTech, Bulgaria. Open access funding provided by Technische Universitat Wien/KEMÖ.

Data Availability Statement

The data that support the findings of this study are openly available in Datasets produced by mpisee profiles at https://zenodo.org/records/15183596.

Endnotes

[1] https://github.com/variemai/mpisee.

[2] https://discoverer.bg.

[3] https://proxyapps.exascaleproject.org.

[4] https://lammps.github.io/fftmpi/.

[5] https://www.open-mpi.org/faq/?category=tuning.

References

1. I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, and N. Sultana, "A Large-Scale Study of MPI Usage in Open-Source HPC Applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19)* (ACM, 2019).

2. Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 4.1," (2023), https://www.mpi-forum.org/docs/mpi-4.1.

3. v K. Kirchbach, M. Lehr, S. Hunold, C. Schulz, and J. L. Träff, "Efficient Process-To-Node Mapping Algorithms for Stencil Computations," in *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)* (IEEE, 2020), 1–11.

4. C. Niethammer and R. Rabenseifner, "An MPI Interface for Application and Hardware Aware Cartesian Topology Optimization," in *Proceedings of the 26th European MPI Users' Group Meeting, EuroMPI '19* (Association for Computing Machinery, 2019).

5. W. D. Gropp, "Using Node and Socket Information to Implement MPI Cartesian Topologies," *Parallel Computing* 85 (2019): 98–108, https://doi.org/10.1016/j.parco.2019.01.001.

6. S. Pronk, S. Páll, R. Schulz, et al., "GROMACS 4.5: A High-Throughput and Highly Parallel Open Source Molecular Simulation Toolkit," *Bioinformatics* 29 (2013): 845–854, https://doi.org/10.1093/bioinformatics/btt055.

7. H. G. Weller, G. Tabor, H. Jasak, and C. Fureby, "A Tensorial Approach to Computational Continuum Mechanics Using Object-Oriented Techniques," *Computers in Physics* 12 (1998): 620–631, https://doi.org/10.1063/1.168744.

8. N. L. Sandia, "fftMPI: A FFT Library for Distributed FFT Grids," 2018, https://lammps.github.io/fftmpi.

9. J. L. Träff and S. Hunold, "Decomposing MPI Collectives for Exploiting Multi-Lane Communication," in *Proceedings of 2020 IEEE International Conference on Cluster Computing (CLUSTER)* (IEEE, 2020), 270–280.

10. A. Knüpfer, C. Rössel, D. Mey, et al., "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Tools for High Performance Computing* (Springer, 2012).

11. J. Vetter and C. Chambreau, *mpiP: A Light-Weight MPI Profiler* (Lawrence Livermore National Laboratory, 2006).

12. R. D. Hipp, "SQLite Database Engine," 2000, www.sqlite.org.

13. J. Borrill, J. Carter, L. Oliker, D. Skinner, and R. Biswas, "Integrated Performance Monitoring of a Cosmology Application on Leading HEC Platforms," in *Proceedings of the 34th International Conference on Parallel Processing (ICPP)* (IEEE, 2005).

14. M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca Performance Toolset Architecture," *Concurrency and Computation: Practice and Experience* 22, no. 6 (2010): 702–719, https://doi.org/10.1002/cpe.1556.

15. A. Nataraj, A. D. Malony, A. Morris, D. C. Arnold, and B. P. Miller, "A Framework for Scalable, Parallel Performance Monitoring," *Concurrency and Computation: Practice and Experience* 22, no. 6 (2010): 720–735, https://doi.org/10.1002/cpe.1544.

16. A. Knüpfer, H. Brunst, J. Doleschal, et al., "The Vampir Performance Analysis Tool-Set," in *Tools for High Performance Computing*, ed. M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz (Springer, 2008), 139–155.

17. L. Adhianto, S. Banerjee, M. Fagan, et al., "HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs," *Concurrency and Computation: Practice and Experience* 22, no. 6 (2010): 685–701, https://doi.org/10.1002/cpe.1553.

18. M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski, "ScalaTrace: Scalable Compression and Replay of Communication Traces for High-Performance Computing," *Journal of Parallel and Distributed Computing* 69, no. 8 (2009): 696–710, https://doi.org/10.1016/j.jpdc.2008.09.001.

19. C. Wang, P. Balaji, and M. Snir, "Pilgrim: Scalable and (Near) Lossless MPI Tracing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'21)* (Association for Computing Machinery, 2021).

20. W. Gropp and R. Thakur, "Thread-Safety in an MPI Implementation: Requirements and Analysis," *Parallel Computing* 33, no. 9 (2007): 595–604, https://doi.org/10.1016/j.parco.2007.07.002.

21. M. Geimer, M. A. Hermanns, C. Siebert, F. Wolf, and B. J. N. Wylie, "Scaling Performance Tool MPI Communicator Management," in *Proceedings of 18th European MPI Users' Group Meeting, EuroMPI '18* (Springer, 2011).

22. I. Vardas, S. Hunold, J. I. Ajanohoun, and J. L. Träff, "Mpisee: MPI Profiling for Communication and Communicator Structure," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium Workshops* (IEEE, 2022), 520–529.

23. J. L. Träff and I. Vardas, "Library Development With MPI: Attributes, Request Objects, Group Communicator Creation, Local Reductions, and Datatypes," in *Proceedings of the 30th European MPI Users' Group Meeting, EuroMPI '23* (ACM, 2023).

24. O. Aaziz, J. Cook, J. Cook, T. Juedeman, D. Richards, and C. Vaughan, "A Methodology for Characterizing the Correspondence Between Real and Proxy Applications," in *IEEE International Conference on Cluster Computing (CLUSTER)* (IEEE, 2018).

25. N. Sultana, M. Rüfenacht, A. Skjellum, P. Bangalore, I. Laguna, and K. Mohror, "Understanding the Use of Message Passing Interface in Exascale Proxy Applications," *Concurrency and Computation: Practice and Experience* 33, no. 14 (2021): e5901, https://doi.org/10.1002/cpe.5901.

26. S. Hunold, J. I. Ajanohoun, I. Vardas, and J. L. Träff, "An Overhead Analysis of MPI Profiling and Tracing Tools," in *Proceedings of the 2nd Workshop on PERMAVOST* (ACM, 2022).

27. A. Faraj, P. Patarasuk, and X. Yuan, "A Study of Process Arrival Patterns for MPI Collective Operations," *International Journal of Parallel Programming* 36, no. 6 (2008): 543–570, https://doi.org/10.1007/S10766-008-0070-9.

28. P. Swartvagher, S. Hunold, J. L. Träff, and I. Vardas, "Using Mixed-Radix Decomposition to Enumerate Computational Resources of Deeply Hierarchical Architectures," in *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W '23)* (Association for Computing Machinery, 2023).