`mpisee`: MPI Profiling for Communication and Communicator Structure

Ioannis Vardas, Sascha Hunold, Jordy I. Ajanohoun, and Jesper Larsson Träff
TU Wien, Faculty of Informatics, Research Group Parallel Computing, Vienna, Austria
Email: {vardas,hunold,ajanohoun,traff}@par.tuwien.ac.at

***Abstract*—Cumulative performance profiling is a fast and lightweight method for gaining summary information about where and how communication time in parallel MPI applications is spent. MPI provides mechanisms for implementing such profilers that can be transparently used with applications. Existing profilers typically profile on a process basis and record the frequency, total time, and volume of MPI operations per process. This can lead to grossly misleading cumulative information for applications that make use of MPI features for partitioning the processes into different communicators.**

We present a novel MPI profiler, `mpisee`, for *communicator-centric profiling* that separates and records collective and point-to-point communication information *per communicator* in the application. We discuss the implementation of `mpisee` which makes significant use of the MPI attribute mechanism. We evaluate our tool by measuring its overhead and profiling a number of standard applications. Our measurements with thirteen MPI applications show that the overhead of `mpisee` is less than $3\,\%$. Moreover, using `mpisee`, we investigate in detail two particular MPI applications, `SPLATT` and `GROMACS`, to obtain information on the various MPI operations for the different communicators of these applications. Such information is not available by other, state-of-the-art profilers. We use the communicator-centric information to improve the performance of `SPLATT` resulting in a significant runtime decrease when run with 1024 processes.

I. Introduction

The Message Passing Interface (MPI) is the most widely used programming model for HPC applications both in the present and in the foreseeable future [1]. The essential abstraction of MPI for communication is the *communicator*, which defines a safe communication domain for an ordered set of processes. Communicators are used for both collective and point-to-point communication, and they cleanly isolate communication between different communicators. Communicators can furthermore partition the set of MPI processes into smaller subsets, which can be used independently and concurrently. Additionally, communicators can be used to control the placement of MPI processes over processors and compute nodes.

Communicators can carry information on the so-called virtual process communication topology, which abstracts the communication pattern between the processes of the application. Ideally, virtual topology information can be exploited by the MPI library to produce a better performing assignment of the MPI process ranks to the processing elements of the system. Deriving efficient mappings is sometimes seen as a way to improve an application's overall performance, for instance by placing heavily communicating processes "close" to each other in the system [2]–[7]. MPI defines two kinds of virtual topologies, namely Cartesian and Distributed graph (Virtual Graph topologies are considered non-scalable, and will eventually be deprecated from the standard [8]). Virtual topologies are defined by creating the corresponding new communicators which gives the MPI library a handle to attempt a beneficial MPI process reordering (by setting the `reorder` flag in the corresponding calls).

Ultimately, we are interested in the process mapping problem of MPI. We therefore want to be able to investigate the communicator and communication structure of third-party applications to gain information on whether process reordering is attempted, and on the type and amount of communication going on in the different communicators of the applications. State-of-the-art MPI profilers [9], [10] typically profile on a global process level and record the information from this point of view. They do not separate the communication over the different communicators, and thus do not provide the information that we are interested in. Static code analysis will not reveal much about the actual number and structure of the communicators created and nothing about the actual amount and type of communication taking place on the different communicators. For our purposes, it would be highly informative to know the following for a given application: First, how many Cartesian (sub-)communicators were created and whether reordering was attempted. Second, which of these communicators were indeed used for communication, and which MPI calls were performed in these communicators.

Although good process mappings can potentially improve the application performance, MPI libraries rarely take advantage of the `reorder` option to derive better mappings. Additionally, only few of the popular MPI applications use the virtual topology mechanism [11]. This leads to a "chicken and egg" problem as expressed by Gropp [12], where developers do not make use of the MPI virtual topologies and thus, MPI implementations do not see the need to optimize these routines. Nevertheless, if the topology-related routines have been optimized, it is important to be able to evaluate these optimizations within existing MPI applications. Tools for assessing and improving application performance are either profilers or tracing tools. While tracing tools are able to provide detailed information on individual operations, they impose a significant overhead on the application possibly even resulting in traces that do not represent the application's actual behavior. For our purposes, we therefore settle for information that can be provided with low overhead by efficient profiling.

A. Motivation

Current state-of-the-art MPI profiling tools [9], [10] track MPI calls on a per process basis, but typically disregard the communicator on which communication is done. This can be misleading, especially for collective calls that take place on different communicators. Consider an `MPI_Comm_split` call that partitions the processes into a number of (nonempty) sub-communicators. Collective operations on these will be counted either only by one process with some specific, global rank, or by processes in the different sub-communicators, and both will lead to a globally meaningless operation count. The same holds for message volumes and other profiled information. Thus, for applications that use (many) sub-communicators, accurate profiling requires separating the communication operations, volume and times over the communicators.

A *communicator-centric profiler* can help us to accurately answer the following general questions:

- Do real-world MPI applications actually use different communicators, in particular those with a virtual topology?
- What is the communicator structure of the application? the number of communicators, their sizes and, by which MPI call were they created?
- How is the actual communication volume divided among the different communicators?
- Which communication operations take place in each of the different communicators?

B. Contribution

We present `mpisee`, our novel MPI profiler for *communicator-centric profiling*. It is a lightweight tool that introduces less than 3% overhead even for applications with exorbitant MPI time. With `mpisee`, we can explore how MPI applications make use of communicators, how the message volume is divided over these communicators, and which and how many communication operations are used among them. Specifically, we study `GROMACS` which is a widely used molecular dynamics package, and `SPLATT`, a library and application for sparse tensor factorization. Furthermore, we demonstrate the use of `mpisee` by utilizing its *communicator-centric* information to improve the performance of `SPLATT`.

We implemented `mpisee` using the standard MPI profiling interface, PMPI. The current version of `mpisee` intercepts 35 of the most common communication operations, as well as the various MPI communicator creation operations. Our implementation makes heavy use of the MPI attribute mechanism for caching and accessing the per communicator profiling information. The remainder of the paper is organized into the following five sections. Section II specifies the information that is accumulated by `mpisee` for the different MPI calls relative to their communicator. Section III discusses technical details regarding the profiler implementation. In Section IV, we evaluate the overhead of `mpisee` and present use-cases with a set of standard MPI applications. We discuss related work in Section V, before concluding the paper in Section VI.

II. Specification

Our communicator-centric profiler gathers information on the MPI operations performed by each process (as do most other MPI profilers). This information is analyzed and summarized after the MPI application has finished execution by an external tool named `mpisee-through`, which is discussed briefly in Section III-E. In contrast to other profilers, the profiling information is assigned to the communicators that are actually being used in the application. For each distinct communicator created by the application, the profiler cumulates the following information:

1) the communicator size,
2) the communication volume in Bytes per MPI operation,
3) the number of calls per MPI operation, and
4) the time spent per MPI operation.

For send and receive operations, the volume for a call is the actual amount of data either sent or received. Tracking both send and receive volumes per communicator provides a profiler sanity check: The cumulated point-to-point send and receive volumes must be equal. Also, the total number of send and receive operations must be equal.

For collective operations, the actual communication volume depends on the implementations (algorithms) used for the collectives in the given MPI library. We therefore define and record for each collective a lower bound on the communication volume. These communication volume lower bounds for the MPI collectives are defined in Appendix A.

The collective communication volume is recorded in a fully distributed way by each process recording its contribution to the volume (e.g., for `MPI_Bcast` the root process records a volume of 0 and non-root processes a volume of m) and the volume for the communicator is computed in post-processing step (cf. Section III-E).

The current profiler version does not cover one-sided communication, since our interest is in point-to-point and collective communication on the various communicators. Also, persistent communication as per MPI 4.0 is not yet covered.

III. Implementation

We use the standard PMPI profiling interface of MPI [13, Section 14.2], and wrap the MPI calls with the corresponding profiling actions. The application can be linked with `mpisee` by using the LD_PRELOAD environment variable before running, in this way, no recompilation with the `mpisee` library is required. Alternatively, the application can be linked by compiling it with the `mpisee` library.

We now present the communicator naming scheme and the function wrappers implemented by `mpisee` that intercept the MPI calls.

A. Communicator Naming Scheme

The tracking of the communicators is done in a fully distributed way, meaning that each MPI process maintains its own profiling data for each communicator that it is part of. To summarize the profiled communicator data for the processes, it is necessary for communicators to be "named" consistently such that all processes that belong to a communicator have the same name for this communicator. This consistency is guaranteed by a global naming scheme for the communicators.

The name given to a communicator consists of a prefix and a suffix. The prefix is the name of the communicator's parent (the communicator used in the call that created the communicator in question). In this way, we can identify all ancestors of a communicator via the prefix. The suffix consists of a single character denoting the communicator creation operation used to create this communicator, and the maximum number of communicators that the processes of the parent communicator are part of. For communicator creation functions that can create several communicators, such as `MPI_Comm_split` or `MPI_Comm_create`, the suffix additionally includes the smallest rank in the parent communicator of the processes that belong to the newly created communicator. Clearly, all processes in any created communicator will have the same name, and different communicators will have different names.

An example of the naming scheme is shown in Fig. 1. Each circle depicts a different communicator with its name and the processes that are part of it. The example consists of four processes (P0–P3) calling the depicted communicator creation primitives in the following order: `MPI_Cart_create`, `MPI_Comm_split`, and `MPI_Comm_dup` and `MPI_Comm_create` concurrently. The last two calls, `MPI_Comm_dup` and `MPI_Comm_create`, cannot be ordered as they are called by different processes. `MPI_Cart_create` creates the communicator `W_a1` by appending `a1` to its parent's name, `W`. Character '`a`' denotes the `MPI_Cart_create`, and '1' is the maximum number of communicators that P0–P3 are part of. The `MPI_Comm_split` is the second communicator creation call and creates two communicators, `W_s2.0` and `W_s2.2`. The second identifier (after the dot) is '0' for the one communicator and '2' for the other. For `W_s2.0`, this is because the rank of P0 in the parent communicator (`W`) is 0 and the rank of P1 is 1, then the smaller of these two is chosen. Similarly, the name of `W_s2.2` communicator is created. Furthermore, the `W_s2_c3.0` communicator has '3' as first identifier since the processes of its parent communicator have no knowledge of the `MPI_Comm_dup` call. Note that `MPI_Comm_dup` is only called by a subset of processes (P0 and P1). Its second identifier is '0', because process P2 is assigned with rank 0 at the parent communicator `W_s2.2`.

An alternative solution to the consistent-naming problem is given in [14]. Each process maintains a variable to count the number of communicators where this process was rank 0. During communicator creation, the process with rank 0 broadcasts its global rank and its communicator count. Thus, each communicator is identified by these two numbers. However, with this method it is not possible to trace the communicator creation history (ancestor tree) since ancestors are not included in the name.

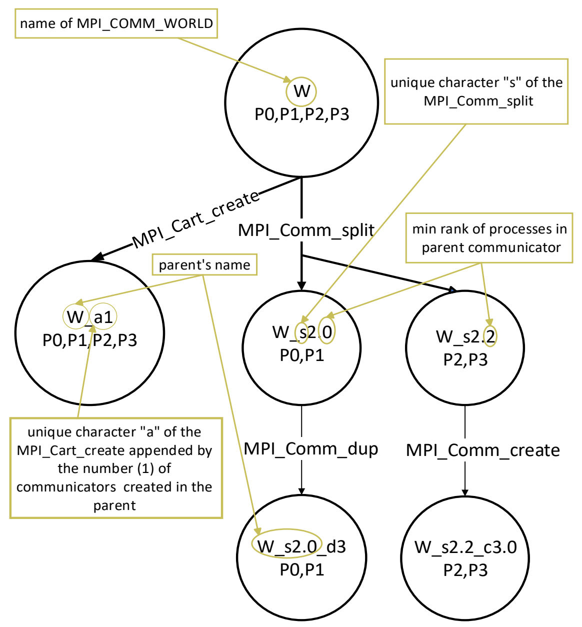

name of MPI_COMM_WORLD
W
P0,P1,P2,P3
unique character "s" of the MPI_Comm_split
MPI_Cart_create
MPI_Comm_split
min rank of processes in parent communicator
parent's name
W_a1
P0,P1,P2,P3
W_s2.0
P0,P1
W_s2.2
P2,P3
unique character "a" of the MPI_Cart_create appended by the number (1) of communicators created in the parent
MPI_Comm_dup
MPI_Comm_create
W_s2.0_d3
P0,P1
W_s2.2_c3.0
P2,P3

Fig. 1: Example of naming scheme in communicator hierarchy.

B. Communicator Creation Functions

The MPI communicator creation functions, such as `MPI_Comm_split`, `MPI_Comm_create` etc., create new, possibly smaller, differently ranked communicators out of existing ones. All these calls, but one, are collective over all processes in the communicator used in the calls, which is important for generating the names we need. The communicator creating functions implement the aforementioned naming scheme (cf. Section III-A).

Here, we present the corresponding wrapper function for `MPI_Comm_split`. For virtual topology functions such as `MPI_Cart_create`, `MPI_Dist_graph_create`, etc., we also record the value of `reorder`, but do not include it in the name. The code of the `MPI_Comm_split` function is shown in Listing 1. The wrapper function does the following:

1) Determine the number of sub-communicators (first suffix identifier): This is done by the `MPI_Allreduce` in Line 9, after which all processes "agree" on the maximum number of communicators in the parent communicator.
2) Find smallest rank in sub-communicator (second suffix identifier): This is achieved with another `MPI_Allreduce` in Line 14.
3) Create names of sub-communicators: For our naming scheme, we first need to look up the parent's name of each sub-communicator (Line 16). The name of the new sub-communicator is created by concatenating the information from the previous two steps. The newly created sub-communicator data structure is stored in the communicator by calling `PMPI_Comm_set_attr`. The `mpisee_namekey()` function creates a new key for the data structure that is used to retrieve the data later on.

Listing 1: `MPI_Comm_split` wrapper function.

```
int MPI_Comm_split(MPI_Comm comm,int color,int key, MPI_Comm *newcomm){
  int ret, ncomms;
  prof_attrs *communicator;
  char *buf;
  int rank, min_rank;
  ret=PMPI_Comm_split(comm,color,key,newcomm);
  // Step 1: Determine the number of sub-communicators
  // lcoms is the number of communicators per process
  PMPI_Allreduce(&lcoms,&ncomms,1,MPI_INT,MPI_MAX,comm);
  if (newcomm == NULL || *newcomm == MPI_COMM_NULL)
    return ret;
  PMPI_Comm_rank(comm, &rank);
  // Step 2: Find smallest rank in sub-communicator
  PMPI_Allreduce(&rank,&min_rank,1,MPI_INT,MPI_MIN,*newcomm);
  // Step 3: Create names of sub-communicators
  communicator = mpisee_get_comm_name(comm);
  buf = (char*)malloc(sizeof(char)* NAMESZ);
  snprintf(buf,NAMESZ,"_s%d.%d",ncomms,min_rank);
  // lcoms gets updated
  mpisee_init_comm(buf,&communicator,comm,newcomm);
  PMPI_Comm_set_attr(*newcomm,mpisee_namekey(),communicator);
  return ret; }
```

We need to be careful with calls to `MPI_Comm_free`, as profiling data is attached to the communicators. When `MPI_Comm_free` is called, we copy out the profiling data structure from the communicator using `PMPI_Comm_get_attr`.

This three-step scheme works for all MPI communicator creation functions except for `MPI_Comm_create_group`. The reason is that `MPI_Comm_create_group` is not collective over all processes in the communicator used in the call. Therefore, `mpisee` at the current state, cannot handle calls to `MPI_Comm_create_group`. Additionally, the current version of `mpisee` does not support profiling of `MPI_Comm_idup`, as it is a non-blocking call and, as we saw, the communicator creation wrappers require a blocking call of `MPI_Allreduce`. However, we plan to handle `MPI_Comm_idup` in a future version of `mpisee`.

C. *Communication Functions*

The communication function wrappers update the profiling data of the communicator at every call. They have the following general structure:

1) Obtain the time stamp at the start of the call.
2) Call the corresponding PMPI function.
3) Obtain the finish time stamp and calculate the duration of the call.
4) Look up the profiling data structure by calling `MPI_Comm_get_attr` and do the following updates:
    a) Increment the call counter,
    b) Update the number of bytes, and
    c) Add the time spent to the total time.

The profiler measures the time by calling `MPI_Wtime` before and after each call and calculating the difference. Our aim is to maintain the communication function wrappers as simple as possible with minimum intrusion to the application's execution.

D. *Wait and Test Functions*

For non-blocking (point-to-point, collective, and persistent) communication, completion is enforced or queried with the numerous wait and test functions of MPI such as `MPI_Wait` and `MPI_Test`.

The wrappers for these functions in `mpisee` present a greater challenge compared to the other MPI functions. This is due to the fact that these functions are not called with a communicator argument, but only take an `MPI_Request` argument, from which it is not possible (with MPI) to find the associated communicator that activated the `MPI_Request` object. Since the communicator is crucial for our profiler, we need a way of looking up the communicator that was used when creating the request. We do this in two steps: When intercepting an non-blocking call, the `MPI_Request` object identifier is used as key to a hash table of communicator values. When intercepting wait or test functions, the `MPI_Request` is again used as a key in the hash table to look up the associated communicator. The implementation of this hash table is taken from Hanson [15].

E. *The Finalize Function*

When the `MPI_Finalize` function is called, each process traverses its table of profiling data structures described in Section III-B to retrieve the profiling data. The root gathers the profiling data from each individual processes. The data of each process for every communicator are written into a CSV file by the root.

The post processing of the CSV file is performed by an additional external tool named `mpisee-through`, which can quickly summarize the profiles collected by `mpisee`. `mpisee-through` provides timing statistics (minimum, maximum, and average) across all processes for each collective in each communicator. We chose to decouple the post-processing of the profile from the profiling process to provide a greater degree of flexibility. In this way, the same profile can be post-processed in various ways resulting in additional views of the application without requiring to profile the application for each additional view.

IV. Evaluation

The evaluation of `mpisee` consists of three parts. First, we evaluate the overhead of the tool. Second, we present and discuss various application profiles produced by `mpisee`. Third, we use `mpisee` to tune the performance of an MPI application. The applications used for evaluation are EigenExa version 2.10 [16], SPLATT (The Surprisingly ParalleL spArse Tensor Toolkit) [17], Snap v1.04 [18], and `GROMACS` v2021.2 [19]. The reason for choosing this set of MPI applications is that they create a large number of communicators. For the purposes of overhead analysis, we used a second set of MPI applications that includes five kernels plus three pseudo-applications from the NAS Parallel Benchmarks v3.4.2 [20]. The runs were performed on two clusters, the *Hydra* cluster at TU Wien and the Vienna Scientific Cluster, *VSC-4* [21]. The *Hydra* cluster consists of 36 dual socket compute nodes, where each socket

TABLE I: Profiler overhead.

SPLATT	LAMMPS	GROMACS	MG	IS	LU	SP
0.8 %	1.9 %	2.2 %	0.1 %	0.3 %	0.5 %	0.5 %
EigenExa	Snap	BT	CG	EP	FT	
1.1 %	0.6 %	0.7 %	0.6 %	0.3 %	0.8 %	

holds a 16-core Intel(R) Xeon(R) Gold 6130F. The nodes are interconnected via Intel Omnipath, with a bandwidth of 100 Gbit/s. The OS is Debian 10 GNU/Linux with kernel v4.19. For compiling the applications and the profiling tool itself, gcc version 11.2 was used with Open MPI 4.1.1 (*VSC-4*) and Open MPI 4.1.2 (*Hydra*). The mpisee was compiled using -O3, -march=native and -lto flags.

A. Overhead Analysis

We expect that, since mpisee is a profiling and not a tracing tool, it should introduce a low overhead to the MPI applications [1]. The overhead of mpisee depends on the MPI time of the applications but also on the implementation of the communicator attribute caching facility that is provided by the MPI library. Thus, the MPI applications better suited for this kind of measurement are those that spend a significant amount of time in MPI. More specifically, the most intrusive MPI functions of the profiling tool are the communicator creation functions since some of them have two additional collective calls, as explained in Section III-B. Hence, we expect that applications that use those MPI primitives should have a higher overhead. In order to measure the overhead, we ran the applications with and without mpisee profiling, where the excess runtime defines the overhead. For each run, every process records its own completion time, and we report the maximum value of the completion times over all processes. We calculate the mean value out of twenty runs after filtering out the outliers.[1] All applications except LAMMPS and BT were run on *Hydra* with 512 processes (16 nodes, 32 processes per node). For LAMMPS and BT, we used 256 processes (8 nodes, 32 processes per node) for maintaining a meaningful computation to communication ratio.

From Table I, we notice that the overhead of mpisee for all applications except GROMACS and LAMMPS is less than 1 %. For GROMACS and LAMMPS, the overhead is 2.2 % and 1.9 %, respectively. This is due to the fact GROMACS creates some hundreds of communicators using MPI_Comm_split and also spends a substantial amount of time in MPI calls. The MPI time for GROMACS with this problem size and number of processes is around 70 %, which is higher than the typical MPI time of applications [1]. For the rest of the applications, the fraction of MPI time varies from 30 % to 40 %. The MPI time of LAMMPS is nearly 85 %, which is even higher than of GROMACS, this can explain why its overhead is higher, although it creates less communicators than GROMACS.

[1]We apply Tukey's rule for outlier detection.

TABLE II: Communicators in SPLATT for different tensors.

Comm. Size	1	4	128	256
enron tensor	0	64	4	2
lbnl tensor	256	0	0	5

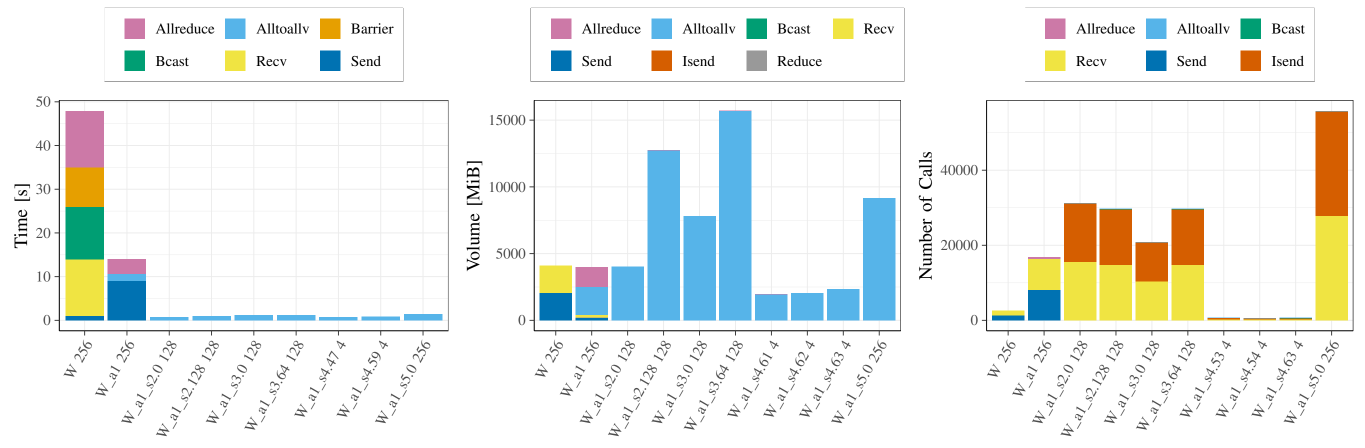
B. Profiling HPC Applications

Now, we present profiles of different MPI applications generated by mpisee in order to demonstrate its use. We chose the profiles of two applications that create large numbers of communicators, SPLATT and GROMACS, both run with 256 processes. SPLATT uses MPI_Cart_create with *reorder* $= 1$, and neither of them uses MPI_Dist_graph_create. For SPLATT, we present three different profiling metrics, the total volume sent by the MPI calls, the total time spent in those calls, and the total number of calls in each communicator created by the application. For GROMACS, we make a different analysis by presenting the total time spent by the MPI calls in each communicator over different runs. We also present the number of created communicators and their corresponding sizes. These profiling metrics of mpisee are collected for each communicator, which is the main difference to state-of-the-art profilers that do not take a communicator-centric approach. In the following graphs, communicator names are on the x-axis followed by their sizes, while the corresponding metric (time, volume, number of calls) is on the y-axis.

1) SPLATT: We use the CPD (Canonical Polyadic Decomposition) routine of SPLATT [22], which attempts to decompose a tensor into a set of rank-one tensors. The CPD routine takes two inputs, a tensor file and a number that corresponds to the CPD rank. We chose two sets of inputs, one with the enron tensor and another with lbnl [23] tensor both with rank-50 CPD.

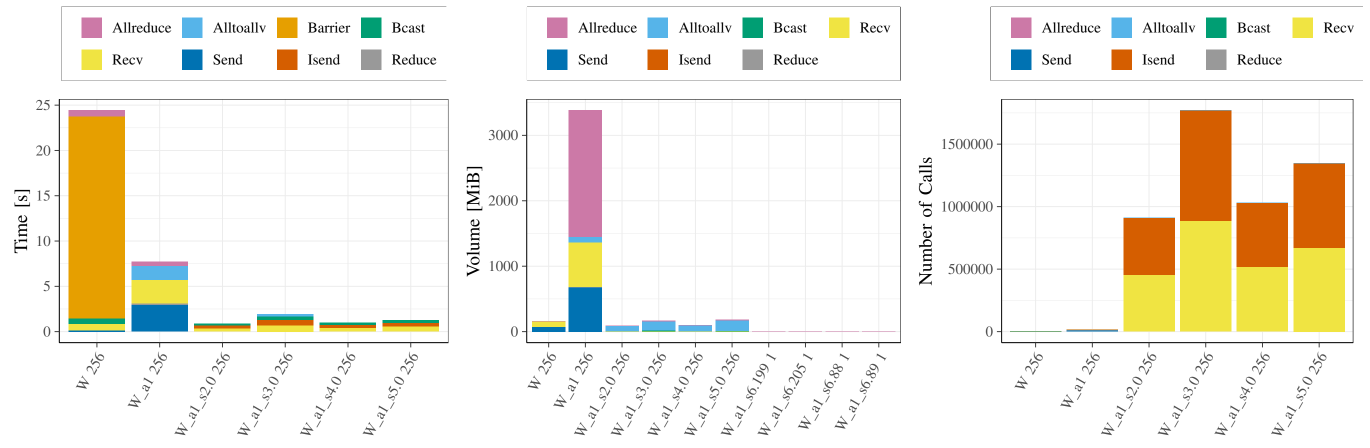
Figures 2a–2c show the application's profile using the enron tensor as input, whereas Figures 3a–3c use the lbnl as input tensor. First, we notice that SPLATT changes its communicator structure depending on the input tensor. The communicators created using the enron and lbnl tensors differ both in size and in number. Table II shows that SPLATT with the lbnl tensor creates 256 communicators of size 1, whereas with the enron tensor, it creates no communicators of size 0. Moreover, 64 communicators of size 4 are created with the enron tensor but none with lbnl.

Figures 2a and 3a show the time spent by MPI calls in different communicators with the two different tensors. The application creates multiple communicators by splitting the initial Cartesian communicator, W_a1, but not all of them are reported in the figures. For example, in Figure 2a, only the communicators with time greater than 0.1 s are shown. Both figures show that most of the time is spent in the MPI_COMM_WORLD communicator, named as W, and the second most-used communicator is the Cartesian communicator, W_a1. Furthermore, we notice that, in Figure 2a, the time spent in the W communicator is almost equally divided among MPI_Allreduce, MPI_Bcast, and MPI_Recv, whereas



(a) running time (b) message volume (c) number of MPI calls

Fig. 2: SPLATT profile, 256 processes, enron dataset (8 × 32 processes; Open MPI 4.1.2; *Hydra*).



(a) running time (b) message volume (c) number of MPI calls

Fig. 3: SPLATT profile, 256 processes, lbnl dataset (8 × 32 processes; Open MPI 4.1.2; *Hydra*).

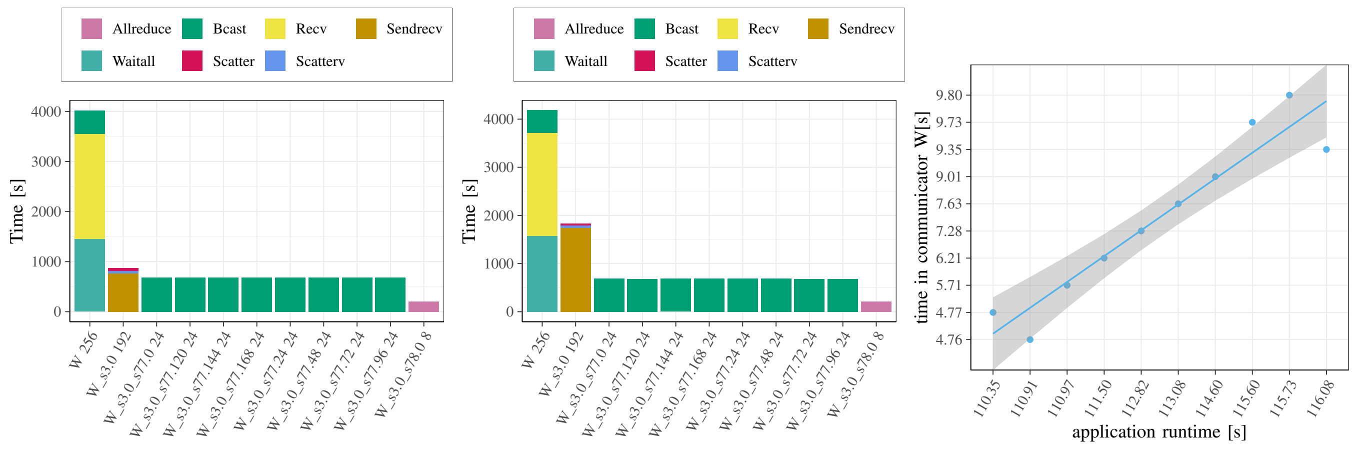
in Figure 3a, the time is spent primarily in MPI_Barrier for the W communicator. Another difference is that in Figure 3a, there are no communicators of size 4 and 128. From Figure 2a, we notice for communicators of size 4 and 128 that most of the time is spent in MPI_Alltoallv. From Figure 2b and Figure 3b, we can see a difference in the traffic pattern and the volume between the two input tensors of SPLATT. In Figure 2b, MPI_Alltoallv causes the most traffic in almost every communicator except W, this differs from what Figure 3b shows. As seen earlier, most of the time is spent in the W communicator for both tensor inputs, however, the volume figures show that the traffic in W communicator is less than other communicators. Furthermore, Figures 2c and 3c show that MPI_Isend and MPI_Recv are the functions with the most calls in almost every communicator except the W and W_a1 communicators where the blocking MPI_Send is used instead.

2) GROMACS: For profiling GROMACS, we used the mdrun function with the benchPep-h benchmark of the free GROMACS benchmarks set [24] with 1000 steps. We conducted ten runs of GROMACS on the *VSC-4*. The profile of the communicator structure shows that GROMACS creates 155 communicators. It uses only the MPI_Comm_split function to create the communicators, and Table III shows the communicator sizes and their count. We notice that the majority of the communicators created are of sizes 3 and 8.

TABLE III: Communicators in GROMACS.

Comm. Size	3	8	24	64	192
Count	64	65	8	1	1

Figs. 4a and 4b show the top 15 communicators where the communication time is greater than 0.1 s for two out of ten runs. For this analysis of GROMACS, showing only the time in communicators is sufficient, and therefore, the other metrics (volume, number of calls) are omitted. From Figures 4a and 4b, we can make the following two observations. First, the W communicator is by far the most used communicator in both



(a) Run A. (b) Run B. (c) Overall runtime against time in W_s3.0.

Fig. 4: `GROMACS` profiles, 256 processes, `benchPep-h` (8×32 processes; Open MPI 4.1.1; *VSC-4*).

runs. Second and more interesting, is that the time spent in communicator `W_s3.0` varies significantly between different runs. We also noticed a difference in the overall runtime of `GROMACS` between the ten different runs. This led us to explore the correlation between the overall runtime of `GROMACS` and the time spent in communicator `W_s3.0` and, present these results in Figure 4c. In Figure 4c, the application runtime is reported on the x-axis and the average time in the corresponding communicator by all processes on y-axis. We can see that the `GROMACS` runtime and the time spent in `W_s3.0` are strongly correlated, specifically the correlation coefficient between them is 0.98. Additionally, we notice that the difference of time spent in communicator `W_s3.0` between different runs is about equal to the difference between the total runtime of the same runs. Moreover, we explored the correlation between the total runtime and `W` communicator, which is the most used communicator. Our findings revealed that `GROMACS` and `W` are not correlated, and specifically, their respective correlation coefficient is -0.18, which shows a light inverse correlation. Interestingly, although processes spend significantly less time in communicator `W_s3.0` than in `W`, the former communicator seems to have more impact on the application's runtime. The analysis demonstrated that `mpisee` can detect which communicator has the most impact on the application's runtime.

C. *Using `mpisee` to improve `SPLATT`*

Now, we show how `mpisee` can indeed help to tune the performance of MPI codes. For this analysis, we again rely on the `SPLATT` application and its CPD routine (cf. Section IV-B1). The CPD routine consists of three steps: (1) preprocessing, (2) solver, and (3) post-processing. Here, we only investigate the solving step, and as `mpisee` supports `MPI_Pcontrol`, we can record a profile of this second step. For comparison reasons, we also record profiles of the second step with mpiP and Score-P. We chose these two tools, as they are widely used and actively maintained, and they entail a very small running-time overhead.

Figure 5 shows the profiling results obtained with mpiP and Score-P, respectively. The mpiP tool accumulates the timing statistics per call site, while Score-P accumulate the data by function per region. Both tools provide very valuable information, as we now know where each of the collective calls are issued and how much overhead they entail overall. We can observe that the CPD method spends a significant fraction of the time in `MPI_Allreduce` and `MPI_Alltoallv`. The question is now whether we can improve the performance of this application by either changing the internal collective algorithms or by using a different process-to-node mapping strategy. To that end, our communicator-centric tool, `mpisee`, can complement the views given by mpiP and Score-P by presenting the timing statistics per communicator. Figure 6 shows the communicator-centric output of `mpisee-through`. In this example, we can observe that 0.8 s and 1.1 s are spent in `MPI_Alltoallv` in the communicators `W_a1_s2.0` and `W_a1_s3.0`, respectively, each of which are of size 128. We first attempted to tune the `SPLATT` application by changing the internally used algorithm for `MPI_Alltoallv`. Since our tool revealed that the most time is spent in communicators of size 256 and 128, we started by adapting the Alltoallv algorithm globally. Open MPI provides two different algorithms for Alltoallv, a *linear* algorithm and a *pairwise* algorithm. We set the Alltoallv algorithm in Open MPI for all communicator sizes to either the linear or the pairwise algorithm and then recorded the profiles using `mpisee`. Figure 7 presents these profiling results. For better readability, we have omitted timing statistics of communicators `W_a1_s4.1`–`W_a1_s4.45`, as they were similar and not significant. The default selection strategy of Open MPI is shown in blue, which select a different Alltoallv algorithm depending on the communicator size. If we select the linear algorithm globally, we can see that the runtime for larger communicators is improved, e.g., the communicator `W_a1_s5.0`, which comprised 256 processes. Selecting the pairwise algorithm did not improve the performance. Thus, we altered the decision logic of Open MPI using a rules file,

```
---------------------------------------------------------------------------
@--- Aggregate Time (top twenty, descending, milliseconds) -----------
---------------------------------------------------------------------------
Call                 Site       Time    App%    MPI%     Count    COV
Alltoallv            1395   3.22e+03    0.18    0.26       184   0.00
Alltoallv             771    3.1e+03    0.17    0.25       184   0.00
Alltoallv             236   3.03e+03    0.16    0.25       184   0.00
Alltoallv             158   2.81e+03    0.15    0.23       184   0.00
Alltoallv             777   2.63e+03    0.14    0.21       184   0.00
Alltoallv            1435   2.44e+03    0.13    0.20       184   0.00
Alltoallv             921   2.41e+03    0.13    0.20       184   0.00
Alltoallv            1483   2.39e+03    0.13    0.19       184   0.00
Alltoallv             338   2.34e+03    0.13    0.19       184   0.00
Alltoallv            1322   2.32e+03    0.13    0.19       184   0.00
Allreduce            1041   2.18e+03    0.12    0.18       180   0.00
..
```

(a) mpiP

```
Estimated aggregate size of event trace:                   13MB
Estimated requirements for largest trace buffer (max_buf): 51kB
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     51,636 200,448 1745.19   100.0        8706.45  ALL
         MPI     51,612 200,192 1144.41    65.6        5716.55  MPI
         COM         24     256  600.78    34.4     2346813.32  COM

         MPI     27,324 105,984  662.58    38.0        6251.66  MPI_Allreduce
         MPI     24,288  94,208  481.83    27.6        5114.55  MPI_Alltoallv
         COM         24     256  600.78    34.4     2346813.32  solver
```

(b) Score-P

Fig. 5: Profiles obtained for SPLATT with 8×32 processes; Open MPI 4.1.2; *Hydra*.

```
COMM                SIZE       PROCS
W_a1_s2.0           128        0-127

        Call                   Mean[s]    Min[s]    Max[s]         Volume     #Calls
        ------------------------------------------------------------------------------
        Alltoallv               0.8016    0.3520    1.1856     3873236800      11776


COMM                SIZE       PROCS
W_a1_s3.0           128        0-63, 128-191

        Call                   Mean[s]    Min[s]    Max[s]         Volume     #Calls
        ------------------------------------------------------------------------------
        Alltoallv               1.1803    0.5428    1.8756     7456121600      11776


COMM                SIZE       PROCS
W_a1_s4.0           4          0, 64, 128, 192

        Call                   Mean[s]    Min[s]    Max[s]         Volume     #Calls
        ------------------------------------------------------------------------------
        Alltoallv               0.1416    0.0020    0.2715        2980800        368


COMM                SIZE       PROCS
W_a1_s5.0           256        0-255

        Call                   Mean[s]    Min[s]    Max[s]         Volume     #Calls
        ------------------------------------------------------------------------------
        Alltoallv               1.0335    0.7398    1.4108     8740110400      23552
```

Fig. 6: A subset of the communicator-centric output produced by `mpisee-through` for SPLATT.

such that only for communicators of size 256, the linear algorithm is used. We call the experiment with this new rules file the "tuned" version, which is shown in green. We can clearly see that the tuned version improves the runtime of the Alltoallv calls in the majority of the communicators. The runtime improvement of the internal Alltoallv calls also translates into an overall improvement of SPLATT, which is shown in Figure 8. In this figure, we compare the total overall runtime of the solver part of SPLATT before and after tuning the internal Alltoallv algorithms. We show the mean runtime of 10 different runs of SPLATT with the CPD routine for 8×32 and for 32×32 processes. In both cases, we can see a significant runtime improvement, which was made possible by being able to inspect the runtime of Alltoallv for each communicator size separately.

Our `mpisee` tool also allows us to extract information about the process-to-node mapping used. In our particular case with SPLATT, we discovered that all processes of every size-4 communicator where scattered across different compute nodes, i.e., requiring internode communication. By exploiting this knowledge, we were able to change the process mapping strategy, such that all size-4 communicators only perform intra-node communication. Although that worked, the overall performance of SPLATT was not improved with this method, as the performance was dominated by the Alltoallv calls in large communicators, which we had just improved.

V. Related Work

Existing performance analysis tools for MPI-based applications can be categorized either as profiling or tracing tools. Profiling tools [9], [10] allow having a summarized view of the application execution by returning aggregated data per event, for example, how many times the event, say `MPI_Send`, occurred, how much time was spent in this event in total, the average or maximum memory buffer used, etc. In contrast, tracing tools [9], [25]–[28] give a more fine-grained view by storing when (timestamp) each event started and ended. This additional information allows for a more detailed analysis of the application. However, the overhead introduced by tracing tools is usually higher than profiling tools, both in time, memory and storage.

Profiling and tracing tools can be further partitioned according to the technique employed to collect the data: instrumentation or sampling. Instrumentation-based tools [9], [10], [26]–[28] as the name suggests, add instructions to the application's code in order to collect information. The instrumentation can be done statically (at the source code level or compilation time), or dynamically (at runtime) with the `LD_PRELOAD` environment variable. Sampling-based tools do not instrument the code but interrupt the execution frequently to retrieve the current execution context. Then, from those sampled execution contexts, they can statistically infer the needed data. The more frequently the application is interrupted, the more accurate the inferred information, but the overhead increases accordingly.

Score-P [9] is a tool that allows all measurement modes. The tool is configurable through a set of environment variables, allowing the user to ask for a specific measurement mode. The instrumentation is done at compilation time using Score-P's compiler, which supports MPI (using PMPI), SHMEM, OpenMP, Pthreads, CUDA, OpenCL, and OpenACC. Unlike Score-P, mpiP [10] only focuses on profiling MPI applications and relies on PMPI underneath. The mpiP tool is the closest to ours in terms of functionality but with an essential distinction: mpiP gives a global view of the communications while our profiler is communicator-centric. HPCToolkit [25] uses a statistical sampling strategy to create program traces,

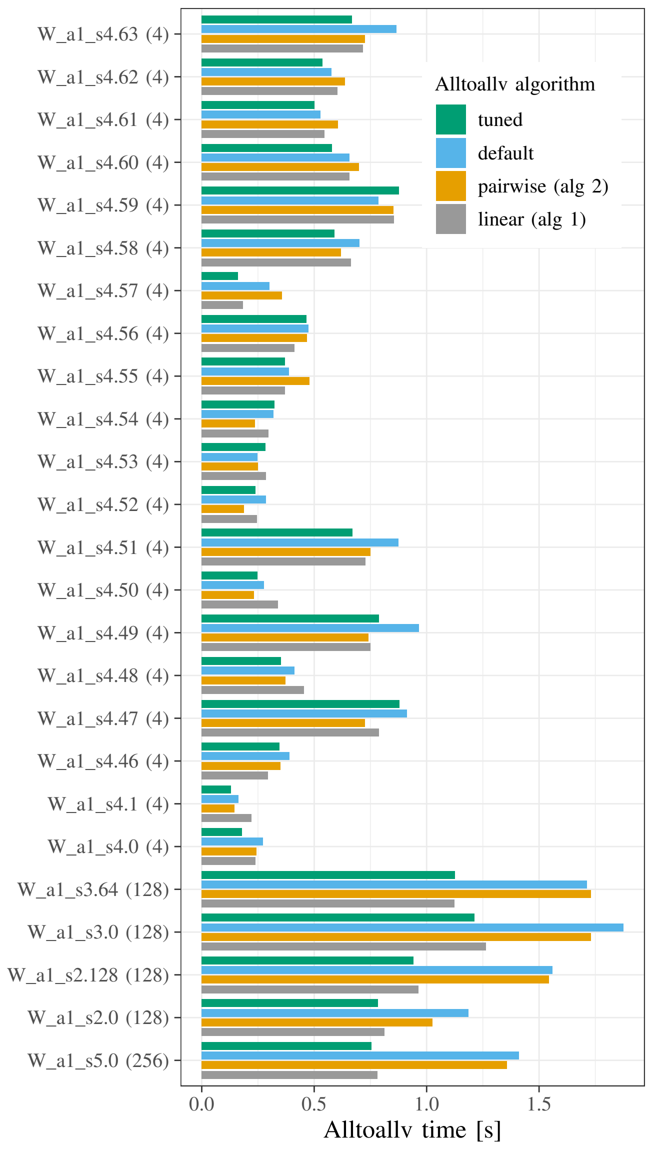


Fig. 7: `MPI_Alltoallv` timing statistics of `SPLATT` recorded with `mpisee`. The size of each communicator is given in the parentheses; Open MPI 4.1.2.

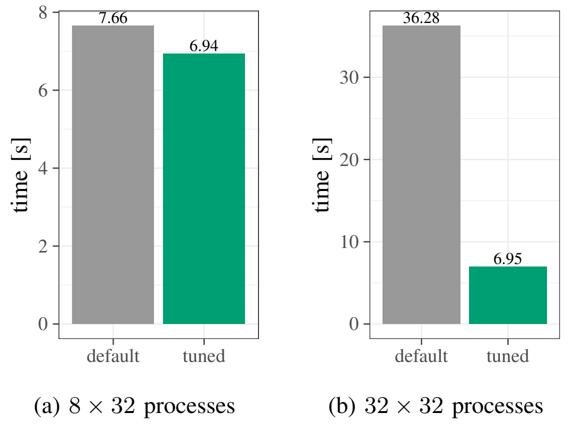


Fig. 8: Runtime improvement for `SPLATT` and different numbers of processes; Open MPI 4.1.2.

where timers and hardware performance counters are recorded periodically. HPCToolkit does not rely on the PMPI interface and can retrieve the current context by unwinding the call stack. Extrae [28] is another feature-rich tracing tool that supports MPI, OpenMP, Pthread, CUDA, OpenCL, and even Java applications. Extrae [28] and Score-P [9] also support the sampling of events. ScalaTrace [26] and Pilgrim [27] are tracing tools for MPI applications. They both rely on the PMPI interface and focus on producing compressed trace files, which should be applicable for trace replay.

VI. Conclusions

We have introduced `mpisee`, an MPI profiler that implements a novel communicator-centric profiling approach. The main component of its implementation is the communicator naming scheme, which achieves consistent communicator naming across all processes. The profiling information includes the time in MPI calls, their volume as well as the number of calls for every communicator. This information is stored in the communicators using the attribute caching facility of MPI. Furthermore, we have evaluated `mpisee` by measuring the overhead and demonstrated its use by presenting the profiles of MPI applications and by tuning the performance of `SPLATT`. Our measurements showed that `mpisee` introduced less than $3\,\%$ of overhead across thirteen MPI applications. Using the profiles produced by `mpisee`, we obtained information regarding the communicators that is not available by the state-of-the-art profilers. With this knowledge, we altered the decision logic of the Alltoallv algorithm in Open MPI for communicators of size 256 in order to tune the performance of `SPLATT`. This resulted in a significant runtime improvement, especially with 1024 processes.

As future work, we plan to support additional MPI primitives, such as `MPI_Comm_idup` and one-sided primitives. Also, we would like to explore the scalability `mpisee` with thousands of processes measuring its performance and profiling more MPI applications. Finally, we will use `mpisee` to benchmark future implementations of MPI virtual topologies.

Appendix
Collective Communication Volume Lower Bounds

Let m, m_i, m_i^j be the sizes of the data blocks to be sent for each process as specified in an MPI collective, where m is total size of a local buffer, m_i the size of a buffer sent by process i (for irregular collectives), and m_i^j the size of a buffer sent by process i to process j. Let p be the number of MPI processes in the communicator, and let n_i be the number of neighbors for process i in a virtual (distributed graph) topology. The total communication volumes are defined as follows:

- `MPI_Bcast`: $(p-1)m$.
- `MPI_Gather` and `MPI_Scatter`: pm.
- `MPI_Gatherv` and `MPI_Scatterv`: $\sum_{i=0}^{p-1} m_i$.
- `MPI_Allgather`: pm.
- `MPI_Allgatherv`: $\sum_{i=0}^{p-1} m_i$.

- `MPI_Alltoall`: ppm.
- `MPI_Alltoall[v,w]`: $\sum_{i=0}^{p-1}\sum_{j=0}^{p-1} m_i^j$.
- `MPI_Reduce` and `MPI_Allreduce`: pm.
- `MPI_Reduce_scatter_block`: pm.
- `MPI_Reduce_scatter`: $\sum_{i=0}^{p-1} m_i$.
- `MPI_Scan` and `MPI_Exscan`: $(p-1)m$.
- `MPI_Neighbor_allgather`: $\sum_{i=0}^{p-1} n_i m$.
- `MPI_Neighbor_alltoall`: $\sum_{i=0}^{p-1} n_i m$.
- `MPI_Neighbor_alltoall[v,w]`: $\sum_{i=0}^{p-1}\sum_{j=0}(n_i-1)m_i^j$.

Acknowledgements

This work was partially supported by the Austrian Science Fund (FWF): project P 31763-N31 and project P 33884-N. We would like to thank Markus Geimer, George Bosilca, and Shaden Smith for their help with Score-P, Open MPI, and `SPLATT`, respectively.

References

[1] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, "Characterization of MPI Usage on a Production Supercomputer," in *Proceedings of the Supercomputing (SC'18)*, vol. 00, 2018, pp. 1–15.

[2] J. L. Träff, "Implementing the MPI process topology mechanism," in *Proceedings of the Supercomputing (SC'02)*. ACM, 2002, pp. 40:1–40:14.

[3] G. Mercier and E. Jeannot, "Improving MPI applications performance on multicore clusters with rank reordering," in *Proceedings of the EuroMPI*, Berlin, Heidelberg, 2011, pp. 39–49.

[4] T. Hoefler and M. Snir, "Generic Topology Mapping Strategies for Large-Scale Parallel Architectures," in *Proceedings of the Supercomputing (SC'11)*, New York, NY, USA, 2011, p. 75–84.

[5] E. Jeannot, G. Mercier, and F. Tessier, "Process Placement in Multicore Clusters:Algorithmic Issues and Practical Techniques," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, pp. 993–1002, 2014.

[6] M. Deveci, K. D. Devine, K. Pedretti, M. A. Pedretti, S. Pedretti, and Ümit V. Çatalyürek, "Geometric Mapping of Tasks to Processors on Parallel Computers with Mesh or Torus Networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, pp. 2018–2032, 2019.

[7] K. von 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)*, 2020, pp. 1–11.

[8] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff, "MPI on millions of cores," *Parallel Processing Letters*, vol. 21, no. 1, pp. 45–60, 2011.

[9] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91.

[10] mpiP: A light-weight MPI profiler. Lawrence Livermore National Laboratory. [Online]. Available: https://github.com/LLNL/mpiP

[11] 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 Supercomputing (SC'19)*, New York, NY, USA, 2019.

[12] W. D. Gropp, "Using node and socket information to implement MPI Cartesian topologies," *Parallel Computing*, vol. 85, pp. 98–108, 2019.

[13] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 3.1*, Jun. 2015. [Online]. Available: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[14] M. Geimer, M. Hermanns, C. Siebert, F. Wolf, and B. J. N. Wylie, "Scaling performance tool MPI communicator management," in *Proceedings of the EuroMPI*, ser. Lecture Notes in Computer Science, vol. 6960, 2011, pp. 178–187.

[15] D. R. Hanson, *C Interfaces and Implementations: Techniques for Creating Reusable Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.

[16] "EigenExa: High Performance Eigen-Solver." [Online]. Available: https://www.r-ccs.riken.jp/labs/lpnctrt/projects/eigenexa/

[17] S. Smith and G. Karypis, "SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit," 2016. [Online]. Available: https://github.com/ShadenSmith/splatt

[18] J. Zerr and R. Baker, "SNAP - SN Application Proxy." [Online]. Available: https://asc.llnl.gov/coral-benchmarks#snap

[19] S. Pronk, S. Páll, R. Schulz, P. Larsson, P. Bjelkmar, R. Apostolov, M. R. Shirts, J. C. Smith, P. M. Kasson, D. van der Spoel, B. Hess, and E. Lindahl, "GROMACS 4.5: a high-throughput and highly parallel open source molecular simulation toolkit," *Bioinformatics*, vol. 29, pp. 845–854, Feb. 2013.

[20] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks Summary and Preliminary Results," in *Proceedings of the Supercomputing (SC'91)*, New York, NY, USA, 1991, p. 158–165.

[21] VSC-4: Vienna Scientific Cluster. VSC Research Center. [Online]. Available: https://vsc.ac.at//systems/vsc-4/

[22] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication," *IEEE International Parallel and Distributed Processing Symposium*, pp. 61–70, 2015.

[23] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. (2017) FROSTT: The formidable repository of open sparse tensors and tools. [Online]. Available: http://frostt.io/

[24] A free GROMACS benchmark set. Max Planck Institute For Biophysical Chemistry. [Online]. Available: https://www.mpibpc.mpg.de/grubmueller/bench

[25] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCTOOLKIT: tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, pp. 685–701, 2010.

[26] 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," *IEEE Trans. Parallel Distrib. Syst.*, vol. 69, pp. 696–710, 2009.

[27] C. Wang, P. Balaji, and M. Snir, "Pilgrim: Scalable and (near) lossless MPI tracing," in *Proceedings of the Supercomtputing (SC'21)*, New York, NY, USA, Nov. 2021, pp. 1–14.

[28] Extrae: A tool that generates paraver trace-files for post-mortem analysis. Barcelona Supercomputing Center. [Online]. Available: https://tools.bsc.es/extrae