Near-Lossless MPI Tracing and Proxy Application Autogeneration

Chen Wang, Yanfei Guo, *Member, IEEE,* Pavan Balaji, *Senior Member, IEEE,* and Marc Snir, *Fellow, IEEE*

Abstract—Traces of MPI communications are used by many performance analysis and visualization tools. Storing exhaustive traces of large-scale MPI applications is infeasible, however, because of their large volume. Aggregated or lossy MPI traces are smaller but provide much less information. In this paper we present Pilgrim, a near-lossless MPI tracing tool that, by using sophisticated compression techniques, generates small trace files at large scales and incurs only moderate overheads. We perform comprehensive studies of various compression techniques used for storing timestamps associated with each call. This timing information is essential for analysis purposes such as skews study. To demonstrate the usefulness of the detailed information stored by Pilgrim, we present a proxy application generator that can generate proxy apps that preserve original communication patterns from the Pilgrim traces.

Index Terms—Communication tracing, MPI tracing, Proxy application generation.

1 Introduction

Traces of Message Passing Interface (MPI) communication calls in a parallel execution are essential to many high-performance computing (HPC) tools. They are used for performance analysis and communication visualization by tools such as Vampir [1] and Scalasca [2], for identifying errors in MPI codes [3], and for guiding source code transformation [4]. Traces are also used to replay application communications, in order to guide the design of future machines [5], [6], [7]. One can also use them to build performance prediction skeletons for estimating the performance of large applications [8], [9].

As systems become larger, trace sizes can get prohibitively large for applications running at large scales. To address this issue, some forms of compression are used by most trace collection tools [2], [10], [11], [12], [13], [14]. The compression schemes used are lossy—even when labeled as lossless. Information is lost at two places. First, numeric performance information, such as call duration, is often aggregated and summarized into a few statistics on the fly by profiling tools such as AutoPerf [15], mpiP [16], and IPM [17]. This does not provide enough information to pinpoint performance issues in applications.

Second, tracing systems typically ignore some MPI functions and some parameters of the traced functions. This improves the compression rate but reduces the usefulness of the traces.

We present in this paper *Pilgrim*,[1] a trace collection and compression tool that we have developed in order to preserve as complete information as possible on executed MPI calls, while achieving good compression. Such detailed information opens possibilities for new postprocessing tasks. We demonstrate one of those, a proxy app generator, which generates proxy apps (more accurately, skeleton apps [18]) automatically from the Pilgrim traces. The proxy apps preserve the same MPI communication patterns as the original apps.

Functions Supported	Cypress	ScalaTrace	Pilgrim
Total: 446	56	125	446
Important Parameters	Cypress	ScalaTrace	Pilgrim
MPI_Status	✓	✓	✓
MPI_Request	×	✓	✓
MPI_Comm	intra	intra and inter	intra and inter
MPI_Datatype	only the size	✓	✓
src/dst/tag	✓	✓	✓
memory pointer	×	×	✓

TABLE 1: Comparison of information collected by different tracing tools. The total MPI (C) function count is based on MPI 4.0 RC [19] (excluding `MPI_Wtime` and `MPI_Wtick`).

Pilgrim advances the state of the art as follows:

1) Pilgrim records all MPI functions and all their parameters. The wrappers and the interception code are generated automatically from the MPI standard to ensure completeness. Table 1 compares the information collected by Pilgrim with two state-of-the-art MPI tracing tools: Cypress [10] and ScalaTrace [11]. They support a small subset of MPI and do not keep enough information to match a nonblocking communication with the wait or test call that completes it (for Cypress), or figure out whether two communication buffers overlap (for both). (The information about Cypress and ScalaTrace was retrieved by manually examining their source code, so small inaccuracies may exist.)
2) All MPI objects (e.g., `MPI_Request` and `MPI_Comm`) and

- *Chen Wang is with Lawrence Livermore National Laboratory. The work was performed when he was with University of Illinois Urbana-Champaign. E-mail: chenw5@illinois.edu.*
- *Marc Snir is with the Department of Computer Science, University of Illinois Urbana-Champaign. E-mail: snir@illinois.edu.*
- *Yanfei Guo is with Argonne National Laboratory. E-mail: yguo@anl.gov.*
- *Pavan Balaji is with Meta Inc. E-mail: pavanbalaji.work@gmail.com.*

Manuscript received Month Day, Year; revised Month Day, Year.

1. Pilgrim is publicly available at https://github.com/pmodels/pilgrim.

memory buffers are encoded in a way that preserves relevant information. Pilgrim intercepts memory allocation operations to match pointers used in MPI calls; it supports both CPU and GPU memories.

3) Pilgrim addresses corner cases that are normally ignored by existing tools, for example, nonblocking communicator creation, intercommunicators, and `MPI_Test*` calls.
4) Pilgrim supports storing the timing information at various levels of detail, whereas most existing tools store only a few statistics that can be computed on the fly.
5) Pilgrim is shown to be scalable and efficient using a variety of codes. *We show that we can achieve smaller trace sizes while preserving more information.*
6) Pilgrim does not require access to the application source code or linking to a special library.
7) We present a proxy app generator that generates, from Pilgrim traces, concise proxy apps that preserve the original communication patterns.

The rest of the paper is organized as follows. Section 2 describes the design and implementation of the tracing process. Optimizations that enable interprocess compression are also discussed. Section 3 describes the proxy app generator. In Section 4 we evaluate Pilgrim and compare its performance with that of ScalaTrace. Related work is discussed in Section 5. We summarize our conclusions in Section 6.

2 Near-Lossless MPI Tracing

Lossless MPI tracing is challenging in that it needs to store a huge amount of information with an acceptable overhead. The overhead consists of two parts: (1) space overhead, which includes the memory footprint during runtime and the storage needed to store the traces after the application run, and (2) time overhead, which is due to the tracing and compression procedure.

Compression can occur at two points: *online compression*, which is performed as traces are collected, and *offline compression*, which occurs after all traces have been collected. Offline compression can be executed in parallel when MPI is finalized, thus reducing I/O. Online compression usually is *intraprocess*, compressing the trace file generated for one process, whereas offline compression usually is *interprocess*, combining the trace files of distinct processes.

The longer an application runs or the more nodes it runs on, the more MPI calls it will make (Figure 10). Fortunately, most codes exhibit *recurring communication patterns*. Good trace compression can be achieved if we recognize and compress as many recurring patterns as possible. We do so by representing the traces using a context-free-grammar (CFG) and a call signature table (CST) as the storage format. Next, we describe the CFG and CST and then show how to use them to represent MPI calls and how to build them incrementally.

2.1 CFG and CST

A formal grammar is defined by a set of production (or term-rewriting) rules that describe all possible strings in a given formal language—namely, all strings of *terminal* symbols that can be obtained by repeatedly applying production rules of the grammar, starting from the initial *nonterminal* symbol S. A *context-free grammar* (CFG) is a formal grammar whose production rules are of the form $A \to \alpha$, where A is a single nonterminal symbol and α is a string of terminal and/or nonterminal symbols. The grammar generates a unique string if there is exactly one rewriting rule for each nonterminal.

```
MPI_Comm_size(comm, &size);
MPI_Comm_rank(comm, &rank);
if (rank == 0) {
  MPI_Comm_set_name(comm, "my-comm");
  for (int i = 0; i < 10; i++)
    MPI_Send(buf, 1, MPI_INT, 1, 999, comm);
}
if (rank == 1) {
  for (int i = 0; i < 10; i++)
    MPI_Recv(buf, 1, MPI_INT, 0, 999, comm);
}
```

CFG of Rank 0

$S \to 1\ 2\ 3\ A$
$A \to 4^{10}$

CFG of Rank 1

$S \to 1\ 2\ A$
$A \to 3^{10}$

CST of Rank 0

Call Signature	Terminal
MPI_Comm_size(comm, 2);	1
MPI_Comm_rank(comm, 0);	2
MPI_Comm_set_name(comm, "my-comm");	3
MPI_Send(buf, 1, MPI_INT, 1, 999, comm);	4

CST of Rank 1

Call Signature	Terminal
MPI_Comm_size(comm, 2);	1
MPI_Comm_rank(comm, 1);	2
MPI_Recv(buf, 1, MPI_INT, 0, 999, comm);	3

Fig. 1: CFG and CST example of a simple code snippet running with two MPI ranks.

A string can be represented by a CFG that uniquely generates that string. If the string has repeating patterns, the grammar can be much shorter than the string. For example, a string a^n, where $n = 2^k$, can be represented by a grammar with $k + 1$ production rules: $S \to A_1A_1$, $A_1 \to A_2A_2$, ..., $A_k \to a$. In the best case a CFG can represent a string of N characters in $O(\log N)$ space. A string is encoded by building a CFG that generates it; it is decoded by repeatedly applying the grammar rules.

Pilgrim builds online for each process a CFG that compresses and stores the sequence of MPI calls made by that process. This sequence is represented by a string of terminal symbols, where each symbol encodes a unique MPI call. To map calls to terminal symbols, Pilgrim maintains a *call signature table* (CST) on each process. A *call signature* consists of a function id and all parameter values of the call. Parameters that are handles to opaque MPI objects are encoded symbolically (we describe the encoding later). Figure 1 shows a simple code snippet and the produced CFG and CST when running with two MPI ranks.

The CFG and CST are constructed independently on each process with no communication overhead, except for calls creating new communicators and similar global objects.

2.2 Optimized Sequitur Algorithm

Both the CFG and CST are built on the fly. Every time Pilgrim encounters an MPI call, it first consults the CST to find the matching terminal symbol or create a new entry if it is its first occurrence. Next, the current grammar is modified in order to handle the terminal symbol occurrence.

We use Sequitur [20] algorithm to build the CFG. It is a good fit for Pilgrim for two reasons: (1) it is an online

algorithm that processes each symbol once, serially, and (2) it has linear time complexity.

The grammar generated by the Sequitur algorithm has two properties:

- P1: No pair of adjacent symbols appears more than once in the grammar.
- P2: Every nonterminal appears more than once on the right-hand side of a production.

The algorithm processes one symbol at a time, replacing the current production rule $S \rightarrow \alpha$ with $S \rightarrow \alpha a$, where a is the current symbol. It then checks whether either P1 or P2 is violated. When property P1 is violated, a new production is formed: A rule $A \rightarrow bcBbc$ will be replaced by $A \rightarrow XBX$ and $X \rightarrow bc$. When property P2 is violated, the useless production is deleted: If we have rules $A \rightarrow Bc$ and $B \rightarrow ab$, and B appears in no other production, then the first rule is replaced by $A \rightarrow abc$, and the second one is deleted.

A more compact grammar is obtained by adding to the notation repetition counts, namely, productions of the form $A \rightarrow B^k$, and replacing each production of the form $A \rightarrow B^i B^j$ by the production $A \rightarrow B^{i+j}$ [21]. This optimization reduces space complexity for regular loops from $O(\log N)$ to $O(1)$. (Strictly speaking, we replace a logarithmic number of productions by counters with a logarithmic number of bits—i.e., we replace recursion with iteration.)

Pilgrim uses this optimized version of the Sequitur algorithm to build each local CFG. For details about the implementation of the Sequitur algorithm we refer readers to [20], [21].

2.3 Implementation Details

Figure 2 depicts the whole tracing and compression process. Intraprocess compression is effected by the first five steps: (1) intercept each MPI call; (2) store the timing information; (3) encode parameters, and compose the call signature; (4) update the CST; (5) use the Sequitur algorithm to grow the CFG. Interprocess compression is performed at the sixth step. We already discussed steps (4) and (5) in the preceding subsections. Here, we describe the remaining steps.

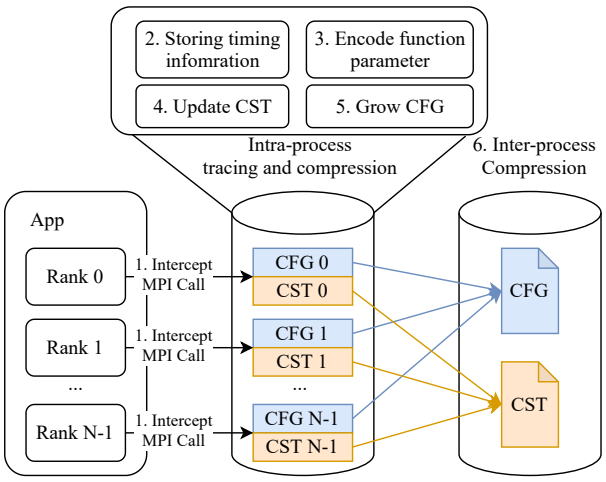


Fig. 2: Tracing and compression process of Pilgrim.

2.3.1 Intercepting MPI Calls

Pilgrim uses the MPI profiling interface (PMPI) to trap and trace MPI calls. The wrappers were automatically generated from the MPI 4.0 RC [19] standard documents (Latex files). We use Latex files instead of MPI header files because they are complete and describe the direction of each function parameter (i.e., input, output, or both). Before compiling Pilgrim, a filtering pass is done automatically to remove the functions that are not supported by the local MPI implementation.

Eventually, the generated wrapper for each MPI function has the following format.

```
prologue();
PMPI_*(); // Call the original function
epilogue();
```

We store the value of each input parameter in the prologue and the value of each output parameter in the epilogue. The prologue and epilogue also record the starting time and duration of the call. Steps (3), (4), and (5) are performed by the epilogue code.

2.3.2 Compressing Timing Information

Pilgrim supports storing the timing information at different levels of detail. The default mode keeps only statistical timing information for each call signature. We keep in the CST the average of the duration of calls with the same signature. This adds negligible overhead and does not increase the number of CST entries. If more details are required for the analysis, Pilgrim can store timestamp information for each call, with either lossless or lossy compression. Timestamps are derived from node-local clocks. These can be tightly synchronized by synchronization techniques that take advantage of MPI [22], [23] or of hardware support [24]. To be specific, Pilgrim keeps the *duration* and *interval* for each MPI call, where duration is the elapsed time of the call and interval is measured from the last call with the same signature. We chose duration and interval instead of the call start and end time because we expected them to compress better.

Lossless timing compression is achieved by using an existing general-purpose lossless compression algorithm Zstandard [25]. We studied several algorithms for lossy compression. These are (1) a CFG-based algorithm first proposed in our previous work [26], (2) a newly designed histogram-based algorithm named HIST, and (3) SZ [27] and ZFP [28], two state-of-the-art floating-points compression algorithms designed for scientific data arrays. All the algorithms except ZFP allow user-tunable relative errors—the larger the error, the higher the compression ratio.

CFG-based algorithm. The use of this algorithm is motivated by the assumption that calls with the same signature have similar durations, and often repeat in a loop at similar intervals. We bin timings using exponential bins: A time d is represented by $\hat{d} = \lceil \log_{(1+\epsilon)} d \rceil$ so that the relative rounding error is no more than ϵ. ϵ can be specified by users on a per-function basis.

And based on the above assumption, the sequences of durations and of intervals (after binning) should exhibit some recurring patterns just like MPI calls. Therefore, we

use the Sequitur algorithm again to build two separate CFGs (one for each sequence) to compress them.

Histogram-based algorithm. Our preliminary experiments showed that the previous assumption is mostly true. However, network noises and other variations across processes reduce the effectiveness of the CFG-based compression algorithm. The top two figures of Figure 3 show the durations of two ranks' `MPI_Ssend` calls in a single-node FLASH [29] run. The durations were binned by using a 10% relative error. The calls were repeatedly invoked in a loop and have an identical call signature across the two ranks. But the durations are highly variable, both within each process and across the two processes.

Another insight is that the durations of identical calls have a bell-shaped distribution, as shown in the bottom two figures of Figure 3. This suggests that we can improve compression using an entropy-encoding method such as Huffman encoding [30]. In Pilgrim, we implemented a simple histogram-based algorithm, named *HIST*. As in the CFG-based algorithm, durations (resp., intervals) are first binned. Then, the HIST algorithm picks the top 2^K frequent bins and uses $K + 1$ bits to encode the durations (resp., intervals) that belong to one of those bins. All others will be stored unchanged. One additional bit is necessary to distinguish these two cases. Since the frequencies of durations and intervals are not known before the program execution, we use the first M (set to 100 in our experiments) observed samples to approximate their distribution.

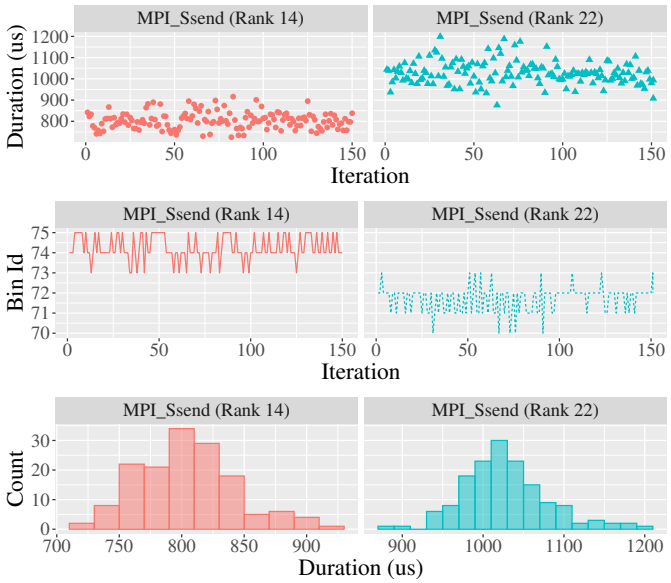


Fig. 3: Top: durations of the first 150 iterations of an identical `MPI_Ssend` call made by two distinct ranks. Middle: the binned duration sequence. Bottom: the histogram of the binned durations.

SZ and ZFP. These two algorithms are designed for compressing scientific data arrays of floating-point values. In our work, durations and intervals are double values stored in 1D memory buffers. Once the buffer is full, we compress the whole buffer using one of the specific algorithms and dump out the compressed data. A small optimization can be applied to both algorithms: instead of buffering durations and intervals in the order of calls, we can cluster them according to the calls' signature before the compression. The motivation comes from the same assumption described earlier—identical calls should have similar durations and intervals that are easier to compress. The modified versions are noted as SZ-Clusterd and ZFP-Clustered, respectively. Table 2 summaries all six lossy timing compression algorithms supported by Pilgrim.

Algorithm	Clustered by call signature
CFG	×
HIST	✓
SZ	×
ZFP	×
SZ-Clustered	✓
ZFP-Clustered	✓

TABLE 2: Summary of the supported timing compression algorithms

2.3.3 *Encoding Function Parameters*

Many function parameters are pointers to the opaque MPI objects or communication buffers. They do not compress well and provide superfluous information. We care whether two MPI calls use the same object, but do not care where that object is stored in memory.

To achieve better compression and to enable useful postprocessings, we use a symbolic representation for such parameters: All occurrences of the same opaque object are identified by the same symbolic id, but the exact memory address is not preserved. For example, an MPI datatype created by `MPI_Type_indexed` and later used in `MPI_Send` will have the same symbolic id in both calls. Since the arguments of the `MPI_Type_indexed` call are also preserved, this allows recreating the layout of the send buffer or properly replaying the call. All other basic type parameters (e.g., numeric values and strings) are stored by values. Next, we describe the symbolic representation algorithm for MPI objects and for memory pointers separately.

MPI Objects. For each process and each MPI object type, Pilgrim maintains a mapping between the object of that type and its symbolic id. Pilgrim also maintains a pool of free ids so that every time a new object is created, we can give it an unused id from the pool. When an object is released (manually by calls such as `MPI_Type_free` or automatically by the MPI library for `MPI_Request` objects), Pilgrim will revoke its id and return it to the pool. In most cases, only a small number of ids are used since the application either reuses the same objects or frees the old objects before allocating more.

If different processes create MPI objects in the same order (as most regular codes tend to do), they will get the same sequence of symbolic ids, which helps the interprocess compression. `MPI_Comm`, however, requires special treatment. A parameter of type `MPI_Comm` is required by all MPI communication calls. Unlike other MPI objects where the symbolic id is only locally unique, we make sure that all processes that belong to the same communicator will get the same id, in order to help compression and help the matching process during postprocessing. Collective communication is used to select such an id. There are two corner cases that require special treatment:

1) If the object created is an intercommunicator, then we need to create a temporary intracommunicator by merging the intercommunicators and use it for collective communication.
2) If the communicator creation function is nonblocking, e.g. `MPI_Comm_idup` then we cannot issue a blocking collective call to decide the id. Instead, we use a nonblocking all-reduce call and keep track of the MPI request generated from it. Later, when we intercept `MPI_Wait*` or `MPI_Test*` calls, we check the completed requests to see whether the symbolic id has been received.

Memory Pointers. Our goal is to have a representation that can tell us whether two buffers overlap. To achieve this, we intercept memory management calls, including `malloc`, `calloc`, `realloc`, and `free`. We also intercept CUDA memory allocation calls such as `cudaMalloc`, `cudaMallocManaged`, and `cudaHostAlloc`. We use an AVL tree to keep track of the currently allocated memory segments. Each node stores a segment's starting address and length, along with the device location if on GPU. All nodes are sorted according to the starting address. Trace records with buffer arguments hold the symbolic id of the containing segment and, optionally, the device location and the displacement from the segment's start. The search for the segment containing an address will take, on average, $O(\log N)$, where N is the current number of nodes in the AVL tree.

This approach handles all memory buffers that are allocated on the heap. For stack variables, since `malloc/free` calls are not invoked, we assign them an id when they are accessed, and the allocated size is assumed to be one byte to be conservative.

2.4 Optimizations

In this subsection we discuss some important optimizations that benefit later interprocess compression.

2.4.1 Relative Ranks

A common pattern of codes using Cartesian meshes is that processes communicate to neighbors in the Cartesian mesh. Consider the pseudocode below, which shows a simple 1D communication pattern.

```
for {
    ...            // computation
    MPI_Recv(src = my_rank - 1);
    MPI_Send(dst = my_rank + 1);
}
```

This code produces different call signatures for the two calls at different ranks because `src` and `dst` are different. A run with N ranks will produce $2N$ call signatures, which is bad for interprocess compression. On the other hand, the displacement from source rank to destination rank has only two values, namely ± 1. Instead of keeping the actual values of `src` and `dst`, we keep the displacement from the caller's rank. This way the code will produce only two unique call signatures regardless of the number of ranks. Note that this simple method also works for regular multidimensional communication patterns (Section 4.1). Moreover, this encoding scheme works for other parameters that may be rank related, e.g., tag in communication calls and color and key in communicator creation calls.

2.4.2 Id of MPI_Request Objects

For most MPI objects, the order of creation, usage, and finalization is deterministic. The `MPI_Request` objects, however, can exhibit randomness because the communication completion order and when the associated request objects are freed are nondeterministic. Consider the following example. At each repeated execution of this `for` loop, request objects may be freed in a different order; the three request objects may be allocated different ids next time the loop is executed. As a result, the code is likely to produce different patterns of call signatures across iterations.

```
for {
    MPI_Irecv(from = my_rank + 1, &req1);
    MPI_Irecv(from = my_rank + 2, &req2);
    MPI_Isend(to = my_rank + 3, &req3);
    while(!(all requests finished)) {
        MPI_Waitany([req1, req2, req3]);
        handle received message;
    }
}
```

While the order in which requests are freed is nondeterministic, the order in which requests are created by communication calls with a given signature, then freed, is usually deterministic. To address this issue we maintain, for each communication call signature (request excluded), a separate pool of symbols for requests created by calls with this signature. With this modification, the three requests in the above example will always get the same ids at every iteration regardless of the request completion order.

2.5 Interprocess Compression

Thus far, we have discussed how Pilgrim encodes and compresses function and function parameters within each process. HPC applications can run on large numbers of processes and will create huge trace files if we keep the CST and CFG separately for each process. Interprocess compression is critical for scalability. While the total number of function calls normally increase linearly with the number of processes, the number of unique call signatures may not grow or grow more slowly if calls have the same signature on different processes, as is the case for many codes. The symbolic representations and the optimizations described earlier increase the similarity across processes.

2.5.1 CST

We merge all CSTs and keep only globally unique call signatures using a parallel merge algorithm with $log_2 P$ phases of pairwise merges, where P is the number of processes. When the merge is completed, the root process broadcasts the merged CST, and every process updates its grammar to use the new symbols assigned to call signatures. Figure 4 shows an example of this process for two MPI ranks.

2.5.2 CFG

Grammars are also expected to have similarities because, in scientific codes, different processes tend to execute the same code blocks (thus the same sequence of MPI calls)

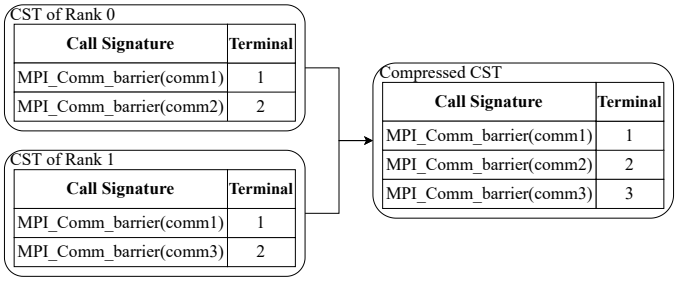


Fig. 4: Example of interprocess compression for CSTs.

but with different data. The symbolic representation we use often allocates an identical sequence of symbolic ids to MPI objects and memory pointers at different processes, resulting in identical signatures (thus the same CFG).

The algorithm for interprocess compression of CFGs also uses $\log_2 P$ stage of pairwise merges. We use a simple example (Figure 5) to illustrate this process. When two grammars are merged, a new rule $S \rightarrow S_1 S_2$ is generated, where S_1 and S_2 are new names for the root symbols of the two grammars. The names of nonterminal symbols are changed to prevent conflicts. The merged grammar encodes the concatenation of the two merged traces. Once all grammars have been merged, we run another Sequitur pass to compress the merged grammar.

Before we merge two grammars, we first check whether the two grammars are identical. The identity check saves time because it is much faster than a merge and identical grammars are frequent. We will show in Section 4 that for many programs the number of *unique grammars* produced is far less than the total number of processes.

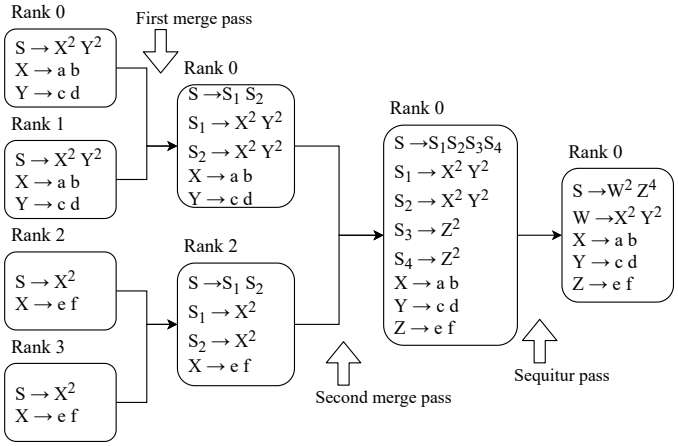


Fig. 5: Example of interprocess compression for CFGs.

The decompression is simply a process of recursive rule application. If the leftmost nonterminal symbol is always expanded first, then the traces of the successive ranks will be obtained in rank order. Parallelism can be easily applied to the decompression; it is also simple to extract the trace of any selected process.

2.6 Other Features

Pilgrim is under active development. This subsection describes two important features that have been added recently.

2.6.1 Multithreaded Support

It is not uncommon for an application to take advantage of both multiprocessing and multithreading. To initialize a multithreaded MPI application, users need to explicitly request a certain level of thread support. This is done by passing one of the following options to the `MPI_Init_thread` call:

- MPI_THREAD_SINGLE: Only one thread will execute.
- MPI_THREAD_FUNNELED: Only the thread that called `MPI_Init_thread` will make MPI calls.
- MPI_THREAD_SERIALIZED: Only one thread will make MPI library calls at one time.
- MPI_THREAD_MULTIPLE: Multiple threads may call MPI at once with no restrictions.

To support multithreaded MPI programs, Pilgrim includes a *tid* field in each call's record that represents the caller's thread id. Similar to the MPI objects, *tid* uses symbolic representation instead of the actual thread handle. No matter whether the thread is created by OpenMP [31] runtime or by explicitly calling pthread functions (e.g., `pthread_create`), when an MPI call is intercepted, Pilgrim uses `pthread_self` to retrieve the thread handle and maps it to a symbolic id. A hash table is used to maintain the map between thread handles and their symbolic ids. If the thread handle is unseen before, a new symbolic id is created. Otherwise, an existing symbolic id is returned. These symbolic ids are unique to the owner process.

Since *tid* is a part of a call's signature, the compression effectiveness may be affected depending on the thread usage. For applications initialized with MPI_THREAD_SINGLE and MPI_THREAD_FUNNELED, *tid* does not introduce any variance as it is always the same both within and across processes. For applications initialized with MPI_THREAD_SERIALIZED and MPI_THREAD_MULTIPLE, however, things can become messy. When different threads are calling MPI functions in a nondeterministic order, or when the MPI calling threads are dynamically created and destroyed, the MPI calls will become very hard to compress due to the unpredictable sequence of tids. Fortunately, such usages are very rare. Most applications we have seen use either a dedicated thread to invoke MPI calls or have a fixed group of threads making MPI calls in a deterministic order. The multithreaded support under these common usages should have no impact on the compression effectiveness.

2.6.2 Dynamic Tracing

Dynamic tracing is a feature that allows users to dynamically enable and disable tracing at runtime. With this feature, users can trace only the desired parts of an application, ignoring unimportant parts. This leads to smaller traces as well as easier postprocessing.

This feature does require minor modifications to the application code. Users need to instruct Pilgrim when to start and stop tracing. But instead of introducing any APIs, Pilgrim monitors an MPI function, `MPI_Info_set`, to receive the instruction. This way, applications only need to be recompiled but do not need to be linked against Pilgrim. We demonstrate the usage of this feature using a real-world application Nek5000 [32]. Below is an example that

shows how to trace only the solving phase of a Nek5000 application. With two extra `MPI_Info_set` calls, one can keep only the solving phase communications. We will show later in Section 4.3.2 the result of this exact usage.

```
// Nek5000 driver code in drive.f
call nek_init(comm)

// Turn on tracing
MPI_Info_set(info, "PILGRIM_TRACING",
                   "ON", ierr)
call nek_solve()
// Turn off tracing
MPI_Info_set(info, "PILGRIM_TRACING",
                   "OFF", ierr)

call nek_end()
```

3 Proxy App Generation

Some trace analysis, e.g., counting the number of occurrences of different MPI calls, can be done on the compressed trace. Others will decompress the trace first. We discuss here one trace consuming tool that, given an MPI code, generates a proxy app that does no computation but makes the same MPI calls. However, designing such a proxy app manually is a laborious process that faces several hurdles: (1) it requires involvement of experts of the original application and a good understanding of the logic of what can be a very large code; (2) it requires access to the source code of the original application, which may not be available; and (3) It is hard to test.

In this section, we propose an algorithm that can generate proxy apps automatically from Pilgrim traces with little or no human intervention. No access to the source code is needed. Correctness checking is easy because we can run the generated proxy app with Pilgrim again and compare its traces with the original application's traces.

A proxy app can be simply generated by replacing each entry in the decompressed trace by a suitable MPI call. But the code size of such an application will be proportional to the number of times MPI is called in the original application, which is not practical. Instead, we want to leverage the same recurring patterns that enable the high compression rate of Pilgrim, to obtain a concise proxy app with code size that is proportional to the size of the compressed trace.

The proxy app generator uses the final merged CST and CFG files as input. The generation process contains three major steps. The first two steps read and partially decompress the final CST and CFG files. The last step traverses the CFG and generates a call for each grammar symbol. A call for a nonterminal symbol on the left-hand side merely invokes recursively the calls for the symbols on the right-hand side, within a loop, if the symbol has a multiplicity count. A call for a terminal symbol uses the information stored in the CST to construct an actual MPI call.

3.1 Step One: Decoding the CST

The first step reads back the merged CST file and replace each CST entry with the information required for reconstructing MPI calls. To do so, we use the information stored in the CST and a table that specifies, for each function id, the function name, the number of parameters, and the "metadata" of parameters such as their type and size. To be specific, for each function parameter of a CST entry, we associate it with the following information:

- Parameter type, for example, int, double, MPI_Request, etc. This information is retrieved from the MPI standard.
- Parameter direction: input, output, or both. We need this information so we can declare variables properly. This information is also documented in the MPI standard.
- Array length. In most functions, the length of an array type argument can be restored from other integer arguments, sometimes indirectly. The most complicated case is possibly `MPI_Graph_map(MPI_Comm comm, int nnodes, const int indx[], const int edges[], int *newrank)`, where the length of array `edges` is `indx[nnodes-1]`. Other cases include functions such as `MPI_Alltoallw`, where the array length is the size of the communicator.

Similar to the wrappers that are generated to intercept MPI calls (Section 2.3.1), the code for retrieving the parameter's metadata is generated automatically from the Latex file of the MPI standard.

3.2 Step Two: Decoding the CFG

This step reads back the CFG and performs a *partial* decompression, where we recover only the starting symbol of each ranks' grammar. We do not further decompress the CFG to ensure the recurring code patterns are still preserved. Figure 6 shows an example of a partially decompressed CFG (bottom right). The left part of the figure shows the original code executed by each rank. Suppose the program is run with four ranks. After the partial decompression, the body of the starting rule (S) is composed of each rank's grammar: S_1 for rank 0, S_2 for rank 1, and S_3 for rank 2 and rank 3. As mentioned earlier, these grammars are not further unrolled. For example, the shared rule A that represents function `func()` is intact. This is critical because we can generate a function for rule A and all ranks can call the same function without duplicating the code. The details are given in the next subsection.

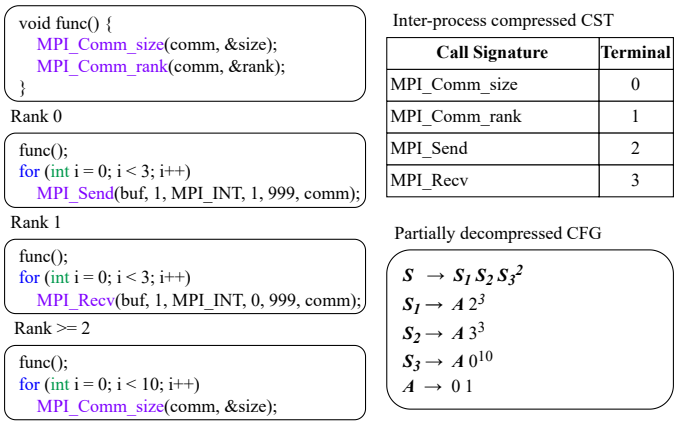


Fig. 6: Example of partially decompressed CFG.

This partially decompressed CFG is stored in memory. We do not reuse the data structure from the Sequitur algorithm because in the proxy app generation the grammar is read-only. Rather, we use a hash table to store each unique rule, where the key is a rule's id and the value is an array of symbols representing the rule's body. The hash-table-based CFG enables fast lookup for rules application (i.e., grammar decompression), which is a core operation that will be executed repeatedly in the next step.

3.3 Step Three: Generating the Proxy App

The partially decompressed CFG from the previous step still captures loop structures and common function calls across ranks. For example, in Figure 6, 2^3 and 3^3 represent the `MPI_Send` and `MPI_Recv` loop for rank 0 and rank 1, respectively. These recurring patterns are retained to reduce redundancies in the generated code. So eventually, the generated proxy app will reconstruct all recurring patterns identified by Pilgrim.

From a high-level view, the third step is similar to a full CFG decompression. Algorithm 1 shows the skeleton algorithm of this step, with many details omitted. The `Generate()` procedure takes the partially decompressed CFG as input and generates one function for each unique rule (e.g., `void A();` for a rule named A). The `Generate_function_pre()` and `Generate_function_post()` procedures are responsible for allocating, initializing, and freeing local variables that are needed by the calls made in that function. In between, the `Generate_function_body()` procedure iterates over every right-hand-side symbol of the given rule. This procedure fully decompresses the grammar by performing a recursive rule application. For each terminal symbol (i.e., an MPI call), it generates the call represented by that symbol by querying the decoded CST. For nonterminal symbols (i.e., grammar rules), it recursively applies the same procedure.

Algorithm 1 Proxy app generation algorithm

```
1:  procedure GENERATE_FUNCTION_BODY(rule)
2:      for each symbol of rule do
3:          if is_terminal(symbol) then
4:              Generate_call(symbol)
5:          else
6:              Generate_function_body(symbol)
7:          end if
8:      end for
9:  end procedure
10: procedure GENERATE(CFG)
11:     for each rule of CFG do
12:         Generate_function_pre(rule)
13:         Generate_function_body(rule)
14:         Generate_function_post(rule)
15:     end for
16: end procedure
```

The resulting code contains one function for each symbol. A rule of the form $A \rightarrow BC$ generates a function f_A that calls f_B and f_C; a rule of the form $A \rightarrow B^n$ will generate a function f_A with a loop where f_B is called n times. We then construct the `main()` function where each rank invokes the function representing its grammar. For example, the proxy app generated from the example in Figure 6 will have the following code structure.

```
void A() {}
void S1() { A(); ... }
void S2() { A(); ... }
void S3() { A(); ... }
int main() {
    if (rank == 0)
        S1();
    if (rank == 1)
        S2();
    if (rank >= 2)
        S3();
}
```

The rest of this subsection gives more implementation details of Algorithm 1.

3.3.1 Declaring Variables

Before generating an MPI call, we need to make sure that all needed variables are properly declared and initialized (performed by `Generate_function_pre()`). To this end, we need to know the type and direction of each call's parameter. This information should be available after the first step (Section 3.1).

Variables that are potentially used across functions will be declared as global variables. As a result, all the functions we generate have an empty input argument list and a void return type. Variables that are not shared across functions will be declared locally at the beginning of the function. In addition, array type variables need to be initialized at declaration time. The length of each array variable is also retrieved in the first step.

3.3.2 Generating MPI Calls

Now we describe how each MPI call and its parameters are generated, namely, the implementation of `Generate_call()` procedure in Algorithm 1.

Output parameters of MPI object types such as MPI_Request will be potentially used by later MPI calls. Thanks to the symbolic representation (Section 2.3.3), we know exactly which MPI calls update an MPI object and which MPI calls consume that object. Thus, during the code generation process, we maintain a table that maps from symbolic ids to variable names. An MPI call that uses an MPI object can consult this table for the actual variable name.

For input parameters, there are two cases based on their data types.

1) Basic data type parameters such as integer and string. Their corresponding variables are not required to be declared in advance. When we generate the MPI call, we can directly use the parameter value retrieved from the trace.
2) Parameters that are MPI objects. If the variable value is a built-in constant such as `MPI_INT`, we directly use that constant. In other cases, that variable must has been set by an earlier MPI call. We can query the <symbolic id, variable name> table to retrieve the variable name.

3.3.3 Memory Buffer Pointers

Arguments of memory buffer pointers require special handling. As discussed in Section 2.3.3, Pilgrim intercepts memory management operations and stores the symbolic representation of memory buffer pointers. During the proxy app generation, we need to place `malloc` and `free` calls in an order that preserves the original memory usage pattern.

A straightforward way is to store all malloc and free calls in the grammar like all other MPI calls. In this way we can faithfully replay all memory operations. However, malloc and free are called not only by an application's code but also by the libraries loaded by the application, including MPI. At the time of a `malloc` interception, we do not know whether the allocated memory buffer will be used by MPI calls. Replaying all malloc and free calls can be extremely inefficient if most are irrelevant to MPI tracing.

We confirmed this with several NAS parallel benchmarks. We ran them on one node with 8 ranks and counted the number of mallocs intercepted. The results are shown in Table 3. As we can see, in all applications only a small portion of malloc calls are performed by the application.

Benchmark	Total mallocs	Mallocs of buffers used by MPI calls
CG	23,563	24
IS	5,723	39
MG	14,522	684
LU	94,173	16
FT	2,871	24
BT	261	0

TABLE 3: Number of mallocs used by application MPI calls

We can know whether a buffer was used by MPI at the free time. Therefore, we intercept all malloc and free calls but record only free calls that freed a buffer used by MPI. With the free calls stored in the grammar, the proxy app generator can place them in the exact locations as in the original application. In addition, the generator will insert a matching malloc call when the buffer is accessed the first time.

3.3.4 Nondeterministic Loops

`MPI_Wait*` and `MPI_Test*` loops can introduce nondeterminism since the set of requests completed at each iteration can change from execution to execution. The code below shows two commonly used `MPI_Test*` loops. Unlike most of the replay work [11], [33], [34] that tries to exactly replay the original completion order, our proxy app tries to reconstruct the original loop structure.

```
while(req not finished) {
    MPI_Test(req);
    ...
}
while(not all reqs finished) {
    MPI_Testany(reqs);
    ...
}
```

Currently, the proxy app generator can handle nondeterministic loops that are controlled by a single list of requests, that is, where the loop terminates once all requests were completed. Since the test/wait call in each iteration may have different call signatures based on the requests finished, the nondeterministic loops stored in the grammar will not have the format of a^c as for deterministic loops. Rather, consider the two `MPI_Test*` loops shown above; they are likely stored in the format of $a_1\alpha a_2\alpha \dots a_n\alpha$, where a_i is the test call of the ith iteration and α represents the sequence of the rest of the MPI calls. For `MPI_Test`, $a_1 = a_2 = ...a_{n-1}$, where the request has not been completed and a_n is the last iteration's `MPI_Test` call that completes the request.

During the proxy app generation, the algorithm detects a_1 and a_n to determine whether we are inside a nondeterministic loop. If so, the algorithm generates a `while(n > 0)` loop, where n is the total number of requests to be completed. Inside this loop, we decrease n by n', the number of requests completed by each iteration. Here n' is a variable returned by the test/wait call of the generated app. For example, in an `MPI_Testsome` loop, $n' =$ `outcount`, the third parameter of `MPI_Testsome`.

3.4 Limitations

The current implementation has a few limitations. First, user-defined MPI functions (e.g., `MPI_Copy_function`) are not supported because Pilgrim traces function calls only at runtime and does not perform compile-time analysis.

Second, for nondeterministic loops, the generator only handles loops of the form described above, where each iteration invokes identical MPI calls (possibly none), in addition to the test/wait call. And the input request array remains the same within the same loop. We assume this is the most common case.

Third, the control logic of the generated proxy app is not guaranteed to be exactly the same as the original application. The potential code discrepancy arises from two factors: (1) it is possible that Pilgrim failed to recognize some recurring patterns in the first place, and (2) Pilgrim could recognize extra repetitions from the call sequence that were not written in the form of loops or functions in the original code. These discrepancies should have no impact on the communication pattern.

4 Evaluation

We evaluate Pilgrim by answering the following questions: (1) What is the trace size for large-scale runs? (2) How do trace size and overhead scale with the number of processes and the number of iterations? (3) How does Pilgrim compare with other systems? In all the experiments, the trace records preserve the values of all the arguments of the MPI calls, and the compression is lossless (except timing); each MPI call that uses an MPI object can be matched to the call that initialized this object. Thus, we can recover complete information on the original MPI calls. Since we developed both a compressor and decompressor, we can check correctness by comparing the original, uncompressed trace with a trace obtained by compressing, next decompressing the original trace.

We selected a variety of codes for the evaluation, as shown in Table 4. All experiments were conducted on Theta at Argonne National Laboratory. Theta is a Cray XC40 system consisting of 4,392 Intel KNL 7230 compute nodes.

Type	Code
Benchmark	2D and 3D Stencils OSU Microbenchmarks [35]
Miniapp	NAS Parallel Benchmark [36]
Production app	FLASH [29] and Nek5000 [32]

TABLE 4: Codes used for evaluation

Each compute node has 64 cores and 192 GB of DDR4 memory. The experiments we ran used up to 16,384 cores on 256 nodes.

4.1 Benchmarks

We tested two stencil codes: a 2D 5-point stencil with nonperiodical boundaries and a 3D 7-point stencil with periodical boundaries. They both use `MPI_Isend`, `MPI_Irecv`, and `MPI_Waitall` for communication. The meshes are distributed by using a block distribution on a Cartesian mesh of processes with the same number of dimensions.

Thanks to the relative rank optimization described in Section 2.4, Pilgrim compresses perfectly for regular stencil codes. The trace file size does not change with the number of iterations or the number of processes beyond a certain number. On an $M \times N$ Cartesian mesh of processes, process i will communicate with processes $i \pm 1$ (horizontal direction) or $i \pm N$ (vertical direction); boundary processes may communicate with `MPI_PROC_NULL`. There are 9 possible communication patterns (four corners, four sides, and interior). All patterns appear when a 3×3 mesh is used. Indeed, the compressed trace size does not grow beyond 9 processes.

Similarly, for the 3D periodical boundary stencil, there will be at most 27 different communication patterns, and the size of the compressed trace does not grow beyond 27 processes.

We also tested all the OSU microbenchmarks. Again, Pilgrim can compress perfectly across processes and iterations for all programs included in the OSU microbenchmarks. Most programs result in a trace file size of a few kilobytes. To save space, we do not include the exact numbers.

4.2 Pilgrim vs. ScalaTrace

Here we compare the compression effectiveness of Pilgrim with that of ScalaTrace [11]. ScalaTrace was built from the latest source code (V4). We configured ScalaTrace to retain tags (ignored by default) and use lossless tracing where possible. We selected six class D NAS parallel benchmarks (NPB) and ran them with increasing numbers of processes (BT, FT, and SP require a square number of processes to run on). Trace sizes are shown in Figure 7 for different NAS benchmarks and different process counts. We did not get the results of ScalaTrace for BT, MG, and SP when running with 16K processes. Those runs did not complete after several hours, whereas they normally finish in a few minutes without ScalaTrace. Of the six benchmarks, ScalaTrace outperforms Pilgrim in only CG when running with more than $8,192$ processes. In all other cases, Pilgrim generates shorter traces while keeping more information. Moreover, the gap between Pilgrim and ScalaTrace becomes larger when increasing the scale. We also performed a detailed comparison between Pilgrim and ScalaTrace using three FLASH simulations in our previous work [26]. We will not repeat the results here, but, to summarize, in all the cases we studied, Pilgrim achieved smaller trace sizes with significantly lower overheads. Next, we discuss Pilgrim's results on the NAS benchmarks in more detail.

FT and LU. Their communication patterns did not change with the number of processes, and Pilgrim was able to recognize all of their recurring patterns. FT uses only three collective MPI functions (`MPI_Reduce`, `MPI_Alltoall`, and `MPI_Bcast`) for communication. The 16K-process FT run invoked `MPI_Reduce` $884,736$ times, but there was only one unique call signature across all processes. LU, on the other hand, makes great use of point-to-point communication calls including `MPI_Send`, `MPI_Irecv`, and `MPI_Recv`. Its behavior is similar to that of the stencil codes discussed previously. Consider `MPI_Irecv` as an example: In the 16K processes LU run, Pilgrim recognized 30 unique `MPI_Irecv` call patterns, for a total of $19,800,316$ invocations.

BT and SP. These two benchmarks have identical communication behaviors. The parallelization technique used by them was first described by Bruno and Cappello [37]. The grid is cubically divided into cells, which are assigned to processors using a gray code. The chosen cell-to-processor mapping guarantees that every processor has a fixed predecessor and successor in any direction (x, y, or z). In each time step, a forward pass and a backward pass were performed in $\sqrt{P}$ stages along each direction, where P is the number of processors. `MPI_Isend` was used to send the surface data to each one's successor, and `MPI_Irecv` was used to receive that data from each one's predecessor. Such a pattern is easy to recognize, as suggested by the result for BT. However, SP's implementation makes matters more complicated. Unlike BT where the surface size moved by each process is fixed, in SP it depends on the sender's "position" (corner, boundary, or interior). The number of unique call signatures is fixed as the number of surface size combinations is finite. However, the order of MPI calls is not fixed because a sender's position changes with stages along a direction. For example, rank 1 may send 10 bytes in the first stage and 20 bytes in the second stage, whereas rank 2 may first send 20 bytes and then 10 bytes. Therefore, for SP, the CST size does not change with the processor count, but the CFG size grows with it.

CG and MG. The trace size for these two benchmarks exhibited sublinear growth with the number of processes. The growth is caused by the source and destination parameters in point-to-point calls. In CG, all processes are laid out in a 2D grid where process (i, j) communicates with process (j, i). On a $P \times P$-processor grid each process communicates with rank $dst = myrank\%P \times P + myrank/P$. This pattern was not recognized by Pilgrim (it recognizes only linear patterns for now). The number of unique $(myrank, dst)$ pairs is P^2. However, with the relative-ranks optimization (Section 2.4.1), this number is reduced to P. Even though Pilgrim does not compress perfectly for transpose communication patterns, it still reduces the growth rate of distinct patterns to be proportional to the square root of the number of processes. As a result, it takes only a few hundred kilobytes for both benchmarks to store all the MPI

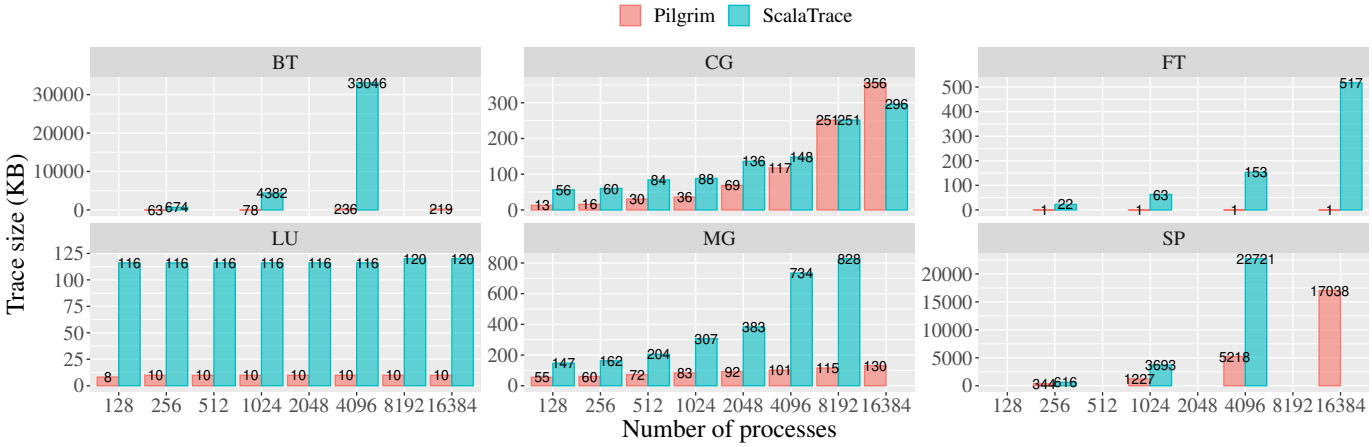


Fig. 7: Comparison of trace file size with NPB.

calls at their largest scales. At the largest scale, ScalaTrace produced a smaller trace size for CG. The reason is unclear to us. We suspect some ad-hoc optimizations are adopted by ScalaTrace to handle this specific communication pattern.

4.3 Scalability

In this subsection, we evaluate the scalability of Pilgrim using scientific simulations, and we discuss additional factors that affect the compression ratio.

4.3.1 FLASH

We tested three simulation problems that come with the FLASH package: Sedov, Cellular, and StirTurb. All are 3D simulations with I/O disabled. To add more variety to the tests, we disabled the adaptive mesh refinement (AMR) feature for Cellular and StirTurb and kept it for Sedov. We also used two different meshes: uniform mesh for Cellular and a PARAMESH [38] generated mesh for Sedov and StirTurb. These differences result in a different set of MPI functions being used, as we will show later.

Figure 10 shows how the trace size scales with the number of processes and the number of iterations. We also plotted in the right of each subfigure the total number of MPI calls Pilgrim encountered. As expected, the number of MPI calls increases linearly with the number of processes and iterations.

4.3.1.1 Trace Size vs. Number of Processes: When varying the process count (Figure 10 (a)), we kept the number of iterations fixed at 500. The problem size per process was also kept unchanged (weak scaling).

For Sedov and StirTurb, the trace file size did not change with the process count, suggesting that all communications patterns were encountered and recognized by Pilgrim. The trace size of Cellular changed slightly with the process count, presumably because the data layout changed according to the number of processes. Overall, in all three simulations the trace file size stayed at a stable level, which shows the perfect scalability of Pilgrim with the number of processes.

To further understand the differences between the three simulations, we show in Table 5 some important statistics of the traces collected from the 16K-process runs. Cellular and StirTurb used only blocking MPI calls, whereas Sedov also used a few nonblocking calls during the adaptive mesh refinement process. The exact set of MPI functions used and the number of calls made by each simulation are shown in Figure 8.

	Cellular	StirTurb	Sedov
MPI functions used	14	20	27
Unique grammars	28	2	74
CST size (KB)	4.68	3.77	410.21
CFG size (KB)	2.07	2.11	147.84

TABLE 5: Statistics of the traces collected from 16K-process FLASH simulations.

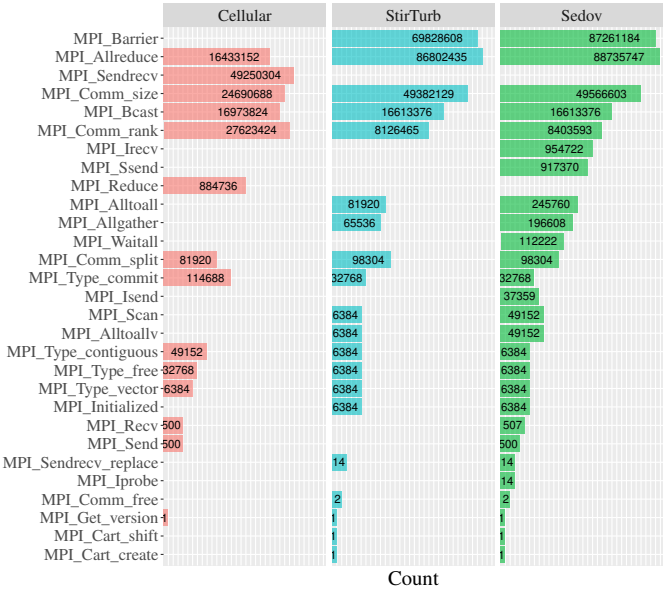


Fig. 8: MPI calls count of 16K-process FLASH simulations. Cellular, StirTurb, and Sedov used 14, 20, and 27 unique MPI functions, respectively.

Also in Table 5, we can see that StirTurb has the simplest communication pattern, since it produced only two unique grammars across 16K processes. This is further confirmed in Figure 9, where we visualize the communication patterns

of each simulation by showing the data size exchanged between processes. We used 64-process 100-iteration runs for easy visualization. In StirTurb, all processes except rank 0 sent and received the same amount of data from and to all others, which corresponds to two unique grammars. More interesting is that the communication pattern of Cellular is almost identical to that of the NAS MG benchmark showed in Figure 7 of [34]. We will not further discuss these communication patterns since this topic is beyond the scope of our work. We note, however, that all the above analyses and visualizations are based purely on the Pilgrim traces, thus demonstrating the usefulness of the detailed information stored by Pilgrim.

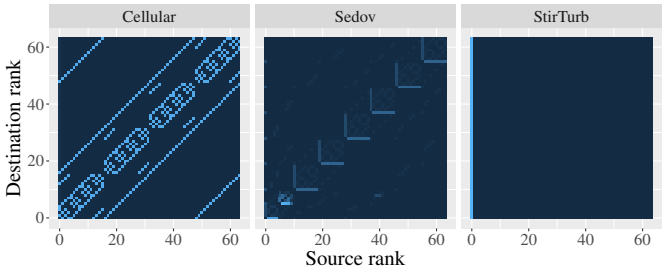


Fig. 9: FLASH communication patterns for 64-process 100-iteration runs.

4.3.1.2 Trace Size vs. Number of Iterations: When varying the number of iterations (Figure 10 (b)), the process count was fixed at 16,384. Cellular and StirTurb produced constant-size trace files. These two simulations did not introduce any new communication patterns when increasing the number of iterations. Sedov internally uses the PARAMESH library to perform parallel AMR. It builds a hierarchy of subgrids to form the compute domain. These subgrid blocks are stored by using a tree data structure. At each refinement phase, new child blocks will be created and added. The blocks are sorted in Morton order so as to compute a load-balanced partition with good locality. Afterwards, data blocks may be moved across processes to rebalance the load. The communication is performed by using point-to-point calls: `Isend`, `Irecv`, and `Waitall`. The communication pattern changes at each refinement. In our runs, the AMR was triggered only once at the beginning of the run. Thus, we did not observe sharp increases in the trace size across iterations. The small trace size increase was due to some extra `MPI_Send/MPI_Recv` being called with new sources and destinations. This is caused by the output mechanism where rank 0 asks for the current minimum simulation time delta; the source of that datum changes every few hundred iterations.

Finally, we note that Pilgrim can store complete traces from the hundreds of millions of MPI calls generated by a multiminute run with 16K processes in 600 KB for Sedov and just 6 KB for Cellular and StirTurb.

4.3.2 Nek5000

Nek5000 [32] is a scalable CFD solver, whose simulations perform heavy communications. The simulation we ran is called *Fully Developed Laminar Flow (FDLF)* [39]. We choose this simulation as it exhibits different communication behaviors in different execution phases, which presents challenges to Pilgrim and raises interesting discussions regarding communication pattern recognition.

We first evaluated how the trace size fluctuates with the number of simulated time steps. Our results showed that for a fixed processor count, the trace size stayed unchanged regardless of the number of time steps. In other words, simulating with more time steps does not introduce new communication patterns. The same phenomena was also observed in the previous FLASH experiments.

Scaling with the number of processes provides a different picture. We fixed the number of time steps to 10 and conducted weak scaling experiments. We found that the trace size grows linearly with the number of processes (Table 6): The FDLF runs exhibited some communications patterns that were poorly compressed by Pilgrim across processes.

Procs	128	256	512
Trace size (KB)	6,733	13,765	27,399

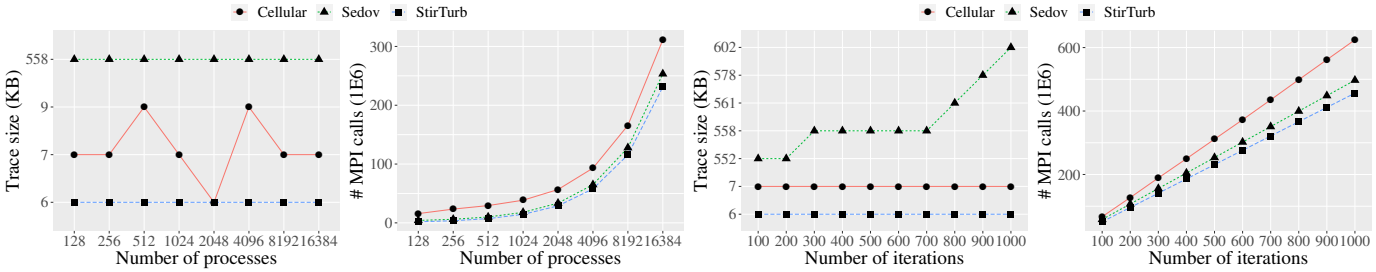
TABLE 6: Trace size of FDLF runs with 128, 256, and 512 processes.

Our first investigation revealed the increase in trace size is mostly due to four point-to-point functions: `MPI_Send`, `MPI_Recv`, `MPI_Isend` and `MPI_Irecv`. As shown in Table 7, Pilgrim observed an increasing number of unique call signatures when more processes were used, which confirms that the communication patterns exhibited by these calls were not fully recognized by Pilgrim. As a result, Pilgrim generated one distinct grammar per process, thus the linear increase in the trace size.

Procs	128	256	512
MPI_Send	2,461	4,623	8,841
MPI_Recv	2,732	4,872	8,956
MPI_Isend	32,496	66,518	131,666
MPI_Irecv	30,810	64,144	128,382

TABLE 7: Unique call signatures count of FDLF runs with 128, 256, and 512 processes.

We then examined the source code of Nek5000 (the part used by FDLF) to understand its exact communication behaviors. A Nek5000 simulation run is divided into three major phases: initialization, solving, and finalization. During the solving phase, FDLF uses a custom communication library called *gslib* to perform gather and scatter communications on a binary tree. We illustrate the gather communication with a 13-process communication tree in Figure 11. The numbers are MPI ranks and the arrows indicate the message flow. Now consider send calls, with Pilgrim's rank-related encoding, there are only 4 unique destinations (-1, -2, -3, and -6), indicated by different arrow colors. Similarly, for receive calls, only 4 unique sources (1, 2, 3, and 6) will be observed. In general, the number of unique displacements (i.e., sources and destinations) is $O(logN)$. So we should expect a logarithmic increase in the trace size as the number of processes, if only considering the solving phase. The initialization and finalization phases, however, have some inefficient communication patterns. For example,



(a) Trace size vs. number of processes.

(b) Trace size vs. number of iterations.

Fig. 10: Evaluation of Pilgrim's trace size with FLASH simulations. When varying the process count (subfigure (a)), the number of iterations was fixed at 500. When varying the number of iterations (subfigure (b)), the process count was fixed at 16K. We also plot on the right of each subfigure the number of MPI calls intercepted by Pilgrim.

at both phases, rank 0 uses point-to-point calls to send and receive data to and from every other process directly, which leads to N distinct grammars when the rank-related encoding is enabled. This is because, for rank i, its send (or receive) will have a unique destination $-i$ (or source i) instead of the same 0. Disabling the rank-related encoding will solve this issue, but it will also jeopardize the pattern recognition during the solving phase.

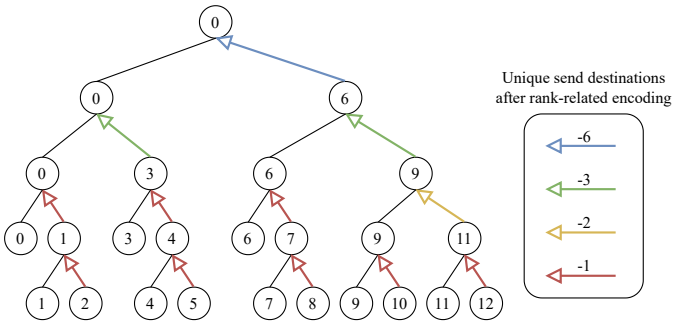


Fig. 11: The gather communication of FDLF on a 13-process communication tree. Eventually, rank 0 receives data from all other ranks. With rank-related encoding, send calls only have four unique destinations as indicated by colors.

To confirm our study about the FDLF's communication patterns. We used the dynamic tracing feature discussed in Section 2.6.2 to help isolating the solving phase. Figure 12 shows how the trace size scales with the number of processes. We used up to 16K processes, and at each scale, we performed two runs, one traced all three phases and the other traced only the solving phase. As expected, when only the solving phase was traced, Pilgrim produced a very small trace size with a slow growth rate since the involved communication patterns compressed well. On the other hand, when all phases were traced, the trace size was significantly larger and increased in a linear fashion. Even though most MPI calls are made during the solving phase, the space needed by Pilgrim to store them only takes up a tiny portion of the entire trace file. So in this case, people interested more in the solving phase can choose to disable tracing during initialization and finalization.

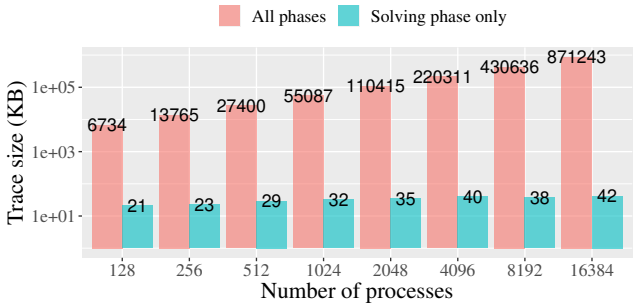


Fig. 12: Evaluation of Pilgrim's trace size with FDLF. The number of time steps was fixed at 10.

4.4 Overhead

Pilgrim's overhead comes from two sources: intraprocess compression overhead, due to the construction of CFG and CST, and interprocess compression at the finalize point. The first part scales well because the intraprocess compression is totally independent across processors. The latter part normally takes up the most of time as we have reported in our previous paper [26]. To break it down, compressing CSTs normally takes negligible time, while the compression for CFGs dominates and is largely due to the sequential final Sequitur pass. This overhead depends on the number of unique grammars. Fortunately, in most cases there are only a few unique grammars. From an application's perspective, the actual overhead incurred largely depends on the ratio between the application's computation and communication. In general, the higher this comp-to-comm ratio, the lower the overhead.

In this subsection, we evaluate Pilgrim's overhead using again FLASH simulations with various process counts and number of iterations. We measured the execution time of FLASH runs with and without Pilgrim. We conducted all experiments using the same number of blocks, but two different block sizes. This does not change the communication pattern (except for message sizes), but changes the comp-to-comm ratio.

4.4.1 Overhead vs. Number of Processes

Figure 13(a) shows the normalized execution time against the process count. Pilgrim shows good weak-scaling

performance—increasing the number of processes does not increase the overall compression time.

For the block size of 8x8x8, the average execution time was increased by about 20.10% for Sedov, 2.64% for StirTurb, and 1.80% for Cellular. The overhead of Sedov is higher because it has more unique grammars than the other two have and thus required more time for the final Sequitur pass. Cellular has a more complex communication pattern than StirTurb has, but nevertheless it shows a lower overhead because it is much more computation intensive than StirTurb. For example, a 16K-process Cellular run takes about 70 seconds to finish whereas StirTurb can complete in about 30 seconds.

When running with a larger block size (16x16x16), the overall execution time was increased, but the communication pattern and the number of unique grammars stayed the same. Therefore, the time for Pilgrim to perform the intra- and interprocess compression remained the same. As a result, the average overhead was reduced to $1.92\%, 1.66\%$, and 0.30% for Sedov, StirTurb, and Cellular, respectively. We expect that the larger the problem size is, the lower the overhead is.

4.4.2 Overhead vs. Number of Iterations

Figure 13(b) shows the normalized execution time against the number of iterations. The relative Pilgrim overhead decreases for Cellular and StirTurb as the number of iterations increases. These codes generate a fixed number of grammars, independent of the number of iterations, so that intraprocess compression time is fixed and amortized against larger execution time with the growth in iteration count. Once again, in all simulations the larger the block size, the lower the overhead.

4.5 Compressing Timing Information

We evaluated all six timing compression techniques discussed in Section 2.3.2. To quickly recap, the first two, CFG and HIST, are proposed by this work, whereas SZ and ZFP are existing lossy compressors. The last two, SZ-Clustered and ZFP-Clustered, are modified versions of SZ and ZFP with binning by call signature applied before compression.

We ran 16K-process FLASH simulations six times, and we enabled one different timing compression technique each time. Unlike the MPI call sequence, the timing information is more difficult to compress because of the intrinsic nondeterminism introduced by noises and irregularity in computations. A trade-off between accuracy and overhead has to be made when storing the timing information. In our experiments the relative error bound was set to 10% for CFG, HIST, and SZ. ZFP supports only an absolute error bound, which we set to $10\mu s$. Moreover, k was set to 8 for HIST. The buffer size was set to 1MB for all algorithms.

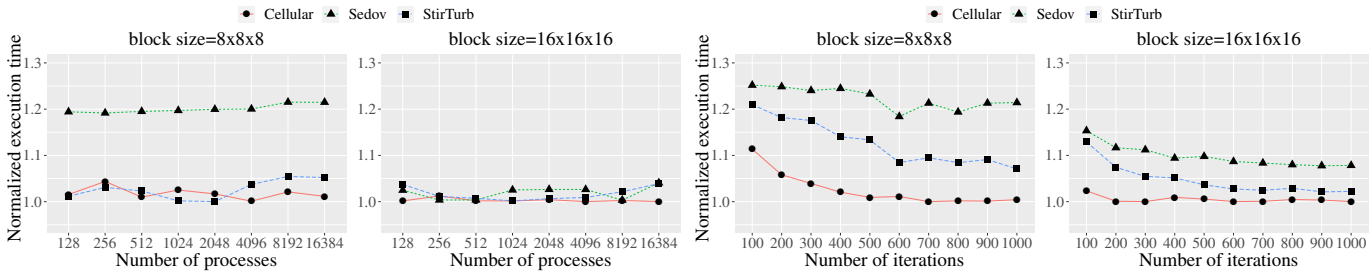
Figure 14 shows the compression ratios of durations and intervals for each simulation. First, for durations, the clustered versions of SZ and ZFP achieved higher compression ratios than did the unmodified versions. Clustering groups the durations according to their calls' signature. Identical calls have similar durations, as discussed in Section 2.3.2, thus making compression easier. On the other hand, clustering hinders the compression ratio for intervals, a phenomenon that was not expected. After investigation, we found that the cause is that clustering brings together intervals of an identical call from different loops that have different intervals. This separates the intervals of adjacent calls in the same loop, where these adjacent calls are more likely to have similar intervals. A better approach would be clustering intervals according to the loop structure instead of the call signature, but we leave this to future work. Second, CFG achieved lower compression ratios than SZ and HIST did in all cases. The results confirmed our discussion in Section 2.3.2 in that the CFG-based algorithm is not a good fit for compressing timestamps. Third, HIST achieved the best compression ratio in all the simulations except intervals of Sedov, where it is close to that of SZ.

Next we study the different algorithms in more detail using the Sedov simulation. In the previous experiment (Figure 14), we used a fixed absolute error bound for ZFP and a relative error bound for all other algorithms. This does not make a fair comparison, as the fixed absolute error bound was too tight and thus hindered compression. So we enlarged ZFP's absolute error bound to 1ms for durations and 10ms for intervals and repeated the experiment. The result is shown in Table 8. We also report a new metric PSNR, which is a commonly used metric for evaluating overall compression qualities (higher the better). It is clear that when a very tight error bound was used, ZFP (and ZFP-Clustered) achieved the lowest compression ratio, but the highest compression quality. When we loosened the error bound of ZFP to 1ms/10ms, the compression ratio improved at the expense of the overall compression quality. Moreover, the results suggest that none of the compression algorithm we tested is clearly superior to the others.

	Duration		Interval	
Algorithm	CR	PSNR	CR	PSNR
ZFP ($10\mu s/10\mu s$)	4.48	108.99	3.62	140.58
ZFP-Clustered ($10\mu s/10\mu s$)	5.26	109.12	3.67	140.61
ZFP (1ms/10ms)	11.51	68.80	8.52	84.97
ZFP-Clustered (1ms/10ms)	14.63	70.88	8.09	84.02
SZ	10.06	60.33	16.03	71.39
SZ-Clustered	12.18	61.47	15.39	71.31
HIST	15.20	42.25	15.36	52.97
CFG	6.92	80.28	7.18	82.89

TABLE 8: Compression ratio (CR) and PSNR of different timing compression algorithms.

We further analyze the throughput of each algorithm as well as the time spent on writing out the compressed timestamps. Once the compression is finished, Pilgrim uses MPI-IO to perform collective writes to output a single file. In our experiment, each Sedov run generated a total of 231,129,864 MPI calls, which requires 3.44GB to store all durations and intervals if uncompressed. Table 9 shows the aggregated compression ratio and the compression and I/O time of each algorithm. For each of the six compression algorithms the I/O time saved by compression is one or two orders of magnitude larger than the time spent compressing. For example, for HIST, the I/O time without compression would have been $15.28 \times 3.27 = 49.97$ (we assume that I/O time is proportional to data size). 0.124 seconds compression time saved 46.70 seconds I/O time.



(a) Normalized execution time vs. number of processes. (b) Normalized execution time vs. number of iterations.

Fig. 13: Evaluation of Pilgrim's overhead with FLASH simulations, with two tile sizes.

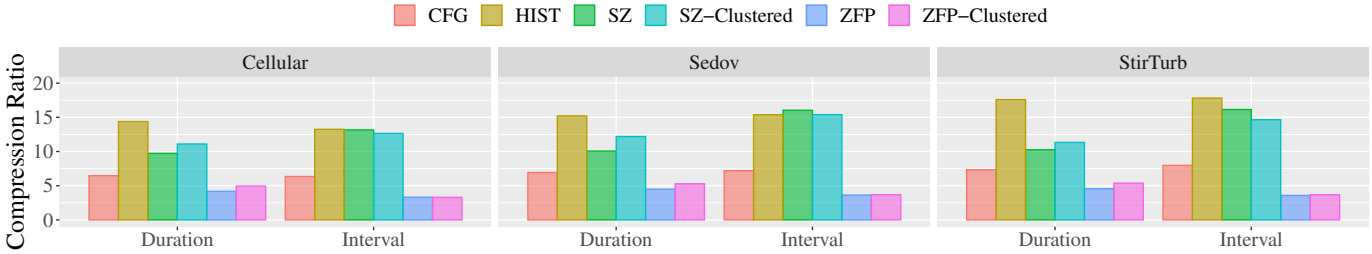


Fig. 14: Comparison of different timing compression techniques with 16K-process FLASH simulations.

Algorithm	Aggregated CR	Elapsed time (seconds) Compression	I/O
ZFP	4.00	0.043	12.13
ZFP-Clustered	4.32	0.038	10.45
SZ	12.36	0.115	4.78
SZ-Clustered	13.60	0.088	4.56
HIST	15.28	0.124	3.27
CFG	7.05	0.689	7.90

TABLE 9: Aggregated compression ratio (CR), compression time, and I/O time (seconds) of each algorithm.

4.6 Automatic Proxy App Generation

We tested the correctness of our proxy app generator using the test suite shipped with MPICH [40]. The test suite includes a few hundred tests of individual MPI routines and combinations of routines and covers most functions specified by the MPI standard. Each test was run with Pilgrim to collect the traces. Next, the corresponding proxy app was generated from the traces. Then, each generated proxy app was run with Pilgrim, and the produced traces were compared with the original traces. Our proxy app generator passed all tests.

Next, we generated proxy apps from FLASH traces with distinct process counts. As mentioned in Section 3, the code size of the generated proxy app should be proportional to the total number of symbols in the final CFG. To confirm this, we show in Figure 15 all these counts for the three FLASH simulations. The results for Cellular clearly show the correlation between the LOCs of the generated proxy app and the final grammar size. Moreover, Cellular and StirTurb generated trace files of similar sizes (Table 5), but Cellular produced more unique process grammars. Therefore, the proxy app of Cellular is larger than that of StirTurb. Furthermore, similar to the trace size experiments (Figure 10 (a)), for Sedov and StirTurb the grammar size is constant regardless of the number of processes, which also leads to a fixed-size proxy app. The LOC count of Sedov's proxy app is much larger because of its complex communication pattern, as suggested by its grammar size.

5 Related Work

5.1 MPI Tracing

There is a significant number of MPI profiling and tracing tools. This includes profiling tools such as AutoPerf [15], mpiP [16], and IPM [17] and tracing tools such as Recorder [41], Vampir [1], TAU [13], Score-P [12], ScalaTrace [42], and Cypress [10]. In this section we focus on tracing tools since they are more related to our work.

Several tools, including Score-p, Vampir, and TAU, support a tracing format named OTF [43] or some optimized versions of it, for example, OTF2 [44], OTFX [45], and [46]. OTF is a general format in that it is not limited only to communication events. It uses ZLIB compression to reduce the trace size, but interprocess compression is not supported. These tools lack structure-aware compression, and achieve lower compression rates.

Recorder [41] traces both communication and I/O events and uses a sliding-window-based approach to compress similar events within the window. But it cannot detect loop structures or repetitions at long ranges. Xu et al. [47] introduced a framework for identifying the maximal loop nest. Their algorithm can also discover long-range repeating communication patterns. However, both these tools do not perform interprocess compression, which is essential at large scales.

ParLOT [48] is a whole-program-tracing library built on top of Pin [49] that traces all function calls (but not

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2022.3215942

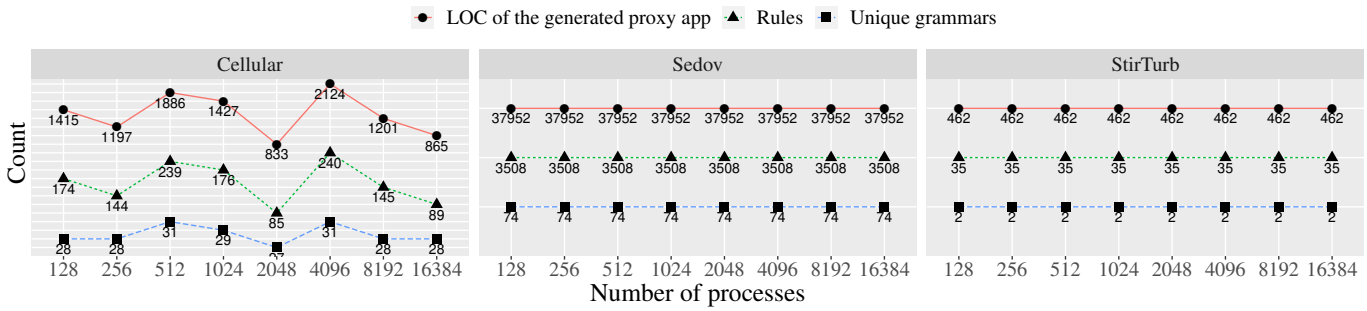


Fig. 15: Lines of code (LOCs) of the generated proxy apps using FLASH traces from runs with increasing process counts. We also show in each subfigure the number of unique grammars and the number of rules in the final CFG.

their arguments). It performs incremental online compression so that each process will never store uncompressed information. The compression is achieved by using two general-purpose compression algorithms, which may not take advantage of the loop structures.

Krishnamoorthy et al. [14] proposed a framework, similar to Pilgrim's framework, that augments the Sequitur algorithm to compress communication traces. The major limitation is that for each intercepted function call, only a small number of parameters are encoded and stored. This approach helps the compression rate but discards important information. In addition, the interprocess compression does not fully exploit the possible redundancy between grammars. It merges rules from multiple grammars without the redundancy check and thus can lead to a high overhead for regular SPMD programs.

ScalaTrace [11] focuses mainly on recording communication events and features on-the-fly compression. It extends regular section descriptors to exploit the patterns of repeating communication events involved in loop structures. It was designed for SPMD-style programs where there is no inconsistent program behavior across processes or time steps. ScalaTrace II [42] addresses this limitation of its predecessor. The intranode and internode loop detection algorithms were redesigned to improve the compression effectiveness for scientific codes that demonstrate inconsistent behavior across time steps and processes. More recently, Bahmani and Mueller [50], [51] proposed a signature-based clustering algorithm and context-aware clustering algorithm for ScalaTrace II to further improve the interprocess compression. However, they require that *markers* be inserted into the user code in order to inform the framework to run the clustering algorithm. The user is responsible for finding good locations for the markers.

Overall, ScalaTrace and its successors follow a bottom-up approach that first compresses the traces locally within each process and then performs an interprocess compression at the finalize point. In comparison, Cypress [10] takes a top-down approach where it first runs a static pass offline to retrieve the loop-and-branch information of the targeting program and then performs the intraprocess compression at runtime. The static pass relies on compiler analysis and normally is more efficient and accurate than online loop detection. A key limitation of Cypress is that it requires access to the source code that needs to be converted into the format of LLVM IR. Also, many functions are not recorded or compressed by Cypress (Table 1), including some popular ones such as `MPI_Wait`. Moreover, both ScalaTrace and Cypress require the user's program to be linked against their library. Pilgrim, on the other hand, performs runtime instrumentation so it does not need to access or rebuild the user's program.

One important goal of Pilgrim is to provide insights to MPI developers who are deploying MPI to the next-generation supercomputers. It is important that we cover the complete set of MPI functions and all their parameters. As far as we know, none of the existing systems achieve this goal. They either miss some functions or keep only part of the parameters (or both). Moreover, many corner cases are ignored in order to simplify the implementation or to achieve a higher compression ratio.

5.2 Proxy App Generation

When it comes to the modeling of MPI performance and the studying of an application's communication characteristics, most existing tools [6], [7], [11], [33], [34], [52], [53] rely on replaying communications using traces as their input. They can either generate and execute the original MPI calls, or drive a simulator using the trace information. For example, ScalaReplay [11] can be used to replay the compressed traces collected by ScalaTrace. ScalaReplay preserves the retained MPI semantics such as the ordering of MPI events. ScalaExtrap [7] takes it even further, it allows replaying the communications at larger scales using the traces collected from a small-scale run. The key idea is to extrapolate the communications parameters (e.g., communication groups and execution time) for the targeted scale and topology. PSINS [6] is a simulator, which models both communication and computation. It also considers factors such as network contentions and CPU burst events. A more recent tool, LCR [34], uses a given trace to train a recurrent neural network that models communication behaviors across different processes at different timestamps. It uses the neural network to reconstruct the trace by predicting a sequence of MPI events. LCR includes a trace-driven simulator as the replayer, which simulates the state change of each process when the MPI events occur, and reproduces required statistics of the original MPI program's communication behaviors.

All the above-mentioned efforts require a trace format specific trace interpreter. In contrast, our proxy app generator generates an actual executable that preserves identical

communication patterns as in the original application. The generated proxy app is just an ordinary MPI application which can be run without the input of the original traces and connected to any MPI performance tool.

6 Conclusion

In this paper we presented Pilgrim, a scalable and near-lossless MPI tracing tool. Compared with existing tools, the key advantage of Pilgrim is that it captures the complete set of MPI calls and their parameters. The proposed CFG- and CST-based compression algorithm empowers the near-lossless tracing at large scales. Our evaluation shows that Pilgrim preserves more information with less space and lower overhead than other tools. Since Pilgrim uses its own trace format, existing postprocessing tools cannot be used directly with Pilgrim traces. We have developed a decoder that decompresses and decodes the traces into original uncompressed trace records.

In addition, a proxy app generator is proposed to show one possibility of utilizing the detailed information stored by Pilgrim. The generation of proxy apps with this generator requires significantly less effort and is significantly less bug-prone than traditional manual generation is. One limitation, however, is that the generated proxy app can only be run with the same number of processes as the original application execution when the traces were collected. We expect future work to address this.

Finally, tracing applications with dynamic communication behaviors raises an interesting open question: Is it possible to automatically enable/disable certain optimizations (like rank-related encoding) or dynamically switch pattern detection algorithms during an application's execution? We need to study more applications and their communication patterns to have a better understanding of this question. We believe this is a research direction worth pursing.

Acknowledgments

This research was supported by NSF OAC grant 19-09144 and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, and by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.

References

[1] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The Vampir Performance Analysis Tool-Set," in *Tools for high performance computing*. Springer, 2008, pp. 139–155.

[2] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca Performance Toolset Architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[3] T. Hilbrich, M. Schulz, B. R. de Supinski, and M. S. Müller, "MUST: A Scalable Approach to Runtime Error Detection in MPI Programs," in *Tools for high performance computing 2009*. Springer, 2010, pp. 53–66.

[4] R. Preissl, M. Schulz, D. Kranzlmüller, B. R. de Supinski, and D. J. Quinlan, "Using MPI Communication Patterns to Guide Source Code Transformations," in *Computational Science – ICCS 2008*, M. Bubak, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 253–260.

[5] N. Jain, A. Bhatele, S. White, T. Gamblin, and L. V. Kale, "Evaluating HPC Networks via Simulation of Parallel Workloads," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 154–165.

[6] M. M. Tikir, M. A. Laurenzano, L. Carrington, and A. Snavely, "PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications," in *European Conference on Parallel Processing*. Springer, 2009, pp. 135–148.

[7] X. Wu and F. Mueller, "ScalaExtrap: Trace-Based Communication Extrapolation for SPMD Programs," *ACM SIGPLAN Notices*, vol. 46, no. 8, pp. 113–122, 2011.

[8] S. Sodhi, J. Subhlok, and Q. Xu, "Performance Prediction with Skeletons," *Cluster Computing*, vol. 11, no. 2, pp. 151–165, 2008.

[9] S. Sodhi and J. Subhlok, "Automatic Construction and Evaluation of Performance Skeletons," in *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2005, pp. 10–pp.

[10] J. Zhai, J. Hu, X. Tang, X. Ma, and W. Chen, "Cypress: Combining Static and Dynamic Analysis for Top-Down Communication Trace Compression," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 143–153.

[11] 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*, vol. 69, no. 8, pp. 696–710, 2009.

[12] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony *et al.*, "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011*. Springer, 2012, pp. 79–91.

[13] S. Shende, A. D. Malony, W. Spear, and K. Schuchardt, "Characterizing I/O Performance Using the TAU Performance System," in *PARCO*, 2011, pp. 647–655.

[14] S. Krishnamoorthy and K. Agarwal, "Scalable Communication Trace Compression," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE, 2010, pp. 408–417.

[15] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, "Characterization of MPI Usage on a Production Supercomputer," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 386–400.

[16] J. S. Vetter and M. O. McCracken, "Statistical Scalability Analysis of Communication Operations in Distributed Applications," in *Proceedings of the eighth ACM SIGPLAN symposium on Principles and Practices of Parallel Programming*, 2001, pp. 123–132.

[17] D. Skinner, "Performance Monitoring of Parallel Scientific Applications," Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US), Tech. Rep., 2005.

[18] M. Heroux, R. Neely, and S. Swaminarayan, "ASC Co-design Proxy App Strategy," *Sandia, LLNL, LANL*, 2013.

[19] M. P. I. Forum, "MPI: A Message-Passing Interface Standard Version 4.0," https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf, 2021.

[20] C. G. Nevill-Manning and I. H. Witten, "Identifying Hierarchical Structure in Sequences: A Linear-Time Algorithm," *Journal of Artificial Intelligence Research*, vol. 7, pp. 67–82, 1997.

[21] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross, "Using Formal Grammars to Predict I/O Behaviors in HPC: The omnisc'IO Approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2435–2449, 2015.

[22] T. Jones and G. A. Koenig, "A Clock Synchronization Strategy for Minimizing Clock Variance at Runtime in High-End Computing Environments," in *2010 22nd International Symposium on Computer Architecture and High Performance Computing*. IEEE, 2010, pp. 207–214.

[23] S. Hunold and A. Carpen-Amarie, "Hierarchical Clock Synchronization in MPI," in *CLUSTER*, 2018, pp. 325–336.

[24] Mellanox, "Highly Accurate Time Synchronization with ConnectX-3 and TimeKeeper," 2020.

[25] Y. Collet, "Zstandard – Real-Time Data Compression," https://facebook.github.io/zstd/.

[26] 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*, 2021, pp. 1–14.

[27] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello, "Error-Controlled Lossy Compression Optimized for

High Compression Ratios of Scientific Datasets," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 438–447.

[28] J. Diffenderfer, A. L. Fox, J. A. Hittinger, G. Sanders, and P. G. Lindstrom, "Error Analysis of ZFP Compression for Floating-Point Data," *SIAM Journal on Scientific Computing*, vol. 41, no. 3, pp. A1867–A1898, 2019.

[29] "Flash Center for Computational Science," http://flash.uchicago.edu, 2019.

[30] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[31] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[32] "NEK - Fast High-Order Scalable CFD," https://nek5000.mcs.anl.gov, Aug 2022.

[33] K. Sato, D. H. Ahn, I. Laguna, G. L. Lee, and M. Schulz, "Clock Delta Compression for Scalable Order-Replay of Non-Deterministic Parallel Applications," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–12.

[34] J. Sun, T. Yan, H. Sun, H. Lin, and G. Sun, "Lossy Compression of Communication Traces Using Recurrent Neural Networks," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[35] "OSU Micro-Benchmarks 5.7," http://mvapich.cse.ohio-state.edu/benchmarks, Dec 2020.

[36] "NAS Parallel Benchmarks," https://www.nas.nasa.gov/publications/npb.html, Feb 2020.

[37] J. Bruno and P. R. Cappello, "Implementing the Beam and Warming Method on the Hypercube," in *Proceedings of the third conference on Hypercube concurrent computers and applications-Volume 2*, 1989, pp. 1073–1087.

[38] P. MacNeice, K. M. Olson, C. Mobarry, R. De Fainchtein, and C. Packer, "PARAMESH: A Parallel Adaptive Mesh Refinement Community Toolkit," *Computer physics communications*, vol. 126, no. 3, pp. 330–354, 2000.

[39] "Fully Developed Laminar Flow," https://nek5000.github.io/NekDoc/tutorials/fdlf.html, Aug 2022.

[40] W. Gropp and E. Lusk, "User's Guide for MPICH, a Portable Implementation of MPI," 1996.

[41] C. Wang, J. Sun, M. Snir, K. Mohror, and E. Gonsiorowski, "Recorder 2.0: Efficient Parallel I/O Tracing and Analysis," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2020, pp. 1–8.

[42] X. Wu and F. Mueller, "Elastic and Scalable Tracing and Accurate Replay of Non-Deterministic Events," in *Proceedings of the 27th international ACM International Conference on upercomputing*, 2013, pp. 59–68.

[43] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel, "Introducing the Open Trace Format (OTF)," in *International Conference on Computational Science*. Springer, 2006, pp. 526–533.

[44] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf, "Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries," in *PARCO*, vol. 22, 2011, pp. 481–490.

[45] M. Wagner, J. Doleschal, and A. Knüpfer, "MPI-focused Tracing with OTFX: An MPI-aware In-Memory Event Tracing Extension to the Open Trace Format 2," in *Proceedings of the 22nd European MPI Users' Group Meeting*. ACM, 2015, p. 7.

[46] M. Wagner, A. Knupfer, and W. E. Nagel, "Enhanced Encoding Techniques for the Open Trace Format 2," *Procedia Computer Science*, vol. 9, pp. 1979–1987, 2012.

[47] Q. Xu, J. Subhlok, and N. Hammen, "Efficient Discovery of Loop Nests in Execution Traces," in *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 2010, pp. 193–202.

[48] S. Taheri, S. Devale, G. Gopalakrishnan, and M. Burtscher, "ParLOT: Efficient Whole-Program Call Tracing for HPC Applications," in *Programming and Performance Visualization Tools*. Springer, 2017, pp. 162–184.

[49] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

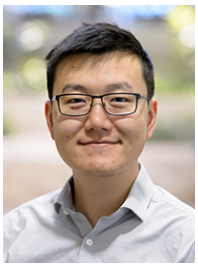
[50] A. Bahmani and F. Mueller, "Scalable Communication Event Tracing via Clustering," *Journal of Parallel and Distributed Computing*, vol. 109, pp. 230–244, 2017.

[51] ——, "Chameleon: Online Clustering of MPI Program Traces," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 1102–1112.

[52] F. Desprez, G. S. Markomanolis, M. Quinson, and F. Suter, "Assessing the Performance of MPI Applications Through Time-Independent Trace Replay," in *2011 40th International Conference on Parallel Processing Workshops*. IEEE, 2011, pp. 467–476.

[53] M.-A. Hermanns, M. Geimer, F. Wolf, and B. J. Wylie, "Verifying Causality between Distant Performance Phenomena in Large-Scale MPI Applications," in *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. IEEE, 2009, pp. 78–84.

Chen Wang is Fernbach Postdoctoral Fellow at Lawrence Livermore National Laboratory. He is currently working at the Center for Applied Scientific Computing at LLNL. He received his Ph.D in computer science from University of Illinois Urbana-Champaign. His research interests include parallel computing, I/O and communication tracing, and parallel storage systems.



Yanfei Guo is an Assistant Computer Scientist at Argonne National Laboratory. He is working in the Programming Models and Runtime Systems (PMRS) group and he is also part of the MPICH development team at ANL. His research interests lie in the fields of parallel and distributed systems, cloud computing systems and data-intensive computing.

Pavan Balaji is an Applied Research Scientist, Technical Lead, and Manager at Meta where he leads the HPC Network Communications and Early Industry Partnerships Group. Before joining Meta, he used to hold appointments as a Senior Computer Scientist and group lead at the Argonne National Laboratory and as an Institute Fellow of the Northwestern-Argonne Institute of Science and Engineering at Northwestern University

Marc Snir is Michael Faiman Emeritus Professor in the Department of Computer Science at the University of Illinois Urbana-Champaign. He was Director of the Mathematics and Computer Science Division at the Argonne National Laboratory from 2011 to 2016 and head of the Computer Science Department at Illinois from 2001 to 2007. Until 2001 he was a senior manager at the IBM T. J. Watson Research Center where he led the Scalable Parallel Systems research group that was responsible for major contributions to the IBM scalable parallel system and to the IBM Blue Gene system.