Pilgrim: Scalable and (near) Lossless MPI Tracing

Chen Wang
University of Illinois at Urbana-Champaign
Champaign, Illinois, USA
chenw5@illinois.edu

Pavan Balaji
Argonne National Laboratory
Lemont, IL, USA
Facebook Inc.
Menlo Park, CA, USA
pavanbalaji.work@gmail.com

Marc Snir
University of Illinois at Urbana-Champaign
Champaign, Illinois, USA
snir@illinois.edu

ABSTRACT

Traces of MPI communications are used by many performance analysis and visualization tools. Storing exhaustive traces of large scale MPI applications is infeasible, due to 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 incurs moderate overheads and generates small trace files at large scales, by using sophisticated compression techniques. Furthermore, for codes with regular communication patterns, Pilgrim can store their traces in constant space regardless of the problem size, the number of processors, and the number of iterations. In comparison with existing tools, Pilgrim preserves more information with less space in all the programs we tested.

CCS CONCEPTS

• **Theory of computation** → **Data compression**; **Pattern matching**; • **Computing methodologies** → *Massively parallel algorithms.*

KEYWORDS

Communication tracing, Lossless MPI tracing, Trace compression

ACM Reference Format:
Chen Wang, Pavan Balaji, and Marc Snir. 2021. Pilgrim: Scalable and (near) Lossless MPI Tracing. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3458817.3476151

1 INTRODUCTION

Traces of MPI communication calls in a parallel execution are essential to many HPC tools. They are used for performance analysis and communication visualization by tools such as Vampir [17] and Scalasca [11], for identifying errors in MPI codes [12], for guiding source code transformation [24], etc. Traces are also used to replay application communications, in order to study communication systems and guide the design of future machines [14, 30]. One can also use them to build performance prediction skeletons for estimating the performance of large applications [27, 28].

However, as systems become larger, trace sizes can get prohibitively large for applications running at large scales. For example, uncompressed traces of NAS parallel benchmarks [4] running on less than 1,000 processors can reach a few gigabytes [23]. To address this, some forms of compression are used by most trace collection tools [11, 18, 19, 23, 25, 37]. We shall mostly compare to Cypress [37] and ScalaTrace [35] as they are state-of-the-art tools that are closest to our work.

The compression schemes used by these systems 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 like AutoPerf [8], mpiP [31] and IPM [26]. These tools are extremely useful for discovering the characteristics across applications as they can be deployed system-widely thanks to the low overhead. The drawback is that they keep only limited information for each application, which is not enough when a more detailed analysis is required. With tracing, the overhead is normally higher, but once the communication traces have been generated, they can be used repeatedly in various post-processing tasks and analyses.

Second, basically no tracing system collects all the information available when MPI calls are intercepted. They ignore some MPI functions and some of the parameters of the traced functions. This improves the compression rate but reduces the usefulness of the traces. Consider the following code:

```
// requests: MPI_Request array of length incount
// incount: assume is the same for all processes
do {
    ...
    MPI_Testsome(incount, requests, outcount,
                 indices, statuses);
    ...
} while(!(all requests were finished));
```

This is not an uncommon pattern where each process loops over a list of requests and handles each completed request. If we keep only the first parameter (`incount`) of this call, then each MPI call on each process will generate an identical trace record and the records will compress well. However, the code is non-deterministic as messages may be received in a different order at each process and at each iteration of this code. In order to know the actual execution order, one needs to handle the remaining arguments and preserve enough information in the compressed trace so that each non-blocking communication can be matched with the test call that completed it. Neither ScalaTrace nor Cypress records `MPI_Testxxx` calls, so the traces they record cannot be used to replay communications in proper order in this example.

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476151

Functions Supported	Cypress	ScalaTrace	Pilgrim
Total: 446	56	125	446

Popular 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 [3] (excluding `MPI_Wtime` and `MPI_Wtick`).

We present in this paper the *Pilgrim*[1] 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. The paper makes the following contributions:

(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, Cypress and ScalaTrace. The information about Cypress and ScalaTrace were retrieved by manually examining their source code, so small inaccuracies may exist.
(2) All MPI objects (e.g., `MPI_Request`, `MPI_Comm`, etc) and memory buffers are encoded in a way that preserves the original information. Pilgrim also intercepts memory allocation operations to match pointers used in MPI calls.
(3) We address the corner cases that are normally ignored by existing work, e.g., non-blocking communicator creation, intercommunicators, `MPI_Testxxx` calls, etc.
(4) We incrementally construct a context-free-grammar and a call signature table for each process to store all information. Optimizations including relative rank encoding and loop detection are proposed to further improve the compression ratio.
(5) We demonstrate the scalability and effectiveness of Pilgrim using a variety of codes. *We show that while we preserve more information, we also achieve a smaller trace size.*
(6) Pilgrim does not require access to the application source code or linking to a special library.

The rest of the paper is organized as follows: Section 2 provides an overview of Pilgrim's design. Section 3 describes the implementation details, along with the proposed optimizations. Next, in Section 4, we evaluate Pilgrim and compare the results with ScalaTrace. The related work is given in Section 5. Finally, we conclude in Section 6.

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

2 OVERVIEW

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 memory footprint during runtime and the disk space required for storing the traces after the application run. (2) Time overhead due to the tracing and compression procedure.

Compression can occur at two points in time: *online compression* that is performed as traces are collected, and *offline compression* that occurs after all traces were collected. The "offline" compression can be executed in parallel when MPI is finalized, thus reducing I/O. Online compression usually is *intra-process*, compressing the trace file generated for one process, while offline compression usually is *inter-process*, 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 6). Fortunately, most codes exhibit recurring communication patterns to some extent. Good trace compression can be achieved, if we recognize and compress as many recurring patterns as possible. We achieve this by representing the traces using a context-free-grammar (CFG) and a call signature table (CST) as the storage format. Next, we will 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 *non-terminal* symbol S. A *context-free grammar* (CFG) is a formal grammar whose production rules are of the form $A \rightarrow \alpha$, where A is a single non-terminal symbol and α is a string of terminal and/or non-terminal symbols. The grammar generates a unique string if there is exactly one rewriting rule for each non-terminal and the grammar is acyclic. (A grammar is cyclic if there is a sequence of derivations so that $A \rightarrow \ldots \rightarrow \alpha A \beta$, for some non-terminal A; it is acyclic otherwise.)

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: $S \rightarrow A_1A_1$, $A_1 \rightarrow A_2A_2$, $\ldots$, $A_k \rightarrow a$. In the best case, a CFG can represent a string of N characters in $O(logN)$ space, whereas in the worst case (e.g., a random string) it requires $O(N)$ space. Conceptually speaking, building the CFG is the process of compressing the string and repeated rule application is the process of decompressing the string.

Pilgrim builds online one CFG for each process to compress and store the sequence of MPI calls made by that process. This sequence can be considered as a string of terminal symbols, where each terminal represents a unique MPI call and the values of its input or output parameters. In order to efficiently map from a call to a terminal symbol, Pilgrim maintains a *call signature table* (CST) on each process. A *call signature* is composed of a function id (every MPI function has a unique id) and the values of its parameters. Parameters that are handles to opaque MPI objects are encoded

```
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

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

CFG of Rank 1

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

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

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

symbolically (the encoding is described later). Figure 1 shows a simple code snippet and the produced CFG and CST when running with two MPI ranks.

The CFG and CST together enable efficient intra-process compression. And since both of them are maintained independently for each process, there is no message exchange or communication overhead until the inter-process compression, except for calls creating new communicators and similar global objects.. Locally, they guarantee that duplicated calls by the same process will be stored only once and the recurring patterns will be compressed by grammar rules. Globally, calls and grammar rules across processes will be merged during the inter-process compression, which will be discussed in Section 3.5.

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 terminal symbol or create a new entry if it is its first occurrence. Next, the current grammar is modified in order to handle the new terminal symbol.

The algorithm we used to build the CFG is the well-known Sequitur [22] algorithm. It is a good fit for Pilgrim for two reasons: (1) It is an incremental algorithm that can grow the CFG at runtime, i.e., one MPI call at a time. (2) It has a linear time complexity in terms of the number of symbols, which is important because the number of MPI calls tends to be large.

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 non-terminal appears more than once on the right-hand side of a production.

The algorithm operates by enforcing the two constraints on the grammar: when property P1 is violated, a new production is formed. Thus, 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. Thus, if we have productions $A \rightarrow Bc$, $B \rightarrow ab$, and B appears in no other production, then the first production is replaced by $A \rightarrow abc$ and the second one is deleted. With these two constraints, a loop of N identical iterations will be compressed to a $O(logN)$ size grammar. A more compact grammar is obtained by adding to the notation repetition counts, i.e. productions of the form $A \rightarrow B^k$, and replacing each production of the form $A \rightarrow B^iB^j$ by the production $A \rightarrow B^{i+j}$ [9]. This optimization reduces space complexity for regular loops from $O(logN)$ to $O(1)$. (Strictly speaking, we replace a logarithmic number of productions by counters with a logarithmic number of bits – we replace recursion with iteration.)

Pilgrim uses this optimized version of the Sequitur algorithm to build each local CFG. We do not describe the implementation of the Sequitur algorithm and refer readers to [9, 22] for details.

3 IMPLEMENTATION

Figure 2 depicts the whole tracing and compression process, which has the following steps: (1) Intercept each MPI call; (2) Store the timing information; (3) Encode parameters and compose the call signature; (4) Update the CST; (4) Use Sequitur algorithm to grow the CFG; and, finally, (6) Inter-process compression.

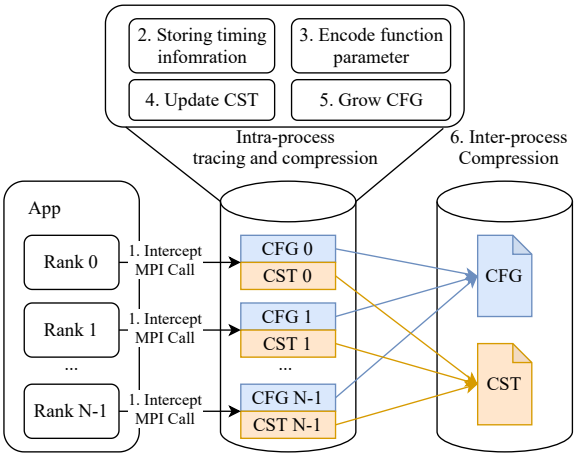

Figure 2: Tracing and compression process of Pilgrim

The intra-process compression, which is done separately for each process, consists of steps (2-5). Steps (4) and (5) have already been discussed in the previous section. In this section, we will describe the remaining steps in detail.

3.1 Intercepting MPI Calls

Pilgrim uses the MPI profiling interface (PMPI) to trap and trace MPI calls. In order to guarantee completeness, the wrappers were automatically generated from the MPI standard documents (Latex files). The reason we use Latex files instead of MPI header files as input is that we need to know the direction of each function parameter (i.e., input, output, or both) and this information is normally not presented in header files. To be specific, we used MPI 4.0 RC [3] to generate the wrappers. Before compiling Pilgrim, a filtering pass is done automatically to remove the calls that are not supported by the local MPI implementation. This step is required because the

targeted MPI can be an older version that does not support recently added MPI calls.

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

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

We need to store the value of each input parameter in the prologue and the value of each output parameter in the epilogue. This is one reason why we need both. The other is to compute the duration of the MPI call. The prologue code records the starting timestamp and starts a timer. All the remaining steps (3, 4, and 5) are performed in the epilogue code.

3.2 Compressing Timing Information

Pilgrim by default keeps only statistical timing information for each call signature. In the CST, we keep the average for calls' duration. This adds negligible overhead and does not change the patterns in the grammar since the information is associated with each entry in the CST.

However, if more information is required for the analysis, e.g., to study the skews in collective call invocations, Pilgrim is also able to store non-aggregated timing information. Lossless compression of time values does not save much space so, instead, we provide a lossy compression with a user tunable relative error. In this mode, Pilgrim keeps the *duration* and *interval* for each MPI call, where duration is the elapsed time of the call, and interval is measured between two calls with an identical call signature. In post-processing, we can use the duration and interval to infer the entry time (noted as *tstart*) and the exit time (noted as *tend*) for every MPI call. Timestamps are most useful when the clocks of distinct processors are well synchronized. This can be achieved by synchronization techniques that take advantage of MPI [13, 15], or of hardware support [21].

We choose to keep the duration and interval instead of *tstart* and *tend* because they have relatively smaller values and are easier to compress. Here are some observations that enable the compression for the duration and interval especially for calls in a loop:

(1) Functions with the same call signature should have similar durations. But network congestion, system noise and other irregularities introduce variations.
(2) Functions with the same call signature should have similar intervals if they occur in a loop and each iteration of the loop takes the same time to execute. Again, variations exist and irregular codes might show even larger divergences.

We bin durations using exponential bins. A duration of d will be represented by $\hat{d} = \lceil \log_b d \rceil$. The relative error in duration will be at most $b-1$. The base b can be specified by users on a per-function basis. Intervals are handled as follows: Assume that a call occurs at time t and the previously stored interval representations for calls with the same signature are $\hat{i}_1, \ldots, \hat{i}_k$. Then the new adjusted interval is $i_{k+1} = t - \sum_{j=1}^{k} b^{\hat{i}_j}$; this interval will be encoded as $\hat{i}_{k+1} = \lceil \log_b i_{k+1} \rceil$. This scheme ensures that the wall clock time for each call will be recovered with a relative error of at most $b-1$.

Based on the observations above, the sequences of durations and of intervals should also exhibit some recurring patterns just like MPI calls. Therefore, we use the Sequitur algorithm again to build two separate grammars (one for each sequence) to compress them.

3.3 Encoding Function Parameters

The objective of Pilgrim is to keep as much information as possible. One challenge is that, for many function parameters, their values obtained at execution time are not significant and are hard to compress. For example, in the `MPI_Send()` call, there is a `void* buf` parameter that points to the memory buffer that the caller wants to send. However, in most cases, the absolute address (or the actual content) of the memory buffer provides little information yet requires many bytes to store.

A similar point can be made about all MPI opaque objects: One wants to know what the object represents, not the value of the pointer or the integer that references it. For this, one needs to be able to associate calls that created the object with calls that use the object. To achieve this, Pilgrim uses locally unique symbolic representations for all MPI objects and memory pointers, so that later we can compare and match them across different calls. 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. An MPI communicator created by `MPI_Comm_split()` and used later in `MPI_Send()` will have the same symbolic id in both calls. This allows recreating the communicator's group and the rank of each group member. As for all other basic type parameters (e.g., numeric values and strings), we simply store their values.

Here, we first describe the symbolic representation algorithm for MPI objects and leave memory pointers in Section 3.3.3 as they require some additional work.

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 like `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 as the application either reuses the same objects or frees the old objects before allocating more.

Other than being able to compare symbolic ids in different calls, another advantage of this design is that 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 inter-process compression.

In the rest of this section, we will describe some of the object types that require special treatment. Pilgrim is designed to handle all MPI calls; due to the space limit, we only cover the implementation of some the more difficult cases.

3.3.1 MPI_Comm. 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 at the post-mortem phase. The algorithm for this follows three steps:

(1) Every process in the group of the new communicator checks for the *maximum symbolic id* locally assigned to a communicator.
(2) An all-reduce operation is used to get the group-wide maximum symbolic id.
(3) Every process in the group uses one plus the maximum id retrieved from the last step as the symbolic id for the newly created communicator.

The group-wide maximum is required so as to avoid assigning the same id to different communicators at the same process.

The above algorithm only works for *blocking intra-communicator* creation calls such as `MPI_Comm_split()`. Inter-communicators are handled differently as we are not able to perform all-reduce on an inter-communicator. The solution is to create a temporary intra-communicator by merging the inter-communicators and then use the same algorithm mentioned above. The non-blocking communicator creation calls such as `MPI_Comm_idup()` are even trickier, as we can not issue a blocking all-reduce within a non-blocking call. Instead, we have to use a non-blocking all-reduce call and keep track of the MPI request generated from it. Later, when we intercepted `MPI_Waitxxx()` or `MPI_Testxxx()` calls we check the completed requests to see if the symbolic id has been received.

3.3.2 MPI_Status. The `MPI_Status` structure includes five fields as shown below.

```
struct MPI_Status {
  int count;      // Number of received entries
  int cancelled;  // if the request was cancelled
  int MPI_SOURCE; // Source of the msg
  int MPI_TAG;    // Tag value of the msg
  int MPI_ERROR;  // Error associated with the msg
}
```

The fields of this structure will be filled after the return of the MPI calls, unless the input argument is `MPI_STATUS_IGNORE`. Pilgrim keeps only two fields of it: `MPI_SOURCE` and `MPI_TAG` as they are important for matching communication calls. The fields `count` and `cancelled` can be inferred during post-processing and `MPI_ERROR` in most times is just zero so we ignore them for current implementation.

3.3.3 Memory Pointers. We assign symbolic ids to MPI objects so that the call creating such an object can be matched with the calls using it. In order to achieve the same goal for memory pointers, we intercept memory management calls including `malloc`, `calloc`, `realloc`, and `free`. We also intercept CUDA memory allocation calls such as `cudaMalloc`, `cudaMallocManaged` and `cudaHostAlloc`, and keep track of the device location of the allocated memory. We use an AVL tree to keep track of the currently allocated memory segments. Each node represents a segment and nodes are sorted according to the starting address of the segment. In addition, each node stores the segment's size and its assigned symbolic id. 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. For each communication buffer, we keep the symbolic id of the containing segment, and optionally, the device location and the displacement from the segment's start.

This 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 were accessed, and the allocated size is assumed to be one byte to be conservative.

3.4 Optimizations

In this section, we discuss some important optimizations applied by Pilgrim. These optimizations are mostly needed for inter-process compression.

3.4.1 Common Communication Patterns. One side benefit of the CST and CFG based compression method is it compresses well for common communication patterns. For example, regular and static communication patterns, whether point-to-point or collective calls will have the identical call signatures across loop iterations, thus they will only be stored once, due to intra-process compression. Another example is provided by applications that use symmetric collective communications, such as `MPI_Allgather` or `MPI_Allreduce`, where all the calling processes pass the same argument values. The call signatures will be stored only once, due to inter-process compression.

The next section discusses additional optimizations that target regular stencil communication patterns. We leave to future work other important special cases.

3.4.2 Relative Ranks. In point-to-point calls such as `MPI_Send()` and `MPI_Recv()`, parameters `src` and `dst` are used to specify the source and destination rank. It is a common pattern that one rank sends and receives messages to and from its neighbors, e.g., in regular stencil codes. For example, consider the pseudocode below, which shows a simple 1-D 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 across ranks because `src` and `dst` are different on different ranks. A run with N ranks will produce $2N$ call signatures, which is bad for inter-process compression.

The solution for this issue is straightforward: Instead of keeping the actual values of `src` and `dst`, we keep the relative value based on the caller's rank. In this way, the above code will produce only two unique call signatures regardless of the number of ranks. Note that this simple method also works for regular multi-dimensional communication patterns (Section 4.1).

Moreover, this encoding schema works not only for source and destination, but also for the parameters that are possibly rank-related (e.g., tag in communication calls and color and key in communicator creation calls).

3.4.3 Id of MPI_Request Objects. For most MPI objects, the order of creation, usage, and finalization are deterministic. The `MPI_Request` objects, however, can exhibit randomness because the order of non-blocking communications completion can be non-deterministic. Requests may be freed in different order at distinct iterations and, therefore, the allocation of symbolic ids can differ, impairing compression.

Consider the example below, every `MPI_Request` object should be assigned a free id from the pool of free ids as mentioned in Section 3.3. Assume the pool is initially empty, then the order of ids assigned within each iteration totally depends on the completion order of the calls, which is non-deterministic. As a result, the code is very 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_Irecv(to = my_rank + 3, &req3);
    while(!(all requests finished)) {
        MPI_Waitany([req1, req2, req3]);
        handle received message;
    }
}
```

The root cause for this is that our algorithm keeps one pool of free ids for the same object type. To address this issue, for `MPI_Request`, we maintain a separate pool for each unique call signature, not including the request in the 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.

3.5 Inter-process Compression

Until now, we have discussed how does Pilgrim encode and compress function and function parameters. In the end, each process will produce its own CST and CFG, which contain all the information required to recover the original MPI calls for this one process.

An HPC application can run on millions of processes and will result in huge trace files if we keep the CST and CFG separately for each process. Inter-process compression is, therefore, critical to avoid this linear increase in trace size. The assumption that the inter-process compression is based on is that many processes tend to call MPI functions in the same order and with the same or similar parameters. Thanks to the symbolic representations and the optimizations described earlier in this section, we should be able to compress the CST and CFG across processes.

3.5.1 CST. The total number of function calls will increase linearly with the number of processes, with weak scaling. But the number of unique call signatures may grow more slowly if calls have the same signature on different processes, as it turns out to be the case for many codes. We leverage this redundancy by merging all CSTs and keeping only globally unique call signatures. We use 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 so as to use the new symbols assigned to call signatures.

Figure 3 shows an example of this process for two MPI ranks. Each rank has only two entries in their CST and one of them has the same call signature. After the merging process, the merged CST will contain three call signatures and the last one `MPI_Comm(comm3)` will be given a new terminal symbol (3) as its old terminal symbol (2) has already been used. If rank 0 performs this merging task, then it will send back the merged CST to rank 1 so rank 1 can update its grammar accordingly.

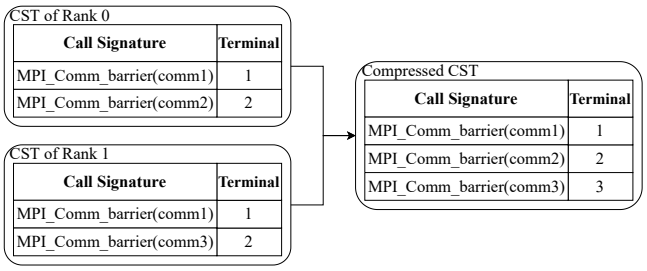

Figure 3: Example of inter-process compression for CSTs.

3.5.2 CFG. Grammars are also expected to have redundancies because, in scientific codes, different processes tend to execute the same code blocks (thus the same sequence of MPI calls) but with different data. The symbolic representation we use for pointers often allocates the same symbol to corresponding buffers at different processes, resulting in identical signatures.

The algorithm for inter-process compression of grammars is similar to the one used for compressing the CSTs: Pairwise merges are executed in parallel until all grammars are merged. 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 non-terminal of the two grammars, so as to concatenate the lists of the two processes. The names of non-terminals are changed in order to prevent conflicts. The merged grammar encodes the concatenation of the process traces.

Before we merge two grammars, we first check if the two grammars are identical. If they are identical, the merge is more efficient as we only keep one of them and do not rename rules. Once all grammars have been merged, we run another Sequitur pass to compress the merged grammar. The identity check is important because it reduces merged grammar size which can significantly reduce the time required for the final Sequitur pass. This check can be done very quickly using a memory comparison operation as our grammar is stored internally as an array of integers. We will show later in Section 4 that in many programs most processes produce identical grammars. We use a simple example (Figure 4) to illustrate this process. The grammars merged in the first pass are identical so we only need to concatenate the two starting rules. At the second pass, the two merged grammars are not identical. Thus, each rule needs to be checked and updated separately to solve conflicts (e.g., X from rank 2 is renamed to Z). Finally, a Sequitur pass is run, which compresses the merged grammar.

As for the decompression, it is simply a process of recursive rule application. If the leftmost non-terminal 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.

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 iterations and the number of processes? (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

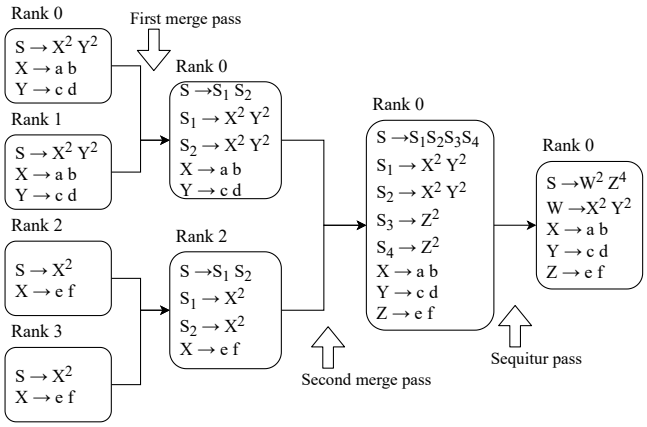

Figure 4: Example of inter-process compression for grammars.

Type	Code
Benchmark	2D and 3D Stencils
	OSU Micro-Benchmarks [5]
Mini app	NAS Parallel Benchmark [4]
Production app	FLASH [2] and MILC [1]

Table 2: Codes used for evaluation

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. As we developed both compressor and decompressor, we can check correctness by comparing uncompressed traces to compressed next decompressed traces.

We selected a variety of codes for the evaluation, as shown in Table 2. The experiments were conducted on two clusters: Catalyst at LLNL and Theta at ANL. The hardware specifications are given in Table 3. Theta was used to run MILC, all other experiments were run on Catalyst.

Spec	Catalyst	Theta
Compute Node	Intel Xeon E5-2695	Intel KNL 7230
Cores	24	64
Memory	128GB DDR4	192GB DDR4
Network	IB QDR	Aries Dragonfly

Table 3: Hardware specification

4.1 Benchmarks

Thanks to the relative rank optimization described in Section 3.4, 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.

The stencil codes we tested were a 2D 5-points non-periodical boundary stencil and a 3D 7-points stencil with periodical boundaries. They both use `MPI_Isend()`, `MPI_Irecv()` and `MPI_Waitall()` for communication. The meshes are distributed using a block distribution on a mesh of processes of the same number of dimensions. Therefore, 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 OSU Micro-benchmarks except osu_latency_mt as Pilgrim currently does not support multi-threaded MPI programs. Again, Pilgrim can compress perfectly across processes and iterations for all programs included in OSU Micro-benchmarks. Most programs result in a trace file size of a few kilobytes. We do not include the results here due to the page limit.

4.2 Pilgrim vs. ScalaTrace

Here, we compare the scalability of Pilgrim with ScalaTrace. 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 C NAS parallel benchmarks (NPB) and ran them with increasing numbers of processes (SP and BT require a square number of processes to run on).

Trace sizes are shown in Figure 5 for different NAS benchmarks and different process counts. For all tested benchmarks, Pilgrim achieves a smaller trace file size while keeping more information. In addition, Pilgrim's trace size scales better than ScalaTrace's in the number of processes. ScalaTrace experienced a linear growth in all cases except LU whereas Pilgrim only exhibited sub-linear growth rates. The advantage comes from the CST and CFG based intra-process and inter-process compression. Although more communication patterns occurred as we increase the number of processes, not all of them are totally new. LU is special in that it has a behaviour similar to the stencil codes we discussed previously. For Pilgrim, the trace size stopped increasing after 16 processes, which suggests that all patterns had occurred at 16 processes.

4.3 Scientific Applications

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

We first tested three simulations that come with the FLASH package, namely Sedov, Cellular, and StirTurb. All are 3D simulations with I/O disabled. In order to add more variances into the simulations, we disabled the adaptive mesh refinement (AMR) feature for Sedov and StirTurb and kept it for Cellular. Sedov and Cellular crashed at the `MPI_Waitall()` call when running with ScalaTrace. So we had to comment out the wrapper for `MPI_Waitall()` in ScalaTrace to run these two simulations. In other words, `MPI_Waitall()` calls were not traced by ScalaTrace for Sedov and Cellular.

Figure 6 shows how the trace size scales with the number of processes and the number of iterations. We also plotted on the secondary y-axis the total number of MPI calls Pilgrim encountered. In comparison with ScalaTrace, Pilgrim produced smaller traces and showed better scalability in all cases. Figure 6 (a-c) suggests that

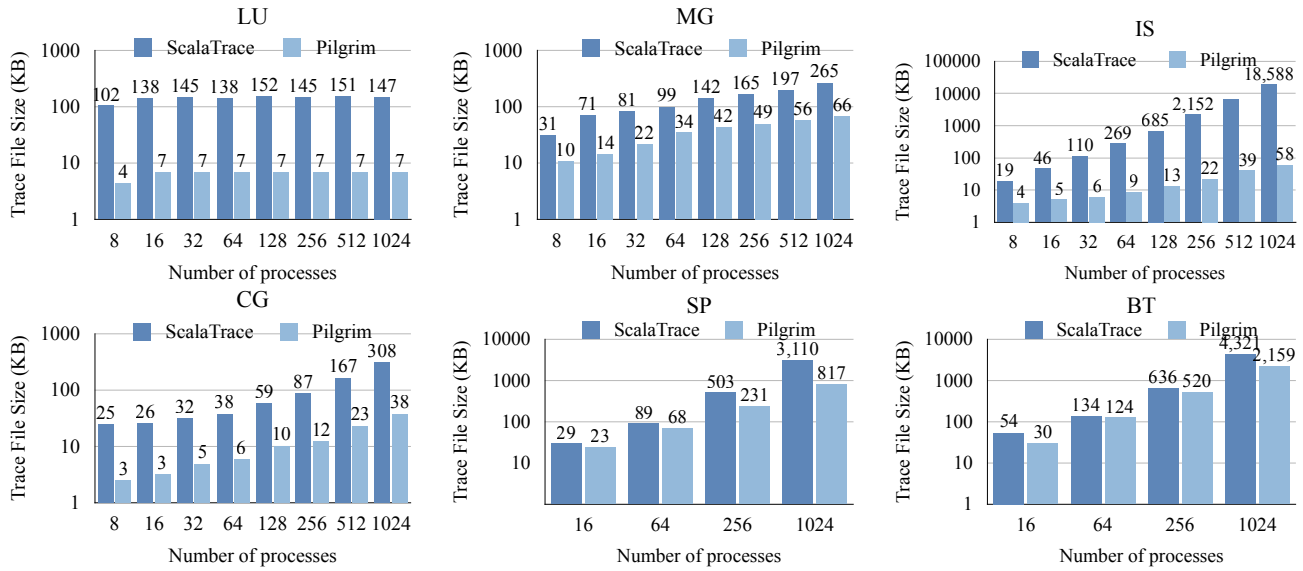

Figure 5: Comparison of trace file size with NPB

the trace size of ScalaTrace growths in proportion to the number of MPI calls. In comparison, the trace size of Pilgrim stopped growing at 64 processors for StirTurb, 128 processors for Sedov, and 1,024 processors for Cellular. Presumably, all signatures were encountered with these numbers of processors. Figure 6 (e-f) shows the results of running the same simulations using 4K processors but with an increasing number of iterations. As expected, the number of MPI calls increased linearly in the number of iterations. StirTurb without the AMR produced constant size trace files for both tools. But for Sedov and Cellular, the more iterations we ran the larger the trace size. However, the causes are different. Cellular internally uses the PARAMESH library to perform parallel AMR. It builds a hierarchy of sub-grids to form the compute domain. These sub-grid blocks are stored 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 in order to rebalance the load. The communication is performed using point-to-point calls: Isend, Irecv, and Waitall. The communication pattern changes at each each refinement. And the trace size grows as more refinements occur. The trace size of Sedov grows slowly once a few hundred iterations due to some extra `MPI_Send/MPI_Recv` with new sources and destinations being called. 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. It is worth noting that Pilgrim can store complete traces from the hundreds of millions of MPI calls generated by a multi-minute run with 4K processes in 2.5MB for Cellular and just 4KB for StirTurb without AMR.

Figure 7 shows the execution time of the same FLASH simulations with and without tracing. The problem size per processor was kept fixed while increasing the number of processors (weak scaling). This set of experiments were designed to evaluate the overhead of Pilgrim. Pilgrim's overhead comes from two sources: intra-process compression overhead due to the construction of CFG and CST; and the inter-process compression at the finalize point. The first part scales well because the intra-process compression is totally independent across processors. For the latter part, compressing CSTs normally takes negligible time, while the compression for CFGs dominates and is largely due to the sequential final Sequitur pass. And this overhead depends on the number of unique grammars. Fortunately, in most cases there are only a few unique grammars.

ScalaTrace achieved comparable overhead for StirTurb but a much larger overhead for the other two simulations. It took several times longer to run Sedov and Cellular with ScalaTrace at large scales. We observed that this slow down of ScalaTrace mostly happened at the refinement time when a burst of non-blocking calls occurred. On the other hand, Pilgrim incurred a maximum overhead of 21%, 29%, and 4% for Sedov, Cellular, and StirTurb. The largest numbers of unique grammars were 74, 498, and 2 respectively. Figure 8 shows Pilgrim's overhead components. As we can see, the inter-process compression for CSTs took up only a tiny fraction of the total time. As expected, the more unique grammars there were, the more time was spent on inter-process compression.

Finally, we ran a MILC simulation using up to 16K processors. The simulation code was taken from MILC's source tree (clover_dynamical). The algorithm we used is *su3_rmd* (refreshed molecular dynamics algorithm). Both strong scaling and weak scaling were performed. For the strong scaling, the problem size was set to $64 \times 64 \times 64 \times 32$ and for the weak scaling the problem size per processor was fixed at $16 \times 16 \times 16 \times 32$. The simulation crashed when running with ScalaTrace, so in Figure 9 we only show the results for Pilgrim. One interesting finding is that the trace size did not change for weak scaling, where we observed 27 unique grammars regardless of the process count. The simulation took around 10 minutes to finish when running on 16K processors for weak scaling and the produced trace file was only 627KB. For strong

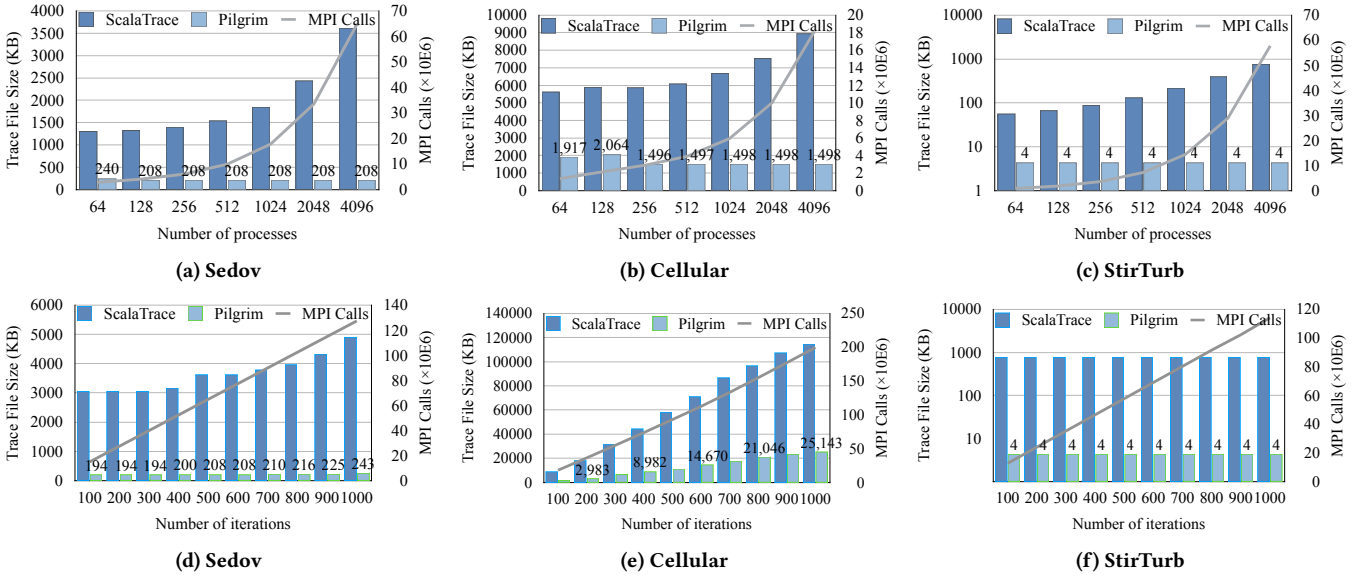

Figure 6: Comparison of trace file size with FLASH simulations. (a-c) show the scalability with respect of the number of processes; (d-f) show how the trace size scales with the number of iterations.

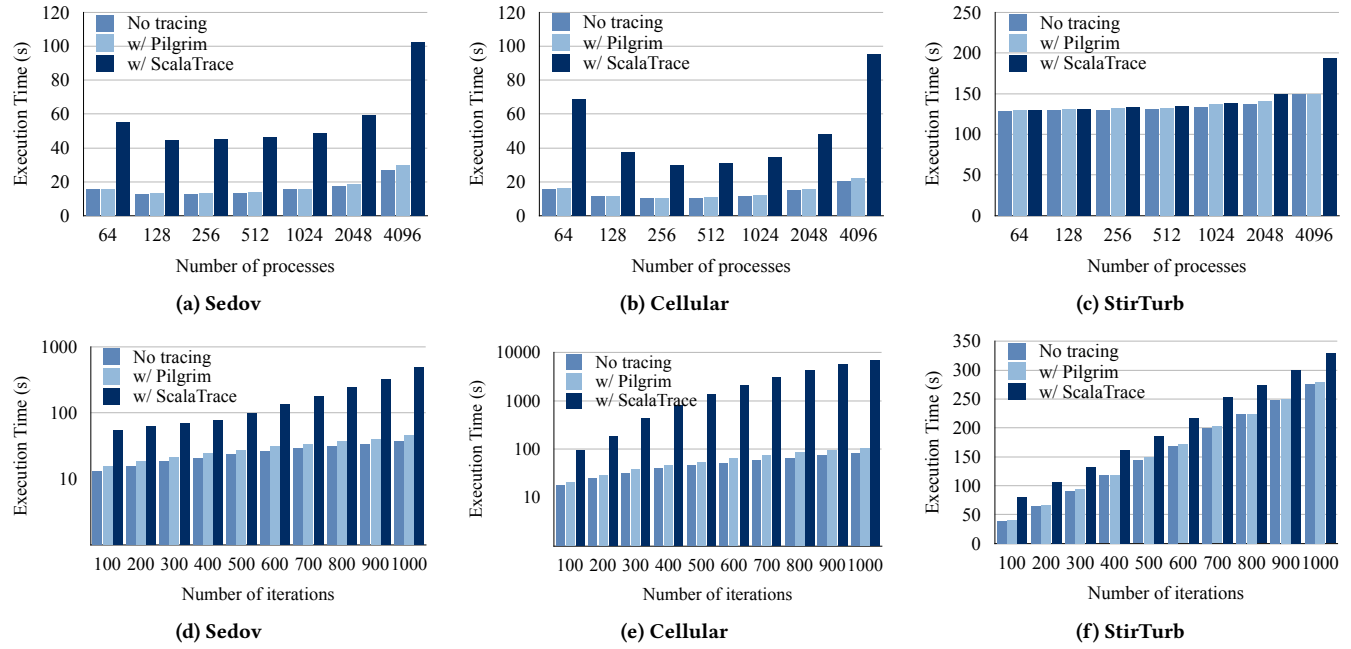

Figure 7: Execution time of FLASH simulations. Note that in (d) and (e) the execution times are plotted in logarithmic scale due to the large range.

scaling, we observed three "stages": 27 unique grammars at 64 processors, 54 unique grammars for runs with 128 to 1,024 processors, and 108 grammars for 2,048 processors and more. Accordingly, the more unique grammars were observed, the larger the trace file was generated. Even though, with 16K processors, the trace file was only about 1MB.

4.4 Non-aggregated Timing Information

As described in Section 3.2, Pilgrim has the ability to store non-aggregated but lossy timing information. However, unlike the MPI call sequence, the timing information is more difficult to compress

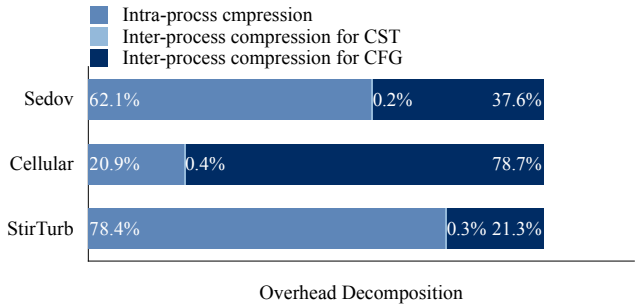

Figure 8: Pilgrim's overhead components for FLASH simulations

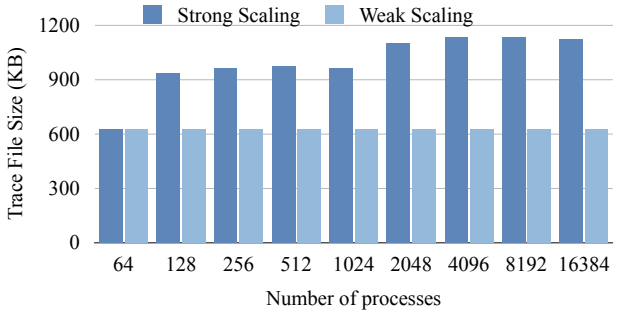

Figure 9: MILC trace size vs. pcos.

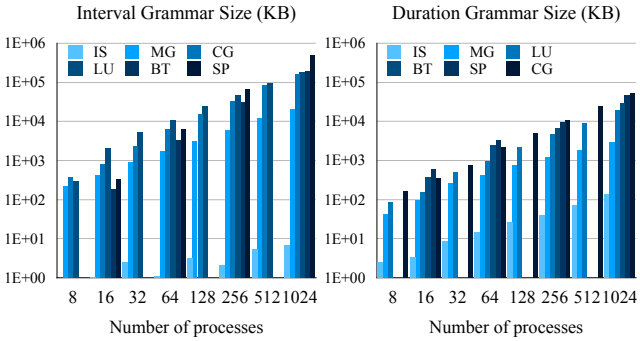

Figure 10: Space required for storing timing information for NPB

due to the intrinsic non-determinism. A trade-off between accuracy and overhead has to be made when storing the timing information.

We evaluate our algorithm for storing timing information using the NAS benchmarks with up to 1,024 processors. The relative error was set to 20%, i.e., $b = 1.2$. Figure 10 shows the produced interval grammar size and duration grammar size. We observed a linear or near linear growth rate in all cases, which suggests that the inter-process compression for timing grammars is not as effective as for MPI calls. Pilgrim required 486MB for the interval grammar and 50MB for the duration grammar in the worst cases (SP and CG at 1,024 processors). In those cases, Pilgrim traced 234 million and 98 million MPI calls from SP and CG, which yields a compression ratio of 3.8× and 15.68×, respectively.

5 RELATED WORK

Significant research efforts have been made in the area of communication profiling and tracing. Many tools have been developed during the years: profiling tools such as AutoPerf [8], mpiP [31], and IPM [26], and tracing tools like Recorder [34], Vampir [17], TAU [25], Score-P [18], ScalaTrace [35], and Cypress [37]. In this section, we focus on tracing tools as they are more related to our work.

Several tools including Score-p, Vampir and TAU support a tracing format named OTF [16] or some optimized versions of it, e.g., OTF2 [10], OTFX [32] and [33]. OTF is a rather general format in that it is not limited only to communication events. It uses ZLIB compression to reduce the trace size but the inter-process compression is not supported. Tools based on it generally lack structure-aware compression, which reduces the compression rate.

Recorder [34] traces both communication and I/O events and it uses a sliding window based approach to compress similar events within the window. But it cant not detect loop structures nor repetitions at long ranges. Xu et al. [36] introduced a framework for identifying the maximal loop nest. Their algorithm can also discover long range repeating communication patterns. However, both of these two tools do not perform inter-process compression, which is essential at large scales.

ParLOT [29] is a whole program tracing library built on top of Pin [20] that traces all function calls (but not their arguments). It performs incremental online compression so that each process will never store uncompressed information. The compression is achieved using two general purpose compression algorithms, which may not take advantage of the loop structures.

Similar to Pilgrim, Krishnamoorthy et al. [19] proposed a 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 helps the compression rate as calls with different signatures can be combined, if they differ in the ignored parameters; but this discards important information. In addition, the inter-process compression does not fully exploit the possible redundancy between grammars. It merges rules from multiple grammars without the redundancy check, which can lead to a high overhead for regular SPMD programs.

ScalaTrace [23] mainly focuses on recording communication events and features on-the-fly compression. It extends regular section descriptors (RSD) 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 [35] addresses this limitation of its predecessor. The intra-node and inter-node loop detection algorithms was redesigned to improve the compression effectiveness for scientific codes that demonstrate inconsistent behavior across time steps and processes. More recently, Bahmani and Mueller [6, 7] proposed a signature based clustering algorithm and context-aware clustering algorithm for ScalaTrace II to further improve the inter-process compression. However, they require *markers* to be inserted into the user code so as 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 inter-process compression at finalizing point. In comparison, Cypress [37] took 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 intra-process compression at runtime. The static pass relies on compiler analysis and normally is more efficient and accurate than online loop detection. The key limitation of Cypress however is that it requires the user's code to be first converted into the format of LLVM IR. Also many functions are not recorded or compressed by Cypress (Table 1), including some popular ones like 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 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 a complete set of MPI functions and all involved parameters. As far as we know, none of the existing work achieves this. They either miss some functions or keep only part of the parameters. And many corner cases are ignored to simplify the implementation or to achieve a higher compression ratio.

6 CONCLUSION

In this paper, we presented a scalable and near lossless MPI tracing tool. In comparison 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. The evaluation showed that Pilgrim preserves more information with less space and lower overhead.

One limitation of the current implementation is that it does not support multi-threaded MPI programs that are initialized with `MPI_THREAD_MULTIPLE`. Moreover, since Pilgrim uses its own trace format, existing post-processing tools can not be used directly with Pilgrim traces. We have developed a decoder that decompresses and decodes the traces into original uncompressed trace records. For the post-processing tools, we are currently working on a mini-app generator that could automatically generate a proxy MPI program that has the same communication patterns as of captured in the traces. It can be helpful for reproducing communication patterns that have performance issues. Another direction is to develop a converter that converts Pilgrim traces into some existing trace formats (e.g., OTF).

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.

We thank Dr. Yanfei Guo for his gracious help.

REFERENCES

[1] 2016. MILC Code Version 7. http://www.physics.utah.edu/~detar/milc/milc_qcd.html.
[2] 2019. Flash Center for Computational Science. http://flash.uchicago.edu.
[3] 2020. MPI: A Message-Passing Interface Standard Version 4.0 (Draft). https://www.mpi-forum.org/docs/drafts/mpi-2020-draft-report.pdf.
[4] 2020. NAS Parallel Benchmarks. https://www.nas.nasa.gov/publications/npb.html.
[5] 2020. OSU Micro-Benchmarks 5.7. http://mvapich.cse.ohio-state.edu/benchmarks.
[6] Amir Bahmani and Frank Mueller. 2017. Scalable Communication Event Tracing via Clustering. *J. Parallel and Distrib. Comput.* 109 (2017), 230–244.
[7] Amir Bahmani and Frank Mueller. 2018. Chameleon: Online Clustering of MPI Program Traces. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1102–1112.
[8] Sudheer Chunduri, Scott Parker, Pavan Balaji, Kevin Harms, and Kalyan Kumaran. 2018. Characterization of MPI Usage on a Production Supercomputer. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 386–400.
[9] Matthieu Dorier, Shadi Ibrahim, Gabriel Antoniu, and Rob Ross. 2015. Using formal grammars to predict I/O behaviors in HPC: The omnisc'IO approach. *IEEE Transactions on Parallel and Distributed Systems* 27, 8 (2015), 2435–2449.
[10] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E Nagel, and Felix Wolf. 2011. Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries. In *PARCO*, Vol. 22. 481–490.
[11] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. 2010. The Scalasca Performance Toolset Architecture. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 702–719.
[12] Tobias Hilbrich, Martin Schulz, Bronis R de Supinski, and Matthias S Müller. 2010. MUST: A Scalable Approach to Runtime Error Detection in MPI Programs. In *Tools for high performance computing 2009*. Springer, 53–66.
[13] Sascha Hunold and Alexandra Carpen-Amarie. 2018. Hierarchical Clock Synchronization in MPI.. In *CLUSTER*. 325–336.
[14] Nikhil Jain, Abhinav Bhatele, Sam White, Todd Gamblin, and Laxmikant V Kale. 2016. 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, 154–165.
[15] Terry Jones and Gregory A Koenig. 2010. 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, 207–214.
[16] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E Nagel. 2006. Introducing the Open Trace Format (OTF). In *International Conference on Computational Science*. Springer, 526–533.
[17] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E Nagel. 2008. The Vampir Performance Analysis Tool-Set. In *Tools for high performance computing*. Springer, 139–155.
[18] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, et al. 2012. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011*. Springer, 79–91.
[19] Sriram Krishnamoorthy and Khushbu Agarwal. 2010. Scalable Communication Trace Compression. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE, 408–417.
[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.
[21] Mellanox. 2020. Highly Accurate Time Synchronization with ConnectX-3 and TimeKeeper.
[22] Craig G Nevill-Manning and Ian H Witten. 1997. Identifying Hierarchical Structure in Sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research* 7 (1997), 67–82.
[23] Michael Noeth, Prasun Ratn, Frank Mueller, Martin Schulz, and Bronis R De Supinski. 2009. ScalaTrace: Scalable Compression and Replay of Communication Traces for High Performance Computing. *J. Parallel and Distrib. Comput.* 69, 8 (2009), 696–710.
[24] Robert Preissl, Martin Schulz, Dieter Kranzlmüller, Bronis R. de Supinski, and Daniel J. Quinlan. 2008. Using MPI Communication Patterns to Guide Source Code Transformations. In *Computational Science – ICCS 2008*, Marian Bubak, Geert Dick van Albada, Jack Dongarra, and Peter M. A. Sloot (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 253–260.
[25] Sameer Shende, Allen D Malony, Wyatt Spear, and Karen Schuchardt. 2011. Characterizing I/O Performance Using the TAU Performance System. In *PARCO*. 647–655.

[26] David Skinner. 2005. *Performance Monitoring of Parallel Scientific Applications*. Technical Report. Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US).

[27] Sukhdeep Sodhi and Jaspal Subhlok. 2005. Automatic Construction and Evaluation of Performance Skeletons. In *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 10–pp.

[28] Sukhdeep Sodhi, Jaspal Subhlok, and Qiang Xu. 2008. Performance Prediction with Skeletons. *Cluster Computing* 11, 2 (2008), 151–165.

[29] Saeed Taheri, Sindhu Devale, Ganesh Gopalakrishnan, and Martin Burtscher. 2017. ParLOT: Efficient Whole-program Call Tracing for HPC Applications. In *Programming and Performance Visualization Tools*. Springer, 162–184.

[30] Mustafa M Tikir, Michael A Laurenzano, Laura Carrington, and Allan Snavely. 2009. PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications. In *European Conference on Parallel Processing*. Springer, 135–148.

[31] Jeffrey S Vetter and Michael O McCracken. 2001. Statistical Scalability Analysis of Communication Operations in Distributed Applications. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*. 123–132.

[32] Michael Wagner, Jens Doleschal, and Andreas Knüpfer. 2015. 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, 7.

[33] Michael Wagner, Andreas Knupfer, and Wolfgang E Nagel. 2012. Enhanced Encoding Techniques for the Open Trace Format 2. *Procedia Computer Science* 9 (2012), 1979–1987.

[34] Chen Wang, Jinghan Sun, Marc Snir, Kathryn Mohror, and Elsa Gonsiorowski. 2020. Recorder 2.0: Efficient parallel I/O tracing and analysis. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1–8.

[35] Xing Wu and Frank Mueller. 2013. Elastic and Scalable Tracing and Accurate Replay of Non-Deterministic Events. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*. 59–68.

[36] Qiang Xu, Jaspal Subhlok, and Nathaniel Hammen. 2010. Efficient Discovery of Loop Nests in Execution Traces. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 193–202.

[37] Jidong Zhai, Jianfei Hu, Xiongchao Tang, Xiaosong Ma, and Wenguang Chen. 2014. 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, 143–153.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We ran several codes to evaluate Pilgrim:

- Custom 2D and 3D stencil
- NAS Parallel Benchmarks v3.4.1
- OSU Micro-Benchmarks v5.7
- FLASH4.4
- MILC-QCD 7.8.1

The first four were run at Catalyst at LLNL (https://hpc.llnl.gov/hardware/platforms/catalyst). Compiler modules: intel/19.1.0; impi/2018.0.

The last one, MILC, was run at Theta at Argonne (https://www.alcf.anl.gov/support-center/theta/theta-thetagpu-overview) . Compiler modules: intel/19.1.0.166; cray-mpich/7.7.14

Execution environment information for both systems: https://github.com/wangvsa/pilgrim/tree/master/SC21_ADAE

Three simulations were run for FLASH. They come with the FLASH package, namely Sedov, Cellular and StirTurb. Link to FLASH software: http://flash.uchicago.edu/site

We ran one simulation for MILC which is included in "clover_dynamic" directory. Link to MILC code: https://github.com/milc-qcd/milc_qcd

The script to run those jobs and the input configuration files needed are available here (under each application directory):

https://github.com/wangvsa/pilgrim/tree/master/SC21_ADAE

Author-Created or Modified Artifacts:

```
Persistent ID: https://github.com/pmodels/pilgrim
Artifact name: Pilgrim
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Catalyst at LLNL

Operating systems and versions: TOSS 3

Compilers and versions: icc 19.1.0

Applications and versions: FLASH4.4, MILC_QCD 7.8.1

Libraries and versions: imp-2018 and cray-mpich-7.7.14

URL to output from scripts that gathers execution environment information.

```
https://github.com/wangvsa/pilgrim/tree/master/SC21\
↪ _ADAE
```