# CYPRESS: Combining Static and Dynamic Analysis for Top-Down Communication Trace Compression

Jidong Zhai[†], Jianfei Hu[1,‡], Xiongchao Tang[†], Xiaosong Ma[2,§] and Wenguang Chen[†,¶]

[†]Tsinghua University, [‡]University of California Irvine, [§]Qatar Computing Research Institute,
[¶]Technology Innovation Center at Yinzhou, Yangtze Delta Region Institute of Tsinghua University, Zhejiang
Email: zhaijidong@tsinghua.edu.cn, {hujianfei258,tomxice}@gmail.com, xma@qf.org.qa, cwg@tsinghua.edu.cn

*Abstract*—**Communication traces are increasingly important, both for parallel applications' performance analysis/optimization, and for designing next-generation HPC systems. Meanwhile, the problem size and the execution scale on supercomputers keep growing, producing prohibitive volume of communication traces. To reduce the size of communication traces, existing dynamic compression methods introduce large compression overhead with the job scale. We propose a hybrid static-dynamic method that leverages information acquired from static analysis to facilitate more effective and efficient dynamic trace compression. Our proposed scheme, CYPRESS, extracts a program communication structure tree at compile time using inter-procedural analysis. This tree naturally contains crucial iterative computing features such as the loop structure, allowing subsequent runtime compression to "fill in", in a "top-down" manner, event details into the known communication template. Results show that CYPRESS reduces intra-process and inter-process compression overhead up to 5× and 9× respectively over state-of-the-art dynamic methods, while only introducing very low compiling overhead.**

*Keywords*—*High Performance Computing, Message Passing, Performance Analysis, Trace Compression.*

## I. INTRODUCTION

Communication traces are indispensable in analyzing communication characteristics of MPI (Message Passing Interface) programs for performance problem identification and optimization [1], [2]. They are also highly useful for designing/co-designing future HPC (High Performance Computing) systems [3], such as Exa-scale systems, where trace-driven simulators (such as DIMEMAS [4], BSIM [5], and SIM-MPI [6]) are often employed to predict and compare application performance under alternative design choices.

Many communication trace collection tools have been developed, such as Intel ITC/ITA [7], Vampir [8], TAU [9], Kojak [10] and Scalasca [11]. Typically, these collection tools instrument MPI programs with the MPI profiling layer (PMPI), and record communication operation details (e.g., message type, size, source/destination, and timestamp) during the program execution.

However, as the applications scale up (in terms of both the number of processes and the problem size), the volume of communication traces increases dramatically. For example, ASC benchmark SMG2000 [12] generates about 5TB communication traces only with a small problem size ($64 \times 64 \times 32$)

for 22,538 processes [13]. A large volume of communication traces not only put pressure on trace collection and storage, but also interfere with the execution of user programs. Ironically, while trace analysis becomes more crucial for Exa-scale applications' performance debugging and system design, trace collection itself becomes less and less affordable on such systems.

```
for (i=0; i<10; i++){
    for(j=0; j<=i; j++){
        if(j%2 == 0)
            MPI_Isend(…);
        else
            MPI_Irecv(…);
    }
    MPI_Waitall(…);
    MPI_Reduce(…);
}
```

| | |
|---|---|
| 1:MPI_Isend(…) | 10:MPI_Isend(…) |
| 2:MPI_Waitall(…) | 11:MPI_Waitall(…) |
| 3:MPI_Reduce(…) | 12:MPI_Reduce(…) |
| 4:MPI_Isend(…) | 13:MPI_Isend(…) |
| 5:MPI_Irecv(…) | 14:MPI_Irecv(…) |
| 6:MPI_Waitall(…) | 15:MPI_Isend(…) |
| 7:MPI_Reduce(…) | 16:MPI_Irecv(…) |
| 8:MPI_Isend(…) | … |
| 9:MPI_Irecv(…) | |

Fig. 1. Sample segments of MPI program and its communication trace

To reduce the size of communication traces for large-scale parallel programs, several recent studies investigated communication trace compression to address the ever-increasing trace size [14]–[18]. For most applications, these approaches can produce orders-of-magnitude reduction in trace sizes, facilitating more efficient trace storage and processing. Meanwhile, dynamic trace compression methods take a "bottom-up" approach to discover patterns from the event sequence itself. As reported by existing research, they may have difficulty in compressing complex communication patterns or have very high computational complexity to process such patterns [15]. Also, trace compression itself brings non-trivial overhead (see Figure 1). In particular, the inter-process compressed trace comparison and merge is a rather expensive procedure, with an $O(n^2)$ complexity in merging a pair of per-process traces (where $n$ is the total length of compressed "patterns" after intra-process compression) [14]. We have found that the inter-process compression overhead grows linearly with the number of processes with a recent compression tool [18], which makes it challenging to scale to serve Exa-scale workloads.

In this paper, we propose a novel "top-down" technique, called CYPRESS, for effective, scalable communication trace compression. CYPRESS combines static program analysis with dynamic runtime trace compression. It extracts program structure at compile time, obtaining critical loop/branch control structures, which enable the runtime compression module to easily identify similar communication patterns. This approach is motivated by the observation that most of communication information needed by trace compression can be acquired from program structure. E.g., in Figure 1, a compiler can effortlessly gather that `MPI_Waitall` and `MPI_Reduce` are in the

---

[1]Jianfei took part in this work at Tsinghua University.
[2]Part of the work was performed during author's leave from North Carolina State University.

outermost loop, while `MPI_Isend` and `MPI_Irecv` are in the innermost loop but in different branch structures.

We propose using compiler techniques to statically extract the program structure as an ordered tree called *communication structure tree (CST)*. Our research has found that such structural information provides dynamic trace compression valuable guidance, by offering a "big picture" naturally lacked in bottom-up runtime pattern search. During intra-process trace compression, only communication operations at the same vertex of the CST need to be compared, resulting in a dramatically smaller search space. During inter-process compression, since all the per-process CSTs have the same structure, merging traces is of $O(n)$ computational complexity, compared to $O(n^2)$ offered by existing dynamic methods.

In addition, we have found that the extra overhead caused by such static analysis is negligible, making this hybrid "top-down" method appealing for Exa-scale applications/systems, where we can choose to pay this small, one-time static analysis cost and significantly trim the trace volume *as well as* trace compression overhead, both growing with the job execution time or the number of processes used. More specifically, we consider the major contributions of this work as:

***Combining static and dynamic analysis for trace compression.*** This approach pays a negligible cost for extra program structure analysis and storage, which is independent of the job execution time or the number of processes used. In exchange, the structural information acquired at compile time enables the dynamic compression to gain enormous enhancement in both compression effectiveness and efficiency.

***Sequence-preserving trace compression and replay for trace-driven performance prediction.*** We enable sequence-preserving trace compression, a feature not afforded by most current trace compression tools, using the CST data structure. This allows more accurate trace-driven simulation or performance analysis.

We implemented CYPRESS in the LLVM compiler framework [19]. We evaluated it using NPB programs and a real-world application with a variety of communication patterns, and compare our method with a state-of-the-art dynamic method. Results show that CYPRESS can improve compression ratios in the majority of test cases, and get an average 5-fold and 9-fold reduction for intra- and inter-process trace compression overhead respectively. Finally, we use a real-world application to demonstrate how to use CYPRESS to analyze program performance and replay program for performance prediction.

This paper is organized as follows. In section II, we present an overview of the CYPRESS system. We introduce the key static data structure, the communication structure tree in Section III, followed by intra-process and inter-process communication trace compression algorithms in Section IV. We describe the design of the replay engine and implementation of CYPRESS in Section V and Section VI. Our experimental results are reported in Section VII. Section VIII gives a discussion of related work. Finally, we conclude in Section IX.

## II. OVERVIEW

CYPRESS is a hybrid communication trace compression system consisting of both a static and a dynamic analysis module, as shown in Figure 2. The rest of our discussion focuses on MPI programs/traces, while the CYPRESS approach is general and can be applied to other communication libraries.
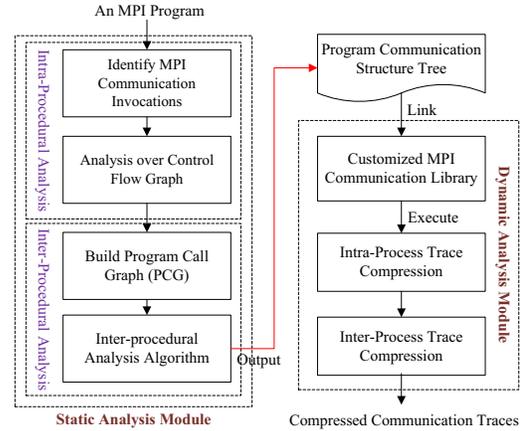


Fig. 2. Overview of CYPRESS

The static analysis module first generates an intermediate summary for each procedure by identifying MPI communication invocations, which collectively determine the program communication patterns. It also identifies control structures, such as *loop* and *branch*, that may affect the execution of communication operations. The static module then constructs the call graph for the whole program, combines the above intermediate summaries through inter-procedural analysis, and stores the resulting program communication structure in a compressed text file.

The dynamic analysis module implements a customized communication library with the MPI profiling layer. Like existing dynamic compression tools, it compresses the intra-process communication traces on-the-fly and conducts an inter-process trace compression at the end of the program's execution. However, the aforementioned program structural information from static analysis contains crucial iterative computing features such as the loop structure, enabling top-down dynamic compression. The key idea is that when informed of such apriori structures, CYPRESS can focus on communication traces *generated by the same piece of code*, which are very likely to have high redundancy.

We illustrate the CYPRESS workflow with a simplified MPI code snippet for Jacobi iteration (Figure 3). In the static analysis module, CYPRESS identifies the loop (line 8) containing four branches (lines 9, 11, 13, 15), eaching calling an MPI routine. Such loop body is likely to generate similar traces across different iterations. Because CYPRESS knows which communication traces correspond to the same call site at runtime, it can pinpoint its similarity search and avoid expensive dynamic probing. Similarly, CYPRESS also tries to merge traces generated by the same piece of code in different processes for further compression. Figure 4 illustrates the compression process, where both intra-process and inter-process repeating patterns are identified. In Figure 4, we use

```
1  int main(int argc, char** argv){
2    MPI_Init(&argc, &argv);
3    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
4    MPI_Comm_size(MPI_COMM_WORLD, &size);
5    // initialization
6    ...
7    double local[(N/size)+2][N], new[(N/size)+2][N];
8    for (k=0; k<steps; k++){
9      if (rank < size - 1)
10       MPI_Send( local[N/size], N, MPI_DOUBLE, rank+1...);
11     if (rank > 0)
12       MPI_Recv( local[0], N, MPI_DOUBLE, rank-1...);
13     if (rank > 0)
14       MPI_Send( local[1], N, MPI_DOUBLE, rank-1...);
15     if (rank < size - 1)
16       MPI_Recv( local[N/size+1], N, MPI_DOUBLE, rank+1...);
17     // compute the variable new and exchange new and local
18     ...
19   }
20   MPI_Finalize();}
```

Fig. 3. A simplified MPI program for Jacobi Iteration

the term of *loop-level* to denote the repeating patterns within each process and *task-level* to denote the repeating patterns between different processes.
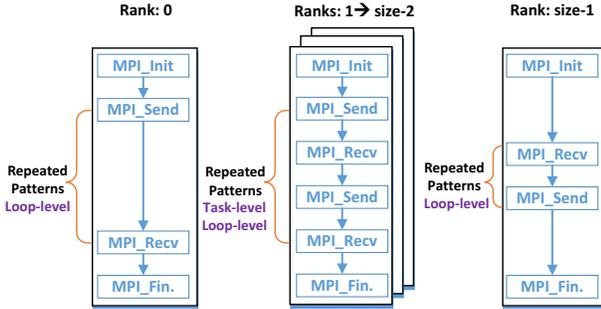


Fig. 4. Communication trace compression (intra-process and inter-process) for Jacobi Iteration

CYPRESS combines the merit of both static analysis and dynamic analysis. The static module can collect the complete program communication structure with little overhead at compile time, while the dynamic module can record the input-dependent control structures, such as loop iteration counts and branch outcomes. When combined, the static program structure enables the dynamic module to perform *informed and scalable* trace processing that compresses more efficiently.

## III. EXTRACTING COMMUNICATION STRUCTURE

As mentioned earlier, CYPRESS leverages static analysis techniques to extract a program communication structure at compile time. The program communication structure records communication invocations and related control structures, which can be used not only to accelerate identification of repetitive communication operations but also to help store compressed communication traces.

To this end, we propose a tree-based data structure called *Communication Structure Tree (CST)*. To retain the original sequence of communication operations traced, the CST is organized into an ordered tree to record the communication invocations and the program control structures. In our MPI-oriented design, the leaf nodes represent MPI communication invocations, while non-leaf nodes represent program control

structures, including branch nodes and loop nodes. Edges represent the hierarchy of the program communication structure. Note that we use an ordered tree to organize all the nodes of the CST, this is because that a pre-order traversal of the CST is completely matched with the static structure of the program. As a result, we can easily capture the program execution at runtime over the static CST.

The CST of a given MPI program is built in two major phases: intra-procedural analysis and inter-procedural analysis. Below we give more details on their perspective processing.

### A. Intra-Procedural Analysis Algorithm

The intra-procedural analysis phase builds an intermediate CST for each procedure. This is done by collecting its control flow graph (CFG) and identifying all the *loop* and *branch* structures. For loops, CYPRESS uses a classic dominator-based algorithm [20].

This phase identifies all the MPI communication invocations and user-defined functions in the program, each of which represented as a leaf node in the intermediate CST. (Note that the user-defined function nodes will be refined during inter-procedural analysis.) If a procedure does not contains any MPI or user-defined function calls, its intra-procedural CST is null. Finally, we add a virtual $root$ vertex to connect all the first-level vertices. For each vertex in the CST, we assign it a unique `global id` (denoted by GID) in pre-order, which is useful for handling MPI asynchronous communications. Algorithm 1 gives the complete process of building an intra-procedural CST. Figure 6 shows the intra-procedural CST for the function $main$ in our sample MPI program (Figure 5).

---

**Algorithm 1** Intra-procedural analysis algorithm in CYPRESS

---

1:  **input:** A CFG for a procedure $p$ (a node of the CFG is a basic block)
2:  **initialize:** $T \leftarrow \phi$
3:  Identify all the loop and branch structures in the CFG
4:  **for all** node $n$ in Post-Order over CFG **do**
5:    **if** $n$ is a loop header node **then**
6:      Insert a loop vertex $n$ into $T$
7:      **for all** vertex $v \in T$ **do**
8:        **if** $v \in$ successor$(n)$ **then**
9:          Insert an edge $e$ from $n$ to $v$ into $T$
10:       **end if**
11:     **end for**
12:   **else if** $n$ is a branch node **then**
13:     For each path insert a branch vertex $n'$ into $T$
14:     **for all** vertex $v \in T$ **do**
15:       **if** $v \in$ successor$(n')$ **then**
16:         Insert an edge $e$ from $n'$ to $v$ into $T$
17:       **end if**
18:     **end for**
19:   **else**
20:     **for all** invocation $i \in n$ **do**
21:       **if** $i$ is an MPI invocation or a user-defined function **then**
22:         Insert a leaf vertex $i$ into $T$
23:       **end if**
24:     **end for**
25:   **end if**
26:  **end for**
27:  Delete the leaf node not an MPI invocation or a user function iteratively

---

### B. Inter-Procedural Analysis Algorithm

The inter-procedural analysis phase combines the intermediate results from intra-procedural phase into a complete

```
1  int main(){
2    for (i=0; i<k; i++) {
3      if (myid % 2 == 0)
4        MPI_Send(buf, size, MPI_INT,
5                 myid+1, 0, MPI_COMM_WORLD);
6      else
7        MPI_Recv(buf, size, MPI_INT,
8                 myid-1, 0, MPI_COMM_WORLD, &status);
9      bar();
10   }
11   foo();
12   if (myid % 2 == 0)
13     MPI_Reduce(sbuf, rbuf, 1, MPI_INT,
14                MPI_SUM, root, comm);
15 }
16 int bar() {
17   for(k=0; k<n; k++)
18     MPI_Bcast(buf, size, MPI_INT, root, MPI_COMM_WORLD);
19 }
20 int foo() {
21   for(j=0; j<m; j++)
22     sum += j;
23 }
```

Fig. 5. An MPI program for illustrating the usage of the CST
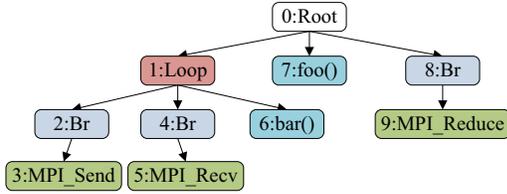


Fig. 6. Intra-procedural CST for the function $main$ in Figure 5

CST. The core idea is to connect all the intra-procedural CSTs according to the relationship of function calls. To do this, we first construct a program call graph (PCG), followed by inter-procedural analysis to iteratively replace user-defined functions with their intra-procedural CSTs. A bottom-up inter-procedural analysis algorithm (Algorithm 2) effectively reduces the iteration number. At the end of this process, the CST of the function $main$ is the final CST for the program. Figure 7 portraits the complete CST for the MPI program in Figure 5.

After analyzing with Algorithm 2, we get an MPI program communication structure tree. However, there are some irrelevant vertices in the CST that will not be used during the trace compression phase, such as the leaf vertices that are not MPI invocations. We conduct a pruning pass over the CST

---

**Algorithm 2** Pseudo code for constructing Communication Structure Tree (CST).

```
1:  input: Intra-procedural CSTs for each procedure p: I_CST(p)
2:  input: The program call graph (PCG)
3:  Change ← True
4:  /* Bottom-Up inter-procedural analysis */
5:  while (Change == True) do
6:      Change ← False
7:      for all  procedure p in Post-Order over PCG do
8:          for all vertex v in Pre-Order over I_CST(p) do
9:              if v is a user-defined function then
10:                 Replace the vertex v with its intra-procedural CST of
                    I_CST(v)
11:                 Change ← True
12:             end if
13:         end for
14:     end for
15: end while
```

---
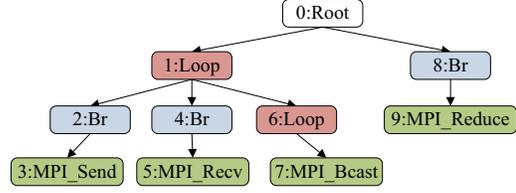


Fig. 7. A complete CST for the program in Figure 5

and delete all the irrelevant vertices. Our pruning algorithm includes two steps: (1) Delete all the leaf vertices in the CST that are not MPI invocations. (2) Repeat step 1 until all the leaf vertices are MPI invocations. We use an iterative DFS (Depth First Search) algorithm over the CST to accomplish the two steps above.

```
1  void foo (float* buf, int num) {
2    if (num == 0) return;
3    else if (num<k && num>j) {
4      MPI_Bcast(...);
5      MPI_Reduce(...);
6      foo(buf, num-1); }
7    else {
8      MPI_Bcast(...);
9      foo(buf, num-1);
10     MPI_Reduce(...); }
11 }
```
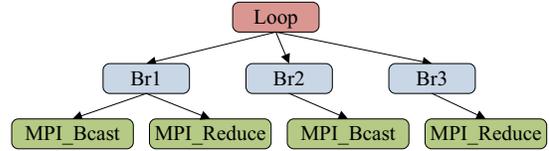


Fig. 8. Conversion of recursive function calls in the CST

Recursive function call creates a challenging problem for building the CST. In CYPRESS, we unroll all the recursions and convert each recursive function call into an approximate *loop* control structure, adopting the method proposed by Emami et al. [21]. At compile time, we insert a pseudo *loop* node at the entry point of the recursive function, and replace all the internal recursive functions with *branch* nodes in the CST. Figure 8 shows an example of converting recursive function calls in CYPRESS. At runtime, we record the branch outcomes and compress the repetitive communication operations. We will further illustrate this in Section IV.

The static CST serves as a template for runtime communication trace compression. Any communication invocation or program control structure has a corresponding vertex in the CST to match. To inform our runtime compression library of the currently executing vertex in the CST, we introduce two extra functions, PMPI_COMM_Structure and PMPI_COMM_Structure_Exit, as shown in Figure 9. CYPRESS automatically inserts codes to bracket each control structure with this pair of functions during static analysis, where the id field assists the runtime analysis to identify the matching CST vertex.

## IV.   RUNTIME COMMUNICATION TRACE COMPRESSION

At runtime, CYPRESS again adopts two-phase communication trace processing, with intra-process and an inter-process trace compression respectively. Both phases utilize

```
PMPI_COMM_Structure (int type, int id)
   type: the program control structure (loop, branch)
         in the CST
   id:   the unique global ID in the CST

PMPI_COMM_Structure_Exit(int id)
   id:   the unique global ID in the CST
```
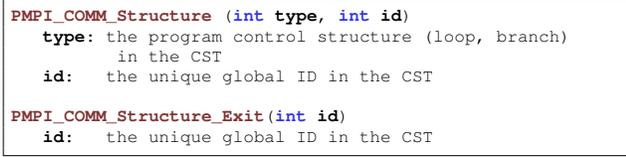
Fig. 9.   Instrumented functions during the static phase in CYPRESS

the program communication structure to achieve effective and low-overhead compression. In order to efficiently organize and store compressed traces, CYPRESS uses a data structure similar to the CST, called the *Compressed Trace Tree (CTT)*. It is an ordered tree with the same edges and the number of vertices as the CST. Each CTT vertex, however, is associated with a linked list storing runtime information.

### A. Intra-Process Communication Trace Compression

During the intra-process compression phase, repeated communication operations for each process are compressed and stored in the CTT. This phase is completed on-the-fly during the program execution.

At the beginning of the program execution, CYPRESS initializes the CTT according to the CST and set the linked list of each CTT vertex to null. It maintains a program pointer, $p$. Facilitated by the ordered nature of the CTT and the instrumented functions at compile time, the pointer $p$ always points to the CTT vertex that is currently being executed. This enables the runtime compression to "fill in" event details into the known communication template. Below we give more details on compressing each type of CTT vertices.

**Communication vertex compression** For each communication operation, the following parameters are collected: *communication type*, *size*, *direction*, *tag*, *context*, and *time*. For the communication time, two types of recording methods are supported in CYPRESS. One records the average time and the standard deviation of repeated communication operations. The other uses a histogram to record the distribution of the communication time [14].

For each incoming communication operation, the current CYPRESS implementation only compares it with the last one in the same CTT vertex, merging them if all their communication parameters (all but the communication time) match. Potentially one can set a larger sliding window for each leaf vertex, to find more similar communication patterns. There is clearly a trade-off between cost and compression effectiveness. We find our current implementation adequate for most of parallel programs.

**Loop vertex compression** For each loop vertex, we need to record its actual iteration counts. This is done by incrementing a certain counter every time the `PMPI_COMM_Structure` function associated with this loop vertex is invoked. The counter stops after `MPI_COMM_Structure_Exit` is called. For nested loops, the inner loop iterations during each round of the outer loop iteration are recorded, to recover the correct communication sequence.

Figure 10 shows a program containing a nested loop with varied inner loop iteration counts. For leaf vertices, similar traces are compressed. $a \times n$ means that the trace $a$ repeats
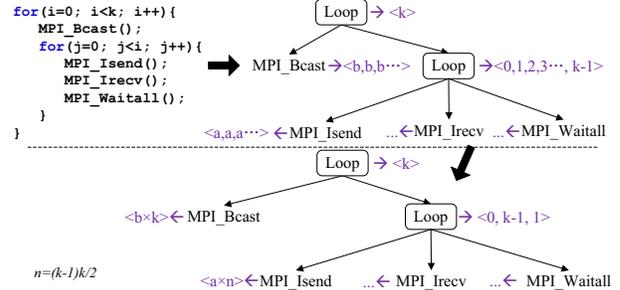


Fig. 10.   Compress communication traces over a nested loop in the CTT.

$n$ times. CYPRESS can further compress loop counts with striding patterns, using tuples like $< 0, k-1, 1 >$, which means that the iteration count is from $0$ to $k-1$ and the stride is $1$. For the outermost loop, we only need to record its iteration count $k$.
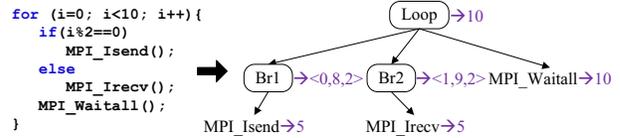


Fig. 11.   Branch compression example

**Branch vertex compression** As mentioned earlier, CYPRESS records all branch outcomes at runtime. Moreover, if a branch vertex is a child of one or more loops, the current iteration number for all the parent loop vertices should also be recorded. Figure 11 shows how to record the branch outcomes in a CTT. Here the branch is selected with an alternating pattern, which can again be denoted by tuples like $< 0, 8, 2 >$ (branch taken at iteration numbers from $0$ to $8$ with a stride of $2$).
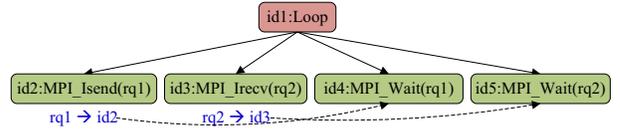


Fig. 12.   Example of mapping between the request handler and GID

**Asynchronous Communication** For asynchronous communication, the MPI library uses request handlers to connect asynchronous communication routines with checking functions (*e.g.*, `MPI_Wait`, `MPI_Waitall`). To associate each non-blocking communication routine (*e.g.*, `MPI_Isend`, `MPI_Irecv`) with its corresponding checking function, we map its request handler to its unique GID in the CST. Then, in the checking function, the request handler is replaced with this GID. Figure 12 shows an example of such mapping. During the decompression phase, the MPI checking function and asynchronous communication routine can be paired again using the GID.

For partial completion in MPI programs, such as `MPI_Waitsome`, `MPI_Testsome`, and `MPI_Testany`, we also use the GID to record the actual non-blocking communication completion. During the decompression phase, we can replay the complete communication sequence using the GID and CTT structure.
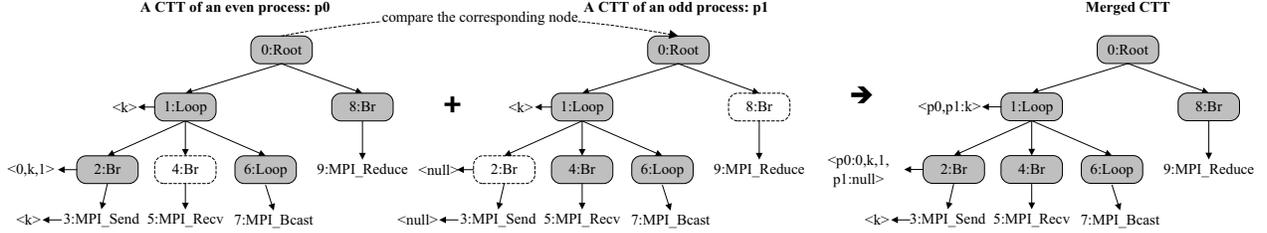
Fig. 13. Perform inter-process communication trace compression with the CTTs (For simplicity, only partial linked lists of the vertices in the CTTs are shown).

**Non-Deterministic Events** Non-deterministic events complicates trace compression. For a non-blocking wildcard receive (*e.g.*, `MPI_Irecv` with `MPI_ANY_SOURCE`), the source is not known when the routine is posted, so the non-blocking wildcard receive cannot be matched upon invocation. In CYPRESS, these wildcard receives are cached, with compression delayed until the corresponding checking functions are executed.

### B. Inter-Process Communication Trace Compression

Finally, the inter-process trace compression phase identifies similar communication patterns *across different processes*, at the end of the program execution (*e.g.*, `MPI_Finalize`). This is where the static-CST-assisted CYPRESS obtains most outstanding advantage against traditional dynamic-only methods: the knowledge obtained at compile time enables CYPRESS to perform informed compression, scalable to large numbers of parallel processes.

Due to the characteristics of the prevailing SPMD (Single Program Multiple Data) model, in common applications today most of the processes execute the same path in the program call graph. As a result, processes generate highly similar communication traces. Still, dynamic-only methods face the challenging communication trace alignment task [14]. For example, when compressing the sequences (a:3, b:4) and (b:2, a:2) (trace:repeating counts) for two processes, dynamic methods may produce three different results using different strategies, (b:2, a:5, b:4), (a:3, b:6, a:2), and (a:5, b:6). Therefore, the computational complexity in compressing a pair of per-process traces for dynamic methods is $O(n^2)$ ($n$ is the number of compressed trace events for each process), which makes it challenging to scale with ever-increasing system size.

CYPRESS solves this problem elegantly, thanks to owning top-down structural information: statically extracted CSTs and dynamically populated CTTs based on the former. The SPMD nature of parallel programs dictates that CTTs across most processes share the common structure existing in the single source code. As any MPI communication invocation corresponds to a unique CTT vertex, we only need to compare the communication invocations *at the same vertex in the CTT*. If a process has not executed a certain call path in the CTT, the call path is ignored for this process.

In Figure 13, we demonstrate the process of merging the CTTs of an even process $p_0$ and an odd process $p_1$ of the program in Figure 5. CYPRESS traverses the CTTs in preorder. It first merges the virtual nodes 0 (referred to by its `GID`). Next, it merges the *loop* vertices 1. Since both processes have the same iteration count, a single value is recorded. After

that, it merges the *branch* vertex 2. Since process $p_1$ does not take this branch, it just records the information for $p_0$. It then merges the leaf vertex 3, where we just record the repeating communication operations for process $p_0$, and so on. Finally, it merges the two CTTs into one. This procedure continues until all per-process CTTs are combined into the merged CTT. We can use a parallel algorithm to merge all the CTTS. Therefore, the computational complexity of CYPRESS is $O(n \log(P))$ when merging $P$ per-process traces, each with length $n$. In contrast, the complexity for the dynamic-only method is dependent on the per-process communication traces, and the complexity for the worst case is $O(n^P \log(P))$.

To effectively compress similar communication invocations for different processes, we need to encode the process rank in a uniform way. To this end, CYPRESS adopts an existing relative ranking method [14]. For example, we use the current process rank $id$ plus or minus a constant value to denote the source process or the destination process. This method is effective for most parallel programs, especially stencil applications.

## V. DECOMPRESSION AND PERFORMANCE ANALYSIS

Compressed communication traces stored in the CTT by CYPRESS can be decompressed by traversing the CTT in preorder and performing the following tasks depending on the vertex type: (1) for a loop vertex, iteratively traversing its child vertices according to the recorded loop count, (2) for a branch vertex, traversing its child vertices according to the recorded branch outcome, and (3) for a communication vertex, printing the communication trace stored in the per-vertex linked list.
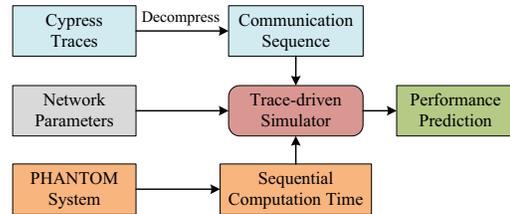


Fig. 14. Integrate CYPRESS with a trace-driven simulator.

Communication traces acquired by CYPRESS can be used for various performance analysis based on communication traces. As a proof-of-concept prototype, we integrated CYPRESS with a trace-driven performance simulator, SIM-MPI [6], as shown in Figure 14. SIM-MPI can simulate various MPI communication routines. The LogGP communication model [22] is used to simulate point-to-point routines, while collective routines are decomposed into point-to-point
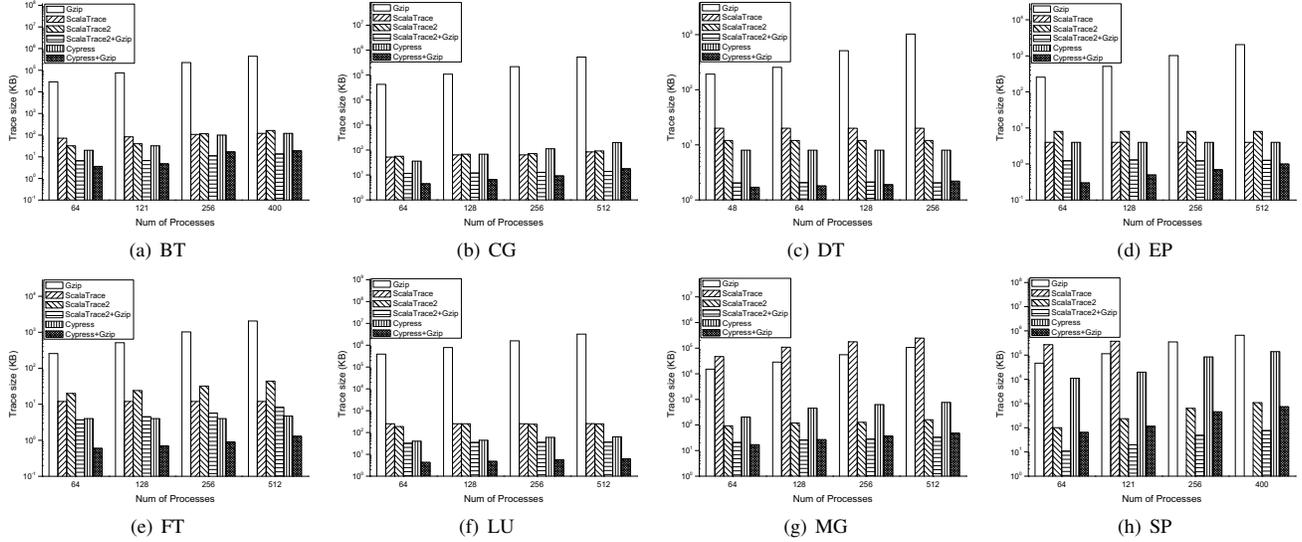
Fig. 15. Total communication trace sizes (KB, log-scale) of NPB programs with different compression tools

operations [23]. In addition to decompressed communication traces, SIM-MPI also needs the sequential computation time of the target program on a target platform, which can be obtained by using deterministic replay on a single node of the target platform [6]. Combining the above results, SIM-MPI predicts the overall performance for a given parallel program.

## VI. IMPLEMENTATION

We implemented the static analysis module of CYPRESS as a plug-in of the LLVM compiler [19], identifying loop and branch structures over the control flow graph at the LLVM intermediate representation (IR) level. Our prototype stores the program CST in a compressed text file. The runtime compression library of CYPRESS is implemented with the MPI profiling layer (PMPI) and is independent of any specific MPI implementation. We have tested CYPRESS with Intel MPI-4.0, Intel MPI-4.1, and MPICH2-1.5. Users do not need to manually modify any application source code to use CYPRESS.

## VII. EVALUATION

### A. Methodology

We evaluate CYPRESS with the NPB benchmark and a real-world application, collectively presenting a variety of communication patterns, to assess the benefits of our approach. We design several groups of experiments to answer the following questions:

1) How does the performance of communication trace compression of CYPRESS compare with that of existing dynamic methods?
2) What is the compression overhead of CYPRESS compared to dynamic methods, both intra-process and inter-process?
3) What is the compilation overhead of CYPRESS to build the CSTs?
4) How to use the compressed traces of CYPRESS to analyze the communication performance for a given parallel program?

We use the Explorer-100 cluster system at Tsinghua University as our experimental platform, which has a peak performance of 104TFlops/s. Compute nodes, each with two 6-core Intel Xeon X5670 processors and 48GB of memory, are interconnected via the QDR Infiniband network. The operating system is RedHat Enterprise Linux Server 5.5 and the MPI library is Intel MPI-4.0.2.

For NAS, we used the NPB 3.3 programs [24], including BT, CG, DT, EP, FT, LU, MG and SP benchmarks. All tests use the CLASS D problem size. We also tested with a real-world application, LESlie3d [25] for computational fluid dynamics, to demonstrate the effectiveness of using CYPRESS compressed traces to analyze the communication performance of a given parallel program. The grid size of LESlie3d is $193 \times 193 \times 193$.

We compare CYPRESS with the other three techniques, Gzip, ScalaTrace [14], and ScalaTrace-2 [18]. Gzip is a popular technique for compressing user documents and data on Linux systems and it is also the trace compression method used in the OTF library [26]. ScalaTrace is the state of the art for lossless dynamic trace compression developed at North Carolina State University. ScalaTrace-2 improves the performance of ScalaTrace by using a loop agnostic inter-node compression scheme. However, the probabilistic method used in ScalaTrace-2 only preserves partial communication information and may lose much information for better compression [18], while CYPRESS retains trace details. CYPRESS supports two types of trace compression files, a normal binary file and a compressed version with Gzip (labeled with CYPRESS+Gzip in the figures). Gzip does bring extra compression at very small overhead, and can be integrated into CYPRESS. ScalaTrace-2 does not support Gzip compression files currently. In order to fairly compare the results, we add extra Gzip support for ScalaTrace-2 (labeled with ScalaTrace2+Gzip in the figures).
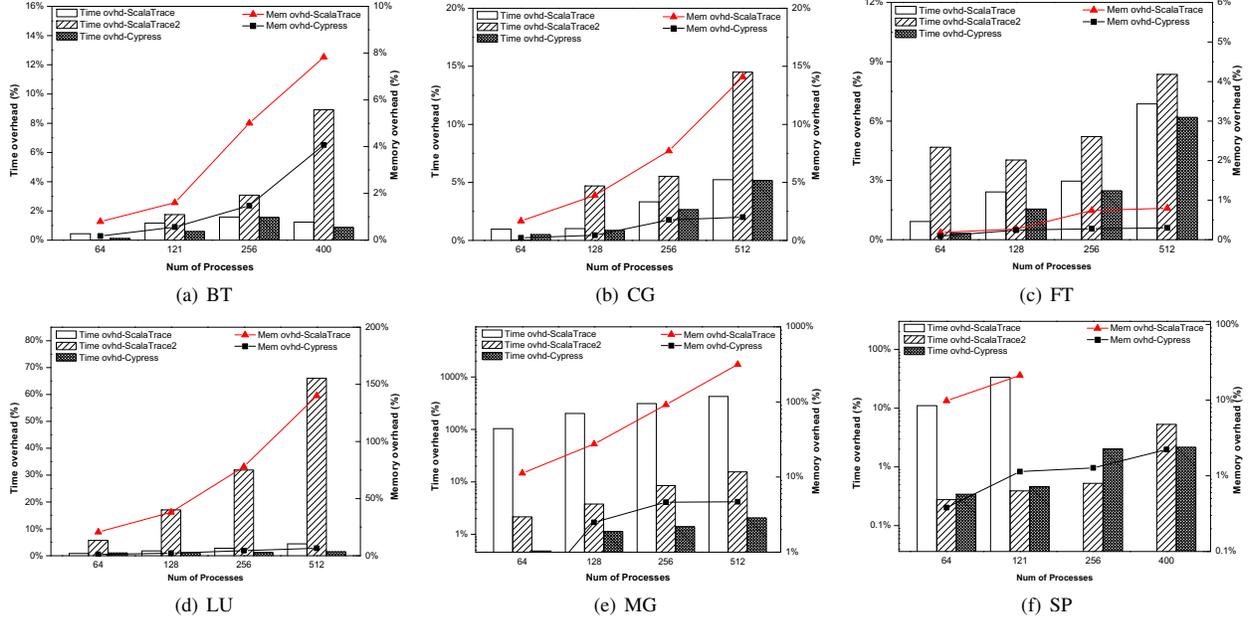
Fig. 16.　Intra-process compression overhead in terms of time and memory (per process) with ScalaTrace and CYPRESS

## B. Communication Trace Size

Figure 15 shows the total trace size of NPB programs in log-scale, collected with different methods. Since Gzip cannot perform inter-process trace compression, the trace sizes increase linearly with the number of processes. In contrast, both CYPRESS and dynamic methods (ScalaTrace and ScalaTrace-2) can get near-constant trace sizes (DT, EP and LU) or sublinear scaling of trace sizes (BT, CG, FT, MG and SP) as the number of processes increases.

For most of NPB programs (DT, FT, LU, MG and SP), CYPRESS shows an order of magnitude improvement over ScalaTrace. Although ScalaTrace-2 has improved the compression ratios over ScalaTrace, CYPRESS outperforms ScalaTrace-2 for DT, EP, FT and LU. At the same time, Gzip appears to bring significant improvement in compression effectiveness for both CYPRESS and ScalaTrace-2. With Gzip incorporated (the "+Gzip" bars), CYPRESS offers up to an order of magnitude improvement over ScalaTrace-2 for the majority of NPB programs. The only case where CYPRESS significantly underperforms is SP, due to its non-uniform communication patterns and varied message sizes. However, from the analysis of compression overhead below, ScalaTrace-2 introduces much more compression overhead than CYPRESS for SP as the number of processes increases.

Among the NPB codes, MG and SP feature complex communication patterns, as shown in Figure 17. MG solves a three-dimensional discrete Poisson equation using a v-cycle multi-grid method. There is a nested 3D torus for some particular communication processes, which results in irregular communication operations between different processes. For example, processes of 8-11 and processes of 12-15 have different communication patterns. SP also presents non-uniform communication patterns between processes. Moreover, for some loops in SP, the message sizes and the message tags of sending
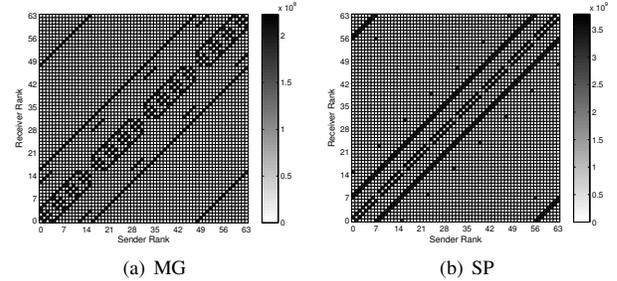


Fig. 17.　Communication patterns of MG and SP (64 processes). The gray level of a cell at the $x^{th}$ row and $y^{th}$ column represents the communication volume (in Byte) between two processes $x$ and $y$.

and receiving communications are varied for each process. Without such information, dynamic methods fail to compress communications effectively (see MG and SP trace sizes of ScalaTrace). To address these complex patterns, ScalaTrace-2 [18] proposes a special algorithm that introduces large compression overhead. However, CYPRESS is able to acquire these information automatically from the source codes and achieve effective trace compression, as discussed further below in overhead analysis.

## C. Trace Compression Overhead

*1) Intra-Process Overhead:* Figure 16 shows the intra-process compression overhead in terms of memory and time for both CYPRESS and dynamic methods. Among the NPB benchmarks, DT, EP, and FT contain few communication operations, resulting in very low memory and time overhead regardless of compression methods. We include only FT to represent this group.

For time overhead (bars in Figure 16), CYPRESS introduces very little overhead on the NPB programs, about 1.58% on

150
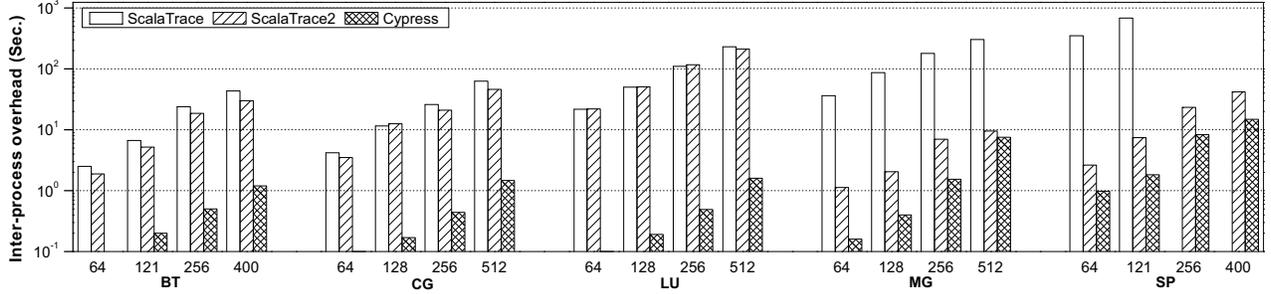
Fig. 18. Inter-process trace compression overhead (in second, log-scale) with different compression methods

average. The maximum overhead of CYPRESS is 5.18% for CG on 512 processes. With both ScalaTrace and ScalaTrace-2, however, the time overhead varies greatly. For example, ScalaTrace incurs a 400%+ overhead on 512 processes for MG. This is caused by the complex nested branches in MG to implement different types of message exchange. For ScalaTrace-2, the time overhead varies from less than 1% to about 60% (LU on 512 processes). In summary, the average intra-process compression overhead for NPB programs is 51.05% for ScalaTrace, 9.1% for ScalaTrace-2, and only 1.58% for CYPRESS.

For memory consumption (line+symbol graphs in Figure 16), CYPRESS outperforms dynamic methods significantly. At runtime, CYPRESS only uses extra memory resources to store the CTT, consuming about 2.2MB (1.79%) memory for each process of NPB programs on average, and the memory consumption changes little with increasing number of processes. ScalaTrace, on the other hand, consumes much more memory, averaging 34MB (36.31%) per process. Its memory consumption also increases rapidly with the number of processes.

*2) Inter-Process Overhead:* Figure 18 shows the inter-process compression overhead (in second) on a log-scale for NPB programs. Due to space limit, we only list the results for BT, CG, LU, MG and SP, considering the small overhead for DT, EP, and FT.

CYPRESS shows 1.5 orders of magnitude improvement for BT and CG, and 2 orders of magnitude improvement for LU over ScalaTrace and ScalaTrace-2. Because the computational complexities for CYPRESS and dynamic methods are $O(n)$ vs. $O(n^2)$ (where $n$ is the length of compressed traces per process), the inter-process overhead is significantly reduced by CYPRESS. For MG and SP, CYPRESS shows about 2-5 times improvement over ScalaTrace-2. In summary, the average inter-process compression overhead for NPB programs is 170.69% for ScalaTrace, 30.3% for ScalaTrace-2, and 3.29% for CYPRESS. These results confirm that our method can be extended to a much larger HPC systems.

TABLE I. COMPILATION OVERHEAD OF CYPRESS (IN SECOND)

| Compile Time | BT | CG | DT | EP | FT | LU | MG | SP |
|---|---|---|---|---|---|---|---|---|
| w/o CYPRESS | 7.19 | 0.95 | 0.40 | 0.34 | 1.56 | 6.57 | 2.51 | 7.40 |
| w/ CYPRESS | 7.44 | 1.06 | 0.45 | 0.43 | 1.61 | 6.73 | 2.60 | 7.51 |
| Overhead(%) | 3.51 | 11.20 | 12.79 | 27.72 | 3.21 | 2.54 | 3.67 | 1.51 |

*3) Compilation Overhead of* CYPRESS*:* To build the CST of an MPI program, we add an extra phase in the LLVM compiler. Table I shows the compilation overhead for NPB programs. We can find that for most of NPB programs the compilation overhead is negligible. The average compilation overhead is 8.27% for NPB programs. The maximum time to built the CST is 0.25 seconds for BT.

*D. Case Study*
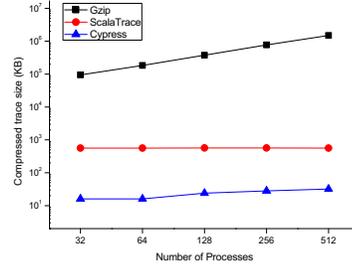


Fig. 19. Compressed communication traces of LESlie3d with Gzip, Scala-Trace and CYPRESS

In this section, we use a real-world application, LESlie3d, to demonstrate the use of CYPRESS to analyze a parallel program performance. LESlie3d (Large-Eddy Simulations with Linear-Eddy Model in 3D) is a computational fluid dynamics program used to investigate a wide array of turbulence phenomena, such as mixing, combustion, acoustics, and general fluid mechanics. Figure 19 shows the trace sizes collected with different methods. Here CYPRESS brings about 1.5 orders of magnitude improvement over ScalaTrace, and 4 orders of magnitude improvement over Gzip.



(a) 32 Processes      (b) 64 Processes
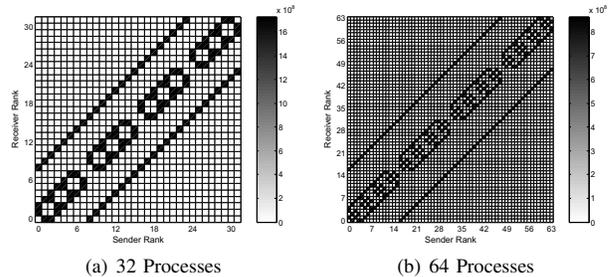
Fig. 20. Extracted communication patterns of LESlie3d with CYPRESS traces

*1) Analyzing Communication Patterns:* The basic function with the compressed traces of CYPRESS is to analyze program communication patterns. Figure 20 shows the communication patterns when the number of processes is 32 and 64. We observe that there is a communication locality in this application. For example, the process 0 only communicates with the processes of 1, 2 and 8. There are only two types of message sizes, *43KB* and *83KB*. With this information, we can perform communication optimization for this application. For instance, on a non-uniform cluster system, we can improve program communication performance with process mapping techniques [27].
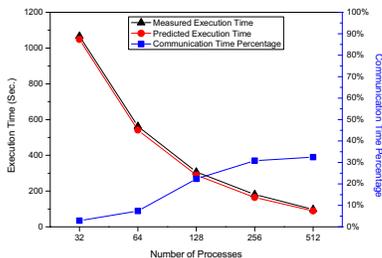


Fig. 21.   Predict execution time of LESlie3d with CYPRESS traces.

*2) Performance Prediction:* CYPRESS preserves the complete communication sequence of the original traces. We decompress LESlie3d communication traces and feed them into the SIM-MPI simulator. The network parameters needed by the SIM-MPI is acquired using two nodes of the Explorer-100 cluster. The sequential computation time is acquired with the deterministic replay technique on a single node of the Explorer-100 cluster. In Figure 21, SIM-MPI obtains high prediction accuracy with the compressed traces, with an average prediction error of 5.9%. We can also see that the program speedup grows slowly as the number of processes increases. This is because the communication time of LESlie3d increases with the number of processes. For example, at process 32 , the average communication time percentage is only 2.85%, whereas the communication time percentage reaches 32.47% at process 512.

## VIII.   RELATED WORK

Communication traces are widely used to characterize communication patterns and identify performance bottlenecks for large-scale parallel applications [1], [2]. Many trace collection tools have been developed, such as Intel ITC/ITA [7], Vampir [8], TAU [9], Kojak [10] and Scalasca [11]. mpiP [28] is a popular communication profiling tool, which collects statistical information for MPI programs and helps developers analyze communication performance.

Open Trace Format (OTF) [26] is a fast and efficient trace format library with regular Zlib compression, but it does not support inter-process communication trace compression, and so the volume of traces size increases with the number of processes. Knupfer et al. [29] proposed using compressed complete call graphs (cCCG) to compress communication traces. However, their method is post-mortem and original traces have to be collected.

Noeth et al. [14] proposed a scalable communication trace compression method, called ScalaTrace. The compression algorithm maintains a queue of MPI events and attempts to greedily compress the first matching sequence. Their method extends regular section descriptors (RSD) and power-RSD (PRSD) to express repeating communication events involved in loop structures. Since all the compression work is completed during the program execution, large compression overhead can be introduced for parallel programs when processing complex communication patterns, especially for inter-process trace compression. Wu et al. proposed a novel framework of ScalaTrace-2 to address the compression inefficiencies of ScalaTrace [18]. Compared to ScalaTrace, ScalaTrace-2 can significantly improve compression rates for scientific applications with inconsistent behavior across time steps and nodes. However, the inter-process compression process still introduces large overhead for large-scale parallel applications.

Xu et al. [15] introduced a framework for identifying the maximal loop nest based on Crochemore's algorithm. Their algorithm can efficiently discover long range repeating communication patterns due to outer loops in MPI traces. However, their algorithm cannot process inter-process trace compression. Krishnamoorthy et al. [16] augmented the SEQUITUR compression algorithm for communication trace compression. Although they employed various optimizations to improve compression overhead for dynamic compression techniques, large overhead is still incurred in their system.

Ratn et al. [30] proposed a method for adding timing information to compressed traces generated by ScalaTrace. They rely on delta times rather than absolute time stamps to express similarities for repeating communication patterns. CYPRESS focuses on trace-driven simulation so it uses two simple mechanisms of histogram and mean values to represent communication time. Their work can be integrated with our work to better record communication time.

Our previous work [31] proposed a slicing technique to efficiently acquire MPI communication traces on a small-scale system for a large-scale application, but it does not compress the collected communication traces. In the future, we can combine both work to effectively analyze communication patterns of parallel applications. Shao et al. [17] proposed a static compiler framework to analyze communication patterns for parallel applications. However, due to pointer alias and program input, their method can only identify static and persistent communication patterns, dynamic communication behavior cannot be processed using their framework.

## IX.   CONCLUSIONS

In this paper, we propose a top-down method, called CY-PRESS, which can effectively compress communication traces for large-scale parallel applications with very low overhead through combining static and dynamic analysis. Our approach leverages a program's inherent static structure to improve the efficiency of trace compression. We implement CYPRESS and evaluate it with several parallel programs. Results show that our method can improve compression ratios significantly in the majority of test cases compared with a state-of-the-art dynamic method, and only incurs 1.58% and 3.29% overhead on average for intra- and inter-process compression respectively.

To the best of our knowledge, CYPRESS is the first work that leverages program structure to improve communication trace compression.

## REFERENCES

[1] J. S. Vetter and F. Mueller, "Communication characteristics of large-scale scientific applications for contemporary cluster architectures," in *IPDPS'02*, 2002, pp. 853–865.

[2] D. Becker, F. Wolf, W. Frings, M. Geimer, B. J. Wylie, and B. Mohr, "Automatic trace-based performance analysis of metacomputing applications," *IPDPS'07*, p. 48, 2007.

[3] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha, "A framework for application performance modeling and prediction," in *SC'02*, 2002, pp. 1–17.

[4] N. Choudhury, Y. Mehta, and T. L. W. et al., "Scaling an optimistic parallel simulation of large-scale interconnection networks," in *WSC'05*, 2005, pp. 591–600.

[5] R. Susukita, H. Ando, and M. A. et al., "Performance prediction of large-scale parallell system and application using macro-level simulation," in *SC'08*, 2008, pp. 1–9.

[6] J. Zhai, W. Chen, and W. Zheng, "Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node," in *PPoPP'10*. ACM, 2010, pp. 305–314.

[7] Intel Ltd., "Intel trace analyzer & collector. http://www.intel.com/cd/software/products/asmo-na/eng/244171.htm."

[8] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," *Supercomputer*, vol. 12, no. 1, Jan. 1996.

[9] S. Shende and A. D. Malony, "TAU: The tau parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, 2006.

[10] B. Mohr and F. Wolf, "KOJAK–A tool set for automatic performance analysis of parallel programs," in *Euro-Par*, 2003.

[11] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[12] Advanced Simulation and Computing Program, "The asc smg2000 benchmark code, https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/smg/."

[13] B. J. N. Wylie, M. Geimer, and F. Wolf, "Performance measurement and analysis of large-scale parallel applications on leadership computing systems," *Sci. Program.*, vol. 16, no. 2-3, pp. 167–181, Apr. 2008.

[14] M. Noeth, F. Mueller, M. Schulz, and B. de Supinski, "Scalable compression and replay of communication traces in massively parallel environments," in *IPDPS'07*, 2007, pp. 1–11.

[15] Q. Xu, J. Subhlok, and N. Hammen, "Efficient discovery of loop nests in execution traces," in *MASCOTS'10*, 2010, pp. 193–202.

[16] S. Krishnamoorthy and K. Agarwal, "Scalable communication trace compression," in *IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, 2010.

[17] S. Shao, A. K. Jones, and R. G. Melhem, "A compiler-based communication analysis approach for multiprocessor systems," in *IPDPS*, 2006.

[18] X. Wu and F. Mueller, "Elastic and scalable tracing and accurate replay of non-deterministic events," in *ICS'13*, 2013, pp. 59–68.

[19] "The LLVM compiler framework. http://llvm.org."

[20] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[21] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive inter-procedural points-to analysis in the presence of function pointers," in *PLDI'94*. ACM, 1994, pp. 242–256.

[22] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "LogGP: Incorporating long messages into the LogP model-one step closer towards a realistic model for parallel computation," in *SPAA'95*. New York, NY, USA: ACM, 1995.

[23] J. Zhang, J. Zhai, W. Chen, and W. Zheng, "Process mapping for mpi collective communications," in *Euro-Par'09*. Springer, 2009, pp. 81–92.

[24] D. Bailey, T. Harris, W. Saphir, R. V. D. Wijngaart, A. Woo, and M. Yarrow, *The NAS Parallel Benchmarks 2.0*. Moffett Field, CA: NAS Systems Division, NASA Ames Research Center, 1995.

[25] J. Wei, C. W. Muelder, K.-L. Ma, S. M. Legensky, C. P. Stone, D. Hiepler, and E. P. Duque, "Ifdtcintelligent in-situ feature detection, extraction, tracking and visualization for turbulent flow simulations," in *ICCFD'12*, Big Island, Hawaii, July 2012.

[26] A. Knupfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel, "Introducing the open trace format (OTF)," in *International Conference on Computational Science*, 2006, pp. 526–533.

[27] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn, "Mpipp: an automatic profile-guided parallel process placement toolset for smp clusters and multiclusters," in *ICS'06*. ACM, 2006, pp. 353–360.

[28] J. S. Vetter and M. O. McCracken, "Statistical scalability analysis of communication operations in distributed applications," in *PPoPP'01*, 2001, pp. 123–132.

[29] A. Knupfer and W. Nagel, "Construction and compression of complete call graphs for post-mortem program trace analysis," in *ICPP'05*, Washington, DC, USA, 2005, pp. 165–172.

[30] P. Ratn, F. Mueller, B. R. de Supinski, and M. Schulz, "Preserving time in large-scale communication traces," in *ICS'08*. New York, NY, USA: ACM, 2008, pp. 46–55.

[31] J. Zhai, T. Sheng, J. He, W. Chen, and W. Zheng, "FACT: Fast communication trace collection for parallel applications through program slicing," in *SC'09*, 2009.