Leveraging Graph Analysis to Pinpoint Root Causes of Scalability Issues for Parallel Applications

Yuyang Jin, Haojie Wang, Xiongchao Tang, Zhenhua Guo, Yaqian Zhao, Torsten Hoefler, Tao Liu, Xu Liu, and Jidong Zhai

***Abstract*—It is challenging to scale parallel applications to modern supercomputers because of load imbalance, resource contention, and communications between processes. Profiling and tracing are two main performance analysis approaches for detecting these scalability bottlenecks. Profiling is low-cost but lacks detailed dependence for identifying root causes. Tracing records plentiful information but incurs significant overheads. To address these issues, we present ScalAna, which employs static analysis techniques to combine the benefits of profiling and tracing - it enables tracing's analyzability with overhead similar to profiling. ScalAna uses static analysis to capture program structures and data dependence of parallel applications, and leverages lightweight profiling approaches to record performance data during runtime. Then a parallel performance graph is generated with both static and dynamic data. Based on this graph, we design a backtracking detection approach to automatically pinpoint the root causes of scaling issues. We evaluate the efficacy and efficiency of ScalAna using several real applications with up to 704K lines of code and demonstrate that our approach can effectively pinpoint the root causes of scaling loss with an average overhead of 5.65% for up to 16,384 processes. By fixing the root causes detected by our tool, it achieves up to 33.01% performance improvement.**

***Index Terms*—Performance analysis, root-cause detection, scalability bottleneck, static analysis.**

Received 27 December 2023; revised 7 October 2024; accepted 21 October 2024. Date of publication 24 October 2024; date of current version 30 December 2024. This work was supported in part by the National Key R&D Program of China under Grant 2022ZD0115304, in part by NSFC for Distinguished Young Scholar under Grant 62225206, in part by the National Natural Science Foundation of China under Grant U23B2027 and Grant 62302251, and in part by China Postdoctoral Science Foundation under Grant BX20230193. Recommended for acceptance by P. Bangalore. *(Corresponding author: Jidong Zhai.)*

Yuyang Jin, Haojie Wang, and Jidong Zhai are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: jinyuyang@tsinghua.edu.cn; wanghaojie@tsinghua.edu.cn; zhaijidong@tsinghua.edu.cn).

Xiongchao Tang is with the Qingcheng.AI Company, Beijing 100084, China (e-mail: txc@qingcheng.ai).

Zhenhua Guo and Yaqian Zhao are with the State Key Laboratory of High-End and Storage Technology, Inspur Electronic Information Industry Company Ltd., Jinan 250013, China (e-mail: guozhenhua@ieisystem.com; zhaoyaqian@ieisystem.com).

Torsten Hoefler is with the Scalable Parallel Computing Laboratory (SPCL), Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland (e-mail: htor@inf.ethz.ch).

Tao Liu is with the Qilu University of Technology (Shandong Academy of Sciences), Jinan 250014, China (e-mail: taoliu@qlu.edu.cn).

Xu Liu is with the Computer Science Department, North Carolina State University, Raleigh, NC 27695 USA (e-mail: xliu88@ncsu.edu).

Digital Object Identifier 10.1109/TPDS.2024.3485789

I. Introduction

As CLOCK frequencies have stalled and Dennard scaling has ended for a decade, the only way to improve computing power is to increase the number of cores, which has led to that top-ranked supercomputers [1] contain millions of processor cores, such as ORNL's Frontier with 8,730,112 cores, RIKEN's Fugaku with 7,630,848 cores, and Sunway TaihuLight with 10,649,600 cores. Due to this rapid growth, developers face significant difficulties, particularly in terms of scaling parallel applications to large-scale supercomputers. However, many parallel applications cannot efficiently utilize the computing resources of modern supercomputers due to poor scalability [2].

Scalability issues are caused by a series of reasons including load imbalance, lock contention, network congestion, serial execution, and so on [3], [4]. They are often deeply hidden by control and data dependence, as well as synchronous and asynchronous communications, making it difficult to locate the underlying root causes. Scalability analysis of parallel applications is thus one of the key aspects of modern performance engineering, especially with the continuing trend of the number of cores increasing. To understand the scalability of parallel applications, researchers have primarily proposed three kinds of approaches:

Profiling records program snapshots at regular clock intervals and summarizes them as statistical data during runtime with very low overhead [8], [9], [10]. However, it misses information between snapshot timestamps, including control/data dependence, inter-process communications, the execution order of events, as well as delay paths, which are critical in identifying the root causes of scaling issues. As a result, profiling-based approaches can only give a coarse insight into scalability bottlenecks, and significant human effort is often needed to locate the exact root causes.

Tracing collects all events and their execution orders as traces, allowing for pinpointing the root causes of scalability issues through analyzing delay paths and complex dependence [6], [11], [12], [13]. However, these approaches suffer from significant runtime and storage costs of recording trace data, which are unacceptable at large scales. In Table I, a comparison of the runtime overhead and storage costs of NPB-CG [5] (NPB-CG represents the Conjugate Gradient program of NAS Parallel Benchmarks) running on 128 processes using tracing and profiling techniques is presented (Please note that it is a run conducted only for the purpose of overhead comparison, and it does not represent a typical use-case for scalability bottleneck

TABLE I
QUALITATIVE RUNTIME AND STORAGE OVERHEAD ANALYSIS ON STATE-OF-THE-ART TOOLS AND SCALANA WITH NPB-CG ON CLASS C FOR 128 PROCESSES [5]

Tools	Approaches	Runtime Ovd.	Storage Cost
Scalasca [6]	Tracing-based	25.3%(w.o. I/O)	6.77 GB
HPCToolkit [7]	Profiling-based	3.41%	11.45 MB
SCALANA	Graph-based	3.73%	125 KB

identification.). Besides, annotation-based tools, such as Caliper [14] and TinyProf [15], provide flexible APIs, which allow performance analysts to focus on specific events and regions, thus reducing the overhead of trace or profile collection.

Modeling can also pinpoint scalability bottlenecks with minimal runtime overhead [16], [17], [18]. Despite this, it usually requires considerable human effort and expert skills to establish accurate performance models. Additionally, it needs a large number of runs for complex real-world applications with numerous input parameters, which is extremely expensive. We thus conclude that pinpointing the root causes of scalability issues for large-scale parallel applications is still an open research problem.

In this work, we present SCALANA, which incorporates static program analysis into dynamic data profiling to automatically and accurately pinpoint root causes of scalability issues with low costs. Specifically, SCALANA leverages static analysis to capture program structures and control/data dependence. Static information is utilized to guide dynamic profiling and reduce runtime overhead and storage costs by avoiding redundant or unnecessary data recording. Then, both static program structures and dynamic performance data are integrated as a parallel performance graph (PPG). With this graph, we propose a backtracking root cause detection approach to pinpoint the underlying issues in large-scale parallel programs where performance issues propagate to other parallel units by complex inter-process communication and inter-thread lock dependence. We evaluate the efficacy and efficiency of SCALANA with benchmarks and several case studies of real-world applications. Experimental results indicate that our tool can pinpoint the underlying root causes of scalability issues more accurately with lower costs compared with the state-of-the-art tools HPCToolkit [7] and Scalasca [6]. Specifically, our approach only introduces 5.65% runtime overhead on average for evaluated applications on 16,384 processes. Optimizing the identified root causes achieves up to 33.01% performance improvement. To summarize, our work has three main contributions:

- We combine static analysis with dynamic profiling, which significantly reduces runtime and storage costs.
- We design a PPG to represent the computation patterns and interactions between different parallel units. This graph is generated with both static program structure and dynamic performance data.
- Based on the PPG, we propose a novel approach to pinpointing the root causes of scalability issues by backtracking complex control, data, lock, and communication dependence.
- We implement the above approaches and develop a performance tool, called SCALANA. We evaluate it with several real-world applications. Experimental results indicate that SCALANA can automatically and effectively pinpoint the root causes of scalability issues.

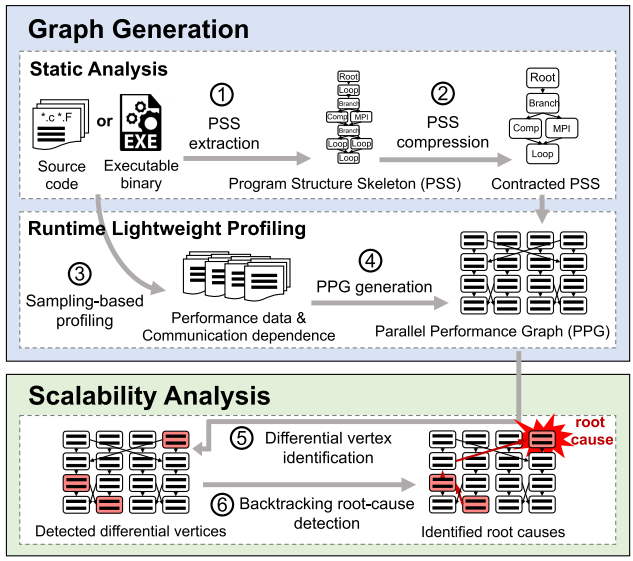


Fig. 1. The overview of SCALANA.

II. DESIGN OVERVIEW

In this section, we first introduce the workflow of SCALANA. Then we present a motivating example to illustrate the scenarios where SCALANA can locate the underlying root causes of scalability issues more accurately compared to existing tools.

A. Overview

SCALANA's main innovation is leveraging static program structure from source codes or executable binaries, to minimize dynamic overhead during runtime. We use a program structure skeleton (PSS) to represent the main patterns of computation and communication in a parallel application with static analysis. At runtime, we utilize lightweight sampling-based techniques to record performance data of computations and dependence data of communications and locks, then generate a parallel performance graph (PPG). Another innovation is the use of graph analysis to automatically locate the root cause of scalability issues by backtracking complex dependence. As shown in Fig. 1, SCALANA has two main modules: graph generation and scalability analysis.

Graph Generation: There are two phases involved in the graph generation module: static analysis and runtime lightweight profiling. Static analysis extracts program structures from source codes or executable binaries and constructs PSSs. PSS vertices represent code snippets of parallel applications, and PSS edges represents execution orders. Runtime lightweight profiling collects performance data and dependence at runtime. PPGs are generated with static PSS and dynamic data and dependence.

- *PSS extraction.*2- It takes the source codes or executable binaries of parallel programs as inputs and generates PSSs by intra- and inter-procedural analysis.

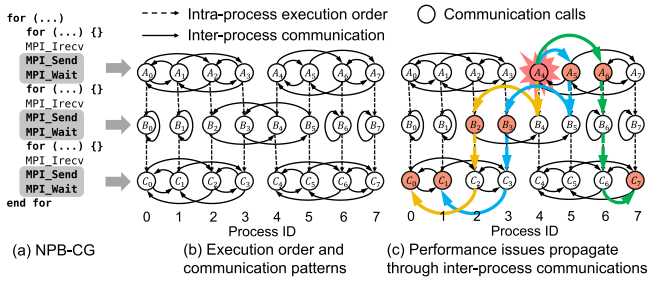


Fig. 2. Motivating example with NPB-CG.

- *PSS compression.* It removes uninformative vertices and edges, as well as combines tiny vertices as a big vertex, which can reduce the costs of offline scalability analysis.
- *Sampling-based profiling.* In this step, we efficiently capture performance data and communication/lock dependence using sampling-based approaches.
- *PPG generation.* This step generates PPGs to effectively represent computation and communication features and the interplay between them across different parallel units.

Scalability Analysis: The offline scalability analysis module contains two phases: differential vertex identification and backtracking root-cause detection. The differential vertex detection phase uses PPG's features to locate scalability issues. The backtracking root-cause detection phase then uses graph analysis techniques to automatically track backward through complex dependence and pinpoint the root causes of the scalability issues.

- *Differential vertex identification:* Based on PPG, we leverage differential analysis to identify the problematic vertex among different parallel scales and parallel units.
- *Backtracking root-cause detection:* This step pinpoints the underlying root causes of identified differential vertices by backward tracking through complex dependence.

B. Motivating Example

A motivating example is provided to illustrate how scalability bottlenecks are deeply hidden and why ScalAna can detect the underlying root causes of scalability issues. Fig. 2(a) shows the body of NPB-CG program's main loop, which implements global synchronizations with three point-to-point (P2P) communications. Fig. 2(b) shows the communication pattern of NPB-CG on 8 processes. Each column represents the execution flow of a process, and each row represents the same P2P communication call of different processes. The edges between different columns stand for inter-process communications. A delay is manually injected in the first P2P communication of process 4, leading to the waiting events in the first P2P communication of processes 5 and 6. This delay further propagates to the following P2P communications of other processes, which leads to a number of performance issues, and finally causes poor scalability (The execution time of NPB-CG with the same problem size on 1,024 processes and 2,048 processes are 49.4 and 49.5 seconds on the Tianhe-2 system, respectively). The *profiling-based tool* HPCToolkit [7] requires significant human effort to pinpoint the root causes of the scalability issue hidden by complex data, control, and inter-process communication dependence. And the *tracing-based tool* Scalasca [6] records over 250 GB of trace logs.

In ScalAna, we establish a PPG combining static and dynamic analysis to capture intra-process execution orders and inter-process communications. Our detection algorithm first detects differential vertices in PPG, then the backtracking algorithm detects the dependence paths between these differential vertices and pinpoints the underlying root causes in order of severity. To summarize, ScalAna is a performance tool for guiding developers to identify the root causes of scalability issues. It takes source codes or executable binaries of parallel applications as inputs, pinpoints the underlying root causes of scalability issues, and reports related information of root causes, including lines of source code and important performance metric counts, for guiding targeted optimizations.

III. Graph Generation

This section outlines our detailed approach to automatically constructing a comprehensive performance representation of a given parallel program, which accurately captures its main computational and communication features. Inspired by many previous works [19], [20], we utilize a graph structure as the representation, where vertices represent computations and program structures, and edges stand for the communications among different processes. The primary component of graph generation is a static analysis module, which is complemented by the runtime lightweight profiling module to process information related to the input.

A. Static Analysis

The purpose of the static analysis module is to extract a program structure skeleton (PSS) that serves as a parallel application's sketch. PSS contains vertices representing the main computation components, communication operations, and program structures, while edges describe the order of execution according to the control and data flow. PSS vertices are classified into several types, containing *Function*, *Call*, *Loop*, *Branch*, *MPI*, as well as *CompIns*, where *CompIns* refers to a set of continuous computational instructions and the others represent basic program structures. The main reason for the type classification is that these types of vertices propagate performance issues in different ways, classifying them could make it easier to backtrack and identify scalability bottlenecks during the offline analysis.

The static PSS extraction involves three primary phases: intra-procedural analysis, inter-procedural analysis, and PSS compression. The intra-procedural analysis first extracts function structure skeleton (FSS) for each function, and the inter-procedural analysis combines FSSs as an overall PSS for the entire program. Finally, PSS compression concises the PSS to reduce the cost of subsequent scalability analysis.

1) Intra-Procedural Analysis: In this phase, an FSS is extracted for each function. To form this skeleton, the control-flow graph of each function is traversed, and branches, loops, and calls are extracted as PSS vertices, control and data flow are identified

Algorithm 1: Intra-Procedural Analysis Algorithm.

Input: The Control-Flow Graph of a function f ($\mathbb{CFG}$).
Output: The Program Structure Skeleton of the input function ($\mathbb{PSS}$).

```
1  Function main(f):
2      PSS ← ∅ ;
3      Insert a Function vertex f into PSS ;
4      b ← the first block of CFG ;
5      handleBlock(b, f) ;
6  Function handleBlock(b, v):
7      if b is visited then return ;
8      if b is a Branch then handleBranch(b, v) ;
9      else if b is a Loop then handleLoop(b, v) ;
10     else
11         forall inst ∈ b do v' ← handleInst(inst, v) ;
12         b ← the next block of b ;
13         handleBlock(b, v') ;
14 Function handleBranch(b, v):
15     Insert a Branch vertex br and an edge ⟨v,br⟩ into PSS ;
16     S ← the first blocks of branch's all successors ;
17     forall s ∈ S do handleBlock(s, br) ;
18 Function handleLoop(b, v):
19     Insert a Loop vertex l and an edge ⟨v,l⟩ into PSS ;
20     lb ← the first block of loop body ;
21     handleBlock(lb, l);
22 Function handleInst(inst, v):
23     if inst is a Call then
24         if the callee is an MPI function then
25             Insert an MPI vertex m and an edge ⟨v,m⟩ into PSS ;
26             return m ;
27         else if the callee is unknown then
28             Insert an Indirect Call vertex i and an edge ⟨v,i⟩ into
                PSS ;
29             return i ;
30         else
31             Insert a Call vertex c and an edge ⟨v,c⟩ into PSS ;
32             return c ;
33     else
34         Insert a CompIns vertex ci and an edge ⟨v,ci⟩ into PSS;
35         return ci ;
```

Algorithm 2: Inter-Procedural Analysis Algorithm.

Input: A map of functions and their Function Structure Skeletons $\mathbb{M}=\langle f,\mathbb{FSS}\rangle$, A Program Call Graph $\mathbb{PCG}$
Output: An overall Program Structure Skeleton $\mathbb{PSS}$

```
1  Function main():
       // Start from the main function
2      interProcedural (M[main]) ;
3      PSS ← M[main] ;
4  Function interProcedural(FSS):
       // Avoid duplicate analysis for recursive
          calls
5      if FSS has been analyzed then return ;
       // Traverse each Call vertex in a FSS
6      forall v ∈ FSS do
7          if v is a Call vertex then
8              f ← the callee of v from PCG ;
9              if f is a user-defined function then
10                 interProcedural (M[f]) ;
11                 Replace v with M[f] ;
```

as edges between these vertices. Algorithm 1 shows the detailed intra-procedural analysis algorithm. The vertices of control-flow graphs are basic blocks, consisting of several instructions, where jump instructions determine program structures, such as loops and branches. First, the control-flow graph is traversed from the first basic block. For basic blocks with branch structures, we add *Branch* vertices and related edges into FSS and then traverse all successors of branches. For basic blocks with loop structures, *Loop* vertices and related edges are added into FSS, then we traverse the loop bodies. For other basic blocks, we identify *MPI*, *Call*, *Indirect Call*, and *CompIns* vertices and add them and related edges into FSS.

2) Inter-Procedural Analysis: During the inter-procedural analysis, we merge all FSSs into an overall PSS using the function call relationships obtained from the program's call graph. As shown in Algorithm 2, the inter-procedural analysis algorithm takes FSSs of all functions and a call graph as inputs, and outputs the overall PSS. Starting from the `main` function's FSS, it traverses all *Call* vertices. For *Call* vertices whose callee functions are user-defined functions, we replace the *Call* vertices with the callee functions' FSSs, and recursively traverse the callee FSSs. We also check whether callee functions' FSSs have been analyzed to avoid duplicate analysis for recursive calls. For *Indirect Call* vertices, their callee functions are unknown at the static phase, and runtime call stack data are further utilized to obtain their function call relationships and to complete the PSS.

3) PSS Compression: The generated PSSs from the previous two steps are usually too large to be practically applied to real-world applications, because these applications normally require a large number of control structures to represent complex algorithms or process various corner cases, and the above steps extract all program structures and create corresponding PSS vertices. We notice that some vertices with a workload can be disregarded because recording performance data for them creates more overhead than benefits. To solve this issue, we apply PSS compression to decrease the PSS size.

The compression rules have a great impact on the graph granularity, as well as the characterization of computation and communication components. We compress PSSs based on four rules: (1) ***MPI* vertices.** All MPI invocations and related control structures are preserved since communication operations are typically the primary scalability bottlenecks as parallel programs scale up. (2) ***Loop* vertices.** We preserve the loop iterations because they dominate the computational performance. A user-defined parameter, `MaxLoopDepth`, is provided to restrict the depth of nested loops and keep the compactness of PSSs. (3) ***CompIns* vertices.** Consecutive vertices are merged into one vertex to reduce PSS size and subsequent offline analysis overhead. (4) **Other vertices.** For the other vertices, including *Call* and *Branch*, if they are not related to the preserved vertices, we temporarily remove them from PSSs. Their runtime performance behavior (hotspot or not) will ultimately determine whether they are preserved or removed.

We take a simple MPI program as an example to further illustrate the process of intra-procedural analysis, inter-procedural analysis, and PSS compression. Fig. 3 gives the code of the MPI program example, which consists of two functions, `main` and `foo`. As shown in Fig. 4(a), the intra-procedural analysis generates two FSSs for the function `main` and `foo`, respectively. After the inter-procedural analysis, the generated final PSS of the entire program is shown in Fig. 4(b). As `MaxLoopDepth` is set to 1, the continuous `Loop1.1` and `Loop1.2` are merged. Fig. 4(c) shows the compressed PSS.

4) Supporting Binary Analysis: In many production environments or under conditions of confidentiality, the source codes of

```
1  int main(){
2      for (int i = 0; i < N; ++i)        //Loop 1
3          A[i] = rand();
4          for (int j = 0; j < i; ++j)    //Loop 1.1
5              sum += A[j];
6          for (int k = 0; k < i; ++k)    //Loop 1.2
7              product *= A[k];
8      foo();
9      MPI_Bcast(...);
10 }
11 void foo() {
12     if (myRank % 2 == 0)
13         MPI_Send(...);
14     else
15         MPI_Recv(...);
16 }
```

Fig. 3. An MPI program example.

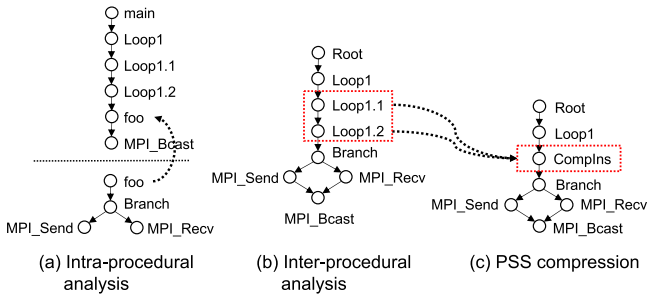


Fig. 4. Illustration of PSS generation.

parallel programs are often not available. Therefore, ScalAna also supports the aforementioned static analysis with executable binaries as inputs. Compared to the source codes or intermediate representations, binaries are more fine-grained, yet some information for structure analysis is missed. For example, loops and branches in the source codes are lowered into multiple jump instructions in the binaries. We utilize the state-of-the-art binary analysis tool Dyninst [21] to extract control-flow graphs and call graphs for the intra-procedural and inter-procedural analysis. However, it is expensive for both static analysis cost and storage cost: 1) fine-grained instruction analysis is time-consuming, incurring significant static analysis overhead, and 2) more structures are extracted and take up more storage space. To address these issues, we propose the following two approaches:

Parallel Intra-Procedural Analysis: Due to the granularity differences between source codes and binaries, even with the advanced tool Dyninst, the static analysis overhead of binaries remains significantly higher than that of source code analysis. This is especially true for large-scale real-world applications, which often contain numerous functions and complex structures, making the analysis overhead unacceptable. Therefore, we employ parallel analysis to accelerate the most time-consuming intra-procedural analysis, with each parallel unit extracting a portion of the functions' FSS. One issue here is that the complexity of functions in parallel programs varies, and directly assigning functions to different parallel units leads to imbalanced workloads. We observe a positive correlation between the size of a function's address space and the complexity of its structure. Therefore, we propose an *address space-based load balancing* approach, following these steps: 1) First, divide functions into different buckets based on the magnitude of their address space (no sorting within each bucket); 2) Then, evenly distribute the functions of each bucket across the parallel units. With this approach, the intra-procedural analysis performs efficiently in parallel.

PSS Granularity Coarsening: The most significant difference between the program structures of binaries and source codes is that loops or branches are converted into a large number of jump instructions. Based on Dyninst, it is possible to extract some loops from these jump instructions, whereas branches cannot, especially when a branch condition contains multiple expressions connected by "or" and "and" operators, the branch is split into multiple conditional jumps. If we add a `Branch` vertex for each conditional jump, it significantly increases the PSS size, which on one hand imposes significant storage cost and on the other hand, increases the complexity of offline scalability analysis. We propose an approach to extract branches and merge multiple conditional jumps into a single branch using debugging information provided in the binaries. Specifically, if the line numbers of conditional jumps in continuous basic blocks are the same, we merge these jump structures into a `Branch` vertex. This approach is simple but can effectively coarsen the granularity of extracted structures and reduce the PSS size.

B. Runtime Lightweight Profiling

In ScalAna, the static data guide dynamic analysis by avoiding unnecessary and redundant performance data collection, allowing the collection of runtime data with very low overhead. Meanwhile, dynamic data can fill in the missed data at the static phase, such as input-dependent information, completing the overall PSS and making it sufficient for offline scalability analysis. Specifically, the runtime lightweight profiling module contains performance metric sampling, indirect call analysis, and inter-process communication data collection. The performance metric sampling collects runtime performance data and associates them with corresponding PSS vertices, and the indirect call analysis captures dynamic call relationships to refine PSS structures. The inter-process communication data collection captures communication patterns to connect per-process PSS and generate a PPG with parallel characteristics.

1) Performance Metric Sampling: ScalAna uses sampling techniques [9] to achieve low overhead collection of program performance metric data and then efficiently associates the performance data to the corresponding PSS vertices. Each vertex in PSS is associated with essential hardware performance metrics, including executed clock cycles, branch prediction success rate, cache miss rate, the number of executed instructions, etc. The main advantage of our approach is collecting fine-grained performance data with low sampling overhead. It is because in ScalAna, the granularity of the performance data depends on the PSS granularity, which is greatly determined by the compression rules in Section III-A3. Note that even though some vertices are merged as one vertex during PSS compression, we record aggregated performance data while retaining performance data for each sampled address associated with this vertex to preserve the ability to identify scalability bottlenecks at the code level.

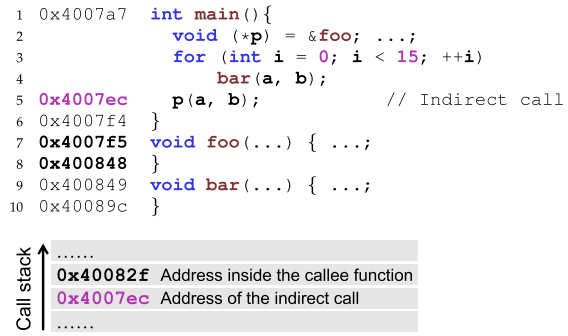


Fig. 5. Illustration for the lightweight indirect call analysis.

The PAPI tool [22] is used to implement the sampling technique. We interrupt each process of parallel programs at fixed intervals of performance events and capture snapshots of each process, including hardware performance monitoring unit (PMU) data and call stack information. Finally, each piece of performance data is associated with the corresponding PSS vertex according to the call stack.

2) Indirect Call Analysis: Real-world applications involve numerous indirect calls, such as function pointers, primarily utilized to process input-dependent cases. However, indirect calls pose a challenge for static analysis in obtaining function call relationships. To address this issue, we gather call relationships for indirect calls at runtime and incorporate them into PSSs. A common indirect call analysis approach is instrumentation, which inserts recording codes before and after the indirect call instruction, as well as at the entry of each function. However, instrumentation usually records full traces and introduces significant overheads.

In SCALANA, we achieve **lightweight indirect call analysis** by analyzing static program address spaces and runtime call stacks, which are obtained through the performance metric sampling (Section III-B1). Call stacks implicitly contain the relationships between indirect calls and callee functions. Specifically, the instruction address above an indirect call address in a call stack must be within the address space of the callee function of the indirect call. Fig. 5 shows an example code with the address space of each function, as well as the address of the indirect call instruction. The sampled call stack in Fig. 5 contains the address of the indirect call in function `main` (`0x4007ec`). And the instruction address (`0x40082f`) above the address of the indirect call falls within the function `foo`'s address space (`0x40082f` $\in$ [`0x4007f5`, `0x400848`]). Therefore, the callee of indirect call in function `main` is the function `foo`. This approach enables SCALANA to capture the function call relationship of indirect calls with minimal overhead. Once the call relationship is obtained, the PSS is further refined by inter-procedural analysis (Section III-A2).

3) Inter-Process Communication Collection: The static analysis module extracts the intra-process control and data dependence. To create a PPG, inter-process communication dependence is further needed to reveal the interactions between processes, which are the most significant differences between parallel and serial programs. This module collects the communication dependence and specialized performance data for `MPI` vertices during runtime, including communication pattern and message size. Traditional approaches trace parallel programs and record complete communication event traces, which incurs significant runtime and storage overhead. To address this issue, SCALANA utilizes two techniques, random sampling-based instrumentation and structure-guided communication data compression.

Random Sampling-Based Instrumentation: Instrumentation techniques repetitively execute inserted codes before and after events at runtime. Full instrumentation records trace logs every time the inserted codes are executed, resulting in huge overhead. To reduce the overhead, some approaches only record trace logs at the first few executions, which is insufficient as they miss programs' dynamic behaviors. To strike a balance between overhead and dynamic behavior capture, we employ a random sampling-based instrumentation technique. Each time before the inserted code is executed, a random number is first generated, and if it falls within a certain interval, the inserted code is executed and communication data are recorded. Otherwise, the original program continues without executing the inserted code. This approach records communication patterns as comprehensively as possible while reducing runtime overhead. Since the main body of most HPC applications is iterative operations, the computation and communication operations that dominate performance or may become scalability bottlenecks will be executed repeatedly. In this iterative mode, the random sampling-based instrumentation technique will not miss the critical information of components that appear repeatedly and dominate performance.

Structure-Guided Communication Data Compression: The executions of parallel programs, especially for real-world parallel applications, contain a large number of communication operations. Most communication operations are conducted by the repeated execution of the same communication function calls in loop iterations, showing great similarity. Thus, recording full data for all communication operations would introduce redundancy. This technique avoids redundant communication data logging by leveraging the program structures from PSS. It works by recording the communication data only once for communication operations that meet both of the following two conditions: (1) They correspond to the same *MPI* vertices in PSS, which means they are conducted by the same communication function calls and have the same calling contexts. (2) Static data dependence analysis indicates that their communication parameters are not repeatedly modified between iterations. This technique reduces storage costs while preserving programs' communication behaviors.

In SCALANA, the PMPI wrapper is utilized to acquire communication data efficiently. PMPI employs dynamic instrumentation techniques and does not require any modification of programs' source code. Different methods are used to gather communication dependence depending on the type of communication operations. We classify communication operations into three types: blocking point-to-point (P2P) communication, non-blocking P2P communication, and collective communication.

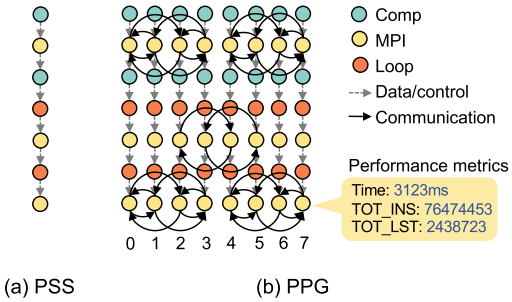


Fig. 6. An example PPG executing on 8 processes.

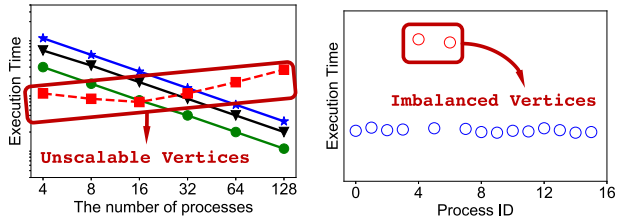


Fig. 7. Examples for two types of differential vertices.

(1) *Blocking P2P communication.* Message type, message size, tag, source process, and target process are recorded. For MPI programs, these data can be extracted directly from the input parameters *datatype*, *count*, *tag*, *source*, and *dest* of the instrumented MPI functions. (2) *Non-blocking P2P communication.* The information to be collected is the same as that for blocking P2P communication, but some of the information cannot be obtained until the check function (e.g., `MPI_Wait` function in MPI programs) is invoked. The *source* process and *tag* can also be obtained from the *status* parameter provided in the `MPI_Wait` function. (3) *Collective communication.* Message type, message size, tag, and communicator are collected, where the communicator refers to the set of processes involved in this communication operation. For MPI programs, the communicator can be obtained through `MPI_Comm_get_info`. This method can also extended to persistent communication under iterative algorithms by hooking related functions like `MPI_Send_Init` and `MPI_Start`.

C. Parallel Performance Graph Generation

The static analysis module and runtime lightweight profiling module prepare sufficient data for building a PPG, which represents the performance behaviors of a parallel program. As parallel programs use the Single Program Multiple Data (SPMD) programming model, each process shares the same source code, which means each process has the same PSS. Thus, we first duplicate PSS for each process. Then, per-process PSSs are connected with inter-process communication dependence edges. For P2P communications, we add communication dependence edges between corresponding *MPI* vertices of the sending and receiving processes. For collective communications, *MPI* vertices of all related processes in a communicator are connected with communication dependence edges. The communication data, including tags, execution time, and waiting time, are associated with communication dependence edges. Fig. 6 gives an example PSS and its PPG on 8 processes.

The overall PPG encompasses the control and data dependence within each process, as well as the communication dependencies between different processes. Furthermore, essential performance data are provided as attributes for each vertex, which are utilized for identifying potential scalability bottlenecks. The performance of a particular vertex in the PPG can be impacted by its computation or memory access patterns, as well as by other vertices' performance behaviors, because performance issues propagate to other processes and vertices through intra-process control and data dependence within processes, and inter-process communication dependence. The following section outlines our approach for pinpointing the root causes of scalability issues for this scenario.

IV. Scalability Analysis

With the acquired PPG, we demonstrate our method for identifying scalability bottlenecks effectively and automatically. Specifically, we first identify differential vertices based on their context-aware location and inter-process behavior, and then backtrack to pinpoint scalability bottlenecks, as well as analyze their severity. The first step aims to detect vertices that exhibit poor scalability and imbalanced behavior. The second step precisely identifies the underlying cause of scalability issues.

A. Differential Vertex Identification

From a given program, we have generated its PPG, which is one of our major advantages. The inter-process communication dependence may vary with different process counts, but the PSS of each process remains unchanged regardless of the problem size or job scale. Based on this observation, we utilize a differential analysis method to identify vertex with imbalanced behaviors or relatively poor scalability. Our approach involves comparing the performance data of vertices in the PPG that correspond to the same vertex in the PSS across different job scales (unscalable vertex identification) and different processes for a given job scale (imbalanced vertex identification).

1) Unscalable Vertex Identification: The main idea is to identify the PSS vertices that exhibit an unusual slope in execution time or other hardware performance metrics compared to other vertices as the process count scales up. For example, Fig. 7(a) illustrates the change trends in execution time for different PSS vertices as the process count scales up. The vertex represented by the red line does not display the same decreasing trend in execution time compared to the other vertices. Scaling issues arise when these vertices consume a substantial proportion of the total execution time.

Detecting unscalable vertices poses the challenge of merging performance data from numerous processes. The straightforward approach involves directly taking the performance data of a specific process as overall data, but this method may overlook important information about other processes. Alternatively, one can utilize the average, mean, or median performance values of all processes and consider the performance variance across

different processes to represent workload distribution. Another approach involves partitioning all processes into distinct groups using clustering algorithms and then aggregating the data for each group. Based on statistical methods of average, mean, and median, we fit the merged performance data from different numbers of processes using a log-log model [23], respectively. We identify potential unscalable vertices by ranking all vertices according to the slope rate as the process or problem scales change based on the fitting results for each statistical method.

2) Imbalanced Vertex Identification: We also analyze the performance data of the same vertex across different processes to detect vertices with significant differences in execution time, potentially indicating imbalanced vertices. In typical Single Program Multi-Data (SPMD) programs, the workload of the same vertex is usually consistent across processes. If a vertex shows noticeably different execution times, it may be flagged as a potentially imbalanced vertex. Imbalanced vertices can be caused by a series of reasons, even without considering performance variance [24]. For example, load balancing issues can lead to imbalanced vertices in certain processes, and communication vertices with excessive synchronization overhead can also be identified as imbalanced vertices. Fig. 7(b) illustrates the execution times of vertices across all processes in a 16-process PPG that correspond to the same PSS vertex. Among these processes, processes 4 and 6 exhibit longer execution times compared to other processes, indicating the presence of imbalanced vertices. Our tool allows users to define a threshold (`ImbThd`) to differentiate between imbalanced and balanced vertices across different processes. For instance, setting `ImbThd` as 2 implies that vertices exceeding twice the average execution time are considered imbalanced vertices.

Algorithm 3: Backtracking Root Cause Algorithm.

Input: A Program Performance Graph PPG, A Set of Unscalable Vertices $\mathbb{U}$, A Set of Imbalanced Vertices $\mathbb{I}$.
Output: A Set of Root Causes $\mathbb{R}$.

```
1  Function Main():
2      ℙ ← ∅ ; // Set of causal paths
3      𝕍 ← ∅ ; // Set of visited vertices
4      forall n ∈ 𝕌 do
5          P ← ∅ ; // Causal path
6          CausalPathBacktracking(n, P) ;
7          Insert P into ℙ ;
8          Insert all v ∈ P into 𝕍 ;
9      forall i ∈ 𝕀 and i ∉ 𝕍 do // Traverse the vertices
         that have not been visited
10         P ← ∅;
11         CausalPathBacktracking(i, P) ;
12         Insert P into ℙ ;
13     ℝ ← RootCauseDetection(ℙ) ;
14     return ℝ ;
15 Function CausalPathBacktracking(v, ℙ):
16     while v is not root or collective communication vertex do
17         Insert v into ℙ ;
18         if v is an MPI vertex then
19             v ← the dest vertex of v's inter-process
                 communication dependence edge ;
20         else if v is an unvisited LOOP or BRANCH vertex then
21             v ← the dest vertex of v's control dependence edge;
22         else
23             v ← the dest vertex of v's data dependence edge ;
24 Function RootCauseDetection(ℙ):
       // Count the number of causal path passes of
          each related vertex
25     𝕄 ← ∅ ; // Map (vertex, path count)
26     forall P ∈ ℙ do
27         forall v ∈ P do 𝕄[v] ++ ;
       // Filter the vertices where the most causal
          paths pass through
28     ℝ ← the elements of 𝕄 with the highest path count ;
29     return ℝ ;
```

B. Backtracking Root Cause Detection

To pinpoint the root causes of the scalability issue, it is crucial to establish connections between the identified differential vertices and determine the causal relationship among them. In SCALANA, we propose a novel approach based on graph analysis, called backtracking root cause detection, which automatically identifies the underlying root causes.

As PPG abstracts dependence that propagates performance issues as edges, we locate the root causes by traversing backward through PPG edges. Algorithm 3 presents the pseudo-code of the *backtracking root cause algorithm*. The algorithm mainly has two steps, which are causal path backtracking and root-cause detection. *(1) Causal path backtracking.* It starts from the unscalable vertices identified in the previous step and then traces back through data/control dependence edges within a process and communication dependence edges across different processes until reaching the root vertices or synchronization vertices, such as blocking communications. During the backtracking process, if unvisited *Loop* or *Branch* vertices have not been visited, the algorithm only traverses the control dependence edges and not the data dependence edges. For instance, when a *Loop* vertex is not visited, the traversal continues from the end vertex of the loop. It is worth noting that complex parallel programs typically contain a large number of dependence edges. Therefore, the search cost would be prohibitively high without optimization. In fact, it is unnecessary to traverse all paths to identify the root causes. In SCALANA, we selectively retain the communication dependence edge only if a waiting event exists, while pruning other communication dependence edges. This approach offers the advantage of reducing both the search space and false positives. Ultimately, we obtain several causal paths that connect identified differential vertices. *(2) Root-cause detection.* We locate the vertices that are at causal path intersections as potential root causes. More paths passing through a vertex indicates more propagation of the performance issue from this vertex, thus the density of path intersections at a certain differential vertex is used to determine the root-cause severity, which is referred to for users to decide optimization targets.

Complexity Analysis: We give a computational complexity analysis on the backtracking root cause algorithm to calculate its cost. In `Main` function, the loops iterate through the unscalable vertex and imbalanced vertex sets, and call a function. The complexity of `Main` function is:

$$T = O((|U| + |I|) \cdot T_P + T_R)$$

where $|U|$ and $|I|$ are the number of unscalable vertices in $\mathbb{U}$ and imbalanced vertices in $\mathbb{I}$, respectively, T_P and T_R are the time complexity of `CausalPathBacktracking`

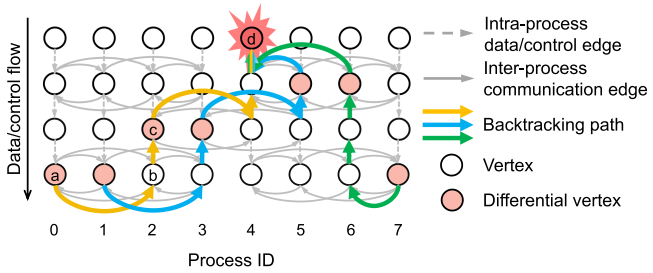


Fig. 8. Differential vertices and backtracking root cause detection on PPG. Yellow, blue, and green arrows all represent backtracking paths and some of them are overlapping.

and `RootCauseDetection` for each vertex, respectively. In `CausalPathBacktracking` function, the while loop runs until a specific condition is met. In the worst case, it could traverse all vertices related to the causal path (Note that the path traversed is from bottom to up, whose length is mostly related to PSS size). The time complexity in the worst cases could approximate:

$$T_P = O(|V|)$$

where $|V|$ is the PSS size. `RootCauseDetection` function iterates over all paths in $\mathbb{P}$ and processes each vertex per path, thus the time complexity is:

$$T_R = O(|P| \cdot |V|)$$

where $|P|$ is the number of identified causal paths. In total, the time complexity of the backtracking algorithm is:

$$T = O((|U| + |I| + |P|) \cdot |V|)$$

$|U|$ is the number of unscalable vertices related to the PSS size, $|I|$ is the number of imbalanced vertices influenced by both the PSS size and the number of processes, and $|P|$ is the number of identified paths primarily associated with the number of processes. Therefore, the cost of the backtracking algorithm is more dependent on the PSS size than on the number of processes, which is the reason why we perform PSS compression to reduce its size (Section III-A3).

Example Case: We take an example in Fig. 8 to further show how to pinpoint scalability bottlenecks. Firstly, we detect the differential vertices, including unscalable and imbalanced vertices, which are filled with red color in the PPG. Then we start from the differential vertex *a* in the lower-left corner, and track through a communication dependence edge to vertex *b* in process 2. Then we backtrack through the data dependence edge to vertex *c* in process 2. We repeat the above steps and finally identify a path with the red color lines connecting all the differential vertices in the processes of 0, 2, and 4. With a similar approach, we backtrack from the other two differential vertices, and then two extra paths are identified in Figure 8, shown as blue and green respectively. With these identified paths, we can connect different differential vertices including MPI invocations and computation components together, and identify the root cause of scalability issues. Finally, vertex *d* is identified as the potential root cause, since most causal paths pass through it.

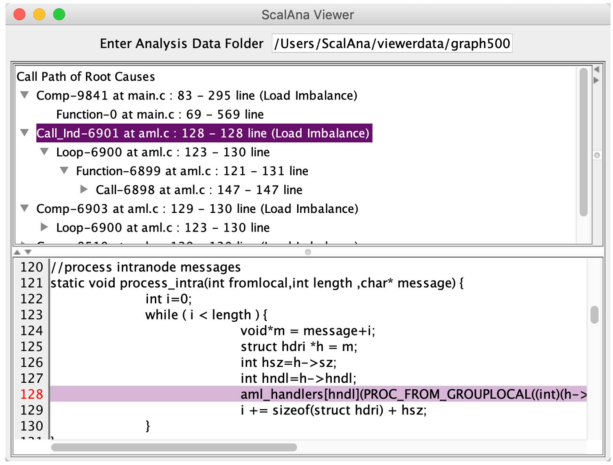


Fig. 9. GUI of ScalAna.

V. Implementation and Usage

A. Implementation

The static analysis module of ScalAna extracts program structures from source codes or binaries. For the CPU programs, we capture their structures based on LLVM [25] from source codes and based on Dyninst(v12.3.0) [21] from binaries. For the GPU programs, we use `nvdisasm` to analyze the structures from the basic block control flow of the cubin file. The runtime profiling module collects performance data. For CPU programs, PAPI [22] is used for sampling performance metrics, and libunwind(v1.6.1) is used to obtain the corresponding call stack for each sampled snapshot. For GPU programs, we leverage the `CUPTI`'s PC sampling module to record instruction-level performance data, which can be associated with source code through instruction addresses dumped by `nvdisasm`. In the current version, ScalAna supports C/C++/Fortran programming languages and MPI/OpenMP/Pthreads/CUDA parallel programming models.

B. Usage

Usage Flows: For end-users, the usage of ScalAna consists five main steps: (1) Set up the configuration file, including binary name, execution command, `MaxLoopDepth`, `ImbThd`, etc. (2) Compile programs with `ScalAna-static.so` to extract program structures, or use `ScalAna-static` to analyze executable binaries of parallel applications and generate PSSs. (3) Execute the applications with `ScalAna-prof` on different process or problem scales and profile runtime performance data. (4) Use `ScalAna-analyzer` to combine static data and runtime data to build PPG, and identify scalability bottlenecks. (5) Use `ScalAna-viewer`, which is a GUI provided by ScalAna, to display the code snippets and severity corresponding to the identified root causes of scaling issues. Fig. 9 is a screenshot of the GUI for ScalAna, which shows the location of identified bottlenecks. In the upper window, root cause vertices are listed along with their calling contexts, sorted by severity. Double-clicking a vertex in the upper window will

provide detailed performance data for each vertex (e.g., Fig. 16). The lower window displays code snippets corresponding to these vertices. Users can further analyze the detailed performance data and source codes to find out the causes of poor performance for scalability bottlenecks.

Configurations: We give some guidance on setting SCALANA's configurable parameters.

(1) `MaxLoopDepth`. We recommend setting this parameter to 10-20 to ensure the completeness of collected program structures for most applications and to avoid the explosion of PSS's *Loop* vertex count in special cases, where the depths of nested loops in specific programs are extremely large.

(2) *Sampling frequency*. The sampling frequency is a key parameter that affects the accuracy of collected performance and runtime overhead. In SCALANA, dependence is more important for backtracking root causes rather than the performance data of computation components. Therefore, we recommend setting the sampling frequency to 200 Hz, which can maintain a low runtime overhead while guaranteeing an acceptable accuracy in performance data collection.

(3) `ImbThd`. This parameter is used to determine the imbalance degree of the vertices marked as imbalanced during the offline analysis period. Based on our experience, we recommend setting `ImbThd` to 1.3 initially, adjusting `ImbThd` based on the analysis results, and re-run offline analysis. If too many root causes are identified, increase the `ImbThd` value, and vice versa.

VI. EVALUATION

A. Experimental Setup

Experimental Platforms: Three cluster testbeds are used for our experiments: (1) *Gorgon*, an eight-node CPU cluster, each node is equipped with dual Intel Xeon E5-2670(v3) processors and 100Gbps EDR Infiniband. (2) *Ja*, a four-node CPU+GPU cluster, each node is equipped with dual AMD EPYC 7742 64-core processors, an NVIDIA Tesla V100, and 100Gbps 2xEDR Infiniband. (3) *Shanhe supercomputer*, each node consists of dual Intel Xeon E5-2692(v2) processors. Tianhe-2 is interconnected through a high-performance high-speed network.

Evaluated Programs: We evaluated the effectiveness of SCALANA using a variety of CPU version and GPU version parallel programs, containing BT, CG, SP, EP, FT, MG, LU, and IS, from NPB benchmark suite(v3.4.2) [5], Sweep3D [26], HPL-GPU [27], HPCG-GPU [28], as well as several real-world applications, ZeusMP [29] and LAMMPS [30]. For NPB programs, we utilize problem size CLASS C on *Gorgon*, and CLASS E on *Shanhe supercomputer*.

Methodology: First, we analyze the features of the generated PSS and PPG for each evaluated program. Subsequently, we evaluate SCALANA's overhead, considering the static binary analysis overhead, runtime overhead, storage cost, and offline detection cost. Furthermore, we demonstrate the process of scalability bottleneck identification using SCALANA with five case studies of real applications, including ZeusMP [29], LAMMPS [30], and Sweep3D [26], to showcase how our approach outstands from existing tools. To mitigate performance variance, we execute each experiment three times and calculate the average results for each process scale. We empirically set `MaxLoopDepth` to 10 and `ImbThd` to 1.3 for all experiments.

Besides, SCALANA is compared with two state-of-the-art performance analysis tools: HPCToolkit [7] and Scalasca [6]. To ensure a fair comparison, we provide the detailed configuration of these tools as follows: (1) Scalasca(v2.6) is a tracing-based tool and it supports using Score-P(v7.1) [31] for data collection. We initially utilize its profiling module to determine specific locations where detailed tracing is required. Then we gradually increase the process count and instrumentation complexity from small-scale runs until scalability bottlenecks are pinpointed. This iterative method allows users to minimize Scalasca's storage cost. (2) For HPCToolkit(v2023.08.1), a profiling-based tool, the sampling frequency serves as a crucial parameter affecting runtime overhead. SCALANA maintains the same sampling frequency (200 Hz) as HPCToolkit for all experiments.

B. Basic Features of PSS and PPG

Table II provides basic features of PSS and PPG, including code sizes, binary sizes, the number of PSS vertices before and after compression, and the number of *Loop* vertices, *Branch* vertices, *CompIns* vertices, and *MPI* vertices, as well as the number of PPG vertices and edges. Note that HPL-GPU and HPCG-GPU binaries are downloaded from the NVIDIA website, and their source codes are not provided, which is marked as '-' in Table II. We run CPU programs on *Gorgon* with 128 processes (BT and SP are executed on 121 processes due to their process count requirements), and GPU programs on *Ja* with 8 processes. From the experimental results, we observe a strong correlation between the total number of vertices and code/binary size in the majority of cases. Besides, the process of compression significantly decreases the number of PSS vertices, averaging a reduction of 36.81%. Additionally, it is worth noting that *Loop*, *CompIns*, and *MPI* vertices occupy more than 66.08% of total vertices, indicating that the PSS effectively represents both computation and communication features of the parallel programs. For instance, LAMMPS has 704.8 K lines of source code (binary size is 14.7 MB) and 89,128 vertices. The number of PSS vertices reduces to 12,113 after compression (reduced by 86.41%). The compressed PSS includes 3,913 *Loop*, 1,309 *MPI*, and 3,728 *CompIns* vertices, which occupy 73.89% of total vertices.

C. Overhead Analysis

The overhead analysis of CPU and GPU programs is conducted on *Shanhe supercomputer* and *Ja*, respectively. The comparison experiments with Scalasca and HPCToolkit are performed on corresponding clusters according to CPU and GPU programs. All CPU programs are executed on 16,384 processes (IS is executed on 1,024 processes due to its program limitation), and GPU programs are executed on 8 processes.

Static Cost: Firstly, as shown in Table III, we evaluate the cost of static binary analysis. The results show that SCALANA incurs a negligible cost, ranging from 0.22 to 23.03 seconds, 2.03 on average. We notice that the static cost is strongly related to the binary size. For instance, the binary size of LAMMPS

TABLE II
CODE SIZE, BINARY SIZE, AND BASIC FEATURES OF PSS AND PPG FOR EVALUATED PROGRAMS

Program	Code(KLoc)	Binary(Bytes)	PSS						PPG	
			#VBC	#VAC	#Loop	#Branch	#CompIns	#MPI	#V	#E
BT	11.3	490K	3,724	2,595	426	257	732	177	332K	374K
CG	2.0	97K	371	307	65	12	103	64	39K	53K
EP	0.6	60K	131	104	17	4	13	25	13K	33K
FT	2.5	222K	2,898	2,524	926	28	277	424	323K	360K
MG	2.8	270K	4,672	3,869	1,370	234	1,061	850	495K	606K
SP	6.3	357K	2,834	2,476	820	41	812	196	317K	251K
LU	7.7	325K	1,782	1,422	491	80	389	225	182K	467K
IS	1.3	37K	392	294	57	13	31	63	38K	66K
ZEUS-MP	44.1	2.2M	28,931	11,317	2,657	1,049	4,912	796	1.4M	2.7M
Sweep3D	3.5	190K	795	417	106	91	129	24	53K	54K
LAMMPS-GPU	704.8	14.7M	89,128	12,113	3,913	1,211	3,728	1,309	97K	106K
HPL-GPU	-	252K	8,378	4,376	628	371	1,913	841	35K	40K
HPCG-GPU	-	4.6M	13,130	8,688	1,341	629	3,133	94	70K	74K

#VBC and #VAC represent the number of PSS vertices before and after compression, while #Loop, #Branch, #CompIns, and #MPI represent the number of Loop, Branch, CompIns, and MPI vertices, respectively.

TABLE III
THE TIME COST OF SCALANA'S STATIC BINARY ANALYSIS

Program	BT	CG	EP	FT	MG	SP	LU	IS	ZMP	SWP	LMP	LMP-G	HPL-G	HPCG-G
Cost (Sec.)	0.41	0.22	0.26	0.34	0.22	0.31	0.25	0.27	0.89	0.32	23.03	21.57	0.59	1.29

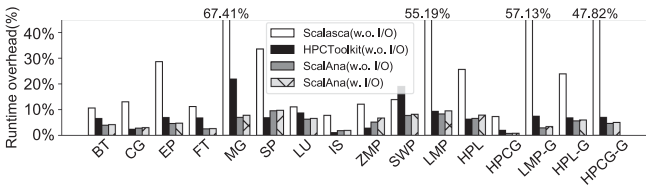


Fig. 10. The runtime overhead of Scalasca [6], HPCToolkit [7], and SCALANA on 16,384 processes.

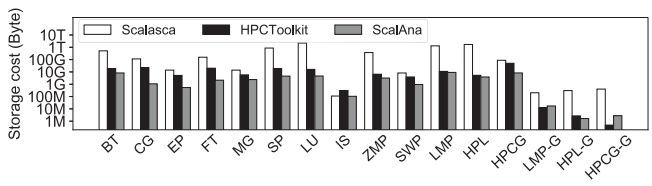


Fig. 11. The storage costs of Scalasca [6], HPCToolkit [7], and SCALANA on 16,384 processes.

is relatively larger (14.7 MB) and its program structures are more complex. The static binary analysis cost of LAMMPS is 23.03 seconds, which is much higher than other programs. Furthermore, the memory cost of static analysis is proportional to the size of the PSS. For example, each vertex in the PSS occupies 32B of memory, and the memory cost of ZeusMP's static analysis is about 9 MB.

Runtime Overhead: As shown in Fig. 10, SCALANA (gray bar) has an average runtime overhead of 5.14% on 16,384 processes, which is significantly lower than that of Scalasca (white bar) and comparable to HPCToolkit (black bar). In Scalasca, we configure the trace buffer size (`SCOREP_TOTAL_MEMORY`) to be large enough to prevent intermediate trace flushing before the program terminates. The light grey bar shows the runtime overhead of SCALANA with I/O. As we only record the compressed PSSs and profiles of each process, the overhead is 5.65% on average (0.51% higher than that without I/O). In addition, we observe that programs with more complex communications usually have higher runtime overhead. For instance, in Sweep3D, processes frequently communicate with neighbors, and its runtime overhead is 7.71%, which is much higher than other programs.

Storage Cost: Fig. 11 presents the storage costs of SCALANA, HPCToolkit, and Scalasca on 16,384 processes as grey, black, and white bars. The average storage cost of CPU programs incurred by SCALANA is 3.77 GB (104 MB to 9.11 GB). SCALANA and HPCToolkit only incur storage costs in the order of Megabytes to Gigabytes, while Scalasca generates Gigabytes to Terabytes of trace data.

Offline Detection Cost: The cost of backtracking root cause detection of SCALANA is also evaluated. In Table IV, comparing with the program's execution time, the offline detection process has a minimal impact, which incurs 35.96 and 6.94 seconds on average for CPU and GPU programs, respectively. We observe that programs with more scaling issues and more complex inter-process communication dependence take a longer execution time for root cause detection. Besides, offline detection's memory consumption depends directly on program structures and the size of the profiling data. For example, it consumes approximately 6.35 Gigabytes of memory for detection analysis of LAMMPS's 16,384-process run.

Scalability: We further evaluate the scalability of SCALANA using HPL and LAMMPS as parameters change, including the number of processes, sampling frequency, and `ImbThd`. (1) *The number of processes.* Fig. 12(a) shows the runtime overhead, storage cost, and offline analysis cost of SCALANA with the process count in the range of 1,024 to 16,384, respectively. The runtime overhead is basically maintained at 6% to 8%, while the

TABLE IV
THE TIME COST OF SCALANA'S OFFLINE ROOT-CAUSE DETECTION ON 16,384 PROCESSES

Program	BT	CG	EP	FT	MG	SP	LU	IS	ZMP	SWP	LMP	LMP-G	HPL-G	HPCG-G
Cost (Sec.)	75.37	18.12	3.76	58.53	17.15	46.88	61.31	2.14	12.73	7.21	92.31	17.36	2.06	1.39

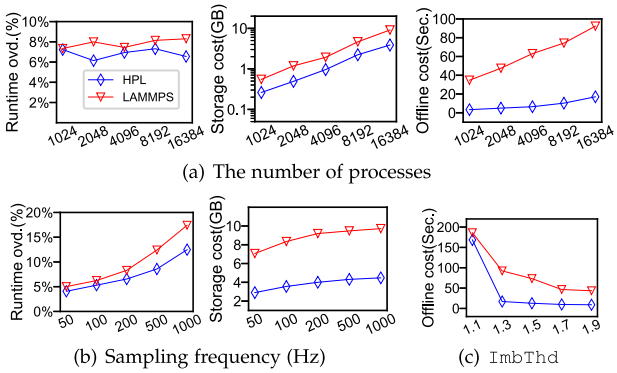


Fig. 12. The scalability of SCALANA on HPL and LAMMPS as parameter changes.

storage cost and offline analysis cost show linear upward trends. (2) *Sampling frequency.* Fig. 12(b) shows the runtime overhead and storage cost of SCALANA with the sampling frequency in the range of 50 to 1,000, indicating superlinear and sublinear trends. For the application runs of the same scale, performance data sizes can certainly represent the accuracy of sampling. An excessively high sampling rate will introduce unacceptable overhead without bringing more accuracy, which is why we recommend 200 Hz as the sampling frequency. (3) **`ImbThd`.** Fig. 12(c) shows the offline analysis time of SCALANA with the `ImbThd` in the range of 1.1 to 1.9. An excessively low `ImbThd` detects too many imbalanced vertices, causing high analysis costs. The analysis cost decreases as `ImbThd` increases. We recommend setting `ImbThd` to 1.3 to balance the analysis cost and accuracy of detection results.

D. Case Study A: ZeusMP

ZeusMP is a computational fluid dynamics application that simulates astrophysical phenomena in 3-D space with the MPI programming model. It employs several non-blocking point-to-point (P2P) communications to implement complex inter-process synchronization. To evaluate its performance and scalability, we execute ZeusMP with a problem size of 64×64×64 on 1 to 128 processes on *Gorgon*. We observe a significant scalability issue, as the speedup of 128 processes is only 55.53×, while the speedup of 64 processes is 35.40×, compared with the baseline of 1 process.

Scalability Analysis with SCALANA: We utilize our performance tool, SCALANA, for root cause analysis and optimization guidance. SCALANA employs the backtracking root cause detection on the generated PPG for ZeusMP on 128 processes to automatically identify the root causes of scalability issues. Fig. 13 illustrates how SCALANA pinpoints scalability bottlenecks in ZeusMP using the backtracking algorithm on the PPG. The elements in this figure are basically the same as those in Fig. 8, the only difference is the way of illustrating normal vertices and differential vertices. In Fig. 13, normal vertices are shown as small points, while differential vertices are shown as circle points, indicating imbalanced performance for the same code snippets. Arrows represent the backtracking causal paths traversing through both intra-process and inter-process dependence. Note that this figure only serves as the illustration for a better understanding but not the standard output in the GUI of SCALANA.

In this case, the `MPI_Allreduce` within the *nudt* function at *nudt.F:610* is identified as unscalable vertices due to its poor scalability in terms of execution time. The proportion of the total execution time of `MPI_Allreduce` increases from 3.70% to 10.85% as the number of processes grows from 16 to 128. In Fig. 13, red (darkest color) lines traverse back from the differential `MPI_Allreduce` vertices and track through both intra-process dependence of control/data flow and inter-process dependence of non-blocking P2P communications at *nudt.F:457, 366, and 285*, forming backtracking causal paths. Similar backtracking paths are represented by orange (lighter color) and yellow (lightest color) lines. Ultimately, SCALANA detects the `Loop_10.1` vertices within the *bvald* function at *bval3d.F:331-359*, which is the top row in Fig. 13, as the root causes of the scalability issues.

Further analysis reveals that the issue arises from only some processes executing the vertices within the *Loop_10.1* function at *bval3d.F:331-359*, while others remain idle, engaged in non-blocking P2P communications within the *nudt* function at *nudt.F:285*. These delays propagate through the non-blocking P2P communications within the function *nudt* at *nudt.F:366, 457*. Then the global synchronization of subsequent `MPI_Allreduce` at *nudt.F:610* results in the observed scalability issues of ZeusMP.

Performance Optimization: In order to fix the scalability bottlenecks identified by SCALANA, we perform targeted optimizations to the program. The load imbalance is caused by the specific computation and communication patterns of ZeusMP, which are difficult to modify. Thus, we introduce multi-thread support at the `Loop_10.1` of *bval3d.F:331-359* using OpenMP pragma to accelerate the execution of busy processes, reducing the waiting time of the idle processes and the load imbalance. Additionally, SCALANA detects other root causes of the scalability issues from the loops at *hsmoc.F: 659-666, 820-835, and 1,048-1,055* with lower severity. By analyzing the Performance Monitor Unit (PMU) data provided by SCALANA, we observe that the number of load/store instructions and cache misses remains high as the number of processes increases. To mitigate these issues, scalar promotion and loop tiling techniques are applied to reduce cache misses and memory accesses. These optimizations bring a significant improvement in the scalability and performance of ZeusMP. The speedup increases from

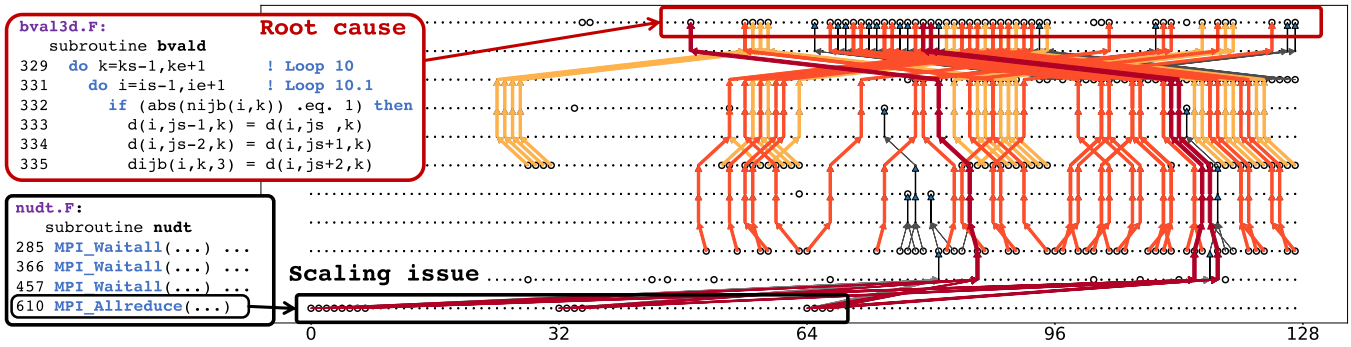
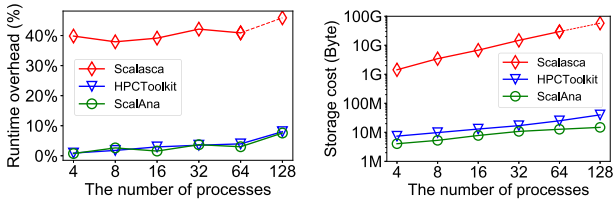


Fig. 13. Backtracking root cause detection on the PPG for ZeusMP executing on 128 processes. The top-down vertical axis indicates control/data flow, and the horizontal axis indicates different parallel processes. Normal vertices are shown as small points, while differential vertices are shown as circle points, indicating imbalanced performance for the same code snippets. Colored arrows represent the backtracking paths traversing through intra- and inter-process dependence.



(a) Runtime overhead (b) Storage cost

Fig. 14. Runtime overhead and storage cost of Scalasca [6], HPCToolkit [7], and ScalAna for ZeusMP (Scalasca detects the scalability bottlenecks when the process count increases to 64.).

55.53× to 61.39× when running with 128 processes, resulting in a performance improvement of 9.55% on *Gorgon*. It is worth noting that there is potential for further exploration of optimization techniques for ZeusMP, but in this study, we only propose simple optimizations to validate that ScalAna does identify the root causes of scalability issues.

Comparison: HPCToolkit detects fine-grained scalability issues of `MPI_Allreduce` at *nudt.F:610* automatically through its profiling data. However, HPCToolkit faces challenges in automatically identifying the root cause `Loop_10.1` at *bval3d.F:331-359* without strong expert skills and manual analysis. It presents multiple scalability issues without a clear analysis of their underlying causal relationships, making it difficult to determine the actual root causes. We use Scalasca to trace, analyze, and visualize ZeusMP on the process count of 1 and increase the process count. Scalasca identifies 16.77% late sender delayed by waiting and propagating from the loops in `bvald`, `hsmoc`, and `lorentz` (the same root causes as that identified by ScalAna) when the process count grows to 64 and with a certain amount of human intervention (annotate the loops as key regions). Fig. 14 illustrates the runtime overhead and storage cost of ScalAna in comparison to Scalasca and HPCToolkit. Both ScalAna and HPCToolkit incur low runtime overhead, averaging 3.19% and 3.52% respectively. However, Scalasca, which is based on tracing, incurs a runtime overhead of 40.89% for trace generation of a 64-process run (excluding I/O). In terms of storage cost, ScalAna outperforms Scalasca by only requiring 14.83 MB of space, whereas the trace data of Scalasca is 28.26 GB for 64 processes.

E. Case Study C: LAMMPS

The LAMMPS [30] (Large-scale Atomic/Molecular Massively Parallel Simulator) package is an open-source software for molecular dynamics simulation, which supports both CPU and GPU architectures using MPI+OpenMP/CUDA programming models. We evaluate the scalability of LAMMPS both on the CPU cluster *Gorgon* and the GPU cluster *Ja*.

1) Lammps-Cpu: On *Gorgon*, we execute LAMMPS under an input (*in.clock.lj*) of the Leonard-Jones system with 32000 atoms for 10000 steps on 1 to 64 processes, and each process only has 1 thread. The results show a speedup of only 32.79× on 64 processes (5.37 seconds) and a speedup of 24.64× on 32 processes (7.15 seconds) compared to a baseline of 1 process (176.14 seconds).

Scalability Analysis with ScalAna: We use ScalAna to detect the scalability bottlenecks of the CPU version of LAMMPS. The analysis results reveal that the `MPI_Send` and `MPI_Wait` within the *CommBrick::reverse_comm* at *comm_brick.cpp:544, 547* are detected as unscalable vertices. As shown in Fig. 15, for `MPI_Send` and `MPI_Wait`, a large number of processes show imbalanced behaviors and have a long waiting time. Fig. 16 shows the PMU data of `MPI_Send`, including the number of executed cycles (`TOT_CYC`), the number of stalled cycles (`TOT_STL_CYC`), and the number of executed load/store instructions (`TOT_LST_INS`). The values of `TOT_LST_INS` in different processes are very similar, indicating the data movement is balanced. The values of `TOT_CYC` vary significantly for different processes, and are strongly related to the values of `TOT_STL_CYC`, which indicates that waiting events dominate the execution of `MPI_Send`. As shown in Fig. 16, a red horizontal line represents the threshold for distinguishing imbalanced and normal vertices, and the imbalanced vertices are marked as little circles in Fig. 15. The backtracking root cause detection is performed on the PPG of LAMMPS for 64 processes. Through the detected backtracking causal paths, `Loop_1.1` within the *PairLJCut::Compute* at *pair_lj_cut.cpp:102-135* is identified as the potential root cause of scaling issues.

We notice that the root cause of scaling issues is workload imbalance in `Loop_1.1`. As shown in Fig. 15, the iteration count *jnum* of `Loop_1.1` is determined by *numneigh*, which indicates the number of neighbors of particle *i*. In this case,

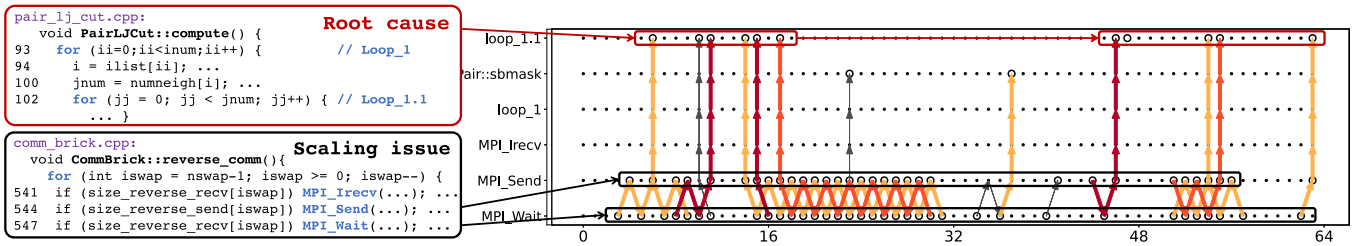


Fig. 15. Backtracking root cause detection on the PPG for LAMMPS executing on 64 processes.

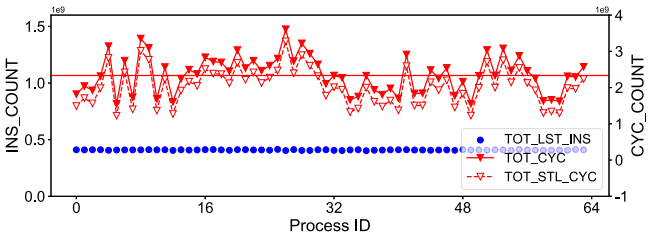


Fig. 16. The PMU data for `MPI_Send` in LAMMPS-CPU executing on 64 processes.

due to the imbalanced distribution of particles in physical space, the iteration counts for different processes vary significantly, which causes the imbalanced behaviors in `Loop_1.1`. And the imbalanced behaviors cause the waiting events in `MPI_Send`. Besides, in the function *CommBrick::reverse_comm*, processes communicate with neighbors in a reverse order, which incurs dependence between P2P communications. This dependence results in `MPI_Send` propagating the waiting event to `MPI_Send` and `MPI_Wait` in other processes, thus a large number of processes in `MPI_Send` and `MPI_Wait` have a long waiting time and poor performance. This scalability bottleneck is also identified as the root cause of performance issues in [20], which is proof that SCALANA can correctly pinpoint the root causes.

Performance Optimization: The root cause is the workload imbalance issue in `Loop_1.1`. We find that LAMMPS provides a `balance` command in the input file to dynamically adjust the workload for load balance. Using the `balance` command, we adjust the subdomain's size and shape for every process in every 250 simulation steps dynamically. The speedup of LAMMPS is improved from 32.79× to 38.04× with this optimization. The performance is improved by 13.81% (the total execution time is reduced from 5.37 to 4.63 seconds).

Comparison: We compare SCALANA with HPCToolkit and Scalasca. HPCToolkit is able to identify the scalability issues in `MPI_Send` and `MPI_Wait`, as well as the imbalanced behaviors in `Loop_1.1`. However, the causal relationship between them for finding the root causes is not easy to obtain without manual code analysis. The full event traces collected by Scalasca make it possible to identify the root causes but incur a 3.25 GB storage cost for 64 processes, whereas SCALANA only needs 16.82 MB space.

2) Lammps-Gpu: On *Ja*, we also run LAMMPS under the *in.clock.lj* input file (with the *balance* command for optimization as mentioned in Section VI-E1) of the Leonard-Jones system with 256000 atoms on 1 to 8 processes, and there is one NVIDIA V100 GPU for each process. The results show a speedup of only 3.80× on 8 processes and a speedup of 1.52× on 2 processes, compared to a baseline of 1 process.

Scalability Analysis with SCALANA: From the analysis results, as GPUs significantly accelerate the kernels, the performance of main computations is greatly improved. For instance, in the GPU version, the function `PairLJCutGPU::compute` at pair_lj_cut_gpu.cpp:76-119, which contains the main CUDA kernel *k_lj*, corresponds to the `Loop_1` within `PairLJCut::compute` at *pair_lj_cut.cpp:93-137* in the CPU version of LAMMPS. It only occupies 45.92% of the total execution time (70.61% for `Loop_1` in the CPU version) on 8 processes. In contrast, the `MPI_Send` calls within the function *CommBrick::reverse_comm* occupy 11.12% of the total execution time (4.73% for the CPU version optimized with *balance* command) on 8 processes. The above results indicate that the performance improvement of computations exposes the communications as new bottlenecks on GPU systems. The PMU data provided by SCALANA shows that the performance behaviors of `MPI_Send` between different processes are relatively balanced, and the transfer costs, especially the costs introduced by memory copy, in the P2P communications are the root causes.

Performance Optimization: One of the optimization strategies for reducing communication time is to overlap communications with computations. As shown in Fig. 17, we observe that there exists a `unpack_reserve` call at line 627 after P2P communications in the function *CommBrick::reverse_comm*, and this call only takes received data *buf_recv* as input, which has no data dependence with the `MPI_Send` at line 617. Thus, we optimize LAMMPS-GPU by overlapping the computations in `unpack_reserve` with the `MPI_Send` communications. The optimized code is shown in the lower box of Fig. 17. With this optimization, the performance of *CommBrick::reverse_comm* is improved by 27.17%, and the performance of the entire application is improved by 7.03%, Besides, the speedup of LAMMPS-GPU is improved from 3.80× to 4.12× for 8 processes.

F. Case Study D: Sweep3D

Sweep3D, which is one of the key components of an application in the ASCI project, is a particle transport simulator that models a single group, time-independent discrete ordinate in a 3-D Cartesian geometric space. We evaluate the scalability for strong scaling of Sweep3D with a fixed total problem size

```
comm_brick.cpp:
  void CommBrick::reverse_comm(){
    for (int iswap = nswap-1; iswap >= 0; iswap--) {
      if (...) { ...;
        if (size_reverse_recv[iswap])
614       MPI_Irecv(buf_recv, ...,sendproc[iswap],...,&request);
        if (size_reverse_send[iswap]) {
          buf = f[firstrecv[iswap]];
617       MPI_Send(buf, ...,recvproc[iswap],0,world);
        }
619     if (size_reverse_recv[iswap]) MPI_Wait(&request, ...);
627     avec->unpack_reverse(..., buf_recv);
      else {...;}
    }
```

optimization

```
    for (int iswap = nswap-1; iswap >= 0; iswap--) {
      if (...) { ...;
        if (size_reverse_recv[iswap]) {
          MPI_Irecv(buf_recv, ...,sendproc[iswap],...,&request);
        }
        if (size_reverse_send[iswap]) {
          buf = f[firstrecv[iswap]];
          MPI_Wait(&send_req, MPI_STATUS_IGNORE);
          MPI_Isend(buf, ...,recvproc[iswap],0,world, &r0);
        }
        if (size_reverse_recv[iswap]) MPI_Wait(&request, ...);
        avec->unpack_reverse(..., buf_recv);
      } else {...;}
    }
    MPI_Wait(&send_req, MPI_STATUS_IGNORE);
```

Fig. 17. Code optimization for the *CommBrick::reverse_comm* function in LAMMPS.

(domain space) of 450×450×450, 32 k-planes for blocking (MK), and the number of processes ranging from 1 to 256 on *Gorgon*. The speedup for 256 processes is only 37.57×, which is even less than that for 128 processes (45.53×), compared to a baseline of 1 process. Sweep3D shows poor scalability.

Scalability Analysis with ScalAna: We use ScalAna to pinpoint the scalability bottlenecks for Sweep3D. As the number of processes increases from 64 to 256, the total time proportion of `mpi_recv` and `mpi_allreduce` within functions *rcv_real* and *global_int_sum* at *mpi_stuff.f:99, 195* rises from 10.14% to 22.03%, which indicates scaling issues. As shown in Fig. 18, the backtracking detection algorithm finds several causal paths (yellow, orange, and red arrows), which start from the scaling issues of `mpi_recv` and `mpi_allreduce`, track backward through inter-process communication dependence to `mpi_send` within function *snd_real*, as well as intra-process data/control dependence to some loops. As a result, ScalAna pinpoints three root causes in order of severity based on the density of causal path intersections: `loop_1.2.2.5.2.6.1`, `loop_1.2.2.5.2.5.1`, and `loop_1.2.2.5.2.2.1` within the *sweep* function at *sweep.f: 486-493, 478-480, 388-389*.

Performance Optimization: We find that the load imbalanced workloads are caused by two aspects both related to the input value of the k-plane size for blocking `MK` from existing research [32]. Firstly, if the k-plane size is not divisible by the k-direction grid size, the k-plane workloads would not be evenly assigned to every process, leading to imbalance. Secondly, Sweep3D implements pipeline execution for k-planes. If the k-plane pipeline is finer-grained, the waiting overhead caused by the imbalance can be effectively hidden and reduced. Thus, we set `MK` to 5, which is divisible by the k-direction grid size (the original value 32 is not divisible) and increases the number of k-plane pipelines. With this simple optimization, the communication time significantly decreases (as shown in Fig. 19, the `TOT_CYC` of the scaling issue `mpi_allreduce` is significantly reduced.), and the overall execution time of Sweep3D decreases from 19.22 to 14.45 seconds. The speedup of Sweep3D is improved from 37.57× to 49.97× on 256 processes.

Comparison: HPCToolkit can efficiently identify potential bottlenecks, containing `mpi_recv` (15.43% of the total execution time) and `loop_1.2.2.5.2.5.1` (14.18% of the execution time), through hotspot analysis. However, the imbalance behaviors of loops that cause the waiting of MPI communications still need manual code and performance data analysis. Scalasca has traced Sweep3D on Blue Gene/P and finds the same root causes as our tool [32], but it introduces 6.75 GB storage costs for 256 processes, which is much more than ScalAna (568.50 KB).

VII. Related Work

In the field of performance analysis, Mohr [33] provides a comprehensive survey of cutting-edge tools that employ both tracing and profiling techniques. Knobloch et al. [34] offer a comprehensive overview of performance tools tailored for heterogeneous HPC applications. In this section, we delve into specific instances of related work in performance analysis.

Tracing: Tracing-based approaches collect critical event traces for understanding programs' fine-grained performance behaviors during runtime. Intel has developed a trace collection and analysis tool, ITA/C [11], which enables a detailed understanding of MPI programs' performance behaviors. Several advanced tools, such as TAU [35], [36], Vampir [37], [38], [39], Scalasca [6], [40], built on the Score-P infrastructure [41], [42], support various parallel programming models including MPI, OpenMP, Pthread, and CUDA. These tools provide visualizations of event traces and facilitate fine-grained performance analysis for developers. Paraver [43], [44], [45] is a tracing-based performance tool that enables flexible collection of performance data and detailed metric variance detection. Becker et al. [46] leverage event traces to diagnose the performance of large-scale parallel programs. Although various techniques for trace compression have been proposed [19], [47], [48], [49], tracing-based approaches still incur significant overhead without human intervention and are thus unsuitable for large-scale parallel applications in production environments.

Profiling: Profiling techniques interrupt programs at regular clock cycles and capture program snapshots. They extract statistical information from programs with minimal overhead. mpiP [8] and INAM [52] are lightweight MPI profiling tools for multi-process applications, which collect context-aware statistical information of MPI function calls, network fabric, and schedulers. Tallent et al. [9], [10] employ a call path profiling technique to analyze and quantify load imbalance in parallel programs. STAT [53] conducts large-scale debugging through stack trace sampling to collect behavior profiles of applications. HPCToolkit [7] utilizes sampling techniques to profile the performance of applications and provides visualizations of the results through *hpcviewer* and *hpctraceviewer*. Nsight [54], Arm MAP [55], and CrayPat [56] are profilers that support GPU,

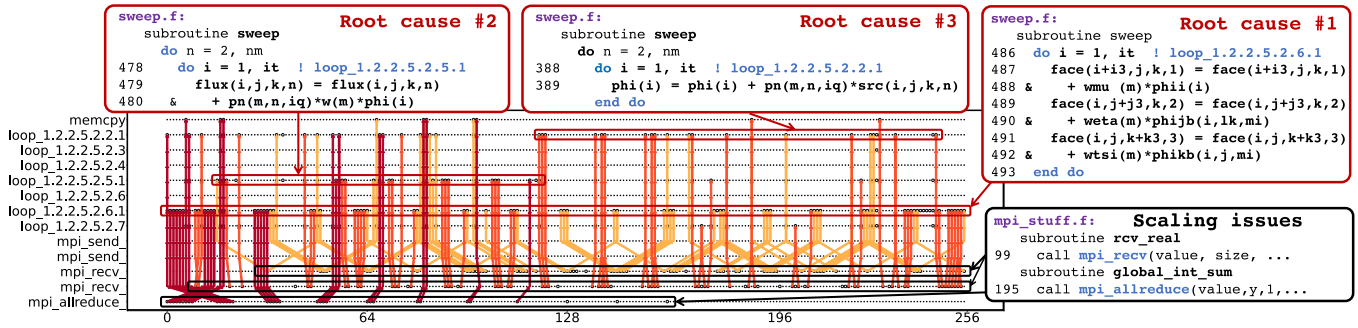


Fig. 18. Backtracking root cause detection on the PPG for Sweep3D executing on 256 processes.

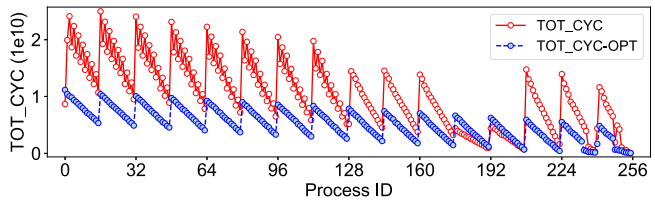


Fig. 19. The PMU data for `mpi_allreduce_` in Sweep3D executing on 256 processes.

TABLE V
A COMPARISON OF SCALANA AND STATE-OF-THE-ART TOOLS

Tools	Approach	Overhead	Scalability analysis
Scalasca [6]/ Score-P [31]	Tracing	High w.o. human intervention	Automatic with [51]
HPCToolkit [7]	Profiling	Low	Basic results + Human analysis
Caliper [14]	Annotation	Low	-
Hatchet [52]	-	-	Provided APIs + Human analysis
SCALANA	Graph	Low	Automatic

'-' means that Caliper is a collection tool and Hatchet is designed as an analysis tool directly using the profiles collected by HPCToolkit or Caliper.

ARM, and XC platforms, respectively. However, profiling often fails to capture critical information so human analysis is needed for fully understanding program behavior.

Annotation-Based Tools: To meet both the requirements of low overhead and detailed tracing, researchers proposed methods using annotations to focus on specific performance events or regions. Caliper [14] provides low-cost and flexible annotation interfaces allowing users to implement specific collection strategies. Score-P [31] allows developers to annotate specific regions to obtain detailed traces. Timemory [57] abstracts the performance collection and analysis using modularity, and gives unified APIs to integrate the capabilities of different tools. Based on the annotation APIs, TinyProf [15] further reduces the I/O cost with a multi-phase I/O strategy. In addition, many tools, like Hatchet [51] and PerFlow [20] give a series of programmable APIs for easing the implementation of specific analysis tasks. Theoretically, as a tool designed for the scalability analysis task, SCALANA could be implemented using the above tools.

Performance Analysis Based on Program Structure: Cypress [19] and Spindle [58] combine static analysis with runtime profiling/tracing for compressing communication traces and monitoring memory access. Program structures extracted at compile time significantly reduce runtime overhead. Program structures are also beneficial in large-scale debugging [59], [60], [61], [62], as they reveal both inter-process and intra-process dependence.

Scalability Issue Diagnosis: Coarfa et al. [63] perform a top-down analysis of HPCToolkit's hpcviewer data to identify scalability bottlenecks. However, this approach does not handle complex communication patterns with massive inter-process dependence. Barnes et al. [23] utilize regression-based techniques for scalability prediction. Calotoiu et al. [17] automate performance modeling techniques to pinpoint scalability bugs, with further improvements using compiler techniques by Bhattacharyya et al. [16]. A scalable performance modeling framework is proposed by Chen et al. [64] based on critical-path candidates. ScaAnalyzer [65] focuses on detecting scalability bottlenecks caused by memory access behaviors by collecting, attributing, and analyzing memory-related metrics during runtime. Bohme et al. [50] pinpoint the root causes of waiting events by performing a forward and backward replay on recorded event traces. Inspired by Bohme's backward-replay approach, our tool, SCALANA, proposes a backtracking root cause detection algorithm. In addition, our approach leverages the program structure to guide minimal runtime profiling, instead of collecting huge-cost event traces. Consequently, SCALANA incurs very low runtime overhead, storage cost, and offline detection cost.

As shown in Table V, our work aims to identify scalability bottlenecks based on lightweight profiling, and graph analysis on PPG. In this way, SCALANA can achieve accurate root cause detection with low overhead. Other tools are general-purpose and need human efforts to pinpoint the root causes.

VIII. CONCLUSION

In summary, this paper introduces SCALANA, a performance tool that effectively pinpoints root causes of scalability issues in parallel applications using a combination of static analysis and dynamic profiling. Through the backtracking root cause detection, SCALANA can automatically identify the underlying root causes of scalability issues in complex parallel applications by traversing the parallel performance graph. Our evaluation

demonstrates that ScalAna can accurately and efficiently pinpoint scalability bottlenecks with minimal overhead for real-world applications with up to 704 K lines of code.

The analysis method in ScalAna has limitations in effectively handling scenarios where imbalances are both significant and highly irregular, such as in emerging applications like graph computing. We aim to address this by enhancing our method with quantitative algorithms. Besides, the visualization of performance analysis for large-scale applications also remains an open problem in the HPC domain. Using interactive modes and hierarchical visualization could be a potential way to address it.

References

[1] "Top500 website," 2020. [Online]. Available: http://top500.org/

[2] J. Y. Shi, M. Taifi, A. Pradeep, A. Khreishah, and V. Antony, "Program scalability analysis for HPC cloud: Applying Amdahl's law to nas benchmarks," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. Piscataway, NJ, USA: IEEE, 2012, pp. 1215–1225.

[3] O. Pearce, H. Ahmed, R. W. Larsen, P. Pirkelbauer, and D. F. Richards, "Exploring dynamic load imbalance solutions with the CoMD proxy application," *Future Gener. Comput. Syst.*, vol. 92, pp. 920–932, 2019.

[4] D. Schmidl, "Openmp scalability limits on large SMPs and how to extend them," Ph.D. dissertation, RWTH Aachen Univ., 2016. [Online]. Available: https://publications.rwth-aachen.de/record/659783/files/659783.pdf

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

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

[7] L. Adhianto et al., "HPCTOOLKIT: Tools for performance analysis of optimized parallel programs," *Concurrency Computation: Pract. Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[8] J. Vetter and C. Chambreau, "mpiP: Lightweight, scalable MPI profiling," 2024. [Online]. Available: https://github.com/LLNL/mpiP

[9] N. R. Tallent, L. Adhianto, and J. M. Mellor-Crummey, "Scalable identification of load imbalance in parallel executions using call path profiles," in *Proc. 2010 ACM/IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2010, pp. 1–11.

[10] N. R. Tallent, J. M. Mellor-Crummey, L. Adhianto, M. W. Fagan, and M. Krentel, "Diagnosing performance bottlenecks in emerging petascale applications," in *Proc. Conf. High Perform. Comput. Netw. Storage Anal.*, 2009, pp. 1–11.

[11] "Intel trace analyzer and collector," 2024. [Online]. Available: https://software.intel.com/en-us/trace-analyzer

[12] J. Zhai, W. Chen, and W. Zheng, "Phantom: Predicting performance of parallel applications on large-scale parallel machines using a single node," *ACM Sigplan Notices*, vol. 45, no. 5, pp. 305–314, 2010.

[13] J. C. Linford, S. Khuvis, S. Shende, A. Malony, N. Imam, and M. G. Venkata, "Performance analysis of OpenSHMEM applications with TAU commander," in *Workshop on OpenSHMEM and Related Technologies*. Berlin, Germany: Springer, 2017, pp. 161–179.

[14] D. Boehme et al., "Caliper: Performance introspection for HPC software stacks," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 550–560.

[15] K. Fan, S. Kesavan, S. Petruzza, and S. Kumar, "TinyProf: Towards continuous performance introspection through scalable parallel I/O," in *Proc. 39th Int. Conf. ISC High Perform. 2024 Res. Paper*, 2024, pp. 1–12.

[16] A. Bhattacharyya, G. Kwasniewski, and T. Hoefler, "Using compiler techniques to improve automatic performance modeling," in *Proc. 24th Int. Conf. Parallel Architectures Compilation*, 2015, pp. 468–479.

[17] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, "Using automated performance modeling to find scalability bugs in complex codes," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2013, Art. no. 45.

[18] F. Wolf et al., "Automatic performance modeling of HPC applications," in *Software for Exascale Computing-SPPEXA 2013–2015*. Berlin, Germany: Springer, 2016, pp. 445–465.

[19] J. Zhai, J. Hu, X. Tang, X. Ma, and W. Chen, "Cypress: Combining static and dynamic analysis for top-down communication trace compression," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 143–153.

[20] Y. Jin, H. Wang, R. Zhong, C. Zhang, and J. Zhai, "Perflow: A domain specific framework for automatic performance analysis of parallel applications," in *Proc. 27th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2022, pp. 177–191.

[21] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *Proc. 10th ACM SIGPLAN-SIGSOFT Workshop Prog. Anal. Softw. Tools*, 2011, pp. 9–16.

[22] "PAPI tools," 2024. [Online]. Available: http://icl.utk.edu/papi/software/

[23] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. De Supinski, and M. Schulz, "A regression-based approach to scalability prediction," in *Proc. 22nd Annu. Int. Conf. Supercomputing*, 2008, pp. 368–377.

[24] X. Tang, J. Zhai, X. Qian, B. He, W. Xue, and W. Chen, "vSensor: Leveraging fixed-workload snippets of programs for performance variance detection," *ACM SIGPLAN Notices*, vol. 53, pp. 124–136, 2018.

[25] "The LLVM compiler framework," 2024. [Online]. Available: https://llvm.org/

[26] Y. Yoon, J. C. Browne, M. Crocker, S. Jain, and N. Mahmood, "Productivity and performance through components: The ASCI Sweep3D application," *Concurrency Computation: Pract. Experience*, vol. 19, no. 5, pp. 721–742, 2007.

[27] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: Past, present and future," *Concurrency Computation: Pract. experience*, vol. 15, no. 9, pp. 803–820, 2003.

[28] J. Dongarra, M. A. Heroux, and P. Luszczek, "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems," *Int. J. High Perform. Comput. Appl.*, vol. 30, no. 1, pp. 3–10, 2016.

[29] J. C. Hayes et al., "Simulating radiating and magnetized flows in multiple dimensions with ZEUS-MP," *Astrophysical J. Suppl. Ser.*, vol. 165, no. 1, 2006, Art. no. 188.

[30] A. P. Thompson et al., "LAMMPS-a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Comput. Phys. Commun.*, vol. 271, 2022, Art. no. 108171.

[31] A. Knüpfer et al., "Score-P: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011*. Berlin, Germany: Springer, 2012, pp. 79–91.

[32] B. J. Wylie, D. Böhme, B. Mohr, Z. Szebenyi, and F. Wolf, "Performance analysis of Sweep3D on Blue Gene/P with the Scalasca toolset," in *Proc. 2010 IEEE Int. Symp. Parallel Distrib. Process. Workshops Phd Forum*, 2010, pp. 1–8.

[33] B. Mohr, "Scalable parallel performance measurement and analysis tools-state-of-the-art and future challenges," *Supercomputing Front. Innovations*, vol. 1, no. 2, pp. 108–123, 2014.

[34] M. Knobloch and B. Mohr, "Tools for GPU computing–debugging and performance analysis of heterogenous HPC applications," *Supercomputing Front. Innovations*, vol. 7, no. 1, pp. 91–111, 2020.

[35] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, 2006.

[36] "TAU homepage. University of oregon," 2024. [Online]. Available: https://www.cs.uoregon.edu/research/tau/

[37] M. S. Müller et al., "Developing scalable applications with vampir, vampirserver and vampirtrace," in *PARCO*. Princeton, NJ, USA: Citeseer, 2007, pp. 637–644.

[38] A. Knüpfer et al., "The vampir performance analysis tool-set," in *Tools for High Performance Computing*. Berlin, Germany: Springer, 2008, pp. 139–155.

[39] "Vampir homepage. technical university dresden," 2024. [Online]. Available: http://www.vampir.eu

[40] "Scalasca homepage. Julich Supercomputing Centre and German Research School for Simulation Sciences," 2024. [Online]. Available: http://www.scalasca.org/

[41] D. an Mey et al., "Score-P: A unified performance measurement system for petascale applications," in *Competence in High Performance Computing 2010*. Berlin, Germany: Springer, 2011, pp. 85–97.

[42] "Score-P homepage. score-p consortium," 2024. [Online]. Available: https://www.vi-hps.org/projects/score-p

[43] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris, "Dip: A parallel program development environment," in *Proc. Eur. Conf. Parallel Process.*, 1996, pp. 665–674.

[44] H. Servat, G. Llort, J. Giménez, and J. Labarta, "Detailed performance analysis using coarse grain sampling," in *Proc. Eur. Conf. Parallel Process.*, 2009, pp. 185–198.
[45] "Paraver homepage. Barcelona supercomputing center," 2024. [Online]. Available: https://tools.bsc.es/paraver
[46] D. Becker, F. Wolf, W. Frings, M. Geimer, B. J. Wylie, and B. Mohr, "Automatic trace-based performance analysis of metacomputing applications," in *Proc. 2007 IEEE Int. Parallel Distrib. Process. Symp.*, 2007, pp. 1–10.
[47] M. Noeth, F. Mueller, M. Schulz, and B. R. De Supinski, "Scalable compression and replay of communication traces in massively parallel environments," in *Proc. 2007 IEEE Int. Parallel Distrib. Process. Symp.*, 2007, pp. 1–11.
[48] S. Krishnamoorthy and K. Agarwal, "Scalable communication trace compression," in *Proc. 2010 10th IEEE/ACM Int. Conf. Cluster Cloud Grid Comput.*, 2010, pp. 408–417.
[49] A. Knupfer and W. E. Nagel, "Construction and compression of complete call graphs for post-mortem program trace analysis," in *Proc. 2005 Int. Conf. Parallel Process.*, 2005, pp. 165–172.
[50] D. Bohme, M. Geimer, F. Wolf, and L. Arnold, "Identifying the root causes of wait states in large-scale parallel applications," in *Proc. 2010 39th Int. Conf. Parallel Process.*, 2010, pp. 90–100.
[51] A. Bhatele, S. Brink, and T. Gamblin, "Hatchet: Pruning the overgrowth in parallel profiles," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2019, pp. 1–21.
[52] P. Kousha et al., "INAM: Cross-stack profiling and analysis of communication in MPI-based applications," in *Practice and Experience in Advanced Research Computing*. New York, NY, USA: ACM, 2021, pp. 1–11.
[53] D. C. Arnold, D. H. Ahn, B. R. De Supinski, G. L. Lee, B. P. Miller, and M. Schulz, "Stack trace analysis for large scale debugging," in *Proc. 2007 IEEE Int. Parallel Distrib. Process. Symp.*, 2007, pp. 1–10.
[54] "NVIDIA Nsight," 2024. [Online]. Available: https://developer.nvidia.com/nsight-systems
[55] C. January, J. Byrd, X. Oró, and M. O'Connor, "Allinea map: Adding energy and openmp profiling without increasing overhead," in *Tools for High Performance Computing*. Berlin, Germany: Springer, 2015, pp. 25–35.
[56] S. Kaufmann and B. Homer, "Craypat-cray X1 performance analysis tool," Cray User Group (May 2003), 2003. [Online]. Available: https://cpe.ext.hpe.com/docs/performance-tools/index.html
[57] J. R. Madsen et al., "Timemory: Modular performance analysis for HPC," in *Proc. 35th Int. Conf.*, 2020, pp. 434–452.
[58] H. Wang, J. Zhai, X. Tang, B. Yu, X. Ma, and W. Chen, "Spindle: Informed memory access monitoring," in *Proc. 2018 Annu. Tech. Conf.*, 2018, pp. 561–574.
[59] I. Laguna et al., "Debugging high-performance computing applications at massive scales," *Commun. ACM*, vol. 58, no. 9, pp. 72–81, 2015.
[60] B. Zhou, M. Kulkarni, and S. Bagchi, "Vrisha: Using scaling properties of parallel programs for bug detection and localization," in *Proc. 20th Int. Symp. High Perform. Distrib. Comput.*, 2011, pp. 85–96.
[61] I. Laguna et al., "Large scale debugging of parallel tasks with automaded," in *Proc. 2011 Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, Art. no. 50.
[62] S. Mitra, I. Laguna, D. H. Ahn, S. Bagchi, M. Schulz, and T. Gamblin, "Accurate application progress analysis for large-scale parallel debugging," *ACM SIGPLAN Notices*, vol. 49, pp. 193–203, 2014.
[63] C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko, "Scalability analysis of SPMD codes using expectations," in *Proc. 21st Annu. Int. Conf. Supercomputing*, 2007, pp. 13–22.
[64] J. Chen and R. M. Clapp, "Critical-path candidates: Scalable performance modeling for MPI workloads," in *Proc. 2015 IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2015, pp. 1–10.
[65] X. Liu and B. Wu, "Scaanalyzer: A tool to identify memory scalability bottlenecks in parallel programs," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015, Art. no. 47.

Yuyang Jin received the BS degree in computer science from the Beijing Institute of Technology, Beijing, in 2017, and the PhD degree in computer science from the Tsinghua University, Beijing, in 2022. He is a postdoctoral researcher with the Department of Computer Science and Technology, Tsinghua University, Beijing. His research interests include performance analysis and optimization for parallel applications.

Haojie Wang received the BS and PhD degrees from Tsinghua University, Beijing, in 2015 and 2021, respectively. He is an assistant researcher with the Department of Computer Science and Technology, Tsinghua University, Beijing. He worked with Google Inc. as an intern in the summer of 2017, and worked as a research assistant with Qatar Computing Research Institute in 2018. His research interests include compiler, program analysis, and AI compiler.

Xiongchao Tang received the BS and PhD degrees from Tsinghua University, in 2013 and 2019, respectively. He is the CEO and co-founder of Qingcheng.AI company. His major research interests include performance analysis and performance optimization for multi-core systems and large-scale clusters.

Zhenhua Guo received the PhD degree from Harbin Engineering University, China. He is currently a researcher with Inspur Electronic Information Industry Company Ltd. His research interests mainly include computer system architecture and artificial intelligence.

Yaqian Zhao is currently a senior engineer with Inspur Electronic Information Industry Company Ltd., Beijing, China. Her research interests include computer architecture and virtual reality.

Torsten Hoefler received the PhD degree from Indiana University. He is currently a professor of computer science with ETH Zurich, where he leads the Scalable Parallel Computing Lab. His research focuses on understanding the performance of parallel computing systems ranging from parallel computer architecture through parallel programming to parallel algorithms.

Tao Liu received the PhD degree from Beihang University, Beijing, in 2017. He is currently a researcher with the Qilu University of Technology (Shandong Academy of Sciences). His research interests include parallel computing and performance optimization.

Xu Liu received the PhD degree from Rice University, in 2014. He is an associate professor with the Department of Computer Science, North Carolina State University. His research interests include parallel computing, compiler techniques, and systems. He focuses on developing tool infrastructures to measure and analyze program executions on emerging parallel architectures.

Jidong Zhai received the BS degree in computer science from the University of Electronic Science and Technology of China, Chengdu, in 2003, and the PhD degree in computer science from Tsinghua University, Beijing, in 2010. He is a tenured professor with the Department of Computer Science and Technology, Tsinghua University, Beijing. His research interests include performance evaluation for high-performance computers, performance analysis, and modeling of parallel applications.