SCALANA: Automating Scaling Loss Detection with Graph Analysis

Yuyang Jin*, Haojie Wang*, Teng Yu*, Xiongchao Tang*, Torsten Hoefler†, Xu Liu‡, Jidong Zhai*
*Tsinghua University, †ETH Zürich, ‡North Carolina State University
{jyy17, wang-hj18}@mails.tsinghua.edu.cn, yuteng@tsinghua.edu.cn, tomxice@gmail.com,
htor@inf.ethz.ch, xliu88@ncsu.edu, zhaijidong@tsinghua.edu.cn

***Abstract*—Scaling a parallel program to modern supercomputers is challenging due to inter-process communication, Amdahl's law, and resource contention. Performance analysis tools for finding such scaling bottlenecks either base on profiling or tracing. Profiling incurs low overheads but does not capture detailed dependencies needed for root-cause analysis. Tracing collects all information at prohibitive overheads.**

In this work, we design SCALANA that uses static analysis techniques to achieve the best of both worlds - it enables the analyzability of traces at a cost similar to profiling. SCALANA first leverages static compiler techniques to build a *Program Structure Graph*, which records the main computation and communication patterns as well as the program's control structures. At runtime, we adopt lightweight techniques to collect performance data according to the graph structure and generate a *Program Performance Graph*. With this graph, we propose a novel approach, called *backtracking root cause detection*, which can automatically and efficiently detect the root cause of scaling loss. We evaluate SCALANA with real applications. Results show that our approach can effectively locate the root cause of scaling loss for real applications and incurs 1.73% overhead on average for up to 2,048 processes. We achieve up to 11.11% performance improvement by fixing the root causes detected by SCALANA on 2,048 processes.

***Index Terms*—Performance Analysis, Scalability Bottleneck, Root-Cause Detection, Static Analysis**

I. INTRODUCTION

A decade after Dennard scaling ended and clock frequencies have stalled, increasing core count remains the only option to boost computing power. Top-ranked supercomputers [1] already contain millions of processor cores, such as ORNL's Summit with 2,397,824 cores, LLNL's Sierra with 1,572,480 cores, and Sunway TaihuLight with 10,649,600 cores. This unprecedented growth in the last years shifted the complexity to the developers of parallel programs, for which scalability is a main concern now. Unfortunately, not all parallel programs have caught up with this trend and cannot efficiently use modern supercomputers, mostly due to their poor scalability [2], [3].

Scalability bottlenecks can have a multitude of reasons ranging from issues with locking, serialization, congestion, load imbalance, and many more [4], [5]. They often manifest themselves in synchronization operations and finding the exact root cause is hard. Yet, with the trend towards larger core count continuing, scalability analysis of parallel programs becomes one of the most important aspects of modern performance engineering. Our work squarely addresses this topic for large-scale parallel programs.

TABLE I: Qualitative performance and storage analysis on state-of-the-art and SCALANA running NPB-CG with CLASS C for 128 processes [6]

Tools	Approaches	Time Overhead	Storage Cost
Scalasca [7]	Tracing-based	25.3% (w/o I/O)	6.77 GB
HPCToolkit [8]	Profiling-based	8.41%	11.45 MB
SCALANA	Graph-based	3.53%	314 KB

Researchers have made great efforts in scalability bottleneck identification using three fundamental approaches: application profiling, tracing, and modeling.

Profiling-based approaches [9], [10], [11] collect statistical information at runtime with low overhead. Summarizing the data statistically loses important information such as the order of events, control flow, and possible dependence and delay paths. Thus, such approaches can only provide a coarse insight into application bottlenecks and substantial human efforts are required to identify the root cause of scaling issues.

Tracing-based approaches [12], [13], [7], [14] capture performance data as time series, which allows tracking dependence and delay sequences to identify root causes of scaling issues. Their major drawback is the often prohibitive storage and runtime overhead of the detailed data logging. Thus, such tracing-based analysis can often not be used for large-scale programs. For example, we show the performance and storage overhead of the NPB-CG running with 128 processes comparing with tracing and profiling in Table I (Note that it is a single run for overhead comparison but not a typical use-case for scalability bottleneck identification.). Both profiling-based approaches and tracing-based approaches can use sampling techniques to reduce overhead but with a certain accuracy loss.

Modeling-based approaches [15], [16], [17], [18], [19], [20], [21] can also be used to identify scalability bottlenecks with low runtime overhead. However, building accurate performance models often requires significant human efforts and skills. Furthermore, establishing full performance models for a complex application with many input parameters requires many runs and prohibitively expensive [22]. Thus, we conclude that identifying scalability bottlenecks for large-scale parallel programs remains an important open problem.

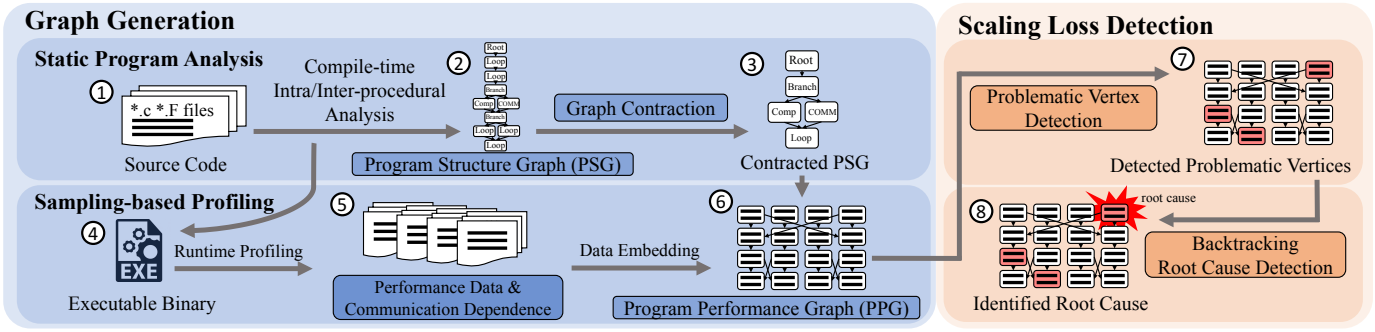


Fig. 1: Overview of SCALANA

To accurately identify scalability problems with low effort and overhead, we consider the program structure during data profiling. SCALANA combines **static program analysis with dynamic sampling-based profiling** into a light-weight mechanism to automatically identify the root cause of scalability problems for large-scale parallel programs. We utilize an intra- and inter-procedural analysis of the source-code structure and record dynamic message matching at runtime to establish an efficient dependence graph of the overall execution. In particular, SCALANA is able to detect latent scaling issues in complex parallel programs where the delay will propagate to the other processes after several time steps through massive communication dependence. In summary, there are three main contributions in our work:

- We design a fine-grained *Program Structure Graph* (PSG) that represents a compressed form of all program dependence within and across parallel processes. Then we generate *Program Performance Graph* (PPG) by enhancing PSG for each execution combining static compile-time analysis with light-weight runtime profiling.
- Based on the PPG, we design a location-aware algorithm to detect problematic vertices with scaling issues. Combining inter-process dependence chains, we further propose a novel graph analysis algorithm, called *backtracking root cause detection*, to find their root cause in source code.
- We implement a light-weight performance tool named SCALANA[1], and evaluate it with real applications. Results show that SCALANA can effectively and automatically identify the root cause of scalability problems.

We evaluate SCALANA with both benchmarks and real applications. Experimental results show that our approach can identify the root cause of scalability problems for real applications more accurately and effectively comparing with HPCToolkit [8] and Scalasca [7]. SCALANA only incurs 1.73% overhead on average for evaluated programs up to 2,048 processes. We achieve up to 11.11% performance improvement by fixing the root causes detected by SCALANA on 2,048 processes.

II. DESIGN OVERVIEW

One main innovation of SCALANA is to build a *Program Structure Graph* (PSG) at compile time and use it during runtime to minimize tracing overheads. The PSG captures the main computation and communication patterns that can be extracted statically from a parallel program. During the execution, SCALANA collects light-weight performance data as PSG vertex attributes as well as communication dependence between different processes and finally forms a *Program Performance Graph* (PPG). Another innovation of SCALANA is that we leverage the features of the generated PPG to locate problematic vertices and then we use graph analysis to automatically identify the root cause of scaling issues in the source code.

In general, SCALANA consists of two main modules, *graph generation* and *scaling loss detection*. Figure 1 shows the high-level workflow of our system. Graph generation contains two phases, static program analysis and sampling-based profiling. Static program analysis is done at compile time while the sampling-based profiling is performed at runtime. We use the LLVM compiler [23] to automatically build a PSG. Each vertex on the program structure graph is corresponding to a code snippet in the source code. The scaling loss detection is an offline module, which includes problematic vertex detection and root-cause analysis. We describe several key steps of these two modules below.

Graph Generation

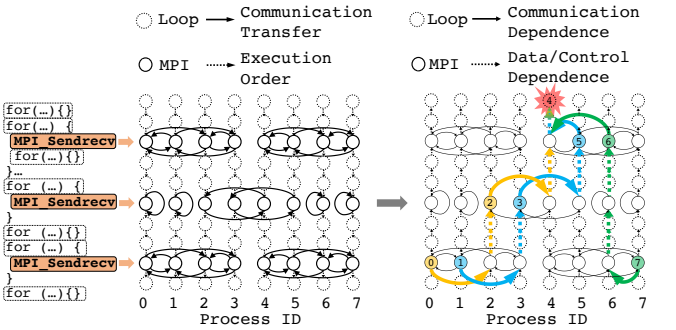
- *Program Structure Graph (PSG).* The input of this module is the source code of a parallel program. Through an intra- and inter-procedural static analysis of the program, we get a preliminary *Program Structure Graph* (Section III-A).
- *Graph Contraction.* In this step, we remove unnecessary edges in the PSG and merge several small vertices into a large vertex to reduce scalability analysis overhead (Section III-A).
- *Performance Data and Communication Dependence.* To effectively detect the scalability bottleneck, we leverage sampling techniques to collect the performance data for each vertex of the PSG and communication dependence data with different numbers of processes (Section III-B).
- *Program Performance Graph (PPG).* To analyze the interplay of computation and communication among different processes, we further generate a *Program Performance Graph* based on per-process PSGs (Section III-C).

Scaling Loss Detection

- *Problematic Vertex Detection.* According to the structure

[1]SCALANA is available at: *https://github.com/thu-pacman/SCALANA*.

of the acquired PPG, we design a location-aware detection approach to identify all problematic vertices (Section IV-A).
- *Backtracking Root Cause Detection.* Combined with identified problematic vertices, we propose a backtracking algorithm on top of the PPG and identify all the paths covering problematic vertices, which can help locate the root cause of the scaling issues (Section IV-B).



(a) NPB-CG (b) Partial PPG of NPB-CG (c) Backtracking root cause detection

Fig. 2: Motivating example with NPB-CG

We give an example to show how our approach is used to detect scaling loss. Figure 2 shows the code snippet of NPB-CG [6] and the partial PPG of NPB-CG generated by SCALANA (due to space limitation, we only draw the PPG for 8 processes). We manually inject a delay in one process (process 4), which causes a scaling loss on the Tianhe-2 system (49.4 seconds at 1,024 processes vs. 49.5 seconds at 2,048 processes). Tracing-based approaches like Scalasca [7] and Vampir [24] generate more than 250 GB of trace data. Due to the covert performance issue mixed by data, control, and inter-process communication dependence, we observe that the traditional profiling-based tool, like HPCToolkit [8], needs significant human efforts to identify the accurate root cause for this case.

In SCALANA, we leverage both the static and dynamic analysis to build a holistic PPG that records program execution order and data flow as well as inter-process communication transfer, as shown in Figure 2(b). With our detection algorithm, we first identify some problematic vertices in this graph. For example, the vertices are marked with red, blue, yellow, or green color. In general, a problematic vertex is a vertex with unusual performance relative to other vertices. Then we perform a backtracking root cause detection on the PPG of NPB-CG, as shown in Figure 2(c). Through backward traversing this graph, we can detect the red vertex of process 4 is the root cause through a path of vertices that traverses different processes.

In summary, SCALANA is a programmer-oriented scalability analysis tool, which takes input as the source code of a parallel program, detects the root cause of scaling bottlenecks and reports back to the programmer which lines of the source code cause the problems to guide further optimization on the program.

III. GRAPH GENERATION

In this section, we describe how we automatically build an appropriate representation to reflect the main computation and communication characteristics for a given parallel program in detail. Our approach mainly relies on a static program analysis module. It also incorporates a sampling-based profiling module to handle input-dependent information.

A. Static Program Structure Graph Construction

In general, the static analysis module is in charge of building a per-process PSG, which can be regarded as a sketch of a parallel program. In a PSG, the vertices represent main computation and communication components as well as program control flow. The edges represent their execution order based on both data and control flow. We group the vertices into different types, including *Branch*, *Loop*, *Function call*, and *Comp*, among which, *Comp* is a collection of computation instructions while the others are basic program structures.

There are three main phases to build a PSG statically: intra-procedural, inter-procedural analysis, and graph contraction. During the intra-procedural analysis, we firstly build a local PSG for each function. And then through an inter-procedural algorithm, we acquire a complete PSG, which will be further refined by graph contraction.

Intra-procedural Analysis During the intra-procedural analysis phase, we build a local PSG for each procedure. The basic idea is that we traverse the control flow graph of the procedure at the level of the intermediate representation (IR) of the program, identify loops, branches, and function calls, and then connect these components based on their dependence to form a per-function local graph.

Inter-procedural Analysis Inter-procedural analysis is to combine all the local PSGs into a complete graph. We start by analyzing the program's call graph (PCG), which contains all calling relationships between different functions. And then we perform a top-down traversal of the PCG from the `main` function and replace all user-defined functions with their local PSGs. For MPI function calls, we just keep them. For indirect function calls, we need to process them after collecting certain function call relationships at runtime. For recursive function calls, their edges are similar to the recursive call edges in the PCG, which means that a circle is formed in the PSG. After the static analysis, the runtime performance data will be attached to these vertices with extra call-stack information for further analysis.

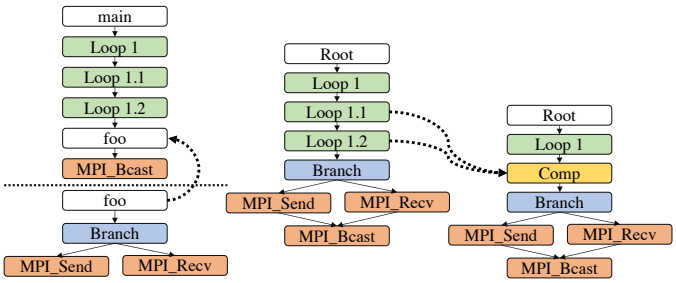
PSG Contraction The PSGs generated in the above step are normally too large to be applied efficiently for real applications since we need to create corresponding vertices for any loop and branch in their source code. However, the workload of some vertices can be ignored as collecting performance data for these vertices only introduces large overhead without benefits. To address this problem, SCALANA performs graph contraction to reduce the size of the generated PSG.

The rules of contraction affect the granularity of the graph and the representation of communication and computation characteristics. Considering that communication is

normally the main scalability bottleneck for parallel programs, SCALANA preserves all MPI invocations and related control structures. For computation vertices in the PSG, we merge continuous vertices into a larger vertex. Specifically, for the structures that do not include MPI invocations, we only preserve *Loop* because computation produced by loop iterations may dominate performance. In addition, SCALANA allows a user-defined parameter, `MaxLoopDepth`, as a threshold to limit the depth of nested loops and keep the graph condensed.

```
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



(a) The local PSG generated by intra-procedural analysis

(b) The complete PSG generated by inter-procedural analysis

(c) The final contracted PSG

Fig. 4: Static Program Structure Graph Generation

For instance, Figure 3 shows a simple MPI program example with two functions. Figure 4(a) shows its local PSGs for each function after the intra-procedural analysis. Figure 4(b) shows a complete PSG after the inter-procedural analysis. Figure 4(c) shows the contracted PSG after merging the sequential `Loop1.1` and `Loop1.2` when `MaxLoopDepth` is set to 1.

B. Sampling-Based Profiling

SCALANA is a hybrid approach. We design a sampling-based profiling module to annotate the PSG with profiling data and also refine it based on runtime information. The sampling-based profiling module includes performance profiling, inter-process dependence profiling, and indirect call analysis. Performance profiling is to collect and fill runtime metrics into the vertices of the graph to handle input-dependent information. We use a sampling technique for performance profiling (Section III-B1). Inter-process dependence profiling is to connect per-function PSG into a larger graph (PPG) that cannot be derived statically (Section III-B2).

1) Associate Vertices with Performance Data: We collect the performance data for each vertex of the PSG at runtime, which is essential for further analysis of scaling issues. Unlike traditional coarse-grained profiling approaches, SCALANA collects performance data according to the granularity of each PSG vertex. One main advantage is that we can combine the graph structure and performance data for more accurate performance analysis. Specifically, we associate each PSG vertex with a performance vector that records the execution time and key hardware performance data, such as cache miss rate and branch miss count.

We use sampling techniques for performance profiling to collect metrics with very low overhead. We use PAPI [25] for sampling and hardware performance data collection, which interrupts the program at regular clock cycles and records program call stack and related performance data. According to the program call stack information, we can associate performance data with the corresponding PSG vertex at the interruption point.

2) Graph-Guided Communication Dependence: During the static analysis, we derive data and control dependence within each process. At runtime, we need to further collect communication dependence between different processes for inter-process dependence analysis. Traditional tracing-based approaches record each communication operation and analyze their dependence, which causes large collection overhead and also huge storage cost [26], [27]. We propose two key techniques to address this problem: sampling-based instrumentation and graph-guided communication compression.

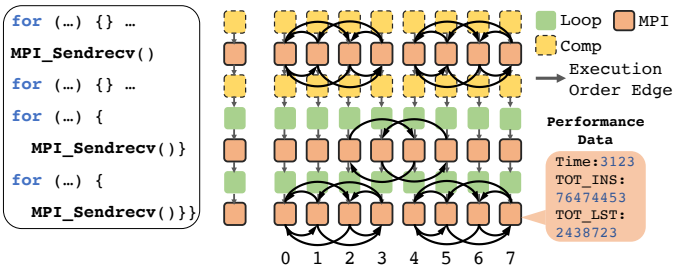
Sampling-Based Instrumentation Full instrumentation always introduces large overheads. The dynamic program behavior may be missed if the instrumentation is recorded only once. To reduce the runtime overhead and still capture the dynamic program behavior along with the program execution, we adopt a random sampling-based instrumentation technique [28]. A random number is generated every time when the instrumentation is executed. When the random number falls into an interval of the pre-defined threshold we record communication parameters. The random sampling technique used here can avoid missing regular communication patterns as much as possible even if they change at runtime.

Graph-Guided Communication Compression A typical parallel program contains a large number of communication operations. Due to the redundancy between different loop iterations, we do not need to record all the communication operations. As the PSG already represents the program's high-level communication structure, we can leverage this graph to reduce communication records. We only record communication operation parameters once for repeated communications with the same parameters of the recorded data, which can reduce the storage cost and ease the analysis of inter-process dependence.

We use PMPI [29] in this work for effective communication

```
map <MPI_Request*, pair<int,int>> requestConverter;
int MPI_Irecv(..., int source, int tag,
                        ..., MPI_Request *request){
    requestConverter[request] = <source,tag>;
    return PMPI_Irecv(...);
}
int MPI_Wait(MPI_Request *request, MPI_Status *status){
    retval = PMPI_Wait(request, status);
    <source,tag> = requestConverter[request];
    if (source or tag is uncertain) {
        commSet.insert(<status.MPI_SOURCE,status.MPI_TAG>);
    } else {
        commSet.insert(<source, tag>);
    }
    return retval;
}
```

Fig. 5: Acquiring communication dependence for non-blocking communications



(a) code (b) PSG (c) the PPG on 8 processes

Fig. 6: A PPG running with 8 processes

collection, which does not need to modify the source code. For different communication types, we adopt different methods to collect their dependence. We distinguish three common classes of communication: (1) For collective communication, we should know which processes are involved in this communication. In MPI programs, we can use `MPI_Comm_get_info` to acquire this information. (2) For blocking point to point communication, we should record the *source* or *dest* process and *tag* directly. (3) For non-blocking communication, some information will not be available until final checking functions are invoked (such as `MPI_Wait`).

We take `MPI_Wait` after `MPI_Irecv` as an example as shown in Figure 5. Firstly, we store the *source* process and *tag* from the parameters associated to the *request* in `MPI_Irecv`. Then in `MPI_Wait`, the *source* and *tag* corresponding to the *request* are recorded into a communication dependence set. If the *source* or *tag* is uncertain, we acquire them from the parameter of *status* in `MPI_Wait`.

3) Indirect Function Calls: Sometimes, the program call graph cannot be fully obtained by the static analysis due to indirect calls, such as function pointers. We need to collect the calling information of indirect calls at runtime and fill such information into the graph. We do necessary instrumentation before the entry and exit of indirect calls and link this information with real function calls with unique function IDs and then refine the PSG obtained after the inter-procedural analysis.

C. Program Performance Graph

After both the static program analysis and sampling-based profiling, we build a final *Program Performance Graph* (PPG). As each process shares the same source code, we can duplicate the PSG for all processes. Then we add inter-process edges based on communication dependence collected at the runtime analysis. For point to point communications, we match the sending and receiving processes. For collective communications, we associate all involved processes. Figure 6 shows a simplified final PPG for an example program running with 8 processes.

Note that the final PPG not only includes the data and control dependence for each process but also records the inter-process communication dependence. In addition, we also attribute key performance data for each vertex, which will be used for further scaling issue detection. For a given vertex in this graph, its performance can be affected by either its own computation patterns or the performance of other vertices connected through data and control dependence within one process as well as communication dependence between different processes. We describe how we locate the performance issue below.

IV. Scaling Loss Detection

In this section, we describe how we leverage the acquired PPG for effective and automatic scaling loss detection. Our approach consists of two key steps, location-aware problematic vertex detection and backtracking root cause identification. The former is to detect problematic vertices with poor scalability or abnormal behavior. The latter is to pinpoint the root cause of scaling loss problems.

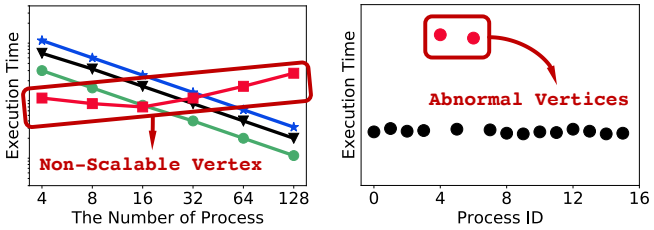
A. Location-Aware Problematic Vertex Detection

One main advantage of our approach is that we have generated a final PPG from a given program. Although the inter-process communication dependence may change with the different numbers of processes, the per-process PSG does not change with the problem size or job scale. Based on this observation, we propose a location-aware detection approach to identify problematic vertices. The core idea of our approach is that we compare the performance data of vertices in the PPG which corresponds to the same vertex in the PSG among different job scales (non-scalable vertex detection) and different processes for a given job scale (abnormal vertex detection).

Non-Scalable Vertex Detection The core idea is to find vertices in the PPG whose performance (execution time or hardware performance data) shows an unusual slope comparing with other vertices when the number of processes increases. For instance, Figure 7(a) shows the change of the execution time of different vertices in a PSG as the process count increases. The execution time of the vertex in the red line does not decrease like other vertices. When the execution time of these vertices accounts for a large proportion of the total time, they will become a scaling issue.

A challenge for non-scalable vertex detection is how to merge performance data from a large number of processes. The simplest strategy is to use the performance data for a

particular process for comparison but this strategy may lose some information about other processes. Another strategy is to use the mean or median value of performance data from all processes and the performance variance among different processes to reflect load distribution. We can also partition all processes into different groups by clustering algorithms and then aggregate for each group. In our implementation, we test all strategies mentioned above and fit the merged data of different process counts with a log-log model [30]. With these fitting results, we sort all vertices by the changing rate of each vertex when the scale increases and filter the top-ranked vertices as the potential non-scalable vertices.



(a) A non-scalable vertex example (b) An abnormal vertex example

Fig. 7: Two kinds of problematic vertices examples

Abnormal Vertex Detection For a given job scale, we can also compare the performance data of the same vertex among different processes. Since for typical SPMD (Single Program Multi-Data) programs, the same vertex tends to execute the same workload among different processes. If a vertex has significantly different execution time, we can mark this vertex as a potential abnormal vertex. A lot of reasons can cause abnormal vertices, even if we do not consider the effect of performance variance [31]. For instance, a load balance problem can cause abnormal vertices in some processes. We can also identify some communication vertices that have much larger synchronization overhead than other processes. Figure 7(b) shows the execution time of the vertices of all processes in a PPG which correspond to the same vertex in the PSG on 16 processes. Among them, process 4 and 6 take longer to execute than the others and yield the abnormal vertices. ScalAna allows a user-defined threshold `AbnormThd` to distinguish both abnormal and normal vertices among different parallel processes. We discuss details in Section VI-D.

As shown in Figure 8, after the analysis of the above two steps, we mark some problematic vertices in the PPG (vertices with blue and red color) generated in Figure 6.

B. Backtracking Root Cause Detection

Furthermore, we need to connect the identified problematic vertices and find the causal relationship between them to locate the root cause of the scaling problem. In this work, we use graph analysis to propose a novel approach, named backtracking root cause detection to automatically report the line number of source code corresponding to the root cause.

To do the backward traversal, first we need to reverse all edges to dependence edges. The pseudo-code of *backtracking root cause algorithm* is shown in Algorithm 1. Our algorithm starts from the non-scalable vertices detected in the above step, then tracks backward through data/control dependence edges within a process and communication dependence edges among different processes until the root vertices or collective communication vertices are found. If unscanned *Loop* or *Branch* vertices are found during the backtrack, our algorithm will traverse only the control dependence edges but not the data dependence edges. For example, when a *Loop* vertex is found, the traversal continues from the end vertex of this loop. One observation is that a complex parallel program always contains a large number of dependence edges. So the search cost will be very high if we would not optimize. However, we do not need to traverse all the possible paths to identify the root cause. In ScalAna, we only preserve the communication dependence edge if a waiting event exists while we prune other communication dependence edges. The advantage of our approach is that we can reduce both searching space and false positives. Finally, we get several causal paths that connect a set of abnormal vertices. Further analysis of these identified paths will help application developers to locate the root cause.

Algorithm 1: Backtracking Root Cause Algorithm

Input: A Program Performance Graph PPG, A Set of Non-Scalable Vertices $\mathbb{N}$, A Set of Abnormal Vertices $\mathbb{A}$.
Output: A Set of Root Cause Paths $\mathbb{S}$.

```
1  Function Main():
2      S ← ∅ ;
3      V ← ∅ ; // Set of scanned vertices
4      forall n ∈ N do
5          P ← ∅ ; // Root cause path
6          Backtracking ( n, P ) ;
7          Insert P into S ;
8          Insert all v ∈ P into V ;
9      forall a ∈ A and a ∉ V do // Traverse the vertices
         that have not been scanned
10         P ← ∅;
11         Backtracking ( a, P ) ;
12         Insert P into S ;
13     return S ;
14 Function Backtracking(v, P):
15     while v is not root or collective communication vertex do
16         Insert v into P ;
17         if v is an MPI vertex then
18             v ← the dest vertex of inter-process communication
                 dependence edge of v ;
19         else if v is an unscanned LOOP or BRANCH vertex then
20             v ← the dest vertex of control dependence edge of v;
21         else
22             v ← the dest vertex of data dependence edge of v ;
```

For example, in Figure 8, we start from the abnormal vertex a in the lower-left corner, and track through a communication dependence edge to vertex b in process 2. Then we can 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 abnormal vertices in the processes of 0, 2, and 4. With a similar approach, we backtrack from the other two abnormal 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 abnormal vertices including MPI invocations and computation components together, and identify the root cause of scaling loss.

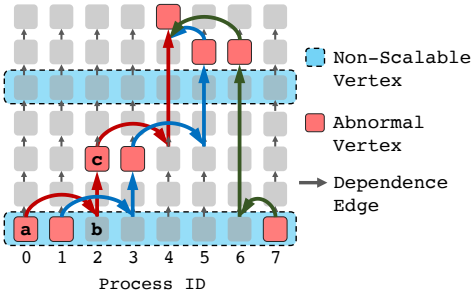


Fig. 8: Problematic vertices and backtracking root cause detection on PPG

Note that some vertices may be both non-scalable and abnormal vertices. The interplay of non-scalable and abnormal vertices can make the program performance even harder to understand. Sometimes, optimizing the performance of some vertices in identified paths can also improve the overall performance of the non-scalable vertex.

V. Implementation and Usage

For the static analysis module of ScalAna, we use LLVM-3.3.0 and Dragonegg-3.3.0 [23] for PSG generation and program instrumentation. For the sampling-based profiling, we use PAPI-5.2.0 [25], [32] to collect hardware performance data, and the PMPI interface to collect communication dependencies. With both static PSG and dynamic profiling data, ScalAna generates PPGs and performs scaling loss detection post-mortem.

In general, there are four main steps for end-users to use ScalAna: (1) Compiling applications with `ScalAna-static` to generate the PSG. (2) Running the instrumented applications with `ScalAna-prof` for different process numbers to collect profiling data. (3) Using `ScalAna-detect` to automatically detect the root cause of scaling loss. (4) We also provide a GUI in ScalAna, `ScalAna-viewer`, to show the code snippets corresponding to the diagnosed root causes. Besides, users can adjust some user-defined parameters like `MaxLoopDepth` and `AbnormThd` to make a trade-off between detection precision and system overhead.

Figure 9 shows a screenshot of ScalAna's GUI. The upper window lists the root cause vertices and their calling paths. The lower window shows the code snippets corresponding to the vertices. The root causes can be further sorted according to the length of execution time and the imbalance among different parallel processes.

ScalAna currently only supports MPI-based programs in C or Fortran. However, all phases in ScalAna (program structure extraction, profiling data collection, and root-cause detection) are general enough to be adapted to other message-passing programs. In addition, our approach can be also extended to other programming models such as OpenMP or Pthreads with additional profiling techniques. We leave it for future work.

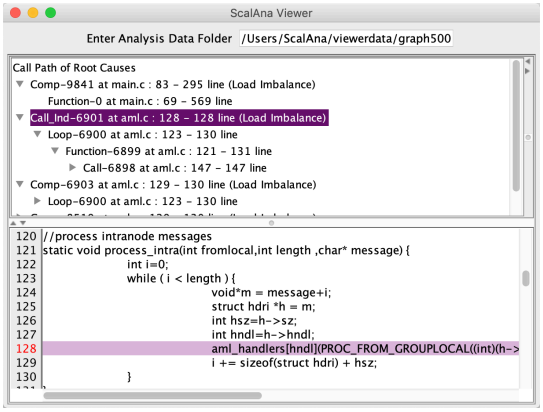


Fig. 9: GUI of ScalAna

VI. Evaluation

A. Experimental Setup

Experimental Platforms We perform the experiments on two testbeds: (1) Gorgon, a cluster with dual Intel Xeon E5-2670(v3) and 100Gbps 4xEDR Infiniband. (2) Tianhe-2 supercomputer. Each node of Tianhe-2 has two Intel Xeon E5-2692(v2) processors (24 cores in total) and 64 GB memory. The Tianhe-2 supercomputer uses a customized high-speed interconnection network.

Evaluated Programs We use a variety of parallel programs to evaluate the efficacy of ScalAna, including BT, CG, SP, EP, FT, MG, LU, and IS, from the widely used NPB benchmark suite [6], plus three real world applications, Zeus-MP [33], SST [34], and Nekbone [35]. For NPB programs, problem size CLASS C is used on Gorgon and CLASS D is used on the Tianhe-2 supercomputer.

Methodology In our evaluation, we first analyze the common features of the generated program structure graphs for each program, and then we present the performance overhead of our tool, including runtime overhead and storage cost (These experiments are for overhead comparison but not typical use-cases for scaling issue detection.). Finally, we use three real applications to demonstrate the benefit of our approach. For all experiments, we run three times and average the results for each process scale to reduce performance variance.

We compare our approach with two state-of-the-art performance tools, HPCToolkit [8] and Scalasca [7]. To ensure the fairness of comparison, we give the detailed configuration of these two state-of-the-art tools: (1) For the tracing-based tool, Scalasca(v2.5), we first use its profiling function to identify where detailed tracing is needed, then we run small scale jobs with limited instrumentation. And increase the process count and the instrumentation complexity iteratively until the scalability bottlenecks are identified. In this way, Scalasca introduces as little storage cost as possible. (2) For the profiling-based tool, HPCToolkit(v2019.08), the sampling frequency is the key parameter that affects the runtime overhead. ScalAna

keeps the same sampling frequency (200Hz) as HPCToolkit in all experiments. For all experiments, `MaxLoopDepth` is set to 10 and `AbnormThd` is set to 1.3 empirically.

TABLE II: Code size and vertices information of PSG for evaluated programs. #VBC and #VAC are the number of vertices in the PSG before and after contraction, while #*Loop*, #*Branch*, #*Comp*, and #*MPI* are the number of *Loop*, *Branch*, *Comp*, and *MPI* vertices respectively.

Program	Code (KLoc)	#VBC	#VAC	#*Loop*	#*Branch*	#*Comp*	#*MPI*
BT	9.3	974	377	39	57	176	103
CG	2.0	431	190	18	10	95	66
EP	0.6	91	32	4	2	13	12
FT	2.5	4,285	241	15	22	118	35
MG	2.8	7,842	1,973	177	233	942	463
SP	5.1	734	278	13	34	138	89
LU	7.7	2,370	663	18	66	327	237
IS	1.3	240	55	1	3	28	19
SST	40.8	23,608	5,217	321	641	1,434	1,303
NEKBONE	31.8	1,289	944	239	162	423	83
ZEUS-MP	44.1	273,715	64,570	1,677	1,304	30,099	11,818

B. PSG Analysis

Table II summarizes the code size and the vertices count for all generated PSGs. Results include the number of lines of source code, the number of vertices before and after graph contraction, the number of *Comp* vertices, and the number of *MPI* vertices. In our experiments, the total vertex count correlates with the number of lines of source code in most cases. Graph contraction reduces the number of vertices by 68% on average. Furthermore, *Comp* and *MPI* vertices make up more than 73% of all vertices, which indicates that the PSG can fully represent computation and communication characteristics.

C. Performance Overhead

We evaluate SCALANA on the Tianhe-2 supercomputer with up to 2,048 processes and the comparison experiments with Scalasca and HPCToolkit are run on Gorgon with up to 128 processes due to the installation limitation of the Tianhe-2 supercomputer's external network.

TABLE III: The static overhead of SCALANA on Gorgon

Programs	BT	CG	EP	FT	MG	SP	LU	IS	SST	NEK	ZMP
Ovd(%)	0.32	0.77	0.38	0.35	0.29	0.31	0.28	0.68	3.01	0.43	2.96

Static Overhead We first evaluate the compilation overhead introduced by static analysis on Gorgon. As shown in Table III, SCALANA only incurs very low compilation overhead comparing to the original LLVM compilation cost (0.28% to 3.01%, 0.89% on average). Besides, the memory cost of static analysis is in proportion to the size of PSG. For example, each vertex of the PSG occupies 32B of memory on Gorgon and the static analysis incurs about 9MB in addition for Zeus-MP.

Runtime Overhead The runtime overhead of SCALANA is shown as the gray bars in Figure 10, which is the average

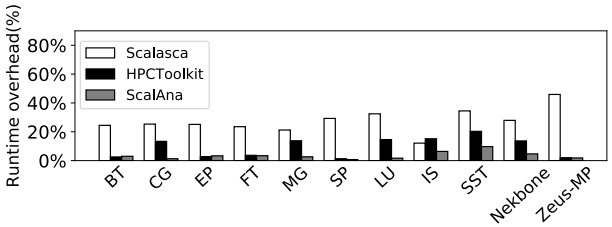


Fig. 10: Average runtime overhead of Scalasca [7], HPCToolkit [8], and SCALANA with 4 to 128 processes (without I/O)

overhead of 4 to 128 MPI processes (4 to 121 processes for BT and SP, due to its requirement for process counts). As shown in Figure 10, SCALANA only brings very small overhead ranging from 0.72% to 9.73%, average at 3.52% on Gorgon, which is much lower than Scalasca [7]. For Scalasca, the trace buffer size (`SCOREP_TOTAL_MEMORY`) is configured large enough to avoid intermediate trace flushing before the program ends. Besides, for SCALANA, the average runtime overhead of the NPB benchmark with 2,048 processes on the Tianhe-2 supercomputer is 1.73%.

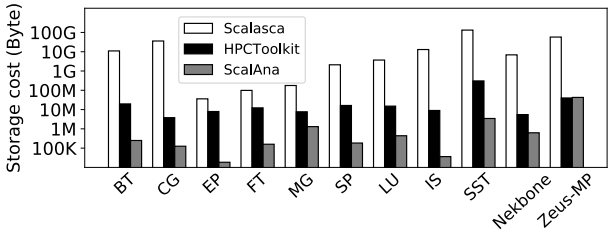


Fig. 11: Storage cost of Scalasca [7], HPCToolkit [8], and SCALANA running with 128 processes

Storage Cost Figure 11 shows the storage costs of SCALANA, HPCToolkit, and Scalasca running with 128 processes (121 for BT and SP) on Gorgon. SCALANA only incurs storage costs in the order of Kilobytes, while Scalasca and HPCToolkit generate Megabytes to Gigabytes of data. Besides, for SCALANA, the average storage cost of the NPB benchmark with 2,048 processes on the Tianhe-2 supercomputer is 4.72MB.

TABLE IV: The post-mortem detection cost of SCALANA with 128 processes

Programs	BT	CG	EP	FT	MG	SP	LU	IS	SST	NEK	ZMP
Cost(Sec.)	3.26	1.74	0.29	2.20	1.80	2.40	6.06	0.50	9.54	8.63	11.81

Post-mortem Detection Cost We evaluate the cost of backtracking root cause detection in SCALANA on Gorgon. As shown in Table IV, the scaling loss detection only introduces little cost comparing to the execution time of the program (up to 11.81 seconds, 8.44% of the execution time) on 128 processes. The memory consumption of post-mortem detection is proportional to the program structure and the size of profiling data (about 50MB for Zeus-MP on 128 processes).

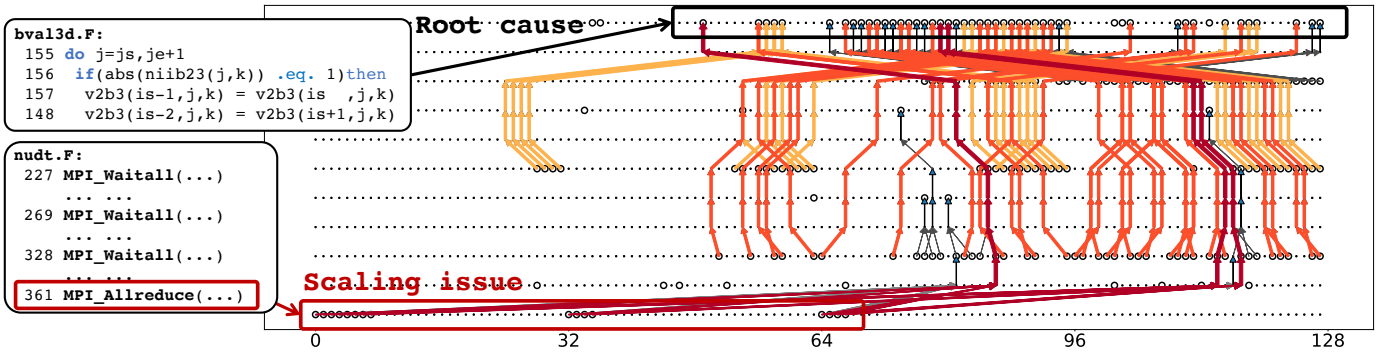


Fig. 12: Backtracking algorithm on the PPG for a Zeus-MP run with 128 processes

D. Case Studies with Real Applications

In this section, we use three real applications, Zeus-MP [33], SST [34], and Nekbone [35], to demonstrate how to diagnose scaling issues with our performance tool. When the root-causes of scaling issues are identified, we optimize the code to improve the scalability of these applications. We also analyze the advantages of our approach over the two state-of-the-art tools HPCToolkit [8] and Scalasca [7].

1) Zeus-MP: Zeus-MP [33], a computational fluid dynamics program, implements the simulation of astrophysical phenomena in three spatial dimensions using the MPI programming model. Non-blocking point-to-point (P2P) communications are used to implement complex inter-process synchronization. We evaluate its performance with a problem size of 64×64×64 for different numbers of processes ranging from 4 to 128. We observe a significant scaling loss for 128 processes and results show that the speedup is only 55.53× on 128 processes while 35.40× on 64 processes (1 process as baseline). ScalAna is then applied to diagnose the problem.

Scaling Loss Detection ScalAna first generates a PPG and then performs the backtracking algorithm on this graph to identify the root causes automatically. Figure 12 shows how ScalAna diagnoses the scaling issues on the PPG of Zeus-MP by its backtracking algorithm. The vertical axis from top to down represents the control/data flow, and the horizontal axis represents different parallel processes. The small points represent the vertices of the PPG with normal performance while the circle points represent problematic vertices with the abnormal performance for the same code snippets. The arrows show the backtracking paths based on intra- and inter-process dependence.

In detail, the `MPI_Allreduce` at *nudt.F: 361* is detected as a scaling issue due to its poor scalability for its execution time. As shown in Figure 12, the dark red (darkest color) lines track backward from the abnormal `MPI_Allreduce` vertices, then go through the intra-process dependence of control/data flow and inter-process dependence of P2P communications at *nudt.F: 328, 269, 227*. The red (lighter color) and orange (lightest color) lines indicate similar backtracking paths. Finally, the `LOOP` vertices at *bval3d.F: 155* (top row in Figure 12) are identified as the root causes of scaling issues.

We find that the underlying reason is that only some busy processes execute the `LOOP` at *bval3d.F: 155* while the others are idle with non-blocking P2P communications at *nudt.F: 227*. Delays in these processes can propagate through the non-blocking P2P communications at *nudt.F: 269* and *nudt.F: 328*. The `MPI_Allreduce` at *nudt.F: 361* synchronizes all processes and leads to the low performance of Zeus-MP.

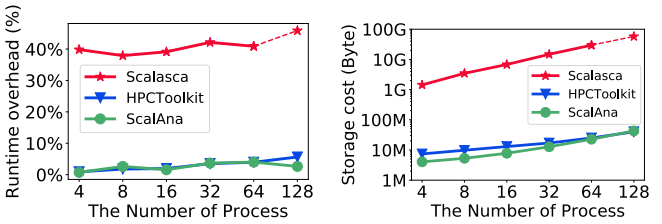
Optimization To fix the performance issue identified by ScalAna, we change the program into a hybrid programming model with MPI plus OpenMP, by adding multi-thread support at the `LOOP` of *bval3d.F: 155*, which can accelerate the busy processes and mitigate the latent load imbalance between busy processes and idle processes. Similarly, ScalAna also detects other root causes of the scaling loss from the `LOOP`s at *hsmoc.F: 665, 841, 1,041*. ScalAna shows that the load/store instruction count and the cache miss count recorded by the PMU (Performance Monitor Unit) stays high with increasing numbers of processes. We use the techniques of loop tiling and scalar promotion to reduce the cache miss and memory access. With these optimizations, the speedup of Zeus-MP is increased from 55.53× to 61.39× (1 process as baseline) on 128 processes and a 9.55% performance improvement is achieved on Gorgon.

We also test the optimized performance of Zeus-MP with a large process number. The speedup of Zeus-MP is increased from 68.41× to 76.15× (16 processes as baseline) on 2,048 processes and 9.96% performance improvement is achieved on Tianhe-2 supercomputer. Note that more optimization techniques can be further explored for Zeus-MP, but we only give some common optimizations here to verify the performance bottlenecks detected by ScalAna.

Comparison As for other state-of-the-art tools, Scalasca can accurately detect the root causes at function-level when the number of processes increases to 64 with some human intervention. The profiling-based HPCToolkit can automatically detect the fine-grained loop-level scaling issues. Specifically, the `MPI_Allreduce` at *nudt.F: 361* and the `LOOP` at *bval3d.F: 155* can be detected as scalability bottlenecks in HPCToolkit. However, profiling-based HPCToolkit cannot easily identify the root cause problem (`LOOP` at *bval3d.F: 155*) without significant human efforts. The outputs from HPCToolkit will show multiple bottlenecks without analysis on their underlying

relationship to infer which one is the actual root cause.

Figure 13 shows the performance and storage analysis of SCALANA against the state-of-the-art Scalasca and HPCToolkit. The lower is better for both Figure 13(a) and 13(b). As for performance, both SCALANA and HPCToolkit have a negligible runtime overhead by 1.85% and 2.01% on average, respectively. However, the tracing-based Scalasca introduces 40.89% runtime overhead on 64 processes (without I/O) to generate traces. For storage, our light-weight SCALANA is better than Scalasca. SCALANA only needs 20MB storage space while Scalasca generates 28.26GB traces of 64 processes.



(a) Runtime overhead (b) Storage cost

Fig. 13: Runtime and storage overhead of Scalasca [7], HPCToolkit [8], and SCALANA when running Zeus-MP (Scalasca detects the root cause when the number of processes increases to 64.)

2) SST: SST (Structural Simulation Toolkit) [34] is a multi-process simulation framework, which simulates for microarchitecture and memory in highly concurrent systems. We execute SST for different numbers of processes ranging from 4 to 128, and results show that the speedup is only 1.20× on 32 processes while 1.28× on 16 processes (4 processes as baseline). We notice that the dependence of simulated events in SST is usually complex so that most events need to be executed sequentially. The parallelism only occurs within each event in most cases, causing relatively low speedup for 32 processes. We use SCALANA to analyze the scaling loss of SST.

Scaling Loss Detection SCALANA finds that the scaling loss mainly comes from the `MPI_Allreduce` in the *RankSyncSerialSkip::exchange* function at *rankSyncSerialSkip.cc:235*. As shown in Figure 14, after backward tracking through P2P communications `MPI_Waitall` in the function *RankSyncSerialSkip::exchange* at *rankSyncSerialSkip.cc:217*, the `LOOP` in the function *RequestGenCPU::handleEvent* at *mirandaCPU.cc:247* is identified as the root cause of scaling issues. The colored lines show some backtracking paths as examples.

Optimization As shown in Figure 15, SCALANA provides the PMU data showing that the total instruction counts (`TOT_INS`) for different processes differ a lot in this loop. Based on the results of SCALANA, we find that this program uses an inefficient data structure (*array*) to process each query in a critical path for each process, which can cause different execution time (`TOT_INS`) to traverse the array for different processes. We modify the code and change the data structure

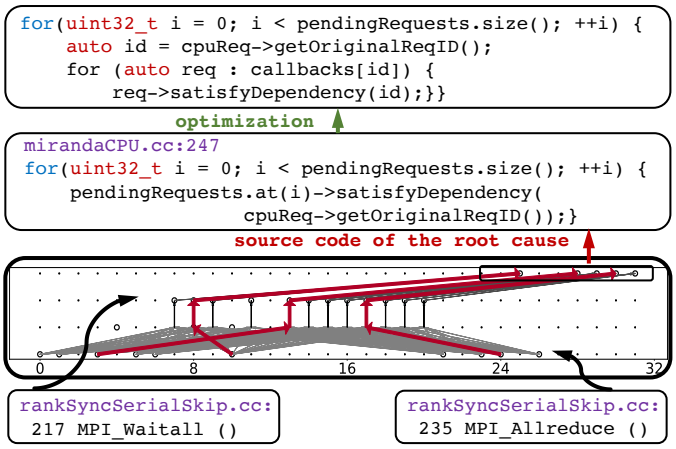


Fig. 14: Backtracking algorithm on PPG and code optimization for an SST run with 32 processes

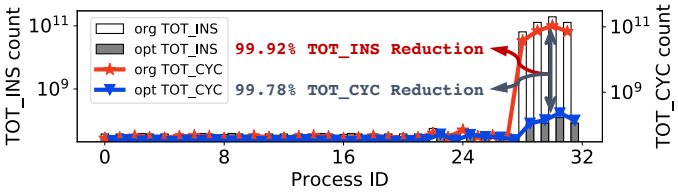


Fig. 15: PMU data for SST running with 32 processes

from *array* to *unordered_map*, which reduces the complexity of the query algorithm from *O(n)* to *O*(log*(n))* and makes the load (execution time of query) of different processes more balanced. Figure 15 also shows `TOT_INS` counts of different processes after our optimization, which are more balanced among different processes than the original SST. After the optimization, the speedup of SST for 32 processes is increased from 1.20× to 1.56× (4 processes as baseline) and the performance is improved by 73.12%.

Comparison The state-of-the-art profiling tool HPCToolkit only locates that `MPI_Waitall` is a scalability bottleneck but not the `LOOP` in the function *RequestGenCPU::handleEvent* because it does not profile on threads created at runtime, although its method is able to profile the threads theoretically. Even if it can do profiling on threads, the root cause identification still needs more human analysis. Besides, SCALANA provides the PMU data of the root causes, which makes it possible to analyze on an architecture level for developers. For storage, SCALANA only needs 1.03MB storage space while Scalasca needs 31.56GB to store the generated traces of 32 processes.

3) Nekbone: Nekbone, the basic structure of Nek5000 [35], uses a spectral element method to solve the Helmholtz equation in three-dimensional space. We execute Nekbone at the scale of 16,384 elements for the number of processes ranging from 4 to 128. Nekbone encounters a scaling issue when running on 64 processes. The speedup is only 31.95× for 64 processes while the speedup of 32 processes is 20.61× (1 process as baseline).

Scaling Loss Detection We use SCALANA to analyze the root cause of the scalability problem. `MPI_Waitall` in the function of *comm_wait* at *comm.h:243* is detected as a non-scalable vertex. Using the backtracking algorithm on the PPG through inter-process dependence, SCALANA finds that the root cause of the scaling loss is the `LOOP` in the function of *dgemm* at *blas.f:8,941*. In this loop, some processes consume significantly less time than others, which causes the waiting time of `MPI_Waitall` to increase and finally leads to the poor scalability of Nekbone.

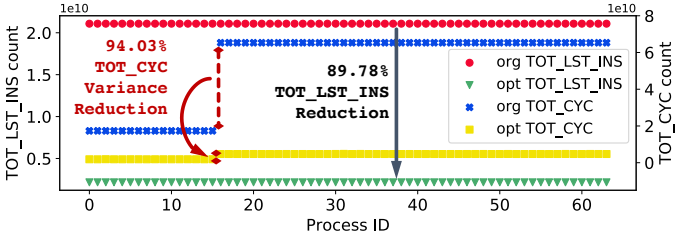


Fig. 16: PMU data for Nekbone running with 32 processes

Optimization As shown in Figure 16, the PMU data provided by SCALANA shows that the load/store instruction count (`TOT_LST_INS`) of this loop is the same among processes while the total cycle count (`TOT_CYC`) of the loop differs. We find that the memory access speed of each processor core differs, and the processes are bound to different processor cores. From the perspective of the code, we optimize it by using a more efficient linear algebra library (BLAS) to reduce the number of `TOT_LST_INS` and mitigate the time variance among processes. Figure 16 also shows that `TOT_LST_INS` decreases by 89.78%, and the execution time variance among different processes is reduced by 94.03%. After the optimization, the speedup on 64 processes is increased from 31.95× to 51.96× (1 process as baseline) and the performance is improved by 68.95%.

We also analyze the optimized performance of Nekbone with a large process number. The speedup on 2,048 processes is increased from 27.08× to 29.97× (64 processes as baseline) and 11.11% performance improvement is achieved on Tianhe-2 supercomputer.

Comparison For HPCToolkit, the `MPI_Waitall` at *comm.h: 243*, the `LOOP` at *blas.f: 8,941*, and some other points are detected as potential bottlenecks, but further manual analysis is needed to find the root cause. For storage, SCALANA only needs 0.32MB storage space while Scalasca needs 3.44GB to store the generated traces of 64 processes.

VII. RELATED WORK

Mohr [36] gives a comprehensive survey of state-of-the-art performance analysis tools including both tracing- and profiling-based methods. Knobloch et al. [37] present a sufficient survey of performance tools for heterogeneous HPC applications. In the remaining part of this section, we discuss representative related work for performance analysis in detail.

Tracing Traces are widely used for analyzing program behavior. Intel provides a trace collection tool to understand MPI program's behavior [12]. Based on Score-P infrastructure [38], [39], TAU [40], [41], Vampir [24], [42], [43], [44], Scalasca [7], [45], and some state-of-the-art tools support various programming models, such as MPI, OpenMP, Pthread, and CUDA. These tools can visualize trace data and provide fine-grained performance analysis for developers. Paraver [46], [47], [48] is a tracing-based performance analyzer that supports flexible data collection and detailed analysis of metrics variability. Becker et al. [49] use event traces to analyze the performance for large-scale programs. Though many works for trace compression are proposed [50], [51], [52], [53], tracing still often brings very large overhead which makes it non-suitable for production environments.

Profiling Profiling can extract the program's statistical information with very low overhead. mpiP [9] is a light-weight profiling library for MPI applications, which can collect statistical information for MPI functions with low overhead. Tallent et al. [10], [11] use call path profiling to identify and qualify the load imbalance for parallel programs. STAT [54] performs large scale debugging by sampling stack trace to assemble a profile for applications' behavior. HPCToolkit [8] uses sample-based techniques to get the profile performance of applications and visualize the results with *hpcviewer* and *hpctraceviewer*. Arm MAP [55] is a light-weight profiler, which is available as a part of Arm Forge debug and profile suite. Cray develops CrayPat [56], supporting both tracing and profiling performance analysis, for XC platforms. However, profiling often misses important information which may prevent us from correctly understanding the program's behavior.

Our approach uses profiling to collect dynamic statistical information, while combining it with static extracted program structure, so that we can achieve high accuracy with low overhead.

Program structure based program analysis Cypress [50] and Spindle [57] use hybrid static-dynamic analysis for communication trace compression and memory access monitoring. By extracting the program structure at compilation time, the runtime overhead can be significantly reduced. Weber et al. [58] presents effective structural similarity measure to classify and store the data for parallel programs. Program structure is also used for large scale debugging [59], [60], [61], [62], since program structure contains the dependence for both inter- and intra-process, which play an important role in large scale debugging.

Detecting scalability bottlenecks Coarfa et al. [63] identify the scalability bottlenecks by analyzing HPCToolkit's [8] hpcviewer data with a top down approach. However, it cannot deal with some communication patterns with complex dependence. Bohme et al. [64] use runtime trace to identify the root cause of wait states. As a tracing-based approach, Bohme's work performs a forward and backward trace replay on collected timeline traces. With the complete traces, delay or root causes can be accurately identified. Inspired by Bohme's backward-replay analysis, we propose a backtracking root cause detection algorithm in SCALANA. Instead of recording a large amount of traces, our approach works on the

program structure based PPG, which contains little profiling data. Therefore, SCALANA introduces very low storage cost and detection overhead. Barnes et al. [30] use regression-based approaches to perform scalability prediction. Calotoiu et al. [18] automate traditional performance modeling to detect scalability bugs. Bhattacharyya et al. [17] improve it using compiler techniques. Chen et al. [65] present a scalable performance modeling framework based on the concept of critical-path candidates for MPI workloads. ScaAnalyzer [3] collects, attributes, and analyzes memory-related metrics at runtime to identify the scalability bottlenecks caused by memory access behavior for the parallel programs running on a single node. COLAB [66] collects and accumulates futexes from Linux kernel at runtime to detect bottlenecks caused by program synchronizations.

Our work targets on detecting scalability bottlenecks using program structure combining with runtime profiling information, which helps address the root cause more accurately.

VIII. CONCLUSION

In this paper, we design SCALANA, a light-weight performance tool that can efficiently detect scalability problems of parallel programs by combining both static and dynamic analysis. SCALANA uses a novel approach to automatically identify the root cause for complex parallel programs, named backtracking root cause detection, through traversing a program performance graph. We evaluate it with both benchmarks and applications. Results show that SCALANA can efficiently identify the scalability bottlenecks with very low overhead and outperform state-of-the-art approaches.

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*. IEEE, 2012, pp. 1215–1225.

[3] X. Liu and B. Wu, "Scaanalyzer: A tool to identify memory scalability bottlenecks in parallel programs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 47.

[4] O. Pearce, H. Ahmed, R. W. Larsen, P. Pirkelbauer, and D. F. Richards, "Exploring dynamic load imbalance solutions with the comd proxy application," *Future Generation Computer Systems*, vol. 92, pp. 920–932, 2019.

[5] D. Schmidl, M. S. Müller, and C. Bischof, "Openmp scalability limits on large smps and how to extend them," Fachgruppe Informatik, Tech. Rep., 2016.

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

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

[8] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[9] J. Vetter and C. Chambreau, "mpip: Lightweight, scalable mpi profiling," 2005.

[10] N. R. Tallent, L. Adhianto, and J. M. Mellor-Crummey, "Scalable identification of load imbalance in parallel executions using call path profiles," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–11.

[11] N. R. Tallent, J. M. Mellor-Crummey, L. Adhianto, M. W. Fagan, and M. Krentel, "Diagnosing performance bottlenecks in emerging petascale applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. IEEE, 2009, pp. 1–11.

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

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

[14] 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*. Springer, 2017, pp. 161–179.

[15] H. Yin, Z. Hu, X. Zhou, H. Wang, K. Zheng, Q. V. H. Nguyen, and S. Sadiq, "Discovering interpretable geo-social communities for user behavior prediction," in *2016 IEEE 32nd International Conference on Data Engineering*. IEEE, 2016, pp. 942–953.

[16] H. Yin, B. Cui, X. Zhou, W. Wang, Z. Huang, and S. Sadiq, "Joint modeling of user check-in behaviors for real-time point-of-interest recommendation," *ACM Transactions on Information Systems*, vol. 35, no. 2, p. 11, 2016.

[17] A. Bhattacharyya, G. Kwasniewski, and T. Hoefler, "Using Compiler Techniques to Improve Automatic Performance Modeling." ACM, Oct. 2015, in proceedings of the 24th International Conference on Parallel Architectures and Compilation.

[18] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, "Using automated performance modeling to find scalability bugs in complex codes," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 45.

[19] F. Wolf, C. Bischof, A. Calotoiu, T. Hoefler, C. Iwainsky, G. Kwasniewski, B. Mohr, S. Shudler, A. Strube, A. Vogel *et al.*, "Automatic performance modeling of hpc applications," in *Software for Exascale Computing-SPPEXA 2013-2015*. Springer, 2016, pp. 445–465.

[20] D. Beckingsale, O. Pearce, I. Laguna, and T. Gamblin, "Apollo: Reusable models for fast, dynamic tuning of input-dependent code," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 307–316.

[21] J. C. Linford, J. Michalakes, M. Vachharajani, and A. Sandu, "Multi-core acceleration of chemical kinetics for simulation and prediction," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 1–11.

[22] A. Calotoiu, D. Beckinsale, C. W. Earl, T. Hoefler, I. Karlin, M. Schulz, and F. Wolf, "Fast multi-parameter performance modeling," in *2016 IEEE International Conference on Cluster Computing*, Sep. 2016, pp. 172–181.

[23] "The LLVM compiler framework." [Online]. Available: http://llvm.org

[24] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "Vampir: Visualization and analysis of mpi resources," 1996.

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

[26] X. Wu and F. Mueller, "Scalaextrap: Trace-based communication extrapolation for spmd programs," in *ACM SIGPLAN Notices*, vol. 46, no. 8. ACM, 2011, pp. 113–122.

[27] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. De Supinski, "Scalatrace: Scalable compression and replay of communication traces for high-performance computing," *Journal of Parallel and Distributed Computing*, vol. 69, no. 8, pp. 696–710, 2009.

[28] J. Vetter, "Dynamic statistical profiling of communication activity in distributed applications," *ACM SIGMETRICS Performance Evaluation Review*, vol. 30, no. 1, pp. 240–250, 2002.

[29] B. Mohr, *PMPI Tools*. Boston, MA: Springer US, 2011, pp. 1570–1575. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_57

[30] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. De Supinski, and M. Schulz, "A regression-based approach to scalability prediction," in *Proceedings of the 22nd annual international conference on Supercomputing*. ACM, 2008, pp. 368–377.

[31] X. Tang, J. Zhai, X. Qian, B. He, W. Xue, and W. Chen, "vsensor: leveraging fixed-workload snippets of programs for performance vari-

ance detection," in *ACM SIGPLAN Notices*, vol. 53, no. 1. ACM, 2018, pp. 124–136.

[32] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[33] J. C. Hayes, M. L. Norman, R. A. Fiedler, J. O. Bordner, P. S. Li, S. E. Clark, M.-M. Mac Low *et al.*, "Simulating radiating and magnetized flows in multiple dimensions with zeus-mp," *The Astrophysical Journal Supplement Series*, vol. 165, no. 1, p. 188, 2006.

[34] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis *et al.*, "The structural simulation toolkit," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 37–42, 2011.

[35] P. F. Fischer, J. W. Lottes, and S. G. Kerkemeier, "nek5000 web page," 2008.

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

[37] M. Knobloch and B. Mohr, "Tools for gpu computing–debugging and performance analysis of heterogenous hpc applications," *Supercomputing Frontiers and Innovations*, vol. 7, no. 1, pp. 91–111, 2020.

[38] D. an Mey, S. Biersdorf, C. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knüpfer, D. Lorenz, A. Malony, W. E. Nagel *et al.*, "Score-p: A unified performance measurement system for petascale applications," in *Competence in High Performance Computing 2010*. Springer, 2011, pp. 85–97.

[39] "Score-p homepage. score-p consortium." [Online]. Available: http://www.score-p.org

[40] S. S. Shende and A. D. Malony, "The tau parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[41] "Tau homepage. university of oregon." [Online]. Available: http://tau.uoregon.edu

[42] M. S. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. E. Nagel, "Developing scalable applications with vampir, vampirserver and vampirtrace." in *PARCO*, vol. 15. Citeseer, 2007, pp. 637–644.

[43] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The vampir performance analysis toolset," in *Tools for high performance computing*. Springer, 2008, pp. 139–155.

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

[45] "Scalasca homepage. julich supercomputing centre and german research school for simulation sciences." [Online]. Available: http://www.scalasca.org

[46] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris, "Dip: A parallel program development environment," in *European Conference on Parallel Processing*. Springer, 1996, pp. 665–674.

[47] H. Servat, G. Llort, J. Giménez, and J. Labarta, "Detailed performance analysis using coarse grain sampling," in *European Conference on Parallel Processing*. Springer, 2009, pp. 185–198.

[48] "Paraver homepage. barcelona supercomputing center." [Online]. Available: http://www.bsc.es/paraver

[49] D. Becker, F. Wolf, W. Frings, M. Geimer, B. J. Wylie, and B. Mohr, "Automatic trace-based performance analysis of metacomputing applications," in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–10.

[50] J. Zhai, J. Hu, X. Tang, X. Ma, and W. Chen, "Cypress: combining static and dynamic analysis for top-down communication trace compression," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 143–153.

[51] M. Noeth, F. Mueller, M. Schulz, and B. R. De Supinski, "Scalable compression and replay of communication traces in massively p arallel e nvironments," in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–11.

[52] S. Krishnamoorthy and K. Agarwal, "Scalable communication trace compression," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE Computer Society, 2010, pp. 408–417.

[53] A. Knupfer and W. E. Nagel, "Construction and compression of complete call graphs for post-mortem program trace analysis," in *2005 International Conference on Parallel Processing*. IEEE, 2005, pp. 165–172.

[54] 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 *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–10.

[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 2014*. Springer, 2015, pp. 25–35.

[56] S. Kaufmann and B. Homer, "Craypat-cray x1 performance analysis tool," *Cray User Group (May 2003)*, 2003.

[57] H. Wang, J. Zhai, X. Tang, B. Yu, X. Ma, and W. Chen, "Spindle: informed memory access monitoring," in *2018 Annual Technical Conference*, 2018, pp. 561–574.

[58] M. Weber, R. Brendel, T. Hilbrich, K. Mohror, M. Schulz, and H. Brunst, "Structural clustering: a new approach to support performance analysis at scale," in *2016 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2016, pp. 484–493.

[59] I. Laguna, D. H. Ahn, B. R. de Supinski, T. Gamblin, G. L. Lee, M. Schulz, S. Bagchi, M. Kulkarni, B. Zhou, Z. Chen *et al.*, "Debugging high-performance computing applications at massive scales," *Communications of the 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 *Proceedings of the 20th international symposium on High performance distributed computing*. ACM, 2011, pp. 85–96.

[61] I. Laguna, T. Gamblin, B. R. de Supinski, S. Bagchi, G. Bronevetsky, D. H. Anh, M. Schulz, and B. Rountree, "Large scale debugging of parallel tasks with automaded," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 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," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 193–203.

[63] C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko, "Scalability analysis of spmd codes using expectations," in *Proceedings of the 21st annual international conference on Supercomputing*. ACM, 2007, pp. 13–22.

[64] D. Bohme, M. Geimer, F. Wolf, and L. Arnold, "Identifying the root causes of wait states in large-scale parallel applications," in *2010 39th International Conference on Parallel Processing*. IEEE, 2010, pp. 90–100.

[65] J. Chen and R. M. Clapp, "Critical-path candidates: Scalable performance modeling for mpi workloads," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2015, pp. 1–10.

[66] T. Yu, P. Petoumenos, V. Janjic, H. Leather, and J. Thomson, "Colab: a collaborative multi-factor scheduler for asymmetric multicore processors," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020, pp. 268–279.