SCALANA: Automating Scaling Loss Detection with Graph Analysis

Yuyang Jin*, Haojie Wang*, Teng Yu*, Xiongchao Tang*, Torsten Hoeffer†, 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.

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.

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

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

---

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

```c
int main() {
    for (int i = 0; i < N; ++i) // Loop 1
        A[i] = rand();
    for (int j = 0; j < i; ++j) // Loop 1.1
        sum += A[j];
    for (int k = 0; k < i; ++k) // Loop 1.2
        product += A[k];
    foo();
    MPI_Bcast(...);
}
void foo() {
    if (myRank == 0)
        MPI_Send(...);
    else
        MPI_Recv(...);
}
```

![Fig. 3: An MPI program example](image)

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 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
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 PGG 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 PPG 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 PPG 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

![Fig. 6: A PPG running with 8 processes](image)

```cpp
for (-) { -
    MPI_Sendrecv()
    for (-) { -
        for (-) {
            MPI_Sendrecv()
            for (-) {
                MPI_Sendrecv()
            }
        }
    }
}
```
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.

**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 N, A Set of Abnormal Vertices A.

**Output:** A Set of Root Cause Paths S.

**Function Main:**

1. $S \leftarrow \emptyset$;
2. $V \leftarrow \emptyset$; // Set of scanned vertices
3. for all $n \in N$ do
4.   $P \leftarrow \emptyset$; // Root cause path
5.   Backtracking ($n, P$);
6.   Insert $P$ into $S$;
7. end for
8. Insert all $v \in P$ into $S$;
9. for all $a \in A$ and $a \notin V$ do // Traverse the vertices that have not been scanned
10.   $P \leftarrow \emptyset$;
11.   Backtracking ($a, P$);
12. end for
13. return $S$;

**Function Backtracking:**

14. while $v$ is not root or collective communication vertex do
15.   Insert $v$ into $P$;
16.   if $v$ is an MPI vertex then
17.     $v \leftarrow$ the dest vertex of inter-process communication dependence edge of $v$;
18.   else if $v$ is an unscanned LOOP or BRANCH vertex then
19.     $v \leftarrow$ the dest vertex of control dependence edge of $v$;
20.   end if
21. end while
22.}

---

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.

![Diagram showing problematic vertices and backtracking root cause detection on PPG](image)

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.

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

![GUI of SCALANA](image)
keeps the same sampling frequency (200Hz) as HPCToolkit in all experiments. For all experiments, AbnormThd is set to 10 and MaxLoopDepth 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.

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

### 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

<table>
<thead>
<tr>
<th>Programs</th>
<th>BT</th>
<th>CG</th>
<th>EP</th>
<th>FT</th>
<th>MG</th>
<th>SP</th>
<th>LU</th>
<th>SST</th>
<th>NEK</th>
<th>ZMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ovd(%)</td>
<td>0.32</td>
<td>0.77</td>
<td>0.38</td>
<td>0.35</td>
<td>0.29</td>
<td>0.31</td>
<td>0.28</td>
<td>0.68</td>
<td>3.01</td>
<td>0.43</td>
</tr>
</tbody>
</table>

**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 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%.

**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.

**Post-mortem Detection Cost** We evaluate the cost of back-tracking 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).
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 $64 \times 64 \times 64$ 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 $55.53 \times$ on 128 processes while $35.40 \times$ 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 LOOPs 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 $55.53 \times$ to $61.39 \times$ (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 $68.41 \times$ to $76.15 \times$ (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...
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 $1.20 \times$ on 32 processes while $1.28 \times$ 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 `LOOOP` 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 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 $1.20 \times$ to $1.56 \times$ (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 $31.95 \times$ for 64 processes while the speedup of 32 processes is $20.61 \times$ (1 process as baseline).
Scaling Loss Detection We use SCALANA to analyze the root cause of the scalability problem. MPI_Waitall in the function \texttt{comm\_wait} at \texttt{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 \texttt{loop} in the function \texttt{dgemm} at \texttt{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](image)

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 \texttt{comm.h:243}, the \texttt{loop} at \texttt{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 unsuitable for production environments.

Profiling Profiling can extract the program’s statistical information with very low overhead. mpiP [9] is a lightweight 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 hpc-traceviewer. Arm MAP [55] is a lightweight 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

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

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


