PerFlow: A Domain Specific Framework for Automatic Performance Analysis of Parallel Applications

Yuyang Jin
Tsinghua University
jyy17@mails.tsinghua.edu.cn

Haojie Wang
Tsinghua University
wanghaojie@tsinghua.edu.cn

Runxin Zhong
Tsinghua University
zrx21@mails.tsinghua.edu.cn

Chen Zhang
Tsinghua University
zhang-c21@mails.tsinghua.edu.cn

Jidong Zhai
Tsinghua University
zhaijidong@tsinghua.edu.cn

Abstract

Performance analysis is widely used to identify performance issues of parallel applications. However, complex communications and data dependence, as well as the interactions between different kinds of performance issues make high-efficiency performance analysis even harder. Although a large number of performance tools have been designed, accurately pinpointing root causes for such complex performance issues still needs specific in-depth analysis. To implement each such analysis, significant human efforts and domain knowledge are normally required.

To reduce the burden of implementing accurate performance analysis, we propose a domain specific programming framework, named PerFlow. PerFlow abstracts the step-by-step process of performance analysis as a dataflow graph. This dataflow graph consists of main performance analysis sub-tasks, called passes, which can either be provided by PerFlow's built-in analysis library, or be implemented by developers to meet their requirements. Moreover, to achieve effective analysis, we propose a Program Abstraction Graph to represent the performance of a program execution and then leverage various graph algorithms to automate the analysis. We demonstrate the efficacy of PerFlow by three case studies of real-world applications with up to 700K lines of code. Results show that PerFlow significantly eases the implementation of customized analysis tasks. In addition, PerFlow is able to perform analysis and locate performance bugs automatically and effectively.

CCS Concepts: • **Software and its engineering → Domain specific languages**; **Software performance**; • **Theory of computation → Program analysis**.

This work is licensed under a Creative Commons Attribution-NonCommercial International 4.0 License.
PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9204-4/22/04.
https://doi.org/10.1145/3503221.3508405

Keywords: Performance Analysis, Domain Specific Framework, Dataflow Graph

1 Introduction

Performance analysis is indispensable for understanding and optimizing applications and is widely used in different fields including scientific computing [10, 47, 51, 67], machine learning [37, 46, 70], and data processing [11, 28]. Due to the complexity of load imbalance, communication dependence, resource contention, etc. [18, 36, 50], significant human efforts and knowledge need to be involved in effective analysis currently. It is challenging to understand the performance behavior of parallel applications with ease.

A large number of performance tools have been proposed to facilitate performance analysis based on either profiling or tracing. *Profiling-based tools* [8, 59, 62] record program snapshots at regular intervals, indicating the overall statistical performance data of programs, with very low overhead. *Tracing-based tools* [4, 31, 44, 48, 56] record all event traces during program execution, which contain plentiful information, including computation, memory access, and communication characteristics. These tools provide various performance data, which are the basis of performance analysis. However, to locate the underlying performance bugs hidden by complex performance data and communication dependence, in-depth analysis is further required.

Researchers have proposed many in-depth performance analysis approaches to locate different kinds of performance bugs in different scenarios, such as critical path analysis [19, 20, 54], root cause analysis [18, 41], etc. Existing approaches only focus on a specific aspect of the performance issue of parallel programs. However, a performance issue for a complex parallel program may involve multiple factors interleaved in a complex way. (1) Complex communications, locks, and data dependence unpredictably hide performance bugs. (2) Different performance bugs interact with each other, which means that the detected performance bugs may come from several kinds of performance issues, including load imbalance, resource contention, etc. Identifying root causes in a new scenario requires specific in-depth analysis approaches, and implementing specific approaches normally

requires significant human efforts and domain knowledge. Therefore, we conclude that an easy-to-use framework for easing the implementation of in-depth performance analysis is necessary.

Designing a general framework for effective performance analysis has two key challenges. (1) Providing a unified form to express different performance analysis tasks is difficult. To meet the needs of various scenarios, the algorithms of constraint-solving-based analysis approaches are designed specifically and differ greatly. We observe that a typical performance analysis approach is a step-by-step process, meaning that each step only performs a basic analysis, and the results from one step are further processed by the next step. Inspired by this observation, we come up with the idea of abstracting performance analysis tasks as a general dataflow graph. The vertex of the dataflow graph corresponds to a step, while the data on the edge of the dataflow graph record intermediate results between steps. (2) Providing a unified form to represent the performance of a program is difficult, since analysis approaches rely on significantly different programs and performance data, including performance monitor unit data, program structure, communication patterns, data dependence, and many more. Many existing works utilize graphs to represent program behavior and design task-driven methods to solve their problems including program debugging [12, 22], performance modeling [17], and communication trace compression [69], etc. [41, 64, 73]. Inspired by these works, we represent the performance of a program as a graph structure.

In this work, we focus on the domain of performance analysis, and propose PerFlow, a domain specific framework to ease the implementation of in-depth performance analysis tasks. *In PerFlow, we abstract the step-by-step process of performance analysis as a dataflow graph* [25], called PerFlowGraph, where the analysis steps, called **passes**, correspond to vertices, and the intermediate results of each analysis step correspond to the data on edges. We leverage hybrid static-dynamic analysis to generate a Program Abstraction Graph (PAG) as a unified form to represent the performance of a parallel program, and then implement tasks of analysis steps with graph operations and algorithms on the generated PAG. We provide a built-in analysis pass library containing several basic performance analysis sub-tasks, and low-level APIs to build user-defined passes. With PerFlow, developers only need to describe their specific performance analysis tasks as PerFlowGraphs. PerFlow is able to automatically perform specific in-depth analysis tasks and report results specified by developers. In summary, there are four main contributions in our work.

- We propose PerFlow[1], a domain specific framework for performance analysis. PerFlow provides a dataflow-based programming interface for developers to customize specific performance analysis tasks with ease.
- We present a Program Abstraction Graph, which is a unified performance representation of parallel programs.
- We provide a performance analysis pass library and some built-in performance analysis paradigms. Developers can directly use passes and paradigms to perform analysis.
- We demonstrate the efficacy and efficiency of PerFlow by three case studies on real-world applications with up to 700K lines of code, leveraging different PerFlow dataflow graphs to detect performance bugs in different scenarios.

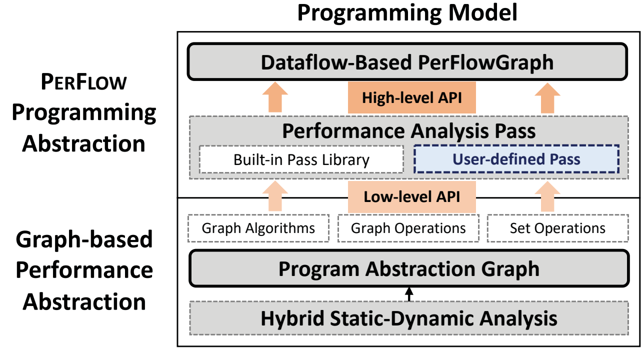


Figure 1. The framework of PerFlow

We evaluate PerFlow with both benchmarks and real-world applications. Experimental results show that PerFlow can detect scalability bugs, load imbalance, and resource contention with different PerFlow dataflow graphs more effectively and efficiently compared with mpiP [62], HPCToolkit [8], and Scalasca [31]. Besides, PerFlow significantly eases the implementation of the scalability analysis task in ScalAna [41]. Applications can achieve up to 25.29× performance improvements by fixing detected performance bugs.

2 Overview

To help developers deal with the complexities in implementing specific performance analysis tasks, we develop PerFlow, a domain specific programming framework that screens developers from all complexities and automatically performs the process of specific performance analysis. In PerFlow, the step-by-step process of performance analysis is abstracted as a dataflow graph, namely PerFlowGraph. Using PerFlow, developers only need to describe performance analysis tasks as PerFlowGraphs, and PerFlow will run the program and perform performance analysis automatically. In this section, we introduce the PerFlow framework, and give an example to illustrate how to program with PerFlow.

2.1 PerFlow Framework

The overview of PerFlow is shown in Figure 1, which consists of two components: a graph-based performance abstraction and a PerFlow programming abstraction.

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

Graph-based performance abstraction. In this component, the performance of a program execution is represented as a Program Abstraction Graph, whose vertices represent code snippets and edges represent control flow, data movement, and dependence (Section 3). Taking an executable binary as input, PerFlow first leverages hybrid static-dynamic analysis (Section 3.2) to extract program structures and collect performance data. Then performance data are embedded into the program structure to build a PAG, describing the performance of a program run (Section 3.3).

PerFlow programming abstraction. This component abstracts the process of performance analysis as a data-flow graph. PerFlow programming abstraction consists of two concepts, performance analysis passes, and dataflow-based programming model. As the core of programming abstraction, the performance analysis pass library provides various built-in passes (Section 4.3.2), which are built with low-level APIs based on the performance abstraction. The passes perform graph algorithms, such as breadth-first search, subgraph matching, etc., on the PAGs and complete basic analysis sub-tasks. Intermediate results, which are the inputs/outputs of passes, are organized as sets. The elements of a set are PAG vertices and edges. A pass takes sets as input, updates the sets, and outputs them. (Note that a PAG is an environment of all passes in a PerFlowGraph, and a set is a subset of PAG vertices flowing along the edges of a PerFlowGraph.)

As a programming framework, PerFlow also provides a dataflow-based API (high-level API), allowing users to analyze the performance of parallel programs in different scenarios with ease and high efficiency. Developers only need to combine passes into PerFlowGraph according to the demand of their analysis tasks. Then PerFlow will automatically run the programs and perform the specific performance analysis. PerFlow currently supports MPI, OpenMP, and Pthreads programs in C, C++, and Fortran. The hybrid static-dynamic module is easy to extend to other programming models, such as CUDA, and other architectures, such as ARM. In our design, the PerFlow is a cross-platform framework.

2.2 Example: A Communication Analysis Task

We take a communication analysis task, as an example to illustrate how to program with PerFlow. When analyzing the communication performance of a program execution, the balance of communications is one of the key points. If communications are detected with imbalanced behavior, developers need to break them down to determine whether the cause of imbalance is different message sizes, the load imbalance before the communications, or others. We conclude the step-by-step process of this communication analysis task as a PerFlowGraph in Figure 2. It reports key attributes (including function name, communication patterns, debug info, and execution time) of detected communication calls with performance bugs. The report module provides both human-readable texts and visualized graphs. Listing 1 shows the implementation of the PerFlowGraph with PerFlow's high-level Python APIs.

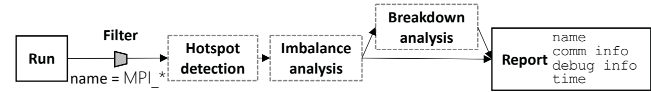


Figure 2. A communication analysis task represented as a dataflow graph (PerFlowGraph)

```
pflow = PerFlow()
# Run the binary and return a program abstraction graph
pag = pflow.run(bin = "./a.out",
                   cmd = "mpirun -np 4 ./a.out")

# Build a PerFlowGraph
V_comm = pflow.filter(pag.V, name = "MPI_*")
V_hot = pflow.hotspot_detection(V_comm)
V_imb = pflow.imbalance_analysis(V_hot)
V_bd = pflow.breakdown_analysis(V_imb)
attrs = ["name", "comm-info", "debug-info", "time"]
pflow.report(V_imb, V_bd, attrs)
```

Listing 1. A communication analysis task written using PerFlow's Python API

3 Graph-Based Performance Abstraction

A program can be naturally represented as a graph. The code snippets of programs correspond to the vertices in the graph, while the relationships between these code snippets, such as control/data flow and dependence across threads/processes, correspond to the edges in the graph. The performance data can be stored as attributes of vertices and edges. In PerFlow, we use a Program Abstraction Graph to represent the performance of a program run. In this section, we first introduce the definition of PAG, and then describe how to leverage hybrid static-dynamic analysis to extract PAG structure and how to embed performance data on a PAG.

3.1 Definition of PAG

A PAG is a (weighted) directed graph $G = (V, E)$.

Vertex(V). Each vertex $v \in V$ represents a code snippet or a control structure of a program, whose labels and properties indicate the types of this vertex and the recorded data on it.

1) The *labels* of a vertex include *function*, *call*, *loop*, and *instruction*. *Call* vertices are divided into user-defined function calls, communication function calls, external function calls, recursive calls, and indirect calls, etc.

2) The *properties* of a vertex are various performance data, including execution time, performance monitor unit (PMU) data, communication data, the number of function calls, and iteration count, etc., depending on the specific requirement of analysis tasks and the view of the PAG.

Edge (E). Each edge $e = (v_{src}, v_{dest}) \in E$ connects two vertices v_{src} and v_{dest}, whose labels and properties indicate the types of this edge and the recorded data on this edge.

1) The *labels* of an edge include *intra-procedural*, *inter-procedural*, *inter-thread*, and *inter-process*. The *intra-procedural edge* represents the control flow of functions. The *inter-procedural edge* represents function call relationships. The *inter-thread edge* represents data dependence across different threads, such as waiting events caused by locks. The *inter-process edge* represents communications between different processes, including synchronous point-to-point (P2P) communications, asynchronous P2P communications, and collective communications.

2) The *properties* of an edge can be the performance data, the execution time of communications, the amount of communication data, as well as the time of waiting events, etc., depending on the types of edges and runtime data.

3.2 Hybrid Static-Dynamic Analysis

Hybrid static-dynamic analysis is leveraged to collect data for PAG generation. Static analysis extracts the main structure of PAG, while the dynamic analysis collects performance data and the required structure that cannot be obtained statically, such as indirect calls, locks, and communications, etc., by monitoring the program at runtime. Static analysis can significantly reduce the runtime overhead of pure dynamic analysis.

Static analysis. PerFlow statically analyzes the binary using Dyninst [66] to extract the static information, including the control flow, static call relationship, and debug information. The static analysis also marks the function calls whose information cannot be obtained at the static phase so that they can be filled in at runtime.

Dynamic analysis. PerFlow provides a built-in runtime data collection module using sampling-based approaches. The collection module collects runtime data that cannot be obtained statically, including the performance monitor unit (PMU) data, communication data, lock information, indirect call relationships, etc.

3.3 Performance Data Embedding

Performance data embedding associates performance data with attributes of the corresponding vertices. We first identify the corresponding vertex through the calling context of each piece of data, and then associate the performance data with these vertices.

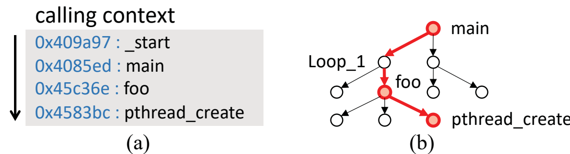


Figure 3. Illustration of performance data embedding

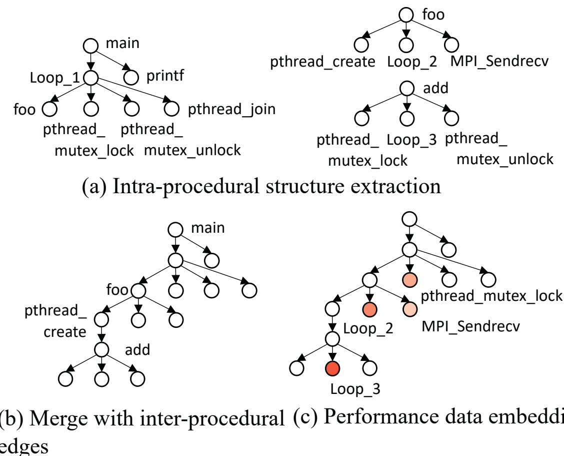
In Figure 3, we give an example to illustrate the process of performance data embedding. Figure 3(a) is a calling context, and Figure 3(b) shows a PAG. Starting from the main vertex, Loop_1, foo, and pthread_create vertices are detected with the calling context during the searching process. Finally, this piece of data is embedded into the pthread_create vertex.

```
1  int main() {
2      while (iter--) {                // Loop_1
3          foo(...);
4          pthread_mutex_lock(...);
5          sum += local_sum;
6          pthread_mutex_unlock(...);
7          pthread_join(...);
8      }
9      printf(...); }
10 void foo(...){
11     pthread_create(..., add, ...);
12     for (...)  sum += B[i];         // Loop_2
13     MPI_Sendrecv(sum, ...); }
14 void* add(...) {
15     pthread_mutex_lock(...);
16     for (...)  sum += A[i];         // Loop_3
17     pthread_mutex_unlock(...); }
```

Listing 2. An MPI+Pthreads program example

3.4 Views of PAG

PerFlow provides two views of PAG: a top-down view and a parallel view. We take an example for explanation. Listing 2 shows an MPI + Pthreads program example with three functions (main, foo, and add) (Static analysis is performed on executable binaries, and the example code is only for ease of understanding.).



(a) Intra-procedural structure extraction

(b) Merge with inter-procedural edges (c) Performance data embedding

Figure 4. Generating the top-down view of PAG. The color saturation of vertices represents the severity of hotspots. Only relevant vertices are marked.

Top-down view of PAG. The top-down view of PAG only contains *intra-procedural* and *inter-procedural* edges. Figure 4(a) shows three PAGs of main, foo, and add generated through intra-procedural structure extraction. Figure 4(b) shows a PAG that merges each function's PAG with *inter-procedural edges* (only the related vertex for merging marked). Figure 4(c) shows a top-down view of PAG with performance data in each vertex after *performance data embedding* (only the vertex with performance data marked). The color saturation of vertices represents the severity of hotspots.

Parallel view of PAG. The parallel view of PAG contains all types of edges including *intra-procedural*, *inter-procedural*, *inter-thread*, and *inter-process* edges. To build a parallel view of PAG, (1) we generate a flow for each process and thread. A flow is the vertex access sequence recorded by pre-order traversal through a specific part of the top-down view of PAG. Figure 5(a) shows the generated flows for all threads. (2) Then we add *inter-thread*, and *inter-process* edges, which represent locks, communications, etc., across flows of different processes and threads. (3) We further embed performance data into the PAG. Finally, a parallel view of PAG is formed. Figure 5(b) shows the generated parallel view of PAG.

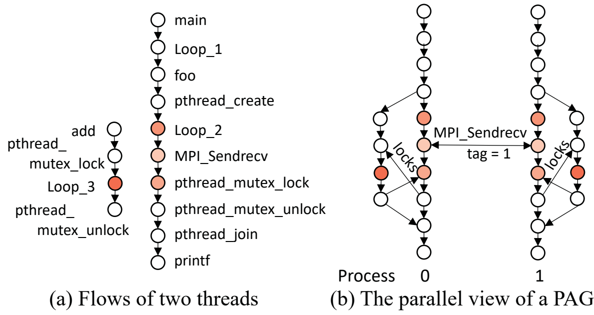


Figure 5. Parallel view of PAG. The color saturation of vertices represents the severity of hotspots. Only relevant vertices are marked.

4 PerFlow Programming Abstraction

4.1 PerFlowGraph

PerFlow uses a dataflow graph (PerFlowGraph) to represent all analysis steps and phases in a performance analysis task, including the running phase, the analysis sub-tasks, and the result reporting phase, etc. The key observation from existing performance analysis approaches and our experience is that the process of performance analysis is similar to a dataflow graph. Developers analyze profiles and traces step by step and finally identify performance bugs. Thus we design a dataflow-based programming abstraction to represent the process of performance analysis. In the rest of this section, we introduce the elements in a PerFlowGraph, as well as performance analysis passes and paradigms.

4.2 PerFlowGraph Element

In a PerFlowGraph, each vertex represents an analysis sub-task, and each edge represents the input to, or output from, a vertex. We use a performance analysis pass to complete a sub-task in a vertex, and use sets as the data flowing along edges. We introduce the elements of the PerFlowGraph below.

Set. The sets can be sets of PAG vertices V or sets of PAG edges E, or both (V, E). In PerFlow, we model all code snippets and program structures as PAG vertices, and all data/control dependence and data movements as PAG edges (details in Section 3.1). The contents of sets are updated as they flow through vertices of PerFlowGraphs.

Pass. A performance analysis pass takes sets as input. After performing its analysis sub-task, it also outputs sets as the input of the next pass. As shown in Figure 6, the input sets flow through a performance analysis pass, and then output sets are generated and continue flowing. The format of inputs and outputs is determined by the design of passes. Developers can flexibly use and combine passes to build the structure of the PerFlowGraph.

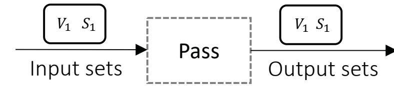


Figure 6. The relationship of sets and performance analysis passes

PerFlow provides high-level APIs and a built-in pass library for PerFlowGraph construction. The built-in pass library provides hotspot detection, differential analysis, critical path identification, imbalance analysis, and breakdown analysis, etc. Besides, PerFlow also provides low-level APIs, which allow developers to write user-defined passes to meet their requirements. We introduce several built-in passes and their implementations using low-level APIs in Section 4.3.2.

4.3 Building Performance Analysis Pass

We introduce the design of low-level API and how to build performance analysis passes with the API below.

4.3.1 Low-level API design. We design three types of APIs: graph operation APIs, graph algorithm APIs, and set operation APIs.

Graph operation APIs provide interfaces for developers to access the attributes of PAG vertex and edge, including name, type, performance data, and debug information, etc., or even to transform the PAG. Here we define the inputs and outputs of a pass, which uses graph operation API, as I and O. It may happen that $O \not\subseteq I$ ($\exists\, e \in O$, but $e \notin I$), which means graph operations can add new elements to the output.

Graph algorithm APIs provide many graph algorithms, such as breadth-first search, subgraph matching, and community detection, etc. Developers can use these algorithms and combine constraints to achieve specific analysis tasks.

Set operation APIs include element sorting, filtering, classification, as well as computing intersection, union, complement, and difference of sets. Different from graph operations, for a pass that only uses set operations, the outputs must be a subset of the inputs ($O \subseteq I$). We take the operation filter as an example. It is designed to deliver specific PAG vertices and edges to specific passes. The metric of a filter can be the type, name, and other attributes of vertices and edges. A filter can distinguish communication vertices by matching the name attribute with the string `MPI_*`, and IO vertices by matching the name with the strings `istream::read` or types of vertices.

4.3.2 Example cases.

We further introduce four built-in performance analysis passes and illustrate how to use PerFlow's low-level API to develop passes with graph algorithms on PAG and set operations.

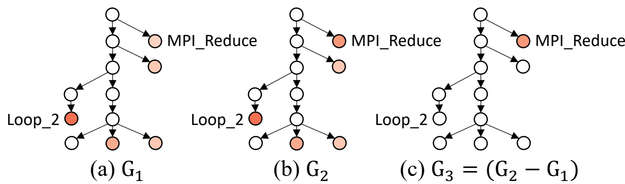
A: Hotspot detection. Hotspot detection refers to identifying the code snippets with the highest value of specific metrics, such as total execution cycles, cache misses, and instruction count, etc. The most common hotspot detection is to identify the most time-consuming code snippets, whose specific metric is total execution cycles or execution time. As shown in Listing 3, a hotspot detection pass is built.

```
# Define an "hotspot detection" pass
# Input:  The vertex set of a PAG - V
#         Sorting metric - m
#         The number of returned vertices - n
# Output: Hotspot vertex set
def hotspot(V, m, n):
  return V.sort_by(m).top(n)
```

Listing 3. The implementation of hotspot detection pass

B: Performance differential analysis. Performance differential analysis refers to a comparison of program performance conducted under the independent variables of input data, parameters, or different executions. The comparison helps analysts understand the trend of performance as the input changes. The performance difference can be intuitively represented on a top-down view of PAG, and we leverage the *graph difference* to perform differential analysis.

The graph difference algorithm is performed on the top-down view of PAG. As shown in Figure 7, G_1 and G_2 are two PAGs with different inputs, and G_3 is the *graph difference* between G_1 and G_2. The color saturation of vertices represents the severity of hotspots. We find that the color saturation of MPI_Reduce in G_1 and G_2 is not the highest, but it is the highest in G_3, which means the performance of non-hotspot vertex MPI_Reduce varies significantly with different inputs. Vertices that behave like the MPI_Reduce are identified with performance issues through performance differential analysis. *Graph difference* intuitively shows the changes in performance between program runs with different inputs. We implement this pass in Listing 4.



(a) G_1 (b) G_2 (c) $G_3 = (G_2 - G_1)$

Figure 7. Graph difference on the top-down view of PAGs. The color saturation of vertices represents the severity of hotspots.

C: Causal analysis. Performance bugs can propagate through complex inter-process communications as well as inter-thread locks, and lead to many secondary performance bugs, which makes root cause detection even harder. Paths that consist of a parallel view of PAG's edges can well represent correlations

```
# Define a "differential analysis" pass
# Input:  Vertex sets of two PAGs - V1, V2
# Output: A set of difference vertices
def differential_analysis(V1, V2):
  V_res = []
  for (v1, v2) in (V1, V2):
    v = pflow.vertex()
    for metric in v1.metrics:
      v[metric] = v1[metric] - v2[metric]
    V_res.append(v)
  return V_res
```

Listing 4. The implementation of performance differential analysis pass

across these performance bugs in different processes and threads. We leverage a graph algorithm, *lowest common ancestor* [53] (LCA), and specific restrictions to detect the correlations and thus achieve the purpose of causal analysis. The goal of the LCA algorithm is to search the deepest vertex that has both v and w as descendants in a tree or directed acyclic graph.

The causal analysis pass is designed based on the LCA algorithm. Listing 5 shows the implementation of the causal analysis pass. This pass takes vertices with performance bugs as inputs, and regards them as descendants. After performing the LCA algorithm, the detected common ancestors of descendants are recorded and output as the vertices that cause performance bugs.

```
# Define a "causal analysis" pass
# Input:   A set of vertices with performance bugs - V
# Output:  A set of vertices that cause the bugs
def casual_analysis(V)
  V_res, S = [], [] # S for scanned vertices
  for (v1, v2) in (V, V):
    if v1!=v2 and v1 not in S and v2 not in S:
      # v1 and v2 are regarded as descendants
      v, path = pflow.lowest_common_ancestor(v1, v2)
      # v is the detected lowest common ancestor
      # path is an edge set
      if v in V:
        V_res.append(v)
  return V_res
```

Listing 5. The implementation of the causal analysis pass

D: Contention detection. Contention refers to a conflict over a shared resource across processes or threads, which leads to a negative impact on the performance of processes or threads competing for the resource. It can cause several kinds of misbehavior, such as unwanted synchronization or periodicity, deadlock, livelock, and many more, which need expensive human efforts to be detected. We observe that misbehaviors have specific patterns on the parallel view of PAGs. *Subgraph matching* [57], which searches all embeddings of a subgraph query in a large graph, is leveraged to search these specific patterns on the PAGs and detect resource contention.

The contention detection pass determines whether resource contention exists in the vertices of input sets. The input of a contention detection pass is a set of vertices detected by the previous pass, while the outputs are the detected subgraph embeddings. We define a set of candidate

subgraphs to represent resource contention patterns. Then we identify resource contentions by searching the embeddings of candidate subgraphs around the vertices of the input set. Listing 6 shows the implementation of the contention detection pass.

```
# Define a "contention detection" pass
# Input:  Vertex set - V
# Output: Subgraph embeddings
def contention_detection(V):
  V_res = []
  # Build a candidate subgraph with contention pattern
  sub_pag = pflow.graph()
  sub_pag.add_vertices([(1,"A"), (2,"B"), (3,"C"),
                                   (4,"D"), (5,"E")])
  sub_pag.add_edges([(1,3), (2,3), (3,4), (3,5)])
  # Execute subgraph matching algorithm
  V_ebd, E_ebd = pflow.subgraph_matching(V.pag, sub_pag)
  return V_ebd, E_ebd
```

Listing 6. The implementation of the contention detection pass

4.4 Performance Analysis Paradigm

A performance analysis paradigm is a specific PerFlowGraph for an analysis task. We summarize some typical performance analysis approaches of existing tools [8, 31, 48, 56, 62] as built-in analysis paradigms, such as an *MPI profiler* paradigm (inspired by mpiP [62]), a *critical path* paradigm (inspired by the work of Böhme et al. [19] and Schmitt et al. [54]), and a *scalability analysis* paradigm (inspired by the work of Böhme et al. [18] and ScalAna [41]), etc.

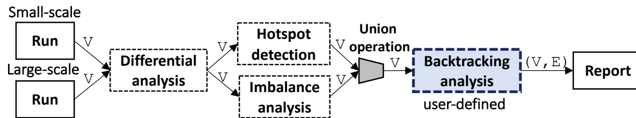


Figure 8. The PerFlowGraph of the scalability analysis paradigm

We take the *scalability analysis* paradigm as an example to show how to implement a performance analysis paradigm. The scalability analysis task in [41] first detects code snippets with scaling loss and imbalance, then finds the complex dependence between the detected code snippets by a backtracking algorithm, and finally identifies the root causes of scaling loss. We decompose the scalability analysis task into multiple steps. Most of the steps can be completed with PerFlow's built-in passes, and we only need to implement the backtracking step as a user-defined pass.

As shown in Figure 8, we build the PerFlowGraph of the scalability analysis paradigm, containing three built-in passes (differential analysis pass, hotspot detection pass, and imbalance analysis pass), a user-defined pass (**backtracking analysis pass**), a union operation, and a report module. Listing 7 shows the implementation of the scalability analysis paradigm, which consists of two parts: **(1) Writing a backtracking analysis pass.** We first write a backtracking analysis pass, which is not provided by our built-in pass

```
# Define a "scalability analysis" paradigm
# Input:  PAGs of two program runs - PAG1, PAG2
def scalability_analysis_paradigm(PAG1, PAG2):

  # Part 1: Define a "backtracking analysis" pass
  # Input:  A set of vertices with performance bugs - V
  # Output: Vertices and edges on backtracking paths
  def backtracking_analysis(V):
    V_bt, E_bt, S = [], [], [] # S for scanned vertices
    for v in V:
      if v not in S:
        S.append(v)
        in_es = v.es.select(IN_EDGE)
        while len(in_es) != 0
                and v[name] not in pflow.COLL_COMM:
          if v[type] == pflow.MPI:
            e = in_es.select(type = pflow.COMM)
          elif v[type] == pflow.LOOP or
                          v[type] == pflow.BRANCH:
            e = in_es.select(type = pflow.CTRL_FLOW)
          else
            e = in_es.select(type = pflow.DATA_FLOW)
          V_bt.append(v)
          E_bt.append(e)
          v = e.src
    return V_bt, E_bt

  # Part 2: Build the PerFlowGraph of scalability
     analysis paradigm
  V1, V2 = PAG1.vs, PAG2.vs
  V_diff = pflow.differential_analysis(V1, V2)
  V_hot = pflow.hotspot_detection(V_diff)
  V_imb = pflow.imbalance_analysis(V_diff)
  V_union = pflow.union(V_hot, V_imb)
  V_bt, E_bt = backtracking_analysis(V_union)
  attrs = ["name", "time", "dbg-info", "pmu"]
  pflow.report([V_bt, E_bt], attrs)

# Use the scalability analysis paradigm
pag_p4 = pflow.run(bin = "./a.out",
                cmd = "mpirun -np 4 ./a.out")
pag_p64 = pflow.run(bin = "./a.out",
                cmd = "mpirun -np 64 ./a.out")
scalability_analysis_paradigm(pag_p4, pag_p64)
```

Listing 7. The implementation of the scalability analysis paradigm

library. As shown in Listing 7, this pass implements a backward traversal through communications, and control/data flow with several graph operation APIs, including neighbor acquisition (`v.es` at Line 13), edge filter (`select` at Line 13, 17, 20, and 22), attribute access (`v[...]` at Line 15-16, 18-19), and source vertex acquisition (`e.src` at Line 25). **(2) Building the PerFlowGraph of the scalability analysis paradigm.** Then, we build a PerFlowGraph with built-in and user-defined passes. The differential analysis pass (Line 30) takes two executions (i.e., a small-scale run and a large-scale run) as input, and outputs all vertices with their scaling loss. Then the hotspot analysis pass (Line 31) outputs vertices with the poorest scalability, while the imbalance analysis pass (Line 32) outputs imbalanced vertices between different processes. The union operation (Line 33) merges two sets (outputs of the hotspot analysis pass and the imbalance analysis pass) as the input of the backtracking analysis pass

(Line 34). Finally, the backtracking paths and the root causes of scalability are stored in (`V_bt`, `E_bt`) and reported (Line 36).

4.5 Usage of PerFlow

In summary, there are two main ways for developers to implement specific analysis tasks with PerFlow's APIs: using paradigms and building PerFlowGraphs.

Using Paradigms. Developers can directly use built-in paradigms to obtain related performance analysis reports. An example that shows how to use a paradigm is given at Line 38-43 in Listing 7. We run a program with two process scales of 4 and 64, and directly input them into the scalability analysis paradigm.

Building PerFlowGraphs. PerFlow provides a built-in performance analysis pass library for building PerFlowGraphs. *For scenarios where analysis tasks have already been designed*, the example in Listing 7 has shown a complete process of implementation. *For scenarios in which developers do not know what analysis to apply*, PerFlow supports an interactive mode. It is advisable to first use a general built-in analysis pass, such as hotspot detection. The output of the previous pass will provide some insights to help determine or design the next passes. Then analysts can add other analysis passes into PerFlowGraph step by step. Finally, PerFlowGraphs are generated according to detailed analysis.

If built-in passes cannot satisfy the demands, developers need to write their own passes and combine these user-defined passes with other built-in passes to build PerFlowGraphs. Developers require some basic knowledge to write user-defined passes. The implementations of several passes have been introduced (four built-in passes in Section 4.3.2 and the backtracking analysis pass at Line 5-26 in Listing 7).

5 Evaluation

5.1 Experimental Setup

Experimental platforms. We perform the experiments on two clusters: (1) Gorgon, a cluster with dual Intel Xeon E5-2670 (v3) and 100 Gbps 4xEDR Infiniband. (2) A national supercomputer *Tianhe-2A*. Each node of *Tianhe-2A* has two Intel Xeon E5-2692 (v2) processors (24 cores in total) and 64 GB memory. The *Tianhe-2A* supercomputer uses a customized high-speed interconnection network. PerFlow uses Dyninst (v10.1.0) [66] for static binary analysis, as well as PMPI wrapper, PAPI library (v5.4.3) [1], and libunwind library (v1.3.1) for dynamic data collection. The PAG is stored in a graph processing system igraph [24].

Evaluated programs. We use a variety of parallel programs to evaluate the efficiency and efficacy of PerFlow, including BT, CG, EP, FT, IS, LU, MG, and SP, from the widely used NPB benchmark suite (v3.3) [14], plus several real-world applications, ZeusMP [35], LAMMPS [13], and Vite [32]. For NPB programs, problem size CLASS C is used.

Methodology. In our evaluation, we first present both the static and runtime overhead, as well as the space cost of the hybrid static-dynamic analysis module (all evaluated programs run with 128 processes on gorgon.). Then we show basic features of the top-down view and the parallel view of PAGs for all evaluated programs (128 processes for the parallel view). Finally, we use three real-world applications to demonstrate the process of performing customized performance analysis with PerFlow. In addition, we compare the results of PerFlow with four state-of-the-art tools, mpiP (v3.5) [62], HPCToolkit (v2020.12) [8], Scalasca (v2.5) [31], and ScalAna [41], by studying the performance of ZeusMP. For HPCToolkit, we set the sampling frequency to 200 Hz, which is the same as PerFlow. For Scalasca, we first profile the program and determine where tracing is needed, which significantly reduces instrumentation overhead.

5.2 Overhead and PAG

Table 1. The overhead of PerFlow

Program	BT	CG	EP	FT	MG	SP	LU	IS	ZMP	LMP	Vite
Static(Sec.)	0.20	0.06	0.03	0.09	0.12	0.19	0.23	0.04	1.50	5.34	0.73
Dynamic(%)	0.44	3.73	0.13	1.83	0.92	1.08	1.42	0.03	1.56	0.71	0.03
Space(B)	346K	57K	35K	215K	464K	449K	184K	28K	2.4M	22M	1.6M

Static analysis. We first evaluate the cost of static analysis on the executable binaries. As shown in Table 1, the static analysis only incurs very low overhead (0.03 to 5.34 seconds, 0.77 seconds on average). For a software package with over 700K lines of code, LAMMPS, the static analysis costs only 5.34 seconds.

Dynamic analysis. For all programs, both PMU data and communication data are collected during dynamic analysis. The runtime performance overhead of dynamic analysis is 1.11% on average (0.03% to 3.73%), as shown in Table 1. The variance in dynamic overhead is caused by the different complexities of communication patterns. CG implements collective communications with three point-to-point communications, which makes its communication pattern more complicated. Thus, the runtime overhead of CG is much higher than that of other programs (3.73%).

Space cost. The space cost of PerFlow is the storage size of PAGs. Table 1 shows that the space costs for evaluated programs range from 28 Kilobytes to 22 Megabytes, and 2.5 Megabytes on average. The storage cost for the LAMMPS package is only 22 Megabytes.

Basic features of PAG. Table 2 shows the code size, the binary size, as well as the vertex and edge counts of both the top-down view and the parallel view of generated PAGs for all evaluated programs. The PAG of the program whose binary size is larger tends to have more vertices and edges.

5.3 Case Study A: ZeusMP

We use PerFlow and four state-of-the-art tools, mpiP [62], HPCToolkit [8], Scalasca [31], and ScalAna [41] to study

Table 2. Code size, binary size, and basic features of top-down view and parallel view of PAG for evaluated programs. |V| and |E| are the number of vertices and edges respectively.

Program	Code (KLoc)	Binary (Bytes)	Top-down view		Parallel view	
			|V|	|E|	|V|	|E|
BT	11.3	490K	3,283	3,282	420,224	462,404
CG	2.0	97K	321	320	41,088	55,176
EP	0.6	60K	111	110	14,208	34,360
FT	2.5	222K	2,904	2,903	371,712	409,128
MG	2.8	270K	4,701	4,700	601,728	712,432
SP	6.3	357K	2,252	2,251	288,256	322,364
LU	7.7	325K	1,566	1,565	200.448	284,780
IS	1.3	37K	325	324	41,600	69,816
ZeusMP	44.1	2.2M	11,981	11,980	1,533,568	2,805,760
LAMMPS	704.8	14.67M	85,230	85,229	10,909,440	16,423,808
Vite	15.9	2.8M	7,118	7,117	970,624	984,866

the performance analysis of ZeusMP. ZeusMP implements a three-dimensional astrophysical phenomena simulation with computational fluid dynamics using an MPI programming model.

We run ZeusMP with a problem size of 256×256×256 for different numbers of processes ranging from 16 to 2,048 on the *Tianhe-2A* supercomputer. Experimental results show that the speedup of ZeusMP does not scale well on 2,048 processes, which is only 72.57× (16 processes as baseline).

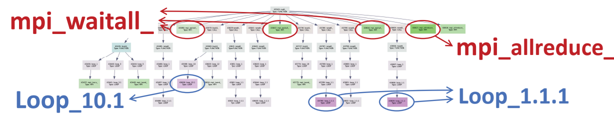


Figure 9. The output vertices of differential analysis pass on the top-down view of PAG

Performance analysis with PerFlow. We use the scalability analysis paradigm in Figure 8 to analyze the scalability problems. PerFlow first runs ZeusMP with 16 and 2,048 processes. Figure 9 shows the output of the differential analysis pass. Loop, mpi_waitall_, mpi_allreduce_ vertices are detected with scaling loss. The output of the imbalance analysis pass is the imbalanced vertices, which are marked with black boxes in Figure 10. Then the backtracking analysis pass builds paths between these imbalanced vertices, which represent how the performance bugs propagate (shown as red bold arrows). Figure 10 shows partial results due to space limitations. Finally, the imbalanced process vertices of loop_10.1 in *bvald_* and loop_1.1.1 in newdt_ are detected as the underlying reasons for ZeusMP's poor scalability.

As shown in Listing 8, the load imbalance of loop_10.1 at *bvald.F*: 358 causes that some processes of mpi_waitall_ at *nudt.F*: 227 wait for others. The delays in these processes cause the waiting events of some processes of mpi_waitall_ at *nudt.F*: 269 and then propagate to mpi_waitall_ at *nudt.F*: 328. Finally, the synchronization in mpi_allreduce_ at *nudt.F*: 361 becomes a scaling issue. In conclusion, the load imbalance propagates through three non-blocking point-to-point communications and causes the poor scalability of mpi_allreduce_ and ZeusMP.

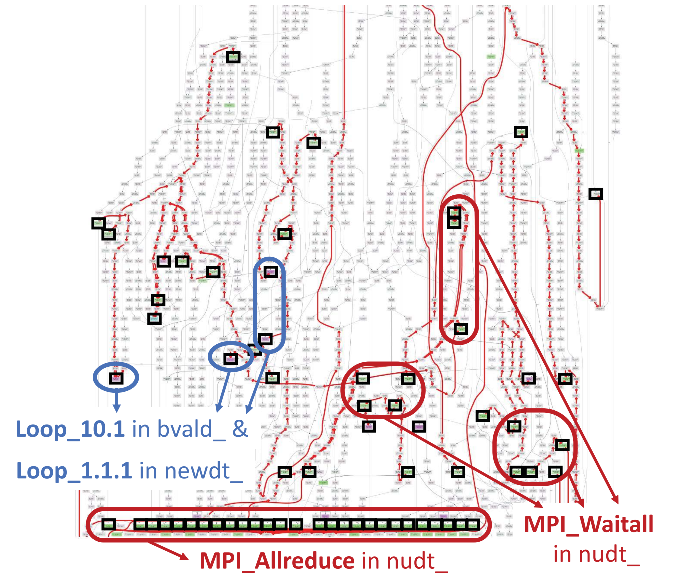


Figure 10. Partial results of the backtracking analysis pass on the parallel view of ZeusMP's PAG. The vertices with boxes are the output of the imbalance analysis pass, and red bold arrows represent the detected edges by the backtracking analysis pass.

```
    subroutine bvald (rl1, ru1, rl2, ru2, rl3, ru3, d)
357     do k=ks-1,ke+1                          ! Loop 10
358       do i=is-1,ie+1                        ! Loop 10.1
359         if (abs(nijb(i,k)) .eq. 1) then
360           d(i,js-1,k) = d(i,js  ,k)
361           d(i,js-2,k) = d(i,js+1,k)
391     call MPI_IRECV(d(1,je+j+uu,1), 1, j_slice, n2p ...
399     call MPI_ISEND(d(1,je+j-ll,1), 1, j_slice, n2p ...
    subroutine nudt
207     call bvald  (1,0,0,0,0,0,d) ...
227       call MPI_WAITALL (nreq, req, stat, ierr) ...
242     call bvald  (0,0,1,0,0,0,d) ...
269       call MPI_WAITALL (nreq, req, stat, ierr) ...
284     call bvald  (0,0,0,0,1,0,d) ...
328       call MPI_WAITALL (nreq, req, stat, ierr) ...
361     call MPI_ALLREDUCE(buf_in(1), buf_out(1), 1 ...
```

Listing 8. ZeusMP code with performance bugs

Comparison. We run ZeusMP with four state-of-the-art tools, mpiP, HPCToolkit, Scalasca, and ScalAna on both 16 and 2,048 processes. (1) **MpiP** generates statistical profiles, which present communication hotspots and other communication data, including message size, call count, and debug information, etc. In the report of mpiP, the mpi_allreduce_ in *nudt_* takes 0.06% and 7.93% of the total time on 16 and 2,048 processes, respectively. However, detecting the scaling loss of each communication call still needs significant human efforts. (2) **HPCToolkit** provides both fine-grained loop-level hotspots. In addition to hotspot analysis, HPC-Toolkit [65] can also detect multiple scalability issues in

`mpi_allreduce_` and `mpi_waitall_`. But the root cause of poor scalability and the underlying reasons cannot be easily obtained without performance analysis skills. (3) **Scalasca**, a tracing-based tool, can automatically detect root causes with event traces. The runtime overhead is 56.72% (not include I/O) and the storage cost is 57.64 Gigabytes on 128 processes for function-level event traces with human intervention, while PerFlow only incurs 1.56% runtime overhead and 2.4 Megabytes storage. (4) Besides, to implement the scalability analysis task with PerFlow, developers only need to write 27 lines of code with 7 high-level APIs and 5 low-level APIs (as shown in Listing 7). In contrast, the source code of **ScalAna** has thousands of lines.

Optimization. We fixed the root cause by changing ZeusMP into a hybrid MPI + OpenMP programming model. OpenMP `#pragma` on `loop_10.1` at *bvald_* allows idle processors to share the workload of busy processors, which mitigates the load imbalance between processes. We also perform this optimization on other detected code snippets with load imbalance. With these optimizations, the speedup of ZeusMP increases from 72.57× to 77.71× on 2,048 processes (16 processes as baseline). Meanwhile, the performance of ZeusMP is improved by 6.91% on 2,048 processes.

5.4 Case Study B: LAMMPS

LAMMPS is an open-source software package for large-scale molecular dynamics simulation. It is implemented with the hybrid MPI + OpenMP programming model. We run LAMMPS with 6,912,000 atoms and 2,048 processes (`in.clock.static` as an input) on the *Tianhe-2A* supercomputer. With simple profiling, we notice that the total communication time is up to 28.91%. In order to analyze the performance issue of LAMMPS, we design a PerFlowGraph in Figure 11. The PerFlowGraph detects imbalanced vertices and performs causal analysis repeatedly until the output set no longer changes, and we identify the outputs as the root causes.

Performance analysis with PerFlow. After running the program, a `PAG` is generated. Passing through the hotspot detection pass and the communication filter, `MPI_Send` and `MPI_Wait` are detected as communication hotspots with 7.70% and 7.42% of the total time. The imbalance analysis pass detects that some processes of `MPI_Send` and `MPI_Wait` are imbalanced vertices with longer execution time. As shown in Figure 12 (We only show a partial parallel view of the `PAG` due to space limitations), the top-down vertical axis represents the data flow, and the horizontal axis represents different parallel processes. The vertices with boxes are imbalanced `MPI_Send` and `MPI_Wait` calls. The output of the causal analysis pass indicates that the long execution time of `MPI_Send` and `MPI_Wait` in `CommBrick::reverse_comm` (*comm_brick.cpp*: 544, 547) is caused by `loop_1.1` in `PairLJCut::compute` (*pair_lj_cut.cpp*: 102-137). Figure 12 shows the result of causal analysis. The paths consisting of bold

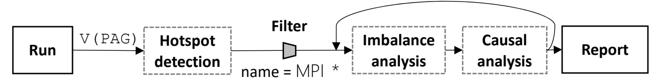


Figure 11. PerFlowGraph designed for performance analysis on LAMMPS

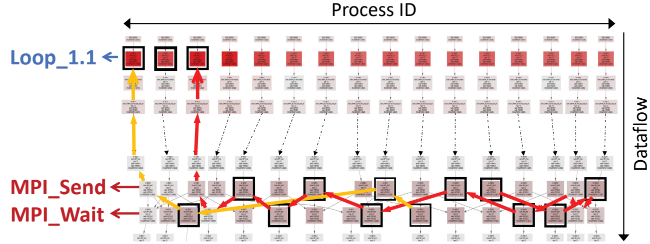


Figure 12. Illustration of the process of PerFlowGraph on the parallel view of LAMMPS's PAG

```
void PairLJCut::compute(){
  for (ii = 0; ii < inum; ii++) { // Loop_1
    for (jj = 0; jj < jnum; jj++) {...} // Loop_1.1
void CommBrick::reverse_comm(){
  for (int iswap = nswap-1; iswap >= 0; iswap--) {
    if (size_reverse_recv[iswap]) MPI_Irecv(...);
    if (size_reverse_send[iswap]) MPI_Send(...);
    if (size_reverse_recv[iswap]) MPI_Wait(...);
```

Listing 9. LAMMPS code with performance bugs

edges are causal relationships, which shows how performance bugs in `loop_1.1` propagate to `MPI_Send` and `MPI_Wait`. The cause is that process 0, 1, and 2 run with a longer time in `loop_1.1` than the others.

As shown in Listing 9, each process sends buffers to its neighbors, and it is implemented with blocking communications. The blocking communication propagates performance bugs in process 0, 1, and 2 (`loop_1.1`) to other processes (`MPI_Send` and `MPI_Wait`).

Optimization. The imbalance in `loop_1.1` is the root cause, and the performance bugs of `MPI_Send` and `MPI_Wait` are secondary bugs, which means that our optimization target is to make `loop_1.1` more balance. We add *balance* commands into the input file to adjust the size and shape of sub-domains of processes every 250 steps during simulation. With the optimization, the performance improves significantly from 118.89 timesteps/s to 134.54 timesteps/s (improved by 13.77%) on 2,048 processes.

5.5 Case Study C: Vite

Vite implements the distributed memory Louvain method for graph community detection using the MPI + OpenMP programming models. We evaluate its performance on a weighted graph with 600,000 vertices and 11,520,982 edges with 8 processes and different numbers of threads per process ranging from 2 to 8 (on gorgon). As shown in Figure 13, the red dotted line represents the execution time of the original version of Vite. We observe that Vite has extremely poor scalability as the number of threads grows. The execution

time on 8 threads is even longer than that on 2 threads. As shown in Figure 14, we design a PerFlowGraph, which sets up different branches for comprehensive diagnosis, to detect the performance issues of Vite.

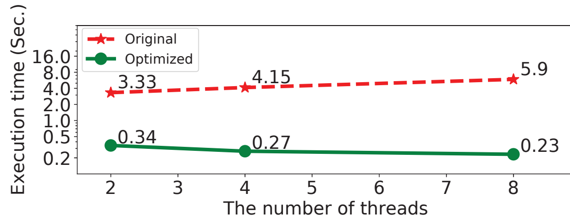


Figure 13. Scalability of Vite with 8 processes and different numbers of threads ranging from 2 to 8

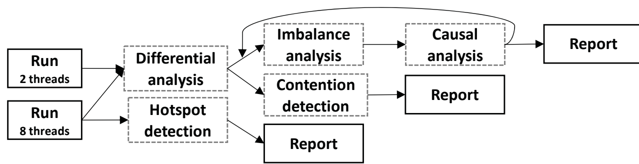
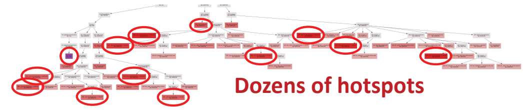


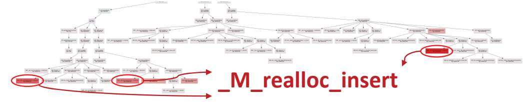
Figure 14. PerFlowGraph designed for performance analysis on Vite

Performance analysis with PerFlow. PerFlow first runs Vite with 2 and 8 threads on 8 processes, and two PAGs are generated. Figure 15(a) shows a partial output of the hotspot analysis pass. The darker the color of a vertex is, the longer the execution time of its corresponding code snippet. We notice that there exist dozens of hotspots, including several operations of _Hashtable. The output of the differential analysis pass is shown in Figure 15(b), from which we detect that _M_realloc_insert calls in *distExecuteLouvainIteration* have scalability issues. The report after the causal analysis presents that the _M_realloc_insert vertices themselves, and _M_emplace vertices are detected as the root causes. As shown in Figure 16, the contention detection pass searches for resource contention around the detected _M_realloc_insert vertices. The vertical direction from top to down represents the control/data flow, and the horizontal direction represents different parallel processes and threads. Each vertex stands for a code snippet in a thread or a process. We hide irrelevant inter-process and inter-thread edges to simplify the parallel view of PAG for better representation (The complete parallel view of PAG is much more complex). Subgraphs in red circles are detected embeddings of the resource contention pattern in different processes and code snippets.

In a zoomed-in subgraph, it can be seen that resource contention exists in allocate, reallocate, and deallocate (called by _M_realloc_insert, and _M_emplace). We find that the reason for resource contention is that memory allocation operations are thread-unsafe. When a thread allocates memory, an implicit lock is needed before the operation is performed. These locks lead to resource contention in memory allocation vertices, thus causing performance degradation and scalability issues as the number of threads grows.



(a) A partial output of the hotspot detection pass on the top-down view of PAG. Dozens of vertices are detected as hotspots.



(b) A partial output of the differential analysis pass on the top-down view of PAG. Only three _M_realloc_insert vertices are detected.

Figure 15. The output of different passes

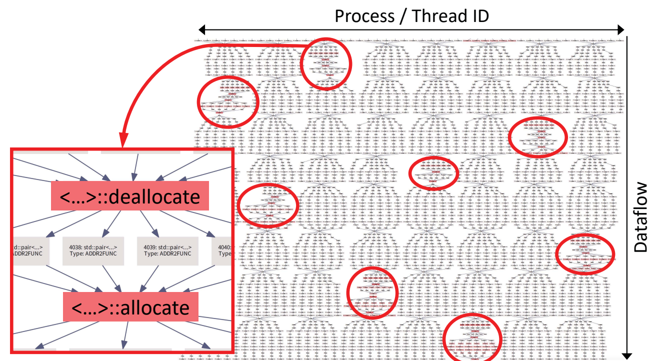


Figure 16. A partial output of the contention detection pass on the parallel view of Vite's PAG. We hide irrelevant inter-process and inter-thread edges for better representation.

Optimization. The results indicate that the key of optimization is to reduce the resource contention in allocate, reallocate, and deallocate. We apply two approaches to optimize it. (1) First, we use static thread-local variables to replace default stack variables so that they are initialized only once, which significantly reduces the number of allocate and deallocate calls. (2) We change the data structure from *unordered_map* to a customized vector-based hashmap for tiny objects, which allocates memory statically to avoid frequent memory reallocation. With these optimizations, the performance and multi-threaded scalability improve significantly. As shown in Figure 13, the performance of Vite is improved by 25.29× for 8 threads, and the speedup increases from 0.56× to 1.46× for 8 threads (2 threads as baseline).

6 Related Work

Performance tools. Existing tools are either based on profiling or tracing. (1) *Profiling-based tools* collect performance data with very low overhead. MpiP [62] is a lightweight profiling library, which provides statistical performance data for MPI functions. HPCToolkit [8], GProf [33], and VTune [52]

are all lightweight profilers for general applications and architectures. Arm MAP [39] and CrayPat [43] are a performance analysis tools specially designed for ARM and Cray X1 platform, respectively. (2) *Tracing-based tools* collect rich information for in-depth analysis [15]. Based on Score-P [4], TAU [5, 56], Vampir [6, 48], and Scalasca [3, 31] provide visualization for generated trace data and provide direct insights. Paraver [2, 45, 55] is a trace-based performance analyzer, which brings great flexibility for data collection and analysis.

Performance analysis. To satisfy the demands of different scenarios, researchers have made great efforts in presenting specific in-depth analysis. Böhme et al. [18] propose an approach to identify the root cause of imbalance by replaying event traces forward and backward. To identify the root cause of bugs, Kairux [71] constructs the longest common prefix of failure and non-failure execution sequence. Tallent et al. [60] propose a light-weight detection technique focusing on lock contention. Böhme et al. [19] and Schmitt et al. [54] detect performance bugs by critical path analysis. Load imbalance analysis [20, 30, 59], critical path analysis [19, 54], and other approaches [21, 27, 68] have been proposed to detect performance bugs.

Graph-based analysis. STAT [12] designs a *3D-Trace/Space/Time Call Graph* with stack traces for large-scale program debugging. CYPRESS [69] and ScalAna [41] generate graphs with program structure and runtime data for communication trace compression and scaling loss detection. wPerf [73] uses a *wait-for graph* with thread-level waiting events to identify bottlenecks. Spindle [64] builds a *Memory-centric Control Flow Graph* for efficient memory access monitoring. PROGRAML [26] represents programs as directed multigraphs and leverages deep learning models for further analysis. Besides, many graph-based analysis approaches are presented for data processing, such as Canopy [42], Dapper [58], X-Trace [29], etc. [34]. These works use graphs to detect information hidden by complex program structures and dependence on traces and profiles.

Dataflow-based programming. There are many frameworks that use dataflow [25] as programming abstraction. TensorFlow [7] uses a dataflow graph to represent machine learning applications. TVM [23], TensorRT [61], PyTorch [49], and Ansor [72] use rule-based strategies to transform dataflow graphs for optimization, while TASO [40] and PET [63] support automatic optimization strategy selection. Dace [16] builds stateful dataflow multigraphs as a unified IR for a pragram. Theano [9] provides a Python framework that allows users to define mathematical expressions. MapReduce [28] eases data processing with two functions, map and reduce. The dataflow-based Dryad graph [38] is proposed for developing large distributed and concurrent applications. The PerFlowGraph is inspired by the above works.

7 Conclusion

In this paper, we present PerFlow, a domain specific programming framework for easing the implementation of in-depth performance analysis tasks. PerFlow provides a dataflow-based programming abstraction, which allows developers to develop customized performance analysis tasks by describing the analysis process as a PerFlowGraph with built-in or user-defined passes. We first propose a Program Abstraction Graph (PAG) to represent the performance of parallel programs, and then build passes with graph operations and algorithms. We also provide some paradigms, which are the specific combinations of passes, for some general and common analysis tasks. Besides, PerFlow provides easy-to-use Python APIs for programming. We evaluate PerFlow with both benchmarks and real-world applications. Experimental results show that PerFlow can effectively ease the implementation of performance analysis and provide insightful guidance for optimization.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. We thank Shengqi Chen, Huanqi Cao, Liyan Zheng, Kezhao Huang, Shiyu Fan, and Xiaoping Huang for their valuable feedback and suggestions. This work is supported by National Key R&D Program of China under Grant 2021YFB0300300, National Natural Science Foundation of China (U20A20226), Beijing Natural Science Foundation (4202031). Jidong Zhai is the corresponding author of this paper (Email: zhaijidong@tsinghua.edu.cn).

References

[1] 2021. PAPI tools. http://icl.utk.edu/papi/software/
[2] 2021. Paraver homepage. Barcelona Supercomputing Center. http://www.bsc.es/paraver
[3] 2021. Scalasca homepage. Julich Supercomputing Centre and German Research School for Simulation Sciences. http://www.scalasca.org
[4] 2021. Score-P homepage. Score-P Consortium. http://www.score-p.org
[5] 2021. TAU homepage. University of Oregon. http://tau.uoregon.edu
[6] 2021. Vampir homepage. Technical University Dresden. http://www.vampir.eu
[7] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI'16)*. 265–283.
[8] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.
[9] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* (2016), arXiv–1605.
[10] Yulong Ao, Chao Yang, Xinliang Wang, Wei Xue, Haohuan Fu, Fangfang Liu, Lin Gan, Ping Xu, and Wenjing Ma. 2017. 26 pflops stencil

computations for atmospheric modeling on sunway taihulight. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS'17)*. IEEE, 535–544.

[11] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1383–1394.

[12] Dorian C Arnold, Dong H Ahn, Bronis R De Supinski, Gregory L Lee, Barton P Miller, and Martin Schulz. 2007. Stack trace analysis for large scale debugging. In *2007 IEEE International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE, 1–10.

[13] Large-scale Atomic and Molecular Massively Parallel Simulator. 2013. Lammps. *available at: http:/lammps. sandia. gov* (2013).

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

[15] Daniel Becker, Felix Wolf, Wolfgang Frings, Markus Geimer, Brian JN Wylie, and Bernd Mohr. 2007. Automatic trace-based performance analysis of metacomputing applications. In *2007 IEEE International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE, 1–10.

[16] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19)*. 1–14.

[17] Arnamoy Bhattacharyya and Torsten Hoefler. 2014. Pemogen: Automatic adaptive performance modeling during program runtime. In *Proceedings of the 23rd international conference on Parallel architectures and compilation (PACT'14)*. 393–404.

[18] David Bohme, Markus Geimer, Felix Wolf, and Lukas Arnold. 2010. Identifying the root causes of wait states in large-scale parallel applications. In *2010 39th International Conference on Parallel Processing (ICPP'10)*. IEEE, 90–100.

[19] David Böhme, Felix Wolf, Bronis R de Supinski, Martin Schulz, and Markus Geimer. 2012. Scalable critical-path based performance analysis. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS'12)*. IEEE, 1330–1340.

[20] D. Bohme, F. Wolf, and M. Geimer. 2012. Characterizing Load and Communication Imbalance in Large-Scale Parallel Applications. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW'12)*. 2538–2541.

[21] Nader Boushehrinejadmoradi, Adarsh Yoga, and Santosh Nagarakatte. 2018. A parallelism profiler with what-if analyses for openmp programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'18)*. IEEE, 198–211.

[22] Greg Bronevetsky, Ignacio Laguna, Saurabh Bagchi, Bronis R de Supinski, Dong H Ahn, and Martin Schulz. 2010. AutomaDeD: Automata-based debugging for dissimilar parallel tasks. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN'10)*. IEEE, 231–240.

[23] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 578–594.

[24] Gabor Csardi, Tamas Nepusz, et al. 2006. The igraph software package for complex network research. (2006).

[25] David E Culler. 1986. Dataflow architectures. *Annual review of computer science* 1, 1 (1986), 225–253.

[26] Chris Cummins, Zacharias V Fisches, Tal Ben-Nun, Torsten Hoefler, and Hugh Leather. 2020. Programl: Graph-based deep learning for program optimization and analysis. *arXiv preprint arXiv:2003.10536* (2020).

[27] Charlie Curtsinger and Emery D Berger. 2015. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. 184–197.

[28] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[29] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. 2007. X-trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI'07).*

[30] T. Gamblin, B.R. de Supinski, M. Schulz, R. Fowler, and D.A. Reed. 2008. Scalable load-balance measurement for SPMD codes. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'08)*. 1–12.

[31] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. 2010. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 702–719.

[32] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, Hao Lu, Daniel Chavarria-Miranda, Arif Khan, and Assefaw Gebremedhin. 2018. Distributed louvain algorithm for graph community detection. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS'18)*. IEEE, 885–895.

[33] Susan L Graham, Peter B Kessler, and Marshall K McKusick. 1982. Gprof: A call graph execution profiler. *ACM Sigplan Notices* 17, 6 (1982), 120–126.

[34] Zhenyu Guo, Dong Zhou, Haoxiang Lin, Mao Yang, Fan Long, Chaoqiang Deng, Changshu Liu, and Lidong Zhou. 2011. G2: a graph processing system for diagnosing distributed systems. In *Proceedings of the 2011 USENIX Conference on Annual Technical Conference (USENIX ATC'11)*. 27–27.

[35] John C Hayes, Michael L Norman, Robert A Fiedler, James O Bordner, Pak Shing Li, Stephen E Clark, Mordecai-Mark Mac Low, et al. 2006. Simulating radiating and magnetized flows in multiple dimensions with ZEUS-MP. *The Astrophysical Journal Supplement Series* 165, 1 (2006), 188.

[36] Mert Hidayetoğlu, Tekin Biçer, Simon Garcia De Gonzalo, Bin Ren, Doğa Gürsoy, Rajkumar Kettimuthu, Ian T Foster, and Wen-mei W Hwu. 2019. Memxct: Memory-centric x-ray ct reconstruction with massive parallelization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19)*. 1–56.

[37] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. Understanding and bridging the gaps in current GNN performance optimizations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'21)*. 119–132.

[38] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. 59–72.

[39] Christopher January, Jonathan Byrd, Xavier Oró, and Mark O'Connor. 2015. Allinea MAP: Adding Energy and OpenMP Profiling Without Increasing Overhead. In *Tools for High Performance Computing 2014*. Springer, 25–35.

[40] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*. 47–62.

[41] Yuyang Jin, Haojie Wang, Teng Yu, Xiongchao Tang, Torsten Hoefler, Xu Liu, and Jidong Zhai. 2020. ScalAna: automating scaling loss detection with graph analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and*

Analysis (SC'20). 1–14.

[42] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. 2017. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. 34–50.

[43] Steve Kaufmann and Bill Homer. 2003. Craypat-cray x1 performance analysis tool. *Cray User Group (May 2003)* (2003).

[44] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E Nagel. 2008. The vampir performance analysis tool-set. In *Tools for high performance computing*. Springer, 139–155.

[45] Jesús Labarta, Sergi Girona, Vincent Pillet, Toni Cortes, and Luis Gregoris. 1996. DiP: A parallel program development environment. In *European Conference on Parallel Processing*. Springer, 665–674.

[46] Sunwoo Lee, Dipendra Jha, Ankit Agrawal, Alok Choudhary, and Wei-keng Liao. 2017. Parallel deep convolutional neural network training by exploiting the overlapping of computation and communication. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC'17)*. IEEE, 183–192.

[47] Paulius Micikevicius. 2009. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. 79–84.

[48] Wolfgang E Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. 1996. VAMPIR: Visualization and analysis of MPI resources. (1996).

[49] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems (NeurIPS'19)* 32 (2019), 8026–8037.

[50] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. 2003. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC'03)*. ACM.

[51] Kiran Ravikumar, David Appelhans, and PK Yeung. 2019. GPU acceleration of extreme scale pseudo-spectral simulations of turbulence using asynchronism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19)*. 1–22.

[52] James Reinders. 2005. VTune performance analyzer essentials. *Intel Press* (2005).

[53] Baruch Schieber and Uzi Vishkin. 1988. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.* 17, 6 (1988), 1253–1262.

[54] Felix Schmitt, Robert Dietrich, and Guido Juckeland. 2017. Scalable critical-path analysis and optimization guidance for hybrid MPI-CUDA applications. *The International Journal of High Performance Computing Applications* 31, 6 (2017), 485–498.

[55] Harald Servat, Germán Llort, Judit Giménez, and Jesús Labarta. 2009. Detailed performance analysis using coarse grain sampling. In *European Conference on Parallel Processing*. Springer, 185–198.

[56] Sameer S Shende and Allen D Malony. 2006. The TAU parallel performance system. *The International Journal of High Performance Computing Applications* 20, 2 (2006), 287–311.

[57] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. GraphPi: high performance graph pattern matching through effective redundancy elimination. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'20)*. IEEE, 1–14.

[58] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).

[59] Nathan R. Tallent, Laksono Adhianto, and John M. Mellor-Crummey. 2010. 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 (SC'10)*. Washington, DC, USA, 1–11.

[60] Nathan R Tallent, John M Mellor-Crummey, and Allan Porterfield. 2010. Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*. 269–280.

[61] NVIDIA TensorRT. 2019. Programmable inference accelerator. *Retrieved August* 1 (2019).

[62] Jeffrey Vetter and Chris Chambreau. 2005. mpip: Lightweight, scalable mpi profiling. (2005).

[63] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. 2021. PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*. 37–54.

[64] Haojie Wang, Jidong Zhai, Xiongchao Tang, Bowen Yu, Xiaosong Ma, and Wenguang Chen. 2018. Spindle: informed memory access monitoring. In *Proceedings of the 2018 USENIX Conference on Annual Technical Conference (USENIX ATC'18)*. 561–574.

[65] Lai Wei and John Mellor-Crummey. 2020. Using sample-based time series data for automated diagnosis of scalability losses in parallel programs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'20)*. 144–159.

[66] William R Williams, Xiaozhu Meng, Benjamin Welton, and Barton P Miller. 2016. Dyninst and MRNet: Foundational infrastructure for parallel tools. In *Tools for High Performance Computing 2015*. Springer, 1–16.

[67] Hisashi Yashiro, Masaaki Terai, Ryuji Yoshida, Shin-ichi Iga, Kazuo Minami, and Hirofumi Tomita. 2016. Performance analysis and optimization of nonhydrostatic icosahedral atmospheric model (NICAM) on the K computer and TSUBAME2. 5. In *Proceedings of the Platform for Advanced Scientific Computing Conference*. 1–8.

[68] Tingting Yu and Michael Pradel. 2016. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 389–400.

[69] Jidong Zhai, Jianfei Hu, Xiongchao Tang, Xiaosong Ma, and Wenguang Chen. 2014. Cypress: combining static and dynamic analysis for top-down communication trace compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. IEEE, 143–153.

[70] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. 2017. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *Proceedings of the 2017 USENIX conference on Annual Technical Conference (USENIX ATC'17)*. 181–193.

[71] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. 2019. The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*. 131–146.

[72] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. 863–879.

[73] Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang. 2018. wPerf: generic Off-CPU analysis to identify bottleneck waiting events. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 527–543.

A Artifact Evaluation Instruction

A.1 Access

Scripts of experiments in this paper are available at https://github.com/thu-pacman/PerFlow.

A.2 Prerequisites

PerFlow is dependent on:

- Dyninst: https://github.com/dyninst/dyninst
- Boost: Boost will be installed automatically with Dyninst.
- PAPI: https://bitbucket.org/icl/papi/src/master/
- igraph: https://github.com/igraph/igraph
- cmake: version >= 3.16

Dyninst and PAPI need to be pre-installed. igraph has been integrated into PerFlow as submodule. Use the following command to download igraph.

```
git submodule update --init
```

There are two ways to build PerFlow. One is to build dependencies with source files and specify their directories when building PerFlow, the other is to use *spack* to build these dependencies.

A.2.1 Building dependencies with source files.

After building dependencies, use the following command to install PerFlow.

```
cmake .. -DBOOST_ROOT=/
  path_to_your_boost_install_dir -DDyninst_DIR=/
  path_to_your_dyninst_install_dir/lib/cmake/
  Dyninst -DPAPI_PREFIX=/
  path_to_your_papi_install_dir

# Make sure that there exists `DyninstConfig.
  cmake` in /path_to_your_dyninst_install_dir/
  lib/cmake/Dyninst,
# and there exist `include` and `lib` in /
  path_to_your_papi_install_dir and /
  path_to_your_boost_install_dir.
```

Note that if Dyninst is built with source files, the Boost will be downloaded and installed automatically in the install directory of Dyninst.

A.2.2 Building dependencies with *spack*.

The recommended way to build Dyninst (with Boost) and PAPI is to use *spack* (https://github.com/spack/spack). First use the following commands to install and load dependencies.

```
spack install dyninst # boost will be installed
   at the same time
spack install papi
spack load dyninst # boost will be loaded at
  the same time
spack load papi
```

Then use the following command to build PerFlow.

```
mkdir build && cd build && cmake ..
```

A.3 Using PerFlow

PerFlow provides built-in analysis passes and paradigms, as well as low-level APIs for developers. An MPI profiler paradigm and a critical path detection task are used to show how to use PerFlow.

A.3.1 MPI profiler.

The MPI profiler paradigm is a built-in analysis paradigm. For evaluation, it is performed on an MPI program NPB-CG (CLASS=B and 8 processes).

Use the following command to perform the MPI profiler paradigm.

```
cd build/example/AE/model_validation
python3 ./model_validation.py # python3
```

A.3.2 Critical path detection.

To implement the critical path detection task, a user-defined pass is written with PerFlow's low-level APIs. For evaluation, this task is performed on a multi-threaded micro-benchmark (a PTthreads program).

Use the following command to perform the critical path detection task.

```
cd build/example/AE/pass_validation
python3 ./pass_validation.py # python3
```