IEEE CLUSTER 2025
Edinburgh, Scotland • 2–5 September

Tutorial: Detecting Performance Variance on Large-Scale Heterogeneous Systems

Xin You

Beihang University

Hands-on Tutorial @ CLUSTER25

北京航空航天大學
BEIHANG UNIVERSITY

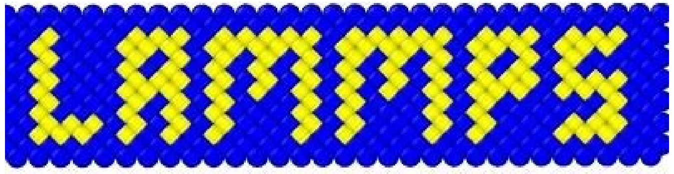
Outline

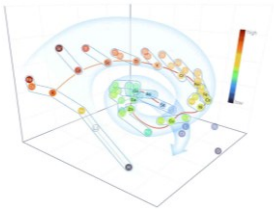
- Introduction
- Design Overview
- Detecting Performance Variance & Implementation
- Evaluation
- Hand-on Tutorial

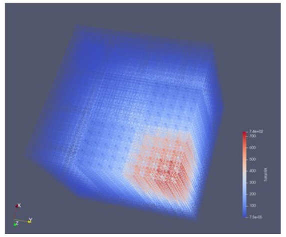
Outline

- Introduction
- Design Overview
- Detecting Performance Variance & Implementation
- Evaluation
- Hand-on Tutorial

Large-Scale Heterogenous System





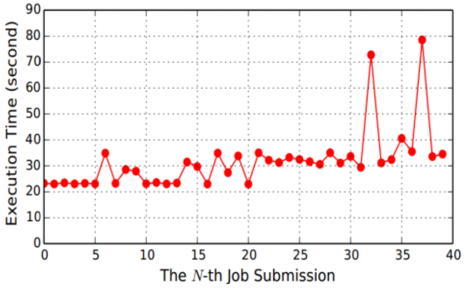


Various production and scientific applications

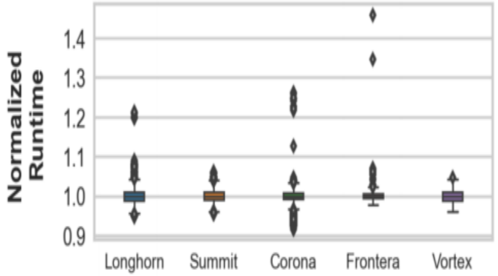
Computational support

Frontier: >37K GPUs
X86 + GPU (#1@2024.6)

Aurora: >63K GPUs
X86 + GPU (#2@2024.6)



Performance variance in homogeneous systems (Zheng et.al., PPoPP22)



Performance variance in heterogenous systems (Sinha et.al., SC22)

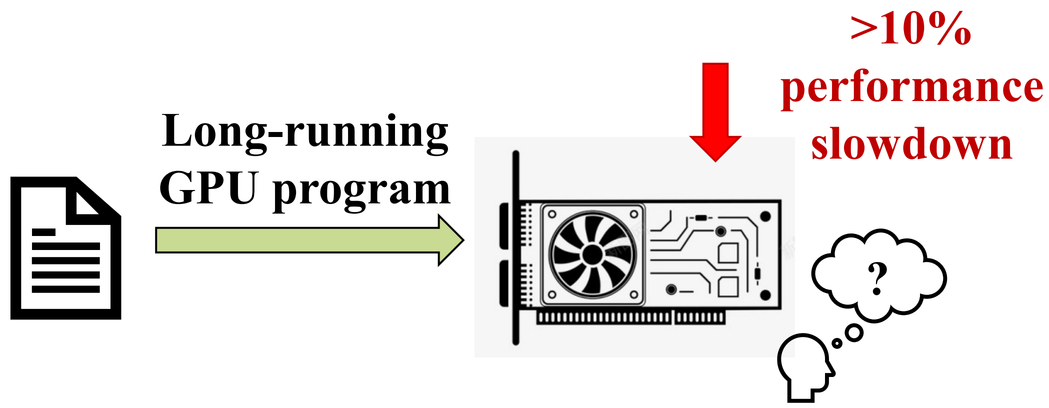
The attainable parallel program performance has **become more and more unstable**

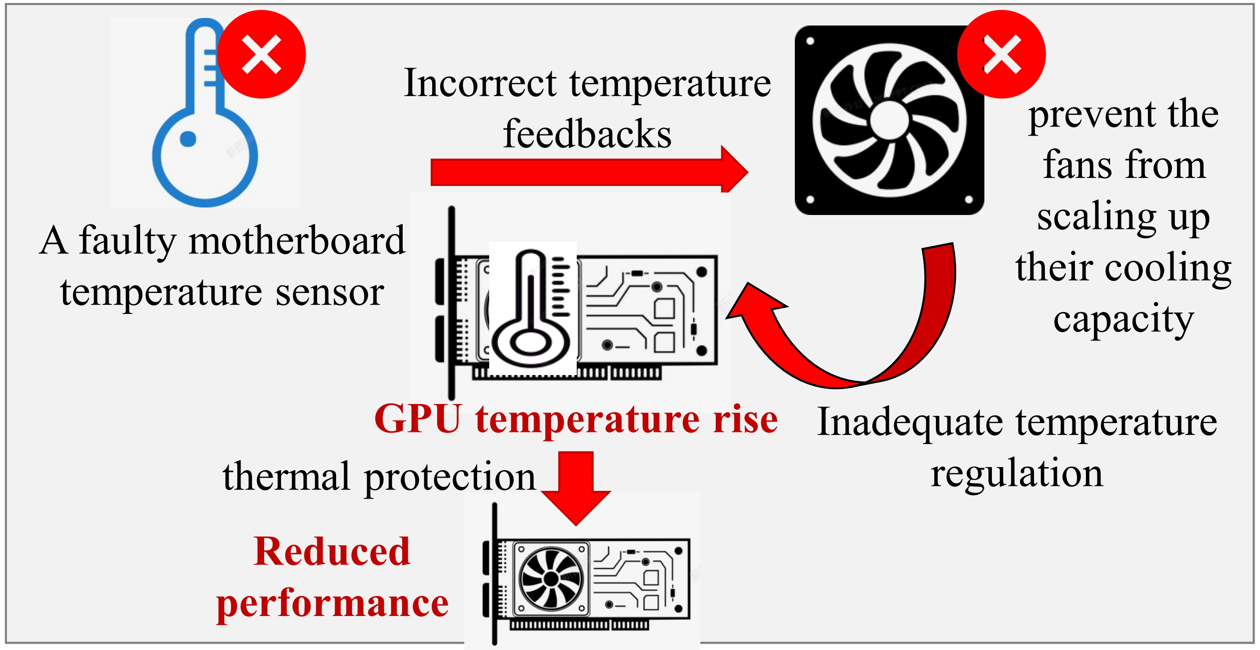
Scale of GPU cluster increases

Performance variance has become one of the nasty pitfalls when running parallel programs on such large-scale heterogeneous systems.

Diagnosing Performance Variance: Real Case

- Performance variance often suffers from its spontaneity, unpredictability, and the diversity of root causes
  - making it exceedingly difficult to detect and pinpoint the underlying reasons for potential performance variance during program execution.



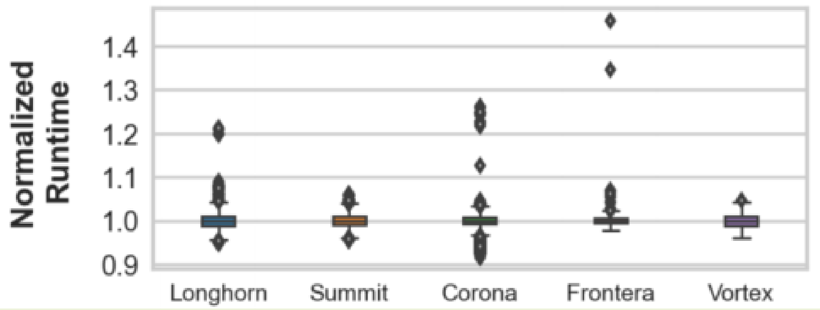


Pinpointing the precise root causes of such performance variances remains a formidable task for both developers and system maintainers

Challenges in Existing Approaches

Microbenchmark based

- The most widely adopted approaches to discover and diagnose the source of the performance variance
- Require the same well-formed workloads running on different computation nodes for performance variance detection



✓ Uncovered performance variances in large-scale heterogeneous systems caused by GPU hardware variances.

- ☐ Based on long-term data collection using specific micro-benchmarks
- ☐ Cannot capture and identify potential performance variances during program execution

Challenges in Existing Approaches

Microbenchmark based

- The most widely adopted approaches to discover and diagnose the source of the performance variance
- Require the same well-formed workloads running on different computation nodes for performance variance detection

detecting and reasoning performance variance within a single parallel execution.

Fixed workload based

- Identified several fixed-workload code snippets that can be treated as probes for performance variance detection of homogeneous systems
  - Based on observations that codes with similar workload should result in similar performance with the same hardware and software specifications

Challenges in Existing Approaches

✓ Address performance variances and root cause analysis to some extent in large-scale homogeneous systems

- ☐ Fail to track asynchronous communications
- ☐ They are not directly applicable to heterogeneous systems
- ☐ GPU diagnosis tools (e.g., DrGPU) incurs high overhead to pinpoint the causes of poor performance on GPU-based programs

Still lack an effective tool to detect performance variance for large-scale heterogeneous systems

Fixed workload based

- Identified several fixed-workload code snippets that can be treated as probes for performance variance detection of homogeneous systems
  - Based on observations that codes with similar workload should result in similar performance with the same hardware and software specifications

Challenges in Existing Approaches

✓ Address performance variances and root cause analysis to some extent in large-scale homogeneous systems

- ☐ Fail to track asynchronous communications
- ☐ They are not directly applicable to heterogeneous systems
- ☐ GPU diagnosis tools (e.g., DrGPU) incurs high overhead to pinpoint the causes of poor performance on GPU-based programs

Key challenges：

1) How to identify fixed-workload probes within parallel programs for heterogeneous systems?

➢ including kernel, CPU-GPU data transfer, sync/async communications, etc.

2) How to collect performance data for performance variance detection at a low cost?

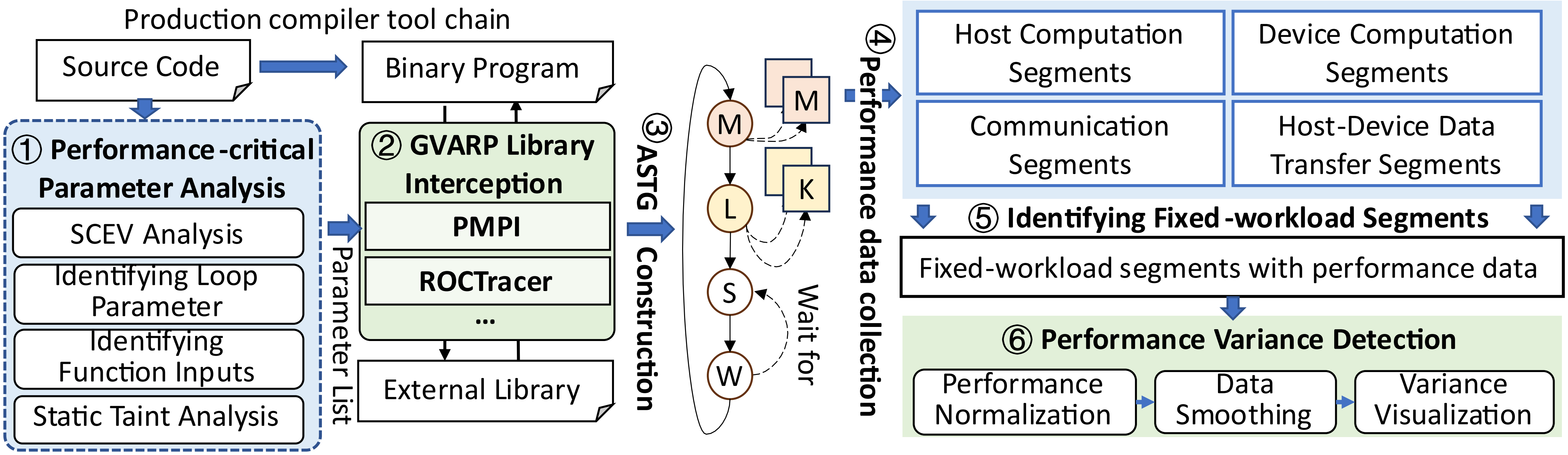
➢ high overhead may mislead the identification of performance variances

Outline

- Introduction
- Design Overview
- Detecting Performance Variance & Implementation
- Evaluation
- Hand-on Tutorial

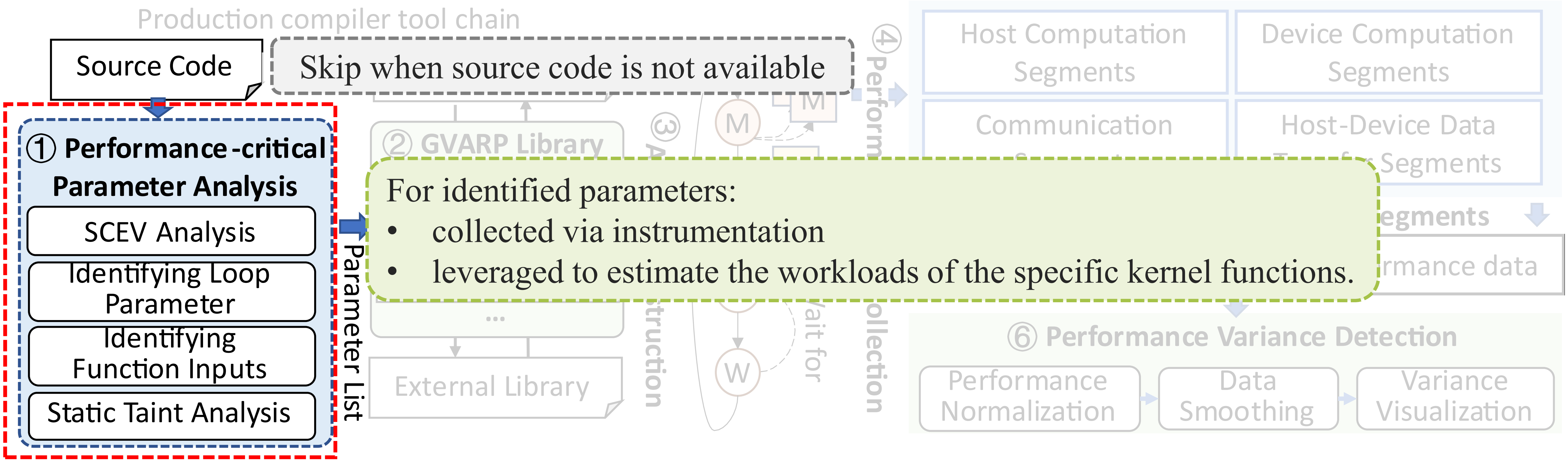
Design Overview

- We propose GVARP, a performance variance detection tool for large-scale heterogeneous systems.
  - No mandatory need for customized compiler chains or recompilation



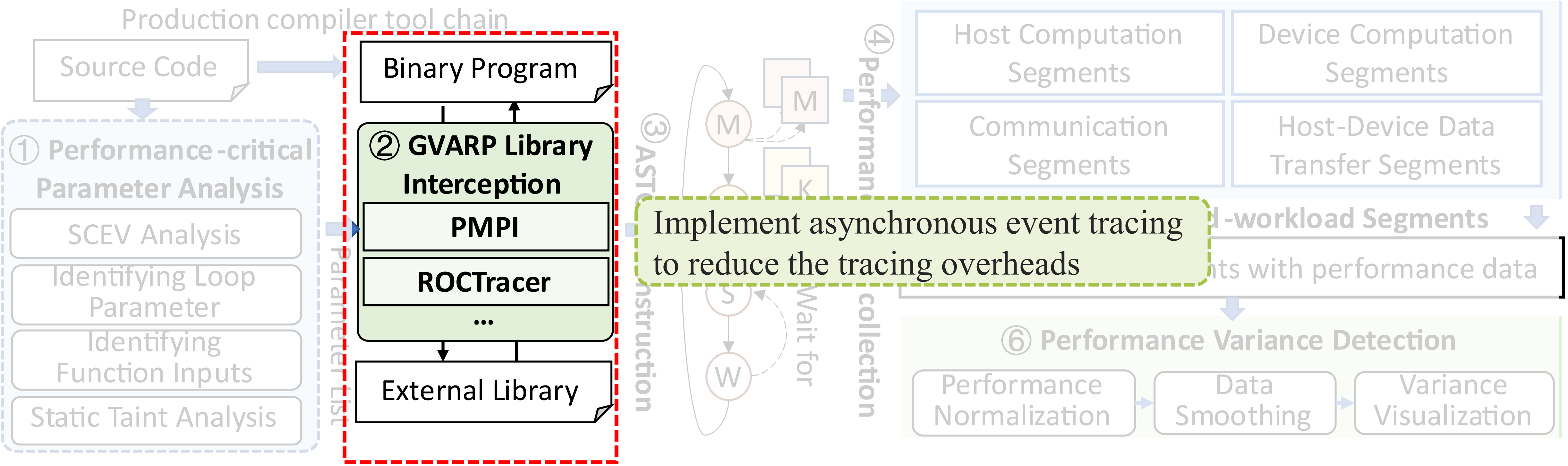
Design Overview

- We propose GVARP, a performance variance detection tool for large-scale heterogeneous systems.
  - 1) Analyzes the source code to identify all accelerated kernel functions on GPU and identify the performance-critical parameters via static taint analysis



Design Overview

- We propose GVARP, a performance variance detection tool for large-scale heterogeneous systems.
  - 2) GVARP divides the execution of a parallel program into internal and external program segments with external library calls.

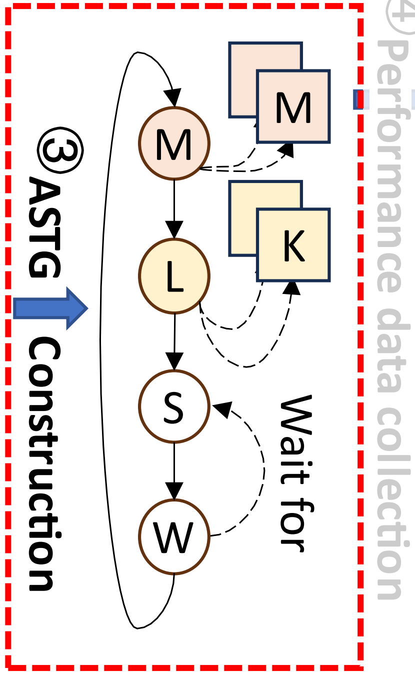


Design Overview

- We propose GVARP, a performance variance detection tool for large-scale heterogeneous systems.
  - 3) GVARP constructs Asynchronous State Transition Graph (ASTG) from the collected traces to represent the program execution

Production compiler tool chain

- Nodes (○, e.g., *M, L, S, W*)
  - host-side external program segments
- Nodes (□, e.g., *M, K*)
  - device-side asynchronous events
- Edges (→, e.g., *M*→*L*)
  - host-side internal program segments
- Edges (⇢, e.g., W→S)
  - asynchronous event annotations



Host Computation Segments

Device Computation Segments

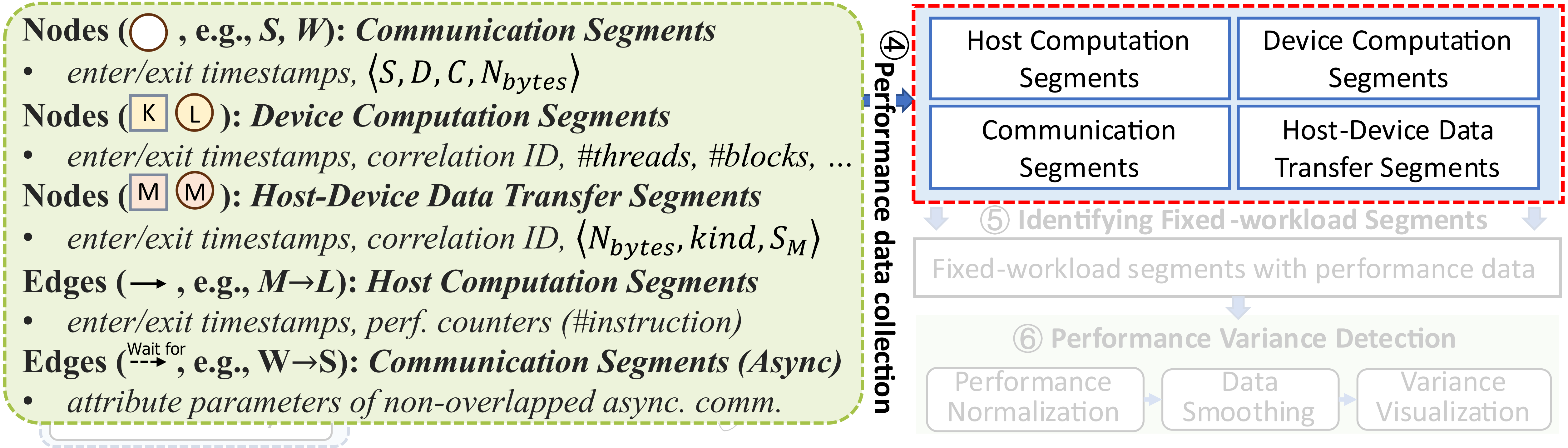
- Asynchronous kernel execution (*L, K*),
- Host-device data transfers (*M*),
- Wait-for relationships of asynchronous MPI communication (*W*→*S*, …)

⑥ Performance Variance Detection

Performance Normalization → Data Smoothing → Variance Visualization

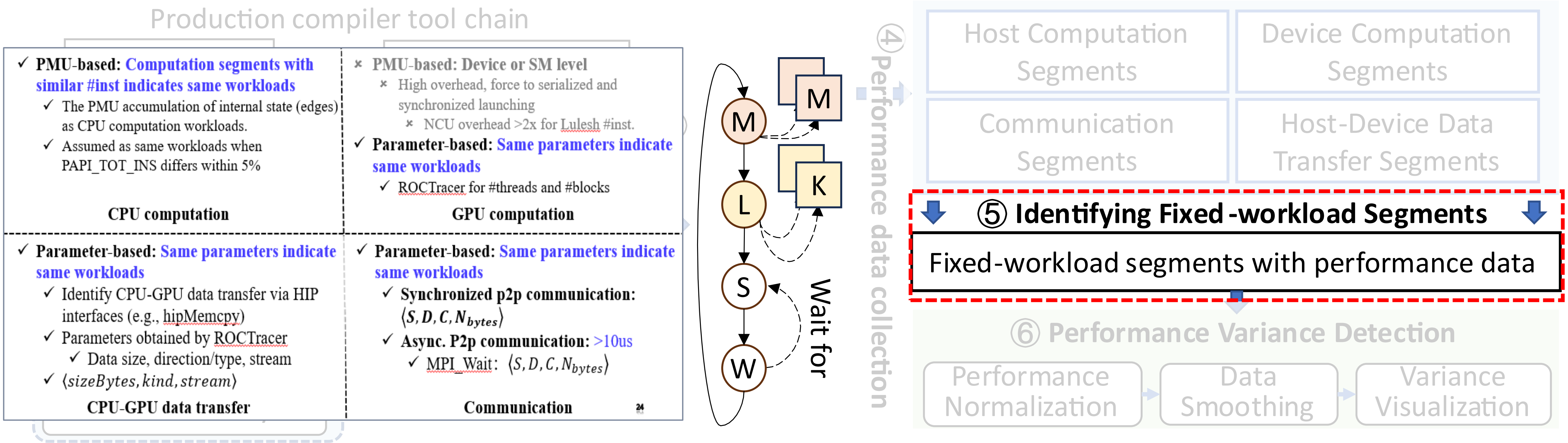
Design Overview

- We propose GVARP, a performance variance detection tool for large-scale heterogeneous systems.
  - 4) Attributes each program segments with performance data, including enter and exit timestamps, correlation identifiers, function parameters, and performance counters.



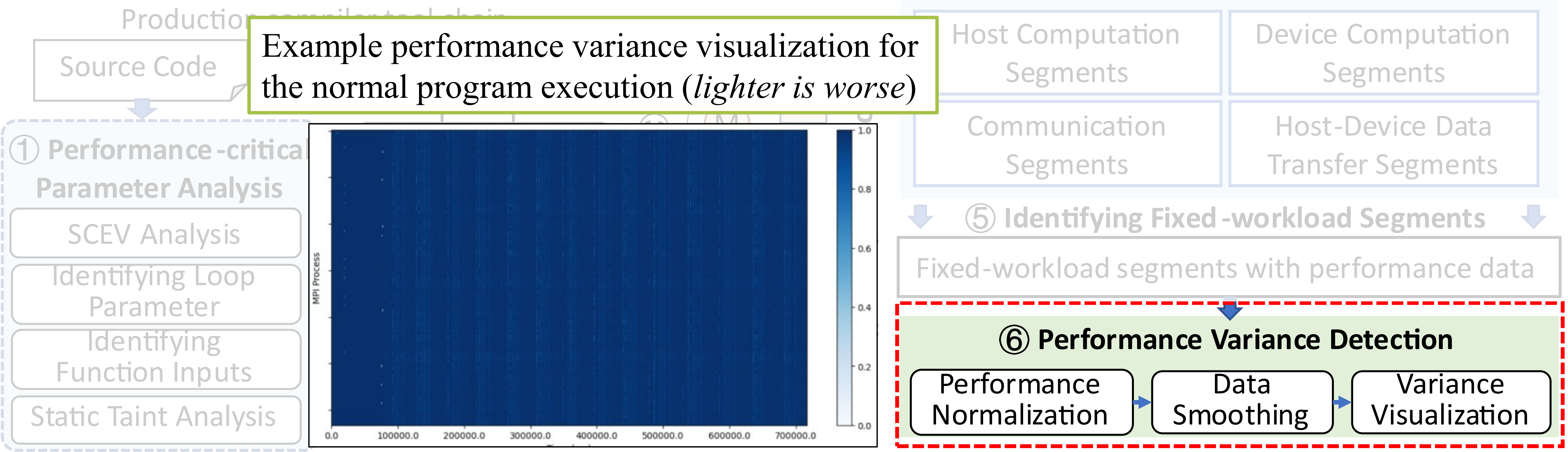
Design Overview

- We propose GVARP, a performance variance detection tool for large-scale heterogeneous systems.
  - 5) For each type of program segment, GVARP leverages the ASTG-based clustering to identify fixed-workload segments for further performance variance detection.



Design Overview

- We propose GVARP, a performance variance detection tool for large-scale heterogeneous systems.
  - 6) For each cluster of fixed-workload segments, GVARP adopts performance normalization and data smoothing for comparable and noiseless variance metrics.

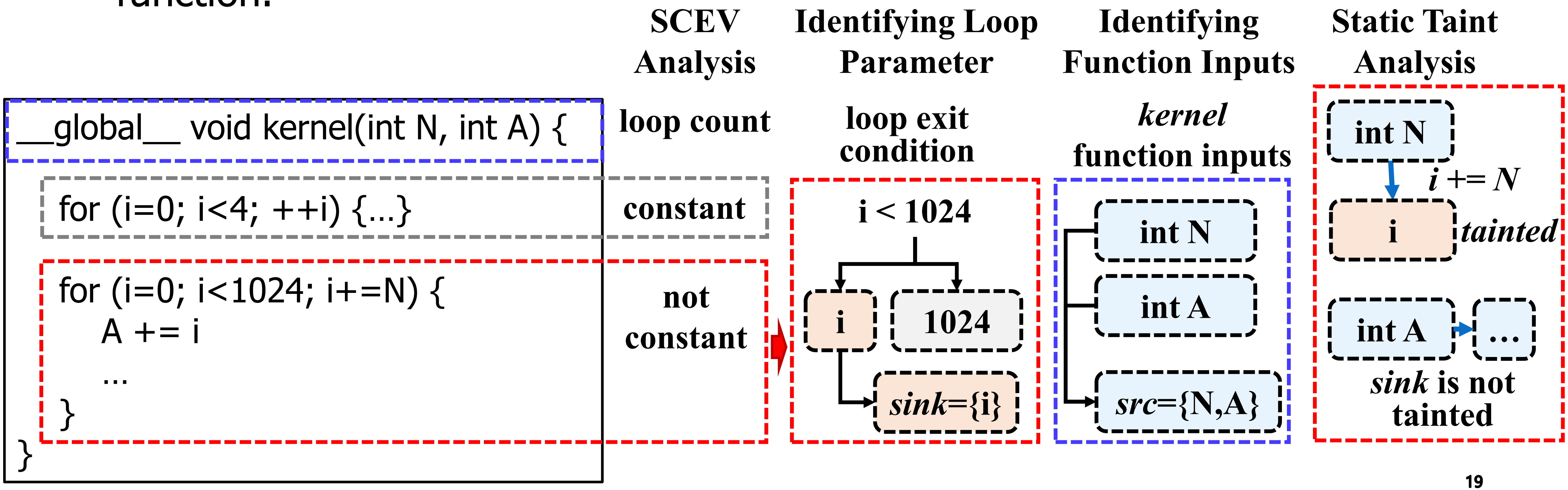


Outline

- Introduction
- Design Overview
- Detecting Performance Variance & Implementation
- Evaluation
- Hand-on Tutorial

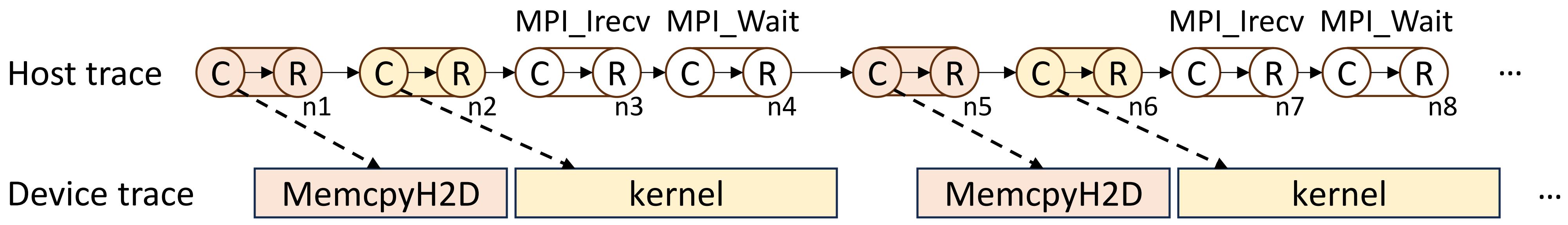
Performance-critical Parameter Analysis

- ***Key Idea*: If the value of kernel function parameter can affect the value of loop condition variables, such parameters are *performance-critical.***
  - The performance-critical parameters can affect the loop counts within the function.



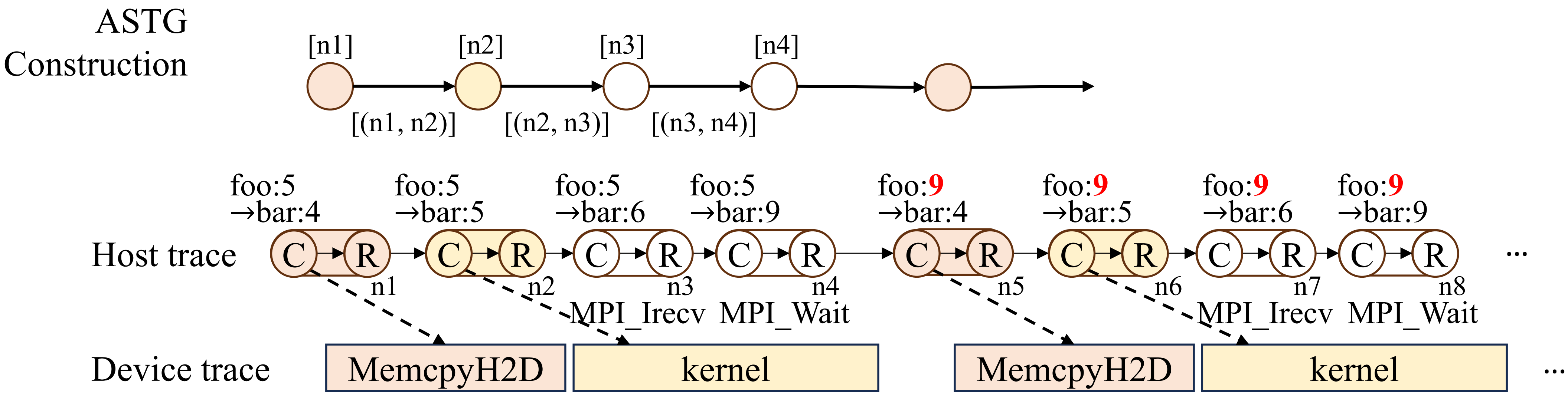
ASTG Construction

- After collecting target program's performance trace, GVARP constructs Asynchronous State Transition Graph (ASTG) based on the collected traces
  - CPU-GPU data transfer (*orange icons*), GPU kernel launch (*yellow icons*) and MPI communication (*white icons*)



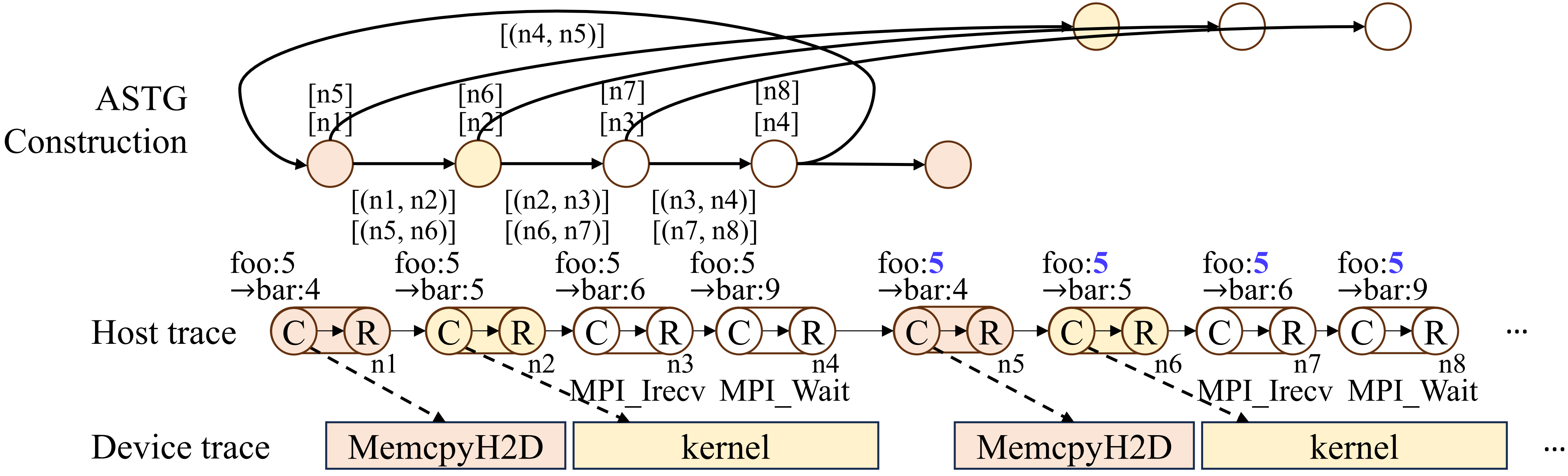
ASTG Construction

- After collecting target program's performance trace, GVARP constructs Asynchronous State Transition Graph (ASTG) based on the collected traces
  - CPU-GPU data transfer (*orange icons*), GPU kernel launch (*yellow icons*) and MPI communication (*white icons*)
  - *C, R* indicates function *Call* and *Return*, where *C*→*R* indicates *external program segments*, and *R*→*C* indicates *internal program segments*



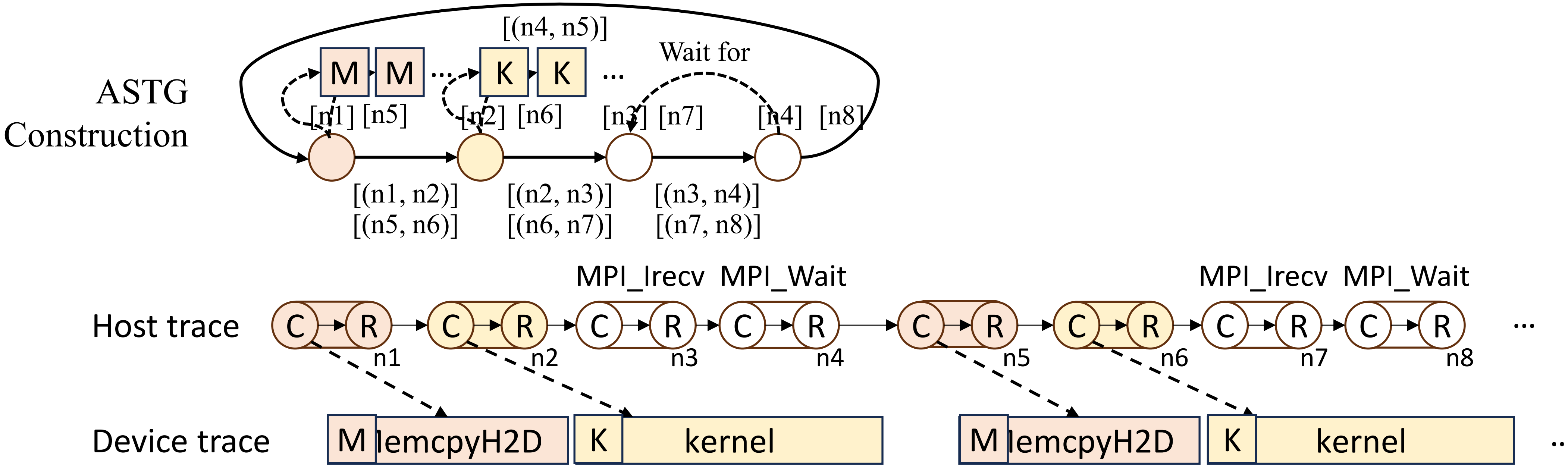
ASTG Construction

- After collecting target program's performance trace, GVARP constructs Asynchronous State Transition Graph (ASTG) based on the collected traces
  - CPU-GPU data transfer (*orange icons*), GPU kernel launch (*yellow icons*) and MPI communication (*white icons*)
  - *C, R* indicates function *Call* and *Return*, where *C*→*R* indicates *external program segments*, and *R*→*C* indicates *internal program segments*



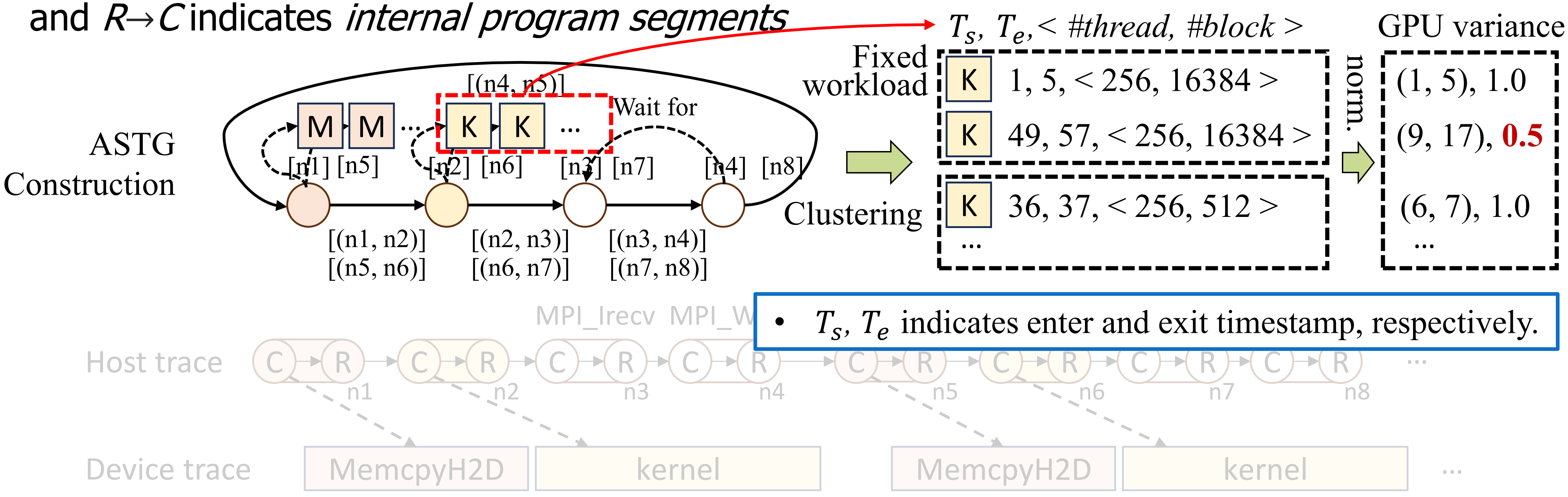
ASTG Construction

- After collecting target program's performance trace, GVARP constructs Asynchronous State Transition Graph (ASTG) based on the collected traces
  - CPU-GPU data transfer (*orange icons*), GPU kernel launch (*yellow icons*) and MPI communication (*white icons*)
  - *C, R* indicates function *Call* and *Return*, where *C*→*R* indicates *external program segments*, and *R*→*C* indicates *internal program segments*



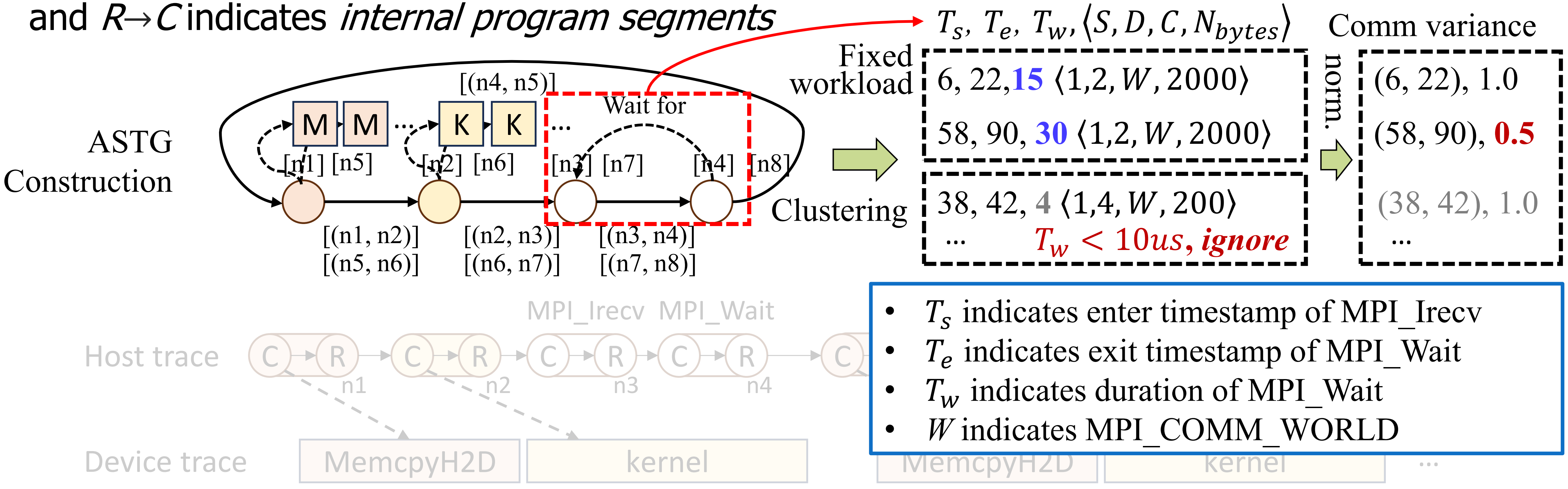
ASTG-based Performance Variance Detection

- Based on the constructed ASTG, GVARP can identify the fixed workload program segments for further performance variance detection
  - CPU-GPU data transfer (*orange icons*), GPU kernel launch (*yellow icons*) and MPI communication (*white icons*)
  - *C, R* indicates function *Call* and *Return*, where *C→R* indicates *external program segments*, and *R→C* indicates *internal program segments*



ASTG-based Performance Variance Detection

- Based on the constructed ASTG, GVARP can identify the fixed workload program segments for further performance variance detection
  - CPU-GPU data transfer (*orange icons*), GPU kernel launch (*yellow icons*) and MPI communication (*white icons*)
  - *C, R* indicates function *Call* and *Return*, where *C*→*R* indicates *external program segments*, and *R*→*C* indicates *internal program segments*

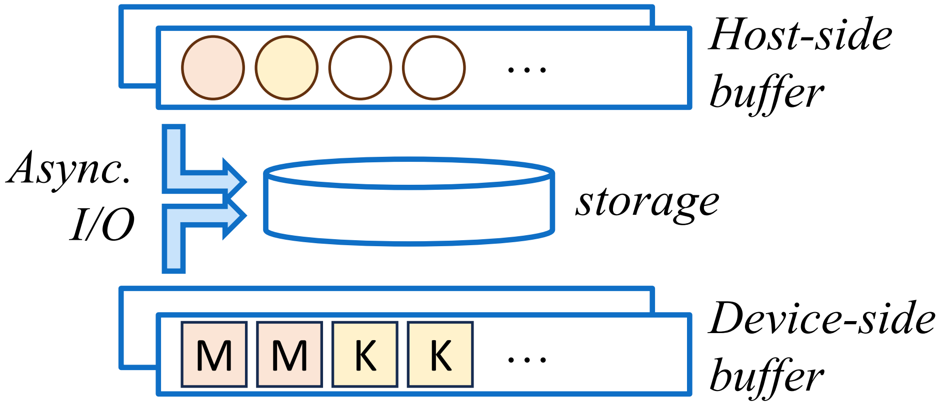


- T_s indicates enter timestamp of MPI_Irecv
- T_e indicates exit timestamp of MPI_Wait
- T_w indicates duration of MPI_Wait
- W indicates MPI_COMM_WORLD

Implementation - Low cost at scale

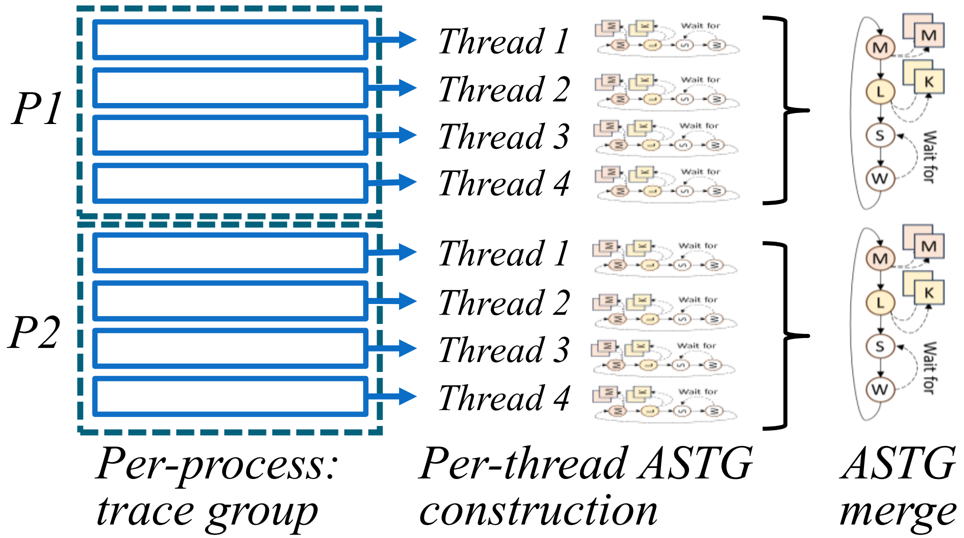
- For detecting performance variances at scale, GVARP implements the above methods efficiently with the following techniques:

1) Tracing efficiency:



Implement host-device dual buffer enabled asynchronous event tracing within GVARP library interception

2) Analysis efficiency:



Implement communication-free MPI and OpenMP hybrid parallelization for efficient ASTG construction

Implementation - Low cost at scale

- For detecting performance variances at scale, GVARP implements the above methods efficiently with the following techniques:

1) Tracing efficiency:

2) Analysis efficiency:

For more details, please refer to our paper.

X. You, Z. Xuan, H. Yang, Z. Luan, Y. Liu and D. Qian, "GVARP: Detecting Performance Variance on Large-Scale Heterogeneous Systems," SC24: International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, 2024, pp. 1-13.

GVARP is available: https://doi.org/10.5281/zenodo.10975567

Implement host-device dual buffer enabled asynchronous event tracing within GVARP library interception

Implement communication-free MPI and OpenMP hybrid parallelization for efficient ASTG construction

Outline

- Introduction
- Design Overview
- Detecting Performance Variance & Implementation
- Evaluation
- Hand-on Tutorial

Evaluation Setup

- We evaluate GVARP on an AMD GPU cluster with up to 16,000 GPUs
- 3 representative GPU-accelerated HPC program
  - (1) HPCG: a high-performance conjugate gradients benchmark
  - (2) ANT-MOC: a scalable neutron transport equation solver
  - (3) LAMMPS: a widely adopted molecular dynamics simulator in various domains
- Both ANT-MOC and LAMMPS achieves large scale parallelization of **16,000 MPI processes and 16,000 GPUs**

Platform	GPU Cluster
CPU	AMD Zen-based processor @ 2.5GHz
GPU	4 AMD Instinct M160 GPUs
Cores	32
Memory	128 GB (host), 16 GB (GPU)
Network	200 Gbps HDR InfiniBand network
Storage	> 200 Gbps
Software	GCC 9.3.1, ROCM≥3.9, OpenMPI 4.0.4

Detection Coverage

- We evaluate with 128 processes + 128 GPUs (small scale varification) and 16,000 processes+16,000 GPUs (whole machine scale)

Program	Scale	Sync-only	Detailed Coverage				Coverage	
		Comm.	Host Comp.	**Comm.**	Device Comp.	Host-device Data Transfer	Host	Device
HPCG	128	**0.00%**	70.60%	**0.02%**	0.01%	0.00%	70.62%	0.01%
ANT-MOC	128	**0.00%**	3.05%	**2.08%**	**67.52%**	**6.20%**	5.13%	73.72%
LAMMPS	16,000	**0.21%**	9.45%	**0.32%**	**0.18%**	**4.46%**	9.77%	4.64%
ANT-MOC	16,000	**0.00%**	9.69%	**0.01%**	**0.92%**	**0.82%**	9.70%	1.74%

GVARP can identify some asynchronous communications as fixed-workload segments through its ASTG-based communication workload identification

GVARP can identify both GPU data transfer and computation as probes for variance detection

Overhead

- We execute at least 3 times for each evaluated program and choose the fastest execution time as the reported evaluation results

Program	Scale	Overhead		
		Time	Storage	Analysis (s)
HPCG	128	1.00×	8.9 GB	877.5
ANT-MOC	128	1.00×	2.0 GB	447.04
LAMMPS	**16,000**	**1.14×**	**70.0 GB**	**288.04**
ANT-MOC	**16,000**	**1.16×**	**60.0 GB**	**390.51**

Analysis with 1 node

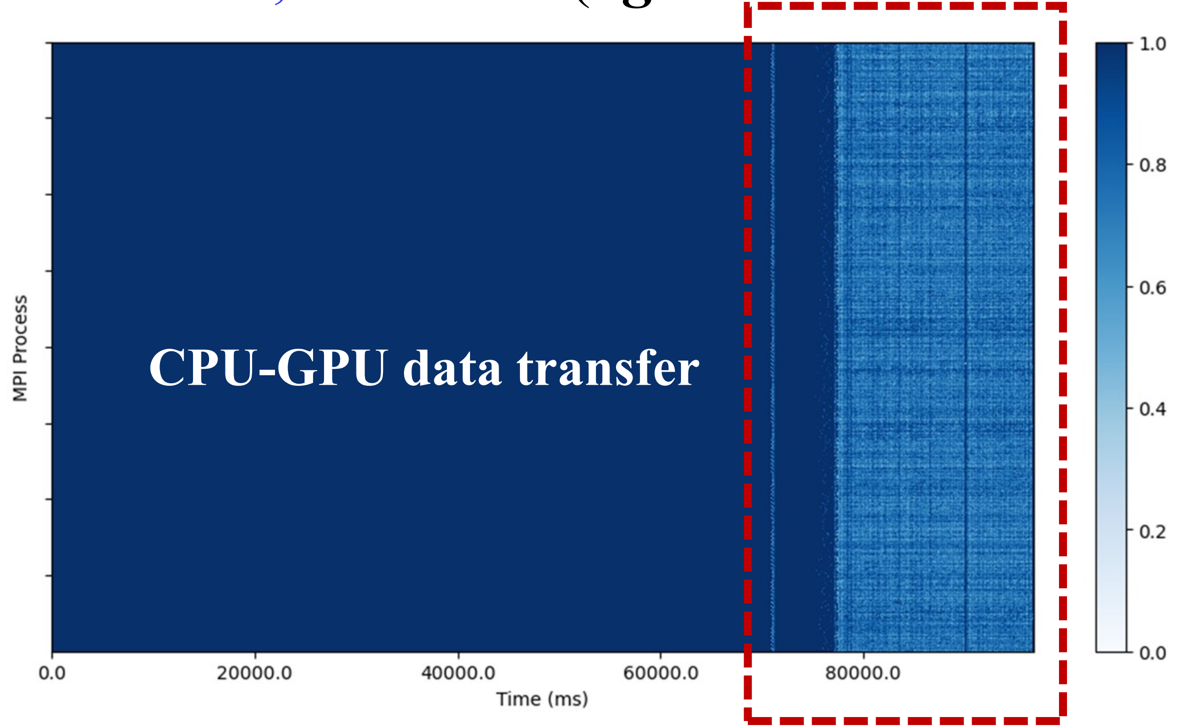
Analysis with 4 node

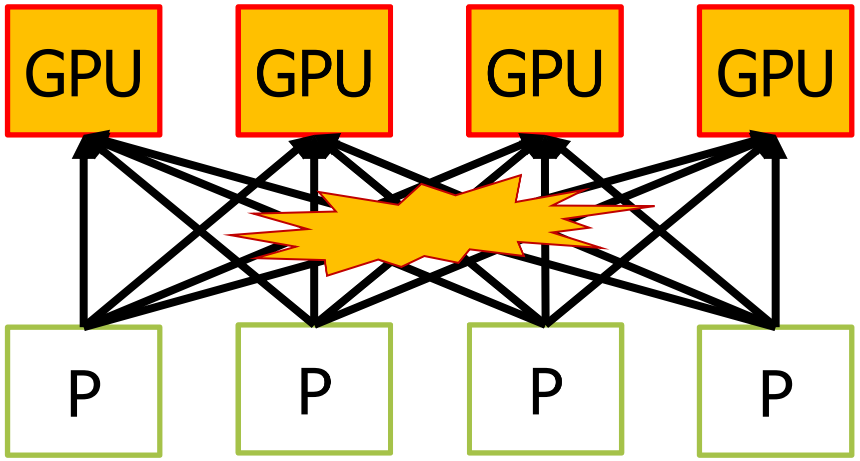
Such low time overheads can be attributed to the asynchronous event tracing techniques adopted in GVARP implementation.

GVARP requires several minutes for performance variance detection analysis, which is also acceptable to perform in free debug nodes

Case Study: Detection of Launching Problem (LAMMPS)

- The performance variance detected in host-device data transfer for executing LAMMPS with 16,000 GPUs (***lighter cells indicates more severe performance variances***)



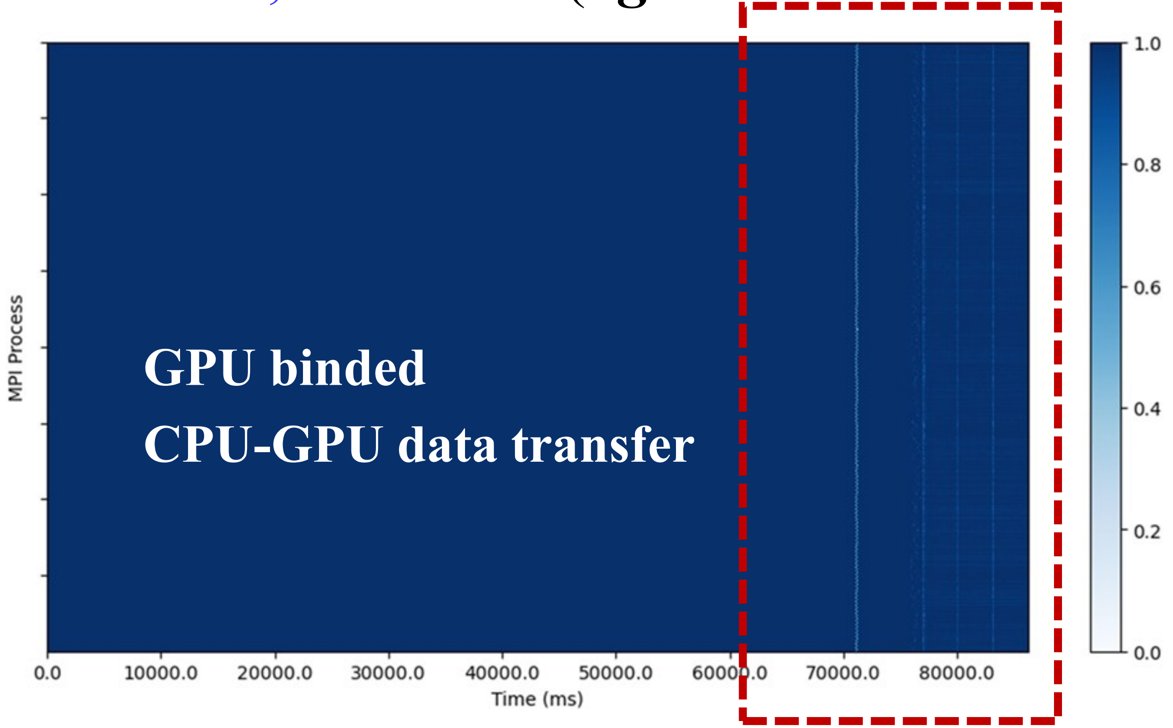


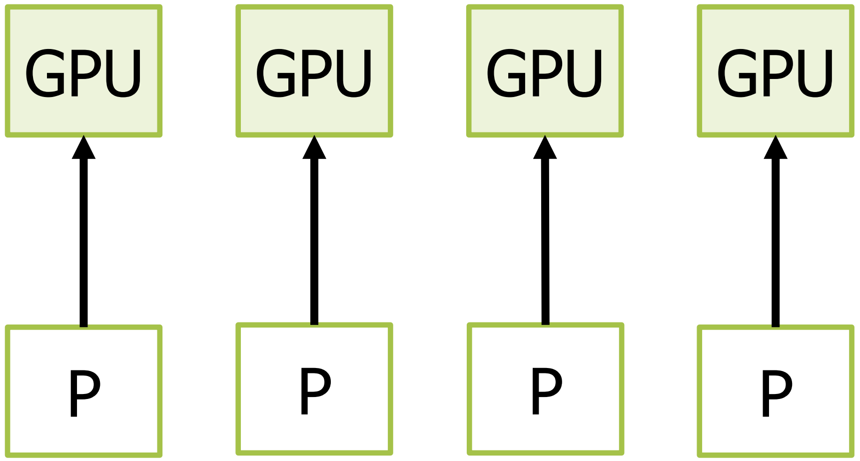
LAMMPS wrongly binds the same GPU for acceleration with improper launching configurations

- Such performance variance must come from strong interference along PCIE data transition

Case Study: Detection of Launching Problem (LAMMPS)

- The performance variance detected in host-device data transfer for executing LAMMPS with 16,000 GPUs (***lighter cells indicates more severe performance variances***)





Properly bind from launch configuration

- Such performance variance must come from strong interference along PCIE data transition
- **After properly binded, we achieve 1.56x performance speedup**

Outline

- Introduction
- Design Overview
- Detecting Performance Variance & Implementation
- Evaluation
- Hand-on Tutorial
  - Installation
  - Case Study – ANT-MOC
  - Case Study – LAMMPS

Installation - GVARP

- **Install with source code:**
  - Dependencies: git, gcc>=12, cmake>=3.20, libunwind, … (*can install with spack*)
  - Source Code:
    ```
    git clone https://github.com/buaa-hipo/MSToolkit.git
    ```
  - Compilation Instruction:
    ```
    cd MSToolkit && ./build.sh
    ```
  - Configure the Path:
    ```
    source env.sh
    ```
- **Use pre-installed version in tutorial cluster:**
  - Configure the Tool Path:
    ```
    source ~/GVARP-Tutorial/examples/ANT-MOC/setup-env.sh
    # source ~/GVARP-Tutorial/examples/LAMMPS/setup-env.sh
    conda activate mstoolkit # load python env to dump figures
    ```
- **Instruction to analyze the target program:**
  - Tracing with ***jsirun***:
    ```
    mpirun –n ${NP} jsirun <tool options> -- <EXE> <ARGS>
    ```
  - Variance detection:
    ```
    variance_analysis –i <TRACE DIR> -o <RESULT_DIR>
    ```

GVARP – More usage details

- **Support various configurations for *jsirun* to collect data including:**
  - MPI events with parameters (***enabled by default***)
  - ROCM/HIP API calls & device events (***--accl***)
  - Backtrace collection (***--backtrace***)
  - Performance counters (***--pmu***), events are defined in environment variable:

    ```
    export JSI_COLLECT_PMU_EVENT=PAPI_TOT_INS,PAPI_L1_DCA
    ```

  - …
- **Performance variance detection with *variance_analysis*:**
  - Resolution (ms) for heatmap generation (***-r, --resolution arg***), default is ***100ms***
  - One can also change the reference PMU events for host computation workload estimation (***-m, --reference-metric arg***), default is ***PAPI_TOT_INS***

Case Study: ANT-MOC: Tracing

~3 min

- Get the tutorial cases for ANT-MOC

```
cp -r /public/home/dfcs2025/GVARP-Tutorial/examples/ANT-MOC ./
```

- Tracing for ANT-MOC normal execution

```
cd ANT-MOC/run-ori
./slurm-tool.job # generate & submit ANT-MOC jobs with and without tool
```

Modify the Job Name

```
#!/bin/bash
NTASKS=16
JOBNAME="MOC-blk"
RUN_DIR=`dirname $0`
echo "TASK MOC C5G7 TEST START NTASK=$NTASKS "
NEWMOC=/public/home/buaa_hipo/CLUSTER25-Tutorial/app/ANT-MOC/ANT-MOC/build/run/newmoc
MEASUREMENT_DIR_ORI=measurement/measurement-antmoc-ori
nowdate=$(date +%Y_%m_%d_%H_%M_%S)
echo $nowdate
sbatch << END
#!/bin/bash
#SBATCH -J $JOBNAME
#SBATCH -o log/c5g7/c5g7-$NTASKS-%j-$nowdate.log
#SBATCH -e log/c5g7/c5g7-$NTASKS-%j-$nowdate.err
#SBATCH -p test
#SBATCH --cpus-per-task=7
#SBATCH --ntasks-per-node=4
#SBATCH --gres=dcu:4
#SBATCH --mem=100GB
#SBATCH -n $NTASKS
cd $RUN_DIR && source $RUN_DIR/../setup-env.sh
export OMP_NUM_THREADS=1
export JSI_BACKTRACE_MAX_DEPTH=5
export JSI_COLLECT_PMU_EVENT=PAPI_TOT_INS,PAPI_L1_DCA
rm -rf $MEASUREMENT_DIR_ORI
/usr/bin/time -v mpirun -n $NTASKS $NEWMOC --config="./config.yaml"
/usr/bin/time -v mpirun -n $NTASKS jsirun --accl --backtrace --pmu -o $MEASUREMENT_DIR_ORI -- $NEWMOC --config="./config.yaml"

END
```

Modify the TRACE FILE PATH

Unoptimize
d time

Profiling time

Slides is available at tutorial home page:

https://buaa-hipo.github.io/blog/mstoolkit-tutorial-cluster25/

Case Study: ANT-MOC: Analysis of normal execution **~2 min**

- Load the mstoolkit environments for variance analysis

```
conda activate mstoolkit
source ../setup-env.sh
```

- Submit an analysis job and the result file is located in folder ***variance/***

```
srun -n 1 --cpus-per-task 2 -p sep2 --exclusive --mem=100GB variance_analysis -i measurement/measurement-antmoc-ori -o variance -f --reference-metric-add PAPI_L1_DCA --resolution 10
```

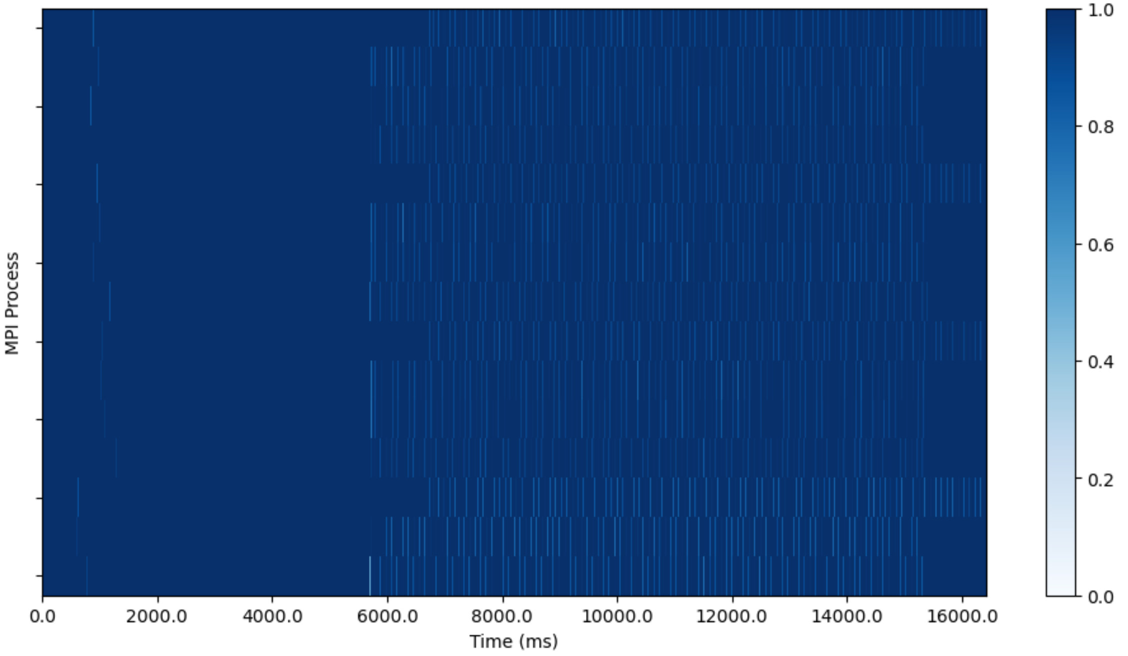
  - ***Contains 5 result files in csv format:*** *accl_calc_heat.csv, accl_memcpy_heat.csv, calc_heat.csv, comm_heat.csv, host.csv*
  - Indicates analysis results for *device computation, host-device transfer, host computation, communication* and *process-host mapping* information, respectively

- Draw heatmap figures with the provided script

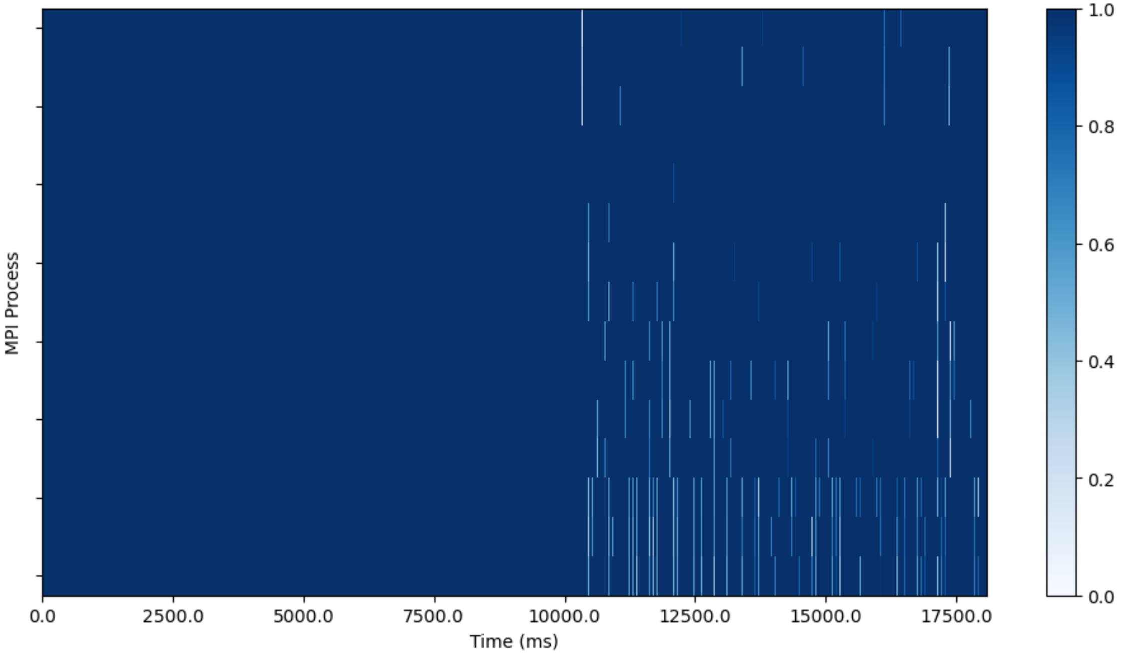
```
HEATMAP_PY=$MSTOOLKIT_PATH/scripts/analysis/variance/heatmap.py
python $HEATMAP_PY --input variance --output figure
```

Case Study: ANT-MOC: Analysis of normal execution

- Resulting files are located in the folder ***figure/***
  - ***Contains 4 result figures:*** *accl_calc.png, accl_memcpy.png, calc.png, comm.png*
  - Indicates analysis results for *device computation, host-device transfer, host computation,* and *communication*, respectively
  - ***The lighter indicates worser performance***



accl_calc.png



comm.png

Case Study: Inject Device Computation Workloads

~6 min

- Now try to inject device-only computation workload along with ANT-MOC execution to simulate severe device-side performance variance.
- Injected computation-intensive kernel
  - Located in **run-accl-calc/inject-accl-calc/stress.cpp**
  - **Injected after 20 seconds, last for 20 seconds**
- Provide a reference job submission scripts for *inject & tracing*

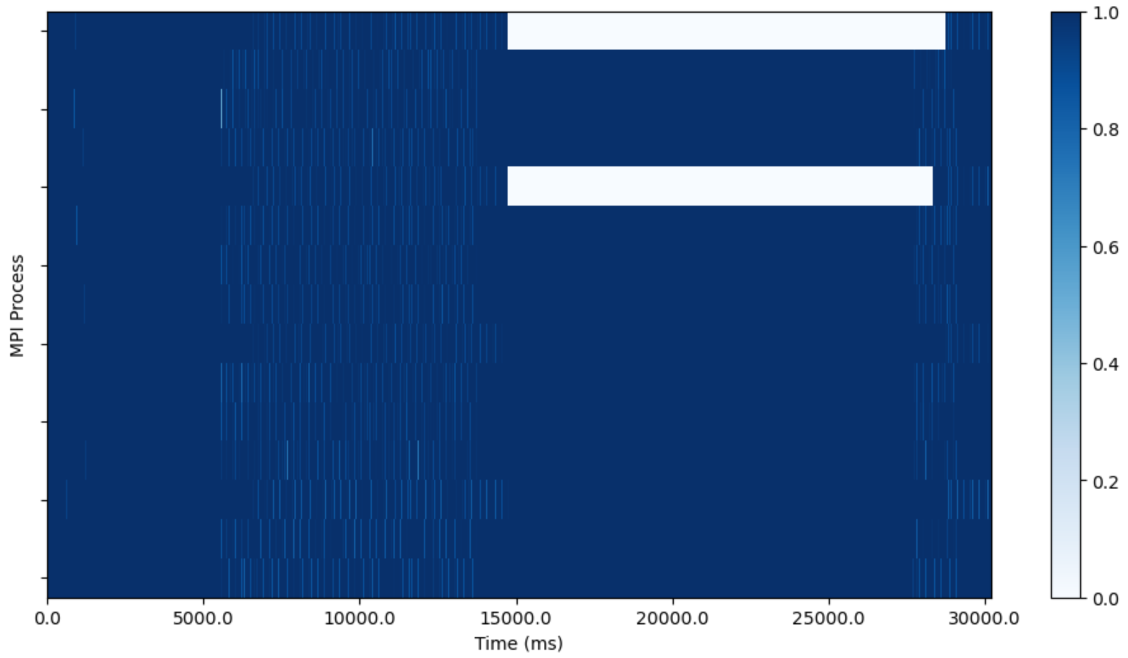
```
cd <path/to/ANT-MOC>
cd run-accl-calc
./slurm-tool.job # inject device-only computation & trace for further analysis
```

- Once the tracing the finished, run the analysis script for heatmap figures

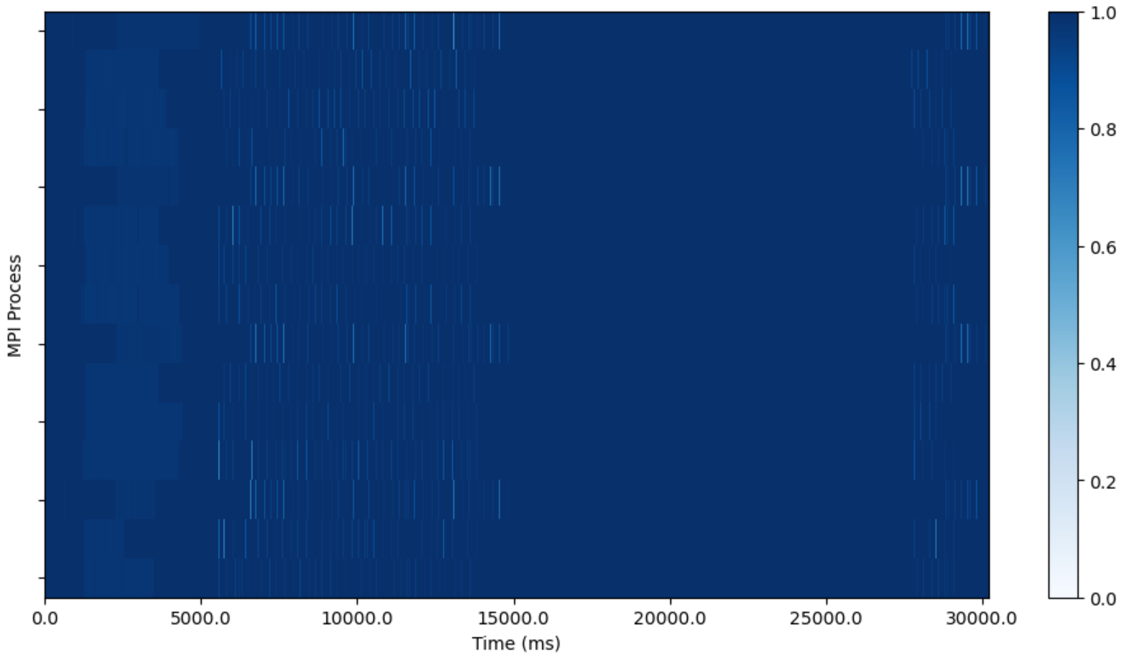
```
conda activate mstoolkit # make sure the python environment is ready
./run_analysis.sh # the previous analysis commands are wrapped into the given scripts
```

Case Study: Inject Device Computation Workloads

- Resulting files are located in the folder ***figure/***
  - ***Target variance only appears in accl_calc.png***
  - ***The lighter indicates worser performance***



accl_calc.png



accl_memcpy.png

Case Study: Inject Host-Device Data Transfer

~6 min

- Now try to inject host-device data transfer workload along with ANT-MOC execution to simulate severe PCIE performance variance.
- Injected computation-intensive kernel
  - Located in **run-accl-mem/inject-accl-mem/inject_accl_mem.cpp**
  - **Injected after 0 seconds, last for 60 seconds**
- Provide a reference job submission scripts for *inject & tracing*

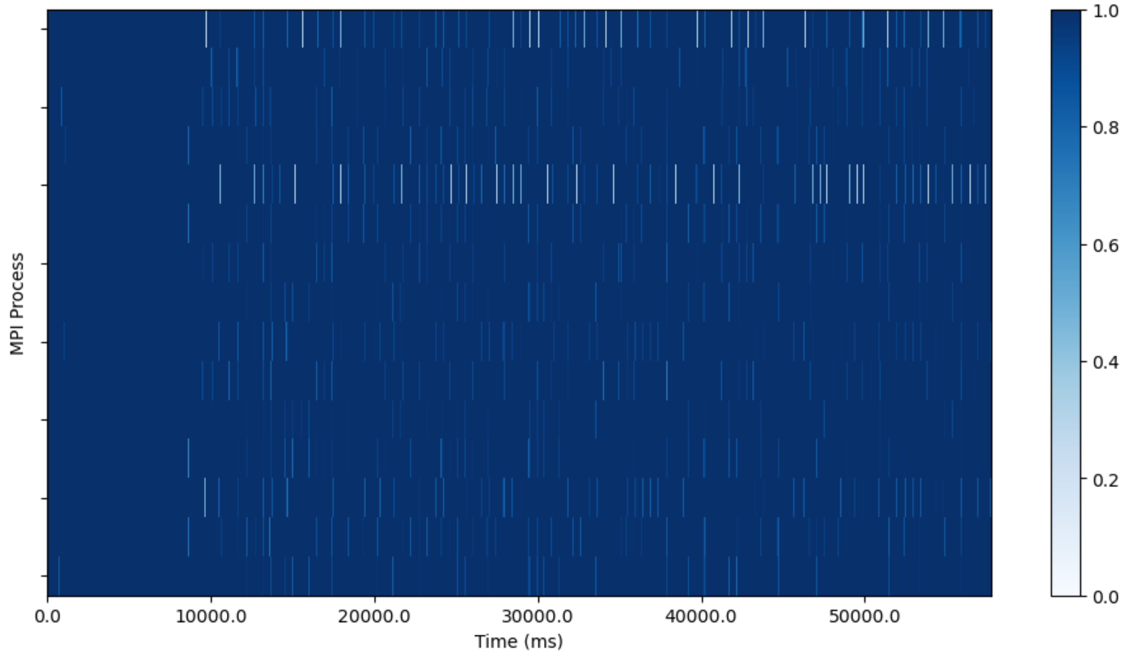
```
cd <path/to/ANT-MOC>
cd run-accl-mem
./slurm-tool.job # inject host-device data transfer & trace for further analysis
```

- Once the tracing the finished, run the analysis script for heatmap figures

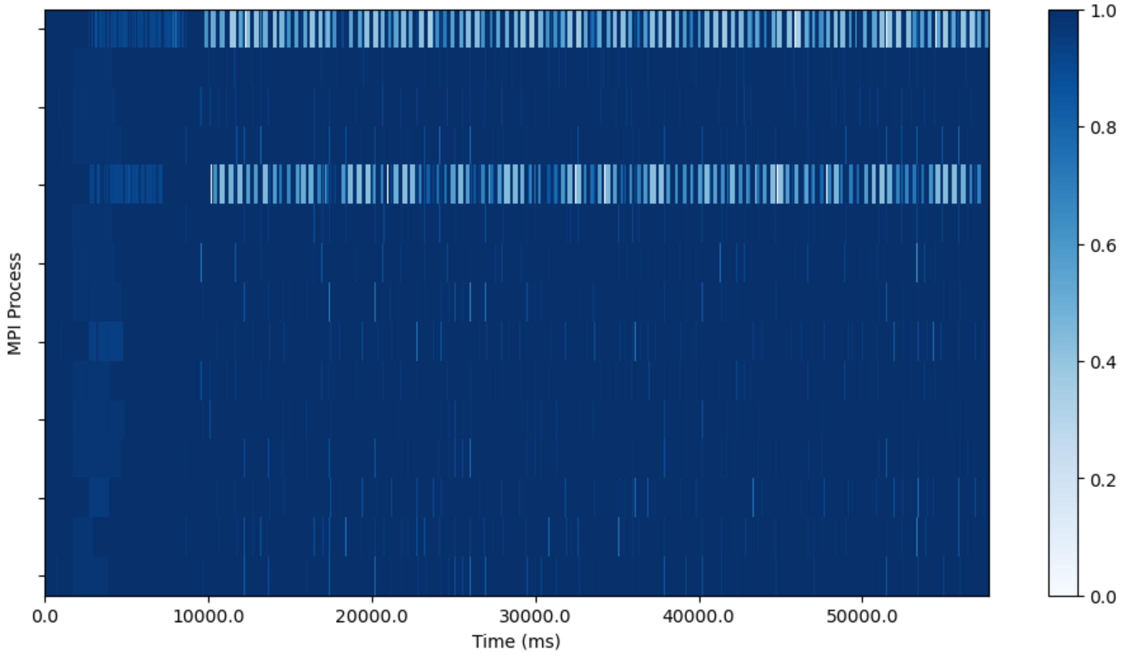
```
conda activate mstoolkit # make sure the python environment is ready
./run_analysis.sh # the previous analysis commands are wrapped into the given scripts
```

Case Study: Inject Host-Device Data Transfer

- Resulting files are located in the folder ***figure/***
  - ***Target variance only appears in accl_memcpy.png***
  - ***The lighter indicates worser performance***



accl_calc.png



accl_memcpy.png

Case Study: Inject Host Computation

~6 min

- Now try to inject host computation workload along with ANT-MOC execution to simulate severe host-side performance variance.
- Injected computation-intensive kernel
  - Located in **run-calc/inject-calc/inject.cpp**
  - **Injected after 0 seconds, last for 60 seconds**
- Provide a reference job submission scripts for *inject & tracing*

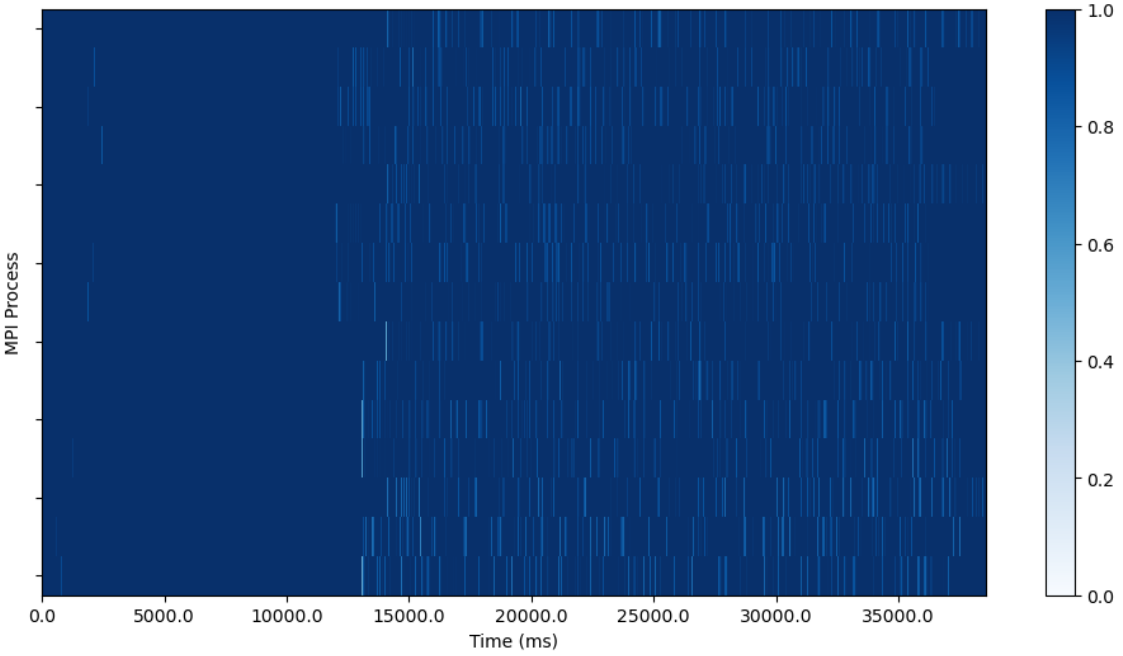
```
cd <path/to/ANT-MOC>
cd run-calc
./slurm-tool.job # inject host computation & trace for further analysis
```

- Once the tracing the finished, run the analysis script for heatmap figures

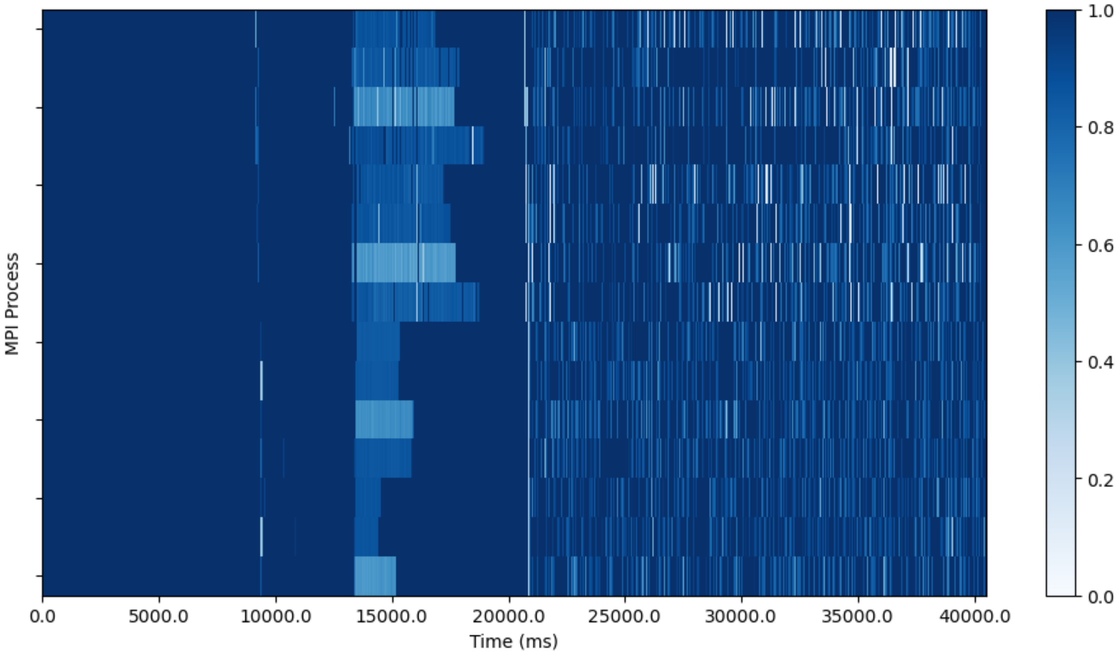
```
conda activate mstoolkit # make sure the python environment is ready
./run_analysis.sh # the previous analysis commands are wrapped into the given scripts
```

Case Study: Inject Host Computation

- Resulting files are located in the folder ***figure/***
  - ***Target variance only appears in calc.png***
  - ***The lighter indicates worser performance***



accl_calc.png



calc.png

Case Study: LAMMPS real case

~8 min

- Get the tutorial cases for LAMMPS

```
cp –r /public/home/dfcs2025/GVARP-Tutorial/examples/LAMMPS ./ && cd LAMMPS/run-1
```

- Submit LAMMPS jobs without tool (***default execution***)

```
sbatch slurm-nobind-ori.job
```

- Submit LAMMPS jobs with tool (***tracing for data collection***)

```
sbatch slurm-nobind-tool.job
```

```
#!/bin/bash
#SBATCH -J LMP-NOBIND-TOOL
#SBATCH -o log/lmp-nobind-tool-%j.log
#SBATCH -e log/lmp-nobind-tool-%j.err
#SBATCH -p test
#SBATCH --cpus-per-task=1
#SBATCH --ntasks-per-node=4
#SBATCH --gres=dcu:4
#SBATCH -n 16
export OMP_NUM_THREADS=1
export JSI_BACKTRACE_MAX_DEPTH=5
export JSI_COLLECT_PMU_EVENT=PAPI_TOT_INS,PAPI_L1_DCA

source /public/home/dfcs2025/GVARP-Tutorial/.test/LAMMPS/setup-env.sh

rm -rf measurement/measurement-nobind

/usr/bin/time -v mpirun -n 16 jsirun --accl --backtrace --pmu -o measurement/measurement-nobind -- lmp_mpi -sf gpu -pk gpu 1 -i in.balance.static.4N16DCU
```

Use DTK23 version

Tracing with ***jsirun***

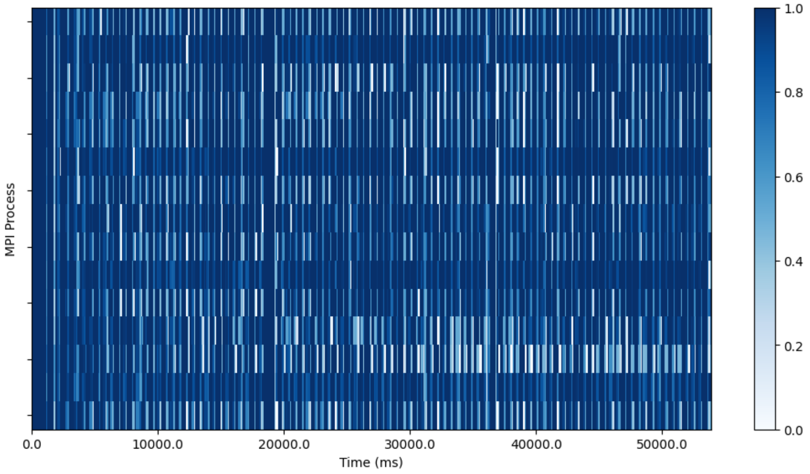
Case Study: LAMMPS real case

- Load the mstoolkit environments for variance analysis

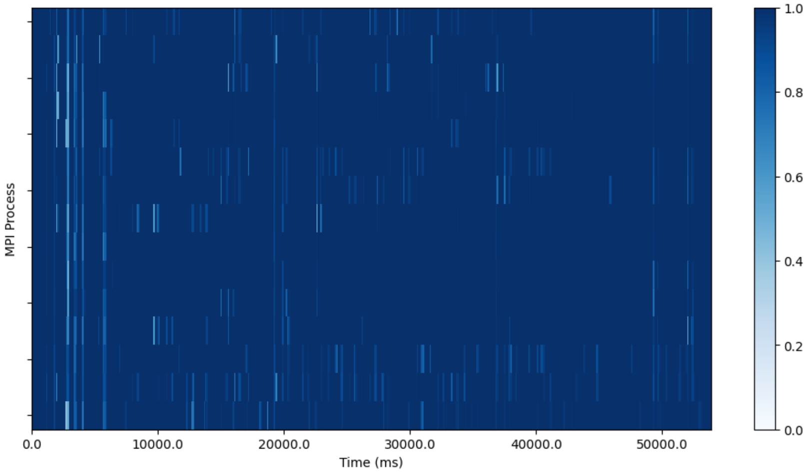
```
conda activate mstoolkit
source ../setup-env.sh
```

- Submit an analysis job and the result file is located in folder ***figure-nobind/***

```
srun -n 1 --cpus-per-task 2 -p sep2 --exclusive --mem=100GB variance_analysis -i
measurement/measurement-nobind -o variance-nobind -f
HEATMAP_PY=$MSTOOLKIT_PATH/scripts/analysis/variance/heatmap.py
python $HEATMAP_PY --input variance-nobind --output figure-nobind
```



accl_calc.png



accl_memcpy.png

Case Study: LAMMPS real case

- Check for log: *log/lmp-nobind-tool-<slurm job id>.log*

```
LAMMPS (2 Aug 2023)
  using 1 OpenMP thread(s) per MPI task

...

--------------------------------------------------------------------------
- Using acceleration for lj/cut:
-  with 4 proc(s) per device.
-  with 1 thread(s) per proc.
-  Horizontal vector operations: ENABLED
-  Shared memory system: No
--------------------------------------------------------------------------
Device 0: Device 66a1, 64 CUs, 16/16 GB, 1.7 GHZ (Mixed Precision)
--------------------------------------------------------------------------

Initializing Device and compiling on process 0...Done.
Initializing Device 0 on core 0...Done.
Initializing Device 0 on core 1...Done.
Initializing Device 0 on core 2...Done.
Initializing Device 0 on core 3...Done.
...
```

Bind to the same GPU!

Case Study: LAMMPS real case

- Bind GPU with its MPI rank via numactl && HIP_VISIBLE_DEVICES

```
#!/bin/bash
APPCMD="$*"
lrank=$(expr $OMPI_COMM_WORLD_LOCAL_RANK % 4)
case ${lrank} in
[0])
 export HIP_VISIBLE_DEVICES=0
 export UCX_NET_DEVICES=mlx5_0:1
 export UCX_IB_PCI_BW=mlx5_0:50Gbs
 numactl --cpunodebind=0 --membind=0 ${APPCMD}
 ;;
[1])
 export HIP_VISIBLE_DEVICES=1
 export UCX_NET_DEVICES=mlx5_1:1
 export UCX_IB_PCI_BW=mlx5_1:50Gbs
 numactl --cpunodebind=1 --membind=1 ${APPCMD}
 ;;
[2])
 export HIP_VISIBLE_DEVICES=2
 export UCX_NET_DEVICES=mlx5_2:1
 export UCX_IB_PCI_BW=mlx5_2:50Gbs
 numactl --cpunodebind=2 --membind=2 ${APPCMD}
 ;;
[3])
 export HIP_VISIBLE_DEVICES=3
 export UCX_NET_DEVICES=mlx5_3:1
 export UCX_IB_PCI_BW=mlx5_3:50Gbs
 numactl --cpunodebind=3 --membind=3 ${APPCMD}
 ;;
esac
```

bind.sh

```
#!/bin/bash
#SBATCH -J LMP-BIND-TOOL
#SBATCH -o log/lmp-bind-ori-%j.log
#SBATCH -e log/lmp-bind-ori-%j.err
#SBATCH -p test
#SBATCH --cpus-per-task=1
#SBATCH --ntasks-per-node=4
#SBATCH --gres=dcu:4
#SBATCH -n 16
#
export OMP_NUM_THREADS=1
export JSI_BACKTRACE_MAX_DEPTH=5
export
JSI_COLLECT_PMU_EVENT=PAPI_TOT_INS,PAPI_L1_DCA

#cd /public/home/buaa_hipo/CLUSTER25-
Tutorial/app/LAMMPS/run-1

module load apps/lammps-DCU/2Aug2023/hpcx-2.7.4-
dtk23.10

/usr/bin/time -v mpirun -n 16 ./bind.sh lmp_mpi -
sf gpu -pk gpu 1 -i in.balance.static.4N16DCU
```

Bind to the different GPU

slurm-bind-ori.job

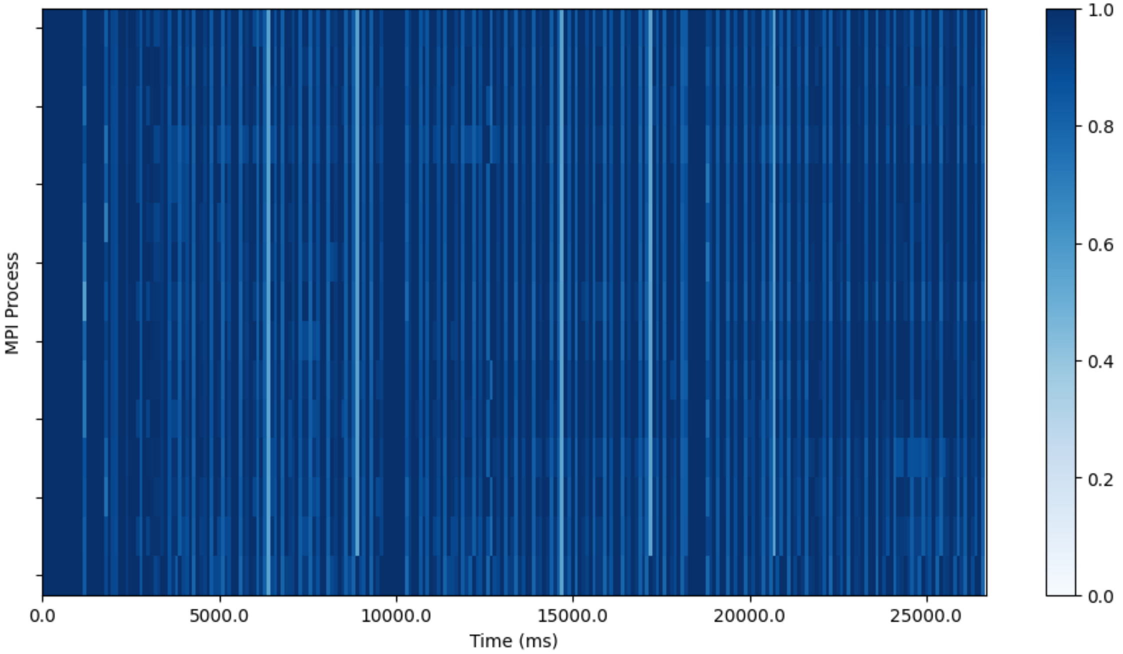
Case Study: LAMMPS real case

- Submit the optimized configuration && tracing for validation

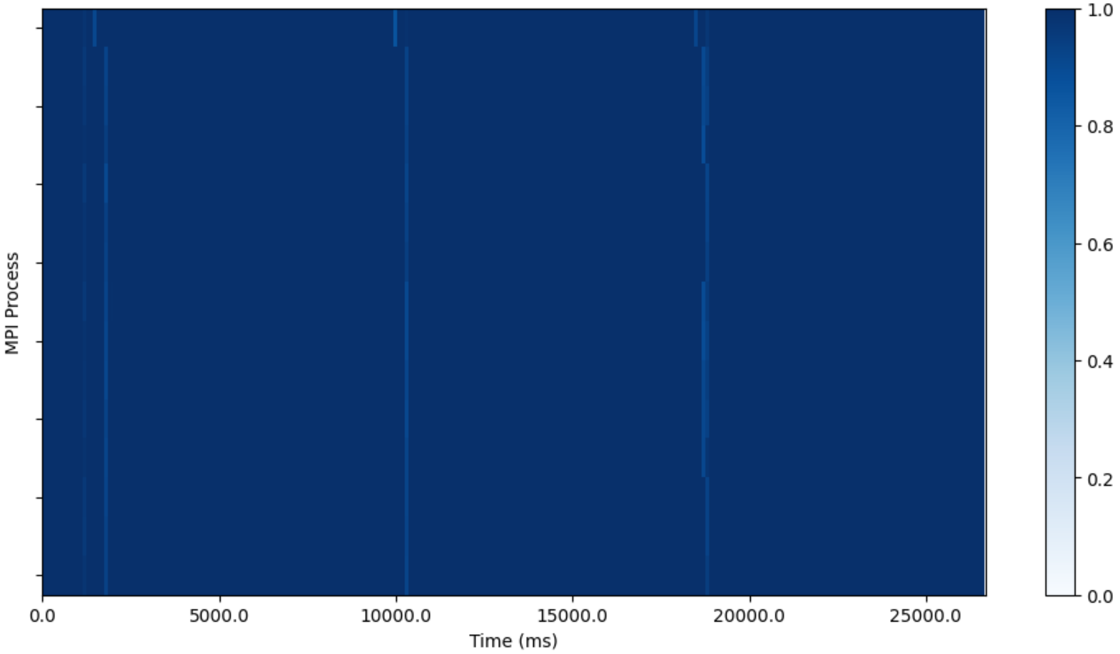
```
sbatch slurm-bind-ori.job
sbatch slurm-bind-tool.job
```

- Run analysis when the job is finished and the tracing data is ready

```
srun -n 1 -p test --mem=100GB variance_analysis -i measurement/measurement-bind -o variance-bind -f
python $ HEATMAP_PY --input variance-bind --output figure-bind
```



accl_calc.png



accl_memcpy.png

Thanks! Q&A

Contact: youxin2015@buaa.edu.cn