



Thicket: Seeing the performance experiment forest for the individual run trees

RADIUSS Tutorial Series 2024

8 August 2024



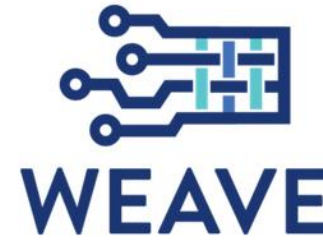
Stephanie Brink, Dewi Yokelson, Olga Pearce





Welcome to the HPCIC 2024 HPC Tutorials!

Go to:
<https://hpcic.llnl.gov/tutorials/2024-hpc-tutorials> to learn more about
our other tutorials and
documentation!



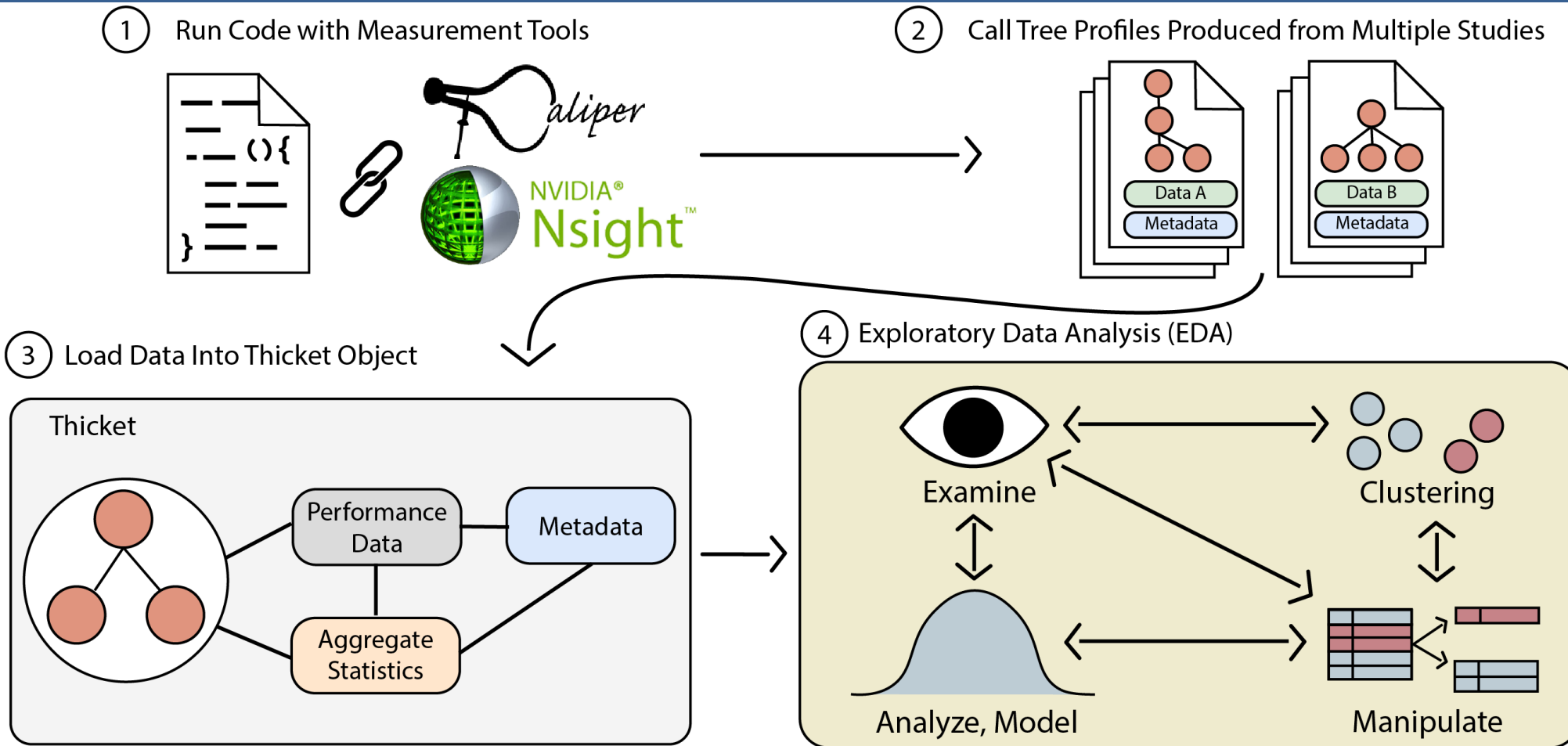
Tutorial Materials: <https://github.com/llnl/thicket-tutorial>

- We will start with a presentation, slides available here:
https://thicket.readthedocs.io/en/latest/tutorial_materials.html
- The tutorial materials include example Jupyter notebooks, Thicket install 2024.1.0, and RAJA Performance Suite datasets in a docker container environment with all dependencies
- We'll use this material in the hands-on portion of the tutorial


Join our mailing list! <https://bit.ly/caliper-thicket-users>



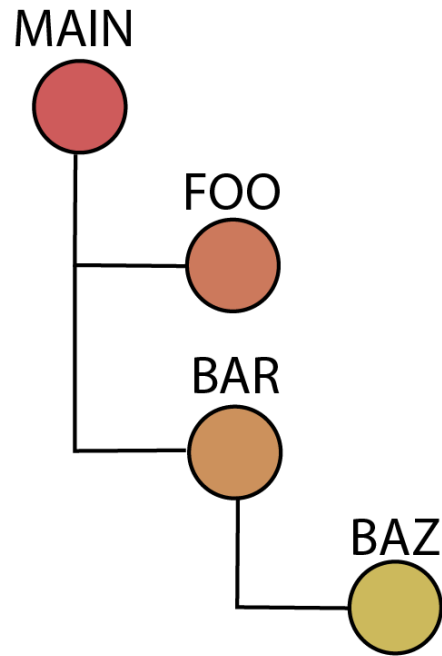
Our big picture solution for analyzing and visualizing performance data from different sources and type



What do profiling tools collect per run?

e.g., 

1) Call Tree



2) Performance data

Node	Cache Misses
MAIN	
FOO	
BAR	
BAZ	

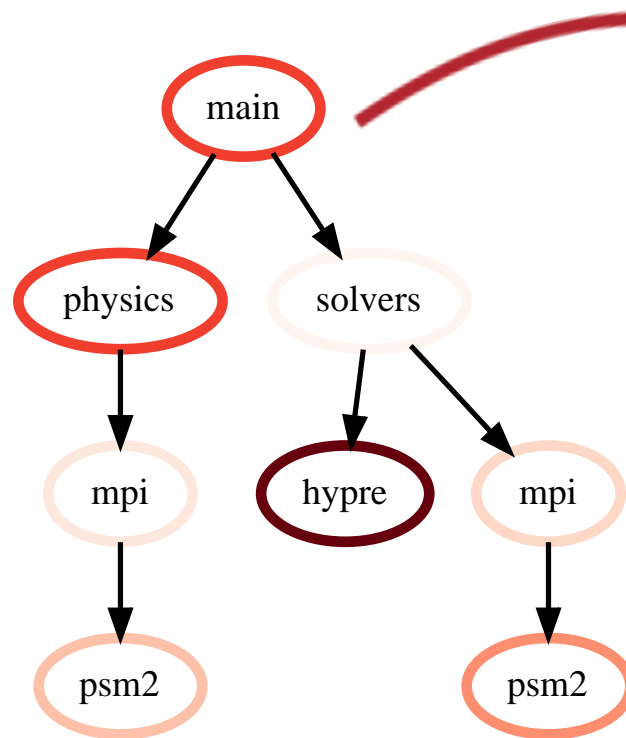
- Time, FLOPS
- Cache misses
- Memory accesses

3) Metadata per run

User	Platform

- Batch submission (user, launch date)
- Hardware info (platform)
- Build info (compiler versions/flags)
- Runtime info (problem parameters, number of MPI ranks used)

Thicket builds upon Hatchet's *GraphFrame*: a Graph and a Dataframe



Graph: Stores relationships between parents and children

	name	nid	node	time	time (inc)
node					
main	main	0	main	40.0	200.0
physics	physics	1	physics	40.0	60.0
mpi	mpi	2	mpi	5.0	20.0
psm2	psm2	3	psm2	15.0	15.0
solvers	solvers	4	solvers	0.0	100.0
hypre	hypre	5	hypre	65.0	65.0
mpi	mpi	6	mpi	10.0	35.0
psm2	psm2	7	psm2	25.0	25.0

Pandas Dataframe: 2D table storing numerical data associated with each node (may be unique per rank, per thread)





Visualizing Hatchet's GraphFrame components

```
>>> print(gf.tree()) # print graph
>>> print(gf.dataframe) # print dataframe
```

```
0.000 foo
├─ 6.000 bar
│   └─ 5.000 baz
├─ 0.000 qux
│   └─ 5.000 quux
│       ├── 10.000 corge
│       ├── 15.000 garply
│       └─ 1.000 grault
└─ 15.000 waldo
    ├── 3.000 fred
    │   └─ 5.000 plugh
    └─ 15.000 garply
```

Legend (Metric: time)

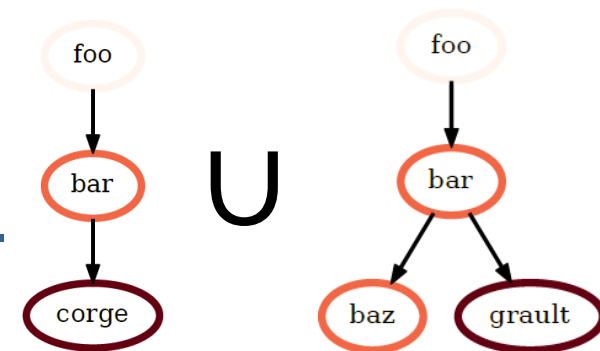
- 13.50 - 15.00
- 10.50 - 13.50
- 7.50 - 10.50
- 4.50 - 7.50
- 1.50 - 4.50
- 0.00 - 1.50

node			
{'name': 'foo'}	foo	0.0	130.0
{'name': 'bar'}	bar	5.0	20.0
{'name': 'baz'}	baz	5.0	5.0
{'name': 'grault'}	grault	10.0	10.0
{'name': 'quux'}	quux	0.0	60.0
{'name': 'quux'}	quux	5.0	60.0
{'name': 'corge'}	corge	10.0	55.0
{'name': 'bar'}	bar	5.0	20.0
{'name': 'baz'}	baz	5.0	5.0
{'name': 'grault'}	grault	10.0	10.0
{'name': 'garply'}	garply	15.0	15.0
{'name': 'grault'}	grault	10.0	10.0

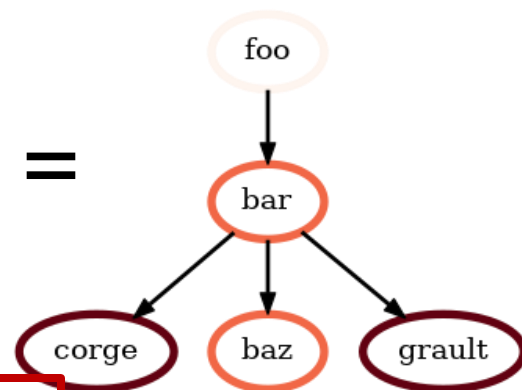
name User code ◀ Only in left graph ▶ Only in right graph

Compare GraphFrames using division (or add, subtract, multiply)

```
>>> gf3 = gf1 / gf2 # divide graphframes
```



*First, unify two trees since structure is different



gf3		gf1		gf2
0.000 foo		0.000 foo		0.000 foo
├ 2.000 bar		├ 6.000 bar		├ 3.000 bar
├ ─ 5.000 baz		├ ─ 5.000 baz		├ ─ 1.000 baz
├ ─ inf qux	=	├ ─ 3.000 qux	/	├ ─ 0.000 qux
├ ─ 4.000 quux		├ ─ 2.000 quux		├ ─ 0.500 quux
├ ─ 2.000 corge		├ ─ 8.000 corge		├ ─ 4.000 corge
├ ─ nan garply				├ ─ 15.000 garply
├ ─ nan grault				├ ─ 0.250 grault

```
>>> gf3 = gf1 + gf2 # add graphframes
>>> gf3 = gf1 - gf2 # subtract graphframes
>>> gf3 = gf1 * gf2 # multiply graphframes
```

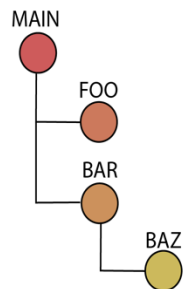
Use Thicket to *compose* performance profiles in Python



P1

Metadata

User	Platform



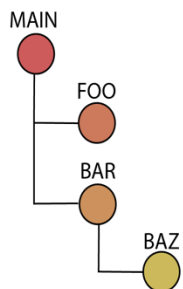
Performance metrics

Node	Cache Misses
MAIN	24
FOO	
BAR	
BAZ	

P2

Metadata

User	Platform



Performance metrics

Node	Cache Misses
MAIN	16
FOO	
BAR	
BAZ	



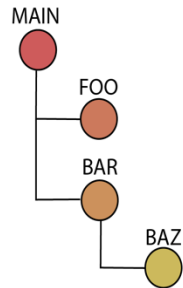
Use Thicket to *compose* performance profiles in Python



P1

Metadata

User	Platform



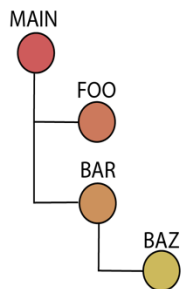
Performance metrics

Node	Cache Misses
MAIN	24
FOO	
BAR	
BAZ	

P2

Metadata

User	Platform



Performance metrics

Node	Cache Misses
MAIN	16
FOO	
BAR	
BAZ	

- 1 Compose functions w/matching call trees

Performance metrics

Node	Profile	Cache Misses
MAIN	P1	24
	P2	16
FOO	P1	
	P2	
BAR	P1	
	P2	
BAZ	P1	
	P2	

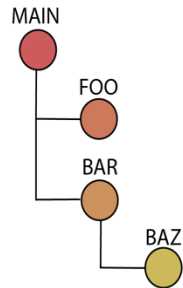
Use Thicket to *compose* performance profiles in Python



P1

Metadata

User	Platform



Performance metrics

Node	Cache Misses
MAIN	24
FOO	
BAR	
BAZ	

2

1

Metadata

Profile	User	Platform
P1		
P2		

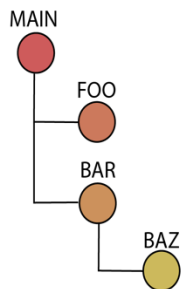
Performance metrics

Node	Profile	Cache Misses
MAIN	P1	24
	P2	16
FOO	P1	
	P2	
BAR	P1	
	P2	
BAZ	P1	
	P2	

P2

Metadata

User	Platform



Performance metrics

Node	Cache Misses
MAIN	16
FOO	
BAR	
BAZ	

- 1 Compose functions w/matching call trees
- 2 Compose metadata with all fields

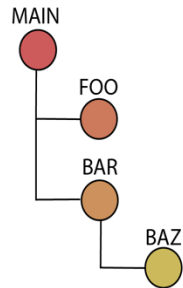
Use Thicket to *compose* performance profiles in Python



P1

Metadata

User	Platform



Performance metrics

Node	Cache Misses
MAIN	24
FOO	
BAR	
BAZ	

2

1

Metadata

Profile	User	Platform
P1		
P2		

Performance metrics

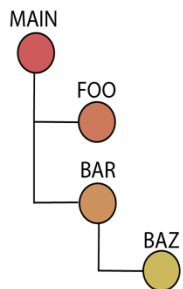
Node	Profile	Cache Misses
MAIN	P1	24
	P2	16
FOO	P1	
	P2	
BAR	P1	
	P2	
BAZ	P1	
	P2	

- 1 Compose functions w/matching call trees
- 2 Compose metadata with all fields
- 3 Aggregate statistics (order reduction)

P2

Metadata

User	Platform



Performance metrics

Node	Cache Misses
MAIN	16
FOO	
BAR	
BAZ	

3

Node	Avg. Cache Misses
MAIN	20
FOO	
BAR	
BAZ	

Thicket components are *interconnected*



Metadata

Profile	User	Platform
P1	Jon	lassen
P2	Bob	lassen

Filtered Metadata

Profile	User	Platform
P2	Bob	lassen

Performance metrics

Node	Profile	Cache Misses
MAIN	P1	
	P2	
FOO	P1	
	P2	
BAR	P1	
	P2	
BAZ	P1	
	P2	

Filter on metadata:
platform=="lassen" &&
user=="Bob"

Filtered Performance metrics

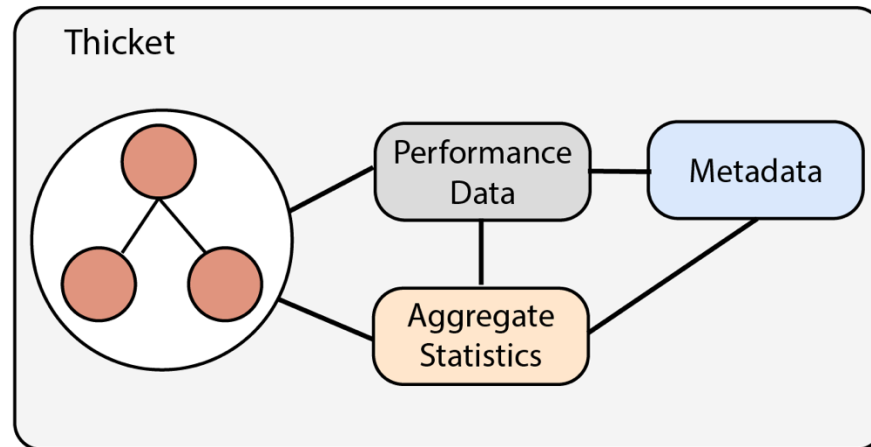
Node	Profile	Cache Misses
MAIN	P2	
FOO	P2	
BAR	P2	
BAZ	P2	

Metadata fields useful for understanding
and manipulating thicket object!

Thicket enables exploratory data analysis of multi-run data



3 Load Data Into Thicket Object

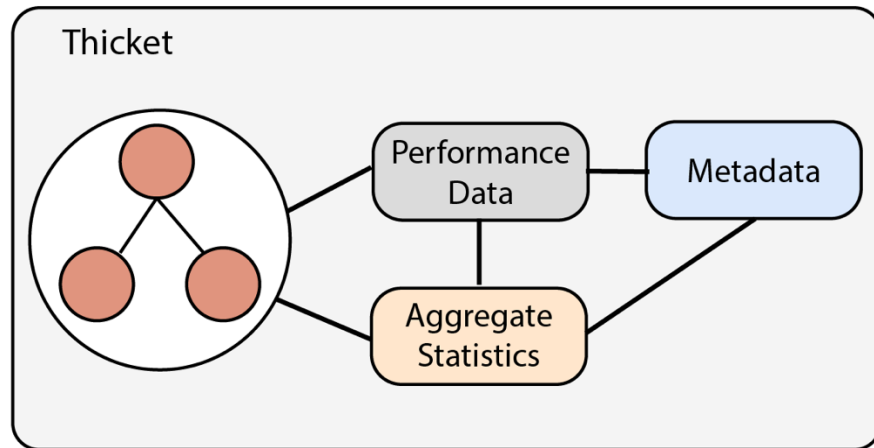


- Compose data from diff. sources and types
 - Different scaling (e.g., strong, weak)
 - Different application parameters
 - Different compilers and optimization levels
 - Different hardware types (e.g., CPUs, GPUs)
 - Different performance tools

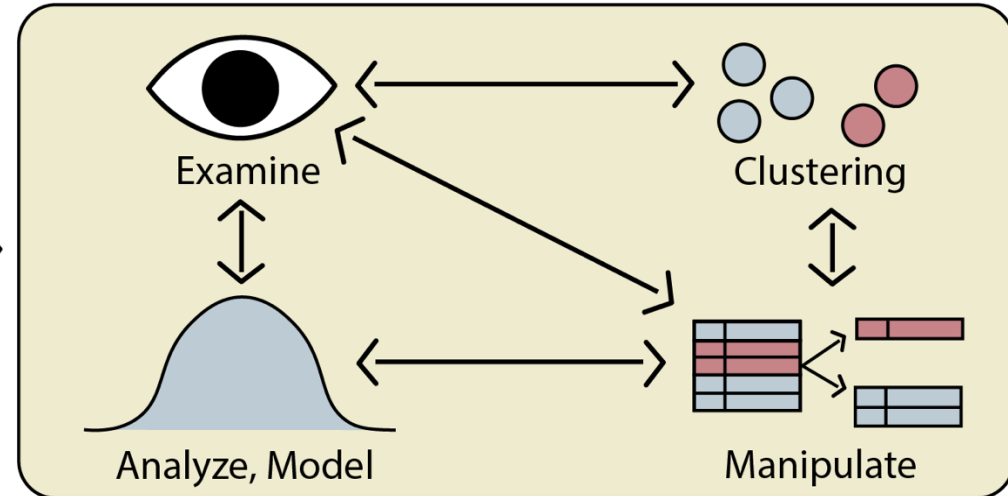
Thicket enables exploratory data analysis of multi-run data



③ Load Data Into Thicket Object



④ Exploratory Data Analysis (EDA)



- Compose data from diff. sources and types
 - Different scaling (e.g., strong, weak)
 - Different application parameters
 - Different compilers and optimization levels
 - Different hardware types (e.g., CPUs, GPUs)
 - Different performance tools

- Perform analysis on the thickets of runs
 - Manipulate the set of data
 - Visualize the dataset
 - Perform analysis on the data
 - Model data
 - Leverage third-party tools in the Python ecosystem

RAJA Case Study 1: RAJA Performance Suite

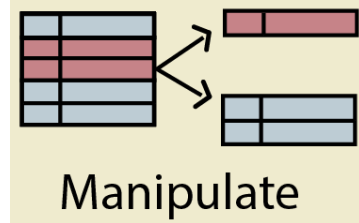


- Open-source suite of loop-based kernels commonly found in HPC applications showcasing performance of different programming models on different hardware
- 560 runs/profiles:
 - 2 clusters (CPU, CPU+GPU)
 - 4 problem sizes
 - 3 compilers, 4 optimizations
 - 3 programming models (sequential, OpenMP, CUDA)
 - 3 performance tools (Caliper, PAPI, Nsight Compute)

	cluster	systype	build	problem size	compiler	compiler optimizations	omp num threads	cuda compiler	block sizes	RAJA variant	#profiles
0	quartz		toss_3_x86_64_ib	[1M, 2M, 4M, 8M]	clang++-9.0.0	[-O0, -O1, -O2, -O3]	1	N/A	N/A	Sequential	160
1	quartz		toss_3_x86_64_ib	[1M, 2M, 4M, 8M]	g++-8.3.1	[-O0, -O1, -O2, -O3]	1	N/A	N/A	Sequential	160
2	quartz		toss_3_x86_64_ib	[1M, 2M, 4M, 8M]	clang++-9.0.0	-O0	72	N/A	N/A	OpenMP	40
3	quartz		toss_3_x86_64_ib	[1M, 2M, 4M, 8M]	g++-8.3.1	-O0	72	N/A	N/A	OpenMP	40
4	lassen	blueos_3_ppc64le_ib_p9		[1M, 2M, 4M, 8M]	xlc++_r-16.1.1.12	-O0	1	nvcc-11.2.152	[128, 256, 512, 1024]	CUDA	160



Use Thicket to *compose* multi-platform, multi-tool data



Thicket object composed of 2 profiles run on CPU

	node	problem_size	time (exc)	Reps	Retiring	Backend bound
Apps_NODAL_ACCUMULATION_3D		1M	0.204583	100	0.144928	0.783786
		4M	0.795511	100	0.139002	0.788017
Apps_VOL3D		1M	0.067061	100	0.402238	0.510525
		4M	0.241508	100	0.400775	0.515976

Thicket object composed of 2 profiles run on GPU

	node	problem_size	time (gpu)	gpu_compute_memory_throughput	gpu_dram_throughput	sm_throughput
Apps_NODAL_ACCUMULATION_3D		1M	0.007478	70.689752	46.724767	7.330745
		4M	0.026951	74.275834	51.257993	7.688628
Apps_VOL3D		1M	0.006028	81.012826	67.751194	35.676942
		4M	0.021422	91.929933	70.122011	35.386470

CPU

GPU

	node	problem_size	time (exc)	Reps	Retiring	Backend bound	time (gpu)	gpu_compute_memory_throughput	gpu_dram_throughput	sm_throughput
Apps_NODAL_ACCUMULATION_3D		1M	0.204583	100	0.144928	0.783786	0.007478	70.689752	46.724767	7.330745
		4M	0.795511	100	0.139002	0.788017	0.026951	74.275834	51.257993	7.688628
Apps_VOL3D		1M	0.067061	100	0.402238	0.510525	0.006028	81.012826	67.751194	35.676942
		4M	0.241508	100	0.400775	0.515976	0.021422	91.929933	70.122011	35.386470

Dataset: 4 types of profiles side-by-side to compare CPU to GPU performance

- 1 Basic CPU metrics from Caliper
- 2 Top-down metrics from Caliper/PAPI
- 3 GPU runtime from Caliper
- 4 GPU metrics from Nsight Compute

Examples of analysis:

- Compute CPU/GPU speedup
- Correlate memory and compute usage on the CPU vs. GPU

1

2

3

4

Derived

CPU

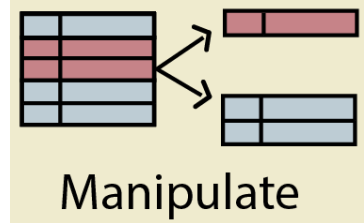
CPU top-down

GPU

GPU Nsight Compute

Node	Problem size	CPU					GPU					speedup
		time (exc)	Bytes/Rep	Flops/Rep	Retiring	Backend bound	time (gpu)	gpu__compute_memory_throughput	gpu__dram_throughput	sm__throughput	sm__warps_active	
Apps_VOL3D	8M	0.498815	282109496	632421288	0.377843	0.540604	0.040761	93.742058	72.140428	36.206767	54.459589	12.237556
s_HYDRO_1D	8M	2.077556	201326600	41943040	0.032965	0.909545	0.242928	92.944968	92.944968	6.595714	95.266148	8.552147

Manipulate: Filter using call path query



```
0.001 Base_CUDA
├── 0.000 Algorithm
│   ├── 0.000 Algorithm_MEMCPY
│   │   ├── 0.002 Algorithm_MEMCPY.block_128
│   │   ├── 0.009 Algorithm_MEMCPY.block_256
│   │   └── 0.006 Algorithm_MEMCPY.library
│   ├── 0.000 Algorithm_MEMSET
│   │   ├── 0.001 Algorithm_MEMSET.block_128
│   │   ├── 0.004 Algorithm_MEMSET.block_256
│   │   └── 0.003 Algorithm_MEMSET.library
│   ├── 0.000 Algorithm_REDUCE_SUM
│   │   ├── 0.003 Algorithm_REDUCE_SUM.block_128
│   │   ├── 0.004 Algorithm_REDUCE_SUM.block_256
│   │   └── 0.002 Algorithm_REDUCE_SUM.cub
│   └── 0.000 Algorithm_SCAN
│       └── 0.006 Algorithm_SCAN.default
```

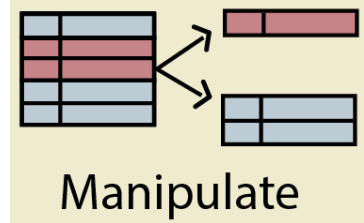
Input call tree

Filter on call path:
(1) Node named
"Base_CUDA"

```
0.001 Base_CUDA
```

Output call tree

Manipulate: Filter using call path query



```
0.001 Base_CUDA
└─ 0.000 Algorithm
   ├── 0.000 Algorithm_MEMCPY
   │   ├── 0.002 Algorithm_MEMCPY.block_128
   │   ├── 0.009 Algorithm_MEMCPY.block_256
   │   └── 0.006 Algorithm_MEMCPY.library
   ├── 0.000 Algorithm_MEMSET
   │   ├── 0.001 Algorithm_MEMSET.block_128
   │   ├── 0.004 Algorithm_MEMSET.block_256
   │   └── 0.003 Algorithm_MEMSET.library
   ├── 0.000 Algorithm_REDUCE_SUM
   │   ├── 0.003 Algorithm_REDUCE_SUM.block_128
   │   ├── 0.004 Algorithm_REDUCE_SUM.block_256
   │   └── 0.002 Algorithm_REDUCE_SUM.cub
   ├── 0.000 Algorithm_SCAN
   │   └── 0.006 Algorithm_SCAN.default
```

Input call tree

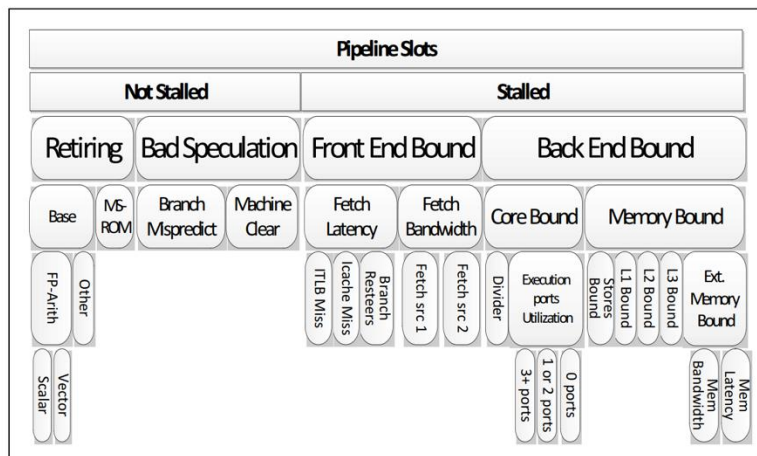
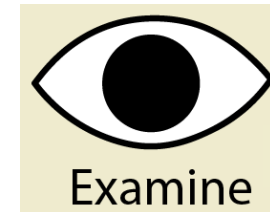
Filter on call path:

- (1) Node named "Base_CUDA"
- (2) Node with "block_128" in name (and any nodes in between)

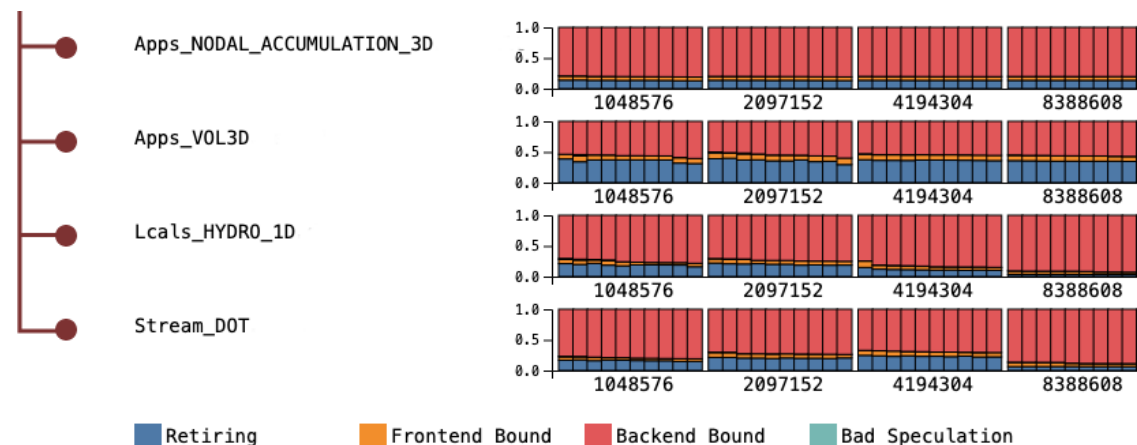
```
0.001 Base_CUDA
└─ 0.000 Algorithm
   ├── 0.000 Algorithm_MEMCPY
   │   └── 0.002 Algorithm_MEMCPY.block_128
   ├── 0.000 Algorithm_MEMSET
   │   └── 0.001 Algorithm_MEMSET.block_128
   ├── 0.000 Algorithm_REDUCE_SUM
   │   └── 0.003 Algorithm_REDUCE_SUM.block_128
```

Output call tree

Visualize: Intel CPU top-down analysis



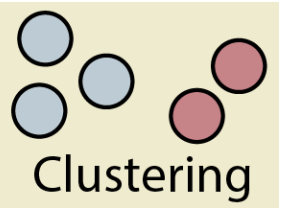
- *Top-down analysis* uses HW counters in a hierarchy to identify bottlenecks*
- Use Caliper's top-down module to derive top-down metrics for call-tree regions



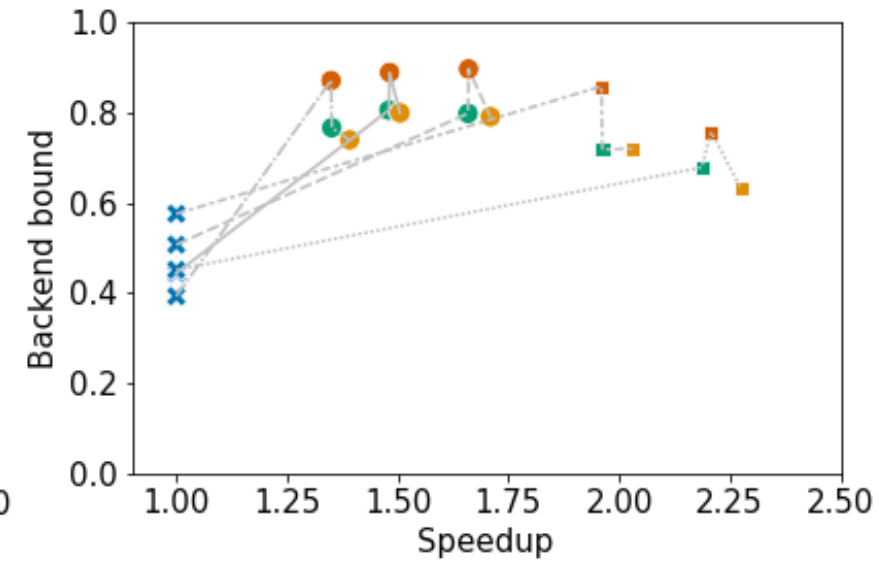
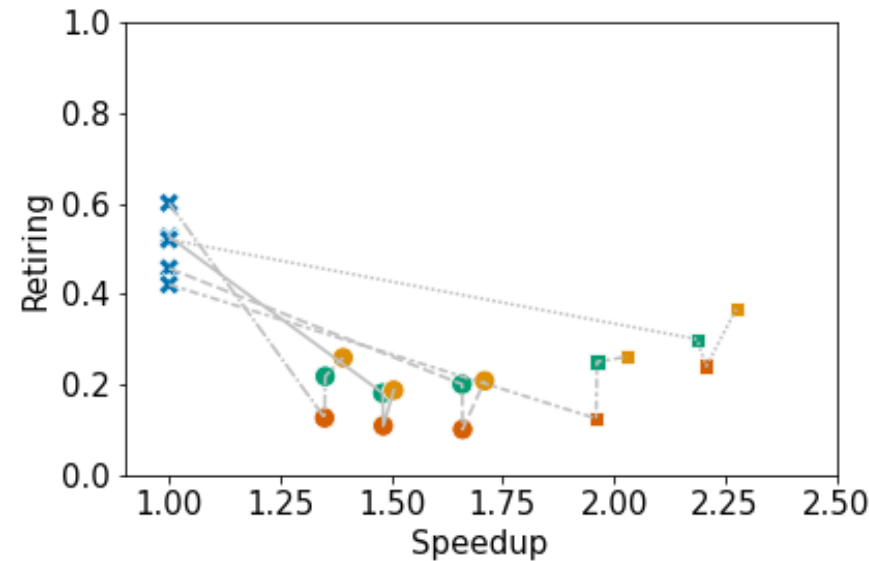
- Thicket's *tree+table* visualization shows top-down metrics as stacked bar charts, each bar is a profile
 - Apps_VOL3D has the highest retiring rates
 - Lcals_HYDRO and Stream_DOT become more backend bound as problem size grows

* Yasin, A.: A Top-Down Method for Performance Analysis and Counters Architecture. In: 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 35–44. IEEE, CA, USA (Mar 2014).

Use third-party Python libraries, e.g., Scikit-learn clustering



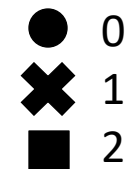
1. Select data of interest
 - Filter 8M problem size
 - Use query language to extract all implementations of the Stream kernel
2. (optional) Normalize data
3. Apply scikit-learn clustering to top-down analysis metrics of runs with different compiler optimization levels



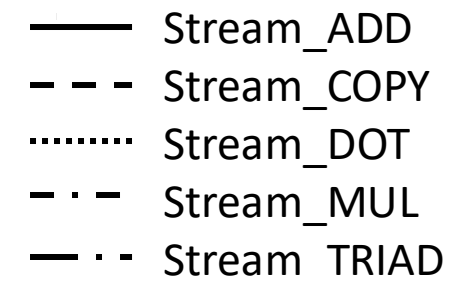
Optimization Level

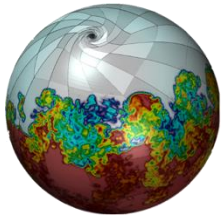


K-Means Clusters



Kernels





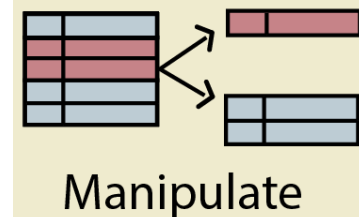
Case Study 2: MARBL multi-physics code



- MARBL is a next-generation multi-physics code developed at LLNL
- 60 runs/profiles:
 - 2 clusters (rztopaz, AWS ParallelCluster)
 - 2 MPI libraries (impi, openmpi)
 - 6 node/rank counts
 - 5 repeat runs per config

	cluster	ccompiler	mpi	version	numhosts	mpi.world.size	#profiles
0	ip----	/usr/tce/packages/clang/clang-9.0.0	impi	v1.1.0-203-gcb0efb3	[1, 2, 4, 8, 16, 32]	[36, 72, 144, 288, 576, 1152]	30
1	rztopaz	/usr/tce/packages/clang/clang-9.0.0	openmpi	v1.1.0-201-g891eaf1	[1, 2, 4, 8, 16, 32]	[36, 72, 144, 288, 576, 1152]	30

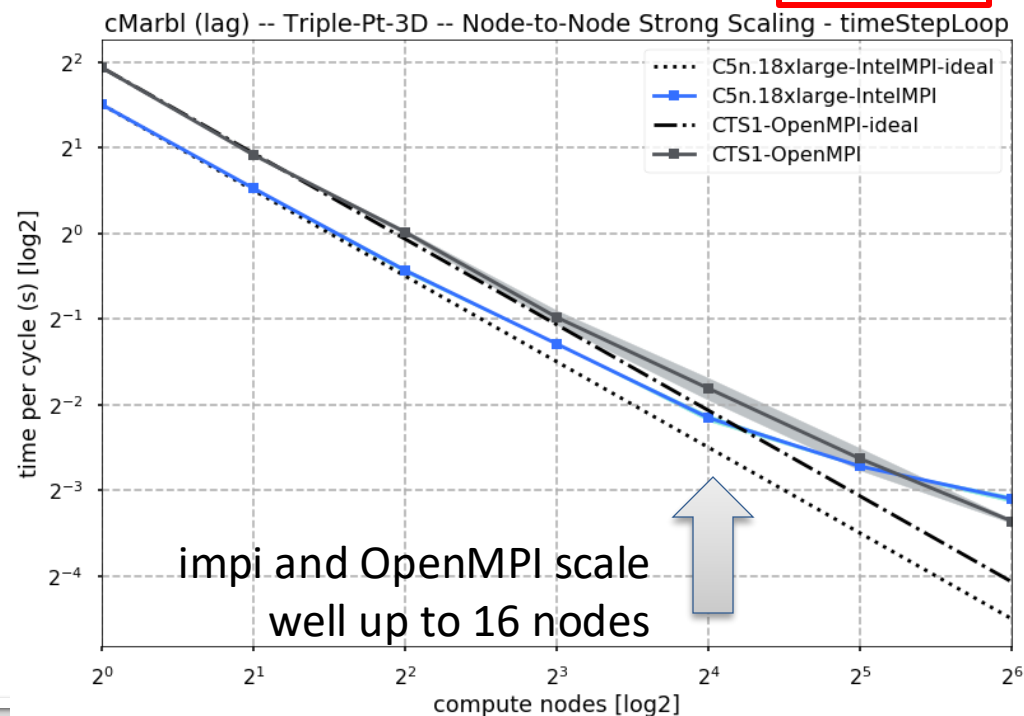
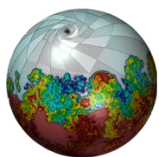
Manipulate: Compute noise and scaling



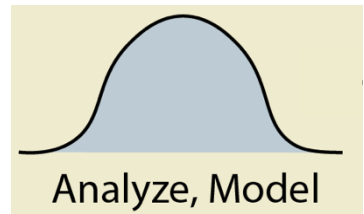
		Total time	name	mpi.world.size
node	profile			
{ 'name': 'main', 'type': 'function' }	-8554409769265002864	58036.664552	main	144
	-7335101512240609798	55318.808836	main	36
	-6029692086108825020	156984.246813	main	2304
	-5606382734792961361	64122.371533	main	288
	-4058809097109060732	155040.998627	main	2304
	-3193575964635936033	71010.504038	main	576
	-2978339073585311581	55910.708449	main	72
	-2939704488254773514	157934.204076	main	2304
	-2771797711381234985	56893.512948	main	144
	-2638513839856695106	97432.260966	main	1152

			Total time	name	mpi.world.size
	node	profile			
{'name': 'main', 'type': 'function'}	-7335101512240609798	55318.808836	main	36	
	-843517585394879415	55110.656885	main	36	
	7720382918482619866	55155.581578	main	36	
	8293335926964337960	55139.134916	main	36	
	8335957980556391465	55013.682102	main	36	

1. Use `groupby(mpi.world.size)` to generate unique subsets of data which are repeated runs; compute noise
2. Compose runs on different platforms and at different scales
3. Generate strong scaling plot with matplotlib
 - Deviation shown in shaded region, dots are average of 5 runs

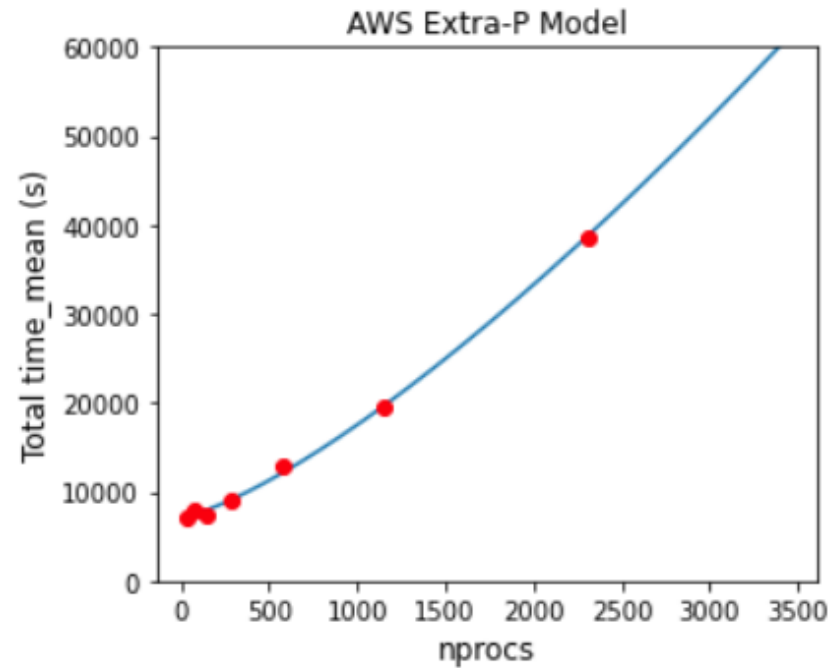


Model: Use third-party Python library, Extra-P

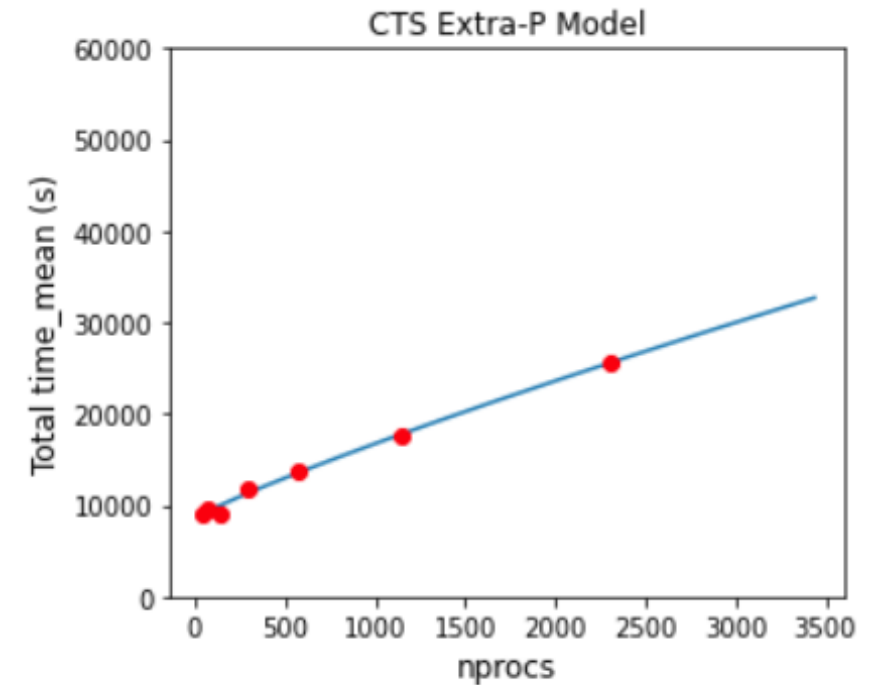


Extra-P derives an analytical performance model from an ensemble of profiles covering one or more modeling parameters
<http://github.com/extra-p/extrap>

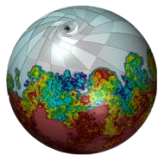
- Select functions of interest
- Call Extra-P to model scaling on different hardware types



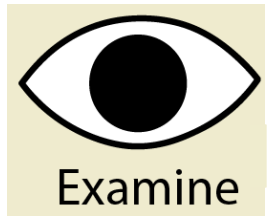
— $7182.4031994090665 + 1.03574959802164 * p^{(4/3)}$
● M_solver->Mult



— $8670.320353773704 + 4.557021276855119 * p^{(3/4)} * \log_2(p)^{(1)}$
● M_solver->Mult



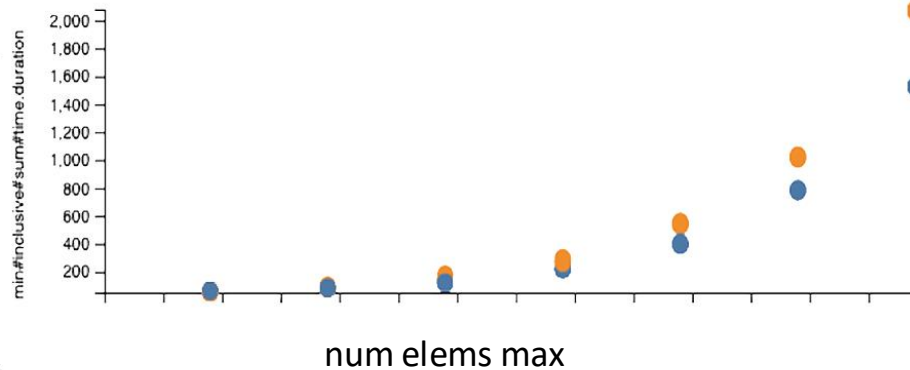
Visualize metadata with parallel coordinates plot



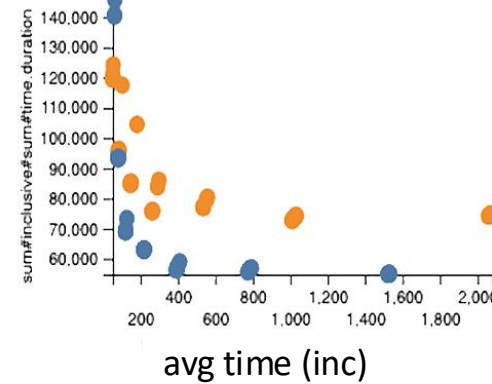
- Thicket's interactive parallel coordinates plot shows relationships between metadata variables, and between metadata and performance data

The metric values are associated with one node in the call tree.

Clicking the crayon separates data by architecture



Each point represents a profile. All profiles are currently selected.



Criss-crossing lines show inverse correlation between number of MPI threads and program runtime

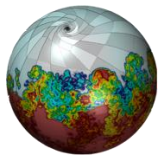
Parallel lines show correlation between program runtime and number of simulated elements

arch

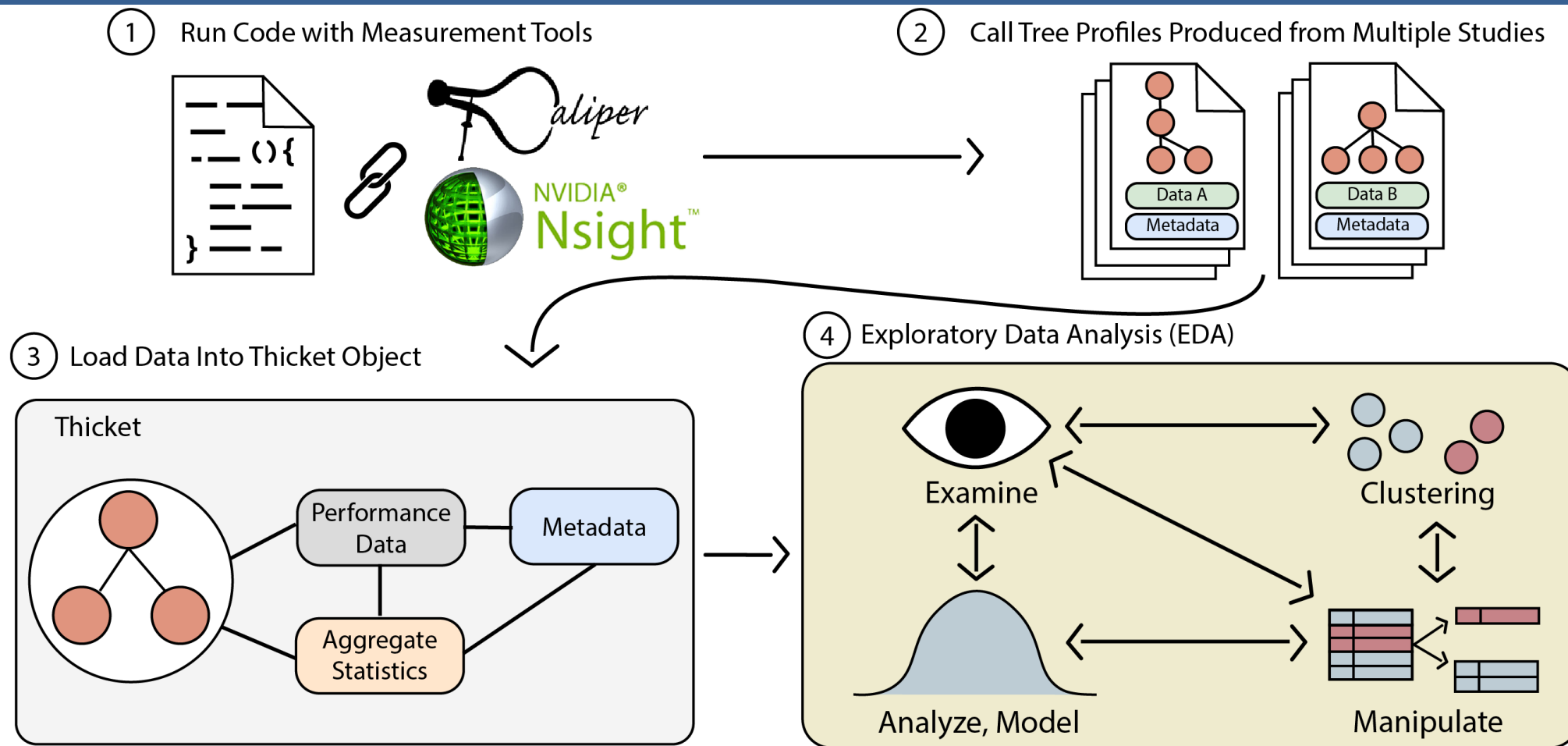
mpi.world.size

walltime

num_elems_max



Thicket is a toolkit for exploratory data analysis of multi-run data



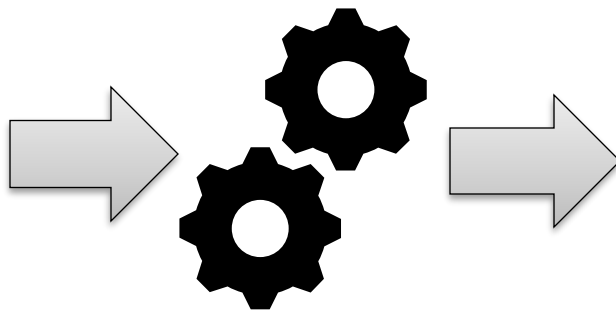
On LC Automated Application Performance Analysis Workflow

Go to <https://lc.llnl.gov/jupyter>

```
#include <caliper/cali.h>

static inline
void LagrangeElements(Domain&
domain, Index_t numElem)
{
    CALI_CXX_MARK_FUNCTION;
    // ...
}
```

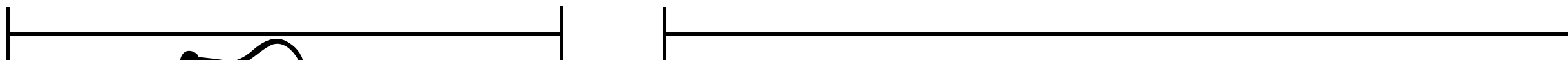
Caliper instrumentation
in the application



At runtime: Performance
and Metadata Collection

The screenshot shows a Jupyter notebook interface with the title 'hatchet-small-graph-example-casc-jan2021'. The code in the notebook includes imports for 'matplotlib inline' and 're', a list of 'WEAK_SCALE_CALI_FILES' with file paths and metric names, and a loop that reads each file into a 'Hatchet GraphFrame', extracts 'nranks' from the filename, filters for functions with a duration greater than 0.25 seconds, and appends the filtered data to a global list of dataframes. Finally, it concatenates the list, formats the 'nranks' column, and creates a line plot of 'nranks' vs. 'inclusive time by function name'.

Visualization and analysis of caliper datasets
using Python in Jupyter notebooks



Jupyter and Thicket

c/o D Boehme



Lawrence Livermore National Laboratory

LLNL-PRES-850268

Documentation: thicket.readthedocs.io



Hands-On Time!

- The <https://github.com/llnl/thicket-tutorial> includes all example Jupyter notebooks, Thicket 2024.1.0 install, and RAJA Performance Suite datasets in a self-contained environment with all dependencies

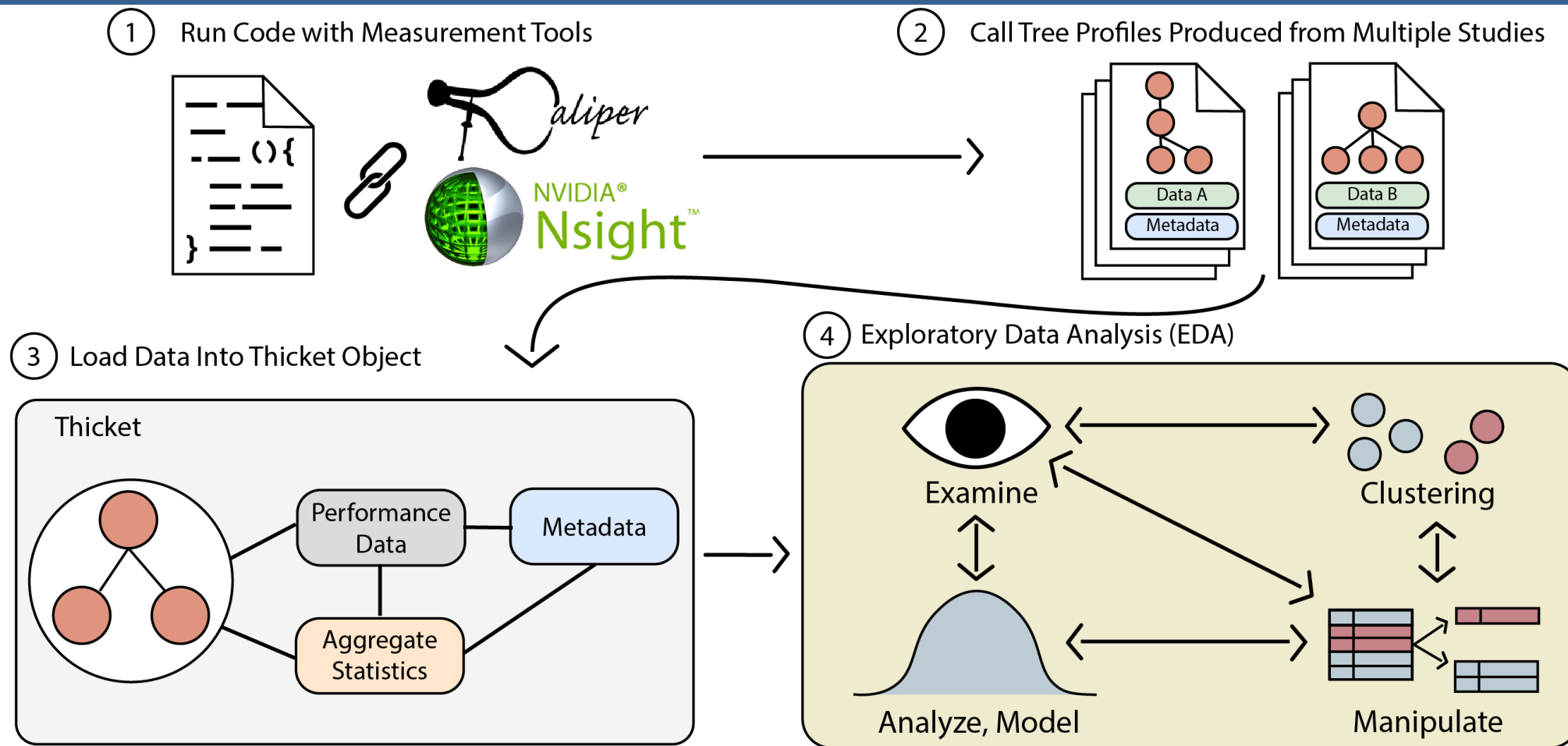
Steps:

- Sign up in the google doc to get your URL for the jupyter notebooks
- We'll start with notebooks/01_thicket_tutorial.ipynb
- Please remember to make a copy of the notebook before you run it!

Join our mailing list! <https://bit.ly/caliper-thicket-users>



Thicket is a toolkit for exploratory data analysis of multi-run data



Join our mailing list! <https://bit.ly/caliper-thicket-users>





CASC

Center for Applied
Scientific Computing



Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.