Enabling Call Path Querying in Hatchet to Identify Performance Bottlenecks in Scientific Applications

Ian Lumsden, Jakob Luettgau, Vanessa Lama, Connor Scully-Allison, Stephanie Brink, Katherine E. Isaacs, Olga Pearce, Michela Taufer

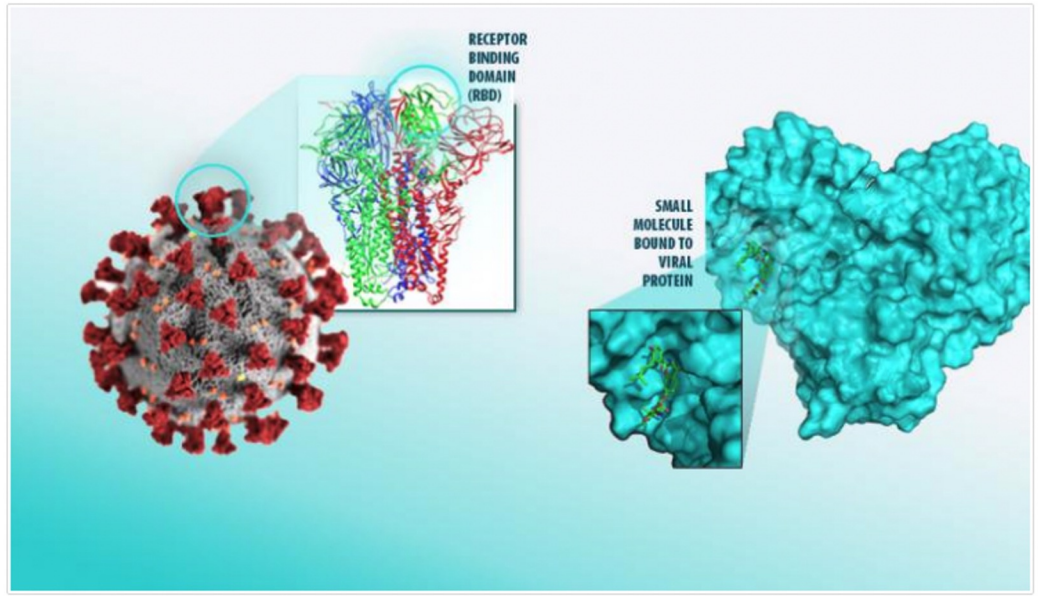
October 13, 2022
2022 IEEE International Conference on eScience

LLNL-PRES-840755

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Importance of HPC in Scientific Computing

Nov. 17, 2020



(Download Image)

Using Sierra, the world's third fastest supercomputer, LLNL scientists produced a more accurate and efficient generative model to enable COVID-19 researchers to produce novel compounds that could possibly treat the disease. The team trained the model on an unprecedented 1.6 billion small molecule compounds and one million additional promising compounds for COVID-19, reducing the model training time from one day to just 23 minutes.

Model for COVID-19 drug discovery a Gordon Bell finalist

A machine learning model developed by a team of Lawrence Livermore National Laboratory (LLNL) scientists to aid in COVID-19 drug discovery efforts is a finalist for the Gordon Bell Special Prize for High Performance Computing-Based COVID-19 Research.

Jeremy Thomas
thomas244@llnl.gov
925-422-5539

Investigating Dyes for Solar Cells from Start to Finish

Finding the right dyes for a new type of solar cell can be challenging, but this study used supercomputers to speed up the process

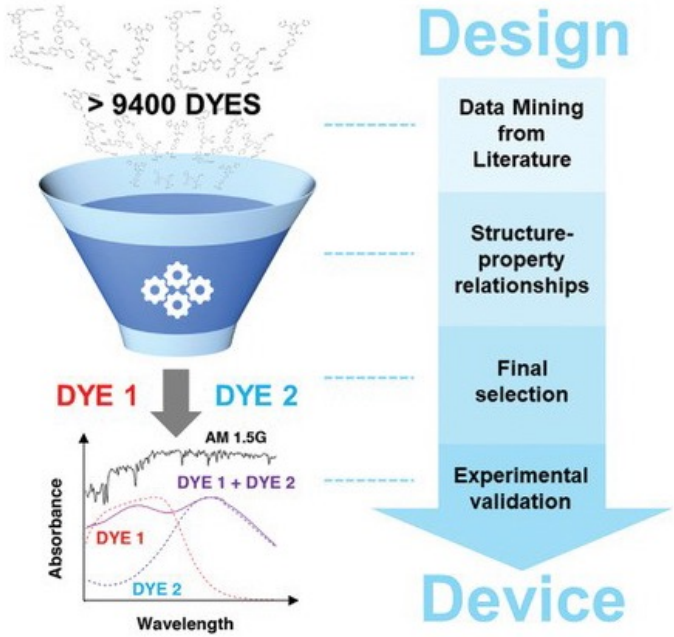
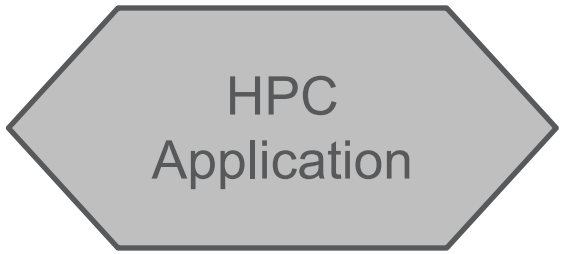


Image courtesy of Dr. Jacqueline M. Cole (University of Cambridge UK and Argonne National Laboratory)

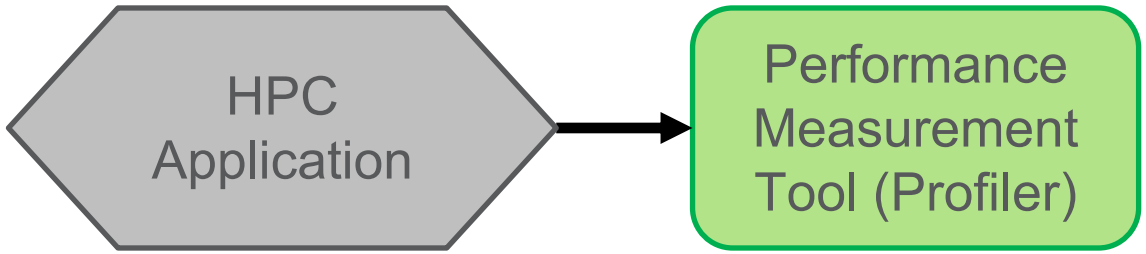
Dataflow in the design-to-device study for a panchromatic photovoltaic cell.

[1] J. Thomas, "Model for COVID-19 drug discovery a Gordon Bell finalist," *Lawrence Livermore National Laboratory*, 2020. Available: https://www.llnl.gov/news/model-covid-19-drug-discovery-gordon-bell-finalist.
[2] "Investigating Dyes for Solar Cells from Start to Finish," *U.S. DOE Office of Science (SC)*, Oct. 28, 2019. Available: https://science.osti.gov/ascr/Highlights/2019/ASCR-2019-10-a.

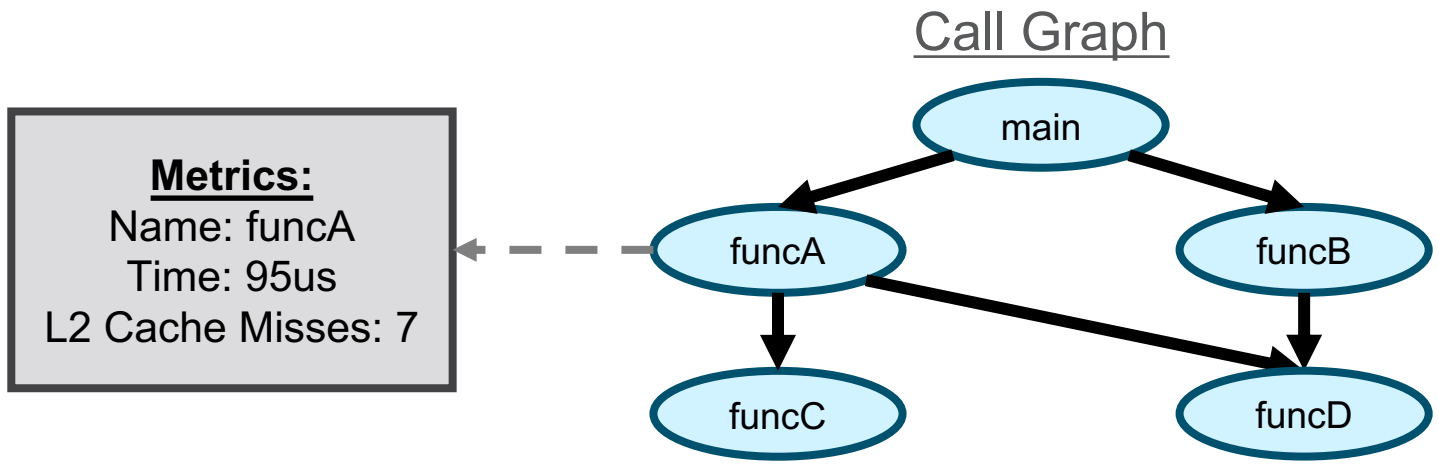
HPC Performance Analysis Workflow



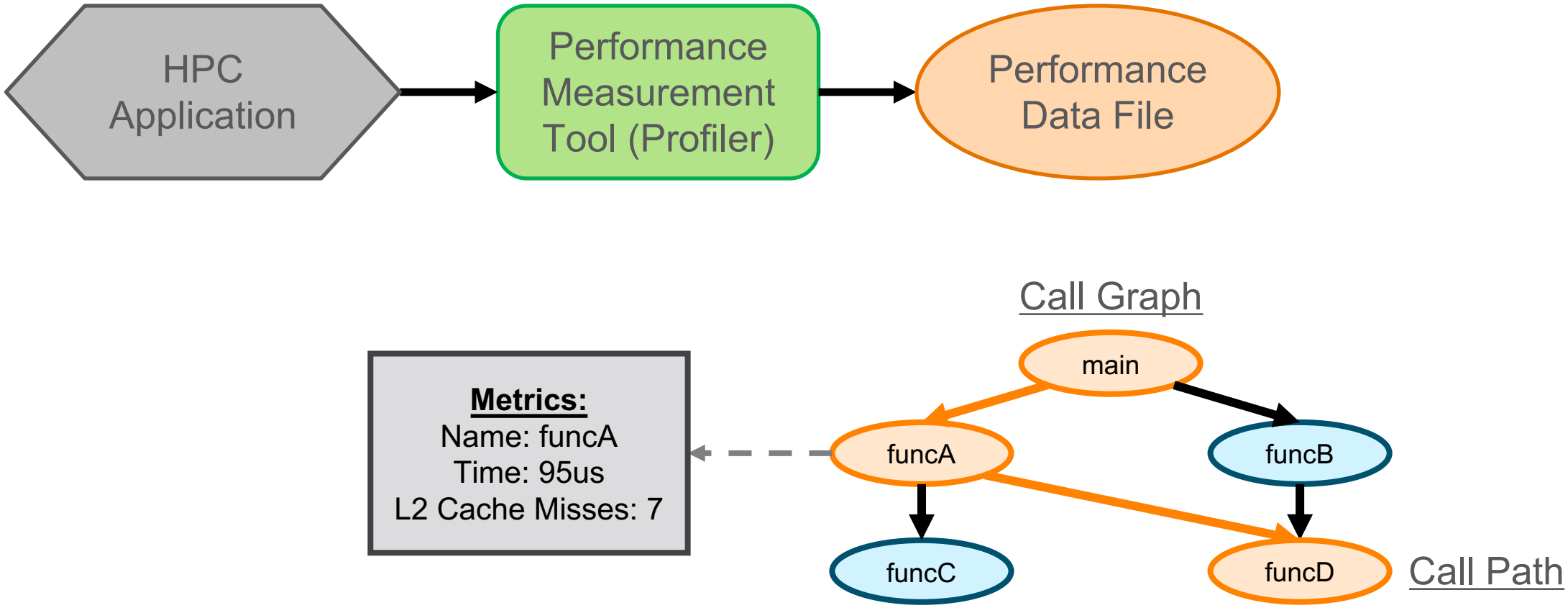
HPC Performance Analysis Workflow



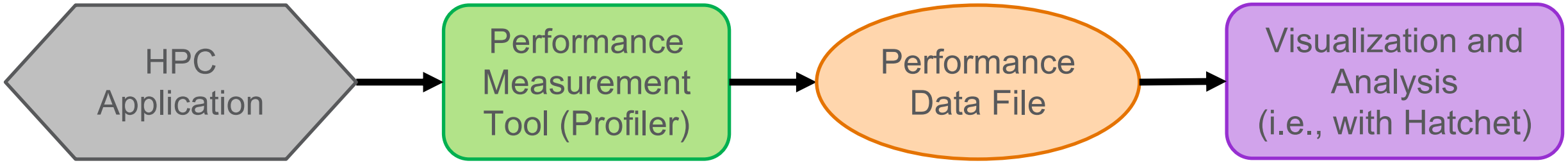
HPC Performance Analysis Workflow



HPC Performance Analysis Workflow

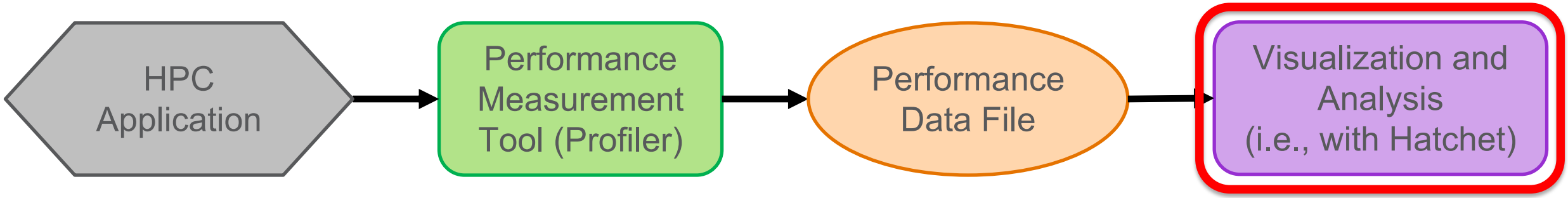


HPC Performance Analysis Workflow

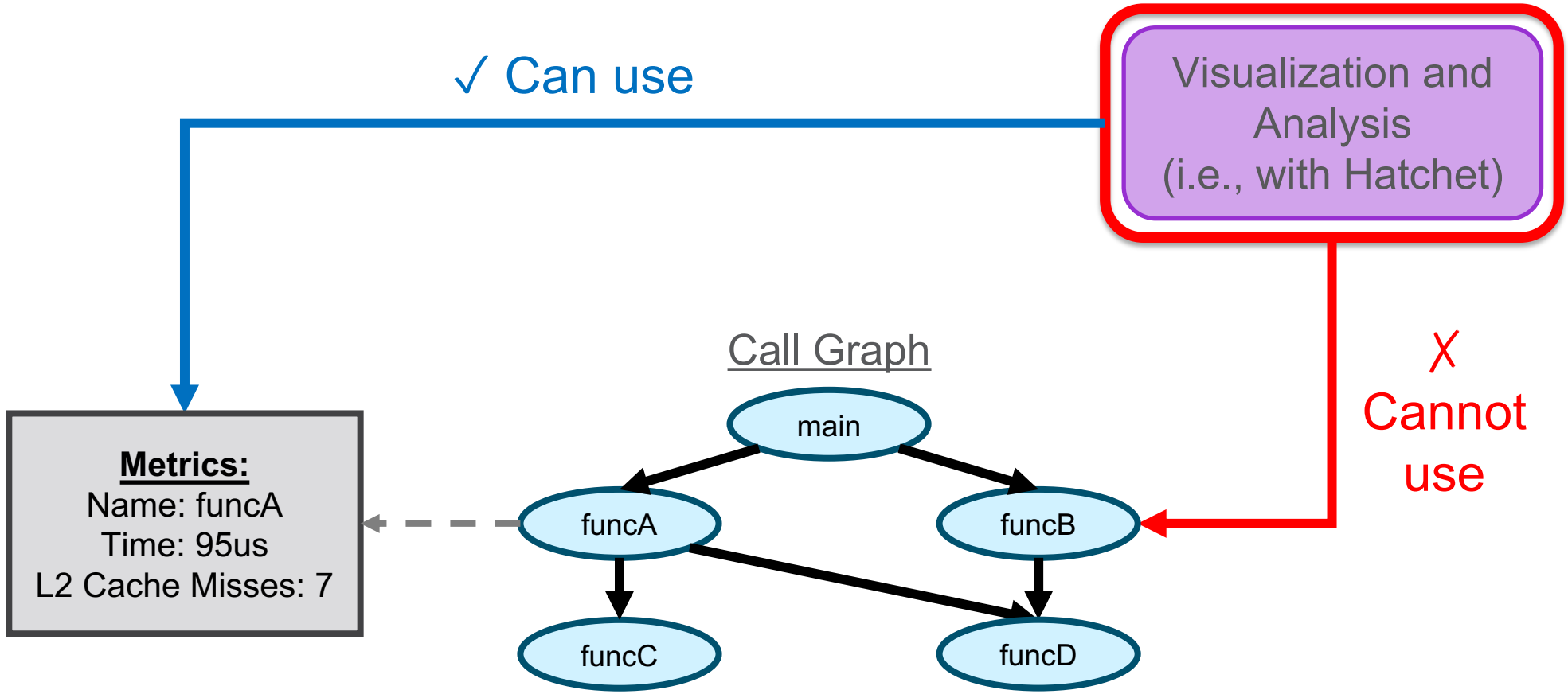


[3] A. Bhatele, S. Brink, and T. Gamblin, “Hatchet: Pruning the Overgrowth in Parallel Profiles,” in Proceedings of SC 2019, November 17-19, 2019. Available: https://doi.org/10.1145/3295500.3356219

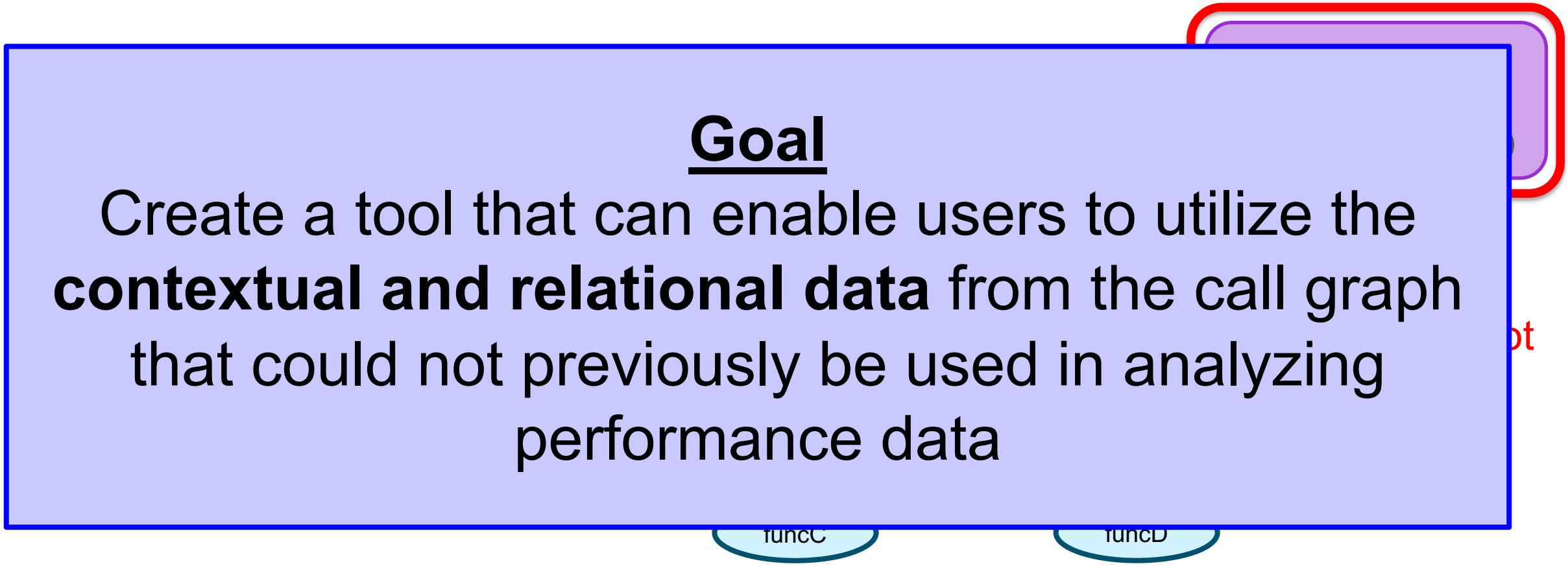
The Problem with Performance Analysis



The Problem with Performance Analysis



The Problem with Performance Analysis



Goal

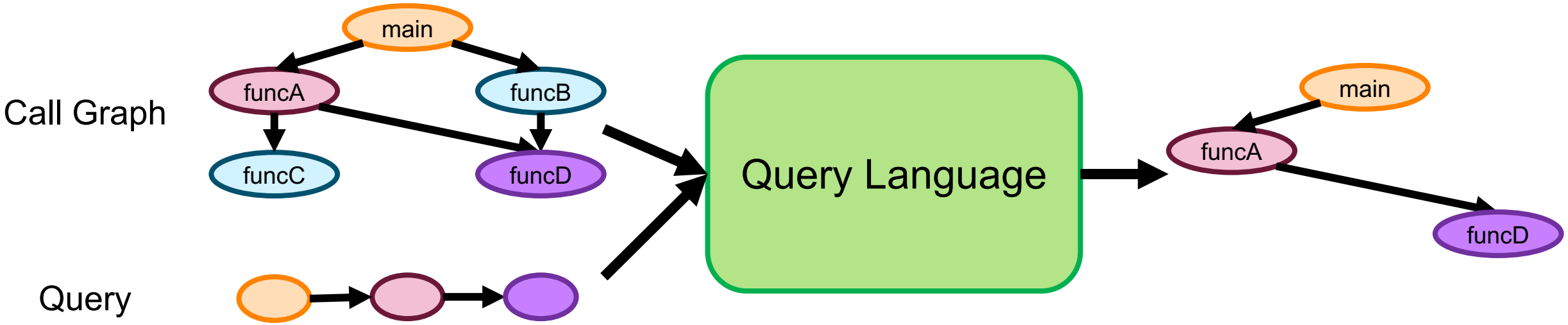
Create a tool that can enable users to utilize the **contextual and relational data** from the call graph that could not previously be used in analyzing performance data

Contributions

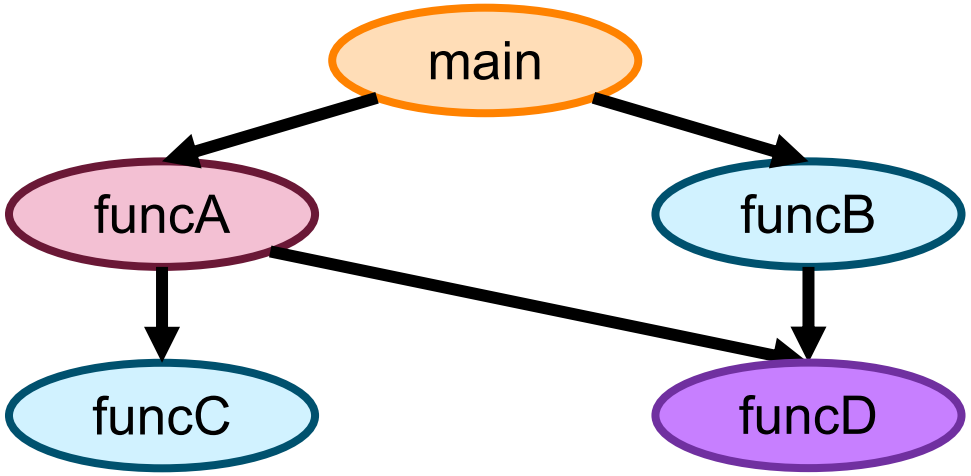
- Design and implement a new *Call Path Query Language* in Hatchet
- Define two dialects for our Query Language to simplify its use under diverse circumstances
- Classify the abilities of our Query Language and its dialects into features and capabilities
- Demonstrate the benefits of our Query Language through three case studies

Call Path Query Language

- Enables Hatchet users to extract a set of paths that match certain properties from a call graph
- Two parts: (1) Query Composition and (2) Algorithm



Call Path Query Language: Composition



Query

Query Node (1)

Quantifier
defines how many "real" nodes in a call path to match to a query node

Predicate
defines what conditions must be satisfied for a "real" node to match a query node

Query Node (2)

Query Node (3)

Call Path Query Language: Composition

“Find all paths that start with a MPI node with more than 5 L2 cache misses, followed by 0 or more of any node”

OR

“Find all subgraphs rooted at a MPI node with more than 5 L2 cache misses”

Call Path Query Language: Composition

"Find all paths that start with a MPI node with more than 5 L2 cache misses, followed by 0 or more of any node"

OR

"Find all subgraphs rooted at a MPI node with more than 5 L2 cache misses"

"Base" Syntax:

```
query = (
    QueryMatcher()
    .match(
        ".",
        lambda row: re.match(
            "MPI_.*",
            row["name"]
        )
        is not None
        and row["PAPI_L2_TCM"] > 5
    )
    .rel("*")
)
```

Call Path Query Language: Composition

Query:
QueryMatcher
Python class

Query Node:

- `match()` sets the first query node in the query
- `rel()` adds a query node to the end of the query

"Base" Syntax:

```
query = (
QueryMatcher()
.match(
    ".",
    lambda row: re.match(
        "MPI_.*",
        row["name"]
    )
    is not None
    and row["PAPI_L2_TCM"] > 5
)
.rel("*")
)
```

Call Path Query Language: Composition

“Base” Syntax:

Quantifier:

- “.”: match 1
- “*”: match 0 or more
- “+”: match 1 or more
- **Integer**: match exact number

```
query = (
    QueryMatcher()
    .match(
        “.”,
        lambda row: re.match(
            “MPI_.*”,
            row[“name”]
        )
        is not None
        and row[“PAPI_L2_TCM”] > 5
    )
    .rel(“*”)
)
```

Call Path Query Language: Composition

“Base” Syntax:

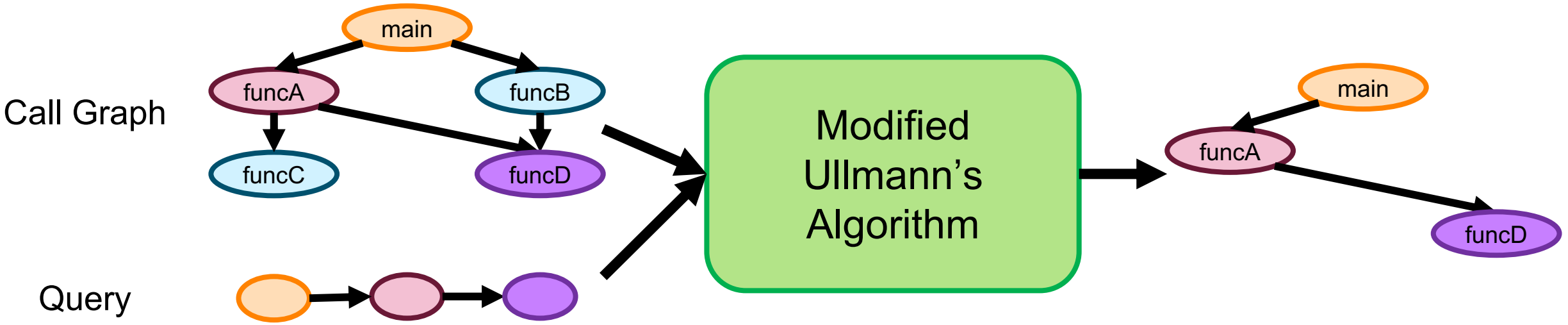
```
query = (
    QueryMatcher()
    .match(
        “.”,
        lambda row: re.match(
            “MPI_.*”,
            row[“name”]
        )
        is not None
        and row[“PAPI_L2_TCM”] > 5
    )
    .rel(“*”)
)
```

Predicate:
Python Callable with metrics as input and Boolean as output

Call Path Query Language: Algorithm

Problem:

Find **all** paths in a call graph that match properties described by the user-provided query (subgraph isomorphism)

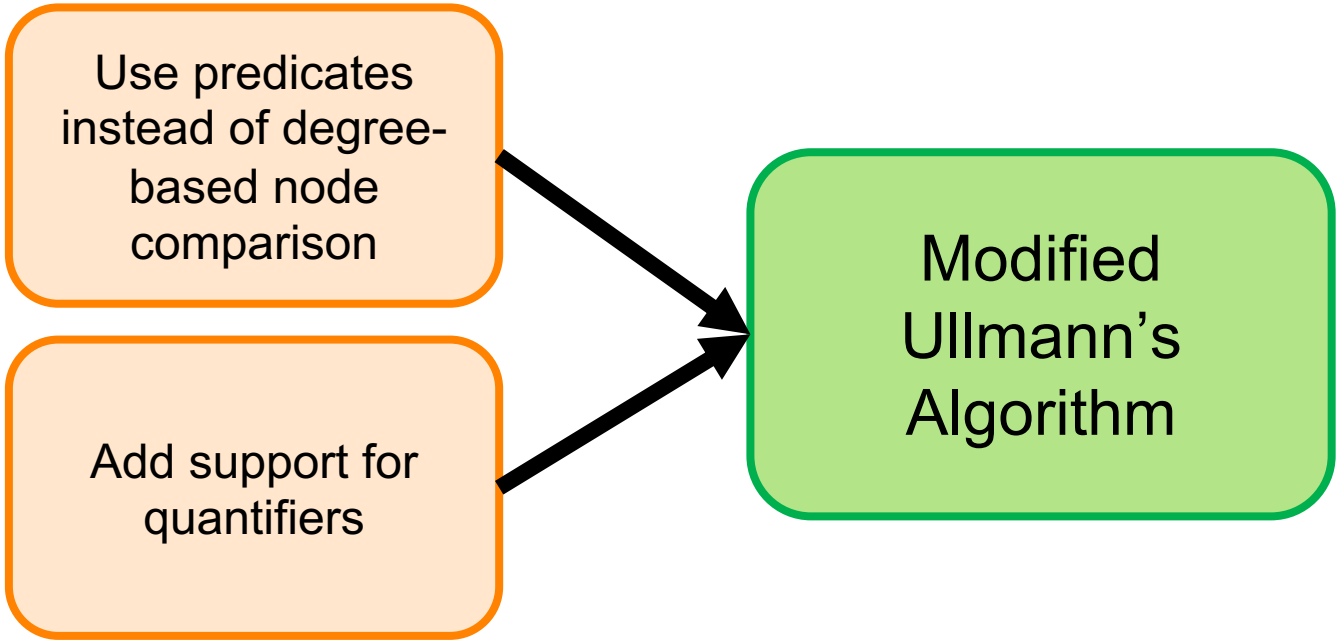


[4] J. R. Ullmann, "An Algorithm for Subgraph Isomorphism," J. ACM, vol. 23, no. 1, pp. 31–42, 1976. Available: http://doi.acm.org/10.1145/321921.321925

Call Path Query Language: Algorithm

Problem:

Find **all** paths in a call graph that match properties described by the user-provided query (subgraph isomorphism)



Call Path Query Language

For some users, this syntax is:

- Too verbose and complex
- Unusable in some situations (e.g., creating a query in another programming language)

"Base" Syntax:

```
query = (
    QueryMatcher()
    .match(
        ".",
        lambda row: re.match(
            "MPI_.*",
            row["name"]
        )
        is not None
        and row["PAPI_L2_TCM"] > 5
    )
    .rel("*")
)
```

Contributions

- Design and implement a new *Call Path Query Language* in Hatchet
- Define two dialects for our Query Language to simplify its use under diverse circumstances
- Classify the abilities of our Query Language and its dialects into features and capabilities
- Demonstrate the benefits of our Query Language through three case studies

Simplifying Queries with Dialects

- Simplify use of Call Path Query Language and allow use under diverse circumstances
- Internally translated into “base” Query Language syntax
- Two Dialects:
  - Object-based Dialect
    - Provides a simpler, less verbose way of writing queries
  - String-based Dialect
    - Provides a way to write queries that is decoupled from Python

Simplifying Queries with Dialects

"Base" Syntax:

```
query = (
    QueryMatcher()
    .match(
        ".",
        lambda row: re.match(
            "MPI_.*",
            row["name"]
        )
        is not None
        and row["PAPI_L2_TCM"] > 5
    )
    .rel("*")
)
```

Simplifying Queries with Dialects

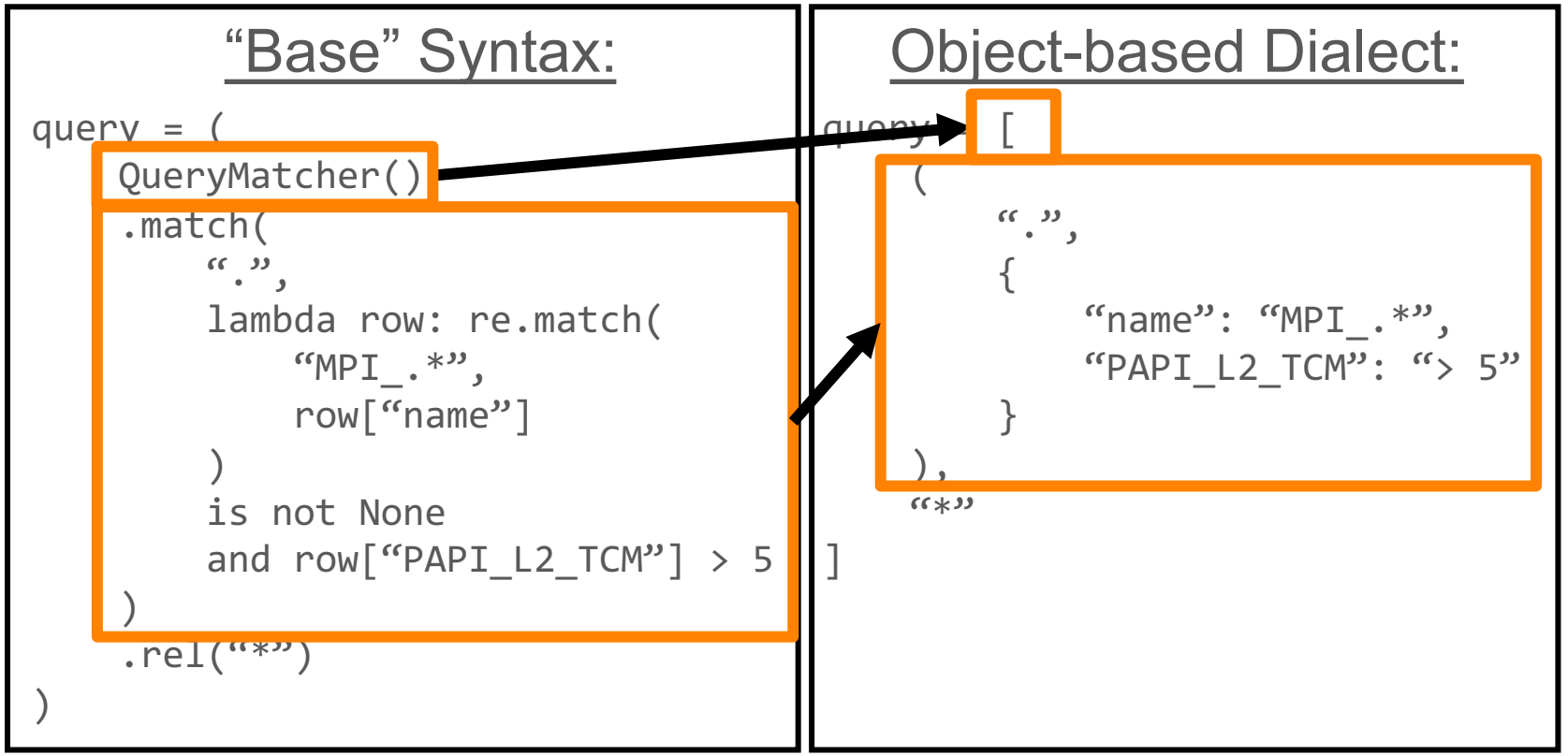
“Base” Syntax:

```
query = (
    QueryMatcher()
    .match(
        “.”,
        lambda row: re.match(
            “MPI_.*”,
            row[“name”]
        )
        is not None
        and row[“PAPI_L2_TCM”] > 5
    )
    .rel(“*”)
)
```

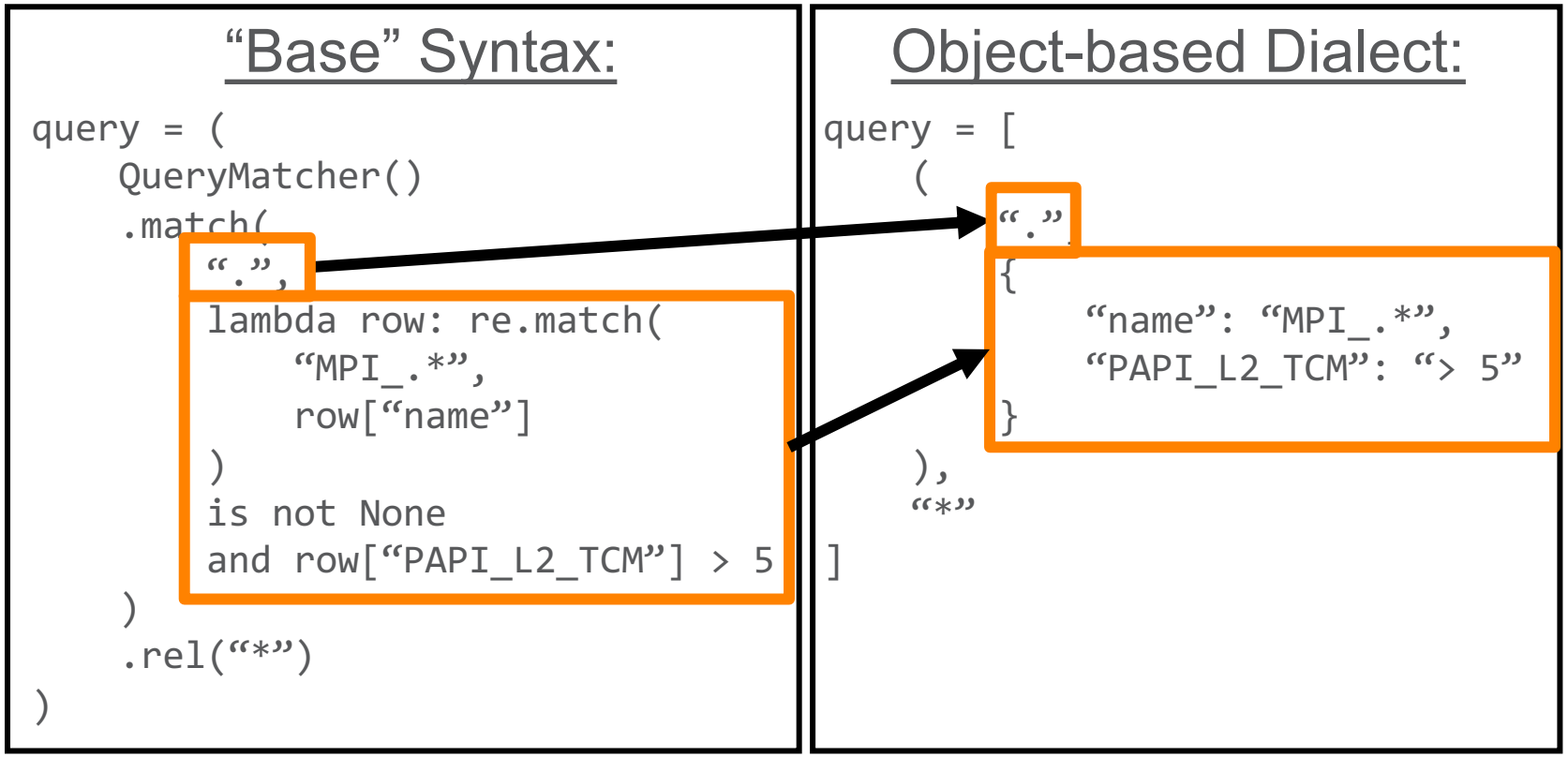
Object-based Dialect:

```
query = [
    (
        “.”,
        {
            “name”: “MPI_.*”,
            “PAPI_L2_TCM”: “> 5”
        }
    ),
    “*”
]
```

Simplifying Queries with Dialects



Simplifying Queries with Dialects



Simplifying Queries with Dialects

“Base” Syntax:

```
query = (
    QueryMatcher()
    .match(
        “.”,
        lambda row: re.match(
            “MPI_.*”,
            row[“name”]
        )
        is not None
        and row[“PAPI_L2_TCM”] > 5
    )
    .rel(“*”)
)
```

Object-based Dialect:

```
query = [
    (
        “.”,
        {
            “name”: “MPI_.*”,
            “PAPI_L2_TCM”: “> 5”
        }
    ),
    “*”
]
```

String-based Dialect:

```
query = """
MATCH (".", p)->("*")
WHERE p."name" =~ "MPI_.*" AND
    p."PAPI_L2_TCM" > 5
"""
```

Simplifying Queries with Dialects

“Base” Syntax:

```
query = (
    QueryMatcher()
    .match(
        “.”,
        lambda row: re.match(
            “MPI_.*”,
            row[“name”]
        )
        is not None
        and row[“PAPI_L2_TCM”] > 5
    )
    .rel(“*”)
)
```

Object-based Dialect:

```
query = [
    (
        “.”,
        {
            “name”: “MPI_.*”,
            “PAPI_L2_TCM”: “> 5”
        }
    ),
    “*”
]
```

String-based Dialect:

```
query = """
MATCH (".", p)->("*")
WHERE p."name" =~ "MPI_.*" AND
    p."PAPI_L2_TCM" > 5
"""
```

Simplifying Queries with Dialects

“Base” Syntax:

```
query = (
    QueryMatcher()
    .match(
        “ ”,
        lambda row: re.match(
            “MPI_.*”,
            row[“name”]
        )
        is not None
        and row[“PAPI_L2_TCM”] > 5
    )
    .rel(“*”)
)
```

Object-based Dialect:

```
query = [
    (
        “.”,
        {
            “name”: “MPI_.*”,
            “PAPI_L2_TCM”: “> 5”
        }
    ),
    “*”
]
```

String-based Dialect:

```
query = """
MATCH (" ", p)->("*")
WHERE p."name" =~ "MPI_.*" AND
    p."PAPI_L2_TCM" > 5
"""
```

Simplifying Queries with Dialects

“Base” Syntax:

```
query = (
    QueryMatcher()
    .match(
        “.”,
        lambda row: re.match(
            “MPI_.*”,
            row[“name”]
        )
        is not None
        and row[“PAPI_L2_TCM”] > 5
    )
    .rel(“*”)
)
```

Object-based Dialect:

```
query = [
    (
        “.”,
        {
            “name”: “MPI_.*”,
            “PAPI_L2_TCM”: “> 5”
        }
    ),
    “*”
]
```

String-based Dialect:

```
query = """
MATCH (".", p)->("*")
WHERE p."name" =~ "MPI_.*" AND
    p."PAPI_L2_TCM" > 5
"""
```

1 3 2

Complexity and Verbosity (Most to Least)

Simplifying Queries with Dialects

"Base" Syntax:

- Pros
  - Can represent complex queries
- Cons
  - Most complex and verbose syntax
  - Requires knowledge of other libraries (e.g., pandas)
  - Can only be created in Python

Object-based Dialect:

- Pros
  - Simplest and least verbose syntax
  - Great for simple queries
- Cons
  - Cannot represent complex queries
  - Can only be created in Python

String-based Dialect:

- Pros
  - Can be created in any programming language
  - Simpler and less verbose than "base" syntax
- Cons
  - Cannot represent some complex queries
  - Requires learning a custom language

Contributions

- Design and implement a new *Call Path Query Language* in Hatchet
- Define two dialects for our Query Language to simplify its use under diverse circumstances
- Classify the abilities of our Query Language and its dialects into features and capabilities*
- Demonstrate the benefits of our Query Language through three case studies

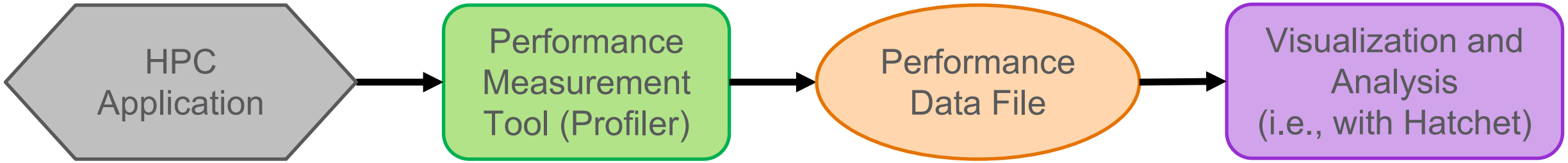
* If you want to see the differences in features between the Query Language and its dialects, check out our paper: “Enabling Call Path Querying in Hatchet to Identify Performance Bottlenecks in Scientific Applications”

Contributions

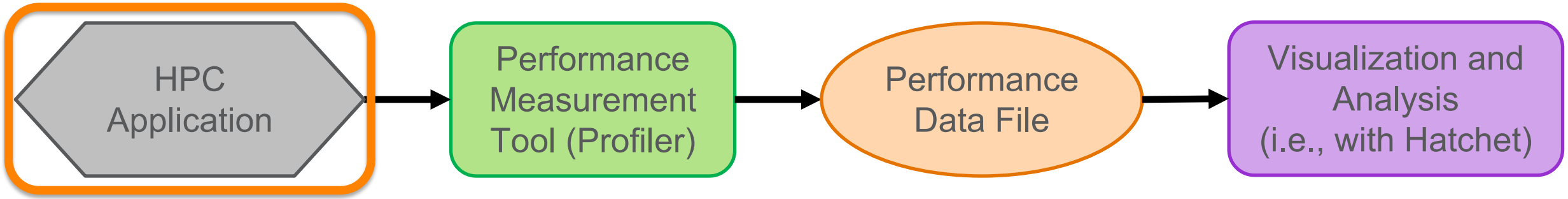
- Design and implement a new *Call Path Query Language* in Hatchet
- Define two dialects for our Query Language to simplify its use under diverse circumstances
- Classify the abilities of our Query Language and its dialects into features and capabilities
- Demonstrate the benefits of our Query Language through three case studies*

* Only one case study is shown due to time. If you want to see the others, check out our paper: "Enabling Call Path Querying in Hatchet to Identify Performance Bottlenecks in Scientific Applications"

Comparing Tools with Queries



Comparing Tools with Queries



- AMG2013:
  - Benchmark from the CORAL-2 suite
  - Parallel algebraic multigrid solver for linear systems on unstructured meshes
    - **Used in applications like Computational Fluid Dynamics**
- MPI Libraries:
  - MVAPICH
  - Spectrum-MPI

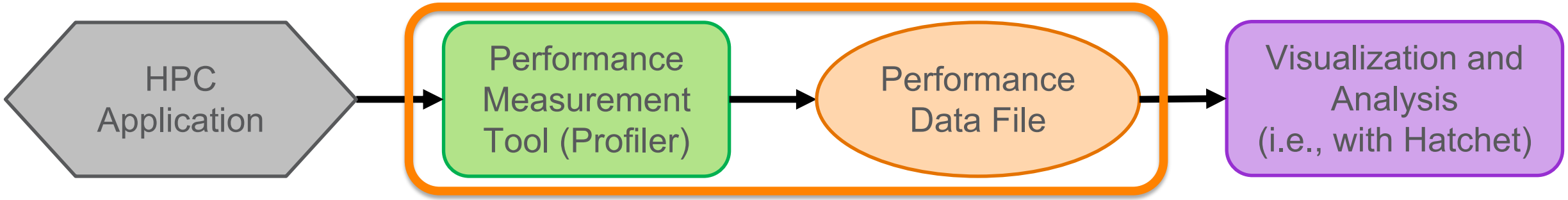
[5] "AMG2013," Lawrence Livermore National Laboratory. Available: https://asc.llnl.gov/codes/proxy-apps/amg2013.
[6] "CORAL-2 Benchmarks," Lawrence Livermore National Laboratory. Available: https://asc.llnl.gov/coral-2-benchmarks.
[7] K. Stüben, "A review of algebraic multigrid," J. Comput. Appl. Math., no. 1–2, pp. 281–309, 2001, Available: https://doi.org/10.1016/S0377-0427(00)00516-1

Comparing Tools with Queries

- All runs performed on LLNL's Lassen supercomputer
  - 795 AC922 nodes
    - **2 IBM POWER9 CPUs per node (20 usable cores per node)**
    - 256 GB Memory
    - NVIDIA V100 GPUs
  - **InfiniBand EDR Interconnect**

[8] "Lassen," HPC @ LLNL. Available: https://hpc.llnl.gov/hardware/compute-platforms/lassen.
[9] "Using LC's Sierra Systems," HPC @ LLNL. Available: https://hpc.llnl.gov/documentation/tutorials/using-lc-s-sierra-systems.

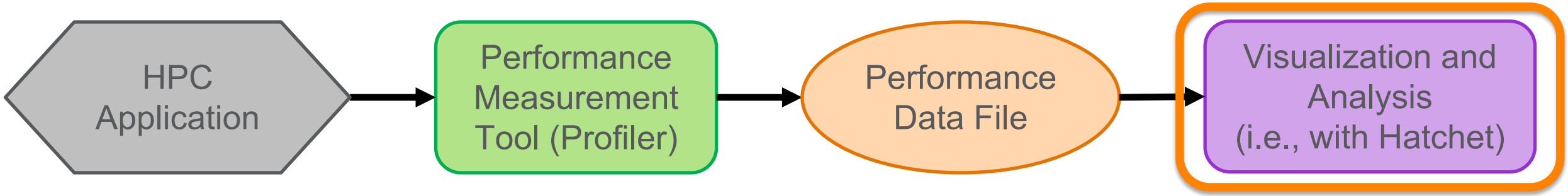
Comparing Tools with Queries



- Run AMG2013 with MVAPICH and Spectrum-MPI while profiling with HPCToolkit
  - 64, 128, 256, and 512 MPI ranks

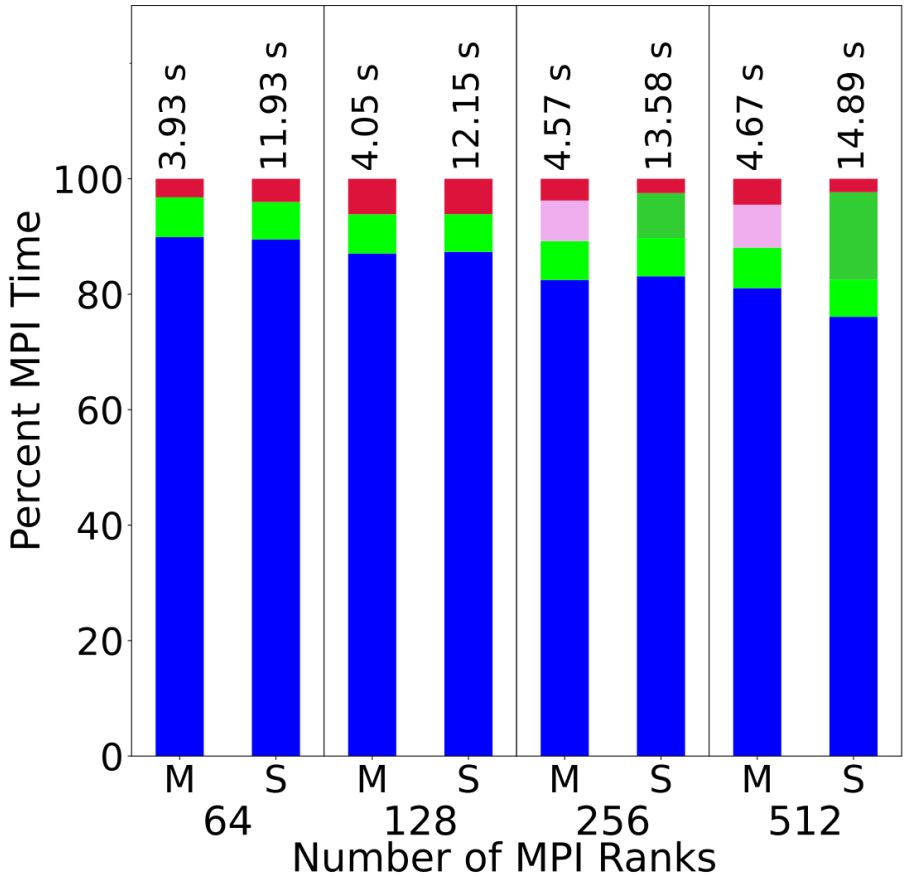
[10] L. Adhianto et al., “HPCTOOLKIT: tools for performance analysis of optimized parallel programs,” Concurrency and Computation: Practice and Experience, 2009. Available: https://doi.org/10.1002/cpe.1553

Comparing Tools with Queries



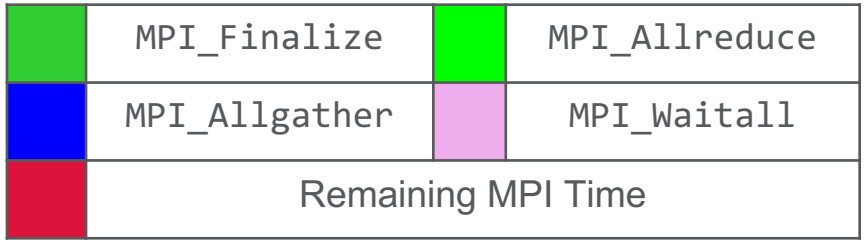
- **Goal:** compare the abilities of existing tools (i.e., Hatchet) and our Query Language at extracting knowledge from performance data
- **Case Study:** compare the performance of MVAPICH and Spectrum-MPI by running AMG2013 at different scales
  - Use Hatchet with and without our Query Language
  - Use Object-based Dialect

Round 1: MPI w/o Query Language

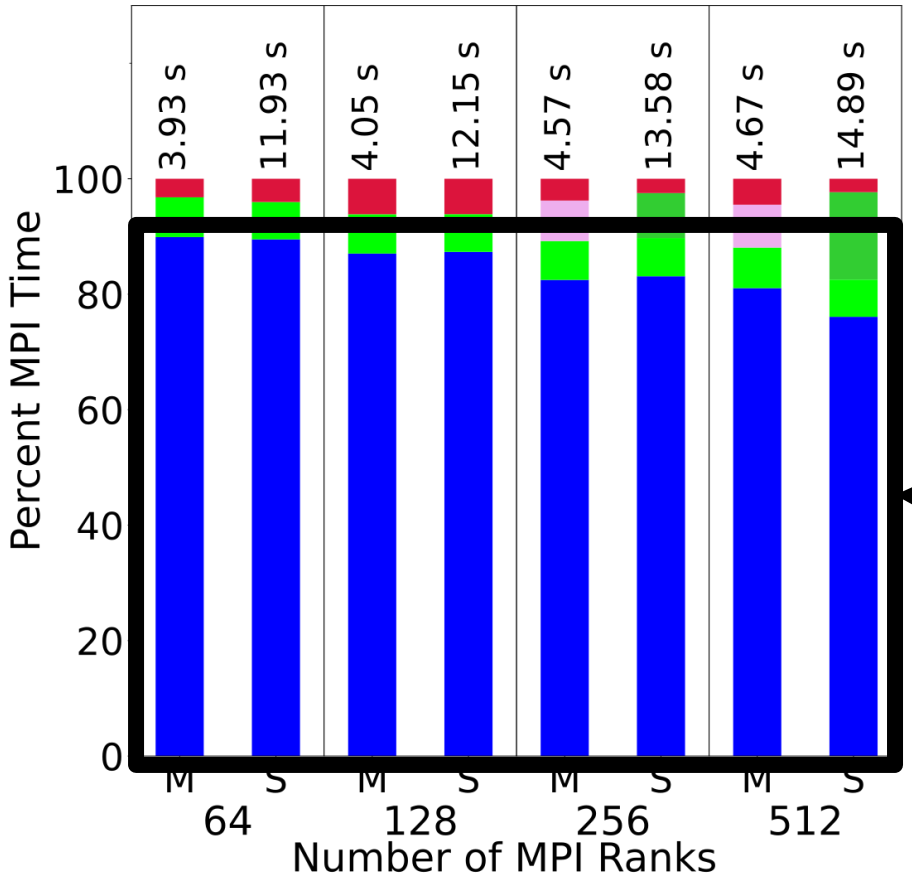


M = MVAPICH S = Spectrum-MPI

- Determine how much time was spent in each MPI function
- Hatchet without Query Language
- “Remaining MPI Time” = functions that take less than 5% of total MPI time



Round 1: MPI w/o Query Language



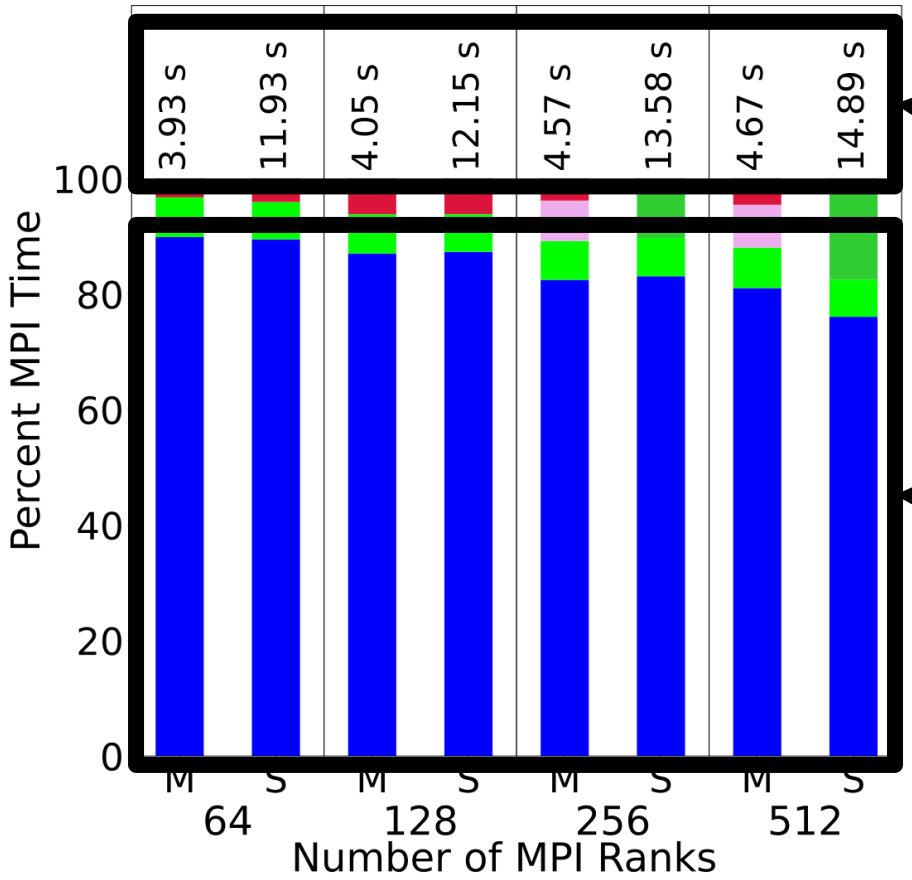
M = MVAPICH S = Spectrum-MPI

`MPI_Allgather` takes more than 75% of total MPI time for all runs

	`MPI_Finalize`		`MPI_Allreduce`
	`MPI_Allgather`		`MPI_Waitall`
	Remaining MPI Time*		

* ”Remaining MPI Time” = functions that take less than 5% of total MPI time

Round 1: MPI w/o Query Language



M = MVAPICH S = Spectrum-MPI

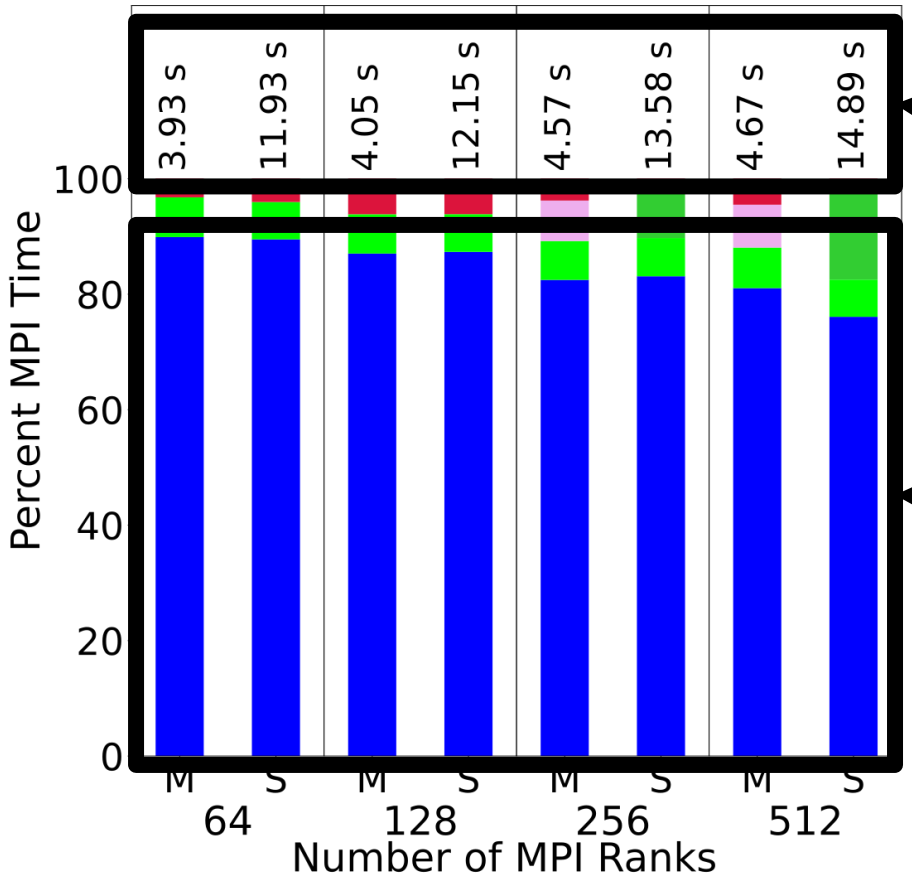
MVAPICH significantly outperforms Spectrum-MPI

`MPI_Allgather` takes more than 75% of total MPI time for all runs

	`MPI_Finalize`		`MPI_Allreduce`
	`MPI_Allgather`		`MPI_Waitall`
	Remaining MPI Time*		

* ”Remaining MPI Time” = functions that take less than 5% of total MPI time

Round 1: MPI w/o Query Language



M = MVAPICH S = Spectrum-MPI

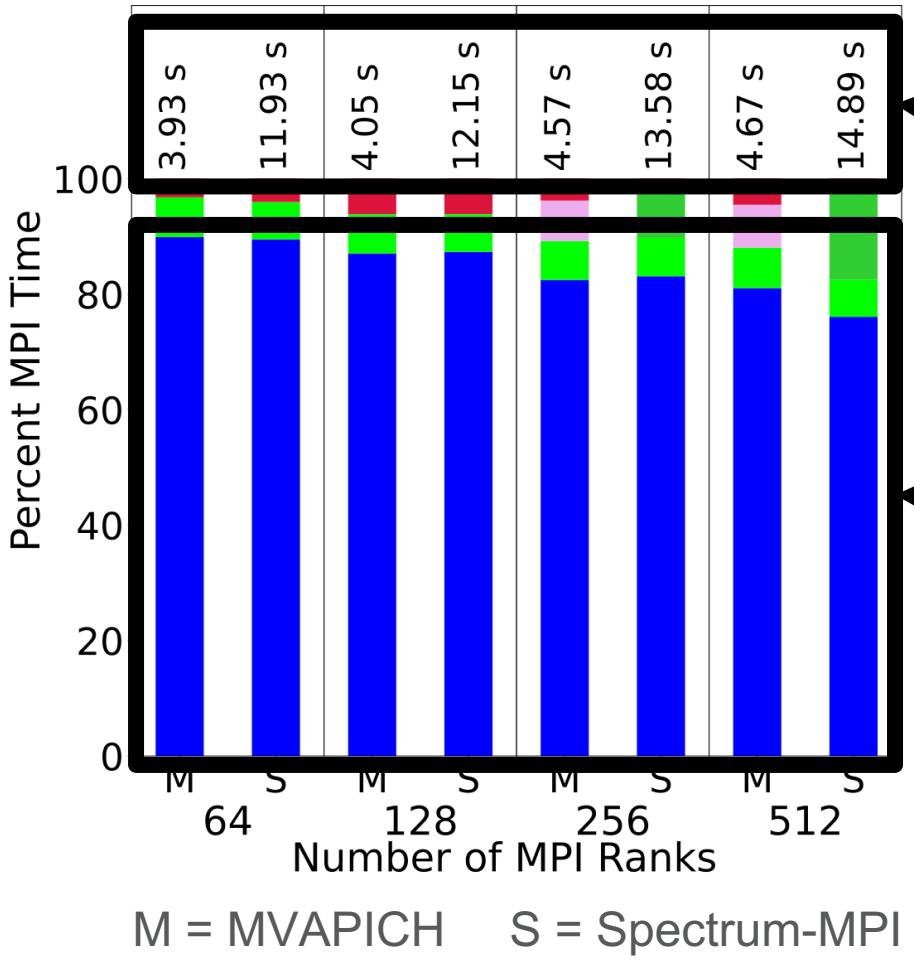
Spectrum-MPI's poor performance can hurt scientific applications that depend on it

`MPI_Allgather` takes more than 75% of total MPI time for all runs

	`MPI_Finalize`		`MPI_Allreduce`
	`MPI_Allgather`		`MPI_Waitall`
	Remaining MPI Time*		

* "Remaining MPI Time" = functions that take less than 5% of total MPI time

Round 1: MPI w/o Query Language

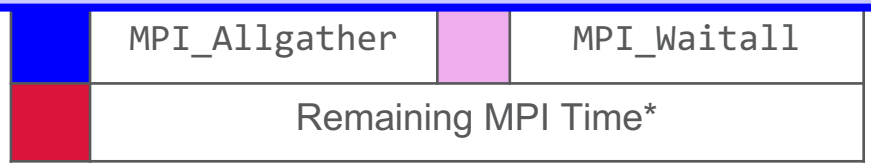


M = MVAPICH S = Spectrum-MPI

Spectrum-MPI's poor performance can hurt scientific applications that depend on it

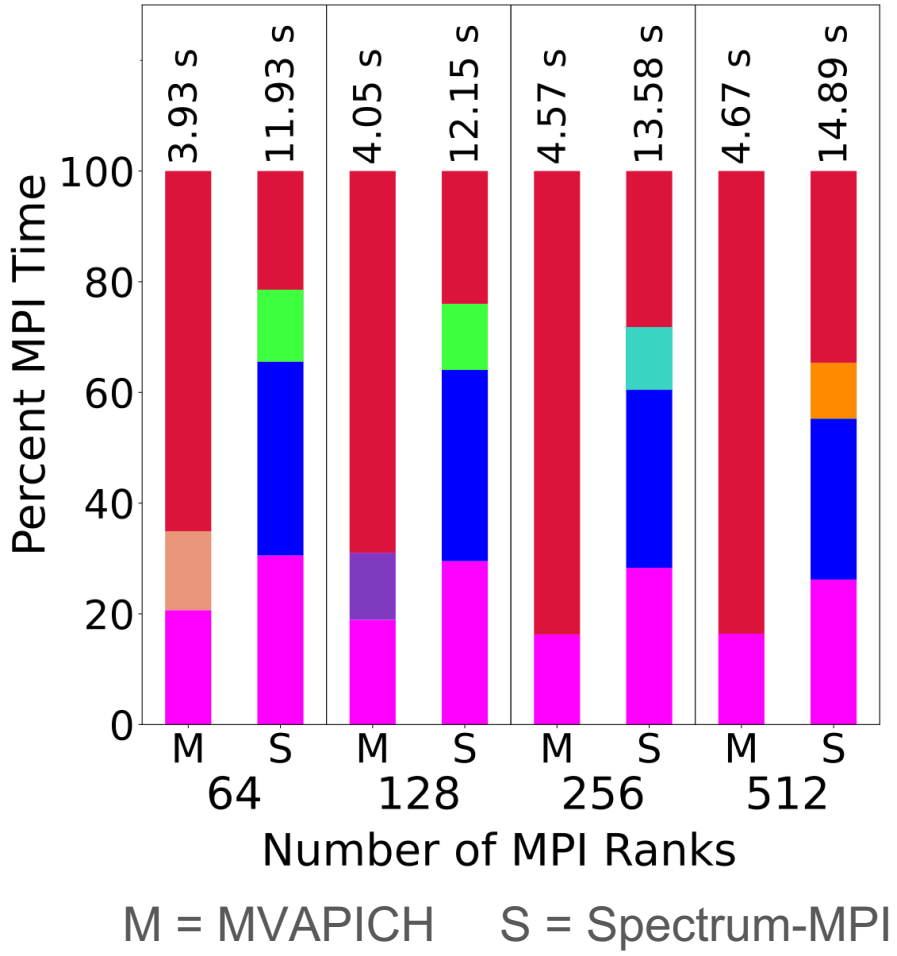
Without Query Language:

We **cannot** determine any specific potential cause of Spectrum-MPI's poor performance



* "Remaining MPI Time" = functions that take less than 5% of total MPI time

Round 2: MPI w/ Query Language



M = MVAPICH S = Spectrum-MPI

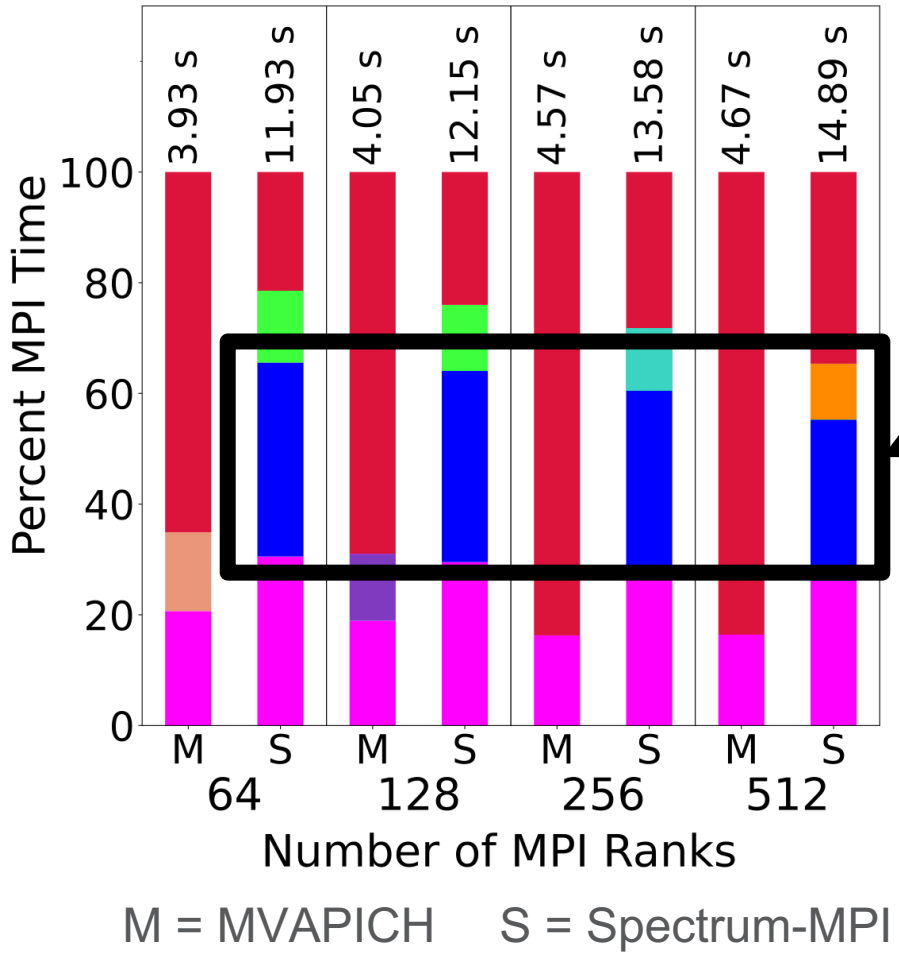
- Determine how much time was spent in the children of the MPI functions
- Hatchet with Query Language
  - Use non-Query Language filtering to remove MPI nodes

```
[{"name": "P?MPI_.*"}, "*"]
```

	pthread_spin_lock.c:26		memset.S:1133
	<unknown file> [libmlx5.so.1.0.0]:0		<unknown file> [libmlx5.so.1.0.0]:1133
	stl_vector.h:0		Geometry.h:0
	malloc.c:0		Remaining MPI Time*

* ”Remaining MPI Time” = functions that take less than 5% of total MPI time

Round 2: MPI w/ Query Language

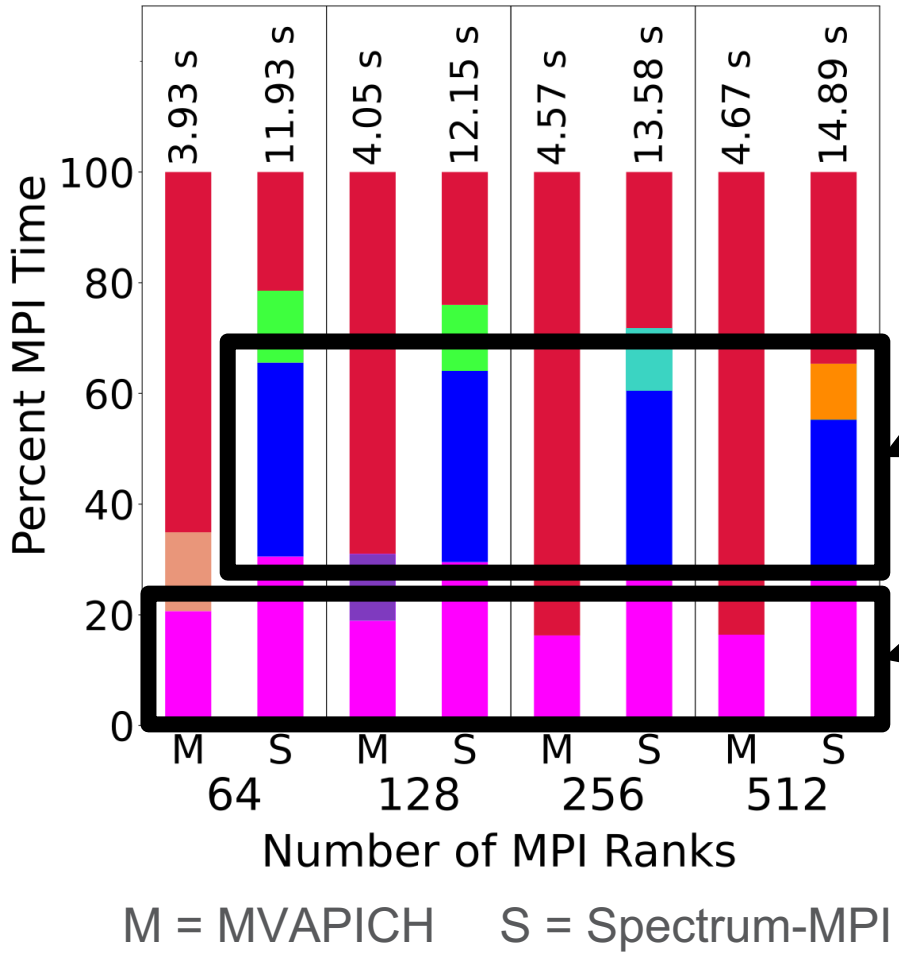


Spectrum-MPI uses libmlx5 (Mellanox InfiniBand driver) much more than MVAPICH

	pthread_spin_lock.c:26		memset.S:1133
	<unknown file> [libmlx5.so.1.0.0]:0		<unknown file> [libmlx5.so.1.0.0]:1133
	stl_vector.h:0		Geometry.h:0
	malloc.c:0		Remaining MPI Time*

* ”Remaining MPI Time” = functions that take less than 5% of total MPI time

Round 2: MPI w/ Query Language



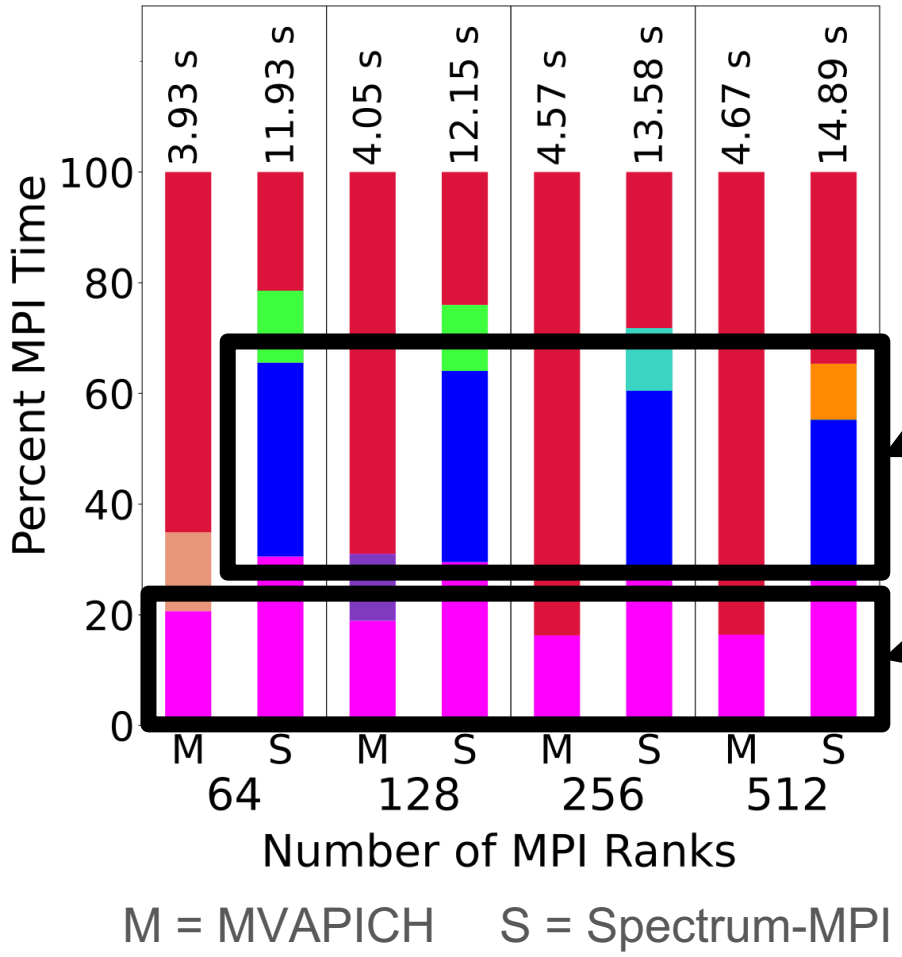
Spectrum-MPI uses libmlx5 (Mellanox InfiniBand driver) much more than MVAPICH

Spectrum-MPI uses `pthread_spin_lock` for a larger percentage of its MPI time than MVAPICH

	`pthread_spin_lock.c:26`		`memset.S:1133`
	`<unknown file> [libmlx5.so.1.0.0]:0`		`<unknown file> [libmlx5.so.1.0.0]:1133`
	`stl_vector.h:0`		`Geometry.h:0`
	`malloc.c:0`		Remaining MPI Time*

* ”Remaining MPI Time” = functions that take less than 5% of total MPI time

Round 2: MPI w/ Query Language



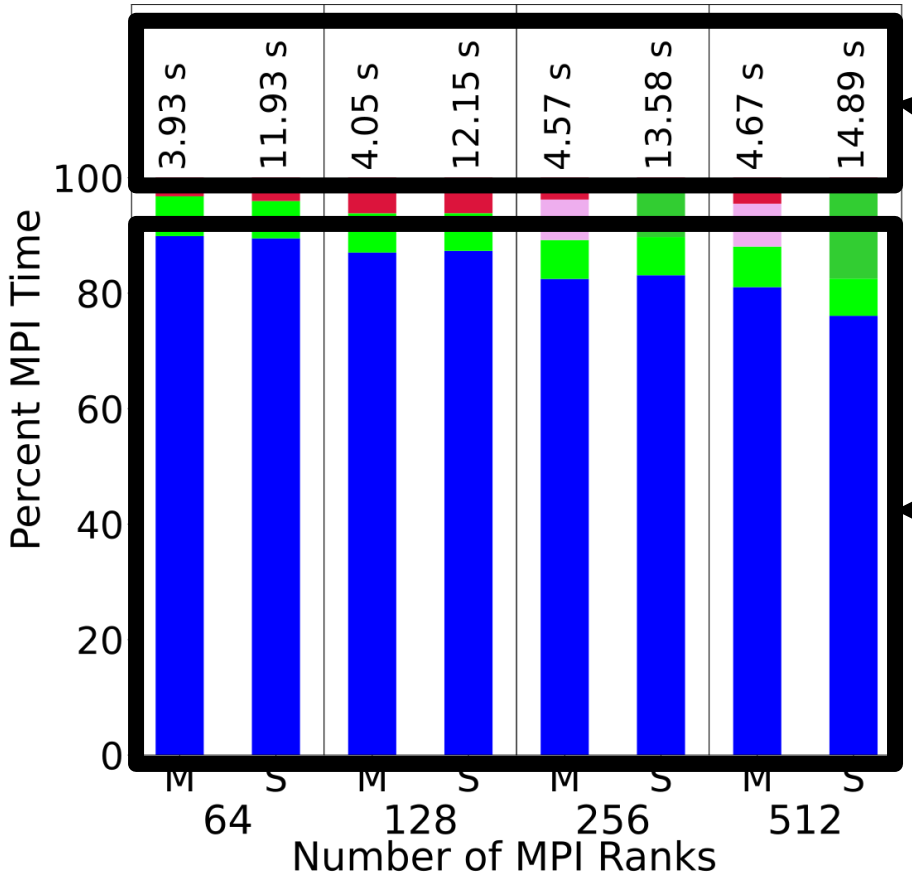
With Query Language:

We can point to Spectrum-MPI's use of **libmlx5** and **`pthread_spin_lock`** as likely causes of worse performance

	`pthread_spin_lock.c:26`		`memset.S:1133`
	`<unknown file> [libmlx5.so.1.0.0]:0`		`<unknown file> [libmlx5.so.1.0.0]:1133`
	`stl_vector.h:0`		`Geometry.h:0`
	`malloc.c:0`		Remaining MPI Time*

* "Remaining MPI Time" = functions that take less than 5% of total MPI time

Round 1: MPI w/o Query Language



M = MVAPICH S = Spectrum-MPI

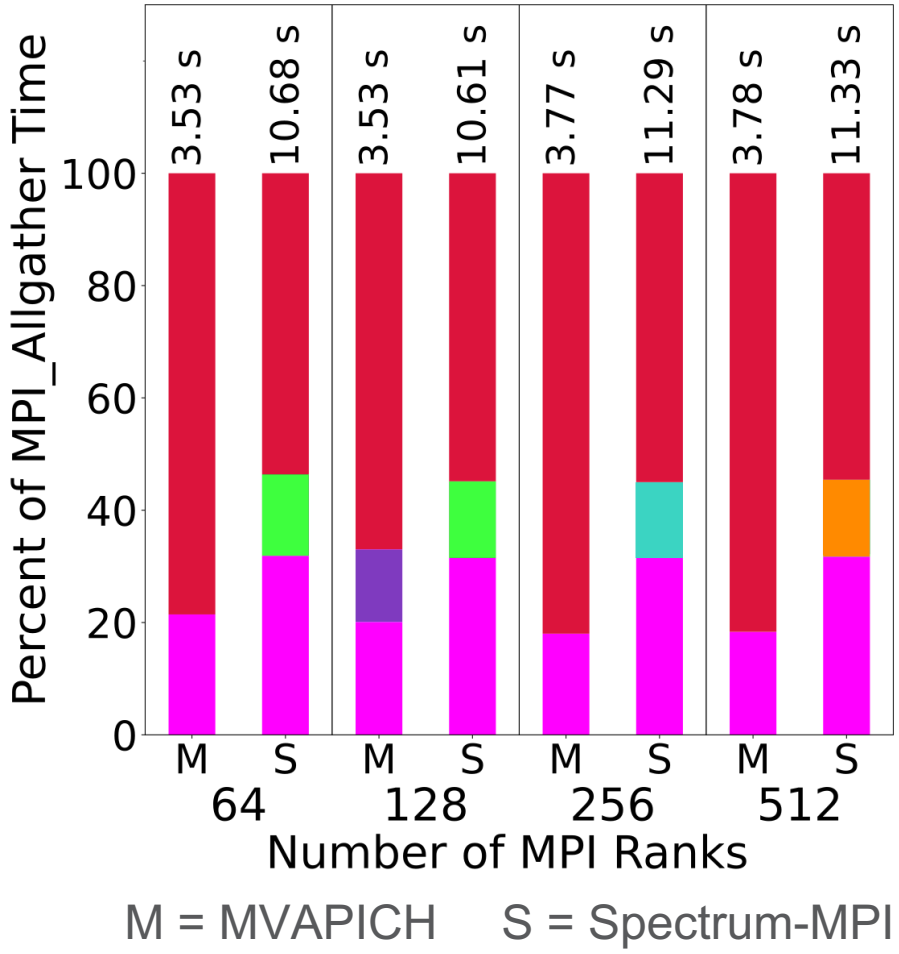
MVAPICH significantly outperforms Spectrum-MPI

`MPI_Allgather` takes more than 75% of total MPI time for all runs

	`MPI_Finalize`		`MPI_Allreduce`
	`MPI_Allgather`		`MPI_Waitall`
	Remaining MPI Time*		

* ”Remaining MPI Time” = functions that take less than 5% of total MPI time

Round 3: MPI_Allgather w/ Query Language



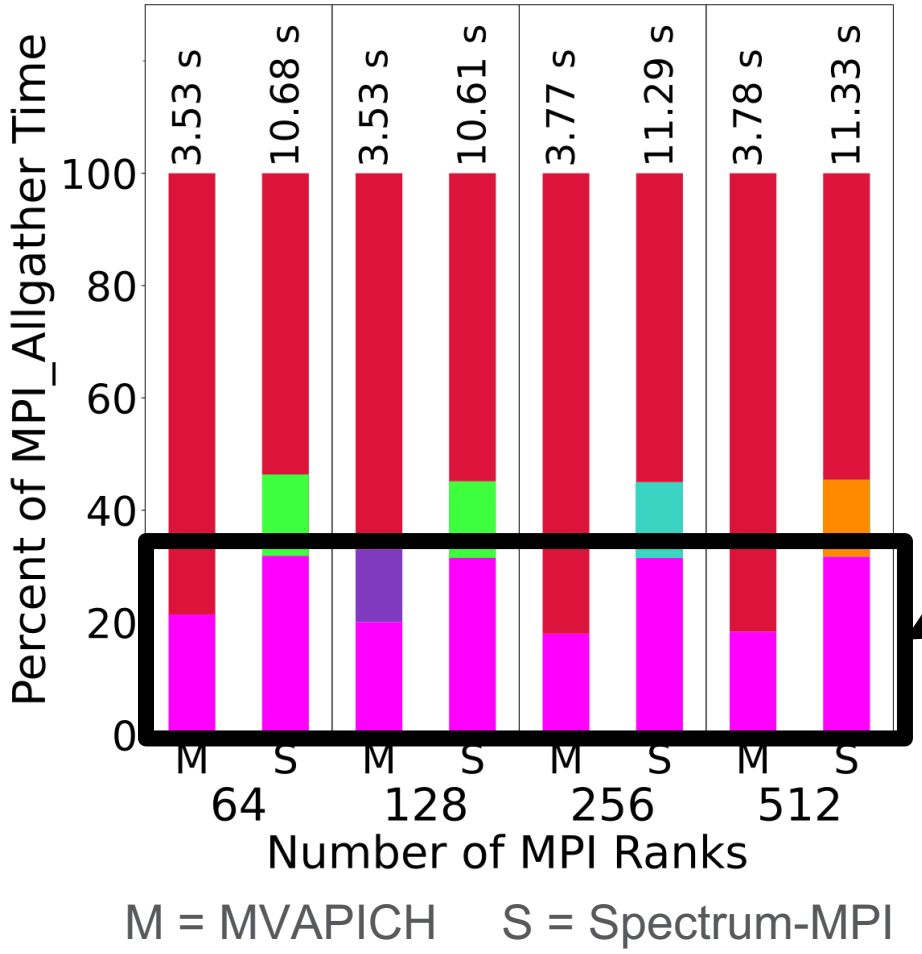
- Determine how much time was spent in the children of `MPI_Allgather`
- Hatchet with Query Language
  - Use non-Query Language filtering to remove `MPI_Allgather` nodes

```
[{“name”: “P?MPI_Allgather”}, “*”]
```

	`pthread_spin_lock.c:26`		`memset.S:1133`
	`<unknown file> [libmlx5.so.1.0.0]:0`		`<unknown file> [libmlx5.so.1.0.0]:1133`
	`stl_vector.h:0`		`Geometry.h:0`
	`malloc.c:0`		Remaining `MPI_Allgather` Time*

* ”Remaining MPI Time” = functions that take less than 5% of total `MPI_Allgather` time

Round 3: MPI_Allgather w/ Query Language

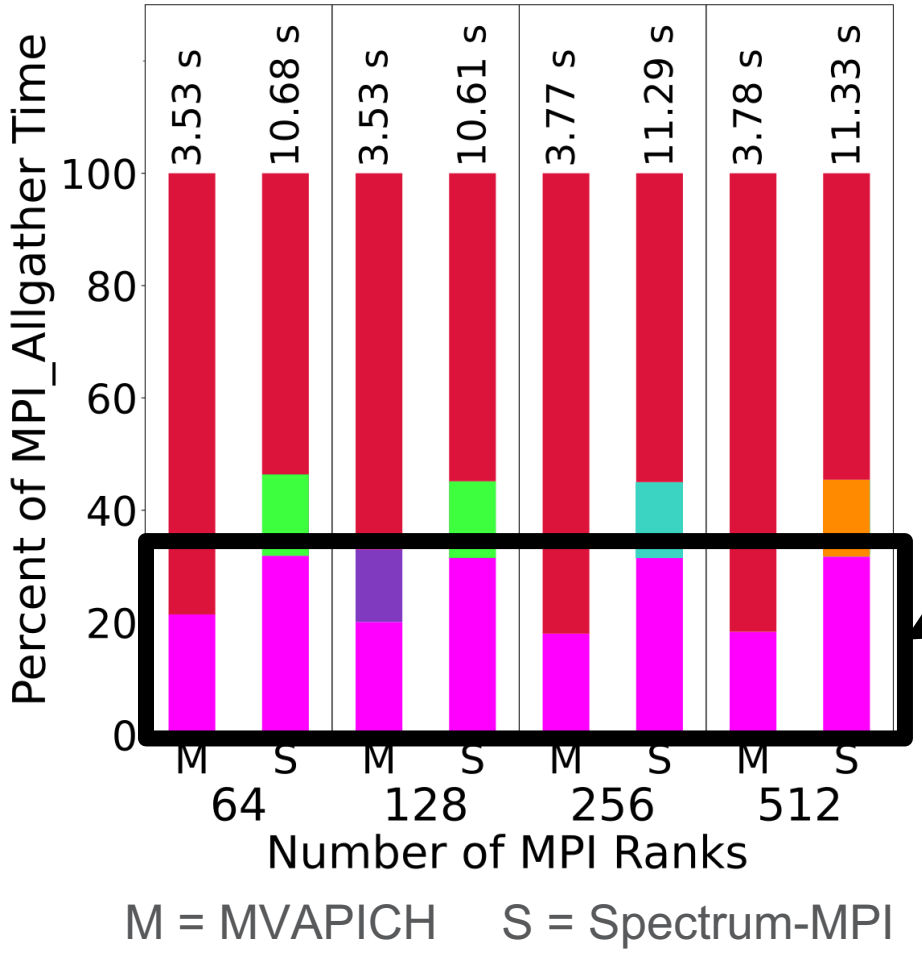


Spectrum-MPI uses `pthread_spin_lock` for a larger percentage of its time in `MPI_Allgather` than MVAPICH

	`pthread_spin_lock.c:26`		`memset.S:1133`
	`<unknown file> [libmlx5.so.1.0.0]:0`		`<unknown file> [libmlx5.so.1.0.0]:1133`
	`stl_vector.h:0`		`Geometry.h:0`
	`malloc.c:0`		Remaining `MPI_Allgather` Time*

* ”Remaining MPI Time” = functions that take less than 5% of total `MPI_Allgather` time

Round 3: MPI_Allgather w/ Query Language



With Query Language:

We can point to Spectrum-MPI's use of **`pthread_spin_lock`** as likely causes of worse performance **in `MPI_Allgather`**

	`pthread_spin_lock.c:26`		`memset.S:1133`
	`<unknown file> [libmlx5.so.1.0.0]:0`		`<unknown file> [libmlx5.so.1.0.0]:1133`
	`stl_vector.h:0`		`Geometry.h:0`
	`malloc.c:0`		Remaining `MPI_Allgather` Time*

* "Remaining MPI Time" = functions that take less than 5% of total `MPI_Allgather` time

What did the Query Language provide?

Spectrum-MPI uses **libmlx5** (Mellanox InfiniBand driver) much more than MVAPICH

Spectrum-MPI uses **`pthread_spin_lock`** for a larger percentage of its **MPI time** than MVAPICH

Spectrum-MPI uses **`pthread_spin_lock`** for a larger percentage of its **time in `MPI_Allgather`** than MVAPICH

What did the Query Language provide?

Spectrum-MPI uses **libmlx5** (Mellanox InfiniBand driver) much more than MVAPICH

Spectrum-MPI uses **`pthread_spin_lock`** for a larger percentage of its **MPI time** than MVAPICH

Spectrum-MPI uses **`pthread_spin_lock`** for a larger percentage of its **time in `MPI_Allgather`** than MVAPICH

We couldn't have found any of this using Hatchet without the Query Language!

Lessons Learned and Future Work

- Our work enables users to discover new insights into their applications' performance
  - Identify specific functions for further optimization
  - Attribute poor performance to specific functions
  - Reduce massively the size of call graphs
  - Enable easy and safe interaction between Hatchet and other tools (through the String-based Dialect)
- We will apply our Query Language and dialects to performance data from scientific applications

Want to try out the Query Language?

GitHub Repo

https://github.com/LLNL/hatchet-tutorial

BinderHub

https://mybinder.org/v2/gh/llnl/hatchet-tutorial/main