Performance Modeling, Analysis, and Tools

Abhinav Bhatele, Department of Computer Science

UNIVERSITY OF MARYLAND

Annoucements

- Assignment 1 is now posted online
  - Due on: Feb 28, 2024 11:59 pm
- More resources:
  - https://www.cs.umd.edu/~mmarsh/books/cmdline/cmdline.html
  - https://www.cs.umd.edu/~mmarsh/books/tools/tools.html
- Late submission policy: submit up to one late day for a 20% penalty
  - For any other exceptions, you need to ask as early as possible, not on the day of the deadline
  - This does NOT apply to extra credit assignments (including scribe notes) - No late submissions for these

Weak versus strong scaling

- Strong scaling: ***Fixed total*** problem size as we run on more processes
  - Sorting n numbers on 1 process, 2 processes, 4 processes, …
  - Problem size per process decreases with increase in number of processes
- Weak scaling: ***Fixed*** problem size ***per process*** but *increasing total* problem size as we run on more processes
  - Sorting n numbers on 1 process
  - 2n numbers on 2 processes
  - 4n numbers on 4 processes

Amdahl's law

- Speedup is limited by the serial portion of the code
  - Often referred to as the serial "bottleneck"
- Lets say only a fraction f of the code can be parallelized on p processes

$$\text{Speedup} = \frac{1}{(1-f) + f/p}$$

Amdahl's law

- Speedup is limited by the serial portion of the code
  - Often referred to as the serial "bottleneck"
- Lets say only a fraction *f* of the code can be parallelized on *p* processes

$$\text{Speedup} = \frac{1}{(1-f) + f/p}$$

Amdahl's law

- Speedup is limited by the serial portion of the code
    - Often referred to as the serial "bottleneck"
- Lets say only a fraction *f* of the code can be parallelized on *p* processes

$$\text{Speedup} = \frac{1}{(1-f) + f/p}$$

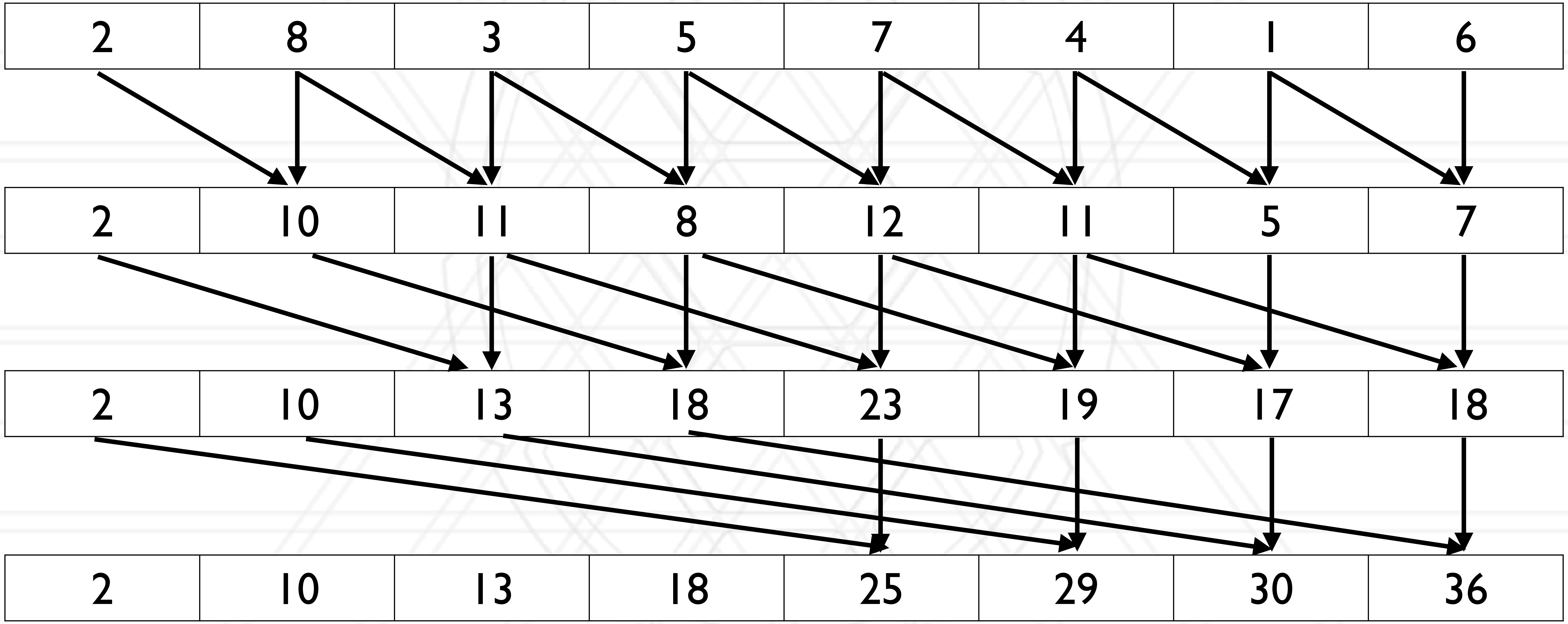
Performance analysis

- Parallel performance of a program might not be what the developer expects
- How do we find performance bottlenecks?
- Performance analysis is the process of studying the performance of a code
- Identify why performance might be slow
  - Serial performance
  - Serial bottlenecks when running in parallel
  - Communication overheads

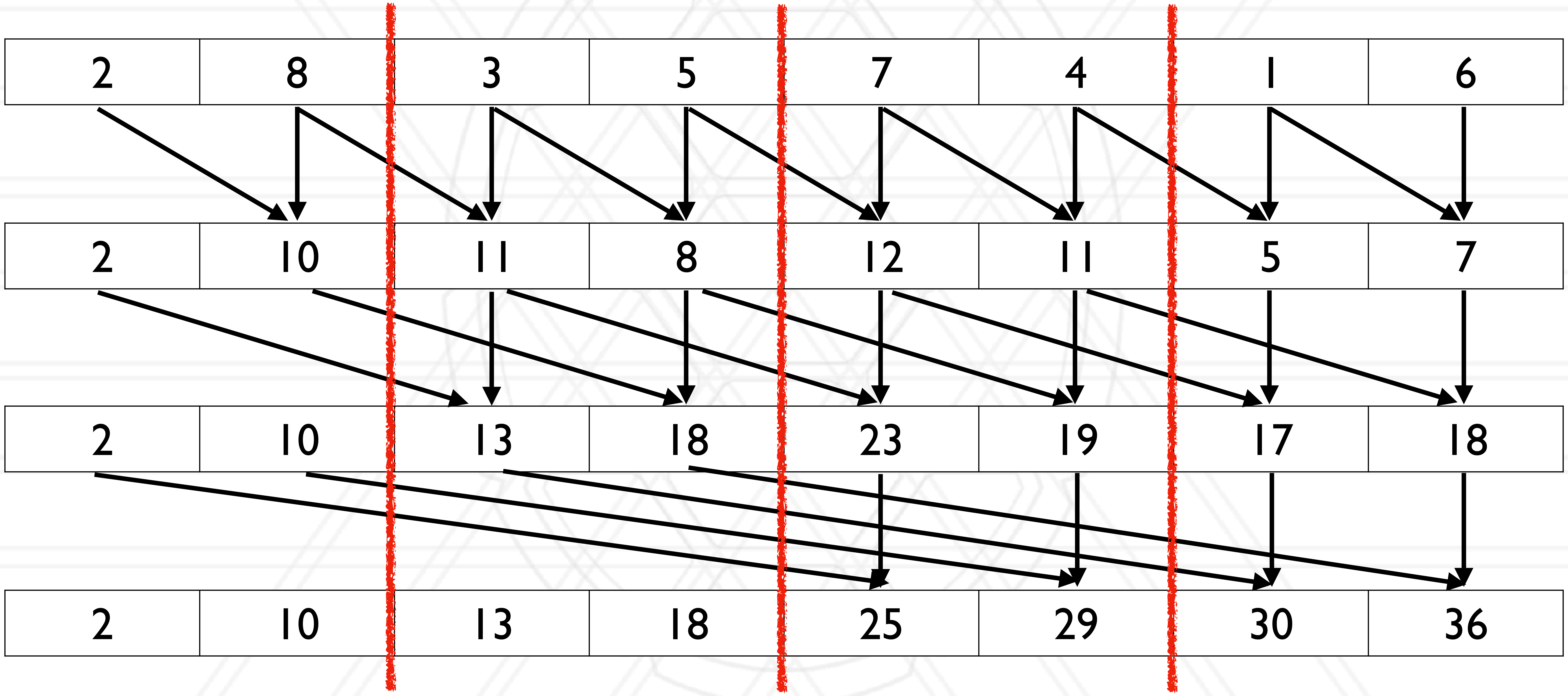
Different performance analysis methods

- Analytical techniques: use algebraic formulae
  - In terms of data size (n), number of processes (p)
- Time complexity analysis: big O notation
- Scalability analysis: Isoefficiency
- More detailed modeling of various operations such as communication
  - Analytical models: LogP, alpha-beta model
- Empirical performance analysis using profiling tools

Parallel prefix sum



Parallel prefix sum



Parallel prefix sum for *n* >> *p*

- Assign *n*/*p* elements (block) to each process
- Perform prefix sum on these blocks on each process locally
  - Number of calculations per processs:
- Then do the parallel algorithm using the computed partial prefix sums
  - Number of phases:
  - Total number of calculations per process:
  - Communication per process (one message containing one key/number):

Parallel prefix sum for *n* >> *p*

- Assign *n*/*p* elements (block) to each process
- Perform prefix sum on these blocks on each process locally
  - Number of calculations per processs: $\frac{n}{p}$
- Then do the parallel algorithm using the computed partial prefix sums
  - Number of phases:
  - Total number of calculations per process:
  - Communication per process (one message containing one key/number):

Parallel prefix sum for *n* >> *p*

- Assign *n*/*p* elements (block) to each process
- Perform prefix sum on these blocks on each process locally
  - Number of calculations per processs: $\frac{n}{p}$
- Then do the parallel algorithm using the computed partial prefix sums
  - Number of phases: $log(p)$
  - Total number of calculations per process:
  - Communication per process (one message containing one key/number):

Parallel prefix sum for *n* >> *p*

- Assign *n*/*p* elements (block) to each process
- Perform prefix sum on these blocks on each process locally
  - Number of calculations per processs: $\frac{n}{p}$
- Then do the parallel algorithm using the computed partial prefix sums
  - Number of phases: $log(p)$
  - Total number of calculations per process: $log(p) \times \frac{n}{p}$
  - Communication per process (one message containing one key/number):

Parallel prefix sum for *n* >> *p*

- Assign *n*/*p* elements (block) to each process
- Perform prefix sum on these blocks on each process locally
  - Number of calculations per processs: $\frac{n}{p}$
- Then do the parallel algorithm using the computed partial prefix sums
  - Number of phases: $log(p)$
  - Total number of calculations per process: $log(p) \times \frac{n}{p}$
  - Communication per process (one message containing one key/number): $log(p) \times 1 \times 1$

Modeling communication: LogP model

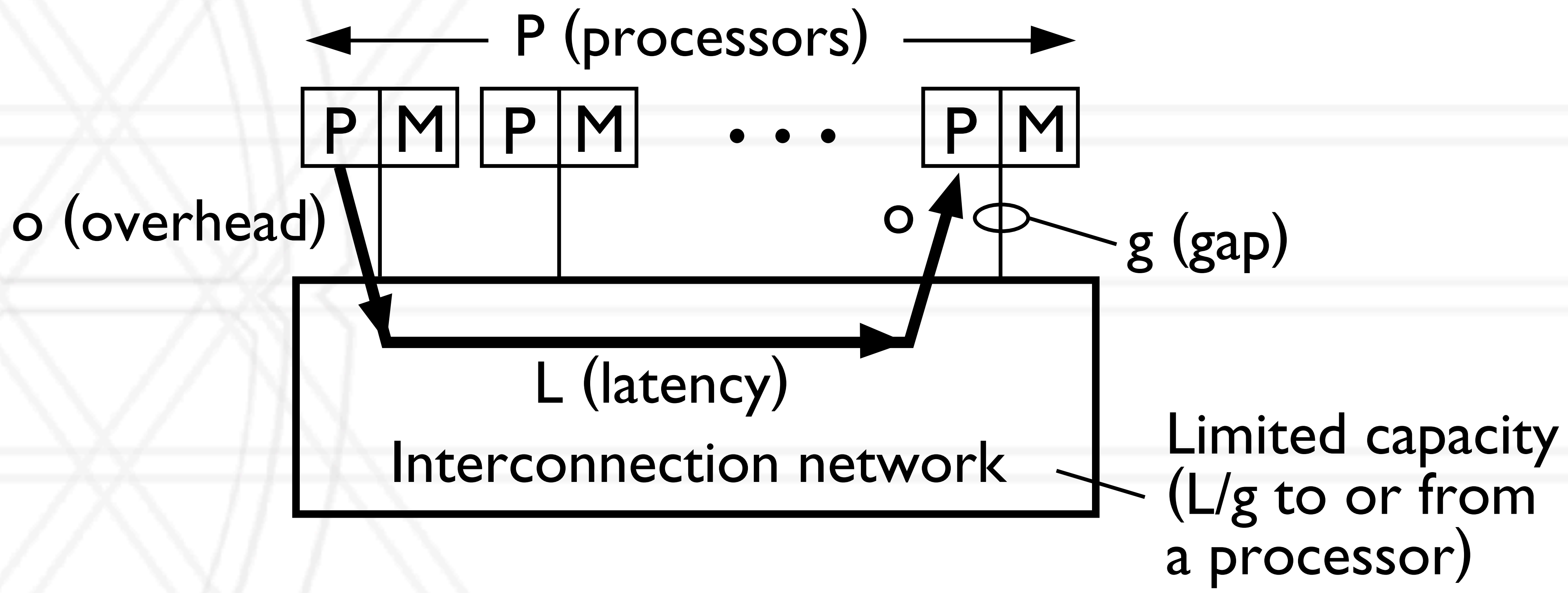
- Used for modeling communication on the inter-node network

L: latency or delay

o: overhead (processor busy in communication)

g: gap (required between successive sends/ receives)

P: number of processors / processes



g is the inverse of bandwidth

1/g = bandwidth

alpha + n * beta model

- Another model for communication

$$T_{\text{comm}} = \alpha + n \times \beta$$

α: latency

n: size of message

1/β: bandwidth

Isoefficiency

- Relationship between problem size and number of processes to maintain a certain level of efficiency
- At what rate should we increase problem size with respect to number of processes to keep efficiency constant (iso-efficiency)

Speedup and efficiency

- Speedup: Ratio of execution time on one process to that on p processes

$$\text{Speedup} = \frac{t_1}{t_p}$$

- Efficiency: Speedup per process

$$\text{Efficiency} = \frac{t_1}{t_p \times p}$$

Efficiency in terms of overhead

- Total time spent in all processes = (useful) computation + overhead (extra computation + communication + idle time + other overheads)

$$p \times t_p = t_1 + t_o$$

$$\text{Efficiency} = \frac{t_1}{t_p \times p} = \frac{t_1}{t_1 + t_o} = \frac{1}{1 + \frac{t_o}{t_1}}$$

Isoefficiency function

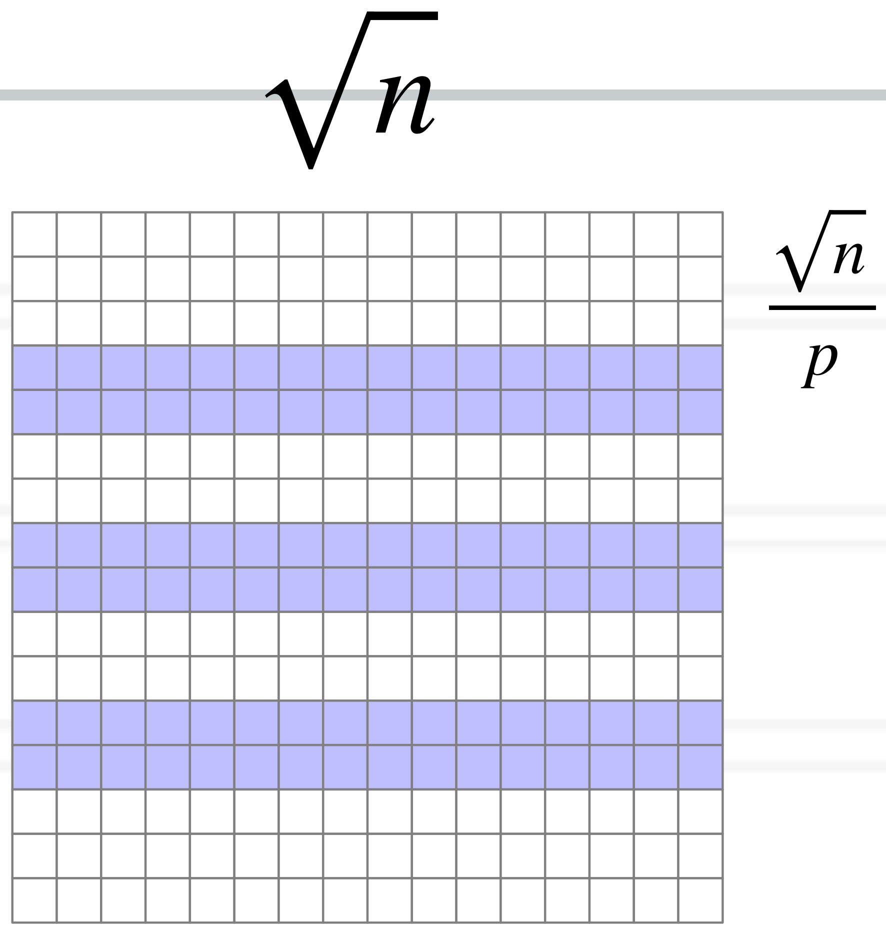
$$\text{Efficiency} = \frac{1}{1 + \frac{t_o}{t_1}}$$

- Efficiency is constant if t_o / t_1 is constant (K)

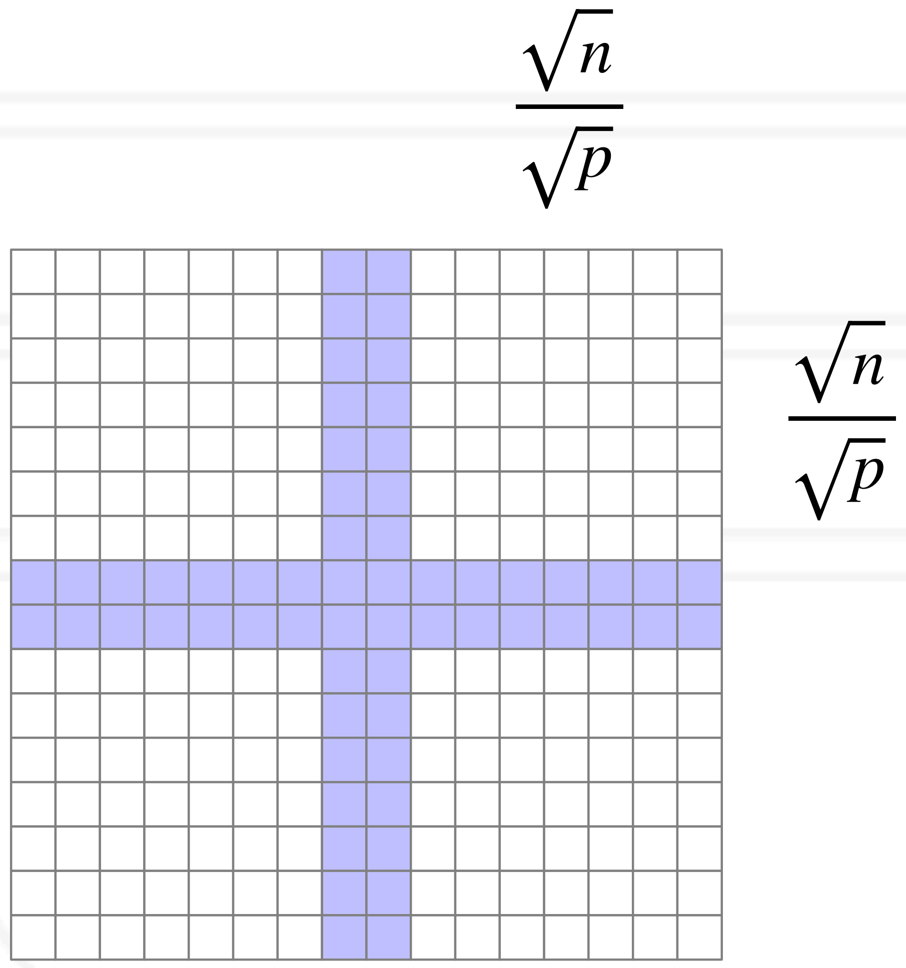
$$t_o = K \times t_1$$

Isoefficiency analysis

- 1D decomposition:
  - Computation:
  - Communication:

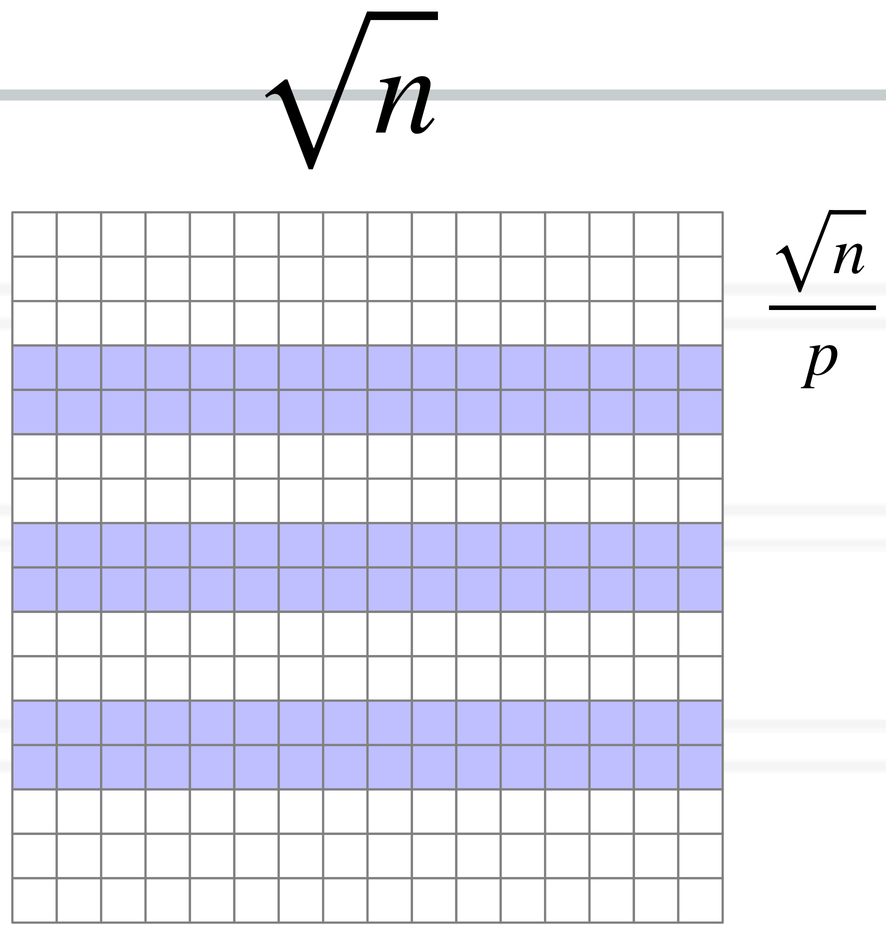


- 2D decomposition:
  - Computation:
  - Communication

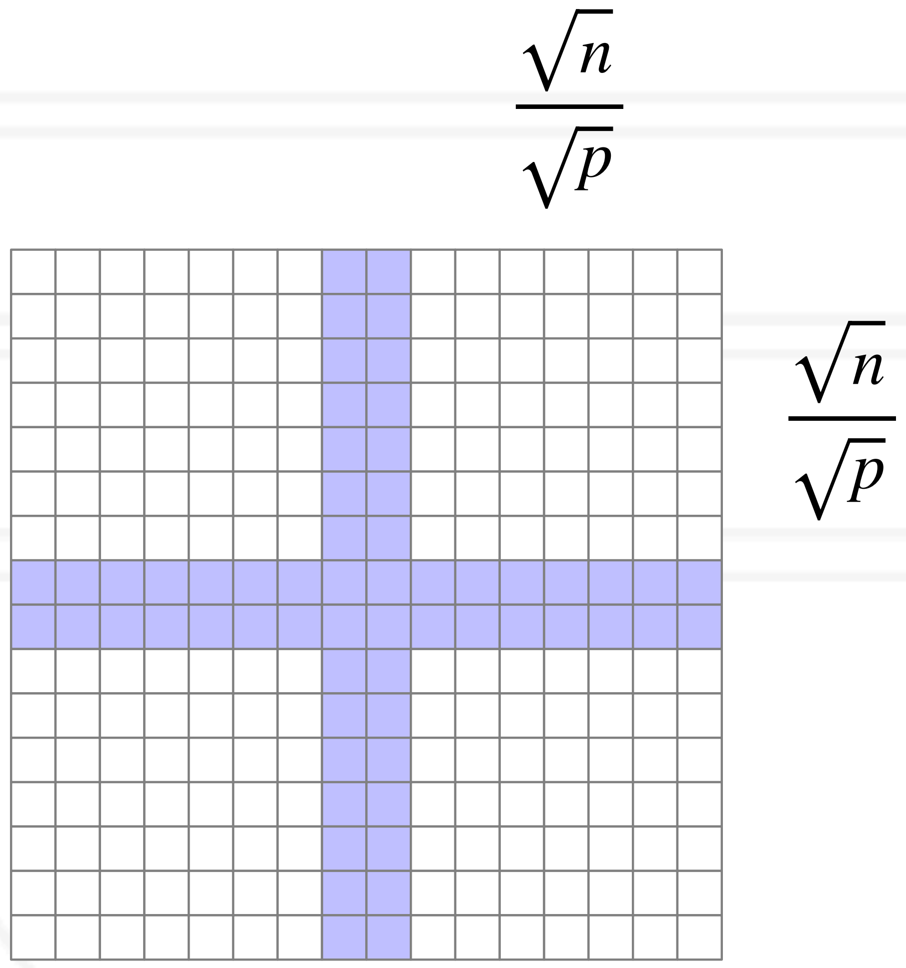


Isoefficiency analysis

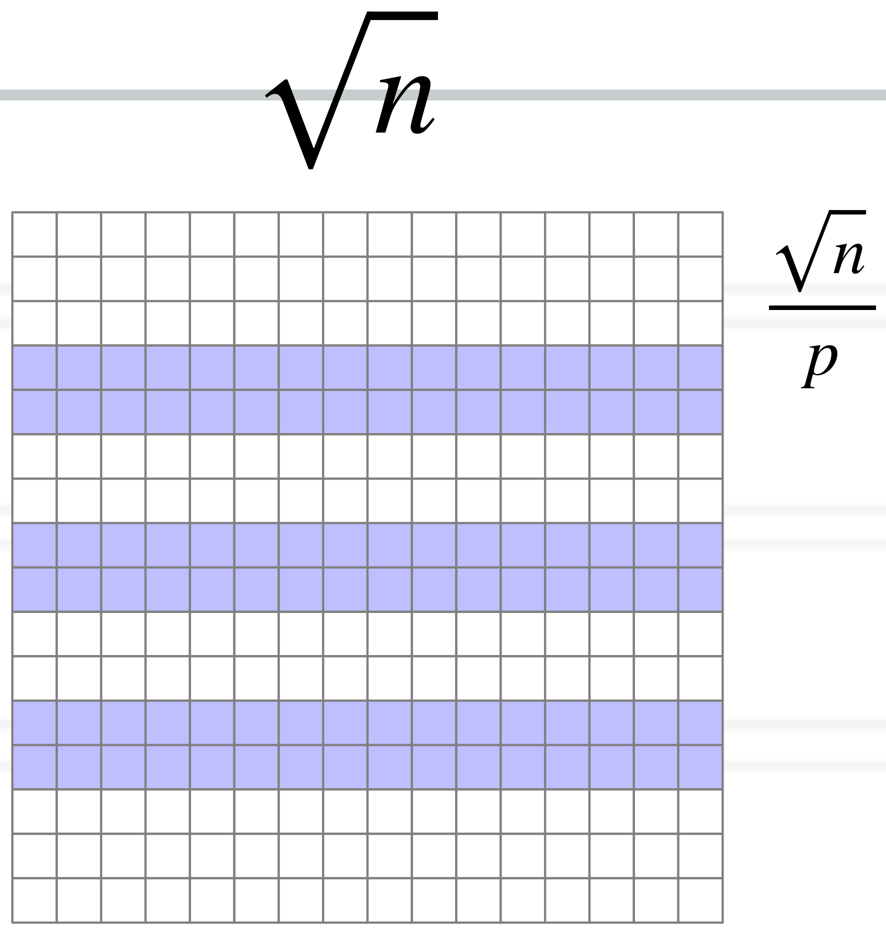
- 1D decomposition:
  - Computation: $\sqrt{n} \times \frac{\sqrt{n}}{p} = \frac{n}{p}$
  - Communication:



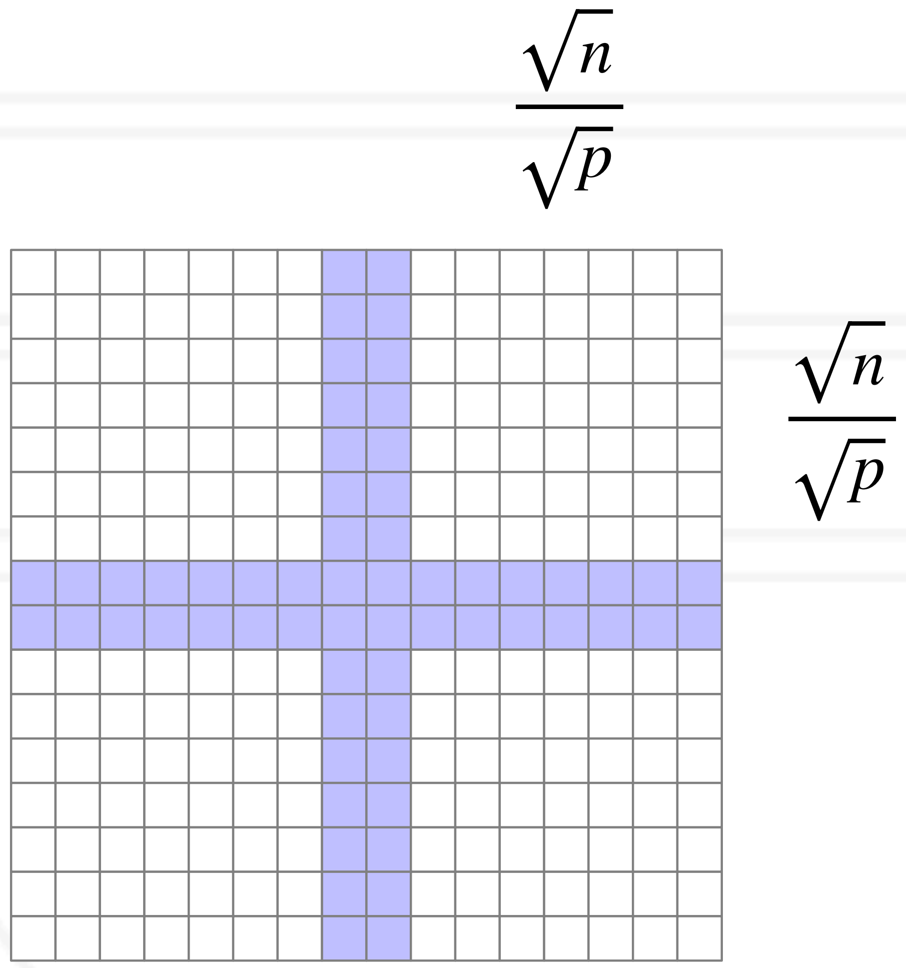
- 2D decomposition:
  - Computation:
  - Communication



Isoefficiency analysis



- 1D decomposition:
  - Computation: $\sqrt{n} \times \frac{\sqrt{n}}{p} = \frac{n}{p}$
  - Communication: $2 \times \sqrt{n}$

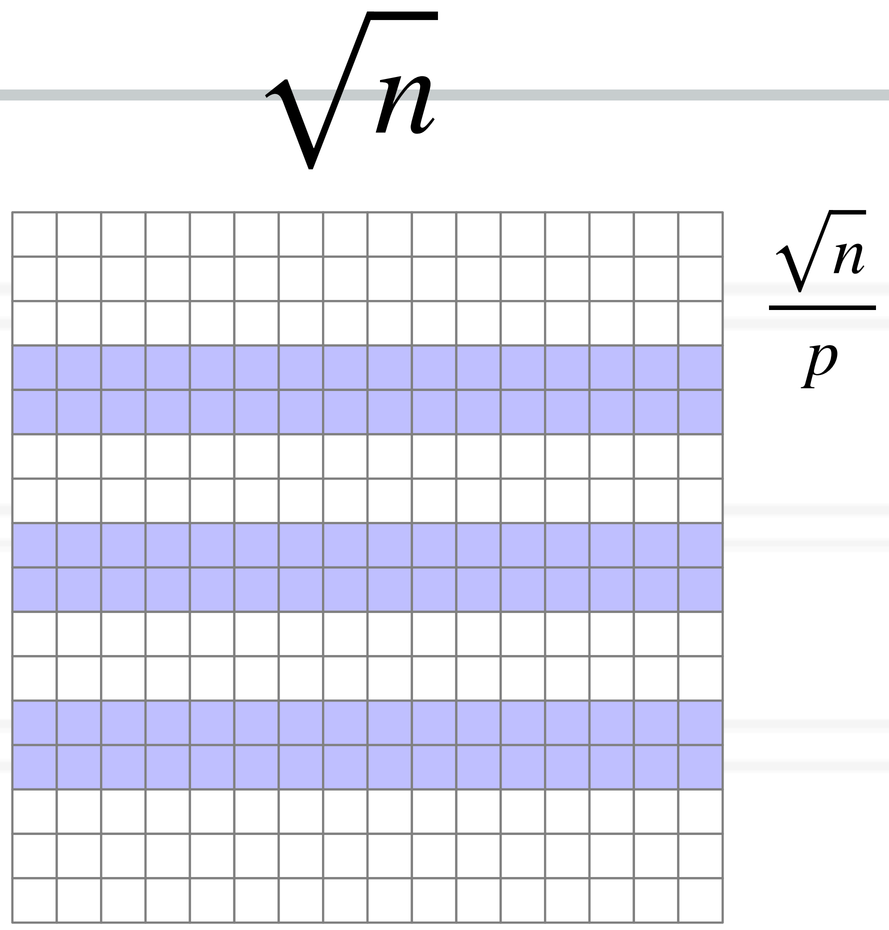


- 2D decomposition:
  - Computation:
  - Communication

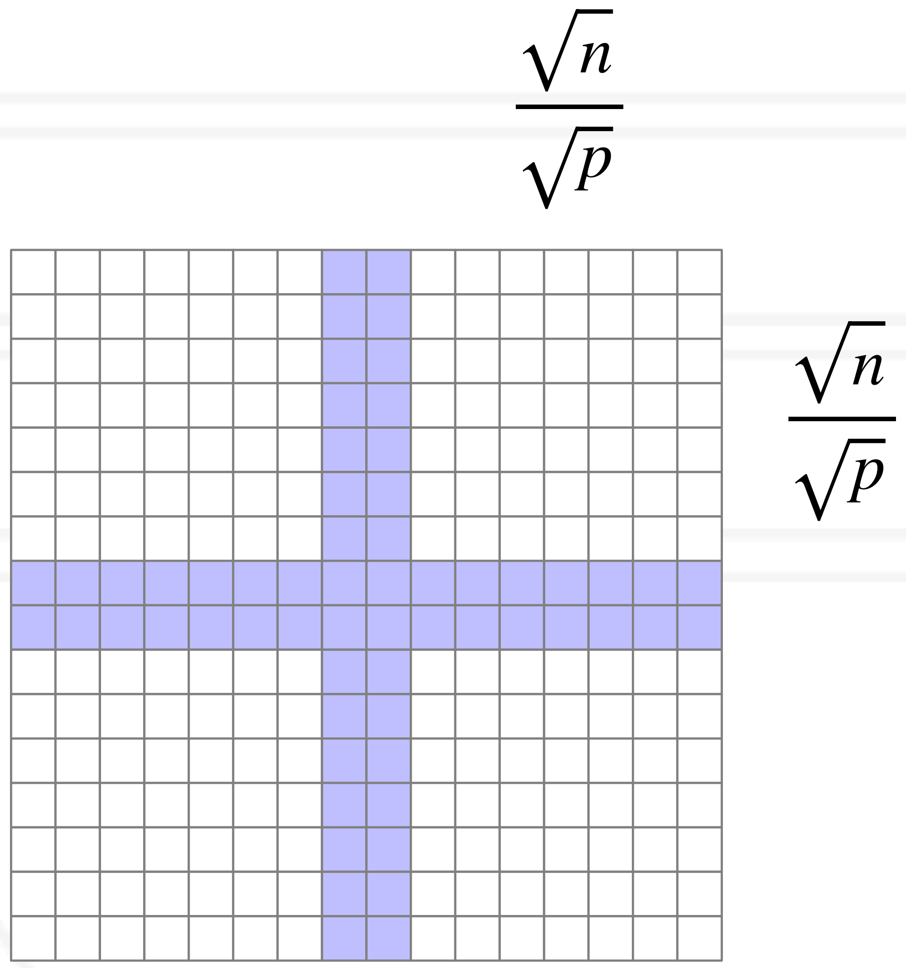
Isoefficiency analysis

- 1D decomposition:
  - Computation: $\sqrt{n} \times \frac{\sqrt{n}}{p} = \frac{n}{p}$
  - Communication: $2 \times \sqrt{n}$

$$\frac{t_o}{t_1} = \frac{2 \times \sqrt{n}}{\frac{n}{p}} = \frac{2 \times p}{\sqrt{n}}$$



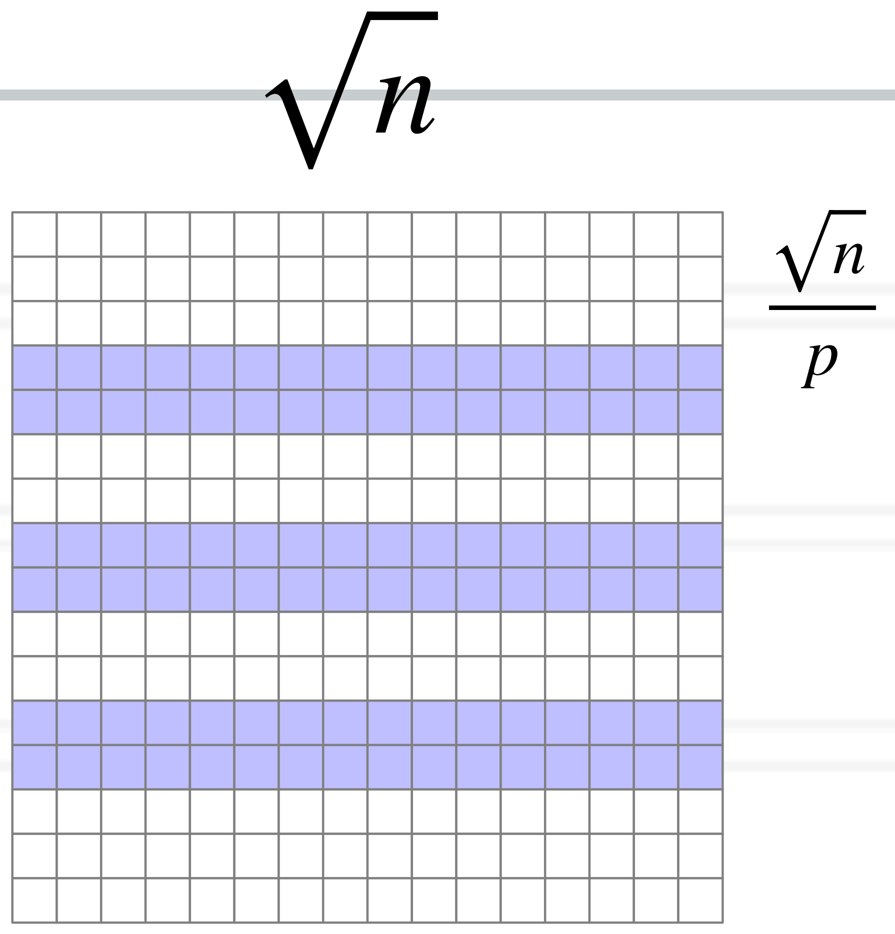
- 2D decomposition:
  - Computation:
  - Communication



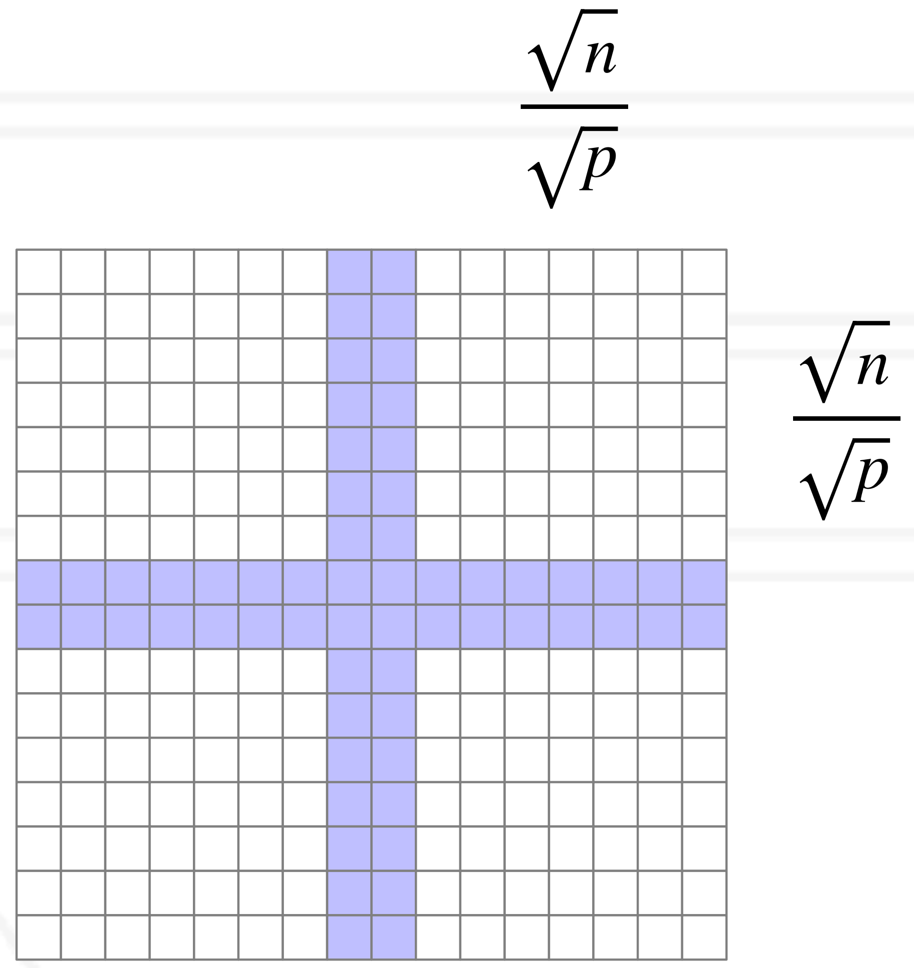
Isoefficiency analysis

- 1D decomposition:
  - Computation: $\sqrt{n} \times \frac{\sqrt{n}}{p} = \frac{n}{p}$
  - Communication: $2 \times \sqrt{n}$

$$\frac{t_o}{t_1} = \frac{2 \times \sqrt{n}}{\frac{n}{p}} = \frac{2 \times p}{\sqrt{n}}$$



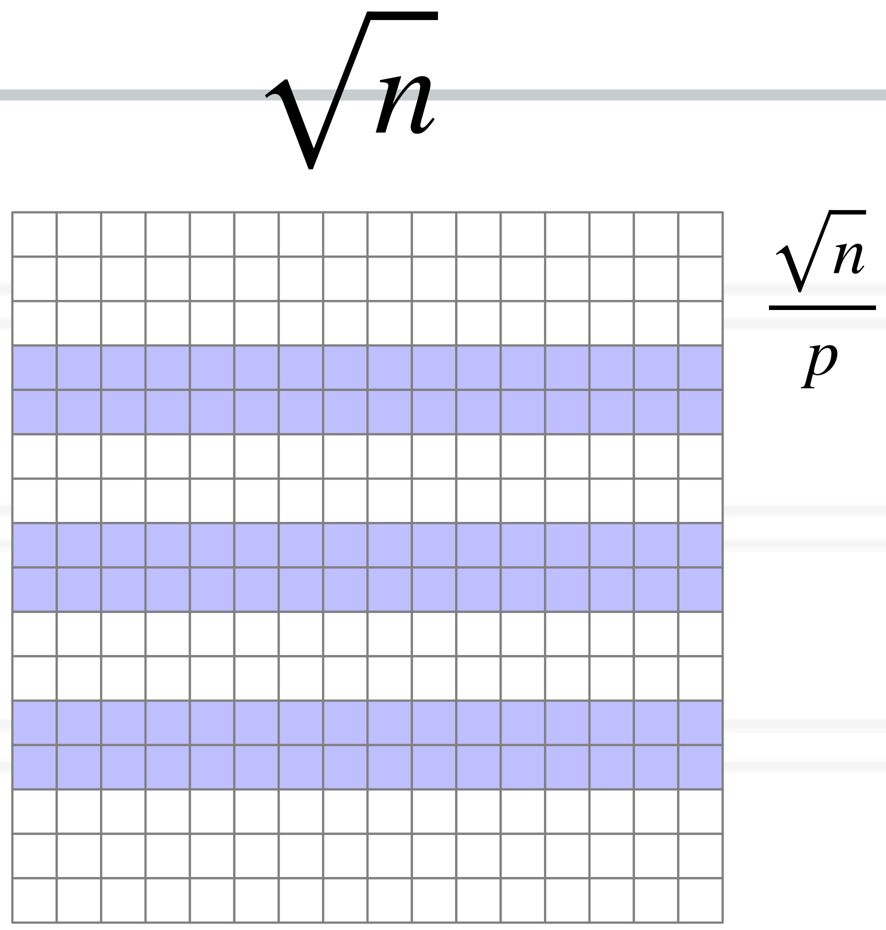
- 2D decomposition:
  - Computation: $\frac{\sqrt{n}}{\sqrt{p}} \times \frac{\sqrt{n}}{\sqrt{p}} = \frac{n}{p}$
  - Communication



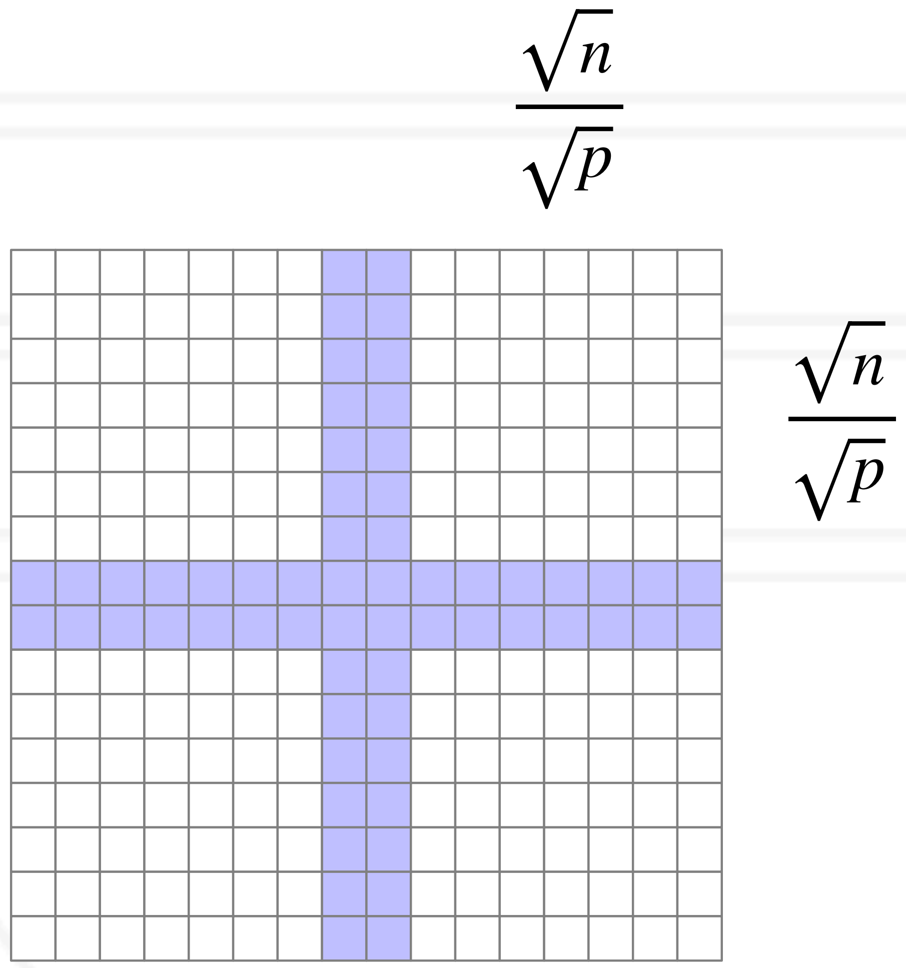
Isoefficiency analysis

- 1D decomposition:
  - Computation: $\sqrt{n} \times \frac{\sqrt{n}}{p} = \frac{n}{p}$
  - Communication: $2 \times \sqrt{n}$

$$\frac{t_o}{t_1} = \frac{2 \times \sqrt{n}}{\frac{n}{p}} = \frac{2 \times p}{\sqrt{n}}$$



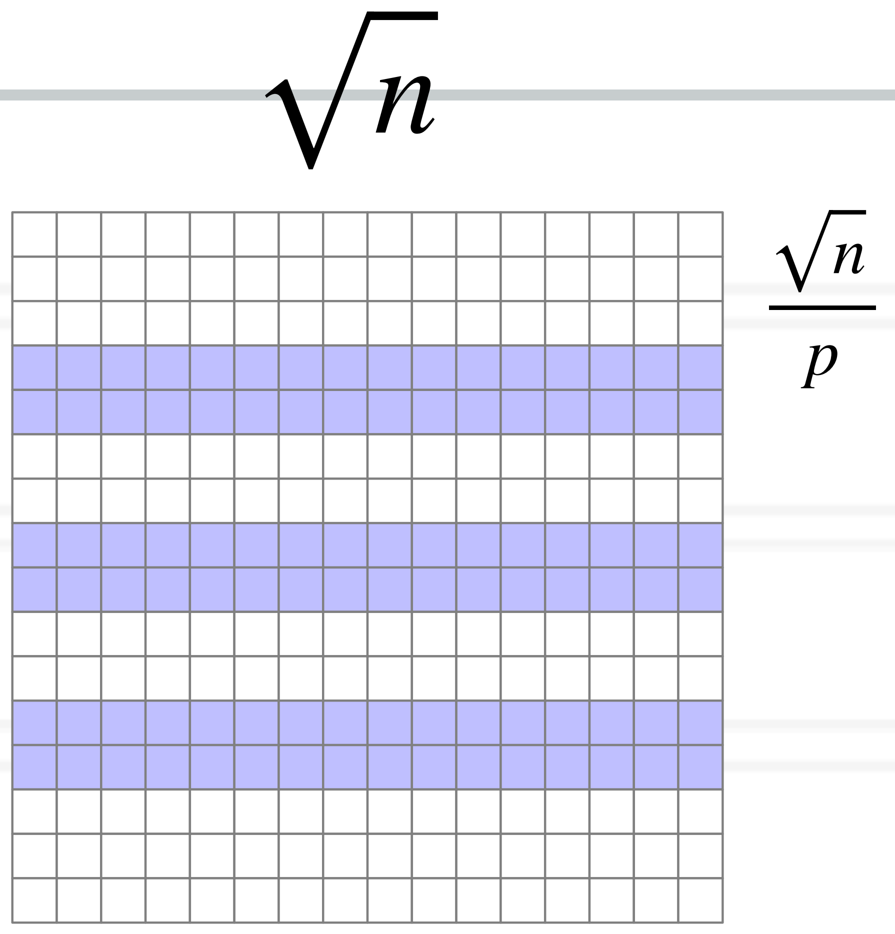
- 2D decomposition:
  - Computation: $\frac{\sqrt{n}}{\sqrt{p}} \times \frac{\sqrt{n}}{\sqrt{p}} = \frac{n}{p}$
  - Communication $4 \times \frac{\sqrt{n}}{\sqrt{p}}$



Isoefficiency analysis

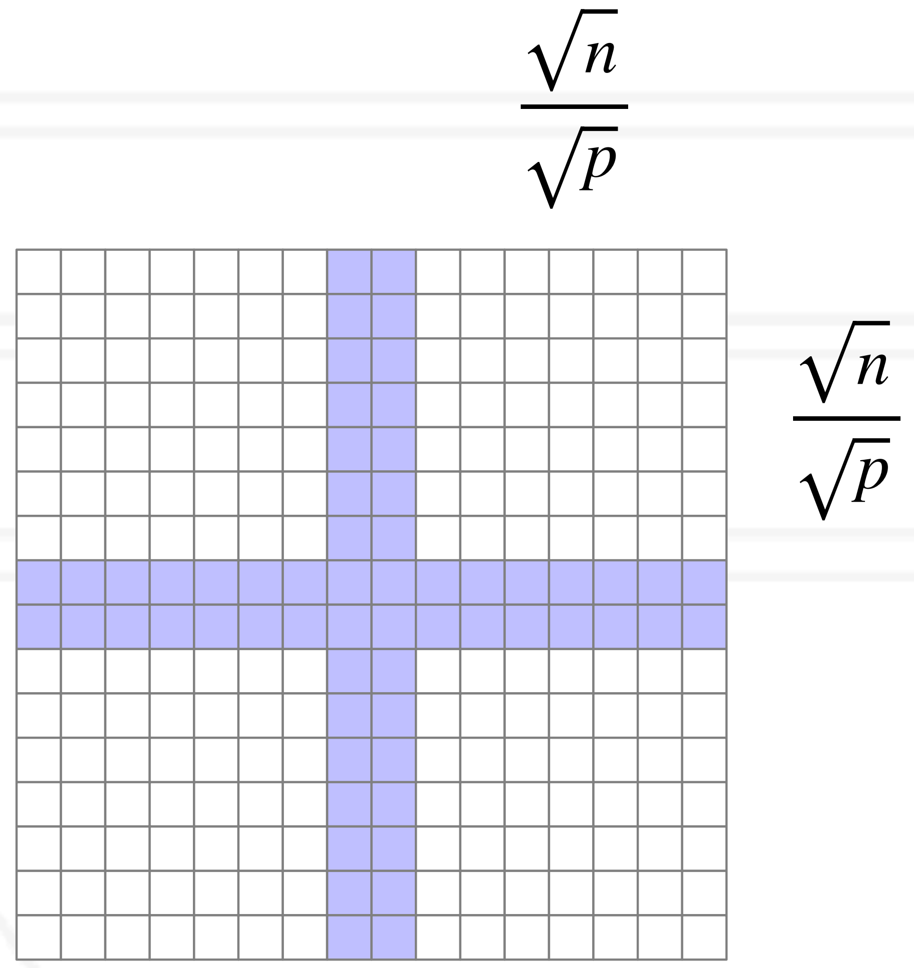
- 1D decomposition:
  - Computation: $\sqrt{n} \times \frac{\sqrt{n}}{p} = \frac{n}{p}$
  - Communication: $2 \times \sqrt{n}$

$$\frac{t_o}{t_1} = \frac{2 \times \sqrt{n}}{\frac{n}{p}} = \frac{2 \times p}{\sqrt{n}}$$



- 2D decomposition:
  - Computation: $\frac{\sqrt{n}}{\sqrt{p}} \times \frac{\sqrt{n}}{\sqrt{p}} = \frac{n}{p}$
  - Communication $4 \times \frac{\sqrt{n}}{\sqrt{p}}$

$$\frac{t_o}{t_1} = \frac{4 \times \frac{\sqrt{n}}{\sqrt{p}}}{\frac{n}{p}} = \frac{4 \times \sqrt{p}}{\sqrt{n}}$$

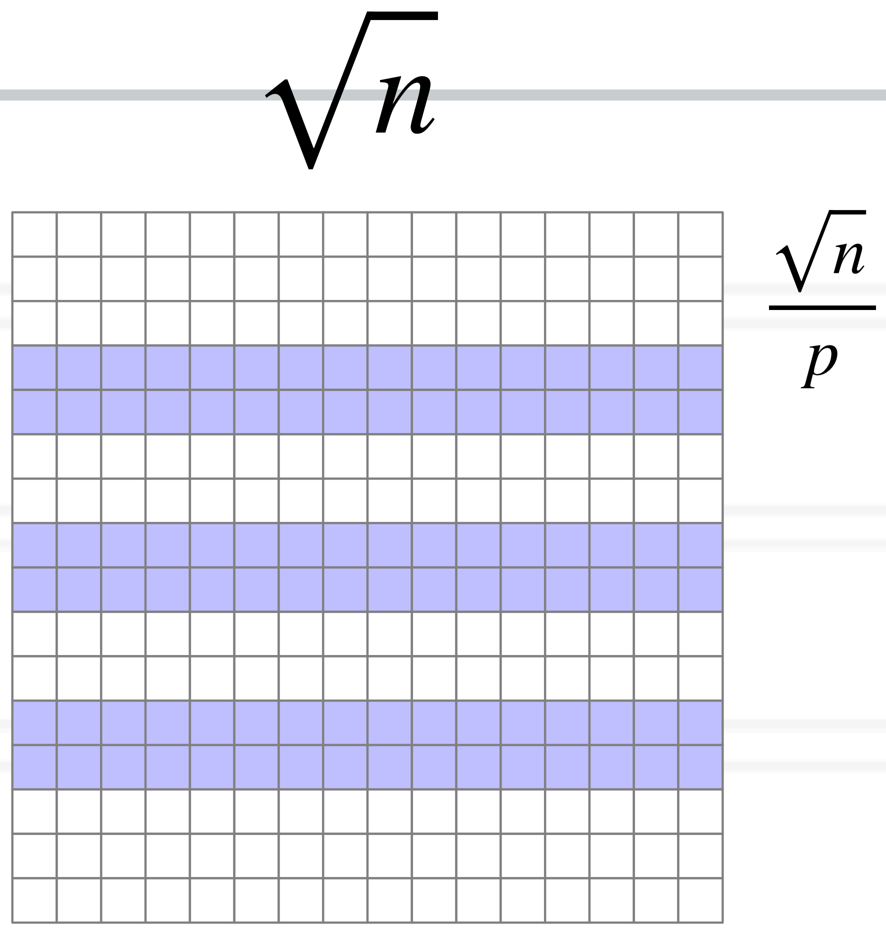


Isoefficiency analysis

> We only consider communication for t_o

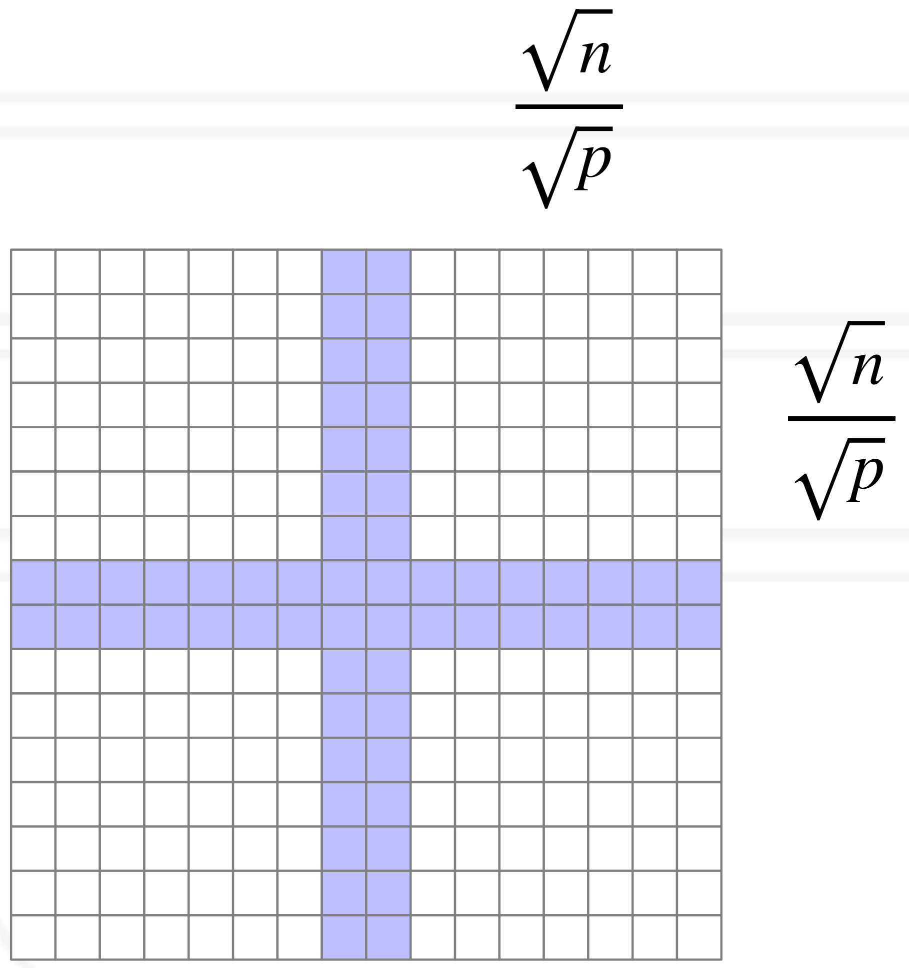
- 1D decomposition:
  - Computation: $\sqrt{n} \times \frac{\sqrt{n}}{p} = \frac{n}{p}$
  - Communication: $2 \times \sqrt{n}$

$$\frac{t_o}{t_1} = \frac{2 \times \sqrt{n}}{\frac{n}{p}} = \frac{2 \times p}{\sqrt{n}}$$



- 2D decomposition:
  - Computation: $\frac{\sqrt{n}}{\sqrt{p}} \times \frac{\sqrt{n}}{\sqrt{p}} = \frac{n}{p}$
  - Communication: $4 \times \frac{\sqrt{n}}{\sqrt{p}}$

$$\frac{t_o}{t_1} = \frac{4 \times \frac{\sqrt{n}}{\sqrt{p}}}{\frac{n}{p}} = \frac{4 \times \sqrt{p}}{\sqrt{n}}$$



Empirical performance analysis

- Two parts to doing empirical performance analysis
  - measurement: gather/collect performance data from a program execution
  - analysis/visualization: analyze the measurements to identify performance issues
- Simplest tool: adding timers in the code manually and using print statements

Using timers

```
double start, end;
double phase1, phase2, phase3;

start = MPI_Wtime();
 ... phase1 code ...
end = MPI_Wtime();
phase1 = end - start;

start = MPI_Wtime();
 ... phase2 ...
end = MPI_Wtime();
phase2 = end - start;

start = MPI_Wtime();
 ... phase3 ...
end = MPI_Wtime();
phase3 = end - start;
```

Using timers

```
double start, end;
double phase1, phase2, phase3;

start = MPI_Wtime();
 ... phase1 code ...
end = MPI_Wtime();
phase1 = end - start;

start = MPI_Wtime();
 ... phase2 ...
end = MPI_Wtime();
phase2 = end - start;

start = MPI_Wtime();
 ... phase3 ...
end = MPI_Wtime();
phase3 = end - start;
```

Phase 1 took 2.45 s

Phase 2 took 11.79 s

Phase 3 took 4.37 s

Performance tools

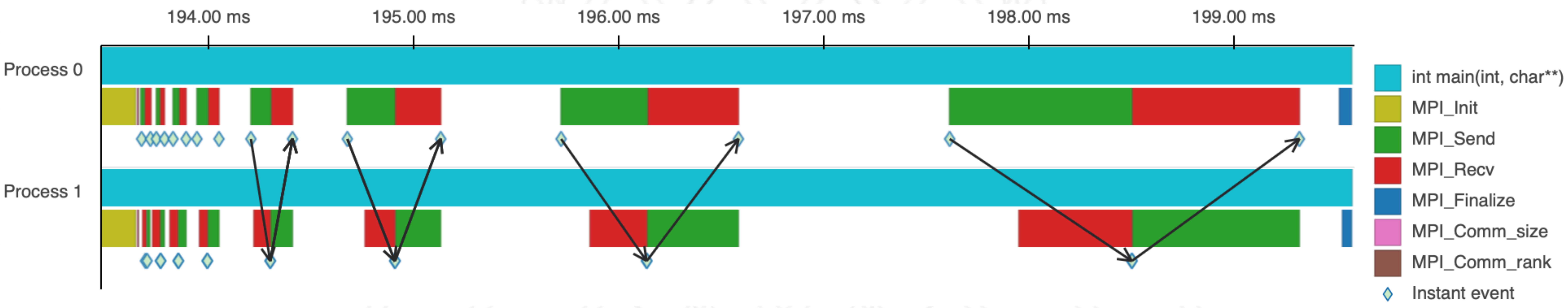
- Tracing tools
  - Capture entire execution trace, typically via instrumentation
- Profiling tools
  - Provide aggregated information
  - Typically use statistical sampling
- Many tools can do both

Metrics recorded

- Counts of function invocations
- Time spent in each function/code region
- Number of bytes sent (in case of MPI messages)
- Hardware counters such as floating point operations, cache misses, etc.
- To fix performance problems — we need to connect metrics to source code

Tracing tools

- Record all the events in the program with enter/leave timestamps
- Events: user functions, MPI and other library routines, etc.



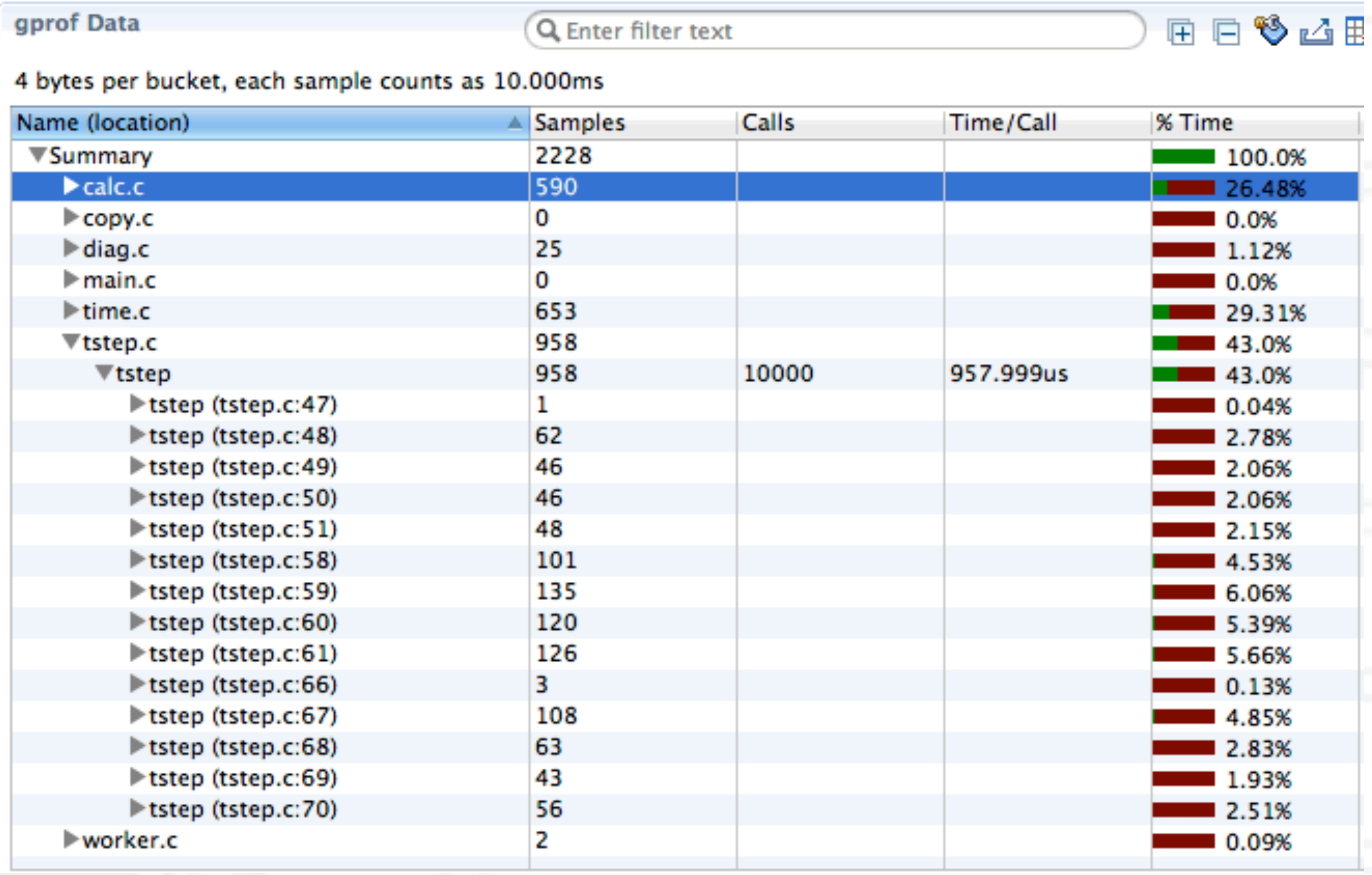
Timeline visualization of a 2-process execution trace

Examples of tracing tools

- VampirTrace
- Score-P
- TAU
- Projections
- HPCToolkit

Profiling tools

- Ignore the specific times at which events occurred
- Provide aggregate information about time spent in different functions/code regions
- Examples:
  - gprof, perf
  - mpiP
  - HPCToolkit, caliper
- Python tools: cprofile, pyinstrument, scalene


gprof Data

Enter filter text

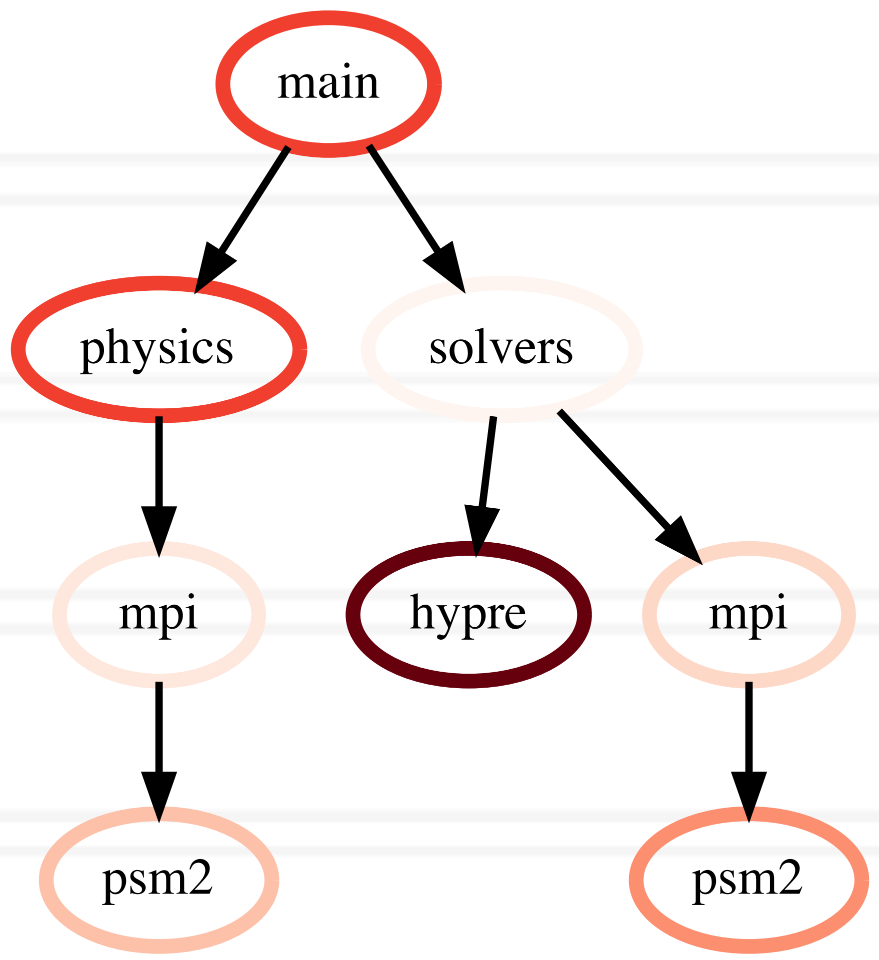
4 bytes per bucket, each sample counts as 10.000ms

Name (location)	Samples	Calls	Time/Call	% Time
Summary	2228			100.0%
calc.c	590			26.48%
copy.c	0			0.0%
diag.c	25			1.12%
main.c	0			0.0%
time.c	653			29.31%
tstep.c	958			43.0%
tstep	958	10000	957.999us	43.0%
tstep (tstep.c:47)	1			0.04%
tstep (tstep.c:48)	62			2.78%
tstep (tstep.c:49)	46			2.06%
tstep (tstep.c:50)	46			2.06%
tstep (tstep.c:51)	48			2.15%
tstep (tstep.c:58)	101			4.53%
tstep (tstep.c:59)	135			6.06%
tstep (tstep.c:60)	120			5.39%
tstep (tstep.c:61)	126			5.66%
tstep (tstep.c:66)	3			0.13%
tstep (tstep.c:67)	108			4.85%
tstep (tstep.c:68)	63			2.83%
tstep (tstep.c:69)	43			1.93%
tstep (tstep.c:70)	56			2.51%
worker.c	2			0.09%

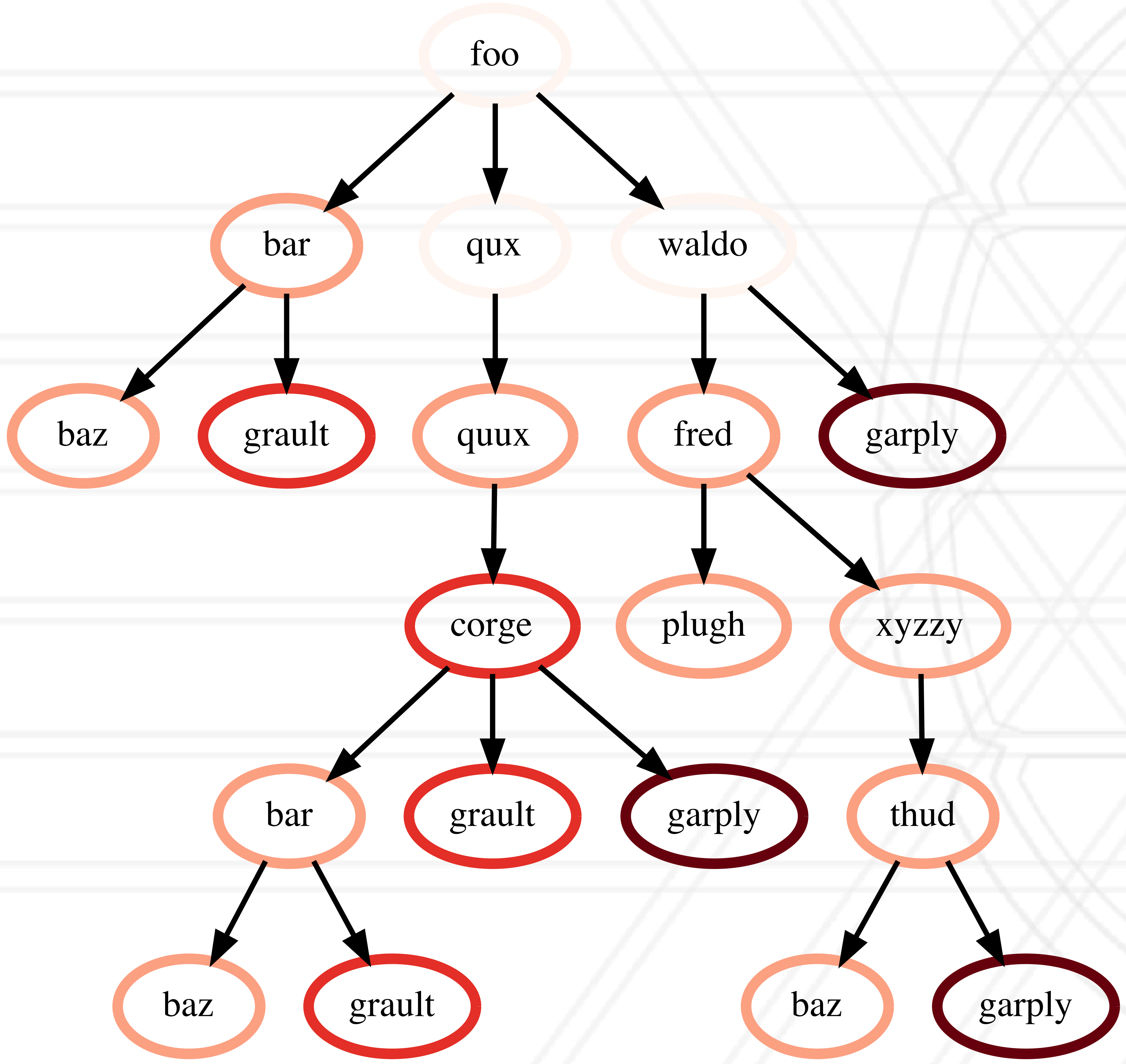
gprof data in hpctView

Calling contexts, trees, and graphs

- Calling context or call path: Sequence of function invocations leading to the current sample (statement in code)
- Calling context tree (CCT): dynamic prefix tree of all call paths in an execution
- Call graph: obtained by merging nodes in a CCT with the same name into a single node but keeping caller-callee relationships as edges

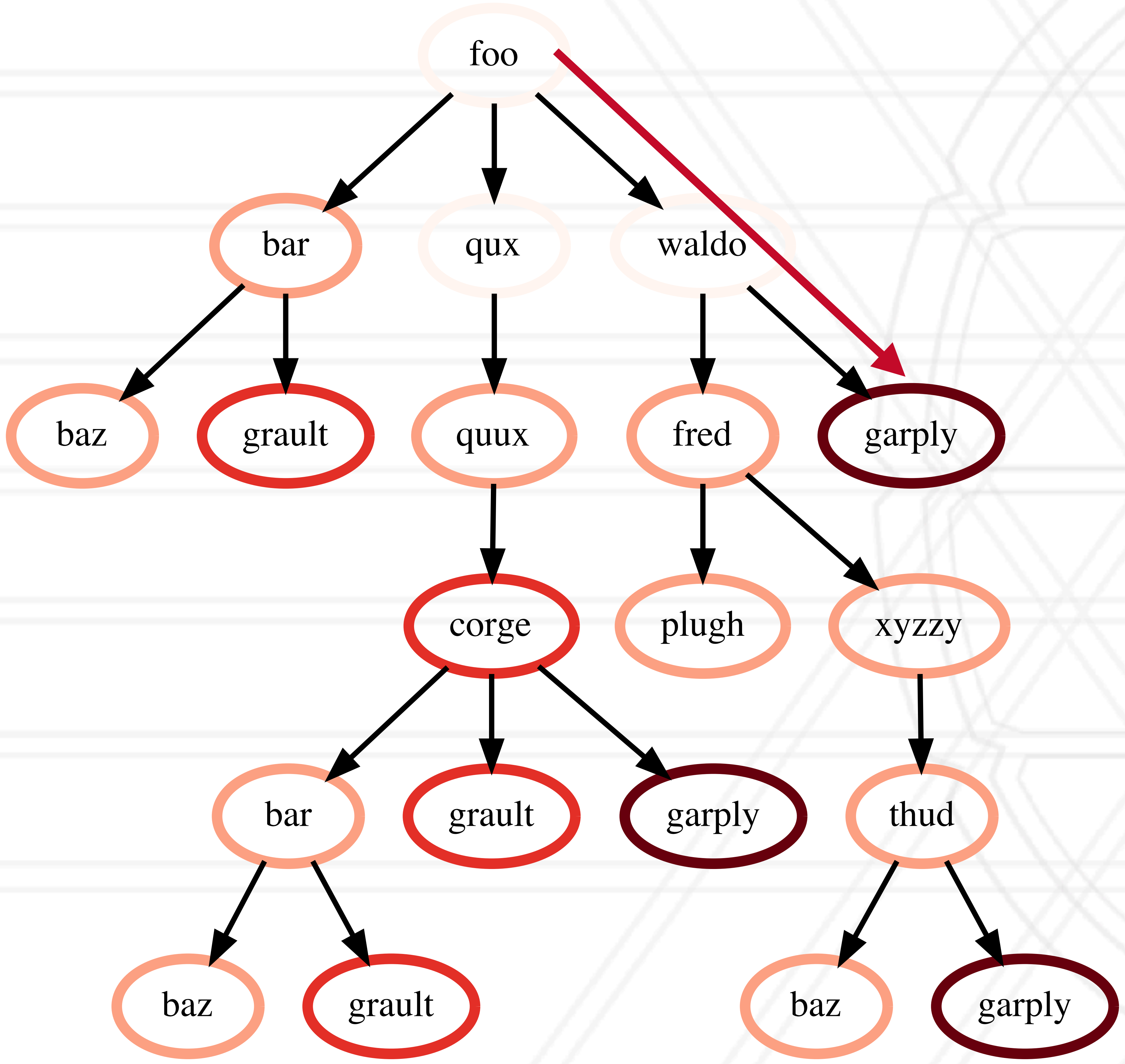


Calling context trees, call graphs, …



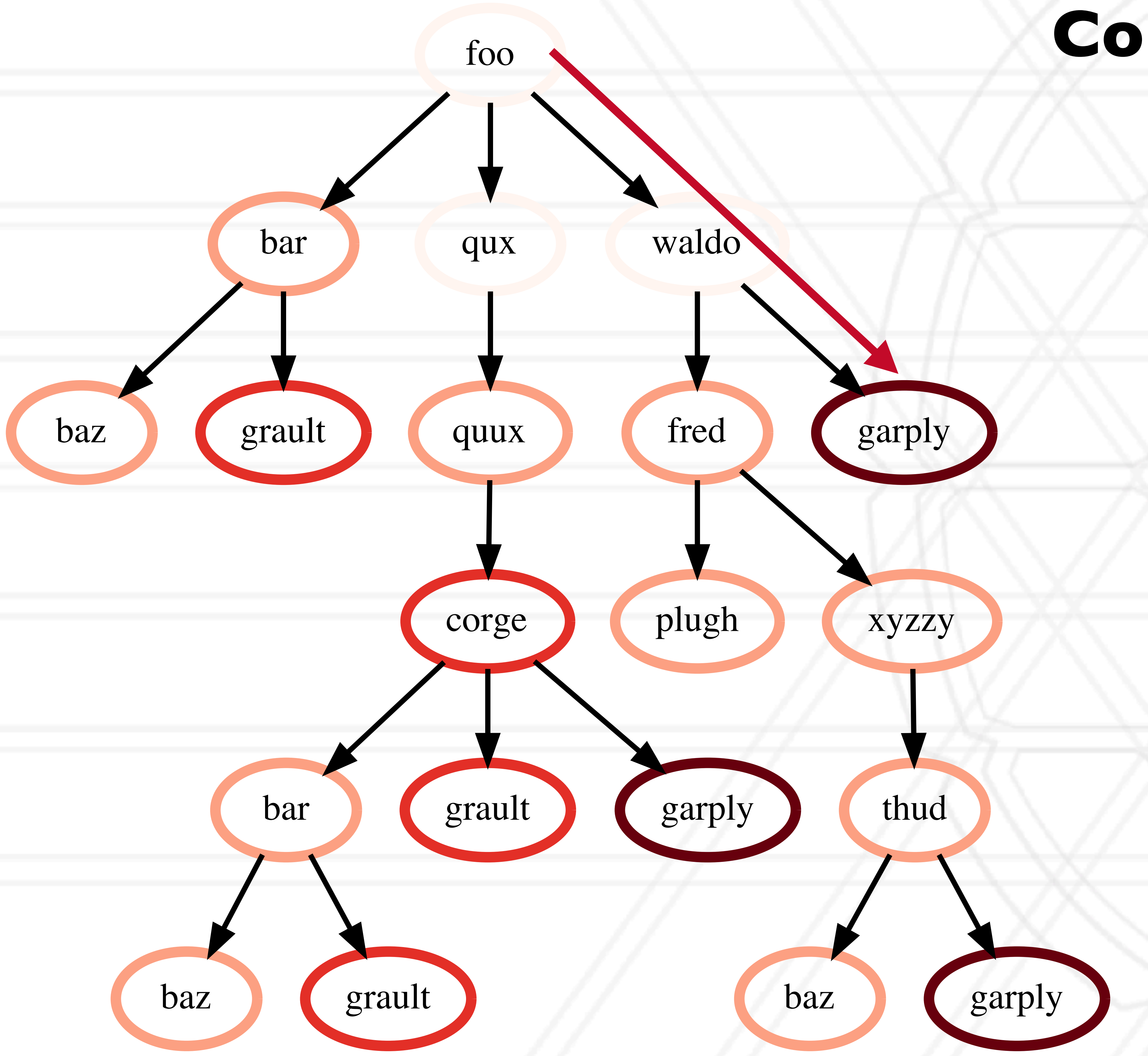
Calling context tree (CCT)

Calling context trees, call graphs, …



Calling context tree (CCT)

Calling context trees, call graphs, …

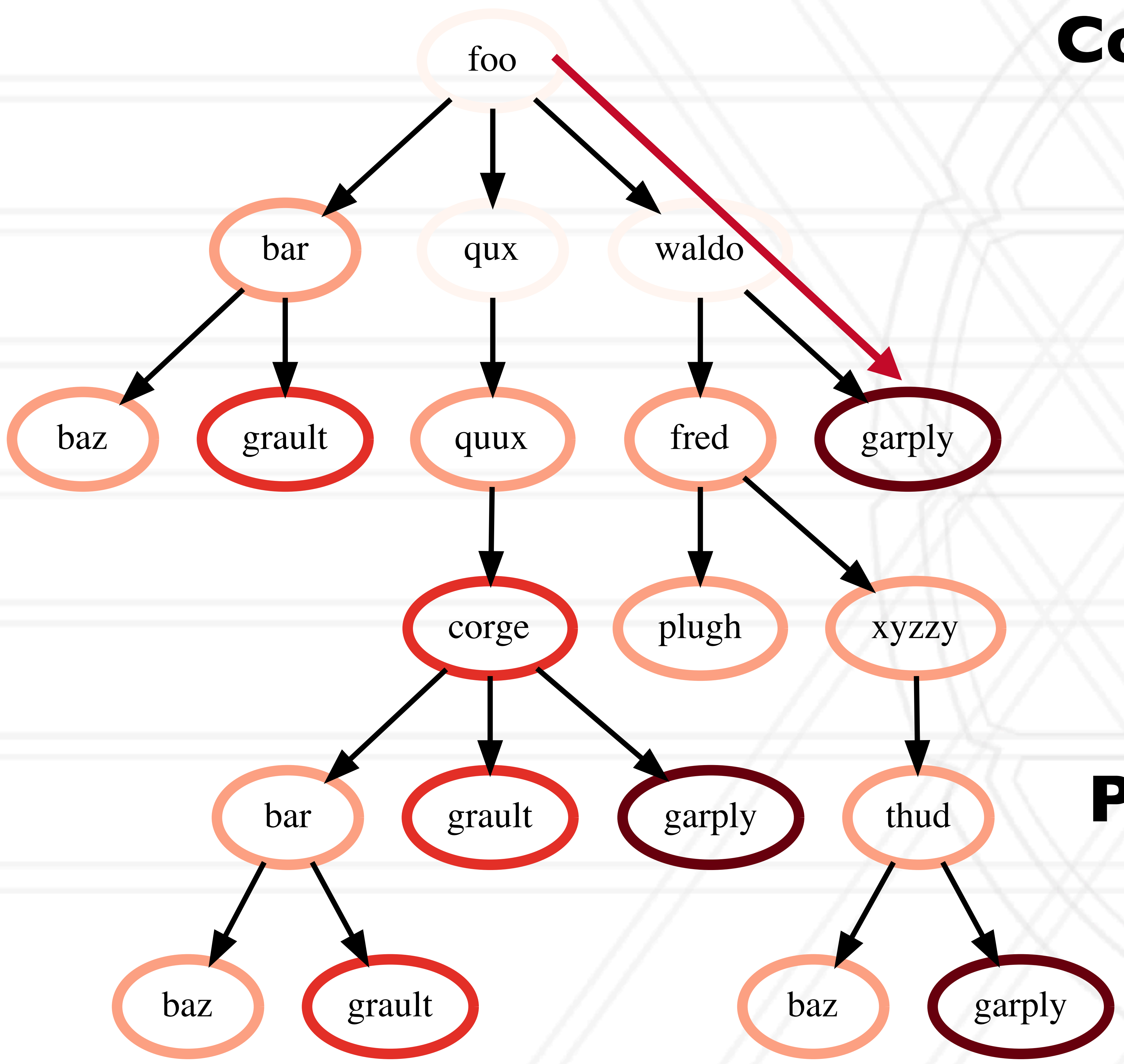


Calling context tree (CCT)

Contextual information

```
File
Line number
Function name
Callpath
Load module
Process ID
Thread ID
```

Calling context trees, call graphs, …



Calling context tree (CCT)

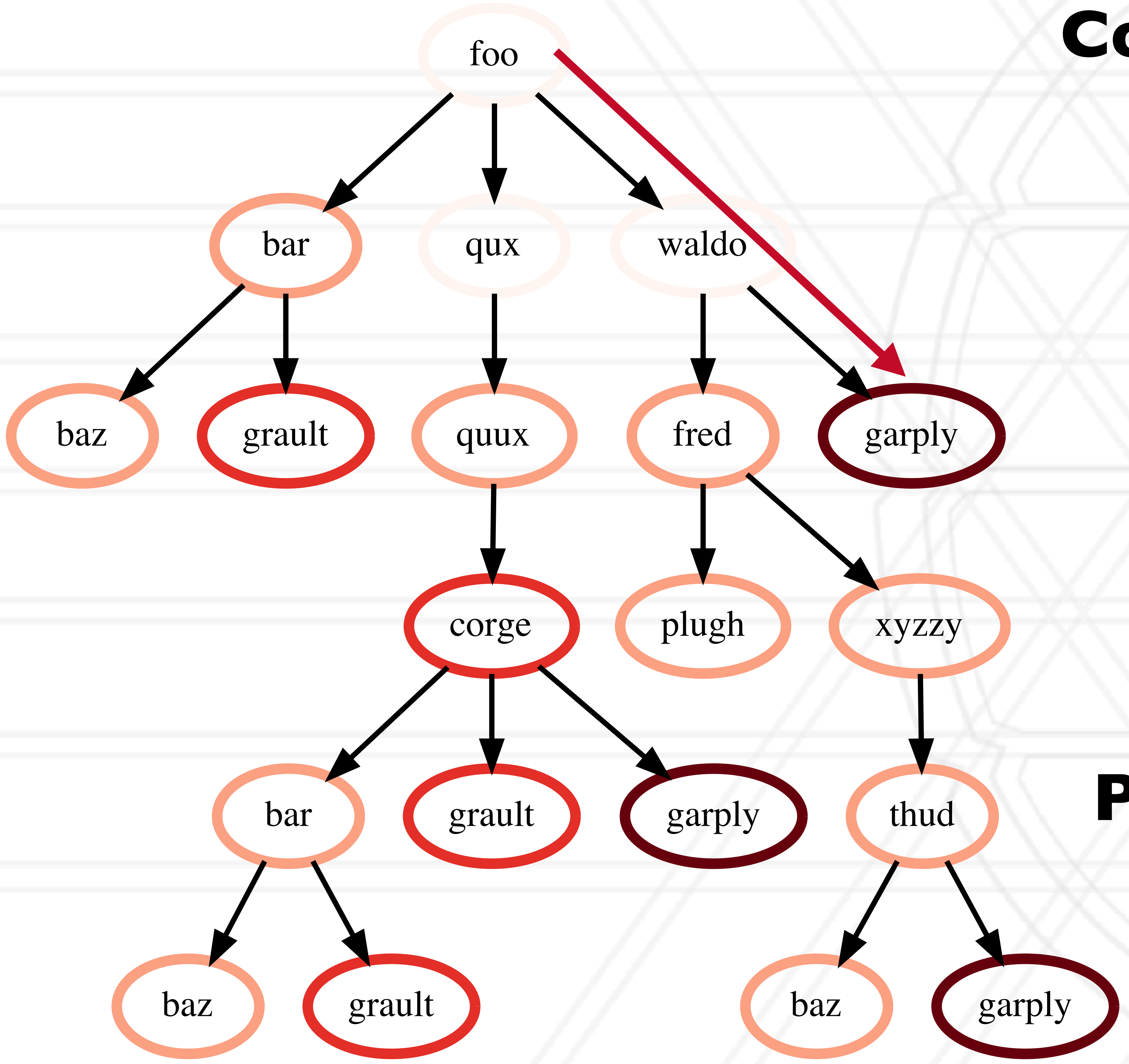
Contextual information

- File
- Line number
- Function name
- Callpath
- Load module
- Process ID
- Thread ID

Performance Metrics

- Time
- Flops
- Cache misses

Calling context trees, call graphs, …



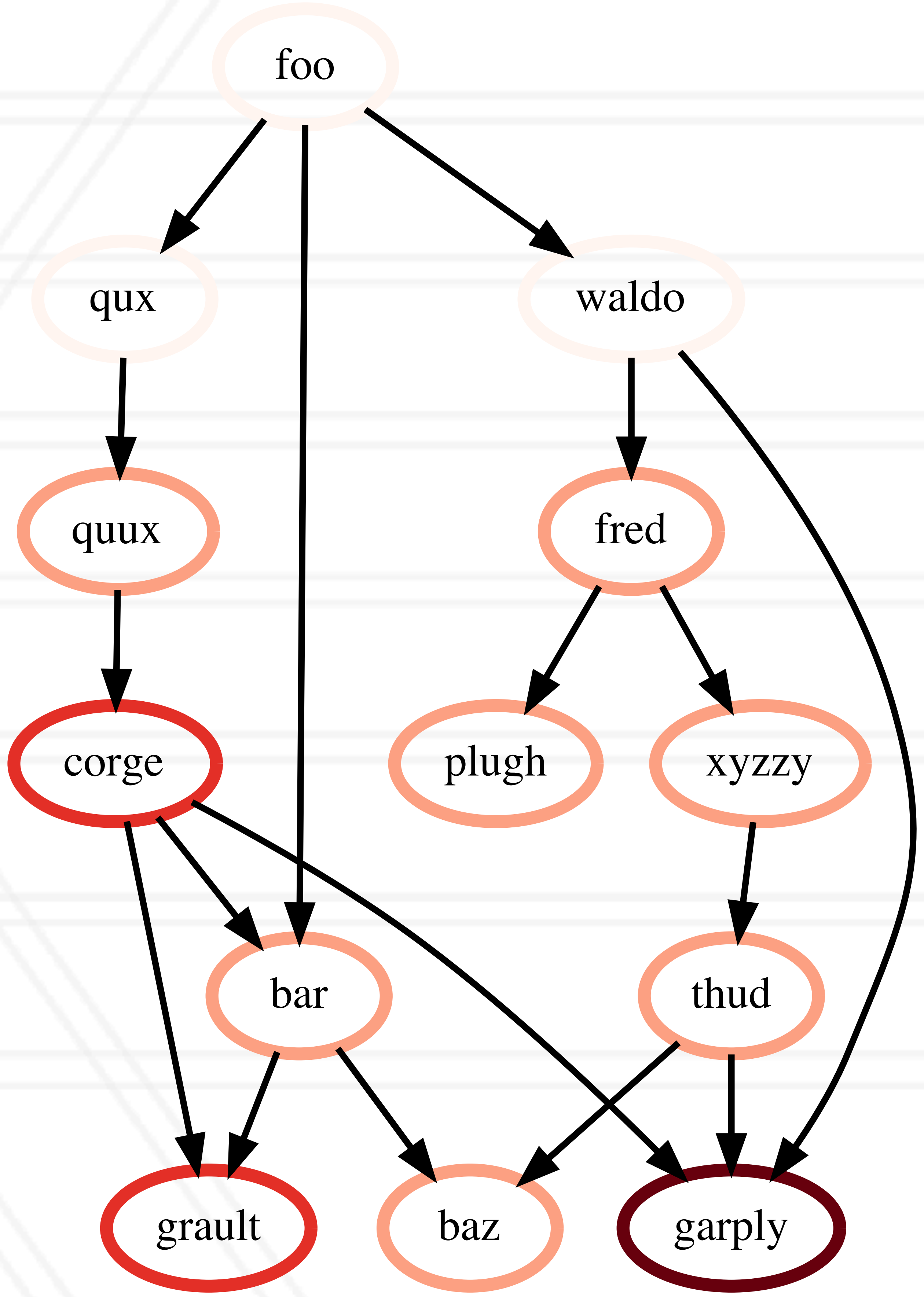
Calling context tree (CCT)

Contextual information

```
File
Line number
Function name
Callpath
Load module
Process ID
Thread ID
```

Performance Metrics

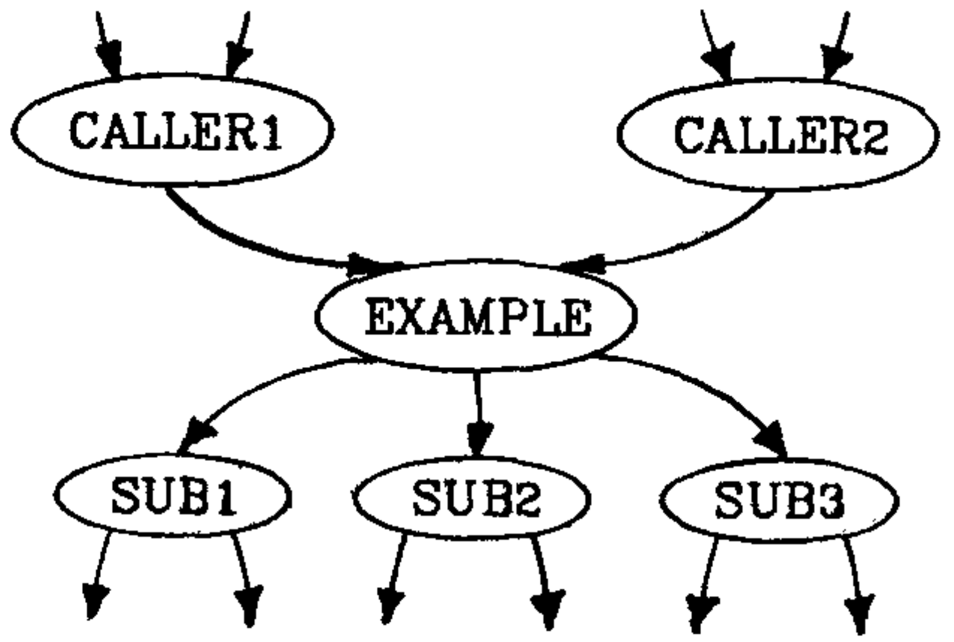
```
Time
Flops
Cache misses
```



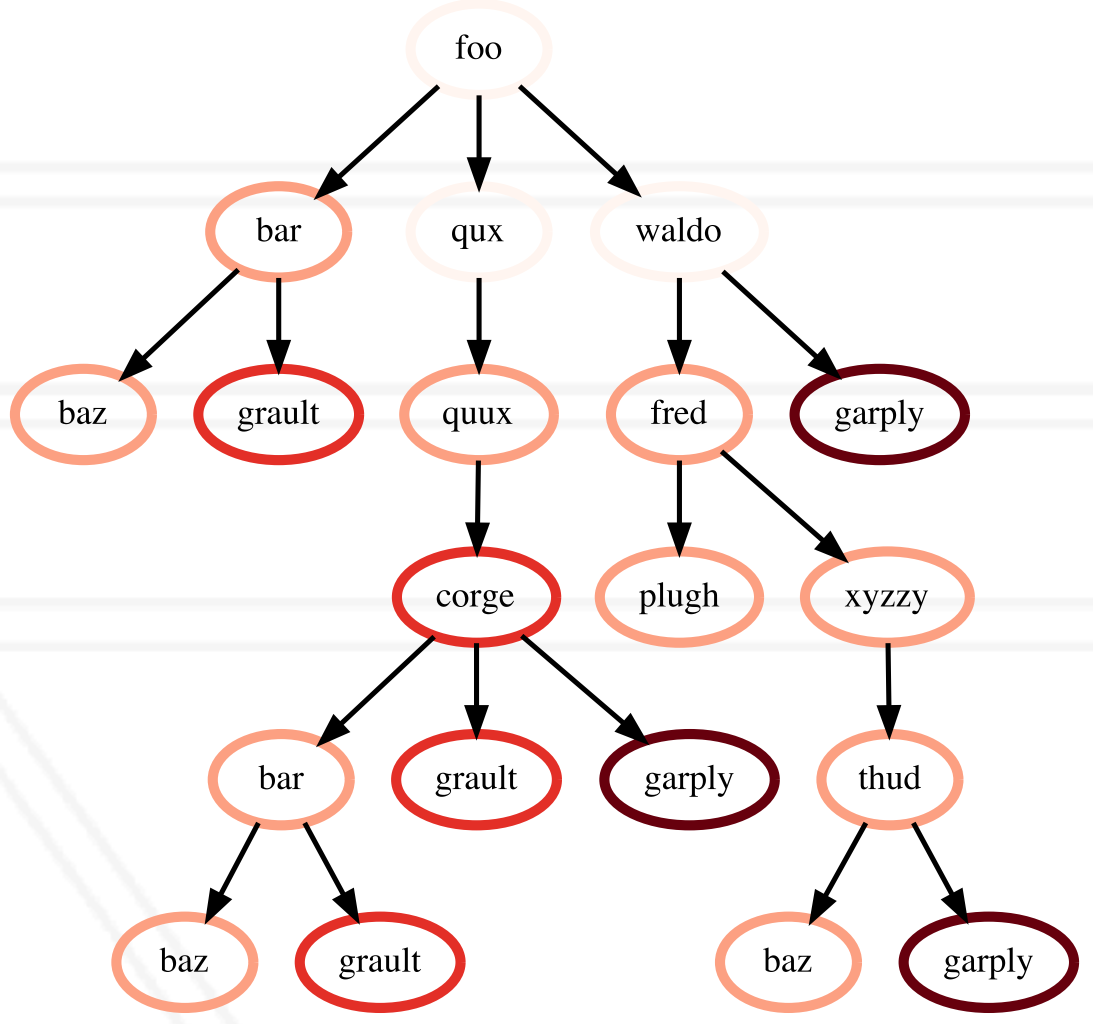
Call graph

Output of profiling tools

Call graph



- Flat profile: Listing of all invoked functions with counts and execution times
- Call graph profile: unique node per function
- Calling context tree: unique node per calling context



Calling context tree

Hatchet: performance analysis tool

- Hatchet enables programmatic analysis of parallel profiles
- Leverages pandas which supports multi-dimensional tabular datasets
- Create a structured index to enable indexing pandas dataframes by nodes in a graph
- A set of operators to filter, prune and/or aggregate structured data

https://hatchet.readthedocs.io/en/latest/

Pandas and dataframes

Pandas and dataframes

- Pandas is an open-source Python library for data analysis

Pandas and dataframes

- Pandas is an open-source Python library for data analysis
- Dataframe: two-dimensional tabular data structure
  - Supports many operations borrowed from SQL databases

Columns

Rows

	node	name	time (inc)	time
0	{'name': 'main'}	main	200.0	10.0
1	{'name': 'physics'}	physics	60.0	40.0
2	{'name': 'mpi'}	mpi	20.0	5.0
3	{'name': 'psm2'}	psm2	15.0	30.0
4	{'name': 'solvers'}	solvers	100.0	10.0
5	{'name': 'hypre'}	hypre	65.0	30.0
6	{'name': 'mpi'}	mpi	35.0	20.0
7	{'name': 'psm2'}	psm2	25.0	60.0

Pandas and dataframes

- Pandas is an open-source Python library for data analysis
- Dataframe: two-dimensional tabular data structure
  - Supports many operations borrowed from SQL databases

Index Columns

Rows

	node	name	time (inc)	time
0	{'name': 'main'}	main	200.0	10.0
1	{'name': 'physics'}	physics	60.0	40.0
2	{'name': 'mpi'}	mpi	20.0	5.0
3	{'name': 'psm2'}	psm2	15.0	30.0
4	{'name': 'solvers'}	solvers	100.0	10.0
5	{'name': 'hypre'}	hypre	65.0	30.0
6	{'name': 'mpi'}	mpi	35.0	20.0
7	{'name': 'psm2'}	psm2	25.0	60.0

Pandas and dataframes

- Pandas is an open-source Python library for data analysis
- Dataframe: two-dimensional tabular data structure
  - Supports many operations borrowed from SQL databases
- MultiIndex enables working with high-dimensional data in a 2D data structure

Index

Columns

Rows

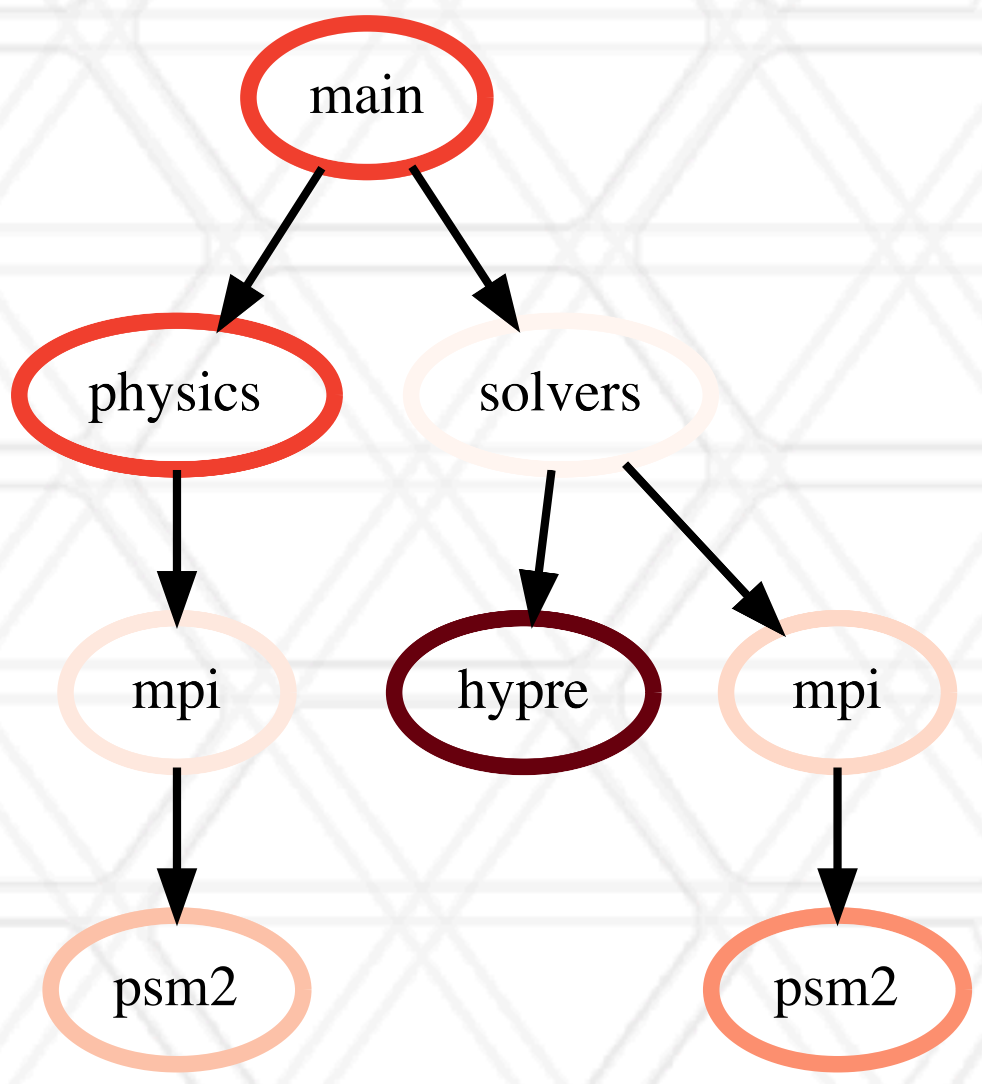
	node	name	time (inc)	time
0	{'name': 'main'}	main	200.0	10.0
1	{'name': 'physics'}	physics	60.0	40.0
2	{'name': 'mpi'}	mpi	20.0	5.0
3	{'name': 'psm2'}	psm2	15.0	30.0
4	{'name': 'solvers'}	solvers	100.0	10.0
5	{'name': 'hypre'}	hypre	65.0	30.0
6	{'name': 'mpi'}	mpi	35.0	20.0
7	{'name': 'psm2'}	psm2	25.0	60.0

Main data structure in hatchet: a *GraphFrame*

- Consists of a structured index graph object and a pandas dataframe
- Graph stores caller-callee relationships
- Dataframe stores all numerical and categorical data for each node in the graph
- In case of multiple processes/ thread, there is a row per node per process per thread

Main data structure in hatchet: a *GraphFrame*

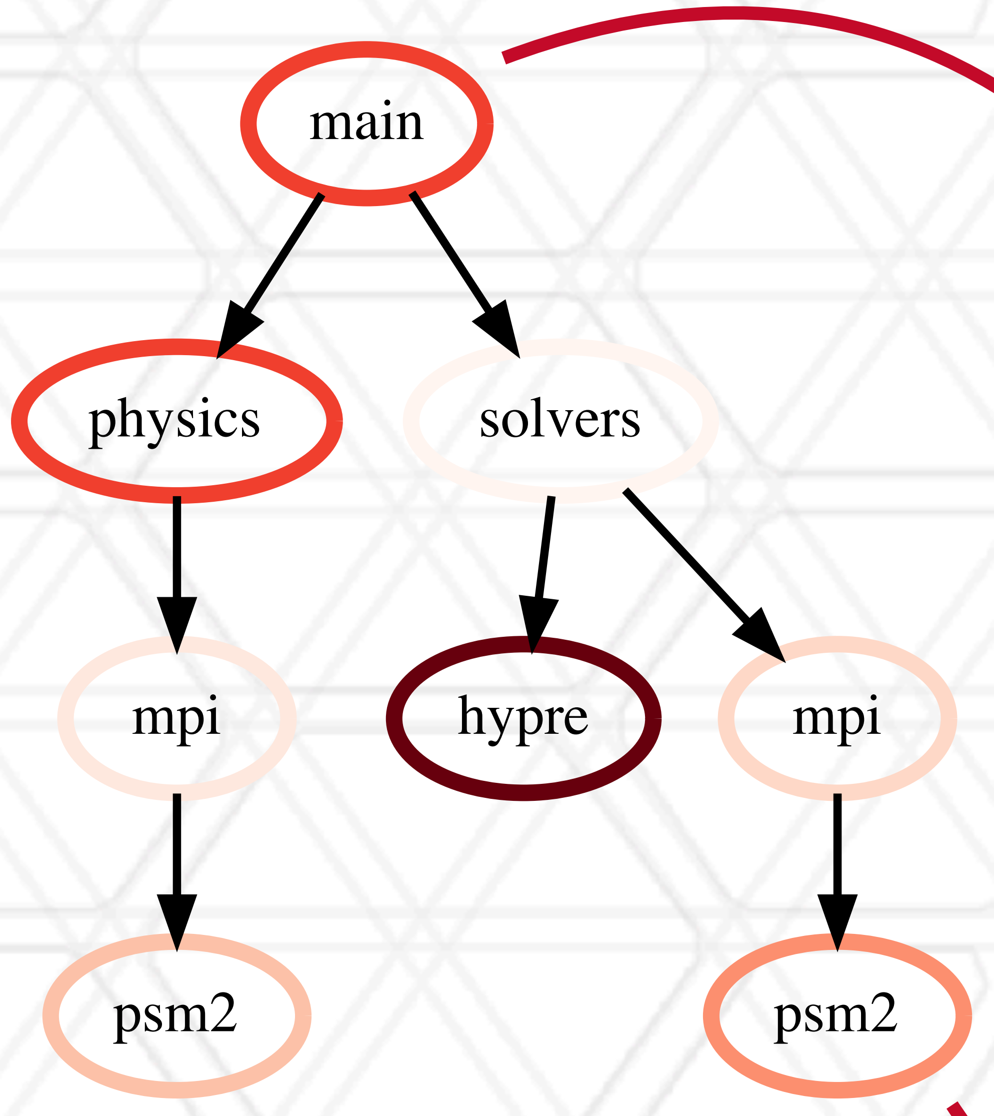
- Consists of a structured index graph object and a pandas dataframe
- Graph stores caller-callee relationships
- Dataframe stores all numerical and categorical data for each node in the graph
- In case of multiple processes/ thread, there is a row per node per process per thread



Graph object

Main data structure in hatchet: a *GraphFrame*

- Consists of a structured index graph object and a pandas dataframe
- Graph stores caller-callee relationships
- Dataframe stores all numerical and categorical data for each node in the graph
- In case of multiple processes/ thread, there is a row per node per process per thread



Graph object

node	name	nid	node	time	time (inc)
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

Dataframe

Dataframe operation: filter

```
filtered_gf = gf.filter(lambda x: x['time'] > 10.0)
```

Dataframe operation: filter

```
filtered_gf = gf.filter(lambda x: x['time'] > 10.0)
```

node	name	nid	node	time	time (inc)
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

Dataframe operation: filter

```
filtered_gf = gf.filter(lambda x: x['time'] > 10.0)
```

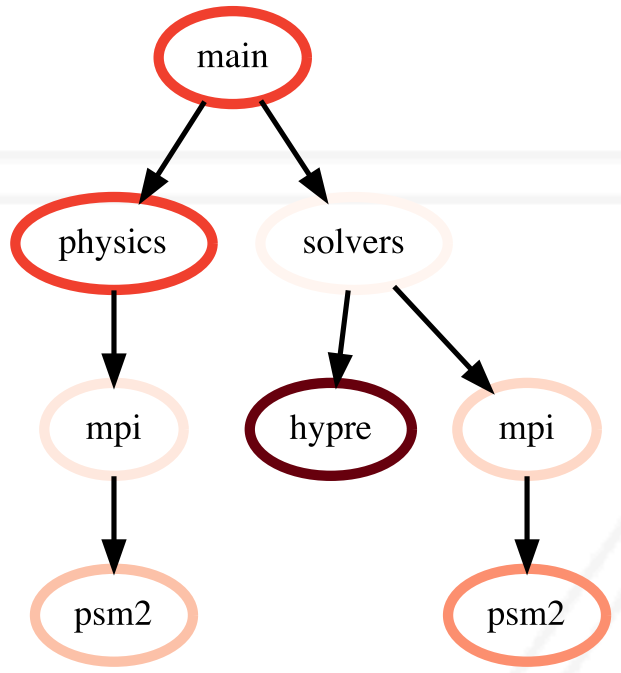
node	name	nid	node	time	time (inc)
main	main	0	main	40.0	200.0
physics	physics	1	physics	40.0	60.0
mpi	mpi	2	mpi	5.0	20.0
psm2	psm2	3	psm2	15.0	15.0
solvers	solvers	4	solvers	0.0	100.0
hypre	hypre	5	hypre	65.0	65.0
mpi	mpi	6	mpi	10.0	35.0
psm2	psm2	7	psm2	25.0	25.0

node	name	nid	node	time	time (inc)
main	main	0	main	40.0	200.0
physics	physics	1	physics	40.0	60.0
psm2	psm2	3	psm2	15.0	15.0
hypre	hypre	5	hypre	65.0	65.0
psm2	psm2	7	psm2	25.0	25.0

Graph operation: squash

```
filtered_gf = gf.filter(lambda x: x['time'] > 10.0)
```

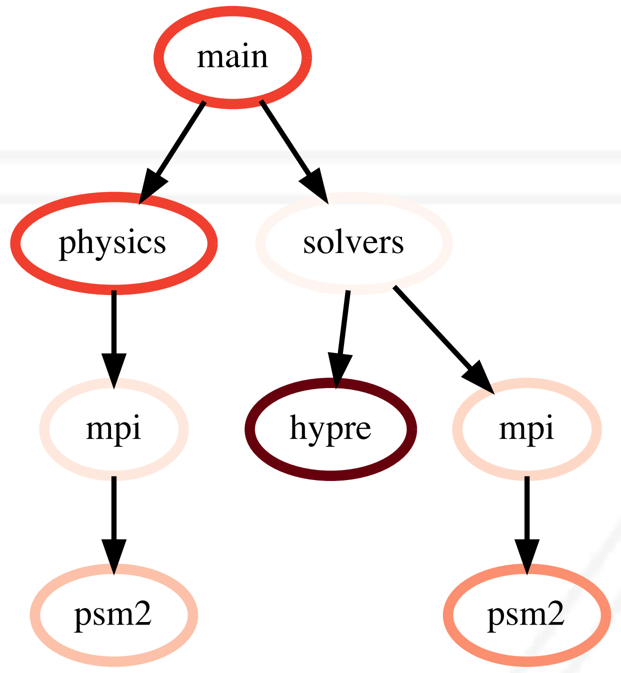
	name	nid	node	time	time (inc)
node					
main	main	0	main	40.0	200.0
physics	physics	1	physics	40.0	60.0
mpi	mpi	2	mpi	5.0	20.0
psm2	psm2	3	psm2	15.0	15.0
solvers	solvers	4	solvers	0.0	100.0
hypre	hypre	5	hypre	65.0	65.0
mpi	mpi	6	mpi	10.0	35.0
psm2	psm2	7	psm2	25.0	25.0



Graph operation: squash

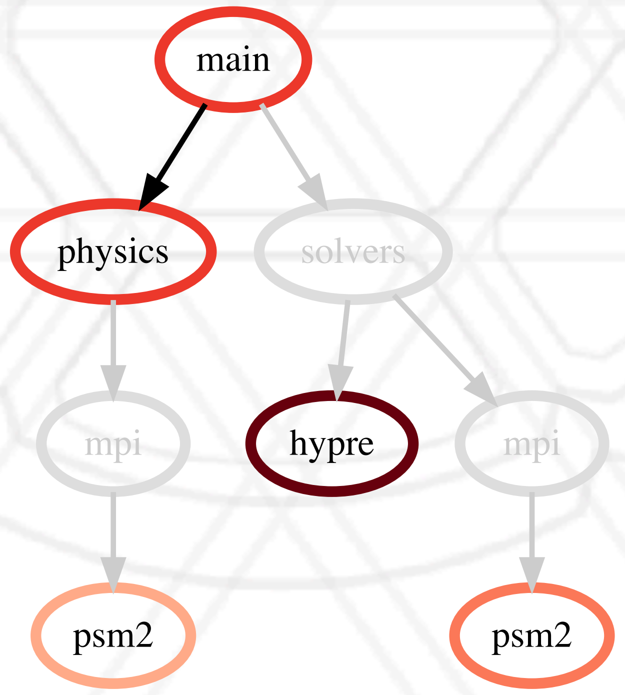
```
filtered_gf = gf.filter(lambda x: x['time'] > 10.0)
```

node	name	nid	node	time	time (inc)
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



filter

node	name	nid	node	time	time (inc)
main	main	0	main	40.0	200.0
physics	physics	1	physics	40.0	60.0
psm2	psm2	3	psm2	15.0	15.0
hypre	hypre	5	hypre	65.0	65.0
psm2	psm2	7	psm2	25.0	25.0

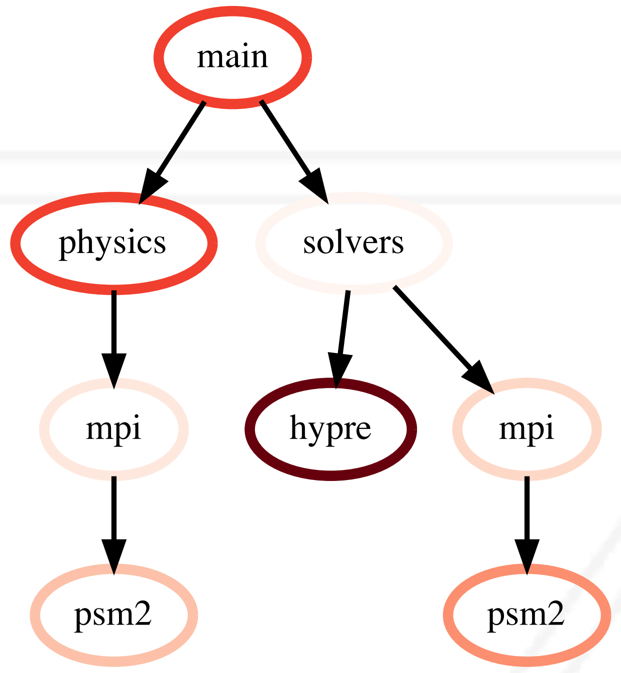


Graph operation: squash

```
filtered_gf = gf.filter(lambda x: x['time'] > 10.0)
```

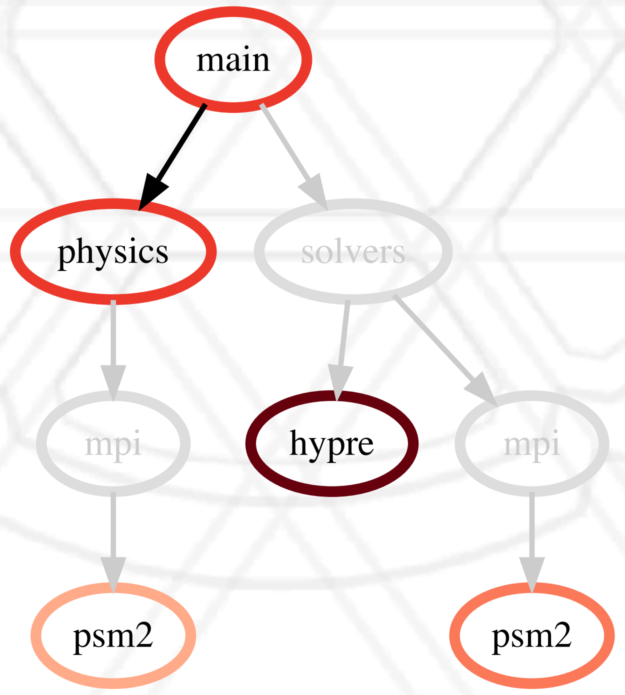
```
squashed_gf = filtered_gf.squash()
```

node	name	nid	node	time	time (inc)
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



filter

node	name	nid	node	time	time (inc)
main	main	0	main	40.0	200.0
physics	physics	1	physics	40.0	60.0
psm2	psm2	3	psm2	15.0	15.0
hypre	hypre	5	hypre	65.0	65.0
psm2	psm2	7	psm2	25.0	25.0

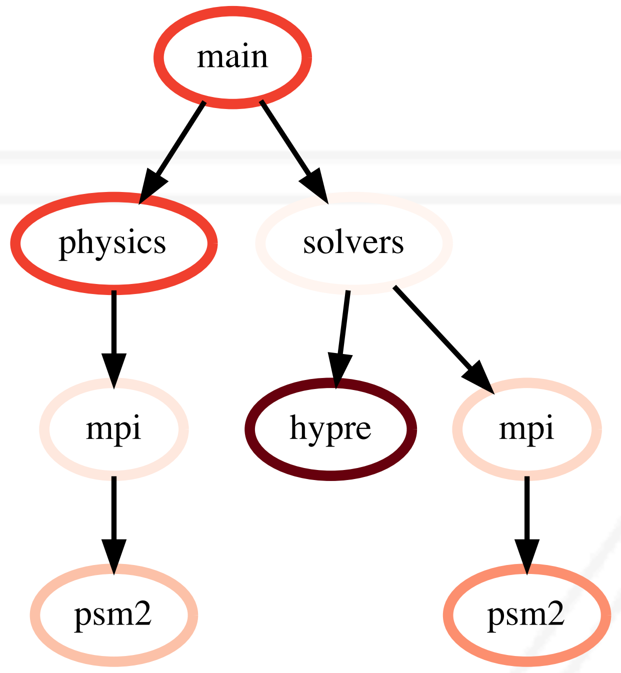


Graph operation: squash

```
filtered_gf = gf.filter(lambda x: x['time'] > 10.0)
```

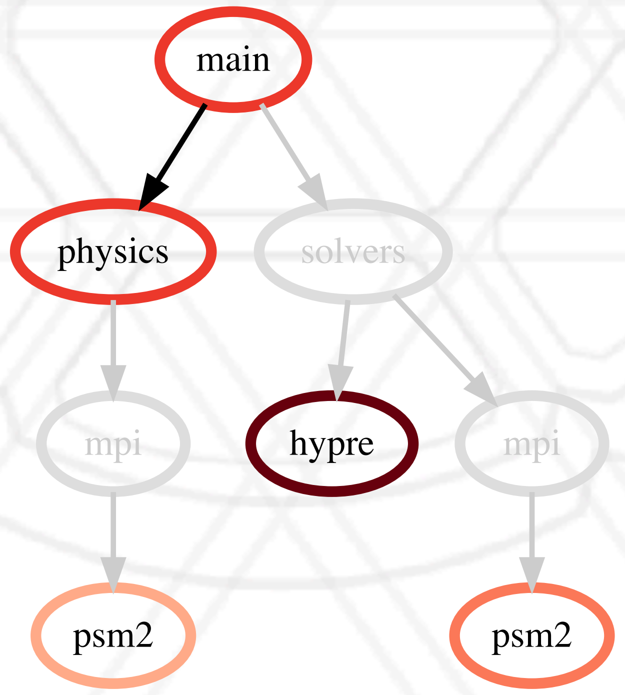
```
squashed_gf = filtered_gf.squash()
```

node	name	nid	node	time	time (inc)
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



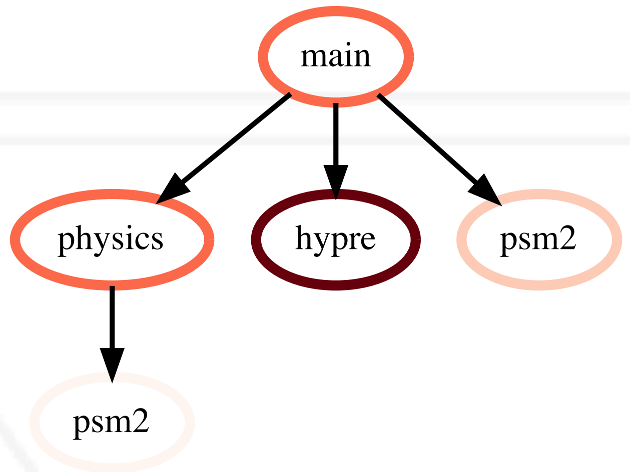
filter

node	name	nid	node	time	time (inc)
main	main	0	main	40.0	200.0
physics	physics	1	physics	40.0	60.0
psm2	psm2	3	psm2	15.0	15.0
hypre	hypre	5	hypre	65.0	65.0
psm2	psm2	7	psm2	25.0	25.0



squash

node	name	nid	node	time	time (inc)
main	main	0	main	40.0	200.0
physics	physics	1	physics	40.0	60.0
psm2	psm2	3	psm2	15.0	15.0
hypre	hypre	5	hypre	65.0	65.0
psm2	psm2	7	psm2	25.0	25.0

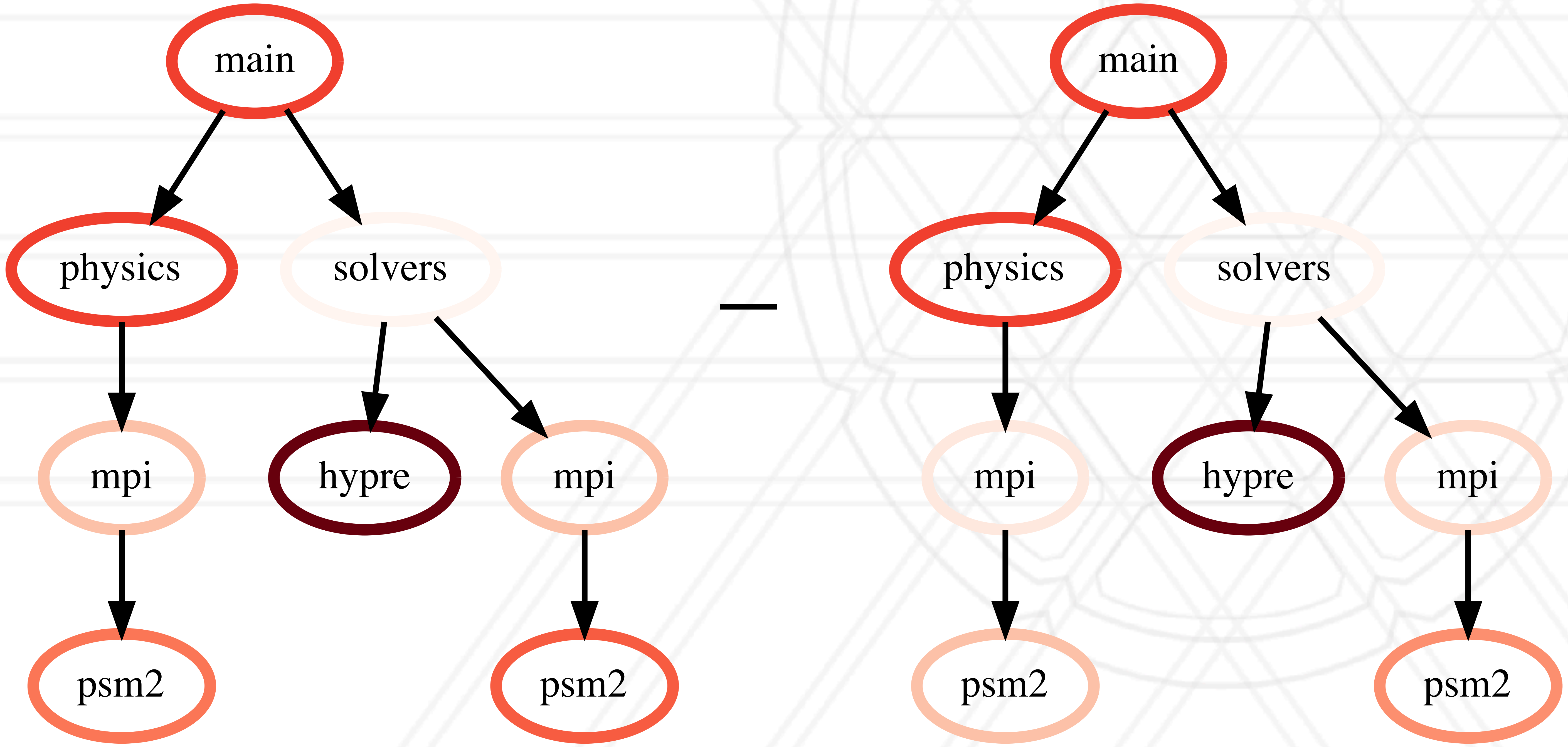


Graphframe operation: subtract

```
gf1 = ht.GraphFrame.from_literal( ... )
gf2 = ht.GraphFrame.from_literal( ... )
gf2 -= gf1
```

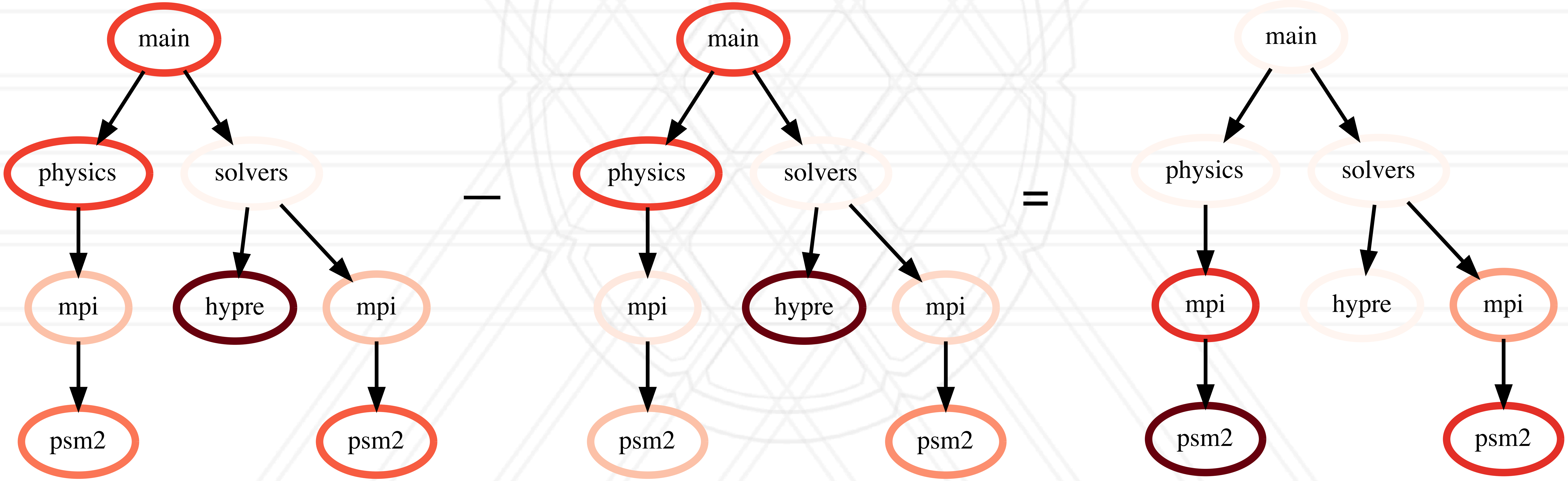
Graphframe operation: subtract

```
gf1 = ht.GraphFrame.from_literal( ... )
gf2 = ht.GraphFrame.from_literal( ... )
gf2 -= gf1
```



Graphframe operation: subtract

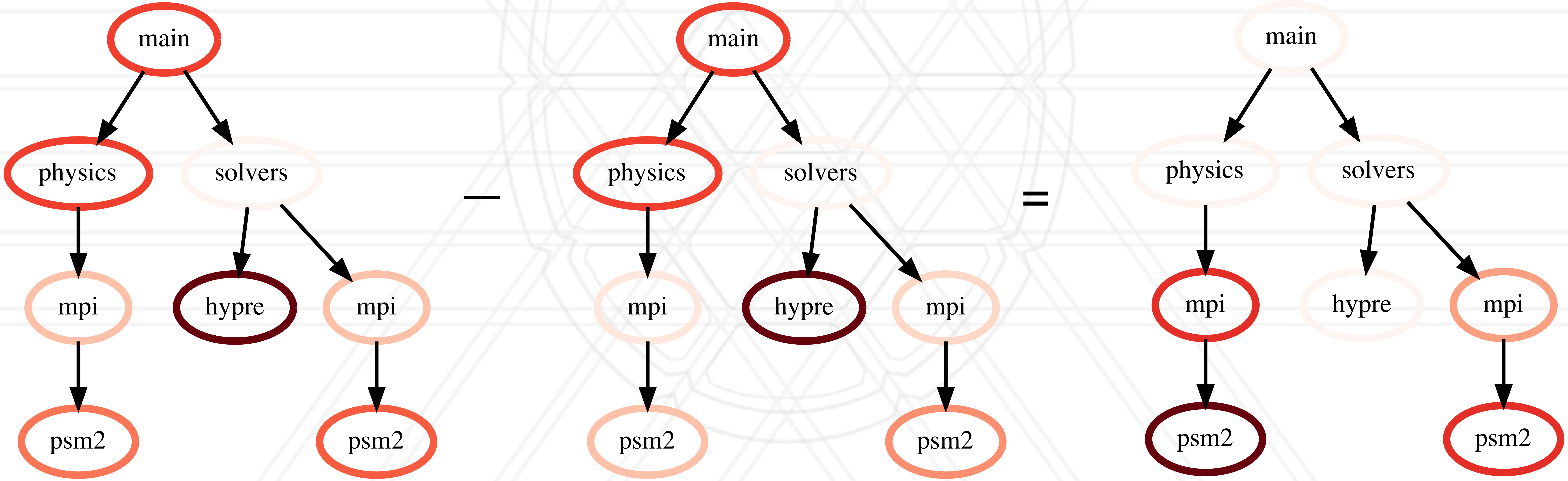
```
gf1 = ht.GraphFrame.from_literal( ... )
gf2 = ht.GraphFrame.from_literal( ... )
gf2 -= gf1
```



Graphframe operation: subtract

```
gf1 = ht.GraphFrame.from_literal( ... )
gf2 = ht.GraphFrame.from_literal( ... )
gf2 -= gf1
```

https://hatchet.readthedocs.io



Visualizing *small* graphs

```
print(gf.tree(color=True))
```

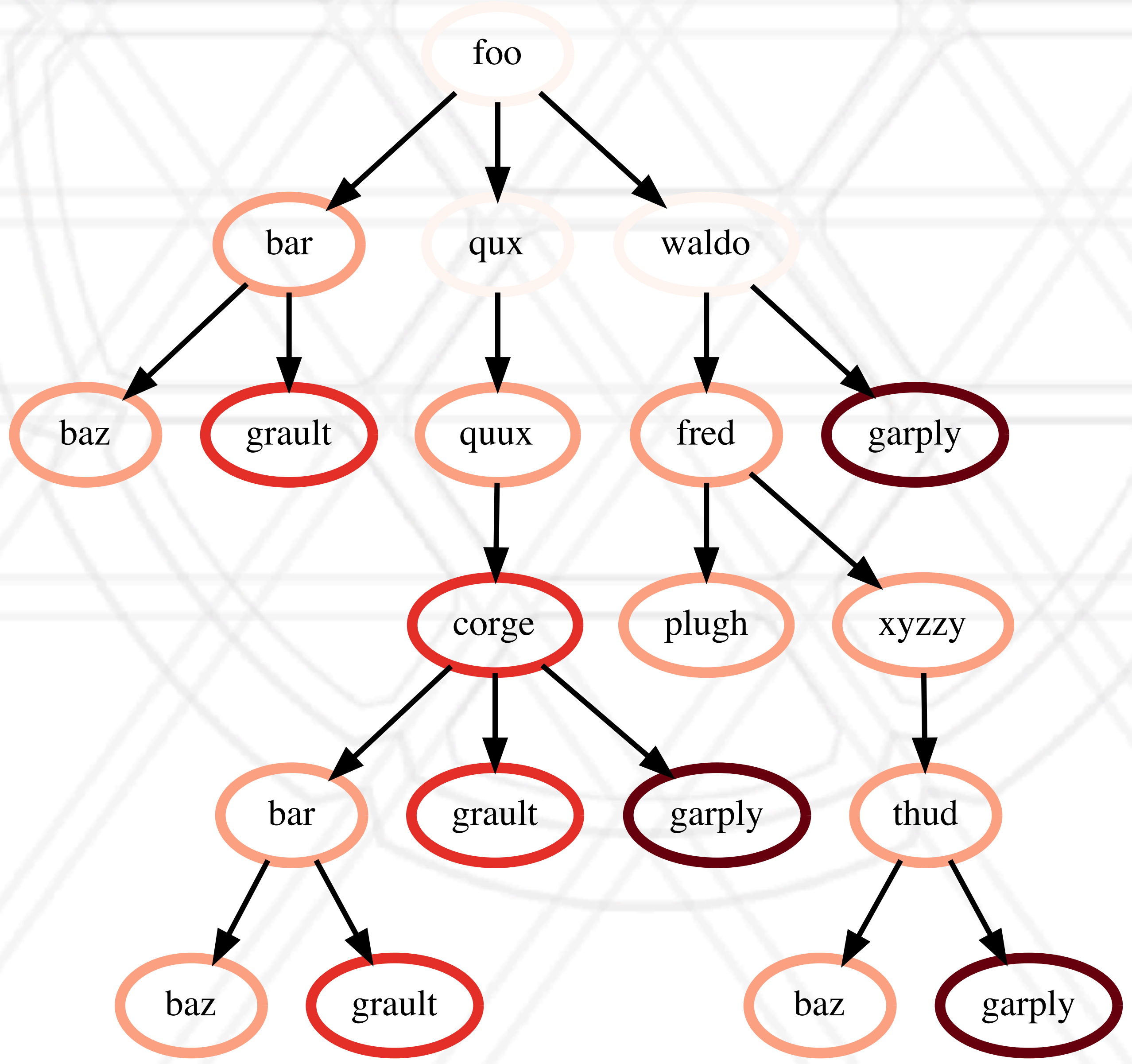
```
0.000 foo
├─ 5.000 bar
│  ├─ 5.000 baz
│  └─ 10.000 grault
├─ 0.000 qux
│  └─ 5.000 quux
│     └─ 10.000 corge
│        ├─ 5.000 bar
│        │  ├─ 5.000 baz
│        │  └─ 10.000 grault
│        ├─ 10.000 grault
│        └─ 15.000 garply
└─ 0.000 waldo
   ├─ 5.000 fred
   │  ├─ 5.000 plugh
   │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

Visualizing *small* graphs

```
print(gf.tree(color=True))
```

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

```
with open("test.dot", "w") as dot_file:
    dot_file.write(gf.to_dot())
```

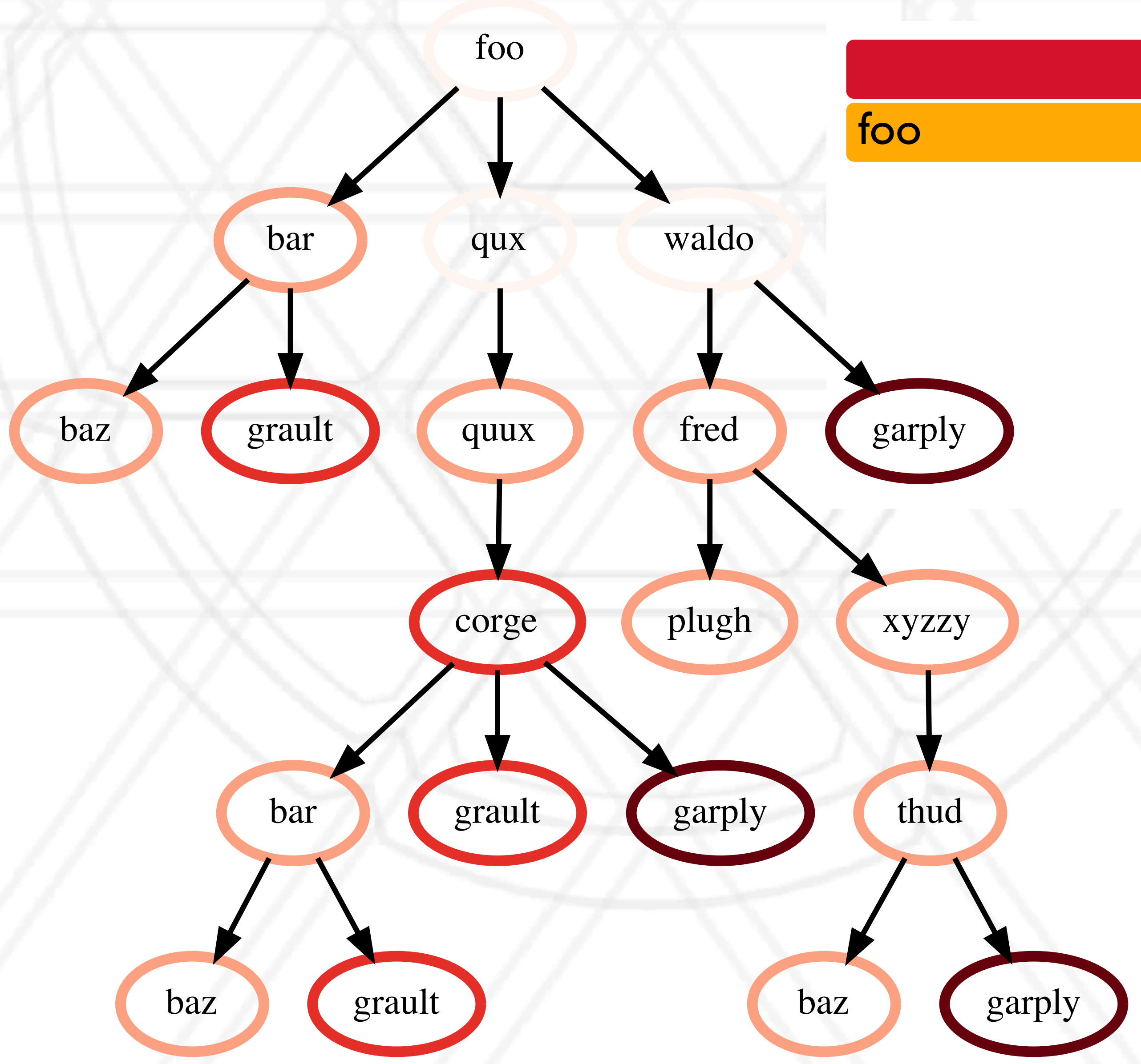


Visualizing *small* graphs

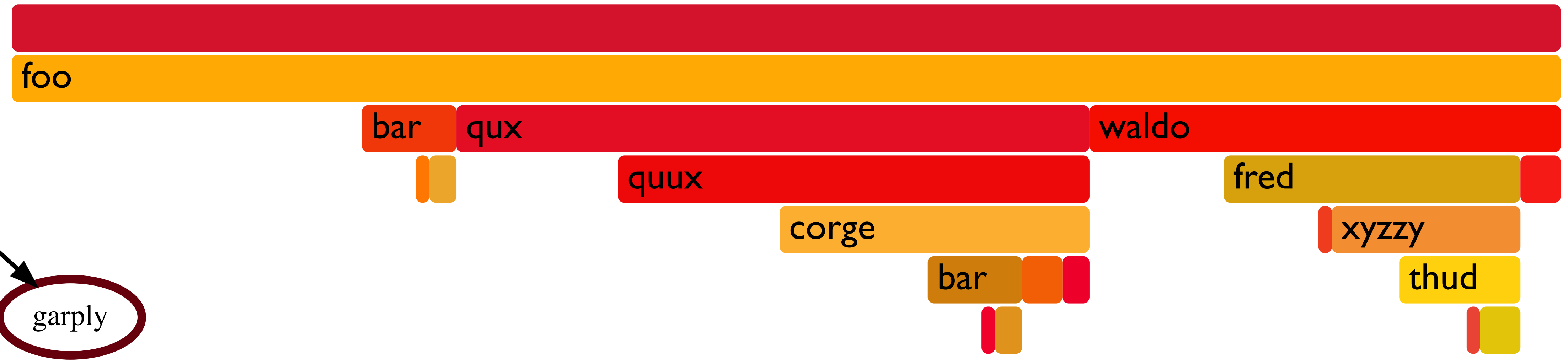
```python
print(gf.tree(color=True))
```

```
0.000 foo
├─ 5.000 bar
|  ├─ 5.000 baz
|  └─ 10.000 grault
├─ 0.000 qux
|  └─ 5.000 quux
|     └─ 10.000 corge
|        ├─ 5.000 bar
|        |  ├─ 5.000 baz
|        |  └─ 10.000 grault
|        ├─ 10.000 grault
|        └─ 15.000 garply
└─ 0.000 waldo
   ├─ 5.000 fred
   |  ├─ 5.000 plugh
   |  └─ 5.000 xyzzy
   |     └─ 5.000 thud
   |        ├─ 5.000 baz
   |        └─ 15.000 garply
   └─ 15.000 garply
```

```python
with open("test.dot", "w") as dot_file:
    dot_file.write(gf.to_dot())
```




```python
with open("test.txt", "w") as folded_stack:
    folded_stack.write(gf.to_flamegraph())
```



Flamegraph

Starter code for reading data

```
import hatchet as ht
import sys

if __name__ == '__main__':
    file_name = sys.argv[1]
    gf = ht.GraphFrame.from_caliper(file_name)

    print(gf.tree())
    print(gf.dataframe)
```

Replace this with another reader depending on data source

Example 1: Generating a flat profile

```
gf = ht.GraphFrame.from_hpctoolkit('kripke')
gf.drop_index_levels()

grouped = gf.dataframe.groupby('name').sum()
sorted_df = grouped.sort_values(by=['time'], ascending=False)
print(sorted_df)
```

Example 1: Generating a flat profile

```
gf = ht.GraphFrame.from_hpctoolkit('kripke')
gf.drop_index_levels()

grouped = gf.dataframe.groupby('name').sum()
sorted_df = grouped.sort_values(by=['time'], ascending=False)
print(sorted_df)
```

Example 1: Generating a flat profile

```python
gf = ht.GraphFrame.from_hpctoolkit('kripke')
gf.drop_index_levels()

grouped = gf.dataframe.groupby('name').sum()
sorted_df = grouped.sort_values(by=['time'], asce
print(sorted_df)
```

	nid	time	time (inc)
name			
<unknown file> [kripke]:0	17234	1.825282e+08	1.825282e+08
Kernel_3d_DGZ::scattering	60	7.669936e+07	7.896253e+07
Kernel_3d_DGZ::LTimes	30	5.010439e+07	5.240528e+07
Kernel_3d_DGZ::LPlusTimes	115	4.947707e+07	5.104498e+07
Kernel_3d_DGZ::sweep	981	5.018862e+06	5.018862e+06
memset.S:99	3773	3.168982e+06	3.168982e+06
memset.S:101	3970	2.120895e+06	2.120895e+06
Grid_Data::particleEdit	1201	1.131266e+06	1.249157e+06
<unknown file> [libpsm2.so.2.1]:0	324763	9.733415e+05	9.733415e+05
memset.S:98	3767	6.197776e+05	6.197776e+05

Example 2: Comparing two executions

```
gf1 = ht.GraphFrame.from_caliper('lulesh-1core.json')
gf2 = ht.GraphFrame.from_caliper('lulesh-27cores. json')

gf2.drop_index_levels()
gf3 = gf2 - gf1

sorted_df = gf3.dataframe.sort_values(by=['time'], ascending=False)
print(sorted_df)
```

Example 2: Comparing two executions

```
gf1 = ht.GraphFrame.from_caliper('lulesh-1core.json')
gf2 = ht.GraphFrame.from_caliper('lulesh-27cores. json')

gf2.drop_index_levels()
gf3 = gf2 - gf1

sorted_df = gf3.dataframe.sort_values(by=['time'], ascending=False)
print(sorted_df)
```

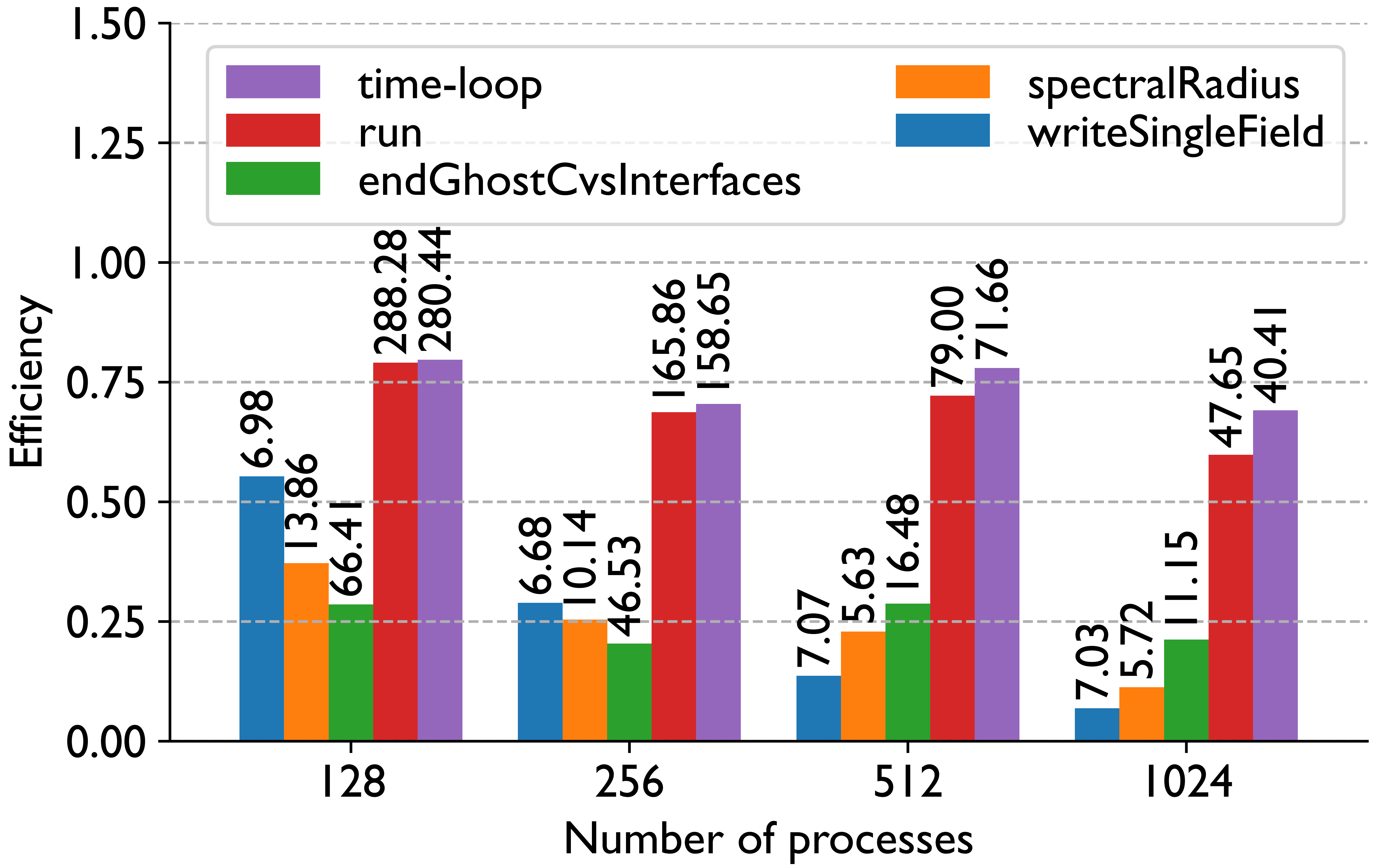
Example 2: Comparing two executions

```
gf1 = ht.GraphFrame.from_caliper('lulesh-1core.json')
gf2 = ht.GraphFrame.from_caliper('lulesh-27cores. json')

gf2.drop_index_levels()
gf3 = gf2 - gf1

sorted_df = gf3.dataframe.sort_values(by=['time'], ascending=False)
print(sorted_df)
```

node	name	nid	time	time (inc)
TimeIncrement	TimeIncrement	25.0	8.505048e+06	8.505048e+06
CalcQForElems	CalcQForElems	16.0	4.455672e+06	5.189453e+06
CalcHourglassControlForElems	CalcHourglassControlForElems	7.0	3.888798e+06	4.755817e+06
LagrangeNodal	LagrangeNodal	3.0	1.986046e+06	8.828475e+06
CalcForceForNodes	CalcForceForNodes	4.0	1.017857e+06	6.842429e+06

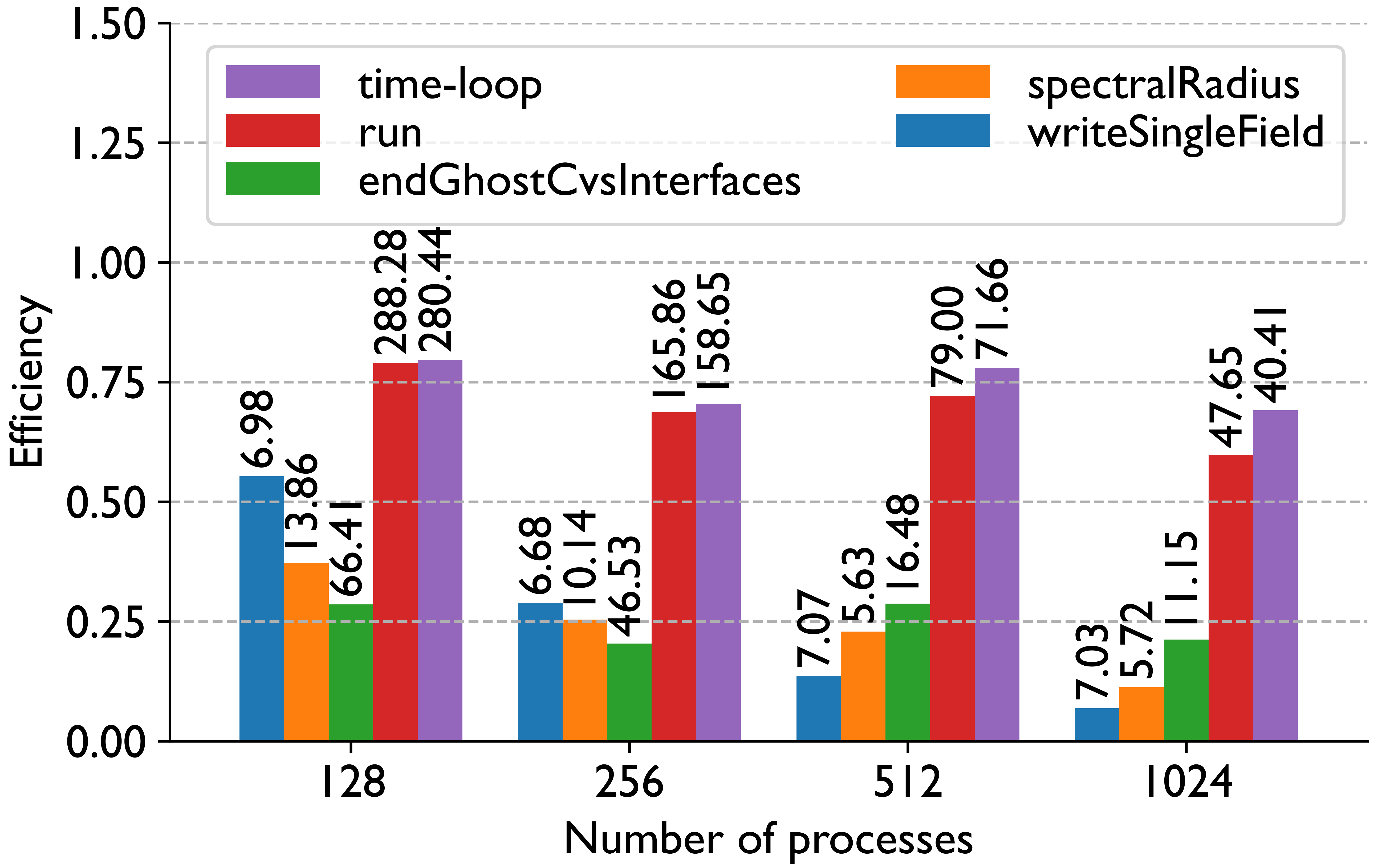
Example 3: Speedup and efficiency

```python
datasets = glob.glob("list_of_tortuga_profiles")
gfs = hatchet.GraphFrame.construct_from(datasets)

df = hatchet.Chopper.speedup_efficiency(gfs, strong=True,
     efficiency=True)
df = df.loc[df['1024'] < 0.7]
df.T.loc[:, :].plot.bar()
```



Example 3: Speedup and efficiency

```
datasets = glob.glob("list_of_tortuga_profiles")
gfs = hatchet.GraphFrame.construct_from(datasets)

df = hatchet.Chopper.speedup_efficiency(gfs, strong=True,
      efficiency=True)
df = df.loc[df['1024'] < 0.7]
df.T.loc[:, :].plot.bar()
```



Example 4: Load imbalance

```python
graphframe = hatchet.GraphFrame.from_hpctoolkit("qs_profile_128")

graphframe_imbalance = graphframe.load_imbalance(verbose=True)
# sort the top 50 nodes that have the highest mean value by imbalance
df_imb = graphframe_imbalance.dataframe.head(50).sort_values("time.imbalance", ascending=False)
print(df_imb.head(4))  # Dataframe Output (a)
```

name	time.imbalance	time.ranks	time.hist	time.percentiles	time.mean
MacroscopicCrossSection.cc:22	4.199311	[39 46 118 33 94]	[105, 14, 2, 1, 0, 0, 0, 0, 1, 5]	[6.296, 7.12, 7.302, 7.67, 39.102]	9.311
MacroscopicCrossSection.cc:32	1.539592	[67 92 39 46 94]	[2, 3, 16, 80, 9, 2, 7, 6, 1, 2]	[21.083, 30.333, 30.946, 31.61, 49.334]	32.043
NuclearData.cc:270	1.329530	[84 119 5 120 118]	[27, 38, 26, 16, 13, 5, 1, 0, 0, 2]	[40.667, 42.775, 44.325, 46.946, 60.088]	45.195
MCT.cc:582	1.319152	[12 79 47 67 15]	[69, 54, 3, 1, 0, 0, 0, 0, 0, 1]	[17.525, 18.032, 18.158, 18.255, 24.019]	18.208

Example 4: Load imbalance

```python
graphframe = hatchet.GraphFrame.from_hpctoolkit("qs_profile_128")

graphframe_imbalance = graphframe.load_imbalance(verbose=True)
# sort the top 50 nodes that have the highest mean value by imbalance
df_imb = graphframe_imbalance.dataframe.head(50).sort_values("time.imbalance", ascending=False)
print(df_imb.head(4))  # Dataframe Output (a)
```

name	time.imbalance	time.ranks	time.hist	time.percentiles	time.mean
MacroscopicCrossSection.cc:22	4.199311	[39 46 118 33 94]	[105, 14, 2, 1, 0, 0, 0, 0, 1, 5]	[6.296, 7.12, 7.302, 7.67, 39.102]	9.311
MacroscopicCrossSection.cc:32	1.539592	[67 92 39 46 94]	[2, 3, 16, 80, 9, 2, 7, 6, 1, 2]	[21.083, 30.333, 30.946, 31.61, 49.334]	32.043
NuclearData.cc:270	1.329530	[84 119 5 120 118]	[27, 38, 26, 16, 13, 5, 1, 0, 0, 2]	[40.667, 42.775, 44.325, 46.946, 60.088]	45.195
MCT.cc:582	1.319152	[12 79 47 67 15]	[69, 54, 3, 1, 0, 0, 0, 0, 0, 1]	[17.525, 18.032, 18.158, 18.255, 24.019]	18.208

UNIVERSITY OF MARYLAND

Abhinav Bhatele

5218 Brendan Iribe Center (IRB) / College Park, MD 20742

phone: 301.405.4507 / e-mail: bhatele@cs.umd.edu