Introduction to Parallel Computing (CMSC416)

Performance Modeling, Analysis, and Tools

Alan Sussman, Department of Computer Science

UNIVERSITY OF MARYLAND

Annoucements

- Assignment 1 will be posted on Monday
  - Due 2 weeks later, on March 7
- 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
- Quiz 1 will be released next Wednesday
  - You will have 24 hours to take it on ELMS

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, …
- 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

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

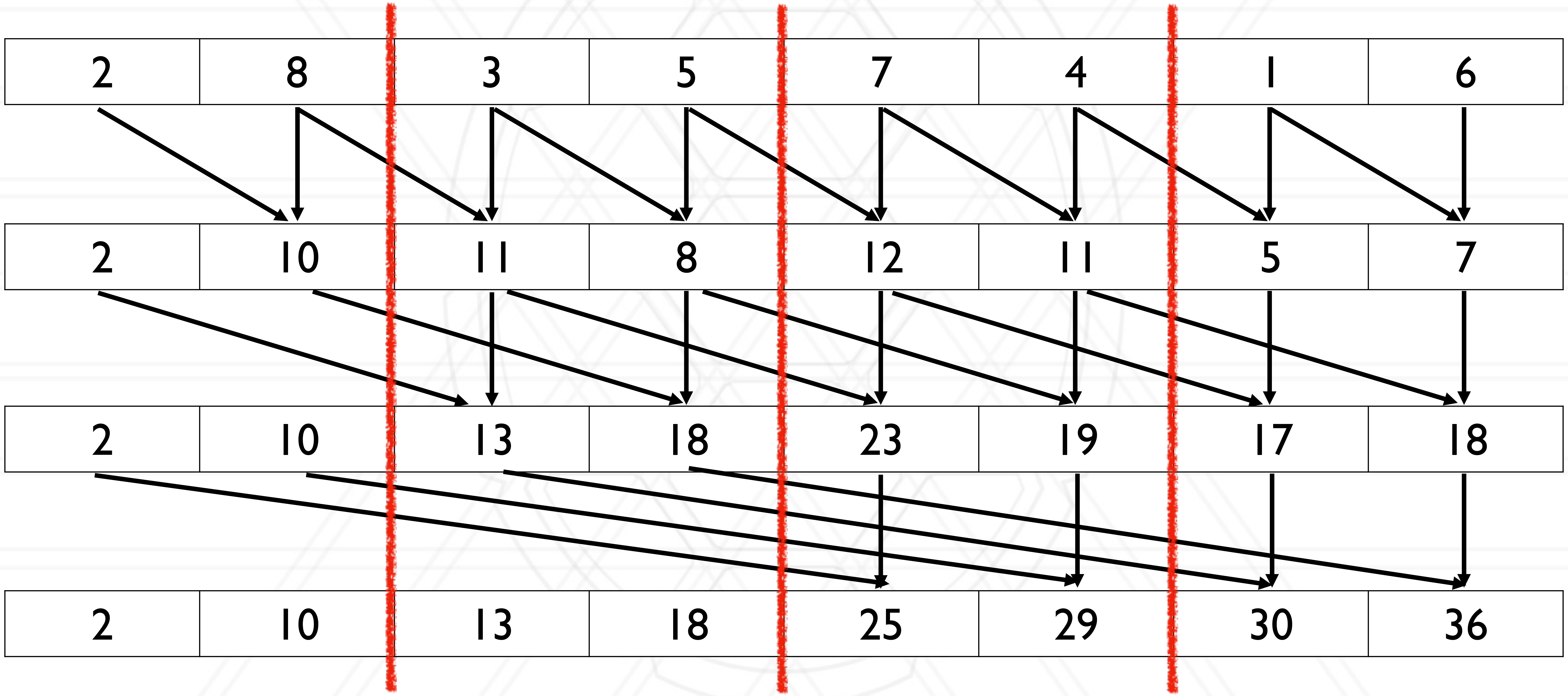
Performance analysis

- 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 parallel code
- Identify why performance might be slow
  - Serial performance
  - Serial bottlenecks when running in parallel
  - Communication overheads

Performance analysis methods

- Analytical techniques: use algebraic formulae
  - In terms of data size (n), number of processes (p)
- Time complexity analysis
- Scalability analysis (Isoefficiency)
- Model performance of various operations
  - Analytical models: LogP, alpha-beta model
- Empirical performance analysis using tools

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: $\frac{n}{p}$
- Then do parallel algorithm with partial prefix sums
  - Number of phases: $log(p)$
  - Total number of calculations: $log(p) \times \frac{n}{p}$

Modeling communication: LogP model

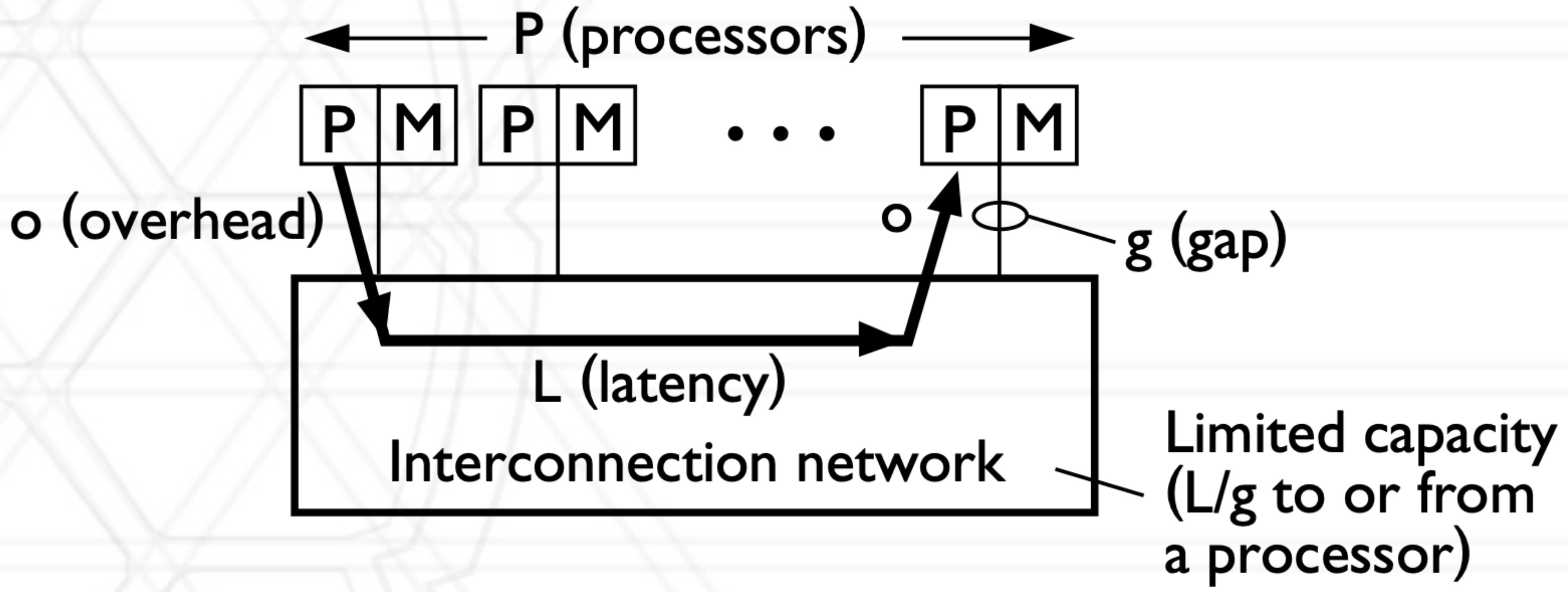
- Model for communication on an interconnection network

L: latency or delay

o: overhead (processor busy in communication)

g: gap (between successive sends/recvs)

P: number of processors / processes



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 processors to maintain a certain level of efficiency
- At what rate should we increase problem size with respect to number of processors to keep efficiency constant

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)

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

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

Isoefficiency function

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

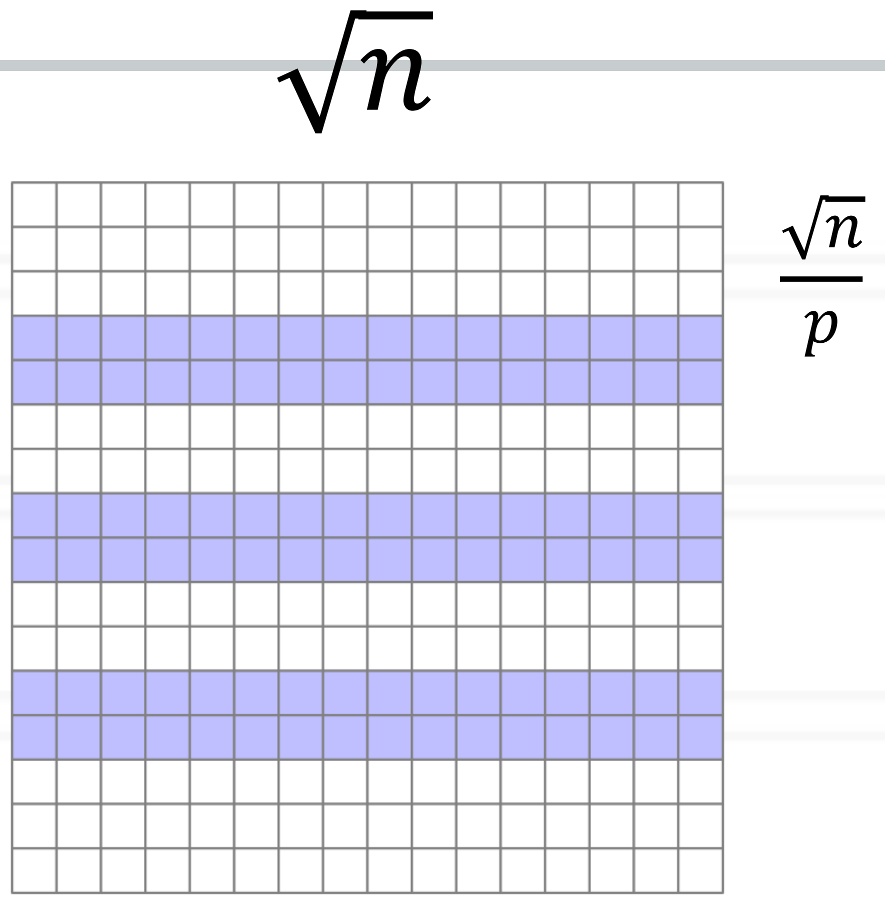
- Efficiency is constant if t_o / t_1 is constant (K)

$$t_o = K \times t_1$$

Isoefficiency analysis

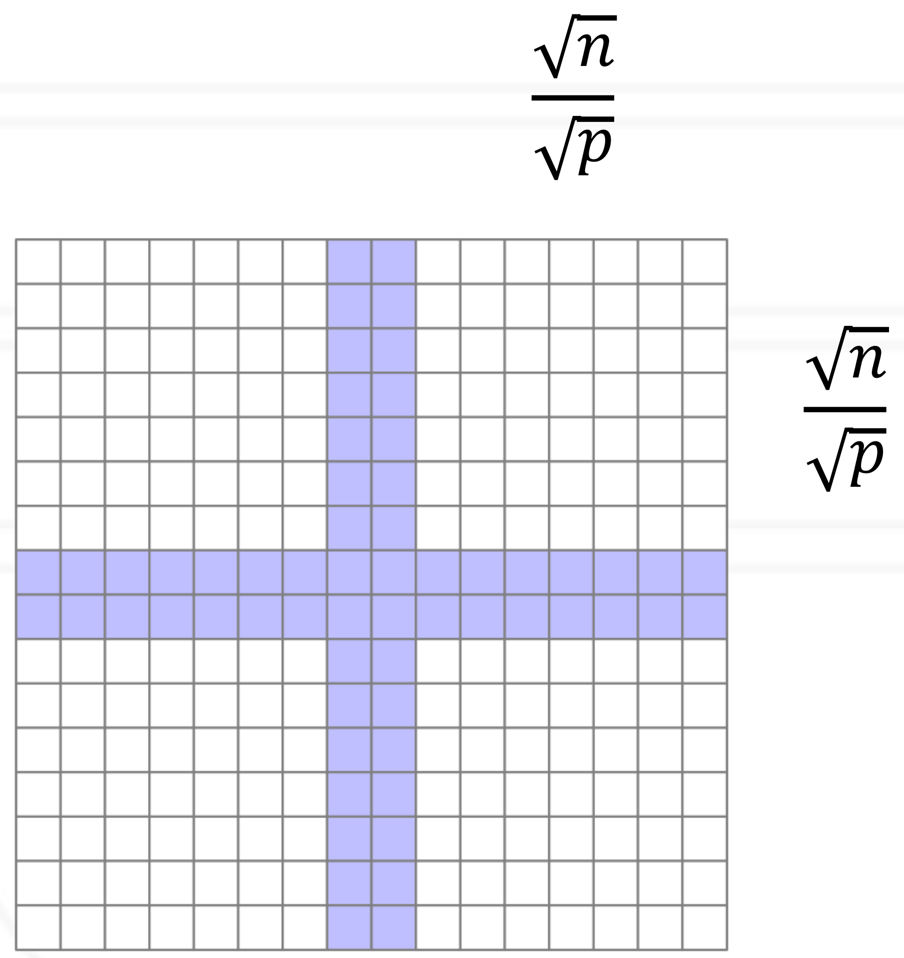
- 1D decomposition:
  - Computation: $\sqrt{n} \times \frac{\sqrt{n}}{p} = \frac{n}{p}$
  - Communication: $2 \times \sqrt{n}$

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



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

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



Performance Modeling, Analysis, and Tools

Alan Sussman, Department of Computer Science

UNIVERSITY OF MARYLAND

Annoucements

- Assignment 1 posted and due Tuesday, March 7
  - Questions?
- Quiz 1 will be released tomorrow, Wed., at 11AM
  - You will have 24 hours to take it on ELMS

Requests for Assignment 1

1. Tar

   - When you tar the directory, make sure it does not contain unnecessary files, such as binary executables.
   - Name the tarball and the directory correctly, following the specified naming format.
   - Often, people tar from their home or scratch directory, which results in an unnecessarily long directory path (which will not be handled by the autograder!). Please tar from the directory with your code and other files, or use the -C flag to tar.
     E.g., **tar -czvf Oh-Keonwoo-assign0.tar.gz -C ~/scratch/Oh-Keonwoo-assign0/.. Oh-Keonwoo-assign0**

2. Batch scripts

   - Allocate by number of tasks, rather than nodes. Each node on Zaratan has many cores. Also, there is no need to set --ntasks-per-node like in the example script
     E.g., **#SBATCH --ntasks=4**
   - Make sure to load the modules you need. E.g., **module load openmpi/gnu**
   - Set OMPI_MCA_mpi_cuda_support to 0 in your batch script if you do not want to see the warning about CUDA when you run mpi.
     E.g., **export OMPI_MCA_mpi_cuda_support=0**

3. Makefile

   - Define CC or CXX correctly to the compiler you are using (gcc, g++, mpicc, mpicxx, etc.)
     E.g., **CXX = mpicxx**

Empirical performance analysis

- Two parts to performance analysis
  - measurement
  - analysis/visualization
- Simplest tool: timers in the code and printf

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

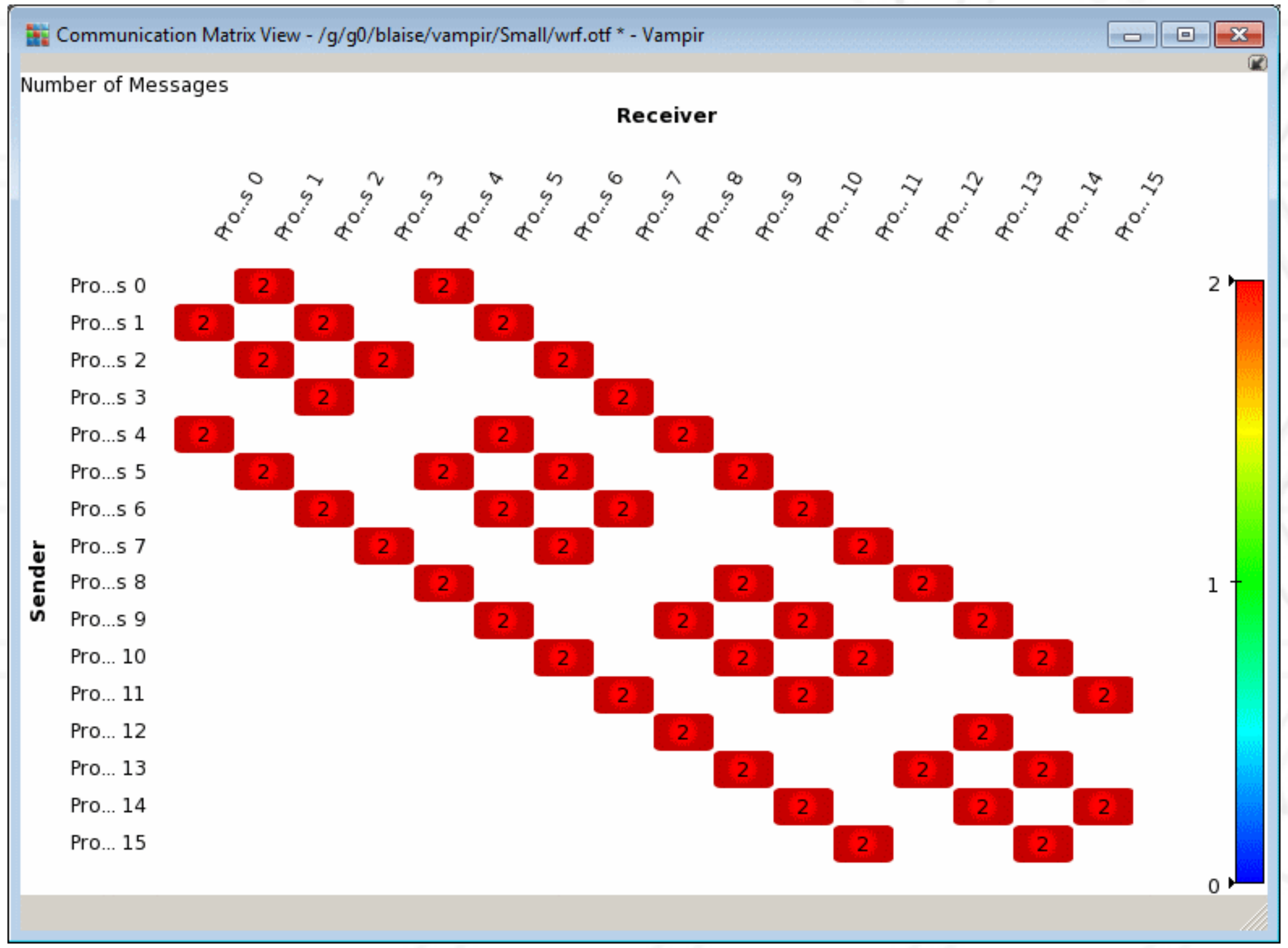
- Tracing tools
  - Capture entire execution trace
- Profiling tools
  - Provide aggregated information
  - Typically use statistical sampling
- Many tools can do both

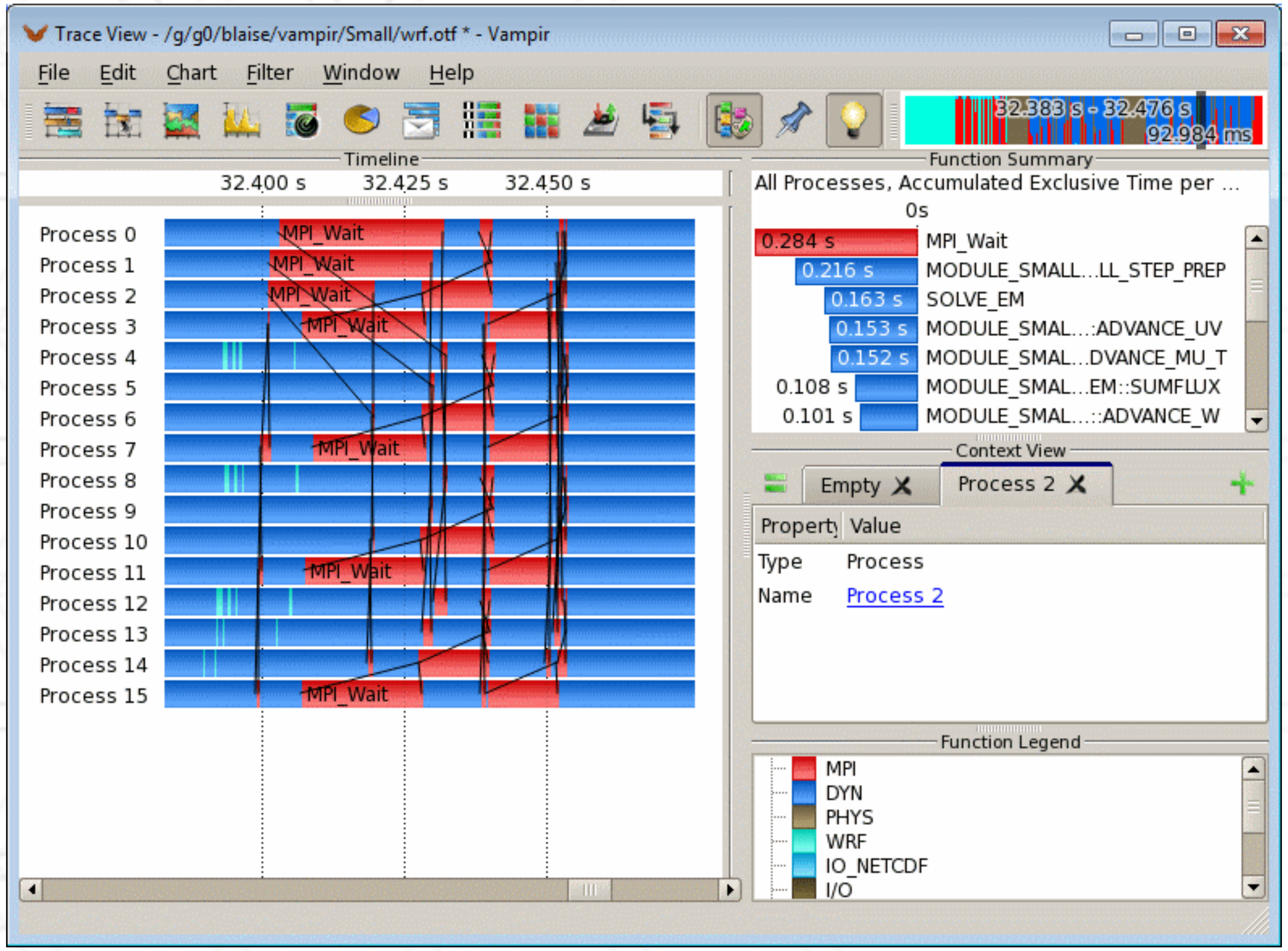
Metrics recorded

- Counts of function invocations
- Time spent in code
- Number of bytes sent/received
- Hardware counters
- To fix performance problems — we need to connect metrics to source code

Tracing tools

- Record all the events in the program with timestamps, typically via instrumentation
- Events: function calls, MPI events, etc.





Vampir visualization: https://hpc.llnl.gov/software/development-environment-software/vampir-vampir-server

Examples of tracing tools

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

Profiling tools

- Ignore the specific times at which events occurred
- Provide aggregate information about different parts of the code
- Examples:
  - Gprof, perf
  - mpiP
  - HPCToolkit, caliper
- Python tools: cprofile, pyinstrument, scalene

gprof Data

Enter filter text

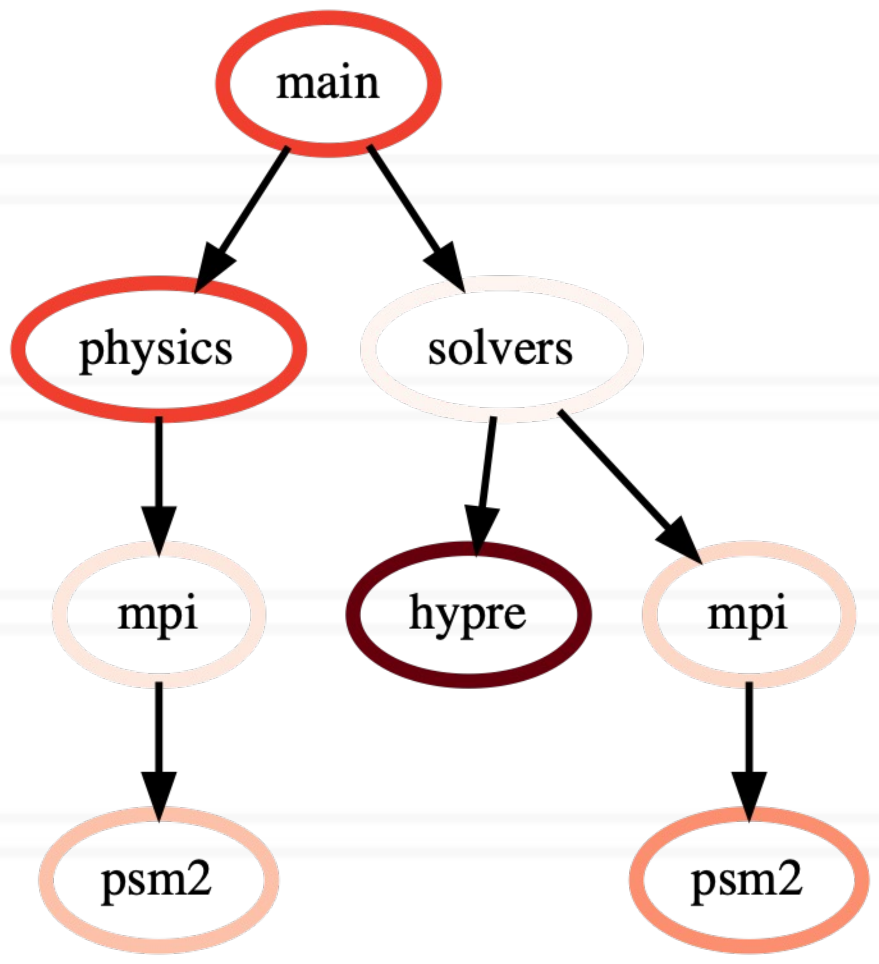
4 bytes per bucket, each sample counts as 10.000ms

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%

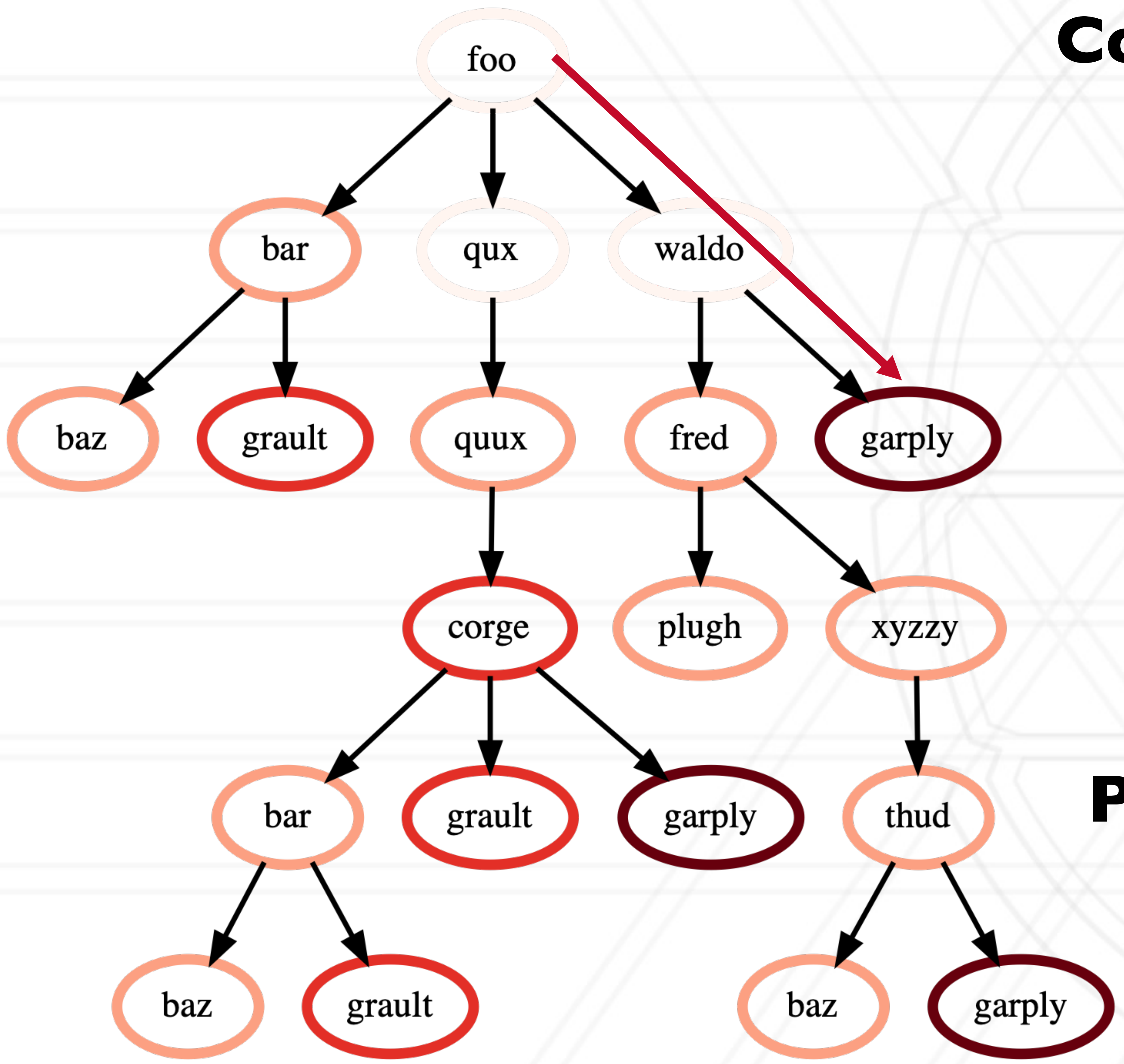
Gprof data in hpctView

Calling contexts, trees, and graphs

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



Calling context trees, call graphs, …



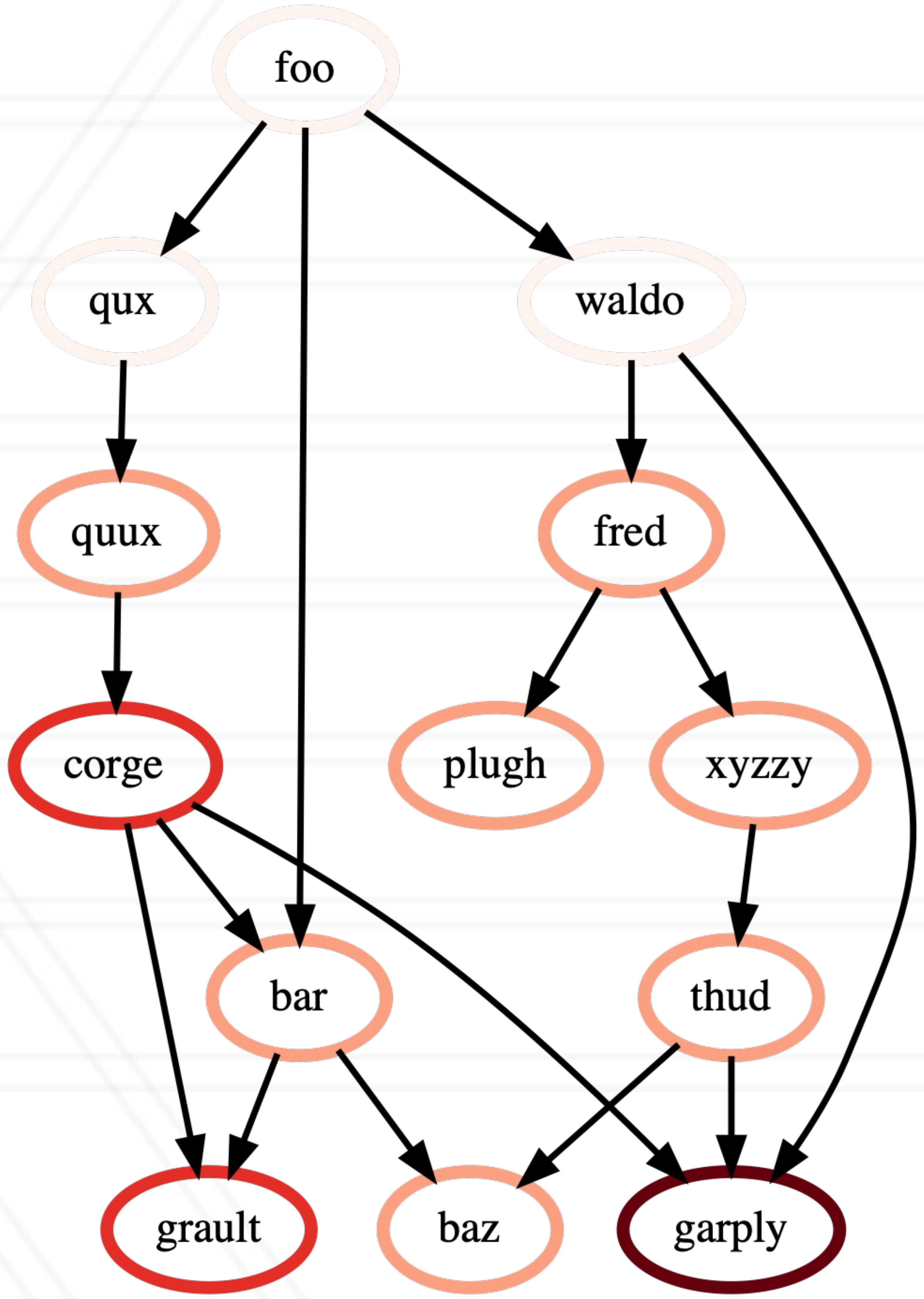
Calling context tree (CCT)

Contextual information

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

Performance Metrics

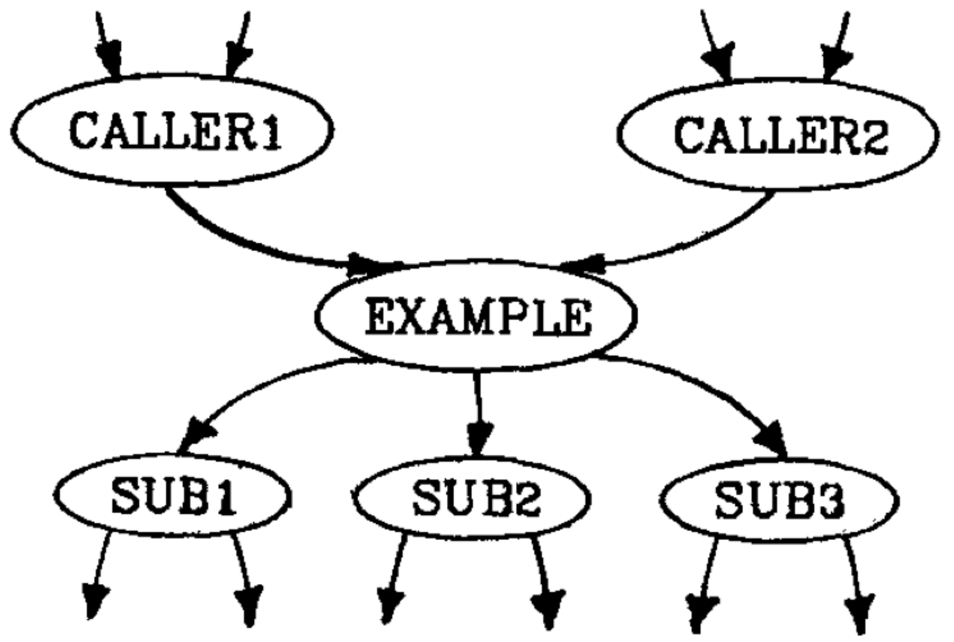
- Time
- Flops
- Cache misses

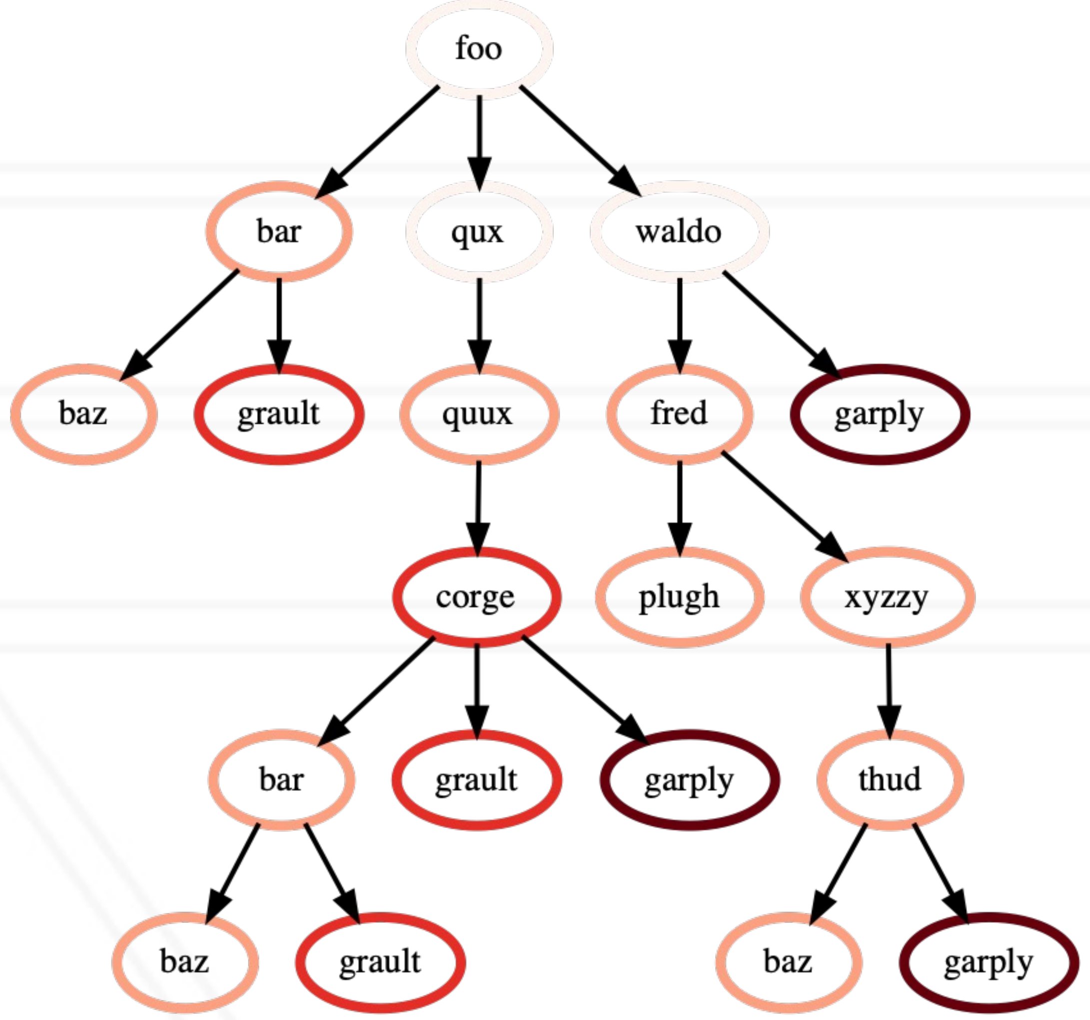


Call graph

Output of profiling tools

- Flat profile: Listing of all functions with counts and execution times
- Call graph profile
- Calling context tree





Hatchet

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

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

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

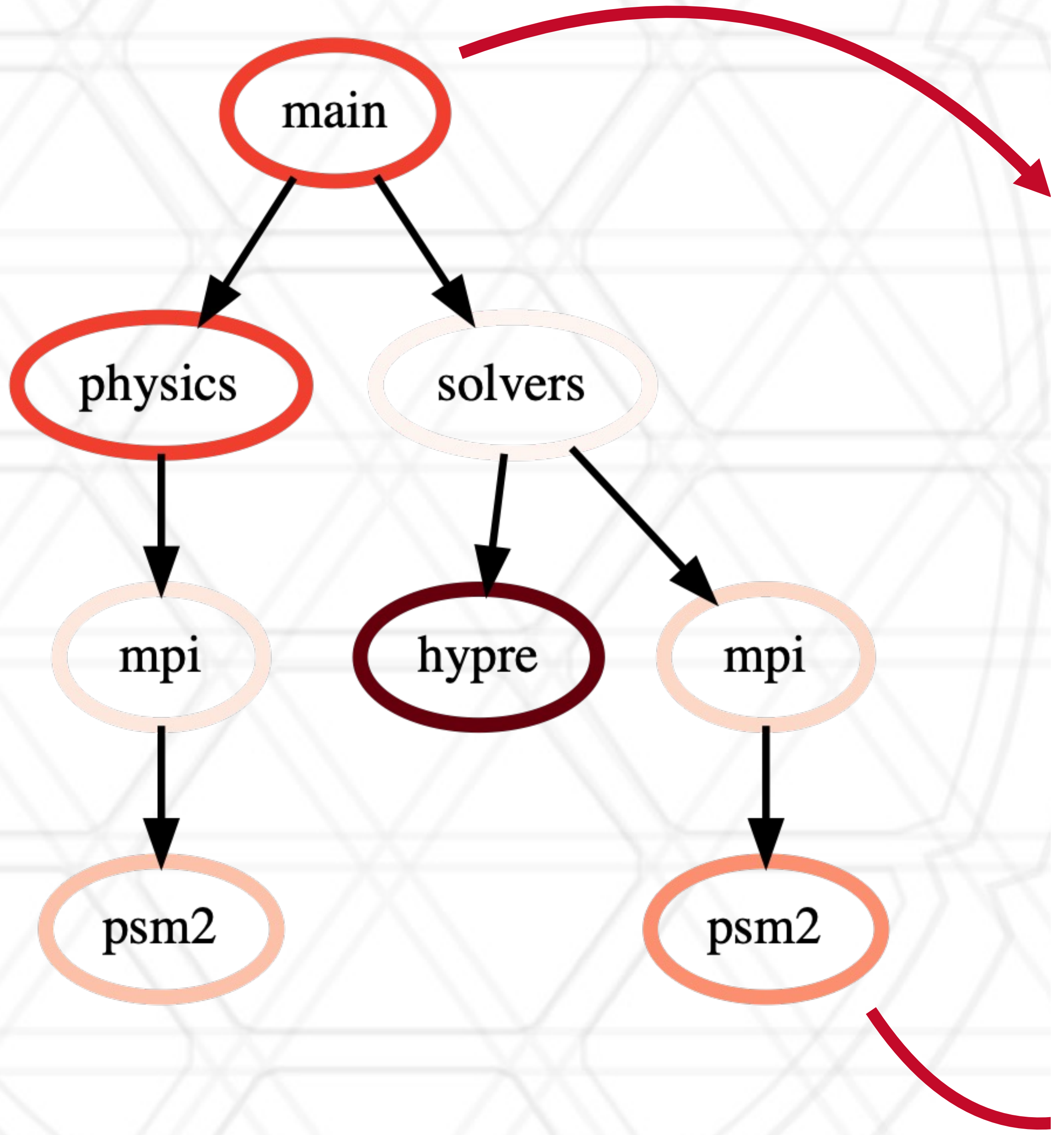
Columns

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

Central data structure: 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



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

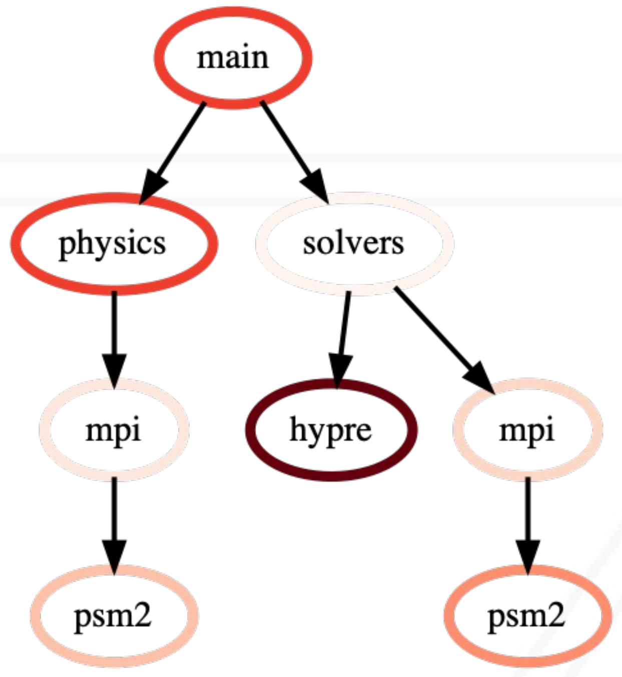
node	name	nid	node	time	time (inc)
main	main	0	main	40.0	200.0
physics	physics	1	physics	40.0	60.0
psm2	psm2	3	psm2	15.0	15.0
hypre	hypre	5	hypre	65.0	65.0
psm2	psm2	7	psm2	25.0	25.0

Graph operation: squash

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

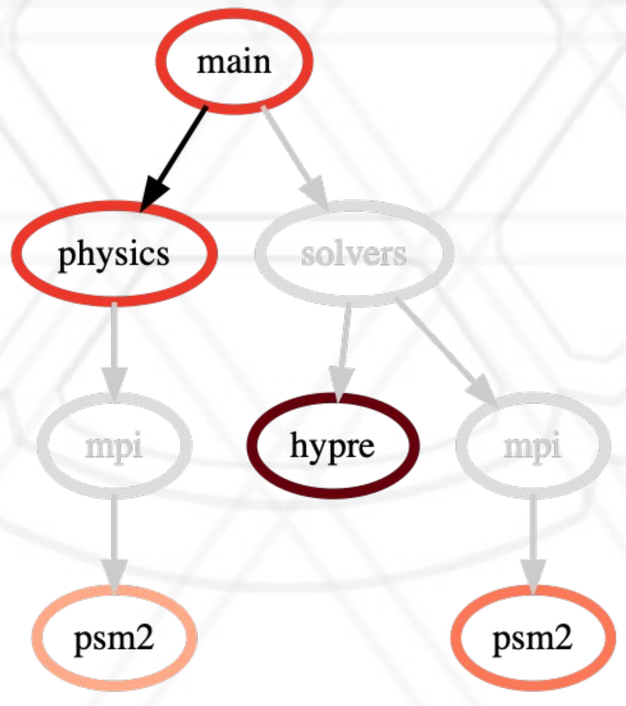
```
squashed_gf = filtered_gf.squash()
```

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



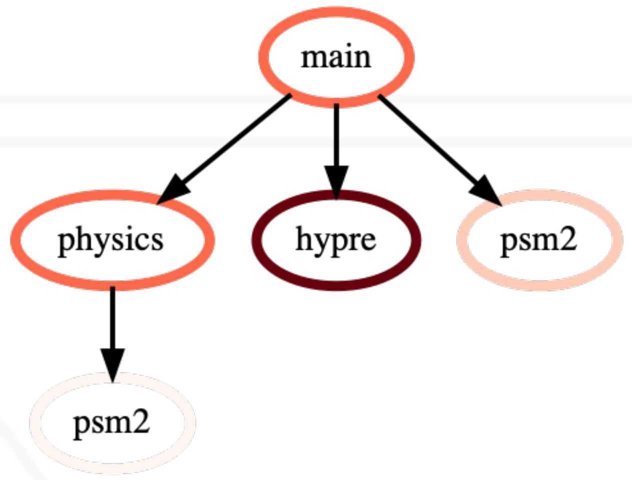
filter

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

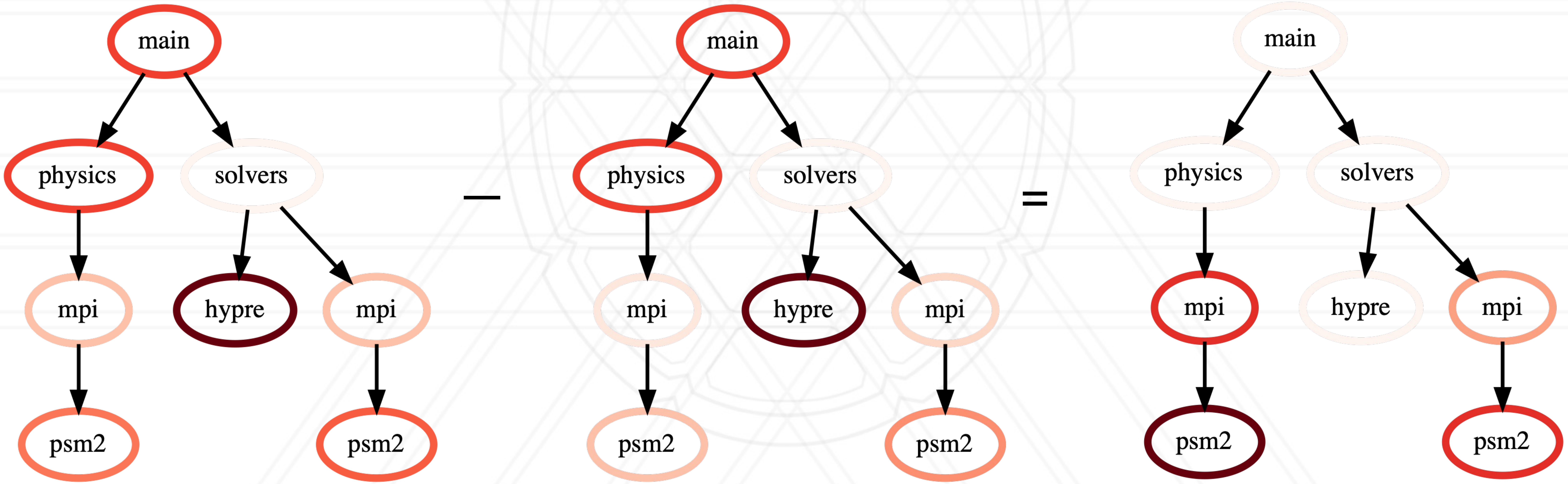
node	name	nid	node	time	time (inc)
main	main	0	main	40.0	200.0
physics	physics	1	physics	40.0	60.0
psm2	psm2	3	psm2	15.0	15.0
hypre	hypre	5	hypre	65.0	65.0
psm2	psm2	7	psm2	25.0	25.0



Graphframe operation: subtract

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

https://hatchet.readthedocs.io

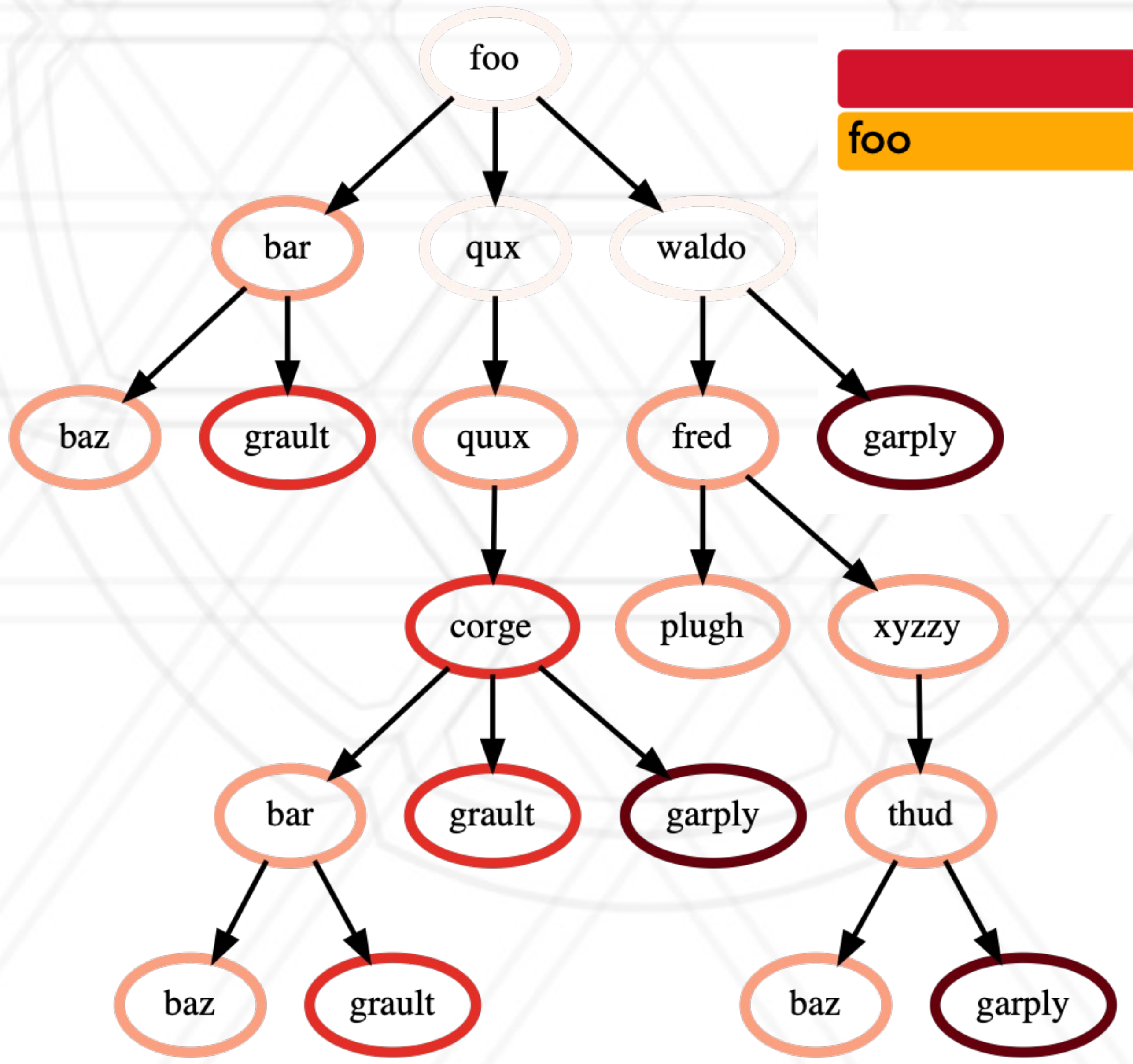


Visualizing *small* graphs

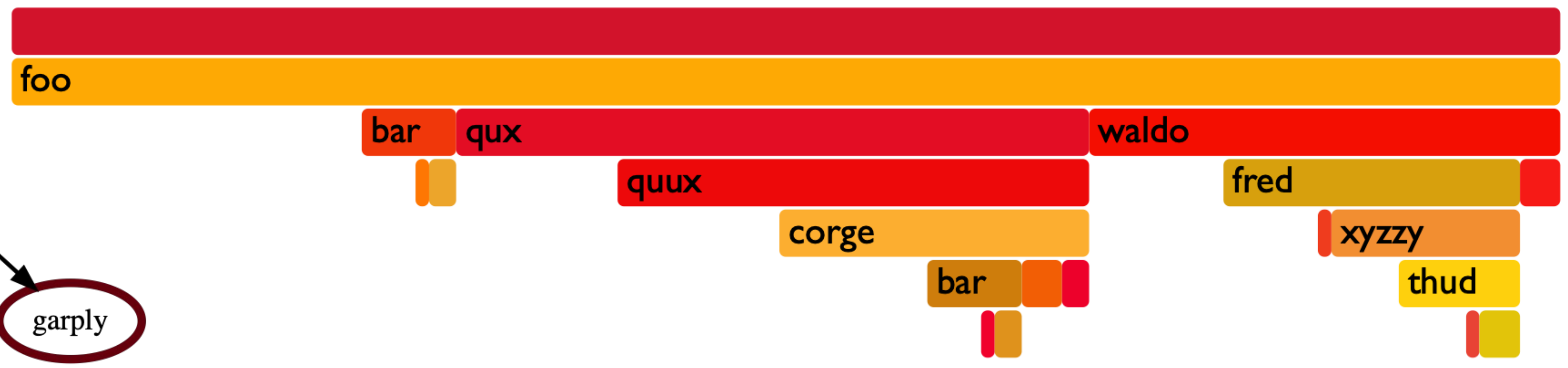
```python
print(gf.tree(color=True))
```

```
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

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'], asce
print(sorted_df)
```

name	nid	time	time (inc)
<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

```python
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

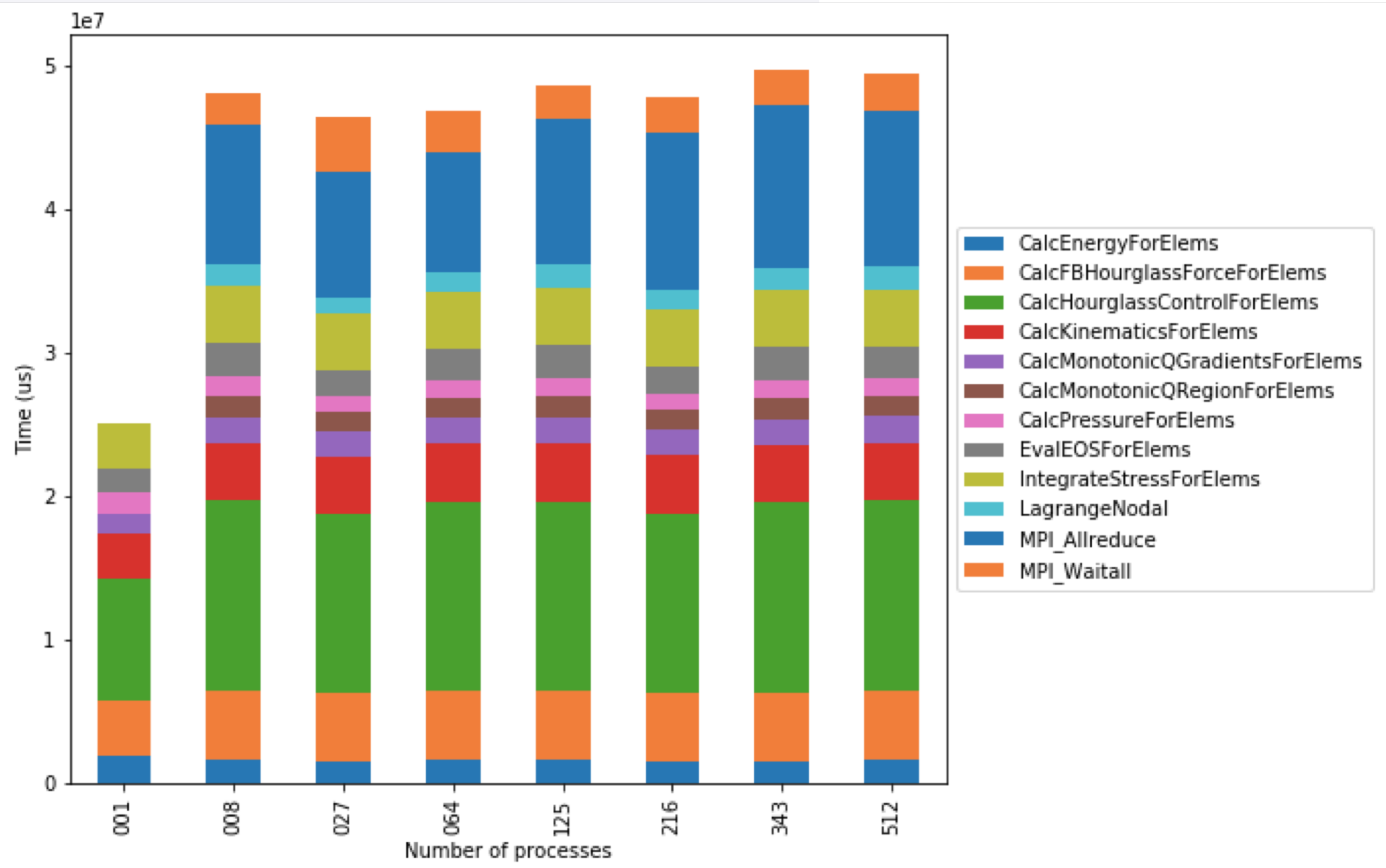
Example 3: Scaling study

```
datasets = glob.glob('lulesh*.json')
datasets.sort()

dataframes = []
for dataset in datasets:
    gf = ht.GraphFrame.from_caliper(datas
    gf.drop_index_levels()

    num_pes = re.match('(.*)-(\d+)(.*)',
    gf.dataframe['pes'] = num_pes
    filtered_gf = gf.filter(lambda x: x['
    dataframes.append(filtered_gf.datafra

result = pd.concat(dataframes)
pivot_df = result.pivot(index='pes', colu
pivot_df.loc[:,:].plot.bar(stacked=True, figsize=(10,7))
```



UNIVERSITY OF
MARYLAND