Performance Modeling, Analysis, and Tools
Abhinav Bhavele, Department of Computer Science
Announcements

• Assignment 1 is posted
  • Due on Oct 10 11:59 pm

• 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
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

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

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

\[
\text{Speedup} = \frac{1}{(1 - f) + \frac{f}{p}}
\]
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

2  8  3  5  7  4  1  6

2  10  11  8  12  11  5  7

2  10  13  18  23  19  17  18

2  10  13  18  25  29  30  36
Parallel prefix sum
Parallel prefix sum for $n \gg p$

- Assign $n/p$ elements (block) to each process
- Perform prefix sum on these blocks on each process locally
  - Number of calculations:
- Then do parallel algorithm with partial prefix sums
  - Number of phases:
  - Total number of calculations:
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:
  - Total number of calculations:
Parallel prefix sum for $n \gg p$

- 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:
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

\( \frac{1}{g} = \text{bandwidth} \)
alpha + n * beta model

• Another model for communication

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

\( \alpha \): latency

\( n \): size of message

\( 1/\beta \): 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)

\[ p \times t_p = t_1 + t_o \]

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

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

\[ \sqrt{n} \]

• 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: \(2 \times \sqrt{n}\)
  \[t_o = \frac{2 \times \sqrt{n}}{\frac{n}{p}} = \frac{2 \times p}{\sqrt{n}}\]

• 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: \( 2 \times \sqrt{n} \)
  - Time:
    \[ \frac{t_0}{t_1} = \frac{2 \times \sqrt{n}}{\frac{n}{p}} = \frac{2 \times p}{\sqrt{n}} \]

- **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: \( 4 \times \frac{\sqrt{n}}{\sqrt{p}} \)
  \[
  \frac{t_o}{t_1} = \frac{4 \times \frac{\sqrt{n}}{\sqrt{p}}}{\frac{n}{p}} = \frac{4 \times \sqrt{p}}{\sqrt{n}}
  \]
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;
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
- 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
- Events: function calls, MPI events, etc.

Tracing tools

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

Tracing tools

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

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 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)
Calling context trees, call graphs, ...

Calling context tree (CCT)
Calling context trees, call graphs, ...

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

Calling context tree (CCT)
Calling context trees, call graphs, …

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, ...

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

Performance Metrics
- Time
- Flops
- Cache misses

Calling context tree (CCT)

Call graph
Output of profiling tools

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

![Diagram of call graph profile]

Abhinav Bhatele (CMSC416 / CMSC818X)
Hatchet

- 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

<table>
<thead>
<tr>
<th></th>
<th>node</th>
<th>name</th>
<th>time (inc)</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{'name': 'main'}</td>
<td>main</td>
<td>200.0</td>
<td>10.0</td>
</tr>
<tr>
<td>1</td>
<td>{'name': 'physics'}</td>
<td>physics</td>
<td>60.0</td>
<td>40.0</td>
</tr>
<tr>
<td>2</td>
<td>{'name': 'mpi'}</td>
<td>mpi</td>
<td>20.0</td>
<td>5.0</td>
</tr>
<tr>
<td>3</td>
<td>{'name': 'psm2'}</td>
<td>psm2</td>
<td>15.0</td>
<td>30.0</td>
</tr>
<tr>
<td>4</td>
<td>{'name': 'solvers'}</td>
<td>solvers</td>
<td>100.0</td>
<td>10.0</td>
</tr>
<tr>
<td>5</td>
<td>{'name': 'hype'}</td>
<td>hype</td>
<td>65.0</td>
<td>30.0</td>
</tr>
<tr>
<td>6</td>
<td>{'name': 'mpi'}</td>
<td>mpi</td>
<td>35.0</td>
<td>20.0</td>
</tr>
<tr>
<td>7</td>
<td>{'name': 'psm2'}</td>
<td>psm2</td>
<td>25.0</td>
<td>60.0</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>index</th>
<th>node</th>
<th>name</th>
<th>time (inc)</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{'name': 'main'}</td>
<td>main</td>
<td>200.0</td>
<td>10.0</td>
</tr>
<tr>
<td>1</td>
<td>{'name': 'physics'}</td>
<td>physics</td>
<td>60.0</td>
<td>40.0</td>
</tr>
<tr>
<td>2</td>
<td>{'name': 'mpi'}</td>
<td>mpi</td>
<td>20.0</td>
<td>5.0</td>
</tr>
<tr>
<td>3</td>
<td>{'name': 'psm2'}</td>
<td>psm2</td>
<td>15.0</td>
<td>30.0</td>
</tr>
<tr>
<td>4</td>
<td>{'name': 'solvers'}</td>
<td>solvers</td>
<td>100.0</td>
<td>10.0</td>
</tr>
<tr>
<td>5</td>
<td>{'name': 'hype'}</td>
<td>hype</td>
<td>65.0</td>
<td>30.0</td>
</tr>
<tr>
<td>6</td>
<td>{'name': 'mpi'}</td>
<td>mpi</td>
<td>35.0</td>
<td>20.0</td>
</tr>
<tr>
<td>7</td>
<td>{'name': 'psm2'}</td>
<td>psm2</td>
<td>25.0</td>
<td>60.0</td>
</tr>
</tbody>
</table>
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
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
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

![GraphFrame Diagram]

With all of these components, a GraphFrame models the edge data as a directed graph. Each node in the graph can have one or multiple parents and children. Hatchet's directed graph could be connected or have multiple disconnected components. Each entity in the graph, such as a callsite, has attributes from each node's FrameID. For example, in the output from tools such as callgrind is already in the form of a DAG. A tree can create edges, turning the tree into a graph. Additionally, a tree generated from call paths can be used, as well.

Because of the way we have architected the Hatchet structured data, we can insert Node objects directly into the pandas index Graph, we can add columns for each associated performance metric (including mpi and nid) for HPCToolkit. This allows us to use regular pandas operations such as selection, sorting, and merging. We can also use regular pandas column operations, as well.

The rows of the data frame are indexed and sorted using their basic attribute. Thus, each node in the call tree or graph has metrics per-MPI process and/or thread. In such cases, each node references, and that any operations that would modify a graph node references, and that any operations that would modify a graph is that client code can extract a subset of a Dataframe and hand it to graph nodes into the Dataframe. The risk this poses in our API is that client code can extract a subset of a Dataframe and hand it to graph nodes into the Dataframe.
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

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

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

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

<table>
<thead>
<tr>
<th>node</th>
<th>name</th>
<th>nid</th>
<th>node</th>
<th>time</th>
<th>time (inc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>main</td>
<td>0</td>
<td>main</td>
<td>40.0</td>
<td>200.0</td>
</tr>
<tr>
<td>physics</td>
<td>physics</td>
<td>1</td>
<td>physics</td>
<td>40.0</td>
<td>60.0</td>
</tr>
<tr>
<td>mpi</td>
<td>mpi</td>
<td>2</td>
<td>mpi</td>
<td>5.0</td>
<td>20.0</td>
</tr>
<tr>
<td>psm2</td>
<td>psm2</td>
<td>3</td>
<td>psm2</td>
<td>15.0</td>
<td>15.0</td>
</tr>
<tr>
<td>solvers</td>
<td>solvers</td>
<td>4</td>
<td>solvers</td>
<td>0.0</td>
<td>100.0</td>
</tr>
<tr>
<td>hypre</td>
<td>hypre</td>
<td>5</td>
<td>hypre</td>
<td>65.0</td>
<td>65.0</td>
</tr>
<tr>
<td>mpi</td>
<td>mpi</td>
<td>6</td>
<td>mpi</td>
<td>10.0</td>
<td>35.0</td>
</tr>
<tr>
<td>psm2</td>
<td>psm2</td>
<td>7</td>
<td>psm2</td>
<td>25.0</td>
<td>25.0</td>
</tr>
</tbody>
</table>
Graph operation: squash

filtered_gf = gf.filter(lambda x: x['time'] > 10.0)
We now describe some of the important operators provided by Hatchet. In this case, the applied function returns all rows where the applied function returns all rows where the conditions are true. The returned dataframe to only modify the dataframe, some only modify the graph, and others perform the aggregation over all MPI processes or threads at the root level.

### The Hatchet API

Hatchet provides readers for several input formats to support most parallel performance analysis tools in the HPC community. These include HPCToolkit, Caliper, and the CCT.

#### 4.1 Dataframe Operations

- **Filter**: When one or more filter operations are performed on the dataframe, they also make connects between the graph and dataframe objects. Typically, the user will perform a filter operation on the dataframe. As shown in Figure 5, there are nodes in the graph that do not return any rows when filtered, aggregated, pruned, etc. Filter is one of the operations that leads to the graph object and dataframe objects consistent again.

- **Squash**: The squash operation is typically performed by the user after filtering the dataframe to make the graph and dataframe objects becoming inconsistent. After a squash operation, all the columns in the ed), all the columns in the dataframe that store inclusive values of some metric become inaccurate, since the parent-child connections are modified. Additionally, a squash operation will make the values in all columns containing inclusive metrics inaccurate, since the parent-child connections are modified.

- **Drop Index Levels**: The drop_index_levels function. Essentially, this performs a drop_index_levels function on the graphframe after a squash operation.

### Example Code

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

### Example Table

<table>
<thead>
<tr>
<th>node</th>
<th>time</th>
<th>time (inc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>40.0</td>
<td>200.0</td>
</tr>
<tr>
<td>physics</td>
<td>40.0</td>
<td>60.0</td>
</tr>
<tr>
<td>mpi</td>
<td>5.0</td>
<td>20.0</td>
</tr>
<tr>
<td>psm2</td>
<td>15.0</td>
<td>15.0</td>
</tr>
<tr>
<td>solvers</td>
<td>0.0</td>
<td>100.0</td>
</tr>
<tr>
<td>hypre</td>
<td>65.0</td>
<td>65.0</td>
</tr>
<tr>
<td>mpi</td>
<td>10.0</td>
<td>35.0</td>
</tr>
<tr>
<td>psm2</td>
<td>25.0</td>
<td>25.0</td>
</tr>
</tbody>
</table>

### Example Diagram

![Graph operation: squash diagram](image)
Graph operation: squash

filtered_gf = gf.filter(lambda x: x['time'] > 10.0)
squashed_gf = filtered_gf.squash()
Graph operation: squash

\[
\text{filtered}_g = \text{gf}.\text{filter}(\lambda x: x[\text{time}] > 10.0)
\]

\[
\text{squashed}_g = \text{filtered}_g.\text{squash}()
\]
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

\[
gf1 = \text{ht.GraphFrame.from_literal}(...) \\
gf2 = \text{ht.GraphFrame.from_literal}(...) \\
gf2 \rightarrow gf1
\]

GraphFrame operations call union on the graphs, and then reindexes the DataFrames if necessary. Once the graphs are unified, it requires the two graphs to be identical. It applies union and subtract operations on two GraphFrames with unequal graphs, a union operation computes the element-wise sum of two DataFrames, and a subtract operation computes the element-wise difference of two DataFrames. In the case where the two graphs are not identical, Hatchet uses an in-place subtraction. Similar to the union operation, it calls union on the respective graphs. It creates a new GraphFrame with a copy of the original GraphFrame, or it modifies the in-place subtraction. This checks whether two graphs have the same nodes and structure. It applies union and subtract operations on two GraphFrames with unequal graphs, a union operation computes the element-wise sum of two DataFrames, and a subtract operation computes the element-wise difference of two DataFrames. In the case where the two graphs are not identical, Hatchet uses an in-place subtraction. A squash operation creates a new DataFrame in addition to the reindexed DataFrame, but its index points to nodes in the new graph. Additionally, a squash operation will make the values in all columns of the reindexed DataFrame accurate again.
Visualizing small graphs

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

```
0.000 foo
├ 5.000 bar
│ └ 5.000 baz
│   └ 10.000 grault
├ 0.000 gux
│ └ 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

\[
\text{print}(gf.\text{tree}(\text{color}=\text{True}))
\]

```
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
│           └── 15.000 garply
├── 0.000 waldo
│   ├── 5.000 fred
│       ├── 5.000 plugh
│       │   └── 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())
```
Visualizing small graphs

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

```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())
```

```
0.000 foo
  | 5.000 bar
  |  | 5.000 baz
  |  | 10.000 grault
  | 0.000 quux
  |  | 5.000 quux
  |  |  | 10.000 corge
  |  |  | 5.000 bar
  |  |  | 5.000 baz
  |  | 10.000 grault
  |  | 15.000 grault
  | 0.000 waldo
  | 5.000 fred
  |  | 5.000 plugh
  |  | 5.000 xzyzy
  |  | 5.000 thud
  |  | 5.000 baz
  | 15.000 garply
```

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'], 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'], 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'], ascending=True)  
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)
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)

<table>
<thead>
<tr>
<th>node</th>
<th>name</th>
<th>nid</th>
<th>time</th>
<th>time (inc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimIncrement</td>
<td>TimIncrement</td>
<td>25.0</td>
<td>8.505048e+06</td>
<td>8.505048e+06</td>
</tr>
<tr>
<td>CalcQForElems</td>
<td>CalcQForElems</td>
<td>16.0</td>
<td>4.455672e+06</td>
<td>5.189453e+06</td>
</tr>
<tr>
<td>CalcHourglassControlForElems</td>
<td>CalcHourglassControlForElems</td>
<td>7.0</td>
<td>3.888798e+06</td>
<td>4.755817e+06</td>
</tr>
<tr>
<td>LagrangeNodal</td>
<td>LagrangeNodal</td>
<td>3.0</td>
<td>1.986046e+06</td>
<td>8.828475e+06</td>
</tr>
<tr>
<td>CalcForceForNodes</td>
<td>CalcForceForNodes</td>
<td>4.0</td>
<td>1.017857e+06</td>
<td>6.842429e+06</td>
</tr>
</tbody>
</table>
Example 3: Scaling study

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

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

    num_pes = re.match('(\d+)(.*)', dataset).group(2)
gf.dataframe['pes'] = num_pes
    filtered_gf = gf.filter(lambda x: x['time'] > 1e6)
dataframes.append(filtered_gf.dataframe)

result = pd.concat(dataframes)
pivot_df = result.pivot(index='pes', columns='name', values='time')
pivot_df.loc[::, :].plot.bar(stacked=True, figsize=(10,7))
Example 3: Scaling study

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

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

    num_pes = re.match('(.*)-(\d+)(.*)', dataset).group(2)
gf.dataframe['pes'] = num_pes
filtered_gf = gf.filter(lambda x: x['time'] > 1e6)
dataframes.append(filtered_gf.dataframe)

result = pd.concat(dataframes)
pivot_df = result.pivot(index='pes', columns='name', values='time')
pivot_df.loc[::,:].plot.bar(stacked=True, figsize=(10,7))
```
Example 3: Scaling study

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

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

    num_pes = re.match('(.*)-(\d+)(.*)', dataset).
    gf.dataframe['pes'] = num_pes
    filtered_gf = gf.filter(lambda x: x['pes'], lambda x: x['pes'] > 1e6)
    dataframes.append(filtered_gf.dataframe)

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

Abhinav Bhatete (CMSC416 / CMSC818X)