Announcements

• Assignment 1 is now posted online
  • Due on: Oct 2, 2023 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

• Quiz 1 will be posted on Wed Sep 20, 2023 11:59 pm
  • You'll have 24 hours to take it
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
  - 2$n$ numbers on 2 processes
  - 4$n$ 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

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>11</td>
<td>8</td>
<td>12</td>
<td>11</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>13</td>
<td>18</td>
<td>23</td>
<td>19</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>13</td>
<td>18</td>
<td>25</td>
<td>29</td>
<td>30</td>
<td>36</td>
</tr>
</tbody>
</table>
Parallel prefix sum

```
<table>
<thead>
<tr>
<th>2</th>
<th>8</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>4</th>
<th>1</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10</td>
<td>11</td>
<td>8</td>
<td>12</td>
<td>11</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>13</td>
<td>18</td>
<td>23</td>
<td>19</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>13</td>
<td>18</td>
<td>25</td>
<td>29</td>
<td>30</td>
<td>36</td>
</tr>
</tbody>
</table>
```
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:
- Then do parallel algorithm with partial prefix sums
  - Number of phases:
  - Total number of calculations:
  - Communication:
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:
  - Communication:
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:
  - Communication:
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: \( \log(p) \times \frac{n}{p} \)
  - Communication: \( \frac{8n}{p \log(p)} \times \frac{n}{p} \)
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

\( L \) = latency or delay

\( o \) = overhead (processor busy in communication)

\( g \) = gap (between successive sends/recvs)

\( P \) = number of processors / processes

\( 1/g \) = bandwidth

Figure 2.
alpha + n * beta model

- Another model for communication

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

\( \alpha \): latency

\( n \): size of message

\( \frac{1}{\beta} \): 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

- **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: \( \frac{\sqrt{n}}{\sqrt{p}} \times \frac{\sqrt{n}}{\sqrt{p}} = \frac{n}{p} \)
  - Communication

\[
\sqrt{n} \quad \frac{\sqrt{n}}{p} \quad \frac{\sqrt{n}}{\sqrt{p}} \]
Isoefficiency analysis

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

- 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
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;

Phase 1 took 2.45 s

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

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 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
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 trees, call graphs, …
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

The output of profiling tools includes:

- **Flat profile**: This provides a listing of all functions with counts and execution times. It gives a straightforward view of the program's performance, showing which functions are taking the most time.

- **Call graph profile**: This view shows the call structure of the program, detailing how different functions call into one another. It helps in understanding the flow of control and dependencies between different parts of the program.

- **Calling context tree**: This tree-like representation shows the hierarchical structure of function calls, making it easy to see how different parts of the code are interacting with each other.

These outputs are crucial for understanding the performance characteristics of a program and identifying bottlenecks or areas for optimization.
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>Columns</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': 'hypre'}</td>
<td>hypre</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.

## Example DataFrame

<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

• MultiIndex enables working with high-dimensional data in a 2D data structure

<table>
<thead>
<tr>
<th>Rows</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>
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
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)
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>
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>

<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>psm2</td>
<td>psm2</td>
<td>3</td>
<td>psm2</td>
<td>15.0</td>
<td>15.0</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>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

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

**Table: Dataframe Before and After Filter Operation**

<table>
<thead>
<tr>
<th>node</th>
<th>nid</th>
<th>parent</th>
<th>name</th>
<th>time</th>
<th>time (in)</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>main</td>
<td>main</td>
<td>main</td>
<td>40.0</td>
<td>200.0</td>
</tr>
<tr>
<td>physics</td>
<td>physics</td>
<td>mpi</td>
<td>physics</td>
<td>40.0</td>
<td>60.0</td>
</tr>
<tr>
<td>mpi</td>
<td>mpi</td>
<td>mpi</td>
<td>mpi</td>
<td>5.0</td>
<td>20.0</td>
</tr>
<tr>
<td>psm2</td>
<td>psm2</td>
<td>psm2</td>
<td>psm2</td>
<td>15.0</td>
<td>15.0</td>
</tr>
<tr>
<td>solvers</td>
<td>solvers</td>
<td>mpi</td>
<td>solvers</td>
<td>0.0</td>
<td>100.0</td>
</tr>
<tr>
<td>hypre</td>
<td>hypre</td>
<td>mpi</td>
<td>hypre</td>
<td>65.0</td>
<td>65.0</td>
</tr>
<tr>
<td>mpi</td>
<td>mpi</td>
<td>mpi</td>
<td>mpi</td>
<td>10.0</td>
<td>35.0</td>
</tr>
<tr>
<td>psm2</td>
<td>psm2</td>
<td>psm2</td>
<td>psm2</td>
<td>25.0</td>
<td>25.0</td>
</tr>
</tbody>
</table>
Graph operation: squash

\[
\text{filtered}_gf = \text{gf.filter}(\text{lambda } x: x['time'] > 10.0)
\]
Graph operation: squash

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
squashed_gf = filtered_gf.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

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

https://hatchet.readthedocs.io
check_equivalence: This checks whether two graphs are exactly equivalent or not in their structures by comparing the call paths of the respective children. If they are equivalent, it returns true, otherwise it returns false.

union: The union function takes two graphs and creates a unified graph, preserving as much structure of the original graphs as possible. Typically, a union is called if the structures of two graphs are different. This operation is useful when we wish to perform an add or subtract operation on two graphframes and the graphs are not structurally equivalent. In this case, a union is performed first before the add or subtract operation.

4.3 Graphframe Operations

copy: The copy operation returns a copy of a graphframe by creating a copy of the dataframe object and the graph object, which in turn involves cloning all the nodes in the graph. Creating a copy enables the user to modify a copy of the graphframe object, while keeping the original object unchanged. This is useful for example, in the case of add and subtract, when there are two operands and the result needs to return a new graphframe.

add: Assuming the graphs in two graphframes are identical (i.e., check_equivalence returns true), the add operation computes the sum of two dataframes column-wise. In the case where the two graphs are not identical, union (described above) is applied first to create a unified graph before performing the sum. The add operation returns a new resulting graphframe or modifies one of the graphframes in place in the case of the following addition assignment: (a += b).

subtract: The subtract operation is similar to the add operation in that it requires the two graphs to be identical. Once the graphs are structurally equivalent, the subtract operation computes the difference between the two dataframes column-wise. The subtract operation returns a new resulting graphframe or modifies one of the graphframes in place in the case of the subtraction assignment (a -= b). Figure 6 shows the subtraction of one graphframe from another and the graph for the resulting graphframe.

4.4 Visualizing Output

Hatchet provides its own visualization as well as support for two other visualizations of the structured data stored in the graph object. The native visualization in Hatchet is a string that can be printed to the terminal to display the graph. Hatchet can also output the graph in the DOT format or a folded stack used by FlameGraph. The dot utility in Graphviz produces a hierarchical drawing of directed graphs, particularly useful for showing the direction of the edges. Flame graphs are useful for quickly identifying the performance bottleneck, that is the box with the largest width. The y-axis of the FlameGraph represents the call stack depth. Figure 7 shows the same Hatchet graph presented in the three supported visualizations: terminal output (left), DOT (right), and FlameGraph (bottom). For particularly large graphs, these visual representations can be useful for quickly identifying caller-callee relationships. However, identifying performance bottlenecks or load imbalance might be easier in the dataframe.

```
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

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

```

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

---

**Figure 6:** Subtraction operation on two graphframes (resulting graph at the bottom).

**Figure 7:** Visualization outputs supported in Hatchet including terminal output (left), DOT (right), and others.

---

4.3 Graphframe Operations

- **union**: Takes two graphs and creates a union graph, preserving as much structure of the original graphs as possible. Typically, a union is called if the structures of two graphs are diiferent. This operation is useful when we wish to perform an operation that combines graphs. For particularly large graphs, smart parallelization or load balancing might be easier in the dataframe.

```
gf2 -= gf1
```

The subtract operation is similar to the add operation (described above) as applied to the graphs. Performing a subtract operation requires the two graphs to be identical. Once the graphs are not identical, the result needs to return a new graphframe.

```
check_equivalence = gf1.equals(gf2)
```

This checks whether two graphs are exactly equivalent or not in their structures by comparing the call paths of the respective children. If they are equivalent, it returns true, otherwise it returns false.

Abhinav Bhatele (CMSC416 / CMSC616)
Visualizing small graphs

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

```
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
      |- 10.000 grault
    |- 15.000 garply
  |- 0.000 waldo
    |- 5.000 fred
      |- 5.000 plugh
      |- 5.000 xzyzy
      |- 5.000 thud
      |- 5.000 baz
    |- 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'], 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_hpc Toolkit('kripke')
gf.drop_index_levels()

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

<table>
<thead>
<tr>
<th>name</th>
<th>nid</th>
<th>time (sec)</th>
<th>time (inc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;unknown file&gt; [kripke]:0</td>
<td>17234</td>
<td>1.82528e+08</td>
<td>1.82528e+08</td>
</tr>
<tr>
<td>Kernel_3d_DGZ::scattering</td>
<td>60</td>
<td>7.68933e+07</td>
<td>7.896253e+07</td>
</tr>
<tr>
<td>Kernel_3d_DGZ::LTimes</td>
<td>30</td>
<td>5.01049e+07</td>
<td>5.240528e+07</td>
</tr>
<tr>
<td>Kernel_3d_DGZ::LPlusTimes</td>
<td>115</td>
<td>4.94770e+07</td>
<td>5.104498e+07</td>
</tr>
<tr>
<td>Kernel_3d_DGZ::sweep</td>
<td>981</td>
<td>5.01886e+06</td>
<td>5.018862e+06</td>
</tr>
<tr>
<td>memset.S:99</td>
<td>3773</td>
<td>3.16898e+06</td>
<td>3.168982e+06</td>
</tr>
<tr>
<td>memset.S:101</td>
<td>3970</td>
<td>2.12089e+06</td>
<td>2.120895e+06</td>
</tr>
<tr>
<td>Grid_Data:particleEdit</td>
<td>1201</td>
<td>1.131263e+06</td>
<td>1.249157e+06</td>
</tr>
<tr>
<td>&lt;unknown file&gt; [libpsm2.so.2.1]:0</td>
<td>324763</td>
<td>9.733415e+05</td>
<td>9.733415e+05</td>
</tr>
<tr>
<td>memset.S:98</td>
<td>3787</td>
<td>6.19777e+05</td>
<td>6.197776e+05</td>
</tr>
</tbody>
</table>
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

```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)
```

<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>

Abhinav Bhatel (CMSC416 / CMSC616)
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

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

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']
    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))