Performance Modeling, Analysis, and Tools

Abhinav Bhatel, Department of Computer Science

Introduction to Parallel Computing (CMSC416 / CMSC616)
Announcements

• 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
  - 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 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

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 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 per process:
- 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 process: $\frac{n}{p}$
- 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): $8n/p$
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 per process: $\frac{n}{p}$
- 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 \( n \gg p \)

- Assign \( n/p \) elements (block) to each process
- Perform prefix sum on these blocks on each process locally
  - Number of calculations per process: \( \frac{n}{p} \)
- 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 process: $\frac{n}{p}$
- 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

\( \frac{1}{g} = \text{bandwidth} \)
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 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)

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

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

\[
\frac{\sqrt{n}}{\sqrt{p}} \times \frac{\sqrt{n}}{\sqrt{p}} = \frac{n}{p}
\]
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: \( 4 \times \frac{\sqrt{n}}{\sqrt{p}} \)
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: \( 4 \times \frac{\sqrt{n}}{\sqrt{p}} \)

  \[
  \frac{t_o}{t_1} = \frac{4 \times \sqrt{n}}{\frac{n}{p}} = \frac{4 \times \sqrt{p}}{\sqrt{n}}
  \]
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: \( 4 \times \frac{\sqrt{n}}{\sqrt{p}} \)

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

We only consider communication for \( t_o \)
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;

Phase 1 took 2.45 s

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

Phase 2 took 11.79 s

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

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 in hpctView](image-url)
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 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 invoked functions with counts and execution times
- Call graph profile: unique node per function
- Calling context tree: unique node per calling context
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

<table>
<thead>
<tr>
<th>Rows</th>
<th>Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>name</td>
</tr>
<tr>
<td>0</td>
<td>{'name': 'main'}</td>
</tr>
<tr>
<td>1</td>
<td>{'name': 'physics'}</td>
</tr>
<tr>
<td>2</td>
<td>{'name': 'mpi'}</td>
</tr>
<tr>
<td>3</td>
<td>{'name': 'psm2'}</td>
</tr>
<tr>
<td>4</td>
<td>{'name': 'solvers'}</td>
</tr>
<tr>
<td>5</td>
<td>{'name': 'hypre'}</td>
</tr>
<tr>
<td>6</td>
<td>{'name': 'mpi'}</td>
</tr>
<tr>
<td>7</td>
<td>{'name': 'psm2'}</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></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>Index</th>
<th>Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>name</td>
</tr>
<tr>
<td>0</td>
<td>{'name': 'main'}</td>
</tr>
<tr>
<td>1</td>
<td>{'name': 'physics'}</td>
</tr>
<tr>
<td>2</td>
<td>{'name': 'mpi'}</td>
</tr>
<tr>
<td>3</td>
<td>{'name': 'psm2'}</td>
</tr>
<tr>
<td>4</td>
<td>{'name': 'solvers'}</td>
</tr>
<tr>
<td>5</td>
<td>{'name': 'hype'}</td>
</tr>
<tr>
<td>6</td>
<td>{'name': 'mpi'}</td>
</tr>
<tr>
<td>7</td>
<td>{'name': 'psm2'}</td>
</tr>
</tbody>
</table>
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
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

<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 object

Dataframe
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)
Graph operation: squash

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

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

filtered_gf

<table>
<thead>
<tr>
<th>name</th>
<th>rid</th>
<th>node</th>
<th>time</th>
<th>time (inc)</th>
</tr>
</thead>
<tbody>
<tr>
<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>1</td>
<td>physics</td>
<td>40.0</td>
</tr>
<tr>
<td>mpi</td>
<td>mpi</td>
<td>2</td>
<td>mpi</td>
<td>5.0</td>
</tr>
<tr>
<td>psm2</td>
<td>psm2</td>
<td>3</td>
<td>psm2</td>
<td>15.0</td>
</tr>
<tr>
<td>solvers</td>
<td>solvers</td>
<td>4</td>
<td>solvers</td>
<td>0.0</td>
</tr>
<tr>
<td>hypre</td>
<td>hypre</td>
<td>5</td>
<td>hypre</td>
<td>65.0</td>
</tr>
<tr>
<td>mpi</td>
<td>mpi</td>
<td>6</td>
<td>mpi</td>
<td>10.0</td>
</tr>
<tr>
<td>psm2</td>
<td>psm2</td>
<td>7</td>
<td>psm2</td>
<td>25.0</td>
</tr>
</tbody>
</table>

squashed_gf = filtered_gf.squash()

squashed_gf

<table>
<thead>
<tr>
<th>name</th>
<th>rid</th>
<th>node</th>
<th>time</th>
<th>time (inc)</th>
</tr>
</thead>
<tbody>
<tr>
<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>1</td>
<td>physics</td>
<td>40.0</td>
</tr>
<tr>
<td>mpi</td>
<td>mpi</td>
<td>2</td>
<td>mpi</td>
<td>5.0</td>
</tr>
<tr>
<td>psm2</td>
<td>psm2</td>
<td>3</td>
<td>psm2</td>
<td>15.0</td>
</tr>
<tr>
<td>solvers</td>
<td>solvers</td>
<td>4</td>
<td>solvers</td>
<td>0.0</td>
</tr>
<tr>
<td>hypre</td>
<td>hypre</td>
<td>5</td>
<td>hypre</td>
<td>65.0</td>
</tr>
<tr>
<td>mpi</td>
<td>mpi</td>
<td>6</td>
<td>mpi</td>
<td>10.0</td>
</tr>
<tr>
<td>psm2</td>
<td>psm2</td>
<td>7</td>
<td>psm2</td>
<td>25.0</td>
</tr>
</tbody>
</table>
We now describe some of the important operators provided by the Hatchet API allowing structured data to be manipulated in the form of a graphframe object. As the readers construct the graphframe object, they also make connections between the graph and dataframe objects using the structured index. These two objects, they also make connects between the graph and dataframe objects using the structured index.

Dataframe Operations

- **Filter**: Takes a user-supplied function and applies that to all rows in the dataframe. As shown in Figure 4, the filter operation removes rows in the dataframe due to a `lambda x: x['time'] > 10.0`. The returned series or dataframe is used to make all inclusive columns consistent again.

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

- **Squash**: Removes nodes from the graph that were previously removed (i.e., the squash operation) by rewiring the graph. In this case, the applied function returns all rows where `x['time'] > 10.0`. After a squash operation, the graph and dataframe become consistent again.

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

Graph Operations

- **Rewire**: The user-supplied function is used to construct the graphframe object. As the readers construct the graphframe object, some operations that lead to changes in the graph structure return a new graphframe. Furthermore, one can provide structured data in Graphviz' DOT format as input.

- **Reader**: Hatchet can read in the database directory generated by HPCToolkit or any other profiling tool that generates CCTs. In this case, the reader is constructed from the database directory.

- **Update inclusive columns**: When a graph is rewired, the `update_inclusive_columns` operation is called to make all inclusive columns consistent again.
Graphframe operation: subtract

```python
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
```

Diagram:

![Diagram showing GraphFrame operations: subtract]
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

```python
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))
```

```
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 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

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

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

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

Flamegraph
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_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)
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 (inc)</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;unknown file&gt; [kripke]:0</td>
<td>17234</td>
<td>1.825282e+08</td>
<td>1.825282e+08</td>
</tr>
<tr>
<td>Kernel_3d_GZ::scattering</td>
<td>60</td>
<td>7.689336e+07</td>
<td>7.896253e+07</td>
</tr>
<tr>
<td>Kernel_3d_GZ::LTImes</td>
<td>30</td>
<td>5.010498e+07</td>
<td>5.240528e+07</td>
</tr>
<tr>
<td>Kernel_3d_GZ::LPlusTimes</td>
<td>115</td>
<td>4.947707e+07</td>
<td>5.104498e+07</td>
</tr>
<tr>
<td>Kernel_3d_GZ::sweep</td>
<td>981</td>
<td>5.018862e+06</td>
<td>5.018862e+06</td>
</tr>
<tr>
<td>memset.S:09</td>
<td>3773</td>
<td>3.168982e+06</td>
<td>3.168982e+06</td>
</tr>
<tr>
<td>memset.S:101</td>
<td>3970</td>
<td>2.120895e+06</td>
<td>2.120895e+06</td>
</tr>
<tr>
<td>Grid_Data:particleEdit</td>
<td>1201</td>
<td>1.131265e+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>6.197776e+05</td>
<td>6.197776e+05</td>
</tr>
</tbody>
</table>
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)
```
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)
```
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>
Example 3: Speedup and efficiency

```python
import glob
import hatchet

# Load the data
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[df['1024'] < 0.7]
df.T.plot.bar()
```

![Diagram showing speedup and efficiency results](image)

This study shows that Chopper significantly simplifies the analysis of parallel profiles using Chopper. It also enables easy plotting of the results via Python libraries. Chopper demonstrates how to optimize performance by identifying and addressing scalability issues. The provided code snippets illustrate how to perform speedup and efficiency analysis using Chopper, including reading data, performing analysis, and visualizing results.
Example 3: Speedup and efficiency

```python
import glob
import hatchet

# Generate datasets
datasets = glob.glob("list_of_tortuga_profiles")

# Create GraphFrames
gfs = hatchet.GraphFrame.construct_from(datasets)

# Calculate speedup and efficiency
df = hatchet.Chopper.speedup_efficiency(gfs, strong=True, efficiency=True)

# Filter results
df = df.loc[df['1024'] < 0.7]

# Plot the results
df.T.loc[:, :].plot.bar()
```

![Graph showing speedup and efficiency results](image)

**Figure 12:** Call paths of the problematic portions of the program before (left) and after (right) the optimization. The time spent on each call is displayed in a bar chart, demonstrating the improvement in efficiency after optimization.

**Figure 13:** Demonstration of scalability analysis by using Chopper. The chart compares the efficiency results for different process counts, showing a significant decrease in execution time as the number of processes increases.

This study shows that Chopper significantly simplifies the analysis of parallel profiles using Chopper by providing a high-level API to avoid having a steep learning curve for new users. It also enables easy plotting of the results via Python libraries, minimizing the need for additional graphical user interfaces.

We identified a scalability issue where the `writeSingleField` function is a potential performance bottleneck. By analyzing the code, we replaced critical parts of the program to write to a function, identifying the code block that causes this scalability issue.

Our analysis demonstrated some useful functionalities such as reading multiple data sets at once and unifying multiple GraphFrames. This capability is particularly useful for performance analysis, which provides programmatic analysis capabilities for single and multiple executions such as detecting load imbalance, finding correlation between metrics and CCT nodes, and causes of performance variation.

We also plan to support customizable plotting capabilities, making it easier to perform with less programming effort. Chopper’s capabilities make performance analysis tasks for single and multi-node granularities with less complexity.

**Conclusion:**

In this study, we proposed Chopper, a Python-based API for performance analysis at per-node granularity by providing functions that are easy to use and integrate into existing workflows. Chopper's programmability and visual capabilities make it a valuable tool for performance engineers in the industry.
Example 4: Load imbalance

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

graphframe_imbalance = graphframe.load_imbalance(verbos=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)
```

<table>
<thead>
<tr>
<th>name</th>
<th>time.imbalance</th>
<th>time.ranks</th>
<th>time.hist</th>
<th>time.percentiles</th>
<th>time.mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>MacroscopicCrossSection.cc:22</td>
<td>4.199311</td>
<td>[39, 46, 118, 33, 94]</td>
<td>[105, 14, 2, 1, 0, 0, 0, 0, 1, 5]</td>
<td>[6.296, 7.12, 7.302, 7.67, 39.102]</td>
<td>9.311</td>
</tr>
<tr>
<td>MacroscopicCrossSection.cc:32</td>
<td>1.539592</td>
<td>[67, 92, 39, 46, 94]</td>
<td>[2, 3, 16, 80, 9, 2, 7, 6, 1, 2]</td>
<td>[21.083, 30.333, 30.946, 31.61, 49.334]</td>
<td>32.043</td>
</tr>
<tr>
<td>NuclearData.cc:270</td>
<td>1.329530</td>
<td>[84, 119, 5, 120, 118]</td>
<td>[27, 38, 26, 16, 13, 5, 1, 0, 0, 2]</td>
<td>[40.667, 42.775, 44.325, 46.946, 60.088]</td>
<td>45.195</td>
</tr>
<tr>
<td>MCT.cc:582</td>
<td>1.319152</td>
<td>[12, 79, 47, 67, 15]</td>
<td>[69, 54, 3, 1, 0, 0, 0, 0, 0, 1]</td>
<td>[17.525, 18.032, 18.158, 18.255, 24.019]</td>
<td>18.208</td>
</tr>
</tbody>
</table>
```
Example 4: Load imbalance

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

<table>
<thead>
<tr>
<th>name</th>
<th>time.imbalance</th>
<th>time.ranks</th>
<th>time.hist</th>
<th>time.percentiles</th>
<th>time.mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>MacroscopicCrossSection.cc:22</td>
<td>4.199311</td>
<td>[39 46</td>
<td>[105, 14, 2, 1, 0 0, 0, 0 1, 5]</td>
<td>[6,296, 7,12, 7,302, 7,67, 39,102]</td>
<td>9.311</td>
</tr>
<tr>
<td>MacroscopicCrossSection.cc:32</td>
<td>1.539592</td>
<td>[67 92 39</td>
<td>[2, 3, 16, 80, 9, 2, 7, 6, 1, 2]</td>
<td>[21,083, 30,333, 30,946, 31,61, 49,334]</td>
<td>32.043</td>
</tr>
<tr>
<td>NuclearData.cc:270</td>
<td>1.329530</td>
<td>[84 119 5</td>
<td>[27, 38, 26, 16, 13, 5, 1, 0, 0, 2]</td>
<td>[40,667, 42,775, 44,325, 46,946, 60,088]</td>
<td>45.195</td>
</tr>
<tr>
<td>MCT.cc:582</td>
<td>1.319152</td>
<td>[12 79 47</td>
<td>[69, 54, 3, 1, 0, 0, 0, 0, 0, 1]</td>
<td>[17,525, 18,032, 18,158, 18,255, 24,019]</td>
<td>18.208</td>
</tr>
</tbody>
</table>
```
Abhinav Bhavele
5218 Brendan Iribe Center (IRB) / College Park, MD 20742
phone: 301.405.4507 / e-mail: bhatele@cs.umd.edu