Lecture 12: Analysis/Visualization Tools
Abhinav Bhave, Department of Computer Science
Summary of last lecture

- Performance analysis
  - Identify performance bottlenecks, anomalies
  - Measurement, analysis, visualization tools
- Tracing and profiling
- Calling context trees, graphs
MPI trace visualization

Vampir

Jumpshot
Projections Performance Analysis Tool

- For Charm++/Adaptive MPI programs
- Instrumentation library
  - Records data at the granularity of chares (Charm++ objects)
**Time Profile**

Figure 3. Time profile for ApoA1 on 1k processors of Blue Gene/L (with PME) in Projections.

Figure 4. NAMD performs well on any given number of processors (Plot shows ApoA1 running on Blue Gene/L, with PME on a range of processor counts varying from 207 to 255).

This decomposition is called 2-Away-X, 2-Away-XY or 2-Away-XYZ etc. depending on which dimension uses $k=2$.

The choice of which decomposition to use for a particular run is decided by the program depending on the atoms-to-processor ratio and other machine-dependent heuristics. NAMD also gives the user flexibility to choose the decomposition for certain scenarios where the automatic choices do not give the best results.

Neither the number of cells nor the number of compute objects need to be equal to the exact number of processors. Typically, the number of cells is smaller than the number of processors, by an order of magnitude, which still generates adequate parallelism (because of the separation of "compute" objects) to allow the load balancer to optimize communication, and distribute work evenly. As a result, NAMD is able to exploit any number of available processors. Fig. 4 shows the performance of the simulation of ApoA1 on varying numbers of Blue Gene/L (BG/L) processors in the range 207-255. In contrast, schemes that decompose particles into $P$ boxes, where $P$ is the total number of processors may limit the number of processors they can use for a particular simulation: they may require $P$ to be a power of two or be a product of three numbers with a reasonable aspect ratio.

We now describe a few features of NAMD and analyze how they are helpful in scaling performance to a large number of processors.

**2.1 Adaptive Overlap of Communication and Computation**

NAMD uses a message-driven runtime system to ensure that multiple modules can be composed concurrently without losing efficiency. In particular, idle times in one module can be exploited by useful computations in another. Furthermore, NAMD uses asynchronous reductions, whenever possible (such as in the calculation of energies). As a result, the program is able to continue without sharp reductions in utilization around barriers. For example, Fig. 3 shows a time profile of a simulation of ApoA1 on 1024 processors of BG/L (This figure was obtained by using the performance analysis tool Projections [10] available in the CHARM++ framework). A time profile shows vertical bars for each (consecutive) time interval of 100 us, activities executed by the program added across all the processors. The red (dark) colored "peaks" at the bottom correspond to the force integration step, while the dominant blue (light) colored regions represent non-bonded computations. The pink and purple (dark at the top) shade appearing in a thin layer every 4 steps represent the PME computation. One can notice...
Usage Profile & Histogram View
Usage Profile & Histogram View
Outlier Analysis
Scripting for multi-run comparisons

Table 10. Comparison of benchmark times (ms/step) for NAMD (running on 2.6 GHz Opterons) and Desmond (running on 2.4 GHz Opterons)

<table>
<thead>
<tr>
<th></th>
<th>NAMD</th>
<th>Desmond</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApoA1</td>
<td>256.8</td>
<td>126.8</td>
</tr>
<tr>
<td></td>
<td>64.3</td>
<td>33.5</td>
</tr>
<tr>
<td></td>
<td>18.2</td>
<td>9.4</td>
</tr>
<tr>
<td></td>
<td>5.2</td>
<td>3.0</td>
</tr>
<tr>
<td></td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>DHFR</td>
<td>41.4</td>
<td>21.0</td>
</tr>
<tr>
<td></td>
<td>11.5</td>
<td>6.3</td>
</tr>
<tr>
<td></td>
<td>3.7</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>1.4</td>
<td>1.4</td>
</tr>
</tbody>
</table>

Figure 8. Percentage increase of different parts of NAMD with increase in number of processors (ApoA1 on BG/L and XT3, without PME)

The needs of biomolecular modeling community require us to pursue strong scaling i.e. we must strive to scale the same molecular system to an ever larger number of processors. We have demonstrated very good scalability with the techniques described in this paper, but challenges remain, especially if we have to exploit the petascale machines for the same problems. To analyze if this is feasible and what avenues are open for further optimizations, we carried out a performance study of scaling that we summarize below.

We used the summary data provided by Projections which gives detailed information about the time elapsed in the execution of each function in the program and also the time for which each processor is idle. This data, collected on the Cray XT3 machine at PSC and the BG/L machine at IBM for 1 to 4,096 processors, is shown in Fig. 8.

To simplify the figures, functions involved in similar or related activities are grouped together. The first observation is that the idle time rises rapidly beyond 256 processors. This is mostly due to load imbalance, based on further analysis. One of the next challenges is then to develop load balancers that attain better performance while not spending much time or memory in the load balancing strategy itself. Also, there is a jump in the non-bonded work from 256 to 512 which can be attributed to the change in the decomposition strategy from 2-Away-X to 2-Away-XY at that point which doubles the number of cells. Since the essential computational work involved in non-bonded force evaluation does not increase, we believe that this increase can be reduced by controlling the overhead in scheduling a larger number of objects. The other important slowdown is because of the increase in communication overhead as we move across the processors. Looking back at Table 1 we see that from 512 to 16K processors, the computation time per processor should ideally decrease 32-fold but does not. This can be attributed in part to the communication volume (per processor) decreasing only by a factor of about 10. Although some of this is inevitable with finer-grained parallelization, there might be some room for improvement.

Summary

The need for strong scaling of fixed-size molecular systems to machines with ever-increasing number of processors has created new challenges for biomolecular simulations. Further, some of the systems being studied now include several million atoms. We presented techniques we developed or used, to overcome these challenges. These included dealing with interaction among two of our adaptive strategies: generation of spanning trees and load balancing. We also presented new techniques for reducing...
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
### Dataframe operation: filter

We now describe some of the important operators provided by Hatchet to manipulate data in the dataframe, a user might be interested in aggregating. For example, to filter out all rows where time is greater than 10.0, the user might use the `filter` operation on the time column.

**Operations that lead to changes in the graph structure** return a new graph object of the graphframe and the performance metrics are updated all inclusive columns. When a graph is rewired (i.e., the parent-child connections are modified), all the columns in the hierarchical index by specifying an aggregation operation on the graphframe. Typically, the user will perform a union operation, the graph and dataframe become consistent again.

**Filter** is one of the operations that leads to the graph object and the respective children. If they are equivalent, it returns true, otherwise, it returns false. The `check_equivalence` function essentially checks whether two graphs are equivalent or not in their structures by comparing the call paths different ways:

1. union: the union function takes two graphs and creates a union of the respective children. If they are equivalent, it returns true, otherwise, it returns false.
2. squash: the squash operation is typically performed by the user to make the graph consistent again. This operation is useful when we wish to perform an operation on the time column.
3. drop_index_levels: this operation is used to index into the dataframe. Typically, the user will perform a drop_index_levels operation on the time column.

#### Code Example

```python
# Assuming gf is a GraphFrame object
filtered_gf = gf.filter(lambda x: x['time'] > 10.0)
```

### Table Example

<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>
We now describe some of the important operators provided by Hatchet. In this case, the applied function returns all rows where the expression is true. The returned dataframe is filtered to only return rows that are true. The returned dataframe is filtered, aggregated, pruned, etc. Even though all columns in the dataframe due to a filter operation on the dataframe. As shown in Figure 5, there are nodes in the graph that do not return any rows when a filter operation is performed on the graphframe. The graphframe preserves the original graph provided as input to the graphframe.

**Graph operation: squash**

<table>
<thead>
<tr>
<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>main</td>
<td>40.0</td>
<td>200.0</td>
</tr>
<tr>
<td>physics</td>
<td>physics</td>
<td>physics</td>
<td>40.0</td>
<td>60.0</td>
</tr>
<tr>
<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>15.0</td>
<td>15.0</td>
</tr>
<tr>
<td>solvers</td>
<td>solvers</td>
<td>solvers</td>
<td>0.0</td>
<td>100.0</td>
</tr>
<tr>
<td>hypre</td>
<td>hypre</td>
<td>hypre</td>
<td>66.0</td>
<td>65.0</td>
</tr>
<tr>
<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>25.0</td>
<td>25.0</td>
</tr>
</tbody>
</table>

1. `gf = GraphFrame(...)`
2. `filtered_gf = gf.filter(lambda x: x['time'] > 10.0)`
We now describe some of the important operators provided by Hatchet, the tool used to construct the graphframe object. As the readers construct the graph object of the graphframe and the performance metrics are included in a CCT structure, the readers in Hatchet read in both the time saved and the per-node granularity.

Hatchet can read in the database directory generated by HPCToolkit and generated by Caliper. In the include code, user can use the graphframe. The users can construct the objects using the structured index. Typically, the user will perform a filter operation on the dataframe, some operations that lead to changes in the graph structure return a new dataframe object becoming inconsistent. After a filter operation, all the columns in the dataframe become inconsistent. Hence, the squash operation also updates the inclusive metrics stored in the dataframe for each node. squash on the graphframe after a filter operation to make the graph preserved.

Most profiling tools in the HPC community are probed (i.e., the child relationships have changed. Hence, the squash operation also updates all inclusive metrics stored in the dataframe for each node.

### Filter

Filter is one of the operations that lead to changes in the graph object and the performance metrics are included. The function performs a post-order traversal of the graph to update all inclusive metrics stored in the dataframe for each node.

```python
1. gf = GraphFrame(...)  
2. filtered_gf = gf.filter(lambda x: x['time'] > 10.0)
```
Graph operation: squash

1. gf = GraphFrame( ... )
2. filtered_gf = gf.filter(lambda x: x['time'] > 10.0)
3. filtered_gf = gf.filter(lambda x: x['time'] > 10.0)
4. squashed_gf = filtered_gf.squash()
Graphframe operation: subtract

1. gf1 = DataFrame(...)
2. gf2 = DataFrame(...)
3. gf2 -= gf1
Visualizing output

Graphviz

Flamegraph

Terminal output
Generating a flat profile

<table>
<thead>
<tr>
<th>name</th>
<th>nid</th>
<th>time</th>
<th>time (inc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;unknown file&gt; [kripe]:0</td>
<td>17234</td>
<td>1.825282e+08</td>
<td>1.825282e+08</td>
</tr>
<tr>
<td>Kernel_3d DGZ::scattering</td>
<td>60</td>
<td>7.669936e+07</td>
<td>7.896253e+07</td>
</tr>
<tr>
<td>Kernel_3d DGZ::LTimes</td>
<td>30</td>
<td>5.010439e+07</td>
<td>5.240528e+07</td>
</tr>
<tr>
<td>Kernel_3d DGZ::LPlusTimes</td>
<td>115</td>
<td>4.947707e+07</td>
<td>5.104498e+07</td>
</tr>
<tr>
<td>Kernel_3d DGZ::sweep</td>
<td>981</td>
<td>5.018882e+06</td>
<td>5.018882e+06</td>
</tr>
<tr>
<td>memset.S:99</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>1.131266e+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:96</td>
<td>3767</td>
<td>6.197776e+05</td>
<td>6.197776e+05</td>
</tr>
</tbody>
</table>

```python
1  gf = GraphFrame()
2  gf.from_hpctoolkit('kripe')
3  grouped = gf.dataframe.groupby('name').sum()
```
Generating a flat profile

<table>
<thead>
<tr>
<th>name</th>
<th>nid</th>
<th>time</th>
<th>time (inc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;unknown file&gt; [kriple]:0</td>
<td>17234</td>
<td>1.825282e+08</td>
<td>1.825282e+08</td>
</tr>
<tr>
<td>Kernel_3d_DGZ:scattering</td>
<td>60</td>
<td>7.669936e+07</td>
<td>7.896253e+07</td>
</tr>
<tr>
<td>Kernel_3d_DGZ:LTimes</td>
<td>30</td>
<td>5.010439e+07</td>
<td>5.240528e+07</td>
</tr>
<tr>
<td>Kernel_3d_DGZ:LPlusTimes</td>
<td>115</td>
<td>4.947707e+07</td>
<td>5.104498e+07</td>
</tr>
<tr>
<td>Kernel_3d_DGZ:sweep</td>
<td>981</td>
<td>5.018862e+06</td>
<td>5.018862e+06</td>
</tr>
<tr>
<td>memset.S:99</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>1.131266e+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:96</td>
<td>3767</td>
<td>6.197776e+05</td>
<td>6.197776e+05</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>module</th>
<th>nid</th>
<th>time</th>
<th>time (inc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kriple/build-mvapich2.3/kriple</td>
<td>14366</td>
<td>1.825802e+08</td>
<td>5.847993e+08</td>
</tr>
<tr>
<td>/usr/lib64/fd-2.17.so</td>
<td>9676</td>
<td>0.000000e+00</td>
<td>9.340625e+02</td>
</tr>
<tr>
<td>/usr/lib64/libc-2.17.so</td>
<td>37970</td>
<td>0.000000e+00</td>
<td>7.150550e+06</td>
</tr>
<tr>
<td>/usr/lib64/libdl-2.17.so</td>
<td>4427</td>
<td>0.000000e+00</td>
<td>2.804062e+02</td>
</tr>
<tr>
<td>/usr/lib64/libpsm2.so.2.1</td>
<td>433252</td>
<td>0.000000e+00</td>
<td>2.496037e+06</td>
</tr>
<tr>
<td>/usr/lib64/libpthread-2.17.so</td>
<td>2679</td>
<td>0.000000e+00</td>
<td>4.674375e+02</td>
</tr>
<tr>
<td>c-4.9.3/lib64/libstdc++.so.6.0.20</td>
<td>14945</td>
<td>0.000000e+00</td>
<td>3.898480e+05</td>
</tr>
<tr>
<td>/intel/mpi/llintc.so.5</td>
<td>1215</td>
<td>0.000000e+00</td>
<td>9.357812e+01</td>
</tr>
<tr>
<td>--intel-18.0.1/lib/libvmi.so.12.1.1</td>
<td>126726</td>
<td>0.000000e+00</td>
<td>7.962225e+06</td>
</tr>
</tbody>
</table>

```python
1. gf = GraphFrame()
2. gf.from_hpctoolkit('kriple')
3. grouped = gf.dataframe.groupby('name').sum()
```
Degree of load imbalance

```
gf1 = GraphFrame()
gf1.from_caliper('lulesh-512cores')
gf2 = gf1.copy()
gf1.drop_index_levels(function=np.mean)
gf2.drop_index_levels(function=np.max)
gf1.dataframe['imbalance']
    = gf2.dataframe['time'].div(gf1.dataframe['time'])```

Figure 10: The subtract operation in Hatchet enables comparing execution profiles. In this figure, the left graph is subtracted from the middle graph to obtain the right graph. When we sort the nodes in the right graph by time, we can easily identify the biggest outliers.

Figure 11: Load imbalance within a single execution is derived by calculating the mean and maximum values of a metric at each node across all MPI processes or threads and then dividing the two values for each node.

Many performance tools provide facilities to store performance data in a database and to apply machine learning and other data analysis tools to it. PerfExplorer provides a database, a GUI analysis environment, and the PerfDMF data format. Open|SpeedShop has an internal SQL database used by the GUI to load parts of performance datasets. However, all of these tools predate the popularization of data analysis frameworks like R and pandas, and they do not provide rich APIs for manipulating data. TauDB, part of PerfDMF, provides language bindings for exploring datasets, but it does not provide the in-memory query or aggregation capabilities that modern frameworks have. All “queries” in these tools must be written in SQL, with a fixed schema, and handed directly to the backend database. There is no in-memory dataframe or abstraction layer as we have leveraged in Hatchet. The closest related work to Hatchet is likely differential profiling. Early work showed the utility of subtracting similar or scaled call trees to pinpoint performance issues. This work was improved upon by techniques for scaling analysis implemented in HPCToolkit.

HPCToolkit provides facilities for calculating derived expressions from performance metrics on call trees within the GUI, and this can be used to scale and subtract columns in the hpcviewer GUI. However, the usage model is cell-based like a spreadsheet; it is not fully programmable or easily integrated with other frameworks.

Likely the most scalable existing call path visualizer is HPCTrace-Viewer, which provides visualizations of call paths over time, MPI ranks, and threads in parallel codes. This tool and Libra are the closest analogs to the per-MPI-rank analyses in this paper. Again, though, these are GUI tools and they do not provide the flexibility to easily script new analyses or to easily query, filter, aggregate, and squash profile data in an indexed dataframe as Hatchet does. Typically, the available analyses are manually selected through drop down menus or some other user-interface, and there is limited flexibility for customization.
Degree of load imbalance

```
gf1 = GraphFrame()
gf1.from_caliper('lulesh-512cores')
gf2 = gf1.copy()
gf1.drop_index_levels(function=np.mean)
gf2.drop_index_levels(function=np.max)
gf1.dataframe['imbalance']
    = gf2.dataframe['time'].div(gf1.dataframe['time'])
```

<table>
<thead>
<tr>
<th>node</th>
<th>name</th>
<th>nid</th>
<th>time</th>
<th>time (inc)</th>
<th>imbalance</th>
</tr>
</thead>
<tbody>
<tr>
<td>LagrangeNodal</td>
<td>LagrangeNodal</td>
<td>3.0</td>
<td>2.242594e+06</td>
<td>2.593621e+07</td>
<td>2.494720</td>
</tr>
<tr>
<td>main</td>
<td>main</td>
<td>0.0</td>
<td>1.106013e+05</td>
<td>5.357208e+07</td>
<td>2.161845</td>
</tr>
<tr>
<td>CalcForceForNodes</td>
<td>CalcForceForNodes</td>
<td>4.0</td>
<td>1.033639e+06</td>
<td>2.369361e+07</td>
<td>2.142526</td>
</tr>
<tr>
<td>CalcQForElem</td>
<td>CalcQForElem</td>
<td>16.0</td>
<td>3.351894e+06</td>
<td>6.649351e+06</td>
<td>2.037651</td>
</tr>
<tr>
<td>CalcEnergyForElem</td>
<td>CalcEnergyForElem</td>
<td>22.0</td>
<td>1.571996e+06</td>
<td>2.807323e+06</td>
<td>2.013174</td>
</tr>
<tr>
<td>CalcPressureForElem</td>
<td>CalcPressureForElem</td>
<td>23.0</td>
<td>1.235327e+06</td>
<td>1.235327e+06</td>
<td>2.005437</td>
</tr>
</tbody>
</table>
Comparing two profiles

```
gf1 = GraphFrame()
gf1.from_caliper('lulesh-27cores')
gf2 = GraphFrame()
gf2.from_caliper('lulesh-512cores')

filtered_gf1 = gf1.filter(lambda x: x['name'].startswith('MPI'))
filtered_gf2 = gf2.filter(lambda x: x['name'].startswith('MPI'))
squashed_gf1 = filtered_gf1.squash()
squashed_gf2 = filtered_gf2.squash()
diff_gf = squashed_gf2 - squashed_gf1
```
Comparing two profiles

```python
gf1 = GraphFrame()
gf1.from_caliper('lulesh-27cores')
gf2 = GraphFrame()
gf2.from_caliper('lulesh-512cores')

filtered_gf1 = gf1.filter(lambda x: x['name'].startswith('MPI'))
filtered_gf2 = gf2.filter(lambda x: x['name'].startswith('MPI'))

squashed_gf1 = filtered_gf1.squash()
squashed_gf2 = filtered_gf2.squash()
diff_gf = squashed_gf2 - squashed_gf1
```
With Hatchet, we provide a common data model for representing structured pro-

dress many common performance analysis tasks on different types of performance data.

These tasks include:

1) identifying regions or call sites with the most load imbalance
2) performing regression analysis of structured pro-

dress many common performance analysis tasks on different types of performance data.

These tasks include:

1) identifying regions or call sites with the most load imbalance
2) performing regression analysis of structured pro-

```python
import os
import time
import pandas as pd
import re

# Example datasets

# Use glob to find all LULESH json files
datasets = glob.glob('lulesh*.json')

# Sort the datasets
datasets.sort()

# Initialize an empty list to hold dataframes
dataframes = []

# Process each dataset
for dataset in datasets:
    # Initialize a GraphFrame
    gf = GraphFrame()
    # Load the dataset into the GraphFrame
    gf.from_caliper(dataset)
    # Drop the hierarchical index levels
    gf.drop_index_levels()
    # Extract the 'pes' time from the dataset
    num_pes = re.match('.*-(\d+)-(.*), dataset).group(2)
    # Select the dataframes corresponding to the current num_pes
    gf.dataframe['pes'] = num_pes
    filtered_gf = gf.filter(lambda x: x['time'] > 1e6)
    # Append the filtered dataframe to the list
    dataframes.append(filtered_gf.dataframe)

# Concatenate all the dataframes
result = pd.concat(dataframes)

# Pivot the result to show the time spent on each node for each core
pivot_df = result.pivot(index='pes', columns='name', values='time')

# Plot the pivot dataframe
pivot_df.loc[:,:].plot.bar(stacked=True, figsize=(10,7))
```
Questions
Scaling Applications to Massively Parallel Machines Using Projections …

- What is AMPI?

- Is there any standardized data format to store performance profiling/analysis results?

- Does Projections support heterogeneous systems (like a node with a CPU and multiple GPUs)?

- Performance analysis and tuning, in general, seems to incorporate a lot of experience and hand crafting. Are there tools that generate suggestions for possible code modifications based on the profiling result?

- Can we go over the load balancing? Why does the balance look a little worse (and the overall load higher) after refinement? The paper talks about quirks in background load leading to underutilization in a range of processors. What sorts of quirks can lead to this type of behavior?

- How do parallel simulators work? The paper mentions BigSim. Is this a popular one? Is it common for people to use a simulator before running on a large supercomputer?
Questions

Hatchet: Pruning the Overgrowth of Parallel Profiles

• What is the definition of reproducibility in performance analysis?

• A programmable tool is great to automate analysis, but I guess a dedicated interactive GUI is also very useful for some analysis. Are there plans to incorporate such elements?

• Which profiling tool is most recommended to generate profile data for the processing with Hatchet?

• Is the library open-sourced, or are there any plans?

• How is it that the drop_index_levels performance is able to remain basically constant until getting to about 256 processors? Also, what's with the strange shape of the filter performance graph? And is 512 processors as the max for the performance test for the tool a little on the low end? Would the analysis tool be usable to look at profiling results from a real or simulated run on a supercomputer?

• Hatchet is ~2.5k lines of code. What were some of the most complicated parts to implement? Could you go over the design of the code briefly
Questions?

Abhinav Bhave
5218 Brendan Iribe Center (IRB) / College Park, MD 20742
phone: 301.405.4507 / e-mail: bhatele@cs.umd.edu