Summary of last lecture

- Scalable networks: fat-tree, dragonfly
  - Use high-radix routers
  - Many nodes connected to each switch
- Low network diameter, high bisection bandwidth
- Dynamic routing
Performance analysis

- Parallel performance of a program might not be what we expect
- How do we find performance bottlenecks?
- Two parts to performance analysis: measurement and analysis/visualization
- Simplest tool: timers in the code and printf
Performance Tools

• Tracing tools
  • Capture entire execution trace
  • Vampir, Score-P

• Profiling tools
  • Typically use statistical sampling
  • Gprof

• Many tools can do both
  • TAU, HPCToolkit, Projections
Metrics recorded

- Counts of function invocations
- Time spent in code
- Hardware counters
Calling contexts, trees, and graphs

- Calling context or call path: Sequence of function invocations leading to the current sample
- Calling context tree: dynamic prefix tree of all call paths in an execution
- Call graph: keep caller-callee relationships as arcs

Figure 3 shows the two objects in a GraphFrame – a graph object and a DataFrame object.

One further consequence of our index model is that to use two GraphFrames, we must use the union of their indices in the next union operation. We default to a lexicographical sort on the first column in the DataFrame together.

The central data structure in the Hatchet library is a GraphFrame, which combines the structured index (the node) is the index column. As a convenience, we may also add an integer representing the address of the Python object in memory. The key is not meant to expose their own names. By default, we use the Python object identifier (OID) as the key. This is equivalent, roughly, to C's "< operator, in that it returns less than for the implementation of the key.

For example, in the graph, we can insert Node objects directly into the pandas DataFrame. Each node knows its children and its ancestors in the graph, and each node has a unique identifier. The key is not meant to be accessed by Hatchet users. Rather, like Frames, Hatchet nodes operate on their own attributes as much as possible over different sources to consider that the nodes are compared by their own semantics. The GraphFrame API is responsible for ensuring that graphs and computing their union according to their connectivity (the node key be with each node). We abstract the details of these graph operations in Hatchet well. We abstract the details of these graph operations in Hatchet to not a requirement.

As we will see later, the graph-semantic functions to be used, as well. Finally, in addition to the identifying information for each node, we need connectivity and structure of the graph, and Node objects de...

MultiIndex can optionally use keys that provide certain useful orderings (like names) based on attributes from each node's Frame.}

Abhinav B hatele, CMSC714

SC '19, November 17–22, 2019, Denver, CO, USA
Output

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

gprof: A Call Graph Execution Profiler

- Execution count: It is highlighted to be two types of counts, which is either an actual count or a boolean. What’s the benefit of introducing the second type?

- It seems that the call to monitoring routine is more informative but slower compared to the inline counter increment. Will the slow down actually affect the accuracy of the monitoring? Also is this trade-off generally worth it (in terms of profiling)?

- It is not immediately clear from the paper how they actually derive the timing approximation from the histogram. If possible I’d like to see if there’s an illustrating example.

- Is there any principled way to extract static call graph from a generic program?

- What are the different types of call graphs? How is each type best used for understanding program performance?

- How much memory does profiling data require usually? Related: how does gprof balance various overheads?

- How does timeslicing work on timeshare machines?
Questions

Binary Analysis for Measurement and Attribution of Program Performance

• The paper states “dynamic instrumentation remains susceptible to systematic measurement error because of instrumentation overhead”. Where do these overheads come from comparing to static and binary instrumentation?

• The loop optimization performed by compiler introduces semantic gap between source code and binary. Is there any effort on incorporating compiler into the profiling system to reduce such gap?

• It seems from the paper that the proposed HPCToolkit is better than gprof. How do they compare practically when used to profile a program?

• How does highly optimized code make it harder to accurately profile? How does binary analysis address these issues?

• What are the measurement techniques for instrumentation?
Questions?