CMSC416
Lecture 2/27/2024

Recall from last class:
Calling Contexts, trees, and graphs

If you combine all the nodes for a function, you transform a context tree into a context graph.
Flat profile is the most coarse, as you just list all the functions with counts and execution times.
Hatchet, which is a python based tool you can use programmatically (as opposed to using a GUI). Hatchet is similar to a Pandas dataframe, but in Hatchet, you can use the handle to a function call to index into a node. There's a graph call object as well as the dataframe. You can also use all the classic pandas functions. Hatchet provides a higher level API filter on top of Pandas'.

More about Hatchet:

Hatchet is a library like numpy. To use it, first point it to the HPCToolkit database directory. There is a family of functions to read initial data (from HPC, Caliper, etc). Hatchet also works in Jupyter notebook, provided you have a local installation of Hatchet (similar to other libraries). Using this tool, you have access to time (inclusive and exclusive), id, name, type, file, etc.

Hatchet provides a *filter* operation similar to Pandas'.
There's also a *squash* operation, which is different from filter operation. When using a filter, the dataframe is reduced, but the graph remains the same. In a squash operation, however, the graph is rewired and becomes smaller. Though in the current version, when filtering, the graph is automatically squashed (in the past this used to be a two-step process). Filter internally calls squash.

Hatchet's GraphFrame object has two distinct objects: the graph object and the dataframe. These are usually kept in sync, but are still distinct objects. The graph is stored as Node objects, similar to a graph implementation in most languages.

Hatchet also provides subtraction functionality. For example, if you run a piece of code with 4 processes then 16 processes, you can see the differences in the execution times by simply subtracting the first GraphFrame object from the second. The time on each node is now the relative difference between the first run and the second. This can help highlight areas where more processes aren't necessarily increasing speedup, which may provide room for improvement. If the graphs end up having distinct nodes, Hatchet internally adds dummy nodes (unifying) where necessary. These dummy nodes are assumed to be time zero.

https://hatchet.readthedocs.io to see Hatchet's documentation.

While Hatchet isn't supposed to be a graph visualization tool, there is still some rudimentary functionality to view small graphs. The first is a String representation that is terminal friendly. It also supports different output formats that work with other visualization tools (GraphViz, Flamegraph)

Example 1: Generate a flat profile

Very simple to go from a calling context tree to a flat profile, with sorted culminations of runtimes.

Example 2: Comparing two executions

Shows the differences between two distinct GraphFrames representing distinct executions. This introduces drop_index_levels() that gives nodes a unique number necessary to have before subtracting.

Example 3: Speedup and efficiency

Hatchet makes it very simple to calculate the speedup and efficiency of your GraphFrames. Some higher-level calls are provided by Chopper. You have to specify whether you were testing strong or weak scaling, and whether you want to see efficiency or speedup. It will create a new dataframe that has the speedup and efficiency per function. Ideally, you want to identify functions where you both spend a lot of time, *and* the efficiency is poor, as improving that performance will result in the most time saved.

Example 4: Load imbalance

Hatchet makes it easy to look at load imbalance on a node to node basis. (Load balance = max_time/mean_time). A load balance close to 1 is a great sign, where anything greater than 1 indicates room for improvement. A verbose flag also outputs this load balance for each node, allowing users to see which functions have the highest load discrepancies.

Shared Memory and OpenMP

We've been focusing on MPI, which is a distributed memory programming model, now we'll switch to OpenMP, which is a shared memory architecture.

Shared Memory Programming

All threads now have access to the entire address space, communicating via just accessing the memory of other threads. Since every thread has access to all memory, the user must now handle synchronization conflicts, which can introduce bugs into your code. Changing from serial code to parallel code is often easier to implement.

OpenMP

A model that you can only run in a single node (though there has been architecture in the past that makes multiple nodes look like a single node, and OpenMP was able to be used).

We'll assume on-node parallelization for this course. There are certain kinds of programs (like arrays and loops) that gain the most benefit from using OpenMP

It is a language extension (and library) that allows parallelization of C/C++/Fortran code.

Programmer uses directives, and then the compiler converts the code to multi-thread.

Fork-join parallelism

Single flow of control, where one main thread spawns in new workers to take care of tasks. Once their work is done, threads are joined together. Threads can do different things in each region,

Race conditions

Since the onus is on the programmer to maintain program correctness, the programmer must ensure that race conditions are not found. A race condition is defined as when the program outcome depenends on the scheduling order of threads. So, the programmer has to be certain that certain reads and writes are done in the correct order, or make data disjoint so that threads aren't accessing the same memory.

OpenMP Pragmas

A pragma is a compiler directive in C/C++, which serves as a mechanism to communicate with the compiler. Certain pragmas may be ignored by the compiler if the correct flags aren't used.

Hello World Example

Adding the directive **parallel** tells the compiler that we want to take the proceeding code block and parallelize it across multiple threads. Without braces, only one line will be parallelized. You can also do functions and code blocks with braces. Where MPI uses processes to parallelize, OpenMP uses threads, which are more lightweight. Nowadays, OpenMP is standardized in C/C++ compilers, so there's a flag to compile OpenMP directives into actual parallel code. To set the number of threads, you need to set the environment variable OMP_NUM_THREADS. There is also a way to dynamically change the number of threads used for each directive. Everytime you open a new shell, you'd have to reset this environment variable.

Parallel for execution

Instead of using just **parallel**, we can use **parallel for** to direct the compiler that the for loop immediately following the pragma should be parallelized. Since we aren't ever reading from the array, there aren't concerns of race conditions. All the threads will divide sections of the iterations to however many threads are specified. Even if you have nested for loops, the directive will only do the first

one. You would need to add separate directives before each in order to parallelize any subsequent loops.

Number of threads

To use a different amount of threads for different sections of code, we can also use a library. You can use it to get the number of available cores (used to be processes) on the node, and that can be used to decide the number of threads to create.