February 27th

Notes:

Performance Analysis and Tools

What is Hatchet? Hatchet is a python library. It builds upon Pandas dataframes, allowing these dataframes to indexed by structured trees and graphs (these call graphs are more condensed versions of the calling context trees but still maintain parent-child relationships). These graph/trees store what is called the original calling context (sequence of function calls by a specific sample program). In other words, every node of these graphs/trees can be used as an index to a specific row in the dataframe.

Once we have these trees/graphs in memory, it allows for all kinds of panda supported analysis. An example of this is leveraging pandas' groupby function. This function will group the data by a certain user provided metric. However, Hatchet provides more tools for program analysis that we can use.

Here is a brief explanation of how to start using Hatchet. First write your python file. Import Hatchet as you would with any other library: "Import hatchet as ht." Provide a path to the PHCToolkit database directory and use Hatchet's "from_hpctoolkitl" API to read in the HPCToolkit database: "ht.GraphFrame.from_hpctoolkit(dirname)." Now from here, there are many things you can do. You can print the data frame (using print) or print the graph ("print(dataframe.tree(args)"). If you have multi process data, each rank has its own row for each node, so it becomes a multiindex dataframe. You would index by the node name and then by rank number, for example. The dataframes themselves also have various columns that have data on the exclusive or inclusive time

Here are some of the tools Hatchet provides. Hatchet provides its own filter operations. This filter operation will filter the dataframe, but also, by default, will squash at the same time. This essentially means that whatever filters we apply to the dataframe and what the result becomes, these changes will be reflected in the graph/tree. Hatchet has a subtract operation. This will compare different executions for each node in the graph, for example, it can compare how

much time you spend in certain nodes. You may wonder, how does this handle differences in graphs. Maybe some graphs don't have all the same nodes. You don't need to worry about this as Hatchet will take care of it. Hatchet will create dummy nodes, similar to the processes of creating a unified graph. It will keep track of the times for each node for each graph and then the subtraction will occur. Note, some nodes may have negative times.
Hatchet also has rudimentary visualization tools for small graphs (ex. "print(gf.tree(color=true))"). Visualization is not the main use of Hatchet so it is not built to handle larger trees well; you can use something like graphviz. You can output your data, for example like "gf.to_dot()" to then feed into graphviz. There are also other functions like "to_flagegraph()" that shows an inverted tree visualization of how much time you spend in what functions. More so, if you have a multiprocess index in your dataframe, you can use the function "drop_index_levels()" so that each no has unique numbers (will be averaged). More information about functions are available on Hatch's documentation which is available online.

Furthermore, Hatchet also lets you calculate speedup and efficiency for a set of graphframes. You can do this and even plot it! Note, for some of the function, you may have to use the Chopper API. More so, you can look into the load imbalance as per single run of a program. You can compare max time versus mean time across all processes; ideally, a value close to 1 is good. We want max time close to mean time. Load imbalance can be used to help figure what functions may need to be looked into more. There are also different flags/parameters like "verbose" which if you sent this to true, will print out the most overloaded processes and other extra information that could be useful.

Shared Memory and OpenMP

OpenMP is a shared memory programing model. All cores have access to all memory, but some memory may be father for some cores; meaning it may take longer to get or send information. This is not a uniform shared memory access architecture. Generally, in shared memory programming models, users create entities via threads that have access to the entire address space. This may make you wonder, how do threads communicate? By looking at the memory of other threads, this is a form of implicit communication. Now this is more tricky in some sense. In the case of the shared memory by thread, every thread can access anything, meaning threads could maybe try to access the same memory at the same time. This can be problematic. If one thread is trying to write to an array but another thread is trying to read the values of it, as a user, you need to make sure this process occurs in a structured order. You wouldn't want the reading thread to read the data when it needed to wait for the other thread to update it. Managing this is what can be tricky. As a user, you can synchronize this with blocking/locking calls (think back to 216). Therefore, writing an initial implementation OpenMP is easier when serial code is less complex.

Here is move of a drive into OpenMP and actually coding. Because we use threads, it is different from MPI in the sense that we only run OpenMP within a node (though there have been few exceptions in the past). OpenMP is considered to be a language extension and library. Programmers use compiler directives and library routines to indicate parallel regions in their code. The compiler then converts the code to multi-threaded code (so it is easier in this sense). OpenMP uses the fork-join model of parallelism

Fork-join Parallelism

In this model, code executes sequentially and then splits off. Essentially, you have a single flow of control and the master thread will spawn worker threads during different stages of execution.

However, you will have to watch out for race conditions. This is the unintended sharing of variables, where the program outcome will differ depending on the order of how the threads execute; meaning it might not give the same results if the program was run serially. To prevent race conditions, programmers must use synchronized reads and writes or change how the data is stored, by this I mean having threads lets say accessing disjoint parts of an array or each thread creates a copy of the array.

OpenMP Pragma

Compiler directives to communicate with the compiler. Here is an example of the standard setup: “#pragma opm [construct] [clause [clause] …].” An example of a construct is “parallel” which takes succeeding code block and parallelizes it. Another example is “#pragma omp parallel for [clause [clause] …].” This directs the compiler that the immediately following for loop should be executed in parallel for not to complex operations. Note, if you have nested for loops or another for loop after, the pragma will not effect it at all, you will need another compiler directive for each subsequent for loop.

To highlight the differences between OpenMP and MPI again, OpenMP uses threads created by the operating system and has the notion/option of thread pools while MPI is just processes.

How to specific number of threads when using OpenMP. There are two main ways to do so. First you can set the environment variable “OMP_NUM_THREADS=#.” This, however, hardcodes all parts of the program. Also, everytime you close and open a shell, you have to reset the environment variable. The other way is to specify the number of threads for each part is by calling a function and setting it before executing the parallelized code.