CMSC416: Introduction to Parallel Computing

Topic: Performance Metrics Ctd. + Shared Memory & OpenMP
Date: February 27, 2024

Performance Metrics Ctd.

- Output of profiling tools
    - Coarsest: **flat profile** - A flat profile has the least amount of information, just a list of all functions and their execution times.
    - Next coarsest: **call graph profile** - A call graph profile contains per node information about the time spent in each function.
    - Least coarse: **calling context tree** - A calling context tree contains the most amount of information.
- Hatchet background
    - Hatchet is a tool to help analyze the performance of parallel programs.
    - Hatchet uses dataframes in pandas for analysis.
    - The main difference between Hatchet and using pandas dataframes directly is that Hatchet provides a graph structure that stores the calling context tree.
    - Hatchet allows us to index into the pandas dataframe using the nodes from the graph.
- pandas background
    - pandas is a popular open-source Python library intended for data analysis.
    - A **dataframe** in pandas is a 2d table data structure.
    - pandas allows us to work with high-dimensional data in a 2d data structure.
- GraphFrame
    - A GraphFrame contains both a structured index graph object and a pandas dataframe.
    - The graph contains the caller-callee relationships, and the dataframe has the tabular data.
- Dataframe operation: **filter**
    - filter is a higher-level API on top of pandas.
    - We can use filter to remove the rows from the dataframe we do not need.
    - In Hatchet, the graph operation squash (explained below) is also included in the filter operation so that the dataframe and the graph remain in sync.
- Graph operation: **squash**
    - squash removes the appropriate nodes from the graph, creating a smaller graph without the removed nodes.
        - The parent-child relationships in the graph can change when the nodes are removed.
    - squash is applied automatically in Hatchet when you apply filter so that the graph and the dataframe are in sync.
- Graphframe operation: **subtract**

    - We can subtract two GraphFrame objects to get the element-wise difference between the dataframes and the graphs.
    - The subtract operation allows us to compare execution times between GraphFrame objects without needing to pull up separate windows side by side.
    - The nodes in the resulting GraphFrame after the subtraction have the differences between the execution times.
- Visualizing small graphs
    - Hatchet is not supposed to be a visualization or graphics tool but provides utilities meant only for small graphs.
    - Terminal visualization
        - print(**gf.tree**)
        - This creates a string representation that prints the parent/child relationships and the default metric for each node (typically time).
    - Dot
        - **gf.to_dot()**
        - This creates a graph output that can be used by graphviz.
    - Flame graph
        - **gf.to_flamegraph()**
        - This shows an inverted tree visualization of where the most time is being spent.
- Hatchet Examples
    - ex 1: generating a flat profile
        - We can use pandas to do a groupby of nodes with the same name and then sum the time spent.
        - If we then sort the resulting values by time, we can get a flat profile showing the functions that are taking the most time.
    - ex 2: comparing two executions with the subtract operation
        - **gf.drop_index_levels()**
            - This drops whatever the second index is in the dataframe.
            - This aggregation allows you to do the subtract operation to get the difference in times between two graph frames.
    - ex 3: speedup and efficiency
        - We can provide Hatchet a set of graphframes and have it calculate the speedup and efficiency for each.
        - The high level API Chopper has the function **speedup_efficiency**.
            - It requires a set of GraphFrame objects, a boolean indicating strong or weak scaling, and another boolean indicating which metric you are interested in (speedup or efficiency).
            - The function looks at each node independently and what speedup you get for each function in the program.
            - It creates a new dataframe that has the speedup and efficiency for each function.

        - We can use the results to find functions where the efficiency is bad and we are spending a lot of time in them because it is less important that a function is inefficient if we are not spending much time in it.
    - ex 4: load imbalance
        - Hatchet lets us look at load imbalance per node, meaning we can see the load imbalance per function.
        - Use the **gf.load_imbalance** function.

Shared Memory & OpenMP

- Shared memory programming
    - If there are multiple cores on a node, in shared memory programming all cores have access to all memory but some parts of memory are easier to reach than others (NUMA architecture).
    - Every thread can access all the memory.
    - There is no explicit communication between threads because they can access the memory being used by other threads.
    - The user has to manage conflicts and how threads are accessing and modifying data.
- OpenMP
    - OpenMP is a shared memory programming model.
    - It is usually only used for on-node parallelization because there is no way to automatically access another node's memory.
        - This means that we can only run OpenMP within a node (in modern contexts).
    - OpenMP is primarily intended for programs/computational kernels that use arrays and loops.
    - To implement a program in parallel using OpenMP, we usually only need small code changes.
    - OpenMP is a language extension to parallelize C/C++/Fortran.
    - The programmer uses compiler directives and optionally library routines to denote parallel regions and say how to parallelize them but the compiler is what figures out how to make the code multi-threaded. This is not the job of the user.
- Fork-join parallelism
    - There is a single thread of control in OpenMP programs.
    - The master thread spawns worker threads each time it encounters a parallel region.
    - The threads join after each parallel task is over and then the master thread spawns new threads if there is another parallel region.
- Race conditions
    - Race conditions are created by unintended sharing of variables.
        - As an example, one thread writing to memory that other threads are trying to access can cause a race condition.

        - We know there is a race condition when the program outcome depends on the scheduling order of threads instead of being determined by the user explicitly.
        - We want the same output no matter the scheduling of the threads.
    - We need to synchronize access and/or change how data is stored and which threads have access to which data to prevent data races.
- Pragmas
    - Pragmas are compiler directives in C and C++.
    - Pragmas are how user communicates with compiler, denoting what the parallel region is and how the user wants it to be parallelized.
    - The compiler can ignore the pragmas without the proper flags.
    - Pragmas always start with “#pragma omp” and are followed by a construct and sometimes clauses.
- Hello World in OpenMP
    - In all OpenMP programs, always include **<omp.h>**.
    - Example pragma:
        - #pragma omp parallel
        - The word parallel is the construct here, and it means that the user wants to parallelize the following code block.
    - Compiling
        - Compiling OpenMP programs is built in to c and c++ standard.
        - For gcc, we need to pass the flag **-fopenmp**, but if we are using a different compiler the flag might be different.
    - We can set the number of threads with the environment variable by entering “export **OMP_NUM_THREADS=NUM**” on the command line.
        - Do this before running the executable.
        - This would hardcode the number of threads for all OpenMP programs.
- The most popular construct is parallel for.
    - The parallel for construct directs the compiler that the immediately following for loop should be parallelized.
    - If there are nested for loops, only the first one would be parallelized.
        - If you want an inner for loop parallelized you would have to separately add another parallel for at the top of that for loop.
    - If there are two for loops one after the other, only the first one is parallelized.
    - A good use case is a for loop where there are no read/write dependencies across loop iterations and no threads are modifying the same data.
    - In parallel for execution, the threads divide the iterations of the loop among themselves.
- Setting the number of threads
    - One option already discussed is to use the environment variable.
        - ex: **export OMP_NUM_THREADS=NUM**
        - This is global and applies for the entire execution.
    - **void omp_set_num_threads (int num_threads)**
        - This can be called from within the code for the following parallel region.

        - This only affects the immediately following code block.
    - **int omp_get_num_procs(void)**
        - This returns the number of available cores, not to be confused with the number of processors.
        - We can use this value to decide how many threads to create.