CMSC416: Introduction to Parallel Computing

Topic: Performance (Hatchet) / OpenMP
Date: February 27, 2024

Lecture Summary

In today's lecture, we discussed the profile analyzer "Hatchet" and began learning the shared memory parallelizing model "OpenMP," not to be confused with OpenMPI.

Performance

First, recall these terms:

- "*Calling Context"*: a function call, which is differentiated from identical calls with the path taken to get to that call. For example, a calling context for Assignment 1 could be (Main -> Compute -> MPI_Send).
- "*Calling context tree*" (CCT): A tree that maps out every calling context in a program. Each calling context is a node in the tree. This is a common output for performance profiling tools.
- "*Call graph*": A simplified calling context tree where all nodes with the same name are combined and all connections maintained. In other words, each node usually has multiple arrows going into it since functions may be called from different places. These are generally less useful than CCT's since they provide less information, but this could be advantageous in a large program.

One way to analyze a calling context tree generated by a profiling tool is the Python tool "Hatchet."

Hatchet

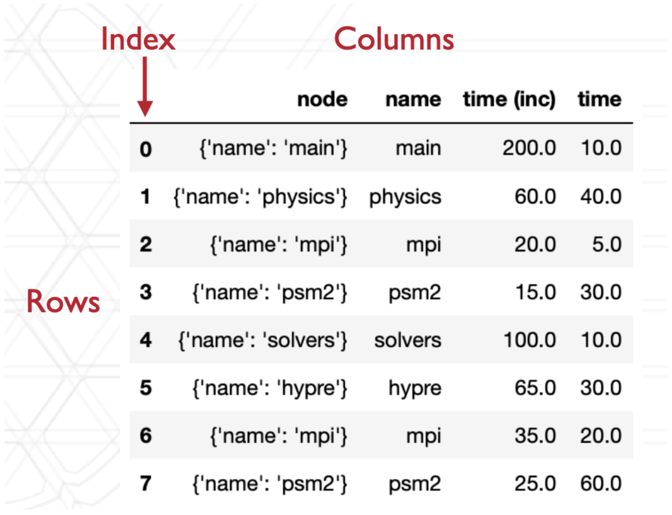
Hatchet is a simple Python tool for analyzing performance profiles. Other tools exist, but we are focusing on this one because Hatchet was developed by the Professor. Hatchet takes the raw output from a profiling tool such as HPCToolkit, puts it in a neat table format, and allows the user to do Python operations on the data such as data trimming, data comparison, and displaying. I will discuss these operations in detail shortly. Here is a simple Python program that uses Hatchet (taken from the Hatchet documentation):

```
import hatchet as ht

if __name__ == "__main__":
        dirname = "hatchet/tests/data/hpctoolkit-cpi-database"
        gf = ht.GraphFrame.from_hpctoolkit(dirname)
```

This could be saved in any Python file, for example "hatchet_program.py." Notably, Hatchet stores its data in a "DataFrame" object from the popular Pandas library. Although not necessary for this course, this means that any Pandas operations one may know how to do will work on Hatchet's graph frame dataset. Typically, Hatchet users call their dataset a "graph frame" (gf), though, as far as I am aware, this is a misnomer since Hatchet actually stores a calling context tree.

GraphFrame


Index
Columns
Rows

	node	name	time (inc)	time
0	{'name': 'main'}	main	200.0	10.0
1	{'name': 'physics'}	physics	60.0	40.0
2	{'name': 'mpi'}	mpi	20.0	5.0
3	{'name': 'psm2'}	psm2	15.0	30.0
4	{'name': 'solvers'}	solvers	100.0	10.0
5	{'name': 'hypre'}	hypre	65.0	30.0
6	{'name': 'mpi'}	mpi	35.0	20.0
7	{'name': 'psm2'}	psm2	25.0	60.0

Picture taken from the Professor's Performance slides

Hatchet stores its data in a "GraphFrame," which is essentially a map of row dictionaries. In other words, calling the following method on the GraphFrame pictured above (assuming its called "*gf*") would result in the following:

print(gf[0])

Output: {'node': {'name': 'physics'}, 'name': 'main', 'time (inc)': 200.0, 'time': 10.0}

We accessed the graph frame's 0th row and printed it out, and the result was a dictionary of all the data in that row. Here is all the data in each row:

- Node: a unique object for a particular calling context .
- Name: the name of the node.
- Time (inc): the amount of time spent in this function and all its children.
- Time: the amount of time spent exactly in this function, not including any children. This is often referred to as "exclusive" time.

Now, let us discuss common Hatchet methods.

Common commands:

1. *Filter*
    a. This method removes nodes that don't meet the filter requirement from the graph frame.
    b. Note: historically, this method did not remove does from the underlying calling context tree. Of course, this suggests that there are actually two key objects in Hatchet: the underlying calling context tree and the graph frame which inspects the underlying calling context tree. However, I presume this led to confusion, so Hatchet now handles synchronizing the underlying calling context tree and the graph frame, so calling "*filter*" and removing nodes from the graph frame will also remove the corresponding nodes from the underlying calling context graph.
    c. **Usage**: *filtered_gf = gf.filter(lambda x: x['time'] > 5.0)*
2. *Squash*
    a. This method removes nodes from the underlying calling context tree that aren't in the graph frame.
    b. DO NOT CALL THIS METHOD. Hatchet will call this automatically whenever you call "filter" to ensure that the underlying calling context tree and the graph frame are in sync.
    c. **Usage**: squashed_gf = filtered_gf.squash()
3. *Subtract*
    a. This operation returns a graph frame resulting from subtracting performance time in identical nodes from two input graph frames. This can help track performance differences between similar graph frames, such as graph frames generated from running on 4 processes and 16 processes.

   b. Note: if the input graph frames are not identical, Hatchet will make “dummy” nodes with a time of 0 until both graph frames are exactly identical and then performs the subtraction operation.
   c. **Usage**: *my_gf = gf2 - gf1*

4. *Displaying*
   a. There are a few methods for visualizing Hatchet’s graph frame, each with strengths and weaknesses.
   b. **Simple approach**: *print(gf.tree(color=True))*
      i. This approach prints the graph to the terminal (stdout). Hatchet will print the time of each node and color relatively “fast” methods green and “slow” ones red. This is best for small graphs since larger graphs won’t fill well in a terminal window.
   c. **Standard approach**:
      with open(my_dot_file.dot’, ‘w’) as my_dot_file:

      my_dot_file.write(gf.to_dot())
      i. This approach makes a nice image for the graph. Each node is an oval and is connected in a tree format.
   d. **Other approach**:
      with open(my_flamegraph.txt, ‘w’) as flamegraph_file:

      flamegraph_file.write(gf.to_flamegraph())
      i. This approach makes a “flamegraph” of the graph frame, where each node is a rectangle that spans horizontally as long as it is active and internal function calls go downward.

5. *Advanced*
   a. Hatchet is also capable of more complex operations, such as creating a calling graph from the calling context tree, comparing the time across two executions of the same program, finding the efficiency as processes are increased, and finding load imbalance across processes. Please refer to the Performance slides (slides 67-74) if you would like more specific details on these operations.

OpenMP (Shared Memory)

Shared Memory Model

First, let us review shared memory systems. In these, all nodes can access all memory, though not necessarily at the same speed. This can make things easier to program, but it also creates the possibility for data races. One common shared memory model is threading.

OpenMP

OpenMP is a way to implement threads in C/C++. Notably, it can only allow for parallelization within one single node (which will, of course, have multiple cores to run threads). Generally, OpenMP requires much less set up than OpenMPI, but it is perhaps also more prone to error. OpenMP is best used in programs with arrays and loops, the reason for which will be clear shortly.

How to use OpenMP

1. Include the omp header file omp.h.
2. Give a compiler directive (#) that tells OpenMP to parallelize the line immediately below the # line.
3. Compile the program with the -fopenmp compile flag (i.e. *gcc -fopenmp my_program.c -o my_program.out*)

OpenMP will actually do the parallelization for you! It will replace the "compiler directive" (recall: simply a hashtag (#) followed by specific words) with the code needed to parallelize the singular command beneath it. OpenMP operates on a "fork-join" model; in other words, everything is serial on a "master thread" until the parallel task. Then, control is split between threads by OpenMP. Finally, control is returned to the master thread after the parallel threads complete their task.
Here is an example program in OpenMP, taken from the OpenMP slides:

```
#include <omp.h>

int main(...) {
        omp_set_num_threads(4);

        #pragma omp parallel
        printf("Hello world\n");
        return 0;
}
```

The output of this program would be:

```
Hello world
Hello world
Hello world
Hello world
```

Let's digest this. We set the number of threads we want to run while we are on the master thread:

```
omp_set_num_threads(4);
```

Then, we give the OpenMP compiler directive to tell OpenMP to parallelize the line beneath it:

```
#pragma omp parallel
```

Finally, we give the command we want to run on 4 threads:

```
printf("Hello world\n");
```

And then control returns to the master thread and exists as expected:

```
return 0;
```

The net result is the "*printf*" statement is run on 4 threads. We could have chosen any number of threads, up to the number of cores on a Zaratan node. Luckily, we can just call

```
omp_get_num_procs(),
```

which will return the max number of threads we could run. Before we move further, we should address the elephant in the room.

Data Races

"Race conditions" are portions of code in a shared memory system that will have different behavior depending on threads' random scheduling order. One common "race condition" is two threads reading and writing the same variable simultaneously. This is very easy to accidentally create in an OpenMP code, and they can be challenging to debug since the errors do not always present themselves every run through. The common way to stop race conditions is with synchronization. For now, the professor simply wants us to be mindful of race conditions and try to avoid them. With that out of the way, let's keep exploring what we can do with OpenMP.

For Loops

Here is the real strength of OpenMP. We can parallelize the completion of a for loop by splitting indices across threads. Here is an example:

```
#include <omp.h>
…
omp_set_num_threads(4);
int my_array[1000000];
```

```
#pragma omp parallel for
for (i = 0; i < 1000000; i++) {
        my_array[i] = i + 7;
}
…
```

In this program, OpenMP will parallelize the for loop below the compiler directive, as expected. However, it will also split the for loop work across the 4 threads such that each thread gets ¼ of the work. One thread could get i=0-249999, another might get i=250000-499999, etc. This may execute faster than simply going through the array serially.

Key OpenMP Commands:

1. *#include <omp.h>*
2. *#pragma omp parallel*
3. *omp_set_num_threads(num_threads);*
4. *omp_get_num_procs()*

Resources:

1. Performance slides
2. Hatchet Documentation
3. OpenMP slides

Acknowledgements

I referenced the above slides and the Hatchet documentation while making these notes.