Utilizing Multi-threading in Order to Optimize Processing

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science
University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree
Bachelor of Science, School of Engineering

Chi Min Jung
May, 2020

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Signature _____ Date _____
Chi Min Jung

Approved _____ Date _____
Henning Mortveit, Department of Systems and Information Engineering

Utilizing Multi-threading in order to Optimize Processing

Ryan Jung and Henning S. Mortveit
Network Systems Science and Advanced Computing;
Department of Engineering Science and Environment;
Charlottesville Virginia, United States
University of Virginia
cj9de, hsm2v (at) virginia.edu

Abstract - In the world today, in all different fields data is enormously abundant. So much so that it is nearly impossible to be able to process and make use of it by the mere layman. Enter the computer, a machine able to perform thousands of calculations of one person. However even the mighty computer has to spend time to process this data. Eventually, with enough data and processes, a computer will no longer be able to instantaneously produce results. The solution is to enable parallelism, or in layman's terms do multiple things at the same time. This technical report covers the threaded implementation of the regular expression (or NFA) constrained router of Jakob et al. This was a previously built router using regular expression, was modified to be able to run parallel instances, so that multiple requests could be handled at once. Instead of processing requests one at a time, the

1 Introduction

The algorithm used in this software involved taking an existing NFA and a graph to produce an auxiliary graph. Using this, an adaptation of Dijkstra's algorithm is used to traverse the graph. Dijkstra's algorithm is an algorithm used to find the shortest path between 2 points. It does this by essentially going through every possible path and keeping track of whatever is the shortest. The modification made was that edges of the vertexes have the same edge label that follow with the NFA. This allows for not only traversal with only a specific set of nodes, but allows also for a number of patterns that the program can map out. [1].
This is performed by utilizing state machines with states that define each mode of transportation. Each state only transitions when a different mode of transportation is encountered and cannot return to the previous state. For example if the NFA was defined as walking-car-bike, and the algorithm encounters a node with a car transportation label, from that point on, that particular path can only follow edges with either the same labels or the label that comes next in the NFA. The software is set up as a router, in which it takes in a text file of requests, each line containing the start and end nodes. Other files that define the NFA and the graph are also included. After the router processes the requests, it produces a text file that contain the nodes that produce the shortest path per request.

2 Design

The basis of the design takes from the original implementation. Quite simply, the focus is on the critical portion of the code base where:

The NFA is created ,The trip requests are processed, The program outputs all the processed request to a text file

Taking this, these implementations are stored in a callable method that can be stored in a thread. An additional method needed to be built that takes in the multiple thread requests and splits them up among however many threads the user decides to run. Alongside these changes, many legacy features were removed to help make the program more lightweight and simpler to understand. An option to allocate a

certain number of cores was also added for testing purposes. The idea behind these changes is by allowing for threading and paralleling operations, the program can handle more requests more quickly and efficiently, making full use of a machine's capabilities.

Once these ideas were firmly set out, the critical sections of the code are as follows.

```
while(!trip_list.empty()){
    Trip_Request trip_request
    = trip_list.back();
    float distance = 0.0;
    plan.path.claer();
    double time_elapsed;
    router.findpath(Algorithm)
        algorithm,
        trip_request, plan
        time_elapsed,
        trip_request.nfaID);
    mtx1.lock();
    out_file << trip_request.id << '\t'
        << trip_request.source << '\t'
        << trip_request.destination
        << '\t';
    outfile << plan << endl;
    mtx1.unlock();
    bool error = false;
    .
    .
    .
    trip_list.pop_back
}
```

Figure 2: Critical components of code

Compared to the original code base, it is quite similar, with much of the components being adapted from the original. Major differences include the addition of locks around file output to ensure thread safety.

A custom "Thread Request" method was also created in to work in conjunction with the Thread method. The function of this is to split up a single request across an appropriate number of threads (can be user generated or defaulted to the number of cores existing on the machine) so that the work can be split up among them.

```
vector<Trip_Request> temp;
vector<vector<Trip_Request> > big_list;
while (!request_vector.empty())
{
    if (count >= vec_size)
    {
        count = 0;
        big_list.push_back(temp);
        temp.clear();
    }
    else
    {
        temp.push_back(request_vector.back());
        request_vector.pop_back();
        count++;
    }
}
if (!temp.empty())
{
    big_list.push_back(temp);
    temp.clear();
}
return big_list
```

Figure 3: Thread Request

3 Scaling Studies

Outline of work:

- Construct e.g. a Python tool that takes as argument the node file of a network, and an integer N, and that produces a complete trip request file containing N random trips. They will all have travel mode 0 which corresponds to automobile. It will likely be useful to have trip request files with 1,000, 10,000, and 100,000, requests. For testing, use the smaller request files; for the serious scaling studies, use the larger one.

- Create a timing framework that can time and report the execution of the router. It may be helpful to separately time initialization code such as construction of networks.

- Create a timing diagram giving time needed for computation as a function of the number of requested compute threads. Conclusions? Linear scaling? If not, why not? What happens if you request more threads than there are available cores? For each timing run, there may be fluctuations due to other computations running on your computer/Rivanna. It will likely be useful to at least conduct two runs per setting.

4 Results

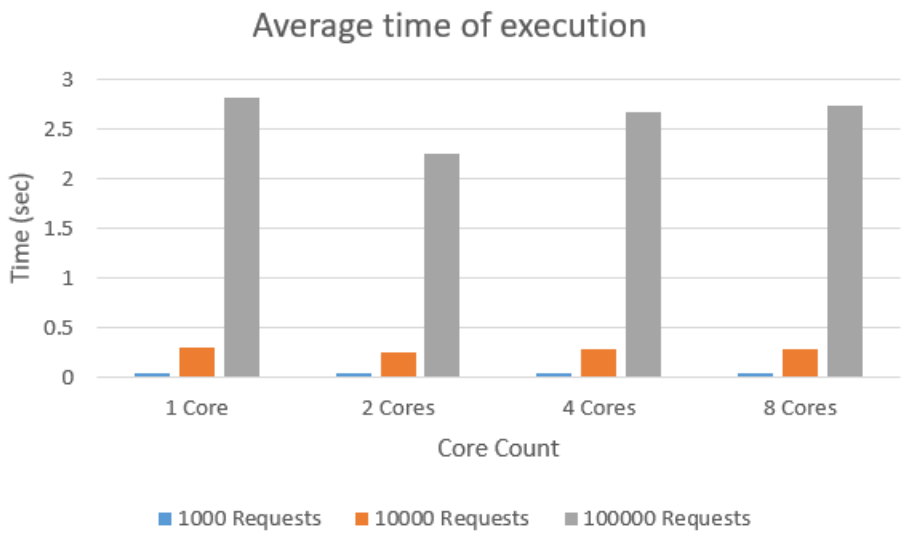

Figure 4: Data

Interestingly, the results show that across the board utilizing only 2 cores would produce the best results. All these tests were performed on a local machine and the only explanation that could be thought of is that the OS is utilizing the other cores for other overhead operation which could cause a slowdown. Further testing on Rivanna with dedicated nodes and cores performing the tests seems like it would be necessary. This was not the expected result where it was initially hypothesized that the relationship between average time and core count would be logarithmic. Still using multiple cores is almost always faster than using one core (save for the test of 1000 trip requests). Thus the goal of applying and implementing the multithreading capabilities was not only useful, but succeeded in solving the problem. Although the 100,000 requests sounds like a huge amount, it is merely s small piece of the total amount of requests that will have to be processed. As this continues to scale, it is safe to conclude that the same trends will follow and therefore be able to save much needed time.

References

[1] Chris Barret, Rico Jacob, and Madhav Marathe *Formal Language Constrained Path Problems* Society for Industrial and Applied Mathematics, 2000

A RE_Router User Documentation

```
./new_main -g <graph> -c <coords> -N <NFAFILE> -s <cores> -t <time> -f <pairs>

  -c <coords>: coordinates (vertex) file

  -g <graph>: pairs from file

  -N <NFAFile>:graph (edge) file

  -s <cores>: specifiying how many cores (and consequently threads) are used

  -t <time>: time of departure

 plans.txt: output file returning request path traversal

./trip_maker.py: prompts user for a number and creates that many trip requests

./super_script.sh:runs program with newly created pairs file from trip_maker.py with 4 cores

./testing_frame: prompts user for number of requests and number of cores.
   Runs 100 trials and returns the average time and all trial times to a .csv file

./data_collect.py: runs 3 tests of 1000, 10,000, and 100,000 requests. Returns 3
   .csv files with results
```

A.1 Input Files

The router uses the following input files:

- Network node file: The text file with specifications of node id and travel options
- Network link file: The text file that puts together the nodes
- Trip Request file: The text file that puts out specific travel routes