BFS Analytical Performance Model

Duncan Bart

March 2024

1 Algorithm Overview

```
 1 void bfs(int *level, int *parent, std::vector<float> *G, int src, int
       num_nodes)
 2     for (int i = 0; i < num_nodes; ++i)
 3         level[i] = -1;
 4         parent[i] = -1;
 5
 6     std::queue<int> queue;
 7
 8     level[src] = 0;
 9     queue.push(src);
10
11     while (!queue.empty())
12         int u = queue.front();
13         queue.pop();
14
15         for (int v = 0; v < num_nodes; ++v)
16             if (G[u][v] > 0 && level[v] == -1)
17                 parent[v] = u;
18                 level[v] = level[u] + 1;
19                 queue.push(v);
```

Listing 1: Naive sequential BFS algorithm, implemented in c. Braces are omitted for compactness.

Lines 2-4 For lines 2-4, we have a for loop which is executed *num_nodes* times. In this for loop, we have two assign operations on the index of an array which is increased by one each iteration. This means that the memory access patterns are optimal, so caching is used optimally.

Lines 6-9 Lines 6-9 we assume to have negligible runtime, since they are executed only once. Could be benchmarked as well.

Lines 11-19 Lines 11-19 are the place where the main algorithm takes place. For simplicity, we assume that the graph G only has a single connected component. This means that the content of the while loop stated in line 11 is executed once for every node. Furthermore, if the graph only has a single connected component, lines 17-19 are also executed exactly once for each node.

We assume that due to branch prediction, the check for an empty queue on line 11 is instant, since it will be predicted correctly every iteration except for the last. We also assume this for the check $v < num\_nodes$ in line 15.

So in the end we have *num_node* iterations of the while loop, which have one `queue.front()` and a `queue.pop()`, and then a for loop with *num_nodes* iteration. The content of this for loop that will be executed each iteration has an addition operation for increasing the loop counter, a memory read of the graph and a memory read of the level. The conditions of the if statement are true *num_nodes* times, so the content will be executed *num_nodes* times.

Since the pattern in which the nodes are visited is depending on the topology of the graph, it is very difficult to make any assumptions about the memory access patterns of lines 17 and 18. Thus, we assume that `parent[v]` and `level[u]` both are cache misses. However, since `level[v]` was already fetched for the if statement, this will be a cache hit.

2 The Symbolical Model

We assume a write back cache, meaning that a memory write takes the same time as a memory read.

$$\begin{aligned}
T_{BFS} &= nT_{init} + nT_{while\_loop} + nT_{visit\_node} \\
T_{init} &= 2T_{mem\_write} \\
T_{while\_loop} &= T_{q\_front} + T_{q\_pop} + n\left(T_{add} + T_{G\_mem\_read} + T_{mem\_read}\right) \\
T_{visit\_node} &= 2T_{DRAM} + T_{L1} + T_{add} + T_{q\_push} \\
T_{mem\_write} &= T_{mem\_read} = (1 - MR_{L1})T_{L1} + MR_{L1}(1 - MR_{L2})T_{L2} + \\
&\quad MR_{L1}MR_{L2}(1 - MR_{L3})T_{L3} + MR_{L1}MR_{L2}MR_{L3}T_{DRAM} \\
MR_{L\{1,2,3\}} &= \frac{1}{cacheLineSize_{L\{x\}}/sizeof(\texttt{int})} \\
T_{G\_mem\_read} &= (1 - G\_MR_{L1})T_{L1} + G\_MR_{L1}(1 - G\_MR_{L2})T_{L2} + \\
&\quad G\_MR_{L1}G\_MR_{L2}(1 - G\_MR_{L3})T_{L3} + \\
&\quad G\_MR_{L1}G\_MR_{L2}G\_MR_{L3}T_{DRAM} \\
G\_MR_{L\{1,2,3\}} &= \frac{1}{cacheLineSize_{L\{x\}}/sizeof(\texttt{float})}
\end{aligned}$$

With n being the number of nodes in the graph.

T_{q_front}, T_{q_pop}, and T_{q_push} are the times for the queue operations `queue.front()`, `queue.pop()`, and `queue.push()`. These times should be benchmarked. Furthermore T_{add} is integer addition, $T_{L\{x\}}$ is the access time to a specific cache level, and T_{DRAM} is the access time to DRAM.

3 Calibration

For compilation, we use g++, thus the size for both integer and float is 4 bytes. The variables that must be benchmarked are: T_{mem_front}, T_{q_pop}, T_{q_push}, T_{add}, and the cache/DRAM access times.

3.1 Anton

Using `getconf -a | grep CACHE` we find the cache sizes of L1, L2, and L3 to be 32kB, 1MB, and 22MB respectively. The cache line size for all levels is 64 bytes.

For memory speed benchmarking, we use the `lat_mem_rd` script from LMBench. The results are shown in Figure 1. The figure shows clear increases in fetch time for the different cache levels. For some reason, stride 256 has a very high DRAM latency, the reason for this is unclear to me...

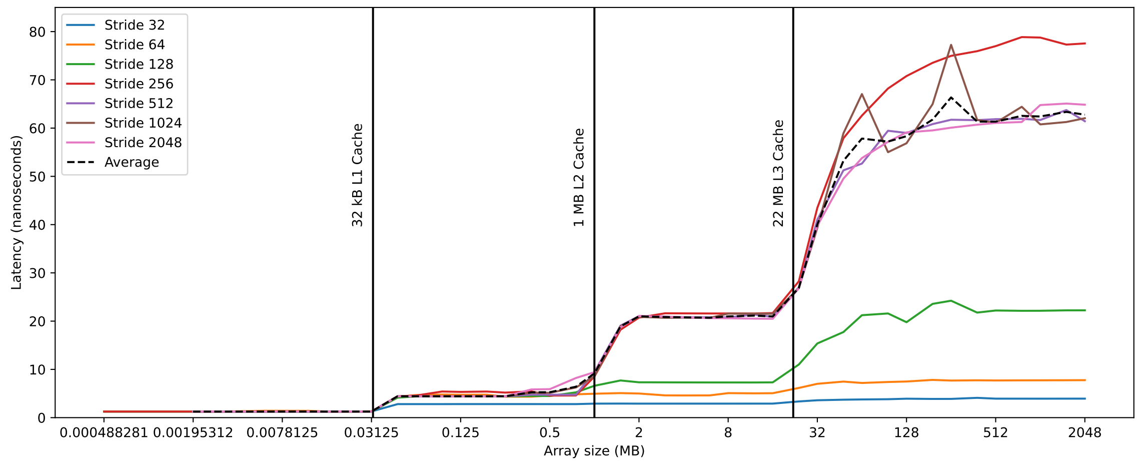


Figure 1: Memory latency benchmarking results from LMbench.

The observed memory latencies are listed in Table 1. The obtained values are the average stable values for the highest three stride values (depicted by the black dotted line in Figure 1).

Memory level	latency (ns)
L1	1.26
L2	4.42
L3	20.9
DRAM	62.5

Table 1: Observed memory latency times.

These values give the following miss rates and memory access times:

$$\begin{aligned} MR_{L\{1,2,3\}} &= G\_MR_{L\{1,2,3\}} = \frac{1}{64/4} = \frac{1}{16} \\ T_{mem\_write} &= T_{mem\_read} = T_{G\_mem\_read} \\ &= \frac{15}{16}1.26 + \frac{1}{16}\frac{15}{16}4.24 + \frac{1}{16}\frac{1}{16}\frac{15}{16}20.9 + \frac{1}{16}\frac{1}{16}\frac{1}{16}62.5 \\ &= 1.521484375 \end{aligned}$$

The operation latencies are microbenchmarked, by running each operation 1 billion times, and measuring the execution time in seconds. This is the execution time of a single operation in nanoseconds. Macros are used to minimize the effect of loop overhead. The observed operation latencies are listed in Table 2.

Operation	latency (ns)
queue push	16.1
queue front	14.5
queue pop	11.2
integer add	0.326

Table 2: Observed operation latency times.

When substituting all the obtained values into our symbolical model, we get the following:

$$\begin{aligned}
T_{BFS} &= nT_{init} + nT_{while\_loop} + nT_{visit\_node} \\
T_{init} &= 2 \cdot 1.521484375 \\
T_{while\_loop} &= 14.5 + 11.2 + n\,(0.326 + 1.521484375 + 1.521484375) \\
T_{visit\_node} &= 2 \cdot 62.5 + 1.26 + 0.326 + 16.1
\end{aligned}$$

This gives:

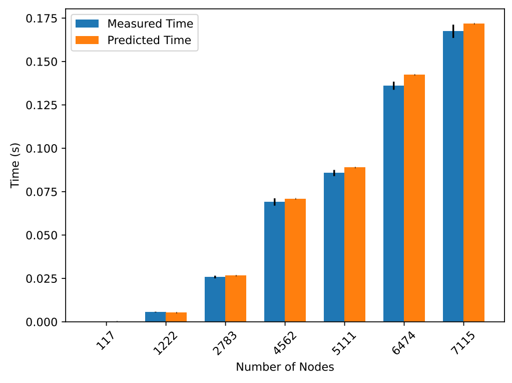
$$\begin{aligned}
T_{BFS} = {} & n(2 \cdot 1.521484375 + 14.5 + 11.2 + n\,(0.326 + 1.521484375 + 1.521484375) \\
& + 2 \cdot 62.5 + 1.26 + 0.326 + 16.1) \\
& = 171.429n + 3.36897n^2
\end{aligned}$$

4 Accuracy

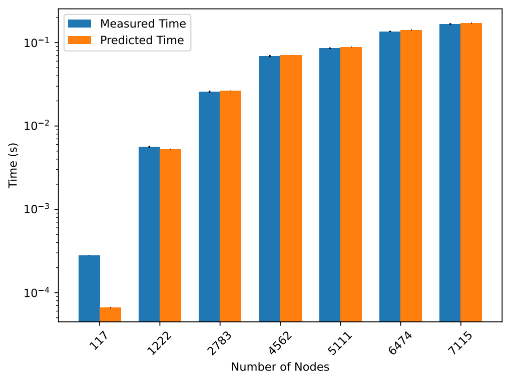
To test the accuracy of the obtained model, we run the algorithm for a variety of input graphs from the Konect database[1] with various number of nodes and report their runtimes. We use the callibrated model ($171.429n + 3.36897n^2$) to make predictions, which are compared against the observed runtimes in Figure 2.

The plots show accurate results, except for the small graph with 117 nodes. The reason for this is probably the fact that we do not include lines 6-9 in the model, which takes up a more significant amount of time for small graphs.

[1]http://konect.cc/networks



(a) Normal y-scale



(b) Logarithmic y-scale

Figure 2: BFS performance prediction vs reported runtimes. Black bars indicate the standard deviation of 10 runs.

4.1 Fully Connected Graphs

Interesting to see is if the model is overly pessimistic for fully connected, since it assumes that `parent[v]` and `level[u]` both are cache misses, but in fully connected graphs, this will not be the case.

Thus, we conduct an experiment where we test on fully connected graphs of sizes.... TODO..... The results are listed in **??**.

5 File locations

The bfs source code is implemented in the file
∼/bgo/bfs.cpp.
The microbenchmarks for queue operations and integer addition can be found in
∼/benchmarks/microbenchmarks.cpp.
All reported runtimes of the benchmarks can be found in csv format in
∼/benchmarks/results/bfs_benchmark.csv.
The figures are stored in the directory ∼/benchmarks/results/, which is also where the visualization code is, in the file visualize.ipynb.
The graphs used for the experiments are listed in ∼/data/bfs_test/. The largest connected component was selected using a python script, and the resulting subgraphs are stored in the files ending with "_largest_cc.mtx".