Searching for a Parent Instead of Fighting Over Children: A Fast Breadth-First Search Implementation for Graph500

Scott Beamer
Krste Asanovic
David A. Patterson

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-117
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-117.html

November 15, 2011

Acknowledgement

The authors would like to thank Intel for their hardware donations which enabled this work and Jason Reidy and Henry Gabb for providing access. Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instru- ments, Nokia, NVIDIA, Oracle, and Samsung.

Searching for a Parent Instead of Fighting Over Children: A Fast Breadth-First Search Implementation for Graph500

Scott Beamer, Krste Asanović, David Patterson
sbeamer@eecs.berkeley.edu, krste@eecs.berkeley.edu, pattrsn@eecs.berkeley.edu

*The Parallel Computing Laboratory,
Electrical Engineering & Computer Science Department,
UC Berkeley*

November 15, 2011

Abstract

This report provides a summary of an efficient breadth-first search implementation that is advantageous for social networks. This implementation uses a hybrid approach, combining a conventional top-down algorithm along with a novel bottom-up algorithm. The bottom-up algorithm can dramatically reduce the number of edges examined, which in turn accelerates the search as a whole. This hybrid approach is used to make a fast implementation for the Graph500 Benchmark [2], achieving 5.1 GTEPs at scale=28 (256M vertices with 4B undirected edges) on a quad-socket 40-core Intel Xeon compute node. It ranks 19th out of 50 on the November 2011 Graph500 Rankings.

1 Introduction

Breadth-First Search (BFS) is a key building block for many graph analysis algorithms, but due to its low computational intensity and low locality (both spatial and temporal), it is usually memory bound. We present a hybrid approach that includes a new algorithm that is advantageous for small-world graphs because it accesses and processes less data. This early report is intended to briefly summarize the implementation used by `mirasol` (5.1 GTEPs at scale=28) for the November 2011 Graph500 rankings. A more thorough evaluation of the algorithm, including an investigation of when it is advantageous and comparing it to prior work is underway, and will be submitted soon. Once released, that publication will supercede this one.

2 Weakness of The Conventional Top-Down Algorithm

A conventional BFS implementation can be thought of as a top-down approach, which starts at the search key and propagates down the created BFS tree during each step (Figure 1). Figure 2 shows a classic implementation of a single step of this top-down approach. Each vertex in the frontier attempts to become the parent of all of its neighbors. This approach results in every edge in the main connected component being traversed.

Implementations of this same basic algorithm can vary in a number of performance-impacting ways, including: data structures, traversal order, parallel work allocation, partitioning, synchronization, or update procedure. The process of checking if neighbors have been visited can result in many costly random accesses. An effective optimization for shared memory machines with large last-level caches is to use a bitmap to mark nodes that have already been visited [1]. The bitmap can often fit in the last-level cache, which prevents many of those random accesses from going out to DRAM.

Social networks have a low effective diameter, so when performing a BFS, even if the graph has hundreds of millions of vertices, the vast majority of them will be reached in the first several steps. Since the BFS starts from a single vertex, this means the size of the frontier ramps up and down exponentially in order to reach so many vertices in so few steps. This exponential growth follows from the defining properties of a social network: scale-free and small-world. Table 1 shows the breakdown of a typical search on a scale=27 graph when traversed by a parallel queue-based top-down traversal, such as the Graph500 omp-csr reference code. The middle steps (2 and 3) consume the vast majority of the runtime, which is unsurprising since the frontier is then at its largest size, requiring many more edges to be examined.

During these steps, there are a great number of wasted attempts to become the parent of a neighbor, since if a vertex is at depth d in the BFS tree, any of its neighbors at depth $d-1$ could also be its parent. Depending on the implementation, neighbors may contend to become the parent of an unvisited vertex. Wasted attempts come not only from other parents in the same step, but the vertex also could have already been visited in the previous step. Using a bitmap to mark visited vertices can reduce the time spent checking neighbors, but each vertex on the frontier is still attempting to become the parent of all of its neighbors. These failed attempts represent redundant work, since a vertex in a correct BFS tree only needs to have one parent.

The theoretical minimum for the number of edges that need to be examined is the size of the

```
function breadth-first-search(graph, key)
  frontier ← {key}
  next ← {}
  tree ← [-1,-1,...-1]
  while frontier ≠ {} do
    top-down-step(frontier, next, tree)
    swap(frontier, next)
    next ← {}
  end while
  return tree
```

Figure 1: Classical BFS Algorithm

```
function top-down-step(frontier, next, tree)
  for v ∈ frontier do
    for n ∈ neighbors(v) do
      if tree(n) = -1 then
        tree(n) ← v
        next ← next ∪ {n}
      end if
    end for
  end for
```

Figure 2: Top-Down Algorithm for a single step

BFS tree minus one, since that is how many edges are required to connect it. For the example in Table 1, only 63,036,116 vertices are in the BFS tree, so at least 63,036,115 edges need to be considered, which is about $\frac{1}{67}^{th}$ of all the edge examinations that would happen during a top-down traversal. This factor of 67 is substantially larger than the input degree of 16, and is caused by two reasons. First, the input degree is for undirected edges, but during a top-down search, both endpoints of each edge will check it, which doubles the number of examinations. Secondly, there are a large number of vertices of zero degree, which reduce the size of the main connected component, which also further increases the effective degree of the vertices it contains. There is clearly substantial room for improvement by checking fewer edges.

3 Bottom-Up Algorithm

When the frontier is large, there exists an opportunity to perform the BFS traversal more efficiently by searching in the reverse direction, i.e. going bottom-up. Instead of each vertex in the frontier attempting to become the parent of *all* of its neighbors, each unvisited vertex

Step	Frontier Size	Fraction of Runtime	Edge Examinations	Failed Attempts	Fraction Failed
0	1	0.00002	242	0	0
1	242	0.01836	5,055,487	2,031,553	0.402
2	3,023,934	0.63358	2,902,729,050	2,847,737,876	0.981
3	54,991,174	0.32917	1,309,552,404	1,304,547,038	0.996
4	5,005,366	0.01755	5,870,543	5,855,182	0.997
5	15,361	0.00133	15,406	15,368	0.997
6	38	0.00001	38	38	1.0
Total	63,036,116	1.0	4,223,223,170	4,160,187,055	0.985

Table 1: Typical BFS on a scale=27 graph (128M vertices with 2B undirected edges)

```
function bottom-up-step(frontier, next, tree)
  for v ∈ vertices do
    if tree(v) = -1 then
      for n ∈ neighbors(v) do
        if n ∈ frontier then
          tree(v) ← n
          next ← next ∪ {v}
          break
        end if
      end for
    end if
  end for
```

Figure 3: Bottom-Up BFS Algorithm for one Step

attempts to find *any* parent among its neighbors. A neighbor is a parent if the neighbor is a member of the frontier, and this can be determined efficiently if the frontier is represented by a bitmap. The advantage of this approach is that once a vertex has found a parent, it does not need to check the rest of its neighbors. Figure 3 shows a single step of this algorithm.

The bottom-up algorithm also removes the need for some atomic operations when parallelized. In the top-down algorithm, there could be multiple writers to the same child, so atomic operations are needed. With the bottom-up approach, the child writes to itself, so there is only one writer, which removes any contention concerns. The bottom-up algorithm is advantageous when a large fraction of the vertices are in the frontier, but it will result in more work if the frontier is small, so an efficient BFS algorithm must combine this with the top-down algorithm.

4 Hybrid Implementation Design

Our hybrid approach uses the top-down algorithm for steps when the frontier is small and uses the bottom-up algorithm for steps when the frontier is large. The runtime for the top-down algorithm is proportional to the size of the frontier, while the runtime for the bottom-up approach is roughly proportional to the number of unvisited nodes divided by the size of the frontier. This pairing is complementary, since when the frontier is its largest, the bottom-up algorithm will be at its best whereas the top-down algorithm will be at its worst, and vice versa.

Switching algorithms requires converting the frontier, which is represented by a standard queue for the conventional top-down algorithm and by a bitmap for the bottom-up algorithm. Converting from the queue to the bitmap can be done most efficiently when integrated into the last top-down step before the first bottom-up step. The integrated transition and top-down step populates the bitmap directly, and is faster than the normal top-down algorithm since it does not refill the queue. Converting the bitmap back to the queue when switching from bottom-up to top-down does not take much time since the frontier is quite small at that point.

To control the hybrid approach, cutoff parameters based on the current and predicted next size of the frontier are used to determine when to switch algorithms, and Figure 4 shows the overall control logic. The next frontier size can be predicted efficiently by summing the degrees of all of the vertices in the current frontier. This prediction is an overestimate, since vertices will be counted multiple times and some of them will have already been visited. When a BFS search begins and the frontier is only the start vertex, this prediction is exact, however, as the search continues, the prediction becomes a greater overestimate. At the step before the frontier is its largest, the prediction will have an error roughly proportional to the degree. If the prediction is greater than the number of vertices divided by the degree, the prediction must be substantially overestimating, so the frontier will be its largest soon.

The top-down algorithm starts the search, and continues until the frontier becomes too big. Initial testing has shown that it is best to switch right before the frontier becomes its largest. If the controller can guess one step before the switch, it can use the transition top-down algorithm to accelerate the conversion work (guarded by c_1). If the size of the frontier ramps up suddenly, the top-down algorithm can bypass the transition top-down algorithm and go straight to the bottom-up algorithm (guarded by c_2). The bottom-up algorithm continues until the frontier becomes small, and then switches back to the top-down algorithm. Making the switch at exactly

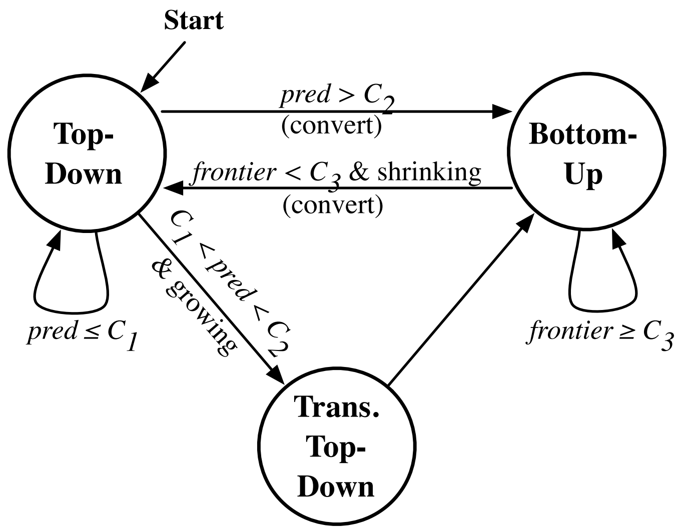

Figure 4: Control algorithm for hybrid approach. (convert) indicates the frontier must be converted from a queue to a bitmap or vice versa between the steps.

the right step is less critical since the late steps take much less time, no matter which algorithm is used.

Using these insights, we developed the following heuristics for the controller. Let n be the number of vertices, e be the number of undirected edges, $k = \frac{e}{n}$ be the average degree, and F be the frontier, the heuristics for the cutoffs are:

$$pred = \sum_{v \in F} degree(v)$$
$$c_1 = \frac{n}{2k}$$
$$c_2 = 2n$$
$$c_3 = \frac{n}{2k}$$

The factors of 2 in the cutoffs are due to k being only half the effective degree, since the traversal will attempt to search from both sides of each undirected edge.

5 Results

The results (Table 3) for Graph500 were run on `mirasol`, a large memory, quad-socket multicore (Table 2). Using the hybrid approach greatly reduces the number of edges checked, which

Parameter	Value
Architecture	Westmere-EX
Model	Intel Xeon E7-8870
Clock rate	2.4 GHz
Cores/socket	10
Threads/socket	20
LLC/socket	30 MB
# Sockets	4
DRAM Size	256 GB

Table 2: `mirasol` specifications (LLC is Last-Level Cache)

Scale	Avg. Search Time (s)	Search Rate (MTEPs)	Checks per Vertex in BFS Tree	Reduction in Checks (x)
27	0.424	5067.7	2.822	24.14
28	0.838	5125.5	2.554	27.76
29	1.734	4954.4	2.878	25.64

Table 3: Hybrid approach on Graph500 graphs

explains the high performance. As shown from the results for large searches (Table 3), the vast majority of edges can go unchecked. As described in Section 2, the potential number of checks performed per vertex is substantially larger (>64) than the input degree of 16 because the input degree counts undirected edges, and zero degree vertices inflate the effective degree for the rest of the graph. The step when the algorithm switches from the top-down to the bottom-up avoids the most edges because the vertices in the frontier after the switch never get to test their edges. During the top-down algorithm, each vertex in the frontier checks all of its edges, but during the bottom-up algorithm, each unvisited vertex (which by definition is not in the frontier), checks some of its edges.

The MTEPs measure is computed according to the Graph500 specifications, where it is based on the number of undirected edges in the traversed connected component. The scale=29 result does not use the validator since there is not enough memory capacity to hold both the input edge list needed for validation and the loaded graph.

6 Implementation Details

The implementation is written in C++/OpenMP and links in the validator and generator from the Graph500 reference code (version 2.1.4 [2]). It uses a standard Compressed Sparse Row (CSR) layout to hold the graph, and each undirected input edge is represented as two directed edges in the loaded graph. We implement our own bitmaps to include atomic versions of all of the needed operations, but STL [3] `vector<bool>`s were sufficient for most of the development. Node identifiers are implemented using 64-bit integers to meet the required size of at least 48-bits.

7 Frequently Asked Questions

In talking to others about these results, some questions have arisen:

Q: Is the Graph500 input graph sufficiently representative to predict performance on all other graphs?

A: No, we believe it would be better if a graph competition included a suite of potential graphs rather than just one. An inspiring example is the Sparse Matrix community which uses a variety of sparse matrices to evaluate innovations [4]. We would be happy to participate in such an endeavor.

Q: What impact does degree have?

A: If the input graph has a low-effective diameter, the degree is the biggest predictor of how much faster this hybrid approach will be.

Q: What about high diameter graphs?

A: When performing BFS on a high diameter graph, the frontier should not get large enough to trigger the change to the bottom-up algorithm, so it will continue with the conventional top-down algorithm. Even if an adversarial graph is constructed to have a frontier just large enough to cause the switch, the crossover point is a heuristic selected where both algorithms have comparable runtime, so performing a step with the bottom-up algorithm should not have significantly worse performance.

Q: What about graphs with many small connected components?

A: Like high diameter graphs, searches on small connected components should not generate a frontier large enough to trigger the switch.

Q: How robust is the heuristic for switching algorithms?

A: The heuristic makes the right decision the vast majority of the time. The few times it makes a premature or late switch, the bottom-up approach is so much faster during the advantageous steps, that the overall search time is still faster than the conventional purely top-down approach.

8 Conclusion

Performing a BFS in the opposite direction by going bottom-up can substantially reduce the number of edges traversed because a child needs to find only one parent instead of a parent attempting to claim all possible children. This technique is advantageous when the search is on a large connected component of low-effective diameter because this will cause the frontier to be a substantial fraction of the vertices. Using a conventional top-down approach works well for the beginning and end of such a search, since the frontier will be a small fraction of the vertices. A hybrid approach can effectively combine these two algorithms, and a simple heuristic of the number of vertices divided by the degree provides guidance of when to switch.

9 Acknowledgements

The authors would like to thank Intel for their hardware donations which enabled this work and Jason Reidy and Henry Gabb for providing access. Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung.

References

[1] V Agarwal, F Petrini, D Pasetto, and David A Bader. Scalable graph exploration on multicore processors. *Supercomputing*, 2010.

[2] Graph500 benchmark. `http://http://www.graph500.org/`.

[3] Standard template library. `http://www.sgi.com/tech/stl/`.

[4] The university of florida sparse matrix collection. `http://www.cise.ufl.edu/research/sparse/matrices/`.