Scientific Programming 21 (2013) 137–148
DOI 10.3233/SPR-130370
IOS Press

Direction-optimizing breadth-first search [1]

Scott Beamer [*], Krste Asanović and David Patterson

Electrical Engineering and Computer Sciences Department, University of California, Berkeley, CA, USA
E-mails: {sbeamer, krste, pattrsn}@eecs.berkeley.edu

Abstract. Breadth-First Search is an important kernel used by many graph-processing applications. In many of these emerging applications of BFS, such as analyzing social networks, the input graphs are low-diameter and scale-free. We propose a hybrid approach that is advantageous for low-diameter graphs, which combines 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. On a multi-socket server, our hybrid approach demonstrates speedups of 3.3–7.8 on a range of standard synthetic graphs and speedups of 2.4–4.6 on graphs from real social networks when compared to a strong baseline. We also typically double the performance of prior leading shared memory (multicore and GPU) implementations.

Keywords: Graph algorithms, breadth-first search

1. Introduction

Graph algorithms are becoming increasingly important, with applications covering a wide range of scales. Warehouse-scale computers run graph algorithms that reason about vast amounts of data, with applications including analytics and recommendation systems [17, 20]. On mobile clients, graph algorithms are important components of recognition and machine-learning applications [19,28].

Unfortunately, due to a lack of locality, graph applications are often memory-bound on shared-memory systems or communication-bound on clusters. In particular, Breadth-First Search (BFS), an important building block in many other graph algorithms, has low computational intensity, which exacerbates the lack of locality and results in low overall performance. To accelerate BFS, there has been significant prior work to change the algorithm and data structures, in some cases by adding additional computational work, to increase locality and boost overall performance [1,9,16, 27]. However, none of these previous schemes attempt to reduce the number of edges examined.

In this paper, we present a hybrid BFS algorithm that combines a conventional top-down approach with a novel bottom-up approach. By examining substantially fewer edges, the new algorithm obtains speedups of 3.3–7.8 on synthetic graphs and 2.4–4.6 on real social network graphs. In the top-down approach, nodes in the active frontier search for an unvisited child, while in our new bottom-up approach, unvisited nodes search for a parent in the active frontier. In general, the bottom-up approach will yield speedups when the active frontier is a substantial fraction of the total graph, which commonly occurs in small-world graphs such as social networks.

The bottom-up approach is not always advantageous, so we combine it with the conventional top-down approach, and use a simple heuristic to dynamically select the appropriate approach to use at each step of BFS. We show that our dynamic on-line heuristic achieves performance within 25% of the optimum possible using an off-line oracle. Our hybrid implementation also provides typical speedups of 2 or greater over prior state-of-the-arts for multicore [1,11,16] and GPUs [21] when utilizing the same graphs and the same or similar hardware. An early version of this algorithm [5] running on a stock quad-socket Intel server was ranked 17th in the Graph500 November 2011 rankings [15], achieving the fastest single-node implementation and the highest per-core processing rate, and outperforming specialized architectures and clusters with more than 150 sockets.

2. Graph properties

Graphs are a powerful and general abstraction that allow a large number of problems to be represented

[1]This paper received a nomination for the Best Paper Award at the SC2012 conference and is published here with permission from IEEE.

[*]Corresponding author. E-mail: sbeamer@eecs.berkeley.edu.

and solved using the same algorithmic machinery. However, there is often substantial performance to be gained by optimizing algorithms for the types of graph present in a particular target workload.

We can characterize graphs using a few metrics, in particular their diameter and degree distribution. Graphs representing meshes used for physical simulations typically have very high diameters and degrees bounded by a small constant. As a result, they are amenable to graph partitioning when mapping to parallel systems, as they have mostly local communication and a relatively constant amount of work per vertex, which simplifies load-balancing.

In contrast, graphs taken from social networks tend to be both *small-world* and *scale-free*. A small-world graph's diameter grows only proportional to the logarithm of the number of nodes, resulting in a low effective diameter [24]. This effect is caused by a subset of edges connecting parts of the graph that would otherwise be distant. A scale-free graph's degree distribution follows a power law, resulting in a few, very high-degree nodes [4]. These properties complicate the parallelization of graph algorithms to efficiently analyze social networks. Graphs with the small-world property are often hard to partition because the low diameter and cross edges make it difficult to reduce the size of a cut. Meanwhile, scale-free graphs are challenging to load-balance because the amount of work per node is often proportional to the degree which can vary by several orders of magnitude.

3. Conventional top-down BFS

Breadth-First Search (BFS) is an important building block of many graph algorithms, and it is commonly used to test for connectivity or compute the single-source shortest paths of unweighted graphs. Starting from the source vertex, the frontier expands outwards during each step, visiting all of the vertices at the same depth before visiting any at the next depth (Fig. 1). During a step of the conventional top-down approach (Fig. 2), each vertex checks all of its neighbors to see if any of them are unvisited. Each previously unvisited neighbor is added to the frontier and marked as visited by setting its parent variable. This algorithm yields the BFS tree, which spans the connected component containing the source vertex. Other variants of BFS may record other attributes instead of the parent at each node in the BFS tree, such as a simple boolean variable that marks whether it was visited, or an integer representing its depth in the tree.

```
breadth-first-search(vertices, source)

frontier ← {source}
next ← {}
parents ← $[-1, -1, \ldots, -1]$
while frontier $\neq$ {} do
    top-down-step(vertices, frontier, next, parents)
    frontier ← next
    next ← {}
end while
return tree
```

Fig. 1. Conventional BFS algorithm.

```
top-down-step(vertices, frontier, next, parents)

for v $\in$ frontier do
    for n $\in$ neighbors[v] do
        if parents[n] $= -1$ then
            parents[n] ← v
            next ← next $\cup$ {n}
        end if
    end for
end for
```

Fig. 2. Single step of top-down approach.

The behavior of BFS on social networks follows directly from their defining properties: small-world and scale-free. Because social networks are small-world graphs, they have a low effective diameter, which reduces the number of steps required for a BFS, which in turn causes a large fraction of the vertices to be visited during each step. The scale-free property requires some nodes to have much higher degrees than average, allowing the frontier growth rate to outpace the average degree. As a result of these two properties, the size of the frontier ramps up and down exponentially during a BFS of a social network. Even if a social network graph has hundreds of millions of vertices, the vast majority will be reached in the first few steps.

The majority of the computational work in BFS is checking edges of the frontier to see if the endpoint has been visited. The total number of edge checks in the conventional top-down algorithm is equal to the number of edges in the connected component containing the source vertex, as on each step every edge in the frontier is checked.

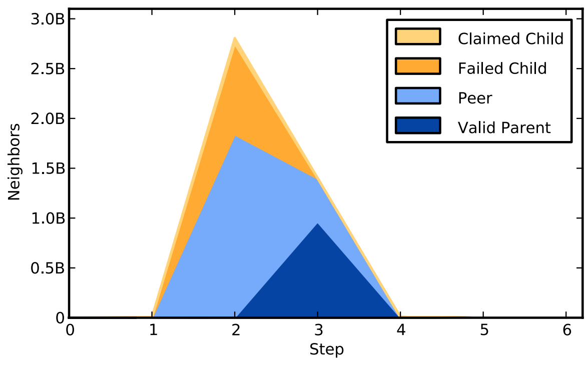

Fig. 3. Breakdown of edges in the frontier for a sample search on `kron27` (Kronecker generated 128M vertices with 2B undirected edges) on the 16-core system. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130370.)

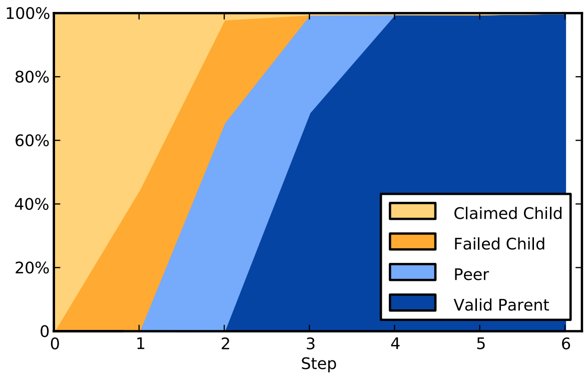

Fig. 4. Breakdown of edges in the frontier for a sample search on `kron27` (Kronecker generated 128M vertices with 2B undirected edges) on the 16-core system. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130370.)

Figure 3 shows a breakdown of the result of each edge check for each step during a conventional parallel queue-based top-down BFS traversal on a Kronecker-generated synthetic graph (used for the Graph500 benchmark [15]). 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. Failures occur when the neighbor has already been visited, and these can be broken down into three different categories based on their depth relative to the candidate parent: *valid parent*, *peer* and *failed child*. A valid parent is any neighbor at depth $d-1$ of a vertex at depth d. A peer is any neighbor at the same depth. A failed child is any neighbor at depth $d+1$ of a vertex at depth d, but at the time of examination it has already been claimed by another vertex at depth d. Successful checks result in a *claimed child*. Figure 3 shows most of the edge checks do fail and represent redundant work, since a vertex in a correct BFS tree only needs one parent.

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 touching off-chip DRAM. These optimizations speed up the edge checks but do not reduce the number of checks required.

The theoretical minimum for the number of edges that need to be examined in the best case is the number of vertices in the BFS tree minus one, since that is how many edges are required to connect it. For the example in Fig. 3, 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 for two reasons. First, the input degree is for undirected edges, but during a top-down search each edge will be checked from both endpoints, doubling the number of examinations. Secondly, there are a large number of vertices of zero degree, which reduces the size of the main connected component and also further increases the effective degree of the vertices it contains. There is clearly substantial room for improvement by checking fewer edges, although in the worst case, every edge might still need to be checked.

Figure 4 zooms in on the edge check results of Fig. 3 for the sample search. This progression of neighbor types is typical among the social networks examined. During the first few steps, the percentage of claimed children is high, as the vast majority of the graph is unexplored, enabling most edge checks to succeed. During the next few steps, the percentage of failed children rises, which is unsurprising since the frontier has grown larger, as multiple valid parents are fighting over children. As the frontier reaches its largest size, the percentage of peer edges dominates. Since the frontier is such a large fraction of the graph, many edges must connect vertices within the frontier. As the frontier size rapidly decreases after its apex, the percentage of valid parents rises since such a large fraction of edges were in the previous step's frontier.

4. Bottom-up BFS

When the frontier is large, there exists an opportunity to perform the BFS traversal more efficiently by searching in the reverse direction, that is, going bottom-up. Note that in Step 3 there are many valid parents, but the other ends of these edges mostly resulted in failed children during Step 2. We can exploit this phenomenon to reduce the total number of edges examined. Instead of each vertex in the frontier attempting to become the parent of *all* of its neighbors, each unvisited vertex attempts to find *any* parent among its neighbors. A neighbor can be a parent if the neighbor is a member of the frontier, which 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 4 demonstrates that for some steps, a substantial fraction of neighbors are valid parents, so the probability of not needing to check every edge is high. Figure 5 shows the algorithm for a single step of this approach.

The bottom-up approach also removes the need for some atomic operations in a parallel implementation. In the top-down approach, there could be multiple parallel writers to the same child, so atomic operations are needed to ensure mutual exclusion. With the bottom-up approach, only the child writes to itself, removing any contention. This advantage, along with the potential reduction of edges checked, comes at the price of serializing the work for any one vertex, but there is still massive parallelism between the work for different vertices. The bottom-up approach is advantageous when a large fraction of the vertices are in the frontier, but will result in more work if the frontier is small. Hence, an efficient BFS implementation must combine both the top-down and bottom-up approaches.

```
bottom-up-step(vertices, frontier, next, parents)

for v ∈ vertices do
  if parents[v] = −1 then
    for n ∈ neighbors[v] do
      if n ∈ frontier then
        parents[v] ← n
        next ← next ∪ {v}
        break
      end if
    end for
  end if
end for
```

Fig. 5. Single step of bottom-up approach.

If the graph is undirected, performing the bottom-up approach requires no modification to the graph data structures as both directions are already represented. If the graph is directed, the bottom-up step will require the inverse graph, which could nearly double the graph's memory footprint.

5. Hybrid algorithm

The pairing of the top-down approach with the bottom-up approach is complementary, since when the frontier is its largest, the bottom-up approach will be at its best whereas the top-down approach will be at its worst, and vice versa. The runtime for either the top-down approach or the bottom-up approach is roughly proportional to the number of edges examined. The top-down approach will examine every edge in the frontier while the bottom-up approach could examine every edge attached to an unvisited vertex, but hopefully fewer. Figure 6 illustrates this behavior by showing the time per step for each approach using the same example search as in Section 3. As the size of the frontier ramps up, the time per step of the top-down approach rises correspondingly, but the time per step for the bottom-up approach drops.

Our hybrid algorithm uses the top-down approach for steps when the frontier is small and the bottom-up approach for steps when the frontier is large. We begin each search with the top-down approach and continue until the frontier becomes too large, at which point we switch to the bottom-up approach. Although it is difficult to tell from Fig. 6, it is usually worthwhile to switch back to the top-down approach for the final steps. During some searches there can be a long tail, and edges that are not in the connected component continue to consume runtime using the bottom-up approach.

The step when transitioning from the top-down approach to the bottom-up approach provides an enormous opportunity, since all of the edge checks that would happen during the top-down step (all of the edges in the frontier) are skipped. For this reason, the optimum point to switch is typically when the number of edges in the frontier is at its largest. This is fortuitous, since the first bottom-up step will probably benefit from a high percentage of valid parents, as Fig. 4 shows.

To control the hybrid approach, we use a heuristic based on: the number of edges to check from the fron-

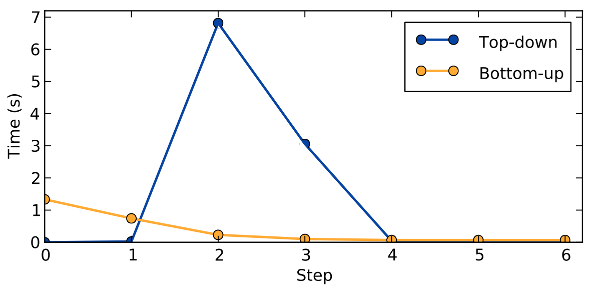

Fig. 6. Sample search on `kron27` (Kronecker 128M vertices with 2B undirected edges) on the 16-core system. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130370.)

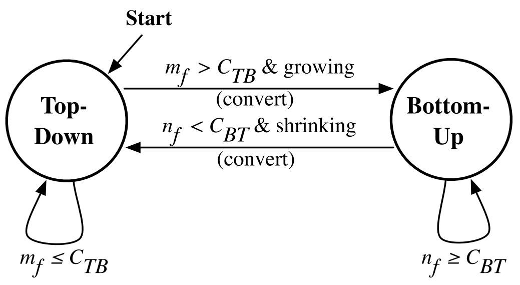

Fig. 7. Control algorithm for hybrid algorithm. (convert) indicates the frontier must be converted from a queue to a bitmap or vice versa between the steps. Growing and shrinking refer to the frontier size, and although they are typically redundant, their inclusion yields a speedup of about 10%.

tier (m_f), the number of vertices in the frontier (n_f), and the number of edges to check from unexplored vertices (m_u). These metrics are efficient to compute, since they only require: summing the degrees of all vertices in the frontier, counting the number of vertices added to the frontier, or counting how many edges have been checked. Note that with an undirected graph, m_f or m_u might be counting some edges twice since each edge will be explored from both ends. Figure 7 shows the overall control heuristic, which employs two thresholds, C_{TB} and C_{BT}, together with m_f and n_f.

Since the top-down approach will always check m_f edges, and the bottom-up approach will check at most m_u, if m_u is ever less than m_f, there is a guarantee that switching to the bottom-up approach at that step will check less edges. Since m_u is an overly pessimistic upper-bound on the number of edges the bottom-up approach will check, we use a tuning parameter α. This results in the condition for switching from top-down to bottom-up being:

$$m_f > \frac{m_u}{\alpha} = C_{TB}.$$

Switching back to the top-down approach at the end should occur when the frontier is small and there is no longer benefit to the bottom-up approach. In addition to the overhead of checking edges outside the main connected component, the bottom-up approach becomes less efficient at the end because it scans through all of the vertices to find any unvisited ones. The heuristic to switch back attempts to detect when the frontier is too small, for which we use another tuning parameter β:

$$n_f < \frac{n}{\beta} = C_{BT}.$$

When switching between approaches, the representation of the frontier must be changed from the frontier queue used in the conventional top-down approach to the frontier bitmap used for the bottom-up approach. Different data structures are used since the frontiers are of radically different sizes, and the conversion costs are far less than the penalty of using the wrong data structure. The bottom-up approach uses a frontier bitmap to allow a constant-time test for whether a particular node is in the frontier. The time to convert the frontier is typically small, since it should be done when the frontier is small.

6. Evaluation

6.1. Methodology

We evaluate the performance of our hybrid algorithm using the graphs in Table 1. Our suite of test graphs includes both synthetic graphs as well as real social networks. The synthetic graph generators as well as their parameters are selected to match the Graph500 Competition [15] (Kronecker (A, B, C) $=$ $(0.57, 0.19, 0.19)$) and prior work [1,16] (Uniform Random and RMAT (A, B, C) $=$ $(0.45, 0.25, 0.15)$). The real social networks are taken from a variety of web crawls [6–8,13,17,22,25,26].

We report results from three multi-socket server systems (8-core, 16-core, 40-core) as shown in Table 2. The 16-core system is used for most of the evaluation, as it is representative of the next generation of compute nodes for clusters. The 40-core system is used to evaluate the parallel scalability of our approach up to 80 threads. The 8-core system is used to perform a direct comparison with prior work [16] utilizing the same hardware and input graphs. On all platforms we disable Turbo Boost to get consistent performance results.

Table 1

Graphs used for evaluation

Abbreviation	Graph	# Vertices (M)	# Edges (M)	Degree	Diameter	Directed	References
kron25	Kronecker	33.554	536.870	16.0	6	N	[15,18]
erdos25	Erdős–Réyni (Uniform Random)	33.554	268.435	8.0	8	N	[3,14]
rmat25	RMAT	33.554	268.435	8.0	9	Y	[3,10]
facebook	Facebook trace A	3.097	28.377	9.2	9	N	[26]
flickr	Flickr follow links	1.861	22.614	12.2	15	Y	[22]
hollywood	Hollywood movie actor network	1.140	57.516	50.5	10	N	[6–8,13]
ljournal	LiveJournal social network	5.363	79.023	14.7	44	Y	[22]
orkut	Orkut social network	3.073	223.534	72.8	7	N	[22]
wikipedia	Wikipedia links	5.717	130.160	22.8	282	Y	[25]
twitter	Twitter user follow links	61.578	1,468.365	23.8	15	Y	[17]

Table 2

System specifications

Name	8-core	16-core	40-core
Architecture	Nehalem-EP	Sandy bridge-EP	Westmere-EX
Intel model	X5550	E5-2680	E7-4860
Release	Q1'09	Q1'12	Q2'11
Clock rate	2.67 GHz	2.7 GHz	2.26 GHz
# Sockets	2	2	4
Cores/socket	4	8	10
Threads/socket	8	16	20
LLC/socket	8 MB	20 MB	24 MB
DRAM size	12 GB	128 GB	128 GB

Our algorithm implementations use C++ and OpenMP, and store the graphs in Compressed Sparse Row (CSR) format after removing duplicates and self-loops. For each plotted data point, we perform BFS 64 times from pseudo-randomly selected non-zero degree vertices and average the results. The time for each search includes the time to allocate and initialize search-related data structures (including `parents`).

To benchmark performance, we calculate the search rate in Millions of Edges Traversed per Second (MTEPS) by taking the ratio of the number of edges in the input graph to the runtime. This metric is artificial in the sense that either the top-down approach or the hybrid approach might not check every edge, but this standardized performance metric is similar in spirit to measuring MFLOPS for optimized matrix multiplication in terms of the classic O(N^3) algorithm.

6.2. Tuning α and β

First we determine values of α and β to use for *hybrid-heuristic*, our hybrid implementation of BFS described above. We first tune α before tuning β, as it has the greatest impact. Sweeping α across a wide range demonstrates that once α is sufficiently large (>12), BFS performance for many graphs is relatively insensitive to its value (Fig. 8). This is because the frontier grows so rapidly that small changes in the transition threshold do not change the step at which the switch occurs. When trying to pick the best value for the suite, we select $\alpha = 14$ since it maximizes the average and minimum. Note that even if a less-than-optimal α is selected, the *hybrid-heuristic* algorithm still executes within 15–20% of its peak performance on most graphs.

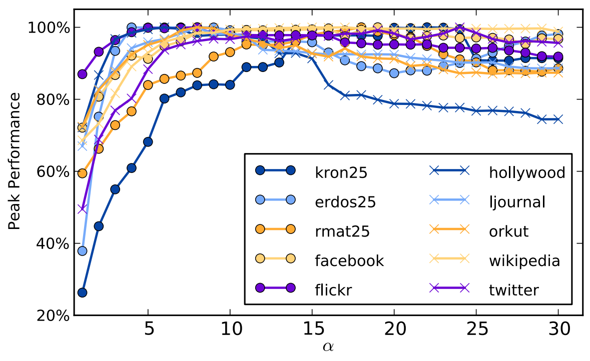

Fig. 8. Performance of *hybrid-heuristic* on each graph relative to its best on that graph for the range of α examined. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130370.)

Tuning β is less important than tuning α. We select $\beta = 24$, as this works well for the majority of the graphs (Fig. 9). The value of β has a smaller impact on overall performance because the majority of the runtime is taken by the middle steps when the frontier is at its largest, even when those steps are accelerated by the bottom-up approach. We explore a much larger range for β since it needs to change by orders of magnitude in order to have an impact on which step the heuristic will switch.

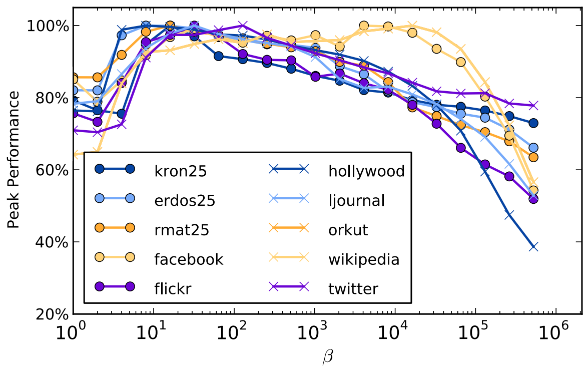

Fig. 9. Performance of *hybrid-heuristic* on each graph relative to its best on that graph for the range of β examined. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130370.)

6.3. Comparing algorithms

We first compare our own implementations of different algorithms on a range of graphs. As a baseline, we use two top-down parallel queue implementations. The *top-down* implementation uses a typical parallel queue, much like the `omp-csr` reference code [15]. We also create an optimized version, *top-down-check*, which adds a bitmap to mark completed nodes. To determine the maximum benefit of our hybrid approach and to evaluate the effectiveness of our on-line heuristic (*hybrid-heuristic*), we also develop *hybrid-oracle*, which repeatedly runs the algorithm trying all possible switch points and then reports the best-performing to represent an oracle-controlled hybrid.

We use the 16-core machine to compare both hybrid implementations (*hybrid-heuristic* and *hybrid-oracle*) against both baselines (*top-down* and *top-down-check*) in Fig. 10. The hybrid provides large speedups across all of the graphs, with an average speedup of 3.9 and a speedup no lower than 2.4. The on-line heuristic often obtains performance within 10% of the oracle. Also notice that the speedups are far greater than the impact of a mistuned α or β in the heuristic. The bottom-up approach by itself yields only modest speedups or even slowdowns, which highlights the importance of combining it with the top-down approach for the hybrid.

6.4. Explaining the performance improvement

The speedup of the hybrid approach demonstrated in Fig. 10 is due to the reduction in edge examinations (Fig. 11). In a classical top-down BFS, every edge in the connected component containing the starting vertex will be checked, and for an undirected graph, each edge will be checked from both ends. The bottom-up approach skips checking some edges in two ways. First, it passes the responsibility of setting the `parents` variable from the parent to the child, who can stop early once a parent is found. Secondly, it skips all of the edges in the frontier on the step that transitions from top-down to bottom-up. As shown by Fig. 11, the edge examinations skipped are roughly split between those skipped in the transition and those skipped by the first bottom-up step. Since the bottom-up approach is used when the frontier is large, the top-down approach in the hybrid implementation processes only a small fraction of the edges. Every edge examination after the first bottom-up step is redundant, since the first bottom-up step will have attempted to examine every edge attached to an unvisited vertex. Since the size of the frontier decreases so rapidly, this redundancy is typically negligible. Unsurprisingly, the graphs with the least speedup (`flickr` and `wikipedia`), skip fewer edges and check the most redundant edges. They have a substantially higher effective diameter, so there is a longer tail after the apex of the frontier size.

Examining where the time is spent during an entire search reveals the majority of it is spent in the bottom-up implementation (Fig. 12). Since the bottom-up approach skips so many examinations, the effective search rate for the bottom-up steps is much higher (order of magnitude) than the top-down approach. Conversion and m_f calculation take a non-negligible fraction of the runtime, but this overhead is worthwhile due to the extreme speedup provided by the bottom-up approach. The nearly constant degree of `erdos25` results in a steady but modest growth of the frontier, so more steps will be run top-down and with a larger frontier, resulting in more work at the conversion point, more degrees to sum each step, and more edge checks run in top-down mode.

Figure 13 plots the speedups for the suite compared to the reduction in number of edges checked. The speedup is reasonably correlated with the reduction in edges checked, and the slope of a best-fit line is approximately 0.3. The slope is less than 1.0 due to a variety of overheads. While the bottom-up approach skips edges, it reduces the spatial locality of the remaining memory accesses, which impacts memory performance in modern processors that bring data from memory in units of cache blocks rather than individual words. Conversion between frontier data structures and m_f calculation add other overhead. Using *top-down-check* as a baseline rather than *top-down*, further re-

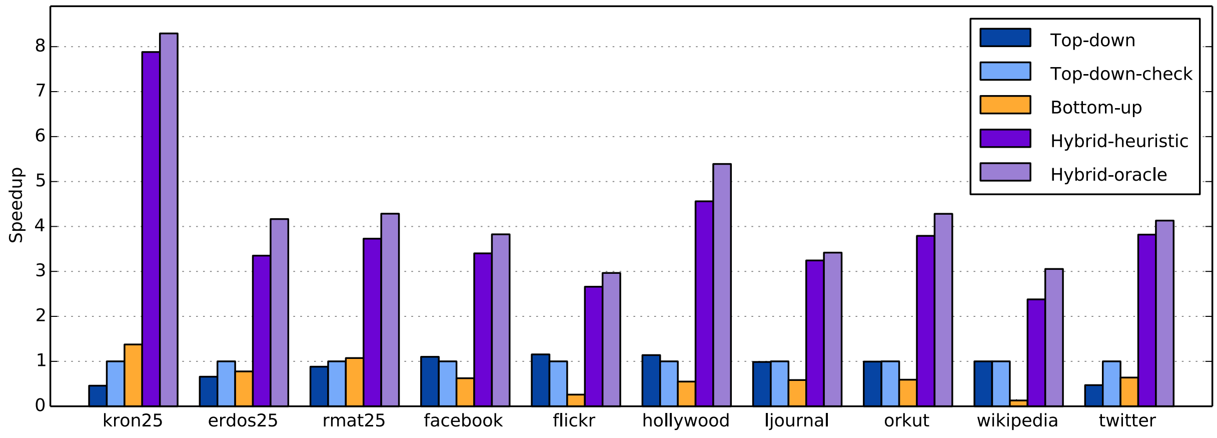

Fig. 10. Speedups on the 16-core machine relative to *Top-down-check*. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130370.)

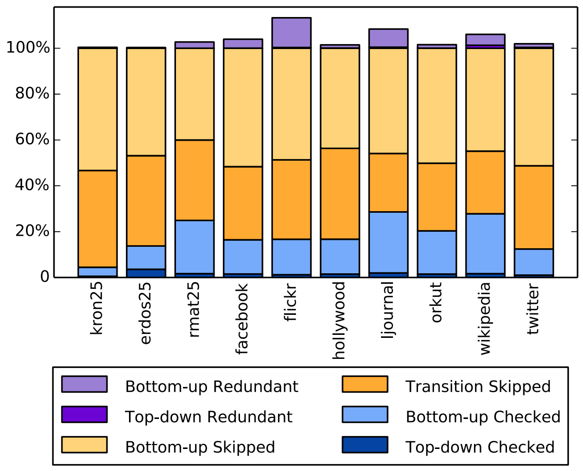

Fig. 11. Breakdown of edge examinations. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130370.)

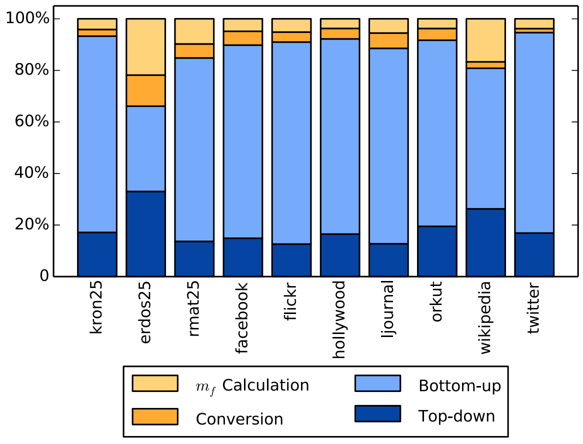

Fig. 12. Breakdown of time spent per search. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130370.)

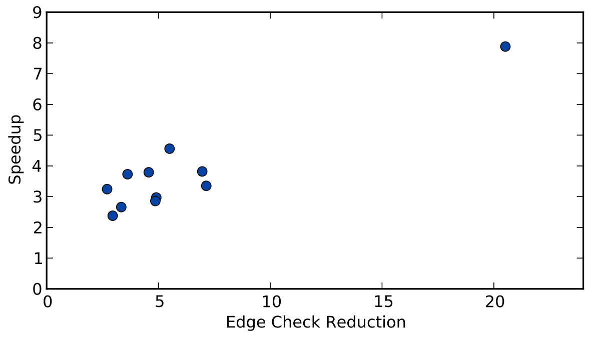

Fig. 13. Speedups on the 16-core system for *Hybrid-heuristic* compared to the reduction in edges checked. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130370.)

duces the relative advantage since edge examinations become less expensive in the baseline as they will often hit the bitmap in the last-level cache.

6.5. Scalability

We evaluate the scalability of our *Hybrid-heuristic* implementation on the 40-core system in a fashion similar to prior work [1]. Our approach sees parallel speedup with each additional core, but does not gain much from hyperthreading on the 40-core system (Fig. 14). Our approach does benefit from larger graphs with more vertices, as more computation helps to amortize away the overheads (Fig. 15). Increasing the degree is particularly beneficial for our approach because this does not significantly increase the runtime but does increase the number of input edges, which results in a higher effective search rate.

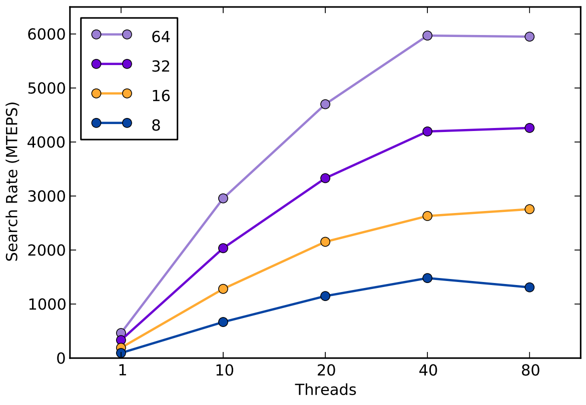

Fig. 14. Parallel scaling of *Hybrid-heuristic* on the 40-core system for an RMAT graph with 16M vertices and varied degree. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130370.)

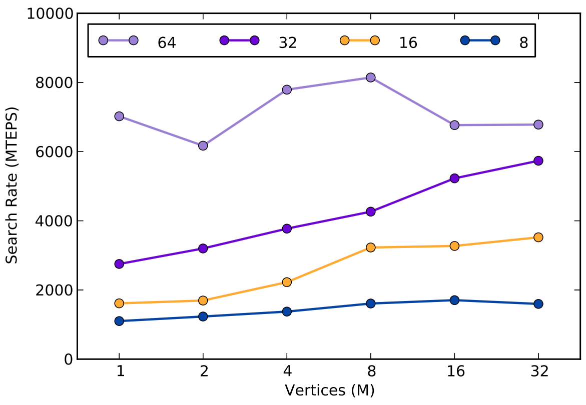

Fig. 15. Vertex scaling of *Hybrid-heuristic* on the 40-core system for an RMAT graph with varied degree. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130370.)

6.6. *Comparison against other implementations*

We also compare our hybrid implementation against previous state-of-the-art shared-memory implementations. We begin with the multicore implementation from Hong et al. [16]. Note the implementation by Hong et al. [16] is faster than that of Agarwal et al. [1] even while using slower hardware. We use the exact same model hardware and the same graph generator source code (SNAP [3]) to provide a point-for-point comparison as shown in Fig. 16. Our implementation is 2.4× faster on uniform random (`erdos25`) and 2.3× faster on scale-free (`rmat25`). Our implementation experiences a slowdown after 8 threads due to the switch to hyperthreads in our tightly synchronized approach, but using 16 hyperthreads is still noticeably faster than just 8 hyperthreads on the 8 available cores.

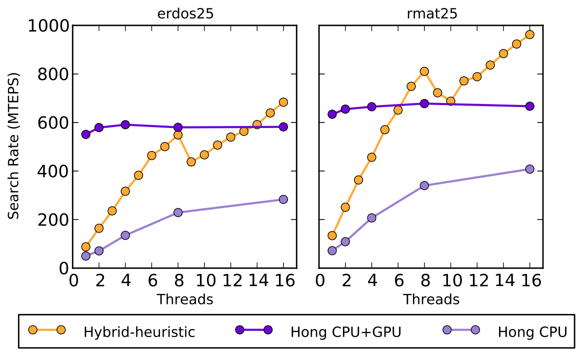

Fig. 16. Search rates on the 8-core system on `erdos25` (Uniform Random with 32M vertices and 256M edges) and `rmat25` (RMAT with 32M vertices and 256M edges). Other lines from Hong et al. [16]. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130370.)

Table 3

Performance in MTEPS of *Hybrid-heuristic* on the 8-core system compared to Chhugani et al. [11]

	rmat-8	rmat-32	erdos-8	erdos-32	orkut	facebook
Prior	750	1100	*295*	*505*	2050	*460*
8-core	1580	4630	850	2250	4690	1360

Notes: Synthetic graphs are all 16M vertices, and the last number in the name is the degree. Due to differences in counting undirected edges for TEPS, we have scaled erdos and facebook appropriately.

Furthermore, our CPU implementation using 16 hyperthreads is even faster than the GPU–CPU hybrid implementation presented by Hong et al. [16], despite the GPU having greater memory bandwidth and supporting more outstanding memory accesses. Note also that not all of the graphs in our study above would fit in the reduced memory capacity of the GPU, and that our implementation records a full parent pointer rather than just the BFS depth [16].

We attempt to compare against Chhugani et al. [11] in Table 3. Compared to 8-core, their platform has a higher clockrate (2.93 GHz vs. 2.67 GHz) and more memory (96 GB vs. 12 GB), but should otherwise be similar (Intel X5570 vs. Intel X5550). For the synthetic graphs, we compare against those with 16M vertices since they are the largest that fit in our memory. Despite using a slower system, hybrid-heuristic is faster, and sometimes by a significant degree. In general, the highest degree graphs seem to enjoy the greatest relative speedups.

We compare our results to that of Merrill et al. [21]. which is the highest published performance for shared memory (Table 4). Our *Hybrid-heuristic* performs well due to the high input degrees of the graphs used in spite of the memory bandwidth advantages of the GPUs.

Table 4

Hybrid-heuristic on multicore systems in this study compared to GPU results from Merrill et al. [21] (in GTEPS)

System	kron_ g500-logn20	random. 2Mv.128Me	rmat. 2Mv.128Me
GPU results from Merrill et al. [21]			
Single-GPU	1.25	2.40	2.60
Quad-GPU	3.10	7.40	8.30
Hybrid-heuristic results on multicore			
8-core	7.76	6.75	6.14
16-core	12.38	12.61	10.45
40-core	8.89	9.01	7.14

The GTEP rates for the GPUs for `kron_g500-logn20` have been halved since we calculated search rates for undirected graphs based on the number of undirected edges.

7. Related work

Buluç and Madduri [9] provide a thorough taxonomy of related work in parallel breadth-first searches. In this section we focus on the work most relevant to this study, principally parallel shared-memory implementations.

Bader and Madduri [2] demonstrate large parallel speedups for BFS on an MTA-2. Parallelism is extracted both at the level of each vertex as well as at each edge check. The fine-grained synchronization mechanisms of the MTA are used to efficiently exploit this parallelism. Since the MTA does not use caches and instead uses many hardware threads to hide its shared main memory latency, the implementation does not need to optimize for locality.

In contrast, Agarwal et al. [1] optimize for locality to push the limits of a quad-socket system built from conventional microprocessors, and show speedups over previous results from clusters and custom supercomputers, including the MTA-2 [2]. Careful programming is used to reduce off-socket traffic (memory and inter-socket) as much as possible. Threads are pinned to sockets and utilize local data structures such that all the work that can be done within a socket never leaves the socket. Bitmaps are used to track completed vertices to avoid reading DRAM. Whenever an edge check must go to another socket, it utilizes a custom inter-socket messaging queue.

Hong et al. [16] improve upon Agarwal et al. with hybrid algorithms that utilize multiple CPU implementations and a GPU implementation. As in our work, the algorithms switch at step boundaries and make decisions based on on-line heuristics, but in this case to select between CPU and GPU implementations of a purely top-down approach. The CPU implementation is accelerated by the read-array approach, which combines the frontier with the output array. The output provided is the depth of each node rather than its parent, and by bounding the depth to be less than 256, the output array can be compacted by storing it as bytes. Instead of appending vertices to a frontier, their output is set to depth $+ 1$. On each step, all vertices are scanned searching for the current depth, and then visits are performed from any vertex found in the frontier. By using the output array to act as the frontier, duplicate entries are squashed, and spatial locality is increased due to the sequential array accesses to the graph. The GPU implementation outperforms the CPU implementation for a sufficiently large frontier, and heuristics select which implementation to use. In Section 6.6, we compare our new scheme directly against the performance in [16] utilizing the same synthetic data on the same model CPU hardware, but without the GPU. Note our scheme retains full parent information, not just search depth.

Merrill et al. [21] improve the performance of BFS on GPUs through the use of prefix sum to achieve high utilization of all threads. The prefix sum is used to compute the offsets from the frontier expansion, which reduces the contention for atomic updates to the frontier queue. This allows the graph exploration to have little control divergence between threads. They also leverage various filtering techniques enabled by bitmaps to reduce the number of edges processed. This approach is beneficial for all diameters of graphs, and they present results indicating they are the fastest published for shared memory systems, especially with their quad-GPU parallelization. Our approach is advantageous for low diameter graphs, and as shown in Section 6.6, our approach even outperforms their quad-GPU implementation when using the 16-core platform.

Chhugani et al. [11] perform a multitude of optimizations to improve memory utilization, reduce inter-socket traffic, and balance work between sockets. Their memory performance is improved by reordering computation and data layout to greatly increase spatial locality. The load balancing techniques proposed are dynamic, and can adapt to the needs of the current graph. Furthermore, they demonstrate an analytic model derived from the platform's architectural parameters that accurately predicts performance. Many of these optimizations are complementary to our work and could be added on top of our implementation. In Section 6.6 we compare directly against them using a slightly slower system with the same graphs and demonstrate reasonable speedups.

An early implementation of our algorithm on the `mirasol` system reached 17th place in the November 2011 rankings of the Graph500 competition [5,15]. This earlier implementation used a cruder heuristic and included an alternate top-down step that integrated the conversion for bottom-up. It achieved the highest single-node performance and the highest per-core processing rate. Using just a single quad-socket system similar to the 40-core system, the hybrid BFS algorithm outperformed clusters of >150 sockets and specialized architectures such as the Cray XMT2 [23] and the Convey HC-1^{ex} [12]. Its processing rate per-core was over $30\times$ the top-ranked cluster system, highlighting the performance penalty a cluster experiences when crossing the network. This implies that scaling across a cluster should be used to solve larger problems (weak scaling) rather than solving fixed-size problems faster (strong scaling), indicating large shared-memory systems are still useful for irregular difficult-to-scale problems like BFS.

8. Conclusion

Performing a BFS in the bottom-up direction can substantially reduce the number of edges traversed compared with the traditional top-down approach. A child needs to find only one parent instead of each parent attempting to claim all possible children. This bottom-up technique is advantageous when the search is on a large connected component of low-effective diameter, as then the frontier will include a substantial fraction of the total vertices. The conventional top-down approach works well for the beginning and end of such a search, where the frontier is a small fraction of the total vertices. We have developed a hybrid approach to effectively combine these two algorithms, using a simple heuristic to guide when to switch. We believe this hybrid scheme yields the current best-known BFS performance for single shared-memory nodes.

The results of this work demonstrate the performance improvement potential of integrating the bottom-up approach into BFS, but it will not always be advantageous. Fortunately, the same top-down implementation that is used for the steps when the frontier is small can be used for entire searches when the bottom-up approach is not advantageous. The heuristic will allow this to happen automatically, as searches that do not generate a massive frontier will not trigger the switch. In this sense, the bottom-up approach can be seen as a powerful way to accelerate a BFS on a low-diameter graph.

High-diameter graphs will not benefit from the bottom-up approach, but they are much easier to partition and thus easier to parallelize than low-diameter graphs. This work presents an algorithmic innovation to accelerate the processing of more difficult-to-parallelize BFS.

Acknowledgements

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. Partially funded by DARPA Award HR0011-11-C-0100.

References

[1] V. Agarwal et al., Scalable graph exploration on multicore processors, in: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.
[2] D. Bader and K. Madduri, Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2, in: *International Conference on Parallel Processing (ICPP)*, 2006.
[3] D. Bader and K. Madduri, SNAP: Small-world network analysis and partitioning: an open-source parallel graph framework for the exploration of large-scale networks, in: *International Parallel and Distributed Processing Symposium (IPDPS)*, 2008.
[4] A.-L. Barabási and R. Albert, Emergence of scaling in random networks, *Science* **286** (1999), 509–512.
[5] S. Beamer, K. Asanović and D. Patterson, Searching for a parent instead of fighting over children: A fast breadth-first search implementation for graph500, Technical Report UCB/EECS-2011-117, EECS Department, University of California, Berkeley, 2011.
[6] P. Boldi et al., Ubicrawler: A scalable fully distributed web crawler, *Software: Practice & Experience* **34**(8) (2004), 711–726.
[7] P. Boldi et al., Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks, in: *International World Wide Web Conference (WWW)*, ACM Press, 2011.
[8] P. Boldi and S. Vigna, The webgraph framework I: Compression techniques, in: *International World Wide Web Conference (WWW)*, Manhattan, USA, ACM Press, 2004, pp. 595–601.

[9] A. Buluç and K. Madduri, Parallel breadth-first search on distributed memory systems, in: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[10] D. Chakrabarti, Y. Zhan and C. Faloutsos, R-MAT: A recursive model for graph mining, in: *SIAM Data Mining*, 2004.

[11] J. Chhugani et al., Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency, in: *International Parallel and Distributed Processing Symposium*, 2012.

[12] Convey HC-1 family, available at: www.conveycomputer.com/Resources/Convey_HC1_Family.pdf.

[13] T. Davis and Y. Hu, The University of Florida sparse matrix collection, *ACM Transactions on Mathematical Software* **38**(1) (2011), 1:1–1:25.

[14] P. Erdős and A. Réyni, On random graphs. I, *Publicationes Mathematicae* **6** (1959), 290–297.

[15] Graph500 benchmark, available at: www.graph500.org.

[16] S. Hong, T. Oguntebi and K. Olukotun, Efficient parallel graph exploration on multi-core CPU and GPU, in: *Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[17] H. Kwak et al., What is Twitter, a social network or a news media?, in: *International World Wide Web Conference (WWW)*, 2010.

[18] J. Leskovec et al., Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication, in: *European Conference on Principles and Practice of Knowledge Discovery in Databases*, 2005.

[19] Y. Low et al., GraphLab: A new framework for parallel machine learning, in: *Uncertainty in Artificial Intelligence*, 2010.

[20] G. Malewicz et al., Pregel: A system for large-scale graph processing, in: *International Conference on Management of Data (SIGMOD)*, June 2010.

[21] D. Merrill, M. Garland and A. Grimshaw, Scalable GPU graph traversal, in: *Principles and Practice of Parallel Programming*, 2012.

[22] A. Mislove et al., Measurement and analysis of online social networks, in: *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2007.

[23] D. Mizell and K. Maschhoff, Early experiences with large-scale Cray XMT systems, in: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2009.

[24] D. Watts and S. Strogatz, Collective dynamics of 'small-world' networks, *Nature* **393** (1998), 440–442.

[25] Wikipedia page-to-page link database 2009, available at: http://haselgrove.id.au/wikipedia.htm.

[26] C. Wilson et al., User interactions in social networks and their implications, in: *European Conference on Computer Systems (EuroSys)*, 2009.

[27] A. Yoo et al., A scalable distributed parallel breadth-first search algorithm on BlueGene/L, in: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2005.

[28] K. You et al., Scalable HMM-based inference engine in large vocabulary continuous speech recognition, in: *IEEE Signal Processing Magazine*, 2010.