CHAPTER

2

Edge v. Node Parallelism for Graph Centrality Metrics

Yuntao Jia, Victor Lu, Jared Hoberock, Michael Garland, and John C. Hart

Graphs help us model and understand various structures, but understanding large graphs, such as those now generated by instruments, simulations and the Internet, increasingly depend on statistics and characterizations. Centrality metrics indicate which nodes and edges are important, to better analyze, simplify, categorize and visualize large graphs, but are expensive to compute, especially on large graphs, and their parallel implementation on the most common "scale-free" graphs can suffer severe load imbalance [1]. This chapter proposes an improved edge-parallel approach for computing centrality metrics that can also accelerate breadth-first search and all-pairs shortest path.

2.1 INTRODUCTION

Centrality metrics help us analyze large graphs, but their computation can be quite time-consuming and is often the main bottleneck for graph processing. For example, computing betweenness centrality (BC) using the state-of-the-art sequential algorithm [2] on a flickr user network of 6.6 million relationships between 800,000 users takes about two days on a single thread of a 3.33 GHz Intel Core i7-975 central processing unit (CPU). Approximate BC computes faster [3], but the error can severely degrade some applications [4, 5].

Several have proposed parallel algorithms to accelerate the computation of centrality metrics. Bader and Madduri [6] implemented BC on the Cray MTA-2 shared memory multiprocessor, and Madduri et al. [7] refined the approach by reducing its need for atomic operations. Tu and Tan [8] implemented BC on Intel Clovertown and Sun Niagara1 multicore processors. To our knowledge, Sriram et al. [1] is the first and only effort to compute BC on the GPU, though similar parallel approaches have been used for GPU implementations of other graph algorithms [9]. Overall, these approaches have been shown to perform well on general graphs, but suboptimally on scale-free networks [1, 9].

Scale-free networks are graphs whose node degree distribution follows a power law [10]. They commonly result from real-world data ranging from natural, such as the protein interactions shown in Figure 2.1, to social, such as online friend networks like the aforementioned flickr example, prompting the development of several recent tools specialized for their analysis [4, 5, 11, 12]. This degree distribution, with many low-degree nodes and few high-degree nodes can severely degrade performance of parallel graph algorithms, including centrality metrics, breadth-first search (BFS) and all-pairs

GPU Computing Gems. DOI: 10.1016/B978-0-12-385963-1.00002-2

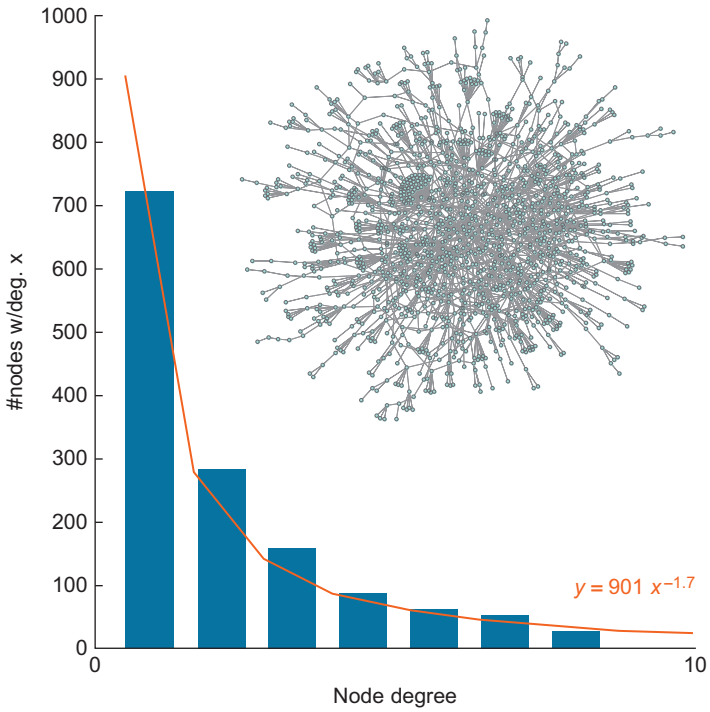

FIGURE 2.1

Node degree distribution of the scale-free network "bo".

shortest path (APSP), that operate per-node with work proportional to node degree, because the degree (workload) variance creates a load imbalance and control flow divergence.

We propose instead a per-edge-parallel approach to computing centrality metrics and other graph operations that balances computation regardless of node degree variance. This new approach comes at the expense of an increase in the number of memory reads/writes (about four to seven times more in our tests), but enables more opportunities for coalesced accesses. Overall its benefits from load balancing outweigh the increase in memory accesses.

The presented solution and discussions assume unweighted and undirected input graphs.

2.2 BACKGROUND

Centrality metrics measure the importance of a node for various definitions of important. Central nodes are close to all nodes, so closeness centrality (1) sums the distance from a node to every other node, and graph centrality (2) uses the distance from a node to its most remote counterpart. Alternatively, crucial nodes might lie on more shortest paths than others. Stress centrality (3) measures the absolute number of shortest paths passing through a node, whereas betweenness centrality (4) measures the fraction of

shortest paths passing through a vertex. Each of these can likewise measure the importance of an edge. Please note that $d_v(t)$ denotes the shortest-path distance from v to t, and $\sigma_{st}(v)$ represents the number of shortest paths from s to t through v.

$$\mathrm{CC}(v) = 1/\sum_{t\in V} d_v(t) \quad (1) \qquad \mathrm{GC}(v) = 1/\max_{t\in V} d_v(t) \quad (2)$$

$$\mathrm{SC}(v) = \sum_{s,t\in V} \sigma_{st}(v) \quad (3) \qquad \mathrm{BC}(v) = \sum_{s,t\in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \quad (4)$$

FIGURE 2.2

Closeness, Graph, Stress and Betweenness Centrality.

Among the four centrality metrics, BC is the most complicated and most popular. Because the BC algorithm can be easily modified to compute the other three centrality metrics, we focus on BC.

The state-of-art sequential algorithm for computing BC [2] runs in time $O(mn)$, where m and n are the number of edges and nodes, respectively. As outlined in Algorithm 1, it computes BC (4) as a sum of *dependencies*

$$\mathrm{BC}(v) = \sum_{s\in V} \delta_{s\cdot}(v), \quad (5) \qquad \delta_{s\cdot}(v) = \sum_{t\in V} \frac{\sigma_{st}(v)}{\sigma_{st}}, \quad (6)$$

where $\delta_{s\cdot}(v)$ is the dependency of s on v. Each outer loop iteration (lines 2–37) maps to a term in the summation of (5). Each iteration computes, for the corresponding node s, the dependencies $\{\delta_{s\cdot}(v)\colon \forall v \in V\}$, which accumulate into $\{\mathrm{BC}(v)\colon \forall v \in V\}$. Each iteration first performs a *forward propagation* phase (lines 3–23, a modified breath-first search) that computes the shortest-path (SP) distances $\{d_s(v) : v \in V\}$ and SP counts $\{\sigma_s(v) : \forall v \in V\}$. $N(v)$ denotes the nodes adjacent to v. Next, a *backward propagation* phase (lines 24–36) computes $\delta_{s\cdot}(v)$ using the recursive relation (line 31) [2]

$$\delta_{s\cdot}(v) = \sum_{w:v\in P_s(w)} \frac{\sigma_s(v)}{\sigma_s(w)}(\delta_{s\cdot}(w) + 1), \quad (7)$$

and accumulates it into $\mathrm{BC}(v)$. Here $P_s(w)$ represents the predecessors of w given the source node s.

Closeness centrality is similarly computed by a forward propagation without path counting from each node s and backward propagation replaced by (1), specifically $\mathrm{CC}(s) \leftarrow 1/\sum_{v\in V} d_s(v)$. Graph centrality likewise propagates forward without path counting, replacing backward propagation with $\mathrm{GC}(s) \leftarrow 1/\max_{v\in V} d_s(v)$. Stress centrality performs forward propagation with path counting, and back propagation, replacing lines 31 and 34 with

$$\delta_{s\cdot}(v) \leftarrow \delta_{s\cdot}(v) + \delta_{s\cdot}(w) + 1, \quad (8) \qquad \mathrm{SC}(v) \leftarrow \mathrm{SC}(v) + \sigma_s(v)\delta_{s\cdot}(v), \quad (9)$$

where $\delta_s(v)$ denotes instead the number of "shortest-path suffixes" starting at s passing through v.

Algorithm 1: Sequential BC Computation

Require: Graph $G = (V, E)$
Ensure: BC(v) at every $v \in V$

```
1:  Initialize BC(v) to 0, for all v ∈ V
2:  for s ∈ V do
3:      ▼ Forward Propagation
4:      Initialize d_s(v) to ∞, for all v ∈ V
5:      Initialize σ_s(v) to 0, for all v ∈ V
6:      d_s(s) ← 0, σ_s(s) ← 1
7:      d ← 0
8:      while not done do
9:          done ← true
10:         for v ∈ V where d_s(v) = d do
11:             for w ∈ N(v) do
12:                 if d_s(w) = ∞ then
13:                     d_s(w) ← d + 1
14:                     done ← false
15:                 end if
16:                 ▼ Path Counting
17:                 if d_s(w) = d + 1 then
18:                     σ_s(w) ← σ_s(w) + σ_s(v)
19:                 end if
20:             end for
21:         end for
22:         d ← d + 1
23:     end while
24:     ▼ Backward Propagation
25:     Initialize δ_s.(v) to 0 for all v ∈ V
26:     while d > 1 do
27:         d ← d − 1
28:         for v ∈ V where d_s(v) = d do
29:             for w ∈ N(v) do
30:                 if d_s(w) = d + 1 then
31:                     δ_s.(v) ← δ_s.(v) + (σ_s(v)/σ_s(w)) (δ_s.(w) + 1)
32:                 end if
33:             end for
34:             BC(v) ← BC(v) + δ_s.(v)
35:         end for
36:     end while
37: end for
```

Algorithm 2: Node-Parallel Forward Propagation Step

```
1: for v ∈ V on wavefront in parallel do
2:     for w ∈ N(v) do
3:         if d_s(w) = ∞ then d_s(w) ← d + 1
4:         if d_s(w) = d + 1 then σ_s(w) ← σ_s(w) + σ_s(v)
5:     end for
6: end for
```

Algorithm 3: Edge-Parallel Forward Propagation Step

```
1: for edge (v,w) incident to wavefront in parallel do
2:     if d_s(w) = ∞ then d_s(w) ← d + 1
3:     if d_s(w) = d + 1 then σ_s(w) ← σ_s(w) + σ_s(v)
4: end for
```

2.3 NODE V. EDGE PARALLELISM

To convert Algorithm 1 into a parallel GPU version, we first map the outer loop iterations (lines 2–37) to coarse-grain parallel thread blocks, then seek finer grain SIMD parallelism opportunities within each iteration. There are dependencies between iterations in both forward and backward propagation, so we look within a propagation step (lines 10–21 and 28–35).

Existing methods [1, 9] map wavefront nodes (the set of nodes with $d_s(v) = d$) to threads. We call this the *node-parallel* approach. As Algorithm 2 indicates, the inner for-loop over the node neighbors $N(v)$ of node v will vary depending on the degree of v, so threads corresponding to low-degree nodes must wait for the few higher-degree node threads in the same SIMD unit to complete, in both the forward and backward propagation steps.

These forward and backward propagation steps iterate across edges incident to nodes in the current front. Unfortunately, this configuration produces load imbalance owing to the varying work required by nodes along the front. For this reason, it is unsuited to parallelism. If we instead parallelize over the edges incident to the front directly, then each edge represents constant complexity: either an update of distance and shortest-path count or a no-op. Hence, such an *edge-parallel* approach, as shown in Algorithm 3, better balances load.

2.4 DATA STRUCTURE

We represent the graph as an edge list to better assign edges to threads. Each edge (v, w) appears twice in this list, as (v, w) and (w, v), to ensure that both "directions of computation" (forward and backward propagation) associated with each edge are properly processed. This edge list is sorted by the first node

in each ordered pair, such that adjacent threads are assigned edges that likely emanate from the same node, leading to more opportunities for coalesced access of node attributes. Likewise, this ordering also encourages edge attribute coalescing.

Graphs are commonly specified by an adjacency list, an array of lists where each array element (corresponding to a node) contains a list of array indices corresponding to neighboring nodes. This structure is often linearized for graphical processing unit (GPU) processing as an indexed neighbor list, consisting of the concatenated neighbor lists `nhbrs` indexed by an `offsets` array, such that the neighbors of a node `i` are found at `nhbrs[offsets[i]:offsets[i+1]-1]`. Each thread of a node-parallel approach would then process a varying length list of neighbors, as shown in Figure 2.3a, resulting in load imbalance and less opportunity for coalescing.

We can convert the indexed neighbor "node" list into an edge list by replacing the node-indexed offset array with an edge-indexed "from" array that aligns with the original `nhbrs[]` array, which records at `froms[j]` the node that `nhbrs[j]` is a neighbor of as shown in Figure 2.3b. Hence (`froms[j]`,`nhbrs[j]`) becomes the aforementioned edge list, and under this construction each edge would appear twice, and the edges are sorted by their first node.

There are a few additional arrays to maintain the propagation status and centrality value, including `distances`, `numSPs`, `dependencies`, `nodeBCs`, and `predecessor`. The first four correspond to the per-node quantities of Algorithm 1. The `predecessor` array records for a given node pair (v,w) whether w is a predecessor of v; `predecessor[i]` is true if `nhbrs[i]` is a predecessor of `froms[i]`.

Listing 2.1 shows the CUDA implementation of these data structures. The `cuGraph` structure is read-only and shared by all thread blocks, whereas the `cuBCData` structure is duplicated across multiple thread blocks so that each thread block can propagate independently. If `gridDim` is the number of thread blocks, then the estimated global memory usage is

$$\text{Memory usage} = 16m + \texttt{gridDim}(16n + 2m) \text{ bytes.} \tag{10}$$

The optimal number of thread blocks `gridDim` would maximize GPU parallelism without exceeding GPU memory space. We set `gridDim` to the number of SMs (30). A larger number of thread blocks would better hide latency, but we found that for BC computation, larger numbers of thread blocks saturated the memory interface. To help hide memory latency, we set the number of threads in a block, `blockDim`, as large as possible (512).

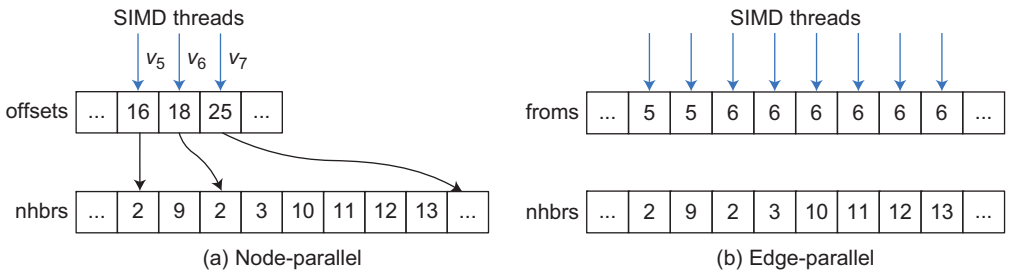

FIGURE 2.3

Illustration of data structure design.

```
struct cuBCData {
    int * distances;          // size n
    int * numSPs;             // size n
    bool * predecessor;       // size 2m
    float * dependencies;     // size n
    float * nodeBCs;          // size n
};
struct cuGraph {
    int * froms;              // size 2m
    int * nhbrs;              // size 2m
};
```

Listing 2.1. Data structures as defined for CUDA implementation of BC computation.

```
__shared__ bool done = false;
while (!done) {
  __syncthreads();
  done = true;
  d++;
  __syncthreads();

  for (int eid = threadIdx; eid < 2 * nedge; eid += blockDim) {
    int from = froms[eid];
    if (distance[from] == d) {
      int nhbr = nhbrs[eid];
      int nhbr_dist = distance[nhbr];
      if (nhbr_dist == -1) {
        distance[nhbr] = nhbr_dist = d + 1;
        done = false;
      } else if (nhbr_dist < d) {
        predecessor[eid] = true;
      }
      if (nhbr_dist == d + 1) {
        atomicAdd(&numSPs[nhbr], numSPs[from]);
      }
    }
  }
  __syncthreads();
}
```

Listing 2.2. CUDA kernel for edge-parallel forward propagation phase.

2.5 IMPLEMENTATION

CUDA algorithms for forward and backward propagation are shown in Listings 2.2 and 2.3, less array and variable initiation shown earlier in Algorithm 1. For each thread `threadIdx`, we map the set of

```
1  while (d > 1) {
2    for (int eid = threadIdx; eid < 2 * nedge; eid += blockDim) {
3      int from = froms[eid];
4      if (distance[from] == d) {
5        if (predecessor[from]) {
6          int nhbr = nhbrs[eid];
7          float delta = (1.0f + dependency[from]) *
8                        numSPs[nhbr] / numSPs[from];
9          atomicAdd(&dependency[nhbr], delta);
10       }
11     }
12   }
13   d--;
14   __syncthreads();
15 }
16 for (int nid = threadIdx; nid < nnode; nid += blockDim) {
17   nodeBC[nid] += dependency[nid];
18 }
```

Listing 2.3. CUDA kernel for edge-parallel backward propagation phase.

edges indexed by `threadIdx`$+ k *$ `blockDim`, for some $k \geq 0$ given by Line 8 in Listing 2.2 and Line 2 in Listing 2.3, which promotes coalesced accesses to edge attributes.

The forward phase propagates the wavefront until all nodes are visited, indicated by the shared boolean `done`, initialized `true` at the start of each step (line 4), but is set `false` when a wavefront node finds a successor. Lines 8–23 implement Algorithm 3.

The backward phase visits nodes in the opposite wavefront propagation order as recorded by the forward phase. Each step identifies wavefront edges whose first-node distance equals the current wavefront distance `d`. These wavefront-edge first nodes accumulate dependency to their predecessor neighbor. The distance `d` is decremented, and the phase concludes when it reaches 1. When the backward phase completes, dependency is accumulated into node BC (Lines 16–18).

The accumulations on Line 20 of Listing 2.2 and Line 9 of Listing 2.3 must occur atomically since two threads could process edges with a common second node. These accumulations always occur from the first node into the second node, either accumulating SP count (from wavefront node to successor) or dependency (from wavefront node to predecessor). This choice specifically avoids contention between concurrent threads that likely share the same first node.

The dependency accumulation of the backward phase (line 9 of Listing 2.3) requires floating-point atomic operations introduced with the Fermi architecture. For earlier GPUs such as the GTX 280, we simply perform a node-parallel backward phase. Most of the results in this chapter were obtained on a GTX 480 using an edge-parallel backward phase.

In the forward phase, the second-node distance computation requires an extra level of indirection: `distance` indexed by the result of `nhbrs`. The backward phase avoids indirection by directly checking the `predecessor` array, which unlike `distance`, is coalesced.

2.6 ANALYSIS

The edge-parallel approach is better load balanced, but it also requires more memory accesses. We analyze this trade-off using a simplified BC kernel consisting only of a forward phase that simply performs distance updates. This step reduces it to a breadth-first traversal.

The number of reads performed by the node-parallel approach is $(In + 2n + 4m)n$, where I is the average number of propagation steps from any of the start nodes, and each propagation step reads the entire n-length `distance` array. The number of reads to `offset` is $2n$, and to `nhbrs` is $2m$. Each neighbor index then reads `distance` another $2m$ times.

The edge-parallel approach reads $(4Im + 4m)n$ times. The difference is the edge-parallel approach must perform $4m$ reads in order to identify wavefront nodes ($2m$ from `froms` and $2m$ from `distance`). Because the number of edges usually exceeds the number of nodes, the edge-parallel approach incurs significantly more memory bandwidth pressure, as measured in Table 2.1.

The number of *writes* by both approaches is similar, bounded from above by $(m - \chi)n$, where χ is the average number of edges lying on any wavefront emanating from a start node. This upper bound assumes each node's distance is set (redundantly) by all of its predecessors, which is not necessarily the case. In particular, predecessors that fetch v's distance after being set by another predecessor will not write to it.

Figure 2.4 shows memory throughput from an application point of view, which is computed by dividing the estimates in Table 2.1 by GPU running time. On the GTX 480, the linear increase in throughput with `gridDim`, modulo the drops[1] at 31 and 46, suggests the memory interface is not being saturated and that the additional memory requests incurred by the edge-parallel approach are being adequately absorbed by cache. This is in contrast to the Tesla S1070, where edge-parallel's additional requests quickly saturate the memory interface as `gridDim` increases.

Table 2.2 reports memory throughput computed using hardware performance counters. This provides further evidence that bandwidth is indeed being saturated on the Tesla S1070 and not on the GTX 480.

Table 2.1 Estimated Global Memory Reads and Writes in GB

	Node-Parallel	Edge-Parallel
sp500-038	0.026	0.117
bo	0.174	0.615
cg_web	0.523	2.362
as-rel.20071008	67.470	320.598
hep-th	213.603	1636.577

[1]The drop at 31 is probably due to an unbalanced assignment of thread blocks to SMs (i.e., among the 15 SMs one will have three thread blocks, whereas the others will have two), and the drop at 46 is probably due to a limited number of thread blocks that can run concurrently.

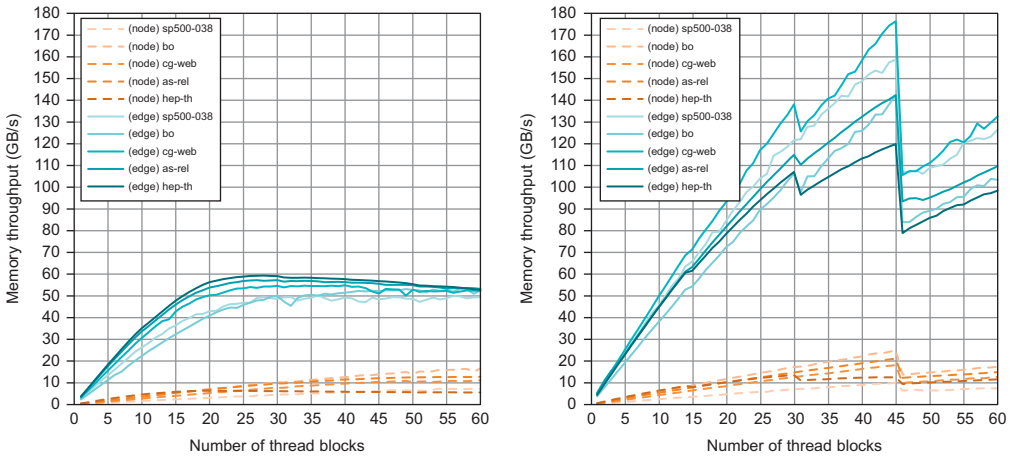

FIGURE 2.4

Memory throughput from application point of view as a function of the number of thread blocks, as measured on the Tesla S1070 (left) and the GTX 480 (right).

Table 2.2 Memory Throughput (GB/s) Measured Using Visual Profiler 3.2. Peak Throughput for the Tesla S1070 and GTX 480 are 102.5 GB/s and 177.4 GB/s, Respectively. Measurements were Performed with `gridDim` Set to 60 and 45 on the Tesla S1070 and GTX 480, Respectively

	Tesla S1070		GTX 480	
	Node-Par.	**Edge-Par.**	**Node-Par.**	**Edge-Par.**
sp500-038	45.4	63.7	2.0	5.0
bo	57.3	74.0	10.9	13.6
cg-web	58.8	69.1	6.5	9.3
as-rel	44.2	65.9	35.5	68.0
hep-th	34.8	61.4	1.6	2.2

The node-parallel approach has a lower memory throughput on both the Tesla S1070 and GTX 480. This can be attributed to SIMD underutilization and uncoalesced memory access. In particular, adjacent threads mapped to nodes with different node degree results in an idle SIMD lane that would otherwise perform useful memory operations. Unlike the edge-parallel approach, adjacent threads in general access nonadjacent items in `nhbrs`, resulting in fewer coalesced loads.

In summary, although the edge-parallel approach must pay the cost of more memory requests, the benefits of less SIMD divergence and more memory coalescing and the support of recent cache-based GPU architectures enable the edge-parallel approach to win over the node-parallel approach in the end.

2.7 RESULTS

We implemented all four centrality metrics using CUDA on the GPU and single-threaded C++ on the CPU for comparison. Except when otherwise specified, all experiments were performed on the UIUC IACAT accelerator cluster [13] of 8GB 3.33GHz Intel i7-975 CPUs and GTX 480 GPUs with 1.5GB memory. We tested these implementations on six real-world graphs of varying complexity, averaging five runtimes each shown in Table 2.3. (Practical BC computation for massive graphs like "flickr" use a random subset of nodes as wavefront start nodes. For "flickr" we used 4,096 start nodes.)

Table 2.3 shows that serial CPU computation of centrality metrics can become cumbersome on large graphs over 10,000 nodes. Figure 2.5 shows how parallelism on CPU and GPU accelerates these computations. Using OpenMP enabled a parallel BC implementation on a CPU with eight threads to be 1.6 times to 4.9 times faster except on the smallest network. Whereas our BC implementation on GTX 480 can be 6.9 times to 10.2 times faster, using 30 blocks of 512 threads. The same algorithm

Table 2.3 Test Dataset Complexity, Average Propagation Steps (I), and Centrality Metric Performance on a Single CPU Thread, in Seconds

Graph	Nodes	Edges	I	CC	GC	SC	BC
sp500-038	365	3206	5.49	0.006	0.006	0.22	0.22
bo	1458	1948	13.28	0.047	0.049	0.119	0.117
cg-web	2269	8131	7.34	0.160	0.160	0.506	0.503
as-rel	26242	53174	14.17	20	20	52	52
hep-th	27400	352021	10.14	76	75	251	225
flickr	820878	6625280	-	341	343	824	854

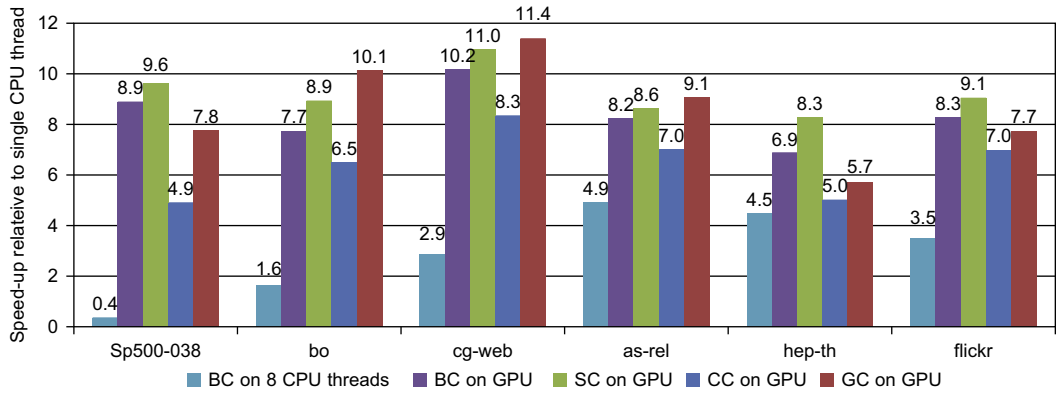

FIGURE 2.5

Speed-ups for computing centrality metrics on select data sets using mutithreaded CPU and GTX 480 GPU, relative to the running time of single CPU thread.

also accelerates SC, CC, and GC, from 4.9 times to 11.4 times, all relative to a single-threaded CPU implementation.

Our edge-parallel centrality algorithm can be modified to compute breadth-first search (BFS). Because BFS propagates a single wavefront, we parallelize a single propagation step using all threads in the grid and synchronize them by relaunching the kernel. Table 2.4 shows "Edge ‖" outperforms a single-threaded "CPU" by 1.8 times to 6.1 times (except on the first small graph) and "Node ‖" by 2.2 times to 7.5 times. To illustrate the relative importance of coalesced access pattern vs. little thread divergence in our edge-parallel approach, we randomly permute the thread-edge mapping in an implementation called "Edge ‖-U." Here, threads still enjoy little divergence, but memory access is totally uncoalesced. As shown in Table 2.4, it performs worse than "Edge ‖," illustrating the impact of *disabling* coalesced accesses. The performance degrades beyond "Node ‖," which has thread divergence because of the larger number of accesses owing to edge parallelism and uncoalesced loads. However, the fact that "Edge ‖-U" is faster than "Node ‖" on "as-rel" clearly illustrates the importance of having less thread divergence.

We have used edge-parallel BC to accelerate two graph processing tools for social network visualization: simplification [4] and clustering [5], as demonstrated on a stock price affinity network in Figure 2.6. Graph simplification removes low-BC edges, retaining the highest-BC edges that likely represent communication pathways in a network. Graph clustering removes high-BC edges to discover low-BC communities, then merges them to discover inter-community affinities. Results in Figure 2.7

Table 2.4 BFS Runtime Performance on a GTX 480, in Milliseconds

	CPU	Node ‖	Edge ‖	Edge ‖-U
cg-web	0.099	0.531	0.225	0.854
as-rel	0.891	3.670	0.489	1.644
hep-th	2.923	3.763	0.753	4.519
flickr	72.397	26.659	11.926	134.809

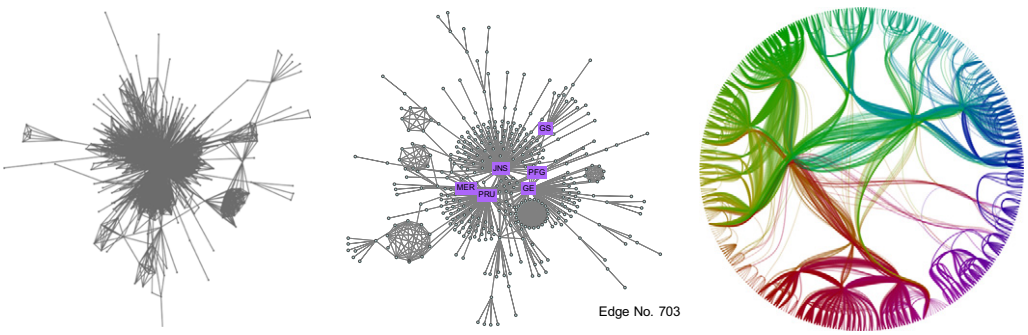

FIGURE 2.6

Input graph S&P 500-038 (left), simplified (center) and hierarchically clustered as edge bundles (right) using fast GPU centrality.

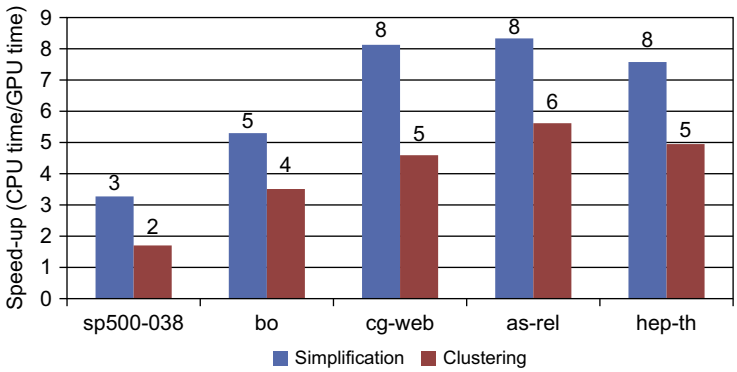

FIGURE 2.7

Speed-up (edge-parallel GPU v. serial CPU) of applications on a GTX 480.

indicates 2 to 8 times performance improvement for both tools by using edge-parallel BC, when compared to a serial CPU implementation.

2.8 CONCLUSIONS

For some graph algorithms such as computing centrality, breadth-first search, and even all-pairs shortest path, an edge-parallel approach improves GPU throughput with better load balancing and less thread divergence on scale-free networks. The edge-parallel approach is less appropriate for grids, meshes, and other graphs with low-degree variance, and it performs less well on dense graphs with many edges. The edge-parallel approach requires more GPU memory than node-parallel, which could limit its application to larger graphs than the ones we considered, for which one could investigate efficient inter-block synchronization techniques [14].

We believe the edge-parallel approach would benefit most scale-free network applications and should be investigated on further graph algorithms such as max flow. We did not consider directed or weighted graphs, which might use Dijkstra's algorithm for non-negative weights and the Bellman-Ford algorithm to handle negative weights.

This chapter is the result of several projects supported in part by the National Science Foundation under grant IIS-0534485, Intel, and the Universal Parallel Computing Research Center at the University of Illinois, sponsored by Intel and Microsoft.

References

[1] A. Sriram, K. Gautham, K. Kothapalli, P.J. Narayan, R. Govindarajulu, Evaluating centrality metrics in real-world networks on GPU, in: Proceedings of the International Conference High Performance Computing, Student Research Symposium, Kochi, India, 2009.

[2] U. Brandes, A faster algorithm for betweenness centrality, J. Math. Sociol. 25 (2001) 163–177.

[3] R. Jacob, D. Koschutzki, K.A. Lehmann, L. Peeters, D. Tenfelse-Podehl, Algorithms for centrality indices, in: U. Brandes, T. Erlebach (Eds.), Network Analysis, Springer, Berlin, 2005, pp. 62–82.
[4] Y. Jia, J. Hoberock, M. Garland, J. Hart, On the visualization of social and other scale-free networks, IEEE TVCG 14 (6) (2008) 1285–1292.
[5] Y. Jia, M. Garland, J. Hart, Hierarchical edge bundles for general graphs, Technical report, Univ. of Illinois Urbana Champaign, 2009.
[6] D. Bader, K. Madduri, Parallel algorithms for evaluating centrality indices in real-world networks, in: W.-C. Feng (Ed.), Proceedings of the International Conference Parallel Processing, IEEE Computer Society, Columbus, OH, 2006, pp. 539–550.
[7] K. Madduri, D. Ediger, K. Jiang, D.A. Bader, D. Chavarria-Miranda, A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets, in: Proceedings of the IPDPS, IEEE Computer Society, Rome, Italy, 2009, pp. 1–8.
[8] D. Tu, G. Tan, Characterizing betweenness centrality algorithm on multi-core architectures, in: International Symposium on Parallel and Distributed Processing with Applications, IEEE Computer Society, Chengdu, Sichuan, China, 2009, pp. 182–189.
[9] P. Harish, V. Vineet, P.J. Narayanan, Large graph algorithms for massively multithreaded architectures, Technical report, Indian Institutes of Information Technology, 2009.
[10] A.-L. Barabasi, R. Albert, Emergence of scaling in random networks, Science 286 (5439) (1999) 509–512.
[11] D.H. Kim, J.D. Noh, H. Jeong, Scale-free trees: the skeletons of complex networks, Phys. Rev. E 70 (4) (2004) 046126+.
[12] A.Y. Wu, M. Garland, J. Han, Mining scale-free networks using geodesic clustering, in: W. Kim, R. Kohavi, J. Gehrke, W. DuMouchel (Eds.), Proceedings of the KDD, ACM, Seattle, WA, 2004, pp. 719–724.
[13] V. Kindratenko, J. Enos, G. Shi, M. Showerman, G. Arnold, J. Stone, J. Phillips, W. Hwu, GPU clusters for high-performance computing, in: Proc. Workshop on Parallel Programming on Accelerator Clusters, IEEE International Conference on Cluster Computing, 2009, pp. 1–8.
[14] S. Xiao, W.-C. Feng, Inter-Block GPU communication via fast barrier synchronization, in: Proceedings of the IPDPS, 24th IEEE International Symposium on Parallel and Distributed Processing, Atlanta, GA, 2010, pp. 1–12.