Parallel Graph Algorithms

Sathish Vadhiyar

Graph Traversal

- Graph search plays an important role in analyzing large data sets
- Relationship between data objects represented in the form of graphs
- Breadth first search used in finding shortest path or sets of paths

Parallel BFS
Level-synchronized algorithm

- Proceeds level-by-level starting with the source vertex
- Level of a vertex – its graph distance from the source
- Also, called **frontier-based** algorithm
- The parallel processes process a level, synchronize at the end of the level, before moving to the next level – Bulk Synchronous Parallelism (**BSP**) model
- How to decompose the graph (vertices, edges and adjacency matrix) among processors?

Distributed BFS with 1D Partitioning

- Each vertex and edges emanating from it are owned by one processor
- 1-D partitioning of the adjacency matrix

$$\begin{bmatrix} A_1 \\ \hline A_2 \\ \hline \vdots \\ \hline A_P \end{bmatrix}$$

- Edges emanating from vertex v is its edge list = list of vertex indices in row v of adjacency matrix A

1-D Partitioning

- At each level, each processor owns a set F – set of frontier vertices owned by the processor
- Edge lists of vertices in F are merged to form a set of neighboring vertices, N
- Some vertices of N owned by the same processor, while others owned by other processors
- Messages are sent to those processors to add these vertices to their frontier set for the next level

Algorithm 1 Distributed Breadth-First Expansion with 1D Partitioning

1: Initialize $L_{v_s}(v) = \begin{cases} 0, & v = v_s, \text{ where } v_s \text{ is a source} \\ \infty, & \text{otherwise} \end{cases}$
2: **for** $l = 0$ to ∞ **do**
3: $F \leftarrow \{v \mid L_{v_s}(v) = l\}$, the set of local vertices with level l
4: **if** $F = \emptyset$ for all processors **then**
5: Terminate main loop
6: **end if**
7: $N \leftarrow$ {neighbors of vertices in F (not necessarily local)}
8: **for all** processors q **do**
9: $N_q \leftarrow$ {vertices in N owned by processor q}
10: **Send** N_q to processor q
11: **Receive** $\bar{N}_q$ from processor q
12: **end for**
13: $\bar{N} \leftarrow \bigcup_q \bar{N}_q$ (The $\bar{N}_q$ may overlap)
14: **for** $v \in \bar{N}$ and $L_{v_s}(v) = \infty$ **do**
15: $L_{v_s}(v) \leftarrow l + 1$
16: **end for**
17: **end for**

$L_{vs}(v)$ – level of v, i.e, graph distance from source vs

BFS on GPUs

```
1  bfs_kernel(int curLevel){
2  v = blockIdx.x * blockDim.x + threadIdx.x;
3  if dist[v] == curLevel then
4  |    forall the n ∈ neighbors(v) do
5  |    |    if visited[n] == 0 then
6  |    |    |    dist[n] = dist[v] + 1;
7  |    |    |    visited[n] = 1;
8  |    |    end
9  |    end
10 end
11 }
```

BFS on GPUs

- One GPU thread for a vertex
- For each level, a GPU kernel is launched with the number of threads equal to the number of vertices in the graph
- Only those vertices whose assigned vertices are frontiers will become active
- Do we need atomics?
- Severe load imbalance among the treads
- Scope for improvement

- MST (Prim’s), SSSP, APSP

Single Source Shortest Path (SSSP)

- Find the shortest distance from a source s to all vertices
- Dijkstra’s algorithm

Single Source Shortest Path (SSSP)

```
input : a graph graph(V,E) with N vertices in a set V, and M edges in a set, E, and M
        weights, w of the edges. A source s
1  forall the v ∈ V do
2  |   dist[v] ←= ∞ ;
3  end
4  dist[s] ← 0 ;
5  Initialize a priority queue, Q, with all the vertices ordered by the distances ;
6  while Q ≠ ∅ do
7  |   Remove u with minimum distance from Q ;
8  |   forall the v ∈ neighbors(u) do
9  |   |   if (dist[u] + w(u,v)) < dist[v] then
10 |   |   |   dist[v] = dist[u] + w(u,v) ;
11 |   |   |   Update position of v in Q ;
12 |   |   end
13 |   end
14 end
```

Fig. 4.21: Sequential Dijkstra's SSSP Algorithm

Single Source Shortest Path (SSSP)

- The operation of updating the distances of neighbors using the minimum distance of a vertex – *relaxation*
- Parallelization
  - Vertices distributed across processors
  - Each processor owns a set of vertices and their outgoing edges
  - Priority queue distributed – each processor updates only its vertices in the priority queue

Parallel SSSP Steps

- In each iteration:
- Minimum of all priority queues found using reduction
- Processor with the lowest rank removes vertex with the minimum distance

Parallel SSSP Steps

- Performs distributed edge relaxations
  - Processor communicates distance of u to processors that owns u's neighbors
  - Processors update the tentative distances of neighbors and update their positions in local priority queue
- Disadvantages?

Parallel SSSP

- It is important to parallelize outer loop
- Some heuristics have been proposed:
- All vertices with distances < threshold L can be removed
- A large L can promote parallelism but can result in poor work efficiency due to unwanted computations due to reinsertions and repeated relaxations

Parallel Dijkstra’s SSSP

- Parallelism depends on graph topology
- Number of vertices that can be removed and processed in parallel
- Number of edges that can be relaxed in parallel

Bellman-Ford

- Larger parallelism, low work efficiency
- All edges are relaxed in all iterations till convergence

input : a graph $graph(V,E)$ with N vertices in a set V, and M edges in a set, E, and M weights, w of the edges. A source s

```
1  forall the v ∈ V do
2      dist[v] ←= ∞ ;
3  end
4  dist[s] ← 0 ;
5  for (i = 0; i < N; i++) do
6      forall the (u,v) edge ∈ E do
7          if (dist[u] + w(u,v)) < dist[v] then
8              dist[v] = dist[u] + w(u,v) ;
9          end
10     end
11 end
```

Fig. 4.22: Sequential Bellman-Ford SSSP Algorithm

- Suitable for GPUs; Inner loop needs to

SSSP: Delta-stepping

- Balance between the two
- Balances work efficiency and parallelism
- Maintains tentative distances in *buckets*
- Each bucket maintains a range of tentative distances
- Range is given by delta

Delta stepping

- Assign source vertex to B0; all vertices to Binf
- Outer loop of phases; inner loop of steps
- In each phase, algorithms considers the non-empty bucket of lowest index
- Let Bj be such a bucket
- At the beginning of the phase, all vertices with final distances less than (delta.j+1) would have been settled

Delta stepping

- Algorithm removes vertices from Bj and relaxes all its outgoing edges
- This can migrate vertices from higher indexed bucket to a lower index bucket, k, with k>=j
- Perform until Bj becomes empty
- What happens if delta=1, and delta=inf?

Delta stepping

input : a graph $graph(V,E)$ with N vertices in a set V, and M edges in a set, E, and M weights, w of the edges. A source s

1 **forall the** $v \in V$ **do**
2 $\quad dist[v] \leftarrow = \infty$;
3 **end**
4 $dist[s] \leftarrow 0$;
5 $B_0 = s$;
6 $B_\infty = v \in V, v \neq s$;

Delta stepping

```
7  while ∃ a non-empty bucket do
       /* For each phase                                        */
8      Find the non-empty bucket, B_j, that is not empty ;
9      while B_j ≠ ∅ do
           /* For each step                                     */
10         forall the u ∈ B_j do
11             forall the v ∈ neighbors(u) do
                   /* Relax (u,v) edge                          */
12                 ;
13                 if (dist[u] + w(u,v)) < dist[v] then
14                     oldDist = dist[v] ;
15                     dist[v] = dist[u] + w(u,v) ;
16                     newDist = dist[v] ;
17                     l = (oldDist − 1)/Δ ;
                       /* old bucket index                      */
18                     k = (newDist − 1)/Δ ;
                       /* new bucket index                      */
19                     Migrate v from B_l to B_k ;
20                 end
21             end
22         end
23     end
24 end
```

Fig. 4.23: Δ-stepping SSSP Algorithm

Delta-stepping Parallelization

- Inner loop is parallelized where all relaxations in a bucket are parallelized
- Shared memory parallelism – updates of the distances will have to be protected by atomics

Delta-stepping Parallelization

- Distributed memory parallelism
  - Vertices distributed
  - Non-empty buckets maintained by all processors
  - Each processor stores and processes only its vertices in its buckets
  - Allreduce for finding the lowest index bucket
  - Simultaneous relaxations using BSP (Bulk synchronous parallelism) model

SSSP on GPUs

- Most follow Bellman-Ford: Large number of edges processes by threads
- Two models:
- Topology-driven: All vertices with non-infinite distances are processed by corresponding threads
- Data-driven: Only those whose distances have changed in the previous iteration are processed. A work list is maintained.

Topology-driven Algorithm

```
1  main(){
2  forall the v ∈ V do
3  |   dist[v] ←= ∞ ;
4  end
5  dist[s] ← 0 ;
6  change = 1 ;
7  while change do
8  |   change = 0 ;
9  |   sssp_kernel(change) ;
10 end
11 }

12 sssp_kernel(INOUT change){
13 forall u ∈ V parallel
14 |   if u ≠ ∞ then
15 |   |   for v ∈ neighbors(u) do
16 |   |   |   if (dist[u] + w(u,v)) < dist[v] then
17 |   |   |   |   dist[v] = dist[u] + w(u,v) ;
18 |   |   |   |   change = 1 ;
19 |   |   |   end
20 |   |   end
21 |   end
22 }
```

Fig. 4.24: Topology-driven SSSP Algorithm

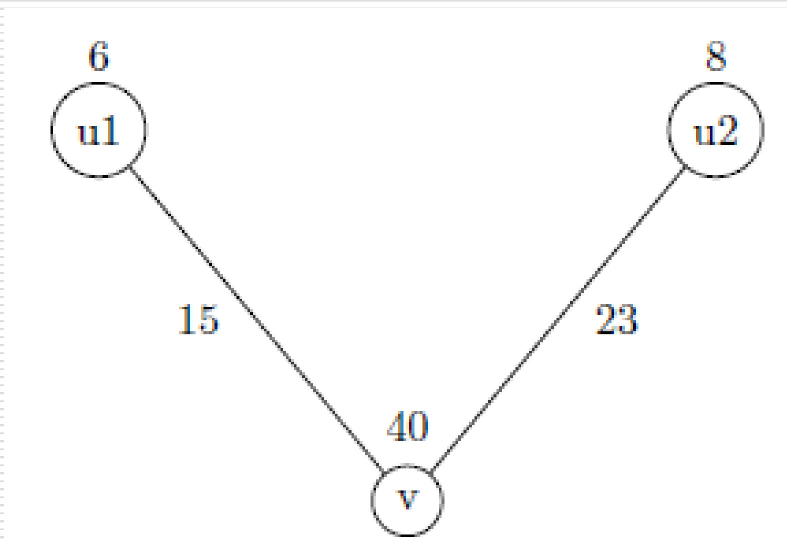
Data-driven Algorithm

```
1  main(){
2  forall the v ∈ V do
3  |    dist[v] ←= ∞ ;
4  end
5  dist[s] ← 0 ;
6  Add s to worklist_in ;
7  worklist_out = ∅ ;
8  while worklist_in ≠ ∅ do
9  |    sssp_kernel(worklist_in, worklist_out) ;
10 |    copy worklist_out to worklist_in ;
11 |    worklist_out = ∅ ;
12 end
13 }
14 sssp_kernel(IN worklist_in, OUT worklist_out){
15 for u ∈ worklist_in parallel
16 |    for v ∈ neighbors(u) do
17 |    |    if (dist[u] + w(u,v)) < dist[v] then
18 |    |    |    atomicMin(dist[v], dist[u] + w(u,v)) ;
19 |    |    |    Add v to worklist_out ;
20 |    |    end
21 |    end
22 }
```

Fig. 4.25: Data-driven SSSP Algorithm

Pros and Cons

- Topology-driven: Low work efficiency
- Data-driven: Need atomics, atomicMin



- Can result in lost updates of minimum distances

Pros and Cons

- Topology-driven version does not need atomics
- This is because SSSP has *monotonicity* property: the distance value of a vertex is non-increasing
- Property utilized in topology-driven version to avoid atomics

Pros and Cons

- Since all active vertices with non-infinite distances are processed in all iterations, the lost updates will be reconsidered in the subsequent iterations.
- Even if there is a lost update, the thread with the minimum distance will get its chance in the next iteration

Reducing Atomics in Atomic Addition to Worklist

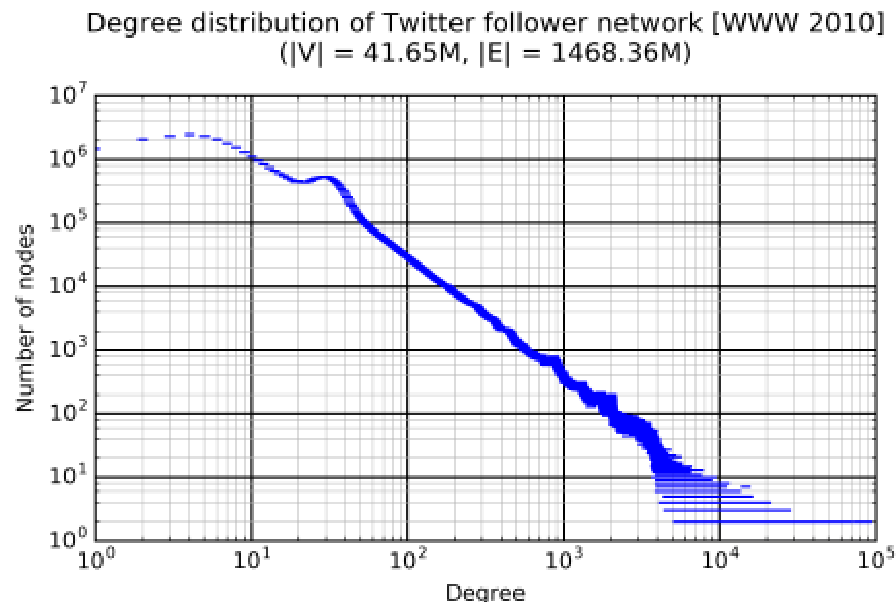
- *Work chunking*: Perform atomic updates for a chunk of elements rather than for every element
- Prefix sums can be used to avoid atomics
- Prefix sums themselves can be hierarchically constructed: *hierarchical prefix sum*

Redundancy in Worklists

- Two threads can add the same neighbour vertex to a worklist
- Duplicates can be huge!
- Two ways to avoid:
  - A post-processing *filtering* kernel
  - *Hash-based culling*; a thread uses hashing to find if its neighbour has been added

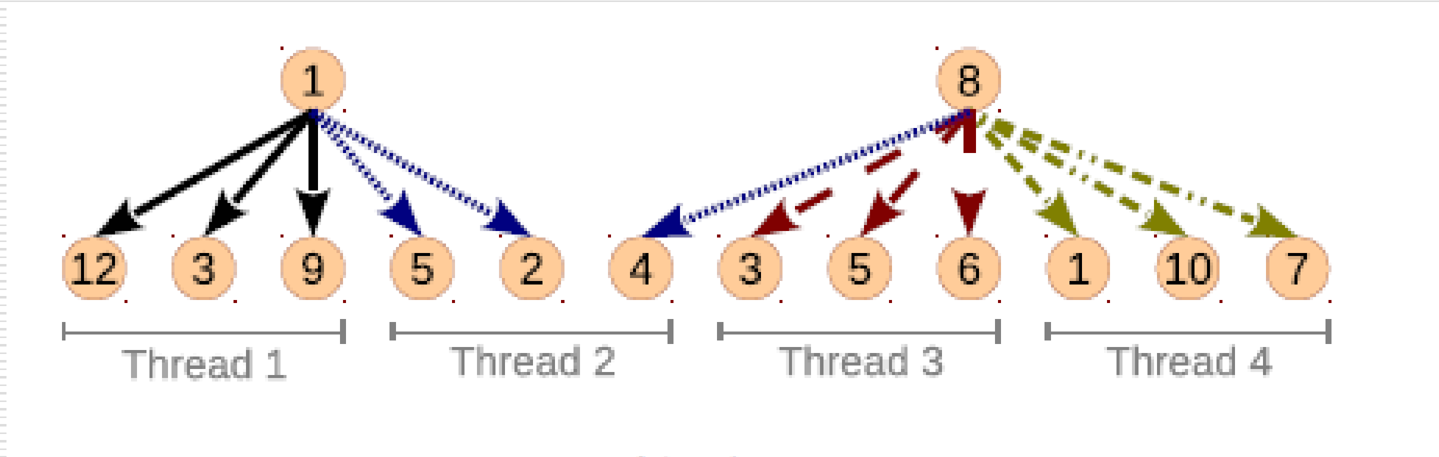
Load Balancing

- A major challenge among graph processing algorithms is the load imbalance among thread
- Modern-day graphs in social networks are scale-free graphs
- These graphs follow power-law distribution of degrees



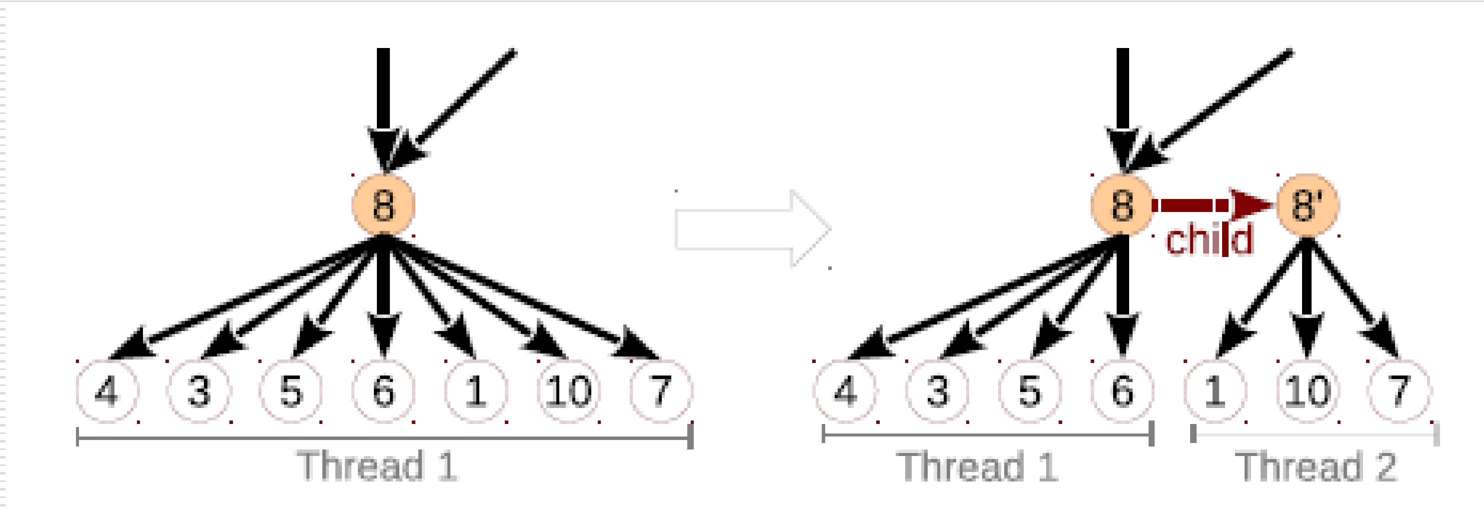
Load Balancing Strategies: Workload Decomposition

- Edges decomposed across threads in a block distribution



Load Balancing Strategies: Node Splitting

- Split each high degree node into multiple low-degree nodes called virtual nodes



- Prim’s Minimal Spanning Tree, APSP using Dijkstra’s
- (Book by Grama et al. – Pages starting from 432, then from 438)

Minimal Spanning Tree – Prim’s Algorithm

- Spanning tree of a graph, G (V,E) – tree containing all vertices of G
- MST – spanning tree with minimum sum of weights
- Follows similar structure as Dijkstra’s SSSP.
- Vertices are added to a set Vt that holds vertices of MST; Initially contains an arbitrary vertex,r, as root vertex

Minimal Spanning Tree – Prim's Algorithm

- An array d such that d[v in (V-Vt)] holds weight of the edge with least weight between v and any vertex in Vt; Initially d[v] = w[r,v]
- Find the vertex in d with minimum weight and add to Vt
- Update d
- Time complexity – $O(n^2)$

Parallelization

- Vertex V and d array partitioned across P processors
- Each processor finds local minimum in d
- Then global minimum across all d performed by reduction on a processor
- The processor finds the next vertex u, and broadcasts to all processors

Parallelization

- All processors update d; The owning processor of u marks u as belonging to Vt
- Process responsible for v must know w[u,v] to update d[v]; 1-D block mapping of adjacency matrix
- Complexity – $O(n^2/P)$ + (OnlogP) for communication

All-Pairs Shortest Paths

- To find shortest paths between all pairs of vertices
- Dijikstra's algorithm for single-source shortest path can be used for all vertices
- Two approaches

All-Pairs Shortest Paths

- Source-partitioned formulation: Partition the vertices across processors
  - Works well if $p<=n$; No communication
  - Can at best use only n processors
  - Time complexity?
- Source-parallel formulation: Parallelize SSSP for a vertex across a subset of processors
- Do for all vertices with different subsets of processors
- Hierarchical formulation
- Exploits more parallelism
- Time complexity?

Graph Partitioning

Graph Partitioning

- For many parallel graph algorithms, the graph has to be partitioned into multiple partitions and each processor takes care of a partition
- Criteria:
  - The partitions must be balanced (uniform computations)
  - The edge cuts between partitions must be minimal (minimizing communications)
- Some methods
  - BFS: Find BFS and descend down the tree until the cumulative number of nodes = desired partition size
  - Mostly: Multi-level partitioning based on coarsening and refinement (a bit advanced)
  - Another popular method: Kernighan-Lin

Partitioning without nodal coordinates - Kernighan/Lin

- Take a initial partition and iteratively improve it
  - Kernighan/Lin (1970), cost = $O(|N|^3)$ but easy to understand
  - Fiduccia/Mattheyses (1982), cost = $O(|E|)$, much better, but more complicated
- Let $G = (N,E,W_E)$ be partitioned as $N = A \cup B$, where $|A| = |B|$
- $T = cost(A,B) = \Sigma$ {W(e) where e connects nodes in A and B}
- Find subsets X of A and Y of B with $|X| = |Y|$ so that swapping X and Y decreases cost:
  - $newA = A - X \cup Y$ and $newB = B - Y \cup X$
  - $newT = cost(newA , newB) < cost(A,B)$
  - Keep choosing X and Y until cost no longer decreases
- Need to compute newT efficiently for many possible X and Y, choose smallest

Kernighan/Lin - Preliminary Definitions

- $T = cost(A, B)$, $newT = cost(newA, newB)$
- Need an efficient formula for newT; will use
  - E(a) = external cost of a in A = Σ {W(a,b) for b in B}
  - I(a) = internal cost of a in A = Σ {W(a,a') for other a' in A}
  - D(a) = cost of a in A = E(a) - I(a)
    - Moving a from A to B would decrease T by D(a)
  - E(b), I(b) and D(b) defined analogously for b in B
- Consider swapping X = {a} and Y = {b}
  - newA = A - {a} U {b}, newB = B - {b} U {a}
- newT = T - (D(a) + D(b) - 2*w(a,b)) = T - gain(a,b)
  - gain(a,b) measures improvement gotten by swapping a and b
- Update formulas, after a and b are swapped
  - newD(a') = D(a') + 2*w(a',a) - 2*w(a',b) for a' in A, a' != a
  - newD(b') = D(b') + 2*w(b',b) - 2*w(b',a) for b' in B, b' != b

Kernighan/Lin Algorithm

```
Compute  T = cost(A,B) for initial A, B                         ... cost = O(|N|^2)
Repeat
    ... One pass greedily computes |N|/2 possible X,Y to swap, picks best
    Compute costs D(n) for all n in N                            ... cost = O(|N|^2)
    Unmark all nodes in N                                        ... cost = O(|N|)
    While there are unmarked nodes                               ... |N|/2 iterations
       Find an unmarked pair (a,b) maximizing gain(a,b)            ... cost = O(|N|^2)
       Mark a and b (but do not swap them)                         ... cost = O(1)
       Update D(n) for all unmarked n,
            as though a and b had been swapped                   ... cost = O(|N|)
    Endwhile
       ... At this point we have computed a sequence of pairs
       ...  (a1,b1), ... , (ak,bk)   and gains gain(1),...., gain(k)
       ... where k = |N|/2, numbered in the order in which we marked them
    Pick m maximizing Gain = Σk=1 to m   gain(k)                    ... cost = O(|N|)
       ... Gain is reduction in cost from swapping (a1,b1) through (am,bm)
    If Gain > 0 then    ... it is worth swapping
       Update newA = A - { a1,...,am } U { b1,...,bm }           ... cost = O(|N|)
       Update newB = B - { b1,...,bm } U { a1,...,am }           ... cost = O(|N|)
       Update T = T - Gain                                       ... cost = O(1)
    endif
 Until Gain <= 0
```

Parallel partitioning

- Can use divide and conquer strategy
- A master node creates two partitions
- Keeps one for itself and gives the other partition to another processor
- Further partitioning by the two processors and so on…

- Multi-level partitioning

K-way multilevel partitioning algorithm

- Has 3 phases: coarsening, partitioning, refinement (uncoarsening)
- Coarsening - a sequence of smaller graphs constructed out of an input graph by collapsing vertices together

Coarsening

- Formulated as a maximal matching problem
- Matching – finding a set of non-adjacent edges, i.e., edges are not incident on same vertices
- Maximal matching: A matching where addition of one more edge results in the loss of matching property
- Commonly used heuristic: heaviest edge matching

K-way multilevel partitioning algorithm

- When enough vertices are collapsed together so that the coarsest graph is sufficiently small, a k-way partition is found
- Finally, the partition of the coarsest graph is projected back to the original graph by refining it at each uncoarsening level using a k-way partitioning refinement algorithm

K-way partitioning refinement

- A simple randomized algorithm that moves vertices among the partitions to minimize edge-cut and improve balance
- For a vertex v, let neighborhood N(v) be the union of the partitions to which the vertices adjacent to v belong
- In a k-way refinement algorithm, vertices are visited randomly

K-way partitioning refinement

- A vertex v is moved to one of the neighboring partitions N(v) if any of the following vertex migration criteria is satisfied
  - The edge-cut is reduced while maintaining the balance
  - The balance improves while maintaining the edge-cut
- This process is repeated until no further reduction in edge-cut is obtained

Graph Coloring

Graph Coloring Problem

- Given G(A) = (V, E)
- $\sigma: V \longrightarrow \{1,2,\ldots,s\}$ is s-coloring of G if $\sigma(i) \neq \sigma(j)$ for every (i, j) edge in E
- Minimum possible value of s is *chromatic number* of G
- Graph coloring problem is to color nodes with chromatic number of colors
- NP-complete problem

Parallel graph Coloring – General algorithm

$ParallelColoring(G = (V, E))$
begin
  $U \leftarrow V$
  $G' \leftarrow G$
  while (G' is not empty) do in parallel
    Find an independent set I in G'
    Color the vertices in I
    $U \leftarrow U \setminus I$
    $G' \leftarrow$ graph induced by U
  end-while
end

Parallel Graph Coloring – Finding Maximal Independent Sets – Luby (1986)

I = null
V' = V
G' = G
While G' ≠ empty
  Choose an independent set I' in G'
  I = I U I';   X = I' U N(I')   (N(I') – adjacent vertices to I')
  V' = V' \ X; G' = G(V')
end

For choosing independent set I': (Monte Carlo Heuristic)

1. For each vertex, v in V' determine a distinct random number p(v)
2. v in I iff p(v) > p(w) for every w in adj(v)

Color each MIS a different color

Disadvantage:

- Each new choice of random numbers requires a global synchronization of the processors.

Parallel Graph Coloring – Gebremedhin and Manne (2003)

$BlockPartitionBasedColoring(G, p)$

begin

1. Partition V into p equal blocks $V_1 \ldots V_p$, where $\lfloor \frac{n}{p} \rfloor \leq |V_i| \leq \lceil \frac{n}{p} \rceil$

 for $i = 1$ to p do in parallel

  for each $v_j \in V_i$ do

   assign the smallest legal color to vertex v_j

   barrier synchronize

  end-for

 end-for

Pseudo-Coloring

Sources/References

- Paper: A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. Yoo et al. SC 2005.
- Paper:Accelerating large graph algorithms on the GPU usingCUDA. Harish and Narayanan. HiPC 2007.
- M. Luby. A simple parallel algorithm for the maximal independent set problem. SIAM Journal on Computing. 15(4)1036-1054 (1986)
- A.H. Gebremedhin, F. Manne, Scalable parallel graph coloring algorithms, Concurrency: Practice and Experience 12 (2000) 1131-1146.

Community Detection

- Given a graph, the goal is to partition into communities such that related vertices are assigned to the same community

Metric

- Modularity – Measure to evaluate the goodness of a community
- Measures the fraction of edges that lie within the community
- Measures the difference between fraction of edges within communities compared to the expected fraction that would exist on a random graph with identical vertex and degree distributions

$$Q = \frac{1}{2m} \sum_{i,j} (A_{ij} - \frac{k_i * k_j}{2m}) \delta(c_i, c_j)$$

where:

$$m = \text{sum of all the edge-weights} \quad (1)$$

$$k_i = \text{weighted degree of vertex } i$$

$$c_i = \text{community that contains vertex } i$$

$$\delta(c_i, c_j) = 1 \text{ if } c_i = c_j, 0 \text{ otherwise.}$$

Modularity

- Or

$$Q = \sum_{c \in C} \left[\frac{e_{ij}}{2m} - \left(\frac{a_c}{2m} \right)^2 \right]$$

where:

$$e_{ij} = \sum w_{ij} : \forall i, j \in c, \text{ and } \{i, j\} \in E \quad (2)$$

$$a_c = \sum_{i \in c} k_i$$

Louvain Method

- Multi-phase, multi-iteration heuristic
- Iteratively improves the quality of the community until the gain in quality becomes negligible
- Complete sweep of a graph per iteration
- Graph coarsenings between phases

Louvain Method

- Each phase runs for a number of iterations until convergence
- Initially, each vertex is a community
- In each iteration:
  - Gain in modularity calculated when moving a vertex to each of its neighboring communities
  - If positive gain moved
- Iterations continued until convergence
- At the end of the phase, the vertices are collapsed

Sequential Algorithm

Algorithm 1: Serial Louvain algorithm.

Input: Graph $G = (V, E)$, threshold τ

Input: Initial community assignment, C_{init}

1: $Q_{prev} \leftarrow -\infty$
2: $C_{prev} \leftarrow$ Initialize each vertex in its own community
3: **while** true **do**
4:     **for all** $v \in V$ **do**
5:         $N(v) \leftarrow$ neighboring communities of v
6:         $targetComm \leftarrow \arg\max_{t \in N_v} \Delta Q(v \text{ moving to } t)$
7:         **if** the gain is positive **then**
8:             Move v to $targetComm$ and update C_{curr}
9:     $Q_{curr} \leftarrow ComputeModularity(V, E, C_{curr})$
10:     **if** $Q_{curr} - Q_{prev} \leq \tau$ **then**
11:         break
12:     **else**
13:         $Q_{prev} \leftarrow Q_{curr}$

Challenges in Parallel Algorithm

- Lag of Community updates
- Significant communication overhead at every iteration of every phase
- Modularity calculation requires global accumulation of weights, hence global collectives
- New vertex-community mapping must be communicated at the end of every phase

Parallel Louvain Algorithm

```
1:  function LouvainIteration(G_i, C_curr)
2:  V_g ← ExchangeGhostVertices(G_i)
3:  while true do
4:     send latest information on those local vertices that are
    stored as ghost vertices on remote processes
5:     receive latest information on all ghost vertices
6:     for v ∈ V_i do
7:        Compute ΔQ that can be achieved by moving v to each
    of its neighboring communities
8:        Determine target community for v based on the migration
    that maximizes ΔQ
9:        Update community information for both the source and
    destination communities of v
10:    send updated information on ghost communities to owner
    processes
11:    C_info ← receive and update information on local
    communities
12:    currMod_i ← Compute modularity based on G_i and C_info
13:    currMod ← all-reduce: Σ_∀i currMod_i
14:    if currMod − prevMod ≤ τ then
15:       break
16:    prevMod ← currMod
17: return prevMod
```

Optimizations

- One of the major contributors of communication is the communication of ghost vertex information
- *Observation*: Rate of modularity increase decreases with the number of iterations – diminishing benefits
- This fact can be used to drop out certain vertices from computations and communications
- Mark vertices as active and inactive probabilistically

Optimizations

- If the vertex has not moved recently, the probability that it will stay active is reduced
- e.g.:

$$P_{v,k} = \begin{cases} P_{v,k-1} * (1-\alpha), & \text{if } C_{v,k-1} = C_{v,k-2} \\ 1, & \text{otherwise} \end{cases} \quad (3)$$

Optimizations within a node

- Within a node, concurrent updates need locking
- Can identify non-colliding vertices and update them concurrently without locks?
- How?

- Paper: Distributed Louvain Algorithm for Graph Community Detection