Parallel Graph Algorithms

What is a graph?

$G = (V, E)$ $\quad E : V \times V$

Weighted graphs

$G = (V, E, W)$ $\quad W : E \to R$

Directed : E is ordered

Undirected : E is unordered

Very large graphs — Stanford Large Network Dataset Collection

Memory can be a bottleneck / compute can be a bottleneck

So both OpenMP parallelization for shared memory and MPI parallelization for distributed memory are important for graph algorithms.

GPU acceleration is hard for graph algorithms as typically graphs are sparse. Adjacency matrix for these large graphs are typically very sparse. Sparse matrix operation are not very optimal for GPU. Dense matrix operation work better — caches can be optimized for a block of contiguous access on CPU, on GPU the sparse access between DRAM and GPU memory is even worse.

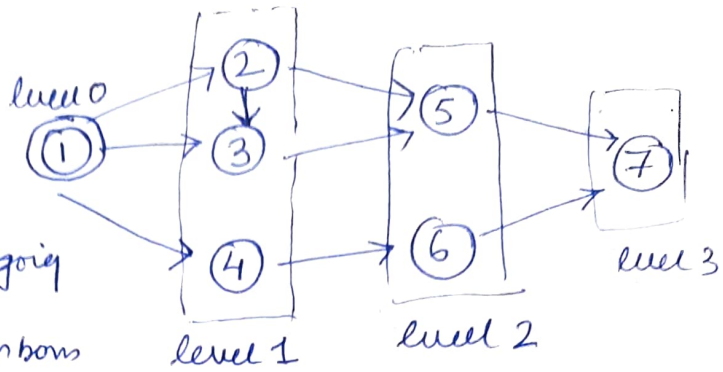
Let's take BFS. We have divided the graph nodes across processors.

$v_1 \ldots v_{100}$	$v_{101} \ldots v_{200}$

the neighbors of a node on a processor might not be on the same processor. Thus to run BFS, communication will be needed. Communication might be much more costly than compute, and therfore the algorithms designed should also minimize communication.

Breadth First Search

Root ①



First step : visit all nodes of ① reachable with outgoing edges.

Second step : visit all neighbours of ②, ③, ④.

Very large graph → so all nodes are not fitting on one processor memory.

Frontier → all nodes which got their levels marked in current iteration

level = 0

$d = [\underset{1}{0} \;\; \underset{2}{\infty} \;\; \underset{3}{\infty} \;\; \underset{4}{\infty} \;\; \underset{5}{\infty} \;\; \underset{6}{\infty} \;\; \underset{7}{\infty}]$

distance from the root / level of the particular node in the BFS tree

$f = [\underset{1}{1} \;\; \underset{2}{0} \;\; \underset{3}{0} \;\; \underset{4}{0} \;\; \underset{5}{0} \;\; \underset{6}{0} \;\; \underset{7}{0}]$ Frontier

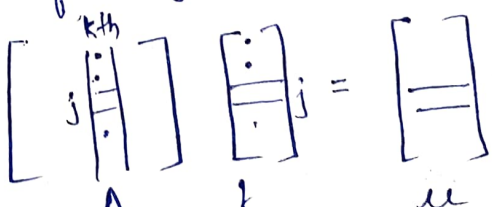
```
while (f != 0)
{
    u = A^T f
    f' = u & (d ≠ ∞)
    update d with all elements f' = 1
    set d[.] = level + 1
    level = level + 1
    f = f'
}
```

Adjacency matrix

	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	0	0	1	0	1	0	0
3	0	0	0	0	1	0	0
4	0	0	0	0	0	1	0
5	0	0	0	0	0	0	1
6	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0

$a_{ij} = 1$ if $(i, j) \in E$, $G(V, E)$

suppose a node j is in frontier i.e $f[j] = 1$, then a node k will be candidate for next frontier if $a_{jk} = 1$



$u[k] = \bigvee_j (f[j] \wedge a[j, k])$

$A^T f$ dot product → addition multiplication replaced $(\vee, \wedge)$

BFS with matrix algebra → can be parallelized.

Non matrix based algorithm for BFS.

```
Sequential_BFS ( G=(V,E), vertex src)
{   ∀ v ∈ V: level[v] = -1
    level[s] = 0;
    curr_level = 0;
    frontier = { }      // set of vertices at the current level.
    next_frontier = { } // set of vertices at the next level
    frontier.add (src);
    while (frontier is not empty)
    {   for each vertex u ∈ frontier:   ← can be executed in parallel
        for each neighbor v of u:
            if (level[v] == -1)   // not visited before.
            {   next_frontier.add (v);
                level[v] = curr_level + 1;
            }                                  → race condition.
    }
    frontier = next_frontier;   ← barrier
    next_frontier = { };
    curr_level ++;
}
```

Two threads looking at two different nodes of the frontier, but one of the neighbors of these two vertices are common. Both threads might see (level[v] == -1) and try to change it.

Parallel BFS using OpenMP

Fork join model

single thread

threads 4.

three additional threads.

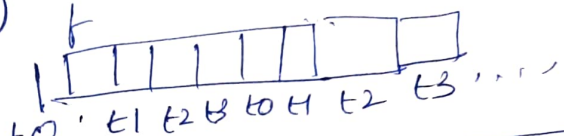
original thread

join

only thread 0 continues.

```
# pragma omp parallel
{
    tid = omp_get_thread_num();
    nthreads = omp_get_num_threads();
    # traverse the frontier together
    for (int i = tid ; i < frontier.size() ; i += nthreads)
    {
        for each neighbor v of u
            if (level[v] == -1)
            {
                # pragma omp atomic capture
                {
                    insert = level[v];
                    level[v] = curr_level + 1;
                }
                if (insert == -1)
                    local_next_frontier.add(v);
            }
    }
    # pragma omp critical
    {
        next_frontier.merge(local_next_frontier);
    }
}
```

frontier : array of vertex ids



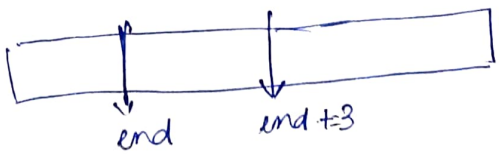
critical section (A)

```
# pragma omp critical
{ enqueue vertex v in next_frontier }
```

(B) not a global next frontier

(-) lock at the code level instruction cannot be executed be any other thread, even if that thread is working on a different piece of data. Instead we do lock on data

↑ how to efficiently do this?

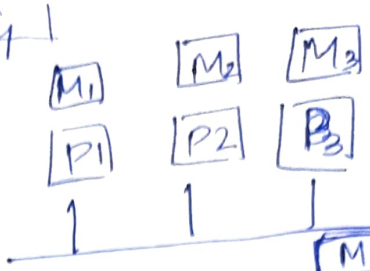


increment pointer only in critical section. A thread reserves space for its own entries. copy into these reserved space outside critical section.

BFS in distributed memory setting

Cannot access the memory of another processor.

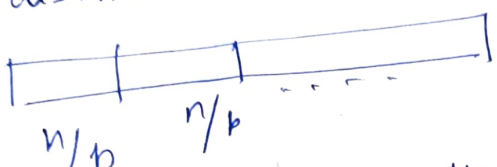
Instead use send/recv



How do we distribute the data? (graph partitioning)

1D - data distribution

Partition vertices



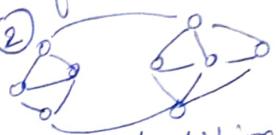
n vertices
p processors

That processor will maintain the information for that vertex.

- ⇒ adjacency list
- ⇒ any other attributes

owner(v) → rank

METIS

1. randomize IDs. then distribute for load balancing
2. 

cut/partition minimize edges across

```
Distributed_BFS_1D ( local G = (V, E), vertex s)
{
    frontier = { };  // local
    next_frontier = { }; // local
    curr_level = 0; // local
    for all v ∈ V : level[v] = -1 ; // not visited yet.
    if (owner(s) == me) // MPI_COMM_RANK used
    {
        level[s] = 0;
        frontier.add(s);
    }
    while (true) // cannot use frontier == NULL as not shared
    {
        my local frontier might be empty, but
        other processors might have nodes in
        their frontier, Those nodes might have
        neighbors, which I own— so I need
        be active in the next iteration. So a
        processor can be inactive in some iteration
```

// frontier contains vertices (local) in current frontier

for each vertex $u \in$ frontier:

for $v \in$ neighbor(u): // send these vertices to their owners.

w = owner(v)

Buffer[w].add(v); // for each processor, I maintain a buffer.

p sends, p receives

for each $p \neq$ me

{ send buffer[p] to p;

recv RecvBuff[p] from p;

}

// send buffers to respective processors.

AlltoAllv(buffer, RecvBuffer ...); // each buffer to respective ranks, and also receive from all other ranks

$\forall p = [0 \ldots numRanks - 1]$

next_frontier.merge(RecvBuf[p]); ← duplicate entries.

frontier = { };

for $v \in$ next_frontier:

if level[v] == -1

level[v] = curr_level + 1;

fronta.add(v); ← v might be neighbor of multiple vertices at current level

only if level has changed, so frontier removes duplicate entries.

next_frontier = { };

curr_level ++;

size = frontier( );

AllReduce(size); // total global frontier

if (size == 0) // all frontiers are empty

	P1	P2	P3	..	P4
	3	2	1		1
	7	7	7		7

break out of while loop

}

}