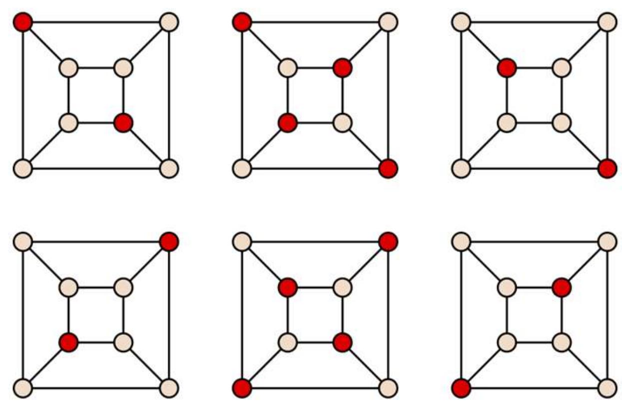
# Random-selection parallel algorithm (Luby's Algorithm) with MPI

Yu Nong

# Background

- Maximum Independent Set (MIS):
  - Independent Set:
    - a set of vertices such that for every two vertices in the set, there is no edge connecting the two.
  - MIS:
    - A set that is not a subset of any independent set.



# How to get an MIS

- Greedy Algorithm *(Time complexity: O(n))*
    - Given a Graph G(V,E):
        - Initialize I to an empty set.
        - While V is not empty:
            - Choose a node v∈V.
            - Add v to the set I.
            - Remove from V the node v and all its neighbors.
    - Return I.

# How to get an MIS

- Random-selection parallel algorithm [Luby's Algorithm]
  - Initialize I to an empty set.
  - While V is not empty:
    - Choose a random set of vertices $S \subseteq V$.
    - For every edge in E, if both its endpoints are in the random set S, then remove from S the endpoint that has fewer neighbors.
      - Break ties arbitrarily, e.g. using a lexicographic order on the vertex names.
    - Add the set S to I.
    - Remove from V the set S and all the neighbors of nodes in S.
  - Return I.

  - *Time complexity O(log n)*

# Implementation

- Storing the graph as an adjacent matrix:
  - Load from file.
- Initialize:
  - I (selected vertices)
  - V (remaining vertices to be selected)

```
struct MIS_graph
{
	int *adjacent_matrix;
	int num_nodes;
	int num_procs;
};
```

```
int *selected_nodes = malloc(sizeof(int) * num_nodes); // set I in wiki
for (int i = 0; i < num_nodes; i++)
{
	selected_nodes[i] = -1;
}
```

```
int *nodes_remain = malloc(sizeof(int) * num_nodes);  //set V in wiki
int nodes_remain_size = num_nodes;
for (int i = 0; i < num_nodes; i++)
	nodes_remain[i] = i;
```

# Implementation

- Split graph:
  - Each process randomly works on round(N/P) vertices, where the last process works on the remainder ones.
- Split adjacent matrix:
  - Each process only need to store the connected vertices information for its processed vertices.

```c
int num_nodes_per_proc = (num_nodes>=num_procs) ? round(1.0 * num_nodes / num_procs) : 1;
int num_nodes_last_proc = num_nodes - num_nodes_per_proc * (num_procs - 1);

int *count = malloc (sizeof(int) * num_procs);  // Number of nodes for current proc
int *displs = malloc (sizeof(int) * num_procs); // Buffer location for the sending proc

for (int i = 0; i < num_procs; i++)
{
	if (num_nodes >= num_procs)
	{
		if (i < num_procs - 1)
			count[i] = num_nodes_per_proc;
		else
			count[i] = num_nodes_last_proc;
		displs[i] = i * num_nodes_per_proc;
	}
	else
	{
		count[i] = (i < num_nodes) ? 1 : 0;
		if (i==0)
			displs[i] = 0;
		else
			displs[i] = displs[i-1]+1;
	}
}
```

```c
// Each process only need to process respective adjacent matrix rows
int **connected_nodes = malloc(sizeof(int*) * count[rank]);
int *num_connected_nodes = malloc(sizeof(int) * count[rank]);
int *num_connected_nodes_remain = malloc(sizeof(int) * count[rank]);
for (int i = rank * num_nodes_per_proc; i < rank * num_nodes_per_proc + count[rank]; i++)
{
	int *this_connected_nodes = malloc(sizeof(int) * num_nodes);
	int this_num_connected_nodes = 0;

	for (int j = 0; j < num_nodes; j++)
	{
		if (adjacent_matrix[i*num_nodes+j] == 1)
		{
			this_connected_nodes[this_num_connected_nodes] = j;
			this_num_connected_nodes++;
		}
	}
	connected_nodes[i - rank * num_nodes_per_proc] = this_connected_nodes;
	num_connected_nodes[i - rank * num_nodes_per_proc] = this_num_connected_nodes;
	num_connected_nodes_remain[i - rank * num_nodes_per_proc] = this_num_connected_nodes;
}
```

# Implementation

- Iterations of Luby's algorithm:
  - Each process stores copies of I (selected vertices), V (remaining vertices).
  - Each process works on its assigned vertices to choose the random set S'.
  - Synchronize the selected random set S by union all the S'.
  - Each process checks and removes the conflict vertices with fewer neighbors.
  - Synchronize the final selected vertices I and remaining set V in this iteration.

```c
while (nodes_remain_size > 0) {
    int *cur_proc_selected_nodes = malloc(sizeof(int) * count[rank]); // S for this proc
    for (int i = 0; i < count[rank]; i++)
                cur_proc_selected_nodes[i] = -1;

    for (int i = 0; i < count[rank]; i++){
        int node = nodes_this_proc[i];
        if (nodes_remain[node] == -1)   continue;
        int degree = num_connected_nodes_remain[node];
        int rand_num;
        if (degree > 0)
            rand_num = rand() % (2 * degree);
        else
            rand_num = 0;
        if (rand_num == 0)  cur_proc_selected_nodes[i] = node;
    }
    int *cur_selected_nodes = malloc(sizeof(int) * num_nodes);
    MPI_Allgatherv(cur_proc_selected_nodes, count[rank], MPI_INT, cur_selected_nodes,
        count, displs, MPI_INT, MPI_COMM_WORLD);
```

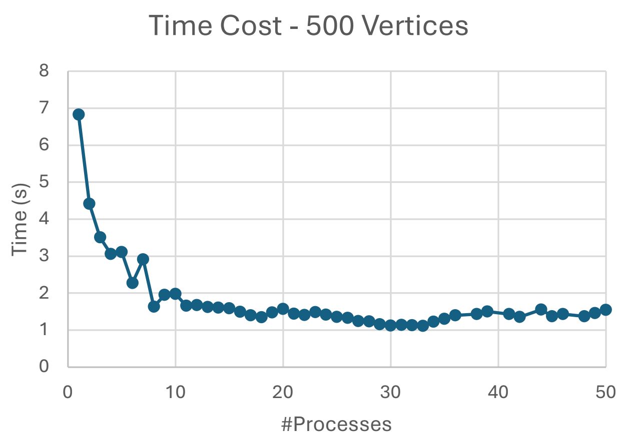
```c
for (int i = 0; i < count[rank]; i++){
        int node = cur_proc_selected_nodes[i];
        if (node == -1)
                continue;
        int degree = num_connected_nodes[node];
        //printf("Proc:%d checking node: %d\n", rank, node);

        int flag = 1;
        if (degree == 0)        selected_nodes[node] = node;
        for (int j = 0; j < degree; j++){
                if (cur_selected_nodes[connected_nodes[node][j]] >= 0){
                        if (degree < num_connected_nodes[connected_nodes[node][j]])
                                flag = 0;
                        if (degree == num_connected_nodes[connected_nodes[node][j]])
                                if (node < connected_nodes[node][j])
                                        flag = 0;
                }
        }
        if (flag)       selected_nodes[node] = node;
}
```

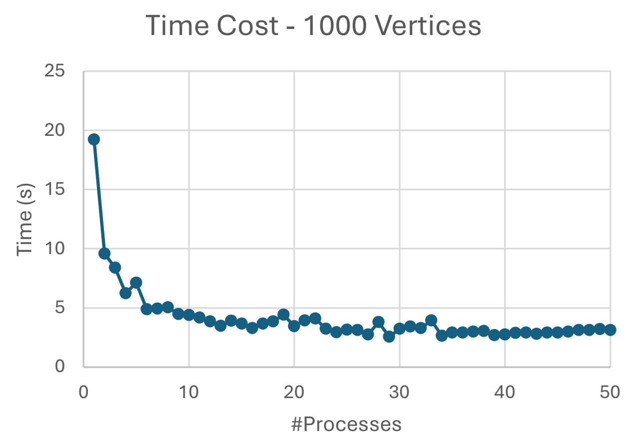
```c
for (int i = 0; i < num_nodes; i++){
    if (selected_nodes[i]>=0){
        int degree = num_connected_nodes[i];
        if (nodes_remain[i] != -1){
            nodes_remain[i] = -1;
            nodes_remain_size--;
        }
        for (int j = 0; j < degree; j++){
            int neighbor = connected_nodes[i][j];
            if (nodes_remain[neighbor] != -1){
                nodes_remain[neighbor] = -1;
                nodes_remain_size--;
            }
        }
    }
}
```

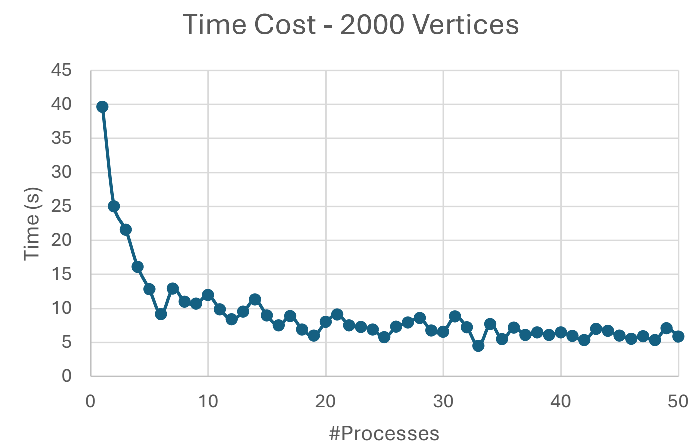
- Initialize I to an empty set.
- While V is not empty:
  - Choose a random set of vertices S ⊆ V
  - For every edge in E, if both its endpoints are in the random set S, then remove from S the endpoint that has fewer neighbors.
    - Break ties arbitrarily, e.g. using a lexicographic order on the vertex names.
  - Add the set S to I.
  - Remove from V the set S and all the neighbors of nodes in S.
- Return I.

# Results

- Given a sample with a fixed number of vertices:
  - When the #processes increase, the time cost decrease and then increase a little bit.

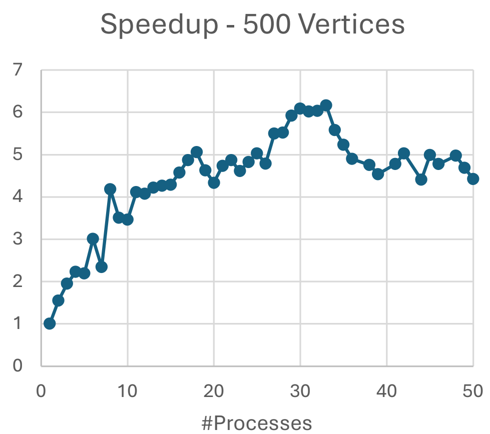


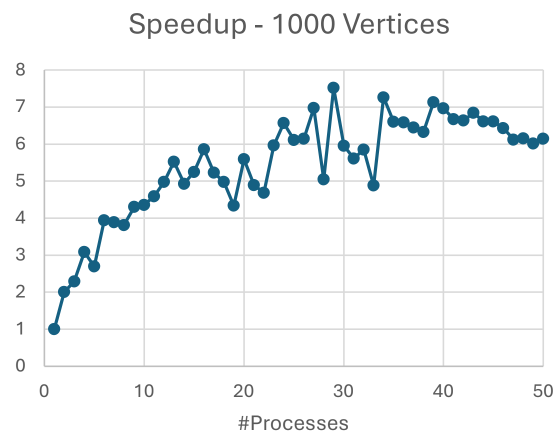



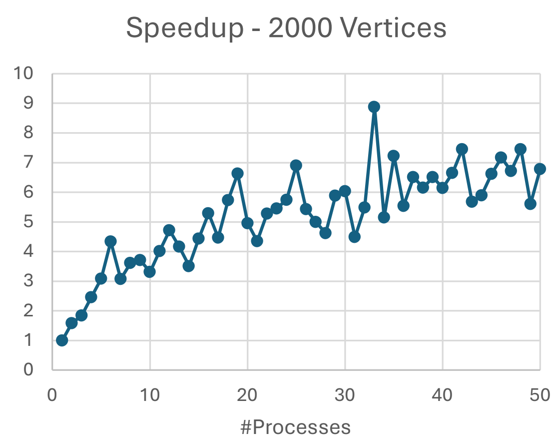



# Results

- When the #processes increase, the speedup first increase and then decrease:
  - Especially on the sample with less vertices.

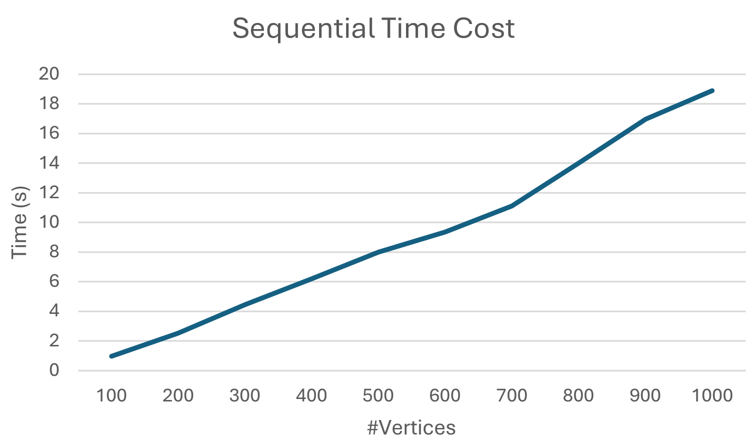


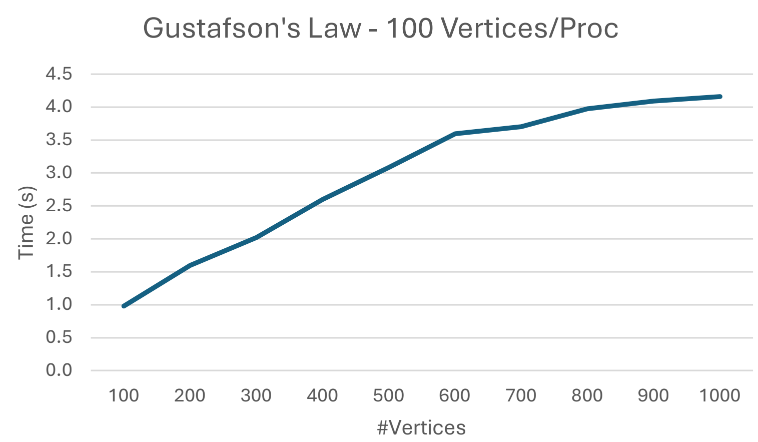



# Results

- Compared with sequential algorithm, the parallel algorithm's time cost increases much slower and tend to be logarithmic.



# Comments?