FINAL PRESENTATION

On Parallelizing Maximal Clique Enumeration (Bron-Kerbosch)

Utkarsh Kumar

13 December 2024

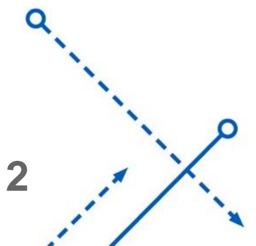
University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

Maximal Cliques: what & why?

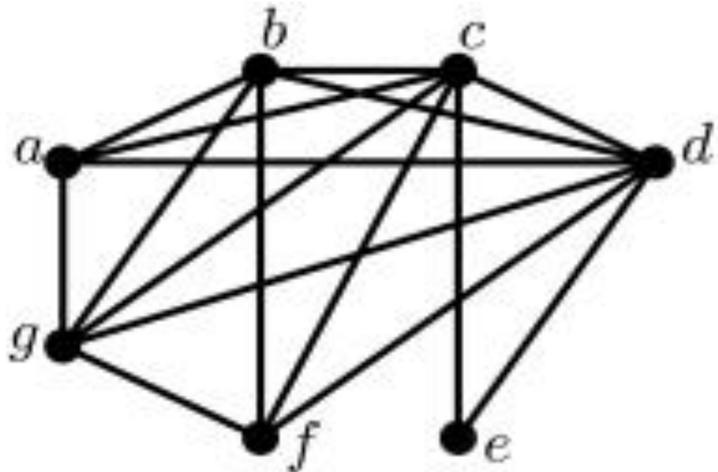
- A clique is a sub graph/graph which is complete: all nodes are connected to each other
- A maximal clique should satisfy the following:
  - The vertices are subset of given graph where the subset is complete (clique part)
  - Adding any new node will mean destroying the complete connectivity (maximal part)

- [Moon & Moser, 1965]* proved that any n-vertex graph has a maximum of 3^(n/3) maximal cliques
- Application areas include:
  - Social network analysis
  - Bio-informatics
  - Telecomm & network design
  - Fraud detection
  - Recommender system
  - Scientific collaborations

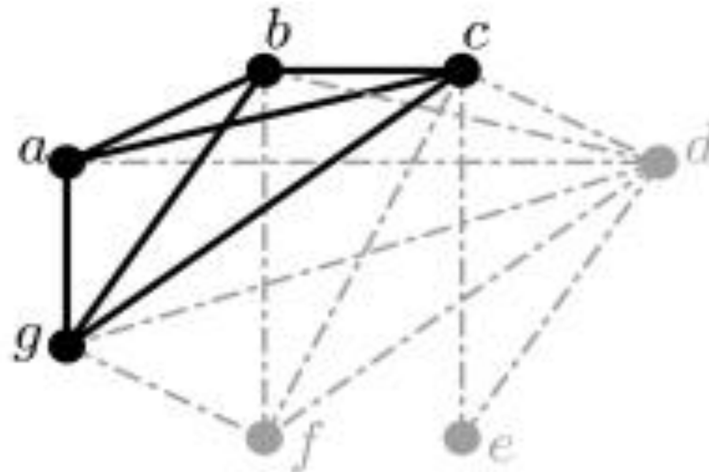
*Moon, John W., and Leo Moser. "On cliques in graphs." *Israel journal of Mathematics* 3 (1965): 23-28.



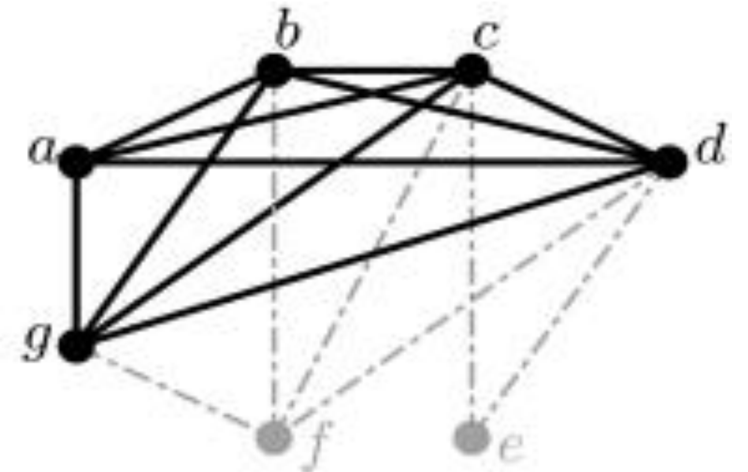
Maximal Cliques: illustrated example



(a) Input Graph G



(b) Non-Maximal Clique in G

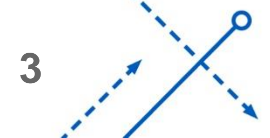


(c) Maximal Clique in G

We can still add **d**, and not break the completeness

For **f** to be considered a candidate to add, it should have been connected to **a**

Diagram from: Apurba Das, Seyed-Vahid Sanei-Mehri, and Srikanta Tirthapura. 2020. Shared-memory Parallel Maximal Clique Enumeration from Static and Dynamic Graphs. *ACM Trans. Parallel Comput.* 7, 1, Article 5



But how do we find maximal clique? The naïve way:

Step 1: Let V be the set of vertices of the graph $G = (V, E)$.

Step 2: Generate all possible subsets $S \subseteq V$.

Step 3: For each subset S:

(a) Check if S is a clique:

- For all $u, v \in S$, ensure $(u, v) \in E$.

(b) If S is a clique, check if it is maximal:

- Ensure there is no vertex $w \in V \setminus S$ such that $S \cup \{w\}$ is also a clique.

Step 4: Output all subsets S that are maximal cliques.

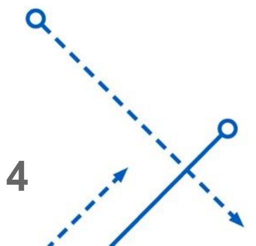
Complexity Analysis

- **Subset Generation:** Generating all subsets $S \subseteq V$ takes 2^n time, where $n = |V|$.
- **Clique Verification:** For each subset S of size k, verifying if S is a clique involves checking $\binom{k}{2} = \frac{k(k-1)}{2}$ edges.
- **Maximal Check:** For a subset S, verifying maximality involves checking $n - k$ vertices outside S.

Overall Complexity:

$$O\left(2^n \cdot k^2\right),$$

where k is the average size of the subsets being processed.



So first, lets speed up serially if possible: enter Bron-Kerbosch

Step 1: **Input:** Graph $G = (V, E)$.

Step 2: **Initialize:**

- $R = \emptyset$: The current clique (starts empty).
- $P = V$: The set of candidate vertices that can be added to R to form a larger clique.
- $X = \emptyset$: The set of excluded vertices that have already been processed to avoid duplication.

Step 3: **Recursive Function:** Call `BronKerbosch`(R, P, X):

(a) If $P \cup X = \emptyset$, then:

- Output R as a maximal clique.

(b) Select a **pivot vertex** $u \in P \cup X$ (heuristically chosen to minimize the size of $P \setminus N(u)$).

(c) For each vertex $v \in P \setminus N(u)$:

- Add v to the current clique: $R' = R \cup \{v\}$.
- Recurse on the subgraph induced by neighbors of v:

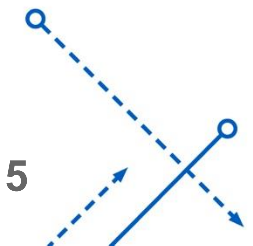
$$\texttt{BronKerbosch}(R', P \cap N(v), X \cap N(v)).$$

- Move v from P to X after returning from recursion.

Step 4: **Output:** All maximal cliques found during recursion.

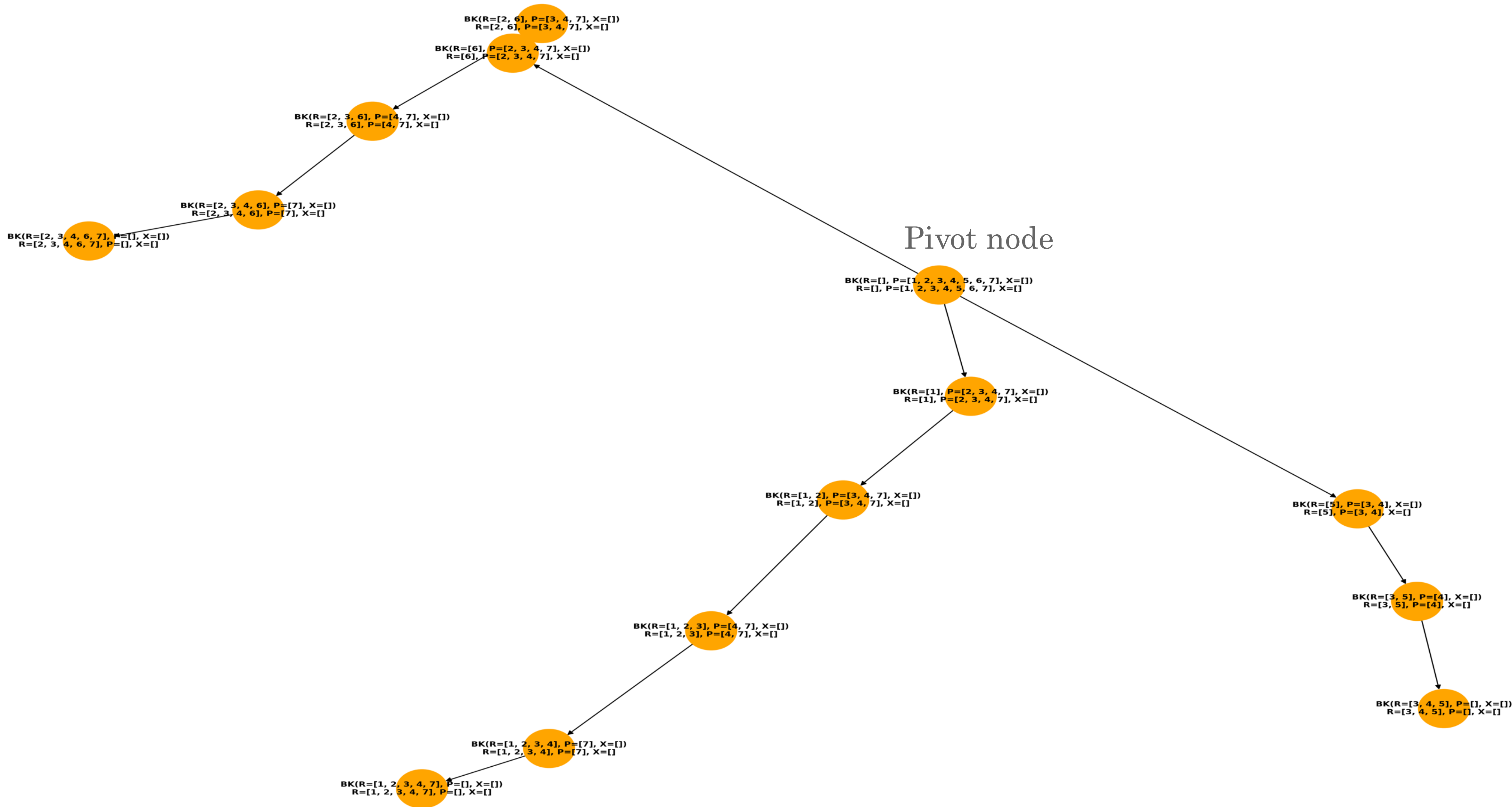
Where,
R (Current Clique): The growing set of vertices forming a clique.
P (Candidates): The set of vertices that can potentially extend R to form a larger clique.
X (Excluded): The set of vertices that have already been processed to ensure no duplicate cliques are found.

Bron-Kerbosch has a worst case run time of $\mathbf{O(3^{n/3})}$ which is far superior to the naïve implementation



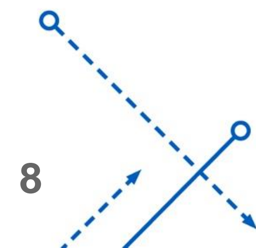
Bron-Kerbosch example

Maximal Clique Found: {1, 2, 3, 4, 7}
Maximal Clique Found: {3, 4, 5}
Maximal Clique Found: {2, 3, 4, 6, 7}



BK(R=[2, 6], P=[3, 4, 7], X=[])
R=[2, 6], P=[3, 4, 7], X=[]
BK(R=[6], P=[2, 3, 4, 7], X=[])
R=[6], P=[2, 3, 4, 7], X=[]
BK(R=[2, 3, 6], P=[4, 7], X=[])
R=[2, 3, 6], P=[4, 7], X=[]
BK(R=[2, 3, 4, 6], P=[7], X=[])
R=[2, 3, 4, 6], P=[7], X=[]
BK(R=[2, 3, 4, 6, 7], P=[], X=[])
R=[2, 3, 4, 6, 7], P=[], X=[]
Pivot node
BK(R=[], P=[1, 2, 3, 4, 5, 6, 7], X=[])
R=[], P=[1, 2, 3, 4, 5, 6, 7], X=[]
BK(R=[1], P=[2, 3, 4, 7], X=[])
R=[1], P=[2, 3, 4, 7], X=[]
BK(R=[1, 2], P=[3, 4, 7], X=[])
R=[1, 2], P=[3, 4, 7], X=[]
BK(R=[5], P=[3, 4], X=[])
R=[5], P=[3, 4], X=[]
BK(R=[3, 5], P=[4], X=[])
R=[3, 5], P=[4], X=[]
BK(R=[1, 2, 3], P=[4, 7], X=[])
R=[1, 2, 3], P=[4, 7], X=[]
BK(R=[3, 4, 5], P=[], X=[])
R=[3, 4, 5], P=[], X=[]
BK(R=[1, 2, 3, 4], P=[7], X=[])
R=[1, 2, 3, 4], P=[7], X=[]
BK(R=[1, 2, 3, 4, 7], P=[], X=[])
R=[1, 2, 3, 4, 7], P=[], X=[]

Parallelization scheme: coarse grained



Parallelization scheme: coarse grained

Algorithm 1 Coarse-Grained Parallel Bron-Kerbosch

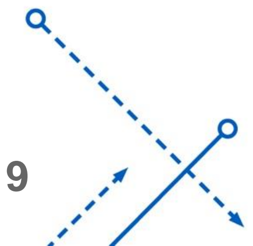
```
1:  procedure ParallelBronKerbosch(R, P, X, graph)
2:      Input: R, P, X (sets), adjacency list of the graph graph.
3:      Output: All maximal cliques in the graph.
4:      Initialize MPI environment
5:      rank ← process rank in MPI world
6:      size ← total number of MPI processes
7:      if rank = 0 then                                  ▷ Master process
8:          Select pivot u ∈ P ∪ X
9:          Compute tasks ← P \ N(u)                      ▷ Divide recursive branches
10:         Distribute tasks to size − 1 worker processes
11:     else                                              ▷ Worker processes
12:         Receive assigned task v from master
13:         Compute R′ ← R ∪ {v}
14:         Compute P′ ← P ∩ N(v), X′ ← X ∩ N(v)
15:         SequentialBronKerbosch(R′, P′, X′, graph)
16:         Send results back to master
17:     end if
18:     if rank = 0 then                                  ▷ Master process collects results
19:         Gather results from all workers
20:         Combine all maximal cliques
21:         Output maximal cliques
22:     end if
23:     Finalize MPI environment
24: end procedure
25: procedure SequentialBronKerbosch(R, P, X, graph)   ▷ Sequential
    version for subproblems
26:     if P ∪ X = ∅ then
27:         Report R as a maximal clique
28:     else
29:         for each v ∈ P do
30:             R′ ← R ∪ {v}
31:             P′ ← P ∩ N(v), X′ ← X ∩ N(v)
32:             SequentialBronKerbosch(R′, P′, X′, graph)
33:             P ← P \ {v}, X ← X ∪ {v}
34:         end for
35:     end if
36: end procedure
```

Here's the thing:
Even before we get into scaling, the speed up using the given algorithm doesn't make sense: the results are very poor

Its like there is insignificant speed up for power-law graphs.

The given coarse-grained parallelism is very very weak

Why is the function SEQUENTIALBRONKERBOSCH() needed if we are going parallel??



Here’s why:

BK(R=[2, 6], P=[3, 4, 7], X=[])
R=[2, 6], P=[3, 4, 7], X=[]

BK(R=[6], P=[2, 3, 4, 7], X=[])
R=[6], P=[2, 3, 4, 7], X=[]

BK(R=[2, 3, 6], P=[4, 7], X=[])
R=[2, 3, 6], P=[4, 7], X=[]

Assigned to worker 1

BK(R=[2, 3, 4, 6], P=[7], X=[])
R=[2, 3, 4, 6], P=[7], X=[]

BK(R=[2, 3, 4, 6, 7], P=[], X=[])
R=[2, 3, 4, 6, 7], P=[], X=[]

This is the only place we are parallelizing

BK(R=[], P=[1, 2, 3, 4, 5, 6, 7], X=[])
R=[], P=[1, 2, 3, 4, 5, 6, 7], X=[]

BK(R=[1], P=[2, 3, 4, 7], X=[])
R=[1], P=[2, 3, 4, 7], X=[]

Assigned to worker 3

BK(R=[1, 2], P=[3, 4, 7], X=[])
R=[1, 2], P=[3, 4, 7], X=[]

BK(R=[5], P=[3, 4], X=[])
R=[5], P=[3, 4], X=[]

Assigned to worker 2

BK(R=[3, 5], P=[4], X=[])
R=[3, 5], P=[4], X=[]

BK(R=[1, 2, 3], P=[4, 7], X=[])
R=[1, 2, 3], P=[4, 7], X=[]

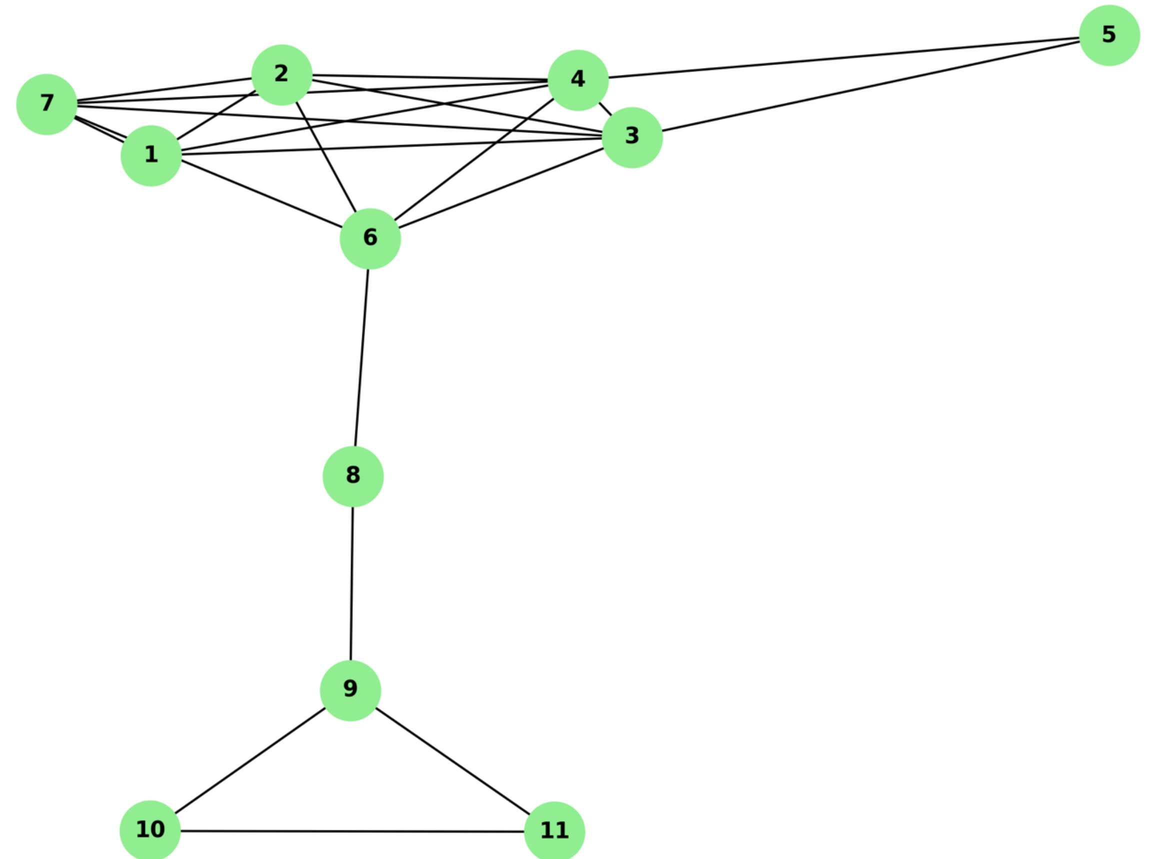
BK(R=[3, 4, 5], P=[], X=[])
R=[3, 4, 5], P=[], X=[]

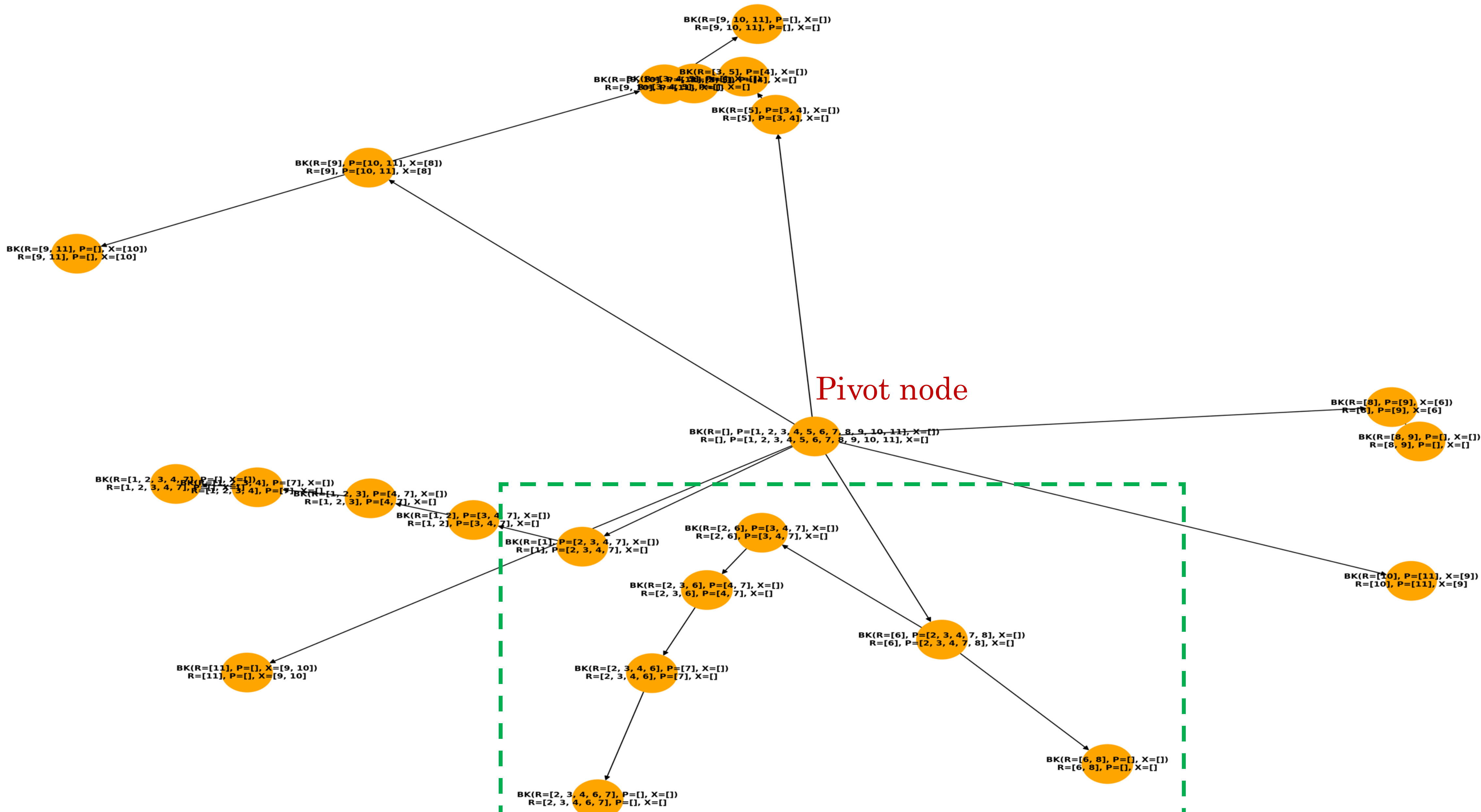
BK(R=[1, 2, 3, 4], P=[7], X=[])
R=[1, 2, 3, 4], P=[7], X=[]

BK(R=[1, 2, 3, 4, 7], P=[], X=[])
R=[1, 2, 3, 4, 7], P=[], X=[]

But what about notoriously branching graphs

Maximal Clique Found: {1, 2, 3, 4, 7}
Maximal Clique Found: {3, 4, 5}
Maximal Clique Found: {8, 6}
Maximal Clique Found: {2, 3, 4, 6, 7}
Maximal Clique Found: {8, 9}
Maximal Clique Found: {9, 10, 11}




BK(R=[9, 10, 11], P=[], X=[])
R=[9, 10, 11], P=[], X=[]
BK(R=[3, 5], P=[4], X=[])
BK(R=[5], P=[3, 4], X=[])
R=[5], P=[3, 4], X=[]
BK(R=[9], P=[10, 11], X=[8])
R=[9], P=[10, 11], X=[8]
BK(R=[9, 11], P=[], X=[10])
R=[9, 11], P=[], X=[10]
Pivot node
BK(R=[], P=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11], X=[])
R=[], P=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11], X=[]
BK(R=[8], P=[9], X=[6])
BK(R=[8, 9], P=[], X=[])
R=[8, 9], P=[], X=[]
BK(R=[1, 2, 3], P=[4, 7], X=[])
R=[1, 2, 3], P=[4, 7], X=[]
BK(R=[1, 2], P=[3, 4, 7], X=[])
R=[1, 2], P=[3, 4, 7], X=[]
BK(R=[1], P=[2, 3, 4, 7], X=[])
R=[1], P=[2, 3, 4, 7], X=[]
BK(R=[2, 6], P=[3, 4, 7], X=[])
R=[2, 6], P=[3, 4, 7], X=[]
BK(R=[2, 3, 6], P=[4, 7], X=[])
R=[2, 3, 6], P=[4, 7], X=[]
BK(R=[2, 3, 4, 6], P=[7], X=[])
R=[2, 3, 4, 6], P=[7], X=[]
BK(R=[2, 3, 4, 6, 7], P=[], X=[])
R=[2, 3, 4, 6, 7], P=[], X=[]
BK(R=[6], P=[2, 3, 4, 7, 8], X=[])
R=[6], P=[2, 3, 4, 7, 8], X=[]
BK(R=[6, 8], P=[], X=[])
R=[6, 8], P=[], X=[]
BK(R=[10], P=[11], X=[9])
R=[10], P=[11], X=[9]
BK(R=[11], P=[], X=[9, 10])
R=[11], P=[], X=[9, 10]
But now multiple branching, has
potential for going more parallel!!

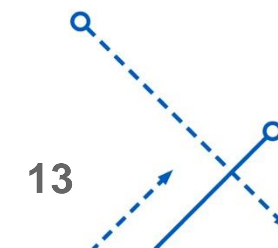
Algorithm 1 Coarse-Grained Parallel Bron-Kerbosch

```
1: procedure ParallelBronKerbosch(R, P, X, graph)
2:     Input: R, P, X (sets), adjacency list of the graph graph.
3:     Output: All maximal cliques in the graph.
4:     Initialize MPI environment
5:     rank ← process rank in MPI world
6:     size ← total number of MPI processes
7:     if rank = 0 then                                   ▷ Master process
8:         Select pivot u ∈ P ∪ X
9:         Compute tasks ← P \ N(u)                       ▷ Divide recursive branches
10:        Distribute tasks to size − 1 worker processes
11:    else                                               ▷ Worker processes
12:        Receive assigned task v from master
13:        Compute R′ ← R ∪ {v}
14:        Compute P′ ← P ∩ N(v), X′ ← X ∩ N(v)
15:        SequentialBronKerbosch(R′, P′, X′, graph)
16:        Send results back to master
17:    end if
18:    if rank = 0 then                                   ▷ Master process collects results
19:        Gather results from all workers
20:        Combine all maximal cliques
21:        Output maximal cliques
22:    end if
23:    Finalize MPI environment
24: end procedure
25: procedure SequentialBronKerbosch(R, P, X, graph)  ▷ Sequential
    version for subproblems
26:    if P ∪ X = ∅ then
27:        Report R as a maximal clique
28:    else
29:        for each v ∈ P do
30:            R′ ← R ∪ {v}
31:            P′ ← P ∩ N(v), X′ ← X ∩ N(v)
32:            SequentialBronKerbosch(R′, P′, X′, graph)
33:            P ← P \ {v}, X ← X ∪ {v}
34:        end for
35:    end if
36: end procedure
```

The issue with this implementation is:
If we do not choose a good pivot node (or even if we do), the multiple branching for each recursion call may explode in some processors, be gentle in others, may completely not utilize some more.....

This is not a good way to parallelize!

Hence, we do dynamic parallelization, where the algorithm when detecting multiple branches assigns tasks on the go!



Algorithm 1 Coarse-Grained Parallel Bron-Kerbosch

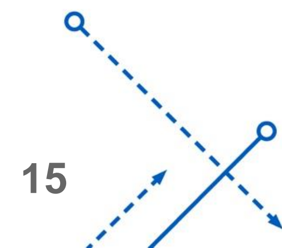
```
 1: procedure ParallelBronKerbosch(R, P, X, graph)
 2:     Input: R, P, X (sets), adjacency list of the graph graph.
 3:     Output: All maximal cliques in the graph.
 4:     Initialize MPI environment
 5:     rank ← process rank in MPI world
 6:     size ← total number of MPI processes
 7:     if rank = 0 then                                   ▷ Master process
 8:         Select pivot u ∈ P ∪ X
 9:         Compute tasks ← P \ N(u)              ▷ Divide recursive branches
10:         Distribute tasks to size − 1 worker processes
11:     else                                              ▷ Worker processes
12:         Receive assigned task v from master
13:         Compute R′ ← R ∪ {v}
14:         Compute P′ ← P ∩ N(v), X′ ← X ∩ N(v)
15:         SequentialBronKerbosch(R′, P′, X′, graph)
16:         Send results back to master
17:     end if
18:     if rank = 0 then                      ▷ Master process collects results
19:         Gather results from all workers
20:         Combine all maximal cliques
21:         Output maximal cliques
22:     end if
23:     Finalize MPI environment
24: end procedure
25: procedure SequentialBronKerbosch(R, P, X, graph)  ▷ Sequential
    version for subproblems
26:     if P ∪ X = ∅ then
27:         Report R as a maximal clique
28:     else
29:         for each v ∈ P do
30:             R′ ← R ∪ {v}
31:             P′ ← P ∩ N(v), X′ ← X ∩ N(v)
32:             SequentialBronKerbosch(R′, P′, X′, graph)
33:             P ← P \ {v}, X ← X ∪ {v}
34:         end for
35:     end if
36: end procedure
```

Note that the function SEQUENTIALBRONKERBOSCH() is missing

Algorithm 1 Dynamic Parallel Bron-Kerbosch using MPI

```
 1: procedure MasterProcess(num_procs)
 2:     Initialize task queue with the root task ⟨R = ∅, P = V, X = ∅⟩
 3:     active_workers ← num_procs − 1
 4:     while active_workers > 0 do
 5:         Receive TASK_REQUEST from any worker w
 6:         if task queue is not empty then
 7:             task ← Dequeue task from task queue
 8:             Send TASK_ASSIGN with task to worker w
 9:         else
10:             Send NO_TASK signal to worker w
11:             active_workers ← active_workers − 1
12:         end if
13:         Receive NEW_SUBPROBLEM or RESULT messages
14:         if NEW_SUBPROBLEM received then
15:             Enqueue received subproblem into task queue
16:         end if
17:         if RESULT received then
18:             Store maximal clique in results list
19:         end if
20:     end while
21:     for each worker w do
22:         Send TERMINATE signal to worker w
23:     end for
24:     Output all stored maximal cliques
25: end procedure
26: procedure WorkerProcess(rank)
27:     while true do
28:         Send TASK_REQUEST to master
29:         Wait for message from master
30:         if message is TASK_ASSIGN then
31:             task ← Received task ⟨R, P, X⟩
32:             if P ∪ X = ∅ then
33:                 Send RESULT with R to master
34:             else
35:                 Select pivot u from P ∪ X
36:                 Q ← P \ N(u)                          ▷ Vertices to explore
37:                 for each v ∈ Q do
38:                     R′ ← R ∪ {v}
39:                     P′ ← P ∩ N(v), X′ ← X ∩ N(v)
40:                     Send NEW_SUBPROBLEM ⟨R′, P′, X′⟩ to master
41:                 end for
42:             end if
43:         else if message is NO_TASK then
44:             Break
45:         else if message is TERMINATE then
46:             Break
47:         end if
48:     end while
49: end procedure
```

Let's break it down



Master Initialization and Task Management

```
MasterProcess():
    Initialize task_queue with root task (R={}, P=V(G), X={})
    active_workers = num_procs − 1

    # Continuously respond to worker requests
    while active_workers > 0:
        Receive TASK_REQUEST from any worker
        if task_queue is not empty:
            # Assign a subproblem for immediate processing
            task = Dequeue(task_queue)
            Send TASK_ASSIGN(task) to that worker
        else:
            # No current tasks available
            # Worker becomes idle, and may be done if no tasks
                return
            Send NO_TASK to that worker
            active_workers −= 1

    # Once all workers signaled no more tasks (active_workers == 0),
    # finalize by sending TERMINATE signals.
    for each worker:
        Send TERMINATE
```

```
// Task is a struct containing (R, P, X) subsets and sizes.
void master_process(int num_procs) {
    int active_workers = num_procs - 1;
    init_task_queue();
    enqueue_task(root_task); // (R={}, P=V, X={})

    MPI_Status status;
    while (active_workers > 0) {
        int request;
        // Workers request task
        MPI_Recv(&request, 1, MPI_INT, MPI_ANY_SOURCE,
                 TASK_REQUEST, MPI_COMM_WORLD, &status);

        if (!is_empty(task_queue)) {
            Task task = dequeue_task(task_queue);
            MPI_Send(&task, sizeof(Task), MPI_BYTE,
                     status.MPI_SOURCE, TASK_ASSIGN, MPI_COMM_WORLD);
        } else {
            // No tasks available at this moment
            MPI_Send(NULL, 0, MPI_BYTE,
                     status.MPI_SOURCE, NO_TASK, MPI_COMM_WORLD);
            active_workers--;
        }
    }

    // Send termination signal to all workers
    for (int i = 1; i < num_procs; i++) {
        MPI_Send(NULL, 0, MPI_BYTE, i, TERMINATE, MPI_COMM_WORLD);
    }
}
```

Master Initialization and Task Management

```
MasterProcess():
    Initialize task_queue with root task (R={}, P=V(G), X={})
    active_workers = num_procs − 1

    # Continuously respond to worker requests
    while active_workers > 0:
        Receive TASK_REQUEST from any worker
        if task_queue is not empty:
            # Assign a subproblem for immediate processing
            task = Dequeue(task_queue)
            Send TASK_ASSIGN(task) to that worker
        else:
            # No current tasks available
            # Worker becomes idle, and may be done if no tasks
                return
            Send NO_TASK to that worker
            active_workers −= 1

    # Once all workers signaled no more tasks (active_workers == 0),
    # finalize by sending TERMINATE signals.
    for each worker:
        Send TERMINATE
```

- Global Task Pool Setup: The master starts with a single root subproblem representing the entire graph. Initially, R=⦰, P=V(G), X=⦰
- On-Demand Assignment: Workers pull tasks by sending **TASK_REQUEST**. If the queue has tasks, the master immediately assigns one
- Idle Workers: If no tasks are currently available, the master sends **NO_TASK**, effectively marking that worker as temporarily idle. Once all workers become idle and no new tasks arrive, the master concludes that the entire search is complete
- Dynamic Adaptation: Because workers will generate more tasks as they expand nodes, the queue may repopulate over time. Idle workers can become active again when new **NEW_SUBPROBLEM** messages from other workers arrive
- Termination: Once no tasks remain and all workers are known to be idle, the master sends **TERMINATE** to finalize execution.

In a coarse-grained approach, the master would distribute large chunks of the recursion tree just once, risking poor load balance if some subproblems are inherently larger. Here, the master only gives out one step at a time and continually re-collects and redistributes tasks, ensuring more uniform workload distribution

Worker internals

```
WorkerProcess():
    while true:
        # Request a task from the master
        Send TASK_REQUEST to master

        # Wait for response
        message = Receive from master

        if message == TASK_ASSIGN(task):
            # Process the assigned task
            (R, P, X) = task
            if (P U X) == empty:
                # Found a maximal clique
                Send RESULT(R) to master
            else:
                # Perform one step of recursion (task expansion)
                subproblems = ExpandTask(R, P, X)
                # Return new subproblems to master
                for each subproblem in subproblems:
                    Send NEW_SUBPROBLEM(subproblem) to master

            # After this, loop again to request a new task
        else if message == NO_TASK:
            # No work available currently
            # Worker remains idle until TERMINATE is received
            # or can re-request after a barrier if implemented
            break
        else if message == TERMINATE:
            # Master signals completion
            break
```

```
void worker_process(int rank) {
    MPI_Status status;
    while (1) {
        int req = 1;
        MPI_Send(&req, 1, MPI_INT, 0, TASK_REQUEST, MPI_COMM_WORLD);

        Task task;
        MPI_Recv(&task, sizeof(Task), MPI_BYTE, 0, MPI_ANY_TAG,
                 MPI_COMM_WORLD, &status);

        if (status.MPI_TAG == TASK_ASSIGN) {
            // Check if this leads to a maximal clique
            if (task.P_size == 0 && task.X_size == 0) {
                // Maximal clique found
                MPI_Send(&task.R, task.R_size * sizeof(Vertex),
                         MPI_BYTE, 0, RESULT, MPI_COMM_WORLD);
            } else {
                // Expand one step and return new subproblems
                expand_task(task);
            }
        } else if (status.MPI_TAG == NO_TASK) {
            // No current work, worker becomes idle
            // Could wait or just break if using a barrier approach
            break;
        } else if (status.MPI_TAG == TERMINATE) {
            // Search completed
            break;
        }
    }
}
```

Worker internals

```
WorkerProcess():
    while true:
        # Request a task from the master
        Send TASK_REQUEST to master

        # Wait for response
        message = Receive from master

        if message == TASK_ASSIGN(task):
            # Process the assigned task
            (R, P, X) = task
            if (P U X) == empty:
                # Found a maximal clique
                Send RESULT(R) to master
            else:
                # Perform one step of recursion (task expansion)
                subproblems = ExpandTask(R, P, X)
                # Return new subproblems to master
                for each subproblem in subproblems:
                    Send NEW_SUBPROBLEM(subproblem) to master

            # After this, loop again to request a new task
        else if message == NO_TASK:
            # No work available currently
            # Worker remains idle until TERMINATE is received
            # or can re-request after a barrier if implemented
            break
        else if message == TERMINATE:
            # Master signals completion
            break
```

- The worker requests tasks when free. This is opposed to the coarse-grained model where tasks are assigned once at the start
- After receiving a task, the worker only performs one level of expansion. It either:
    - Reports a maximal clique if no further branches exist
    - Generates new tasks and returns them to the global pool
- By returning new subproblems promptly, the worker never gets overloaded. Other workers can process these subproblems, achieving **dynamic load balance**
- Advantage Over Coarse-Grained: Instead of a worker being stuck in a long recursion (potentially minutes or hours), it always returns to the master for the next assignment. This ensures that slow-growing or complex parts of the recursion tree do not stay with one worker but are spread out among multiple workers over time.

Expanding task in one step

```
ExpandTask(R, P, X):
    # Select a pivot u from (P U X)
    u = choose_pivot(P, X)
    Q = P \ N(u)

    subproblems = []
    for each v in Q:
        R1 = R union {v}
        P1 = P intersect N(v)
        X1 = X intersect N(v)
        subproblems.add( (R1, P1, X1) )

    return subproblems
```

```
void expand_task(Task t) {
    int u = choose_pivot(t.P, t.X);
    VertexSet Q = set_difference(t.P, neighbors(u));

    // For each v in Q, create a new subproblem
    for (int i = 0; i < Q.size; i++) {
        int v = Q[i];
        Task new_task;
        // Copy R, then add v
        memcpy(new_task.R, t.R, t.R_size * sizeof(Vertex));
        new_task.R[new_task.R_size = t.R_size] = v;
        new_task.R_size = t.R_size + 1;

        // Compute P' and X' as intersection with N(v)
        new_task.P_size = intersect_with_neighbors(new_task.P, t.P, v);
        new_task.X_size = intersect_with_neighbors(new_task.X, t.X, v);

        // Send new subproblem to master
        MPI_Send(&new_task, sizeof(Task), MPI_BYTE, 0,
                 NEW_SUBPROBLEM, MPI_COMM_WORLD);
    }
}
```

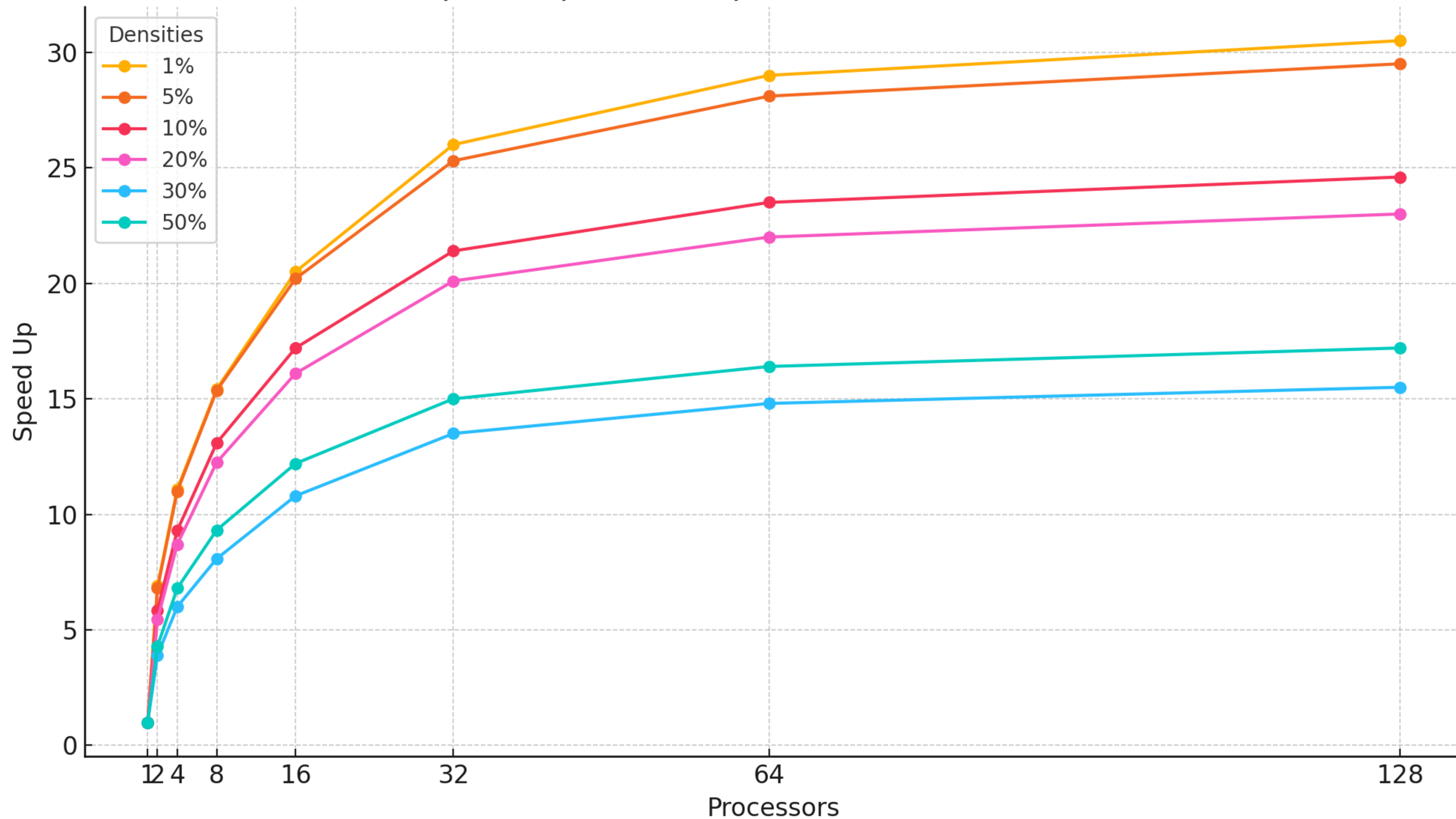
We analyze average of 10 parallel runs for speed ups (X) with respect to serial on an 8-core parallel:

%Density/#Nodes	10,000	30,000	50,000	80,000	100,000
1%	15.42	9.53	8	9.11	8.11
5%	15.36	9.56	14.8	8.19	10.69
10%	13.09	9.22	8.23	14.87	9.65
20%	12.25	11.23	8.9	10.15	10.43
30%	8.08	9.53	13.01	8.95	9.95
50%	9.32	7.95	8.16	6.54	6.88

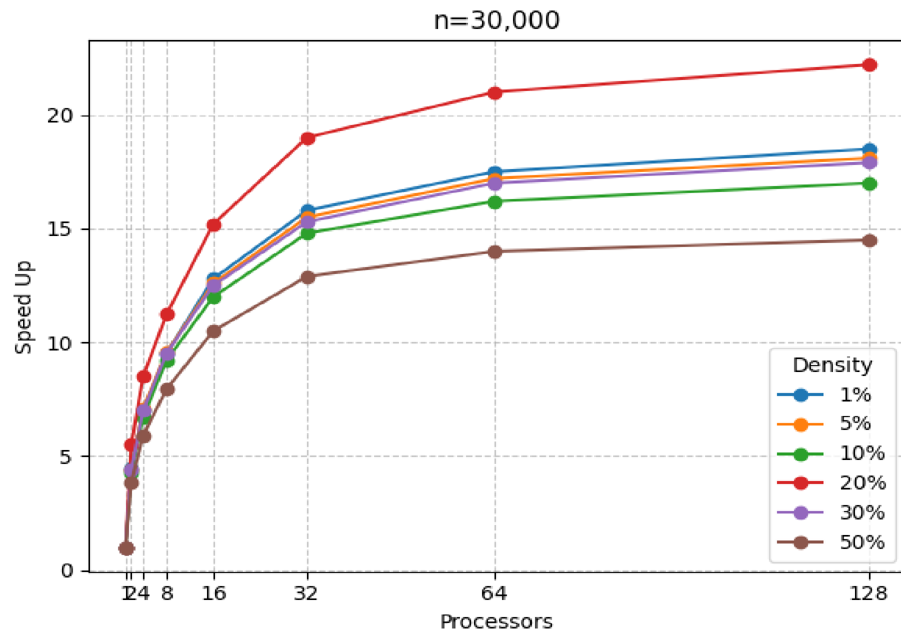
On average:

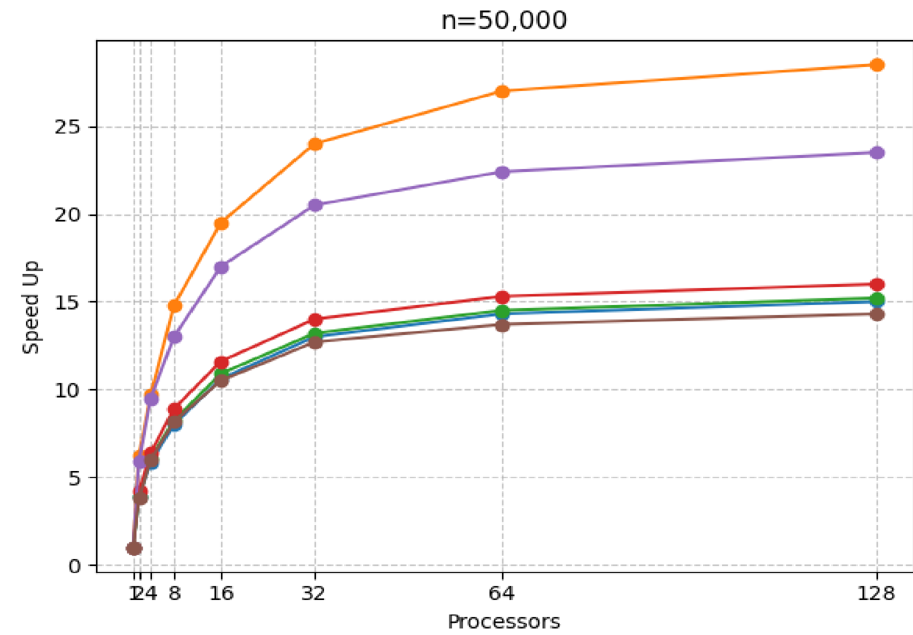
- At worst, the speed ups are > 6 times
- At best, it is up to 15 times
- Less dense graphs show better results as there are less sub problem creation

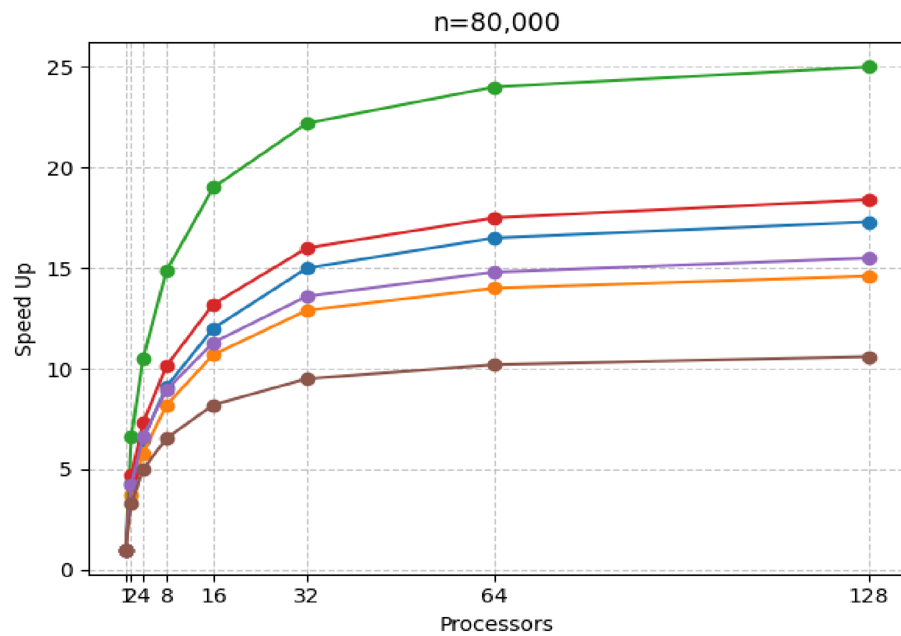
Strong scaling N = 10,000

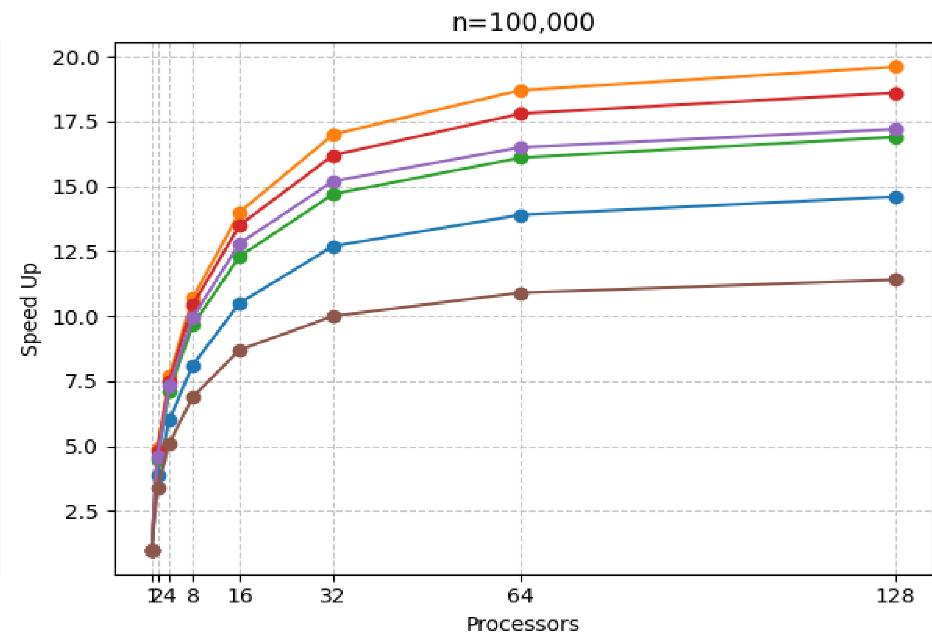


Speed Up with Respect to Serial for Different n









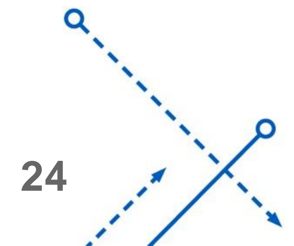
Strong scaling Density = 10%

Speed Up Across Processors for 10% Density

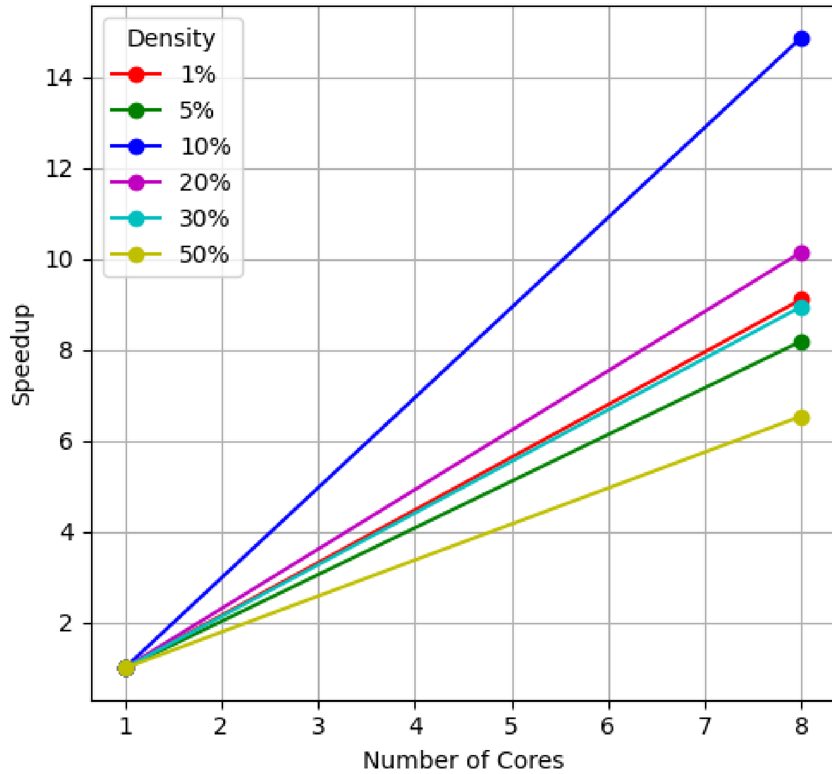
Nodes (n)
n=30,000
n=50,000
n=80,000
n=100,000

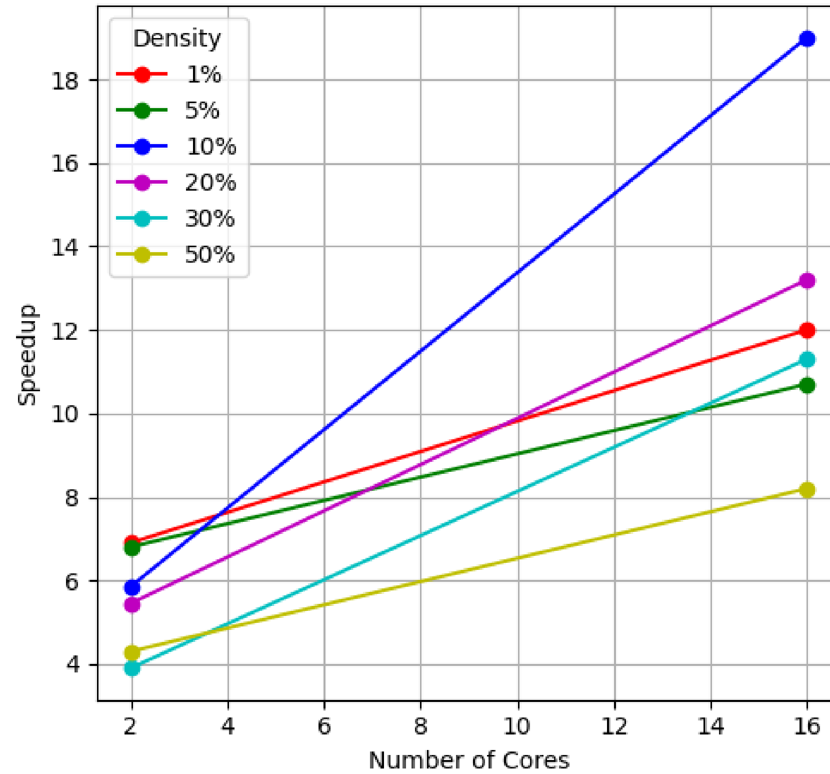
Speed Up
0
5
10
15
20
25

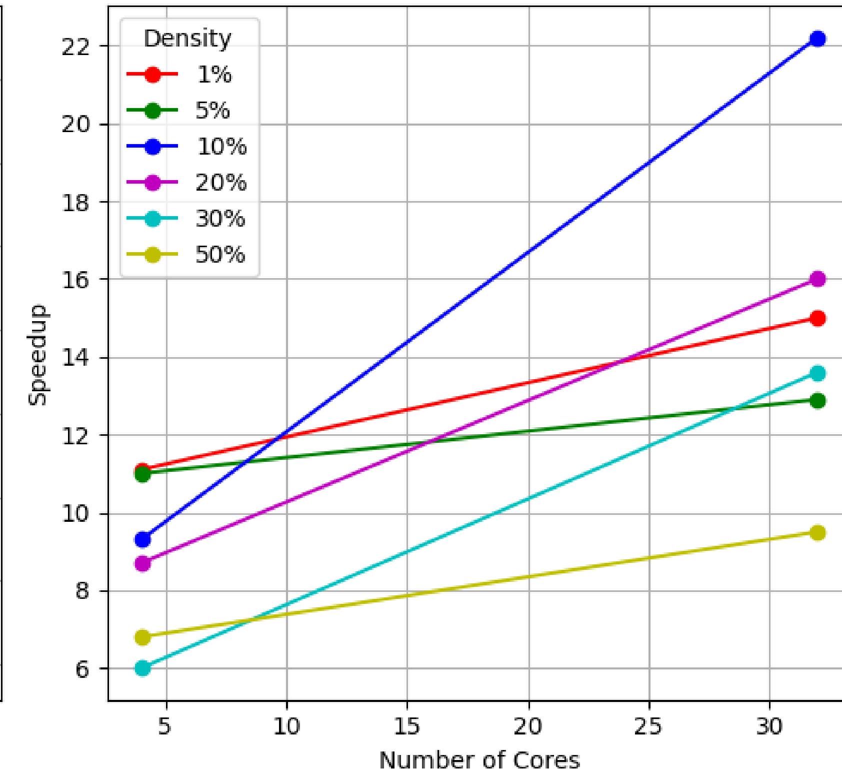
Processors
1
2
4
8
16
32
64
128



Weak scaling measurements from the experiments





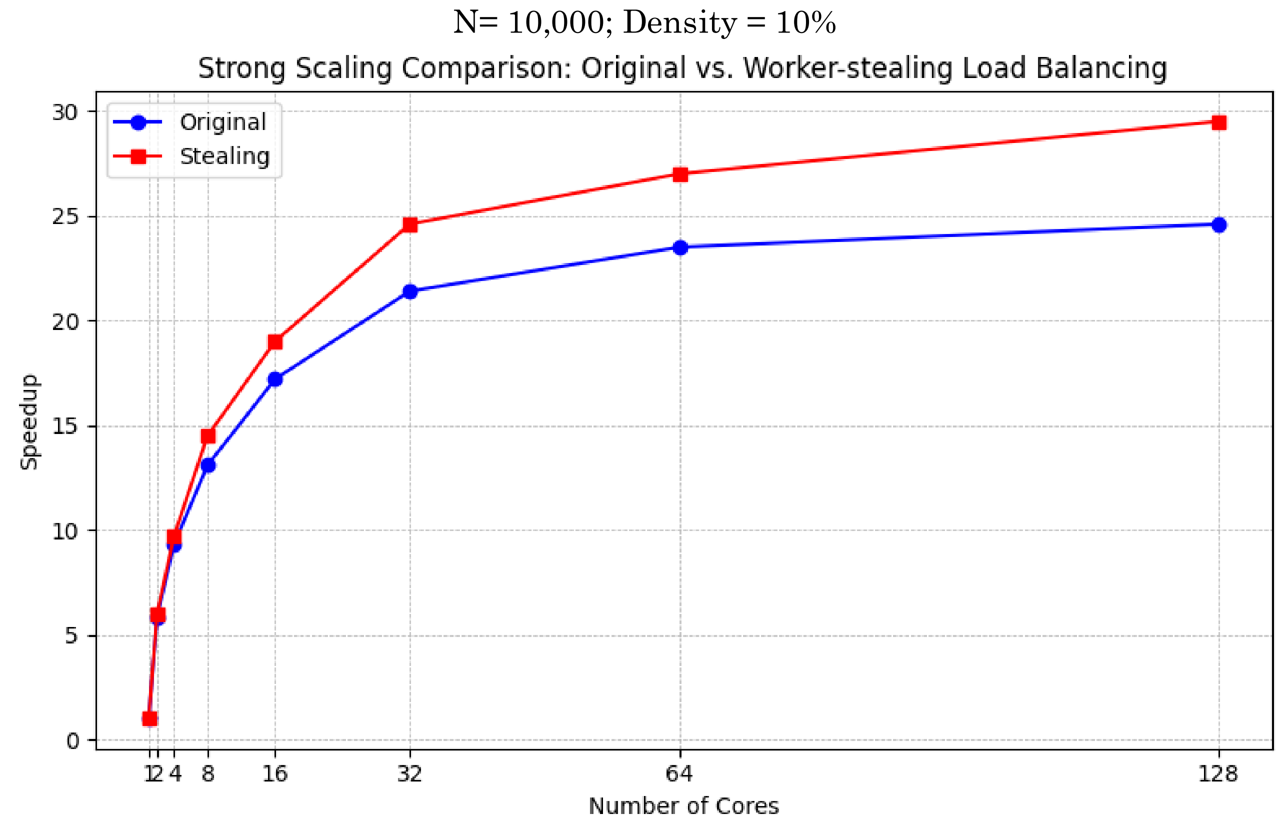


Making workers communicate amongst themselves: strong scaling improvements

- Instead of sending task to master, each worker maintains a local queue of created jobs
- When a worker is free, it searches for list in worker + 1 if they have any jobs in the queue
- In case of a job, the worker will steal the job and execute

Please note- This is inconclusive, even if it is promising. One data point improvement does not generalize across all graphs

N= 10,000; Density = 10%



What's next?

- Load balancing is a promising avenue:
  - Worker stealing logic needs to be better
  - We should be able to dynamically predict load requirements based on node characteristics
- Read level statistics:
  - When reading the graph we can note certain stats in O(E) to inform us of the structure and keep workers ready
- GPU worker stealing:
  - Worker stealing for maximal enumeration on GPU is unexplored, given GPUs massive parallelism and different memory hierarchy it offers a lot more options