A* ALGORITHM

Single-thread and Multi-thread implementation of A* optimal path-searching algorithm

Group members:

- Paoli Leonardi Francesco (s297078)
- Pasqualini Federico (s296488)

System and Device Programming course project (a.a. 2021/2022)

PROJECT OUTLINE

1. Graph data structures
2. Graph algorithms design and implementation
3. A* algorithms design and implementation
4. A* performance comparison

GRAPH DATA STRUCTURES

GRAPH NODE STRUCTURE

```
// set of links (and corresponding weights) that connect a node with its neighbors
typedef std::unordered_map<unsigned int, unsigned int> link_weight_umap;

...

/******************** Graph's nodes structure ********************/
class Node {

    public:
        /* copy control features definition */
        ...

        /* overloaded operators */
        ...

        /* getters and setters */
        ...

    private:
        unsigned int id;    // node's ID
        int x;              // node's coordinate along x axis
        int y;              // node's coordinate along y axis
        std::unique_ptr<link_weight_umap> neighbor;     // node's links to neighbor nodes
};
```

Highlighted implementation choices:

- Copy-control features, overloaded operators and getters/setters for better interface to *Node* objects.
- Pointer to *link_weight_umap* to have constant *Node* size.
- Links-weights hash map for constant random-access complexity.

GRAPH PATH STRUCTURE

```
// set of nodes in a graph's path (the order in which the nodes must be traversed is given
by the key of the hash map)
typedef std::unordered_map<unsigned int, Node> path_umap;

...

/******************** Path (i.e. sequence of nodes) of the graph ********************/
typedef struct graph_path_s {

    std::unique_ptr<path_umap> path_ptr;    // path itself
    unsigned int path_num_nodes;            // number of nodes contained in the path
    std::atomic<int> path_cost;             // path's overall cost

} graph_path_t;
```

It will store the information of best path found by A* (concurrently modified by running threads, so we consider it as a critical section).

Highlighted implementation choices:

- Atomic path's overall cost to guarantee mutual exclusion without any mutex.
- Pointer to *path_umap* to have constant *graph_path_t* size.
- Path nodes sequence hash map for constant random-access complexity.

GRAPH ALGORITHMS
DESIGN AND IMPLEMENTATION

GRAPH GENERATION AND STORAGE

First accomplished task was the implementation of functions for:

- Generating of a random graph (with given number of nodes).
- Storing of the generated graph in long-term memory.

Highlighted implementation choices:

- Graphs up to millions of nodes, so we implemented both single and multi-thread algorithms.
- C *pthread_mutext* and *sem_t* and C++ *std::mutex* used for synchronization (e.g. access mutually exclusive critical sections, implementation of thread barriers).
- Graph divided in partitions for efficient multi-thread graph generation and storage.
  - Managed by C++ *std::atomic_flag::test_and_set* (<<*std::mutex::try_lock* method is allowed to fail spuriously>>[1]).

[1] https://en.cppreference.com/w/cpp/thread/mutex/try_lock

NODES GENERATION

- The graph gets divided as a 2D squared grid, where is grid cell is a graph partition.
- Each graph partition gets filled up of nodes by the first thread that locks it (critical section).
- Each node has *x-y* coordinates generated randomly within its partition coordinates bound.

Highlighted implementation choices:

- The number of graph partitions is quite larger than the number of threads.
  - Fastest threads can fill up more partitions.
  - Avoid bottleneck of slowest thread.
- Each partition needs to be visited only once.
  - C++ *std::atomic_flag::test_and_set* was used to avoid useless thread waits.

LINKS GENERATION

- Links are generated between nodes belonging to
  1. the same partition.
  2. to adjacent partitions.

1. The number of links both starting and ending in a partition A is
   - Random.
   - Proportional to $\log_2 N_A$ [1].
2. The number of links from partition A to B (adjacent partitions) is
   - Deterministic
   - Proportional to $\sqrt{N_B}$ [1].

Highlighted implementation choices:

- Each node has at least one incoming and one outcoming link
  - Graph guaranteed to be strongly connected (there's always a feasible path between any nodes pair).

[1] N_A and N_B are the number of nodes belonging to partition A and B, respectively.

GRAPH STORAGE IN LONG-TERM MEMORY

After the graph generation, the nodes and links information are serialized in a file (specified by the user) from which it can be retrieved when needed.

Ad-hoc serialization protocol:

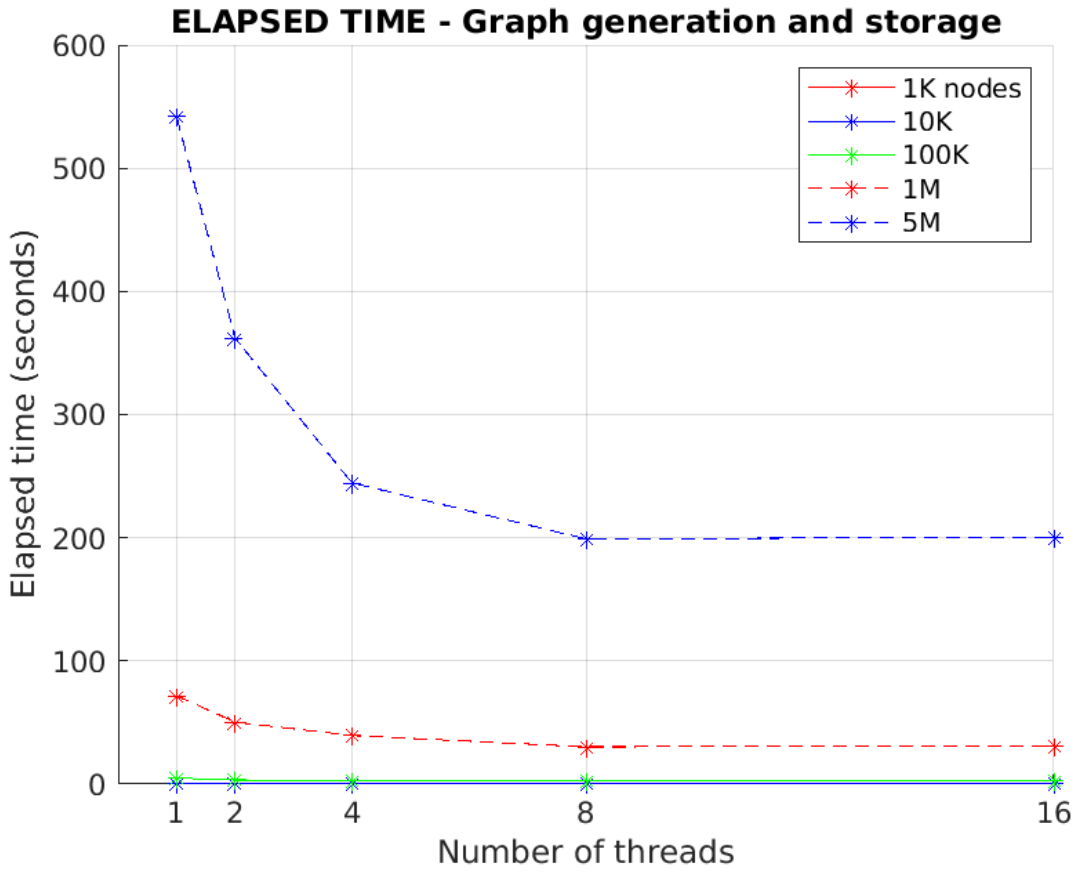
1. 12 bytes (3 *unsigned int*) for
   - Number of generated nodes.
   - Number of generated links.
   - Number of generated graph partitions.
2. For each graph partition, 4 bytes (1 *unsigned int*) for the number of links inside that partition.
3. For each graph partition, 12 bytes (3 *unsigned int*) for information regarding the nodes of that partition:
   - *ID* of the node.
   - *x* coordinate of the node.
   - *y* coordinate of the node.
4. For each graph partition, 12 bytes (3 *unsigned int*) for the information regarding the links of that partition:
   - *ID* of the starting node.
   - *ID* of the ending node.
   - Link's weight.

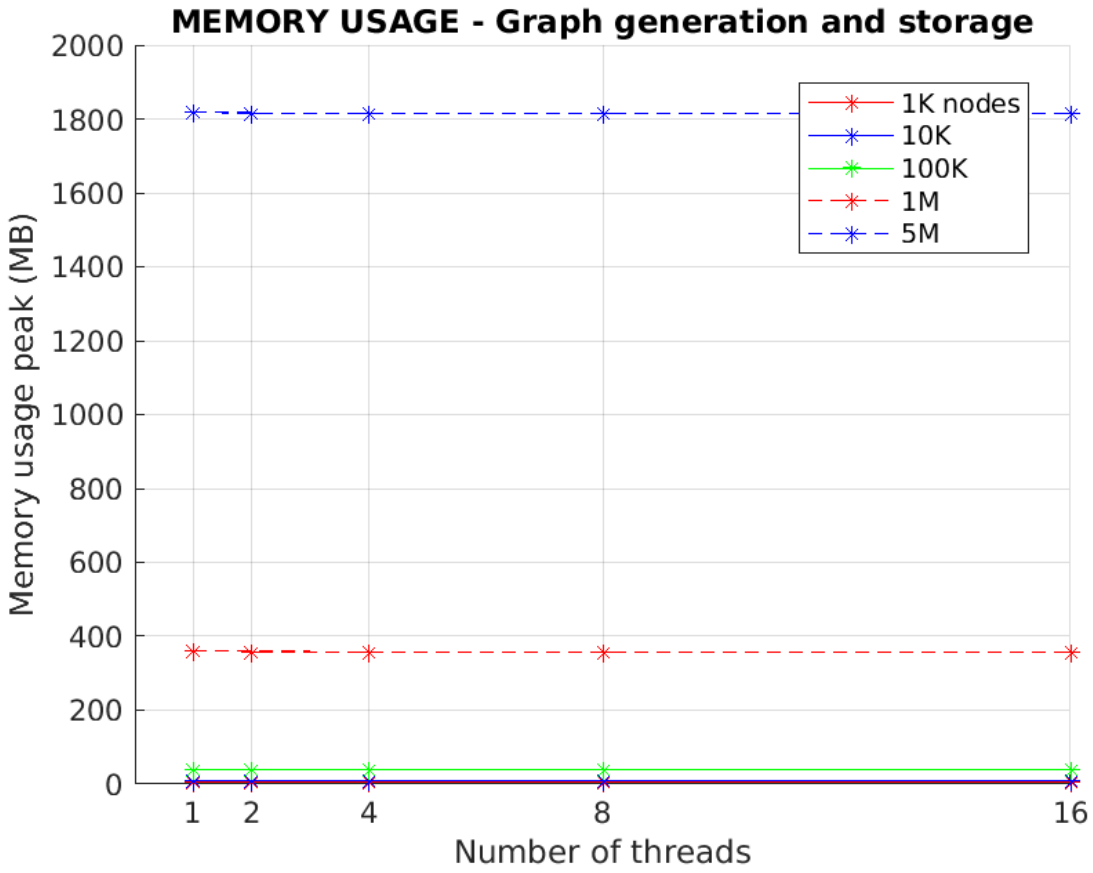
GRAPH STORAGE IN LONG-TERM MEMORY

Highlighted implementation choices:

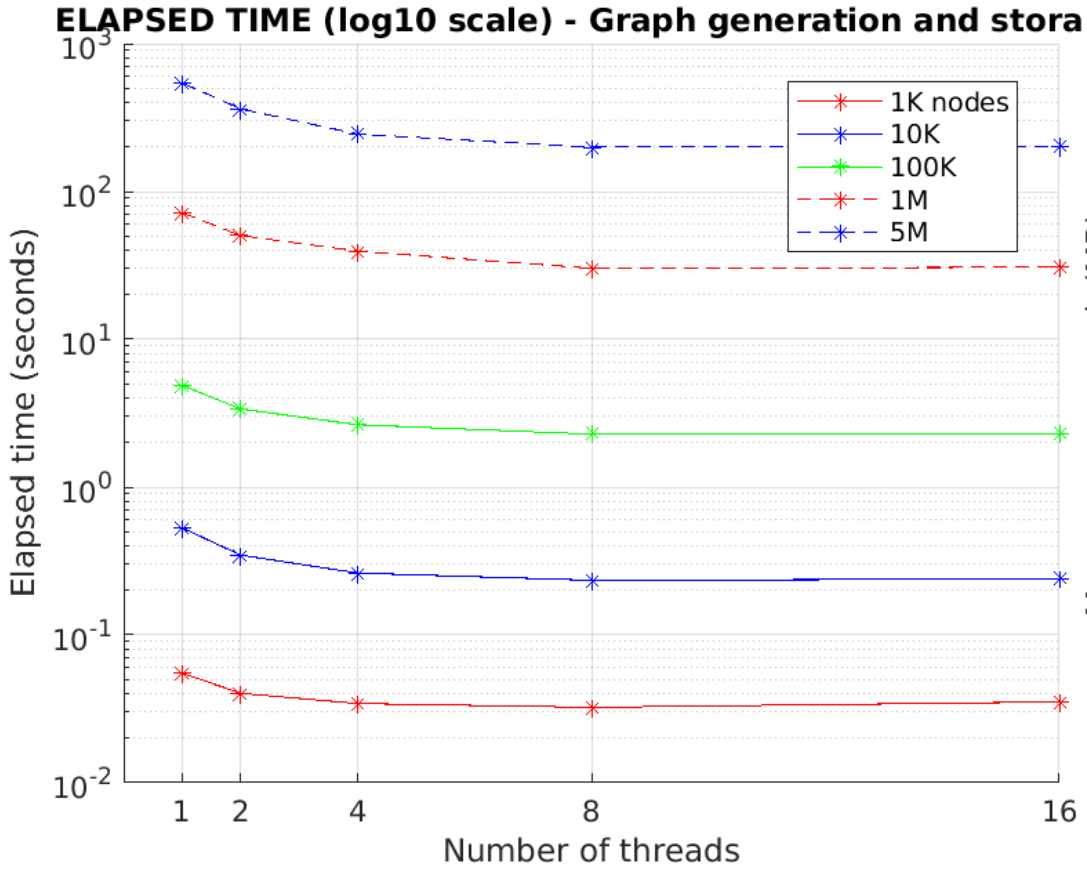
- Graph data get serialized in binary format.
  - Occupy less disk space.
- The graph storing procedure gets performed in parallel (with proper synchronization strategies).
- Each thread performs file locking of the file region it is about to write on.
  - Guarantees that two or more threads never write on the same file region at the same time.
- After each file write, threads check that the number of actually written bytes is what they expect.

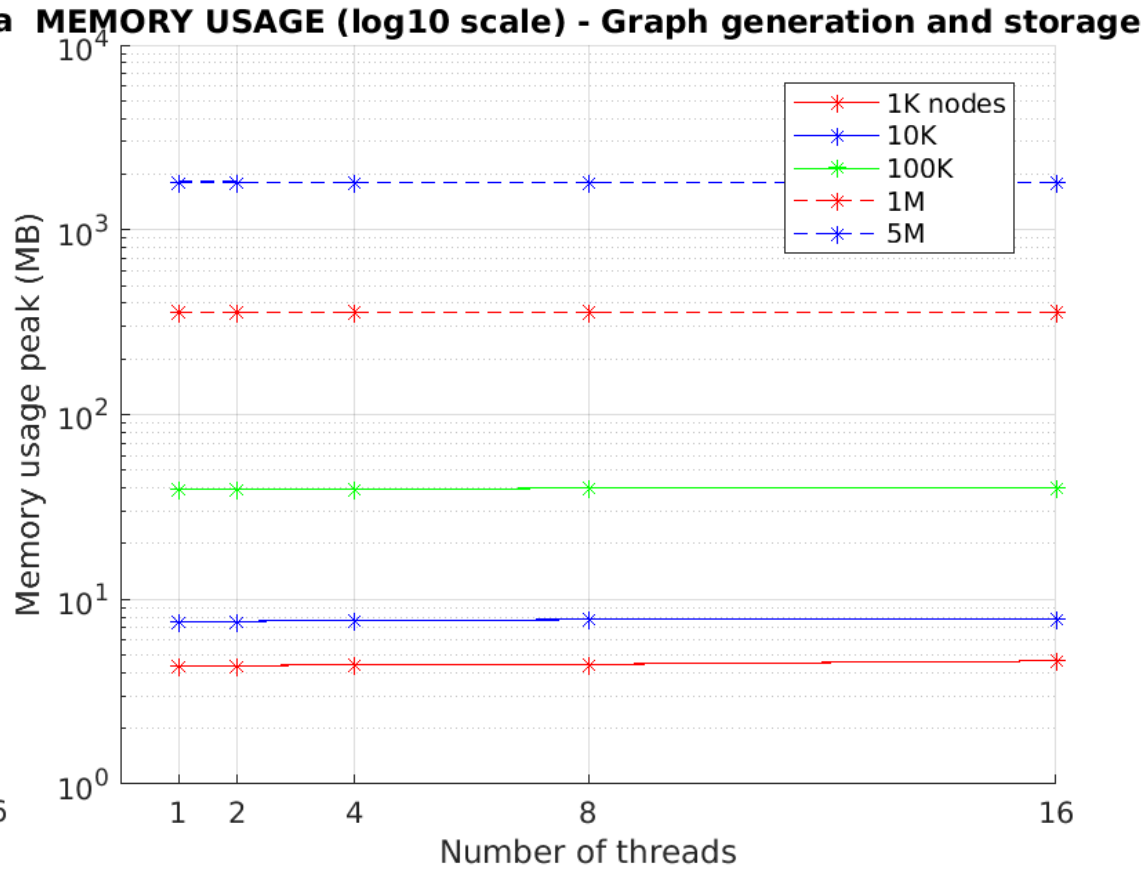
GRAPH GENERATION AND STORAGE PERFORMANCE





GRAPH GENERATION AND STORAGE PERFORMANCE





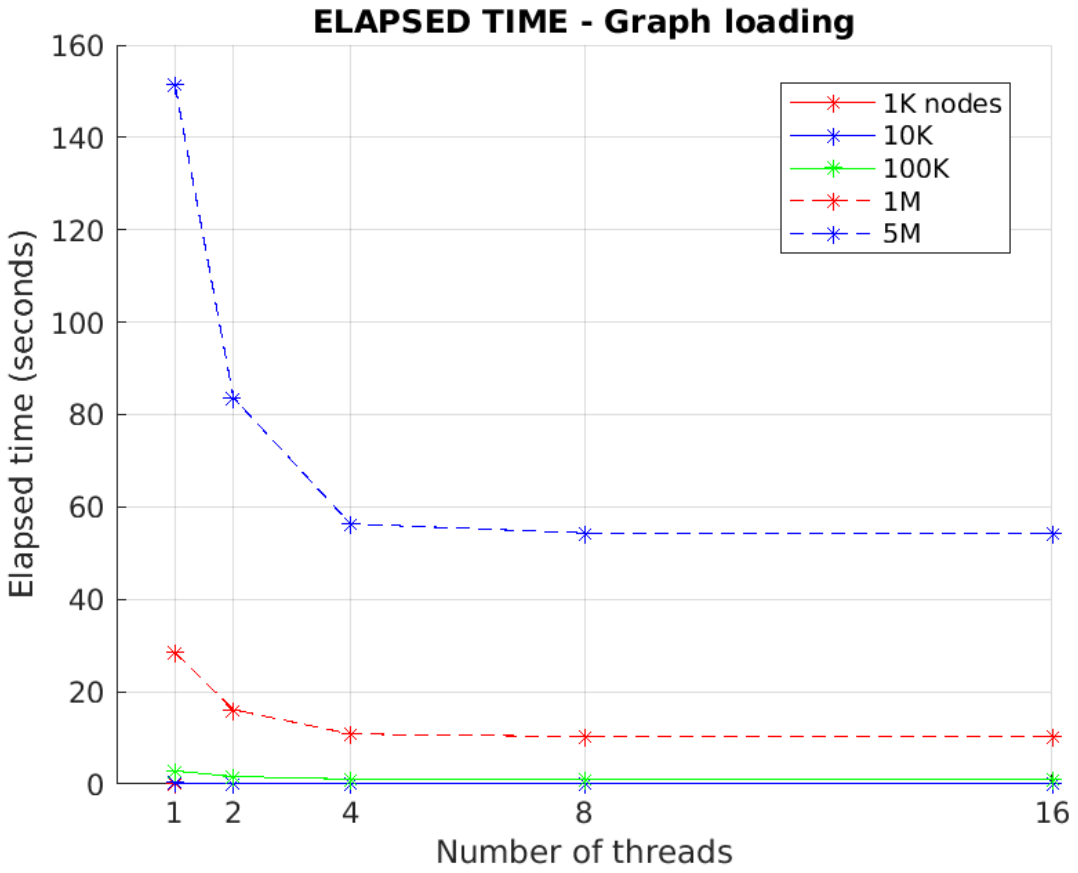
GRAPH LOADING FROM LONG-TERM MEMORY

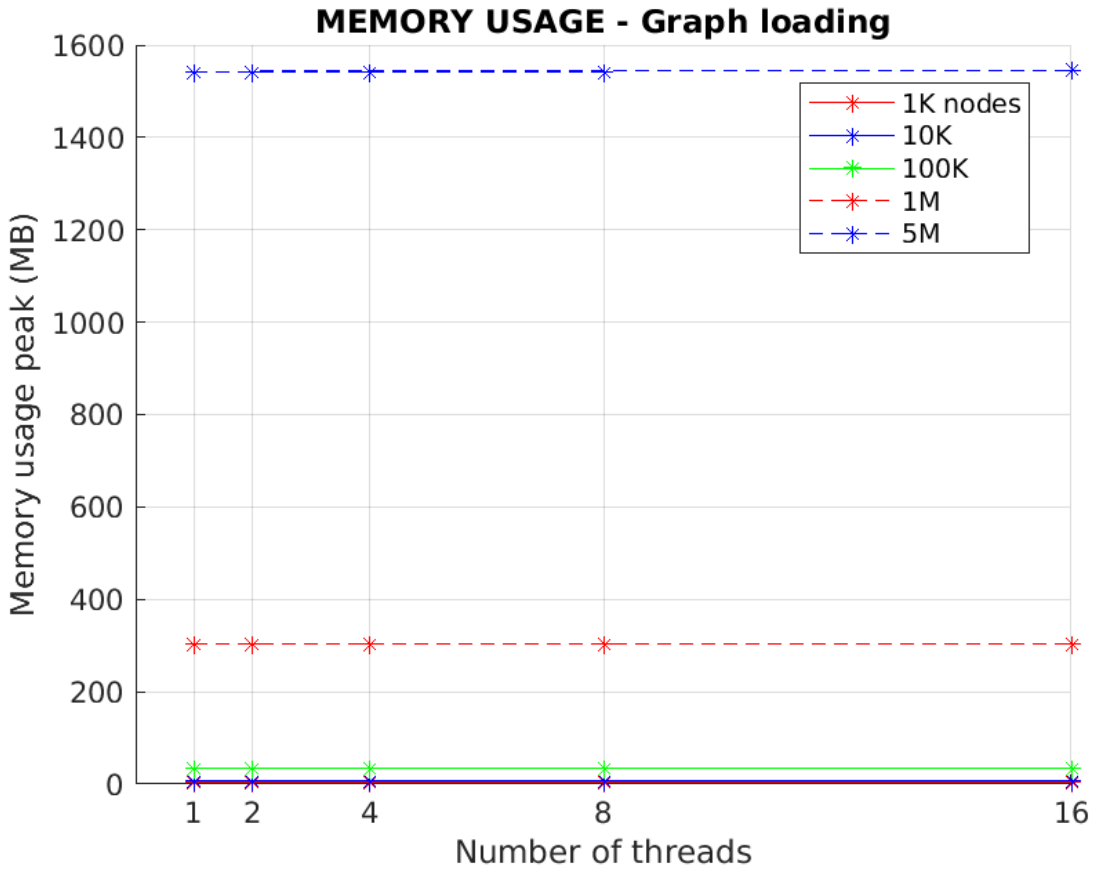
Following the same serialization protocol used for graph storage, a graph previously generated can be retrieved from disk, deserialized and used to run A* on it.

Highlighted implementation choices:

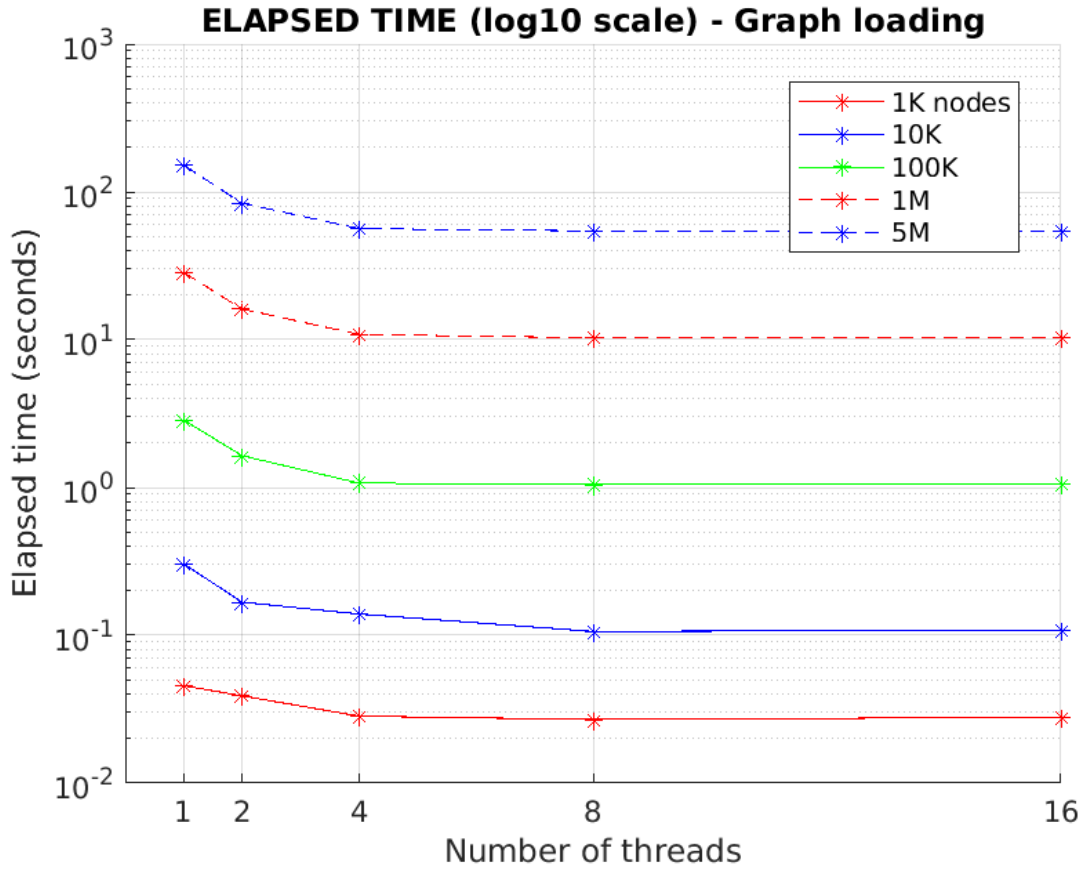
- The graph loading procedure gets performed in parallel (with proper synchronization strategies).
- After each file read, threads check that the number of actually read bytes is what they expect.

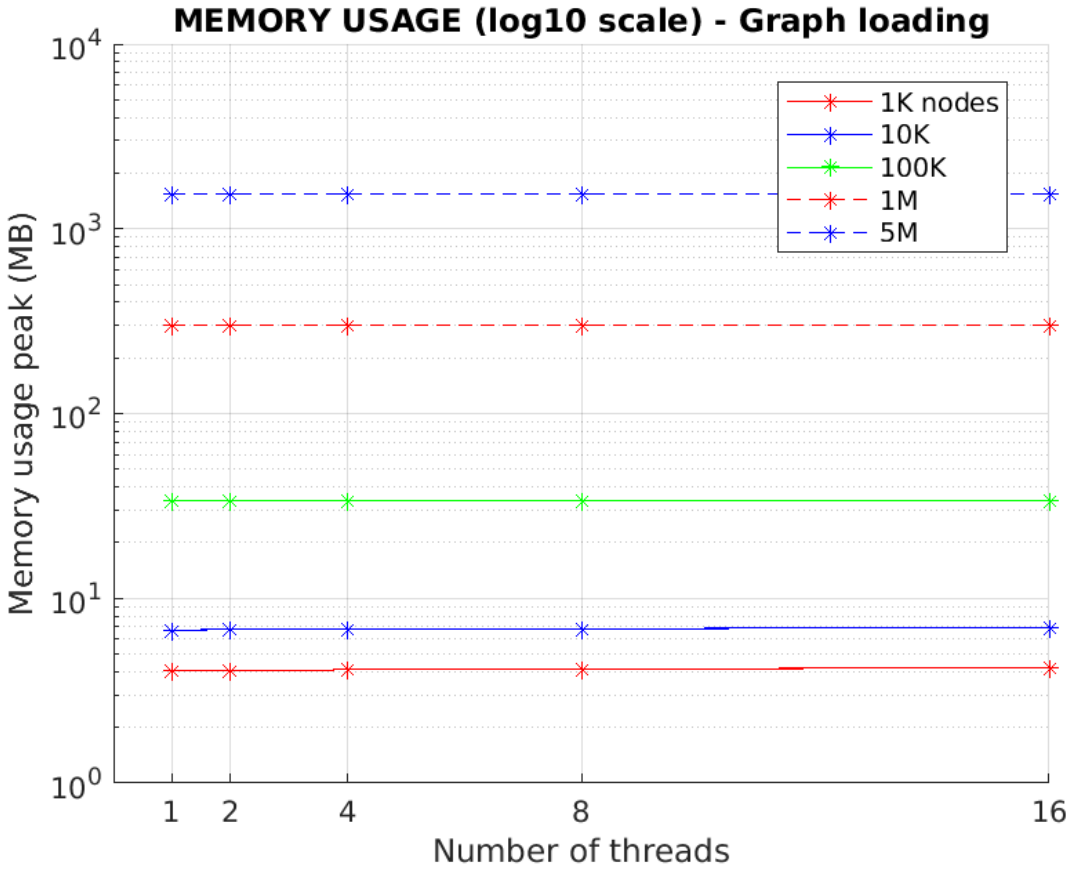
GRAPH LOADING PERFORMANCE





GRAPH LOADING PERFORMANCE





GRAPH GENERATION, STORAGE AND LOADING OBSERVATIONS

- **Elapsed time** during graph generation, storage and loading procedures **decreases** significantly with a **higher number of threads**.
- **Memory usage** remains approximately **constant** and **independent on the number of threads**.
  - The memory needed to run a new thread is negligible with respect to the memory needed to generate/store/load a graph.

A* ALGORITHMS
DESIGN AND IMPLEMENTATION

A* ALGORITHM VERSIONS

3 different versions have been implemented:

- Sequential A*: classic A* algorithm running on a single thread.
- Centralized A*: runs on multiple threads and all the data structures are shared between them and protected by appropriate synchronization primitives.
- Decentralized A*: highly optimized parallel version where each thread has its own data structures, so that locks and resources shared among all threads are reduced as much as possible.

Through the provided user interface (*menu.cpp/.h*), the user can choose the A* version to run and its parameters:

- Name of the file storing the graph.
- Number of threads to launch.
- *Start* and *stop* nodes between which A* has to find a path.
- Hash-type (for Decentralized A*).

SEQUENTIAL A* DATA STRUCTURES

The Sequential A* version mainly exploits the following data structures:

- g_{cost}: cost of traversing the currently best path found from *start* node to the current node.
- h_{cost}: estimation of the cost to arrive from the current node to the *stop* node.
- f_{cost}: estimated overall path cost passing through the current node ($f_{cost} = g_{cost} + h_{cost}$).
- *open* list: keep track of the nodes that still need to be expanded, together with their f_{cost} and h_{cost}.
- *closed* list: keep track of the nodes already expanded.
- *from* list: for each graph node, keep track of its current parent (i.e., the node that, along the current best path, immediately precedes it).
- *cost* list: keep track of the current best cost (i.e., g_{cost}) of each node.

SEQUENTIAL A* DATA STRUCTURES

Highlighted implementation choices:

- *open* list is a C++ *std::priority_queue*
  - Allows fast insertion and fast removal of the lower cost node (*O(1)* and *O(logN)* complexity, respectively).
  - Doesn't allow random-access but it is not needed.
- *closed*, *from* and *cost* lists are C++ *std::unordered_map*
  - Allow random-access with constant complexity.
- The heuristic h_{cost} is the Euclidean norm distance because it is:
  - Admissible, i.e., it never overestimates the real distance of *stop* node from the current node.
  - Consistent, i.e., it satisfies the triangle inequality.

  This guarantees that the first path found by Sequential A* is always the globally optimal one (not true for Centralized and Decentralized A*).

SEQUENTIAL A* ALGORITHM (OUTLINE)

1. Extract from *open* list the node with the lowest cost.
2. Check if already in *closed* list:
   - If yes, discard it (we already visited it with a lower cost).
   - If no, expand it and continue.
3. Check if the expanded node is *stop*:
   - If yes, the algorithm terminates and the path from *start* to *stop* nodes gets rebuilt.
   - If no, continue.
4. For each neighbor of the expanded, check if the neighbor is worth expanding too:
   - If yes, add it to the *open* list.
   - If no, discard it.
5. Loop through the steps 1-4.

CENTRALIZED A*

Data structures:

- Same as Sequential A*.
- Synchronization primitives to access thread-shared data.

Algorithm implementation:

- Similar to Sequential A*.
- Managing of critical sections.
- Ad-hoc termination detection algorithm.
  - Advantages given by an admissible and consistent heuristic are not valid for multi-thread A*.

CENTRALIZED A* TERMINATION DETECTION

A thread can stay idle if at least one of the following conditions is met:

- *open* list is empty.
- The node with lower cost still has a higher cost than the current best path to *stop* node.

At every iteration, each thread checks both conditions for itself.

- If both are false, continue.
- If at least one is true, check them also for every other thread.
  - If, during their last check, every thread satisfied at least one of the above conditions, then the Centralized A* can terminate.
  - Otherwise, continue.

DECENTRALIZED A*

Decentralized A* optimizes thread parallelization by

- Removing shared data structures (except few necessary ones).
- Assigning nodes to threads
  - Each node is “owned” by only one thread, which will be responsible for expanding it.
- Implementing thread communication through messages.

The Decentralized A* have been implemented as HDA* (Hash Distributed A*).

- Nodes are assigned to threads by a hash function.

Highlighted implementation choices:

- The possibility of assigning nodes **randomly** to threads was discarded in the first place.
  - To avoid useless duplication of nodes expansion, nodes must be assigned to threads in a **deterministic** way.

DECENTRALIZED A* HASH METHODS

3 different hash functions (can be chosen through the user interface):

- Multiplicative Hash (MDHA*)
  - Given a node, the thread ***ID*** of its owner gets computed as $N \cdot (k \cdot A \ - \ \lfloor k \cdot A \rfloor)$, where:
    - N is the number of running threads.
    - A can be any number (golden ratio showed to work well [1]).
    - k is obtained by hashing the ***x-y*** coordinates of the node.
- Zobrist Hash (ZHDA*)
  - Given a node, the thread ***ID*** of its owner gets computed as $R[x'] \oplus R[y']$, where:
    - $\oplus$ is the XOR operator.
    - x' and y' are positive projections of node's ***x*** and ***y*** coordinates, respectively.
    - R is a random-bit-strings table (i.e., vector filled by random bits).
- Abstract Zobrist Hash (AZHDA*)
  - Given a node, the thread ***ID*** of its owner gets computed as $R[A(x')] \oplus R[A(y')]$
    - Same as ZHDA* but uses a projection function A to assign near nodes to the same thread.

[1] "A Survey of Parallel A*". Alex Fukunaga, Adi Botea, Yuu Jinnai, Akihiro Kishimoto. August 18, 2017.

DECENTRALIZED A* HASH METHODS

Multiplicative Hash (MDHA*)

Pros:

- Simple.
- Doesn't need to fill R vector.

Cons:

- Highly relies on a good hash function to compute k (not trivial).

Zobrist Hash (ZHDA*)

Pros:

- Almost perfect work balance distribution.

Cons:

- Non-negligible overhead due to thread communication.

Abstract Zobrist Hash (AZHDA*)

Pros:

- Uses a projection function to reduce thread communication overhead.
- Still almost perfect work balance distribution.

DECENTRALIZED A* DATA STRUCTURES

Data structures:

- Same as Sequential A*.
- Synchronization primitives to manage **best path information** (critical section).
- Thread messages for **information exchange** and **path rebuild**:

```
// message exchanged between threads in multi-threaded Decentralized A* algorithm (the tuple contains
node N, cost g(N) and parent(N))
typedef std::tuple<Node, double, Node> msg_t;

// buffer of messages
typedef std::queue<msg_t> msg_buffer_t;

/* parent request message (key-value are, respectively, the node of which it is requested to know the
parent and the flag that represents
if a new parent request arrived or not) */
typedef std::pair<Node, bool> parent_request_t;

/* parent reply message (key-value are, respectively, the requested parent node and the flag that
represents if a new parent reply is available or not) */
typedef std::pair<Node, bool> parent_reply_t;
```

DECENTRALIZED A* ALGORITHM

Algorithm implementation:

- Same idea of Sequential A*, but multi-thread.
- Graph knowledge is distributed among all threads.
  - When a new node gets discovered, send it to its owner thread.
  - To rebuild the best path, it is needed the knowledge of multiple threads.
    - Parent requests.
- Termination detection algorithm.
  - Vector counters algorithm [1].
  - Advantages given by an admissible and consistent heuristic are not valid for multi-thread A*.

[1] “Algorithms for distributed termination detection”. Friedemann Mattern. 1987.

DECENTRALIZED A* ALGORITHM (OUTLINE)

1. Fill up R random vector (potentially in parallel).
2. Check if current thread has nodes to be expanded or if its knowledge is needed from another thread to rebuild the best.
   - If yes, continue.
   - If no, the thread starts sleeping on a condition variable to save CPU computation.
     - It will be woken up again when needed.
3. Check if current thread has any message to be read containing information about new nodes to be expanded.
   - If yes, read all the messages and them to the *open* list if it is worth it.
   - If no, continue.
4. Expand the lowest cost node (if it is actually worth it).
5. Check if expanded node is *stop* node.
   - If no, look at all the node's neighbors and send them as messages to their owners.
   - If yes, start the path rebuild procedure.
     - Exchange a series of messages among threads in order to rebuild the path from *start* to *stop*.
6. Loop through steps 2-5.

DECENTRALIZED A* TERMINATION DETECTION

Each thread manages:

- A local counter vector containing counters of messages sent/received to/by other threads.
- An additional *acc_msg_counter* vector that
  - Holds the overall count of every thread-local vector.
  - Goes from one thread to another in a circular way.

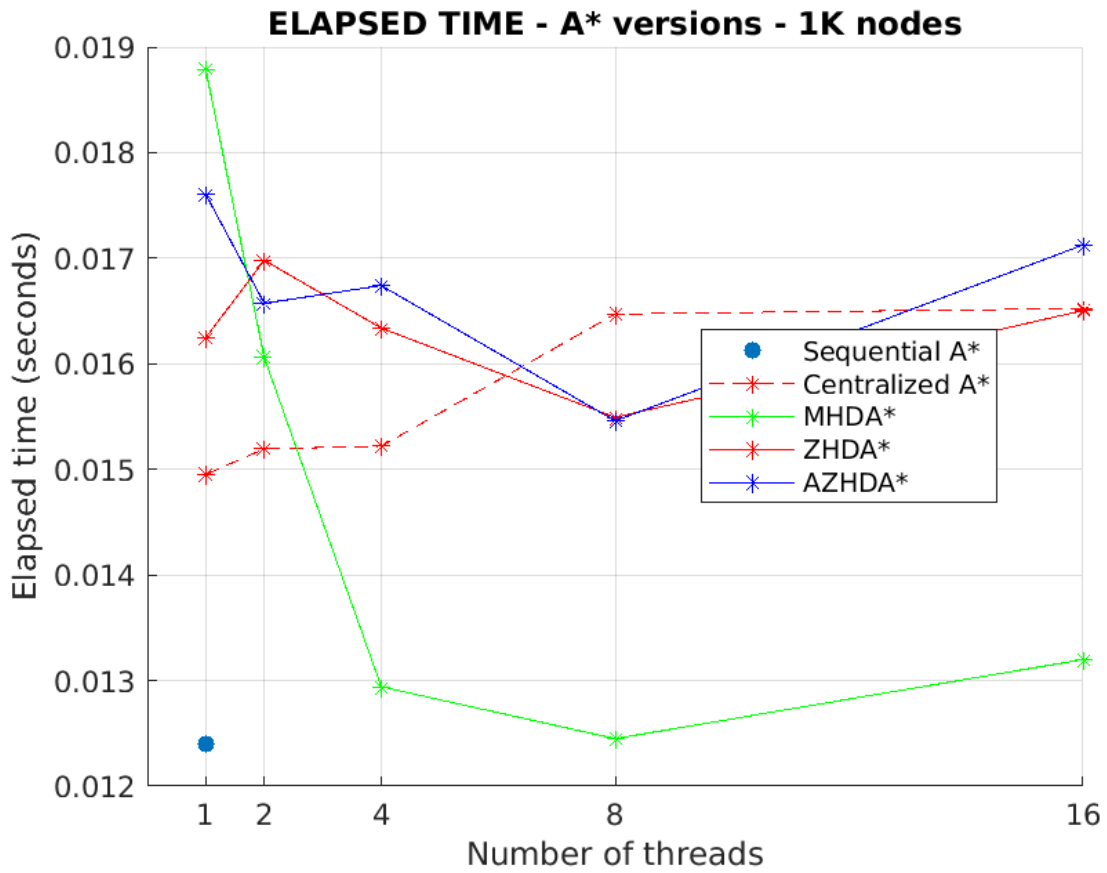
When a thread receives *acc_msg_counter:*

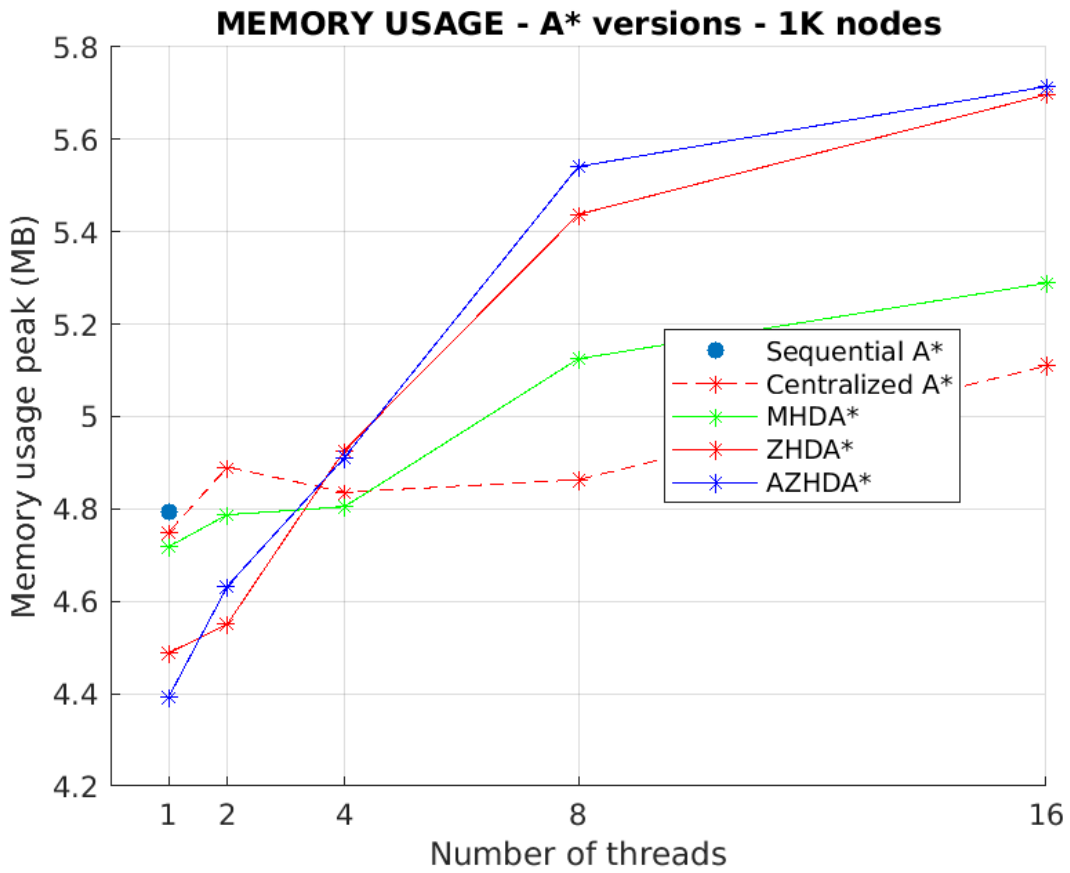
- Add the local counter vector to *acc_msg_counter*.
- Send it to next thread.

If *acc_msg_counter* completes a cycle of all threads being full of 0s for the whole time then the A* can terminate, otherwise continue.

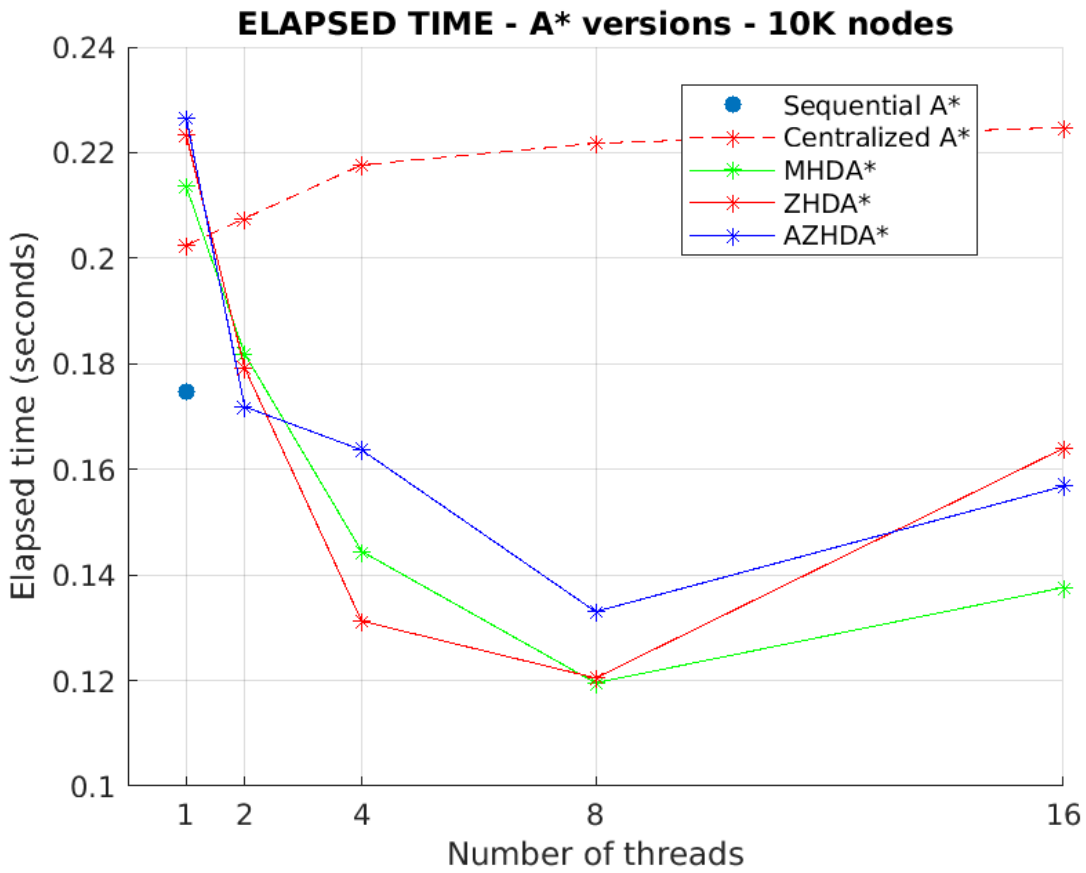
A* PERFORMANCE COMPARISON

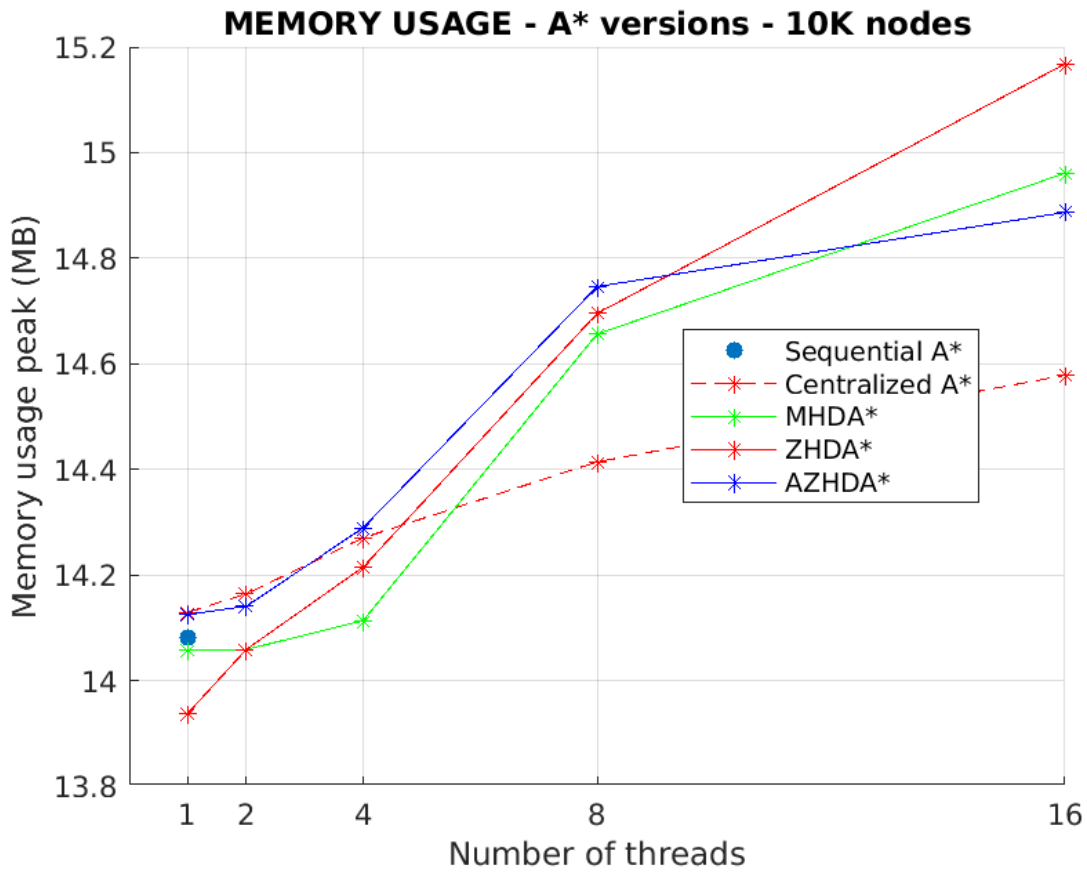
A* PERFORMANCE – 1K NODES



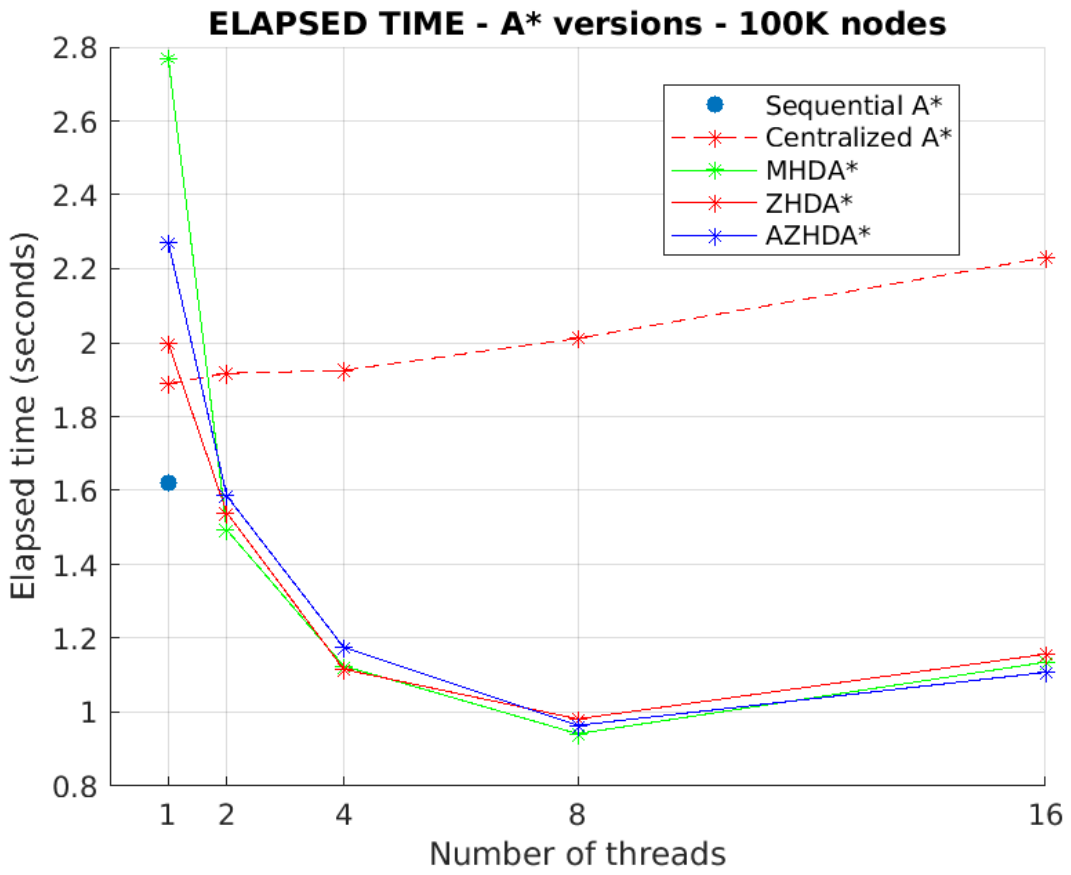


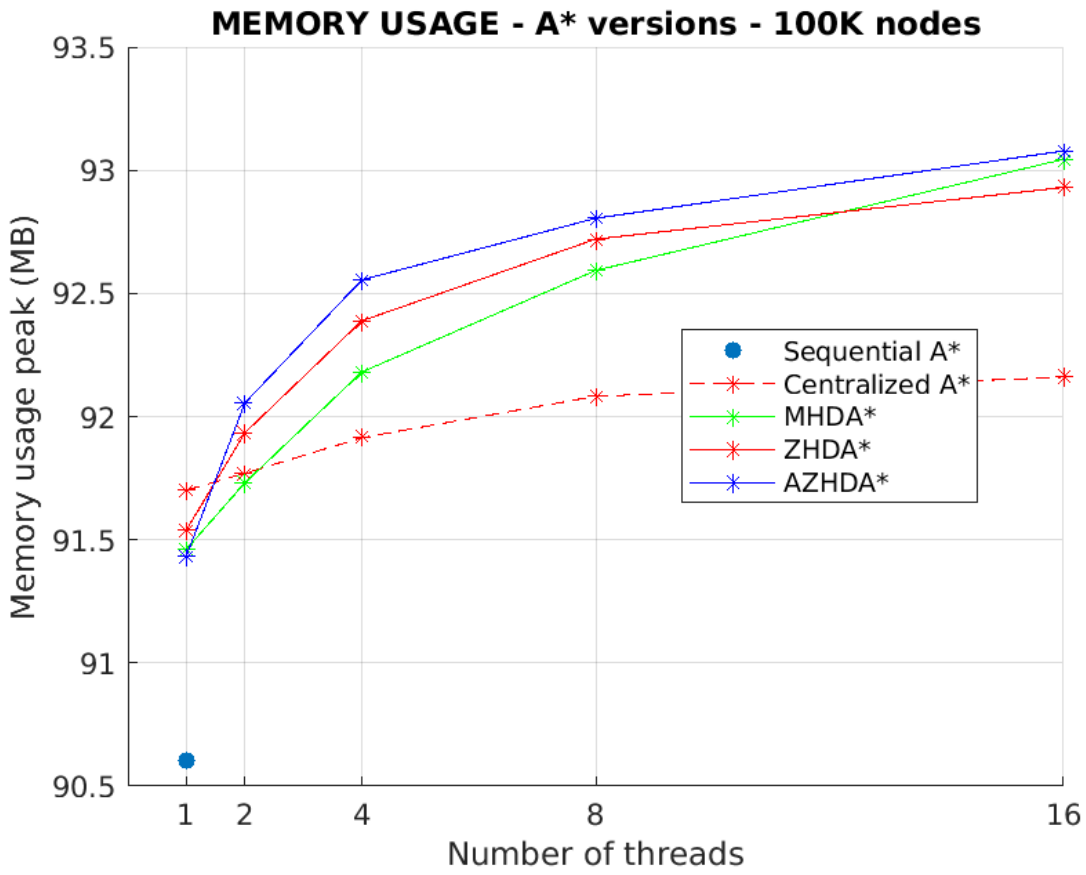
A* PERFORMANCE – 10K NODES



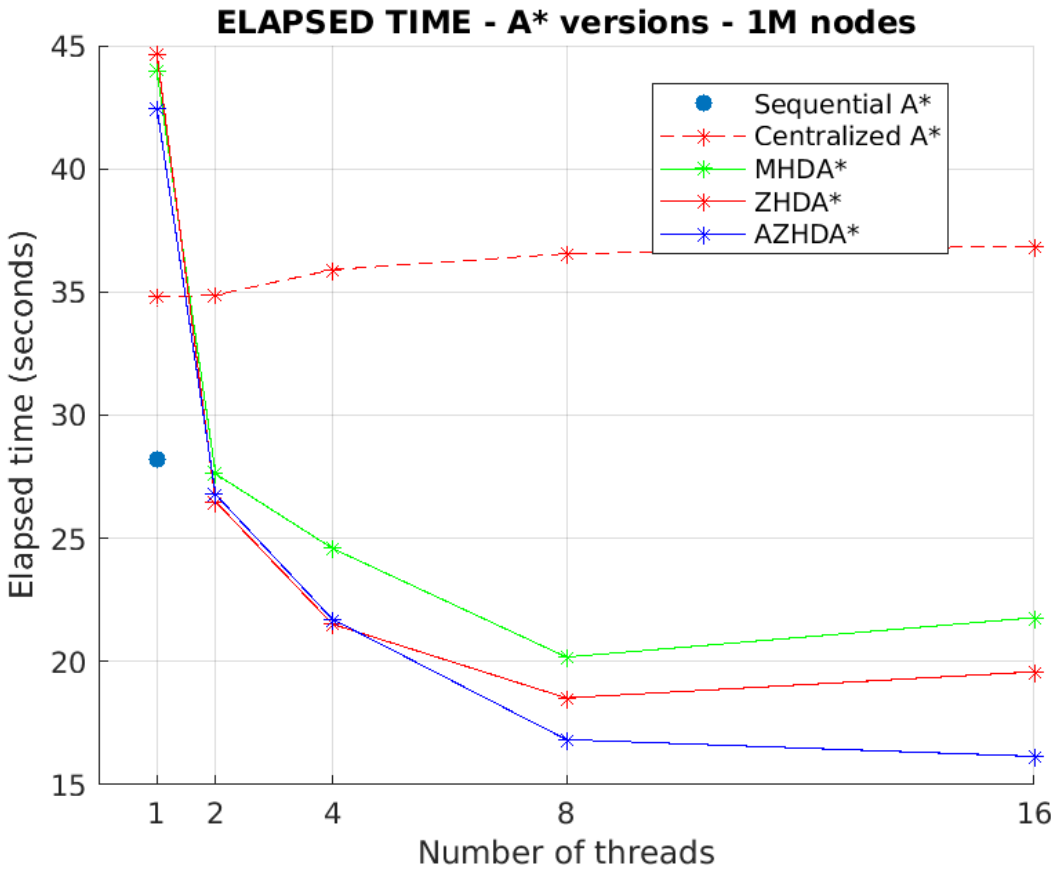


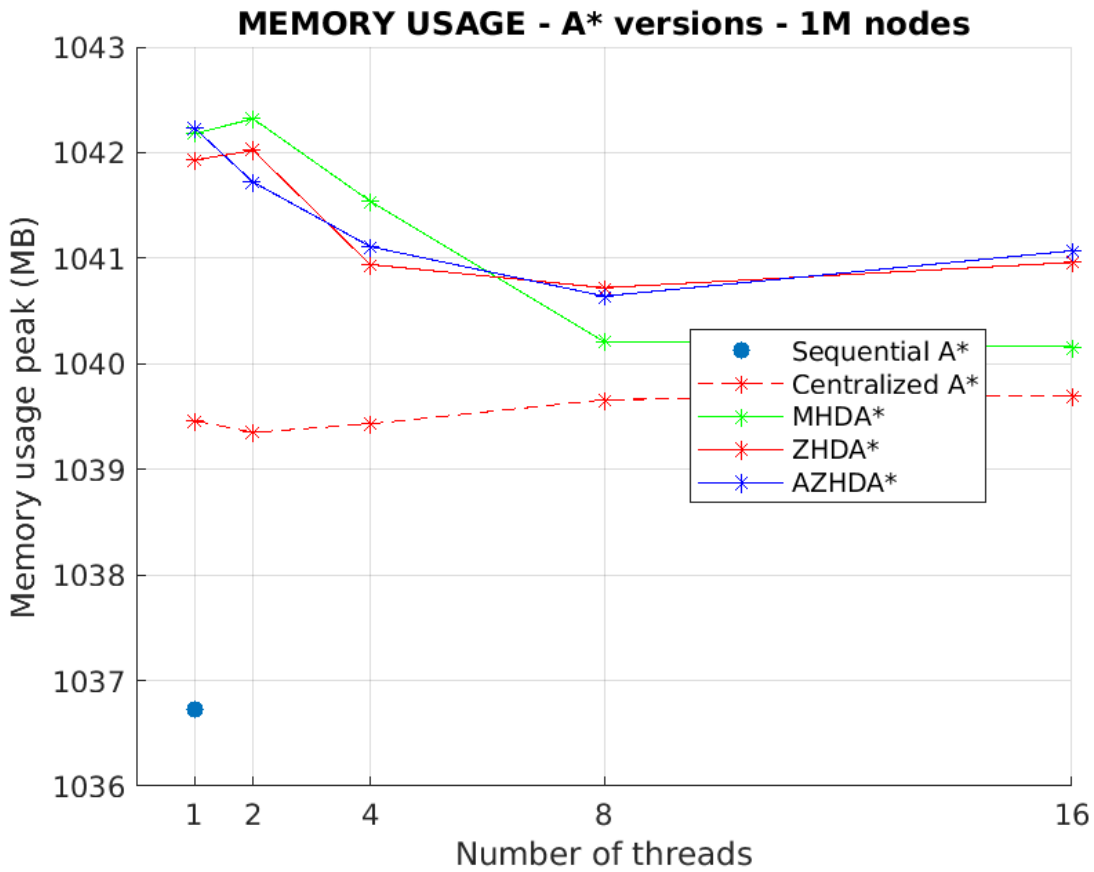
A* PERFORMANCE – 100K NODES



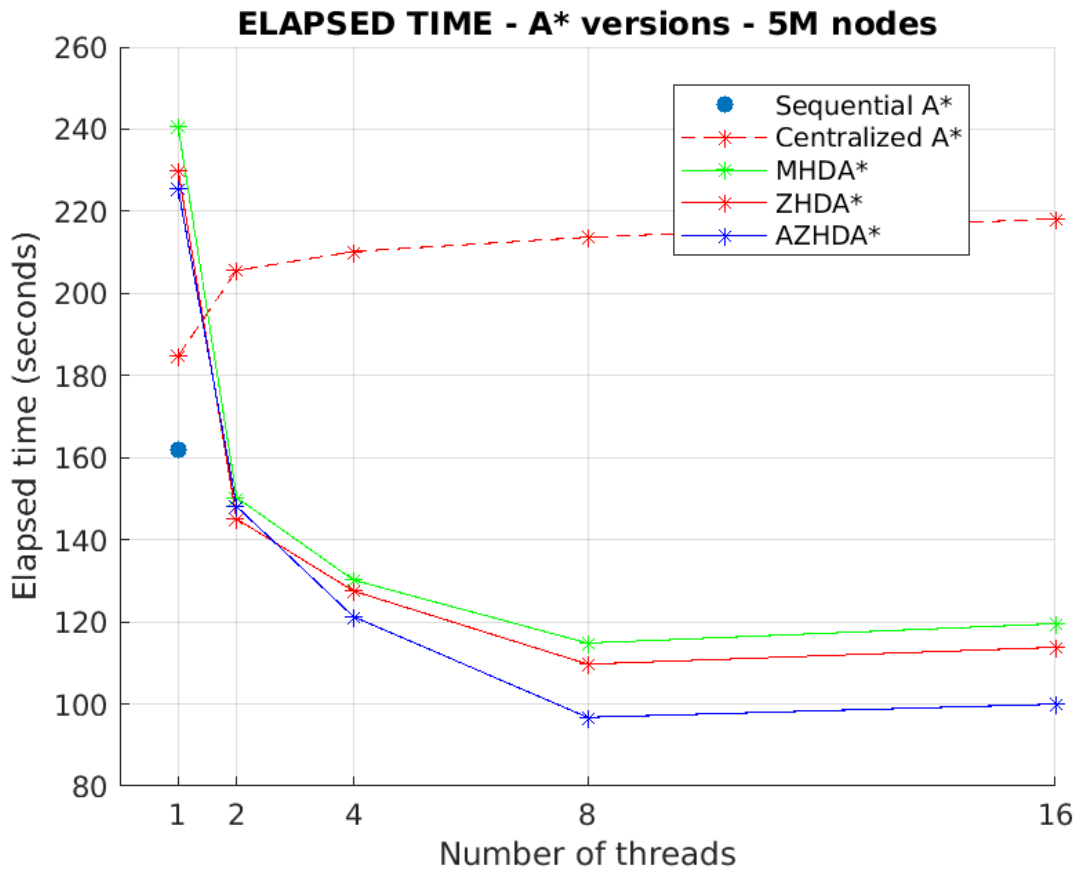


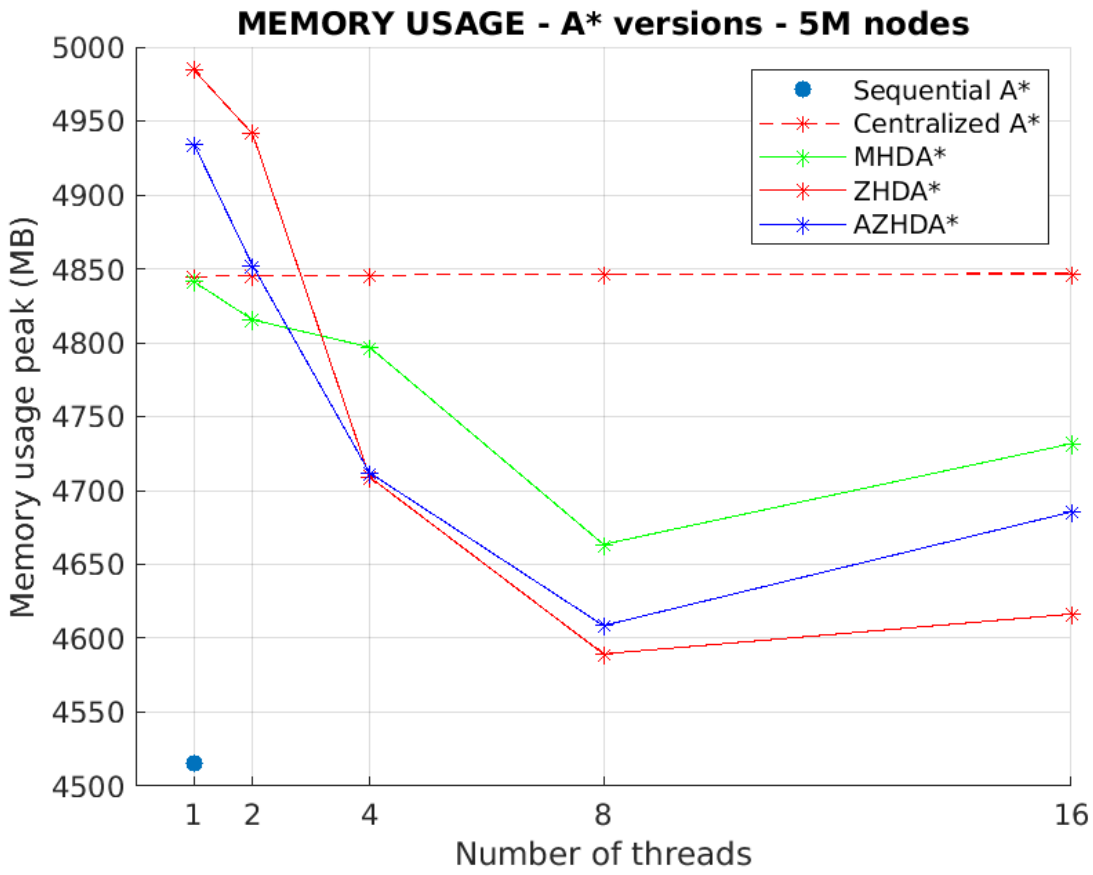
A* PERFORMANCE – 1M NODES





A* PERFORMANCE – 5M NODES





A* PERFORMANCE CONCLUSIONS

Elapsed time:

- **Sequential A*** performs **better** than Centralized and Decentralized A* in **single thread** mode.
  - With small graphs, Centralized and Decentralized A* have too much overhead.
- **Decentralized A* scales very well** with the number of threads.
- **AZHDA*** has **better** performance than ZHDA* and MHDA*
  - The former combines the advantages of the two latter.
- There is a **platoon** at **8 threads**
  - Tests have been conducted on an 8 core CPU.
- **Centralized A* scales bad** with the number of threads.
  - Access to *open* list is **bottleneck**
    - Negligible with respect to expanding a node.

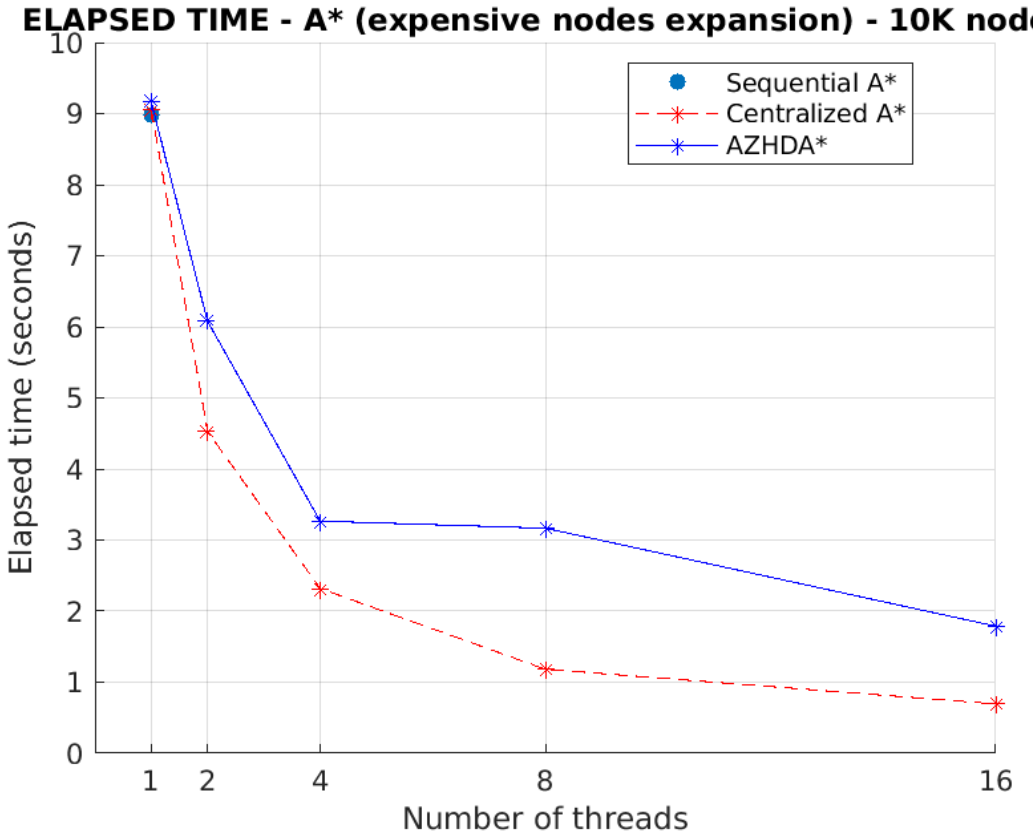
Memory usage:

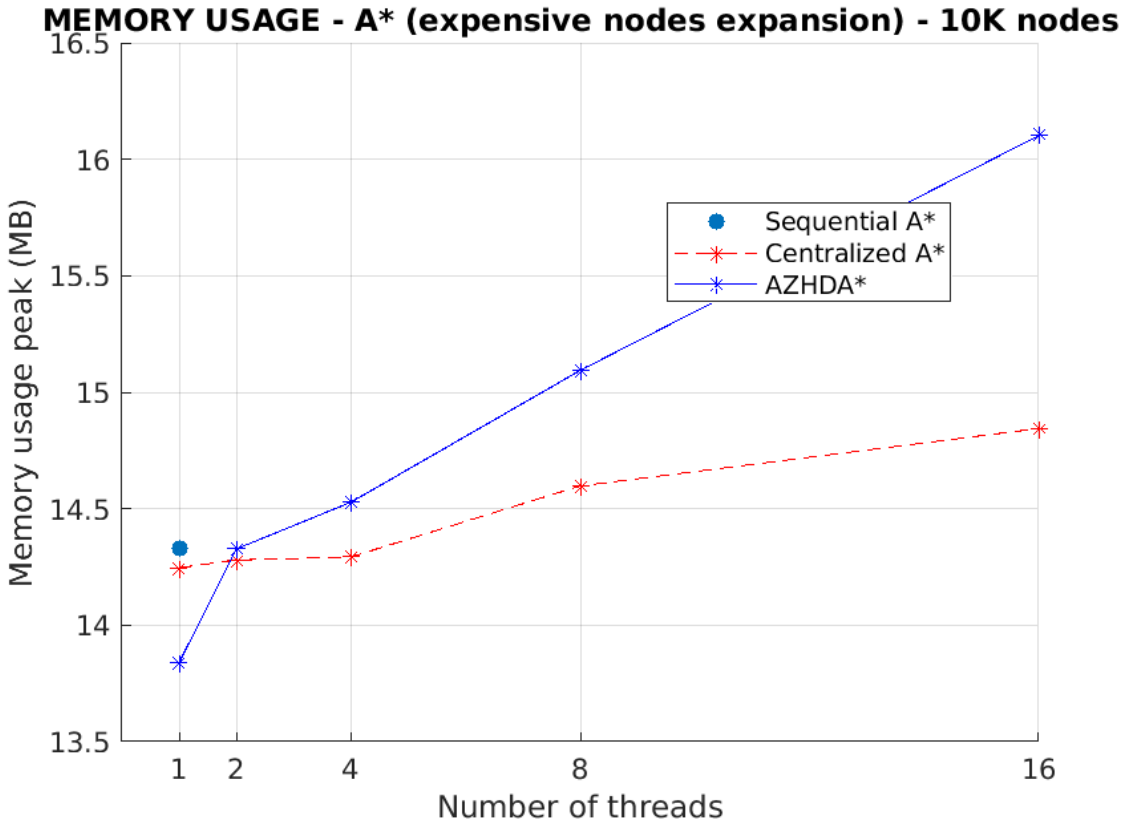
- For **small graphs**, more threads means **more memory**.
  - Memory needed for actual A* is little.
  - Adding a new thread has a non-negligible impact.
- For **big graphs**, more threads means **less memory**.
  - Less execution time.

A* PERFORMANCE CONCLUSIONS

Let’s try to compare the A* versions with a **1 millisecond delay** in every **node expansion**...

A* PERFORMANCE CONCLUSIONS





No more bottleneck of *open* list access for Centralized A*