Politecnico di Torino

System and Device Programming

The A* Path-Planning Algorithm

Authors:
Cavelli Rosario Francesco
Cerra Michele
Colotti Manuel Enrique

September 7, 2022

Contents

1 Introduction

A* is a widely used pathfinding algorithm for traversing Graphs; it extends Dijkstra's algorithm by introducing the concept of **Heuristic Function** which plays a key role in making more optimal decisions at each step.

The project was developed in *C* on a *Unix-like* architecture and implements several versions of this algorithm, both *sequential* and *parallel* ones, with the goal of testing one against the other and getting a final result that summarizes their overall performances.

2 Input file

In this section we describe how graph files are organized, how they have been generated and eventually how they are loaded at execution time. Since we are dealing with graphs made of up to 25 millions vertices, we opted for input files in binary format in order to store information in a more compact and efficient way.

2.1 File Structure

Each file is organized in 3 main parts.

2.1.1 Part 1: number of vertices

The first part corresponds to the first 4 bytes of each file: they represent an integer describing the number of vertices a graph is made of. This number is then used while initializing the Graph data structure.

2.1.2 Part 2: nodes list

The second part is a list of all nodes: each node (or vertex) is totally described by its coordinates. Indeed, for each vertex, we stored only 2 float numbers (4 bytes each) representing both a pair of Cartesian coordinates, by means of normalized x and normalized y for 2D-graphs, or Geographical coordinates, by means of latitude and longitude. The id of a vertex, that is the integer representing it, is not explicitly stored as and additional field: it can be deduced implicitly considering the position of a vertex within the list. Following this logic, the first vertex in the list has id equal to 0, the second one has id equal to 1, and so on.

2.1.3 Part 3: edges list

The third and last part is a list of all edges that starts immediately after the nodes list and finishes with the end of the file. Each edge is identified by means of 3 fields: 2 integers (4B each) and 1 float (4B). The first int is the id of the edge's precursor, the second int is the id of edge's successor, and the float number represents the weight.

2.2 File Generation

Since binary graph files we are using have a specific format, we implemented an additional feature that allows us to obtain binary files of the format described in the previous section by merging 2 ASCII files containing all information. The C code can be found following the path *"/src/parseGraphFile/parseGraphFile.c"*.
The algorithm used to implement this feature simply loops over all subdirectories of the *"files"* folder (that can be found in the same folder as the .c file). Each subfolder has a name made of 2 fields: a number on 2 digits and a set of 3 characters recalling the graph contained in it.
For the algorithm to work properly, each folder must contains at least 2 ASCII files with a specific name format. The first file, with title *"YYY_coordinates.txt"*, where *"YYY"* is the same set of 3 characters of the folder's name, must contain the list of coordinates of each vertex. Each vertex must be stored in the format of 1 int and 2 floats: first int is the vertex's id, and 2 floats represent vertex's coordinates. The

second one, with title *"YYY_distance.txt"*, must contain a list of all edges; each edge must be identified by means of a first integer indicating the number of the edge, then 2 integers indicating the identifiers of the 2 nodes the edge links, and at the end a float for the edge's weight representing the distance between 2 nodes. An additional ASCII file called *"YYY_time.txt"* can be also used: it contains the list of all edges where the weight is expressed as time in milliseconds spent to reach one node starting from another.
For each subfolder in *"files"*, the algorithm starts two threads: one for creating a binary graph file with weights represented by distances, and the other for creating a binary graph files with weights represented by times. Each thread retrieves the paths of the source files (that are ones containing information about vertices and edges), creates the final binary file and then starts merging the file containing the nodes list with the one containing the edges list. Before start reading the nodes list, since we don't know a priori the number of vertices, the algorithm skips first 4 bytes of the binary file, and after having read all vertices it writes back the correct number.

2.3 File Loading

We created 2 different version of the function that is in charge of loading a graph file: one complete this task in a **sequential** fashion, the other in a **parallel** one.
Both of them perform some preliminary operations:

- Open the binary graph file
- Read the number of vertices from the input file
- Initialize the Graph data structure

After these common operations, the 2 functions perform different operations that are described more in details in the following sections.

2.3.1 Sequential Loading

After having initialized the graph, this function reads all information about graph's vertices and edges in a sequential fashion.
More in details we have 2 loops: the first one is a for-loop that is in charge of reading all vertices, while the second one is a while-loop that starts immediately after the previous and ends when the end of file is reached.
The reading of all vertices is performed using read binary operation of the POSIX standard library and a Vert structure (declared in Graph.h) containing 2 floats that is used to describe a vertex. Each time a vertex is correctly read from the input file, it is inserted in the Graph's Symbol Table by calling *STinsert* function.
The reading of all edges is performed in a similar way using the Edge structure (declared in Graph.h) that contains 2 int for the vertices' identifiers and 1 float for the weight. Every time an edge is correctly read, it is inserted in the adjacency list of its starting node by calling *GRAPHinsertE* function.
After all vertices and all nodes have been correctly read, the input file is closed and the function returns.

2.3.2 Parallel Loading

The parallel loading function requires to specify the number of threads we want to use. The parallelization divide threads in 2 groups: who reads the nodes and who reads the edges, then both of the groups work in a similar way. There is a global variable that points to the next line to read, the first thread who takes that position reads it.
The reading is done by *lseek* to the correct "line" to read, this thanks also to the fact that each thread has opened its own file descriptor.
Therefore, there are two mutex that protect the two variable for the next node and the next edge. Additionally there are two mutex inside the Symbol Table and the Adjacency List, to avoid race conditions.

3 Design

This section provides the main design choices about the organization of the data structures and the parallelisation. Most of the types we used are ADT and semi-ADT.

3.1 Graph

In some domains the problem graph is sufficiently small to fit into the memory of the computer; in such cases the search graph can be defined explicitly, enumerating all nodes and edges. In many other problems instead, the search graph is very large and for this reason it has to be defined implicitly. In the latter case portions of the graph are generated on demand. In the following we use a explicitly defined graph, taken from these sources: [4], [6].

Given a weighted, directed graph G=(V, E), with a weight function $W : E \rightarrow R$ mapping edges to real-value weights, we used to represent them as *semi-ADT* Graph with an array of *Adjacency lists* instead of an Adjacency Matrix, this is because most of the benchmark-graphs for path finding are sparse with lots of nodes.
Since there is the possibility to load the graph in a parallel way, we added a mutex to protect the adjacency lists during the insertion of edges. The adjacency is implemented with the method of the *sentinel node*: it is a fake node with id = -1 and negative weight that is placed at the end of each adjacency list. The presence of the sentinel node makes recognizing the end of a list easier.

3.1.1 Symbol Table

Nodes in the graph are identified by a number starting from 0, but each node is characterized by a state, each state has different feature. The mapping from node to state is done by a symbol table (ST) of states, in our case the coordinates of the point in the map, that could be (latitude, longitude) or (x, y).

The ST is an ADT with a *dynamic array* of Coordinates that expand itself if the max size is reached with the *power of 2 method*. The array is protected by a mutex during the parallel loading.

3.2 Priority Queue

3.2.1 Introduction

A very important data structure on which we rely for the execution of each variant of A* is the **priority queue**; this data structure is implemented as a **min-heap of Items**, each item contains two values identifying: the index of the node represented by the item, and as priority an estimate of the cost of the whole path that would be obtained by passing by that node (or **Fscore** 4.1 from now on).

The choice of implementing the priority queue as a min-heap on the Fscore stems from the fact that we needed a data structure which would allow us to retrieve the item with the lowest Fscore in the fastest way possible; by using a min-heap we achieve this purpose by accessing that specific item in **O(1)**, although we'll have to take into account the additional costs to keep the structure of the heap that amounts to **O(n)**.

3.2.2 Search methods for the Priority Queue

Searching for an item within the priority queue is a key functionality required for proper efficient functioning of the algorithm; this observation led us to develop three different search implementations that will extensively be explained here.

- **Linear Search:** Scans each item of the heap until the desired one is found. With a complexity of O(n) it may slow down execution when amount of nodes and searches to be performed are high; the main advantage is that it doesn't require any additional data structure since the scan is made directly on the heap.

- **Constant Search:** This type of search is extremely fast as it allows to find an element inside the heap in constant time. Although its drawback is that an additional data structure containing as many integers as the total number of nodes is required. This supplementary data structure links the index of each node with it's position inside the heap in a way that knowing the index of a node guarantees to know it's position within the heap.

- **Parallel Search:** The last type of search implemented was specifically designed for the sequential version of A* and it aims at exploiting thread-level parallelism in order to speed-up searches within a heap.

  When the heap is firstly initialized, as many threads as the number of cores available in the machine are started, each of them will then wait for a search to be scheduled. When a search is requested each thread starts exploring for the desired element inside a sub-set of the heap; the first thread that finds it, notifies the main thread and goes back waiting followed by the other threads.

  Synchronization and work scheduling has been attained using one master thread in addition to the others (slaves) and three *pthread barrier*. The first one, called *masterBarrier*, is initialized with a value of 2 and it is used by the master to wait the completion of the search: the master, after having assigned a job to all slaves, waits on this barrier and it can continue only after one of the slave-threads perform a wait on the same barrier. The second barrier is *inBarrier*, and it is initialized at the number of slaves plus 1: in this way, all slave-threads are stuck on it waiting for a job to be scheduled by the master, and only when the latter will perform the *pthread_barrier_wait* all slaves will be able to resume execution. The last barrier is *outBarrier*: it is initialized at the number of slave-threads, such that when all threads end the search-job they wait on this barrier before looping again.

  Although we were expecting with this last approach an improvement in search performances, we acknowledged that synchronization overhead strongly affects overall performances; nevertheless we decided to keep this approach as an additional benchmark with respect to the two other previously explained techniques.

3.3 Queue

The implementation of Decentralized A* requires a data structure for the sending of the newly generated state/node to the processor/thread that owns it (that is the one that is in charge of exploring it). The sending must be *asynchronous* and *non-blocking*, in order to avoid synchronization overhead. For this reason we designed an ADT Queue FIFO, a *half-duplex linked list queue* of HItems with pointers to head and tail. We chose a linked list in order to avoid the problem of a full queue, even though this could lead to a raise of the synchronization overhead.

From the implementation of HDA* we know that only one thread can extract a node and only one thread can insert a new node simultaneously, as there are *numThread-1* queues for each thread that lead to a fully meshed net. Therefore, the problem of synchronization is reduced only when there are an insertion and an extraction at the same time. The goal was to make these two operation independent, but when the queue has only 0 or 1 elements, the head and tail points to the same node.

We implemented a solution in which there are two *mutexes* to protect head and tail pointers respectively, and in order to reduce the number of locks, we introduced a sentinel in the head of the queue; this allowed us to lock during insertions only the tail and not the head, while the extraction can lock the tail only if there is an element in the queue, otherwise the two operation are fully independent.

3.3.1 Multiple queues

In the implementation of HDA* a thread has a queue for each other thread, for this reason we implemented special version of Extract and Empty.

The function *QUEUEsAreEmpty()*, receives an array of queues, and returns true if some of the queues is not empty and returns also the first queue not empty if exists. While the function *QUEUEsHeadExtract()*, extracts the head of one of the queues not empty, in order to save some time it also receives the index of a queue for which a scan is wanted.

3.3.2 HItem

The nodes of the linked list are *semi-ADT HItem* (declared in Item.h) with all the information needed for the communication between the threads useful for HDA* algorithm.

4 Implementation

This section goal is to explain in detail how we proceeded implementing different versions of A* exploiting the previously introduced Design. We mainly developed three versions of the algorithm, one sequential and two concurrent; the latter ones can further be divided in two distinct versions as it will be explained in sections 4.4 and 4.5

4.1 Glossary of Data Structures

In the following list, some concepts and data structures that have been used in A*'s implementations will be briefly discussed.

- **Concepts**
  - **Gscore**: Gscore of a node represents the cost to reach that node from the source of the path
  - **Hscore**: The Hscore is a measure obtained by means of an Heuristic function 4.2 that estimates the cost to reach the destination from a specific node.
  - **FScore**: Adding up the Gscore and the HScore we obtain the Fscore of a node, which represents an estimate of the cost of the overall path that we would obtain by passing by that node.
  - **nMsgSnt/Rcv**: total number of messages sent and received by the slave threads, useful for the implementation of the termination condition with the Mattern's method.
- **Data Structures**
  - **OpenSet**: Priority Queue 3.2 containing nodes that are still to be visited.
  - **ClosedSet**: Integer array that keeps track of visited nodes by saving their Fscore in the proper index.
  - **Path**: Integer array that saves for each node it's predecessor; once the end node has been reached it is possible to rebuild the constructed path by visiting the predecessors backwards starting from the end.
  - **HScores**: Array of integers that stores the precomputed Hscores for each node of the graph. This data structure has been introduced to speed-up the computation. Indeed, it could be removed, but every time a new node is analyzed we need to compute its Hscores on the fly. Since the evaluation of the h function may be really heavy, computing this value every time may slow-down the overall performances.
  - **QueueArr/Mat**: Array 1D or 2D of Queues 3.3, used for the communication between master to slave or slave to slave in HDA*.

4.2 Heuristics

A* selects the path that minimizes $f(n) = g(n) + h(n)$.
$g(n)$ is the cost of the best known path from the root to the current node n, while $h(n)$ is an estimation of the cost from n to the goal node. Therefore having an *admissible* heuristic function is crucial to have *optimal results*. If h is an admissible function, then A* using h is admissible, which means that will always return an optimal solution.

A heuristic function h is *admissible* if $h(n) < C^*(n)$, where $C^*(n)$ is the cost of the minimal path from n to the goal, so h is a lower bound on C^*.
The nodes that have been closed could be re-opened if an heuristic is not *consistent*, this leads to a reduction of the *efficiency*. A heuristic function is consistent if $h(n) \leq c(n, n') + h(n')$ for all nodes n and n' such that n' is a successor of n, and $h(target) = 0$ for all target.

In our study, most of the graphs are networks on a map, it means that nodes are coordinates. On a spherical map coordinates are *latitude* and *longitude* while in a Euclidean space they are x and y. We use three different *admissible* heuristic functions for our study:

- *Euclidean distance* Heuristic,

$$h(n_1, n_2) = \sqrt{(n_1.x - n_2.x)^2 + (n_1.y - n_2.y)^2}$$

- *Haversine formula* Heuristic [7], it determines the *great-circle* distance in meters between two points on a sphere given their longitudes and latitudes.

4.3 Sequential A*

Now that all fundamental concepts have been explained it is possible to discuss algorithms implementation more in details, starting from the sequential version of A*. The algorithm starts by visiting the *start* node that is inserted in the path and it's Fscore saved in the closed set. Next step is to visit all neighbours of this node, for each of them a temporary Gscore and Fscore is computed and used for the following comparison:

If the neighbour has already been completed, that is when a node is extracted from the open set, added to the closed set and all its neighbours put in the open set, and it's temporary Gscore is less than it's current one, it means that this vertex may be reached by a path with a lower cost; therefore it is removed from the closed set and added again to the open set; in case the Gscore would have been greater, the algorithm would have just continued with the next neighbour.

Otherwise if this neighbour hasn't already been visited it is searched for it's presence inside the open set: if it's not there it gets immediately inserted, otherwise the same comparison on the Gscore made above is performed again; this time though if a better Gscore is found, the lowest will be substituted in place and the node will remain in the open set with a new priority.

Once these comparisons have been completed, only in cases where the Gscore has improved or the neighbour hadn't already been visited, the path array is updated by inserting as predecessor of the neighbour, the node from which it has been reached.

This process of extracting nodes with the lowest Fscore from the open set and evaluating their neighbours, goes on until the destination node is reached or the open set is empty: if the latter termination condition occurs, it means that a path linking the starting node with the final one doesn't exist.

4.4 Simple Parallel A*

Parallelization of A* search is important because memory usage continuously increases during the run, as it must keep all the expanded nodes in memory in order to guarantee an optimal solution and completeness of the algorithm. In this section we describe the first basic approach to parallelization of A*: the centralized parallelization.

In this approach all the data structures used in Sequential A* are *shared* between threads, thus is necessary to add a *synchronization overhead* to regularize the access to them. We implemented two different version of the algorithm: the first one use a *single mutex* to lock all the nodes of the graph in the closed set during an expansion of a node, while the second one have an *array of mutexes* that lock only the single node that the algorithm is expanding. In both version there is a mutex to protect the *bestCost* shared variable. This variable is used to save and compare the costs found by different threads, and also to prune the algorithm: if a priority of an extracted node is greater than the *bestCost*, the node is skipped. The other shared variable, *path array*, is not protected using an its own mutex: mutually exclusive accesses to this variable are done while expanding a node, so *path* is implicitly protected by the mutex used to synchronize nodes access.

4.4.1 Termination Condition

The termination condition is important in SPA* to let other threads know when to terminate.
For stopping the threads we used a conditional variable (*cv*), a *counter* and a *mutex* that is different depending on the two versions. If the *open Set* is empty, normally, a thread goes to sleep until a new node is inserted in the queue, but if all the threads are sleeping, the last thread will wake up all the other threads and they will exit.

4.4.2 Extraction of a node

The extraction of a node is a *Critical Operation* that must be in a critical section, the same locked for the *conditional variable* of the termination condition, since we are sure to extract a node from the open set. In addition the *extraction* from the open set and the *insertion* in the closed set must be done in *Atomic* way, otherwise another thread will treat an already discovered node as a new.

These operations are easy if they are done with a single mutex (first version) that locks both all the nodes in the closed set and the open set, while in the second version, in which there is a mutex for each node in the closed set, we cannot lock the open set, for this we need another mutex for locking it. But this is a problem because extraction from open set and insertion in the closed set must be atomic.

The solution we adopted in the second version peeks the node will be extracted and try to lock it, if we get the lock, we unlock the open set and do the operations, otherwise we continue, because the node is already locked by another thread. We must use try lock because otherwise would be created a *deadlock*.

4.5 Hash-Based Decentralized A*

SPA* suffers from severe synchronization overhead due to the need of constantly access shared open/closed lists.
In a *decentralized* approach to parallel best-first search, each thread has its own open list. Decentralizing the open-list eliminates the concurrency overhead associated with shared, centralized open list, but *load balancing* becomes necessary.

4.5.1 Architecture

Our implementations is composed by two types of threads:

- The *Master*: it starts the algorithm, by *sending the start node to its owner-thread*. It also check the *termination condition*.

  In addition, depending of the implementation, the master can receive and send messages from and to the slaves, like a telephone exchange.

- The *Slaves*: they are the threads who done the most of the work. Each slave extracts a message from the *message queue(s)* and do the same analysis as the sequential A*. When its message queue is empty (or all its message queues are empty, depending on the implementation), the thread extract

from the open set the next node to *expand*. For each vertex in the adjacency list, it computes the owner-thread through an *hash function*, and send it to the Master o the owner Slave, depending on the implementation.

4.5.2 Implementation

We propose two different solution:

- **With coordinator** (Master): each slave talk only with the master, that is in charge of extracting messages received and forwarding them to proper threads.

  This solution to work properly needs 2 message queues for each thread, leading to have $2 * nThreads$ queues. The *S2M queue* is used by the slave to send messages to the master: every time a node is found in the adjacency list of the vertex being currenctly expanded, the slave-thread crates a new message to send to the master using this queue. The *M2S queue* is used by the master to forward messages to the correct slave-thread: every time the master detects that a message has to be sent to a given slave, it extracts the message from the incoming message queue and insert it into the M2S queue of the owner.

- **Without coordinator**: each slave can talk directly to all other threads. Each slave-thread has 1 message queue for each other thread it can talk to. In this way, every time a slave finds a node in the adjacency list of the vertex currently being expanded, it computes the owner using an hash function and it directly sends the message to it.

  This solution uses $nThreads * (nThreads - 1)$ queues.

4.5.3 Avoid busy waiting

HDA* algorithm must run until the *Termination Condition* is satisfied, but sometimes the message queue or the open set (or both) are empty: in this cases the thread would loop as long as something is put in one of the two queues, leading to a *busy waiting*.

To avoid this we introduced **task counters**.

Each thread (including the Master) has a *semaphore* that counts the be of tasks that it has to do inside the loop: when something is put inside the queue *sem_post()* is called to increase the counter, while when something is taken away from the queue *sem_wait()* is called to decrease the counter. In addition, at the begin of the while loop there is a something like a barrier that stop the thread when the task are terminated (counter == 0).

4.5.4 Termination detection

Termination detection check two different conditions:

- Mattern's method condition [1]: the method counts the number of received and sent messages between slave-threads. This counters are implemented through array in order to avoid the synchronization overhead. In addition the method take in account also the number of the previous check. If the previous value are pR and pS, and the new values are R and S, the condition to be true must be: $pR == S$ **and** $S! = 0$ **and** $pR == R$.

  It means that: the previous value of the received message must be equal to the number of sent message, the number of sent message must be different than 0, otherwise the algorithm is not started yet, and the number of received message must be equal during the two cycles.

- All the slaves must have the task counter to 0: each time a slave-thread has all queues empty, it increments the counter of stopped slaves. When the counter reach the total number of threads, the master is woken up.

4.5.5 Hash functions

The performance of HDA* algorithms depend entirely on the characteristics of the hash function used for assigning nodes to processors. During our experimentation we used different hash functions. *Random hash function* give the owner-thread's id using the following formula: $h(v) = v\%numThreads$. This resulted to be the worst hash function because there is no relation between different threads and the owner-thread.
To increase the overall performances, we had to find a state in the node that can categorise it, in our case we use as state two features (latitude, longitude) or (x, y).

From now on, our state s is now defined as: $s = (x_0, x_1)$, with x_0 = Latitude or x of the vertex, and x_1 = Longitude or y. We decided to represent s inside an integer (4Byte): in which in the 2 MSB will be x_0, and x_1 in the 2 LSB. From here we could apply different hash functions [3]:

- Multiplicative Hashing, with this method has been observed to hash a random key to P slots with almost equal likelihood.

$$A := \frac{\sqrt{5}-1}{2}$$

$$H(s) := \lfloor numThread * (k(s) * A - \lfloor k(s) * A \rfloor) \rfloor$$

- Zobrist Hashing: it is a *Tabulation Hashing*, this algorithm has proven to be very fast and of high quality.

$$Z(s) := R[x_0]\ xor\ \cdots\ xor\ R[x_n]$$

  With s (state) defined as before and R a table containing preinitialized random bit strings.

- Abstract States Zobrist Hashing: the abstract function is applied to the state

$$A(s) = (A(x_0), \cdots, A(x_1))$$

$$Z(s) = R[A(x_0)]\ xor\ \cdots\ xor\ R[A(x_n)]$$

- Abstract Feature Zobrist Hashing

$$Z(s) = A(R[x_0])\ xor\ \cdots\ xor\ A(R[x_n])$$

The difference between the two latest is where the abstract function is applied.

4.5.6 Abstract function

Abstract function is a feature projection function, which is a many-to-one mapping from each raw feature to an abstract feature.

In our main cases we use Latitude and Longitude feature, for this reason we use a feature projection function that map the points on a section of the map through an approximation of the *Open Location Code*. Open Location Code[8] define an area in which a point could be, thanks that we assign more points in a defined area, and assign that area to a thread. More the approximation is high bigger will be the area.

5 Experimental Results

5.1 Timer

To know the time elapsed for the algorithms we use an *ADT Timer* that can record the time. But first we have to understand what type of time we have to know. There are 2 kind of times: Wall time and CPU time.

- *Wall Time*: it is simply the total time elapsed during the measurement.
- *CPU Time*: it refers to the time the CPU was busy processing the program's instructions. The time spent waiting for things is not included.

In all of our experimental result we use a Wall time, because we want to measure also the scheduling time between the threads.

Our solution uses *<sys/time.h>* and *gettimeofday()*, because it returns the *struct timeval* that can reach an accuracy of microsecond. Microseconds are useful for comparing times for small graph or brief paths

5.1.1 Timer for parallel algorithms

The ADT Timer works well even in *multi-threading* codes, during the initialization it receives the number of threads, and when the algorithm begin, the time starts when the first thread active the timer, and end when the last thread stop the timer. Everything works thanks a *mutex* and a *counter*.

5.2 Test

Our goal in this section is to do some tests to compare our different solution for path-finding with A*. In order to automate this process we wrote a new main "Test". This ADT random generate 10 different path and it loops on them trying all the algorithms, for each of them it take some information about time, memory and other statistics. For each it prints:

- Name of the algorithm
- Number of thread used for that algorithm
- The total cost of the path
- Number of hops of the path
- Total time spent from when the algorithm function is called and the return of the function, here is included the time to setup all the structures
- Algorithm time spent, here is only the time from the begin of the algorithm to the end, setup isn't included
- Speed Up for both times, is the division factor of the actual time repect the time of sequential A*.

$$SpeedUp := \frac{Time\_Sequential}{Time\_parallel}$$

- Number of expanded nodes, it is the number of discovered nodes during the algorithm
- Number of extracted nodes, they are the nodes that are extracted from the open Set. For parallel this number is grater than number of expanded
- Communication Overhead (CO) refers to the cost of exchanging information between threads

$$CO := \frac{\#nodes\_sent\_to\_other\_thread}{\#nodes\_generated}$$

- Search Overhead (SO) is the rapport between the number of nodes expanded in parallel way and sequential way. SO can arise due to inefficient load balance.

$$SO := \frac{\#nodes\_expanded\_in\_parallel}{\#nodes\_expanded\_in\_sequential}$$

- Load Balance (LB) denotes if there are some thread which are assigned more nodes than the others.

$$LB := \frac{Max\_\#nodes\_assigned\_to\_a\_thread}{Avg\_\#nodes\_assigned\_to\_a\_thread}$$

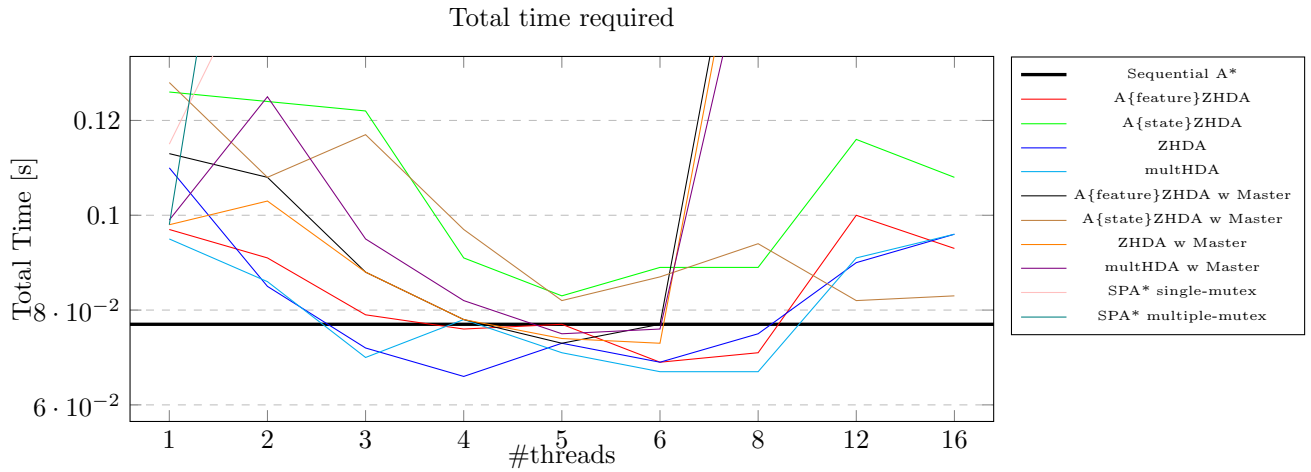
- Memory Size: it is the memory allocated by the algorithm in Byte (the memory of the graph excluded).

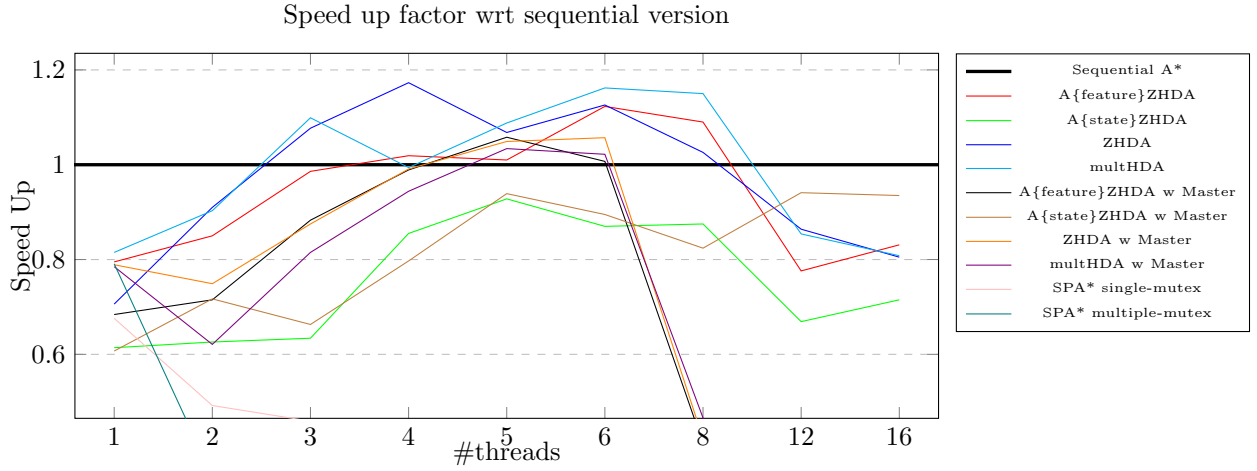
6 Plots

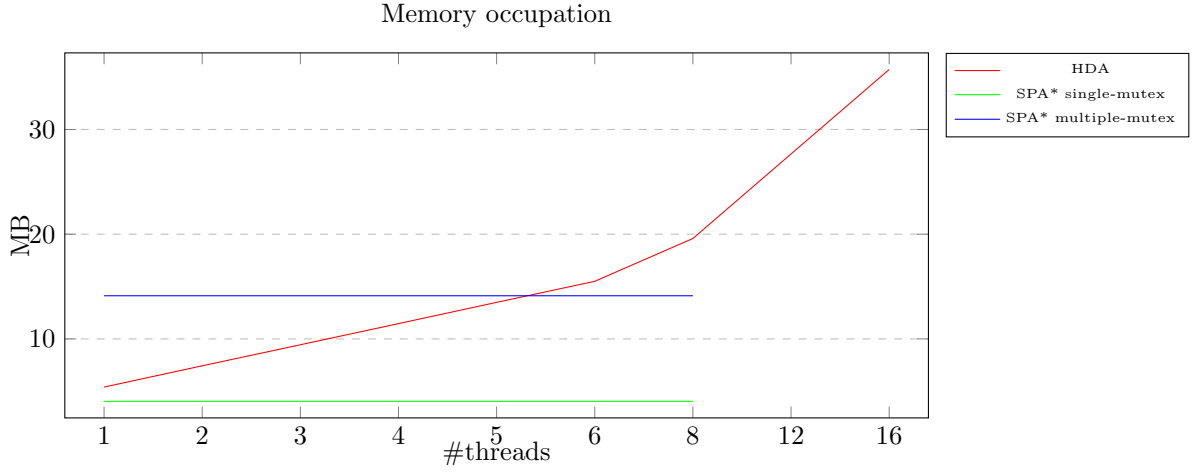
In this section are reported a set of 3 plots for each benchmark-graph. For each of the latter we conducted a preliminary analysis to find a proper path length. We ran 100 tests for each graph considering random starting and ending points to compute the average path length. At the end we used this average to build an interval to which the length of the test path belongs. Average path length and Interval are reported before each benchmarked graph.

6.1 New York

New York City road network with 264,346 nodes and 733,846 edges. Average path length: 400 hops. Interval: [200;600].

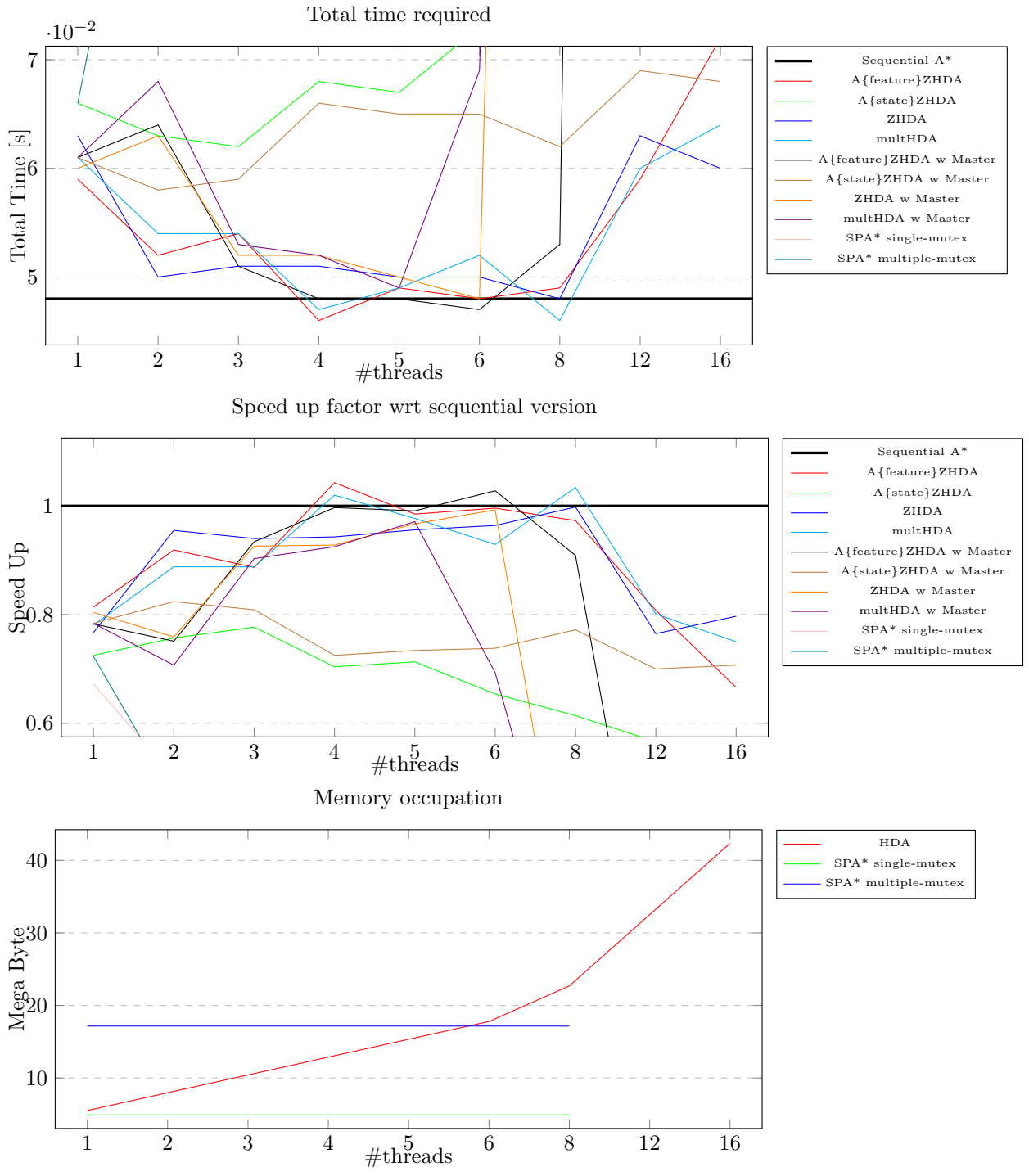






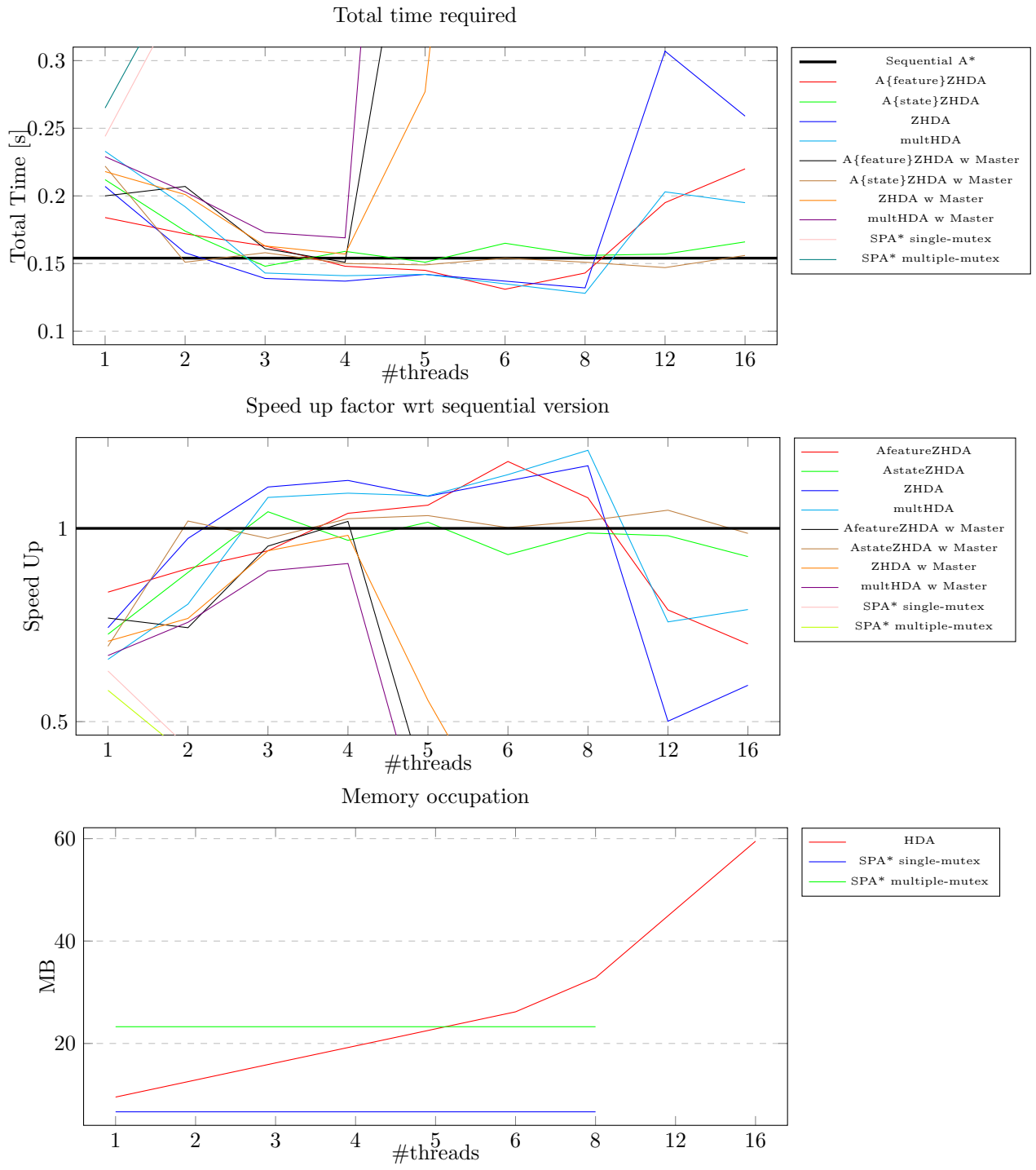
6.2 San Francisco Bay Area

San Francisco Bay Area road network with 321,270 nodes and 800,172 edges. Average path length: 500 hops. Interval: [300;800].



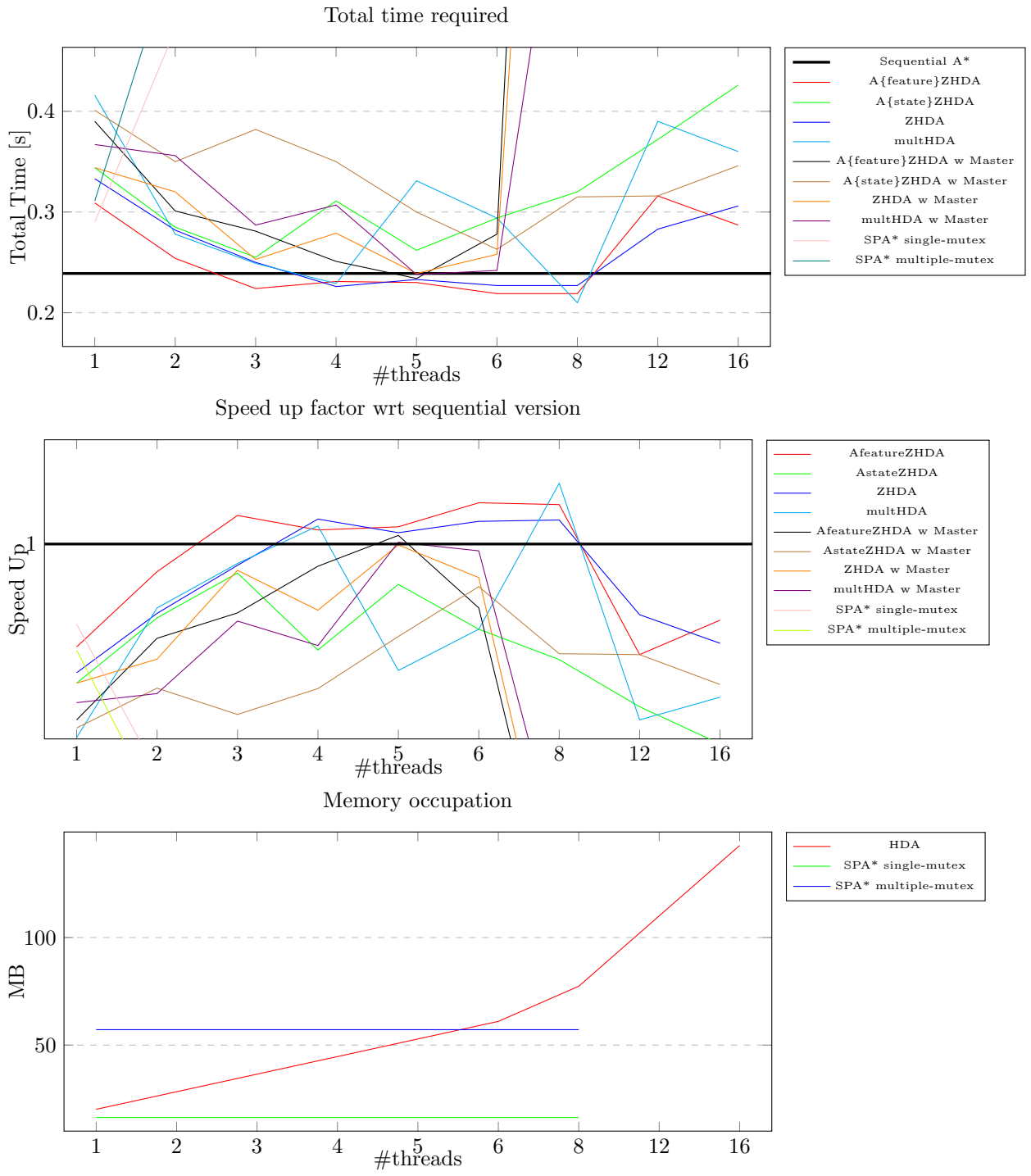
6.3 Colorado

Colorado road network with 435,666 nodes and 1,057,066 edges. Average path length: 700 hops. Interval: [500;1000].



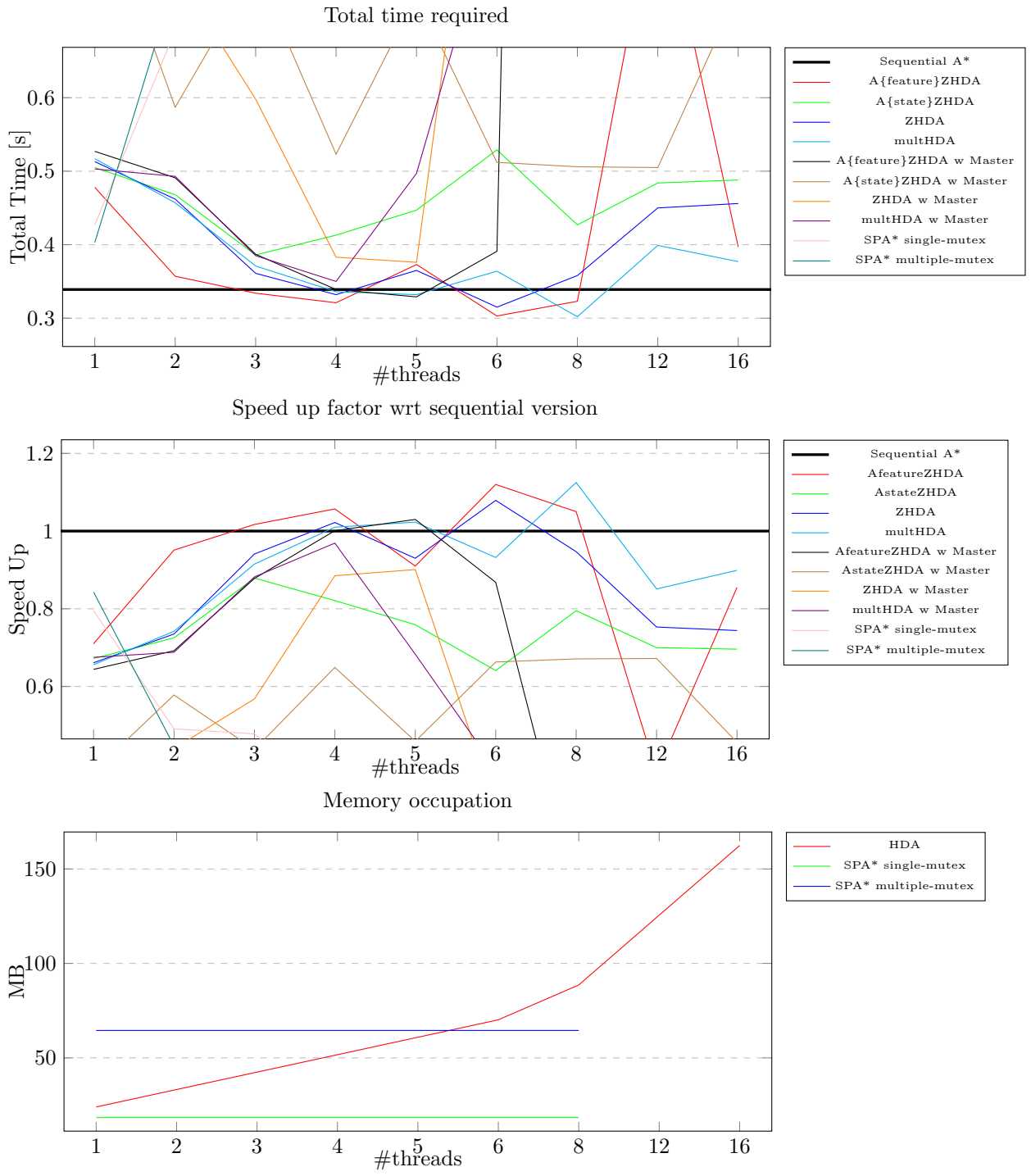
6.4 Florida

Florida road network with 1,070,376 nodes and 2,712,798 edges. Average path length: 1100 hops. Interval: [800;1200].



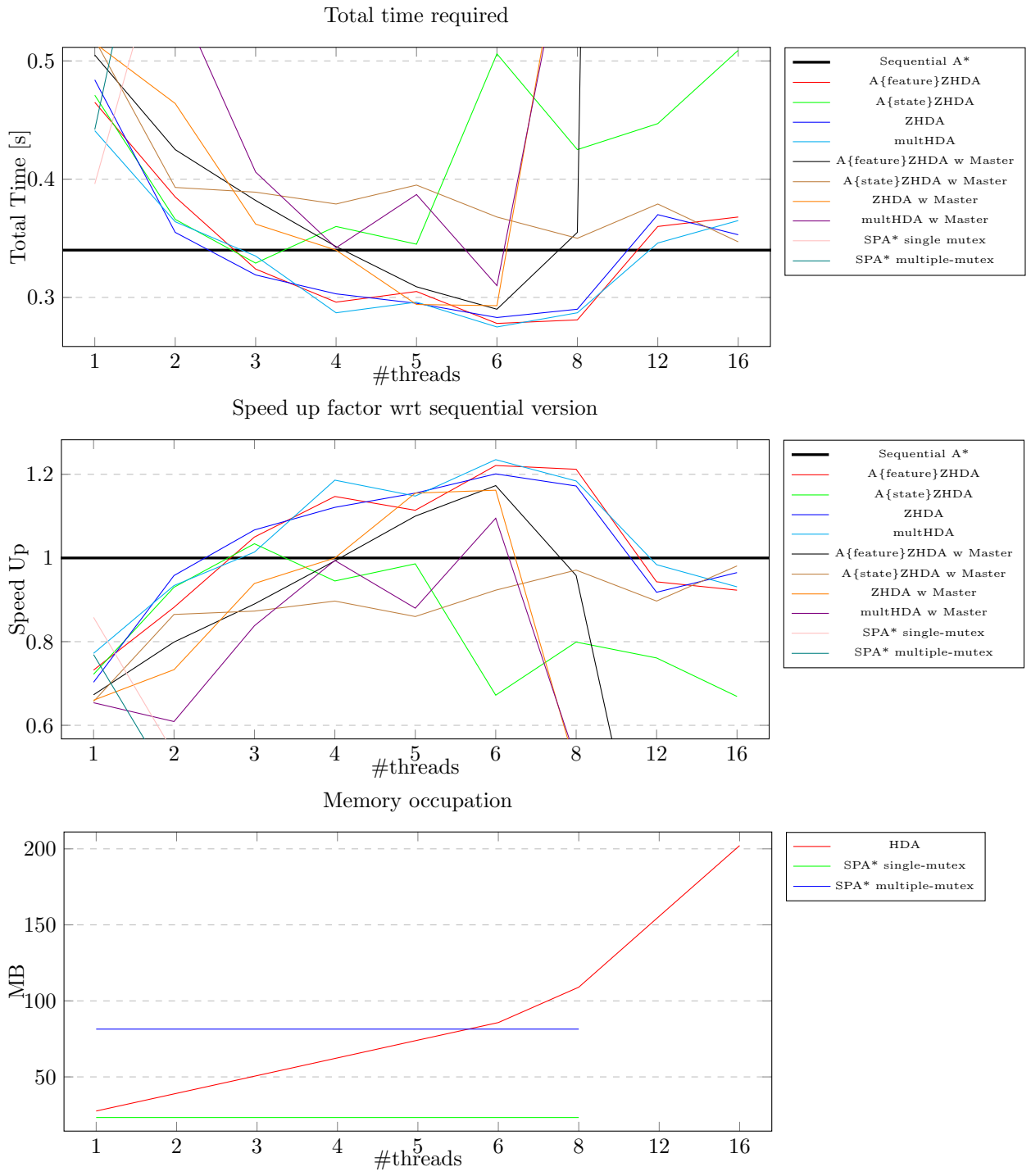
6.5 Northwest USA

Northwest USA road network with 1,207,945 nodes and 2,840,208 edges. Average path length: 1100 hops. Interval: [800;1200].



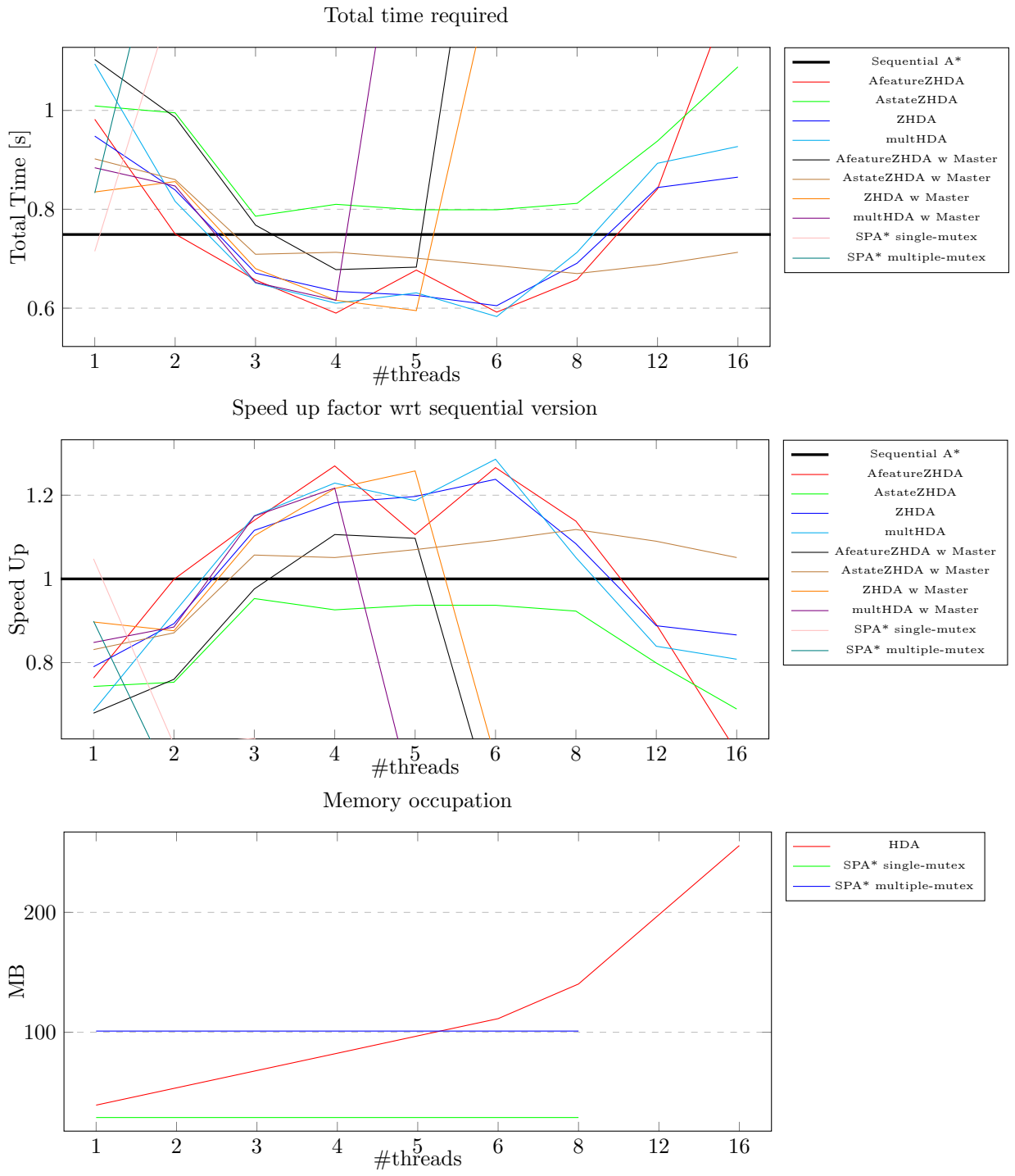
6.6 Northeast USA

Northeast USA road network with 1,524,453 nodes and 3,897,636 edges. Average path length: 1200 hops. Interval: [900;1300].



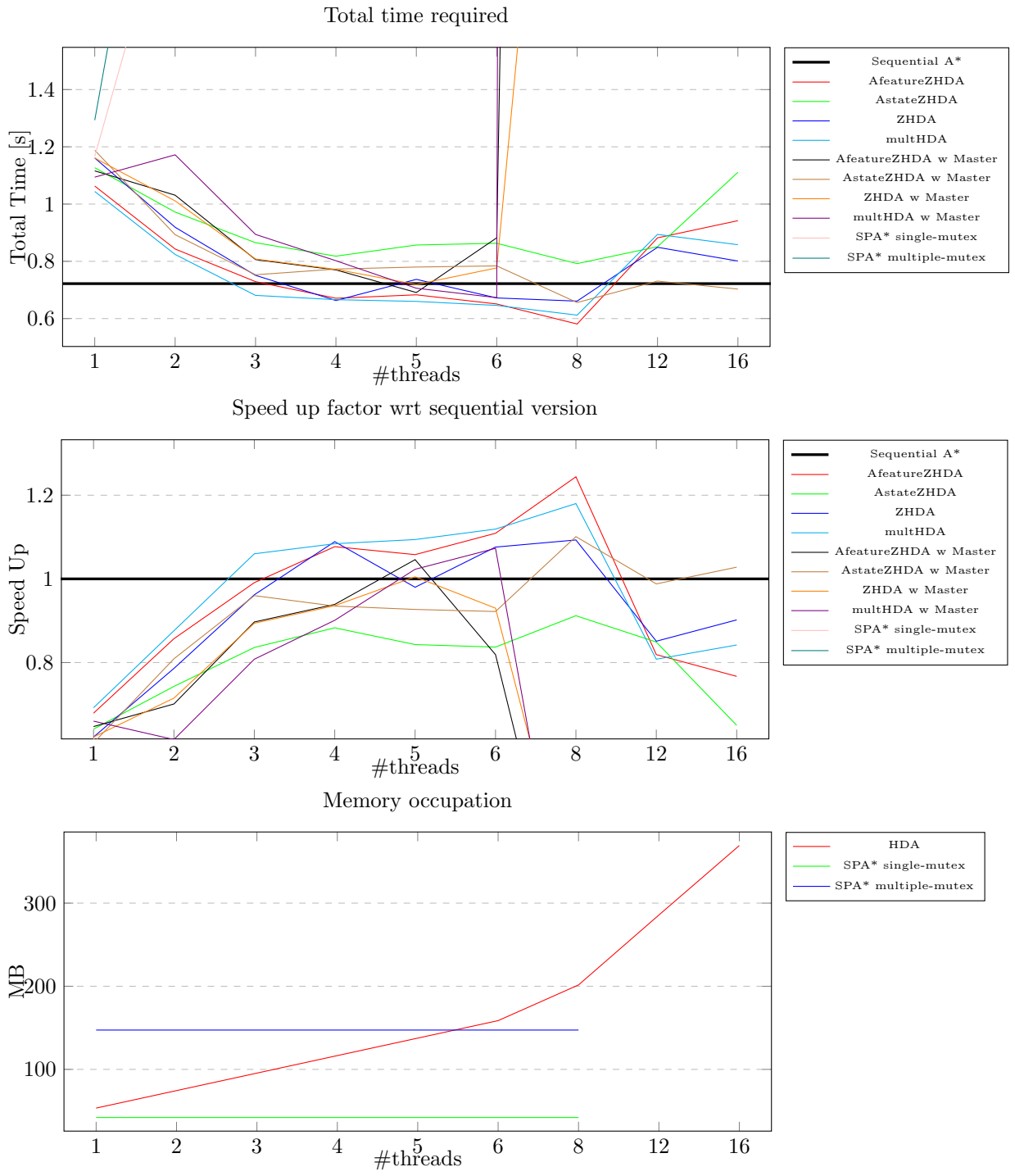
6.7 California and Nevada

California and Nevada road network with 1,890,815 nodes and 4,657,742 edges. Average path length: 1200 hops. Interval: [900;1300].



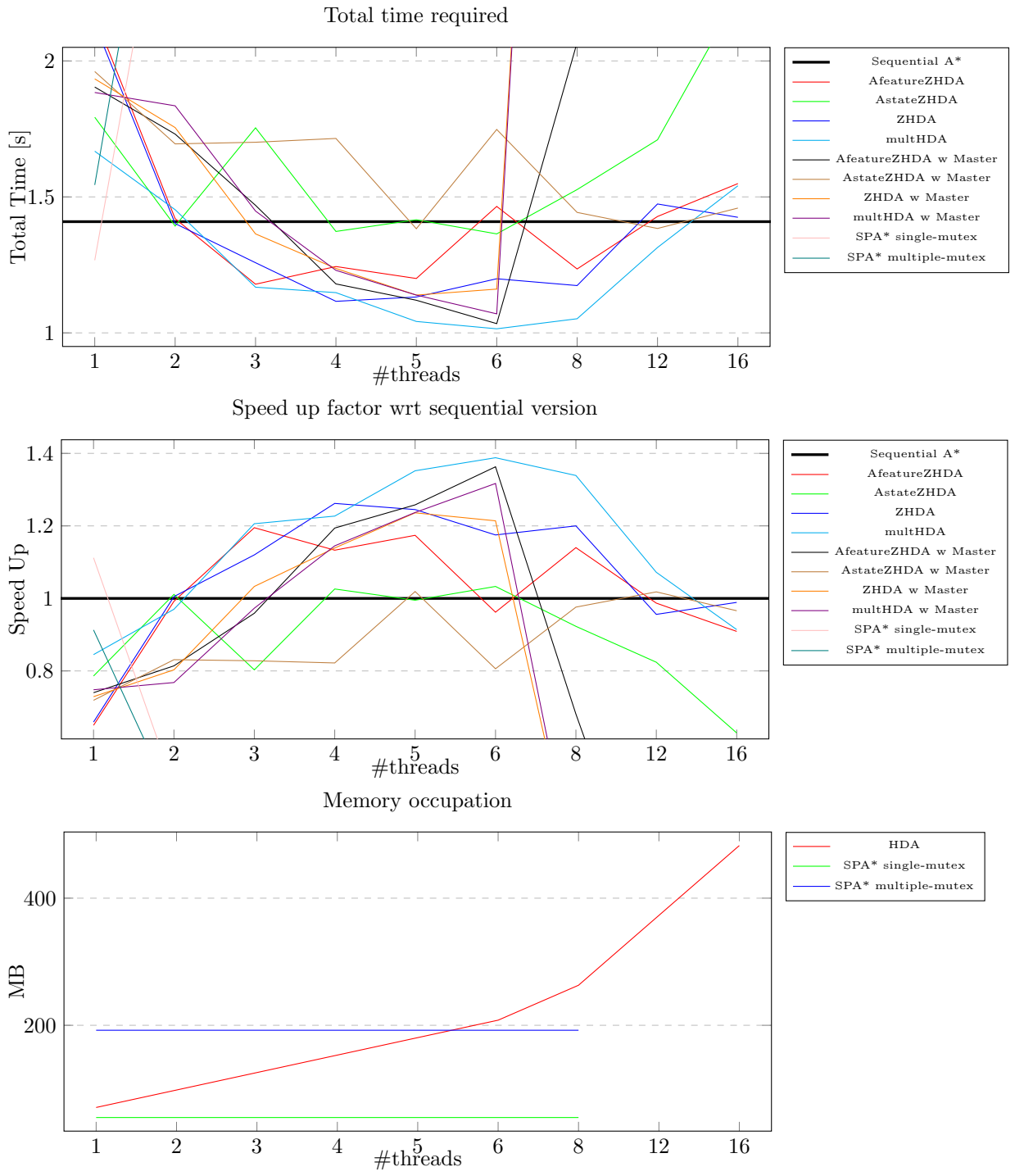
6.8 Great Lakes USA

Great Lakes USA road network with 2,758,119 nodes and 6,885,658 edges. Average path length: 2200 hops. Interval: [2000;2500].



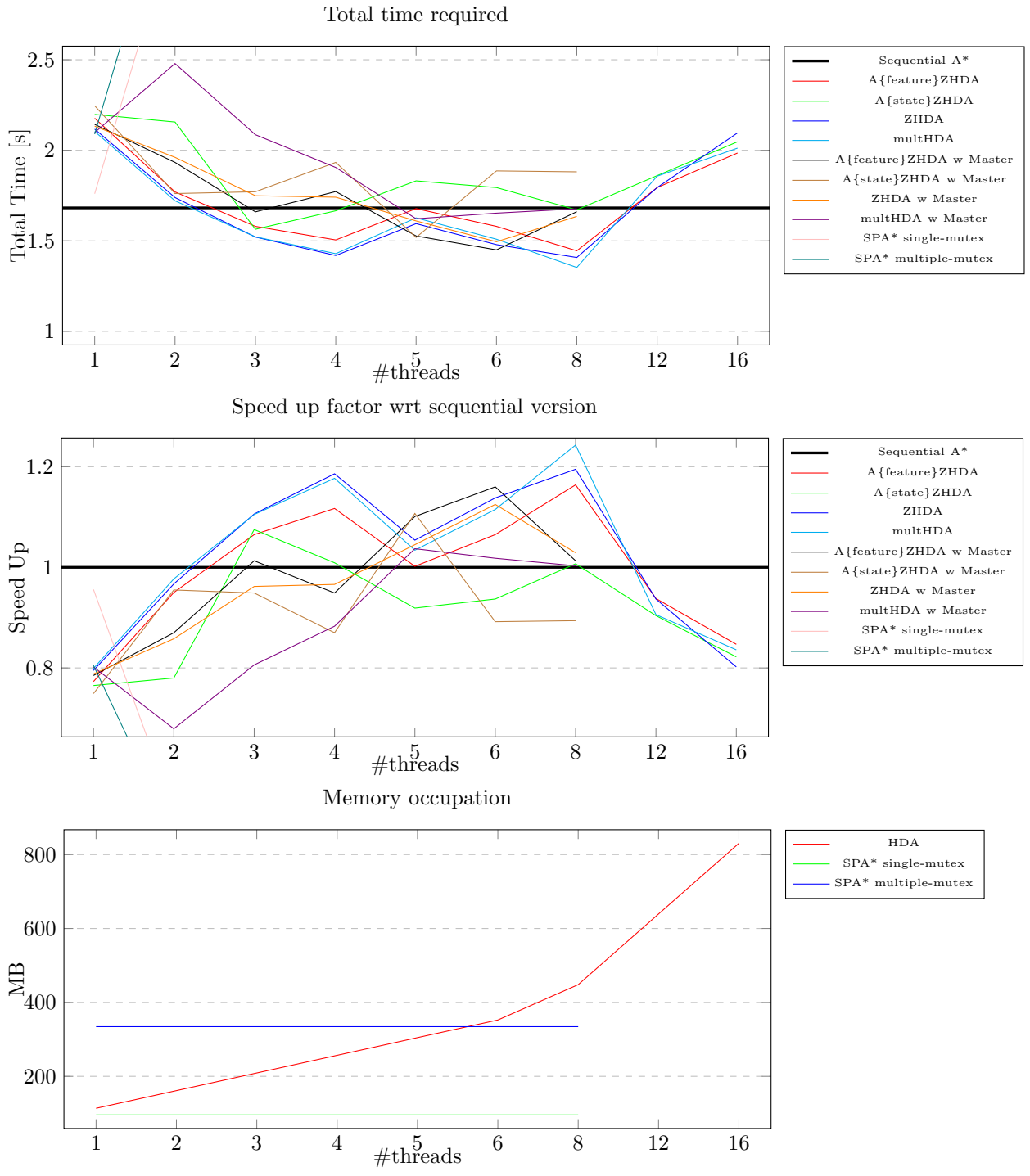
6.9 Eastern USA

Eastern USA Area road network with 3,598,623 nodes and 8,778,114 edges. Average path length: 2100 hops. Interval: [2000;2400].



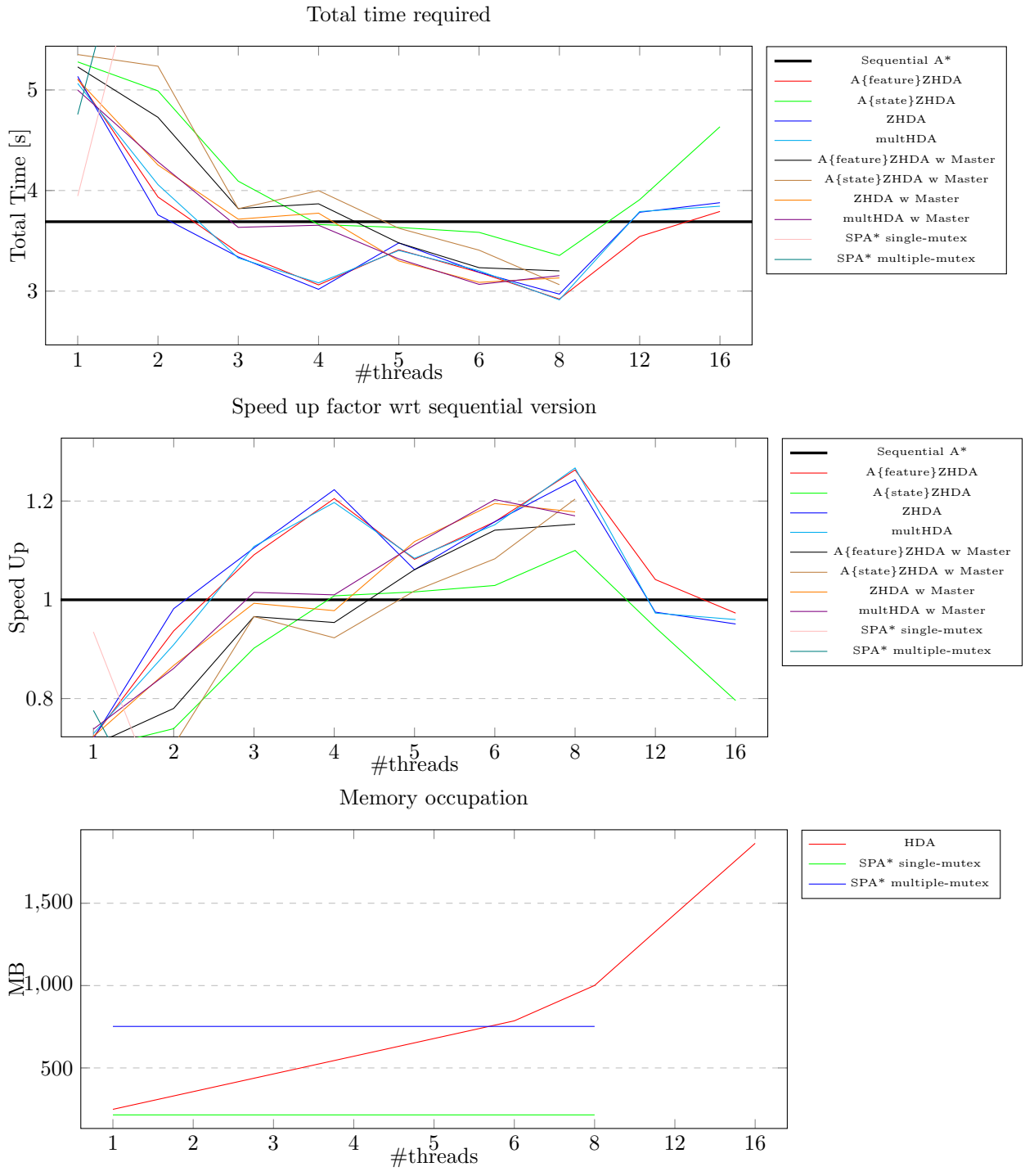
6.10 Western USA

Western USA Area road network with 6,262,104 nodes and 15,248,146 edges. Average path length: 2700 hops. Interval: [2500;3000].



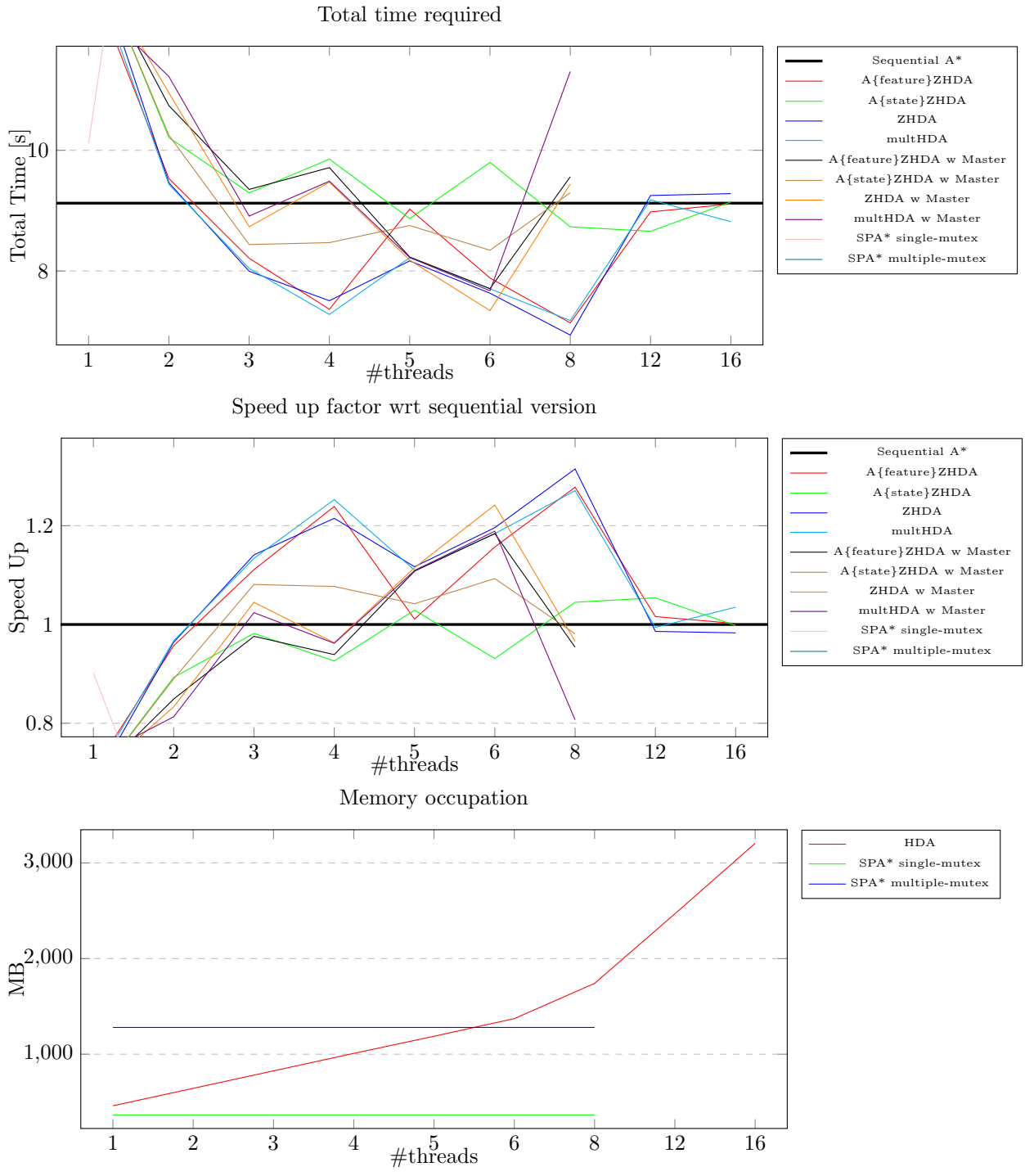
6.11 Central USA

Central USA road network with 14,081,816 nodes and 34,292,496 edges. Average path length: 3300 hops. Interval: [3000;3500].



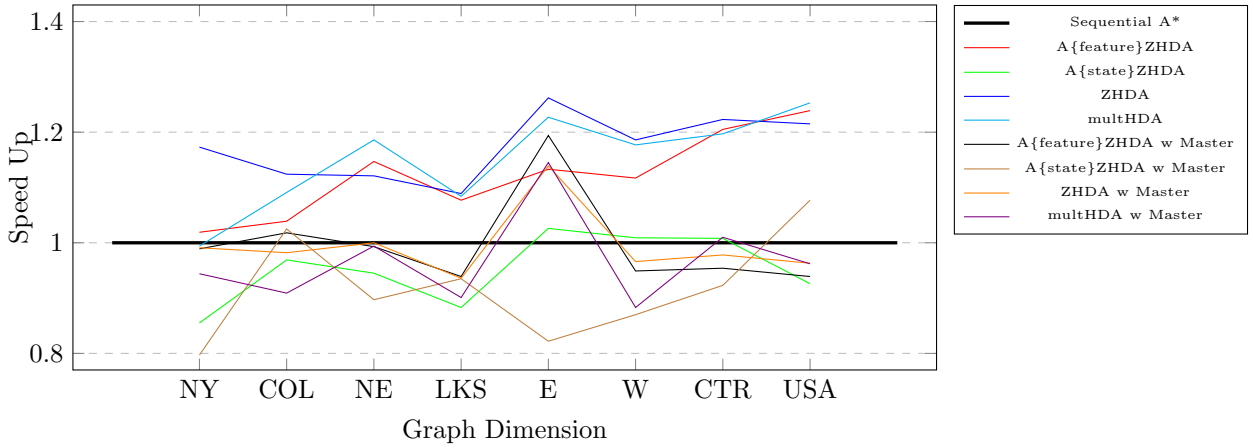
6.12 Full USA

Full USA road network with 23,947,347 nodes and 58,333,344 edges. Average path length: 5100 hops. Interval: [4800;5500].

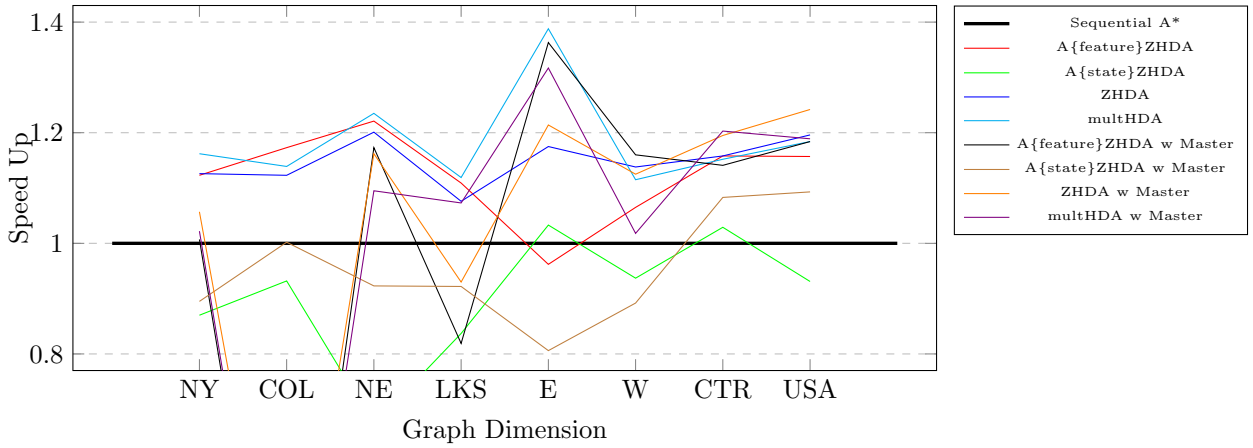


6.13 Comparisons between different graph dimensions and number of threads

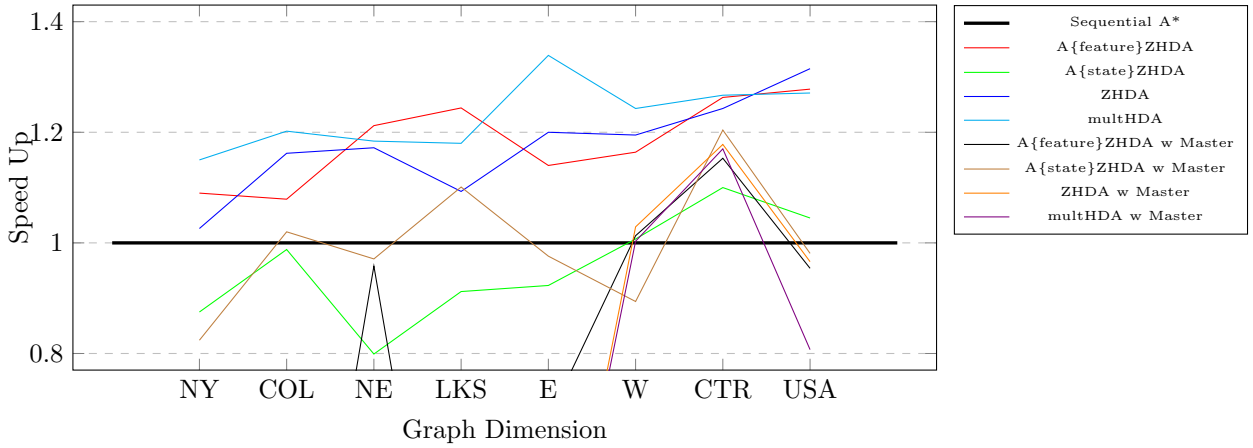
Speed up factor wrt sequential version and **4** Threads



Speed up factor wrt sequential version and **6** Threads



Speed up factor wrt sequential version and **8** Threads



6.14 Conclusions

We will first analyze the difference between SPA* *Simple Parrallel A** and HDA*: the first turned out to be the worst algorithm, it starts with a lower time than HDA* with 1 thread, but when running on 2 or more threads it's performances get worse and worse; in support of this it is possible to notice how the SPA*'s traces tend to immediately disappear in each of the *Total time* and *Speed up* charts.

This decrease in performance is, in our opinion, due to the shared resources employed: openSet, closedSet and path, that contribute significantly to increase the *Synchronization overhead*. In contrast, in a *decentralized* approach (HDA*) synchronization overhead is reduced as each thread/process has its own openSet and closedSet.

From the charts we can notice a decrease in performance when the *abstract* function is applied to the *state* rather than to the *feature*. The *abstract function* maps a point within a limited area of the map, and assigns all the points in that area to the same thread/processor. As a result of this, *Load Balancing* and *Communication overhead* decrease because there are threads that expand and assign to themselves more nodes, reducing overall performances.

Another difference is between the two types of architectures: with Master and without it. Most of the time spent in the Master version is for sending messages from Slave to Master and vice versa, furthermore communication is realized by exploiting *queues* that have to be synchronized. For this reasons the Master is a *bottleneck*, the latter problem has been solved in the second version of HDA*. In some cases this no-master HDA* version has best performance than all the others.

In general we can say that the *Speed Up* increases with the increasing of the *number of nodes* in the graph. And, overall the best algorithms are: *multHDA**, *A{feature}ZHDA** and *ZHDA**, with **6** and **8** threads. These algorithms *always* give a speed up in our tests using 4, 6 and 8 threads.

References

[1] Mattern. "Algorithms for distributed termination detection. Distributed Computing 2". In: (1987).

[2] Alex Fukunaga et al. "A Survey of Parallel A*". In: (Aug. 2017).

[3] Alex Fukunaga Yuu Jinnai. "On Hash-Based Work Distribution Methods for Parallel Best-First Search". In: (2017).

[4] Università Roma 1. *Graph Benchmark*. http://www.diag.uniroma1.it/challenge9/download.shtml.

[5] Rachel Holladay Ariana Weinstock. *Parallel A* Graph Search*. https://people.csail.mit.edu/rholladay/docs/parallel_search_report.pdf.

[6] University of Utah. *USA Road Networks*. https://www.cs.utah.edu/∼lifeifei/SpatialDataset.htm.

[7] Wikipedia. *Haversine Formula*. https://en.wikipedia.org/wiki/Haversine$_f ormula$/.

[8] Wikipedia. *Open Location Code*. https://en.wikipedia.org/wiki/Open_Location_Code.