Parallel A* project

September 10, 2022

Lorenzo Ippolito, Mattia Rosso, Fabio Mirto

Abstract

This project is part of the *System And Device Programming Exam (year 2022, Politecnico di Torino)*. The goal was to implement one or more parallelized versions of the famous path searching A* (A star) algorithm. It was requested us firstly to analyze the behaviour of the sequential version of A* with respect to the Dijkstra algorithm and secondly to implement a parallel version of A* to obtain results in terms of execution time, cpu usage and memory usage. The results we have obtained were compared with the sequential version to understand which can be the most effective approach to reach the initial goal.

1 Introduction: about the A* algorithm

A* is a graph-traversal and path-search algorithm. It is used in many contexts of computer science and not only. It can be considered as a general case of the Dijkstra algorithm. It is a Greedy-best-first-search algorithm that uses an heuristic function to guide itself. What it does is combining:

- **Dijkstra** approach: favoring nodes closed to the starting point(source)
- **Greedy-best-first-search** approach: favoring nodes closed to the final point(destination)

Using the standard terminology:

- $g(n)$: exact cost of moving from source to n.
- $h(n)$: heuristic estimated cost of moving from a node n to the destination.
- $f(n) = g(n) + h(n)$: combination of the actual cost with the estimated one.

At each iteration the node n that has the minimum $f(n)$ is examinated(expanded).

1.1 Heuristic design

Premises We are going to work with weighted oriented graphs where V is the set on nodes/vertices and E is the set of edges of the form (x, y) to indicate that an oriented edge from x to y exists and it has weight indicated as $cost(x, y)$.

Heuristic properties The heurstic function represents the actual core of the A* algorithm. It represents a prior-knowledge that we have about the cost of the path from every node (source included) to the destination.

- If we have not this prior information($h(n) = 0\ \forall n \in V$) we are turning the A* algorithm into Dijkstra (this is why A* can be considered as a more general case of Dijkstra algorithm) but we always have the guarantee of finding the shortest path.
- *Admissible* Heuristic: if $h(n) < cost(n, dest)\ \forall n \in V$ (so the we never over-estimate the cost to get to the destination from a node n) A* will always find the shortest path and the heuristic function is called *admissible*. The more inaccurate is the estimation the more nodes A* will need to expand (with the upper bound of expanded nodes if $h(n) = 0$).
- *Consistent* Heuristic: if $h(x) \leq cost(x, y) + h(y)$ for every edge (x, y) (so the triangular inequality is always satisfied) A* has the guarantee of finding an optimal path without processing any node more than once.

Corner cases

- **Dijkstra**: As already discussed if $h(n) = 0$ for every node in the graph A* turns into the Dijkstra algorithm.
- **Ideal**: We would obtain a perferct behaviour in case $h(n)$ is exactly equal to the cost of moving from n to the destination (A* will only expand the nodes on the best path to get to the destination).
- **Full greedy-best-first search**: if $h(n) \gg g(n)$ than only $h(n)$ plays a role and A* turns into a completly greedy-best-first search algorithm.

2 A* project application

Problem definition Given a wide range of fields where the A* algorithm can be applied we have choosen the one of optimal path searching in geographical areas where the goal is to find the minimum distance path from a node source to a node destination.

Notation We work with a weighted oriented graph G that is made of nodes $n \in V$ that represent road-realated points of interest and edges $(x, y) \in E$ that represent unidirectional connections between these points. Each edge (x, y) is associated to a weight that is the great-circle distance between x and y measured in meters.

Benchmark We will exploit the DIMACS benchmark to make robust estimates of the designed algortithms. Starting from the FIPS system format files provided we have adapted them (as better explained in section 3) to provide to the algorithms a file containing information structured as:

- Nodes: each node n is defined as $(index, longitude, latitude)$ where $index$ is a natural progressive number starting from 0 used to univocally identify the node and $(longitude, latitude)$ are the geographical coordinates of the node.

- Edges: each edge (x, y) is defined as $(x, y, weight)$ and it represents a unidirectional connection from x to y (a road) with length *weight* (great-circle distance from x to y).

2.1 Heuristic function: the great-circle distance

As previoulsly discussed the A* algorithm needs an *admssible* and *consistent* heuristic to properly work and this function is tipically problem-specific. Given the type of problem we are going to apply A* to we are going to use a measure of geographical distance that extends the concept of heuclidean distance between two points: the great-circle distance (that is the shortest distance over the earth surface measured along the earth surface itself). We will employee the **Haversine formula** to compute

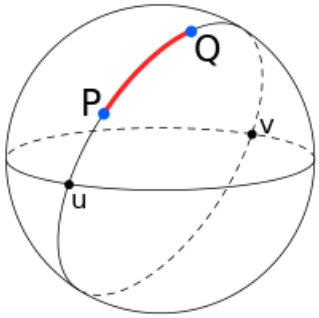

Figure 1: Great circle distance from P to Q

the distance from node (ϕ_1, λ_1) and node (ϕ_1, λ_1) where ϕ is the latitude and λ is the longitude:

$$d = R \cdot c$$
$$\text{where } c = 2 \cdot atan2(\sqrt{a}, \sqrt{1-a})$$
$$a = sin^2\left(\frac{\Delta\phi}{2}\right) + cos(\phi_1) \cdot cos(\phi_2) \cdot sin^2\left(\frac{\Delta\lambda}{2}\right)$$

R is the earth radius that we have fixed to $R = 6.371km$

3 GRAPH FILE INPUT

File input format Now we start discussing the A* algorithm implementation and to do that we need to specify which types of files we will need to provide to the algorithm to load the graph of interest. Each file has the format:

- First line: the number of nodes $N[int]$
- N following lines: nodes appearing as $(index[int], longitude[double], latidue[double])$
- E following lines (with E unknown): edges appearing as $(x[int], y[int], weight[double])$

DIMACS benchmark The benchmark files we have used come from the DIMACS benchmark. Here each geographic map is described by:

- *.co* file: a file containing the coordinates of the nodes following the FIPS system notation.
- *.gr* file: a file containing the edges and the relative weight(distance) expressed in meters.

The generation of a file consistent with the format described above happens by merging these files into a new one (in binary format). One of the challenge we are going to undertake is the one of parallelizing the reading of these huge files (that we will show having an high impact in terms of execution time over the total time spent by the algorithm).

Test paths To analyze the performance of the different versions of A* algorithms we have used these paths: They have

Table 1: Test paths for A*

Nodes	*Edges*	*Source*	*Dest*
California(BAY)			
321270	800172	321269	263446
Florida(FLA)			
1070376	2712798	0	103585
Western USA(W)			
6262104	15248146	1523755	1953083
Full USA(USA)			
23947347	58333344	14130775	810300

been chosen ad-hoc to make the algorithms find a way that crosses from side to side each one of the these benchmark maps as showed in figure 2.

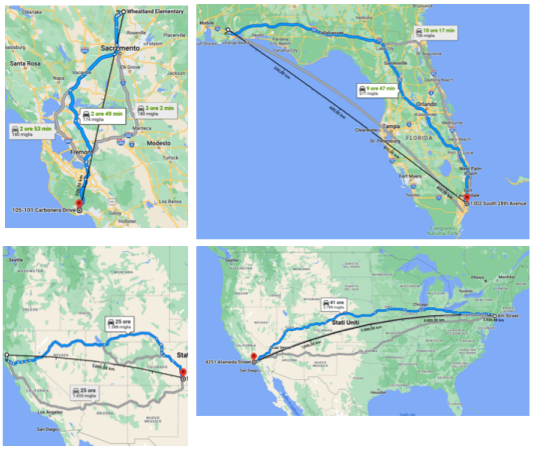

Figure 2: From left to right top to bottom BAY, FLA, W, USA

4 SEQUENTIAL A* ALGORITHM

Sequential A* algorithm is the one we will start with to see how it works and performs. The first step consists of a precomputation of:

- The heurstic $h(n)$ for each node (by defintion $h(dest)$ will be 0) computed through the Haversine formula. We thus keep a data structure `h` to do this.
- The initial values of $f(n)$ and $g(n)$ that will be set to $DOUBLE\_MAX$ for each node except for the source node that will have $f(source) = h(source)$ because $g(soruce)$ is clearly 0. We thus keep data structures `f` and `g` to do this.

- The *open set* contains all the nodes that still have to be explored(at the beginning only the source node).
- The *closed set* contains all the nodes that have already been visited and don't have to be re-evaluated.

We will also need two additional data structures:

- The *costToCome* table (where `costToCome[i]` contains the current best cost to reach node `i`) that is initialized to $DOUBLE\_MAX$.
- The *parentVertex* table (where `parentVertex[i]` contains the parent node of node `i` according to the current best path found to reach the destination) that is initialized with -1.

The outer loop is based on the nodes extraction from a *open set*, the one containing the nodes that have still to be explored (it is implemented as a Priory Queue where the priority is associated to the $f(n)$ of the nodes). At each iteration the node a with minimum $f(a)$ is extracted from the *open set* and its neighbors are expanded: the inner loop is repeated once for each neighbor b of the extracted node a(b is neighbor of a if the edge (a,b) exists). A tentative score is computed for each node b as $g(b) = g(a) + weight(a,b)$. If $g(b)$ is less than current `g[b]` these data structures are updated:

```
g[b] = g[a] + weight(a, b)
costToCome[b] = g[b]
parentVertex[b] = a
f[b] = g[b] + h[b]
```

Since the heuristic function we have chosen is both *admissible* and *consistent* we have the guarantee that the first time a node is extracted from the *open set* we have found a best path to it (this is why when the destination node is extracted we can terminate having found the best path). So despite one node may be added to the *open set* more than once (when discovered as a neighbor of different nodes) it will be expanded only one time.

Algorithm 1: Sequential A*

Data: Graph G(V,E), Source s, Destination d, Heurstic h
Result: Best path from Source to Destination and relative cost

$g[i] \leftarrow DOUBLE\_MAX \ \forall i \in V$;
$f[i] \leftarrow DOUBLE\_MAX \ \forall i \in V$;
$h[i] \leftarrow h(i,d) \ \forall i \in V$;
$costToCome[i] \leftarrow DOUBLE\_MAX \ \forall i \in V$;
$parentVertex[i] \leftarrow -1 \ \forall i \in V$;
$f[s] \leftarrow h[s]$;
$g[s] \leftarrow 0$;
$openSet := \{(s, f[s])\}$;
while $!openSet.EMPTY()$ **do**
  $a \leftarrow openSet.POP()$;
  if $a == d$ **then**
    $pathFound \leftarrow true$;
    reconstructPath();
  end
  if $a \in closedSet$ **then**
    CONTINUE
  end
  $closedSet.PUSH(a)$;
  foreach *neighbor b of a* **do**
    if $b \in closedSet$ **then**
      CONTINUE
    end
    $wt \leftarrow weight(a,b)$;
    $tentativeScore \leftarrow g[a] + wt$;
    if *tentativeScore is less than* $g[b]$ **then**
      $parentVertex[b] \leftarrow a$;
      $costToCome[b] \leftarrow wt$;
      $g[b] \leftarrow tentativeScore$;
      $f[b] \leftarrow g[b] + h[b]$;
      $openSet.PUSH((b, f[b]))$;
    end
  end
end

4.1 Results

We can realize from table 2 that the reading time has a very high impact on the overall execution time of the algorithm and in section 6 we will investigate one technique for parallelizing the reading of the file.

Table 2: Sequential reading + Sequential A* performance

	File Size	Reading	A*	Total	Reading Impact
BAY	20.51MB	0.9538s	0.2197s	1.1735s	81.3%
FLA	69.09MB	3.1551s	0.7174s	3.8725s	81.5%
W	394.26MB	18.3065s	2.5890s	20.8955s	87.6%
USA	1292.40MB	56.9942s	13.6716s	70.6658s	80.6%

5 A* AND DIJKSTRA: A COMPARISON

As already mentioned the Dijkstra algorithm can be considered as a particular case of A* where we don't have any prior knowledge about the distances beteween the nodes ($h(n) = 0 \ \forall n \in V$). We have also seen that the more precise is the heuristic function we provide the less nodes the algorithm will expand to get to the destination. To investigate this point we have run Dijkstra algorithm on the same graph comparing the number of expanded nodes by the two algorithms:

Table 3: Expanded nodes in different maps

	Dijkstra	Sequential A*
BAY	318725 of 321270	157137 of 321270
FLA	996956 of 1070376	592480 of 1070376
W	5470394 of 1070376	1600083 of 1070376
USA	16676528 of 1070376	8998767 of 1070376

The number of expanded nodes is clearly much higher when Dijkstra algorithm is used and the picture 4 cleary show in blue the nodes expanded by the sequntial A* algorithm while the red nodes are the ones expanded by the Dijkstra algorithm. In figure 5 we can notice that the execution time is not necessary much better for A*. This could depend both on the path we are looking for and on the precision of the heuristic function with respect to the cost of the optimal path (in FLA map the heurist

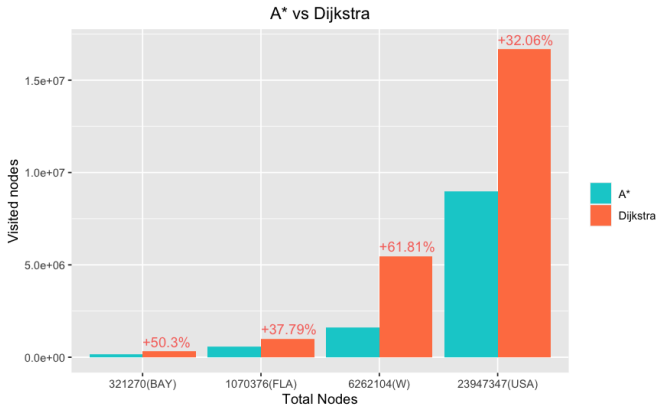

Figure 3: Expanded nodes: A* vs Dijkstra

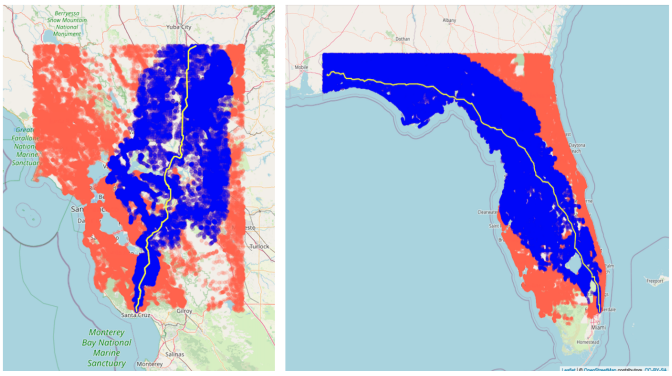

Figure 4: Test paths on BAY(left) and FLA(right)

estimate is the point-to-point distance from source to destination that is clearly different w.r.t. the actual best path). From figure 6 we realize that considering CPU and Memory usage A* needs more memory because of the higher number of data structures allocated but the less number of nodes expanded makes the CPU usage inferior w.r.t Dijkstra algorithm in some cases.

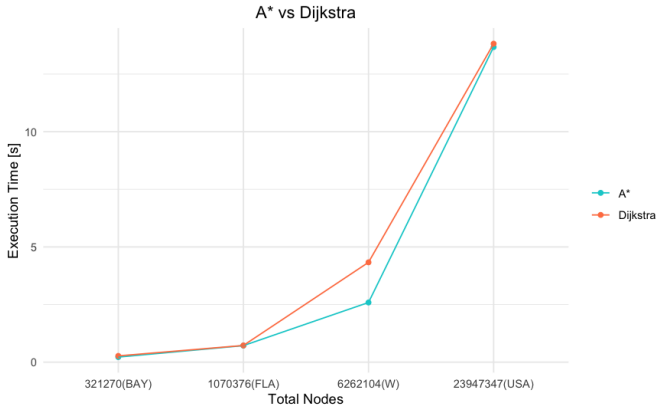

Figure 5: Execution time: A* vs Dijkstra

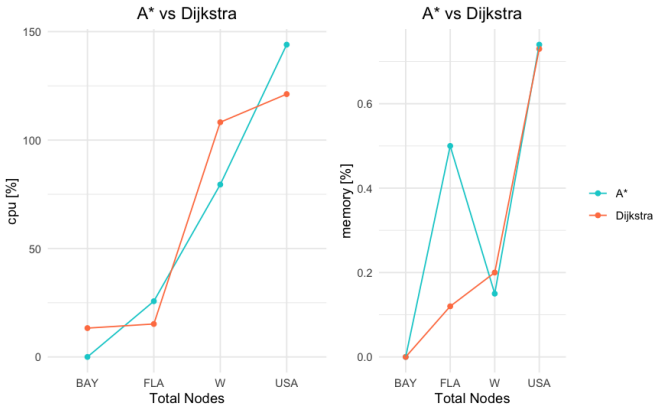

Figure 6: Exploited resources: A* vs Dijkstra

6 PARALLEL READING OF THE INPUT FILE

As we have previously observed the time spent by the algorithm to read the input graph (the input binary file) used by the A* algorithm is very high w.r.t. the total amount of time. This is the reason why we have decided to inspect different techniques of parallelization of the reading phase to speed it up (but only the presented one proved to be really effective). The input file, as already discussed, is divided in two different sections (nodes and edges) so, in general, we need to take care of which section a given thread is working on because different data structures of the graph need to be loaded in the two sections (the symbol table when reading a *node line* and the linked list when reading a *edge line*).

6.1 *Parallel Read: approach 1 (RA1)*

This parallel reading techinique is based on:

- Memory mapping of the input file
- Concurrent reading of the memory mapped data structure

The reading of the input file in a parallel way happens using N threads that concurrently access the pointer to the memory mapped file. While the reading of the input file can happen in whatever order and concurrency doesn't represent a problem the loading of Graph data structures must happen in mutual exclusion and this is why two locks are used:

- One lock to protect the loading of the nodes in the Graph (to fill the Symbol Table)
- One lock to protect the loading of the edges in the Linked List data structure of the Graph

Results on FLA As we can notice from figure 7 the most effective improvement comes probably from the memory mapping. The increasing number of threads is infact not impacting well the reading time.

6.2 *Parallel Read: approach 2 (RA2)*

This parallel read approach is simply an extension of RA1. The only difference is that instead of loading only the input graph G

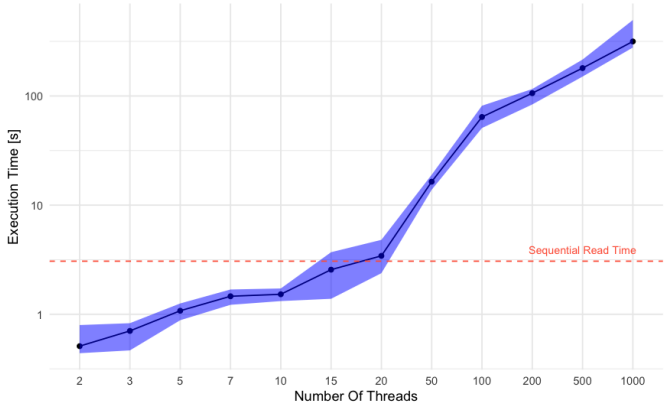

Figure 7: Performance of RA1 for different number of threads on FLA

it is also loaded the reversed graph R. This type of reading is indeed applied only for using the PNBA* algorithm.

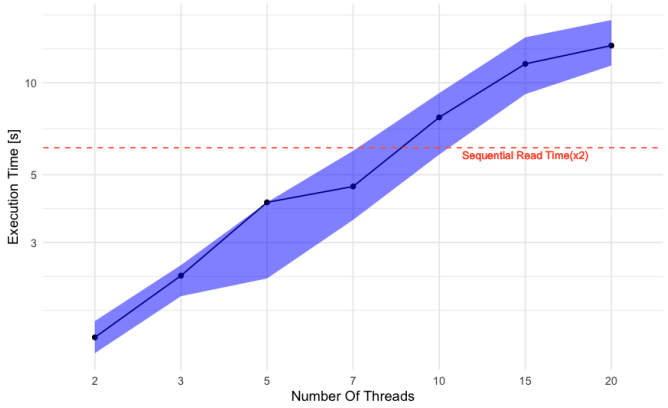

Figure 8: Performance of RA2 for different number of threads on FLA

6.3 *Final results*

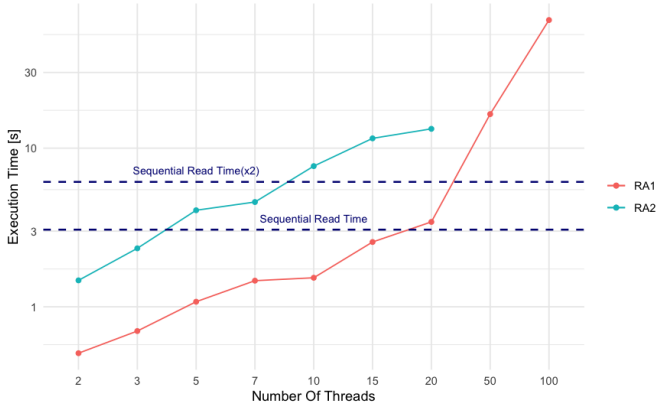

Figure 9: RA(Read Approach) 1,2 compared with sequential reading on FLA - execution time

Figure 10: RA(Read Approach) 1,2 compared with sequential reading on all maps - exploited resources

For the sake of completeness we have also test the most promising reading approach (RA1 and RA2 both with 2 threads) on all the maps (BAY, FLA, W, USA) and the results are showed in table 5 where the speed-up with respect to the sequential reading time is evident. Due to the Memory Mapping mechanism

Table 4: RA1 results against sequential reading

	RA1 (2 threads)	**Sequential**	**Speed-Up**
BAY	0.1936s	0.9366s	79.3%
FLA	0.5103s	3.0650s	83.4%
W	3.3303s	17.8834s	81.4%
USA	14.1492s	56.4445s	74.9%

Table 5: RA2 results against sequential reading

	RA2 (2 threads)	**Sequential(x2)**	**Speed-Up**
BAY	0.5313s	1.8732s	71.6%
FLA	1.4682s	6.1300s	76.1%
W	10.4959s	35.7668s	70.7%
USA	35.4505s	112.8890s	68.6%

the resources exploited are higher in terms of memory usage and also CPU usage has been increased w.r.t the sequential file reading.

7 PARALLEL A*

The goal of the project was to find one or more parallel versions of the A* algorithm and showing their performances w.r.t. the sequential version. We have choosen two approaches to face the problem of parallelizing the A* algorithm: a first approach(known in literature as HDA*) that puts in action a complex way of parallelizing the algorithm by defining a hash-based work distribution strategy and a second approach that makes use of the parallelism in order to make the work of finding the shortest path from *source* to *dest* split between only two threads where one looks for the path from *source* to *dest* and the other one looks for the path from *dest* to *source* in the reversed graph (the PNBA* algorithm).

7.1 HDA*

The Hash-Distributed-A* (HDA*) algorithm works is based on the fact that each thread is *owner* of a specific set of nodes of the Graph: given a node n it is defined a hash function $f: \ f(n) = t$ where $t \in \{1..N\}$ with N the number of threads. When a thread extracts from the *open set* (expands) a node all its neighbors are added to the *open set* of the owner thread of the expanded node. One important fact is that HDA* doesn't provide the same guarantees of the sequential algorithm:

- In sequential A* if it's provided an heuristic function that is both *admissible* and *consistent* we have the guarantee that each node will be only expanded once and that the first time we expand that node we have found a shortest path to it.
- In HDA* we loose these guarantees: since we don't know in which order nodes will be processed it could happen that a longer path to *dest* is found before the shortest one so a node could be opened more than once and expanding the *dest* node doesn't mean that we have terminated.

Hash Function The way how the threads divide among themself the work to be done happens using a *hash function*. The hash function that we have employeed at the beginning of our experiments was simply:

$$hash_1(node\_index, num\_threads) = node\_index \ \% \ num\_threads$$

As we can see in figure 11 with 3 thread using this *modulo* hash function the work distribution is so equal among the 3 threads that the way how each one works for finding an optimal path to destination could be not the optimal one. What proved to be

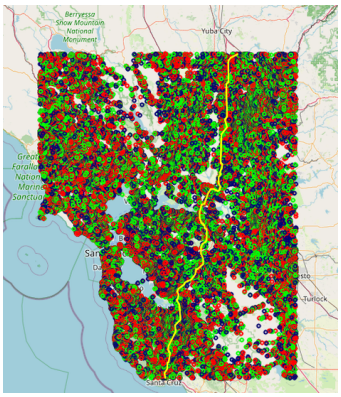

Figure 11: HDA* work distribution among 3 threads in BAY map (using $hash_1$ function)

a better approach was to implement another hash function that works in a different way:

$$hash_2(node\_index, num\_threads, V) = i - 1$$

$$i = min_i : \frac{V}{num\_threads} \cdot i > node\_index \ , i \in \{1, ..., num\_threads\}$$

What can be not optimal if we exploit the *modulo* hash function above could be that the fully randomly work distribution would make the work of each thread much harder because lots of communication is needed and the termination condition is reached many times with the result of more CPU consumption and time needed to find the solution. We can define this as an absence of autonomy for each thread that causes a drastic communication overhead. What tries to do the second hash function (the one that we have used to measure performances) is simply assigning nodes to threads following their index numbering (e.g. nodes from 0 to K to thread t_0, nodes from $K+1$ to H to thread t_1 and so on and so forth). In this way we have tried to overcome the limiations of the *modulo* hash function. Different solutions that could be implemented in future works will be explained later.

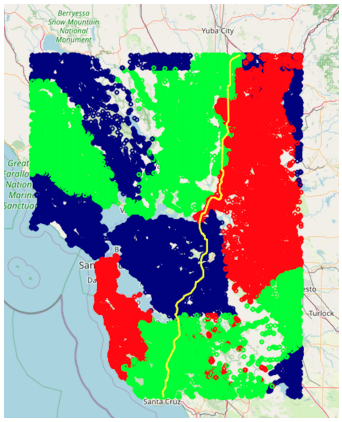

Figure 12: HDA* work distribution among 3 threads in BAY map (using $hash_2$ function)

Distributed Termination Condition If in the sequential algorithm expanding the node *dest* triggers the end of the algorithm (despite the fact that the *open set* could be not empty) in HDA* this is not valid anymore: when a thread is working on its *open set* it is expanding nodes putting their neighbors in the *open set* of another thread. When thread t_i has its *open set* empty it could think to have finished its work but this might not be true because another thread t_j could be sending to t_i some nodes that have to be be processed in the meanwhile. We have adopted two different approaches for the distributed termination condition that are:

- **Barrier method (B)**: when a thread realizes that its *open set* is empty a barrier is hit and when all the threads have hit the barrier each one makes a check to confirm(or not) that all the *open sets* of all the threads are still empty. If this is not true it means that there are nodes that have still to be processed and the best path to *dest* found so far could not be the optimal one otherwise all the threads can terminate.
- **Sum-Flag method (SF)**: the idea behind the sum flag method comes from the fact that the Barrier mechanism could be quite expensive. In this termination condition method each thread keeps a binary flag saying whether its *open set* is empty or not. When no more nodes are inside it the flag is set and if $\sum_{i=1}^{N} flag[i] = N$ all the threads can correctly terminate.

Communication methodology What it differentiates the algorithms we have implemented is not only the distributed termination condition (B or SF) but also the way how the threads communicate with each other. This can be done using a shared address space (SAS) approach or a message passing (MP) model.

7.1.1 Message Passing Model (Using Message Queues) - MPMQ

One way of achieving the communication among threads is message passing. In this first attempt we will rely on Message Queues: when thread t_i expands a node and computes(through the hash function) its owner t_j a message is sent from t_i to t_j. Since the message queue is unique for all the threads each thread needs to be able to process only messages directed to itself. The termination condition that has been implemented in this case is the Barrier Method (B).

Data structures sharing Each thread mantains all the data structures as private (*open set*, *parentVertex*, *costToCome*) and the only way it has communicate with other threads is via message passing.

Message Structure Each message sent from t_i to t_j regarding node n contains:

- Message id: the identifier of the owner thread t_j
- The index of the node n
- The parent node of n (the one expanded from the *open set* of t_i)
- The value of $g(n)$ according to t_i
- The value of $f(n)$ according to t_i

Message Based Path Reconstruction What it makes not trivial the path reconstruction phase in the Message Passing model is that $parentVertex$ and $costToCome$ data structures are not shared among threads. This means that thread t_i will have inside $parentVertex_i$, once the algorithm is terminated, a consistent value of $parentVertex_i[n]$ only for the nodes n it is the owner of. This implies that if we want to know which is the parent node of node n in the final path this information is stored in $parentVertex_i[n]$ (where t_i is the parent of n). Path reconstruction needs to be done in a message passing fashion starting from the destination's thread owner t_d. What happens is that t_d sends a message to the owner of $parentVertex_d[dest]$ (abbreviated as $pV_d[dest]$) that we call t_k. Immediately after t_k will send a message to $pV_k[pV_d[dest]]$ and so on and so forth till we have reached the first node of the path (the one with $pV_x[source] = -1$ where t_x is the owner of *source*).

Technical Issues Due to the fact that Message Queues have a limited size that is not enough to support any of our selected benchmark graphs we have implemented the code but we weren't able to correctly test it. The measures that will be reported next will only regard the MP model that exploits Shared Memory.

7.1.2 Message Passing Model (Using Shared Memory) - MPSM

The Message Passing model that exploits Message Queues has been designed to work under the assumption that any data structure can be shared among threads. We are now going to relax this constraint by keeping some data structures shared among threads (like $parentVertex$ so that the path reconstruction can happen in the standard way) and the exchange of messages will be based on Shared Memory.

Data structures sharing Each thread mantains only some data structures as shared but the only way how a thread t_i can send a node to be processed to its owner t_j is by writing on Shared Memory. This buffering mechanism has the great advantage of minimizing the resource contention among threads (something that will strongly penalize the SAS model explained after).

Message Passing Each message (data written on the Shared Memory) sent from t_i to t_j regarding node n contains:

- The index of the node n
- The parent node of n (the one expanded from the *open set* of t_i)
- The weight of the edge (a, n) where a is the parent of node n
- The value of $g[n]$ according to thread t_i

The way how messages are read and written by the different threads is via *Readers and Writers* model where:

- Thread writer t_i writes on Shared Memory increasing a global pointer smp_G.
- Thread reader t_j reads from the Shared Memory by increasing a local pointer smp_j. The reading of messages terminates when $smp_j == smp_G$. Reading, in this context, means inseriting inside the *open set* all the nodes (and related information) received from other threads.
- Nodes expansion from *open set* is similar w.r.t the sequential algorithm but a node will be either pushed insided $openSet_i$ of thread t_i or written (as a message) on the Shared Memory if t_i is not its owner.

Message Based Path Reconstruction Since the data structure $parentVertex$ is shared we no longer need message based path reconstruction.

7.1.3 Shared Address Space Model

This approach applies the explained concepts of HDA* by using a communication method among threads that exploits the shared address space (SAS). This means that every data structure is shared and this makes at the same time simpler the way how threads can communicate but more complex the concurrent access to the resources. There is:

- A global array of *open sets* of size N (number of threads) where $openSet[i]$ contains a pointer to the *open set* of thread t_i.
- The $parentVertex$ and $costToCome$ data structures are shared among all the threads.

This approach clearly requires locks so that the operations on the shared data structures can happen in mutual exclusion. In particular we need:

- *N mutex_threads* locks (one for each *open set* so *N* overall).
- *V mutex_nodes* locks (one for each node of the graph so *V* overall)

With SAS both the barrier (B) and the sum-flag (SF) distributed termination conditions have been implemented to compare their performances.

7.2 Results

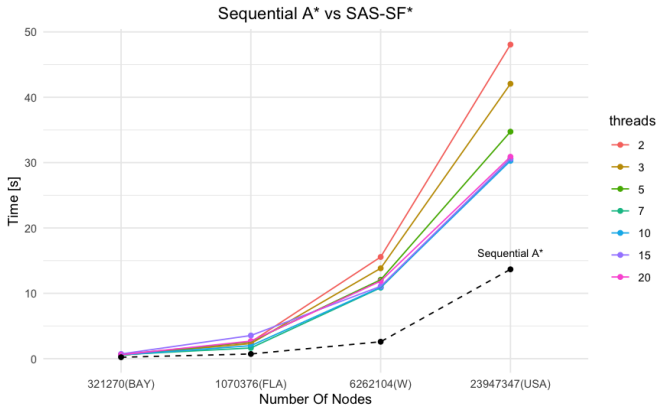

Figure 13: HDA* - SAS-SF - time

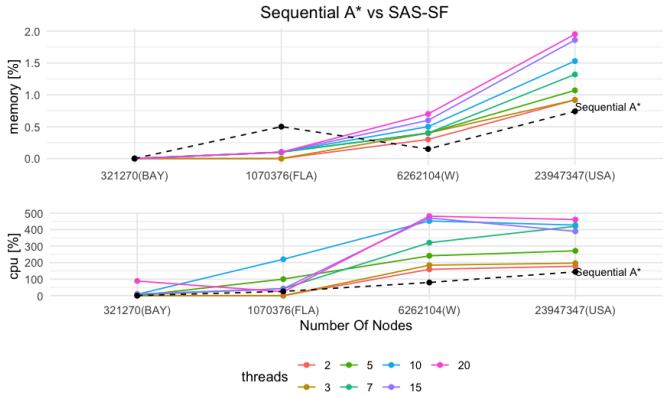

Figure 14: HDA* - SAS-SF - resources

Table 6: SAS-SF with best number of threads time performances

	Threads	SAS-SF	Sequential A*	Perf.
BAY	5	0.5097s	0.2647s	-92.6%
FLA	7	1.6383s	0.7174s	-128.3%
W	7	10.8626s	2.5890s	-319.6%
USA	7	30.5655s	13.6716	-123.6%

These are the troubles found during implementation and testing:

- The MP-MQ variant of HDA* is not scalable. This is due to the fact that we have used Linux Message Queues that have a limited size (few bytes) and when the communication overhead becomes huge the queue gets full causing a impressive slowdown.

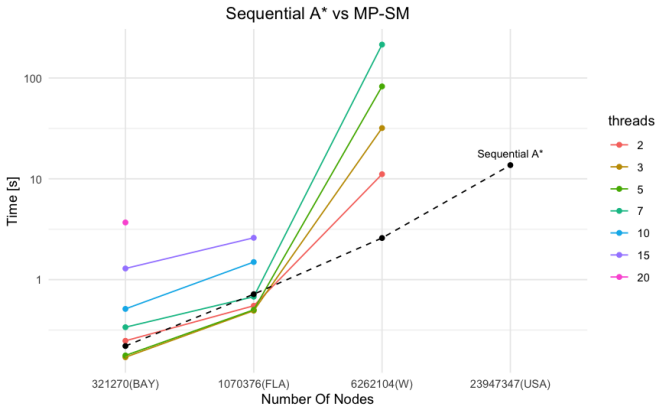

Figure 15: HDA* - MPSM - time

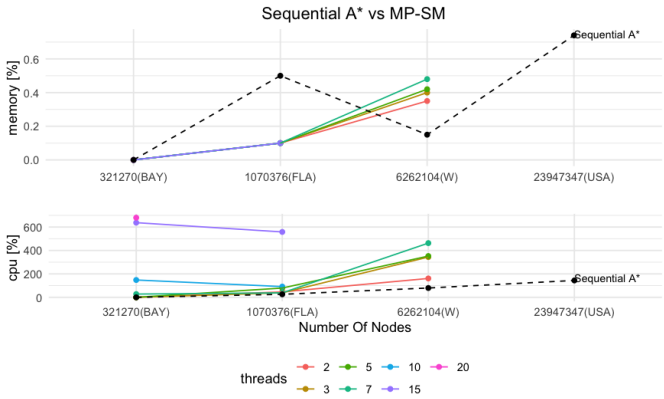

Figure 16: HDA* - MPSM - resources

Table 7: MP-SM with best number of threads time performances

	Threads	MPSM	Sequential A*	Perf.
BAY	5	0.1770s	0.2647s	+33.1%
FLA	3	0.4922s	0.7174s	+31.4%
W	2	11.1136s	2.5890s	-329.3%
USA	-	-	13.6716	-

- Better performances are achieved by the MP-SM variant. On BAY and FLA it achieves very good results w.r.t SAS-SF and also w.r.t. the sequential algorithm. The working mechanism makes the algorithm not properly working on larger graphs obtaing bad performances. Both the size of the graph and the increasing number of threads require too much memory(more than the available one).
- The SAS HDA* that exploits the Barrier termination condition (SAS-B) is more scalable than MP-MQ but the termination condition is not well-performing on large graphs. This happens because a thread has no way to shortcut the barrier if it's receiving work from another thread but it has to wait until all the threads have reached the barrier. Despite the fact that we noticed an

improvement when the number of threads increases the execution time degenerates when using maps W and USA.

- The SAS HDA* that exploits the Sum-Flag termination condition (SAS-SF) behaves overall better. Despite the fact that on maps BAY, FLA, W it is difficult to notice the improvements when the number of threads increases this is more evident on USA map. Performances are always better compared to the SAS-B algorithm and this is confirmed as the most well-performing distributed termination condition.
- About the resources consumption both SAS-B and SAS-SF are more expensive in terms on CPU and memory used w.r.t the sequential algorithm. This is not valid for MP-SM where the consumption of resources is lower in terms of CPU usage(because of the lower degree of concurrency). The Shared Memory mechanism that we have adopted tries to speed-up the algorithm but has the drawback of being worse in terms of memory used (and not fully scalable in this version).

HDA* has been proven not to work very well in general in our application. We have tried to explain these results by observing that:

- The hash function has an important impact on the performances of A* algorithm. The one that we have used is probably not the optimal(for instance tring to assign near nodes to the same threads maybe by pre-clustering the nodes of the map in a number of clusters equal to the number of threads could be a better option).
- The termination condition is clearly a bottleneck, in particular if the Barrier method is used. This can be less evident with small maps but we can appreciate it on bigger graphs. By increasing the number of threads both SAS-SF and SAS-B perform better in terms of execution time even if the resource consumption increases but this is not enough and the performances scalability and improvements that should have been obtained with large graphs were not proved.
- Resource contention is another important element. We have proved by using MP-SM that when the number of shared resources is minimum and thus the need of mutual exclusion access to data structures is almost absent threads work better (sometimes achieving better results w.r.t the sequential algorithm). On the other hand this forces threads to use communication method like Shared Memory that can be quite memory expensive and not fully scalable.

7.3 New Bidirectional A*(NBA*)

The NBA* algorithm is a version of the bidirectional search that uses a data structure M to keep track of the nodes in the middle between the two searcher threads t_G and t_R. M initially contains all the nodes of the graph. The nodes in the search frontiers are the ones that:

- Belongs to M
- Have been labelled: $g_G(n) < \infty$ or $g_R(n) < \infty$

The threads t_G and t_R share a variable L initialized to ∞ that contains the cost of the best path from *source* to *dest*. Other common variables are:

- F_G: lowest f_G value on t_G frontier.
- F_R: lowest f_R value on t_R frontier.
- Variables F_p, f_p, g_p (with $p \in \{R, G\}$) are written on only one side but read by both sides.

These are the initialization steps done by t_G (same for t_R)

- $g_G(source_G) = 0$, $F_G(source_G) = f_G(source_G)$

At each iteration it is extracted a node x such that:

- $x \in M$
- $x : f_G(x) = min f_G(v) \forall v \in openSet_G$

The node is removed from M and pruned (not expanded) if $f_G(x) \geq L$ or $g_G(x) + F_R - h_R(x) \geq L$. Otherwise all its successors y are generated. In the first case it is classified as *rejected* while in the other situation it is *stabilized* because $g_G(x)$ won't be changed anymore. For each y we update:

- $g_G(x)$: $min(g_G(y), g_G(x) + d_G(x, y))$
- L: $min(L, g_G(x) + g_G(y))$

The algorithm stops when no more candidates have to be expanded in one of the two sides.

7.4 Parallel New Bidirectional A*(PNBA*)

The PNBA* algorithm improves the NBA* algorithm by letting the two threads working in parallel and not in a alternate mode. This requires to cope with mutual exclusion on some data. In

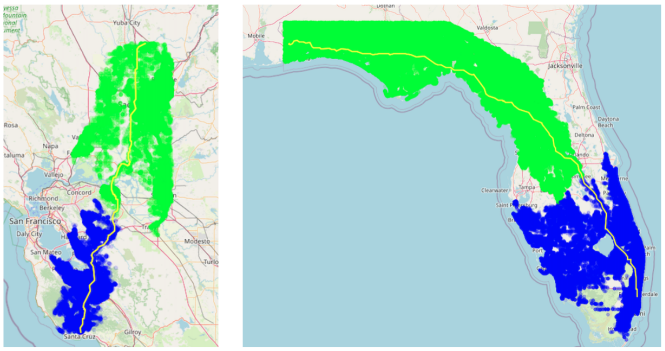

Figure 17: PNBA* work on BAY(left) and FLA(right)

figure 17 we realize that the work distribution is not equal and changes at each iteration (depending on which is the common node found).

7.4.1 Results

- The PNBA* is able to outperform the sequential algorithm in terms of execution time in all the graphs we have tested it on. The speed-up increases as the number of nodes increases and this can be a good news if we

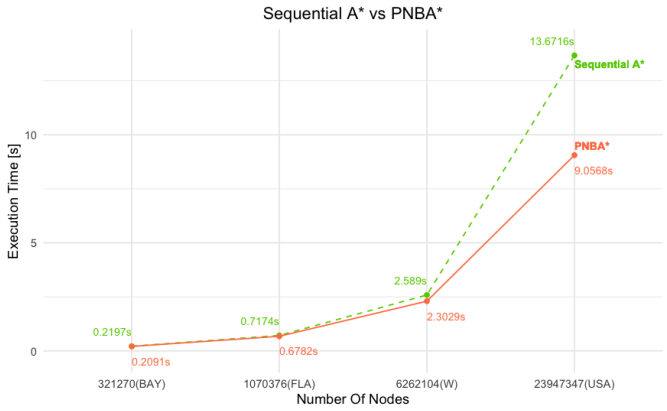

Figure 18: PNBA* performances - time

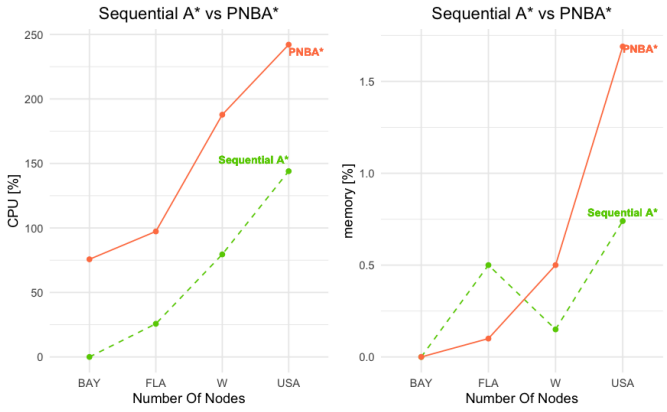

Figure 19: PNBA* performances - resources

Table 8: PNBA* - time performances

	PNBA*	Sequential A*	Speed-Up
BAY	0.2091s	0.2197s	4.82%
FLA	0.6782s	0.7174s	5.46%
W	2.3029s	2.5890s	11.1%
USA	9.0568	13.6716s	33.8%

will try to implement it on much bigger graps. The execution time and the number of nodes have been proved to strongly depend on the position of the common node found. Best performances are achieved when the common node found is approximately in between of source and destination nodes.

- Resource consumption is almost 2x w.r.t. the sequential algorithm and this is reasonable considering that it is like running two sequential algorithm in concurrency
- HDA* with MP-SM (using SF termination condition) is the one that behaves better in terms of execution time both on BAY and FLA but it's not fully scalable. Moreover good results are achieved in terms of CPU because of the limited resource contention among threads.
- PNBA* is the more scalable one and the only one that was proved to achive better results in terms of execution time on all the benchmark maps we have used. Also reasonable performances are achived in terms of CPU and memory usage.

8 CONCLUSIONS

By observing figures 20 and 21 that shows performances achived by the different algorithms (for each map the best number of threads for each algorithm):

- HDA* with SAS-SF is the one that achives worst results in terms of execution time and CPU usage.

To conclude, among all the different approaches we have implemented to parallelize A*, PNBA* seems to be the more promising one. Despite the fact that HDA* using MP-SM model behaves better in terms of execution time on BAY and FLA graphs it is not fully scalable in this version. We can say that, probably, for these type of problem where A* can be applied (geographical path searching) the techniques that deserve to be more deeply investigated are the ones that could come from variants and improvements of PNBA* algorithm.

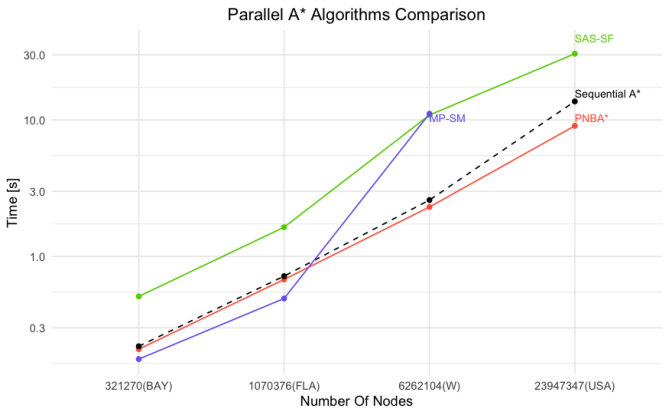

Figure 20: A* algorithms comparison - time (log. scale)

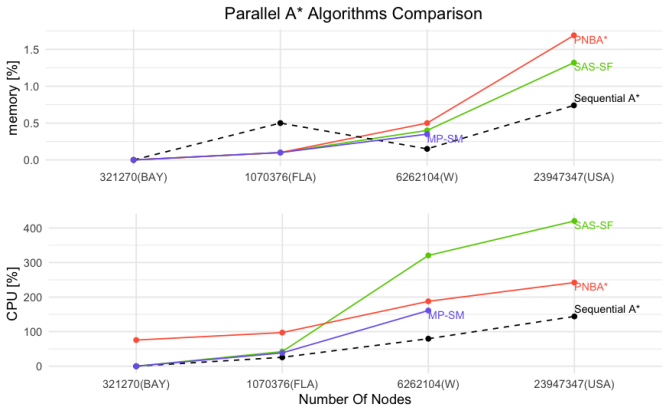

Figure 21: A* algorithms comparison - resources

9 COMPUTING FACILITIES PLATFORM

We have tested all our work on the *SmartData@PoliTO* Cluster. There are 33 storage workers equipped with:

- 216 TB of raw disk storage
- 384 GB of RAM
- Two CPUs with 18 cores/36 threads each
- Two 25 GbE network interfaces
- More than 50 GB/s of data reading and processing speed

10 FUTURE WORK

Features improvements could be realated to:

- Investigate more techniques of parallel file reading to achieve results that improves when the number of threads increases. This could be done by directly modifing the functions of the graph that are in charge of loading all its data structures. This could bring to a higher degree of parallelism.
- Updating the current version of HDA* with MP-SM by handling the Shared Memory in a more sophisticate way to make it more scalable. This could bring to optimal results if the performances will be proportional to the ones observed in BAY and FLA.
- Other techniques starting from PNBA* can be explored: for instance make it working with more than 2 threads. This could happen by splitting the path from *source* to *dest* in more sections and then applying PNAB* on each section. This could also be combined with a trade-off between performance and cost of the path found(accepting to find not optimal paths but in a very small time).
- In terms of performance measuring we will look for more stable and detailed way to inspect CPU usage and memory usage (using instrumentes like *Valgrind*).

11 COMPLETE RESULTS

Here also detailed results obtained on each map by HDA* algorithms are showed.

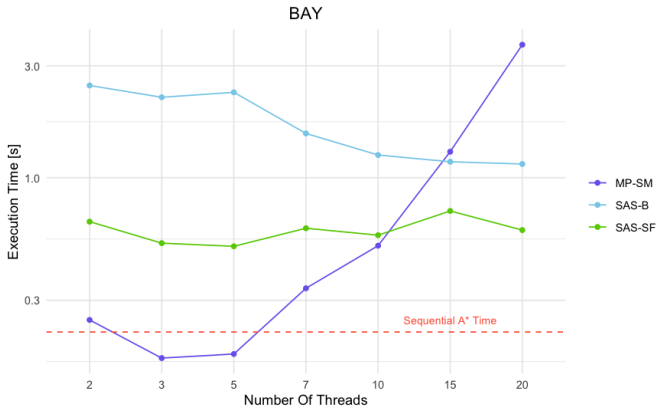

Figure 22: HDA* on BAY map - time

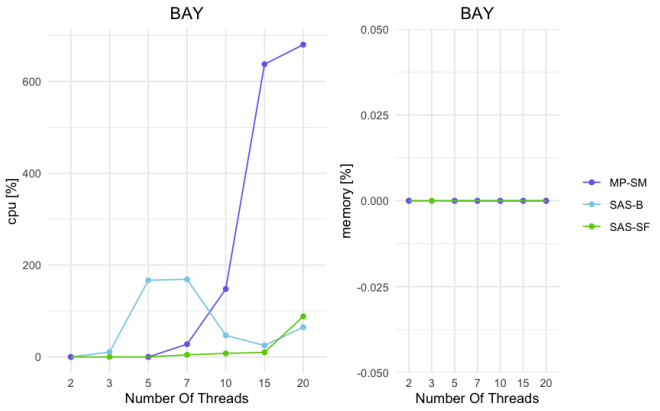

Figure 23: HDA* on BAY map - resources

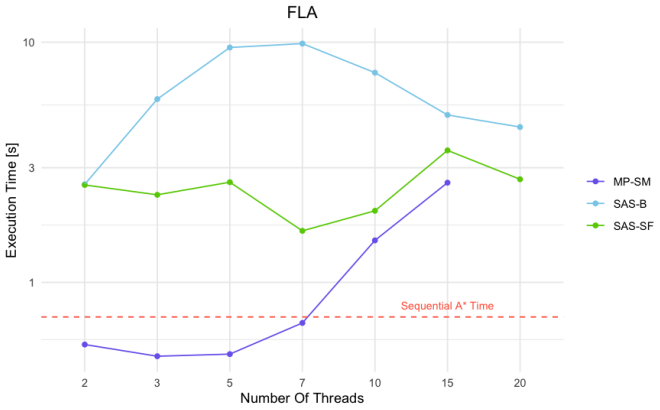

Figure 24: HDA* on FLA map - time

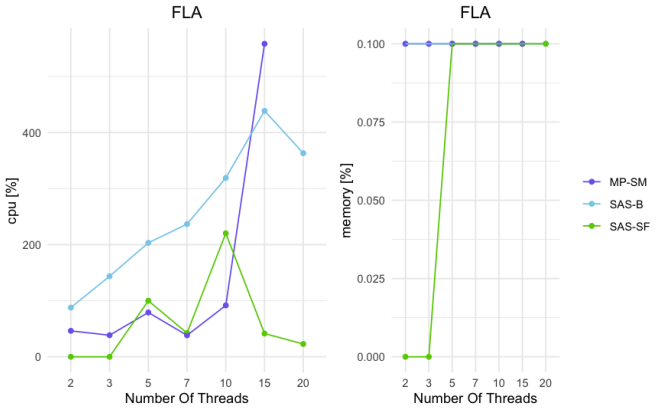

Figure 25: HDA* on FLA map - resources

REFERENCES

[1] A parallel bidirectional heuristic search algorithm. (n.d.). Retrieved September 7, 2022, from https://homepages.dcc.ufmg.br/~chaimo/public/ENIA11.pdf

[2] Parallel A* graph search - massachusetts institute of technology. (n.d.). Retrieved September 7, 2022, from https://people.csail.mit.edu/rholladay/docs/parallel_search_report.pdf

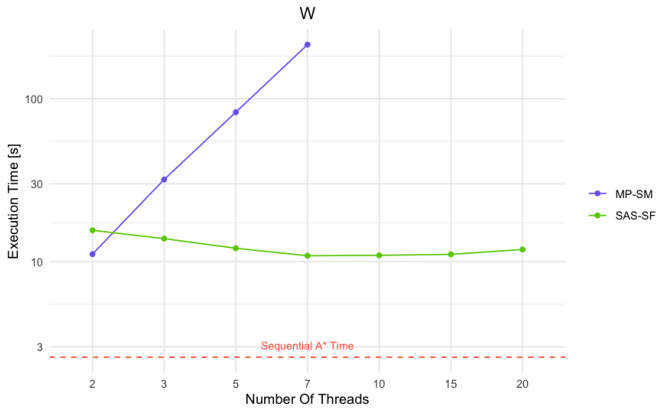

Figure 26: HDA* on W map - time

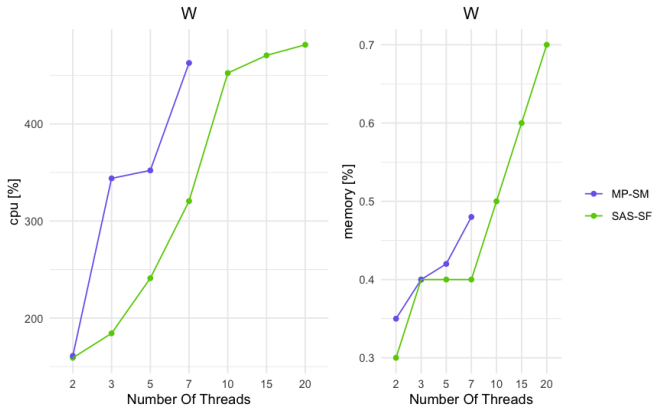

Figure 27: HDA* on W map - resources

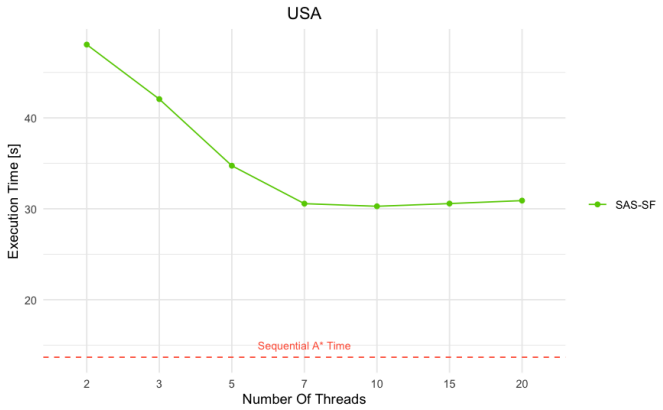

Figure 28: HDA* on USA map - time

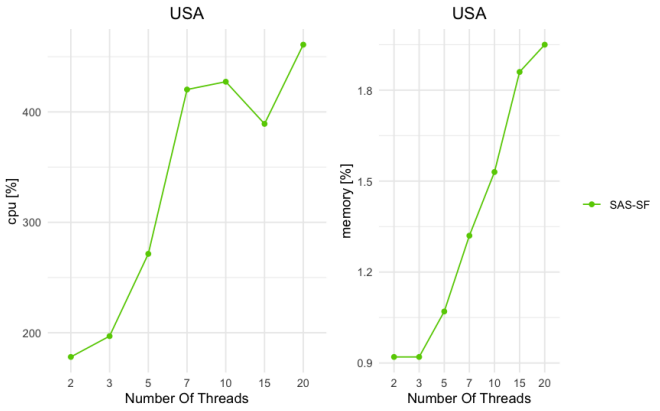

Figure 29: HDA* on USA map - resources

[3] Heuristics. (n.d.). Retrieved September 7, 2022, from http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

[4] Wikimedia Foundation. (2022, August 18). A* search algorithm. Wikipedia. Retrieved September 7, 2022, from https://en.wikipedia.org/wiki/A*_search_algorithm

[5] Proceedings of the twenty-second international joint conference - IJCAI. (n.d.). Retrieved September 7, 2022, from https://www.ijcai.org/Proceedings/11/Papers/105.pdf