

# Parallel A\* Project

## System And Device Programming

Lorenzo Ippolito, Fabio Mirto, Mattia Rosso

Politecnico di Torino

September 10, 2022

# Outline I

## 1 Introduction: about the A\* algorithm

- Problem definition
- Heuristic design

## 2 A\* project application

- Geographical pathfinding
- Problem-specific heuristic function

## 3 Graph file structure

- File input format
- DIMACS benchmark
- Selected benchmark paths

## 4 A\* sequential algorithm

- Data structures
- Pseudocode

## 5 A\* and Dijkstra

- Expanded nodes

# Outline II

- Time and resources comparison

## 6 Parallel input file reading

- Approach 1 (RA1)
- Results on FLA map
- Approach 2 (RA2)
- Results on FLA map
- RA1 and RA2 on FLA map
- Results on all maps

## 7 Parallel A\*

- HDA\*
  - The hash function
  - Work distribution on BAY map
  - Distributed termination condition
  - Communication methodology
  - Message Passing Model (MP)
  - Shared Address Space (SAS)

# Outline III

- SAS pseudocode
- Results
- Comments and overall results
- PNBA\*
  - NBA\*
  - PNBA\*
  - PNBA\* pseudocode

## 8 Conclusions

## 9 References

# Outline

- 1 Introduction: about the A\* algorithm
- 2 A\* project application
- 3 Graph file structure
- 4 A\* sequential algorithm
- 5 A\* and Dijkstra
- 6 Parallel input file reading
- 7 Parallel A\*
- 8 Conclusions
- 9 References

# Introduction: about the A\* algorithm

## Problem definition

- A\* is a graph-traversal and path-search algorithm 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

# Introduction: about the A\* algorithm

## Problem definition

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)

# Introduction: about the A\* algorithm

## Problem definition

According to the standard terminology:

- $g(n)$ : exact cost of moving from source to  $n$
- $h(n)$ : heuristic estimated cost of moving from a node  $n$ (source included) to the destination
- $f(n) = g(n) + h(n)$ : in this way we are able to combine the actual cost with the estimated one

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

# Introduction: about the A\* algorithm

## Heuristic design: properties

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

### Heuristic function properties

- Admissibility:  $h(n) < \text{cost}(n, \text{dest}) \forall n \in V$
- Consistency:  $h(x) \leq \text{cost}(x, y) + h(y) \forall (x, y) \in E$

# Introduction: about the A\* algorithm

Heuristic design: corner cases

Three relevant situations are:

- **Dijkstra**: if  $h(n) = 0$  for every node in the graph.
- **Ideal heuristic**: if  $h(n)$  is exactly equal to the cost of moving from  $n$  to the destination.
- **Full greedy-best-first search**: if  $h(n) \gg g(n)$  than only  $h(n)$  plays a role.

# Outline

- 1 Introduction: about the A\* algorithm
- 2 A\* project application
- 3 Graph file structure
- 4 A\* sequential algorithm
- 5 A\* and Dijkstra
- 6 Parallel input file reading
- 7 Parallel A\*
- 8 Conclusions
- 9 References

# A\* project application

## Geographical pathfinding

We will work with a weighted oriented graph  $G$  that is made of nodes  $n \in V$  that represents 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.

## 2. A\* project application

### The great-circle distance

We will apply the Haversine formula to compute the distance from node  $(\phi_1, \lambda_1)$  to node  $(\phi_2, \lambda_2)$  where  $\phi$  is the latitude and  $\lambda$  is the longitude:

#### Haversine Formula

$$d = R \cdot c$$
$$c = 2 \cdot \text{atan2}(\sqrt{a}, \sqrt{1 - a})$$
$$a = \sin^2\left(\frac{\Delta\phi}{2}\right) + \cos(\phi_1) \cdot \cos(\phi_2) \cdot \sin^2\left(\frac{\Delta\lambda}{2}\right)$$
$$R = 6.371\text{km}$$

# Outline

- 1 Introduction: about the A\* algorithm
- 2 A\* project application
- 3 Graph file structure
- 4 A\* sequential algorithm
- 5 A\* and Dijkstra
- 6 Parallel input file reading
- 7 Parallel A\*
- 8 Conclusions
- 9 References

# Graph file structure

## File input format

The format the file we have worked with is:

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

# Graph file structure

## DIMACS benchmark

The benchmark files we have used come from the 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

We have properly converted these files to obtain binary files with the previously described format

# Graph file structure

## Selected benchmark paths

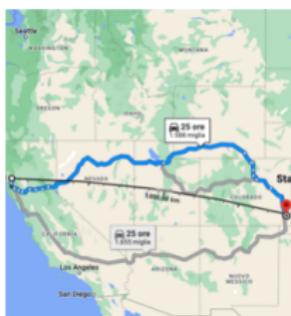
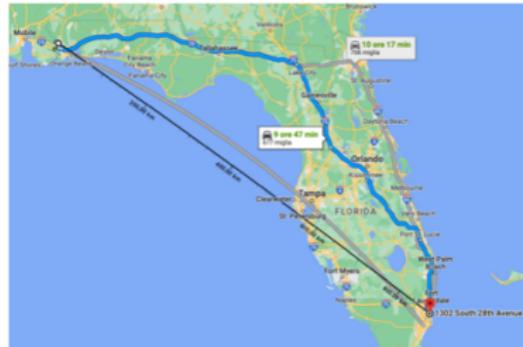
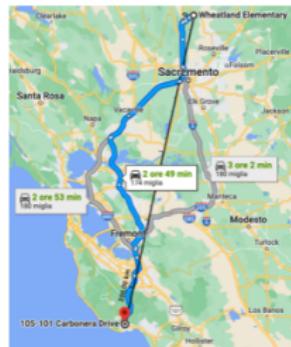


Figure: From left to right top to bottom BAY, FLA, W, USA

# Outline

1 Introduction: about the A\* algorithm

2 A\* project application

3 Graph file structure

4 A\* sequential algorithm

5 A\* and Dijkstra

6 Parallel input file reading

7 Parallel A\*

8 Conclusions

9 References

# A\* sequential algorithm

## Data structures

The first step consists of a pre-computation of:

- The heuristic  $h(n)$  for each node computed through the Haversine formula.
- 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  $f(source) = h(source)$  and  $g(source) = 0$ .

Relevant data structures are:

- $costToCome$ :  $costToCome[i]$  contains the current best cost to reach node  $i$
- $parentVertex$ :  $parentVertex[i]$  contains the parent node of node  $i$  according to the current best path found to reach the destination
- $openSet$ : nodes that have to be explored (implemented as a Priority Queue with  $f(n)$  as priority)
- $closedSet$ : already explored nodes

# A\* sequential algorithm I

## Pseudocode

```
1: function astar( $G$ ,  $source$ ,  $dest$ ,  $h$ )
2:    $g[i] \leftarrow DOUBLE\_MAX \forall i \in V$ 
3:    $f[i] \leftarrow DOUBLE\_MAX \forall i \in V$ 
4:    $h[i] \leftarrow h(i, d) \forall i \in V$ 
5:    $parentVertex[i] \leftarrow -1 \forall i \in V$ 
6:    $f[source] \leftarrow h[source]$ 
7:    $g[source] \leftarrow 0$ 
8:    $openSet := \{(source, f[source])\}$ 
9:   while  $!openSet.EMPTY()$  do
10:     $a \leftarrow openSet.POP()$ 
11:    if  $a == dest$  then
12:      reconstructPath()
13:      return
14:    end if
```

# A\* sequential algorithm II

## Pseudocode

```
15:   if  $a \in closedSed$  then
16:     continue
17:   end if
18:   closedSed.PUSH( $a$ )
19:   for all neighbors  $b$  of  $a$  do
20:     if  $b \in closedSed$  then
21:       continue
22:     end if
23:      $wt \leftarrow weight(a, b)$ 
24:      $tentativeScore \leftarrow g[a] + wt$ 
25:     if  $tentativeScore < g[b]$  then
26:        $parentVertex[b] \leftarrow a$ 
27:        $costToCome[b] \leftarrow wt$ 
28:        $g[b] \leftarrow tentativeScore$ 
```

# A\* sequential algorithm III

## Pseudocode

```
29:            $f[b] \leftarrow g[b] + h[b]$ 
30:           openSet.PUSH((b, f[b]))
31:       end if
32:   end for
33: end while
34: end function
```

## 4. Sequential A\* Algorithm

### Results

Table: 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%

The reading phase has a high impact on the total execution time

# Outline

1 Introduction: about the A\* algorithm

2 A\* project application

3 Graph file structure

4 A\* sequential algorithm

5 A\* and Dijkstra

6 Parallel input file reading

7 Parallel A\*

8 Conclusions

9 References

# A\* and Dijkstra

## Expanded nodes

Table: Expanded nodes in different maps

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

# A\* and Dijkstra

## Expanded nodes

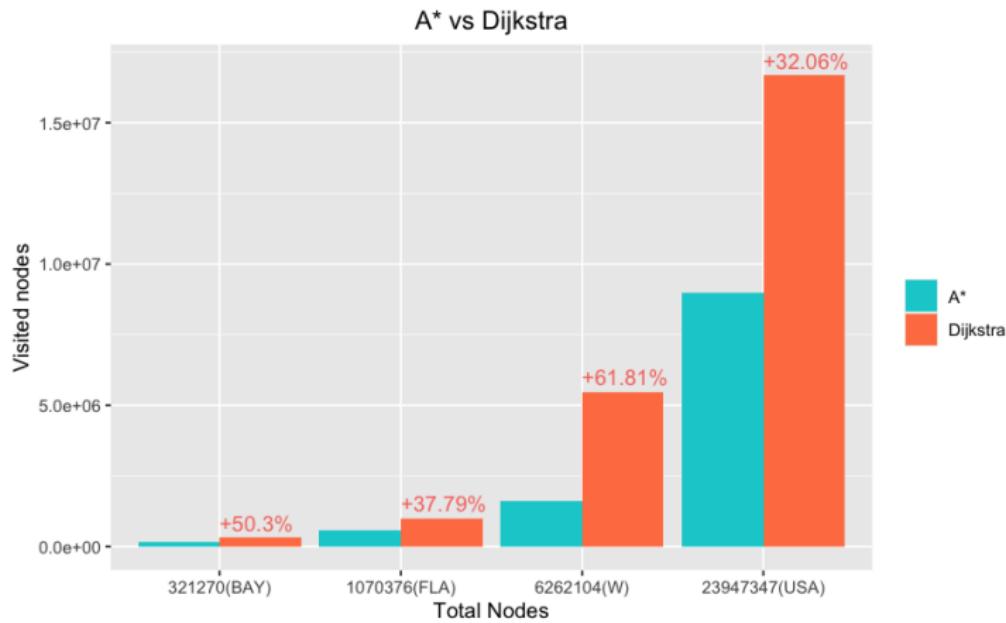


Figure: Expanded nodes: A\* vs Dijkstra

TODO We can notice that...

# A\* and Dijkstra

Expanded nodes

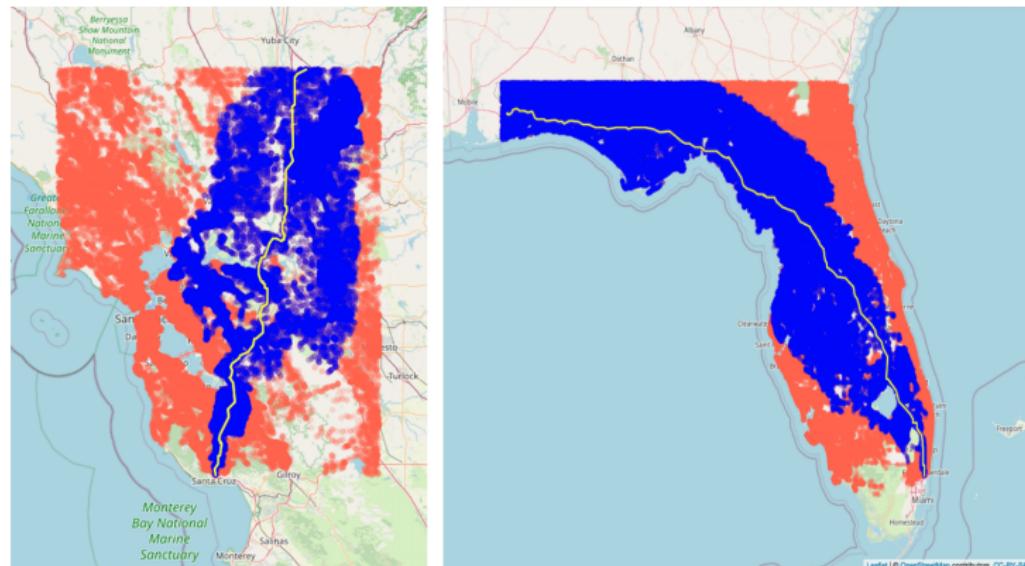


Figure: Test paths on BAY(left) and FLA(right)

# A\* and Dijkstra

## Time and resources comparison

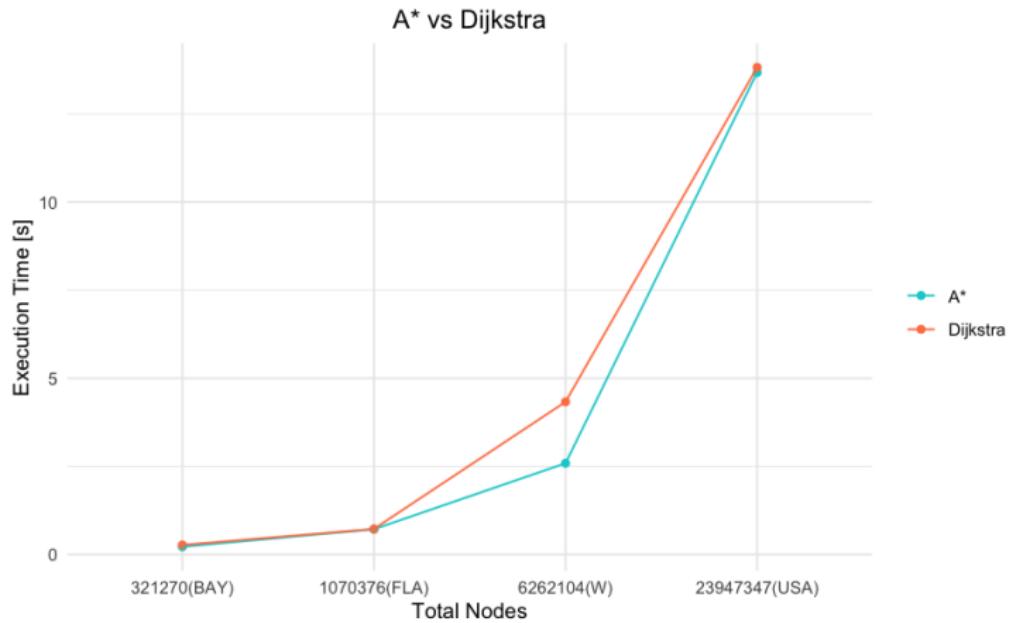


Figure: Execution time: A\* vs Dijkstra

# A\* and Dijkstra

## Time and resources comparison

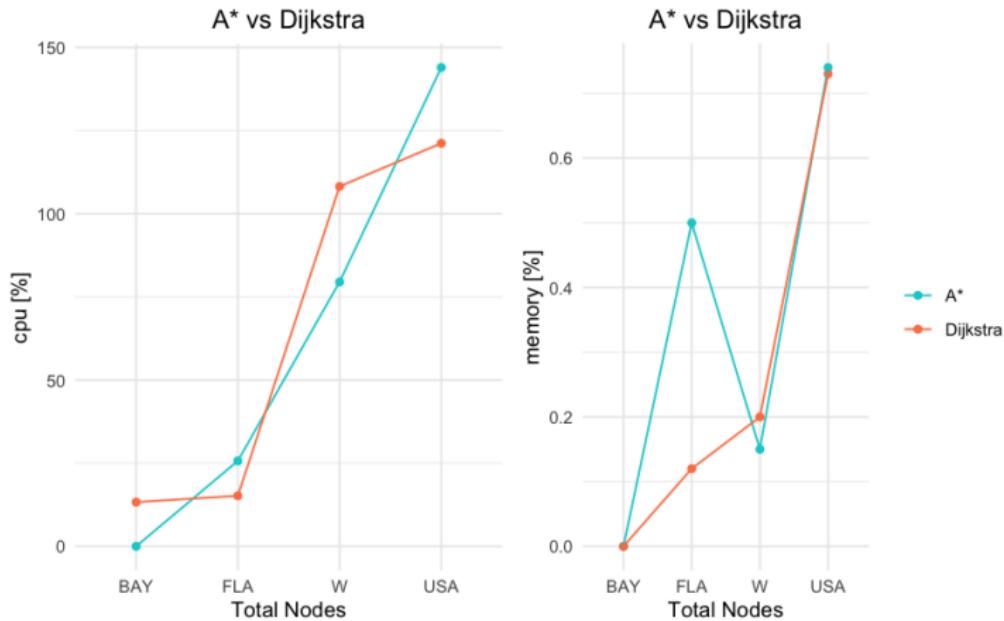


Figure: Exploited resources: A\* vs Dijkstra

# Outline

1 Introduction: about the A\* algorithm

2 A\* project application

3 Graph file structure

4 A\* sequential algorithm

5 A\* and Dijkstra

6 Parallel input file reading

7 Parallel A\*

8 Conclusions

9 References

# Parallel input file reading

## Approach 1 (RA1)

This reading approach 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

# Parallel input file reading

Results on FLA map

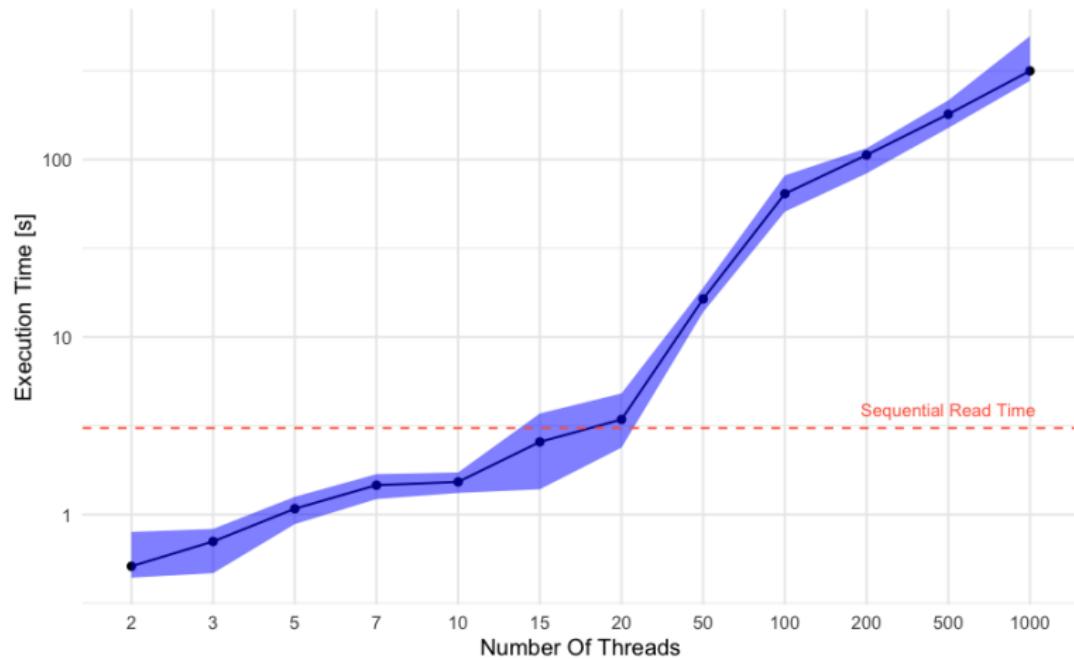


Figure: Performance of RA1 for different number of threads

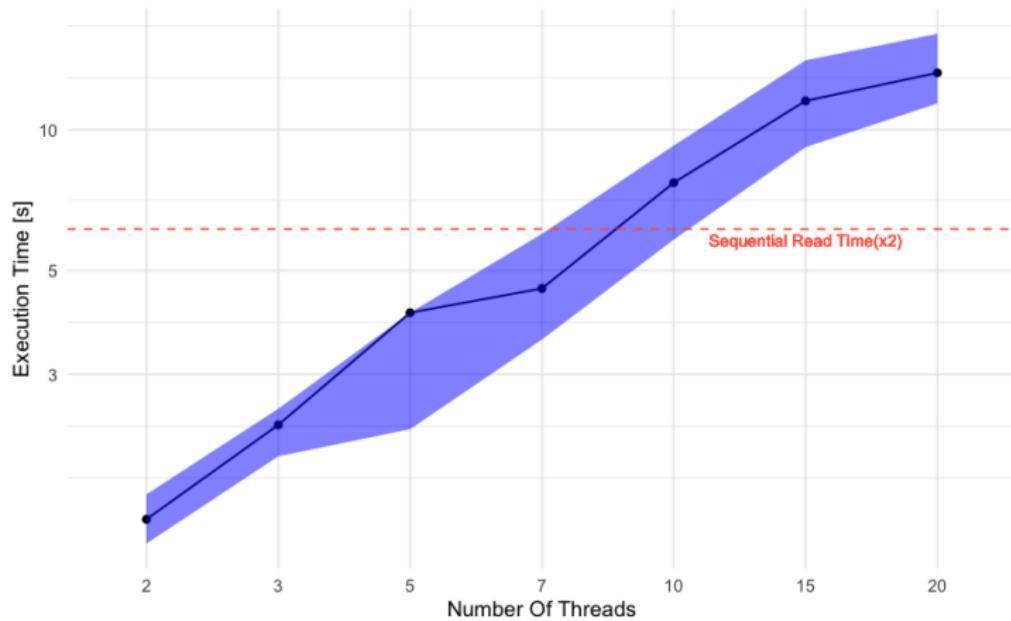
# Parallel input file reading

## Approach 2 (RA2)

This is not properly a different approach of parallel reading but simply a version of the *RA1* needed when the *PNBA\** algorithm will be applied. Since we are loading two graphs data structures at the same time the performances will be compared with the sequential reading as if it was run twice (the first time to read  $G$  and the second time to read  $R$ )

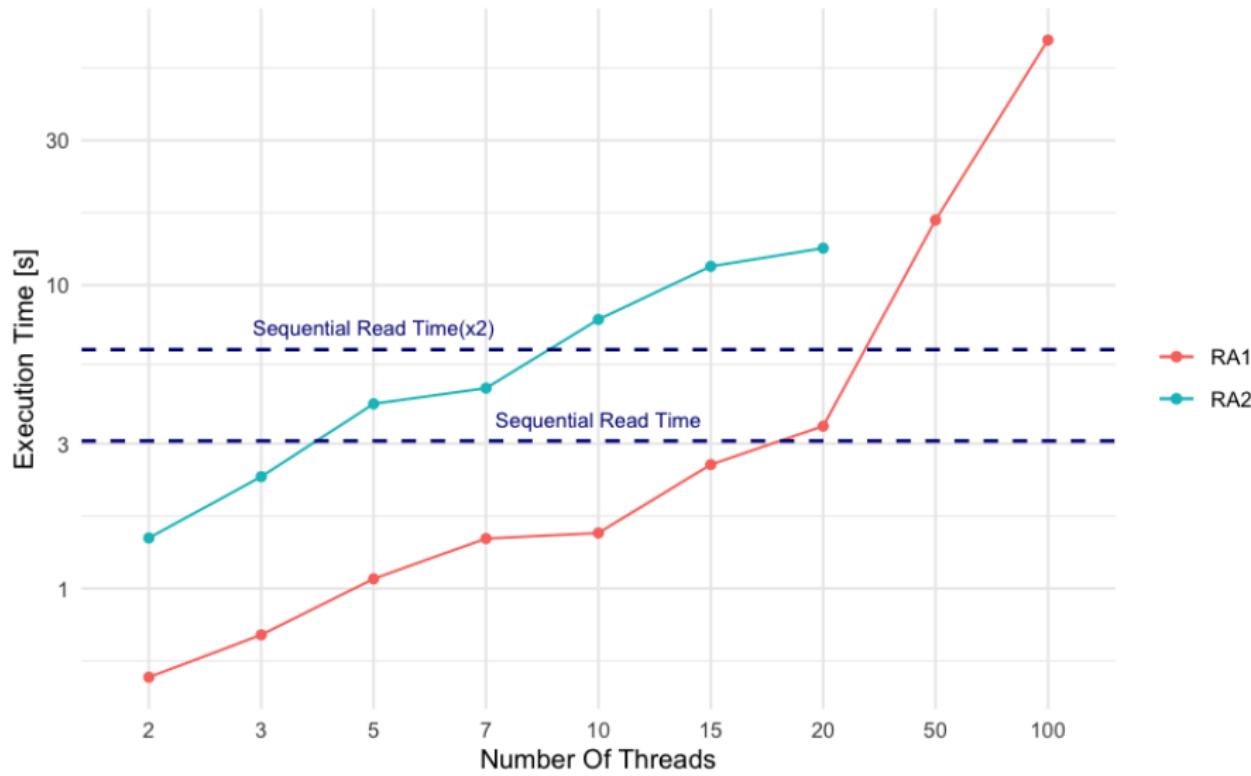
# Parallel input file reading

Results on FLA map



# Parallel input file reading

RA1 and RA2 on FLA map



# Parallel input file reading

Results on all maps

Table: RA1 results against sequential reading

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

# Parallel input file reading

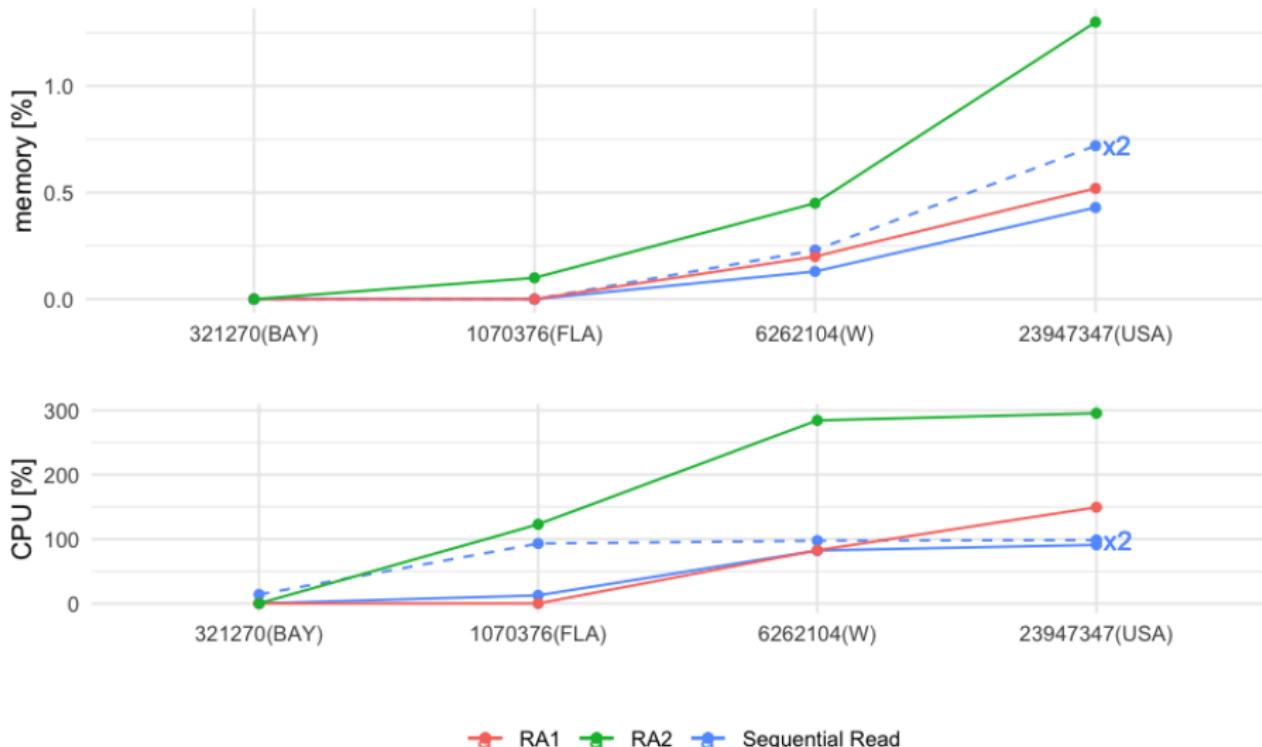
Results on all maps

Table: RA2 results against sequential reading

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

# Parallel input file reading

Results on all maps



# Outline

- 1 Introduction: about the A\* algorithm
- 2 A\* project application
- 3 Graph file structure
- 4 A\* sequential algorithm
- 5 A\* and Dijkstra
- 6 Parallel input file reading
- 7 Parallel A\*
- 8 Conclusions
- 9 References

# 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 followed two approaches to face this problem::

- **Hash Distributed A\* (HDA\*)**: it puts in action a complex way of parallelizing A\* by defining a hash-based work distribution strategy.
- **Parallel New Bidirectional A\* (PNBA\*)**: parallel search of the path from *source* to *dest* and of the path from *dest* to *source* in the reversed graph.

# Parallel A\*: HDA\*

- *Ownership*: HDA\* work is based on the fact that each thread is *owner* of a specific set of nodes of the Graph -*i* 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.

## Parallel A\*: HDA\*

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.

# Parallel A\*: HDA\*

## The hash function

We have implemented two types of hash functions:

### Hash functions definitions

$$\text{hash}_1(\text{node\_index}, \text{num\_threads}) = \text{node\_index \% num\_threads}$$

$$\text{hash}_2(\text{node\_index}, \text{num\_threads}, V) = i - 1 \text{ with } i = \min_i : \frac{V}{\text{num\_threads}} \cdot i > \text{node\_index} , i \in \{1, \dots, \text{num\_threads}\}$$

- The first one simply assigns a node to a thread in a randomic fashion w.r.t to its position inside the map
- 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).

# Parallel A\*: HDA\*

Work distribution on BAY map

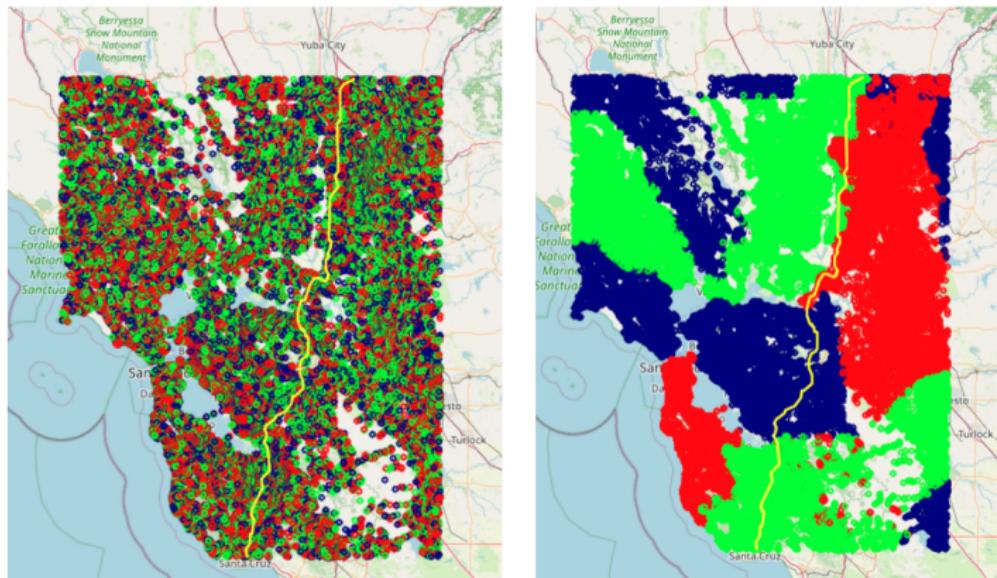


Figure: HDA\* work distribution among 3 threads in BAY map

# Parallel A\*: HDA\*

## Distributed termination condition

- **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  $\sum_{i=1}^N \text{flag}[i] = N$  all the threads can correctly terminate.

# Parallel A\*: HDA\*

## Communication methodology

- **Shared Address Space (SAS)**: threads share pointers to common data structures (we need to cope with mutual exclusion)
- **Message Passing (MP)**: threads can communicate only through messages via:
  - ▶ Message Queues
  - ▶ Shared Memory

# Parallel A\*: HDA\*

## MP using Shared Memory (MPSM)

We don't report here all the details regarding Message Passing implemented using Linux Message Queues because of its lack of scalability. About MPSM:

- Some data structures are shared: *parentVertex*, *costToCome*
- The exchange of information about discovered nodes that have to be explored is done using Linux Shared Memory.
- This has the great advantage of minimizing the resource contention among threads (something that will strongly penalize the SAS model).
- The termination condition that was implemented is Sum-Flag(SF)

# Parallel A\*: HDA\*

## MP using Shared Memory (MPSM)

Each message 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 node of  $n$
- The value of  $g[n]$  according to thread  $t_i$

The way how messages are read and written by the different threads is done adopting the *Readers & 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  $smp_j == smp_G$ .

# Parallel A\*: HDA\*

## MP using Shared Memory (MPSM)

The termination condition we have implemented is Sum-Flag method.

- A *globalDestCost* variable is combined with the distributed termination condition in order to know whether threads can correctly stop (an alternative was to check whether *parentVertex[dest] == -1*).
- *Readers & Writers* mechanism was implemented by using Linux Semaphores.

# Parallel A\*: HDA\* I

MPSM pseudocode

```
1: function hda_mp_sm
2:    $smp_L \leftarrow smp_G$ 
3:   while 1 do
4:     while  $smp_L \neq smp_G$  do
5:       wait(meR)
6:        $nR++$ 
7:       if  $nR == 1$  then
8:         wait(w)
9:       end if
10:      post(meR)
11:      if  $hash_2(index, N, V) == index$  then
12:        if  $smp_L.g < g[smp_L.n]$  then
13:           $g[smp_L.n] \leftarrow smp_L.g$ 
14:           $f[smp_L.n] \leftarrow g[smp_L.n] + h[smp_L.n]$ 
```

# Parallel A\*: HDA\* II

## MPSM pseudocode

```
15:           parentVertex[smpL.n] ← smpL.previous
16:           costToCome[smpL.n] ← smpL.wt
17:           openSet.PUSH({f[smpL.n], smpL.n})
18:       end if
19:   end if
20:   smpL ++
21:   wait(meR)
22:   nR --
23:   if nR == 0 then
24:       post(w)
25:   end if
26:   post(meR)
27: end while
28: if openSet.EMPTY() and globalDestCost < ∞ then
```

# Parallel A\*: HDA\* III

## MPSM pseudocode

```
29:           Terminate according to SF
30:    end if
31:    if openSet.EMPTY() then
32:         $a \leftarrow openSet.POP()$ 
33:        if  $a == dest$  then
34:            LOCK
35:             $globalDestCost \leftarrow g[a]$ 
36:            UNLOCK
37:        end if
38:        for neighbor  $b$  of  $a$  do
39:             $owner\_b \leftarrow hash_2(b, N, V)$ 
40:            if  $owner\_b == index$  then
41:                 $parentVertex[b] \leftarrow a$ 
42:                 $costToCome[b] \leftarrow weight(a, b)$ 
```

# Parallel A\*: HDA\* IV

## MPSM pseudocode

```
43:            $g[b] \leftarrow g[b] + weight(a, b)$ 
44:            $f[b] \leftarrow g[b] + h[b]$ 
45:           openSet.PUSH((b, f[owner_b][b]))
46:       else
47:           wait(w)
48:           smp_G.n  $\leftarrow b$ 
49:           smp_G.g  $\leftarrow g[b]$ 
50:           smp_G.wt  $\leftarrow weight(a, b)$ 
51:           smp_G.previous  $\leftarrow a$ 
52:           smp_G ++
53:           post(w)
54:       end if
55:   end for
56: end if
```

# Parallel A\*: HDA\* V

MPSM pseudocode

57: **end while**

58: **end function**

# Parallel A\*: HDA\*

## Shared Address Space (SAS)

The shared data structures are:

- A global array of *open sets* of size  $N$  (number of threads) where  $\text{openSet}[i]$  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)

# Parallel A\*: HDA\* I

SAS pseudocode

```
1: function hda_sas
2:   while 1 do
3:     while !openSet[index].EMPTY() do
4:       LOCK(mutex_threads[index])
5:       a  $\leftarrow$  openSet.POP()
6:       UNLOCK(mutex_threads[index])
7:     end while
8:     if a is duplicate then
9:       continue
10:    end if
11:    for neighbor b of a do
12:      wt  $\leftarrow$  weight(a, b)
13:      tentativeScore  $\leftarrow$  g[a] + wt
14:      if tentativeScore < g[b] then
```

# Parallel A\*: HDA\* II

## SAS pseudocode

```
15:           owner_a ← hash2(a, N, V)
16:           owner_b ← hash2(b, N, V)
17:       end if
18:       if b is duplicate then
19:           continue
20:       end if
21:       tentativeScore ← g[a] + wt
22:       LOCK(mutex_nodes[b])
23:       if tentativeScore is less than g[b] then
24:           parentVertex[b] ← a
25:           costToCome[b] ← wt
26:           g[b] ← tentativeScore
27:           f[b] ← g[b] + h[b]
28:           LOCK(mutex_threads[owner_b])
```

# Parallel A\*: HDA\* III

SAS pseudocode

```
29:            $f[owner\_b][b] \leftarrow f[b]$ 
30:           openSet.PUSH({b, f[owner_b][b]})
31:           UNLOCK(mutex_threads[owner_b])
32:       end if
33:       UNLOCK(mutex_nodes[b])
34:   end for
35: end while
36: if openSet[index].EMPTY() and parentVertex[dest] ≠ -1
    then
37:     Terminate according to B or SF
38:   end if
39: end function
```

# Parallel A\*: HDA\*

## Results

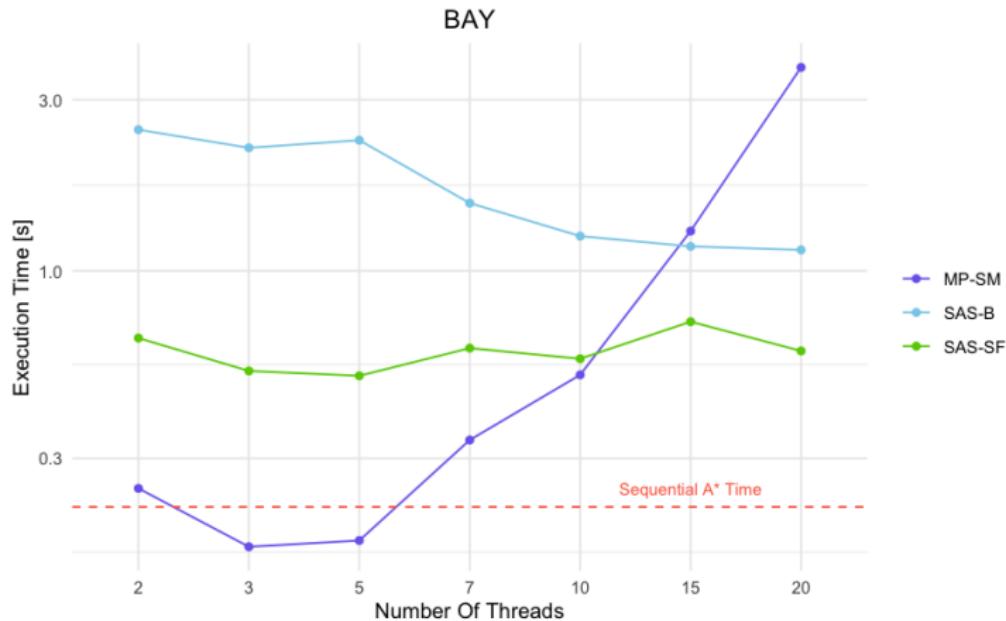


Figure: HDA\* on BAY map - time

# Parallel A\*: HDA\*

## Results

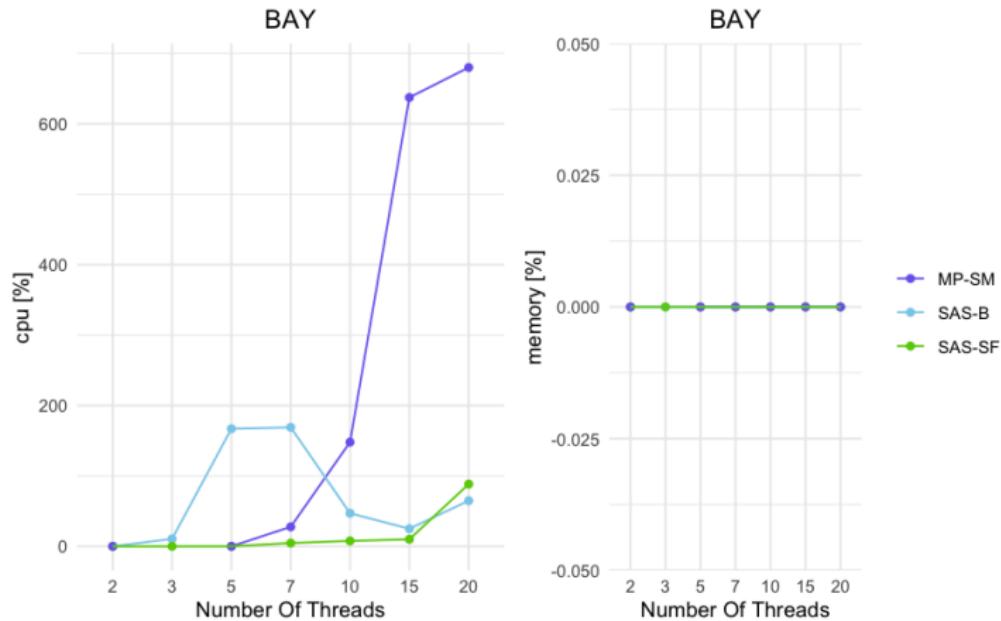


Figure: HDA\* on BAY map - resources

# Parallel A\*: HDA\*

## Results

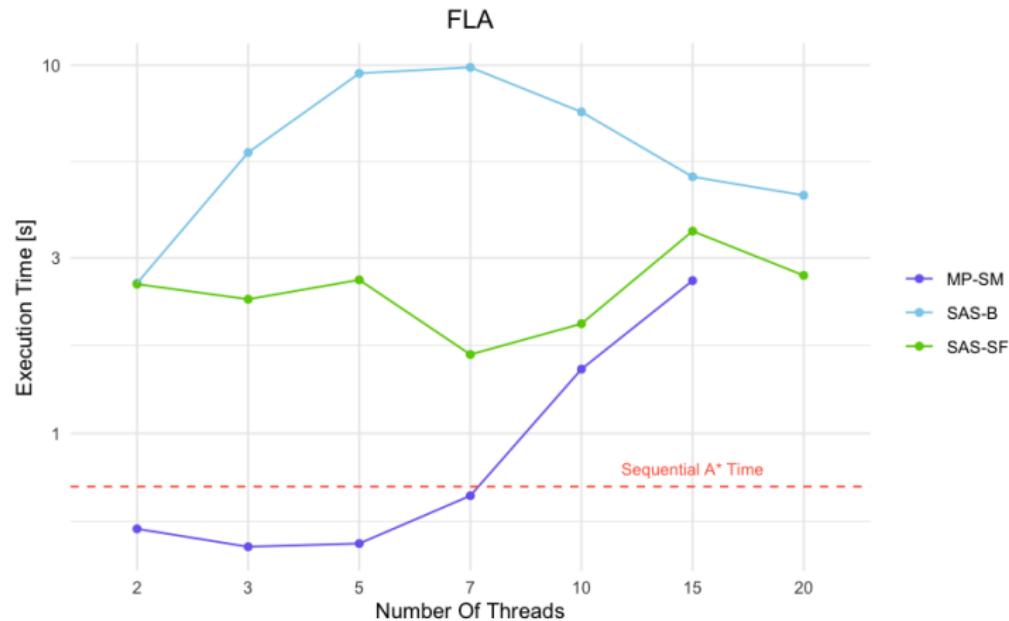


Figure: HDA\* on FLA map - time

# Parallel A\*: HDA\*

## Results

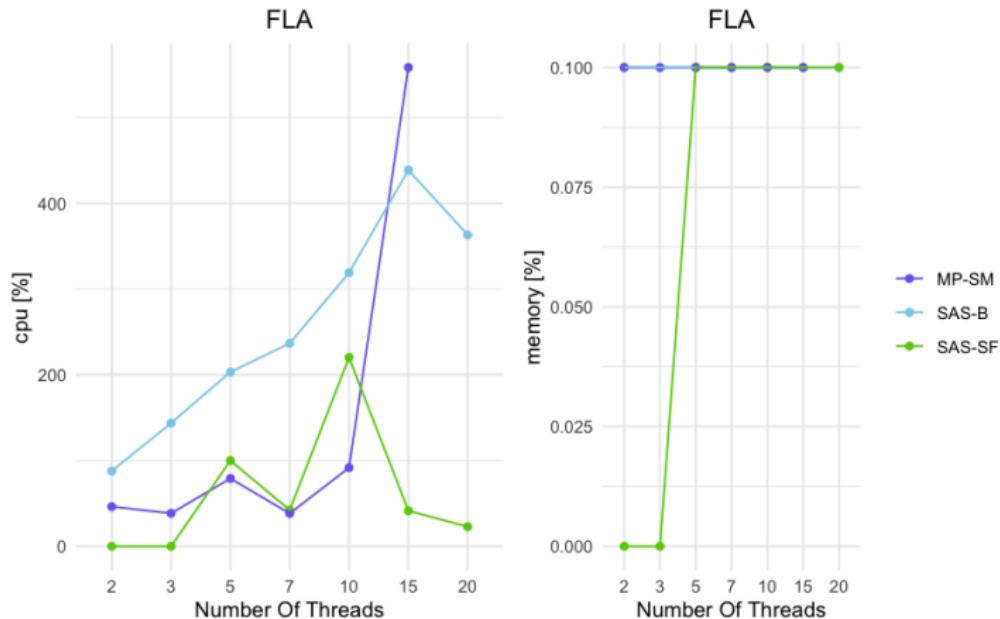


Figure: HDA\* on FLA map - resources

# Parallel A\*: HDA\*

## Results

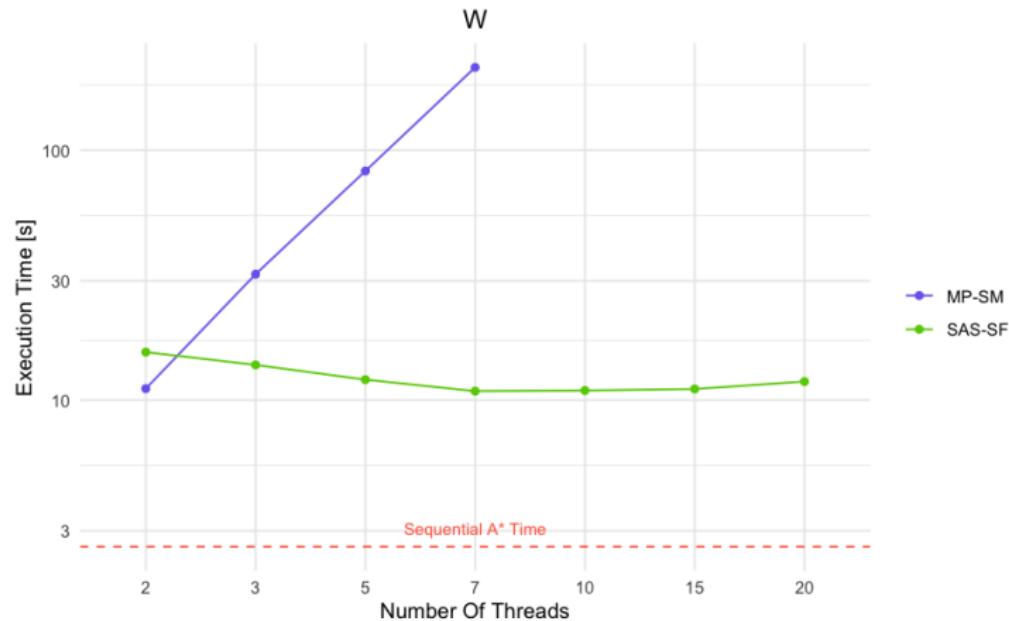


Figure: HDA\* on W map - time

# Parallel A\*: HDA\*

## Results

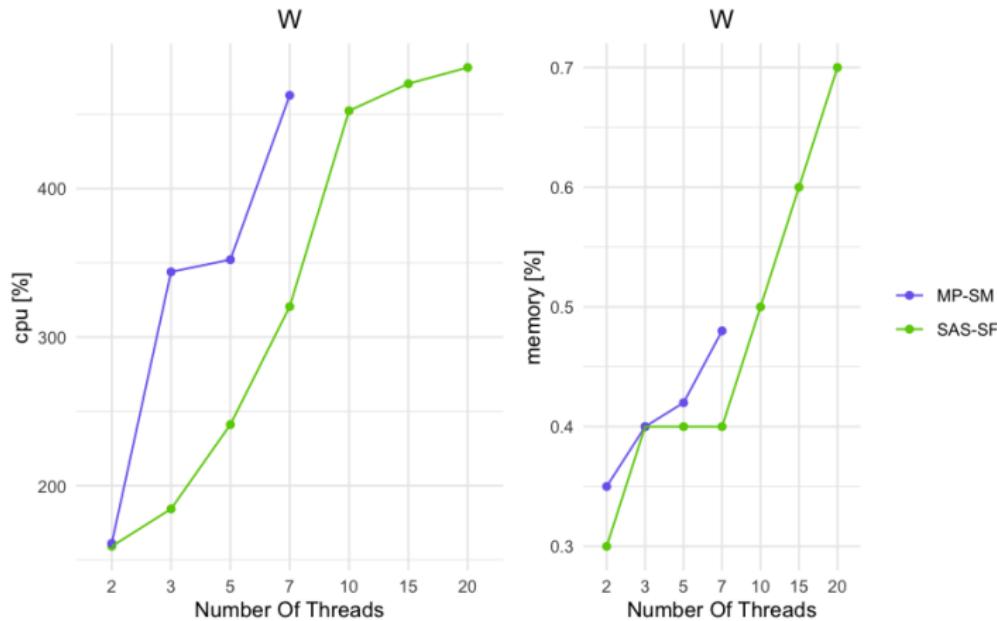


Figure: HDA\* on W map - resources

# Parallel A\*: HDA\*

## Results

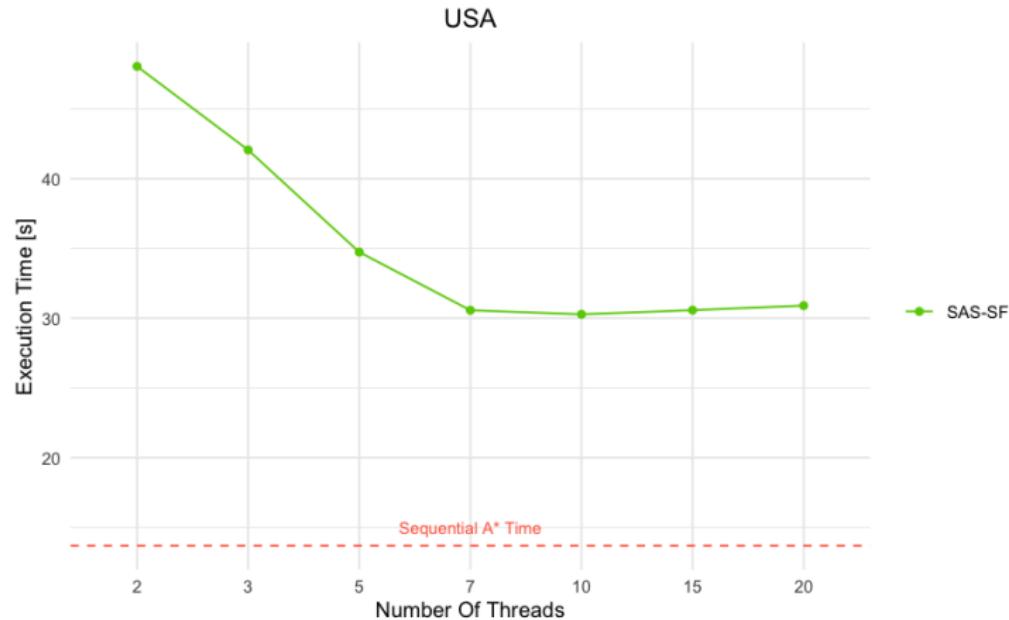


Figure: HDA\* on USA map - time

# Parallel A\*: HDA\*

## Results

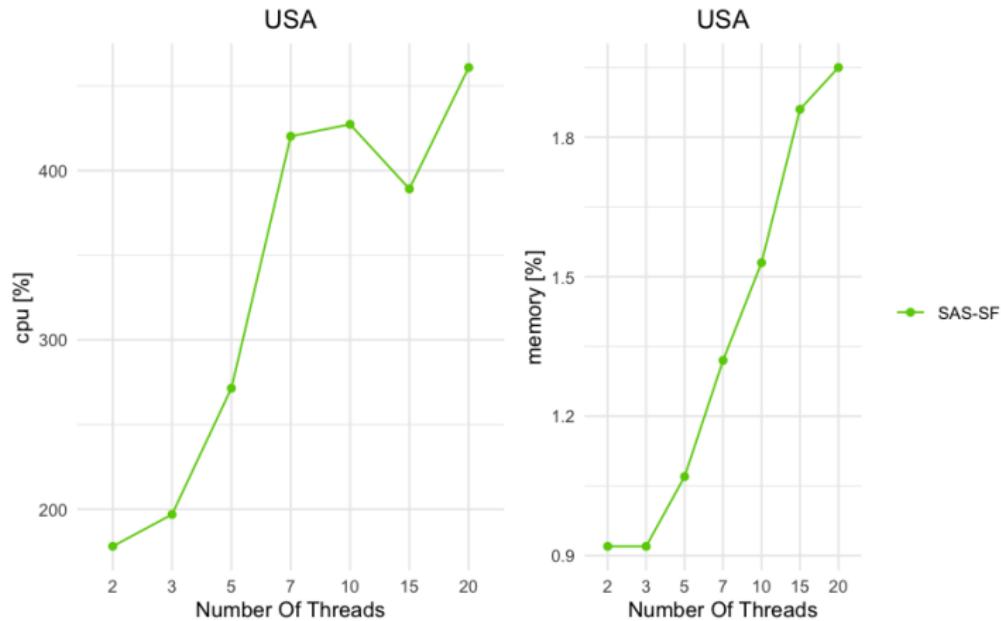


Figure: HDA\* on USA map - resources

# Parallel A\*: HDA\*

## Results

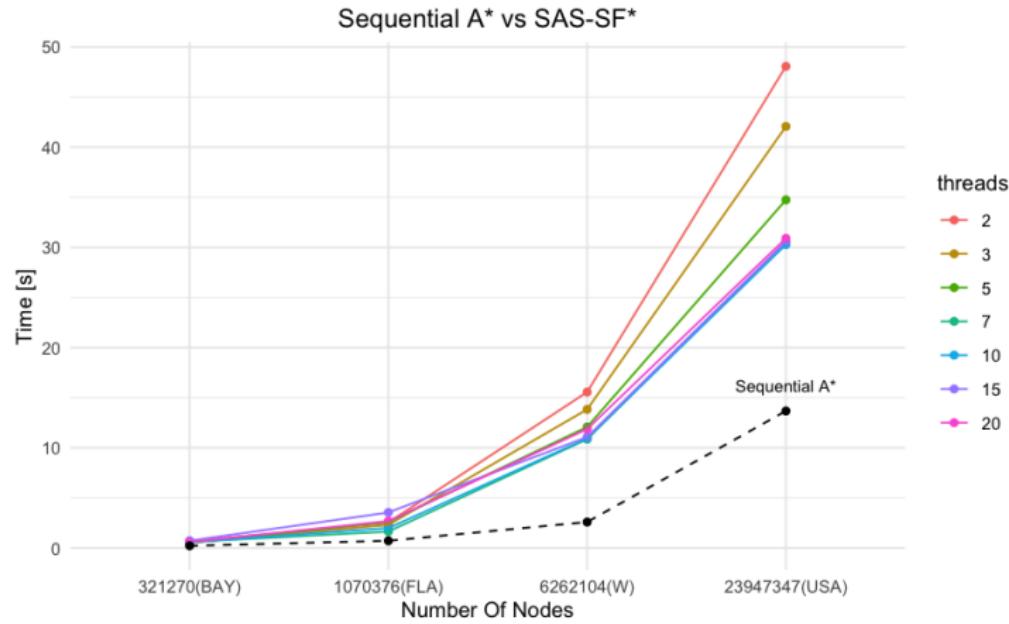


Figure: HDA\* SAS-SF vs Sequential A\* on all maps - time

# Parallel A\*: HDA\*

## Results

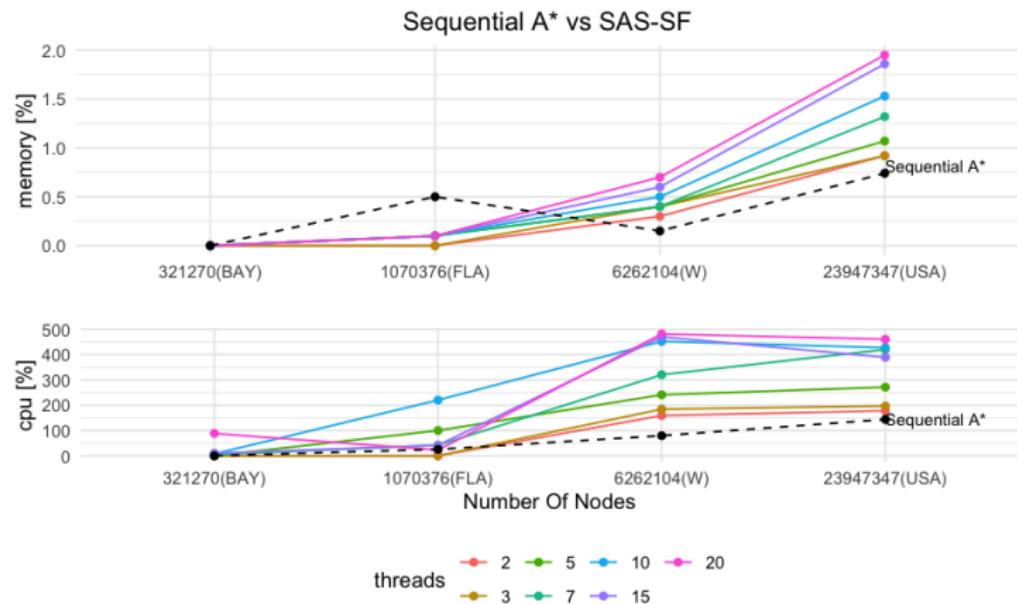


Figure: HDA\* SAS-SF vs Sequential A\* on all maps - resources

# Parallel A\*: HDA\*

## Results

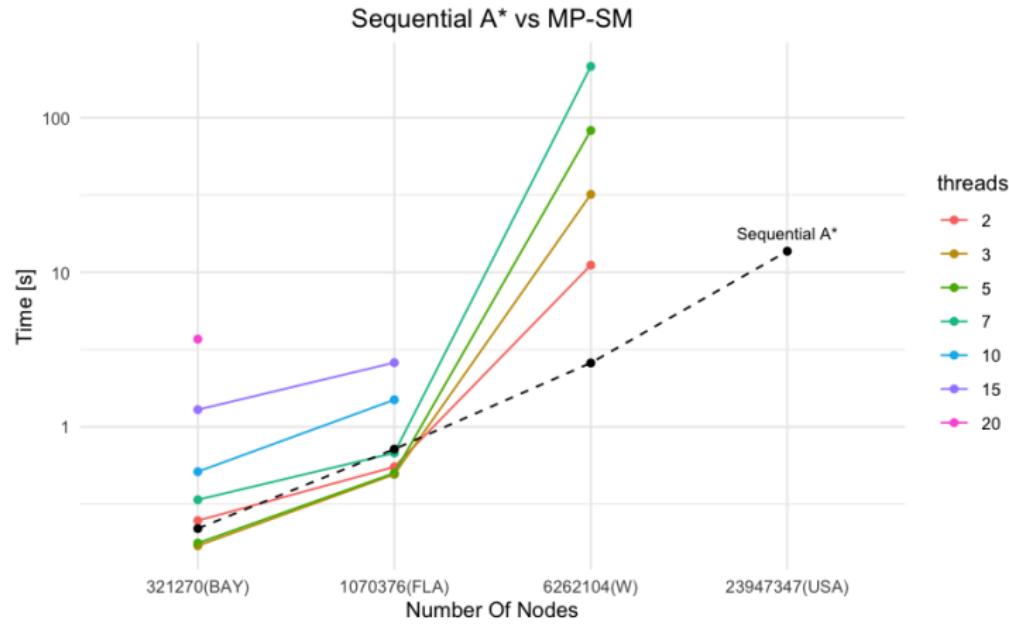


Figure: HDA\* MP-SM vs Sequential A\* on all maps - time

# Parallel A\*: HDA\*

## Results

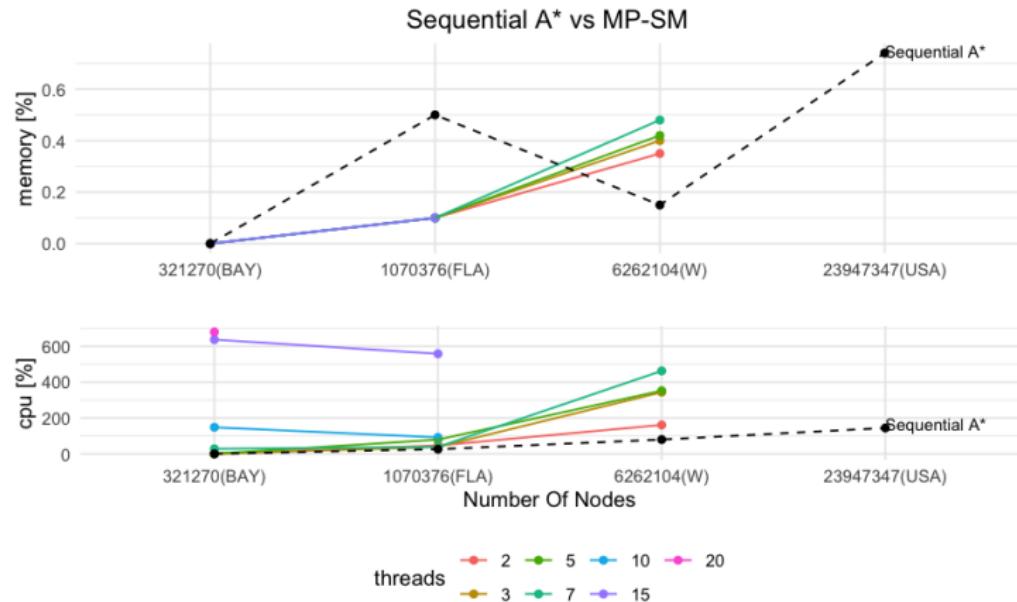


Figure: HDA\* MP-SM vs Sequential A\* on all maps - resources

# Parallel A\*: HDA\*

## Results

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

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

# Parallel A\*: HDA\*

## Results

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

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

# Parallel A\*: HDA\* I

## Comments and overall results

- The MP-MQ variant of HDA\* is not scalable.
- 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. But not fully scalability
- SAS-B 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.
- SAS-SF behaves overall better. Performances are always better compared to the SAS-B algorithm and this is confirmed as the most well-performing distributed termination condition.

# Parallel A\*: HDA\*

## Comments and overall results

HDA\* is not performing really well on our problem. Some reasons could be:

- The hash function has an important impact on the performances of A\* algorithm. The one that we have used is probably not the optimal.
- The termination condition is clearly a bottleneck, in particular if the Barrier method is used.
- 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 resource expensive and not fully scalable.

# Parallel A\*: PNBA\*

## 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  $\infty$  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.

# Parallel A\*: PNBA\*

## NBA\*

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

- $g_G(\text{source}_G) = 0, F_G(\text{source}_G) = f_G(\text{source}_G)$

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

- $x \in M$
- $x : f_G(x) = \min_{v \in \text{openSet}_G} f_G(v)$

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.

# Parallel A\*: PNBA\*

## PNBA\*

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



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

# Parallel A\* I

## PNBA\*

```
1: function pnbaG()  
2:   while !finished do  
3:      $x \leftarrow openSet_G.POP()$   
4:     if  $x \in M$  then  
5:       if ( $f_G(x) < L$ )  $\wedge$  ( $g_G(x) + F_R - h_R(x) < L$ ) then  
6:         for all edges  $(x, y)$  do  
7:           if ( $y \in M$ )  $\wedge$  ( $g_G(x) > g_G(y) + d_G(x, y)$ ) then  
8:              $g_G(y) \leftarrow g_G(x) + d_G(x, y)$   
9:              $f_G(y) \leftarrow g_G(y) + h_G(x, y)$   
10:            if  $y \in openSet_G$  then  
11:               $openSet_G.REMOVE(y)$   
12:            end if  
13:             $openSet_G.PUSH(\{f_G(y), y\})$   
14:            if  $g_G(y) + g_R(y) < L$  then
```

# Parallel A\* II

## PNBA\*

```
15:                                lock
16:                                if  $g_G(y) + g_R(y) < L$  then
17:                                     $L \leftarrow g_G(y) + g_R(y)$ 
18:                                end if
19:                                unlock
20:                            end if
21:                        end if
22:                    end for
23:                end if
24:                 $M \leftarrow M - \{x\}$ 
25:            end if
26:            if ! $openSet_G.EMPTY()$  then
27:                 $F_G \leftarrow f[openSet_G.PEEK()]$ 
28:            else
```

# Parallel A\* III

## PNBA\*

```
29:           finished  $\leftarrow$  true
30:       end if
31:   end while
32: end function
```

# Parallel A\*: PNBA\*

## 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 will try to implement it on much bigger graphs.
- 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

# Parallel A\*: PNBA\*

## Results

Table: PNBA\* - time performances

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

# Parallel A\*: PNBA\*

## Results

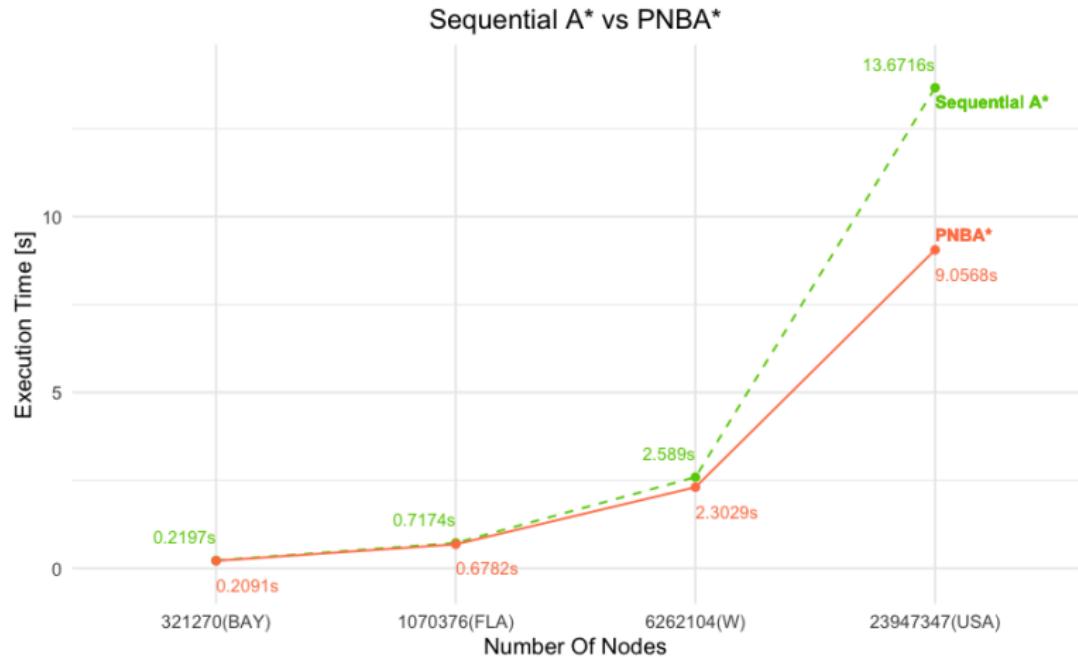


Figure: PNBA\* overall performances

# Parallel A\*: PNBA\*

## Results

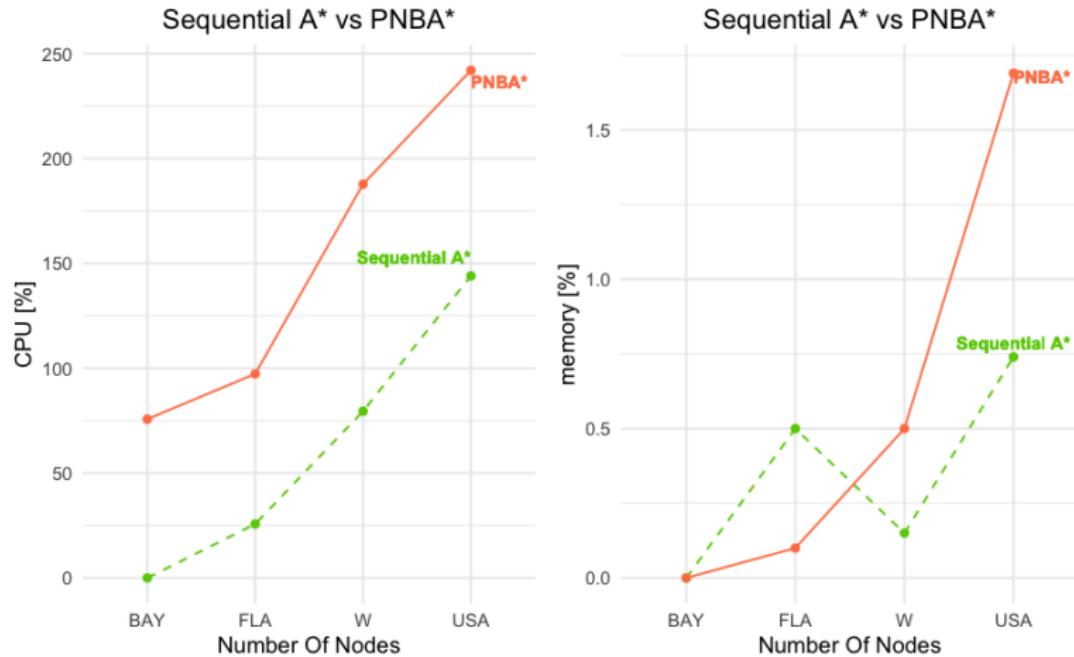


Figure: PNBA\* overall performances - time

# Outline

- 1 Introduction: about the A\* algorithm
- 2 A\* project application
- 3 Graph file structure
- 4 A\* sequential algorithm
- 5 A\* and Dijkstra
- 6 Parallel input file reading
- 7 Parallel A\*
- 8 Conclusions
- 9 References

# Conclusions

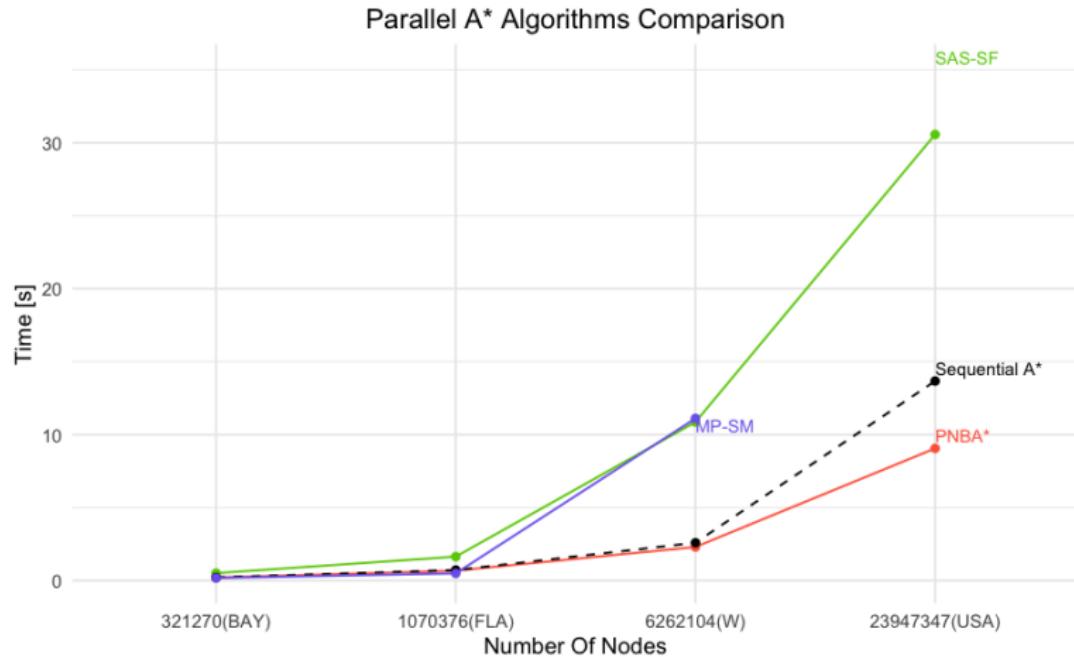


Figure: Parallel A\* overall performances - time

# Conclusions

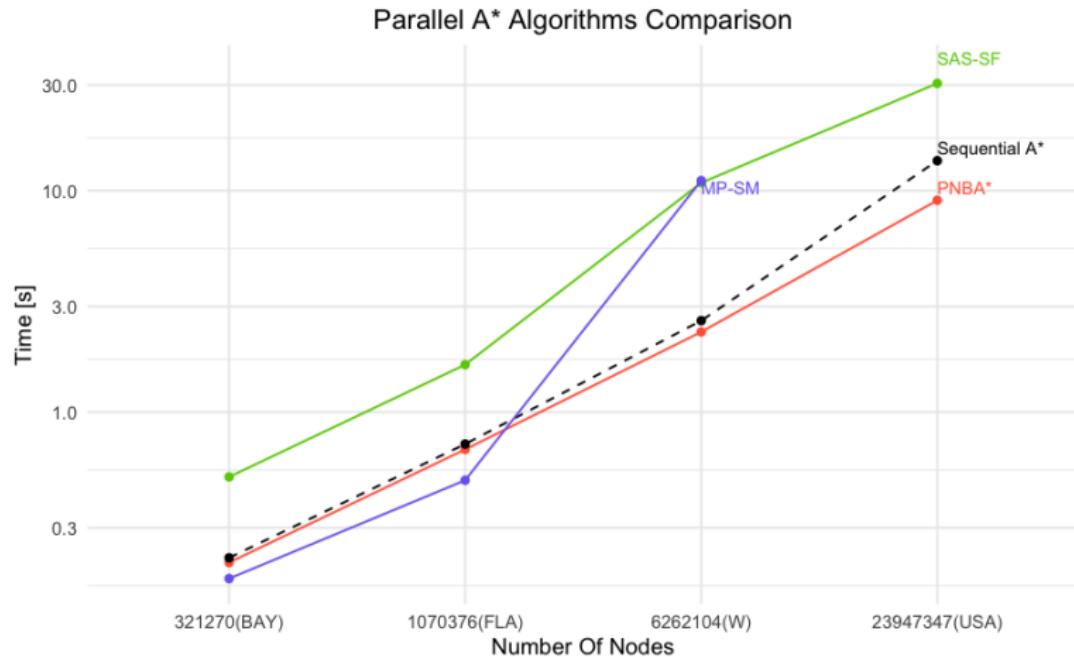


Figure: Parallel A\* overall performances - time (log. scale)

# Conclusions

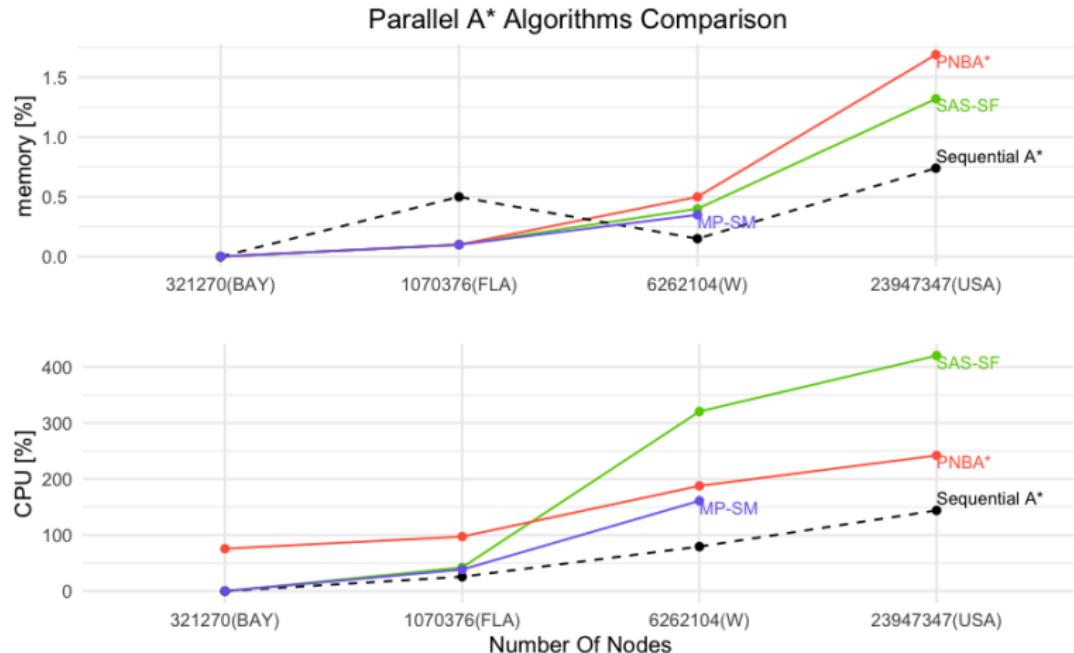


Figure: Parallel A\* overall performances - resources

# Conclusions

- HDA\* with SAS-SF is the one that achieves worst results in terms of execution time and CPU usage.
- 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 achieve better results in terms of execution time on all the benchmark maps we have used. Also reasonable performances are achieved in terms of CPU and memory usage.

# Outline

- 1 Introduction: about the A\* algorithm
- 2 A\* project application
- 3 Graph file structure
- 4 A\* sequential algorithm
- 5 A\* and Dijkstra
- 6 Parallel input file reading
- 7 Parallel A\*
- 8 Conclusions
- 9 References

# References |

-  A parallel bidirectional heuristic search algorithm. (n.d.). Retrieved September 7, 2022, from <https://homepages.dcc.ufmg.br/~chaimo/public/ENIA11.pdf>
-  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](https://people.csail.mit.edu/rholladay/docs/parallel_search_report.pdf)
-  Heuristics. (n.d.). Retrieved September 7, 2022, from <http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>
-  Wikimedia Foundation. (2022, August 18). A\* search algorithm. Wikipedia. Retrieved September 7, 2022, from [https://en.wikipedia.org/wiki/A\\*\\_search\\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm)

## References II

-  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>