Parallel A* Project

System And Device Programming

Lorenzo Ippolito, Fabio Mirto, Mattia Rosso

Politecnico di Torino

September 9, 2022

Outline I

Outline II

Outline III

Outline

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 examinated(expanded).

Introduction: about the A* algorithm

Heuristic design: properties

The heurstic function represents the acutal 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: $h(n) < cost(n, dest)\ \forall n \in V$
- Consistency: $h(x) \leq cost(x, y) + h(y)\ \forall (x, y) \in E$

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

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

Haversine Formula

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

Outline

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 $(index[int], longitude[double], latidue[double])$
- 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

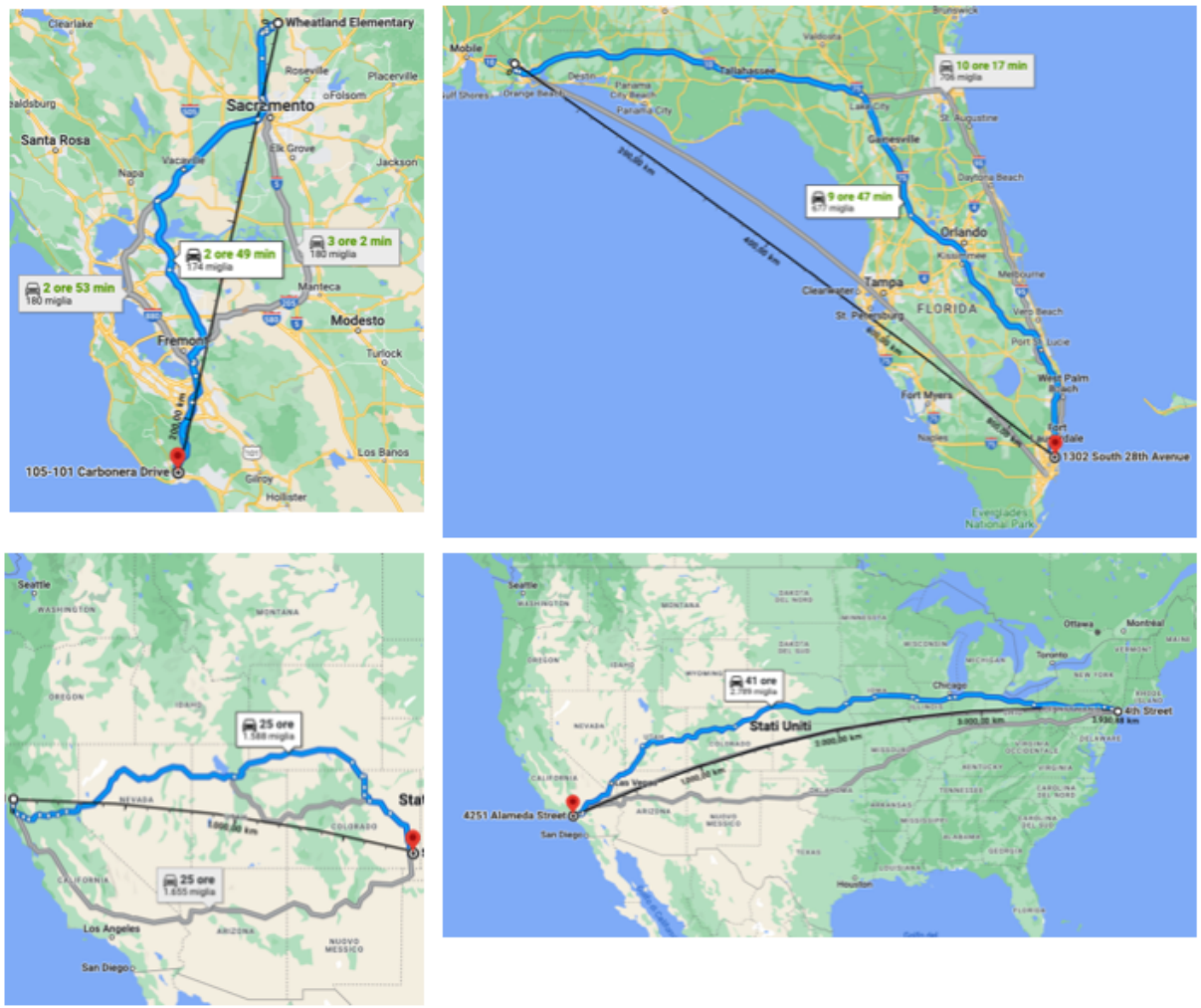


Figure: From left to right top to bottom BAY, FLA, W, USA

Outline

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 explored (implemented as a Prioriy Queue with $f(n)$ as prioriry)
- *closedSet*: already explored nodes

A* sequential algorithm I

Pseudocode

```
1:  function astar(G, source, dest, h)
2:      g[i] ← DOUBLE_MAX ∀i ∈ V
3:      f[i] ← DOUBLE_MAX ∀i ∈ V
4:      h[i] ← h(i, d) ∀i ∈ V
5:      parentVertex[i] ← −1 ∀i ∈ V
6:      f[source] ← h[source]
7:      g[source] ← 0
8:      openSet := {(source, f[source])}
9:      while !openSet.EMPTY() do
10:         a ← openSet.POP()
11:         if a == dest then
12:             reconstructPath()
13:             return
14:         end if
```

A* sequential algorithm II

Pseudocode

```
15:     if a ∈ closedSed then
16:         continue
17:     end if
18:     closedSed.PUSH(a)
19:     for all neighbors b of a do
20:         if b ∈ closedSed then
21:             continue
22:         end if
23:         wt ← weight(a, b)
24:         tentativeScore ← g[a] + wt
25:         if tentativeScore < g[b] then
26:             parentVertex[b] ← a
27:             costToCome[b] ← wt
28:             g[b] ← 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 an high impact on the total execution time

Outline

A* and Dijkstra

Expanded nodes

Table: 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

A* and Dijkstra

Expanded nodes

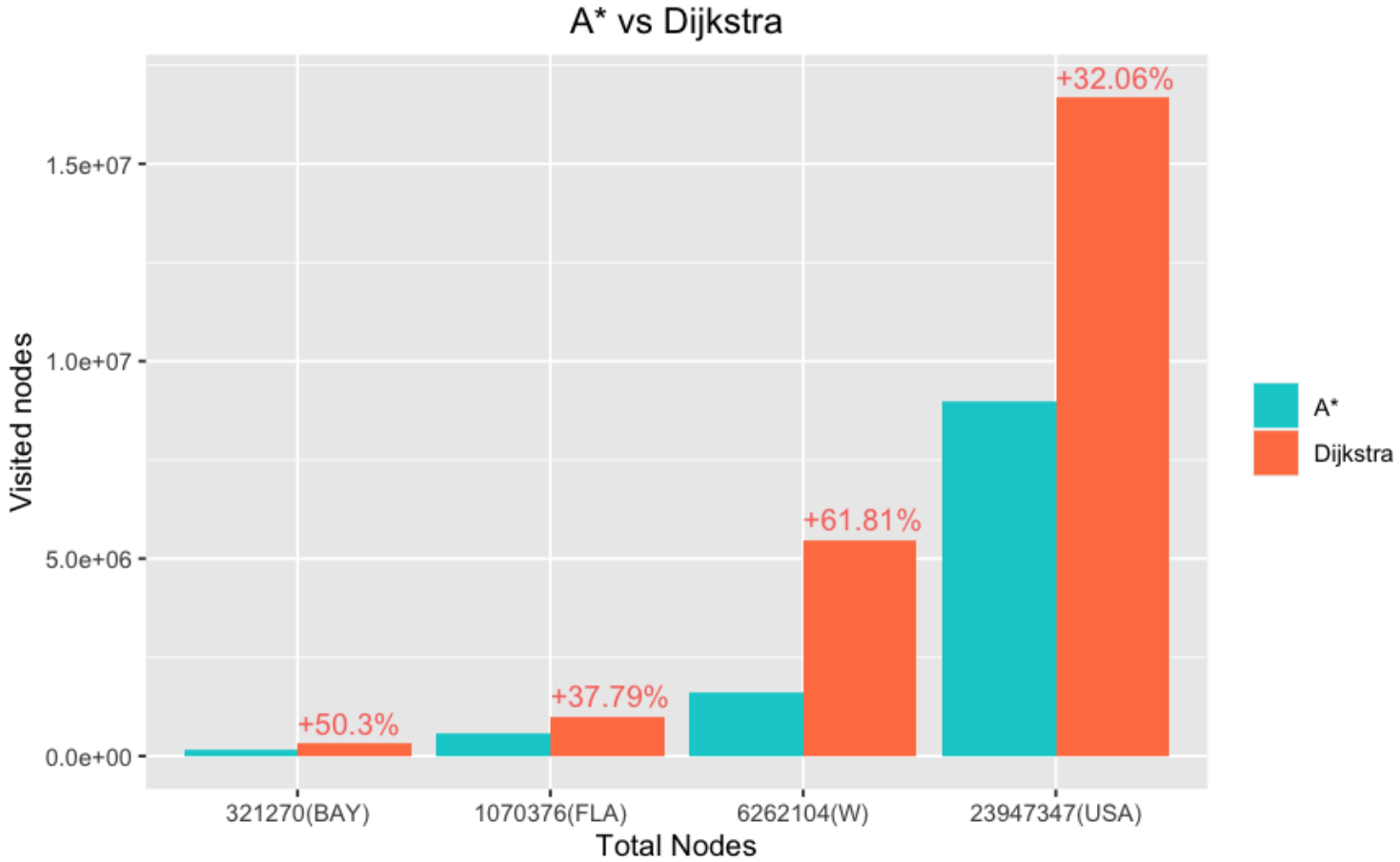


Figure: Expanded nodes: A* vs Dijkstra

TODO We can notice that...

A* and Dijkstra

Expanded nodes

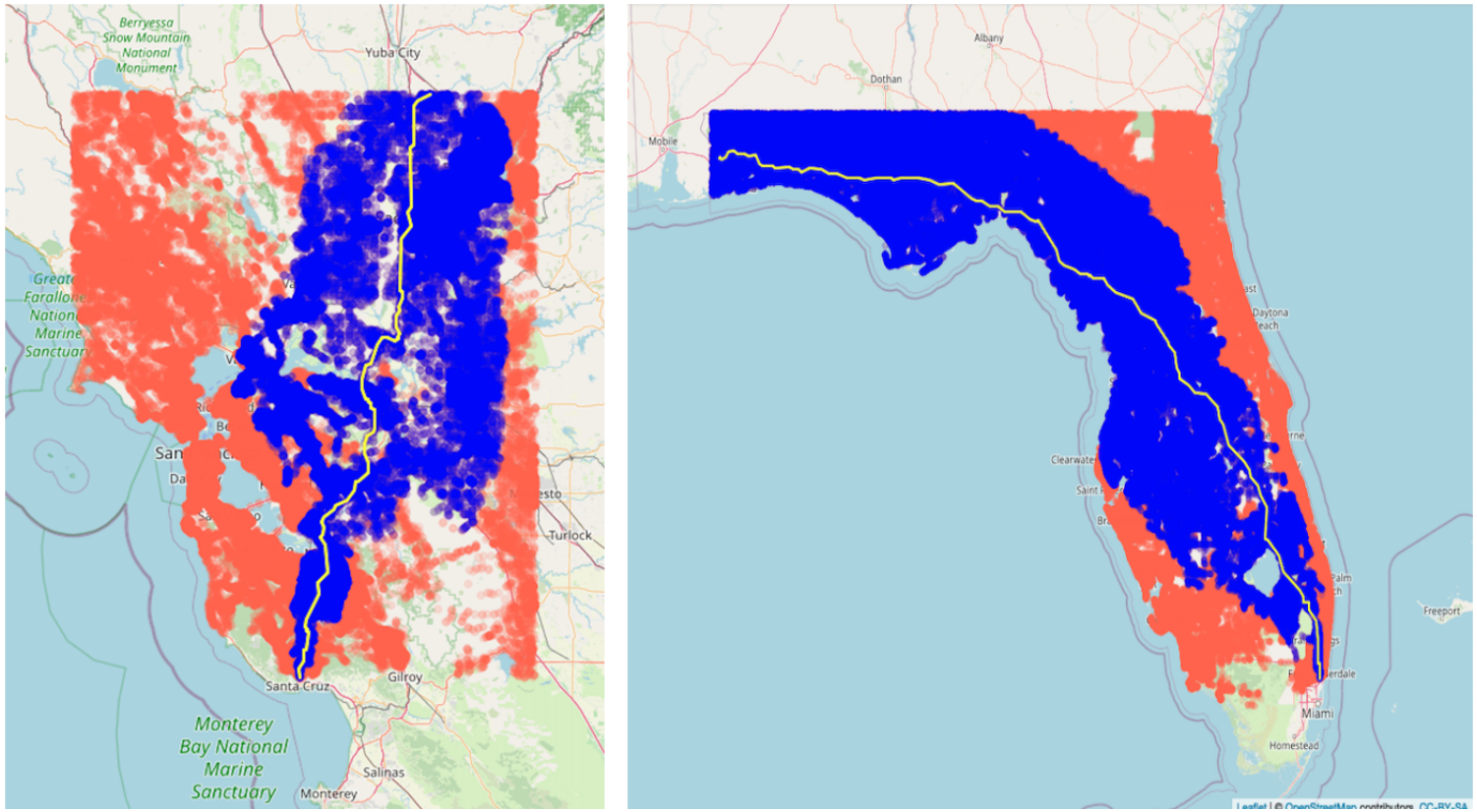


Figure: Test paths on BAY(left) and FLA(right)

A* and Dijkstra

Time and resources comparison

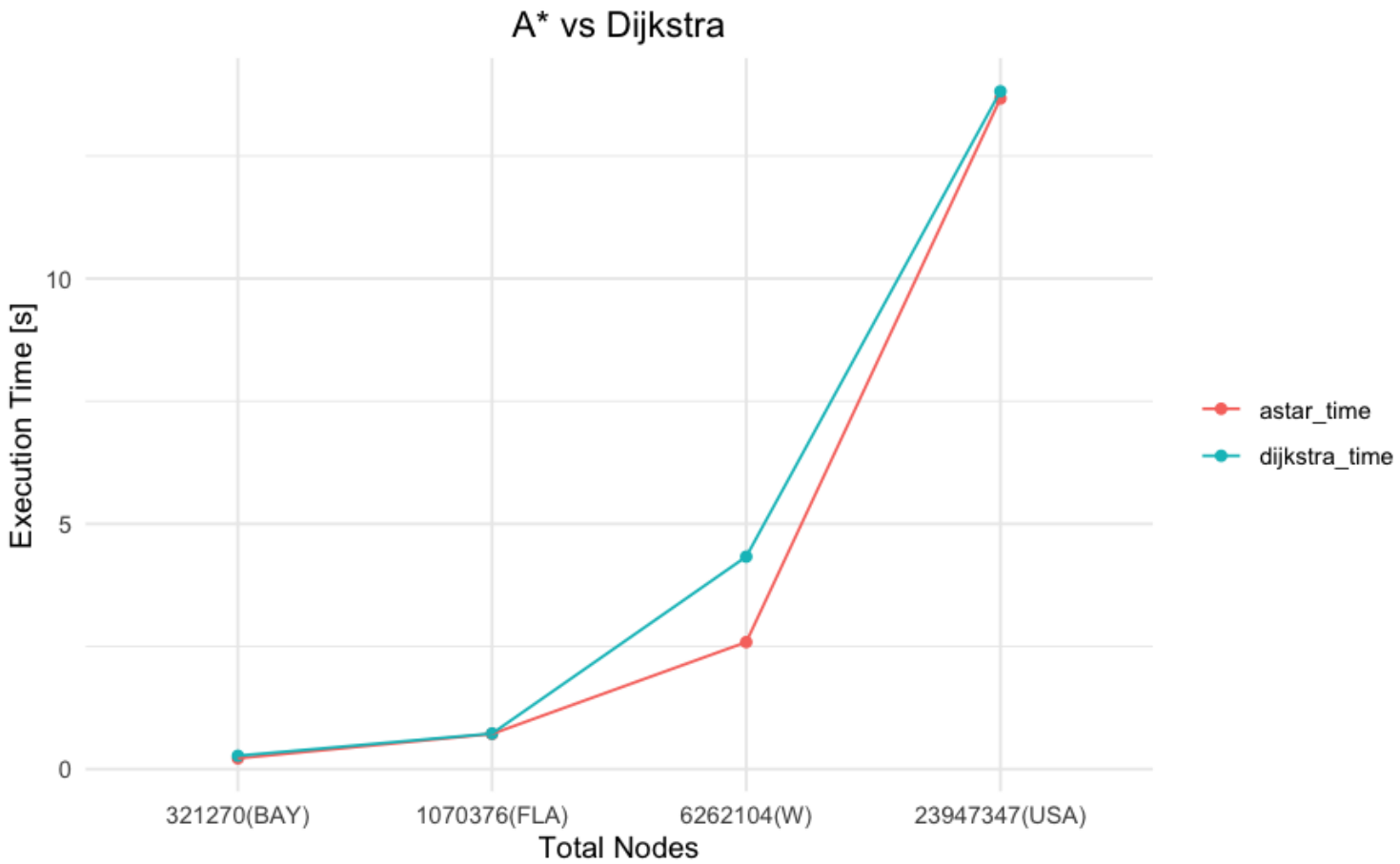


Figure: Execution time: A* vs Dijkstra

A* and Dijkstra

Time and resources comparison

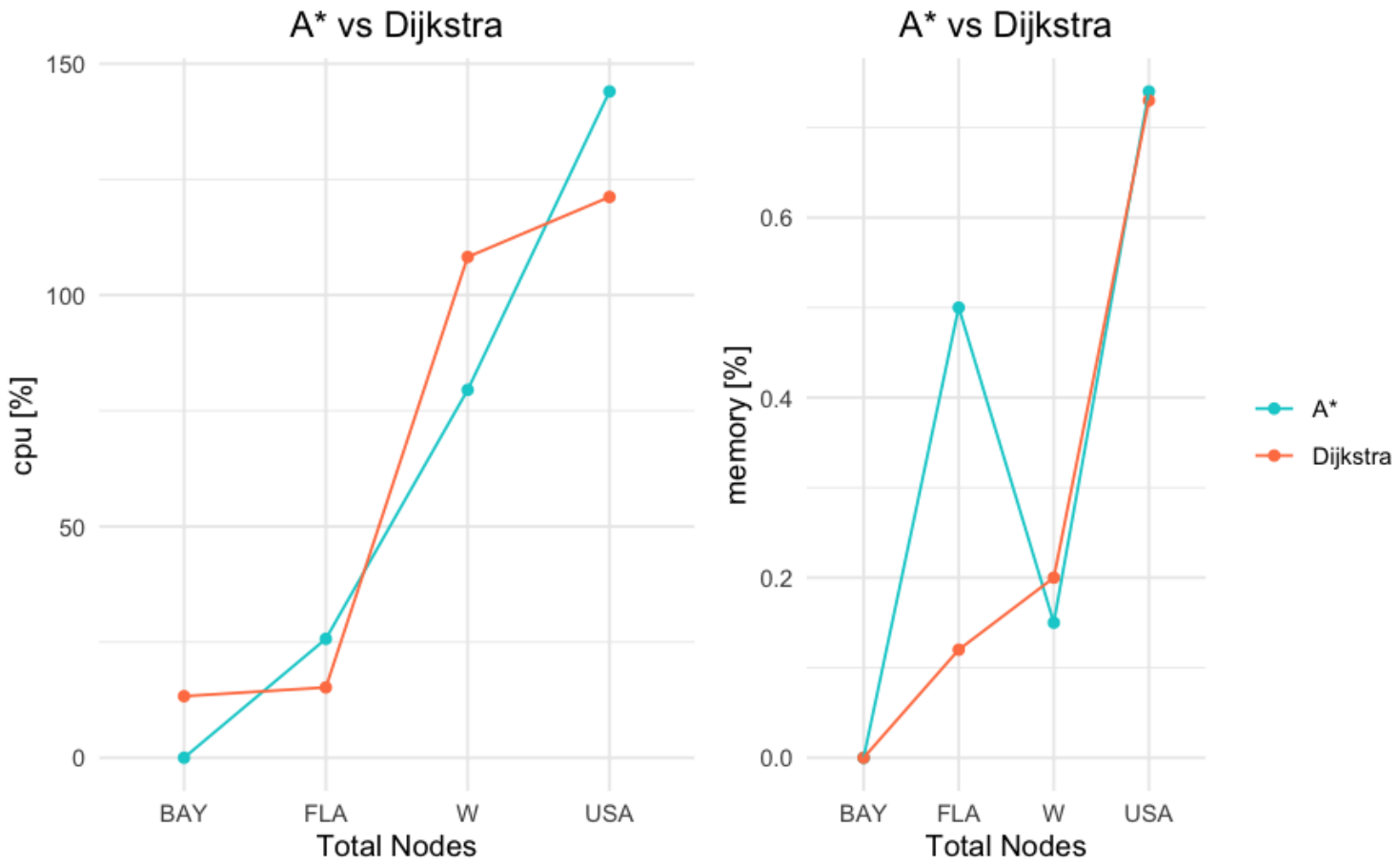


Figure: Exploited resources: A* vs Dijkstra

Outline

Parallel input file reading

Approach 1 (RA1)

In this first approach we have implemeted a solution on which:

- The input file is memory-mapped before being read
- N threads runs freely to read the entire file (array in RAM)

Parallel input file reading

Results on FLA map

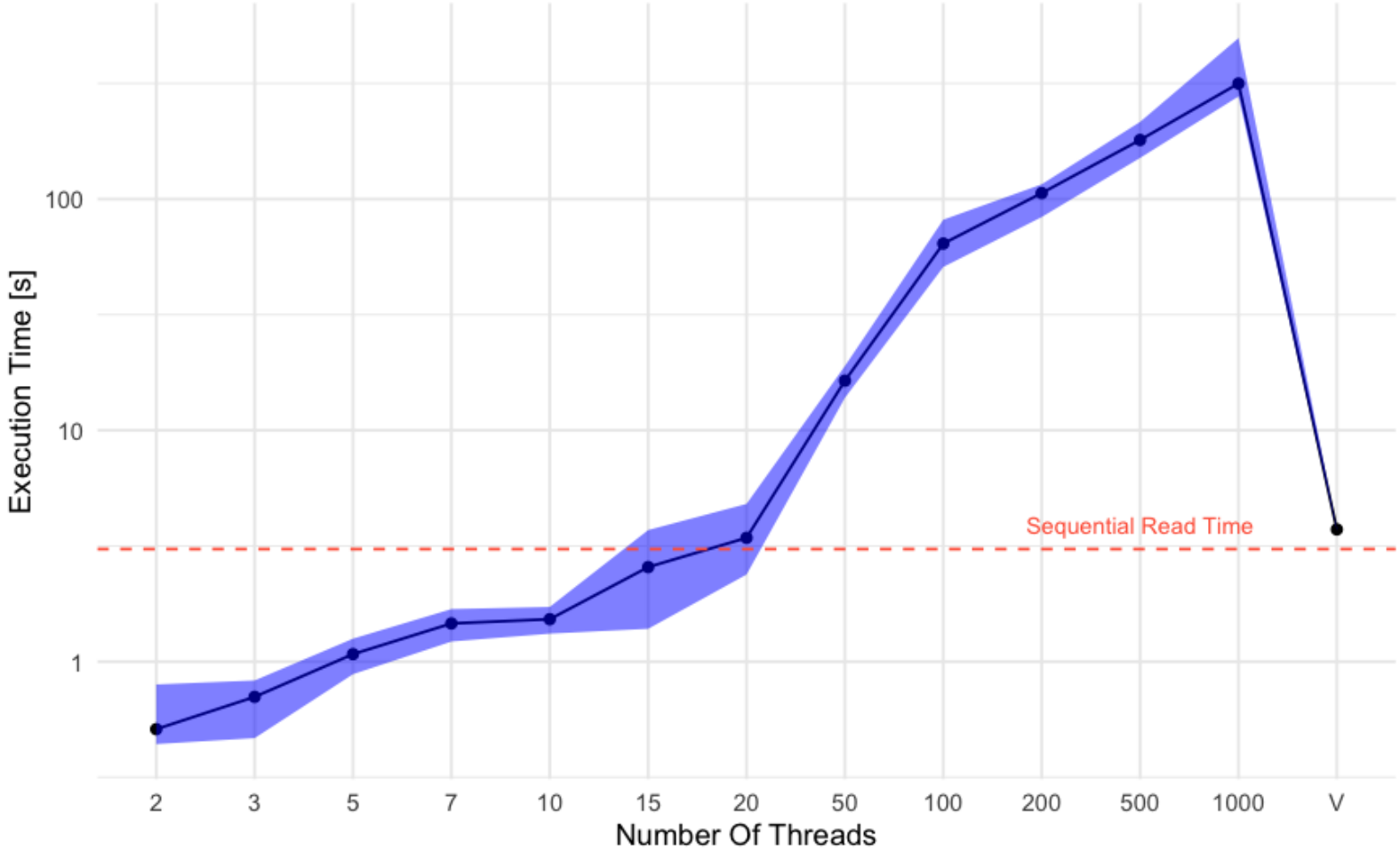


Figure: Performance of RA1 for different number of threads

Parallel input file reading

Approach 2 (RA2)

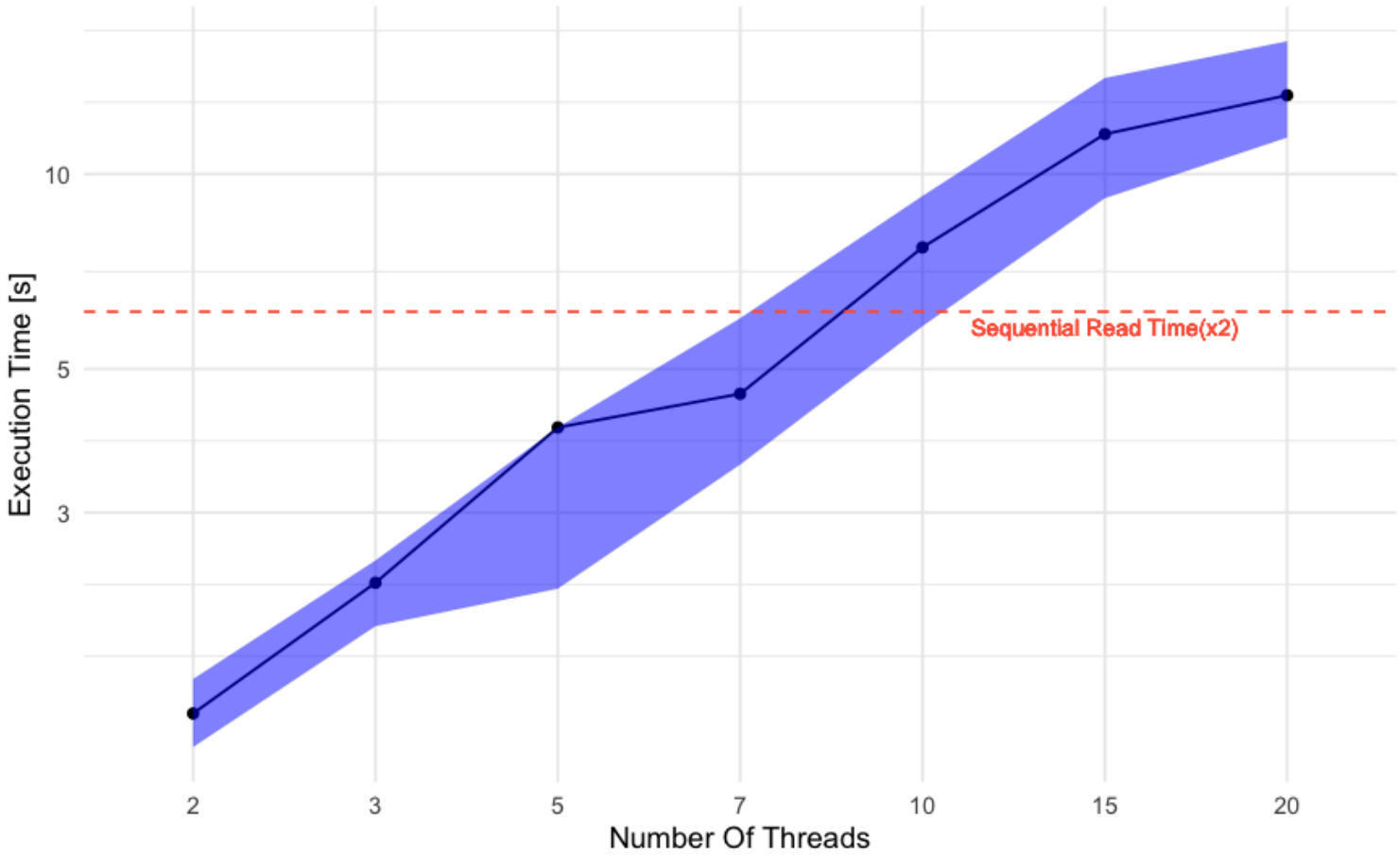
This is not properly a different approach of parallel reading but simply a version of the *RA1* needed when the *PNBA** algorithm will be applied:

- The input file is memory-mapped before being read
- N threads runs freely to read the entire file (array in RAM)
- Two different graphs are loaded: G and the reversed graph R

Since we are loading two graphs data structures at the same time the performances will compared with the sequential reading as if it was run twice (the first time to read G and the second time to read R)

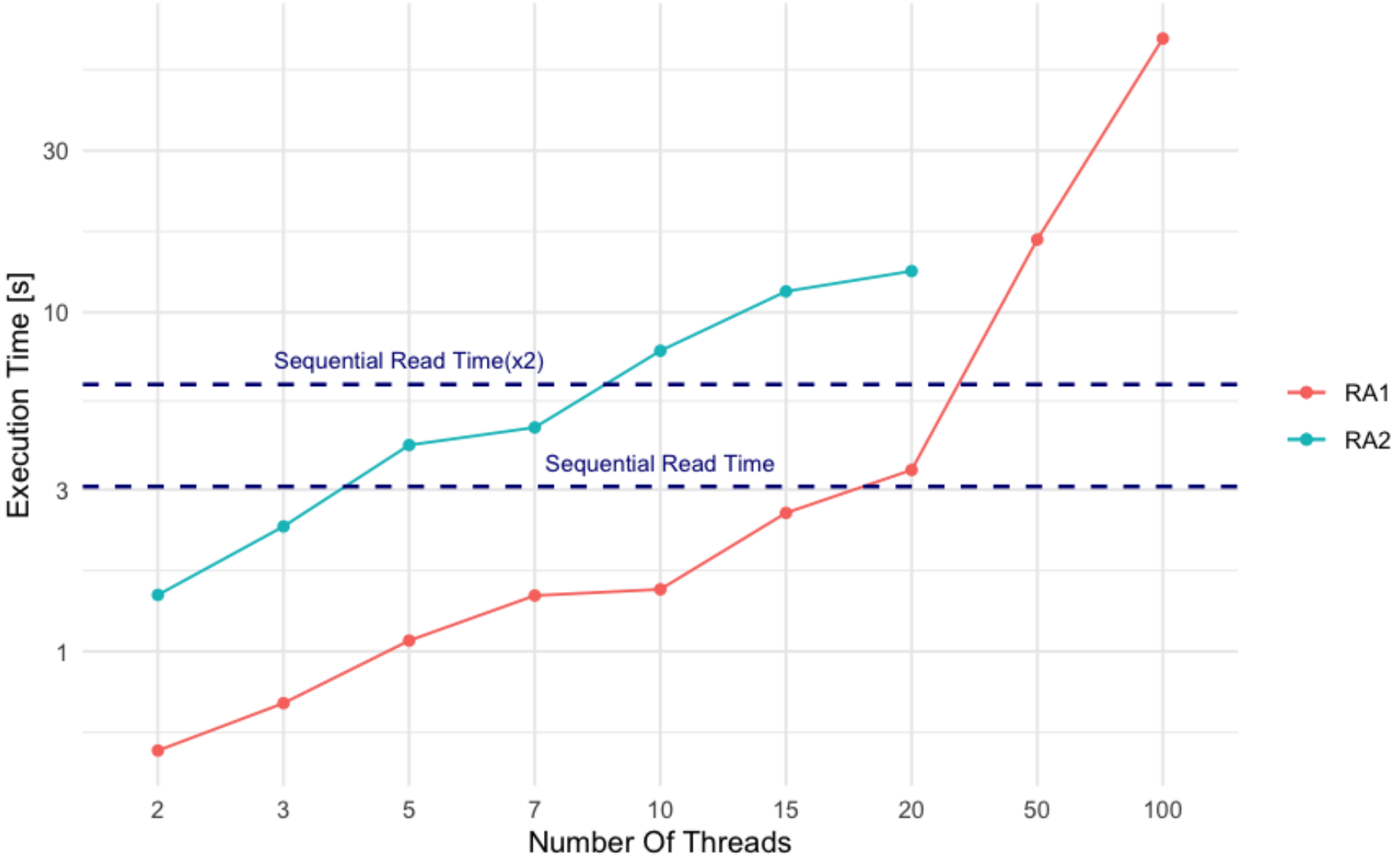
Parallel input file reading

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

	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%

Parallel input file reading

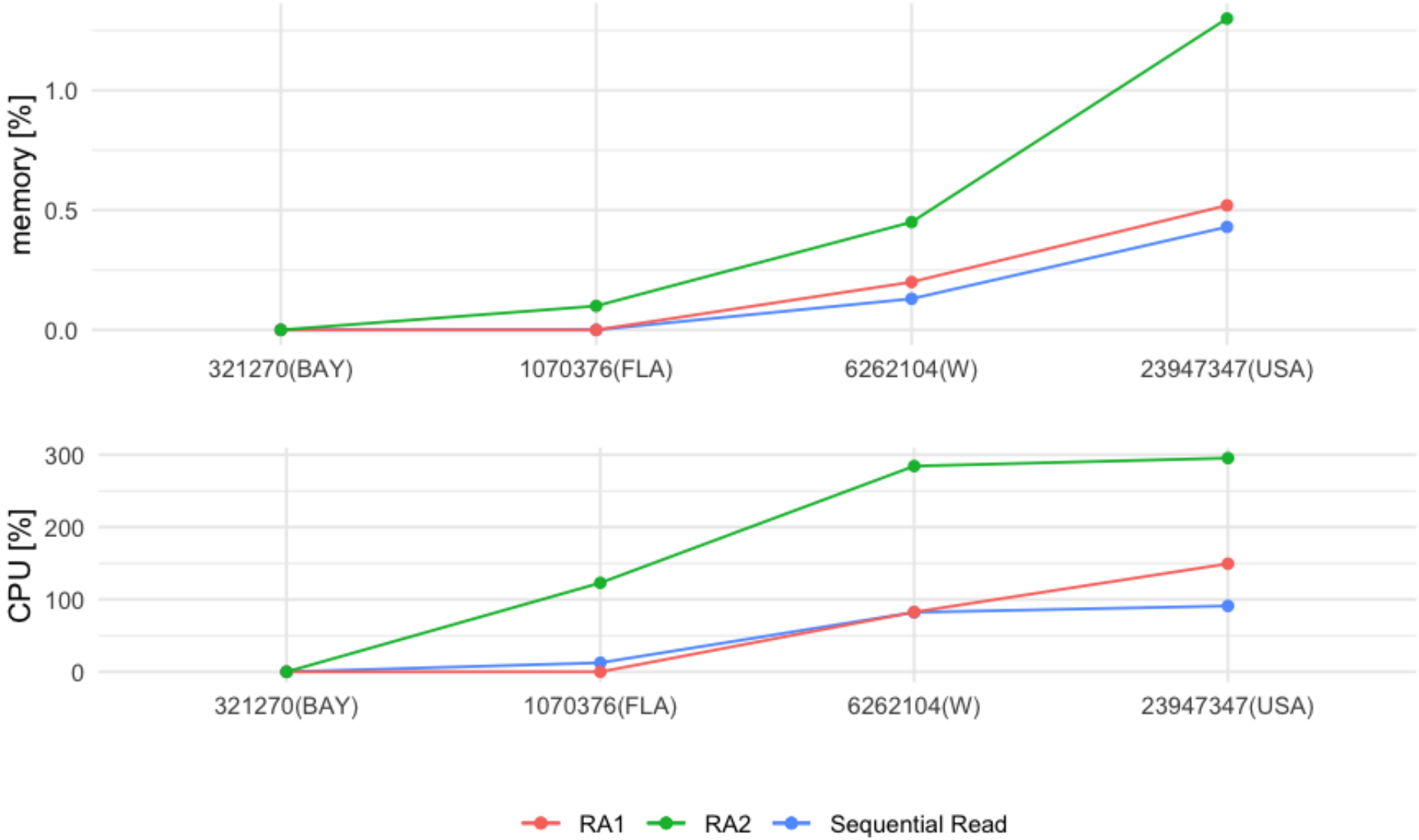
Results on all maps

Table: 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%

Parallel input file reading

Results on all maps



Outline

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 -¿ 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

$hash_1(node\_index, num\_threads) = node\_index \ \% \ num\_threads$
$hash_2(node\_index, num\_threads, V) = i - 1 \ with \ i = min_i : \frac{V}{num\_threads} \cdot i > node\_index \ , i \in \{1, ..., num\_threads\}$

- 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

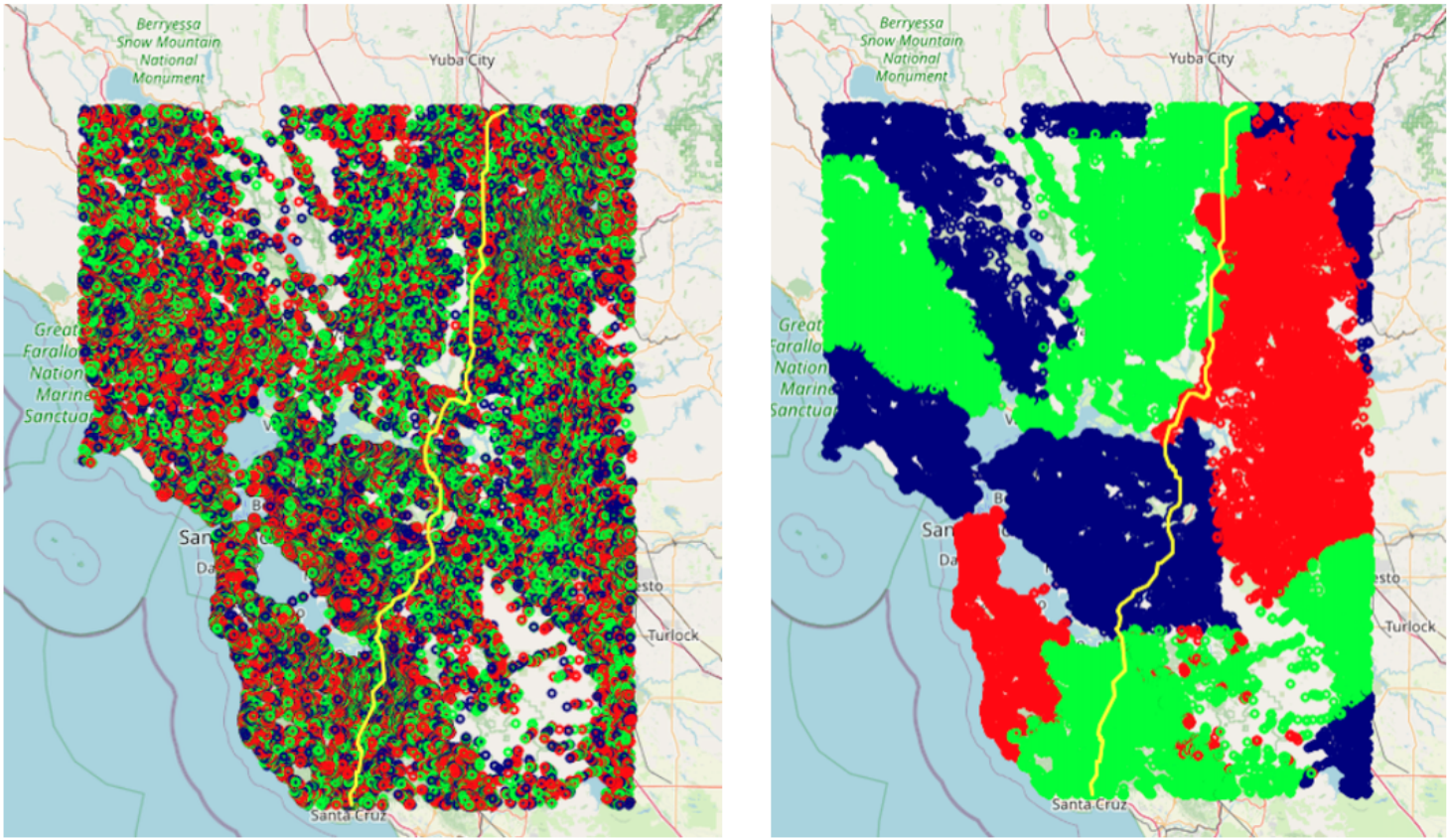


Figure: HDA* work distribution among 3 threads in BAY map

Parallel A*: HDA*

Distributed termination condition

- **Barrier method (B)**: when a thread realizes that its *open set* is empty a barrier is hitten and when all the threads have hitten 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.

Parallel A*: HDA*

Duplicate nodes

We also need to check whether the cost associated to an expanded node when it was previously added to the *closed set* is less or equal to the cost computed at the time n is re-expanded. This is called duplicate checking so, a node n is a duplicate of node m if:

- n is equal to m
- *closedSet* of thread t_i contains the node m
- $g(m) \leq g(n)$

A duplicate node can be discarded.

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. Aobut MPSM:

- Some data structures are shared: TODO
- 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).

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 value of $g(n)$ according to 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 p_G.
- Thread reader t_j reads from the Shared Memory by increasing a local pointer p_j. The reading of messages terminates when $p_j == p_G$.

Parallel A*: HDA*

MPSM pseudocode

TODO

Parallel A*: HDA*

Shared Address Space (SAS)

The shared data structures are:

- A global array of *open sets* that here we call A where $A[i]$ contains a pointer to the *open set* of thread t_i and the size of A is N (the number of threads).
- 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:

- One lock $L1$ for each *open set* so for each $A[i]$ $\forall i \in \{1..N\}$.
- One lock $L2$ for each node of the graph in order to correctly update *parentVertex* and *costToCome* data structures.

Parallel A*: HDA* I

SAS pseudocode

1: **function** $astar(G, source, dest, h)$
2: $g[i] \leftarrow DOUBLE\_MAX \; \forall i \in V$
3: $h[i] \leftarrow heuristic(i, dest) \; \forall i \in V$
4: $parentVertex[i] \leftarrow -1 \; \forall i \in V$
5: $t\_owner \leftarrow hash(source, num\_threads)$
6: $f[t\_owner][source] \leftarrow h[s]$
7: $g[source] \leftarrow 0$
8: $openSet[t\_owner] := \{(source, f[t\_owner][source])\}$
9: Initialize N $mutex\_threads$
10: Initialize V $mutex\_nodes$
11: Launch N threads
12: Join N threads
13: $reconstructPath()$
14: **end function**

Parallel A*: HDA* II

SAS pseudocode

```
15: function hda_sas
16:     while 1 do
17:         while !openSet[index].EMPTY() do
18:             LOCK(mutex_threads[index])
19:             a ← openSet.POP()
20:             UNLOCK(mutex_threads[index])
21:         end while
22:         if a is duplicate then
23:             continue
24:         end if
25:         for neighbor b of a do
26:             wt ← weight(a, b)
27:             tentativeScore ← g[a] + wt
28:             if tentativeScore is less than g[b] then
```

Parallel A*: HDA* III

SAS pseudocode

```
29:             owner_a ← hash(a, N)
30:             owner_b ← hash(b, N)
31:         end if
32:         if b is duplicate then
33:             continue
34:         end if
35:         LOCK(mutex_nodes[a])
36:         tentativeScore ← g[a] + wt
37:         UNLOCK(mutex_nodes[a])
38:         LOCK(mutex_nodes[b])
39:         if tentativeScore is less than g[b] then
40:             parentVertex[b] ← a
41:             costToCome[b] ← wt
42:             g[b] ← tentativeScore
```

Parallel A*: HDA* IV

SAS pseudocode

43: $f[b] \leftarrow g[b] + h[b]$
44: LOCK($mutex\_threads[owner\_b]$)
45: $f[owner\_b][b] \leftarrow f[b]$
46: $openSet.PUSH((b, f[owner\_b][b]))$
47: UNLOCK($mutex\_threads[owner\_b]$)
48: **end if**
49: UNLOCK($mutex\_nodes[b]$)
50: **end for**
51: **end while**
52: **if** $openSet[index].EMPTY()$ and $parentVertex[dest] != -1$ **then**
53: Terminate according to B or SF
54: **end if**
55: **end function**

Parallel A*: HDA*

Results

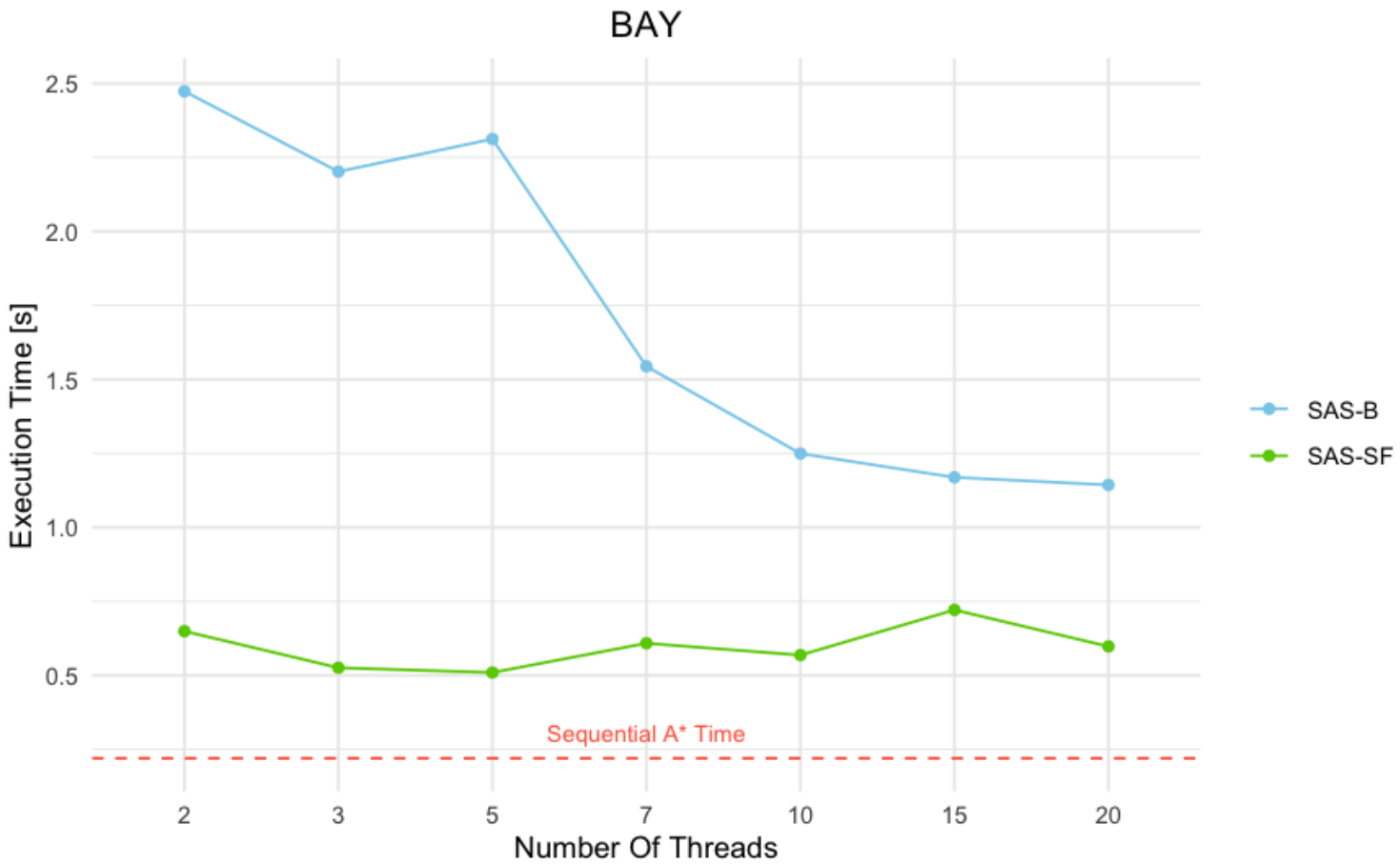


Figure: HDA* on BAY map

Parallel A*: HDA*

Results

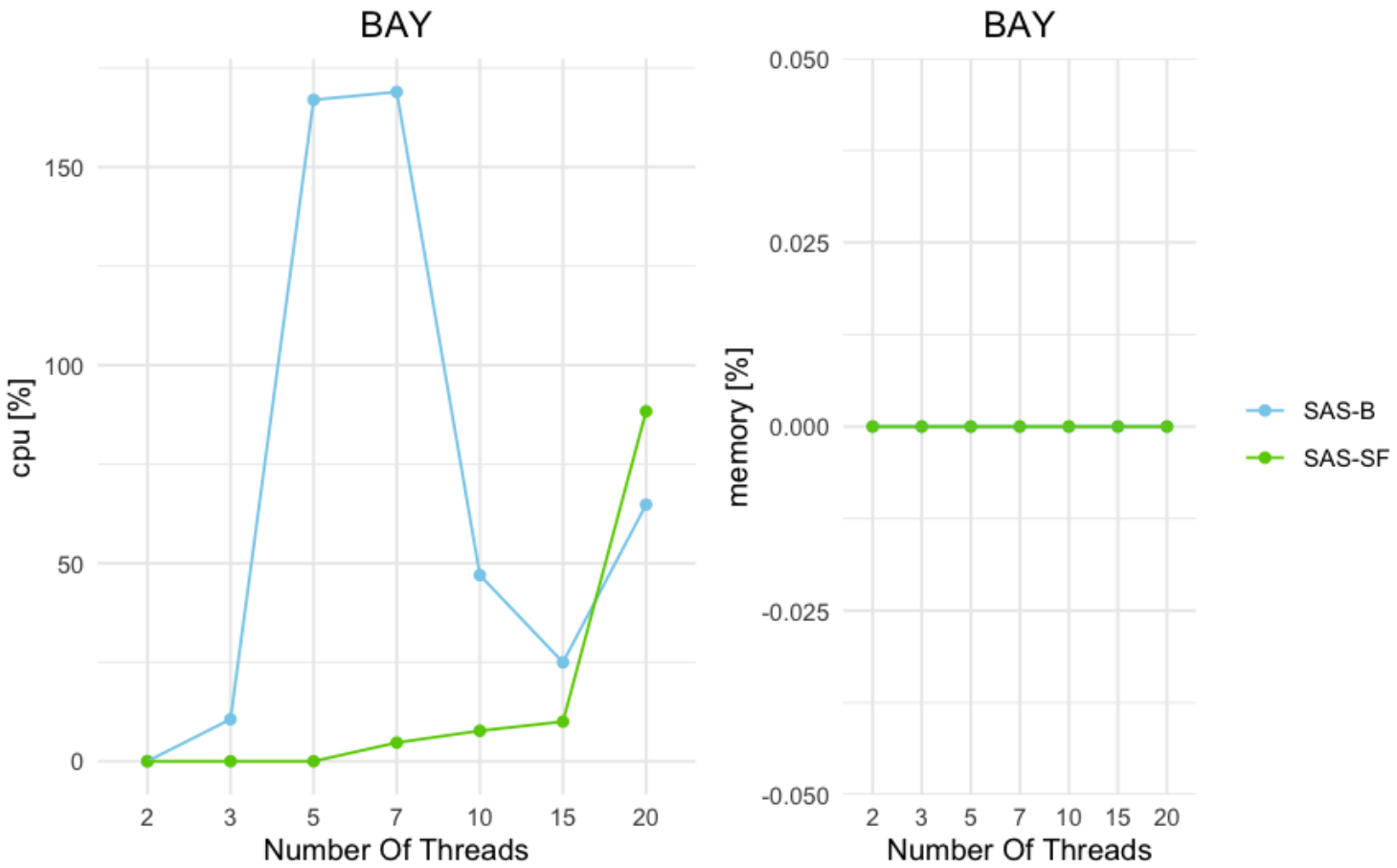


Figure: HDA* on BAY map

Parallel A*: HDA*

Results

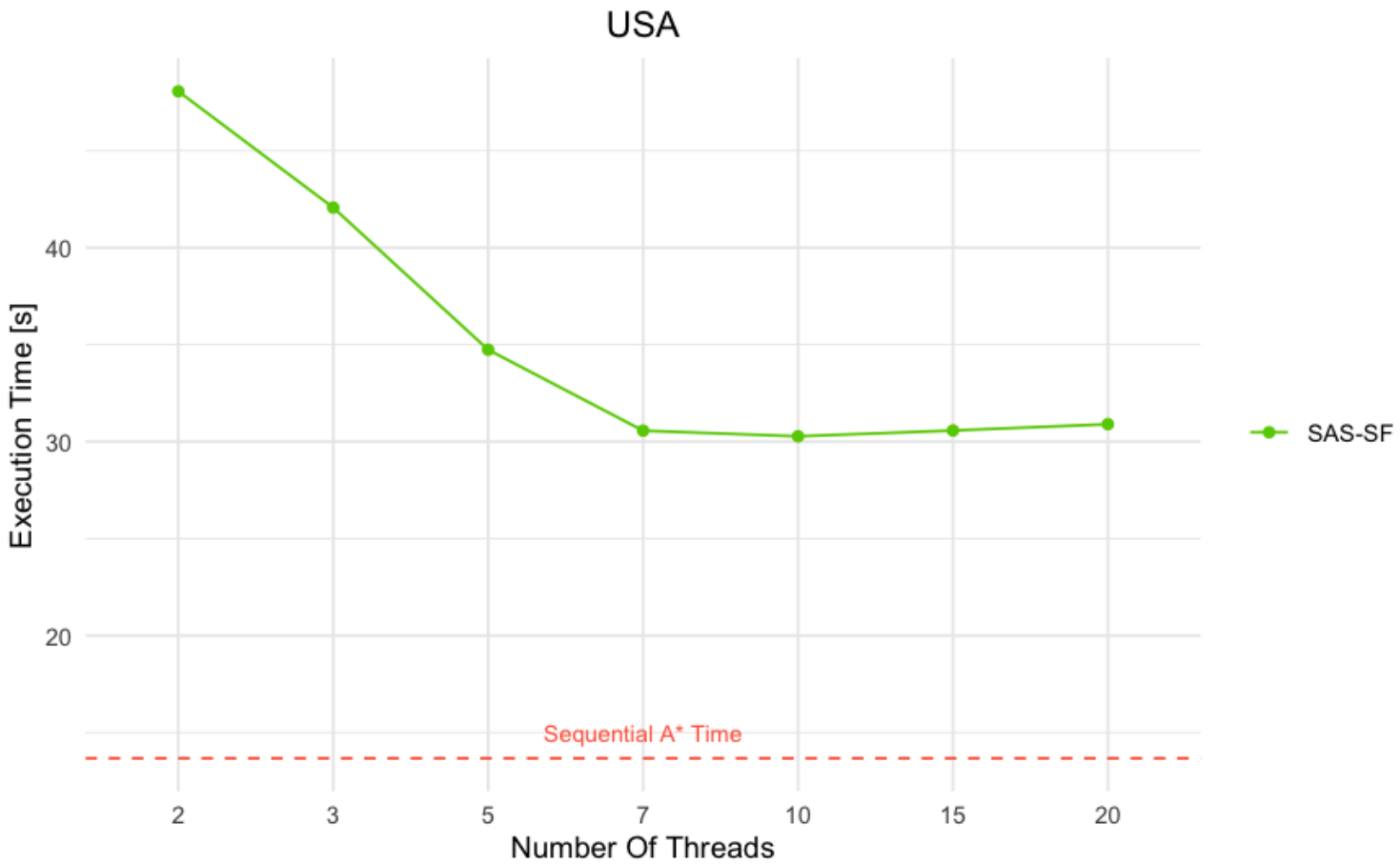


Figure: HDA* on USA map

Parallel A*: HDA*

Results

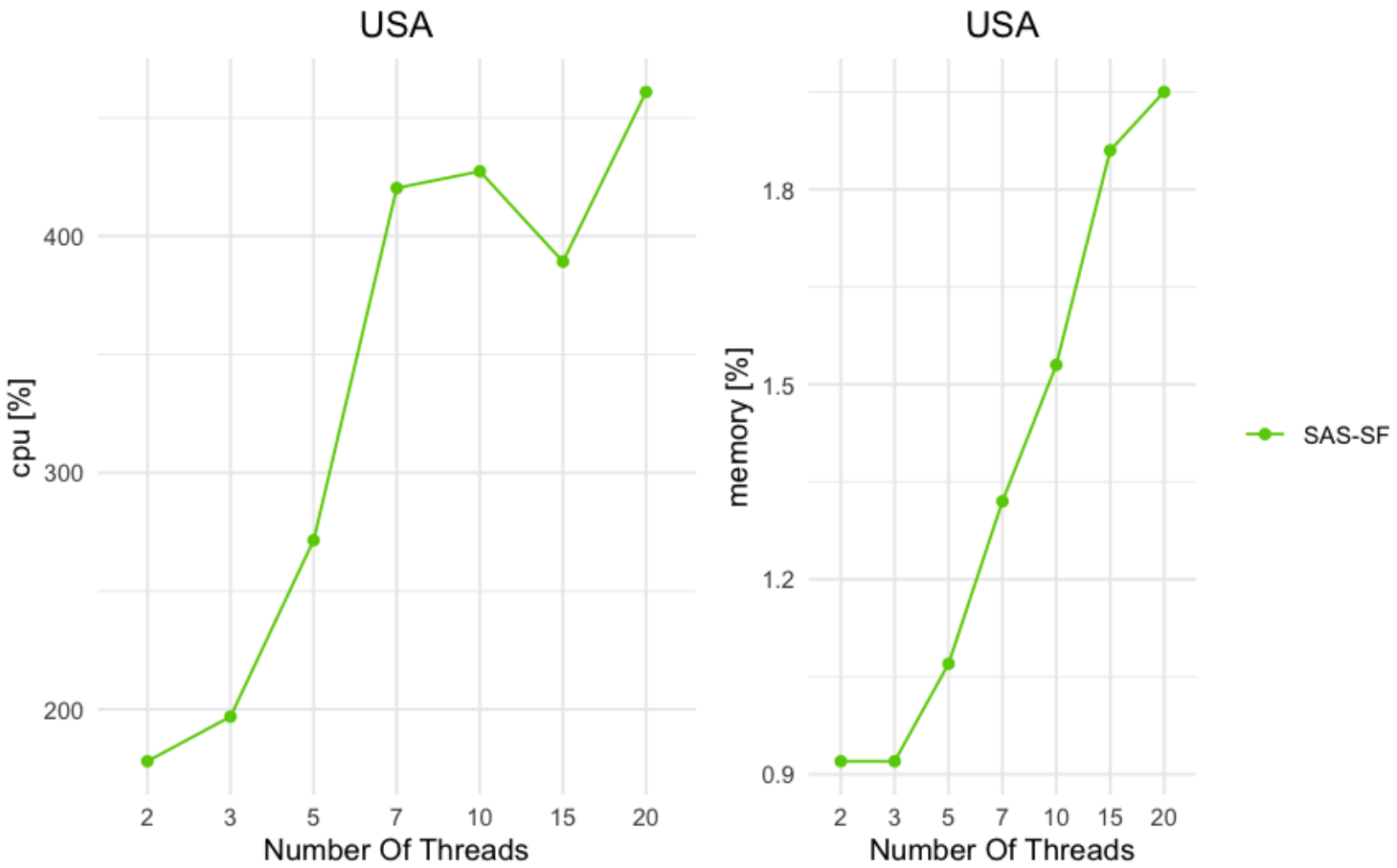


Figure: HDA* on USA map

Parallel A*: HDA*

Comments and overall results

- The SAS-MP variant of HDA* is not scalable.
- The SAS-B is more scalable than SAS-MP but the termination condition is not well-performing on large graphs.
- The 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.
- About the resource consumption both SAS-B and SAS-SF are more expensive in terms on CPU and memory used w.r.t the sequential algorithm. The resource used increase as the number of threads increase.

Parallel A*: HDA*

Comments and overall results

Here we only write the performances of SAS-SF that is the only one able to achieve reasonable results on all the benchmark graphs:

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

	Threads	**SAS-SF**	**Sequential A***	**Slow-Down**
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%

Parallel A*: HDA*

Comments and overall results

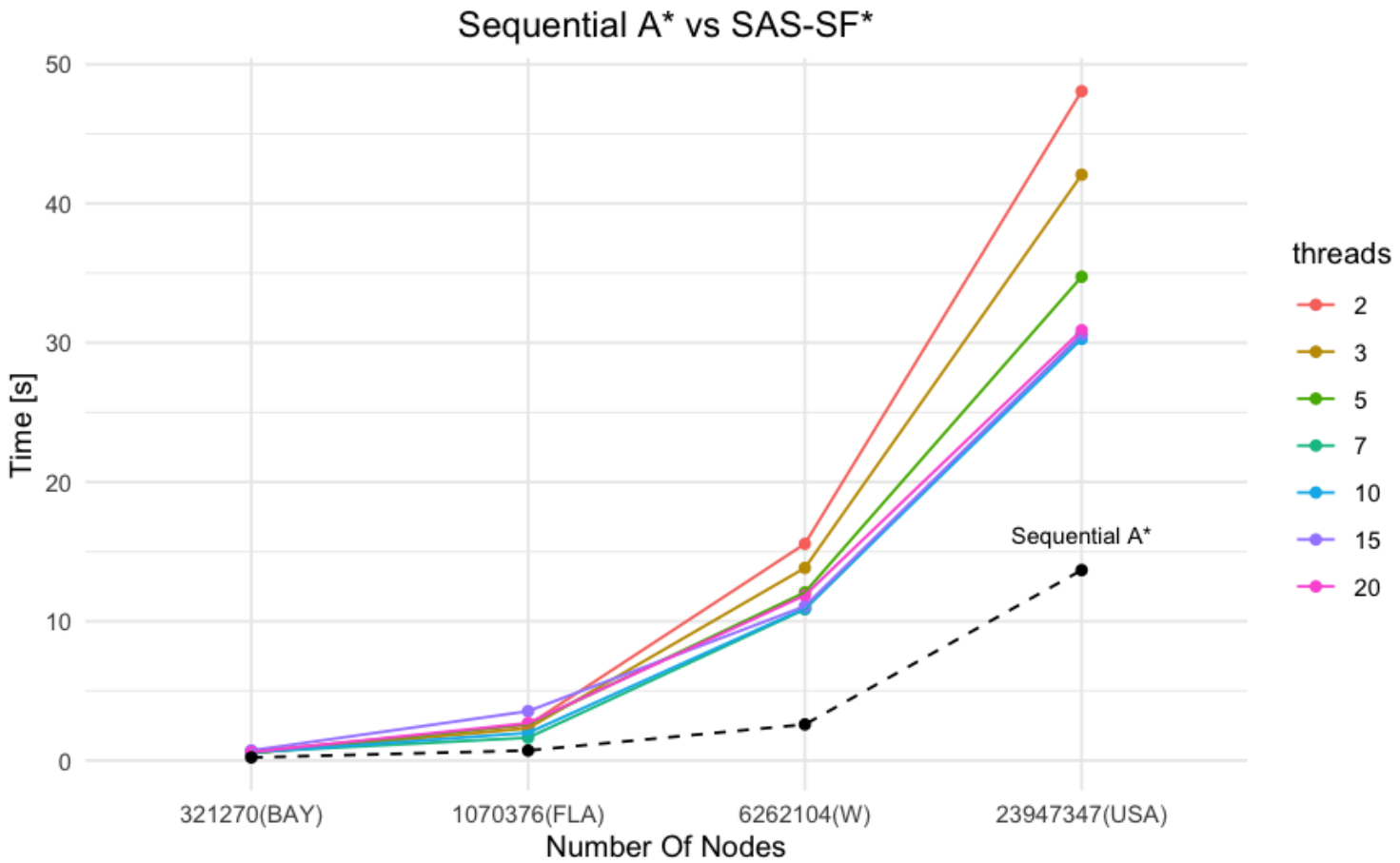


Figure: HDA* overall performances

Parallel A*: HDA*

Comments and overall results

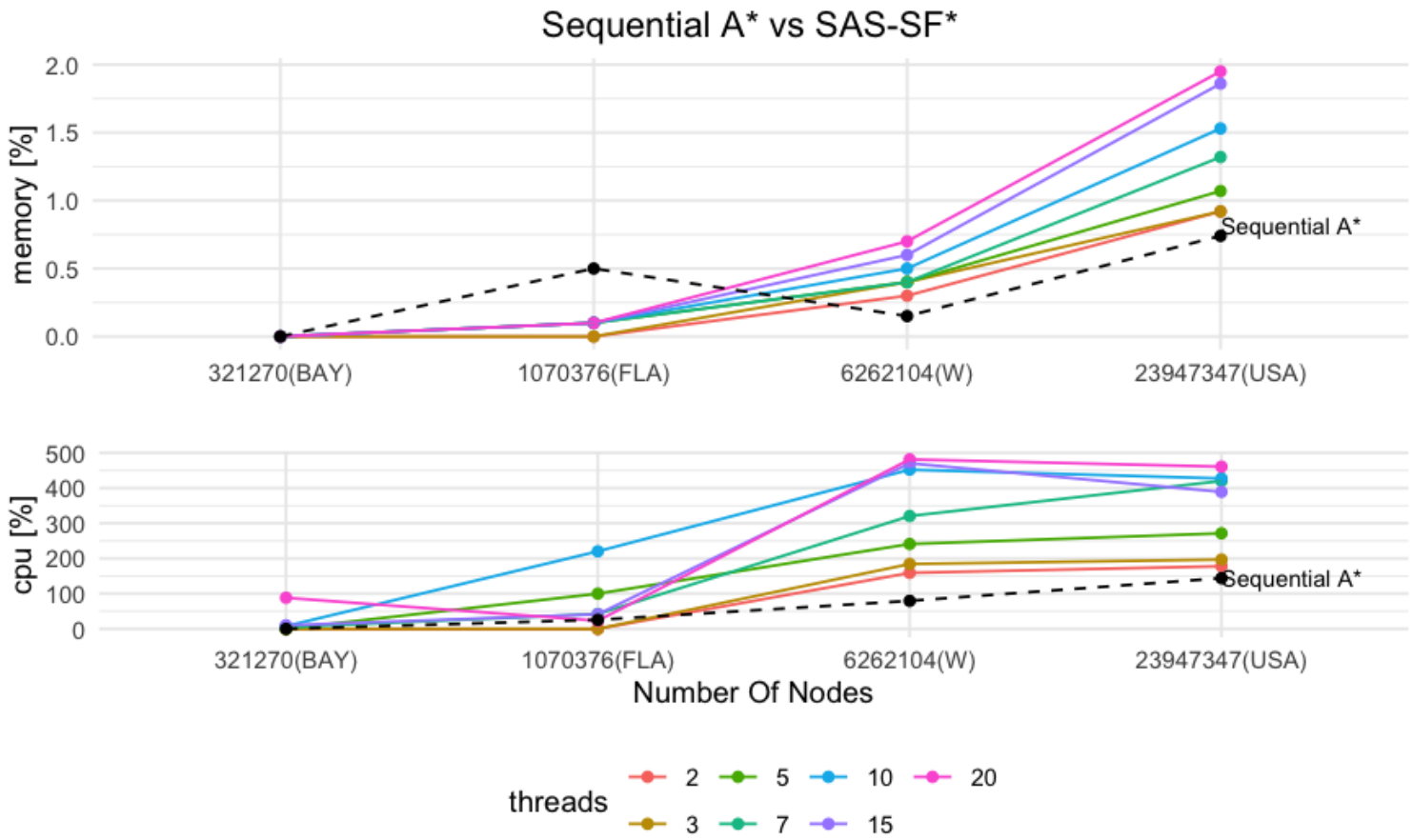


Figure: HDA* ooverall performances

Parallel A*: HDA*

Comments and overall results

- The hash function has an important impact on the performances of A* algorithm. The one that we have used is probably not optimal. Using a more efficient one could lead to obtain better performances but we suppose that the sequential algorithm would still be the "best case" instead of being the "worst case" in terms of execution time.
- 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: the performance scalability and improvements that should have been obtained with large graphs have not been verified.

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 ∞ 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)

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

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.

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.

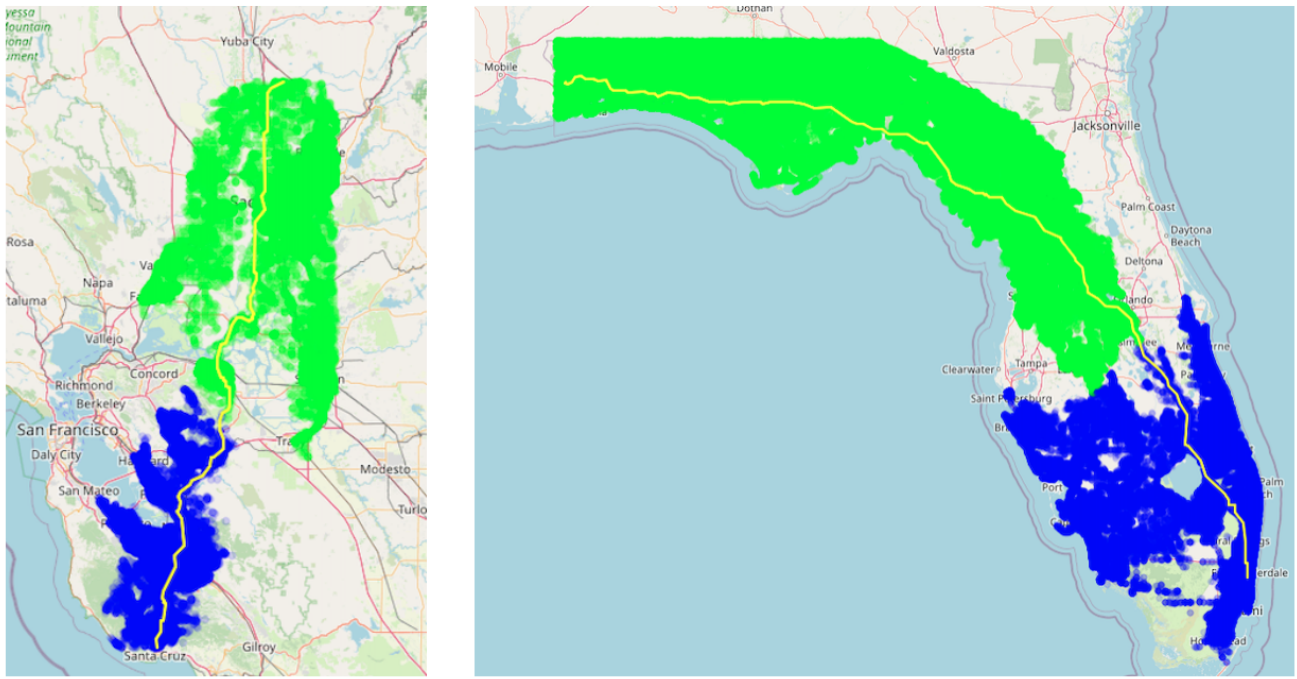


Figure: PNBA* work on BAY(left) and FLA(right)

Parallel A* I

PNBA*

1: **function** $pnba_G()$
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** $(y \in M) \wedge (g_G(x) > g_G(x) + d_G(x, y)$ **then**
8: $g_G(y) \leftarrow g_G(x) + d_G(x, y)$
9: $f_G(y) \leftarrow g_G(x) + h_G(x, y)$
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* ← *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 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

Parallel A*: PNBA*

Results

Table: 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%

Parallel A*: PNBA*

Results

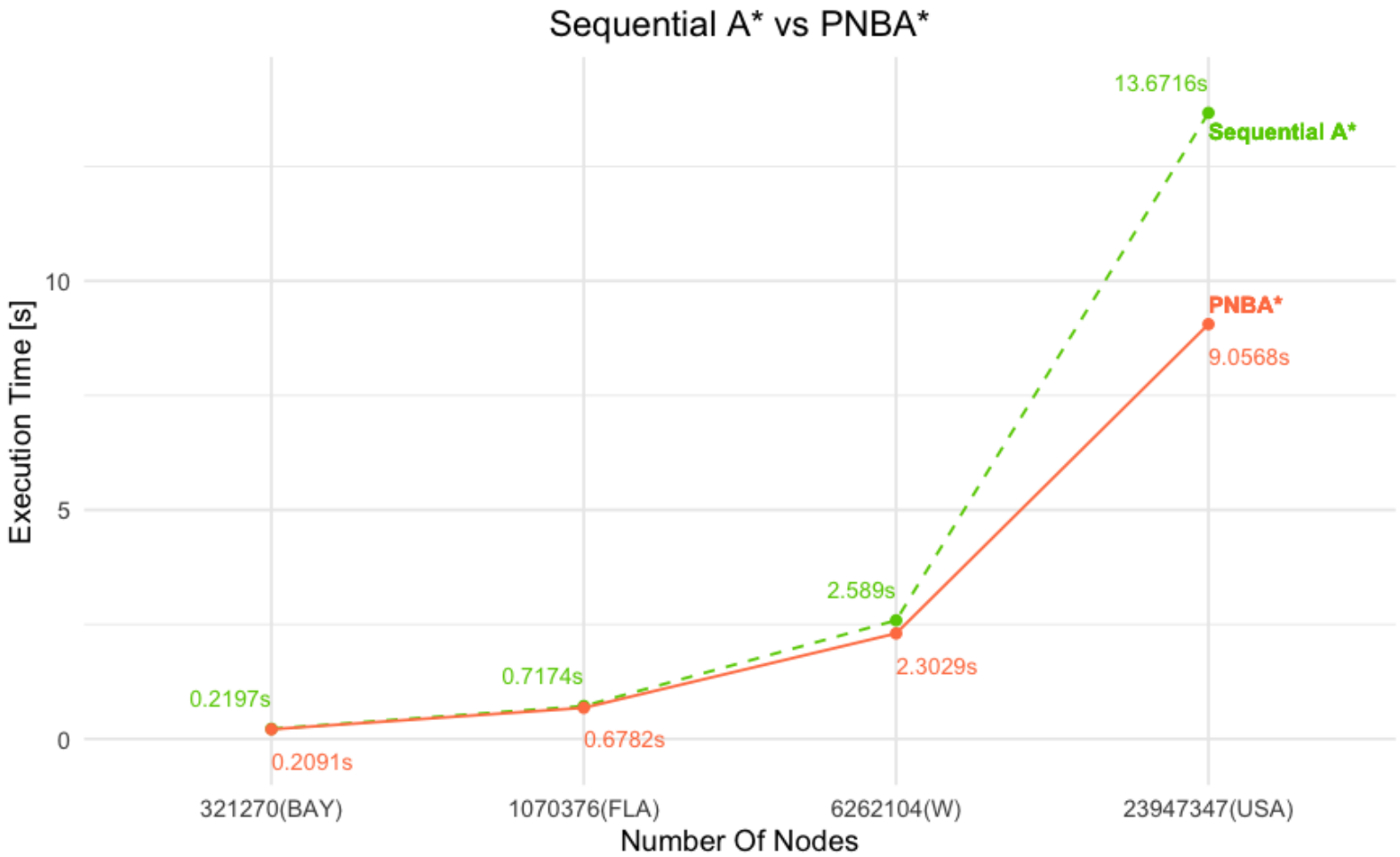


Figure: PNBA* overall performances

Parallel A*: PNBA*

Results

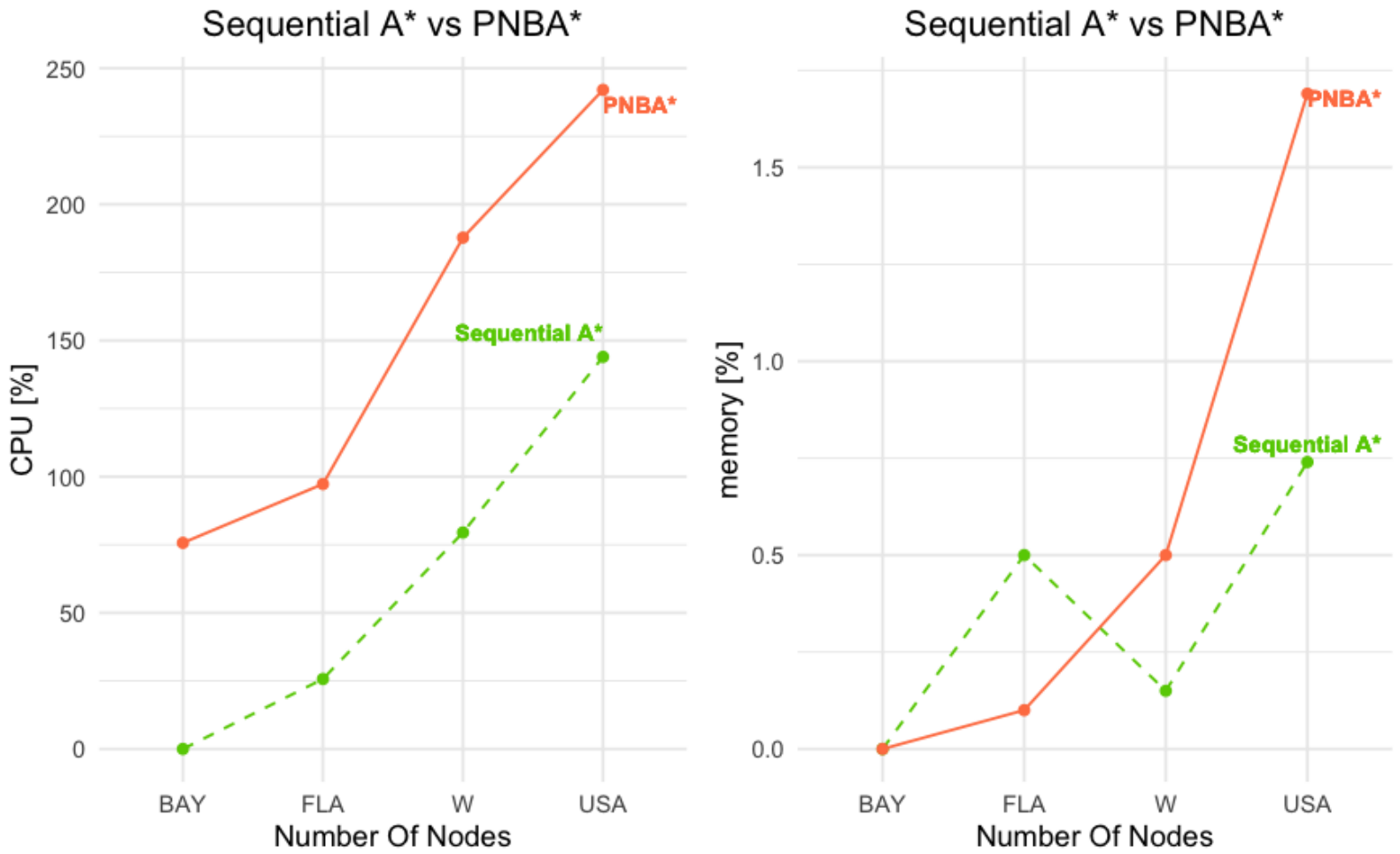


Figure: PNBA* overall performances - time

Outline

Conclusions

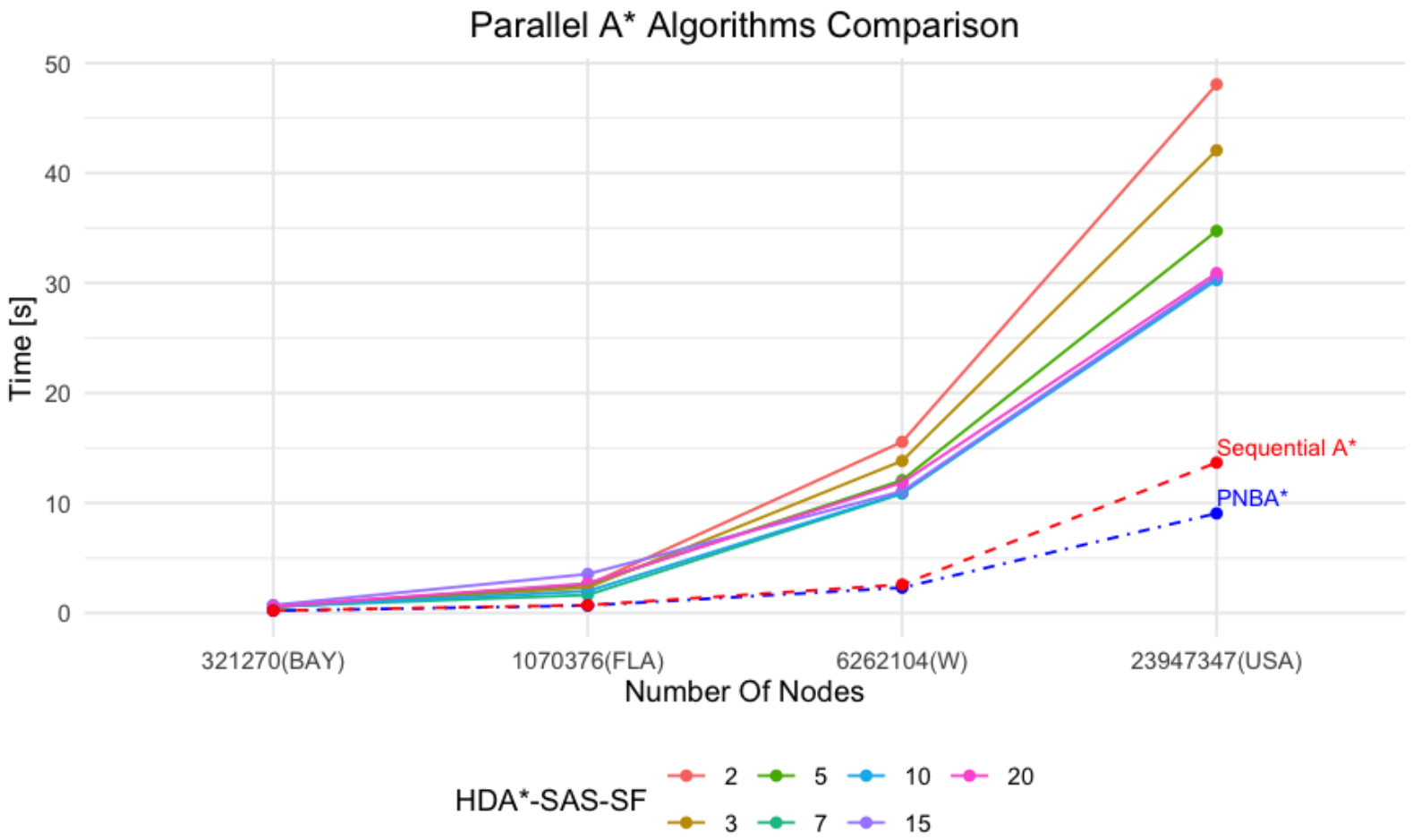


Figure: Parallel A* overall performances - resources

Conclusions

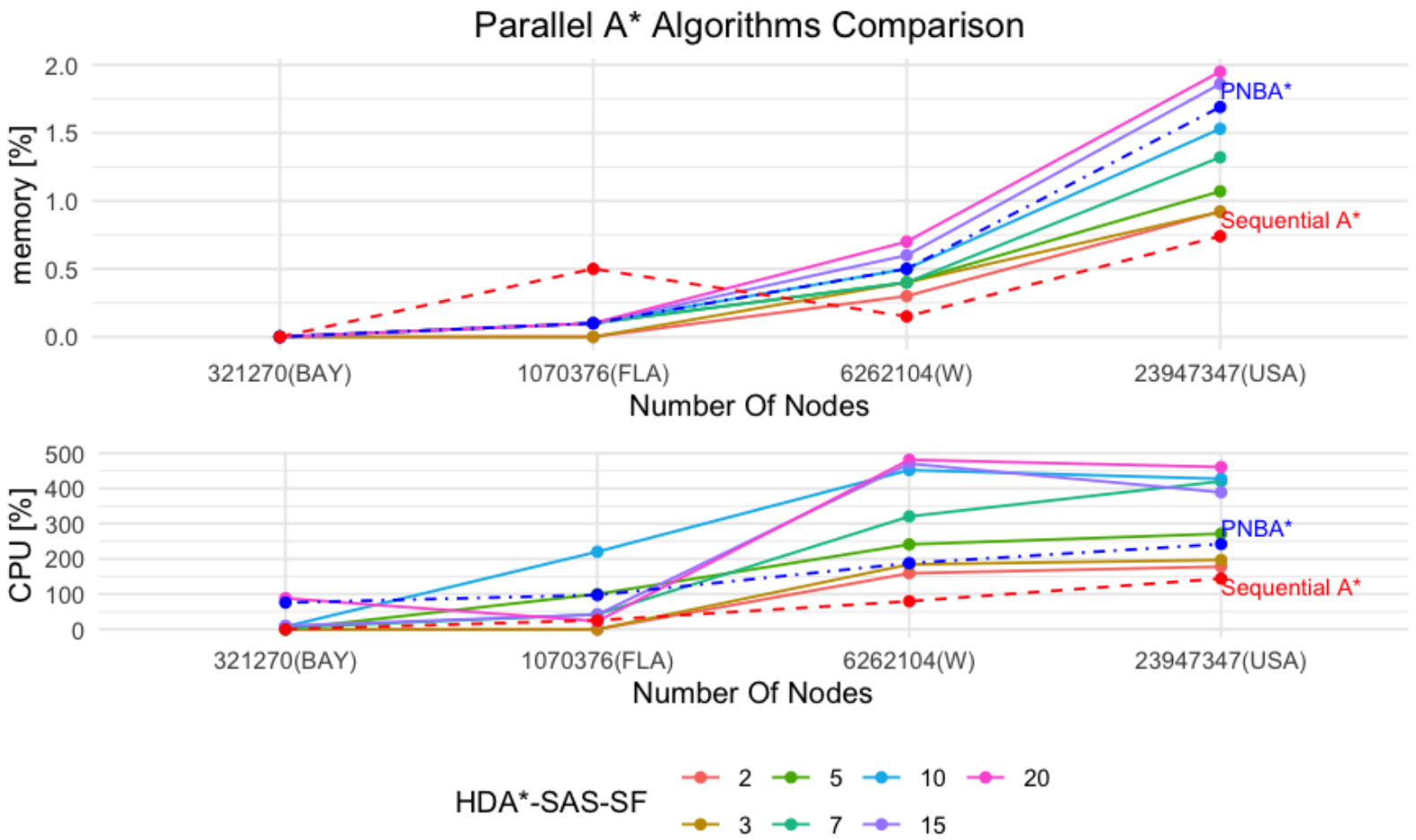


Figure: Parallel A* overall performances

Outline

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

Outline

Future Works

TODO

Outline

References

TODO