Parallel A* Project

System And Device Programming

Lorenzo Ippolito, Fabio Mirto, Mattia Rosso

Politecnico di Torino

September 6, 2022

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 and it achieves better performaces with respect to it. It is a Greedy-best-first-search algorithm that uses an heuristic function to guide itself.

1. Introduction: about the A* algorithm

What it does is combining:

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

1. Introduction: about the A* algorithm

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 (main loop) iteration the node n that has the minimum $f(n)$ is examinated.

1. Introduction: about the A* algorithm

Heuristic design - properties

The heurstic function represents the acutal 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. The properties of an heuristic function are:

- Admissible Heuristic: if $h(n) < cost(n, dest) \; \forall n \in V$
- Consistent Heuristic: if $h(x) \leq cost(x, y) + h(y)$ for every edge (x, y)

(TODO: what happens if no admissible/consistent)

1. 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: 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.

2. A* project application

Optimal path searching in geographical areas

We work with a weighted oriented graph G that is made of nodes $n \in V$ that represents road-realated points of interest and edges $(x, y) \in E$ represent the unidirectional connections among 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

Heuristic function: the great-circle distance

We will employee 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

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

3. Graph file input

File input format

The files we have used use this 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 $(x[int], y[int], weight[double])$

3. Graph file input

Random test graph

We have tested the designed algorithms also on a random generated graph that is built starting from:

- Which path we want to find: given the couple (source, destination) it is generated a graph of $max(source, destination) + 1$ nodes.
- How many paths at most have to be generated from source to destination
- The maximum length of these paths (that will be randomly chosen for each path)

In this way we have ad-hoc files to stress the algorithm having the guarantee that more than one path exists from source to destination. To be consistent with benchmark files also these random graphs represents geographic points with longitude and latitude.

3. Graph file input

DIMACS benchmark

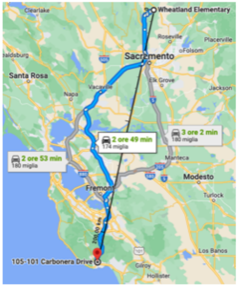
The benchmark files we have used come from the DIMACS benchmark. Here each geographic map is described by:

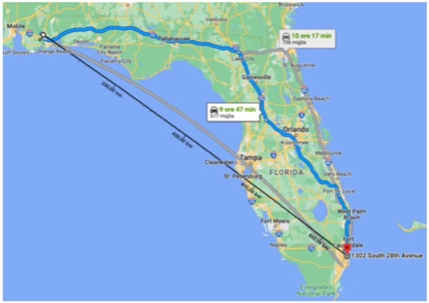
- *.co* file: a file containing the coordinates of the nodes following the FIPS system notation
- *.gr* file: a file containing the edges and the relative weight(distance) expressed in meters

We have properly converted these files to obtain binary files with our chosen input format

3. Graph file input

Test paths





Seattle
25 ore
1.588 miglia
25 ore
1.655 miglia
1.000,00 km
Los Angeles
San Diego

41 ore
2.789 miglia
Stati Uniti
4251 Alameda Street
4th Street
3.930,48 km

4. Sequential A* Algorithm

The first step consists of a pre-computation of:

- The heurstic $h(n)$ 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* table
- *parentVertex* table

4. Sequential A* Algorithm I

```
function $astar(G, source, dest)$
    $g[i] \leftarrow DOUBLE\_MAX\ \forall i \in V$
    $f[i] \leftarrow DOUBLE\_MAX\ \forall i \in V$
    $h[i] \leftarrow h(i, d)\ \forall i \in V$
    $parentVertex[i] \leftarrow -1\ \forall i \in V$
    $f[source] \leftarrow h[ssource]$
    $g[source] \leftarrow 0$
    $openSet := \{(source, f[source])\}$
    while $!openSet.EMPTY()$ do
        $a \leftarrow openSet.POP()$
        if $a == dest$ then
            reconstructPath()
        end if
        for all neighbors $b$ of $a$ do
            $wt \leftarrow weight(a, b)$
```

4. Sequential A* Algorithm II

```
            $tentativeScore \leftarrow g[a] + wt$
            **if** $tentativeScore < g[b]$ **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 if**
        **end for**
    **end while**
**end function**
```

4. Sequential A* Algorithm

Results

	File Size	Reading	A*	Total	Reading Impact
RND	2876B	0.0011s	0.0862s	1.1714s	1.2872%
BAY	20.51MB	0.9365s	0.2349s	1.1714s	79.9477%
FLA	69.09MB	3.0728s	0.5893s	3.6621s	83.9075%

TODO why we need parallel reading...

5. A* and Dijkstra: a comparison

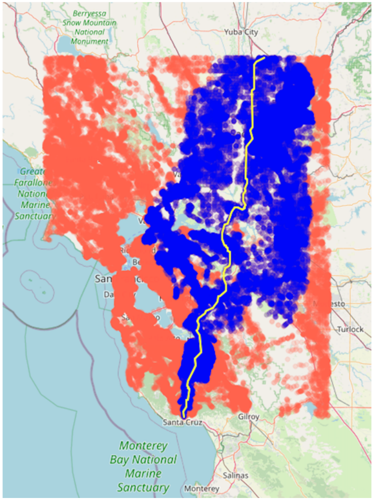
Results

Expanded nodes	**Dijkstra**	**Sequential A***
RND	15 of 101	13 of 101
BAY	318725 of 321270	156950 of 321270
FLA	996956 of 1070376	591926 of 1070376

TODO We can notice that...

5. A* and Dijkstra: a comparison

Results


Berryessa Snow Mountain National Monument
Yuba City
Greater Farallones National Marine Sanctuary
Santa Cruz
Gilroy
Turlock
Monterey Bay National Marine Sanctuary
Salinas
Monterey

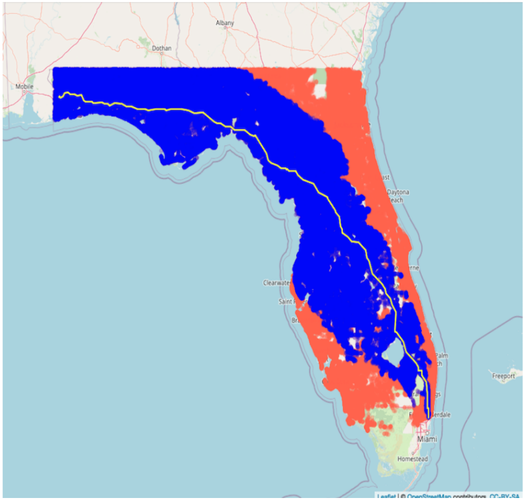


Figure: Test paths on BAY(left) and FLA(right)

6. Parallel reading of the input file

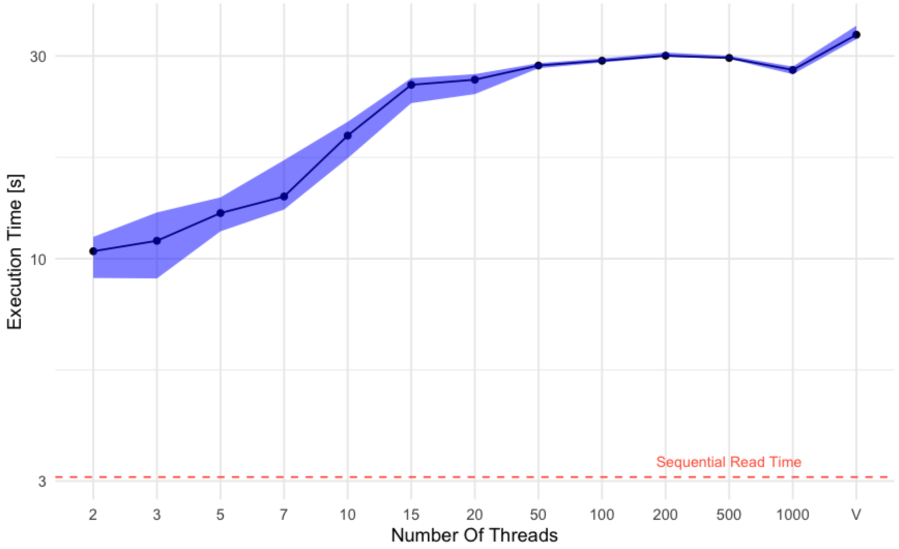
Parallel Read: approach 1

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

- N threads runs freely to read the entire file
- Only one file descriptor is shared among all the threads (this means that when thread t_i performs a `read` opearation all the other threads are waiting for it to finish)

6. Parallel reading of the input file

Parallel Read: approach 1 - Results on FLA map



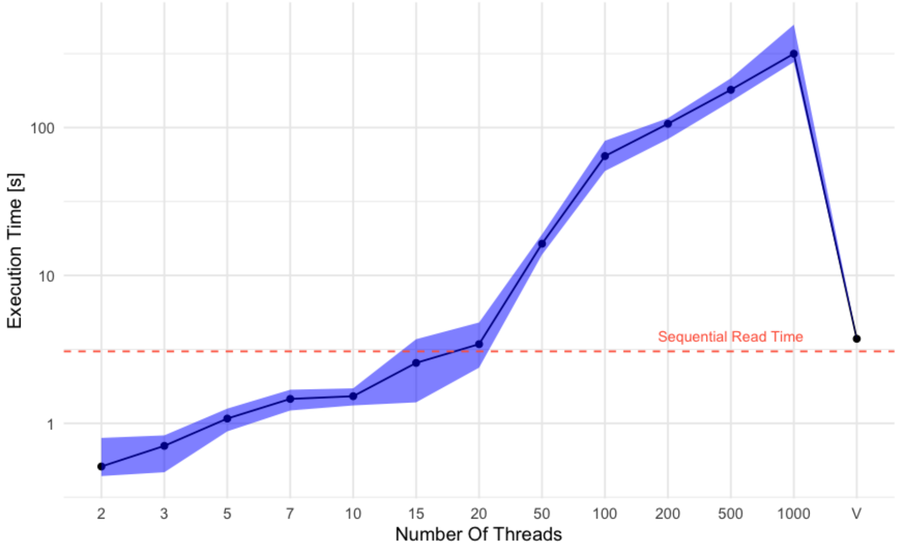
6. Parallel reading of the input file

Parallel Read: approach 2

TODO explain...

6. Parallel reading of the input file

Parallel Read: approach 2 - Results on FLA map



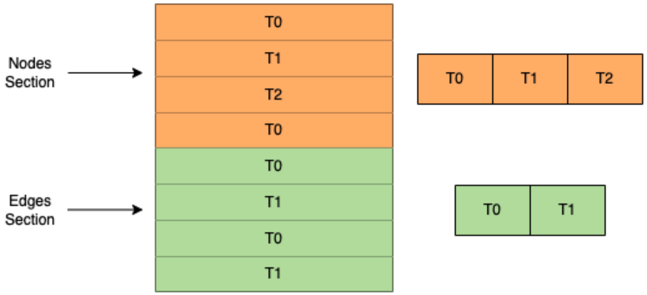
6. Parallel reading of the input file

Parallel Read: approach 3

- Letting threads differentiate among *nodes section* and *edges section* to be able to read them togheter.
- Using a (NP, NT) mechanism to read each section of the input file (where NP is the number of partitions the section is divided in and NT is the number of threads that have to read all the partitions).

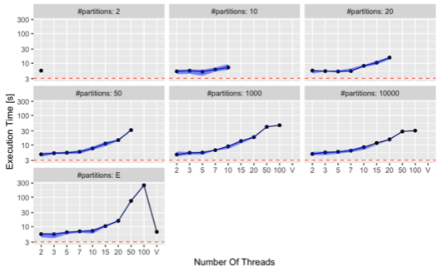
6. Parallel reading of the input file

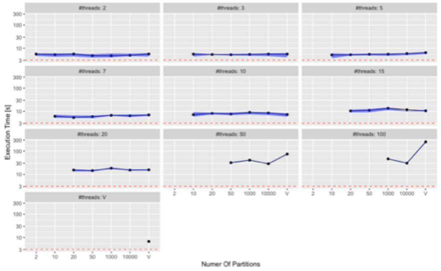
Parallel Read: approach 3



6. Parallel reading of the input file

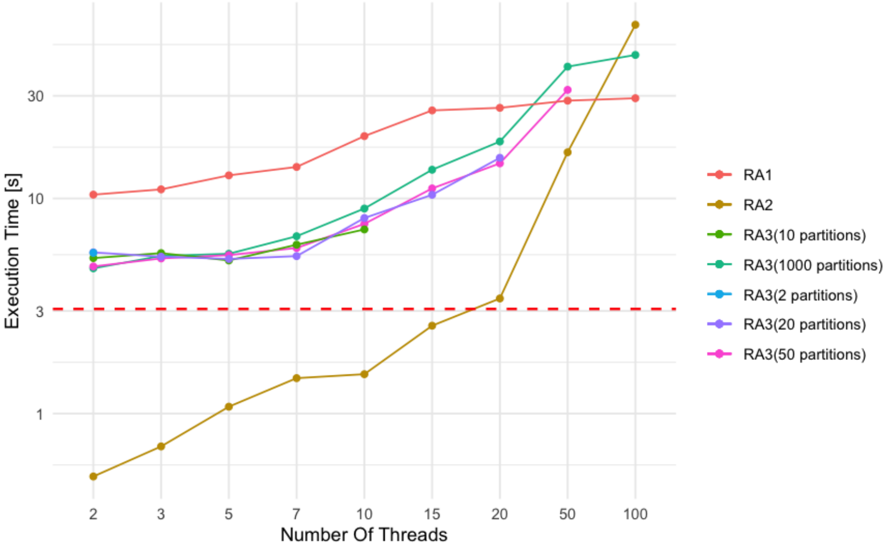
Parallel Read: approach 3 - Results on FLA map





6. Parallel reading of the input file

Final Results on FLA map



6. Parallel reading of the input file

Best performing model results

By using the most effective model we have found, Parallel Read Approach 2 with 2 threads, we have measured time performance over all the benchmark files:

	RA2 (2 threads)	**Sequential**	**Speed-Up**
BAY	0.1936s	0.9366s	79.3%
FLA	0.5103s	3.0650s	83.4%
USA	14.1492s	56.4445s	74.9%

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 three approaches to face the problem of parallelizing the A* algorithm:

- **First Attempt (FA)**: a trivial algorithm that simply works as the sequential algorithm but gives the possibility of executing it in a multithread fashion by sharing the common data structure *OPEN SET* among a variable number of threads.
- **Hash Distributed A* (HDA*)**: it puts in action a more 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.

7. Parallel A*

First Attempt (FA)

7. Parallel A*

HDA

7. Parallel A*

PNBA*