A*: Analysis and Comparison

Salvatore Licata
Lorenzo Ferro
Alessandro Zamparutti

August 2022

Contents

1 Introduction

We implemented the A* algorithm, both parallel and sequential versions, on large and weighted benchmark graphs. Then we studied the performance and compared the two algorithms.
The project implied:

- reading of large weighted benchmark graphs
- run of A* parallel, sequential and standard Djkstra algorithm
- comparison considering computation time, memory usage and visited nodes

2 Graph reading

Starting from a .map file (standard format for maps representation, using ascii characters to represent walls and passable terrain) the `graph_generator.py` python script is able to create a .txt with the format required by our A* implementation.
In the benchmark section we have:

- huge-graphs: graphs with millions of nodes, used for comparisons on large dimensions
- maze: graph with a small number of solutions (few path available from a point to another)
- street-maps: true city grid maps with different sizes
- small-graphs: generated by us used for testing

The weight of the edges can be randomized by the script in order to raise the level of difficulty in finding a path (all the grids have only edges with unitary weight).
The file is processed sequentially, creating all the nodes first and then the edges.

3 Basic structures

The project started by implementing all the basic structures needed by the algorithm, which are as generic as possible to encourage reuse and are dynamically allocated and implemented using lists to have no static limitations.
Each structure has a create function for allocation and a destroy one for disallocation of the memory.

3.1 Hash table

It does not allow duplicate keys and it uses multiplicative modular method with golden ratio:

$$golden\_ratio = \frac{(\sqrt{5}-1)}{2} \tag{1}$$

$$P = 8191 \tag{2}$$

$$hash(key, module) = ((key * golden\_ratio)\%P)\%module \tag{3}$$

The structure re-allocates itself when it reaches 75% of the maximum capacity. In case of collisions, it concatenates the elements in a list (keeps in the first place the last inserted item, exploiting the locality principle).

3.2 Heap

It is used as a priority queue, with the possibility to have MAX or MIN priority, and it uses a hash table for faster addressing and as a support for changing priorities. It has a lower and upper threshold for reallocation.

3.3 Queue

Generic list with head and tail pointers to an item. FIFO queue with head extraction and tail insertion.

3.4 Stack

LIFO implementation with both insertion and extraction occurring on the head.

3.5 Graph

Generic graph with the possibility to be UNDIRECTED or DIRECTED. It has a pointer to a generic data structure (useful when creating the graph from different domains such as 2D or 3D maps). A custom function can be passed as a parameter to properly read all the additional data.
Each vertex has a true_cost, which is the cost of the path from the start to that specific vertex, and a heuristic_cost, an estimation of the cost from that node to the destination node. The graph also contains a hash table of all the nodes used to search efficiently (a good improvement in the graph generation).

3.6 Message queue

It is composed by an array of queues, with customizable length. The receive and send methods are protected by a mutex to avoid race conditions, and they accept an id as parameter to send/receive a message to/from a specific queue. It also has a function to check if all the queues are empty, which is used in the parallel A* algorithm for the terminate detection.

4 A* Algorithms

A* is a best-first search algorithm based on the use of a heuristic function which estimates the cost from a source node to a destination node.

Both parallel and sequential implementations are based on two sets, the OPEN and the CLOSED one. The former is implemented using a priority queue (our heap 3.2) and contains nodes that have been discovered but not expanded yet, the latter is implemented with a hash table (3.1) and contains nodes that have been expanded.

All nodes contain a link to their parent, if any, which is used to keep track of the path found by the algorithm. At the end, the solution is inserted in a stack (3.4) pushing from the goal node and going backward, this way we do not need to reverse the path.

4.1 Sequential A*

Starting from pseudo-code [2]

```
1 Initialize OPEN to {s0};
2  while OPEN ≠ ∅  do
3    Get and remove from OPEN a node n with a smallest f(n);
4    Add n to CLOSED;
5    if n is a goal node  then
6       Return solution path from s0 to n;
7    for every successor n' of n  do
8       g1 = g(n) + c(n, n');
9       if n' ∈ CLOSED  then
10          if g1 < g(n')  then
11             Remove n' from CLOSED and add it to OPEN;
12          else
13             continue;
14      else
15          if n' ∉ OPEN  then
16             Add n' to OPEN;
17          else if g1 ≥ g(n')  then
18             continue;
19     Set g(n') = g1;
20     Set f(n') = g(n') + h(n');
21     Set parent(n') = n;
22 Return failure (no path exists);
```

where:

- s_0 : is the starting node
- $g(n)$: is the cost of the best known path from s_0 to n
- $h(n)$: is the heuristic function
- $f(n)$: is the sum $g(n) + h(n)$, used in the OPEN set to calculate the priority

4.2 Parallel A*

Implementation of the decentralized A* algorithm, where each thread has its own CLOSED and OPEN sets. When a thread discovers a neighbouring node, it chooses the thread which will expand it and sends a message to it.
This version of parallel A* is often referred to as HDA* (Hash Distributed A*).
When a thread finds a solution, it compares the cost with the global one and if it is lower, it changes the cost with the new one (the global cost is protected with a lock).

4.2.1 Pseudo-code

Starting from pseudo-code [2]

```
1 Initialize OPEN_p for each thread p;
2 Initialize incumbent.cost = ∞;
3 Add s_0 to OPEN_ComputeRecipient(s_0);
4 In parallel, on each thread p, execute 5-31;
5  while TerminateDetection() do
6     while BUFFER_p ≠ ∅ do
7       Get and remove from BUFFER_p a triplet (n', g_1, n);
8       if n' ∈ CLOSED_p then
9           if g_1 < g(n') then
10              Remove n' from CLOSED_p and add it to OPEN_p;
11          else
12              Continue;
13      else
14          if n' ∉ OPEN_p then
15              Add n' to OPEN_p;
16          else if g_1 ≥ g(n') then
17              Continue;
18      Set g(n') = g_1;
19      Set f(n') = g(n') + h(n');
20      Set parent(n') = n;
21    if OPEN_p = ∅ or Smallest f(n) value of n ∈ OPEN_p ≥ incumbent.cost then
22      Continue;
23   Get and remove from OPEN_p a node n with a smallest f(n);
24   Add n to CLOSED_p;
25    if n is a goal node then
26       if path cost from s_0 to n < incumbent.cost then
27          incumbent = path from s_0 to n;
28          incumbent.cost = path cost from s_0 to n;
29    for every successor n' of n do
30      Set g_1 = g(n) + c(n , n');
31      Add (n', g_1, n) to BUFFER_ComputeRecipient(n);
32  if incumbent.cost = ∞ then
33    Return failure (no path exists);
34  else
35    Return solution path from s_0 to n;
```

where:

- $BUFFER$: structure used for thread communication (4.2.3)
- *incumbent* : the shared best known path
- $TerminateDetection$: function to check if the path search is completed 4.2.4
- $ComputeRecipient$: function that choses the thread that will expand the node (4.2.2)

4.2.2 Compute recipient

A function which is able to map the node id into a thread identifier, and which is used to decide to which thread send a message to, with the node to be expanded. This function can considerably change the performance of the algorithm.
We provided two different implementations:
The basic module is the simplest one and it is computed as:

$$node\_id \ \% \ number\_of\_threads$$

A more efficient function is the multiplicative hashing computed as [6]:

$$h(node\_id) = ((a * node\_id \ + \ b) \ \% \ p) \ \% \ number\_of\_threads$$

where:

- p: large prime number (we used INT_MAX $= 2^{31} - 1$)
- a: random integer [1,p-1]
- b: random integer [0,p-1]

There are other well known hash functions suitable for parallel A* (see 6.3), but we did not implement them.

4.2.3 Thread communication

A message queue (3.6) structure is used to allow threads to communicate: a thread computes the recipient and sends a message using the thread id as recipient with the nodes of the edge and the cost.

4.2.4 Terminate Condition

We implemented a custom terminate detection, starting from the idea of the "sum flags" method (mentioned in the paper "Parallel A* Graph Search", [4]), also exploiting our data structures.
When a thread has its OPEN set empty or there aren't any elements that will improve the cost of the path, it sets its termination_flag.
A thread waits on its condition variable only if it has no message pending and its termination_flag is set. If all threads are waiting, the program terminates.
When a thread sends a message to another one, it signals its condition variable in order to wake it up.

```
1 lock mutex
2 while termination_flags[this_thread] and BUFFER_(this_thread) = ∅ and
        not program_terminated do
3     if counter ≥ n_threads - 1 and all BUFFER empty then
4         program_terminated = 1
5         signal all condition variables
6     else
7         counter++
8         wait on condition variable of this_thread
9         counter--
10 unlock mutex
11 if program_terminated then
12   return true
13 else
14   return false
```

where:

- *termination_flags*: array of flags set by a thread when the OPEN set is empty or it does not have any node with a minor cost then the actual path
- *program_terminated*: flag set by last thread awake when it detects the termination of the program
- *counter*: counter of sleeping threads
- *mutex*: global mutex to protect all shared resources
- $BUFFER$: message queue of each thread

5 Experimental evaluation

Here we report the results of the tests executed to compare the performances between the sequential and parallel versions of A*.
All the tests were performed on a Linux environment.

5.1 Time performance

To test the parallel A* algorithm we changed the number of threads and the function used to compute the recipient (4.2.2), while using the same input data (same map, same starting node and destination node), then we compared the results with the ones from sequential A*.

5.1.1 Grid Milan

The graph generated from the grid of Milan (1024x1024) has around 800k nodes and has randomly generated weights to increase the difficulty to find the path with the best cost.
The Figure 1 represents a drawing of the path found by the algorithms with the data that we used for testing.

Figure 1: Milan map with path found

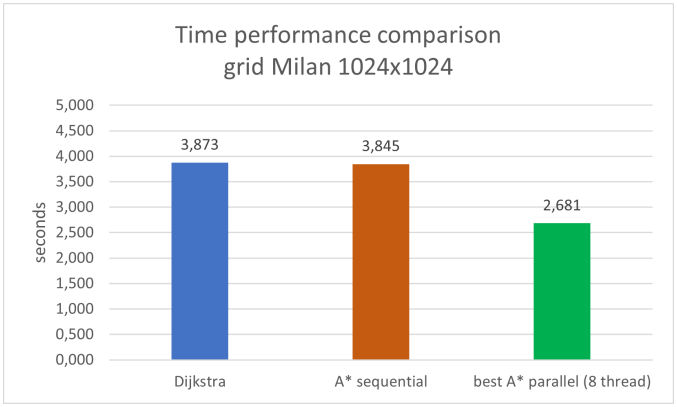

Figure 2: Performance comparison between different algorithms on grid Milan

Looking at Figure 2, and taking into account that we chose the most efficient number of threads for parallel A*, we can notice a better performance compared to the sequential A* and the Dijkstra algorithms.
In order to find the best result, we tried parallel A* changing the number of threads and the compute recipient function: Figure 3 shows all the experimental results of the tests.
Comparing the two compute recipient functions (simple module and multiplicative hash) on this relatively small graph, it can be observed that when running with limited number of threads the performances are similar.
With the increase of the level of parallelism the performances are no longer comparable, and the algorithm with multiplicative hash function is faster.

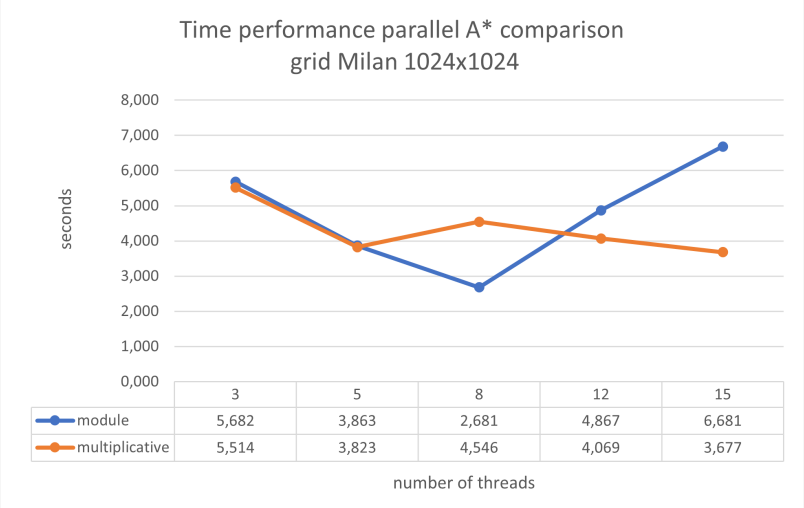

Figure 3: Performance comparison with different number of threads and compute recipient function

5.1.2 Large Map

To better test our algorithms we used larger graphs, always with random weights, with about 4 Millions nodes.

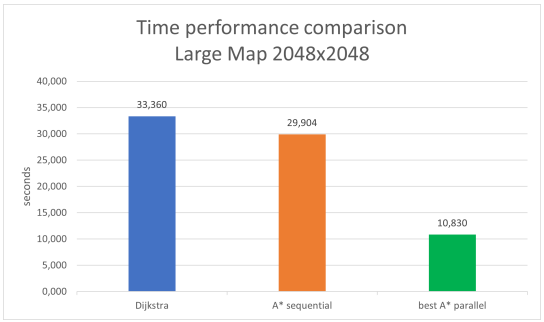

Figure 4: Performance comparison between different algorithms on large map

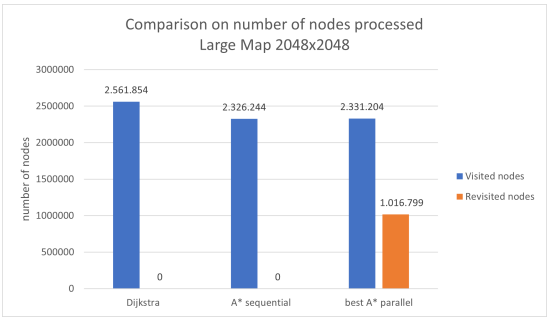

Figure 5: Performance comparison between different algorithms on large map

As we can see above (Figure 4 and Figure 5), the parallel version outperforms both the sequential one and Dijkstra algorithm.
Instead, the sequential version is the one which visits the least number of nodes (as expected).
Since the heuristic is admissible, the sequential version does not revisit any node, while the parallel one revisits a large number of nodes due to the OPEN and CLOSED list being local to the thread.
In the following page we can see a visual representation of the path found inside the map (Figure 6).

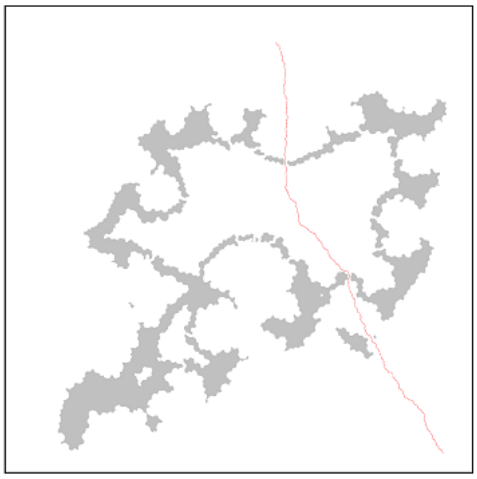

Figure 6: Large map with path found, highlighted in red

Now we will turn our attention to the choice of the compute recipient function.

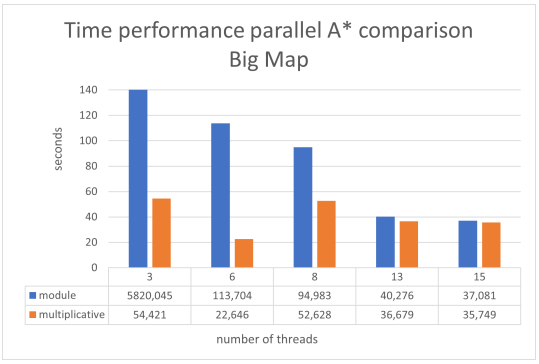

Figure 7: Performance comparison with different number of threads and compute recipient function

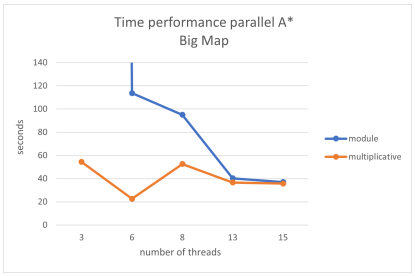

Figure 8: Tendency comparison with different number of threads and compute recipient function

By looking at the results on Figure 7, it can be observed that the multiplicative hash function is more stable when changing the number of threads.
For small level of parallelism (eg. 3 threads) the module hash needs almost two hours to find the correct path, while exploring and revisiting a large number of nodes.
The seconds needed to find a path decrease considerably when increasing the number of threads, but for some values (mostly even values of threads eg. 12), the time to converge to a result greatly increases again.
Instead, the multiplicative hash has generally better performances, but we found some limits on the number of threads (again even values from 16 onwards) beyond which the time increases.
We think the problem is due to the module operation with respect to the number of threads. A possible solution is discussed later (6.2).

5.2 Memory occupation

To track the memory occupation, we used the Valgrind-Massif [1] tool, which is a heap profiler. It measures how much heap the program uses: this includes both the useful space and the extra bytes allocated for book-keeping and alignment purposes.
The graph generated by the program shows the memory occupation on specific time snapshots. The X axis indicates the bytes allocated/deallocated on the heap and stack(s), while the Y axis shows the actual memory occupation on that specific snapshot.

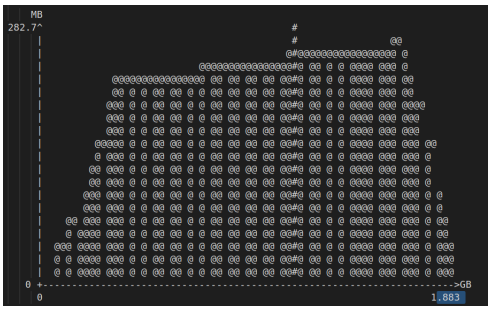

Figure 9: Memory occupation of sequential A* with Milan grid

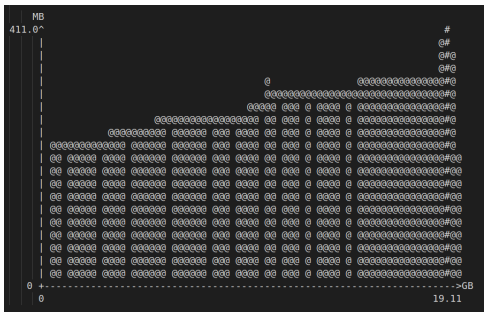

Figure 10: Memory occupation of parallel A* with Milan grid

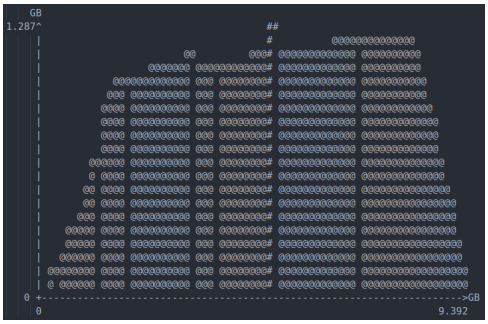

Figure 11: Memory occupation of sequential A* with large map

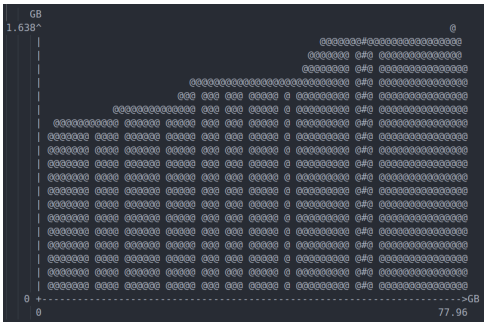

Figure 12: Memory occupation of parallel A* with large map

It is worth to mention that more than half of the used space is occupied by the graph structure and it is not due to the algorithms.
As expected, the memory occupation of the sequential A* is lower since it has only one OPEN set and one CLOSED set and it does not need to allocate additional structures for threads communication and synchronization. It can be also noticed the difference between the sum of all the allocation made by the two algorithms (78 GB allocated totally by parallel A* on Figure 12 against 9 GB by sequential A* for the large map on Figure 11).
To ensure that there are no memory leaks we used an option of the Valgrind tool which keeps track of all heap blocks allocated by the program.
As shown in the Figure 13, the algorithm does not have any memory leak.

```
HEAP SUMMARY:
    in use at exit: 0 bytes in 0 blocks
  total heap usage: 34,140,234 allocs, 34,140,234 frees, 415,487,890 bytes allocated

All heap blocks were freed -- no leaks are possible

ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 13: Valgrind output of memory leak of parallel A* with Milan grid.

6 Possible Enhancements

Looking at the experimental results (Section 5), our parallel A* implementation has good performance overall, but we thought about possible enhancements for our algorithm. In particular:

- parallelization of graph reading
- balancing multiplicative hash for different number of threads
- implementation of abstract zobrist hashing for the compute recipient function

6.1 Parallel Graph Reading

With the current sequential graph reading, the time for the graph creation is slower than path finding: for the large map it can last a couple of minutes. To improve the performance, a possible solution would be parallelizing the graph generation from the input file, although we don't know if it would be really effective.

6.2 Multiplicative Hash Balancing

During the tests on the compute recipient function, we noticed that with certain number of threads we have lower performances even if we expected a different result (Figure 8). A possible reason to this strange behaviour is due to the module operation at the end of our hash functions. We tried to normalize the result of the hash between 0 and 1, then multiplying it by the number of threads and taking the floor of the result. However, this solution was not as effective as we thought, so we decided to keep the unbalanced version.

6.3 Abstract Zobrist Hashing

Looking at the experimental tests (Section 5), we understood the importance of a good hash for the compute recipient function. Many scientific papers (eg. [2]) explain that the abstract zobrist hashing should have better performances than the multiplicative one, because it can balance better the load of the threads and minimize the communication overhead. We tried to implement it achieving poor results and, since it needed to store additional data inside the graph, it penalized other implementation's performances, thus we decided not to further explore it.

References

[1] Valgrind™ Developers. *Valgrind Documentation*. 2022. URL: https://valgrind.org/docs/manual/valgrind_manual.pdf.

[2] Alex Fukunaga et al. "A Survey of Parallel A*". In: (2017). URL: https://arxiv.org/pdf/1708.05296.pdf.

[3] N. Sturtevant. "Benchmarks for Grid-Based Pathfinding". In: *Transactions on Computational Intelligence and AI in Games* 4.2 (2012), pp. 144–148. URL: https://www.cs.du.edu/~sturtevant/papers/benchmarks.pdf.

[4] Ariana Weinstock and Rachel Holladay. "Parallel A* Graph Search". In: (2016). URL: https://people.csail.mit.edu/rholladay/docs/parallel_search_report.pdf.

[5] Wikipedia. *A* search algorithm*. 2022. URL: https://en.wikipedia.org/wiki/A*_search_algorithm.

[6] Kevin Wortman. *Multiplicative Hashing*. URL: https://www.youtube.com/watch?v=Kf2V77ut-B0.