PARALLEL A-STAR

Salvatore Licata
Lorenzo Ferro
Alessandro Zamparutti

August 2022

Contents

1 Introduction

Implementation of the A-Star algorithm both parallel and sequential versions on large and weighted benchmark graphs. Study on performance and comparison between the two algorithms.

The project implies:

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

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-Star 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 finiding a path (all the grids have only edges with unitary weight).

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 is implemented with multiplicative modular method with golden ratio:

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

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

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

The structure re-allocates itself when reaches the 3/4 of the maximum capacity, in case of collisions it concatenates the elements in a list (keeps in the first place the last inserted item).

3.2 Heap

It uses a hash table for faster addressing and it is used as a priority queue, with the possibility to have MAX or MIN priority as first element.

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 from the head.

3.5 Graph

Generic Graph with possibility to be UNDIRECTED or DIRECTED, has a pointer to a generic data (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, as long as the number of threads of the program. The receive and send methods are protected by a mutex to avoid race conditions, and receive an Id 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-Star Implementation

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

Both implementations are based on two set, the OPEN and the CLOSE one. The first is implemented using a priority queue (our heap 3.2) and contains nodes that have been discovered but not expanded yet, the second with a hash table (3.1) and contains nodes that have been expanded.

All vertexs contains a link to a parent vertex which is used to keep trace 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.

4.1 Sequential A*

Starting from pseudo-code [2]

```
Initialize OPEN to s_0;
 while OPEN ≠ 0  do
   Get and remove from OPEN a node n with a smallest f(n);
```

```
  Add n to CLOSED;
  if n is a goal node then
     Return solution path from s_0 to n;
  for every successor n' of n do
     g_1 = g(n) + c(n, n');
     if n' ∈ CLOSED then
         if g_1 < g(n') then
            Remove n' from CLOSED and add it to OPEN;
         else
            continue;
     else
         if n' ∉ OPEN then
            Add n' to OPEN;
         else if g_1 ≥ g(n') then
            continue;
     Set g(n') = g_1;
     Set f(n') = g(n') + h(n');
     Set parent(n') = n;
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 AStar algorithm, where each thread has its own CLOSED and OPEN set. When a thread discovers a neighbouring node it choses the thread that will expand it and sends a message to it.
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 OPENp for each thread p;
2 Initialize incumbent.cost = ∞;
3 Add s0 to OPENComputeRecipient(s0);
4 In parallel, on each thread p, execute 5-31;
5  while TerminateDetection() do
6     while BUFFERp ≠ 0 do
7        Get and remove from BUFFERp a triplet (n', g1, n);
8         if n' ∈ CLOSEDp then
```

```
         if g1 < g(n') then
            Remove n' from CLOSEDp and add it to OPENp;
         else
            Continue;
     else
         if n' ∉ OPENp then
            Add n' to OPENp;
         else if g1 ≥ g(n') then
            Continue;
    Set g(n') = g1;
    Set f(n') = g(n') + h(n');
    Set parent(n') = n;
  if OPENp = 0 or Smallest f(n) value of n ∈ OPENp ≥ incumbent.cost then
    Continue;
 Get and remove from OPENp a node n with a smallest f(n);
 Add n to CLOSEDp;
  if n is a goal node then
     if path cost from s0 to n < incumbent.cost then
        incumbent = path from s0 to n;
        incumbent.cost = path cost from s0 to n;
  for every successor n' of n do
    Set g1 = g(n) + c(n , n');
    Add (n', g1, n) to BUFFER ComputeRecipient(n);
if incumbent.cost = ∞ then
  Return failure (no path exists);
else
  Return solution path from s0 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 that is able to map the nodeId into a thread identifier, and which is used to decide to which thread send a message to with the node to expand. This function can considerably change the performance of the algorithm.
We provided two different implementation:
The basic module is the simplest one and it is computed as:

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

A more efficient function, but still simple to implement, is the multiplicative hashing computed as [4]:

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

where:

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

4.2.3 Thread communication

A message queue (3.6) structure is used to allows threads to communicate: a thread computes the recipient and sends a message using the thread destination with the node and cost.

4.2.4 Terminate condition

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 other threads are waiting the program terminates instead. When a thread sends a message to another one, it signals its condition variable in order to wake it up.

```
lock mutex
while termination_flags[this_thread] and BUFFER(this_thread) = 0 and
      not program_terminated do
    if counter ≥ n_threads - 1 and all BUFFER empty then
        program_terminated = 1
        signal all condition variables
    else
        counter++
        wait on condition variable of this_thread
        counter--
unlock mutex
if program_terminated then
  return true
else
  return false
```

Where

- *termination_flags*: array of flags set by a thread when the OPEN set is empty or 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 condition variables
- $BUFFER$: 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 the A* implementation.
All the tests are 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 the 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

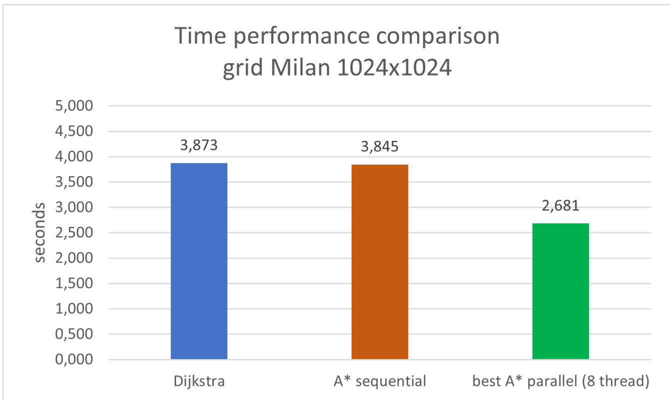

Figure 2: Performance comparison between different algorithms on grid Milan

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. To find the best result we tried the parallel A* changing the number of threads and the compute recipient function: the 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 the multiplicative hash function is faster.

5.1.2 Large Map

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

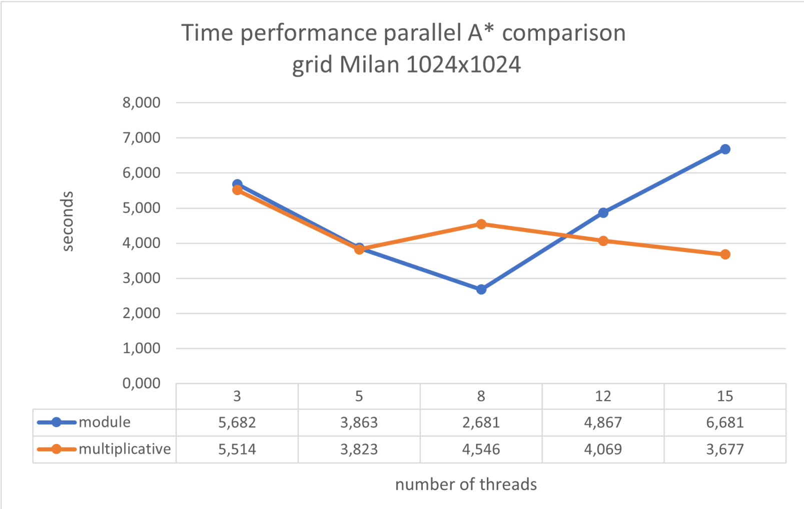

Figure 3: Performance comparison with different thread number and compute recipient function

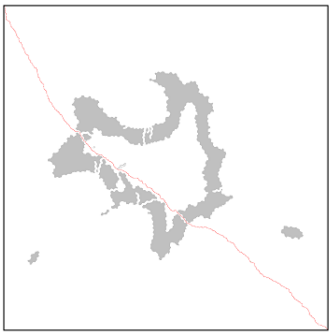

Figure 4: Large map with path found

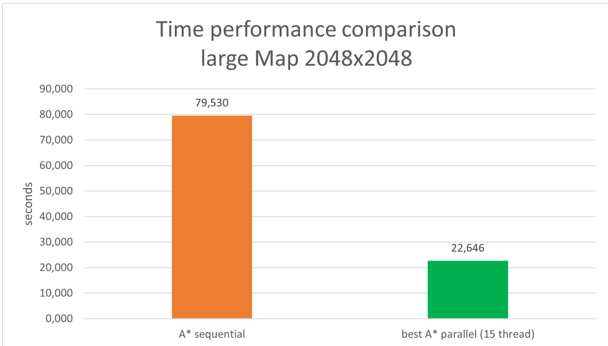

Figure 5: Performance comparison between different algorithms on large map

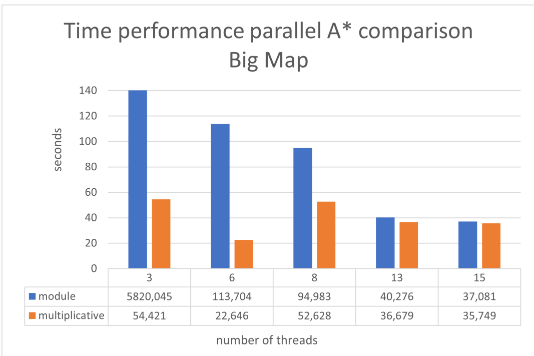

Figure 6: Performance comparison with different thread number and compute recipient function

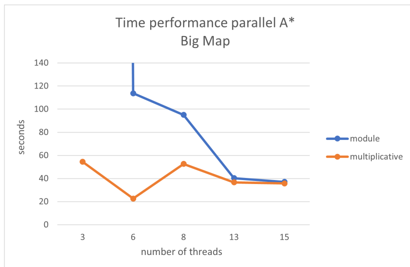

Figure 7: Tendency comparison with different thread number and compute recipient function

By looking at the results on Figure 6, 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. The multiplicative hash instead, 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.

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 memory 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.

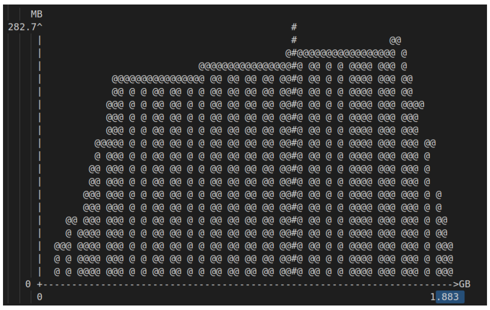

Figure 8: Memory occupation of sequential A* with Milan grid

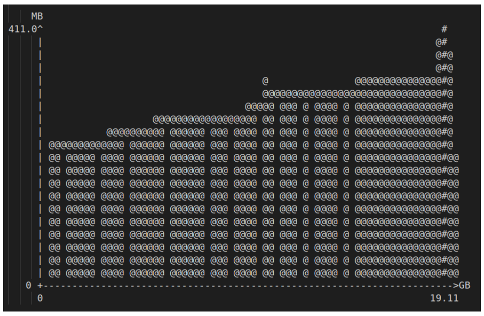

Figure 9: Memory occupation of parallel A* with Milan grid

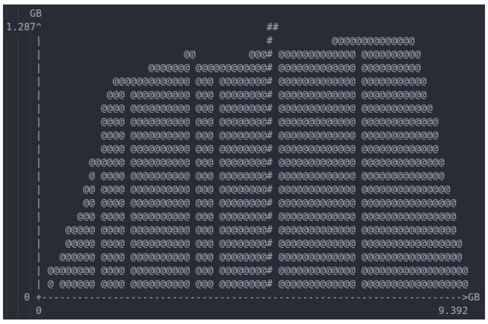

Figure 10: Memory occupation of sequential A* with large map

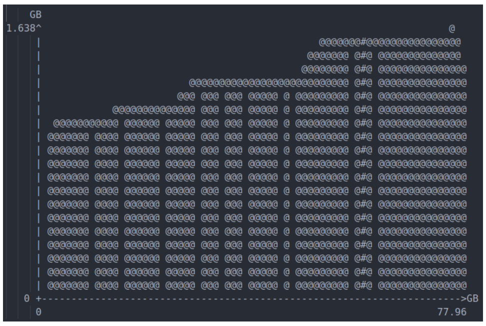

Figure 11: Memory occupation of parallel A* with large map

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 11 against 9 GB by sequential A* for the large map on Figure 10).
To ensure that there are no memory leaks we used an option of the Valgrind tool which keeps track of all heap blocks issued in response to calls to malloc. As shown by the the Figure 12 the algorithm does not have any memory leakage.

```
HEAP SUMMARY:
    in use at exit: 472 bytes in 1 blocks
  total heap usage: 85,036,921 allocs, 85,036,920 frees, 1,030,741,704 bytes allocated

Searching for pointers to 1 not-freed blocks
Checked 108,560 bytes

472 bytes in 1 blocks are still reachable in loss record 1 of 1
   at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
   by 0x48F16CD: __fopen_internal (iofopen.c:65)
   by 0x48F16CD: fopen@@GLIBC_2.2.5 (iofopen.c:86)
   by 0x10AF74: util_fopen (util.c:9)
   by 0x10A752: main (main.c:138)

LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 472 bytes in 1 blocks
        suppressed: 0 bytes in 0 blocks

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

Figure 12: Valgrind output of memory leak of parallel A* with Milan grid. The leak summary section shows the memory leakage of the program

6 Possible Enhancement

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

- parallelization of graph reading
- 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 is up to 70 seconds. To improve the performance a possible solution would be parallelizing the graph generation from the input file, but we did't find an efficient way to parallelize it at thread level.

6.2 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, 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 to remove 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] Wikipedia. *A* search algorithm.* 2022. URL: `https://en.wikipedia.org/wiki/A*_search_algorithm`.

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