Department of Information Engineering and Computer Science (DISI)
University of Trento, Italy

High Performance Computing - A.Y. 2022/2023

Parallel A-Star

Group 4
Matteo Greco 229328
Nicola Marchioro 232227

Contents

1 Introduction

A* (A-star) is a well-known search algorithm for finding the shortest path between two points in a graph. It combines the concept of Dijkstra's algorithm, with the concept of a heuristic function, which assists in guiding the search towards the goal by providing an estimate of the cost to reach the desired destination. The A* algorithm is widely used in computer science and artificial intelligence, most notably in pathfinding and navigation in video games, robotics, and autonomous vehicles.

1.1 Scoping

The focus of this project is to utilize parallel computing techniques to optimize the performance of the A* algorithm and to evaluate the results using benchmarking metrics. The parallel implementation of the A* algorithm will be developed in C language and by employing the MPI library. The algorithm will be implemented and tested on the *HPC@Unitrento* cluster to determine the impact of parallelism on the computational efficiency. The benchmarking of the parallel implementation will be conducted using a set of heterogeneous test cases, which will be used to evaluate the performance of the algorithm in terms of speedup and efficiency. The results will be compared with the performance of the sequential version of the A* algorithm to determine the impact of parallelism on the overall performance.

1.2 Background Research

The A* algorithm is widely recognized for its ability to reduce the search space through the use of a heuristic function [3]. In our background research, we aimed to gain a comprehensive understanding of previous studies that explored the potential of a parallelized version of the A* algorithm. Over the years, various implementations have been analyzed and evaluated. In [6], a parallel implementation of the A* algorithm, known as the Parallel New Bidirectional A* algorithm, was proposed. The authors claimed that this implementation combines the advantages of both bidirectional search and parallel execution, resulting in an efficient A* based search algorithm. Rafia's master's research project [4] explored the use of the A* algorithm on multicore graphics processors. In her study, she proposed a novel parallel implementation of the A* algorithm that targeted specific segments of the grid illustration of a map, rather than finding the shortest path as a whole. Another implementation was developed by Soha et al. [5]. They divided the starting cell's neighbors into local starting points and ran a sequential A* algorithm on each, utilizing threads. Once the threads returned their paths and scores, the one with the best score was selected as the final solution.

2 Problem Analysis

2.1 Shortest Path Finding problem

The Shortest Path Finding problem can be mathematically stated as follows:
Given a weighted graph, G = (V, E), where V is the set of nodes and E is the set of edges, and a source node, s, and a destination node, d, find the path, P = {v1, v2, ..., vk}, from s to d such that the total cost of the path, c(P), is minimized.

2.2 Sequential A* algorithm

Given a graph G=(V,E), a starting node s, and a destination node d, the A* algorithm aims to find the shortest path from s to d in G. We started our work from a C++ implementation of the algorithm, which can be found in [1], and heavily modified it, also porting it to C language. The algorithm maintains two sets of nodes, the open set and the closed set. The open set contains the nodes that have yet to be explored, and the closed set contains the nodes that have already been explored.
At each step of the algorithm, the node n in the open set with the lowest f(n) value is selected, where f(n) is defined as f(n) = g(n) + h(n). g(n) is the cost of reaching node n from the starting node s, and h(n) is a heuristic function, which provides an estimate of the cost of reaching the destination node from node n. The selected node is then moved from the open set to the closed set.
Next, the neighbors of the selected node are added to the open set, and their g(n) values are updated if necessary. The algorithm continues this process until the destination node is found or the open set is empty, indicating that the goal node is unreachable.

Finally, the path from the starting node to the destination node is reconstructed by following the chain of parent nodes from the destination back to the starting node. The path found by the A* algorithm is guaranteed to be the optimal solution if the heuristic function h(n) satisfies the following properties:

1. h(x) is admissible: h(x) never overestimates the actual cost of reaching the goal node from node x.
2. h(x) is consistent: for any edge (x,y) in the graph, $h(x) \leq w(x, y) + h(y)$, where w(x,y) is the weight of the edge and y is a successor of x.

3 Proposed Solution

In this section we present our implementation of the parallel A* algorithm. After reviewing the various academic papers concerning the different design implementations of the parallel algorithm, we concluded that Rafia's research [4] was the most promising, although the most complex. We then started to design our own implementation of parallel A* starting from the breakdown concept she proposed.

3.1 Design

The high-level design of the proposed solution is expressed in Figure 1. The splitting process is performed once at the start of the program after the root process imports the input data. Subsequently, root and worker nodes will cooperate to explore the grid and compute the shortest path. This approach enables the concurrent calculation of a larger number of routes on vast maps.

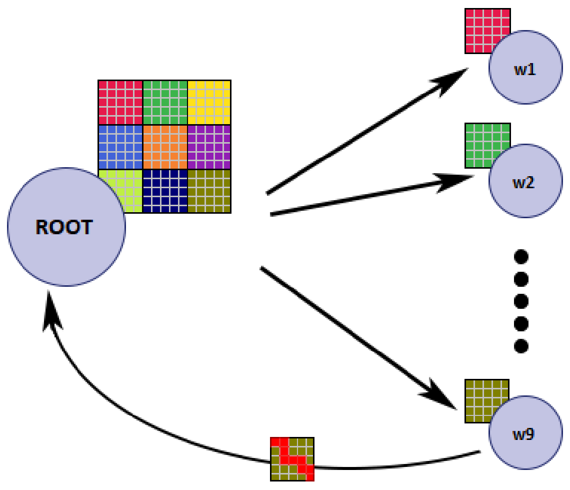

Figure 1: High-level design of the proposed solution

More in detail the algorithm follows these main steps:

1. The initial grid is divided into smaller chunks.
2. Each sub-grid is assigned to a worker node.
3. Every time the root explores a new cell, each worker node executes a local A* algorithm to compute the path from the localSource to the localDestination.
4. When needed the computed paths are transferred back to the root process, which will combine them to obtain the global path from Source to Destination.

3.2 Implementation

The experiment involves the exploration of a graph represented by a 2D matrix, edges are created between adjacent cells and all have the same weight associated with them. Each cell retains a pointer to its parent this way it's easy to traverse the path once the destination is reached. MPI directives are employed to allow communication between the root node and the workers, in particular.

- `MPI_Scatter`: is used to perform the initial matrix split. The root node reads the input file, gets the number of worker nodes requested and equally divides the workspace among them.
- `MPI_Send`: is used to share localGrid states. When a worker computes a relevant localPath it uses consecutive transfers to communicate the state to the root node.
- `MPI_Isend`: is used to increase communication performances. In particular, it is used to make the grid sending asynchronous. Since we must communicate the grid dimension to the root node, we can do it asynchronously while actually preparing the grid.
- `MPI_Recv`: is used to receive both synchronous and asynchronous communications between the processes.

A localGravity is assigned to each worker node, a cell, within the localGrid, which is estimated to be the closest cell to the destination, that will be traversed by an optimal path. Moreover, at each root iteration on a new cell, its position is sent to the workers, they will use the information to calculate a local starting cell using the process just described. Combining this information they are able to compute an optimal path between the candidate localSource and the localGravity, store the path, and send it to the root when requested to do so.

3.3 Testing Setup

3.3.1 Maze Generation

To evaluate our implementation, a custom python script was developed to convert a bi-tonal image into a text file. The script maps black pixels as 0s and white pixels as 1s, providing a convenient method for adding test cases for the parallel algorithm. This allows us to effectively test the performance of our implementation. The file is then enriched with information regarding the grid dimension, source position and destination position.

3.3.2 PBS Configuration

A custom bash script was developed to measure the performance of our application through benchmarking. The script automates the submission of jobs to the *HPC@Unitrento* cluster, allowing for the testing of multiple scenarios. The script performs multiple runs on each test maze, dynamically altering the number of CPUs, nodes, processes, and place strategies. Performance measurements are then taken and recorded for analysis. This approach provides a systematic and efficient method for evaluating the performance of the application.

3.4 Benchmarks

3.4.1 Speedup & Efficiency

Figure 2 represents the algorithm speedup, it shows how the speedup increases proportionately with the number of processors involved in the computation. However, it is also evident that the speedup is significantly low compared to the performance obtained through a serial approach. After analyzing the phenomenon we started wondering what could cause it, our idea is that this may be due to the time consumed in copying the data structure during communication between the root and worker nodes. A similar discussion can be posed for the efficiency graph (Figure 3), where it's even more evident how increasing the number of workers decreases the algorithm efficiency.

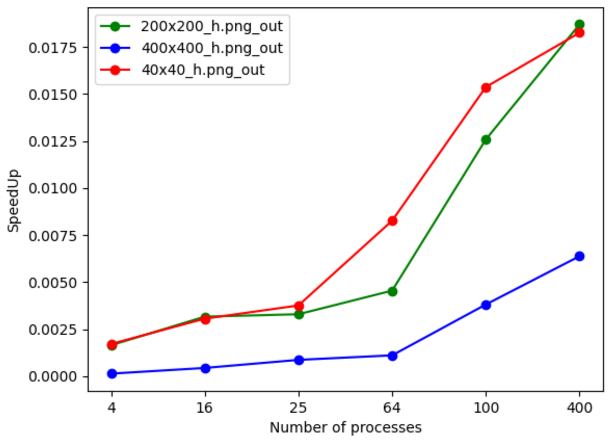

Figure 2: Speedup graph

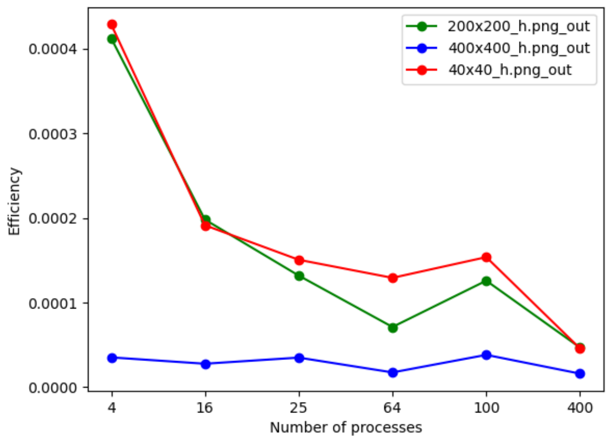

Figure 3: Efficiency graph

Figure 4 illustrates the average execution time on a single input sample, but with a varying number of workers. Three metrics were tracked for each computation, the worker time represents the total time spent by the assigned workers to compute or retrieve the relevant path for the cell being analyzed. It is worth noting that the total time is not simply the sum of the other metrics, as it also takes into account the MPI communication overhead. This overhead increases with the number of workers, as each worker computes shorter routes, therefore the algorithm requires a higher amount of communication to generate the complete path. The graph also reveals a strange behaviour of the algorithm, the execution time starts low, before having a spike followed by a descending curve which is the expected behaviour. We suppose this happens because having fewer workers means that each worker covers a wider problem space, hence an exploration in the wrong direction has a bigger impact on the algorithm.

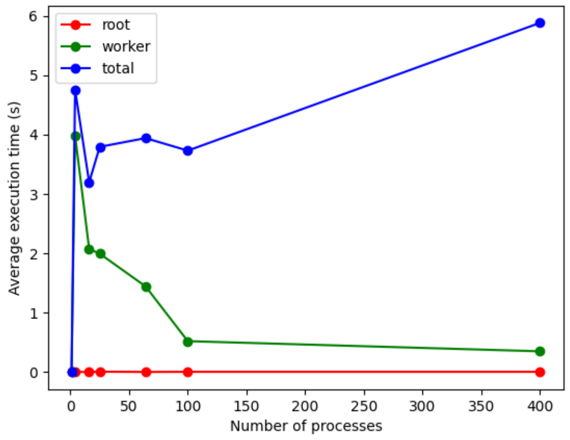

Figure 4: Average execution time graph for one specific input sample ($200x200_h$)

4 Discussion

There are several reasons of why parallelizing the A* algorithm with MPI may not result in improved performance. Some of the possible causes of inefficiency are the following:

- *Load balancing*: In order to achieve a speedup from parallelization, the search space must be distributed evenly across the processors. However, in a search algorithm like A*, the cost of exploring different parts of the search space can vary significantly, leading to unbalanced workloads and poor performance.
- *Limited parallelism*: The A* algorithm is inherently a sequential algorithm, and the order in which nodes are explored is dependent on the cost function. This limits the potential for parallelism, as it is not possible to simply divide the search space into separate regions and have each processor work on its own region.
- *Synchronization overhead*: The open list must be updated dynamically as the search progresses, which requires synchronization between the processors. This synchronization overhead can offset any potential speedup from parallelization.
- *Overhead of MPI communication*: MPI communication between processes can introduce significant overhead. This can be particularly problematic since the open list employed by A* needs to be updated continuously as the search advances.

In conclusion, the testing phase revealed that our parallel solution was inefficient, with the sequential algorithm outperforming it in every test case. Besides the motivations mentioned above, we think that the key design assumption of relying on the local gravity computed by each worker may result in the exploration of a misleading path, ultimately ensuing in a costly search.

References

[1] A* search algorithm. `https://www.geeksforgeeks.org/a-search-algorithm/`. Accessed: 01/02/2023.

[2] Alex Fukunaga, Adi Botea, Yuu Jinnai, and Akihiro Kishimoto. A survey of parallel a*. 2017.

[3] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[4] Rafia Inam. A* algorithm for multicore graphics processors. *Gotemburgo, Suecia. Recuperado el*, 19, 2010.

[5] Al-Jebreen Mariam Arshad, Soha S. Zaghloul. Parallelizing a* path finding algorithm. *International Journal of Engineering and Computer Science*, 2017.

[6] Luis Henrique Oliveira Rios and Luiz Chaimowicz. Pnba*: A parallel bidirectional heuristic search algorithm. In *ENIA VIII Encontro Nacional de Inteligência Artificial*, 2011.

[7] Pratyaksa Ocsa Nugraha Saian, Suyoto, and Pranowo. Optimized a-star algorithm in hexagon-based environment using parallel bidirectional search. *2016 8th International Conference on Information Technology and Electrical Engineering (ICITEE)*, pages 1–5, 2016.

[8] Daniel Sundfeld, George Teodoro, and Alba Cristina Magalhaes Alves de Melo. Parallel a-star multiple sequence alignment with locality-sensitive hash functions. In *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*, pages 342–347, 2015.

5 Appendices

Table 1: Execution times on $40x40_{\mathrm{e}}$

Mode	Root_time	Workers_time	Total_time
default_4	0.00003	0.01712	0.01750
default_16	0.00005	0.06783	0.07992
default_25	0.00009	0.04863	0.07972
default_64	0.00012	0.05709	0.11004
default_100	0.00015	0.01773	0.10169
default_400	0.00033	0.03034	0.59217
scatter_4	0.00004	0.01682	0.01743
scatter_16	0.00005	0.03703	0.04495
scatter_25	0.00008	0.04329	0.06693
scatter_64	0.00013	0.05364	0.10596
scatter_100	0.00018	0.01599	0.10495
scatter_400	0.00032	0.02803	0.57809
pack_4	0.00003	0.01468	0.01529
pack_16	0.00005	0.03739	0.04601
pack_25	0.00009	0.03912	0.05858
pack_64	0.00015	0.04616	0.10694
pack_100	0.00017	0.01775	0.09932
pack_400	0.00026	0.02661	0.57526

Table 2: Execution times on $40x40_{\mathrm{h}}$

Mode	Root_time	Workers_time	Total_time
default_4	0.00036	0.59478	0.59996
default_16	0.00062	0.26356	0.45571
default_25	0.00048	0.22354	0.42676
default_64	0.00053	0.11376	0.43067
default_100	0.00050	0.08660	0.72016
default_400	0.00057	0.05045	1.23501
scatter_4	0.00032	0.54468	0.58662
scatter_16	0.00057	0.39754	0.51828
scatter_25	0.00052	0.36806	0.69413
scatter_64	0.00056	0.14331	0.68782
scatter_100	0.00048	0.04752	0.52733
scatter_400	0.00056	0.06182	1.37156
pack_4	0.00033	0.59394	0.59896
pack_16	0.00053	0.31003	0.40982
pack_25	0.00047	0.19839	0.43761
pack_64	0.00053	0.10255	0.44712
pack_100	0.00050	0.05936	0.58760
pack_400	0.00050	0.05063	1.18448

Table 3: Execution times on $100x100_e$

Mode	Root_time	Workers_time	Total_time
default_4	0.09793	6.05510	7.95376
default_16	0.08091	3.46151	4.48120
default_25	0.05907	1.90087	4.43408
default_100	0.02998	0.65150	5.23498
default_400	0.02046	0.57603	12.04240
scatter_4	0.08943	6.56746	8.36262
scatter_16	0.07358	3.15791	4.22788
scatter_25	0.05579	3.88656	5.59675
scatter_100	0.02999	0.66761	4.55272
scatter_400	0.02167	0.48597	11.90989
pack_4	0.09281	5.93750	8.29816
pack_16	0.07959	4.53864	6.17387
pack_25	0.05771	3.34479	5.04067
pack_100	0.03157	0.57784	5.19769
pack_400	0.02167	0.61029	12.25260

Table 4: Execution times on $100x100_h$

Mode	Root_time	Workers_time	Total_time
default_4	0.00289	1.52752	1.88941
default_16	0.00345	0.94192	1.87576
default_25	0.00182	1.17671	3.04552
default_100	0.00192	0.47774	3.14469
default_400	0.00238	0.24406	4.72265
scatter_4	0.00290	1.27277	1.62992
scatter_16	0.00408	1.51796	3.23104
scatter_25	0.00151	1.13715	2.15900
scatter_100	0.00228	1.05210	4.06896
scatter_400	0.00260	0.22200	4.39471
pack_4	0.00266	2.04702	2.64128
pack_16	0.00352	1.17124	1.69271
pack_25	0.00160	0.85425	2.04219
pack_100	0.00206	0.66268	3.63341
pack_400	0.00254	0.20181	4.61407

Table 5: Execution times on 200x200$_e$

Mode	Root_time	Workers_time	Total_time
default_4	0.05919	1.77402	2.17227
default_16	0.05542	6.35919	10.80072
default_25	0.07479	4.09839	7.23302
default_64	0.07254	1.67025	5.99685
default_100	0.06941	1.20728	6.17822
default_400	0.07142	0.52078	12.45779
scatter_4	0.05668	1.60814	1.98857
scatter_16	0.06113	6.66903	11.16506
scatter_25	0.07464	3.68859	7.33833
scatter_64	0.06812	1.41157	5.29199
scatter_100	0.06656	0.77025	5.69546
scatter_400	0.07087	0.58740	12.74499
pack_4	0.05343	1.46354	1.77567
pack_16	0.05696	6.73161	11.79154
pack_25	0.08433	5.26751	8.38850
pack_64	0.06876	1.61976	5.89070
pack_100	0.07027	0.94856	5.28656
pack_400	0.07385	0.62775	13.04039

Table 6: Execution times on 200x200$_h$

Mode	Root_time	Workers_time	Total_time
default_4	0.00386	4.07563	4.85368
default_16	0.00446	2.05330	3.05646
default_25	0.00944	1.84745	4.05313
default_64	0.00413	1.00838	3.11592
default_100	0.00605	0.36100	3.50903
default_400	0.00609	0.41201	6.71705
scatter_4	0.00385	4.30821	5.07092
scatter_16	0.00481	2.12179	3.44477
scatter_25	0.00963	1.65172	3.55891
scatter_64	0.00405	1.32200	3.74400
scatter_100	0.00665	0.39562	3.17326
scatter_400	0.00671	0.28362	5.54846
pack_4	0.00408	3.57745	4.34580
pack_16	0.00446	2.05245	3.08872
pack_25	0.00892	2.48594	3.76999
pack_64	0.00494	2.00588	4.96128
pack_100	0.00665	0.81145	4.50808
pack_400	0.00686	0.35982	5.37692

Table 7: Execution times on 400x400$_e$

Mode	Root_time	Workers_time	Total_time
default_4	0.07400	326.92282	552.46929
default_16	0.05728	124.79024	252.65306
default_25	0.06207	106.06339	152.07017
default_64	0.10632	0.34083	2.11263
default_100	0.16579	1.85223	8.26130
default_400	0.16623	0.92686	21.31885
scatter_4	0.04386	292.42886	527.79505
scatter_16	0.06502	123.23797	239.65723
scatter_25	0.05890	92.05637	139.64421
scatter_64	0.08634	0.53971	1.71350
scatter_100	0.15507	1.80222	8.67292
scatter_400	0.17275	0.95221	22.57917
pack_4	0.05411	395.60722	502.93413
pack_16	0.06416	130.59612	263.91182
pack_25	0.06281	114.25955	149.38535
pack_64	0.09096	0.44745	1.71285
pack_100	0.15456	1.65811	7.63781
pack_400	0.17466	0.92676	21.28423

Table 8: Execution times on 400x400$_h$

Mode	Root_time	Workers_time	Total_time
default_4	0.04077	296.28888	433.82836
default_16	0.03240	78.92290	110.38712
default_25	0.08401	39.30401	84.43172
default_64	0.06159	37.90685	65.97487
default_100	0.07644	8.75207	45.72861
default_400	0.10113	3.02767	75.52922
scatter_4	0.02156	245.90339	395.88138
scatter_16	0.03547	74.29171	96.60233
scatter_25	0.08523	32.20123	86.59043
scatter_64	0.06656	25.56536	55.30160
scatter_100	0.08037	9.66273	50.01189
scatter_400	0.09722	6.12871	79.22000
pack_4	0.01565	193.47293	402.74459
pack_16	0.03342	80.46836	104.58298
pack_25	0.08147	46.97282	98.42070
pack_64	0.06105	30.34232	54.62038
pack_100	0.08267	8.70071	54.43644
pack_400	0.17274	6.71462	81.80030

Table 9: Execution times on 400x400$_{\mathrm{m}}$

Mode	Root_time	Workers_time	Total_time
default_4	0.04076	85.56207	86.83680
default_16	0.04461	48.08263	51.75972
default_25	0.28874	71.50039	90.21049
default_64	0.06449	21.74818	41.28634
default_100	0.07101	39.11695	62.38118
default_400	0.26801	4.48529	106.64461
scatter_4	0.03614	77.71897	80.52644
scatter_16	0.09265	68.05092	73.11216
scatter_25	0.04129	70.17407	89.70544
scatter_64	0.06039	23.32382	43.97327
scatter_100	0.06244	20.64644	43.47245
scatter_400	0.09436	6.24572	121.35374
pack_4	0.04013	86.96227	88.31409
pack_16	0.04746	58.20016	63.82634
pack_25	0.04966	61.68717	78.80059
pack_64	0.05353	22.23551	38.75310
pack_100	0.06537	24.83908	48.97418
pack_400	2.77893	16.12055	142.68611