www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 6 Issue 9 September 2017, Page No. 22469-22476
Index Copernicus value (2015): 58.10 DOI: 10.18535/ijecs/v6i9.13

Parallelizing A* Path Finding Algorithm

Soha S. Zaghloul[1], Hadeel Al-Jami[2], Maha Bakalla[3], Latifa Al-Jebreen[4], Mariam Arshad[5] Arwa Al-Issa[6]

College of Computer and Information Sciences King Saud University,
Riyadh, KSA

Abstract:
Parallel processing is adopted with the aim to run multiple independent program tasks simultaneously on multiple CPUs with the objective of minimizing the execution time of the whole program. In this research, parallel A* is implemented on a shared-memory multiprocessor system. Experimental results are compared with the sequential A* to find an optimum or near-optimum path between two points in a grid map. The most prominent drawback of the A* algorithm is the long execution time since it examines each of the neighbor nodes, beginning at the start node going through to the goal node. Therefore, having different threads running simultaneously to find the path from each neighbor of start node to goal node is expected to reduce the computation time dramatically. Different parameters are used to assess the performance of the parallel version of the A* algorithm; namely, the execution time, the speedup, the scalability, and the efficiency. The experiments are conducted with different number of threads. Interesting results are given with respect to the previously mentioned four metrics.
Keywords: Parallel A*, Shared-memory multiprocessors, Path finding**.**

1. Introduction

A* is an informed search algorithm used to find the shortest path from a given start node to a goal node. It uses heuristic functions to reduce search space like Manhattan distances. Usually, the A* algorithm is used in games, robotics, path finding in grid maps, and motion planning problems. In general, the sequential implementation of A* requires a lot of computation time, which makes it impractical for domains that have huge search space.

Parallelizing the A* algorithm will overcome timing constraints problem, which is our main focus in this paper. Parallelization can be made by splitting path finding problem in sub parts where search space is divided among multiple threads. This allows different numbers of available cores to work simultaneously to find the best path, which will result in less computational time. Our contribution here will be beneficial for domains like motion planning problems where finding optimal solution requires a lot of time and memory-consuming expansions.

In this paper, we have introduced a parallel version of the A* algorithm, which can be implemented to make use of the available number of cores to speedup the execution of the A* algorithm. In most cases, our algorithm showed better performance than the sequential version in terms of execution time and speedup, particularly in large-sized grids.

This paper is organized as follow: in section 2, a background of the search algorithm and parallel processing is provided; section 3 comprises a comprehensive review of the most prominent state-of-the-art algorithms. In Section 4, the proposed solution is presented. The result and discussion of the proposed algorithm is presented in Section 5, and the last section 6 contains our conclusion.

2. Background

This section includes basics about Parallel processing and the A* algorithm. It describes features of parallel computing and how it is implemented by using parallel computers. Moreover, the architecture of parallel computers and parallel programming models commonly used is discussed in this section. The section further gives an insight about working of the traditional A* algorithm.

2.1 Parallel Processing

Previously, in software computation, the program was divided into several instructions; then these instructions were executed in a single processor sequentially. This process is called serial or sequential computing. In parallel computing, the program is divided into parts that can be executed in parallel. Each of these parts also consist of several instructions executed separately on different processors.

Parallel computers can be classified into three categories: the First is the Single Instruction Multiple Data (SIMD) type, where an instruction is executed by all processors operating on different data; in this type of parallel computing, the pipeline CPU architecture is used; the Second is the Multiple Instruction Single Data (MISD) type, where multiple processors with a single data stream are used and each processor operates on data using separate instructions. The Multiple Instruction Multiple Data (MIMD) is the third type of parallel computing; each processor deals with different instructions and works with different data.

Parallel programming models can be classified into three categories, which are the shared memory processor (SMP), the distributed memory (cluster), and the hybrid model. The model extensively used is SMP, where all the processors can share the main memory with a global address space that allow processors to use the data. In contrast, in the distributed memory architecture, a separate memory for each processor is used instead of having a shared memory with a global address space. Communication between processors occurs through message passing. However, the hybrid model integrates both shared and distributed memory. In this model, the threads on the same processor share the same address space, and the threads on different processors communicate with each other through messages to transfer data [1], [2] and [3].

In our proposed approach, we follow the second model in implementing the parallel A* algorithm. Thus we rely on a shared single memory for sharing the data between threads.

2.2 A* (A Star) Algorithm

The A* (A star) algorithm is a classic, widely-used search algorithm, developed in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael. It is a best-first search strategy that uses a heuristic to guide itself to find the shortest and optimal path between two points. Its effectiveness is based on producing a good heuristic estimator on the remaining distance from the current state to the destination.

To clarify the concept of A*, lets assume the environment as shown in Figure 1. The blue square is the starting point S, and red square is the ending point D. The white squares are walkable nodes, and the green squares are the obstacles or walls between them.

The algorithm uses two lists: an Open List that contains squares that are being considered to find the shortest path, and a Closed List that contains squares that do not have to be considered again. The A* algorithm begins at the starting point S and adds it to the Open List of squares to be considered. After examining all of its neighbour squares (The yellow square in Figure 2), the starting point S will move from the Open List to the Closed List, such that we don't need to look at it again. All of the neighbour squares will be added to the Open List to be checked. Then, choosing one of the neighbour squares on the Open List will depend on the path cost. The process is repeated until the destination is found. For each time, the parent square of its neighbours will be saved to help in tracing the path back to the source cell.

The A* algorithm uses an evaluation function f (n)

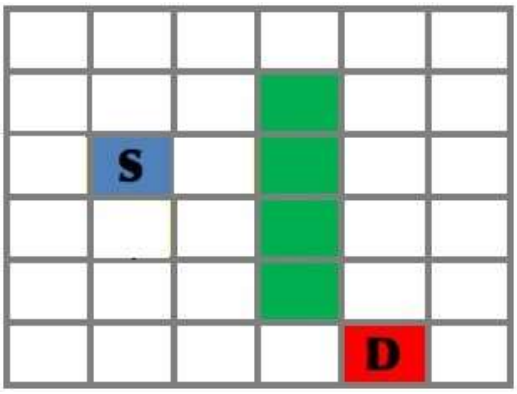

Figure 1. Start, destination and obstacle nodes in A* Search Algorithm

to guide the selection of the neighbour square.

$$f(n) = g(n) + h(n) \quad (1)$$

Where g (n) represents the cost of the path from the starting node S to any node, and h(n) represents the heuristic estimated

DOI: 10.18535/ijecs/v6i9.13

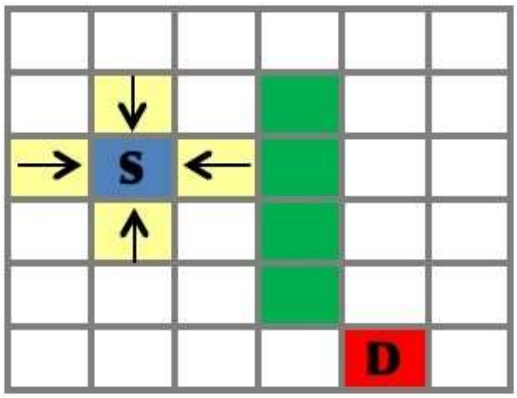

Figure 2. Parent relation to start point

cost from any node to the destination D. The A* turns into Dijkstra's algorithm if h(n) is 0; then only g(n) performs a role in the evaluation function. In addition, if h (n) is always less than or similar to the cost of traveling from n to the goal, A* can be used with certainty to find the shortest path. Figure 3 shows the pseudo code of A* algorithm.

The heuristic h (n) can be estimated in different ways depending on the allowed movement [4]; some of these ways are:

1. Manhattan distance
   It allows a move to a square grid in four directions (North, South, East and West). The following equation is used to calculate the Manhattan distance:

$$h(n) = D*(|\, node.x - goal.x\,| + |\, node.y - goal.y\,|) \quad (2)$$

   Where D represents the cost of moving one node to one of its neighbours.

   In the simple case, we can set D to be 1. The advantage of the Manhattan heuristic is that it runs faster than the other distance measures, and the main disadvantage is that it is used to find the shortest path to the goal; however, an optimal solution is not a certainty with this approach

2. Diagonal distance
   In this method, moving to a square grid in eight directions is allowed; however, computation speed is slower than that in the Manhattan method. The equation used to calculate the Diagonal distance is as follows:

$$h(n) = D*\max(|\, node.x - goal.x\,|, |\, node.y - goal.y\,|) \quad (3)$$

3. Euclidean distance
   It allows moving to a square grid in any direction. The equation used to calculate the Euclidean distance is as follows:

$$h(n) = D*\sqrt{((node.x - goal.x) \wedge 2 + (node.y - goal.y) \wedge 2)} \quad (4)$$

   Using the Euclidean heuristic as a heuristic is admissible, but usually results in an underestimation. Compared with the Manhattan heuristic, this heuristic is more expensive to apply as it additionally involves two multiplication operations and calculating the square root [4].

3. Related Work

Several modifications have been suggested ever since the first version of the sequential A* algorithm was introduced. Evidence suggests that there has been a remarkable progress in the performance of the A* algorithm due to these; however, there is room for its improvement. In this section, we present some of the recent and prominent state-of-the-art algorithms for A* search, namely, sequential and parallel. Mentioned below are the contributions of well-known A* algorithms proposed for different search domains.

3.1 Sequential A* Algorithm

Liang Zhang et al. in [5] proposed a novel hybrid method to find the optimum path planning for mobile robots. The A* algorithm and the genetic algorithm form the basis for this method. The workspace of the robot in this paper is previously known and consists of a grid map of its indoor environment. At first, the method find the shortest path by the A* algorithm using Manhattan distance as heuristic function. Next, it uses the genetic algorithm to optimize the path found. Matlab was used to simulate the proposed method and to compare it with the results of the two previously proposed methods. They achieved good results, one of them having the length of the path 26.26 in 0.92 seconds, which is less than that observed with method [6] 27.35 in 1.15 seconds.

Barnouti et al. [7] built a GUI application system that implements the A* search algorithm, which

```
1.  Create a node containing the goal state node_goal
2.  Create a node containing the start state node_start
3.  Put node_start on the open list
4.  While the OPEN list is not empty {
5.      Get the node off the open list with the lowest f and call it node_current
6.      if node_current is the same state as node_goal we have found the solution;
7.      break from the while loop
8.
9.      Generate each state node_successor that can come after node_current
10.     for each node_successor of node_current {
11.     Set the cost of node_successor to be the cost of node_current plus the cost
        to get to node_successor from node_current
12.     find node_successor on the OPEN list
13.     if node_successor is on the OPEN list but the existing one is as good or
        better
14.     then discard this successor and continue
15.     if node_successor is on the CLOSED list but the existing one is as good or
        better
16.     then discard this successor and continue
17.     Remove occurences of node_successor from OPEN and CLOSED
18.     Set the parent of node_successor to node_current
```

Figure 3. Pseudo code of A* algorithm

addresses the pathfinding problem to find the shortest path between source and destination. The problem of pathfinding in commercial computer games has to be solved in real-time, usually requiring memory and CPU resources. The algorithm is tested by using images that represent either a map that belongs to strategy games or a maze. The system performance is tested using 100 images for each map and maze. The overall performance of the system is acceptable; for more than 85% of the images, the shortest path between the source and destination could be found.

3.2 Parallel A* Algorithm

Luis Henrique and Luiz Chaimowicz in [8] aimed to improve the A* algorithm performance by implementing the bidirectional search algorithm as a parallel version for robotics, digital games, and DNA alignment. The sequential version of New Bidirectional A* (NBA*) uses two processes that execute in a sequential manner to find the path. In this paper, the researchers proposed PNBA* (Parallel New Bidirectional A*). In PBNA*, both search processes execute in parallel and that enhances its performance. Authors implemented the algorithm using C++ programming language on Linux operating system. They compared their algorithm with A* and NBA* for fifteen puzzles and grids pathfinding scenarios. They used random mazes with uniform and non- uniform costs. As a result, the execution time decreased significantly when using PNBA*.

Sandy Brand and Rafael Bidarra in [9] implemented several versions of parallel pathfinding algorithms, namely, Parallel Bidirectional Search (PBS), Distributed Fringe Search (DFS), and Parallel Hierarchic Search (PHS) to analyse their behaviors. They found that these three algorithms suffer from some limitations such as a large overhead yielding from far optimality, not scaling up to many cores, or their cache being unfriendly. Therefore, they proposed Parallel Ripple Search (PRS), which employs two essential cores for flooding at the path boundaries like PBS; then, each core uses A* flooding behavior for expanding toward each other. PRS relies on the opportunistic use of node collisions between collaborating cores. For assessing the performance of the PRS algorithm, it was compared with PBS, Fringe Search (FS), and classic A*. To conclude, in a PRS, the fact that more than two cores in large and complex maps can be utilized is exploited; the speedup factor is in the range 2,510 when compared with classic A*.

Mahafzah in [10] presented a parallel multithreaded A*heuristic search algorithm for a 15-puzzle problem. When using this algorithm, in the first phase, the search tree is expanded to a certain level by implementing the Limited Depth BFS algorithm. A number of threads are created depending upon the number of nodes generated within each level. It was implemented using the POSIZ threads library (Pthreads). The proposed algorithm has been evaluated analytically using different parameters; time complexity, space complexity, optimality, and completeness. Better performance was observed in case of parallel A* in terms of various performance measures. Particularly, in case of the parallel multithreaded A* algorithm, the time complexity was $O(\beta k/t)$; however, for the sequential A* algorithm, the time complexity was $O(\beta k)$, where βrepresents the branching factor, k represents the depth of the shallowest solution, and t is the number of threads.

In [11], Rafia presents her masters research project based on A* Algorithm for Multicore Graphics Processors. The author proposed a parallel version of A* to find the shortest path in several segments, instead of the complete path as a whole in the grid illustration of a map. They create eight threads in parallel, as each node has a maximum of eight neighbours. The program is implemented in CUDA, NVIDIAs programming platform for graphics processors, using data structure temporary list that

divides the shared memory into an array of eight places, one accessible by each thread. Experimental results of the improvement are presented in the paper demonstrating the efficiency of the proposed algorithm.

Phillips et al. [12] presented PA*SE, a parallel implementation of A* search algorithm, another version based on a relaxed independence check that allows a trade-off between higher parallelization at the cost of optimality, and finally a parallel implementation of weighted A*. The experimental results on up to 32 processors demonstrated a linear speedup over A* and weighted A*.

4. Proposed Solution

The main focus of this paper is to measure the performance of the parallel A* algorithm and compare it with a sequential version. We began by programing the well-known A* algorithm. Then, a parallel version was developed as shown in the Pseudo code section. Finally, we compared the results obtained from our experiments.

4.1 Pseudo Code

Pseudo Code of Parallel A*

1. Initialize the *grid*
2. Create a cell containing the start cell ***start_cell***
3. Create a cell containing the destination cell ***goal_cell***
4. Create a list of neighbors of ***start_cell***, ***adjacent_list***
5. Create the threads for each cell in ***adjacent_list*** as ***local_start***
6. For each thread, follow these steps:

    4.1 Put ***local_start*** on the open list
    4.2 While the ***open*** *list* is not empty {
        4.2.1. Get the cell off the ***open*** list with the lowest cost and call it ***current_cell***.
        4.2.2 If ***current_cell*** is the same state as ***goal_cell***, we have found the solution:
            Break from the while loop
        4.2.3. Generate each state ***successor_cell*** of ***current cell***
        4.2.4. For each ***successor_cell*** of ***current_cell*** {
            4.2.4.1 Set the cost of ***successor_cell*** to be the cost of ***current_cell*** plus the cost to get to ***successor_cell*** from ***current_cell***.
            4.2.4.2 If ***successor_cell*** is in ***open*** *list* but the existing one is as good or better.
                Then discarded this ***successor*** and continue.
            4.2.4.3 Remove occurrences of ***successor_cell*** from ***open*** list and ***closed***
            4.2.4.4. Set *the* ***parent*** of ***successor_cell*** to ***current_cell*** }

7. Wait for all threads to finish (join)
8. Get the path found by each thread
9. Find the best path (i.e. the one with minimum cost)

Figure 1. Parallel A* Algorithm

In the parallel A* algorithm, we set the neighbours of the start cell to be a local start and assigned one to each thread. The goal cell is the same for all the neighbours. In all threads, the A* algorithm will be initiated to find the best path from their respective local start to the goal node. Each thread will return the path along with its score, and the best path score will be chosen as the final path. Figure 4 shows our proposed algorithm.

5. Discussion

In this section, we present the results obtained from our experiments and dis- cuss the improvement in performance of the A* algorithm due to our proposed method. We compared the performance of sequential and parallel versions of the A* algorithm in terms of speedup and running time. For better analysis, we have tested our method on different problem sizes and number of processors. Following are the experiments environment that contain the specifications of hardware and software we have used to conduct all experimental runs. Subsequently, the obtained result will be discussed. Finally, we analyse our results with respect to other theoretical laws.

5.1 Experiment's Environment

The computer hardware used in the experiment comprises an Intel CPU with 4 processors and 8 cores at a speed of 2.40 GHz; the memory consisted of 16.00 GB RAM. The operating system used is Windows 10 operating system, and the code is written in Java using the Eclipse IDE (Integrated Development Environment).

In our experiments, we executed the parallel A* algorithm and the sequential A* algorithm on three problem sizes; first, with the problem size that has at least 60 seconds running time for sequential version, which is 2000*2000. Second, for 3000*3000, and finally for 4000*4000 (maximum size allowed based on available memory size). We ran the sequential version seven times for each problem size; then we took the minimum execution time value. Next, we ran the parallel version on different number of threads, ranging from one to eight (maximum number of processors available). Each run was performed seven times; after these iterations, we took the minimum value of execution time. Moreover, we tested our experiment on three different cases based on the number of neighbours of the start cell: first, a 3-neighbour configuration was used with the global start located at the corner of the grid. Second, a 5- neighbour configuration was used with the

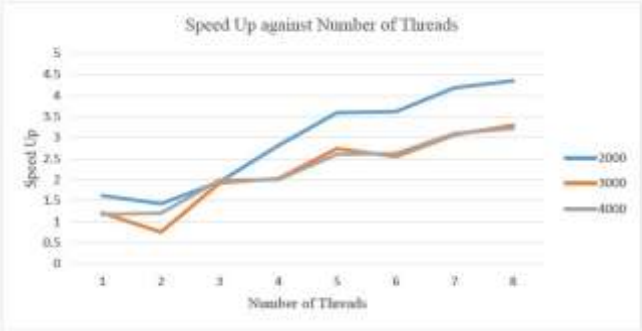

Figure 2. The speedup gained when increasing the number of threads

global start cell located at the border of the grid;

Table 1. Speedup results

Size of grid	Sequential execution time	Parallel execution time	Speedup
2000*2000	72.895	16.752	4.351
3000*3000	173.669	52.725	3.294
4000*4000	396.468	123.189	3.218

and last, a 8- neighbour configuration was used with the global start cell located at the centre of the grid.

5.2 Result

Results obtained from parallel version were used to

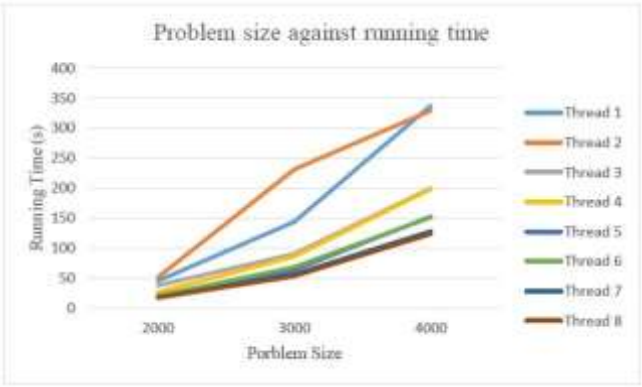

Figure 3. The speedup gained for each thread when we increases the problem

calculate the gained speedup when utilizing the maximum number of threads as depicted in Table 1. Figure 5 shows a graph obtained by plotting the number of threads and gained speedup on x and y axis, respectively, in an 8-neighbour scenario. Each line represents the result corresponding to a different grid size used in our experiments. By in- creasing the number of threads, a considerable increase in the gained speedup can be observed. However, with a certain number of threads, gained speedup will reach a saturation point; at that point, increasing the number of threads will not make any difference. Note that the result of one thread shows an exception in gained speedup compared with two threads, because there is no scheduling overhead of parallelism between two threads.

The next graph is obtained by plotting problem size (N) on x-axis and gained speedup on y-axis when we have 8 neighbours. As shown in Figure 6, each line represents different number of threads used in different runs, ranging from minimum (1) to maximum (8). The graph shows clearly that greater the number of threads, the more we can observe the effect of parallelism. We can also see that gained speedup is maximum in case of maximum number of processors (8) and smallest problem size (2000*2000), because of more number of threads used for a small search space.

In the third graph, we have plotted problem size versus running time, where problem size (N) is on x-axis and running time (T) is on y-axis, when we have 8 neighbours. As shown in Figure 7, each line represents the result obtained by using a different number of threads. The graph shows that there is a significant decrease in running time because of parallelism, particularly in case of the biggest problem size and maximum number of processors (8).

As mentioned previously, our algorithm uses various number of threads to find the shortest path, based on location of global start. Possible number of threads are 3, 5 or 8 in case of corner cell, border cell or middle cell, respectively. Figure 8 shows the result of finding the shortest path from different global start cells to the same goal cell, where x-axis represents gained speedup, y-axis represents grid size, and each line represents one of the three possible cases using different number of threads. As is clear from the graph, speedup is maximum when the maximum number of threads, i.e. 8, are used for searching the shortest path.

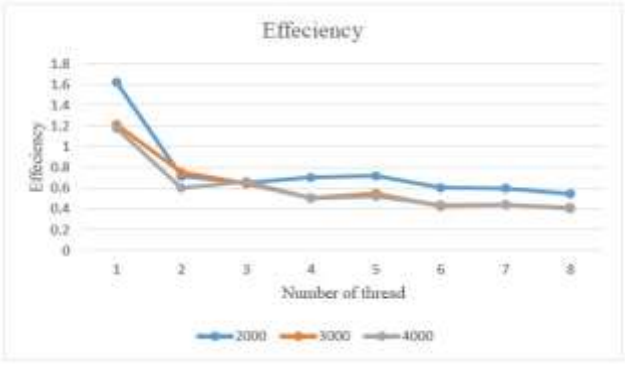

Figure 4. Efficiency gained number of threads

Also, speedup is maximum in case of the smallest grid size because of a smaller search space as compared to a bigger grid size.

As we have observed previously, increasing the number of threads results in an increase of the gained speedup. However, when we plotted efficiency against the number of threads, we could observe in Figure 9 that efficiency decreases with an increase in the number of threads. Efficiency can have a value between 0 and 1; in our case, it is the

Table 2. Amdahl's Speedup

Size of grid	Sequential Fraction (F)	Amdahl's Speedup	Actual
2000*2000	0.120	4.348	4.351
3000*3000	0.204	3.295	3.294
4000*4000	0.212	3.221	3.218

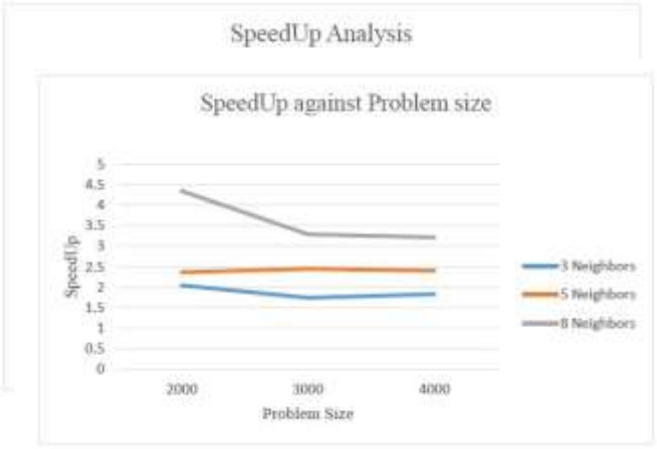

Figure 6. Speedup gained problem size in different cases

maximum in case of one thread because of no overhead caused by parallelism, and keeps decreasing with subsequent increase of processors. This can also be justified by the fact that speedup and efficiency are inversely proportional to each other.

From the results shown above, we concluded that parallelizing the A* decreases the overall execution time of the algorithm.

5.3 Analysis

To analyse the obtained result, first, we applied Amdahl's law given in Equation 5 where F represents sequential fraction and K is the number of processors. F was calculated from the Experimentally Determined Sequential Fraction (EDSF) formula, given in equation 6. Using Amdahl's law, when we are utilizing the maximum number of threads, we achieved a speedup as shown

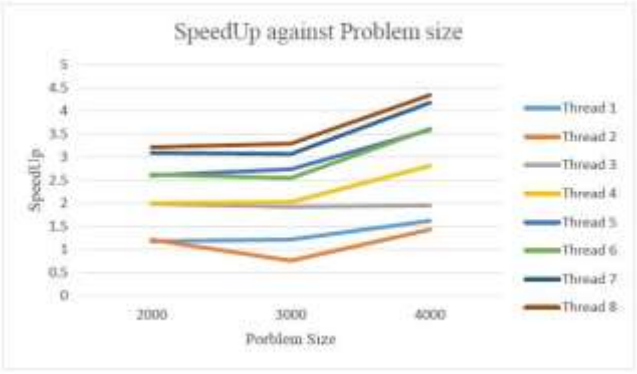

Figure 5. The execution time for each thread when we increases the problem size

in Table 2 and Figure 10. This proves that our proposed algorithm enhanced the gained speedup considerably.

$$SpeedUp(N,K) = \frac{1}{F + \frac{1-F}{K}} \quad (5)$$

$$F = \frac{K*T(N,K) - T(N,1)}{K*T(N,1) - T(N,1)} \quad (6)$$

Secondly, in term of scalability, we started with problem size N in the sequential version that gives more than 60 second as execution time which is a comparable start. Then we used this problem size also in executing the parallel version. After that, we increased the problem size till getting out of memory exception.

Our program scalability can be calculated using the formula 7. As a result, the scalability when we have different number of neighbors is shown in Table 3 and Figure 11.

$$Scalability = \frac{\max imumproblemsize(N)}{\min imumproblemsize(N)} \quad (7)$$

Table 3. Scalability

Number of Neighbor	scalability
3	1.5625
5	2.0408
8	4

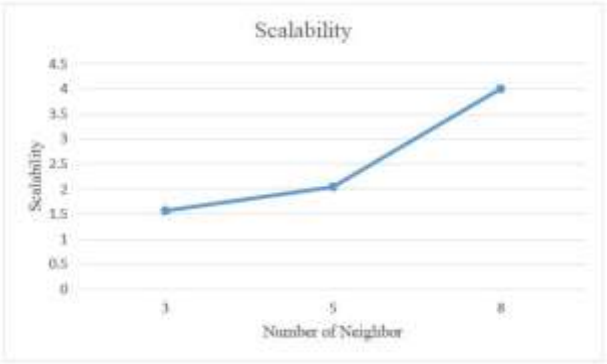

Figure 8. Scalability

6. Conclusion

We have contributed by implementing parallel version of A* for finding the shortest path in a grid map. Based on our results, we concluded that parallelizing the A* decreases the overall execution time and increases the gained speedup of the algorithm. Therefore, a considerable improvement in performance of A* was observed. With the advent of recent technological developments users are getting their hands on more powerful hardware, thus this type of parallelism will be very beneficial for people dealing with path finding search domains.

Acknowledgements

The authors would like to extend their sincere appreciation to the Deanship of Scientific Research at King Saud University for its funding this Research group NO (RG 1435-077). In addition, all authors have contributed equally to this paper.

References

1. P. Pacheco, An Introduction to Parallel Programming, 1st Edition, Massachusetts, USA, Elsevier, Morgan Kaufmann, 2011.
2. D. B. Skillicorn, Foundations of Parallel Programming, 6th Edition, New York, USA, Cambridge University Press, 2005.
3. W.-m. Hwu and D. Kirk, ”Programming massively parallel processors,” 3rd Edition, MA, USA, Elsevier, Morgan Kaufmann 2016.
4. M. Kilinarslan, ”Implementation of a path finding algorithm for the navigation of visually impaired people,” in MS Thesis in Computer Engineering, Atilim University, Ankara,Turkey, pp.68-73, 2007.
5. Zhang, Liang, et al. ”Global path planning for mobile robot based on A* algorithm and genetic algorithm,” Robotics and Biomimetics (ROBIO), 2012 IEEE International Conference on. IEEE, California, USA, pp. 1795-1799, 2012.
6. C.Zeng,Q.Zhang,X.Wei, “GA-based global path planning for mobile robot employing A* algorithm,” Journal of Computers, pp. 470474, 2012.
7. N. H. Barnouti, S. S. M. Al-Dabbagh, and M. A. S. Naser, ”Pathfinding in Strategy Games and Maze Solving Using A Search Algorithm,” Journal of Computer and Communications, p. 15, 2016.
8. L. H. O. Rios and L. Chaimowicz,”PNBA*: A Parallel Bidirectional Heuristic Search Algorithm,” Proceedings of the XXXI Congressa da Sociedade Brasileira de Computaao (CSBC), Brazil, Brasilia, 2011.
9. S. Brand and R. Bidarra, “Multi-Core scalable and efficient pathfinding with Parallel Ripple Search,” Computer. Animation. Virtual Worlds, pp. 7385, 2012.
10. B. A. Mahafzah,”Performance evaluation of parallel multithreaded A* heuristic search algorithm,” Journal of Information Science, p.363375, 2014.
11. I. Rafia, ”A* Algorithm for Multicore Graphics Processors,”, M.S. thesis, Department of Computer Science and Engineering Division of Computer Engineering, CHALMERS UNIVERSITY OF TECHNOLOGY, Gteborg, sweden, 2010.
12. M. Phillips, M. Likhachev, and S. Koenig,”PA* SE: Parallel A* for Slow Expansions,” in ICAPS, New Hampshire, USA, pp. 208216, 2014.