Parallel Real-Time Path planning using RRA* algorithm

Zhiwei Kang
BME
Duke University

Steven Hua
MEMS
Duke university

Abstract

In this paper, we introduced a novel real-time path-planning algorithms RRA* with two parallel processing method to solve a robot evacuation problem and to achieve reasonable speed up. The robot evacuation problem in this paper describes a simulation scenario where all the robots are required to evacuate from the simulation domain within a set time. To achieve reasonable speedup running this simulation, two parallelism method domain decomposition and robot supervision were implemented for two versions of the simulation. Performance and scalability results are presented for Stampede, a petascale supercomputer. The parallelism method respectively achieved maximum around 50 and six times speed up running on up to 256 tasks on the Stampede supercomputer. Also, good weak scaling is achieved for both the parallelism method.

KEYWORDS

Path planning, real time, parallelism, MPI, RRA* algorithm, domain decomposition, robot supervision

1 Introduction

Path-planning performs the fundamental function of the autonomous mobile robots to help the robot to find the closest path between two points while avoiding the obstacles in the ambient environment and even collisions between robots. It is one of the major challenges in improving quality of surgical interventions in surgical robotics system as incorrect path planning may cause unnecessary damage to patients and lower the efficiency using the robotics system［1］

The basic setting of path planning is it requires the robot to have a map of the environment which contains the obstacles information, and understand the locations of the terminal point and itself in the map. In addition, all the robots are assumed to have the map, be able to position itself in the map, and be aware of the temporary surrounding environment before making the path-planning decision.

There are a few basic but effective algorithms that can achieve path-planning ranging from Dijkstra, to A*, and RRT. The A* search algorithm is one of the most popular path-planning algorithms that is widely used in many games and web-based grid maps. The basic idea of A* algorithm is at every time step the algorithms would consider a node according to both the distance from the destination and the distance from the starting point [2]. Another popular path-planning algorithm-rapidly exploring random tree (RRT), designed to achieve path planning in the nonconvex, high-dimensional spaces. However, as pointed out by Qu[3], the path planning algorithms are likely to be computationally expensive in complex environments. Therefore, there should be a demand of applying parallel computing in robot path-planning and many resources are already put into the research of novel methods to realize path-planning with parallel processing for speedup [4].

In this paper, based on the existing path-planning algorithms, we first introduced a new path-planning algorithms RRA* that provides collisions and obstacles avoidance function. Then we designed a game where robots are required to evacuate from the game domain containing obstacles within a set time. Here, we designed two versions of the game. One of which accepts robot collision and the other not. The parallelism of the path-planning is achieved only using MPI. For each game version, a designed parallelization method is introduced. To examine our RRA* algorithm and parallelization method, strong and weak scaling was carried out, and a validation section to demonstrate that the same results as the serial version of the code. In addition, we examined the effect of the complexity of the map, the probability value in out RRA* algorithm, and robot density on the game result. Finally, in the discussion, we discussed the pros and cons of the two parallelism method and possible solutions to solve the performance bottlenecks.

2 Method

In this section, the simulation scenario will be first introduced to give an overview, followed by two of the most popular path-planning algorithms and the RRA* algorithm introductions, parallelism methods and the hardware in the end. Two different versions of the game will also be clarified.

2.1 Simulation settings

The simulation used in this paper for performance and correctness testing is an evacuation problem that a certain number of robots need to evacuate from the game domain within a set time. The size of the simulation domain used in this paper except for in the weak scaling testing is 1000*1000. The robot number in the simulations except for weak scaling is 1000. Every single robot on the map can only occupy one grid and is represented by value one. Each obstacle would be represented by value minus one.

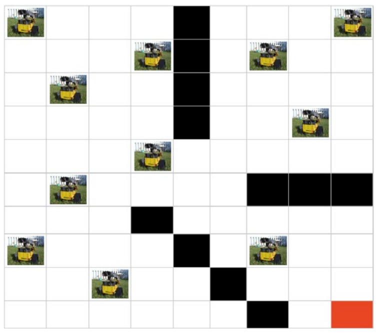


Figure 1: Sketch of the simulation domain. The robots are randomly distributed on the map before the game starts. The black grids in figure 1 illustrate the obstacles which robots cannot pass. The red grid here represents the exit point

We wrote three maps to run this simulation respectively Map01, Map02, and Map03. Map01 is the default map we used in this paper. Map01 contains four sets of obstacles, while Map02 has 6 sets, and Map03 has 8 sets. The size of all the three maps is 1000*1000. Then a random number generation process would generate initialization locations for each robot participating in the simulation. The random number seed in the code is different every time we run the simulation.

As the game starts, each robot will start moving towards the exit point. At each time step, the robot can move to all 8 surrounding cells. Then the robot will determine which direction to go using the RRA* algorithm. At each time step, a single robot can only move up to one step. By the time a robot reaches the exit point, this robot will be removed from the game. The simulation will stop after running a certain number of time steps and calculate the number of robots successfully reach the exit point.

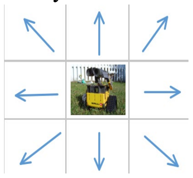


Figure 2: 8 possible move directions for a single robot at a specific time step

2.1.1 Success criteria

The success criteria of this simulation are in a certain amount of time (1000 steps) if a robot can move from is starting location to the exit point, then this would be considered as one success. Robots reach the exit point at any time step within the designated time interval will be viewed as one success.

2.1.2 Robot collision tolerance

In this paper, the simulation has two versions. The difference between the two versions is whether more than one robot can occupy one same grid on the map. In other words, whether the value of on a single grid can be larger than one or not. The collision version allows more than more robots occupy the same grid at any time step. On the contrary, the no collision version only allows one robot to occupy one grid except for the exit point at any time step. Both versions of the simulation don't allow robots to occupy the grids that are considered as an obstacle.

2.1.3 Simulation Results

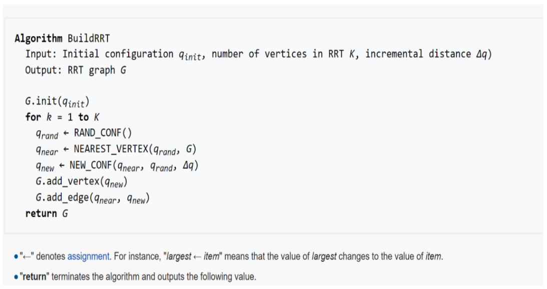
At the end of each time running the simulation, the number of robots successfully move to the exit point in the designated time would be summed together. In addition, this number would be divided by the total number of robots in the simulation results in a percentage to determine how successful this time we run the simulation. The closer this percentage close to one, we would view this run being more successful.

2.2 Algorithm

Currently, there are many mature path planning algorithms that work well in different situations. The most popular ones are the RRT (Rapidly-exploring random tree) and A* search algorithm. They are both effective but have weaknesses respectively as well.

2.2.1 RRT

Compared to traditional path planning algorithms, RRT can effectively solve the path planning problem of high-dimensional space and complex constraints. RRT is an efficient planning method in multidimensional space. It uses an initial point as the root node to generate a random expansion tree by randomly sampling the leaf nodes. When the leaf nodes in the random tree contain the target points or enter the target area, they can find a random tree shown in Figure 3 below).



```
Algorithm BuildRRT
  Input: Initial configuration $q_{init}$, number of vertices in RRT $K$, incremental distance $\Delta q$)
  Output: RRT graph $G$

  $G$.init($q_{init}$)
  for $k$ = 1 to $K$
    $q_{rand}$ ← RAND_CONF()
    $q_{near}$ ← NEAREST_VERTEX($q_{rand}$, $G$)
    $q_{new}$ ← NEW_CONF($q_{near}$, $q_{rand}$, $\Delta q$)
    $G$.add_vertex($q_{new}$)
    $G$.add_edge($q_{near}$, $q_{new}$)
  return $G$
```

- "←" denotes assignment. For instance, "*largest* ← *item*" means that the value of *largest* changes to the value of *item*.
- "**return**" terminates the algorithm and outputs the following value.

Figure 3: RRT algorithm pseudocode

("Rapidly-exploring random tree", En.wikipedia.org, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Rapidly-exploring_random_tree. [Accessed: 18- Dec- 2018].)

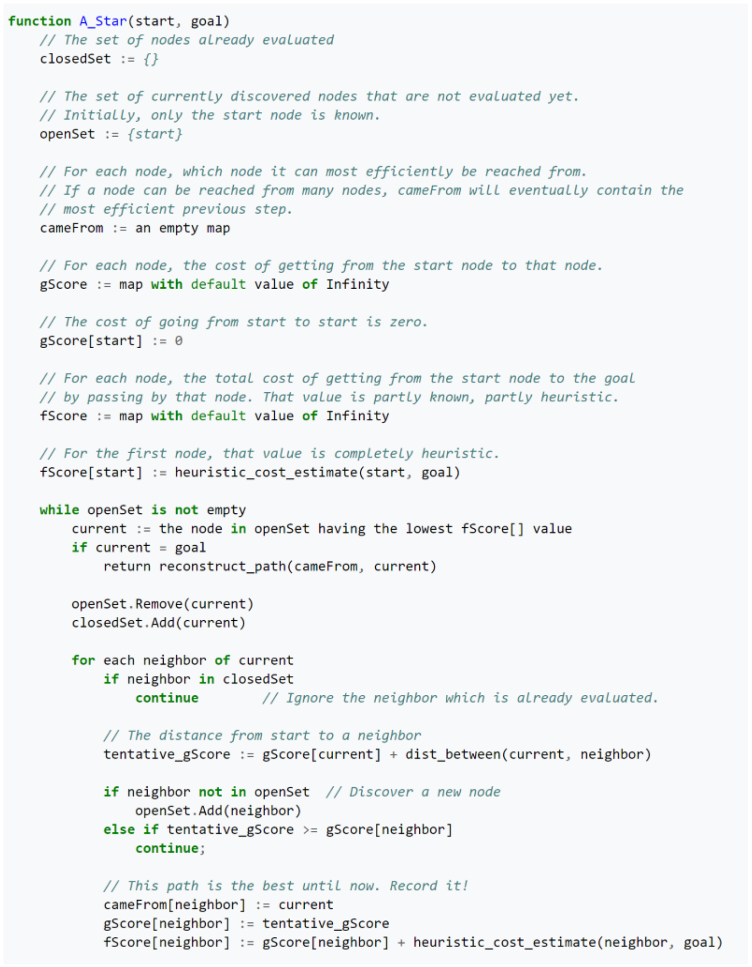
In this pseudocode, first, an initial point is generated as the root point. Then all the vertices are traversed to add new points with the same rules. The first step to add a new point is to find the closest point towards the target. Next, a random number is generated to compare with a preset probability, whose result will decide either the closest point or the random point to be added. The whole process will end when the target is found or is close enough.

The shortcomings are also evident. Since RRT algorithm is a purely random search algorithm that is not sensitive to the environment type, the convergence speed of the algorithm is slow and the efficiency is greatly reduced once space contains a large number of obstacles or narrow channel constraints. If the channel is narrow enough, it is hard to find a path with the RRT algorithm since the possibility for the path to touch the channel is very low.

2.2.2 A* Search Algorithm

Another useful path planning algorithm is A* Search Algorithm. As one of the heuristic search algorithms, this algorithm uses multiple nodes on the graphics plane to find the path with the lowest cost. It also does not need to pre-process the whole graph.

The basic idea of the algorithm is to calculate the cost of selecting one grid on the map, and estimate the cost required to extend from the selected grid all the way to the goal to decide which way to extend (Figure 4). So, in each step, it has to calculate the total cost of all eight neighbors of the node. This algorithm ends when the path it chooses is able to reach the goal or if there are no paths eligible to be extended. The implementation is conducted with the help of the priority queue.



```
function A_Star(start, goal)
    // The set of nodes already evaluated
    closedSet := {}

    // The set of currently discovered nodes that are not evaluated yet.
    // Initially, only the start node is known.
    openSet := {start}

    // For each node, which node it can most efficiently be reached from.
    // If a node can be reached from many nodes, cameFrom will eventually contain the
    // most efficient previous step.
    cameFrom := an empty map

    // For each node, the cost of getting from the start node to that node.
    gScore := map with default value of Infinity

    // The cost of going from start to start is zero.
    gScore[start] := 0

    // For each node, the total cost of getting from the start node to the goal
    // by passing by that node. That value is partly known, partly heuristic.
    fScore := map with default value of Infinity

    // For the first node, that value is completely heuristic.
    fScore[start] := heuristic_cost_estimate(start, goal)

    while openSet is not empty
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        closedSet.Add(current)

        for each neighbor of current
            if neighbor in closedSet
                continue        // Ignore the neighbor which is already evaluated.

            // The distance from start to a neighbor
            tentative_gScore := gScore[current] + dist_between(current, neighbor)

            if neighbor not in openSet  // Discover a new node
                openSet.Add(neighbor)
            else if tentative_gScore >= gScore[neighbor]
                continue;

            // This path is the best until now. Record it!
            cameFrom[neighbor] := current
            gScore[neighbor] := tentative_gScore
            fScore[neighbor] := gScore[neighbor] + heuristic_cost_estimate(neighbor, goal)
```

Figure 4: A* Search algorithm pseudocode
(A* search algorithm", En.wikipedia.org, 2018. [Online]. Available: https://en.wikipedia.org/wiki/A*_search_algorithm. [Accessed: 18- Dec- 2018].)

Since the algorithm does not need to pre-process the whole space, it is replaced by other algorithms in some cases. But the efficiency cannot be denied. One of the shortcomings is that this algorithm cannot guarantee optimal search path. Plus, it also needs a large number of calculations, especially a widely traverse conduction is used.

2.2.3 RRA* Algorithm

Based on the above content, both of the algorithms have strengths and weaknesses, and are not suitable in our scenario since we need to find the optimal path in real time efficiently without knowing the whole environment. The RRT algorithm is efficient enough but it has to be conducted with the knowledge of the whole environment. The A* Search Algorithm can be conducted without the information of the whole map but it has a lot of redundant calculation and has to store all the path it has chosen. Thus, a new algorithm is introduced here, which combined both the strengths of RRT algorithm and the A* Search Algorithm. The RRA* algorithm uses the search concept of the A* Search Algorithm while adopting the extension concept of the RRT algorithm.

An example scenario is shown in Figure 5, and the pseudocode is shown in Figure 6. The robot position is marked with the red circle and the obstacles are marked with the blue rectangles. The target is highlighted with the green circle. As we can see, the robot has six out of eight neighbors, marked with the yellow faces means it can go. The first thing to do is to store all the pair information of the available neighbors' positions and the cost to go to that place. The neighbor with the smallest cost is the best choice and is marked with the purple face in figure 5. And a random number will then be generated to compare with the given probability (e.g. 0.5). If the number is smaller than the probability, the best choice will be chosen; Otherwise, a random direction (including the best choice) will be chosen. The reason why we introduce the probability here instead of always using the best choice is that the latter one could lead to a dead corner. As can be seen in Figure 7, the robot in the left corner is in a deadlock. Since it can only choose the best choice, it will always be stuck in that place.

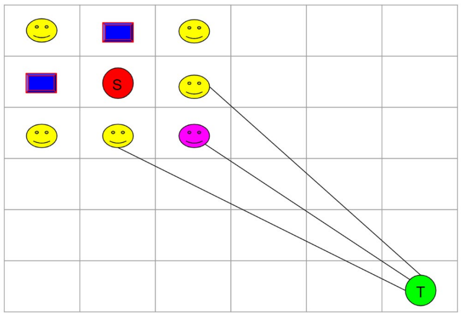
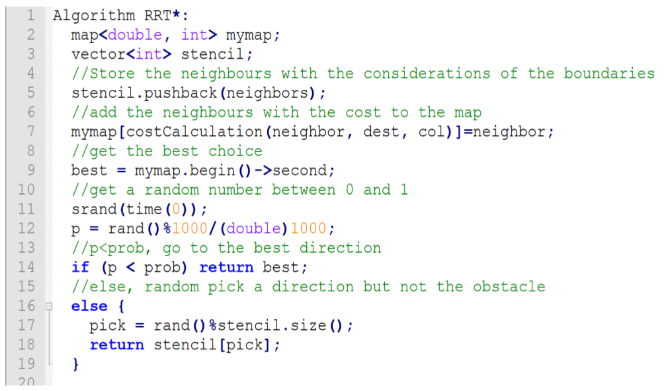


Figure 5: RRA* algorithm example



```
Algorithm RRT*:
  map<double, int> mymap;
  vector<int> stencil;
  //Store the neighbours with the considerations of the boundaries
  stencil.pushback(neighbors);
  //add the neighbours with the cost to the map
  mymap[costCalculation(neighbor, dest, col)]=neighbor;
  //get the best choice
  best = mymap.begin()->second;
  //get a random number between 0 and 1
  srand(time(0));
  p = rand()%1000/(double)1000;
  //p<prob, go to the best direction
  if (p < prob) return best;
  //else, random pick a direction but not the obstacle
  else {
    pick = rand()%stencil.size();
    return stencil[pick];
  }
```

Figure 6: RRA* algorithm pseudocode

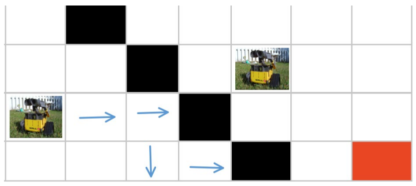


Figure 7: The result of keep applying the best choice in path-planning

This algorithm is not limited to the environmental constraints. The path is determined with the only knowledge of the neighbors. And it can be implemented in the real-time situation. As it is characterized by the strengths of both the RRT algorithm and the A* Search Algorithm, it is also efficient and heuristic.

2.2.4 RRA* Algorithm with non-collision rules

The section above talked about the situation where the collision is allowed, which means two or more robots can be placed on the same grid. In real life, it may not be the case. Figure 8 shows this situation. S1 and S2 both want to go to the purple face. With the non-collision rules, only one of them is able to go to that place.

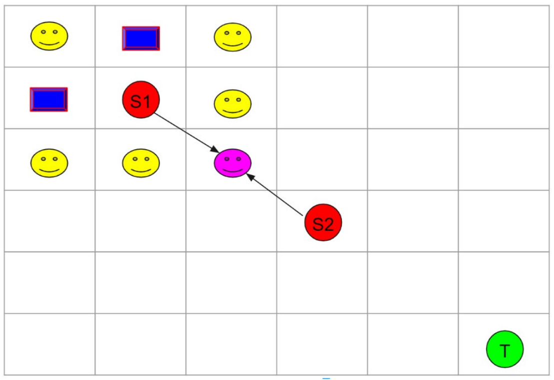
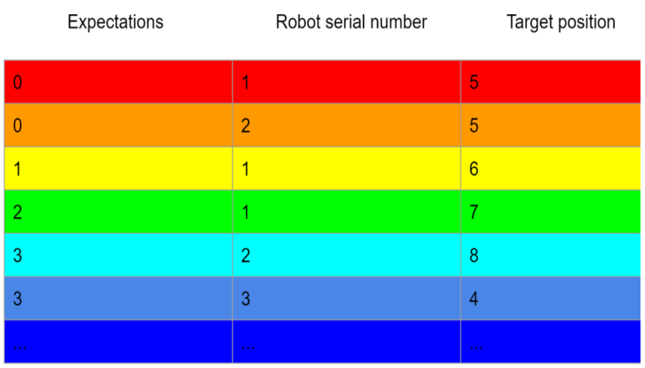


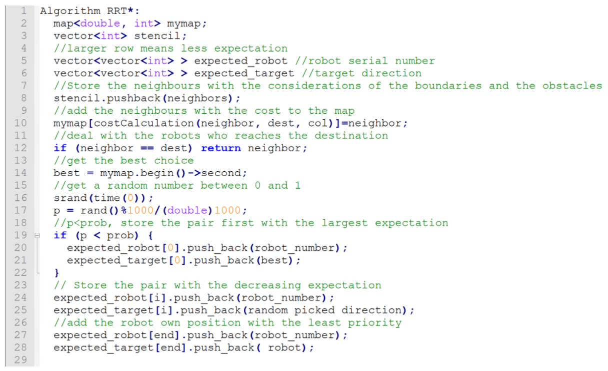
Figure 8: Robots collision situation

In order to meet the requirements of the non-collision policy. An updated RRA* algorithm is developed. To avoid the duplicated implementations of the same robots, a priority queue is used. The basic idea is to store the pair of robot serial number and its target with the expectation (Figure 9). The key of the priority queue is the expectations. The smaller the number is, it indicates higher priority. In other words, the more the robot wants to go to that point. The data matched to the key "expectation" is the pair of robot serial number and the target positions. Thus, there will be up to 9 x Total_robot_numbers lines in the priority queue. Then, at each stage, the top line will be popped out. The robot serial number and the target position pair will be taken out and store in the memory. Once the robot serial number or the target position has been seen before, the storing step will be skipped. In this way, the collision avoidance can be implemented well. In our pseudocode, two 2-D arrays (Figure 10) has been used to implement the priority since it is more convenient to send and receive data in the array type with MPI. As we can see, the expectation is realized using the row number. The smaller the row is in the array, the larger the expectation is. And we use the probability to decide whether to store the best choice with the highest priority. Therefore, the essence of the RRT* algorithm has not been changed. One more difference from the original RRT* algorithm is that the original position of the robot is also pushed into the priority queue since the robot may have to stay at the same place in some cases.



Expectations	Robot serial number	Target position
0	1	5
0	2	5
1	1	6
2	1	7
3	2	8
3	3	4
...	...	...

Figure 9: Priority queue Implementation



```
Algorithm RRT*:
  map<double, int> mymap;
  vector<int> stencil;
  //larger row means less expectation
  vector<vector<int> > expected_robot //robot serial number
  vector<vector<int> > expected_target //target direction
  //Store the neighbours with the considerations of the boundaries and the obstacles
  stencil.pushback(neighbors);
  //add the neighbours with the cost to the map
  mymap[costCalculation(neighbor, dest, col)]=neighbor;
  //deal with the robots who reaches the destination
  if (neighbor == dest) return neighbor;
  //get the best choice
  best = mymap.begin()->second;
  //get a random number between 0 and 1
  srand(time(0));
  p = rand()%1000/(double)1000;
  //p<prob, store the pair first with the largest expectation
  if (p < prob) {
    expected_robot[0].push_back(robot_number);
    expected_target[0].push_back(best);
  }
  // Store the pair with the decreasing expectation
  expected_robot[i].push_back(robot_number);
  expected_target[i].push_back(random picked direction);
  //add the robot own position with the least priority
  expected_robot[end].push_back(robot_number);
  expected_target[end].push_back( robot);
```

Figure 10: RRA* algorithm with non-collision rules pseudocode

2.3 Parallel processing using MPI

This paper highlights the parallelism of the code. As we have discussed above, there are two simulation scenarios, collision , and non-collision. The serial code of both versions will be paralleled. We used MPI to implement the parallelism since MPI is a very powerful tool and easy to use for communications. More resources can be used for SIMD structures.

2.3.1 Parallel processing with collision version

Considering the simulations has two versions, we designed one parallelism method for each version. The first is the domain decomposition and the second is the robots-supervision.

2.3.1.1 Domain decomposition

The whole map is 1000x1000. Here, we use the domain decomposition in the row direction (Figure 11). The rows are divided evenly for each task. Thus, each task only takes in charge of its own domain which means each task only deals with the robots in its domain. In this way, it is necessary to conduct the communications between the tasks since there will be robots move from one task to a nearby task. Figure 12 explicitly explains the communications. Each task has two buffer layers on the two sides, which are marked with green color. The buffers are used to store the robots that move from the domains of the neighbor tasks. By the end of each time step, the robots in the buffers would be sent to its neighbor rank. Buffer 1 will be sent to the previous rank (except for rank 0) and buffer 2 will be sent to the next rank (except for rank size-1). Each rank will update their top (except for rank 0) and the bottom layer (except for rank size-1) after receiving the data. The robot s in the buffers will be cleared before starting a new time step.

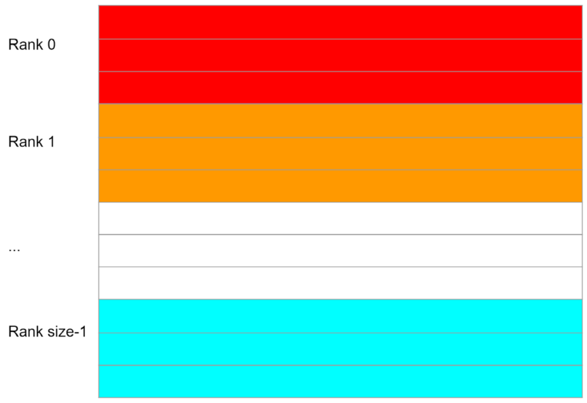


Figure 11: Domain Decomposition

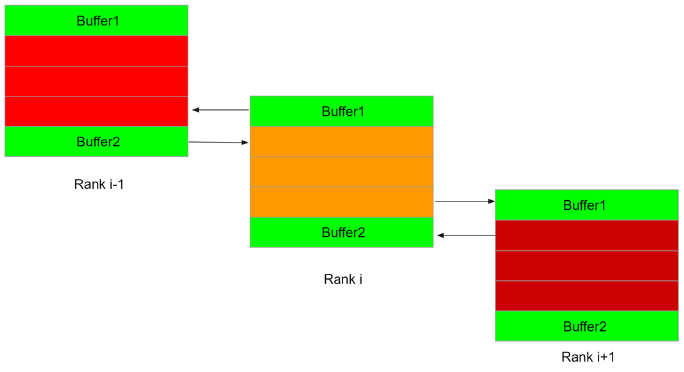


Figure 12: Communication between ranks

2.3.1.2 Robot Supervision

This method did a better job in load-balancing since each task is assigned the nearly same number of robots at the beginning of the simulation (Figure 13). Thus, the only communication will be the reduction operation at the end of the simulation, for calculating the total number of robots who reach the destination. Since robot supervision method doesn't fit simulation tolerates collision as it could only achieve embarrassingly parallelism, in this paper, the robot supervision method was only used for no collision simulation.

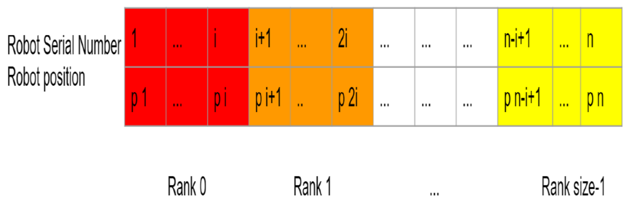


Figure 13: Robot Supervision

2.3.2 Parallel processing with non-collision version

Added the non-collision rules, the situation became more complicated to parallel using domain decomposition method. Thus, for no collision version, the robot supervision parallel method was implemented. This parallelism is illustrated in Figure 14. At each step, the master rank (rank 0 in this paper) would distribute all the robots, which did not exist the simulation yet, to each task with robot serial number and the position information of the robots. This process is the same as what shows in Figure 13. Each task will then run the RRA* algorithm for each robot assigned to them and generate a priority queue. Those robots who have reached the destination will not be included in the priority queue. However, the number of robots that successfully reach the destination would be stored. At each step, the priority queue from each task will be gathered to rank 0. And rank 0 will then process the combined priority queue and get the updated pair of information (robot number -new location). Then, these robots' serial number and their positions would be distributed back to each task. At the end of the game, the number of robots who reach the destination would be sent to rank 0 through reduction operation.

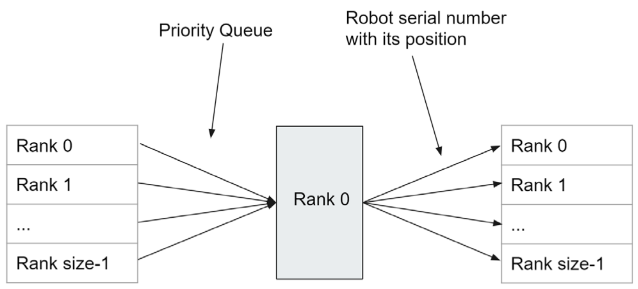


Figure 14: Robot Supervision with non-collision rule parallelism structure

2.4 Supercomputer Architecture

Our simulation in this paper was run on the Stampede petascale supercomputer UT Austin.

2.4.1 Stampede

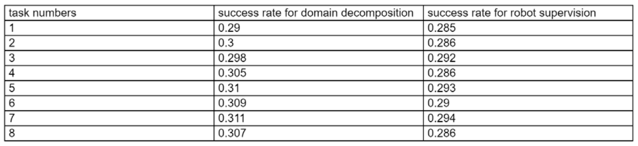
The Stampede system is considered as one of the most powerful open source supercomputer in the US. It is built out of 2 8-core Xeon E5 processors, 1 61-core Xeon Phi coprocessor, and sums up to 6400 Nodes. A single node on Stampede can process 32GB main memory and the whole Stampede system has up to 205TB aggregate memory. The interconnection of the Stampede system is made of an FDR InfiniBand network of Mellanox switches, which can achieve data transmission speed at 56 Gb/s.
The Stampede system was upgraded in 2016 to equip with the latest Intel Xeon Phi many-core, x86 architecture, known as Knights Landing. The upgraded Stampede is ranked #116 in the Top500 list by June 2016 [stampede].

3. Results

In this section, we first present a validation test was conducted to check if the result received from the serial version is the same as the paralleled version. Then we present the result of running our two versions simulations for a single task on the Stampede. Then we would run the serial code with different probabilities used in the RRA* algorithm. Next, we will present the performance values for running our two versions of simulation on more than one node on Stampede. Strong and weak scaling was conducted to examine our scaling properties and see if reasonable speedup was achieved. Also, we did some experiments running the simulations on maps with different complexity. A more complex map indicates that it contains more obstacles in it. The probability and the density of the robots are also considered as parameters may affect the experiment result.

3.1 Validation

The first thing we need to do is to validate that our parallelism does not change the results. The testing use the following settings. The map size is 1000x1000. The robot number is 1000. And we'll use 1000 time-steps. In order to ensure the initial places generated for all robots are the same, we removed the srand() function to give a seed for the random number generation process. The results are shown for two versions, domain decomposition and robot supervision. The results of the serial code (task numbers = 1) and the parallel code (task numbers >1) are shown in the table 1.



task numbers	success rate for domain decomposition	success rate for robot supervision
1	0.29	0.285
2	0.3	0.286
3	0.298	0.292
4	0.305	0.286
5	0.31	0.293
6	0.309	0.29
7	0.311	0.294
8	0.307	0.286

Table 1: Validation results table

Since we use MPI for the parallelism, the rand() function may defer by the number of the tasks used. But we think the results should be in a small range of differences. The Table 2 shows the summary of the results. In the table, we can see that the results have very small variances. Thus, we conclude that parallelism will not change the results of the code

Groups	Count	Sum	Average	Variance
success rate for domain decomposition	8	2.43	0.30375	5.25E-05
success rate for robot supervision	8	2.312	0.289	1.34E-05

Table 2: Summary of the results

3.2 Single task performance

The serial code of the two simulation versions was run on a single Stampede computer node using one task. The map used here is the Map 01. The robot number is set at 1000. The total time step is 1000 step. The probability in the RRA* algorithm is set as 0.6. The random number seeds using in the robot initialization step are different every time running the simulation.

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
No-collision run time	45.634350	47.687971	46.262225	51.733932	48.823151
No-collision successful rate	0.380000	0.39	0.380000	0.248	0.346000
collision run time	46.227816	44.321666	43.402446	45.669420	43.885222
collision successful rate	0.254000	0.379000	0.412000	0.185000	0.361000

Table 3: Runtime and success rate running the serial code on a single task on Stampede

3.3 Multi-task performance

3.3.1 Performance with domain decomposition parallelism

We run the collision version simulation with domain decomposition parallelism method using multitasks on the Stampede computer to do the strong and weak scaling experiment.

Strong scaling: For the strong scaling experiment, we run the code up to 256 tasks. The map used for strong scaling is Map01. The robot number is 1000. Total time step is 1000 step, and the probability in the RRA* algorithm is set as 0.6.

Weak scaling: To achieve weak scaling, we kept the region each task is responsible for the same while increasing the whole size of the simulation domain. Each task took charge of a 100*1000 region. Also, the robot density was kept the same – (100 robot)/(100,1000 grid) as more tasks were put in to the simulation. The results of strong and weak scaling are shown below in figure 15.

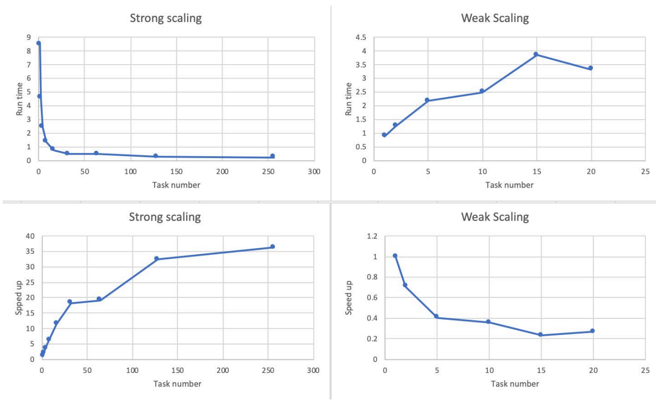


Figure 15 Strong and weak scaling experiment for domain decomposition parallelism method.

3.3.2 Performance with robot supervision parallelism

We used the robot supervision parallelism method on no-collision version simulation using more than one tasks on the Stampede computer to do the strong and weak scaling experiment.

Strong scaling: For the strong scaling experiment, up to 256 tasks were used on multiple nodes on Stampede. The map used for strong scaling is Map01. The robot number is 1000. Total time step is 1000 step, and the probability in the RRA* algorithm is set as 0.6.

Weak scaling: For robot supervision parallelism method, the number of robots each task takes control of are the same no matter how many tasks run the code. In this case, each task would control 100 robots. Total time step is 1000 step, and the probability in the RRA* algorithm is set as 0.6.

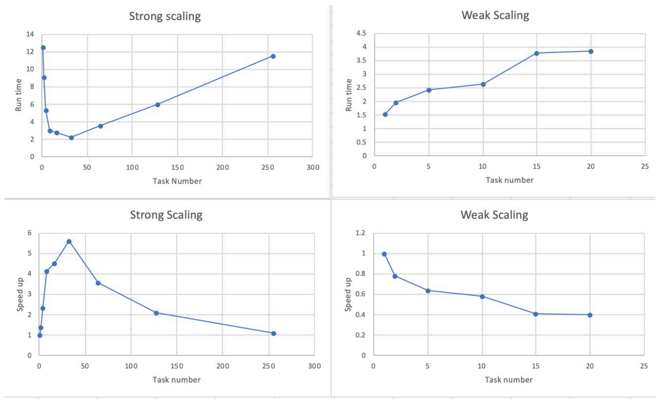


Figure 16 Strong and weak scaling experiment for robot supervision parallelism method.

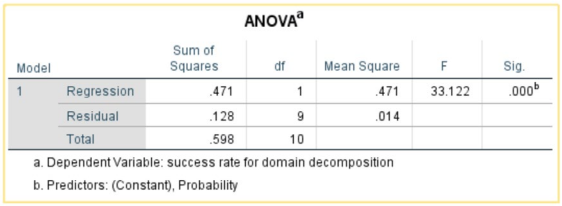
3.4 Performance with different probability settings in the RRA* algorithm

An important parameter in this paper is the probability in the RRA* algorithm.Iit is essential for us to dig into the influence of

the probability for our own algorithm—RRA* and try to find the best choice.
The environment we use to do the experiment is a 1000x1000 map and 1000 robots are generated randomly on the map. We use the least complex map-Map01 and conducted 1000 time-steps. The probability is classified into 11 groups (0, 0.1, 0.2, … 1). Two versions of results will be discussed: domain decomposition where the collisions are accepted and robot supervision where the non-collision rules are applied.

3.4.1 Domain Decomposition

From the table 4 we can see that the probability is significantly influencing the success rate (Significance value = 0.000 while α=0.05). And combined with the Figure 17, we found that the higher possibility can lead to higher success rate. But we can also find that the success rate is not the highest when the probability is 1. The best probability is around 0.9.



ANOVA[a]

Model		Sum of Squares	df	Mean Square	F	Sig.
1	Regression	.471	1	.471	33.122	.000[b]
	Residual	.128	9	.014		
	Total	.598	10			

a. Dependent Variable: success rate for domain decomposition

b. Predictors: (Constant), Probability

Table 4: One-way Anova for domain decomposition

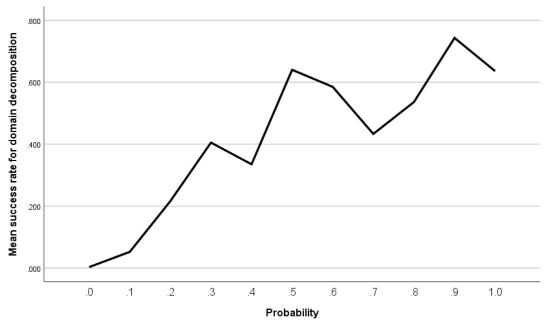
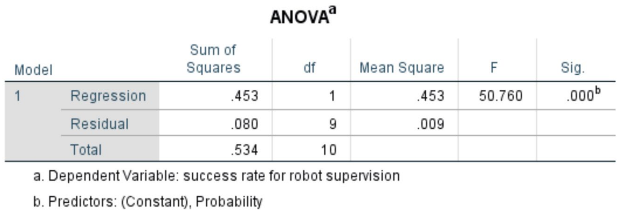


Figure 17: Success rate versus Probability for domain decomposition

3.4.2 Robot Supervision

When applied with the non-collision rules, we can also see that the probability is significantly influencing the success rate (Significance value = 0.000 while α=0.05) from Table 5. And combined with the Figure 18 we can notice that the increasing possibility can lead to higher success rate. But the same turning point occurs when the probability is around 0.8 .



ANOVA[a]

Model		Sum of Squares	df	Mean Square	F	Sig.
1	Regression	.453	1	.453	50.760	.000[b]
	Residual	.080	9	.009		
	Total	.534	10			

a. Dependent Variable: success rate for robot supervision

b. Predictors: (Constant), Probability

Table 5: One-way Anova for robot supervision

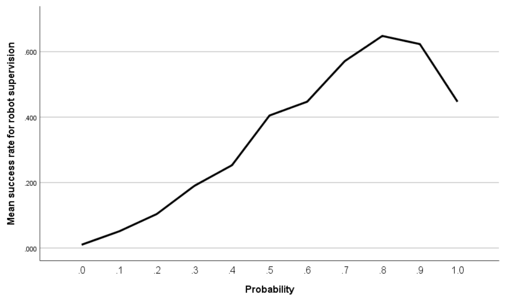


Figure 18: Success rate versus Probability for robot supervision

3.5 Performance with different simulation map complexity and robot density

In this paper, we designed three different kinds of map complexity using our map generator code. The robot density is also defined into three situations. Since our map is 1000x1000, we generate 100, 500, 1000 robots to satisfy different situations. Thus, the robot density is classified as 0.0001, 0.0005, and 0.001. Plus, as we know the performance may change according to the probability we gave, five different probabilities are defined in the coming test. Thereby, a total of 45 combinations of the parameters are to be tested. And for each combination, five replication experiments will be done. We'll show the results in two versions of collision and non-collision.

3.5.1 Domain Decomposition

The means of all the combinations are used to plot. In the Figure 19, we can see that, the more complex of the map is, the less success rate will be for the robots to reach the destination except for when the probability is 0. But it is noticed that the success rate is nearly 0 for all the cases if the probability is 0. Thus, we can say that the increasing complexity of the map offers a decreasing trend of success rate.

In addition, the trend of the success rate with the increasing density is not obvious, especially when the probability is 0.25 and 0.75.

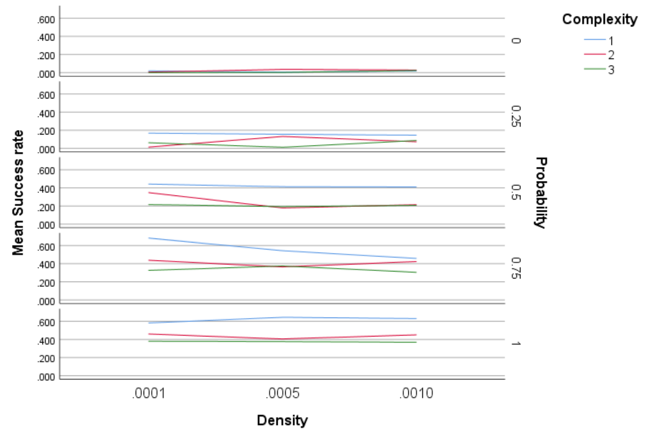
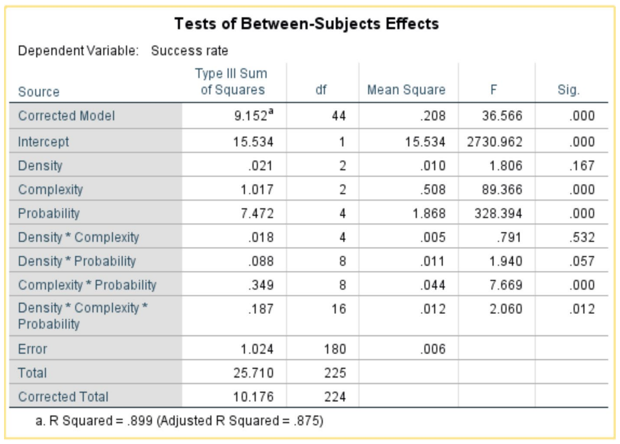


Figure 19: Variables Combinations plot with collision

For further study, a three-way Anova is conducted (Table 4). From the table, we can notice that the success rate is significantly influenced by the probability (Significance value = 0.000 while α=0.05). And it is also significantly influenced by the complexity (Significance value = 0.000 while α=0.05). But the density is not significantly influencing the results (Significance value = 0.167 while α=0.05). These results are matched to what we talked about above. Combined with Figure 20, we can conclude that the increasing complexity of the map leads to the decreasing mean success rate. The density doesn’t significantly influence the success rate when the non-collision rules are not applied.



Tests of Between-Subjects Effects

Dependent Variable: Success rate

Source	Type III Sum of Squares	df	Mean Square	F	Sig.
Corrected Model	9.152[a]	44	.208	36.566	.000
Intercept	15.534	1	15.534	2730.962	.000
Density	.021	2	.010	1.806	.167
Complexity	1.017	2	.508	89.366	.000
Probability	7.472	4	1.868	328.394	.000
Density * Complexity	.018	4	.005	.791	.532
Density * Probability	.088	8	.011	1.940	.057
Complexity * Probability	.349	8	.044	7.669	.000
Density * Complexity * Probability	.187	16	.012	2.060	.012
Error	1.024	180	.006		
Total	25.710	225			
Corrected Total	10.176	224			

a. R Squared = .899 (Adjusted R Squared = .875)

Table 4: Three-way Anova Table without non-collision rules

3.5.2 Robot Supervision

The same way of plotting is used in this part. In the Figure 13, we can see that, the more complex of the map is, the less success rate will be for the robots to reach the destination. This is the same as collision part. Thus, we can say that the increasing complexity of the map offers a decreasing trend for the mean success rate.

In addition, the trend of the success rate with the increasing density is more complicated. It seems that the larger the density is, the higher the success rate is given higher probability. But the success rate will be lower with the increasing density given the lower probability. We’ll discuss more in next section.

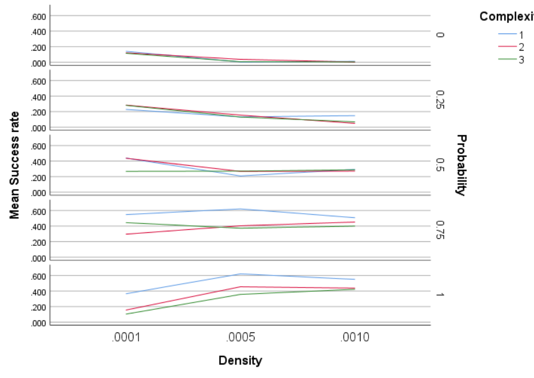
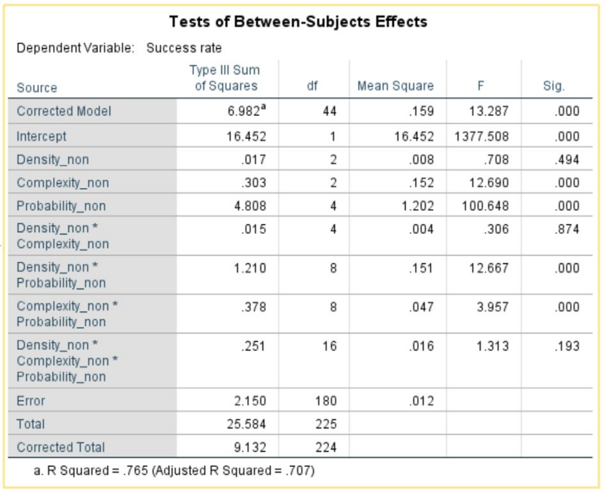


Figure 20: Variables Combinations plot with non-collision

The three-way Anova was also built here (Table 5). From the table, we can also find that the success rate is significantly influenced by the probability (Significance value = 0.000 while α=0.05). And it is also significantly influenced by the complexity (Significance value = 0.000 while α=0.05). But the density does not affect the results with statistical significance (Significance value = 0.494 while α=0.05), which was unexpected. But as we dig into the table, we found that the combined influences of the density and the probability are significant (Significance value = 0.000 while α=0.05). This finding meets what we discussed above. With Figure 20, we can conclude that the increasing complexity of the map leads to the decreasing average success rate. The robot density doesn’t significantly affect the success rate when the non-collision rules are applied. But the density can significantly influence the results when considering the effect of probability together. Higher robot density along with higher probability leads to the higher success rate but results in lower success rate with lower probability.



Tests of Between-Subjects Effects

Dependent Variable: Success rate

Source	Type III Sum of Squares	df	Mean Square	F	Sig.
Corrected Model	6.982[a]	44	.159	13.287	.000
Intercept	16.452	1	16.452	1377.508	.000
Density_non	.017	2	.008	.708	.494
Complexity_non	.303	2	.152	12.690	.000
Probability_non	4.808	4	1.202	100.648	.000
Density_non * Complexity_non	.015	4	.004	.306	.874
Density_non * Probability_non	1.210	8	.151	12.667	.000
Complexity_non * Probability_non	.378	8	.047	3.957	.000
Density_non * Complexity_non * Probability_non	.251	16	.016	1.313	.193
Error	2.150	180	.012		
Total	25.584	225			
Corrected Total	9.132	224			

a. R Squared = .765 (Adjusted R Squared = .707)

Table 5: Three-way Anova Table with non-collision rules

4. Discussion

4.1 Two parallelism method

In this paper, two parallelism method respectively domain decomposition and robot supervision were introduced for parallel processing in two versions of the simulation.

4.1.1 Domain decomposition method: This method received rather good strong scaling and weak scaling result shown in fig. However, one significant weakness of this method is that the computation load on each task would be unbalanced as the simulation progresses. Since most robots would concentrate in the region close to the exit point, the task deal with the region close to the exit point would have rather heavy computation load.

4.1.2 Robot supervision method: As for the robot supervision method, as shown in figure 16, the computation decreases at first as more tasks participated in the simulation. However, roughly when task number larger than 50, the code didn't achieve the speedup as predicted by the Amdahal law. The possible reason is that the total number of robot (1000) is rather small when the task number is more than 50 and the communication time between task become more inevitable. Also, each collective communication method implicitly has an MPI_barrier() in it which should slow down the code because all the task needs to wait until the slowest task finish the job. The outcome of this implicit barrier should become more significant when the task number is larger.

4.2 Parameters influence

There are three parameters discussed in this paper: probability, robot density, and map complexity. The results have been shown above.

4.2.1 Probability in the RRA* algorithm: The probability is a significant influential parameter. One point has to be admitted is that path planning algorithm cannot be a local optimal solution since it may be locked at a dead corner. A good choice of the probability can make the whole process more efficient and optimal. The results show that the probability of 0.9 in domain decomposition condition and 0.8 in robot supervision condition can lead to best performance. But we also see that the probability may lead to other kinds of effects with the power of map complexity and the robot density. For example, the higher robot density along with higher probability leads to the higher success rate but results in lower success rate with lower probability. Thus, the choice of the best probability to be applied should be considered with the real environment. The strength is that the specific environment situation does not need to be provided. Instead, a general evaluation should be enough.

4.2.1 Map complexity and robot density: The map complexity is also proved to be significant. The higher map complexity leads to lower success rate, which matches the real situation. This has already been a big headache in this area. But given enough time steps, our algorithm is confident to get a better performance. The robot density is proved to be non-significant. This result surprises us. But it is good to know that the RRA* algorithm has a large tolerance of the robot density.

For now, the results have shown a greatly practical application of RRA* algorithm. But a lot more experiments need to be done to further validate the results. For example, more subdivisions need to be specified for these parameters. In this paper, only 3 refinements are implemented for the map complexity and the robot density.

4.3 Performance Bottleneck and Future improvements

In this section, performance bottlenecks would be discussed and possible reason for each bottleneck would be provided. Then ideas to further improve or solve the bottleneck would also be presented.

4.3.1 Load balancing problem of domain decomposition method: As discussed above, the domain decomposition method may cause load unbalancing as simulation progresses. Regarding this problem, there are two potential solutions. The first one is "unevenly domain decomposition". For now, our domain decomposition method distributes the whole simulation evenly domain to each rank before the game starts. The size of region each rank takes control of is close to the same. But for unevenly domain decomposition we proposed, the rank far away from the exit point would be assigned with a larger region, and the rank closes to the exit point would only be assigned to deal with a few layers. The other method to solve this problem is "Redistribution during the simulation". The method would keep the evenly distribution part unchanged. However, as the simulation progresses, those layers- idle layers at the top may contain zero robots. Then a domain redistribution process would redistribute the whole simulation domain again according to the robot density in each layer. For example, the rank takes charge of the top layers should be reassigned more layers, but the rank at the bottom would be assigned few layers.

4.3.2 Robot Supervision: Since we used MPI_Gatherv and MPI_Broadcast in our code at each step. The implicit barriers may slow down the whole process, especially when the number of tasks is very large. In our experiment, the strong scaling becomes weird when the number of tasks reaches 50, which means the robot number a task can take the most is 20 without the influence of the overhead for the communications. This bottleneck gives us a ceiling of 6 times speed up. Plus, another point that results in the bottleneck is that rank 0 still needs to do part of the serial job while other ranks can only wait. To get further improvements, a new idea is to improve the RRA* algorithm. The updated algorithm should have less data dependency. Plus, use send and receive instead of gathering and broadcasting. Another try could be done is to use domain decomposition for the non-collision rules to break the bottleneck. But the domain decomposition has its own bottleneck.

5. Conclusion

In this paper, we implanted two parallelism method, domain decomposition and robot supervision, on our RRA* real-time path-planning algorithm. The code run efficiently on a petascale supercomputer Stampede, and according to the strong and weak scaling experiments, the parallelism method both have acceptable scaling properties. In the strong/weak scaling experiment, the code was run up to 256 tasks from 8 nodes on the Stampede super computer. We discussed that the map complexity, robot density and probability in the RRA* may affect the ultimate success rate in some degree. We also found that the two parallelism still can be improved to solve performance bottleneck such as load unbalance. In the future work, we plan to improve the two parallelism method as discussed above and reduce the data dependencies of our RRA* algorithm to make it more useful for robotic and medical applications.

ACKNOWLEDGMENTS

Thank Professor Amanda Randles and Daniel Puleri for the help of the project idea.
Thank Zizhao Fang for his help of data structure thought
Thank Rich Wang for his accompany and bad performance

REFERENCES

[1] Burghart, C., Wurll, C., Henrich, D., Raczkowsky, J., Rembold, U., & Worn, H. (1998). On-line motion planning for medical applications. In IECON'98. Proceedings of the 24th Annual Conference of the IEEE Industrial Electronics Society (Cat. No. 98CH36200) (Vol. 4, pp. 2233-2238). IEEE.

[2]"A* Search Algorithm - GeeksforGeeks", GeeksforGeeks, 2018. [Online]. Available: https://www.geeksforgeeks.org/a-search-algorithm/. [Accessed: 16- Dec- 2018].

[3]H. Qu, S. Yang, A. Willms and Zhang Yi, "Real-Time Robot Path Planning Based on a Modified Pulse-Coupled Neural Network Model", IEEE Transactions on Neural Networks, vol. 20, no. 11, pp. 1724-1739, 2009.

[4]Chinnaiah, M. C., Sanjay, D., Kumar, P. R., & Savithn, T. S. (2012, December). A Novel approach and Implementation of Robot path planning using Parallel processing algorithm. In Communications, Devices and Intelligent Systems (CODIS), 2012 International Conference on (pp. 345-348). IEEE.

[5]"Stampede - Texas Advanced Computing Center", Tacc.utexas.edu, 2018. [Online]. Available: https://www.tacc.utexas.edu/systems/stampede. [Accessed: 16- Dec- 2018].

[6] "Stampede2 User Guide - TACC User Portal", Portal.tacc.utexas.edu, 2018. [Online]. Available: https://portal.tacc.utexas.edu/user-guides/stampede2. [Accessed: 16- Dec- 2018].