CISTER

Research Centre in
Real-Time & Embedded
Computing Systems

Conference Paper

On the Robot Path Planning using Cloud Computing for Large Grid Maps

Imen Chaari
Anis Koubaa
Basit Qureshi
Habib Youssef
Ricardo Severino
Eduardo Tovar

CISTER-TR-180411

On the Robot Path Planning using Cloud Computing for Large Grid Maps

Imen Chaari, Anis Koubaa, Basit Qureshi, Habib Youssef, Ricardo Severino, Eduardo Tovar

*CISTER Research Centre

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail:

http://www.cister.isep.ipp.pt

Abstract

Global path planning consists in finding the optimalpath for a mobile robot with the lowest cost in the minimumamount of time, without colliding with the obstacles scatteredin the workspace. In this paper, we investigate the benefits ofoffloading path planning algorithms to be executed in the cloudrather than in the robot. The contribution consists in developinga vertex-centric implementation of R A "17 [1], a version of A "17that we developed for grid maps and that was proven to bemuch faster than A "17, using the distributed graph processingframework Giraph that rely on Hadoop. We also developed acentralized cloud-based C++ implementation of the algorithm forbenchmarking and comparison purposes. Experimental results ona real cloud shows that the distributed graph processing Giraphfails to provide faster execution as compared to centralized C++implementation for different map sizes and configuration due tonon-real time properties of Hadoop.

On the Robot Path Planning using Cloud Computing for Large Grid Maps

Imen Châari *, Anis Koubâa †‡§,Basit Qureshi†, Habib Youssef ¶, Ricardo Severino §, Eduardo Tovar §
* PRINCE Research Unit, University of Manouba (ENSI), Tunisia.
† Prince Sultan University, College of Computer and Information Sciences, Saudi Arabia
‡ Gaitech International Ltd, China
§ CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Porto, Portugal.
¶ University of Sousse, PRINCE Research Unit, Sousse, Tunisia.

imen.chaari@coins-lab.org, akoubaa@coins-lab.org, qurshi@psu.edu.sa, habib.youssef@fsm.rnu.tn,
RARSS@isep.ipp.pt, emt@isep.ipp.pt

***Abstract*—Global path planning consists in finding the optimal path for a mobile robot with the lowest cost in the minimum amount of time, without colliding with the obstacles scattered in the workspace. In this paper, we investigate the benefits of offloading path planning algorithms to be executed in the cloud rather than in the robot. The contribution consists in developing a vertex-centric implementation of RA^* [1], a version of A^* that we developed for grid maps and that was proven to be much faster than A^*, using the distributed graph processing framework Giraph that rely on Hadoop. We also developed a centralized cloud-based C++ implementation of the algorithm for benchmarking and comparison purposes. Experimental results on a real cloud shows that the distributed graph processing Giraph fails to provide faster execution as compared to centralized C++ implementation for different map sizes and configuration due to non-real time properties of Hadoop.**

I. Introduction

This research work addresses the global path planning problem for large environment. We consider that the robot has a complete knowledge about its workspace modeled as a grid map that contains static obstacles. This problem becomes more challenging for large grid environments as the process of searching a path in large grid maps containing different obstacles requires large execution time [2].
In [1], we presented a relaxed version of the A^* algorithm called relaxed astar RA^* to solve the path planning problem. This algorithm has a linear complexity, it provides near optimal solutions with less than 2% error and with an extremely reduced execution time as compared to A^*.

Nowadays, with the emergence of cloud robotics [3], recent studies have proposed to offload heavy computation from robots to the cloud as robots are usually small, lightweight, and have minimal hardware configurations. This leads to having limited processing capabilities that is insufficient for handling computation-intensive tasks which will not allow the robot to complete its mission in a short period of time. In fact, when connected to the cloud, robots can benefit from the powerful computational, storage, and communications resources in the cloud. This solution has a number of advantages over traditional approach, where all actions are performed on the robot. This led us to think in outsourcing the path calculation process using the cloud computing techniques in order to investigate the the benefits of the cloud robotic solution to solve grid path planning. Our objective is to reduce the execution time of the path searching process in large grid maps as far as possible.
Accordingly, we used the open source graph processing framework Giraph [4] that rely on the open source Hadoop framework [5] and we designed a vertex-centric RA^* algorithm. Giraph was our first choice to implement RA^* for two reasons, first Giraph has an increasing number of contributors and companies that were already running clusters based on it, including Facebook, Twitter and Linkedin. In addition, there is no major research efforts that presented a comprehensive solution on how to use Hadoop and Giraph to solve the path planning. This represent the main motivation of this work. The rest of this paper is organized as follows. In Section 2, we give an overview of some relevant works dealing with Giraph and shortest path problem. Section 3 presents a background about Giraph framework. The vertex-centric RA^* algorithm is described in Section 4. In Section 5, we present the simulation results and finally Section 6 concludes the paper.

II. Related Works

To the best of our knowledge, no previous research works addressed the robot path planning problem in large grid maps using the Giraph framework. In our previous works [6], [7], we designed and developed a cloud robotics management system that allows the access an real-time control of robots and drones over the Internet. In this paper, we are interested in investigating the other facet of cloud robotics related to computation offloading, with the particular emphasis on robot path planning using cloud computing applications.
Some research works have tackled the shortest path problem without considering the obstacle avoidance. For example, the Giraph package [4] provides an implementation of the Dijkstra algorithm. In reference [8], the authors firstly implement the Breadth-first search algorithm using Giraph to solve the

shortest path problem. They tested their algorithms in two large graphs; the first one contains 10000 vertices and 121210 edges and the second has 100000 vertices and 1659270 edges. The authors compared the implementation of breadth-first search on Giraph with a Hadoop MapReduce solution. The simulation results confirm the superiority of Giraph over Hadoop for this kind of task. The intresting point of this paper is the proposition of new improvements that modify Giraph framework in order to support dynamics graphs, in which the input may change while the job is being performed. In [9], the authors descriped how they modifiy the Giraph framework to be able to support managing Facebook-scale graphs of up to one trillion edges. They presented new graph processing techniques such as composable computation and superstep splitting. The work [10] is centered around the development of a cloud-based system, called Path Planning as a Service (PPaaS) for robot path planning. The authors proposed a three-layered system architecture, which facilitates on-demand path planning software in the cloud. They have implemented the proposed system with the Rapyuta cloud engine and used Robot Operating System (ROS) platform as the communication framework for the entire system.

The Okapi project [11] presents a multiple source shortest paths algorithm. The idea of the algorithm consists to apply the single source shortest path algorithm (SSSP) for a given number of vertices. A number of vertices are chosen and the SSSP algorithm is run in parallel at the aim to improve the performance by reducing the number messages. The work [12] compared the Hadoop Map Reduce and the Bulk Synchronous Parallel approach, used in Giraph, using the single source shortest path computation, existing in the giraph package, and the relational influence propagation algorithms. They tested the algorithm in cluster containing 85 machines each has 7500 MB of RAM; They used various datasets, with different sizes, to evaluate the performance of the approaches. They varied the number of nodes in the cluster and measured the execution times of the algorithm. The results revealed that iterative graph processing with BSP implementation significantly outperform MapReduce. Pace et al. in [13] provided a theoretical comparison of the Bulk Synchronous Parallel and the MapReduce models. In terms of graph processing, they noticed, that Breadth First Search algorithm cannot be efficiently implemented by means of the MapReduce model. The iGiraph framework for Processing Large-scale Graphs is described in [14]. iGiraph is a modified version of Giraph. The authors use a new dynamic re-partitioning approach to minimize communication between computation nodes and thus reduces the cost of resource utilization. To evaluate the performance of the new framework, the authors tested the shortest path, the connected components and the PageRank algorithms on a cloud formed by 16 machines each has 8 GB of RAM. They used three graphs containing between 403394 and 1632803. They compared the iGiraph and Giraph frameworks in terms of number of messages exchanged between partitions, the execution time and the number of workers used and they proved that the execution time of the shortest path algorithm using iGiraph is not reduced as compared to Giraph, it takes around 5 minutes. For the other algorithms the execution time is decreased and the number of messages passing through network is reduced significantly. The authors in [15] proposed two models to maximize the performance of graph computing on heterogeneous cluster (different nodes with various bandwidth or CPU resource). They presented a greedy selection model to select the optimal worker set for executing the graph jobs for graph processing systems using hash-based partition method and a heterogeneity-aware streaming graph partitioning to balance the load among workers. Experiments were made using five different algorithms (page rank, shortest path, random walk, kcore and weakly connected-components) on two different clusters: the university lab cluster (46 machines) and EC2 cluster (100 instances). The experiments prove that the execution time of the algorithms is reduced by 55.9% for lab cluster and 44.7% for EC2 cluster as compared to traditional method. In [16], the authors compared Giraph against GraphChi. In 2012, it was proven that GraphChi is able to perform intensive graph computations in a single PC in just under 59 minutes, whereas the distributed systems were taking 400 minutes using a cluster of about 1000 computers. In this work, the authors compared the new versions of the two frameworks by testing three different algorithms (PageRank, Shortest path and weakly connected-components) and they concluded that even for a moderate number of simple machines (between 20 and 40 each has 1GB of RAM), Apache Giraph outperforms GraphChi in terms of execution time for all the algorithms and datasets used. [17] proposed a graph-oriented mechanism to achieve the smart transportation system. The overall traffic information were obtained from road sensors which forms big data. Graphs were generated from big data. Various graph algorithms were implemented using Giraph to acheive real time transportation. Dijkstra was implemented to select the quickest and shortest path. In addition to the path cost, other parameters are taken into consideration to evaluate the path quality in the process of searching the shortest path such as the current traffic intensity as well as the vehicles speeds. However, experiments were conducted only on a single node machine and they have used not very large graph (less than 90000 nodes and edges).

III. Giraph: How parrallel Algorithms are Implemented and Processed?

In 2012, Apache introduced Giraph [4] an open source distributed graph processing framework. It is a loose implementation of Google's Pregel [18]. Giraph is widely used due to its increasing number of contributors and the companies that were already running clusters based on it that included Facebook, Twitter and Linkedin.

Giraph follows the Bulk Synchronous Parallel (BSP) computation model. A Giraph program (Giraph job) consists of three main steps: An input step, where the graph is loaded and distributed among the workers machines, followed by a sequence of iterations called supersteps and finally an output

step to write down the results. This process is illustrated in Figure 1.

At the beginning, the master starts by loading the input file (graph) from the Hadoop Distributed File System (HDFS). Giraph provides tools called Inputs Format that define how to read data from the input file into the mapper instances. A large variety of Input Formats are already implemented within the Giraph package and we could also define our own Input Format implementations to format the input to the program. Then, the master splits the input graph file among the workers using a partitioner. By default Giraph use a HashPartitioner, distributing the vertices among the workers at random. At the end of the Input Superstep, the workers become ready to perform computation. Computation proceeds as a sequence of iterations, called supersteps. In each superstep, the workers run in parrallel a user defined function `compute()` for the active vertices in their respective partitions. Initially, every vertex is active, then a vertex can turn into inactive state by calling voteToHalt() method. The overall program terminates if every vertex is inactive.

The user-defined function specifies the behaviour at a single vertex and a single superstep S and there is no access to other vertices from the current vertex. The function can read messages that are sent to the vertex in superstep $S-1$, send messages to other vertices that will be received at superstep $S+1$, and modify the state of the vertex and its outgoing edges. Messages are typically sent along outgoing edges, but you can send a message to any vertex with a known identifier. After all active vertices finish their local computation in a superstep, a global synchronization phase allows global data to be aggregated, and messages created by each vertex to be delivered to their destinations.

Finally, when there are no more messages to process or the computation is halted, the workers store the output graph back in HDFS during the Output Superstep. We should distinguish two different types of messages: Local messages between vertices (or subgraphs) belonging to the same worker co-located on the same machine, and remote messages transferred over network between vertices (subgraphs) on different machines. In fact, before the computation phase Vertices are hashed and distributed across multiple machines. To exchange their status, the vertices send messages. When the vertex sends a message, the worker first determines whether the destination vertex of the message is owned by a remote worker or the local worker.

IV. RAStar Using Giraph and Hadoop Frameworks

This section presents the vertex-centric RA^* algorithm. To design a Giraph algorithm we should consider the ‘‘think like-a-vertex’’ programming model, i.e. the algorithm should be transformed to show the behavior of one vertex in a given superstep.

Step 1: Input Format Selection:
As explained in the previous section, the first step of any Giraph application is to choose a MapReduce Input Format. For the RA^* job, we choose the `JsonLongDoubleFloatDoubleVertexInputFormat`

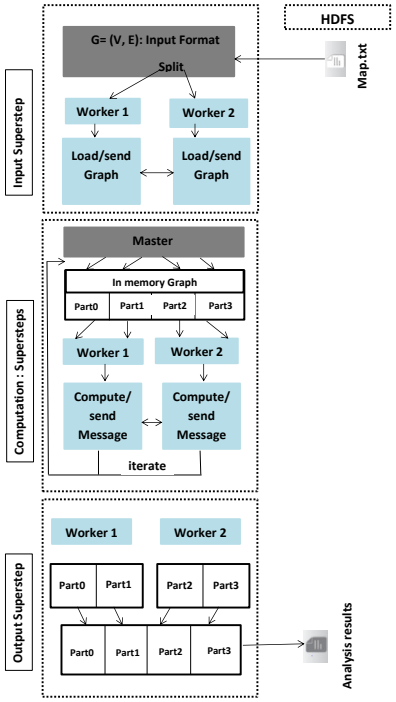

Fig. 1: Apache Giraph Dataflow

format to present the grid map as a graph. The grid will be transformed to a serie of lines that represent the graph (the input file of the job), each line has the following format: $JSONArray(vertexID, vertexValue, JSONArray(JSONArray($ $DestVertexID, EdgeValue), ...))$

Each line is composed from three elements to describe one cell in the grid. The $vertexID$ is the cell ID, $vertexValue$ is the cell value which can have one of these two possible values 0 or 100, where 0 means the cell is free and 100 means the cell is occupied by an obstacle and finally a list of adjacent neighbors, each neighbor is represented by its ID $DestVertexID$ and the edge value $EdgeValue$. The edge value has two possibles value 1 or 1.4; it is equal to 1 if the neighbor vertex exists in the vertical or horizontal direction in the map and 1.4 if the adjacent vertex exists in the diagonal direction.

Step 2: Implementation of the Compute Method:
The RA^* algorithm is implemented as a subclass of the `AbstractComputation` class that provides a `compute()` method which will be executed by each *active* vertex in the graph in each superstep. Vertices begin the process as actives, then they become inactives if they vote to halt and they can come back to activity only if they receive a message from neighbors. At the end of the superstep, it is ensured that all active vertices have executed the compute method and messages are being delivered for the next superstep.
This method has two inputs as shown in the compute

The compute method signature

```
@Override
public void compute (Vertex < LongWritable,
DoubleWritable,FloatWritable > vertex, Iterable < DoubleWritable >
messages)
{
...
}
```

Superstep 0 of The Algorithm

```
// if we are in superstep 0 and the processed
vertex is the start vertex
if (getSuperstep() == 0) then
    if (isStart(vertex)) then
        // update the h_score value of the start
        vertex
        h_score=calculateHScoreValue();
        // send message containing its g_score
        value to the neighbors
        sendCurrentVertexMessages(vertex);
        // update the f_score of the vertex
        vertex.setValue(h_score);
    else
        vertex.setValue(MAX_VALUE));
    end
    vertex.voteToHalt();
end
```

Send message method

```
public void sendCurrentVertexMessages(Vertex) {
for each edge of the current vertex do
    sendMessage(DestVertexID,
    Message(SenderVertexID,SenderVertexGScore));
end
}
```

Receive message method

```
for (each received message) do
    CurrentVertexGScore=getGScoreOfSenderVertex
    (message);
    CurrentVertexID=getIDOfSenderVertex (message);
end
```

Vertex computation method

```
g_score=CurrentVertexGScore+dist_edge (VertexID,
CurrentVertexID);
h_score=calculateHScoreValue (VertexID, CurrentVertexID);
vertex.setValue(g_score+h_score);
AddToOpenList(vertexID, f_score);
aggregate(OPENLIST_AGG, newElementAddedToOPL);
```

method signature, the vertex expanded $Vertex < LongWritable, DoubleWritable, FloatWritable >$ and the message received by this vertex. The same compute method is called for all iterations, so any special cases that need to be handled in a particular superstep must be done through a call to the `getSuperstep()` function. For example, in superstep 0, all vertices are actives at the begining. However, only the start vertex of the robot updates its f_score and sends messages to its neighbors. Other free vertices set their values to the maximum double value. At the end of superstep 0, all vertices vote to halt and become inactives. This is presented in the second method.

In the next supersteps, an active vertex that invokes the compute method can: ***(1)*** **Send messages to its adjacent nodes**, if it is elected to be the next current vertex, in order to update the g_score values of the neighbors. The `sendMessage()` implemented within the Giraph package enables the vertex to send only its value to its neighbors, thus, we implemented our own `sendMessages()` method in order to send both the ID and the value (g_score) of the current vertex to the neighbors.

(2) **Receive a message from its neighbors:** (Receive message method) neighbors receive the current vertex ID and g_score in order to update their g_score and f_score. ***(3)*** **Make computation:** (Computation method) the vertex that receives message updates its value (g_score) then it will be added to the open List.

Finally, after sending all messages, the vertex will vote to halt. The overall execution will be halted when the target vertex is found or the master stops the computation using `haltComputation()` method if the path has not been found.

Step 3: Implementation of the RAStar Master Compute class:

The RA^* Master Compute `RAStarMasterCompute` is a subclass of `DefaultMasterCompute` class, it is a way to introduce centralization into our algorithm. It executes some computation between supersteps. In each superstep, the RAStarMasterCompute runs first then the workers execute the compute method for their active vertices. Before each superstep, the RAStarMasterCompute is called to register the aggregators using the `initialize()` function (this will be explained in detail in the next paragraph), then the `compute()` method of the RAStarMasterCompute class is invoked, in this method we choose the vertex that has the minimum f_score existing in the open list to be the next current vertex. The new current is removed from the open list and shared among all the workers. If the open list is empty and the target is not found then the algorithm computation is stopped using the function `haltComputation()`, otherwise the search process continue until reaching the goal cell if a path exists.

Aggregators: They are global objects visible to all vertices, they are used for coordination and data sharing. Each aggregator is assigned to one worker that receives the partial values from the all the other workers and is responsible for performing the aggregation. Afterwards, it sends the final value to the master. Moreover, the worker in charge of an

RAStarMasterCompute class: compute method

```
1  if (getSuperstep()>1) then
2  |
3  |   if (openList is not empty && goalVertex is not found) then
4  |   |   currentVertex = the node in openList having the lowest
   |   |   f_score;
5  |   |   newOpenList=remove currentVertex from
   |   |   openList;
6  |   end
   |   // add the current vertex and the new open
   |   list to the aggregator to be shared between
   |   workers;
7  |   Add currentVertex to the current vertex aggregator;
8  |   Add newOpenList to the openList aggregator;
9  end
```

LongDenseVectorOverwrite Aggregator

```
1  @Override
2  public LongDenseVector createInitialValue() {
3  return new LongDenseVector();
4  }
5
6  @Override
7  public void aggregate(LongDenseVector vector) { if (vector.length()!=0)
   then
8  |   getAggregatedValue().overwrite(vector);
9  end
10 }
```

aggregator sends the final value to all the other workers.
During supersteps, vertices provide values to the aggregator. These values will be available for other vertices in the next superstep. Different types of aggregators are used our algorithm. We use an aggregator for the openList `MyDoubleDenseVectorSumAggregator` , each expanded vertex is added to the openList if it does not exist. Another aggregator is used to indicate if the goal vertex is found or not `BooleanOverwriteAggregator`. Providing a value to aggregator is done by calling the `aggregate()` function. To get the value of an aggregator during the previous supestep we used `getAggregatedValue()` function. The aggregators are registered in the RAStarMasterCompute class in $initialize()$ fuction by calling `registerAggregator()` function or `registerPersistentAggregator()` according to the aggregator type chosen regular or persistent. The value of a regular aggregator will be reset to the initial value in each superstep, whereas the value of persistent aggregator will live through the application.
Many aggregators are already implemented in the giraph package. In RA^* job, we implemented three new aggregators. The new aggregator extends `BasicAggregator` class, two functions must be implemented the `aggregate()` function and the `createInitialValue()` function (LongDenseVectorOverwrite Aggregator example)

V. Performance Evaluation

In this section we present the results of various experiments carried out to investigate the performance of the RA^* algorithm implemented using Giraph framework. We compared the results of the vertex-centric RA^* algorithm against the C++ version.

A. Cloud Framework

The vertex-centric RA^* algorithm is tested on a small Dreamhost cloud cluster [19] formed by seven virtual machines, one Master and six slaves, each with three 2.10GHz processors Intel (R) Xeon (R) CPU E5-2620, 8GB RAM and a 80 GB disk, running Ubuntu 14.4 as an operating system. The master node runs only the namenode task whereas all computations are performed in the datanodes running in the slaves machines. We used Hadoop 2.4.0 and Giraph 1.2.0. The same scenarios are used to test the centralized version of RA^* under one node from the cluster.

B. Experimental Scenarios

We considered three different maps for test: The first with dimensions 500*500 cells, the second is of size 1000*1000 and the third map is 2000*2000 map. Table I provides details about the maps that we have used for simulation. We considered 21 different scenarios to test the three maps, where each scenario is specified by the coordinates of randomly chosen start and goal cells. Each scenario, is repeated 5 times (i.e. 5 runs for each scenario) and with six different numbers of workers. In total, 630 runs to evaluate the performance of the vertex-centric RA^*. For each run, we recorded the length of the generated path and the execution time of RA^* (without hadoop initialisation time) and the time required by hadoop to initialise the job. The execution time of an algorithm in a given map is the average of the 5 execution times recorded, calculated with 95% of confidence interval.

TABLE I: Grid Maps Characteristics

Map size	Number of vertices	Number of edges
500*500	250000	499000
1000*1000	1000000	1998000
2000*2000	4000000	7996000

In what concerns paths found, the paths generated by the two implementation of RA^* (vertex-centric and C++) are the same of all maps and for all runs.
Figure 2 compares the average execution times consumed by the two versions of RA^* and the Hadoop initialization. We clearly observe from this figure, that the difference in execution time between the two algorithms is significant, for example for 1000*1000 grid map RA^* implemented using Giraph and executed in a cluster composed from 3 machines is seventy thousand times larger than RA^* implemented using C++. This can be explained by different reasons; First, the numbers of iterations of RA^* implemented using Giraph is higher than RA* implemented on C++. This is related to the Giraph programming concepts. In fact, in superstep i, the

current vertex sends a message to their neighbors in order to update their g_score values, the messages will be received in the next superstep (i+1), thus to update the g_score we need two superstep one for sending messages, one for updating g_score of neighbors which contributes in increasing the number of superteps and then the execution time of the whole algorithm. Moreover, the openList in the algorithm cannot be implemented only as an aggregator as it is must be shared between all workers. The Aggregators are manipulated only by the master, it can add/remove values to/from the openList, so if the neighbors of the current vertex are added to the openList in superstep i, this will be visible for workers only in superstep i+1 after the master update which also contributes in increasing the number of supersteps and then the execution time of the algorithm. In addition, we should note that the time consumed by the RA^* is the time required to do computation plus the time required to communication between workers which also contribute in the increasing of the run time. In addition, we see that the increasing of the number of machines in the cluster from 3 to 7 contributes in reducing the execution times; for 500*500 and 1000*1000 grid map the runtime is reduced up to 34,2% and for 2000*2000 grid map up to 16.2%. However, the C++ version of RA^* always exhibits the best execution time. Also looking to Figure 2, we observe that the time required by hadoop for the initialisation and the shutdown of the job, without considering the computation time, is greater than the runtime of the RA^* implemented using C++. This clearly proves that Giraph/Hadoop framework (using 7 machines each has 8GB of RAM) is not an appropriate technique for solving the path planning problem in large environments.

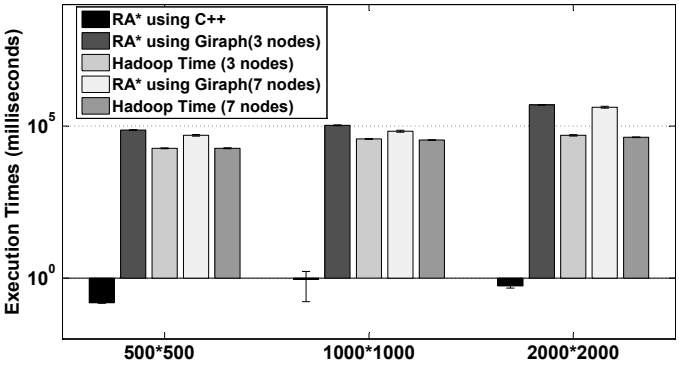

Fig. 2: Average Execution Times of the different implementation of RA* and hadoop initialisation time for 500*500, 1000*1000 and 2000*2000 grid maps

VI. Conclusion

In this paper, we implemented RA^* using Giraph that works on the top of Hadoop. We compared the efficiency of the new version of RA^* with a C++ implementation of RA^*. Our objective was to assess the efficiency of Giraph platform in solving robotics path planning algorithms in terms of solution quality and real-time performance. Our results showed that cloud computing techniques are not efficient for the robot global path planning problem due to the non-real time properties of Hadoop and the Giraph programming concepts. Currently, we working towards investigating the capabilities of other frameworks in solving this problem.

Acknowledgement

This work is supported by Gaitech Robotics in China. It is also supported by Robotics and Internet-of-Things (RIOTU) Lab and Research and Initiative Center (RIC) of Prince Sultan University, Saudi Arabia.

References

[1] A. Ammar, H. Bennaceur, I. Châari, A. Koubâa, and M. Alajlan, "Relaxed dijkstra and a* with linear complexity for robot path planning problems in large-scale grid environments," *Soft Computing*, vol. 20, no. 10, pp. 4149–4171, 2016.

[2] I. Chaari, A. Koubaa, H. Bennaceur, A. Ammar, M. Alajlan, and H. Youssef, "Design and performance analysis of global path planning techniques for autonomous mobile robots in grid environments," *International Journal of Advanced Robotic Systems*, vol. 14, no. 2, pp. 1–15, 2017.

[3] R. chaari, F. Ellouze, A. Koubaa, B. Quershi, N. Pereira, and E. Youssef, Habib an Tovar, "Cyber-physical systems clouds: A survey," *Computer Networks*, vol. 108, pp. 260–278, 2016.

[4] Giraph. [Online]. Available: http://giraph.apache.org/

[5] Apache hadoop. [Online]. Available: https://hadoop.apache.org/

[6] A. Koubaa and B. Quershi, "Dronetrack: Cloud-based real-time object tracking using unmanned aerial vehicles," *IEEE Access*, 2018.

[7] A. Koubaa, B. Quershi, M.-F. Sriti, Y. Javed, and E. Tovar, "A service-oriented cloud-based management system for the internet-of-drones," in *Autonomous Robot Systems and Competitions (ICARSC), 2017 IEEE International Conference on*. IEEE, 2017, pp. 329–335.

[8] M. Aurelio, B. Fagnani, and G. Lotz, "Dynamic graph computations using parallel distributed computing solutions." [Online]. Available: http://www.marcolotz.com/wp-content/uploads/2014/05/LotzReport.pdf

[9] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[10] M.-L. Lam and K.-Y. Lam, "Path planning as a service ppaas: Cloud-based robotic path planning," in *IEEE International Conference on Robotics and Biomimetics (ROBIO)*, 2014, pp. 1839–1844.

[11] The okapi project. [Online]. Available: https://github.com/grafos-ml/okapi

[12] T. Kajdanowicz, W. Indyk, P. Kazienko, and J. Kukul, "Comparison of the efficiency of mapreduce and bulk synchronous parallel approaches to large network processing," in *Data Mining Workshops (ICDMW), 2012 IEEE 12th International Conference on*. IEEE, 2012, pp. 218–225.

[13] M. F. Pace, "Bsp vs mapreduce," in *Proceedings of the International Conference on Computational Science*, vol. 9, 2012, p. 246255.

[14] S. Heidari, R. N. Calheiros, and R. Buyya, "igiraph: A cost-efficient framework for processing large-scale graphs on public clouds," in *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*. IEEE, 2016, pp. 301–310.

[15] W. C. M. H. C. W. A. in Graph Computation, "Jilong xue and zhi yang and shian hou and yafei dai," in *2015 IEEE International Conference on Big Data (Big Data)*, 2015, pp. 154–163.

[16] J. Lu and A. Thomo, "An experimental evaluation of giraph and graphchi," in *Advances in Social Networks Analysis and Mining (ASONAM), 2016 IEEE/ACM International Conference on*. IEEE, 2016, pp. 993–996.

[17] M. M. Rathore, A. Ahmad, A. Paul, and G. Jeon, "Efficient graph-oriented smart transportation using internet of things generated big data," in *Signal-Image Technology & Internet-Based Systems (SITIS), 2015 11th International Conference on*. IEEE, 2015, pp. 512–519.

[18] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[19] Dreamhost. [Online]. Available: https://www.dreamhost.com/