**Streaming Parallel Rao-Blackwellized Particle Filtering SLAM for Robots in the Cloud**

**Abstract** In this paper we propose a cloud based distributed architecture to solving the Simultaneous Localization and Mapping (SLAM) problem. We introduce a distributed stream processing based parallel version of a Rao-Blackwellized Particle Filtering SLAM algorithm. With this approach we obtained significant efficiency improvements in the computation time. The gained efficiency allows the algorithm to increase the number of particles and the frequency of the calculations which are factors for increasing the accuracy of the maps built. Because the computation happens in a cloud environment the robots onboard computer can be a low end computer. Our method is generic and can be applied to any computationally intensive particle filtering algorithm.

## Introduction

Cloud Computing has long being identified as a key enabling technology for Internet of Things applications. Internet of things applications will connect everything ranging from simple devices like our thermostat to complex industrial machinery to complex robots to the services running in the cloud. The cloud services are used by these devices to do both real time and offline analytics at large scale to process the huge amount of data produced by these devices. In one dimension there are computationally intensive algorithms for processing device data that can benefit from cloud processing for real time response. The methods used by these computationally expensive algorithms are powerful but impossible to run near the devices due to high computational and specialized hardware requirements. In the other dimension there are applications that have to be scaled to support vast number of devices and are inherently suitable for central data processing. This paper explores the first dimension of applications by implementing a computationally expensive robotics application to showcase how to achieve complex parallelism for real time applications in the cloud.

Parallel implementations of the real time robotics algorithms mostly focus on running on multicore machines using threads as the primary parallelization mechanism. Scaling such applications using threads in multicore machines is bounded by the number of CPU cores available and the amount of memory in a single machine; which are often not enough for computationally expensive algorithms to provide a real time response. Being able to execute computations in parallel, in a distributed environment can be beneficial to computationally expensive robotics applications requiring low latencies because of the virtually unlimited CPU and memory availability.

Simultaneous localization and mapping (SLAM) is a very important capability for mobile robots and has been studied extensively in the literature. Computing the position of a robot in an unknown environment amidst measurement errors and simultaneously computing a map of the environment can be a computationally challenging task. SLAM algorithms can use various inputs like distance readings from a laser rangefinder, images of the environment and images combined with distances etc. We have chosen a popular SLAM algorithm called GMapping; which uses distance measurements from a laser range finder. GMapping is a Rao-Blackwellized Particle Filtering(RBPF) based SLAM algorithm[[1](#_ENREF_1), [2](#_ENREF_2)]. The algorithm maintains a number of particles each containing a probable map of the environment and a possible trajectory of the robot. Robot uses the laser sensor to find the distances to the objects in its path. As the robot moves through the environment it gets new distance reading as well as new position measurements. For each of these new readings, algorithm calculates a weight for each particle depending on how probable that particle is given the readings. Then it draws particles with replacement from this set according to their weights in a step called resampling. Resampled particles are used with the next reading. At each reading the algorithm takes the map associated with the particle of highest weight as the correct map. The computation time of the algorithm depends on the number of particles used, size of the environment and the number of points in the distance reading. In general by increasing the number of particles, the accuracy of the algorithm can be improved. Moving some of the expensive computations to the cloud allows the robots onboard computer to be a low end computer consuming less power. This algorithm has an open source reference implementation and is used by off the shelf robots like Turtlebot.

Profiling has shown that RBPF SLAM algorithm spends nearly 98% of its computation time on the Scan Matching step which is done for each particle for each laser reading and this computation can be done for a particle independently of the other particles making this algorithm ideal for parallel execution. In a distributed environment the particles can be moved to different computation nodes and the computation can be executed in parallel. Even though the computations over the particles can be easily parallelized, the resampling which requires information about all the particles is complex because the particles are distributed over distributed nodes. The resampling removes some of the existing particles and duplicates the particles in the system. After resampling some of the particles has to be re-distributed over the cluster. To enable continuous processing of the incoming laser readings, the distributed algorithm has to work on a stream of laser readings and odometry readings coming from the robot in real time.

At Indiana University, we have developed framework called IoTCloud, which can transfer data from devices to a cloud computing environment for scalable data processing for real time response. The data from the devices are encapsulated into events and sent to cloud systems in real time. IoTCloud employs a distributed stream processing frameworks (DSPF)[[3](#_ENREF_3)] for developing and executing scalable real time applications. In general a real time application running in a DSPF can be modeled as a directed acyclic graph (DAG) consisting of streams and stream processing tasks. Stream tasks are at the nodes of the graph and streams are the edges connecting the nodes. A stream is an unbounded sequence of events flowing through the edges of the graph and each such event consists of data represented in some format. The processing tasks at the nodes consume input streams and produces output streams. A DSPF provides the necessary API and infrastructure to develop and execute such applications in a cluster of computation nodes. Their main tasks include 1. Providing an API to develop streaming applications 2. Distributing the stream tasks in the cluster and managing the life cycle of tasks 3. Creating the communication fabric 4. Monitoring and gathering statistics about the applications 4. Provide mechanisms to recover from faults. In general DSPF allows the same task to be executed in parallel and provide rich communication channels among the tasks. Some DSPF’s allow the applications to define the stream workflow graph explicitly and some creates the graph dynamically at run time from implicit information. We have developed a distributed streaming parallel version of the RBPF SLAM algorithm by mapping it to a stream processing DAG within the IoTCloud framework.

The main contribution of this paper is to propose a novel framework to compute particle filtering, specifically RBPF based SLAM in a distributed cloud environment to achieve higher efficiency in computation time. In the rest of the paper we will first discuss the related work, then we introduce the IoTCloud framework. After this we discuss how to develop the robotics applications using the SLAM algorithm and then discuss the design of the parallel RBPF SLAM algorithm. Finally we will conclude with the results and discussion.

## Related Work

According to the knowledge of the Authors, using distributed cloud infrastructure to execute particle filtering based SLAM algorithms has not being studied in the literature. Recent work by[[4](#_ENREF_4)] has exploited multicore and GPU architectures to speed up the particle filtering based computations and [[5](#_ENREF_5)] has used multicore architecture to create a thread based parallel implementation of the GMapping algorithm with good performance gains. Our approach depends on a distributed environment where multicore architecture of individual machines and multiple such nodes are being exploited by algorithm.

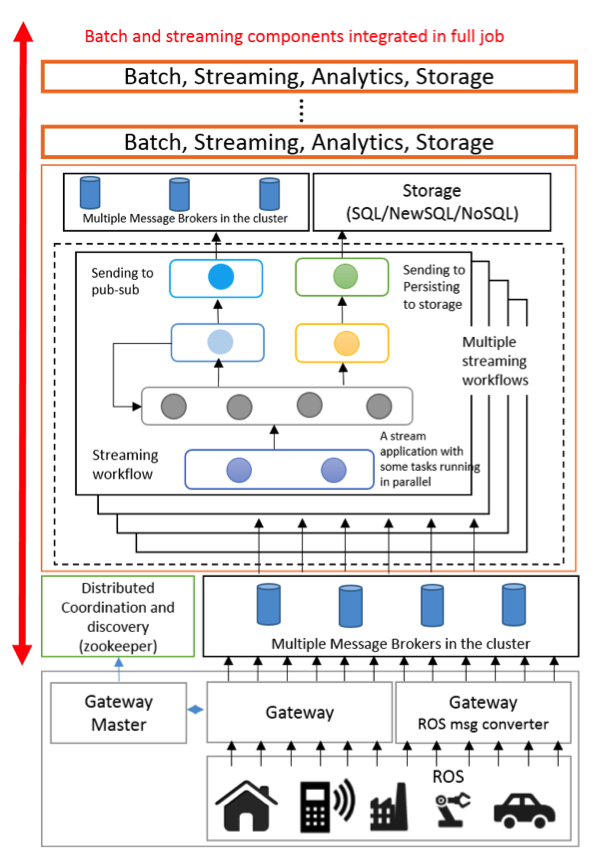
C2TAM[[6](#_ENREF_6)] is a framework developed to move some of the expensive computations of a SLAM algorithm in to a cloud environment for processing. The SLAM algorithm in C2TAM is different from the algorithm used in this work and has different computation requirements. Also our work propose a generic scalable real time framework for computing the maps online with significant gains in the processing time while C2TAM doesn’t provide a real time framework. Zhang *at el* [[7](#_ENREF_7)] describes an approach where CUDA API is used to run the scan matching step of GMapping algorithm in GPUs to improve the performance of the algorithm.

Distributed streaming algorithms have being developed for tasks like clustering social data in stream[[8](#_ENREF_8)] with excellent performance enhancements. The algorithm we have developed is different from those implementations because of the nature of the parallelism and the real time constraints. Those applications are mostly data parallel applications where in this application we focus on a computationally parallel application. Executing computationally parallel algorithms in a real time distributed streaming environment is not well studied in the literature and we would like to continue to explore this area.

## IoTCloud framework

IoTCloud[[9](#_ENREF_9)] is an open source framework developed at Indiana University to connect IoT devices data to cloud services. IoTCloud consists of a set of distributed nodes running close to the devices to gather data; a set of publish-subscribe brokers to relay the information to the cloud services and a distributed stream processing framework (DSPF) coupled with batch processing engines in the cloud to process the data and return (control) information to the IoT devices. Real time applications execute data analytics at the DSPF layer achieving streaming real-time processing. The IoTCloud platform uses Apache Storm[[10](#_ENREF_10)] as the DSPF, RabbitMQ[[11](#_ENREF_11)] or Kafka[[12](#_ENREF_12)] as the message broker and an OpenStack academic cloud[[13](#_ENREF_13)] (or bare-metal cluster) as the platform. To scale the applications with number of devices we need distributed coordination among parallel tasks and discovery of devices; both achieved with a ZooKeeper[[14](#_ENREF_14)] based coordination and discovery service.

To connect a device to the cloud services, a user must develop a gateway application that connects to the device’s data stream. Underlying details of the communication between the gateway and the cloud services is abstracted and a simple API to send and receive data is provided to the gateway application. The real time applications are developed at the streaming layer according to the API’s provided by the DSPF. Common functions of a gateway application includes transformation, filtering and enrichment of data between the cloud services and the robots. IoTCloud doesn’t enforce a communication channel between the device and the gateway. The dataflow between the application and the device can happen via TCP, device specific message protocols, message brokers etc and it is the responsibility of the user to implement necessary mechanisms to connect to the robot. Once an application is deployed in an IoTCloud gateway the cloud applications can discover those applications and connect to them for data processing using the discovery service.

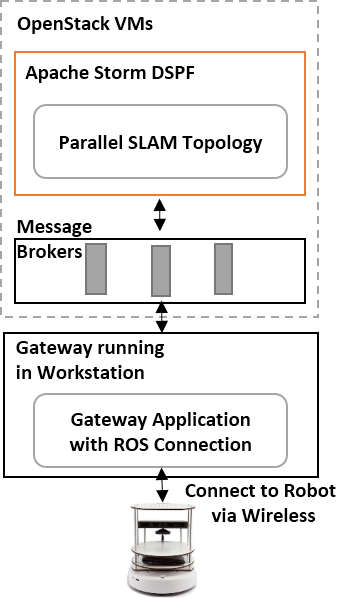


**Figure 1 IoTCloud Architecture**

## Design of Robot Applications

The parallel implementation of the algorithm is generic and can be used to build a map for a mobile robot with a laser scan traversing an unknown area. To validate the algorithm’s practical use we have developed an application to connect to the TurtleBot[[15](#_ENREF_15)] robot by Willow Garage. TurtleBot is an off the shelf differential drive robot equipped with a Microsoft Kinect sensor. The Microsoft Kinect consists of an IR camera, an RGB camera, an IR emitter, and several auxiliary features and can produce distance measurements to objects in front of it functioning like a laser scanner but with less accuracy. TurtleBot has a ROS[[16](#_ENREF_16)] driver and a supporting software stack; which can be used to retrieve information such as odometry, laser scans from the robot as well as controlling the movement of the robot.

The application that connects to the ROS based API of the robot is deployed in an IoTCloud Gateway running in a desktop machine. The application subscribes to laser scans coming from the IR sensor of the Kinect and odometry readings of the Turtlebot. It converts the ROS messages to a format that suits the cloud application and sends transformed data to the application running in the FutureGrid OpenStack[[13](#_ENREF_13)] based VMs using the message brokering layer. The correlation between the odometry readings and the laser scans are done at the gateway to reduce the complexity of the cloud application and to keep it generic. This is a nice example of balancing the computation between cloud and robot’s local computers. The application running in the cloud generates a map according to the information it receives and sends this map back to the workstation running the Gateway. The gateway saves this map and publishes it back to ROS for viewing.

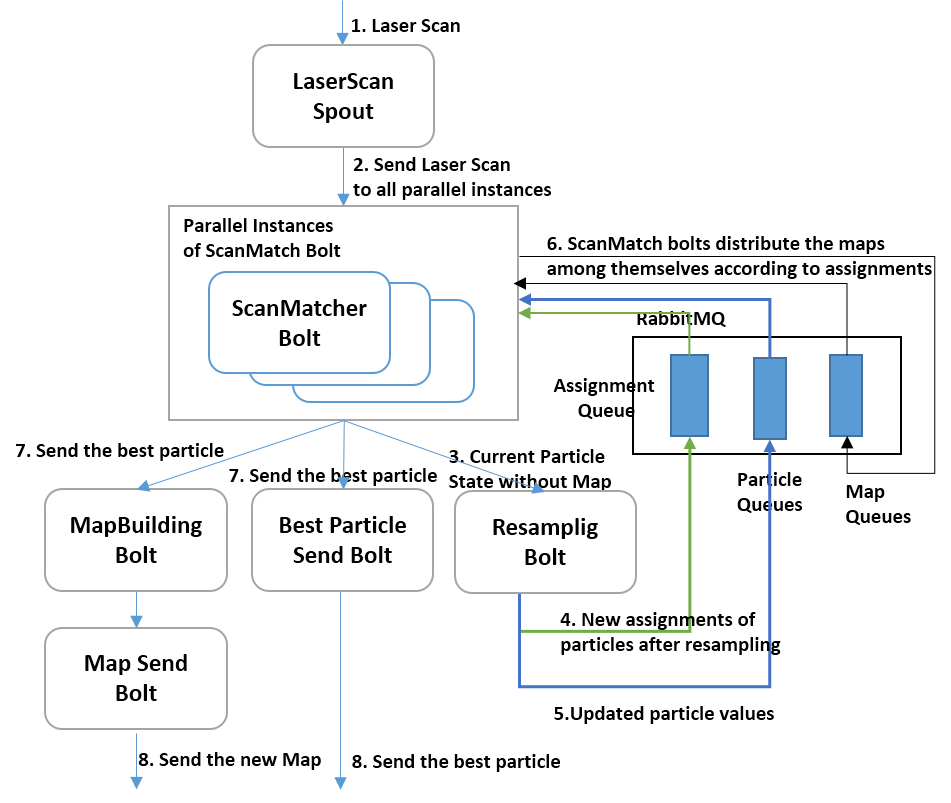


**Figure 2 Turtlebot Application**

## Algorithm Design

Open source serial version of the algorithm implemented in C++ language is available through the OpenSlam.org. Because of the C++ implementation, this algorithms is not suitable for our platform, which mainly focuses on Java based applications.  This has being identified as a shortcoming of our platform because there are many device related algorithms written in C/C++ and we would like to address this in the future. A parallel version of the algorithm was implemented in Java with the API provided by the DSPF.

The most expensive computation of the algorithm happens over the particles. For each pair of laser and odometry readings the algorithms accepts to process, it loops through the particle list and does an expensive computation. The parallel algorithm distributes this expensive computation across a set of tasks running across a cluster in parallel. To achieve this distributed computation the total number of particles are distributed among the parallel tasks and each laser scan-odometry pair is sent to all these parallel workers. After this parallel computation step, a global weighting and resampling process has to be done for involving all the particles. Algorithm gathers the results of the parallel computation into a single node and does this global computation in a single node, which is order of magnitude less computationally expensive than the parallel computations. After the global computation is finished the states of the particles changes and some particles are being eliminated and replaced by existing particles. To facilitate this step the task that does the global calculation sends the results back to the parallel tasks. To reduce the overhead of the communication, the algorithm doesn’t send the maps associated with each particle to the task that does the resampling. Because of this after the parallel tasks receives the updated particles, they have to distribute the maps to correct tasks.



**Figure 3 Storm Streaming Work Flow for Parallel RBPF SLAM**

The algorithm is implemented as an Apache Storm topology. The topology defines the data flow DAG for the application with Java based tasks implementations at the nodes and communication links defining the edges. The resampling step of the algorithm requires the downstream tasks to update the upstream tasks, which is not supported by Apache Storm. Algorithm uses an external RabbitMQ message broker to create these upstream communication channels in the topology DAG. The data flow of the algorithm is shown in Figure 3 and the data flow steps are describe below. All the data flowing through the various communication channels are in byte format and algorithm uses Kryo to serialize the objects to bytes.

1. Laser scans and odometry readings are received by the LaserScan spout. The spout receives the messages through the message broker layer and it discovers communication channels using the ZooKeeper based discovery service.

Each message contains a laser scan and a corresponding odometry reading in byte format.

2. The laser scan is distributed to ScanMatch bolt running parallel task instances. Each parallel task does the calculations with the particles assigned. Each task loops through the assigned particles and does the computation serially.

3. The ScanMatch bolt sends the updated particle values to the Resampling bolt. Only one instance of the resampling bolt is executed. The implementation doesn’t send the maps associated with the particles to the resampling bolt because resampling bolt doesn’t need the map to do its computation and maps can be large objects depending on the world size.

4. After resampling the resampling bolt calculates new particle assignments to the ScanMatch bolts. This reassignment is done considering the old assignment and relocating cost using the Hungarian algorithm. This new global particle assignment is broadcast to all the task instances of the ScanMatch bolts using an external RabbitMQ topic.

5. Simultaneously the resampling bolt sends the resampled particle values to their new destinations according to the assignment. This also uses RabbitMQ queues to directly send the messages to the tasks. A task is identified by an id and this id is used as a routing key in the messages.

6. After the parallel tasks of ScanMatch bolt receive the new assignment, they distribute the maps associated with the resampled particles to their correct destination. All the task instances of the ScanMatch bolts does this simultaneously.

7. The ScanMatch bolt with the best valued particle sends its values and the map to a bolt that builds a renderable map from the map used by the particle. Also it will send the best particle to another bolt that will directly send the map out to the gateways.

8. At last the built map and the best particle will be sent to the gateway application.

This algorithm doesn’t require all the laser scan readings from the robot to compute the map correctly and can lose some of the messages. The parallel algorithm exploits this feature and drops the messages that are coming in between the computations to avoid memory overflow of the system. The original serial algorithm runs every 5 seconds for the Turtlebot map building and we can run much faster than that speed allowing robot to move faster. Because of the design of the GMapping algorithm only few resampling steps are needed during a map building. This reduces the number of times the algorithm distributing particle maps among the tasks. Nevertheless we need the gathering step at Resampling Bolt after each parallel computation to calculate the weights and determine the best particle at that time.

## Results & Discussion

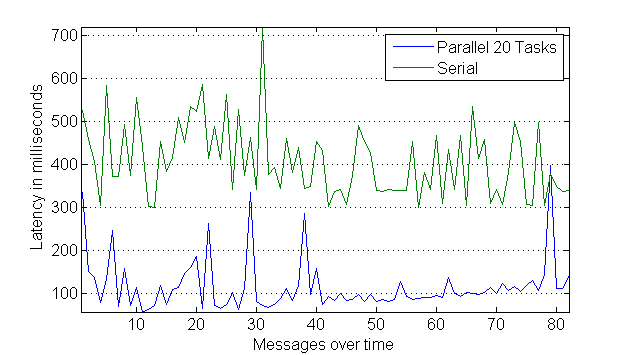
The goal of our experiments was to verify the correctness of our approach and its practical use as well as measure the scalability of the algorithm. We conducted experiments with the real robot, a robot simulator as well as a SLAM benchmark dataset. All the experiments were running in FutureGrid[[13](#_ENREF_13)] OpenStack VMs. The OpenStack experiments used 5 large VM instances for Apache Storm Workers, 1 large instance for RabbitMQ message broker and 1 large instance for ZooKeeper and Storm master (Nimbus) node. A FugureGrid Large instance VM has 8GB memory and 4 CPU cores running at 2.8 GHz. For all the tests the gateway node was running in a desktop machine inside the Indiana University network. Each instance of the Storm worker nodes runs 4 Storm worker processes with 1.5GB of memory allocated.

|  |  |
| --- | --- |
| **Figure 4*a* Aces Building RBPF Computation Speed Up for 60 Particles** | **Figure 4b Simbad Speed up of RBPF Computation for 20, 60 and 100 Particles with varying number of parallel tasks** |

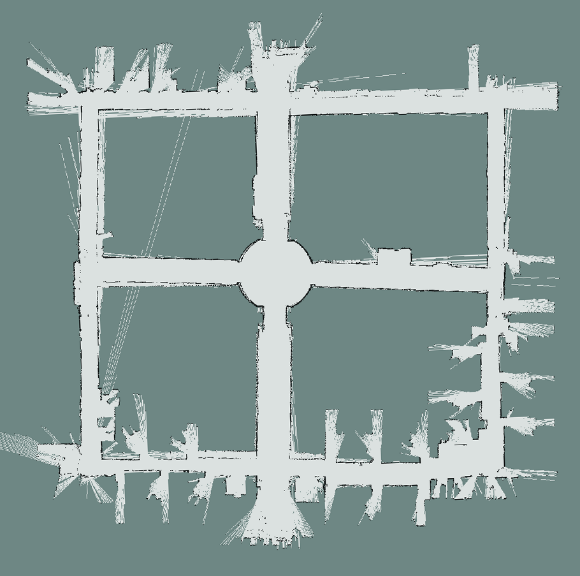
First we verified the accuracy of the algorithm using the ACES building SLAM benchmark data set described in [11]. We used the ROS rviz to visualize the maps that are being built by the application. The obtained map is shown in the figure 6. The speedup of the algorithm was measured by recording the time required to compute a single laser reading and averaging out these times over a set of laser readings. To measure the parallel speedup we first measured the time it takes for the serial version using the same number of particles in a FutureGrid Large VM instance. Then we measured the averaged latency of the parallel implementation for the same number of particles. For the ACES building data set we tested with 60 particles by varying the parallel tasks from 4 to 20. The speedup is shown in the Figure 5a. The ACES building laser readings has 180 distance measurements per laser reading. Because the number of laser reading are relatively low the computation at the ScanMatching bolts are relatively less, making the increase in speedup low after 12 particles. Then we used a robot simulator called Simbad[[17](#_ENREF_17)] to create a small test environment and recorded the laser reading of a simple robot moving in that environment. In the Simbad robot each laser reading has 640 distance measurements which is greater than the 180 measurements in the SLAM benchmark laser readings. We conducted experiments for 20, 60 and 100 particles. The speedup is shown in Figure 4b and it can be seen that the gain is much higher for this experiment compared to the first experiment because of the increased computation required in the Scan Matching parallel tasks. The results clearly shows that we can gain considerable speedups when the complexity of the algorithm increases.

Each particle carries a map of the environment and when the number of particles increases these maps can take a considerable amount of memory for maps requiring more memory due to large map sizes or higher resulutions. We observed this problem when running the serial version of the algorithm with higher number of particles and had to increase the memory of the application to considerably higher levels. The distributed algorithm doesn’t have this problem because only a subset of the particles are in a node. Also the worker model of the DSPF makes the particles in a physical machine to be distributed among several processes with small memory footprint, further reducing the overhead in managing large memories.

Latency measurements in both experiments are subjected to the variations in the network delays between the node generating the data and the cloud. This network delay is relatively small (less than 10ms) but nonexistence in the serial version and hence the actual parallel speedup of the algorithm should be slightly higher than what is shown in the graphs. One observation we had while running this application was that there are sudden variations in the individual laser scan processing times as shown in Figure 5. Both serial version and the parallel version latency variations are shown for ACES dataset with 60 particles. For SLAM applications this is not a big concern but there are applications that can affect from such variations. Note that the two lines in the Figure 5 have minimal correlation between them, due to the un-deterministic nature of the algorithm.

The algorithm doesn’t use guaranteed message processing features of the DSPF allowing it to run with the lowest latency possible through the DSPF. Also there is no coordination among the parallel tasks for each parallel computation of a laser reading. If a laser reading processing fails at a parallel task the only way algorithm can recover is using timeouts. 

**Figure 5 Latency variation for each individual laser reading of RBPF computation over time for 60 Particles with 20 task parallel and serial versions**

Because the parallel algorithm runs much faster than the serial version of the algorithm, it can be used to build a map of a fast moving robot. Also the accuracy of the maps built are increased because of the increased number of readings the algorithm is able to use for calculations. One of the biggest challenges in the particle filtering based methods is the time required for the computation increases with the number of particles. A higher number of particles generally means increased accuracy for the algorithm. By distributing the particles across machines application can utilize high number of particles, improving the accuracy of the algorithms.

**Figure 6 Aces building map built with 30 Particles, Angular Update 0.5 and Linear Update 1.0**

## Conclusion

In this paper we discussed how to develop distributed parallel robotics applications in the cloud using a generic framework. The results show some significant improvements in the performance gains and the system can be extended for many such applications.

At the moment it is still difficult to develop and scale IoT applications with the modern distributed stream processing engines due to the complex programming required. More robust frameworks with support to such applications can be really beneficial for application developers. The parallelization of an application can be embedded into the programming model rather than asking using to program the parallelization by themselves; which can take a considerable effort. Our work has identified difficulties in meeting real time constraints in cloud controlled IoT due to either the intrinsic time needed to process events or due to fluctuations in processing time caused by virtualization, multi-stream interference and messaging fluctuations. We would like to address these fluctuations in the computation time in the future. One possible approach is to use duplicate computation to avoid the random fluctuations at the cost of more resource utilization. Having efficient duplication of computation can be a challenging task. Another important area to focus on the scalability is how to schedule the tasks in a dynamic environment of robots. As the robots connect and disconnect from the system the application resources must be rescheduled to get the optimum performance out of the system. Having such dynamic resource scheduling is difficult because the applications keep the state in the memory. Approaches like distributed in memory key value stores can be used to keep the state so that applications can be migrated to different computation nodes at runtime.

The algorithm implementation is specific to SLAM but the methods used can be easily generalized to any particle filtering algorithm. Extending this work to extract out a generic API to develop any particle filtering distributed parallel algorithm can be a worthy experiment.

## References

1. Grisetti, G., C. Stachniss, and W. Burgard. *Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling*. in *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*. 2005. IEEE.

2. Grisetti, G., C. Stachniss, and W. Burgard, *Improved techniques for grid mapping with rao-blackwellized particle filters.* Robotics, IEEE Transactions on, 2007. **23**(1): p. 34-46.

3. Kamburugamuve, S., et al., *Survey of Distributed Stream Processing for Large Stream Sources.*

4. Chitchian, M., et al., *Particle filters on multi-core processors.* Dept. Comput. Sci., Delft Univ. Technology, Delft, The Netherlands, Tech. Rep. PDS-2012-001,(Feb. 2012)[Online]. Available: <http://www>. pds. ewi. tudelft. nl/fileadmin/pds/reports/2012/PDS-2012-001. pdf. Code available at: https://github. com/alxames/esthera, 2012.

5. Gouveia, B.D., D. Portugal, and L. Marques. *Speeding up rao-blackwellized particle filter SLAM with a multithreaded architecture*. in *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*. 2014. IEEE.

6. Riazuelo, L., J. Civera, and J. Montiel, *C 2 TAM: A Cloud framework for cooperative tracking and mapping.* Robotics and Autonomous Systems, 2014. **62**(4): p. 401-413.

7. Zhang, H. and F. Martin. *CUDA accelerated robot localization and mapping*. in *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*. 2013. IEEE.

8. Gao, X., E. Ferrara, and J. Qiu, *Parallel Clustering of High-Dimensional Social Media Data Streams.*

9. Community Grids Lab, I.U. *IoTCloud*. 2015; Available from: <http://iotcloud.github.io/>.

10. Anderson, Q., *Storm Real-time Processing Cookbook*. 2013: Packt Publishing Ltd.

11. Videla, A. and J.J. Williams, *RabbitMQ in action*. 2012: Manning.

12. Kreps, J., N. Narkhede, and J. Rao. *Kafka: A distributed messaging system for log processing*. in *Proceedings of the NetDB*. 2011.

13. Fox, G., et al., *FutureGrid—A reconfigurable testbed for Cloud, HPC and Grid Computing.* Contemporary High Performance Computing: From Petascale toward Exascale, Computational Science. Chapman and Hall/CRC, 2013.

14. Hunt, P., et al. *ZooKeeper: Wait-free Coordination for Internet-scale Systems*. in *USENIX Annual Technical Conference*. 2010.

15. Garage, W., *TurtleBot.* Website: <http://turtlebot>. com/last visited, 2011: p. 11-25.

16. Quigley, M., et al. *ROS: an open-source Robot Operating System*. in *ICRA workshop on open source software*. 2009.

17. Hugues, L. and N. Bredeche, *Simbad: an autonomous robot simulation package for education and research*, in *From Animals to Animats 9*. 2006, Springer. p. 831-842.