Streaming Parallel Implementation of Rao-Blackwellized Particle Filtering SLAM in the Cloud for Mobile Robots

Supun Kamburugamuve, Hengjing He, Geoffrey Fox, David Crandall

School of Informatics and Computing, Indiana University, Bloomington, USA

Abstract. The Simultaneous Localization and Mapping (SLAM) for a mobile robot is a computationally expensive task. A robot capable of SLAM needs a powerful onboard computer which can limit its mobility because of the weight and power consumption. In this paper we propose a cloud-based distributed architecture for real time computations of robots and implement a Rao-Blackwellized Particle Filtering-based SLAM algorithm in a multi-node cluster in the cloud. With our distributed implementation, we obtained significant efficiency improvements in computation time. This allows the algorithm to increase its complexity and frequency of calculations which are factors for enhancing the accuracy of the maps built. The onboard computer of the robot is free to do other tasks while offloading the heavy computation to the cloud. Our method for implementing particle filtering in the cloud is not specific to SLAM and can be applied to other computationally intensive particle filtering algorithms.

1 Introduction

Cloud Computing has long being identified as a key enabling technology for Internet of Things applications, which connect everything ranging from such simple devices as thermostats to complex industrial machinery, robots to the Internet. Cloud services are used by IoT applications for doing large scale offline and real time analytics on the data produced by these devices. There are computationally intensive algorithms for processing device data that can benefit from powerful resources in the cloud for real time response. The methods used by these computationally expensive algorithms are powerful, but hard to run near the devices due to high computational and specialized hardware requirements. At the same time these applications have to be scaled to support vast numbers of devices and are inherently suitable for central data processing. This paper investigates a computationally expensive robotics application to showcase a means of achieving complex parallelism for real time applications in the cloud.

Parallel implementations of real time robotics algorithms mostly deal with 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 these robotics applications requiring low latencies. Also these applications can be scaled up and down depending on the processing requirements, making clouds a cost effective solution.

Simultaneous Localization and Mapping (SLAM) is an important capability for mobile robots that has been studied extensively in the relevant literature. Computing the position of a robot in an unknown environment amidst measurement errors while 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. We have chosen a popular SLAM algorithm called GMapping to implement in the cloud. GMapping builds a grid map and uses distance measurements from a laser range finder and odometer measurements of the robot for its calculations. It is a Rao-Blackwellized Particle Filtering(RBPF) based SLAM algorithm [7] [8]. It is known to work well in practice and has been integrated into robots like TurtleBot. The algorithm is computationally expensive and can produce better results using more resources.

IoTCloud [4] is a framework which can transfer data from devices to a cloud computing environment for scalable data processing with real time response. The data from the devices is encapsulated into events and sent to cloud systems in real time. IoTCloud employs a distributed stream processing framework (DSPF) [10] for developing and executing scalable real time applications in the cloud. We have implemented the GMappning algorithm to work in the cloud on top of the IoTCloud platform. Laser scans and odometer readings are sent from the robot to the cloud as a stream of events where they are processed by the SLAM application and results are sent back to the robot immediately. The algorithm runs in a fully distributed environment where different parts of it is run in different machine. To reduce the time required, the most expensive computation of the algorithm is run in parallel in a distributed set of nodes.

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

2 Related Work

To the best of our knowledge, using distributed cloud infrastructure to execute particle filtering-based SLAM algorithms has not been studied in the literature. Recent work in[4] has exploited multicore and GPU architectures to speed up the particle filtering-based computations and [3] has used multicore architecture to create a 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 the algorithm. [13] is a framework developed to move some of the expensive computations of a SLAM algorithm into a cloud environment for processing. The SLAM algorithm in C2TAM is different from the version used in this work and has different computation requirements. Also our work proposes a generic scalable real time framework for computing the maps online with significant gains in the processing time. C2TAM does not provide such a framework. Zhang et al [15] describe 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 been deployed for tasks like clustering social data in stream [6] with excellent performance enhancements. The algorithm we developed is different from those implementations because of the nature of the parallelism and the real time constraints. Those applications are mostly data parallel, whereas we focus on a computationally parallel application.

3 Background

3.1 IoTCloud framework

IoTCloud [4] is an open source framework developed at Indiana University to connect IoT devices to cloud services. It 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 [2] as the DSPF, RabbitMQ [14] or Kafka [11] as the message broker and an OpenStack academic cloud [5] (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 are achieved with a ZooKeeper [9] based coordination and discovery service.

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 produce output streams. A DSPF provides the necessary API and infrastructure to develop and execute such applications in a cluster of computation nodes. In general DSPF allows the same task to be executed in parallel and provides rich communication channels among the tasks. To connect a device to the cloud services, a user develops a gateway application that connects to the devices data stream. Underlying details of the communication between the gateway and the cloud services is abstracted and a simple API is provided to communicate with cloud applications. 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.

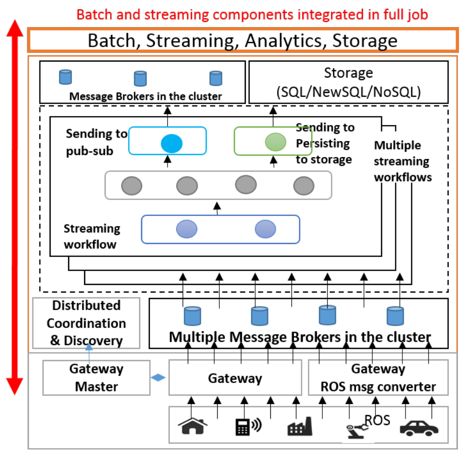

Batch and streaming components integrated in full job
Batch, Streaming, Analytics, Storage
Message Brokers in the cluster
Storage (SQL/NewSQL/NoSQL)
Sending to pub-sub
Sending to Persisting to storage
Multiple streaming workflows
Streaming workflow
Distributed Coordination & Discovery
Multiple Message Brokers in the cluster
Gateway Master
Gateway
Gateway ROS msg converter
ROS

Fig. 1: IoTCloud Architecture

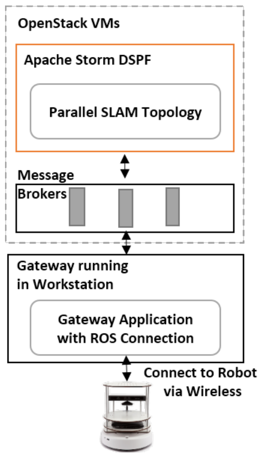

OpenStack VMs
Apache Storm DSPF
Parallel SLAM Topology
Message Brokers
Gateway running in Workstation
Gateway Application with ROS Connection
Connect to Robot via Wireless

Fig. 2: GMapping Robotics application

3.2 Design of GMapping application

The design of the GMapping application with connection to a real robot is shown in Figure 2. In this end-to-end application we have connected TurtleBot [1] by Willow Garage to the GMapping algorithm running in the cloud using the IOT-Cloud platform. TurtleBot is an off-the-shelf differential drive robot equipped with a Microsoft Kinect sensor. It has a ROS [12] driver and a supporting software stack which can be used to retrieve information such as odometer and laser scans, as well as controlling its movement.

The application that connects to the ROS-based API of the robot is deployed in an IoTCloud Gateway running in a desktop machine, where it subscribes to laser scans coming from the IR sensor of the Kinect and odometer 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 [5] VMs using the message brokering layer. The application running in the cloud generates a map according to the information it receives and sends this back to the workstation running the Gateway, which saves and publishes it back to ROS for viewing.

3.3 RBPF SLAM Algorithm

As described in [7] [8] Rao-Blacwellized particle filter for SLAM is estimating the posterior $p(x_{1:t}, m|z_{1:t}, u_{1:t-1})$ where $x_{1:t} = x_1, ..., x_t$ is the trajectory of the robot and m is the map. $z_{1:t} = z_1, ..., z_t$ are the laser readings observed and are the odometer measurements.

$$p(x_{1:t}, m|z_{1:t}, u_{1:t-1}) = p(x_{1:t}, m|z_{1:t}, u_{1:t-1}) \tag{1}$$

The above factorization first estimates the position of the robot given the observations, and then calculates the map using the trajectory of the robot. Map calculation can be done efficiently if the trajectory is known. To estimate the position of the robot over possible trajectories, it uses a particle filter. The particle filter maintains a set of particles, with each one containing a probable map of the environment and a possible trajectory of the robot. The map associated with the particle is built using the robots trajectory associated with the particle and the laser readings observed. To calculate the trajectory of the robot a new reading z_t, u_{t-1} is used. A standard implementation of the algorithm executes the following steps for each particle i using that its information:

1. Make an initial guess $x_t^{'i} = x_{t-1}^{'i} \oplus u_{t-1}$, where $\oplus$ is standard pose compounding operator. The algorithm incorporates the motion model parameters of the robot when calculating the initial guess.
2. Use the ScanMatching algorithm shown in Algorithm 1 with cutoff of ∞ to optimize initial guess $x_t^{'i}$ using the map m_{t-1}^i and laser reading z_t. If the ScanMatching fails, use the previous guess.
3. Update the weight of the particle
4. The map m_t^i of the particle is updated with the new position x_t^i and z_t.

After updating each particle, the algorithm calculates $N_{eff} = \frac{1}{\sum_{i=1}^{n}(w^{(i)})^2}$ using the weight of each particle and does resampling according to the calculated value. When resampling happens the algorithm draws particles with replacements from the set according to their weights. 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.

4 Streaming parallel algorithm design

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 independently of the others. Because the computation on a particle is independent of other particles, this algorithm is well suited for parallel execution. In a distributed environment the particles can be moved to different computation nodes

```
input  : pose $u$ and laser reading $z$
output: $bestPose$ and $l$
1  $steps \leftarrow 0$; $l \leftarrow -\infty$; $bestPose \leftarrow u$; $delta \leftarrow InitDelta$;
2  $currentL \leftarrow$ likelihood($u, z$);
3  for $i \leftarrow 1$ to $n_refinements$ do
4      $delta \leftarrow delta/2$; $pose \leftarrow bestPose$; $l \leftarrow currentL$;
5      repeat
6          for $d \leftarrow 1$ to $K$ do
7              $xd \leftarrow$ deterministicsample($pose, delta$);
8              $localL \leftarrow$ likelihood($xd, z$);
9              $steps+ = 1$;
10             if $currentL < localL$ then
11                 $currentL \leftarrow localL$; $bestPose \leftarrow xd$;
12             end
13         end
14     until $l < currentL$ and $steps < cutoff$;
15 end
```

Algorithm 1: Scan Matching

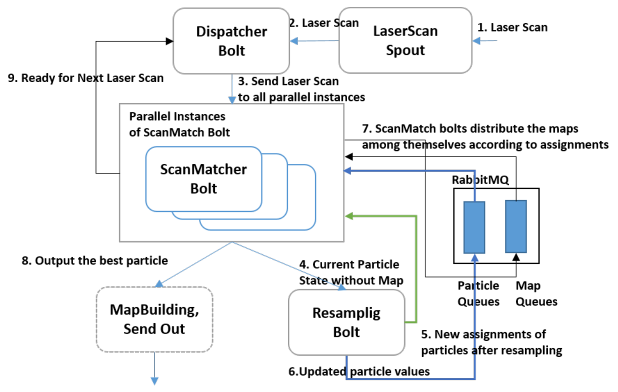

Dispatcher Bolt
2. Laser Scan
LaserScan Spout
1. Laser Scan
9. Ready for Next Laser Scan
3. Send Laser Scan to all parallel instances
Parallel Instances of ScanMatch Bolt
ScanMatcher Bolt
7. ScanMatch bolts distribute the maps among themselves according to assignments
RabbitMQ
Particle Queues
Map Queues
8. Output the best particle
4. Current Particle State without Map
MapBuilding, Send Out
Resamplig Bolt
5. New assignments of particles after resampling
6.Updated particle values

Fig. 3: Storm Streaming Work Flow for Parallel RBPF SLAM

and computation on particles can be executed in parallel. Even though the Scan-Matching computations can be easily made parallel, resampling (which requires information about all the particles) needs to be executed serially and must gather results from the parallel execution of particles. The resampling removes some of the existing particles and duplicates them in the system. Because of this, some of the particles have to be redistributed over the cluster after resampling.

The stream workflow of the algorithm is shown in Figure 3 implemented as an Apache Storm topology. The topology defines the data flow graph of the application with Java-based task implementations at the nodes and communication links defining the edges. The different components of this workflow run in a cluster of nodes in the cloud. The arrows in the diagram show the communication between these components as it occurs through TCP. As we can see, the main tasks of the algorithm are divided into ScanMatcherBolt, ReSamplingBolt and MapBuilding bolt. The LaserScanBolt receives the data from the robot and sends it to the rest of the application. The BestParticleSend bolt and MapSend bolt send the results back to the robot. The MapBuilding bolt builds a renderable map expected by the robot and is not a part of the core algorithm.

A key idea behind our distributed implementation is to scatter the particles across a set of tasks running in parallel across a cluster of nodes and do the expensive Scan Matching operation in parallel. This particle specific code (steps 1, 2, 3 and 4 of the algorithm) is encapsulated in the ScanMatcher bolt of the workflow and we can configure how many instances of that bolt are running in parallel which defines the parallelism of the algorithm. The resampling bolt requires the result of the ScanMatcher bolts running in parallel, so it waits until it receives them from the ScanMatcher bolts. After a resampling happens, algorithm can remove some existing particles and duplicate others. Because of this the particles assigned to ScanMatcher tasks have to be rearranged after a resample. The directed communication required among the parallel ScanMatcher tasks to do the reassignment is not well supported by Apache Storm, so we use an external RabbitMQ message broker for such communications. All the data flowing through the various communication channels are in byte format and the algorithm uses Kryo to serialize the objects to bytes. The dataflow steps as shown in Figure 3 are described below.

1. Laser scans and odometry readings are received by the LaserScan spout through the message broker layer. 2. Laser scans are sent to a Dispatcher bolt that controls the parallel algorithm. The bolt broadcasts the same Laser Scan to the parallel tasks. When the parallel ScanMatcher tasks complete, they send a message confirming this back to the Dispatcher bolt, which sends the next reading to the parallel tasks. It always uses the latest readings and drops the readings it receives while the parallel tasks are running. 3. Each parallel ScanMatcher task receives the same laser reading and does calculations with the particles assigned. After this it sends the updated particle values to the Resampling bolt. 5. After resampling, the resampling bolt calculates new particle assignments to the ScanMatcher bolts. This reassignment is done considering the old assignments and relocating costs using the Hungarian algorithm. Afterwards new particle assign-

ment is broadcast to all the ScanMatcher instances. 6. In parallel to Step 5, the resampling bolt sends the resampled particle values to their new destinations according to the assignment. This 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. 7. After the parallel tasks of ScanMatcher bolt receive the new assignment, they distribute the maps associated with the resampled particles to their correct destination. All the task instances of the ScanMatcher bolts do this simultaneously. 8. The ScanMatcher bolt with the best valued particle outputs its values and the map. 9. ScanMatcher bolts send messages indicating their willingness to accept the next reading to the dispatcher bolt.

The parallel version exploits the algorithms ability to loose readings and drops the messages at a Dispatcher bolt that are coming while a computation is ongoing to avoid memory overflow of the system. Owing to the design of the GMapping algorithm, only a few resampling steps are needed during map building. This reduces the number of times the algorithm has to distribute particle maps among the tasks. An open source serial version of the algorithm implemented in C++ language is available through OpenSlam.org. This implementation is not suitable for our platform, which mainly focuses on Java-based applications. The algorithm described above was implemented in Java with the API provided by the DSPF.

5 Results and Discussion

The goal of our experiments was to verify the correctness of our approach and its practical use in addition to measuring the scalability of the algorithm. We conducted experiments with the real robot and a robot simulator as well as a SLAM benchmark dataset. The experiments with the real robot were conducted in small indoor environments and the results are not shown here. All the experiments ran in [5] 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 FutureGrid 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 another large instance VM of FutureGrid. Each instance of the Storm worker nodes runs 4 Storm worker processes with 1.5GB of memory allocated.

To verify the accuracy of the algorithm, we use the ACES building SLAM benchmark data set described in [11]. We used the ROS rviz to visualize the maps being built by the application. The obtained map is shown in Figure 9. GMapping is a well-known and well-tested algorithm. We did not try to extensively verify the accuracy of the algorithm on different datasets due to that fact and instead focused on the parallel behavior of the algorithm in our experiments. Parallel speedup of an algorithm is defined as (Time Serial Algorithm takes)/(Time parallel algorithm takes), i.e Ts/Tp. The speedup of the algorithm was measured by recording the time required to compute each laser reading and getting the average of these individual times. For ACES data set we use a map of size 80x80m with a .05 resolution and for Simbad the map was 30x30m with .05 resolution. We tested the algorithm with 20, 60 and 100 particles for both data sets. The serial time was measured in a FutureGrid machine that we used for running the parallel version. In the DSPF cluster we had 5 worker nodes with total 20 CPUs, hence each worker utilized a single CPU core. To test the parallel behavior of the algorithm we used 4, 8, 12, 16 and 20 parallel tasks.

The parallel speedup gained for ACES building dataset and Simbad dataset is shown Figure 4. For ACES, the number of laser reading are relatively low and because of this computation at the ScanMatcher bolts is correspondingly less, making the increase in speedup low after 12 particles. On the other hand, Simbad dataset has about 4 times more distance measurements per reading and produces higher speed gains.

Ideally the parallel speedup should be close to 20 when we have 20 parallel tasks and we investigated any factors that could drag the speedup down. Only the scan matching step of the algorithm is executed in parallel; the resampling step is done serially. Because this serial computation is relatively less expensive than the Scan Matching computation, the speedup loss is not significant at this step. The main factor for reducing the distributed parallel computations is the I/O time. I/O time is not present in the serial version of the algorithm and is a totally new addition to the computation time. Because our computation is done in Java, Java garbage collection also can have an effect on the performance. Figure 5a shows the I/O time, GC time and computation time for different parallel tasks and particle sizes. As clearly seen from these results, there is a nearly a constant average I/O overhead in the parallel algorithm. When the number of parallel tasks increases, the time decreases, and because of the I/O overhead the speedup reduces. The average GC time was negligible but we have seen instances where it increases the individual computation times.

Another factor that affects parallel computation is the computation time difference among parallel tasks. Lets assume we have n parallel ScanMatcher tasks taking $t_1, ..., t_m, ..., t_n$ times and take t_m as the maximum time among those times. In the serial case the total time for Scan Matching procedure will be $t_1 + ... + t_m + ... + t_n$. For the parallel case the time will be t_m because the Resampling has to wait for all the parallel tasks to complete. The ideal case for parallel is when all the times in parallel tasks are equal. The overhead introduced because of the difference in times will be $t_{overhead} = t_m - (t_1 + ... + t_m + ... + t_n)/n$. When the difference between the maximum time and average time increases, parallel overhead increases. Figure 5b shows the average overhead calculated for the Simbad dataset against the total time. The calculations are done for cases where particles are distributed equally among the parallel tasks. The average overhead remained constant and the total time decreases as parallel tasks increase, producing less speedup.

To further investigate the behavior of the algorithm we drew the individual times as shown in Figure 6a and 6b. There are high peaks in the individual times in both serial and parallel algorithms. The while loop ending in line 18 of Algorithm 1 can execute an arbitrary number of steps. Sometimes this results in large loops compared to the average. Figure 5c shows the average steps count and

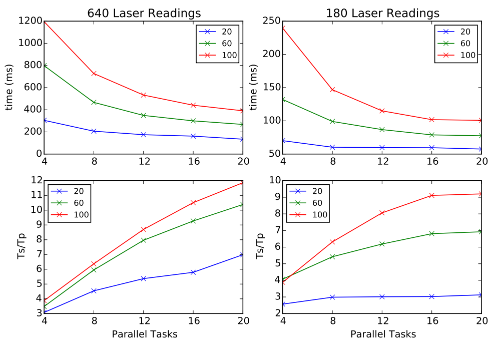
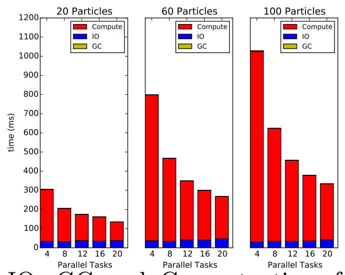
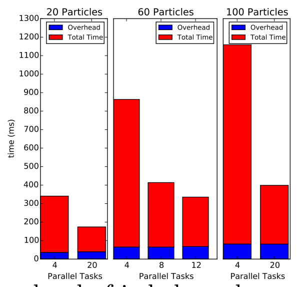


Fig. 4: Parallel behavior of the algorithm for 180 and 640 laser readings. The two graphs at the top show the actual time and bottom graphs show the speedup



(a) IO, GC and Compute time for 640 readings



(b) Overhead of imbalanced parallel computation

Fig. 5: Overheads with random variations in parallel tasks

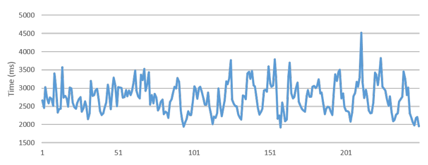
standard deviation of steps executed by the ScanMatching algorithm for Simbad dataset. The standard deviation can be large and sometimes we have seen 2 to 3 times more steps than the average. This is especially problematic for the parallel case because one or two particles can significantly increase the response time. Since we have a large number of particles, cutting off the optimization for one or two of them prematurely shouldnt affect the algorithm. Also we can easily increase the number of particles if needed to compensate for the premature cutoff of the optimization in the parallel case. Another observation was that these large numbers of steps occur at later refinements in the ScanMatching algorithm with small delta values. So the corrections gained executing many loops is minimal in most cases. Considering these factors, we changed the original algorithm as shown in Algorithm 1 with a configurable cutoff for the number of steps and performed experiments by setting the max number of steps to 140, which is close to the average. The changed ScanMatching algorithm is shown in Algorithm . Any maps built by the algorithm were of comparable quality to the previous algorithm. The resulting time variations for two tests are shown in Figure 6c and 6d. Now we no longer see some of the big peaks and variations we saw in Figure 6b. The relatively high peaks are due to minor garbage collections occurring. Figure 7a shows the average time reduction and speedup after the cutoff. As expected, we see an improvement in speedup as well, because the parallel overhead is now reduced as shown in 7. This demonstrates that the cutoff is an important configuration parameter for parallel versions that can be tuned case by case to obtain optimum performance and correctness.

Even though the resampling only happens occasionally in the GMapping algorithm it can introduce a large overhead to the parallel algorithm because of the I/O requirements for redistributing the particles and the maps associated with them. Also the stream processing engines are not optimized for group communications required among the parallel tasks. In our case we were relying on an external broker. Figure 8 shows the difference in calculations when we conducted the resampling step for each operation with the Simbad dataset.

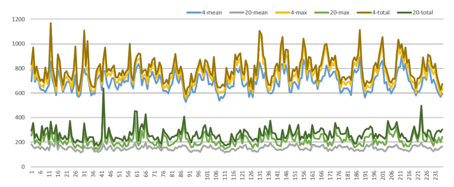
The original serial algorithm for Turtlebot runs every 5 seconds. Because the parallel algorithm runs much faster than the serial version, it can be used to build a map for a fast-moving robot. Also the accuracy of the maps built is increased due to the increased number of readings our algorithm is able to use for calculations. One of the biggest challenges in particle filtering-based methods is that time required for the computation increases with the number of particles. By distributing the particles across machines, an application can utilize a high number of particles, improving the accuracy of the algorithm.

6 Conclusion & Future Work

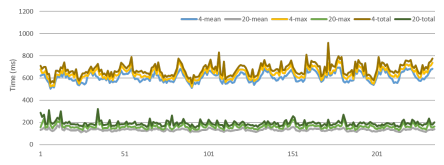
We discussed how to develop distributed parallel robotics applications in the cloud using a generic framework. The results show some significant improvements in the performance gains, and the system can be extended for many such applications. Because the algorithm runs on a distributed cloud infrastructure, it has access to a large amount of memory and CPU power. For map building in



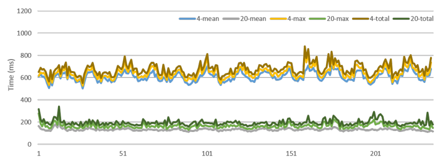
(a) Serial time



(b) Without cut-off

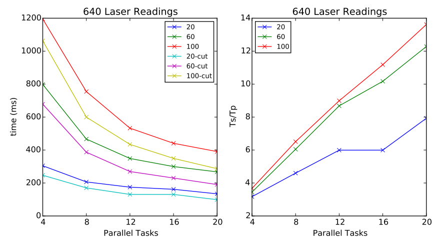


(c) Cutoff 140 steps

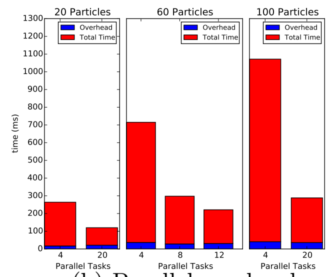


(d) Cutoff 140 steps

Fig. 6: Time variations of individual times for 60 particles with 4 and 20 parallel tasks, mean and max of parallel times along with total time is shown



(a) Parallel behavior of the algorithm for 640 laser readings with cutoff at 140



(b) Parallel overhead

Fig. 7: Overheads after cutoff at 140 steps

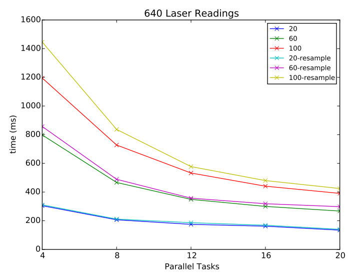


Fig. 8: Resampling overhead

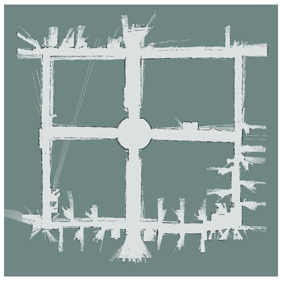


Fig. 9: Map of ACES

large environments where the algorithm needs an increased number of particles or for cases where robots have dense laser readings, the methods introduced in the paper can be used effectively. Also when the robot is equipped with a low end computer it can offload the SLAM to cloud using our method.

Random increases in the number of iterations for a particle can produce computation time imbalances between the parallel workers and reduce the response time, as well as hurt the overall parallel speedup. For this application we addressed the problem by introducing an upper bound to the number of steps, which works perfectly well in practice for this scenario. Another factor for speedup reduction is the I/O time. We have observed that the broadcast and gathering of results in the streaming tasks takes considerable I/O time.

Complex programming is required to develop and scale intricate IoT applications with modern distributed stream processing engines, mainly due to the low level APIs exposed. High level APIs are required in order to handle such complex interactions by abstracting out the underlying details. Our work has identified difficulties in meeting real time constraints in cloud controlled IoT due to the intrinsic time needed to process events or fluctuations in processing time caused by virtualization, multi-stream interference and messaging fluctuations. In the future we would like to address these fluctuations in computation time. Duplicate computations for such applications can be a more generic method that can be applied irrespective of the application at the expense of more resources. At the moment the state distribution between the parallel workers requires a third node, such as an external broker or another streaming task acting as an intermediary. A group communication API between the parallel tasks can be a worthy addition to a DSPF. 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 algorithm in a distributed environment can be a worthy experiment.

Acknowledgments

The authors would like to thank the Indiana University FutureGrid team for their support in setting up the system in FutureGrid NSF award OCI-0910812. This work was partially supported by AFOSR award FA9550-13-1-0225 Cloud-Based Perception and Control of Sensor Nets and Robot Swarms.

References

1. Turtlebot (2014), `http://wiki.ros.org/Robots/TurtleBot`
2. Anderson, Q.: Storm Real-time Processing Cookbook. Packt Publishing Ltd (2013)
3. Chitchian, M., van Amesfoort, A.S., Simonetto, A., Keviczky, T., Sips, H.J.: 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)

4. Community Grids Lab, I.U.: Iotcloud (2015), `http://iotcloud.github.io/`
5. Fox, G., von Laszewski, G., Diaz, J., Keahey, K., Fortes, J., Figueiredo, R., Smallen, S., Smith, W., Grimshaw, A.: Futuregrida reconfigurable testbed for cloud, hpc and grid computing. Contemporary High Performance Computing: From Petascale toward Exascale, Computational Science. Chapman and Hall/CRC (2013)
6. Gao, X., Ferrara, E., Qiu, J.: Parallel clustering of high-dimensional social media data streams
7. Grisetti, G., Stachniss, C., Burgard, W.: 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. pp. 2432–2437. IEEE
8. Grisetti, G., Stachniss, C., Burgard, W.: Improved techniques for grid mapping with rao-blackwellized particle filters. Robotics, IEEE Transactions on 23(1), 34–46 (2007)
9. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: Wait-free coordination for internet-scale systems. In: USENIX Annual Technical Conference. vol. 8, p. 9
10. Kamburugamuve, S., Fox, G., Leake, D., Qiu, J.: Survey of distributed stream processing for large stream sources
11. Kreps, J., Narkhede, N., Rao, J.: Kafka: A distributed messaging system for log processing. In: Proceedings of the NetDB
12. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: Ros: an open-source robot operating system. In: ICRA workshop on open source software. vol. 3, p. 5
13. Riazuelo, L., Civera, J., Montiel, J.: C 2 tam: A cloud framework for cooperative tracking and mapping. Robotics and Autonomous Systems 62(4), 401–413 (2014)
14. Videla, A., Williams, J.J.: RabbitMQ in action. Manning (2012)
15. Zhang, H., Martin, F.: Cuda accelerated robot localization and mapping. In: Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on. pp. 1–6. IEEE