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

Supun Kamburugamuve, Geoffrey C. Fox
School of Informatics and Computing and CGL
Indiana University
Bloomington, USA
[skamburu, gcf]@indiana.edu

Hengjing He
State Key Lab. of Power System, Dept. of Electrical Engineering, Tsinghua University
Beijing, China
hehj11@mails.tsinghua.edu.cn

Abstract In this paper we propose a cloud-based distributed architecture for solving the Simultaneous Localization and Mapping (SLAM) problem and implement a Rao-Blackwellized Particle Filtering-based SLAM algorithm in a multi-node cluster environment in the cloud. With this approach we obtained significant efficiency improvements in computation time. This gain in efficiency allows the algorithm to increase its complexity and frequency of calculations, which are factors for increasing the accuracy of the maps built. Because the computation happens in a cloud environment the robot's onboard computer can be a low end computer. Our method for implementing particle filtering in the cloud environment is not specific to the SLAM algorithm and can be applied to any computationally intensive particle filtering algorithm.

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, and even 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 large amounts of data produced by these devices. On the one hand 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. At the same time 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 type of applications by implementing 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 and 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 uses distance measurements from a laser range finder and odometer measurements of the robot for its calculations and is a Rao-Blackwellized Particle Filtering(RBPF) based SLAM algorithm[1, 2]. 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 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)[3] for developing and executing scalable real time applications in the cloud. We have implemented the RBPF SLAM 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 in which different parts run on different nodes. 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 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.

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 [5] 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.
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 C^2TAM 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. C^2TAM does not provide such a framework. Zhang *et al* [6] 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 [7] 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[8] 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[9] as the DSPF, RabbitMQ[10] or Kafka[11] as the message broker and an OpenStack academic cloud[12] (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[13] based coordination and discovery service.

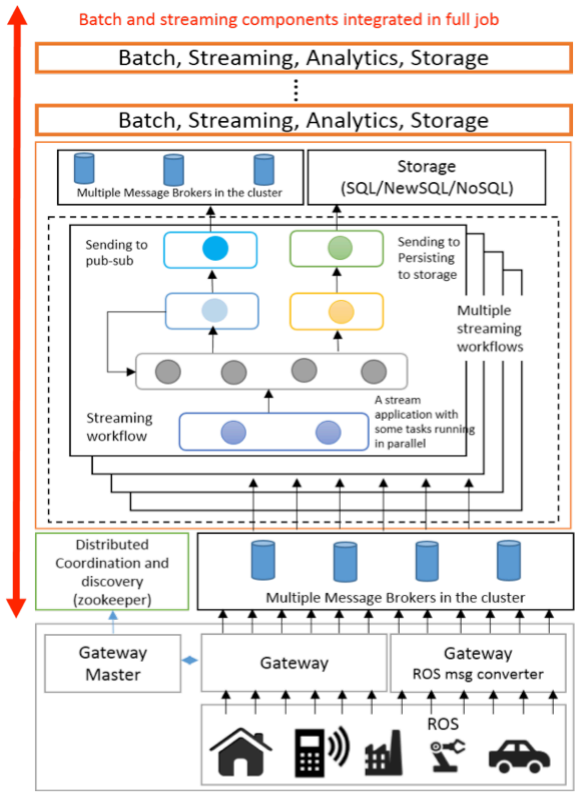

Figure 1 IoTCloud Architecture

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. 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; 5. Provide mechanisms to recover from faults. In general DSPF allows the same task to be executed in parallel and provides rich communication channels among the tasks. Some DSPF's allow the applications to define the stream workflow graph explicitly, while others create 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.

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 is provided to send and receive data to the gateway application. The real time applications are developed at the streaming layer according to the API's provided by the DSPF. Dataflow between the application and the device can happen via TCP, device specific message protocols, message brokers, etc. 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.

3.1.1 Design of GMapping application

One of our main goals was to develop a generic parallel version of the GMapping algorithm that can be used with any robot. The parallel version of this algorithm as a streaming workflow in Apache Storm and code is open source. To validate its practical use, we have developed an application to connect the TurtleBot[14] robot by Willow Garage to the GMapping algorithm running in the cloud using the IOTCloud platform. TurtleBot is an off-the-shelf differential drive robot equipped with a Microsoft Kinect sensor. It has a ROS[15] 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 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

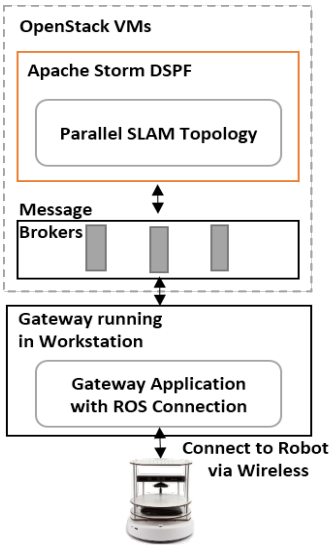

Figure 2 Turtlebot Application

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[12] VMs using the message brokering layer. Correlation between the odometer readings and the laser scans is done at the gateway to reduce the complexity of the cloud application and keep it generic. 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.

3.2 RBPF SLAM Algorithm

As described in [1, 2] Rao-Blacwellized particle filter for SLAM is estimating the posterior $p(x_{1:t}, m|z_{1:t}, u_{1:t-1})$ where $x_{1:t} = x_1, \dots, x_t$ is the trajectory of the robot and m is the map. $z_{1:t} = z_1, \dots, z_t$ are the laser readings observed and $u_{1:t-1} = u_1, \dots, u_{t-1}$ are the odometer measurements.

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

The above factorization first estimates the position of the robot given the observations, and then calculates the map given and 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 robot's 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 particle using that particle's 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 to optimize initial guess $x_t'^{(i)}$ using the $m_{t-1}^{(i)}$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 $N_{eff} = \frac{1}{\sum_i^n (w^{(i)})^2}$ 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.

```
1 function scanMatch(pos_t,readings)
2  l = -∞
3  bestPose = pos_t
4  likelihood = likelihood(pos_t, readings)
5  delta = presetDelta
6  for i = 1 to n reffinements do
7    delta = delta/2
8    pose = bestPose
9    repeat
10      for d = 1 to K do
11        xd = deterministicsample(pose_t,delta)
12        localL = likelihood(xd,readings)
13        if localL > l
14          l = localL
15          bestPose = xd
16        end if
17       end for
18   until l > likelihood
19  end for
20  return l, bestPose
21 end function
```

Algorithm 1 Scan Matching Algorithm

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 robot's onboard computer to be a low-end computer consuming less power.

4. Streaming Parallel Algorithm Design

To enable continuous processing of the incoming laser readings, the distributed algorithm has to work on a stream of laser readings and odometer readings coming from the robot in real time. In our platform such applications are written over a DSPF. Applications logic is divided into small components which are distributed in the cluster and connected by streams of events.

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 and computation on particles can be executed in parallel. Even though the computations over the particles 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. After resampling, some of the particles have to be re-distributed over the cluster.

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 and it happens 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 ScanMatching 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. The number of instances of the ScanMatcher bolt running parallel defines the parallelism of the algorithm. When multiple ScanMatcher bolts are running

in parallel, the algorithm partitions the particles into these bolts. The ScanMatcher task does the computation on the assigned particles serially.

The resampling bolt requires the result of the ScanMatcher bolts running in parallel, so it waits until all the resampling bolts are finished for one reading of the computation and sends the results. After a resampling happens, it can remove some existing particles and duplicate others. Because of this

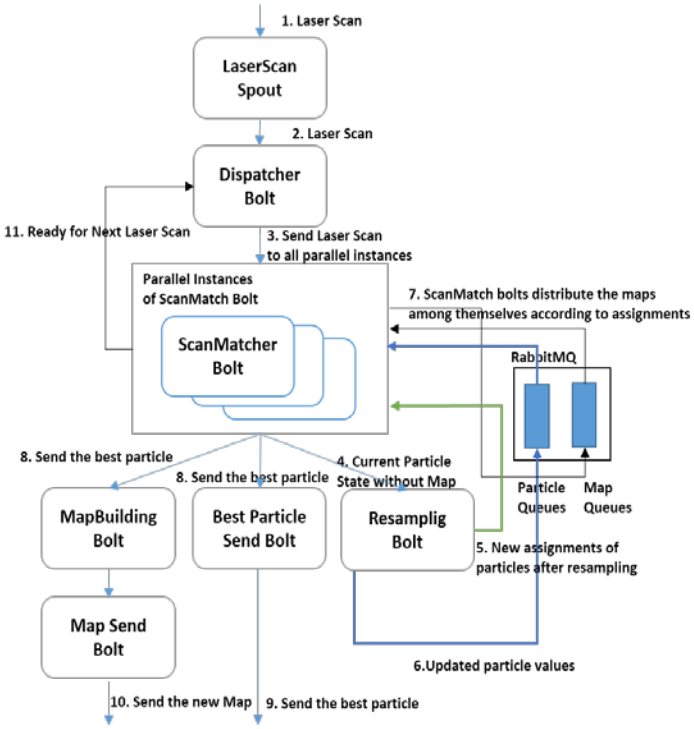

Figure 3 Storm Streaming Work Flow for Parallel RBPF SLAM

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. It discovers communication channels using the ZooKeeper-based discovery service of IoTCloud. Each message contains a laser scan and a corresponding odometer reading in byte format. This spout sends the bytes it receives to the Dispatcher bolt without any modifications.

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. The implementation doesn't send the maps associated with the particles to the resampling bolt because a resampling bolt doesn't need the map to do its computation and maps can be large objects depending on the world size.

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

6. In parallel to Step 5, 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.

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 sends its values and the map to the MapBulding bolt. This then builds a renderable map from the map used by the particle. It will also send the best particle to the BestParticle Send bolt which will directly send the information to gateways.

9. Best particle information will be sent to gateways.

10. The map expected by the visualizer will be sent to gateways.

11. ScanMatcher bolts send messages indicating their willingness to accept the next reading to the dispatcher bolt.

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 at a Dispatcher bolt 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. We can run our algorithm much faster than that speed, thereby allowing the robot to move faster. 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. 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.

An open source serial version of the algorithm implemented in C++ language is available through OpenSlam.org. Because of the C++ implementation, this algorithm is not suitable for our platform, which mainly focuses on Java-based applications. This has been 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. The algorithm described above was implemented in Java with the API provided by the DSPF.

5. Results & 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 FutureGrid[12] 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. The renderable map building happens asynchronously after a configurable time has passed and is not a core part of the algorithm. We did not measure the time it takes to build the maps; instead we focused on the computation.

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 4. 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. It gives a measurement of how much better a parallel algorithm can perform compared to the serial version. 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. For each of these datasets, the serial version time was measured for different particle sizes in a FutureGrid machine that we used for running the parallel version. In the DSPF cluster we had 5 worker nodes with 20 CPU cores, 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.

Table 1 Futuregrid VM Configuration

CPU Model	**Intel Core i7 9xx**
CPU Frequency/Mhz	2933.436
Cores	4
Thread per core	1
Memory/MB	8192
OS	Ubuntu 12.04.4 (Linux 3.2.0)
Hypervisor	KVM

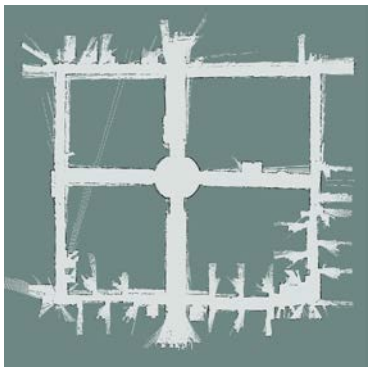

Figure 4 ACES building MAP built with Angular Update .25 and Linear Update 0.5

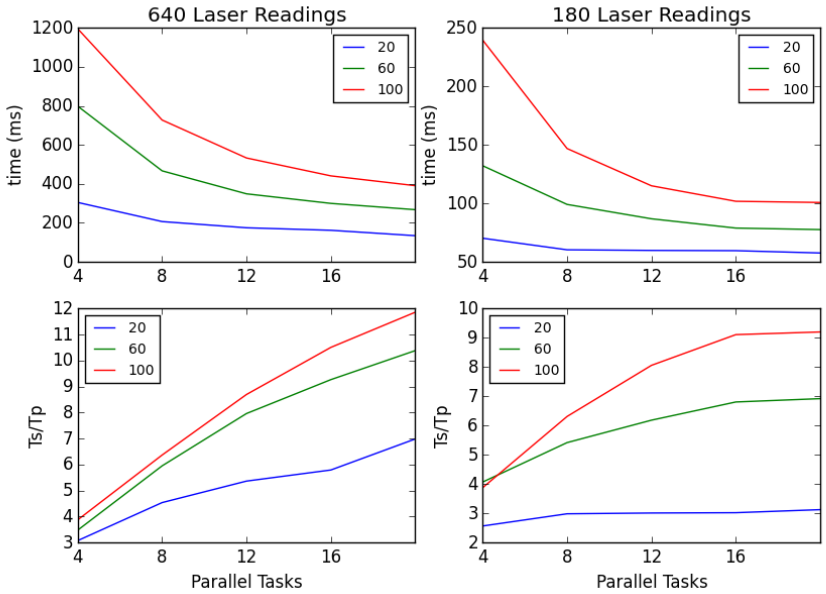

Figure 4 Parallel behavior of the algorithm for 180 and 640 laser readings. The two graphs at the top show the actual time and bottom graphs show the speedup

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.

Table 2 Serial average time for different laser readings and particles

Laser\Particles	20	60	100
640	987.8	2778.7	4633.84
640 with Cutoff	792.86	2391.4	4008.7
180	180	537	927.2

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. Let's assume we have n parallel ScanMatcher tasks taking $t_1, t_2, \dots, t_m, \dots, t_n$ 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 $t_1 + t_2 + \cdots t_m + \cdots + t_n$. 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

$$t_{overhead} = t_m - (t_1 + t_2 + \cdots t_m + \cdots + t_n)/n.$$

As we can clearly see, 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.

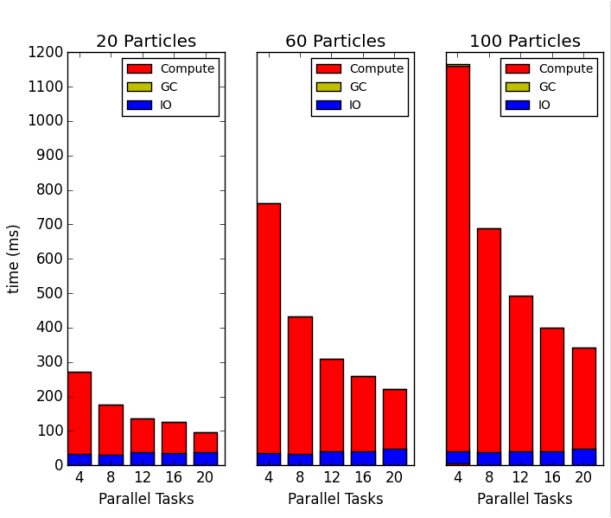

Figure 5a IO, GC and Compute time for 640 readings

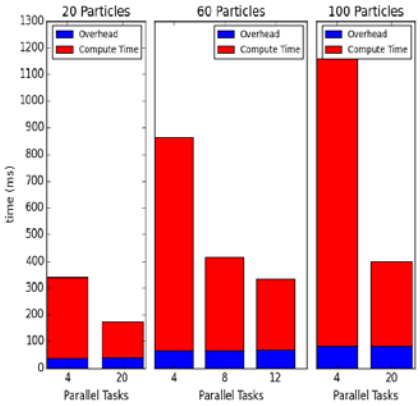

Figure 5b Overhead of imbalanced parallel computation

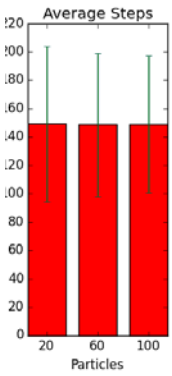

Figure 5c Average step count with standard deviation

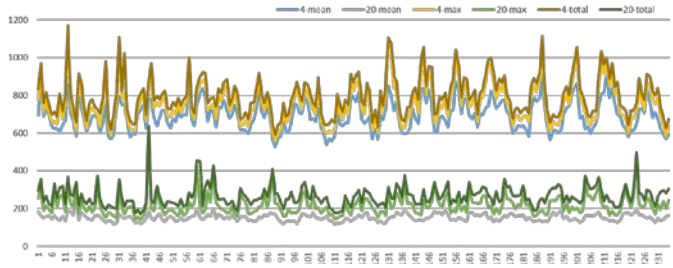

Figure 5d Time variations without limit on steps for 60 particles with 4 and 20 parallel tasks, mean and max of parallel times along with total time is shown

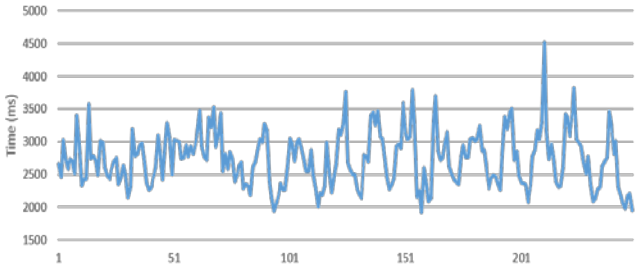

Figure 5e Time variations with in serial algorithm without limit on steps

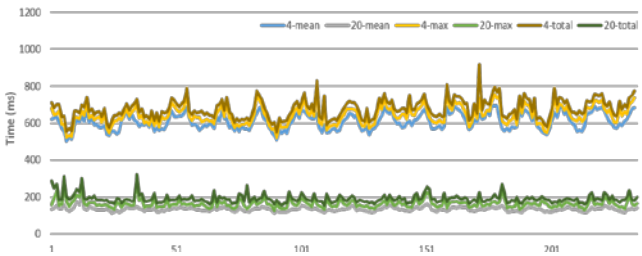

Figure 5f Time variations with limit of 140 steps for 60 particles with 4 and 20 parallel tasks, , mean and max of parallel times along with total time is shown

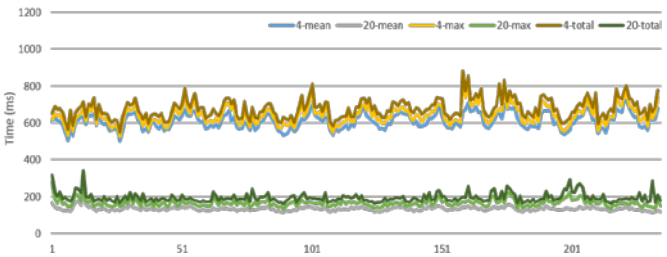

Figure 5g Time variations with limit of 140 steps for 60 particles with 4 and 20 parallel tasks, mean and max of parallel times along with total time is shown

To further investigate the behavior of the algorithm *we* drew the individual times as shown in Figure 5d and 5e. 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 standard deviation of steps executed by the ScanMatching algorithm for Simbad dataset. The standard deviation can be large and sometimes we have seen 2~3 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 shouldn't 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 algorithm to have 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 2. 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 5f and 5g. Now we no longer see some of the big peaks and variations we saw in Figure 5d. The high peaks are due to minor garbage collections occurring. Figure 6 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 Figure 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.

```
function scanMatch(pose_t,readings)
  steps = 0
  l = −∞
  bestPose = pose_t
  likelihood = likelihood(pose_t, readings)
  delta = presetDelta
  for i = 1 to n reffinements do
    delta = delta/2
    pose = bestPose
    repeat
      for d = 1 to K do
        xd = deterministicsample(pose_t,delta)
        localL = likelihood(xd,readings)
        steps++
        if localL > l
          l = localL
          bestPose = xd
        end if
      end for
    until l > likelihood && steps < cutOff
  end for
  return l, bestPose
end function
```

Algorithm 2 ScanMatching algorithm with configurable cutoff

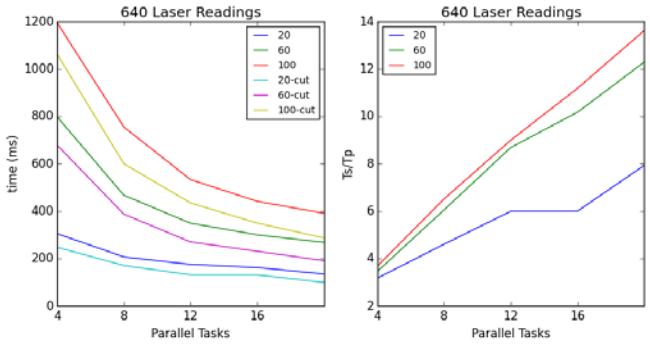

Figure 6 Time and Speedup after cutoff of 140

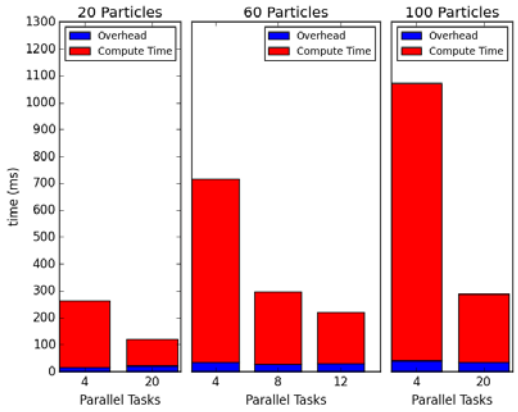

Figure 7 Overhead introduced because of compute time differences after step cutoff of 140

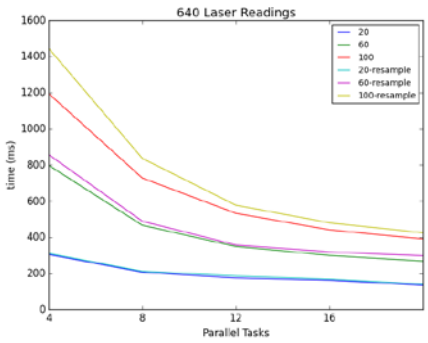

Figure 8 Resampling overhead

Because we have chosen an upper bound to cutoff iterations in particles which occur infrequently, we didn't observe a large speedup gain or reduction in average time. Instead now the algorithm won't have very large peaks that increase the individual computation times. For our experiments we cut the iteration count to prevent very large variations. It would be a worthy exercise to cut these further to gain even more processing time over the distributed tasks and see how the algorithm behaves. This will be an environment and robot dependent experiment and can potentially produce better results.

Even though the resampling only happens occasionally in the GMapping algorithm it can introduce a large overhead to the parallel algorithm because of the IO 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 for achieving such distribution tasks. In our case we were relying on an external broker. Figure 7 shows the difference in calculations when we conducted the resampling step for each operation with the Simbad dataset.

While this is not a major concern for SLAM applications, there are applications that can by affected by 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 does not 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 the algorithm can recover is by using timeouts.

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. A higher number of particles generally means increased accuracy for the algorithm. By distributing the particles across machines, an application can utilize a high number of particles, improving the accuracy of the algorithms.

6. 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. 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 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.

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 approach would be to introduce duplicate computations for such applications to get the computation with minimum time and discard the rest. This is a more generic method that can be applied irrespective of the application at the expense of more resources.

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. 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.

7. Future Work

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. For example, parallelization of an application can be embedded into the programming model of a DSPF rather than programming the parallel execution manually.

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. One possible approach is to use duplicate computation to avoid random fluctuations at the cost of more resource utilization, although developing efficient duplication of computation can be a challenging task.

Another important area is how to schedule the tasks in a dynamic environment where devices connect and disconnect randomly. As this occurs the application resources must be rescheduled to get 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 preserve 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 algorithm in a distributed environment can be a worthy experiment.

Acknowledgement

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] G. Grisetti, 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, pp. 2432-2437.

[2] G. Grisetti, C. Stachniss, and W. Burgard, "Improved techniques for grid mapping with rao-blackwellized particle filters," *Robotics, IEEE Transactions on*, vol. 23, pp. 34-46, 2007.

[3] S. Kamburugamuve, G. Fox, D. Leake, and J. Qiu, "Survey of Distributed Stream Processing for Large Stream Sources."

[4] M. Chitchian, A. S. van Amesfoort, A. Simonetto, T. Keviczky, and H. J. Sips, "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] B. D. Gouveia, 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, pp. 1583-1588.

[6] H. Zhang and F. Martin, "CUDA accelerated robot localization and mapping," in *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, 2013, pp. 1-6.

[7] X. Gao, E. Ferrara, and J. Qiu, "Parallel Clustering of High-Dimensional Social Media Data Streams."

[8] I. U. Community Grids Lab. (2015). *IoTCloud*. Available: http://iotcloud.github.io/

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

[10] A. Videla and J. J. Williams, *RabbitMQ in action*: Manning, 2012.

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

[12] G. Fox, G. von Laszewski, J. Diaz, K. Keahey, J. Fortes, R. Figueiredo, *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.

[13] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *USENIX Annual Technical Conference*, 2010, p. 9.

[14] (2014). *TurtleBot*. Available: http://wiki.ros.org/Robots/TurtleBot

[15] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, *et al.*, "ROS: an open-source Robot Operating System," in *ICRA workshop on open source software*, 2009, p. 5.