Cloud-based Parallel Implementation of SLAM for Mobile Robots

Supun Kamburugamuve[1], Hengjing He[2], Geoffrey Fox[1], David Crandall[1]

[1] School of Informatics and Computing, Indiana University, Bloomington, USA
[2] Dept. of Electrical Engineering, Tsinghua University, Beijing, China

Abstract. Simultaneous Localization and Mapping (SLAM) for mobile robots is a computationally expensive task. A robot capable of SLAM needs a powerful onboard computer, but this can limit the robot's mobility because of weight and power demands. We consider moving this task to a remote compute cloud, by proposing a general cloud-based architecture for real-time robotics computation, and then implementing a Rao-Blackwellized Particle Filtering-based SLAM algorithm in a multi-node cluster in the cloud. In our implementation, expensive computations are executed in parallel, yielding significant improvements in computation time. This allows the algorithm to increase its complexity and frequency of calculations, which in practice could enhance the accuracy of the resulting map, while freeing the robot's onboard computer for other tasks. 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, which brings Internet connectivity to devices ranging from simple thermostats to highly complex industrial machinery and robots. Many potential applications involve analyzing the rich, large-scale datasets that these devices produce, either offline or in real-time. But powerful analytics algorithms are often also computationally expensive, and difficult 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. Cloud services are thus attractive for doing large scale offline and real time analytics on the data produced in IoT applications. 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 run on multicore machines using threads as the primary parallelization mechanism, which bounds parallelism by the number of CPU cores available and the amount of memory in a single machine. This degree of parallelism is often not enough for computationally expensive algorithms to provide real time responses. Being able to execute computations in parallel in a distributed environment could give a cost-effective option for these applications to meet low-latency requirements while also scaling up or down depending on the processing requirements.

Simultaneous Localization and Mapping (SLAM) is an important problem in which a mobile robot tries to estimate both a map of an unknown environment and its position in that environment, given imperfect sensors with measurement error. This problem is computationally challenging and has been studied extensively in the literature. Here we consider a popular algorithm called GMapping, which builds a grid map by combining (noisy) distance measurements from a laser range finder and robot odometer using Rao-Blackwellized Particle Filtering (RBPF) [1, 2]. It is known to work well in practice and has been integrated into robots like TurtleBot [3]. The algorithm is computationally expensive and produces better results if more computational resources are available.

IoTCloud [4] is a framework for transfering data from devices to a cloud computing environment for scalable data processing in real time. The data from the devices is encapsulated into events and sent to cloud systems in real time, where it is processed using a distributed stream processing framework (DSPF) [5]. We have implemented GMapping to work in the cloud on top of the IoTCloud platform. Laser scans and odometer readings are sent from the robot as a stream of events to the cloud, where they are processed by SLAM and results are returned to the robot immediately. The algorithm runs in a fully distributed environment where different parts of it can be run on different machines, taking advantage of parallelism to split up the expensive computation of the algorithm.

Our main contribution is to propose a novel framework to compute particle filtering-based algorithms, specifically RBPF-based SLAM, in a cloud environment to achieve higher computation time efficiency. We first discuss related work before introducing the IoTCloud framework and background on the SLAM algorithms. Then we show how to design and implement parallel RBPF SLAM in IoTCloud. Finally we conclude with results and discussion.

2 Related Work

Kehoe et al. [6] summarize existing work and architectures for cloud robotics, and our architecture is similar to several of them. A key difference is we are proposing a generic streaming architecture coupled with other big data platforms to build applications. Hu et al. [7] describe some of the challenges in cloud robotics such as communication constraints. Chitchian et al. [8] exploit multicore and GPU architectures to speed up particle filtering-based computations, while Gouveia et al. create thread-based implementations of the GMapping algorithm with good performance gains [9]. In contrast, our approach takes advantage of distributed environments with multiple multi-core machines. The C2TAM [10] framework considers a similar problem of moving some of the expensive computation of SLAM to a cloud environment. They use a visual SLAM algorithm called PTAM [11] which uses a video camera. Also in contrast to C2TAM, we propose a generic scalable real time framework for computing the maps online with significant gains in the processing time. Zhang et al. [12] describe an approach where the CUDA API is used to run part of GMapping (specifically the Scan Matching step) on GPUs to improve performance, and Tosun et al. [13] address particle filter-based localization of a vehicle with multicore processors. But neither of these is not a multi-node implementation as we create here.

3 Background

3.1 IoTCloud framework

IoTCloud [4] is an open source framework developed at Indiana University to connect IoT devices to cloud services. As shown in Figure 1, it consists of a set of distributed nodes running close to the devices to gather data, a set of publish-subscribe brokers to relay 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. Applications execute data analytics at the DSPF layer, achieving streaming real-time processing. The IoTCloud platform uses Apache Storm [14] as the DSPF, RabbitMQ [15] or Kafka [16] as the message broker, and an OpenStack academic cloud [17] (or bare-metal cluster) as the platform. To scale applications with number of devices, we use a coordination and discovery service based on ZooKeeper [18].

In general, a real time application running in a DSPF can be modeled as a directed graph with streams defining the edges and processing tasks defining the nodes. A stream is an unbounded sequence of events flowing through the edges of the graph and each such event consists of a chunk of data. 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 applications on a cluster of nodes. In Storm these tasks are called Spouts and Bolts. To connect a device to the cloud services, a user develops a gateway application that connects to the device's data stream. Once an application is deployed in an IoTCloud gateway, the cloud applications discover those applications and connect to them for data processing using the discovery service.

3.2 Design of robot applications

We designed a cloud-based implementation of GMapping for a real robot, the TurtleBot [3] by Willow Garage, using the IOTCloud platform. TurtleBot is an off-the-shelf differential drive robot equipped with a Microsoft Kinect sensor. An overview of the implementation is shown in Figure 2. The application that connects to the ROS-based [19] Turtlebot API is deployed in an IoTCloud Gateway running on a desktop machine, where it subscribes to the TurtleBot's laser scans and odometer readings. 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 [17] VMs using the message brokering layer. The cloud application generates a map and sends this back to the workstation running the gateway, which saves and publishes it back to ROS for viewing.

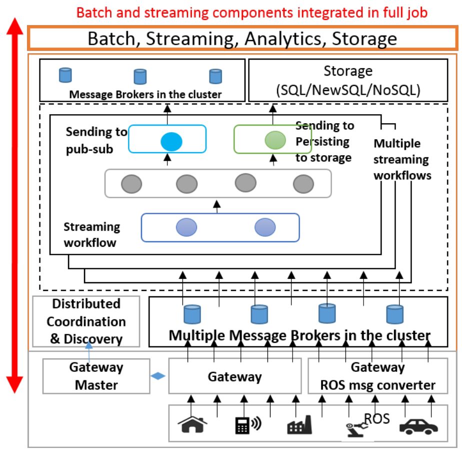


Fig. 1: IoTCloud Architecture

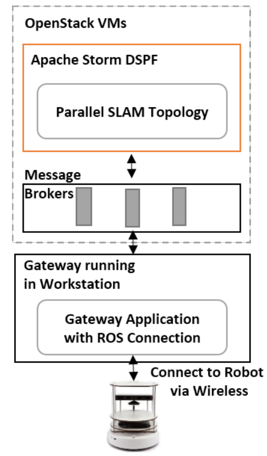


Fig. 2: GMapping Robotics application

3.3 RBPF SLAM Algorithm

A detailed description of the Rao-Blackwellized particle filter for SLAM is given in [1, 2], but we give a brief overview here. Suppose we have a series of laser readings $z_{1:t} = (z_1, ..., z_t)$ over time, as well as a set of odometer measurements $u_{1:t-1} = (u_1, ..., u_{t-1})$ from the robot. Our goal is to estimate both a map m of the environment and the trajectory of the robot, $x_{1:t} = (x_1, ..., x_t)$. For any time t, we can sample from the posterior probability,

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

by sampling from the first term on the right hand side to produce an estimate of the robot's trajectory given just the observable variables, and then sample from the second term to produce an estimate of the map using that sampled trajectory. The particle filter maintains a set of particles, each including a possible map of the environment and a possible trajectory of the robot, along with a weight which can be thought of as a confidence measure. A standard implementation of the algorithm executes the following steps for each particle i as follows:

1. Make an initial estimate of the position of the robot at time t, using the estimated position at time $t-1$ and odometry measurements, i.e. $x_t'^i = x_{t-1}'^i \oplus u_{t-1}$ where $\oplus$ is a pose compounding operator. The algorithm also incorporates the motion model of the robot when computing this estimate.
2. Use the ScanMatching algorithm shown in Algorithm 1 with cutoff of ∞ to refine $x_t'^i$ using the map m_{t-1}^i and laser reading z_t. If the ScanMatching fails, use the previous estimate.
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.

```
input  : pose u and laser reading z
output: bestPose and l
steps ← 0; l ← −∞; bestPose ← u; delta ← InitDelta;
currentL ← likelihood(u, z);
for i ← 1 to nRefinements do
    delta ← delta/2;
    repeat
        pose ← bestPose; l ← currentL;
        for d ← 1 to K do
            xd ← deterministicsample(pose, delta);
            localL ← likelihood(xd, z);
            steps+ = 1;
            if currentL < localL then
                currentL ← localL; bestPose ← xd;
            end
        end
    until l < currentL and steps < cutoff ;
end
```

Algorithm 1: Scan Matching

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

We found that RBPF SLAM spends nearly 98% of its computation time on Scan Matching. However, because Scan Matching is done for each particle independently, it is well-suited for parallel execution, and in a distributed environment the particles can be partitioned into different computation nodes and computation can be done in parallel. However, the resampling step requires information about all particles so it needs to be executed serially, after gathering results from the parallel computations. Resampling also removes some particles and duplicates them, which means that some particles have to be redistributed over the compute cluster after resampling.

Our 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

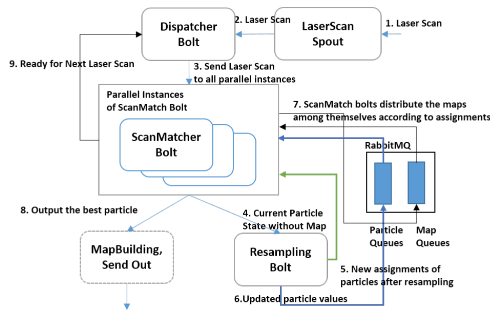


Fig. 3: Storm streaming workflow for parallel RBPF SLAM.

cluster of nodes in the cloud. The main tasks of the algorithm are divided into ScanMatcherBolt and ReSamplingBolt. The LaserScanBolt receives data from the robot and sends it to the rest of the application. After computation, results are passed to SendOut bolts which send it back to the robot. If required, data can be saved to persistent storage as well.

A key idea behind our implementation is to distribute the particles across a set of tasks running in parallel. This particle-specific code is encapsulated in the ScanMatcher bolt, and so that we can control the parallelism of the algorithm by changing the number of ScanMatcher bolt instances. The Resampling bolt must wait until it receives the results of the ScanMatcher bolts. After a resampling happens, the algorithm removes some existing particles and duplicate others, so the assignments of particles to ScanMatcher tasks have to be rearranged. 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. All the data flowing through the various communication channels are in a byte format serialized by Kryo. The steps for a single reading as shown in Figure 3 are:

1. LaserScan spout receives laser and odometer readings via the message broker.
2. The reading is sent to a Dispatcher, which broadcasts it to the parallel tasks.
3. Each ScanMatcher task receives the laser reading, updates its assigned particles, and sends the updated values to the Resampling bolt.
4. After resampling, the Resampling bolt calculates new particle assignments for the ScanMatchers, using the Hungarian algorithm to consider relocation costs. The new particle assignment is broadcast to all the ScanMatchers.
5. In parallel to Step 5, the Resampling bolt sends the resampled particle values to their new destinations according to the assignment, using RabbitMQ queues to send messages directly to tasks.
6. After ScanMatchers receive new assignments, they distribute the maps associated with the resampled particles to the correct destinations.
7. The ScanMatcher with the best particle outputs its values and the map.
8. ScanMatcher bolts send messages to the dispatcher, indicating their willingness to accept the next reading.

Table 1: Serial average time (in ms) for different datasets and numbers of particles.

Data set	Particle count 20	60	100
Simbad	987.8	2778.7	4633.84
Simbad, ScanMatching Cutoff = 140	792.86	2391.4	4008.7
ACES	180	537	927.2

The parallel version exploits the algorithm's ability to lose readings by dropping messages that arrive at a Dispatcher bolt while a computation is ongoing to avoid memory overflow. Owing to the design of the GMapping algorithm, only a few resampling steps are needed during map building. If resampling does not happen then steps 5 and 6 are omitted. Although an open source serial version of the algorithm in C++ is available through OpenSlam.org, it is not suitable for our platform which requires 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 and practical feasibility of our approach, as well as to measure its scalability. All experiments ran in FutureGrid [17] OpenStack VMs each having 8GB memory and 4 CPU cores running at 2.8GHz. Our setup had 5 VMs for Apache Storm Workers, 1 VM for RabbitMQ and 1 VM for ZooKeeper and Storm master (Nimbus) node. For all the tests the gateway node was running in another VM in FutureGrid. Each Storm worker instance ran 4 Storm worker processes with 1.5GB of memory allocated. We used the ACES building data set [20] and a small environment generated by the Simbad [21] robot simulator for our experiments. ACES has 180 distance measurements per laser reading and Simbad has 640 measurements per laser reading. For the ACES dataset we used a map of size 80x80m with 0.05 resolution, and for Simbad the map was 30x30m with 0.05 resolution.

A sample result from ACES is shown in Figure 10. Since GMapping is a well-studied algorithm, we did not extensively test its accuracy on different datasets, but instead focused on the parallel behavior of our implementation. We measured parallel speedup (defined as serial time over parallel time, i.e. T_s/T_p) by recording the time required to compute each laser reading and then taking an average. We tested the algorithm with 20, 60 and 100 particles for each dataset. The serial time was measured on a single FutureGrid machine, as shown in Table 1, and the parallel times were measured with 4, 8, 12, 16 and 20 parallel tasks.

The parallel speedups gained for the ACES and Simbad datasets are shown Figure 4. For ACES, the number of points per reading is relatively low, requiring relatively little computation at the the ScanMatcher bolts, which results in only a modest parallel speedup after 12 particles. On the other hand, the Simbad dataset has about 4 times more distance measurements per reading and produces higher speed gains of about a factor of 12 for 20 parallel tasks with 100 particles. However, ideally the parallel speedup should be close to 20 with 20 parallel tasks, so we investigated factors that could be dragging down the speedup. Only the scan matching step of the algorithm is executed in parallel; the resampling step is done serially. Because this serial computation is relatively inexpensive compared with Scan Matching, the speedup loss is not significant at this step. The main factor for reducing the distributed parallel computations is the I/O time. Because our computation is done in Java, garbage collection also has an effect on performance. Figure 5a shows the I/O, garbage collection, and computation times for different parallel tasks and particle sizes. When the number of parallel tasks increases, the compute time decreases, and because of the I/O overhead the speedup reduces. The average garbage collection 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. 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 time imbalance is $t_{overhead} = t_m - (t_1 + ... + t_m + ... + t_n)/n$. Figure 5b shows the average overhead calculated for the Simbad dataset compared with the total time. The average overhead remained almost constant while the total time decreased as parallel tasks increase, resulting in less speedup.

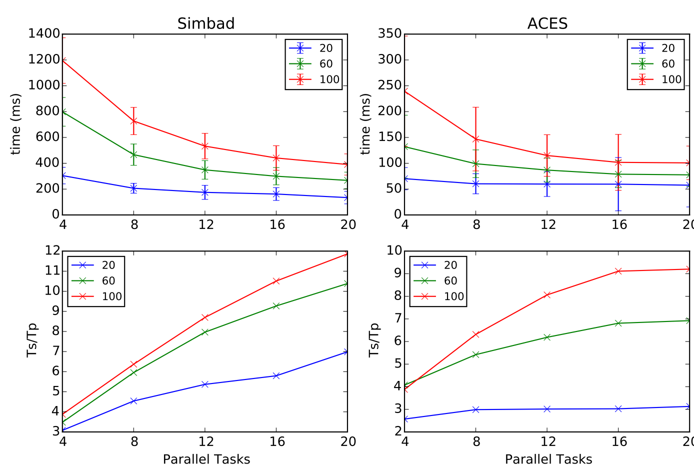
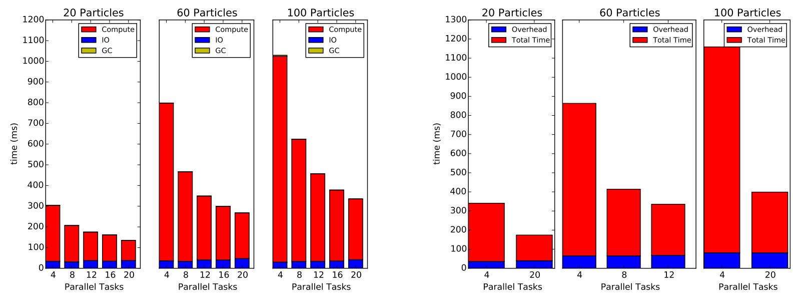


Fig. 4: Parallel behavior of the algorithm for the Simbad dataset with 640 laser readings (left) and the ACES dataset with 180 readings (right). For each dataset, the top graph shows mean times with standard deviations and the bottom graph shows the speedup.



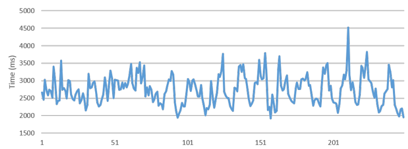
(a) (b)

Fig. 5: Overhead on the Simbad dataset: (a) I/O, garbage collection, and Compute time. (b) Overhead of imbalanced parallel computation.

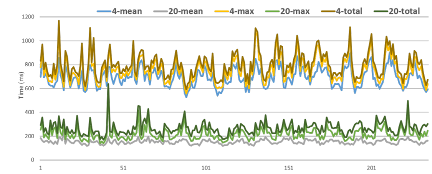
To further investigate the behavior of the algorithm, we plotted the computation times for each reading, as shown in Figures 6a and 6b. There are high peaks in the individual times in both serial and parallel algorithms. This is caused by the while loop ending in line 5 of Algorithm 1, which can execute an arbitrary number of times if $cutoff = \infty$. We have observed a mean of about 150 and standard deviation around 50 for number steps executed by the Scan-Matching algorithm for the Simbad dataset. 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. An advantage of nondeterministic particle-based algorithms is that if there are a sufficiently large number of particles, cutting off the optimization for one or two of them prematurely should not affect the algorithm's results. 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 with small delta values. So the corrections gained executing many loops is minimal in most cases.

We thus changed the original algorithm shown in Algorithm 1 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 empirical average. We found that maps built by this modified algorithm were of comparable quality to the originals. The resulting time variations for two tests are shown in Figures 6c and 6d. Here we no longer see as many large peaks as in Figure 6b, and the remaining peaks are due to minor garbage collection. Figure 7 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 9 after cutoff at 140 steps. This demonstrates that the cutoff is an important configuration parameter for the parallel version that can be tuned to obtain optimum performance and correctness.

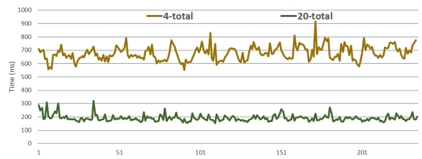
Figure 8 shows the difference in calculations when we conducted the resampling step for every reading with the Simbad dataset. When the number of



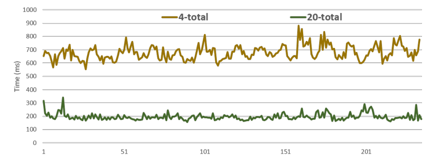
(a) Serial time



(b) Without cut-off



(c) Total time after Cutoff 140 steps



(d) Total time after Cutoff at 140 steps

Fig. 6: Computation time across individual readings, for Simbad and 60 particles.

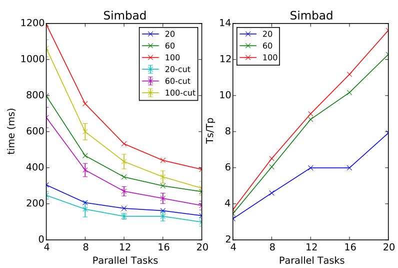


Fig. 7: Average time and speedup for Simbad with cutoff at 140

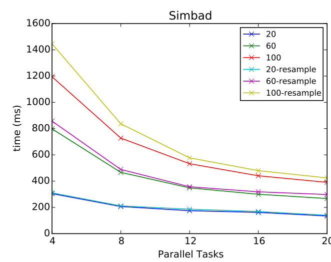


Fig. 8: Resampling overhead with Simbad

particles is high, the overhead is large. When we have more parallel workers, the map distribution happens simultaneously using more nodes, which reduces the I/O time. 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. In particle filtering-based methods, the 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 Conclusions & Future Work

We have shown how to offload robotics data processing to the cloud through a generic real-time parallel computation framework, and our results show significant performance gains. Because the algorithm runs on a distributed cloud, it has access to potentially unbounded memory and CPU power. This allows

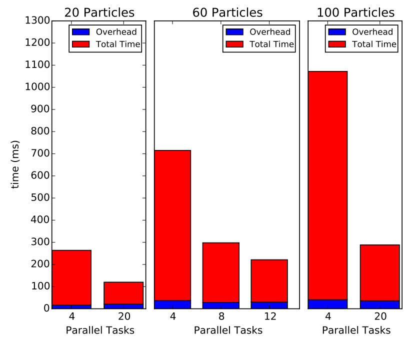


Fig. 9: Parallel overhead with cutoff at 140

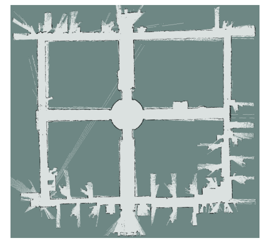


Fig. 10: Map of ACES

the system to scale well, and for example could build maps in large, complex environments needing a large number of particles or dense laser readings.

There are several possible enhancements to our system. Imbalances in computation times across particles can reduce both parallel speedup and response times. We addressed this problem with an upper bound on particle computation time, which works well for this particular algorithm but may not for others. Investigating more generic solutions would be a useful direction. Also, we have observed that result broadcast and gathering in the streaming tasks takes considerable time, so reducing I/O would also significantly improve performance.

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. Future work should propose higher-level APIs to handle complex interactions by abstracting out the details. For example, distributing state between parallel workers currently requires a third node, such as an external broker or a streaming task acting as an intermediary. A group communication API between the parallel tasks would be a worthy addition to DSPF. Extending our work to abstract out a generic API to quickly develop any particle filtering algorithm would also be interesting.

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. Grisetti, G., Stachniss, C., Burgard, W.: Improving grid-based SLAM with Rao-Blackwellized particle filters by adaptive proposals and selective resampling. In: IEEE International Conference on Robotics and Automation. (2005) 2432–2437

2. Grisetti, G., Stachniss, C., Burgard, W.: Improved techniques for grid mapping with Rao-Blackwellized particle filters. IEEE Transactions on Robotics **23**(1) (2007) 34–46
3. Willow Garage: Turtlebot. http://turtlebot.com/
4. Kamburugamuve, S., Christiansen, L., Fox, G.: A framework for real time processing of sensor data in the cloud. Journal of Sensors **2015** (2015) 11
5. Kamburugamuve, S., Fox, G., Leake, D., Qiu, J.: Survey of distributed stream processing for large stream sources. Technical report, Indiana University (2013)
6. Kehoe, B., Patil, S., Abbeel, P., Goldberg, K.: A survey of research on cloud robotics and automation. IEEE Transactions on Automation Science and Engineering **12**(2) (2015)
7. Hu, G., Tay, W.P., Wen, Y.: Cloud robotics: architecture, challenges and applications. IEEE Network **26**(3) (2012) 21–28
8. Chitchian, M., van Amesfoort, A.S., Simonetto, A., Keviczky, T., Sips, H.J.: Particle filters on multi-core processors. Technical Report PDS-2012-001, Dept. Comput. Sci., Delft Univ. Technology (2012)
9. Gouveia, B.D., Portugal, D., Marques, L.: Speeding up Rao-blackwellized particle filter SLAM with a multithreaded architecture. In: IEEE/RSJ International Conference on Intelligent Robots and Systems. (2014) 1583–1588
10. Riazuelo, L., Civera, J., Montiel, J.: C2TAM: A Cloud framework for cooperative tracking and mapping. Robotics and Autonomous Systems **62**(4) (2014) 401–413
11. Klein, G., Murray, D.: Parallel tracking and mapping for small ar workspaces. In: IEEE and ACM International Symposium on Mixed and Augmented Reality. (2007) 225–234
12. Zhang, H., Martin, F.: CUDA accelerated robot localization and mapping. In: IEEE International Conference on Technologies for Practical Robot Applications. (2013) 1–6
13. Tosun, O., et al.: Parallelization of particle filter based localization and map matching algorithms on multicore/manycore architectures. In: Intelligent Vehicles Symposium. (2011) 820–826
14. Anderson, Q.: Storm Real-time Processing Cookbook. Packt Publishing Ltd (2013)
15. Videla, A., Williams, J.J.: RabbitMQ in action: distributed messaging for everyone. Manning (2012)
16. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: A distributed messaging system for log processing. In: Proceedings of 6th International Workshop on Networking Meets Databases. (2011)
17. Fox, G., et al.: FutureGrid: A reconfigurable testbed for Cloud, HPC and Grid Computing. Contemporary High Performance Computing: From Petascale toward Exascale (2013)
18. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: Wait-free coordination for internet-scale systems. In: USENIX Annual Technical Conference. Volume 8. (2010) 9
19. 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. Volume 3. (2009) 5
20. Kümmerle, R., Steder, B., Dornhege, C., Ruhnke, M., Grisetti, G., Stachniss, C., Kleiner, A.: On measuring the accuracy of SLAM algorithms. Autonomous Robots **27**(4) (2009) 387–407
21. Hugues, L., Bredeche, N.: Simbad: an autonomous robot simulation package for education and research. In: From Animals to Animats 9. (2006) 831–842