Multi-threaded Particle Filter Localization

Jordan Charest, Kevin Zhan

Rensselaer Polytechnic Institute

Troy, NY 12180

Code in PPCzhank2@kratos:/project

May 1, 2018

Abstract

Particle filter localization is a powerful technique used in modern autonomous robotics. Being able to quickly and effective determine its pose is extremely important for an autonomous robot. The particle filter offers several advantages over other common techniques, but is highly compute-intensive. Optimizing localization for faster runtime is necessary for real time applications. In this experiment, we attempt to significantly reduce the runtime of a stochastic variant of the particle filter algorithm by parallelizing it using multi-threading in C++14.

1 Introduction

Particle filter localization is an important technique used in modern autonomous robotics for the purpose of state estimation. Particle filtering is so powerful because it is continuous and multimodal, in contrast to similar probabilistic localization techniques such as the Kalman filter, which is continuous, but unimodal, and the histogram filter, which is multimodal, but discrete.

As is necessary with all localization techniques, the particle filter technique requires the robot to have an accurate map of its environment, sensors or encoders to measure its motion, and sensors to take distance measurements (or even color measurements) to known landmarks. A robot with an unknown map requires an entirely different and more complex solution known as Simultaneous Localization And Mapping, or SLAM. A robot that cannot sense its environment will never localize itself.

The state space of a ground-based mobile robot is three dimensional(x-location, y-location, and orientation), as is the state space of the robot in the simulation, therefore any given robot pose can be represented as a 3x1 state vector in the state space. The ultimate goal of localization is for the robot's belief function, a probability density function over the set of possible robot poses, to converge around a small subset of state vectors that is likely to be where the actual robot lies. The nature of probabilistic robotics means that the robot can never know its exact pose, largely due to measurement uncertainty, but it can narrow down the subset of possible poses to a small distribution, of which the robot can be very confident (but never certain) that it lies within.

In particle filter localization, the robot makes guesses as to where it is relative to its environment, in the form of particles, which are simulated copies of the robot. These particles can simulate motion and sensing just as the robot does, however, the particles "know" their exact location within the simulated world at all times. The initial state of the robot is the state of maximum uncertainty. In other words, its belief function is a uniform distribution about all possible poses. This manifests in the particle filter as particles that are randomly scattered about the robot's internal map, as shown in Figure 1.

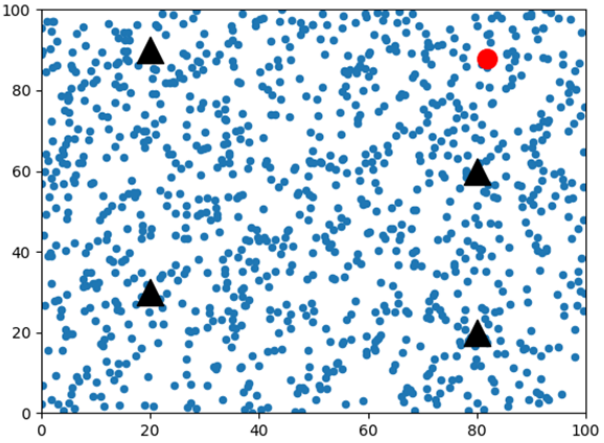

Figure 1: Initial belief; particle representation

In this particle representation of the belief function, the black triangles represent known landmarks, the large red circle near the top right corner represents the robot's actual location, and the small blue dots scattered throughout are the particles, or the robot's guesses as to its true location. Although not shown, the robot and all particles each have a defined orientation. As previously stated, this is the state of maximum uncertainty. Each particle represents a single guess and they are all equally weighted. In other words, the robot has no idea where it lies in the state space.

After initializing the filter as shown in Figure 1, the physical robot enters a loop where it performs two discrete steps: a motion update, and a sensor update. In the motion update, the robot reads and incorporates its odometry data into its estimated pose. The motion update *increases* uncertainty. Odometry data is typically noisy and inaccurate, and the further the robot travels without performing a sensor update, the lower its confidence in its current pose is. The sensor update incorporates measurements to known landmarks into its estimated pose. The sensor update *decreases* uncertainty. The physical robot continues this loop until it is finished exploring its environment, even after the particle filter has converged. Internally, the simulated particles undergo the same motion and sensor updates that the physical robot does, as well as additional calculations to determine importance weights, mean error, and resampling.

2 Background

It is widely recognized [1], [3], [4], [5] that the particle filter is a computationally intensive algorithm and as a result, is difficult to effectively implement on real-time embedded systems, where they are needed. Despite the computational complexity, the effectiveness of particle filtering is too good to ignore, and there have been many attempts to parallelize the particle filter to an effective degree. Most of these attempts focus on defining new ways to resample that have better computational complexity or that parallelize better than common algorithms in use today. The resampling algorithm used in this simulation is known as the Resampling Wheel algorithm or Stochastic Universal Sampling. It is an algorithm that is commonly used for particle filtering due to good empirical results.

3 Implementation

The serial and parallel particle filter simulation was implemented in C++14. In the simulation, the robot and particles are represented by the same data object, implemented in the `class Robot`. Landmarks are represented by dimensionless points with an x-coordinate and a y-coordinate. The robot and particles alike can move directly over the landmarks and occupy the same two-dimensional space as them, similar to lane lines on a road. The pseudo-random number generator is optionally seeded from the command line or from the current time. The program first simulates the serial particle filter, then the particle filter in the same run. Program execution can be broken up into a few distinct sections: initialization, robot/particle motion update and robot sensor update, importance weight calculation, resampling, and mean error evaluation. These section will be discussed for both the serial and parallel runs together.

The parallel simulation uses multi-threading with a shared memory paradigm, and therefore involves no message passing. All necessary data is available to all threads with atomic, mutex, and barrier functionality used as necessary. Despite the lack of message passing, synchronization is crucial for the particle filter to work properly. All threads must complete each step before any thread moves on to the next. The only two steps that threads can perform out of sync are the motion update and importance weight calculation. The importance weight calculation is dependent on the particle motion update, however, each thread always works on the same subset of particles, and the importance weight calculation for each particle is independent of all other particles. Therefore, the threads can perform two subsequent steps, motion update and importance weights before resynchronizing. C-style `p_thread_barrier`'s are used to achieve synchronization since C++ barriers are currently in an experimental state.

3.1 Initialization

The map is initialized to a constant size, with both the x and y boundaries being congruent. The map has a defined boundary that appears as a square, but in reality it is toroidal. If the robot or any particle drives past the right boundary, it reappears at the left boundary, and vice versa. Similarly, if the robot or any particle drives past the upper boundary, it reappears at the lower boundary. The world size is arbitrary and we simulated several different sizes ranging from 256 to 4096 'units'. The number of particles used was always two times the world size, ranging from 512 to 8192 particles. The robot and all particles were initialized to random poses (location and orientation) throughout the world using a pseudo-random number generator fed into a uniform distribution. Similarly, the landmarks (kept at a constant quantity of 12) were randomly initialized using the same method. Finally, for the parallel run, the threads were created. An example visualization was shown in Figure 1, and is repeated here for clarity.

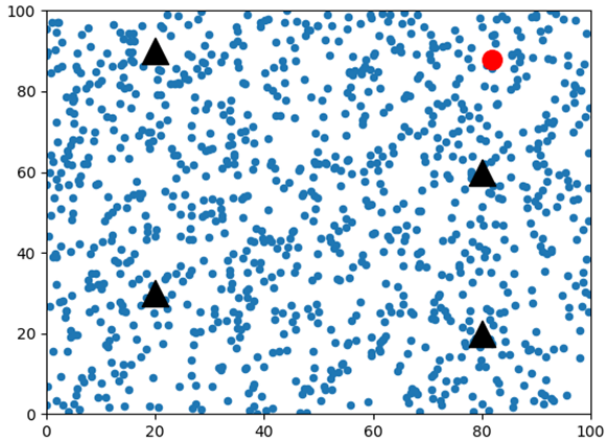

Figure 2: Random initial state

In this visualization, the black triangles represent known landmarks, the large red circle near the top right corner represents the robot's actual location, and the small blue dots scattered throughout are the particles, or the robot's guesses as to its true location. Note that the map size, particle quantity, and landmark quantity are reduced in this visualization to reduce clutter on the small image.

3.2 Motion and Sensor Updates

The robot's movement command (forward distance and orientation delta) is randomly generated, which is executed on the robot and every particle. After the motion update, the robot "senses" by measuring its Euclidean distance to each of the known landmarks. This simulation ignores the complex problem of data association. On a physical robot, the robot must associate each sensor measurement with a previously sensed landmark, which is a difficult task that gets exceedingly more difficult if the landmarks themselves are in motion (e.g. other robots). Since this project was intended to explore parallelization rather than data association, this problem was ignored in the simulation by always sensing the landmarks in the same order (a solution not possible on a real-world robot).

Sensor noise and motion noise was added by randomly pulling values from a gaussian distribution with a mean of 0 and a constant variance, with different variances for the sensor and motion updates. Sensor noise is used to determine the importance weight of a particle and is necessary for the particle filter to work. A sensor noise value of 0.0 would throw a floating point exception since the noise is used in the denominator of the importance weight calculation (see Section 3.3). The motion noise is also crucial for the particle filter to operate properly. Without it, the solution is highly unlikely to ever converge, and would only due so through random luck. The reason why will be further discussed in the resampling section.

3.3 Importance Weight Calculation

The importance weight of a particle plays an important role in resampling. The importance weight represents the likelihood that a given particle's pose is equal to the actual robot's pose. This is done by comparing the difference, in terms of Euclidean distance, between the robot's sensor measurements and the given particle's sensor measurements. The calculation of a single sensor measurement to a single landmark is computed using the probability density of a gaussian distribution:

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where the mean, μ is the distance to landmark i measured by the robot, the variance, σ^2 is the sensor noise, and x is the distance to landmark i measured by the robot. This equation is used to calculate how close a particle's sensor measurements are to the robot's and each calculation (one for each landmark) is multiplied together to form the importance weight for that particle. As you might imagine, for particles that are very far away from the robot's pose, their importance weights can underflow, even with double precision arithmetic. A particle with an importance weight that has underflowed will have a 0.0 probability of being resampled. This is usually not a problem, since this typically occurs in particles with very high error and will aid rapid convergence. However, as the number of landmarks increases, the probability of underflow increases as well, and eventually even nearby particles may have importance weights that underflow. To avoid this problem, the number of landmarks was held static at twelve in the simulation, but on larger problems with hundreds of landmarks, the importance weight calculation may need to be adjusted.

The importance weight of each particle is calculated independently from all other particles. Therefore, as previously discussed, each thread is permitted to calculate the importance weight of each particle in its subset after the motion update without synchronizing with the remaining threads.

3.4 Resampling

The resampling algorithm chosen is know as the Resampling Wheel algorithm or Stochastic Universal Sampling. This algorithm was chosen because of its strong empirical results as well as its apparent ability to be done in parallel. Stochastic Universal Sampling is non-deterministic in both results *and* time complexity. The following pseudocode describes the algorithm, where calls to random() return a double from the uniformly distributed range (0,1), max_weight is the maximum importance weight of all particles, and N is the total number of particles:

```
beta = 0.0
```

```
index = integer(random() * N)
for n from 1...N:
    beta += random() * 2 * max_weight
    while beta > particles[index].weight:
        beta -= particles[index].weight
        index = (index + 1) % N

    resampled_particles[n] = particles[index]
```

In English, choose a random value between 0 and the max importance weight, and choose a random index in the particle array to start at. For every particles, while the weight of the particle at that random index is greater than the random number (beta), step forward one index, and reduce beta by the weight at that index. Then, resample the particle at the index you end up at. Hopefully it is clear to see how particles with small importance weights are frequently passed over, and particles with large importance weights may be chosen multiple times in succession.

The way this algorithm is implemented in parallel ostensibly generates different results than the serial implementation, even with a pseudo-random generator. This is because an additional layer of randomness is introduced by the kernel, since threads will call the global random number generator in a non-deterministic order. Each thread has access to read all particles at once and builds their own subset of resampled particles which are combined into one vector of particles. At the end, the number of resampled particles is equivalent to the original number of particles. Interestingly, no mutexes or atomic operations are necessary during resampling, even though each thread uses the entire set of original particles in their resampling procedure. However, each thread must synchronize before the resampling, to ensure the importance weights are updated, and after resampling, to ensure the correct mean error is evaluated. It is currently unknown whether the increased randomness of the parallel implementation results in any change to the the asymptotic time complexity or the resampling quality. That is something that will need to be explored in future work, since there was no intention to measure resampling quality for this paper.

While the resmapling quality was not the focus of this paper, there are two important aspects to the resampling procedure that are necessary to know before properly interpreting the results. The first is that, due to the additional complexity, the importance weight calculation, and therefore the resampling algorithm, *ignores* the orientation of particles. As a result, certain random seeds will actually result in rapid divergence of the particles, but will eventually converge, after many resamples. Since the orientation is ignored, random resampling may occasionally result in a large majority of the particles close to the robot, but facing the wrong direction. Once the motion step occurs again, the particles and robot move in opposite directions, resulting in divergence. It is only due to motion noise that the particle group is able to gradually drift back to converge around the robot. In practice, there are ways to tune to particle filter to prevent this, at the cost of additional complexity.

Second, the motion noise discussed in Section 3.2 is necessary for any convergence, not just convergence under the initially divergent conditions. Consider the case of having no movement noise: the simulation is initialized as a uniform distribution as described previously. As the motion updates and resamplings occur, the number of unique poses begins to drop. This occurs because when any given particle is resampled twice, there are now two particles with the same exact pose. Without any movement noise, those two particles(and eventually many more if they are continually resampled) will always have equivalent poses, no matter how many motion updates occur. Introducing motion noise on a gaussian distribution solves this problem. After resampling, there are identical particles, but after another motion update, the particles will have slightly different poses. When many particles exist at the same location, this allows the particle cloud to drift toward the robot's actual location, since the particles whose noise brings them closer to the robot have a higher likelihood of being resampled. This is illustrated in Figure 3.

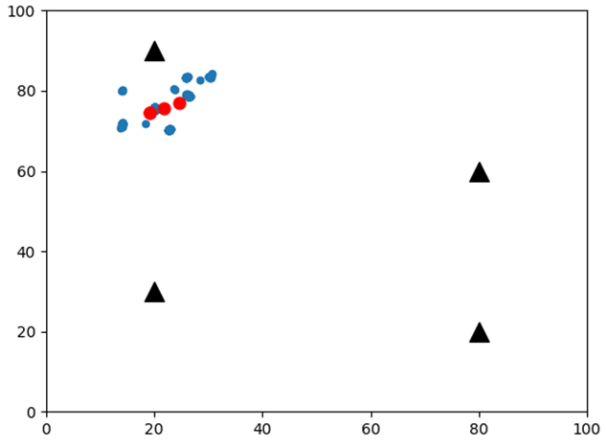

Figure 3: After a few resamplings

In this example, the robot is shown moving to the right. After two motion updates (the two previous robot locations are also shown), the particles have converged to the general area of the robot. Upon close inspection, it becomes clear that each particle location is actually a very small distribution. This is due to the previously described motion noise. The number of particles remains the same as the initial quantity, but their poses are close enough to visually overlap.

3.5 Evaluating the Mean Error

This step is a little different in the simulation than it would be on a real robot. To evaluate the mean error, the average Euclidean distance between the

robot and all the particles is used. This is possible in the simulation because the robot has a known location although it is ignored for all other intents and purposes. The mean error evaluation is a way to end the simulation after achieving convergence. In reality, particle filter localization would continue to run until the robot was done exploring its environment. Convergence may look something like what is shown in Figure 4.

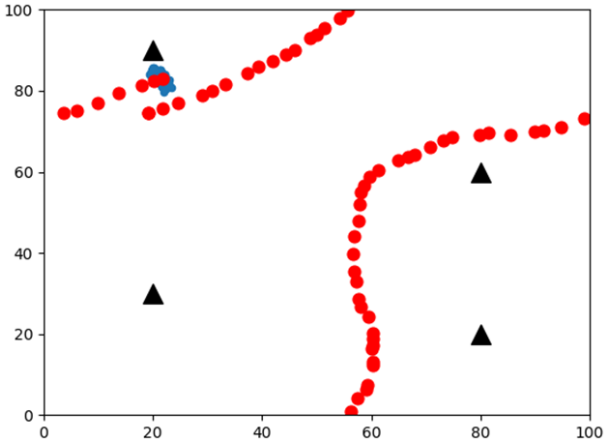

Figure 4: Solution converged

In this example, the robot starts around (20, 75) and starts moving to the right, passes through the top boundary, reappearing on the bottom, then passes through the right boundary, reappearing on the left, and converges when the robot is slightly above where it originally started. Note that the size of the robot steps are exaggerated for visual clarity. In reality, the particle filter can perform a few hundred updates per second.

4 Testing Methodology

The goal for this experiment was to observe the performance differences between multi-threaded and serial implementations of a Particle Filter Localization algorithm across different levels of parallelization and different scales.

4.1 Process

All data was gathered using RPI's Kratos system through SSH. Parallel tests were run from 2 threads to 64 threads, stepping up by powers of 2. For each thread configuration, separate tests were run for different particle counts ranging from 512 to 8192, also in powers of 2. The size of the world was set to half of the particle count for consistency and stability. This allowed for the particles to be initialized at roughly the same density for every run. For each test, both a parallelized and a serial version of the filter were run with the exact same starting conditions. The serial run acts as a control point to help reveal any inconsistencies in performance caused by specific starting conditions, and gives a baseline to measure the performance differences from.

We measured performance using the average time taken per step. Recording the overall runtime would be an inaccurate measurement of performance due to the non-deterministic nature of this algorithm causing the number of steps to be highly variable. By recording the time taken per step, we effectively remove this variability from our data. All turn counts were recorded along with the tests in case of future use. Every test is run in a world generated with exactly 12 landmarks. This is to help keep consistency throughout the tests. The number 12 was chosen to strike a balance between high computational costs and avoidance of the underflow issue discussed in Section 3.3.

4.2 Testing Problems

Our testing process was not without issues and setbacks due to both the nature of the subject matter, and inconsistencies in our testing environment. These issues were largely able to be compensated for.

4.2.1 Operating System Jitter

Kratos is, unfortunately, far from jitter free. Occasionally an update step would take significantly longer than usual, presumably as a result of daemons running in the background. This is further exacerbated by the algorithm's high reliance on random number generation. This means individual steps are relatively inconsistent in their runtime, and the number of steps taken before success is up to chance. We also noticed during testing that the very first step of every run tended to be longer than those that followed. This is likely due to the resampling method. When there are more particles with very low importance weights, resampling takes longer. Luckily our method of recording the average step time mitigates this, as the overall performance naturally evens out over many steps. We average a mean of 116.23 steps through all of our tests. 116 turns is enough to practically ignore most OS jitter and give us relatively consistent results.

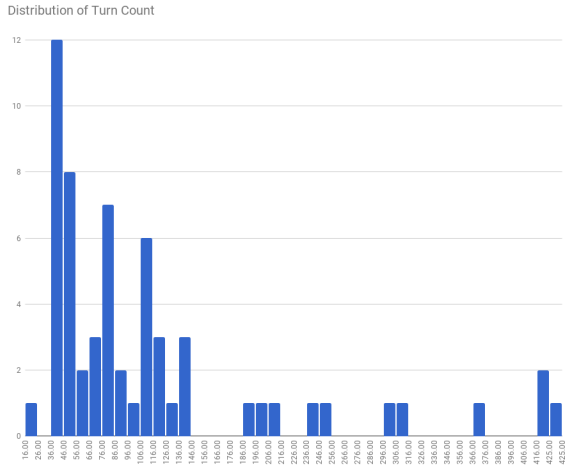

Figure 5: Distribution of Turns Across Tests

4.2.2 Seeding

The starting conditions of this algorithm, including the initial placement of particles, is determined by a Random Number Generator, which takes a seed that is provided on the command line. This seed has proved to greatly effect the behavior of the algorithm. Overall runtime can vary wildly depending on the what seed is paired with what hardware configuration and world size. Choosing the wrong seed can occasionally cause the algorithm to diverge instead of converge as discussed in the implementation, while other times the starting conditions will result in a world that converges after a single update. We tried to use the same seed value between tests whenever possible in order to preserve a sense of consistency, but different configurations often behaved differently under the same seed, and the results data were rarely consistent. Although it did not actually affect the time of an update step, runs that resulted in the divergence problem were ignored. All seeds were chosen completely arbitrarily, and recorded along with the tests.

4.3 Large Scale Instability

We observed that our largest test, the 8192 particle world, tended to yield somewhat inconsistent and unstable results. While the performance results were reproducible to a reasonable degree given the same seed, performance scaling across different hardware configurations and different seeds was somewhat unpredictable and resulted and data that does not completely follow the pattern of the rest of the data. We decided to include this data in our results regardless, as it still useful for demonstrating the overall performance trends when scaling the world size, along with certain behaviors caused by a large world. We believe that the a particle vector of that size filled the cache, resulting in longer memory access times. This was exacerbated by the random memory accesses that occurred in the multi-threaded runs, resulting in more page faults than in smaller runs. In reality, particle filters localization performs well with a few thousand particles, and 8192 is usually more than necessary.

5 Results

We will observe the results of this experiment from two views: The change in runtime as the world grows in scale, and the increase in performance of different parallel configurations relative to the serial implementation.

5.1 World Scaling

First, we compare the average runtime of each step at varying world sizes with all hardware configurations. The X axis is logarithmic as this more clearly represents the different world sizes. This means the Serial curve is actually relatively linear in reality, but is shown as an exponential curve due to the scaling.

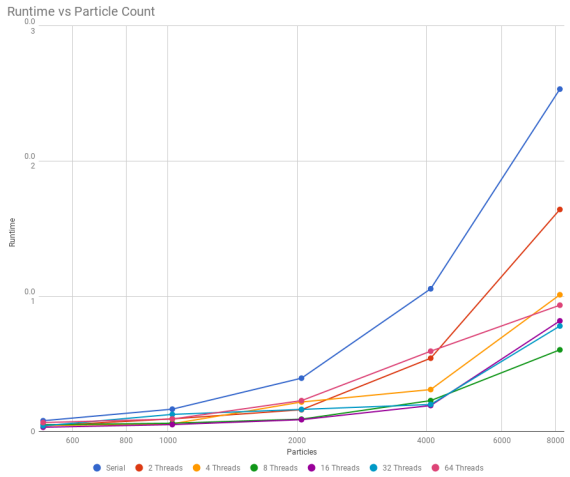

Figure 6: Runtime as as the World grows

While the threading solution is clearing significantly faster than the serial one, we can see that performance scaling is not entirely predictable. The best performing configuration for almost all world sizes is the 16 thread setup. This trend is broken by the 8192 particle world, where the order gets scrambled significantly, which we predict to be a result of the memory access issue discussed earlier. We can see the effects of diminishing returns here, as the change in runtime between each increase in threads decreases, until eventually increasing nodes begins to have a negative impact on performance. This is likely due the the overhead caused by managing a large quantity of threads overcoming the inherent performance benefits of running the algorithm in parallel. Naturally, the gap between configurations grows as the world size grows. This is to be expected, as parallelization divides a program's runtime, rather than subtracting from it. The higher thread configurations seem to result in curves that grow less steeply. Notably the 64 thread tests result in a curve that is almost linear in appearance, despite the graph having a logarithmic X axis. This is likely due to the high overhead of 64 threads slowing down the small world tests relative to other hardware configurations. It is not until the 8192 particle world that the 64 thread configuration is able to properly take advantage of its extra threading.

5.2 Impact of Parallelization

Next, we compare each parallel implementation relative to the serial algorithm using the absolute speedup equation $serial/parallel$. The serial data is added both as a reference point, and for visual consistency. Note that the serial bars are all 1.0, as they are being compared to themselves. Since our program is set up to run a serial version of each test with the exact same starting conditions, each data point is being compared to its serial counterpart. This gives us a more accurate measure of relative speedup, as

it eliminates any potential variation that could be caused by a difference in initial conditions.

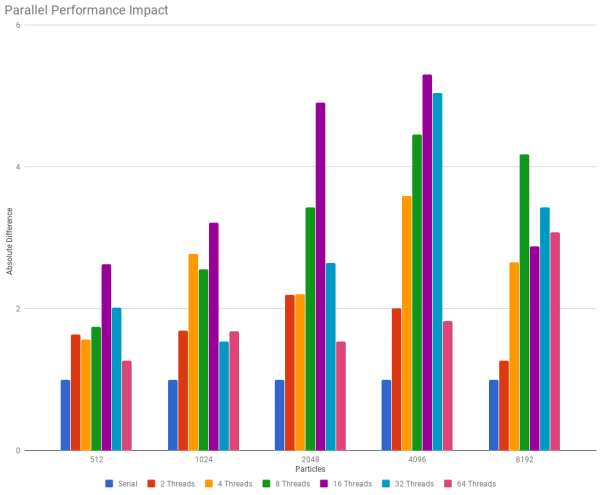

Figure 7: Performance Increase Relative to Serial

Notice that as the world size increases, so does the performance improvement caused by parallelization. This is likely due to the sheer amount of work done per thread overwhelming the inherent overhead caused by multi- threading. The exception to this trend is the 8192 particle run, which sees a marginal decrease in performance gain, which, as discussed previously, is possibly a result of the large world size causing issues with resource allocation and management. It likely doesn't help that using Kratos does not feature hardware optimizations for running Pthread programs at a large scale. This is an inherent limitation of our implementation, as most large scale parallelizations are using hardware optimized APIs on specialized machines, such as using MPI on Blue Gene. For our purposes, however, MPI was considered impractical to implement, as well as being not entirely relevant.

Hardware configurations with large numbers of threads seem to scale more steadily with larger worlds. In comparison, the lower thread tests tends to peak, at smaller sizes, then degrade with further increases in scale. The 64 thread tests are a great example of this, as they consistently lag behind other simpler configurations in almost every test, but suddenly see a dramatic performance gain relative to the other tests in the 8192 particle world. In fact, this is the only configuration we tested that saw a performance improvement at 8192 particles relative to 4096 particles. This makes sense, as the 64 thread configuration has by far the most overhead caused by parallelization, but potentially has the most raw processing power. By comparison, the 2 thread configuration only sees marginal, albeit consistent improvement across the board. This is because the low overhead of only 2 threads means these tests are always limited by raw processing power. The 16 thread setup seems to represent a happy medium for our algorithm and consistently outperforms the others by a significant margin in all but the largest world. The sudden massive drop in performance at 8192 particles is likely due to the previously aforementioned issues.

5.3 Potential Improvements

Our implementation of this algorithm is a work in progress and far from perfect. There are multiple ways we can further improve on the performance and accuracy of this algorithm.

5.3.1 Hardware Optimizations

Our choice of PThreads allows for use to take advantage of shared memory, and thus reduce message passing overhead. However, this comes at the cost of dedicated hardware optimization, and scalability at higher levels of parallelization. For better performance on mass parallel machines, an implementation that uses MPI with proper message passing would allow for better hardware optimization, and the ability better use several independent nodes. Supercomputing class machines are not relevant to the robotics world, but particle filters have relevance that far outreaches the robotics world.

5.3.2 Stability and Consistency

In an effort to improve performance, our implementation uses a non-deterministic approach with high synchronization. This is not a major issue as the accuracy of the result of this algorithm is not significantly affected by the randomness. This does, however, result in somewhat inconsistent results across multiple runs, and occasional stability issues. During testing, we ran into a problem where some seeds would cause the algorithm to diverge instead of converge. This is largely due to the random conditions of this algorithm, and the decision to ignore the orientation of particles. This could be fixed by incorporating the orientation into importance weight calculations.

6 Conclusion

The experiment shows a clear and substantial improvement in performance, even with configurations containing relatively few threads. The gap between the serial algorithm and the 2 thread implementation is enough alone to justify the conversion to a parallel approach. Performance is shown to scale smoothly all the way up to 16 threads with minimal overhead issues. Given that almost all modern CPUs support parallel processing, most commonly with 4 cores and 8 logical threads, our program is cable of significant performance improvements on the vast majority of consumer devices. At high scales, however, our implementation begins to fall short, although we note that 8192 particles is typically much more than necessary. This is largely due to the inherent limitations of Pthreads as for large scale operations, which could likely fixed by retooling our implementation to

utilize MPI, or another API better suited for mass parallelization.

References

[1] Brun, Olivier, et al. "Parallel Particle Filtering." Journal of Parallel and Distributed Computing, vol. 62, no. 7, 2002, pp. 1186–1202.

[2] Douc, R., and O. Cappe. "Comparison of Resampling Schemes for Particle Filtering." ISPA 2005. Proceedings of the 4th International Symposium on Image and Signal Processing and Analysis, 2005., 2005, doi:10.1109/ispa.2005.195385.

[3] Murray, Lawrence M., et al. "Parallel Resampling in the Particle Filter." Journal of Computational and Graphical Statistics, vol. 25, no. 3, 2016, pp. 789–805.

[4] Schwiegelshohn, Fynn, et al. "A Resampling Method for Parallel Particle Filter Architectures." Microprocessors and Microsystems, vol. 47, 2016, pp. 314–320.

[5] Sutharsan, S., et al. "An Optimization-Based Parallel Particle Filter for Multitarget Tracking." IEEE Transactions on Aerospace and Electronic Systems, vol. 48, no. 2, 2012, pp. 1601–1618.