ParSwarm: A C++ Framework for Evaluating Distributed Algorithms for Robot Swarms

Zhi Wei Gan
MIT
Cambridge
Massachusetts, USA
zgan@mit.edu

Grace Cai
MIT
Cambridge
Massachusetts, USA
gracecai@mit.edu

Noble Harasha
MIT
Cambridge
Massachusetts, USA
nharasha@mit.edu

Nancy Lynch
MIT
Cambridge
Massachusetts, USA
lynch@csail.mit.edu

Julian Shun
MIT
Cambridge
Massachusetts, USA
jshun@mit.edu

Abstract

Due to the increasing complexity of robot swarm algorithms, analyzing their performance theoretically is often very difficult. Instead, simulators are often used to benchmark the performance of robot swarm algorithms. However, we are not aware of simulators that take advantage of the naturally highly parallel nature of distributed robot swarms. This paper presents ParSwarm, a parallel C++ framework for simulating robot swarms at scale on multicore machines. We demonstrate the power of ParSwarm by implementing two applications, task allocation and density estimation, and running simulations on large numbers of agents.

CCS Concepts

• **Computing methodologies** → Shared memory algorithms; **Distributed algorithms**; **Massively parallel algorithms**.

Keywords

robot swarms, parallel algorithms, parallel simulations

ACM Reference Format:
Zhi Wei Gan, Grace Cai, Noble Harasha, Nancy Lynch, and Julian Shun. 2023. ParSwarm: A C++ Framework for Evaluating Distributed Algorithms for Robot Swarms. In *The 5th workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems (ApPLIED 2023), June 19, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3584684.3597269

1 Introduction

Social insects like ants and bees are able to achieve spectacular feats by distributing tasks to hundreds or thousands of individual agents. Researchers are becoming increasingly interested in modeling these insects [3, 4] and are even creating robot swarms of their own that achieve tasks that are only possible with hundreds of agents, such as "crop pollination, search and rescue missions, surveillance, as well as high-resolution weather, climate, and environmental monitoring" [1, 6].

Individual agents within robot swarms typically do not have sophisticated long-range communication capabilities. Thus, they are often unable to communicate with a central coordinator that tells them what to do or where to go. For this reason, algorithms for controlling these robot swarms are an emerging field that combines ideas from robotics and distributed algorithms. However, due to the complexity of robot swarm algorithms, analyzing their performance theoretically is difficult. Applying theoretical results to real-life experiments is also challenging because there are often many parameters that have to be tuned in order optimize the performance of the algorithm. Therefore, simulators are often used to benchmark the performance of robot swarm algorithms. However, most simulators run sequentially and do not make use of the naturally highly parallel nature of distributed robot swarms.

This paper presents ParSwarm, a highly parallel C++ framework for simulating robot swarms at scale using multicore machines. With the significant speedups that ParSwarm can provide to robot swarm simulations, we can efficiently simulate large swarms of agents. Researchers are able to quickly run experiments of different sizes without having to analyze theoretical bounds to draw meaningful conclusions about their algorithms.

The paper first introduces the theoretical model that is the basis for ParSwarm in Section 2 before diving into the framework itself in Section 3. Finally, we use ParSwarm to implement two applications, task allocation and density estimation, and benchmark their performance on large numbers of agents in Section 4.

2 Background

In designing robot swarm algorithms, simulators serve as an important intermediary step between mathematical models and real-life testing. The current state-of-the-art robot simulators support physics simulations [10, 12] and are compatible with real-world robots [13]. ARGoS [12] is a modular swarm simulator that allows different physics engines to be used with different parts of the environment. Stage [14] currently offers the best performance for large swarms [5], supporting up to 10^5 agents at 1/50 of real-time speed, which is slower than ParSwarm. In Stage, the highly parallel nature of the robots is not fully utilized since only the behavior of *stationary* agents can be evaluated in parallel.

ParSwarm aims to provide support for even larger scale robot simulations while maintaining the ability to emulate sensors working in 2D and 3D environments.

2.1 Modeling Robot Swarms

Cai et al. proposed a model for general robot swarm problems [3, 4]. This section gives a high-level introduction to this model, which will be the basis for the ParSwarm framework. The model is a probabilistic, synchronous distributed system. Its behavior is defined by the agents, the environment, and the agent-environment interactions that are specific to each experiment.

ApPLIED 2023, June 19, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0128-3/23/06…$15.00
https://doi.org/10.1145/3584684.3597269

Concretely, the model operates on an environment that is modeled as a directed graph $G = (V, E)$. For example, for an $N \times M$ 2D torus grid, a vertex $(x, y) \in V$ has edges to the vertices $((x+1) \bmod N, y)$, $((x-1) \bmod N, y)$, $(x, (y+1) \bmod M)$ and $(x, (y-1) \bmod M)$. A particular *configuration* of this model is described by some environment graph G as well as a set of agents that can manipulate vertices (locations) on the graph.

In particular, the system moves through a sequence of configurations using a sequence of probabilistic global transitions. These global transitions are resolved from a set of local transitions which are generated by agents.

Given some integer parameter I, each agent has a defined *local-mapping*, which maps vertices around an agent (the vertices $[m_x, m_y]$ such that $x - I \leq m_x \leq x + I$ and $y - I \leq m_y \leq y + I$ if the particular agent is at vertex (x, y)) to its associated information. Each agent then uses their *local-mapping* to generate a local transition using a user-defined function that describes the agent's new state, a new vertex state for the vertex it is currently on, as well as a new direction for the agent to move in.

Then, a user-defined resolution function accepts or rejects proposed local transitions generated by each agent. For example, if two agents are trying to move to the same vertex, some resolution rule may prohibit two agents from being on the same vertex at the same time. So, one agent's local transition would be accepted while the other's is rejected.

2.2 Parallelism

This paper will use the standard work-span model for analyzing shared-memory parallel algorithms. The ***work*** is defined to be the total number of operations in the computation and the ***span*** is defined to be the longest dependent path in the computation [8].

Prefix sum takes as input a sequence A of length n, an identity element e, and an associative binary operator +, and returns the sequence A' of length n where $A'[i] = e + \sum_{j<i} A[j]$ as well as the overall sum $e + \sum_{i=0}^{n-1} A[i]$. ***Filter*** takes a sequence A of length n and a predicate function f as input, and outputs a sequence A' containing all $a \in A$ such that $f(a)$ return true, in the same relative order as they appear in A. Both prefix sum and filter take $O(n)$ work and $O(\log n)$ span [8]. ***Semisort*** takes a sequence A as input, and reorders A such that equal-valued elements appear contiguously. The semisort algorithm runs in $O(n)$ expected work and $O(\log n)$ span with high probability [7].

3 ParSwarm

We introduce the ParSwarm framework, a high-performance parallel C++ implementation of the model defined in Section 2. We use C++ templates, which allows for user-defined code to be efficiently incorporated into the framework at compile-time instead of at runtime, which would be the case if polymorphism were used instead. We leverage the shared-memory multi-core programming paradigm, where each process spawns any number of threads that all have read/write access to any location in a global memory. The ParSwarm code is publicly available at https://github.com/zhiweigan/geo-swarm-framework.

The goal of the ParSwarm framework is to minimize the amount of parallel code that a user has to write to model complex agent functionalities. ParSwarm enables users to develop high-performance programs for simulating robot swarms while requiring only minimal knowledge about parallel programming. ParSwarm currently supports agents operating on both 2D and 3D torus grids.

Algorithm 1 Simulate

```
1: config = initialize new configuration
2: Add agents to config
3: Initialize vertices on config
4: while not config.is-finished() do
5:     config.transition()
```

Algorithm 1 shows the simplified pseudo-code for the the user-defined simulate file, which is the entry point for the program. An empty configuration has to first be created (Line 1). The empty configuration initializes an empty 2D or 3D torus grid with associated helper functions to add agents and edit vertices on the graph (Lines 2–3). Then, agents have to be added and specific vertex states can be changed. The program then enters the simulation loop (Lines 4–5). On each round of the simulation, the simulator runs the `transition()` function which is provided by the framework. After each transition, it checks if the configuration's user-defined `is-finished()` function returns true, and if so the simulation ends.

3.1 User-Defined Functions

For each agent, the user defines an internal state `astate` as well as two functions, `generate-message()` and `generate-transition()`. On each round, agents can optionally generate a message for other agents in its vicinity to read. Then, the `generate-transition()` function is given the agent's *local-mapping*, which contains the states of neighboring cells of the grid as well as the messages that were generated by agents that are currently positioned on those cells. The `generate-transition()` function proposes a transition for the agent, which consists of

(1) A new state for the agent,
(2) A suggested new vertex state, and
(3) An edge to a neighboring vertex for the agent to traverse in the next round.

Each vertex on the grid also has a user-defined internal state `lstate` that can be read and modified by each agent.

The configuration itself then has to take the transitions that the agents generate and resolve them using the `update-agents()` and `update-locations()` functions that the user defines. The framework parallelizes the transition generation as well as the agent and location updates behind the scenes without requiring any additional user code. This function is further detailed in Section 3.2. The user defines the `is-finished()` function, which determines when the simulation terminates.

3.2 Transition

This section introduces the function `transition()`. The function handles updating the overall configuration based on the proposed transitions generated by each agent. The implementation uses the parallel primitives introduced in Section 2.2.

Algorithm 2 shows the pseudocode for the transition function. Every iteration of the **parfor** loops can be run in parallel. None of the loops have different iterations that write to the same memory location, nor does any iteration depend on the execution of another. Each agent is equipped with its own random number generator, seeded with its identifier. The default `rand()` function in C++ has

contention on multiple threads, so we instead use a good hash function for randomness.

Line 1 semisorts the agents by their location, placing co-located agents contiguously in memory. Lines 2–11 determine the number of agents that reside in each location. Lines 2–5 keep track of the indices at which neighboring vertices differ. Since the agents are semisorted by location, the difference between any two adjacent indices is also the number of agents currently on that location. These counts are calculated in Lines 6–9. A parallel filter (Line 6) is first needed to make all non-zero elements of `diffIdx` adjacent to one another so that the counts can be calculated in parallel (Lines 8–9). The prefix sums on Lines 10–11 are needed so that `counts`[i] returns the starting index of the agents at the i'th location and `isDiff`[i] is used to index the location of the i'th agent. These computed arrays (`counts`, `offsets`, and `is-diff`) can be used to (1) find the number of agents at each location, (2) iterate through the agents at each location, and (3) determine the location of each unique location.

Lines 12–13 generate messages for each agent. Each location has an associated vector of messages, so Lines 14–16 deposit the messages at the particular location sequentially (this could be parallelized in theory). Lines 17–18 use the generated messages to generate transitions for each agent. On Lines 19–20, for each unique location, the framework iterates through agents on the location and accepts or rejects proposed agent transitions. On Lines 21–22, based on the accepted agent transitions, each agent is iterated over and their states will be updated accordingly.

The following theorem gives the work and span bounds of the transition function.

Theorem 3.1. *Let α be the number of agents, α_{max} be the maximum number of agents at a location, ℓ be the number of locations, U_w be the maximum work of a user-defined function, and U_s be the maximum span of a user-defined function. The transition function takes $O(U_w(\alpha+\ell))$ expected work and $O(U_s+\log(\alpha)+\log(\ell)+\alpha_{max})$ span with high probability.*

Proof. The semisort on Line 1 takes $O(\alpha)$ expected work and $O(\log\alpha)$ span with high probability. The parallel loop on Lines 2–5 takes $O(\alpha)$ work and $O(1)$ span. The filter on Line 6 and prefix sum on Line 11 take $O(\alpha)$ work and $O(\log\alpha)$ span. The loop on Lines 8–9 take $O(\ell)$ work and $O(1)$ span. The prefix sum on Line 10 takes $O(\ell)$ work and $O(\log\ell)$ span. The loops on Lines 12–13 and Lines 17– 22 takes $O(U_w(\alpha+\ell))$ work and $O(U_s)$ span. Lines 14–16 take $O(\alpha_{\max})$ work and span, assuming constant-sized messages. The total work and span of the transition function is $O(U_w(\alpha+\ell))$ and $O(U_s+\log(\alpha)+\log(\ell)+\alpha_{\max})$, respectively. □

The overall work and span of a ParSwarm simulation is the product of the bounds in Theorem 3.1 and the number of iterations until termination.

4 Experiments and Results

In this section, we present two applications, task allocation [3] and density estimation [11], implemented using the ParSwarm framework. For both simulations, we use a 2D torus grid environment. We were able to replicate these simulations with very few lines of non-boilerplate code (≈ 200 lines for task allocation and ≈ 50 lines for density estimation) while being able to easily scale to larger simulations with 10^7 agents.

Algorithm 2 Transition

```
1:  agents = semisort(agents, key=agent.location)
2:  parfor each agent i do
3:      if agents[i].location ≠ agents[i + 1].location then
4:          diffIdx[i] = i + 1
5:          isDiff[i] = 1
6:  offsets = filter(diffIdx)
7:  unique-locations = |offsets|
8:  parfor i from 1 to unique-locations do
9:      counts[i] = offsets[i] − offsets[i − 1]
10: startIdx = prefixSum(counts)
11: vtxId = prefixSum(isDiff)
12: parfor each agent i do
13:     msgs[i] = agents.generate-message()
14: parfor i from 0 to unique-locations do
15:     for j from startIdx[vtxId[i]] to startIdx[vtxId[i+1]] do
16:         deposit(&local-mapping, msgs[j])
17: parfor each agent i do
18:     transitions[i] = agents.generate-transition(&local-mapping)
19: parfor each unique-location i do
20:     update-location(i)
21: parfor each agent i do
22:     update-agent(i)
```

We ran our experiments on a machine with 24 cores, with a 2.2GHz Intel Xeon Processor (E5-2699 v4) and 96 GiB of main memory. We use parallel primitives from the ParlayLib [2] library. Our programs are compiled with g++ (version 7.5.0).

4.1 Task Allocation

The application showcases the parallel speedups that the framework is currently able to achieve. The setup of this problem is as follows. On an $N \times M$ torus grid with α agents, there are T tasks that have some demand $d > 0$. The demand of all tasks sum to a total of D.

Every agent starts at the same home vertex and follows a simple set of rules to traverse the grid until it finds a vertex that has a non-zero residual demand, and stays on that vertex until the simulation completes. For this experiment, each agent performs a random walk to traverse the grid until a task vertex with non-zero residual demand appears in its local-mapping.

Each agent has a destination task and a committed task that are both initially set to `null`. If there is a task nearby, the `generate-transition()` function sets it as the agent's destination task and moves towards it every round. Otherwise, it chooses a random direction and moves in that direction for a pre-defined number of rounds. If the agent is on a task vertex with non-zero residual demand, it proposes to commit to the task in its agent transition. If the `update-location(i)` function is called on a task vertex, it iterates over the agents that are on that location and accepts all transitions until the residual demand is reduced to 0. The updated residual demand is calculated at the end of the function. If i is not a task vertex or it is a task vertex with residual demand 0, then it accepts all transitions. The `update-agent(i)` function resolves all accepted transitions and sets the agent to be inactive if it has a committed task. The `is-finished()` function scans through the task vertices and agents, returning true if all vertices have zero residual

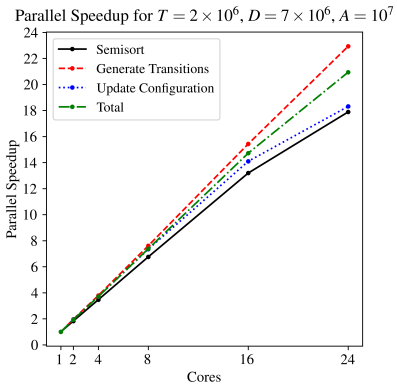

Figure 1: A plot of the parallel speedup for each part of the transition function, as well as the speedup of the overall code.

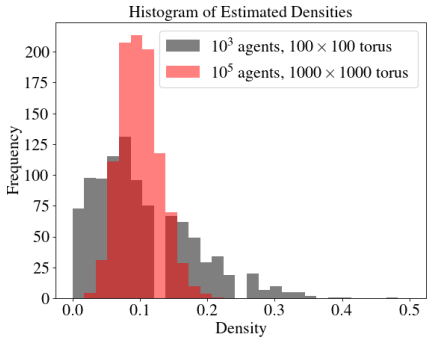

Figure 2: A histogram of the agents' estimated densities. The height of the red histogram has been scaled by 0.01 since there are 100× more agents in that experiment.

demand or if all agents are committed to a task. The framework function `generate-message()` is empty for this application.

On a 50×50 grid with 100 agents, 16 tasks and a total demand of 80, ParSwarm achieved a 500× speedup over a sequential Python implementation by Cai et al. [4], completing the simulation in 19 milliseconds (the Python implementation took over 10 seconds to complete). In addition to the speedup from parallelization, the speedup can also be attributed to a modified way of computing transitions. The sequential implementation iterates over every cell of the grid and operates on agents separately, while ParSwarm semisorts agents before each round so agents can find agents within its local mapping to generate transitions in parallel. This means ParSwarm does not need to iterate over the whole grid, which is much larger than the number of agents.

To benchmark the parallel speedup, we ran a simulation of the task allocation problem with a larger instance of the problem ($N = M = 1000$, $\alpha = 10^7$, $T = 2 \times 10^6$, and $D = 7 \times 10^6$) for 100 iterations on an increasing number of cores. We measured the time that it took to complete each part of the transition function (sorting, generating transitions, and updating agents and locations). We repeated each measurement 5 times and calculated the mean. This particular problem did not make use of message passing so Lines 12–16 were omitted. A plot of the parallel speedup is shown in Figure 1. Each trial on 1 core took 48 minutes, whereas it only took 2.2 minutes on 24 cores. The speedups of each of the steps as well as the overall speedup is at least 18, indicating very good scalability.

4.2 Density Estimation

We use the density estimation application to showcase the simplicity of the framework and the strengths of conducting an experiment with many agents. The application works as follows: given an $N \times M$ 2D torus grid with α agents, the agent density of the system is defined to be $\alpha/(N \times M)$. Each agent on this torus grid is tasked with estimating the agent density of the system. A simple algorithm for estimating the density was presented by Musco et al. [11].

In their paper, each agent runs a random walk in 4 directions on the 2D grid. At each step, each agent counts the number of agents that share its position and increments a variable c by that amount. At the end of T rounds, its density estimate is returned as c/T. According to Musco et al. [11], for any $\delta > 0$ the density estimate $\tilde{d}$ is within the range of $[(1-\epsilon)d, (1+\epsilon)d]$, where $\epsilon = \Theta(\sqrt{\log(1/\delta)\log(2T)/Td})$, with probability at least $1-\delta$.

The implementation for this application using ParSwarm took fewer than 50 lines of non-boilerplate code. Each agent's `generate-transition()` function creates a transition that simply chooses a random cardinal direction for the agent to walk in. `update-location(i)` iterates through every agent on its location and increments the number of agent collisions. Every agent transition is accepted. The `update-agent(i)` function resolves all accepted transitions. The `is-finished()` function returns true if the number of rounds has exceeded T. The framework functions `generate-message()` is empty for this application.

After implementing the problem in the framework for general N, M, α, and T, we chose to run two experiments: in the first experiment, we set $N = M = 100$ and $\alpha = 10^3$; in the second experiment, we set $N = M = 1000$ and $\alpha = 10^6$. In both experiments, $T = 50$ and the exact density is 0.1. The first experiment completed in 50 milliseconds, and the second completed in 4,500 milliseconds. Figure 2 shows the results of the density estimate of each agent for the two experiments.

In Figure 2, the gray histogram shows the density estimates for the first experiment. As we can see the distribution of the densities in gray is not centered around 0.1 as we expect them to be. The variation of densities is also not uniform and it would be difficult to tell that the experimental results align with the theoretical bounds that were produced.

However, this is not the case for the red histogram for the second experiment, where the mean is clearly around 0.1 and there is minimal variation around the mean. This experiment shows the strength of running an experiment with many more agents, which is enabled by ParSwarm, because it is easier to show that the density estimates are concentrated around the mean.

Musco et al. [11] gave proofs for the 3D torus density estimation case, which we confirmed with another experiment similar to this one. Musco et al. also discussed the possibility of noisy collision detection (i.e., detecting collisions with some probability p) and we found empirically that the density is around $p \cdot d$.

5 Conclusions and Future Work

We presented ParSwarm, a publicly-available parallel C++ framework that massively speeds up robot swarm simulations on 2D and 3D torus environments. We used ParSwarm to implement the task allocation and density estimation applications, and showed that our

framework is able to scale to large numbers of agents while achieving good parallel speedups. For density estimation, we showed that our empirical results match what is predicted by theory.

Kiszli et al. showed that simple swarm algorithms may fail when the number of agents increase past a certain threshold [9]. We plan to run more experiments with more agents for other algorithms to show behavior that may not otherwise be evident in a small number of agents. In the near future, we also plan to add more environments beyond the 2D and 3D torus.

Acknowledgements This research is supported by DOE Early Career Award #DE-SC0018947, NSF CAREER Award #CCF-1845763, and NSF Award #CCF-2003830.

References

[1] Florian Berlinger, Melvin Gauci, and Radhika Nagpal. 2021. Implicit coordination for 3D underwater collective behaviors in a fish-inspired robot swarm. *Science Robotics* 6, 50 (2021).

[2] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib - A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *ACM Symposium on Parallelism in Algorithms and Architectures*. 507–509.

[3] Grace Cai, Noble Harasha, and Nancy Lynch. 2023. A Comparison of New Swarm Task Allocation Algorithms in Unknown Environments with Varying Task Density. In *International Conference on Autonomous Agents and Multiagent Systems*.

[4] Grace Cai and Nancy Lynch. 2022. A Geometry-Sensitive Quorum Sensing Algorithm for the Best-of-N Site Selection Problem. In *Swarm Intelligence*. 1–13.

[5] Cindy Calderón-Arce, Juan Carlos Brenes-Torres, and Rebeca Solis-Ortega. 2022. Swarm Robotics: Simulators, Platforms and Applications Review. *Computation* 10, 6 (2022).

[6] Yufeng Chen, Huichan Zhao, Jie Mao, Pakpong Chirarattananon, E. Farrell Helbling, Nak-seung Patrick Hyun, David R. Clarke, and Robert J. Wood. 2019. Controlled flight of a microrobot powered by soft artificial muscles. *Nature* 575, 7782 (Nov. 2019), 324–329.

[7] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. 2015. A Top-Down Parallel Semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures*. 24–34.

[8] Joseph Jaja. 1992. *Introduction to Parallel Algorithms*. Addison-Wesley Professional.

[9] Zalan Kiszli, Seongin Na, and Farshad Arvin. 2022. Toward a Myriad Robot Swarm Aggregation. In *International Conference on Control and Robotics Engineering*. 1–4.

[10] N. Koenig and A. Howard. 2004. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vol. 3. 2149–2154 vol.3.

[11] Cameron Musco, Hsin-Hao Su, and Nancy A. Lynch. 2017. Ant-inspired density estimation via random walks. *Proceedings of the National Academy of Sciences* 114, 40 (sep 2017), 10534–10541.

[12] Carlo Pinciroli, Vito Trianni, Rehan O'Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni Di Caro, Frederick Ducatelle, Mauro Birattari, Luca Maria Gambardella, and Marco Dorigo. 2012. ARGoS: a Modular, Parallel, Multi-Engine Simulator for Multi-Robot Systems. *Swarm Intelligence* 6, 4 (2012), 271–295.

[13] E. Rohmer, S. P. N. Singh, and M. Freese. 2013. CoppeliaSim (formerly V-REP): a Versatile and Scalable Robot Simulation Framework. In *International Conference on Intelligent Robots and Systems*.

[14] Richard Vaughan. 2008. Massively multi-robot simulation in stage. *Swarm Intelligence* 2, 2 (01 Dec 2008), 189–208.