# Parallel Flocking with PFlockC

Joseph Horne

May 10, 2016

## 1 Team & Contributions

This project was done solo, with no teammates. The code is in my directory (PPChornej) in the "pflockc" directory.

## 2 Introduction & Motivation

The motivation for this project was taken from a 1995 paper by Professor Tamas Vicsek [5], working on conjunction with my old mentor from Oak Ridge National Laboratory, Miguel Fuentes-Cabrera. This type of simulation is now known as a "flocking simulation", inspired by the way each agent in the system interacts with neighboring agents–like a of birds flock (colloquially now known as "boids").

The idea was as follows: start with randomly generated boids, and set a few simple rules for the movement of each boid throughout the system. Although many different sets of rules have been studied, and have gotten more complex over time, the original paper Vicsek by used only one: a boid wants to move in the same direction as its nearest neighbors.

Although this is a simple rule, it can lead to surprisingly complex and realistic behavior, when studied in conjunction with some noise parameter $\eta$ (see [5]).

Although much work has been done in terms of extending this idea since, such as adding predators [1] and adding multiple classes of interactions [3], as well as understanding the behaviors in a more continuum sense [4] [2], most extensions of Professor Vicsek's original work have all been done in serial. While this approach works perfectly fine for small-scale systems, this ultimately limits what you can do. Here, my goal is to build a fairly general parallel flocking framework which can be built upon to study a larger variety of systems.

## 3 Implementation

### 3.1 Philosphy

Although the naive serial implementation of this behavior is simple, the simulation becomes much more complex when made parallel. The implementation here was done for boids in 2D space, although extension to 3D space is trivial. It should also be noted that periodic boundary conditions are in place, so when a boid goes out one of the box, it reappears on the other side. This is done to mimic an infinite 2D plane.

The general parallel philosophy is this: each MPI rank gets an exactly square chunk of space, and controls all the boids in that space and nothing else. In other words, it is the space that is split between ranks, rather than the boids. The reasoning for this will be discussed more down below.

### 3.2 Simulation Overview

Before discussing the implementation, some notation is defined. It is assumed there are $n$ boids in the simulation, $m$ MPI ranks, the required minimum distance between two boids for them to be considered neighbors is $r_{min}$, and the 2D simulation box is exactly square, and is defined by $\{(x, y) \mid x, y \in [0, l)\}$, for some input parameter $l$.

Each point in space is uniquely assigned to a rank in the simulation. Rank 0 is at the origin of the 2D plane in the simulation, and each progressive rank fills up the first row from left to right, then the second row, etc.

In order to ensure that all MPI ranks get spaces of exactly the same size, each MPI rank gets a square of size $l/\sqrt{m}$, having an area of $l^2/m$. This condition adds the restriction that $m = 4^x$, for some $x \in \mathbb{N}$. The program will not run if this condition is not true.

As the neighbors of a boid are defined by a radius around the boid $r_{min}$, and each MPI rank controls a set space, it is possible that the neighbors of some boid in the global simulation are in one of the surrounding ranks, if the boid is close to the edge of the rank controlling it. Assuming $r_{min} < l/\sqrt{m}$, the only possible neighbors for the set of boids in rank $i$ are the boids controlled by the immediately touching neighboring ranks for rank $i$. This condition adds the further restriction that $r_{min} < l/\sqrt{m}$, which is easily achievable for any reasonable simulation.

### 3.3 Running Time

A classical naive nearest neighbor search runs in $O(n^2)$ time, where each boid checks every other boid to see if it is within the required distance. Although there are many ways of reducing this, getting as low as linear $O(n)$ time, this approach requires *significant* extra complexity on top of our already complex parallel implementation, so here we do a naive nearest neighbor search.

However, due to the parallel nature of the simulation, we actually get better performance than the classic $O(n^2)$ algorithm. We get similar benefits automatically of solutions such as kd-trees by partitioning up the space into the ranks. If the boids are roughly evenly distributed, then there are, on average, $n/m$ boids per rank, and $9n/m$ neighboring boid candidates. Given this, the naive algorithm for our parallel system actually runs in $O(9(n/m)^2) = O((n/m)^2)$. This leads to superlinear speedups on strong scaling studies as $m$ increases, which is indeed what is observed (see performance benchmarks). Additionally, keeping the assumption that $r_{min} < l/\sqrt{m}$, any solution we implement to reduce the complexity of the nearest neighbor search (for example, using Tamas's original approach), will gain benefits from the natural space partitioning of the simulation.

### 3.4 Initialization

The initialization of all boids is done on one single rank, rank 0. This is because the cost of initializing boids is low ($O(n)$), and it was the easiest way to ensure it was possible to a natural distribution of boids. The naive implementation of parallel initialization, implementing $n/m$ boids on every $m$ rank, would distribute boids more evenly than would be natural than doing it serially on rank 0.

### 3.5 Iteration

At each iteration, several steps take place. Before any movement of the boids takes place, each boid needs to know who its neighboring boids are, which requires each rank to know who its neighboring ranks are. First, each rank calculates who its neighbor ranks are, taking advantage of the way the space is initially partitioned in the simulation (see above).

Then, because the number of boids controlled to each simulation is not constant, each rank needs to know how much memory to allocate for incoming boids, so, using the nonblocking calls `MPI_Isend` and `MPI_Irecv`, each rank sends its neighbor ranks how many boids it plans to send them, and receives how many boids it needs to allocate space for from the same ranks. After all this information is received, space is allocated and the actual boids are sent and received.

Once all neighbors are calculated, the rule stated above, where all boids want to move in the same direction as their neighbors, is enforced. This is done by a simple averaging of all the velocities from a boid's neighbors, which is then normalized to a value $v$, which is passed into the simulation as an input parameter. Once the new velocity $\vec{v}(t)$ is calculated and set, a random value in the interval $[-\eta/2, \eta/2]$ is added to *angle* of the velocity vector, taking care not to change the magnitude, and where $\eta$ is the tunable noise parameter mentioned above (note: $\eta = 0$ corresponds to a deterministic simulation). The boids are then moved according to $\vec{r}(t + dt) = \vec{r}(t) + \vec{v}(t)dt$, where $dt$ is a tunable simulation parameter.

As boids move through the simulation, it is possible they can move out of space controlled by the rank they are currently in. When this happens, after the position update takes place, if the boid is found to be outside of the bounds of its current rank, it is moved to the appropriate rank. This is done by going through the boids in each rank, and splitting them up based on a function that returns which rank is appropriate based on the boid's current position–if the boid's appropriate rank is its current rank, everything is fine, otherwise it needs to be moved into a list to be moved, and deleted from the current rank's boid list.

Once this is done, the "averaged normalized velocity" is calculated using Eq. 1, as per Tamas' original paper. The value $v$, as discussed above, the magnitude of the velocities of all boids in the system (tunable as a simulation parameter), $n$ is the number of boids, $\mathbf{v}_i$ is the velocity of the $i$-th boid, and $||\cdot||_2$ is the $L^2$ norm, otherwise known as the Euclidean norm. The reason this is calculated is because this gives an idea of how ordered the simulation is, which is typically compared to the noise parameter $\eta$. The reason this calculation is mentioned here is because you have to sum over the velocities of all the boids, which are spread across all the ranks.

The solution to parallelize this is fairly simple: since summations can be trivially split up, have each rank sum up its boids velocities, and then have one rank (say, rank 0) use `MPI_Gather` to get each rank's summed velocities, and sum those up, then divide by the appropriate value and print it out.

I want to point out that I considered using `MPI_Reduce` here, which is, after all, exactly the kind of thing its built for. I decided against it however, as the exactly number of boids per rank is indeterminate (and thus is the number of velocities to sum over for each rank), and trying to use `MPI_Reduce` in that situation would complicate things considerably. Therefore I went with the manual sum and gather approach, which works very

naturally if the number of boids on each rank is indeterminate.

$$v_a = \frac{1}{nv} \left|\left| \sum_{i=1}^{n} \mathbf{v}_i \right|\right|_2 \quad (1)$$

Although the functionality for parallel IO using `MPI_File_write_at` is built in, it isn't used. Since the point of the code is to be for massive scale simulations, writing out all the positions/velocities ofthe data in an attempt to visualize/analyze it is impractical, and also an extreme bottleneck on the system. The IO is written contiguously, so no preprocessing is required to look at the data, but again, since it isn't touched at all here, it isn't worth discussing, given the actual relevant parameter in the system is $v_a$, discussed above. In general, even in the original flocking papers (see [5]), only agregate statistics are relevant, which are best calculated within the simulation itself, rather than from an output file.

# 4 Performance

## 4.1 Initialization

As mentioned above, the initialization of boid positions takes place all on one rank, so that there is a realistic distribution of boids. Having an embarrassingly parallel approach where each rank generates $n/m$ boids isn't realistic, and having a parallel approach where each rank somehow generates a random number of boids that all sum to $n$ while keeping the distribution realistic would probably cause more overhead than it saves. Therefore, the initialization was done serially on rank 0, and then rank 0 sends the appropriate boids to the appropriate places, sending a couple messages per rank: one for the amount of data to be received, and one as the actual data.

Of course, this approach incurs a communication penalty, as the number of messages grows as the number of ranks increases. We see the amount of time the initialization takes below in Figure 1. Although the initialization time does grow roughly linearly as expected, it's worth noting that the actual initialization time is small (less than a second) even for 1024 ranks.

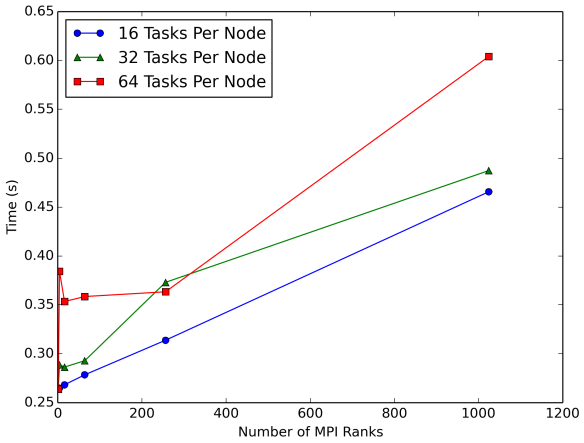


Figure 1: Initialization Time vs MPI Ranks

However, somewhat counterintuitively, the more tasks you have on a given node, the longer initialization takes. Although usually communication overhead is decreased as the number of tasks per node increases, the reason the trend is reversed here is due to the fact that, since communication for initialization is master-slave, and all communications go out "simultaneously", this floods the intranode bandwidth, and since 64 tasks per node get more messages than 16, the result is that 64 tasks per node takes longer.

## 4.2 Speedups & Nearest Neighbor

Due to the $O(n^2)$ scaling of the naive nearest neighbor search, the strong scaling study has some inherent limits. Since all runs (including serial runs) were to be done on the BG/Q, and even on the small allocation we are limited to a maximum of 60 minutes, the largest strong scaling study that could be done was 15000 boids for 200 timesteps, where the serial case runs in $\sim$50 minutes. Simulations several orders of magnitude larger are possible if the serial case is not used as a base-case for speedups.

As explained above, the number of MPI ranks used is required to be a power of 4, so the strong scaling study was done for 1, 4, 16, 64, 256, and 1024 ranks, with 16, 32, and 64 tasks per node, where 1024 is the maximum number of MPI ranks our allocation is allowed to use with 16 tasks per node. The speedup graph is shown below in Figure 2.

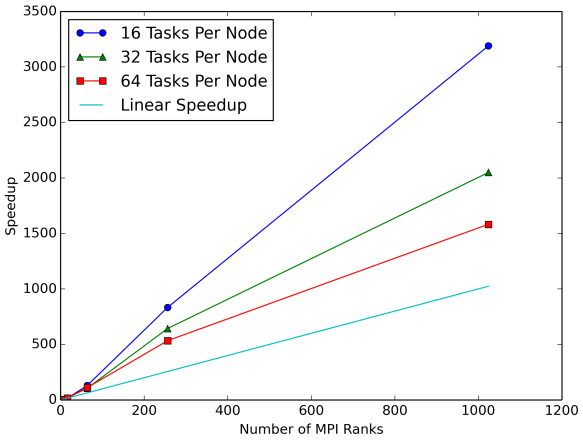


Figure 2: Speedup vs MPI Ranks

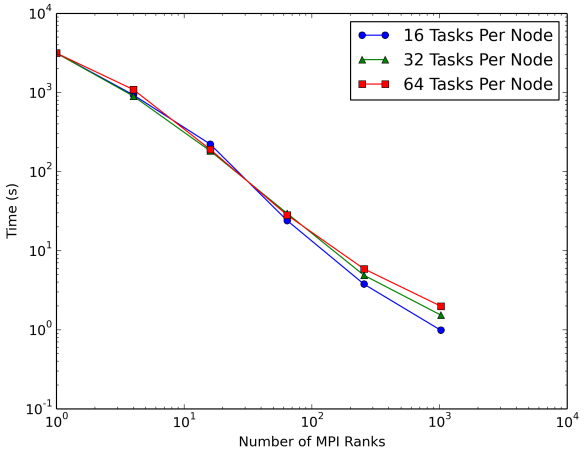


Figure 3: Total Compute + Communication Time vs MPI Ranks

The speedups are strongly superlinear, as expected, with a maximum speedup of $\sim 3190$ for 1024 ranks on 16 ranks per node. Its worth noting that a lot of this speedup is due to the naive implementation of the nearest neighbor search, and a different implementation would result in better performance, but smaller speedups, but this was beyond the scope of this project.

The reason for this extreme superlinear speedup is, as discussed above, actually algorithmic rather than parallel, although being parallel obviously helps. Since, for any normal parameter set, the neighbor cutoff radius is such that for a boid in a given rank, the only possible neighbors for that boid are in the boids own rank, plus the 8 ranks immediately touching it. This means that, for a strong scaling study, as the number of ranks increases, the number of possible neighbors for a boid decreases as $1/m$, which gives that the naive nearest neighbor search runs in $O(n^2/m^2)$ time instead of the classical $O(n^2)$.

It's important to note that this speedup is only algorithmic–it could be achieved serially, and sometimes is implemented using similar ideas. So in addition to this, we can split up each rank, running in $O(n^2/m^2)$ time for each iteration, onto separate cores (or hyperthreads), which gives us a parallel speedup in addition to algorithmic speedup.

Figure 3 shows the total time spent vs the number of MPI ranks. The figure is on a log-log scale to more clearly display how the time decreases as the number of processors increases. This figure similarly agrees with the speedup graph in Figure 2 in that the amount of time taken follows a power law (which superlinear is), since the points are roughly linear in the log-log space.

## 4.3 Communication

There is, of course, some communication overhead. The time spent in communication is somewhat more difficult to understand. There are three relevant figures: the total time spent in communication vs the number of MPI ranks in Figure 4, the communication time per rank vs the number of MPI ranks in Figure 5, and the and the fractional time spent in communication vs the number of MPI ranks in Figure 6, where the fractional time is (communication time) / (total time). All three figures are on a semi-log scale in the number of MPI ranks, so the increases in in the numbers used appear equally spaced and linear

Figures 4 and 5 show similar information, although Figure 5 is, in some sense, scaled more appropriately. Looking at Figure 5, we notice that for 1 MPI rank, there is obviously no communication time, and for 4 MPI ranks the communication time spikes (compared to 0), although the 64 tasks per node (TPN) case, not nearly as much. This shows up in both Figures 4 and 5. The reason for this is that, for 64 TPN, there are 4 tasks per processor. Here with 4 MPI ranks, all tasks get put on the same processor, and can take advantage of the extremely fast communication benefits that come with this. Although the simulation is slower overall (see Figure 3) for this case, the communication time is drastically reduced. Once the strong scaling study progresses to 16 MPI ranks, the 3 different TPN cases match up more closely, with both total time spent in communication being reduced, and communication time per rank being reduced. This reduction is mostly just due to the fact that the simulations take less time.

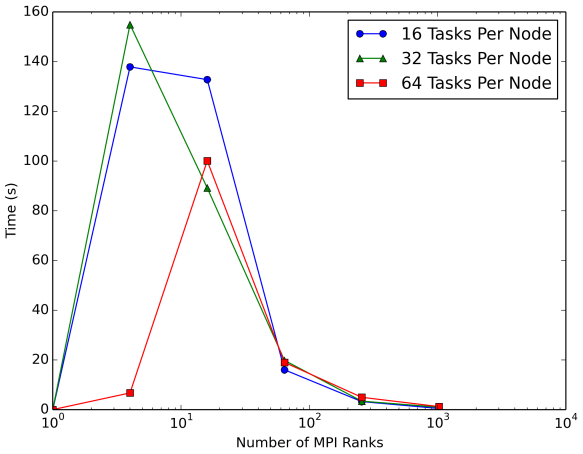


Figure 4: Total Communication Time vs MPI Ranks

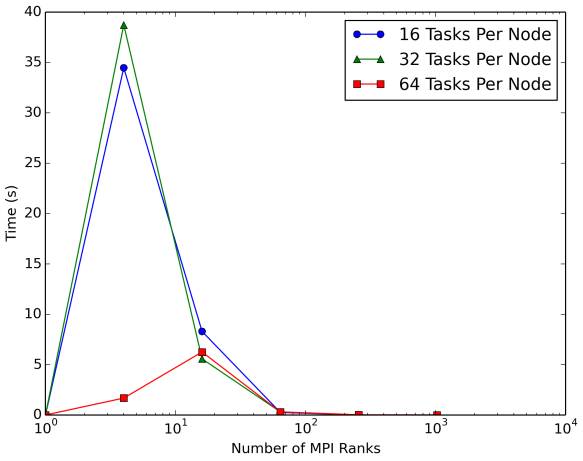


Figure 5: Communication Time Per Rank vs MPI Ranks

Although the total time spent in communication reduces, it is instruction to look at Figure 6, which shows the fraction of time the program spent in communication. This tells somewhat of a different story than Figures 4 and 5, where the total time spent in communication grows until the 1024 MPI ranks case. The anomaly where the 64 TPN case at 4 MPI ranks spends basically none of its time in communication is explained in the same way as above, where all 4 MPI ranks are on the same processor.

As the number of MPI ranks grows, so does the fraction of its time spent in communication. The reason for this is that, as the number of ranks grows, the amount of stuff each rank has to compute grows less and less, but the number of total messages sent across the simulation grows. The result is that the time spent in communication by the simulation steadily grows until it its 1024 MPI ranks. At 1024 MPI ranks, what ends up happening is that, since each rank communicates only with its 8 neighbor ranks, and the average number of boids per rank has steadily decreased, the message sizes about the boids being passed between ranks become small enough that communication becomes less of an issue.

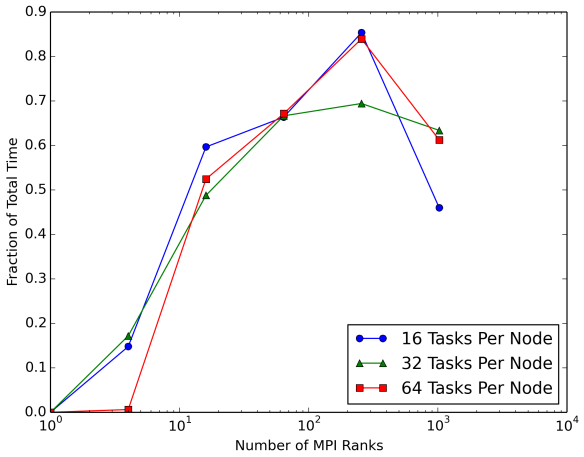


Figure 6: Fraction of Time in Communication vs MPI Ranks

# 5 Future Work

For future work, by far the largest, and easiest, optimization to make would make it so that the nearest-neighbor search runs $O(n)$ time, as opposed to $O(n^2)$. Although this would make the speedups less impressive, it would make the overall simulation run much faster. In addition, this is mostly just a framework for very basic parallel flocking, which has been studied to death. Most commonly, flocking studies published are done in serial, which limits the scale of things they can investigate. Using this parallel framework as a basis, more diverse systems of at scales previously uninvestigated could be looked at.

# References


[1] Thomas Caraco, Steven Martindale, and H Ronald Pulliam. Avian flocking in the presence of a predator. *Nature*, 1980.

[2] Seung-Yeal Ha and Eitan Tadmor. From particle to kinetic and hydrodynamic descriptions of flocking. *arXiv preprint arXiv:0806.2182*, 2008.

[3] Reza Olfati-Saber. Flocking for multi-agent dynamic systems: Algorithms and theory. *Automatic Control, IEEE Transactions on*, 51(3):401–420, 2006.

[4] John Toner and Yuhai Tu. Flocks, herds, and schools: A quantitative theory of flocking. *Physical review E*, 58(4):4828, 1998.

[5] Tamas Vicsek, Andras Czirok, Eshel Ben-Jacob, Inon Cohen, and Ofer Shochet. Novel type of phase transition in a system of self-driven particles. *Physical Review Letters*, 75(6), August 1995.