Investigation of High Performance Computing using Boid Simulation

Xuyang Fang
Level 7 MSci. Laboratory, Department of Physics, University of Bristol.
(Dated: February 14, 2020)

This report investigates the timings of Boid simulation using cell linked list data structure. The cell linked list was tested on N no. of Boids ranging from 10000 to 200000 using Apple MacBook with Intel i5 processor. The Boids are bounded in a space with periodic boundary condition. The size of the space is 20000^2, the maximum perception distance (p_{max}), velocity and acceleration of each Boid are 200, 40 and 5 respectively. When the cell linked list is unsorted, the best performance is achieved with only 5 cells per row instead of 100 (where the cell width equals p_{max}). The former reduces the time complexity to $O(N^{1.6})$, whilst the latter only to $O(N^{1.8})$. When the cell linked list is sorted, with 100 cells per row, the time complexity scaling is further reduced to $O(N^{1.4})$. The sorted cell linked list is then parallelized on Bluecrystal Phase 3 Linux cluster. When $N = 2000000$, MPI performed significantly better than OpenMP with same no. of processors. However, the efficiency either MPI or OpenMP decreases as no. of processors increases.

1. INTRODUCTION

The Boid model was developed to study the flocking behavior of animals. It assumes flocking is simply the result of the interaction between individual animals[1]. In the model each Boid obeys the following 3 rules within its maximum perception distance (p_{max}). It tends to avoid collisions with nearby Boids, match velocity with nearby Boids, and stay close to nearby Boids[1].

Such a model is usually chaotic and cannot be solved analytically for most initial conditions. But it can be studied using numerical simulations. Computing the N-Boid problem involves many independent yet similar calculations, thus it is particularly suited to be run on modern day CPUs. This is due to the so called instruction pipelining. For example, assume two independent floating point operations are given to a CPU. After finishing step 1 in operation 1, it can simultaneouly compute both step 2 in operation 1 and step 1 in operation 2, and so on forth. The length of modern CPU pipeline is about 20 steps. Therefore, calculation of interactions between each and every other Boid directly (brute force method) is efficient at small N.

However, the brute force method is not favoured at large N, since it requires $\frac{1}{2}N(N-1)$ calculations at each time step. Fortunately, the dynamics of the Boid model can be described by much less calculations. Two Boids would not interact unless they are within (p_{max}) apart, a number much smaller compared to the full simulation space. Thus, it makes sense to divide the full space into domains. One example is known as the cell linked list method which splits up the space into cells with unit width equals to p_{max} (See FIG. 1). Each Boid only considers the few others within its own cell and the 8 neighbours, and discards rest of the calculations. This reduces the time complexity scaling below $O(N^2)$.

Even with domain decomposition, the time complexity still scales non-linearly with N. To further reduce the computational time on one CPU core, one could instruct the compiler to execute independent floating point operations in parallel. This very complicated approach is known as instruction level parallelism[2]. Another way is to increase the power output of the CPU by increasing the voltage, which, unfortunately, would cause the core to overheat. A multi-core computing architecture can resolve both the two problems above. The computing power of a single core can be achieved by two cores operating under half the voltage. Therefore, the power dissipation of two cores is only a fourth of that of one core.

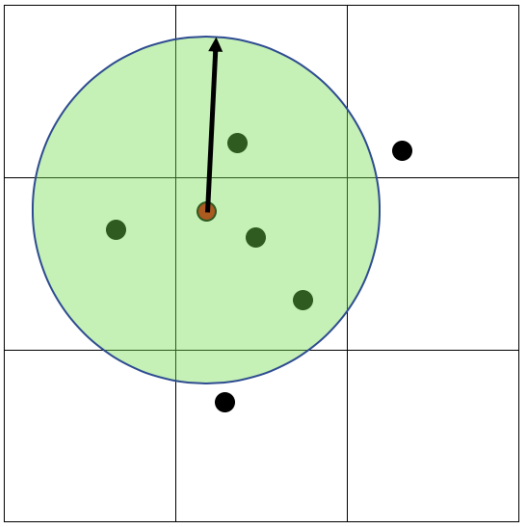


FIG. 1: Dividing up the space into cells. Each opaque coloured dot represents a Boid. The maximum perception distance of each Boid, marked by the black arrow, equals to the width of each cell. The Boid, marked in red, interacts with ever other boid within the green circle.

Multiple Instructions Multiple Data (MIMD) parallel architecture is particularly suitable for Boid simulation with domain decomposition. Each Boid needs to search through the data structure which stores the decomposition to find its nearest neighbours then compute the interactions. Each search consists of multiple instructions and is completely independent from the others.

A cluster is a parallel machine built on Single Program Multiple Data (SPMD) architecture, a subcategory of MIMD. It consists of a set of nodes. Each node has 2 sockets, each

socket has its own independent memory and 8 processors. Within a node, the memory of a SPMD program can be either shared or distributed. But across nodes memory can only be distributed. In strict distributed memory, each processor corresponds to one process, which can only access its own memory. Within a node, cross talks between processes are achieved through message passing via interconnect buses. Between nodes, messages are passed via interconnect networks. In strict shared memory, each processor corresponds to one thread, which uniformly access the memory of the others (up to level 3 cache within a socket[2]). Memory access of threads between sockets (also known as non-uniform memory access, or NUMA) are through the interconnect buses.

In light of the previous discussion, the report follows with discussions on the time complexity of the cell linked list method, and whether distributed or shared memory is more suited for this method.

2.1 DEVELOPMENT ENVIRONMENT AND TIMINGS FOR SINGLE CORE PROGRAMS

The program for Boid simulation on a single core was developed in C++. It is a compiling language that is lower level than Python, but higher level than C or Fortran. As a superset of C, it inherits C's flexible memory management. It supports user-defined data types (objects) that can contain private attributes and member functions. This enables more data hiding within the program itself, which reduces overall debug time.

For simplicity, the simulation is only limited to the 2-D case. The program complexity is further reduced by introducing Boid, a user-defined data type. It stores the position and velocity of the Boid it represents as floats. For the rest of the report, all Boid types/objects are referred to as Boid/Boids. All N Boids are declared on the stack as an object array with size N. Mention of the Boid index, unless specified, refers to the *i*th element of the Boid array. The initial position and velocity of each Boid are generated by sampling from the uniform distribution of a circle. This ensures the initial positions of the Boids are evenly spaced from each other instead of accumulating around the vertical and horizontal symmetry lines of the full space.

All executables intended to be run on a single core were compiled using Apple LLVM version 10.0.1 with following flags to increases the performance: *-O3*, *-funroll-loops*, *-march=native* and *-mtune=native*. All timings on single-core executables were taken using the Apple Instrument profiling tool. The computational time for a program to update positions of all Boids per unit time step were calculated by averaging the time to update Boid positions for no less than 20 time steps.

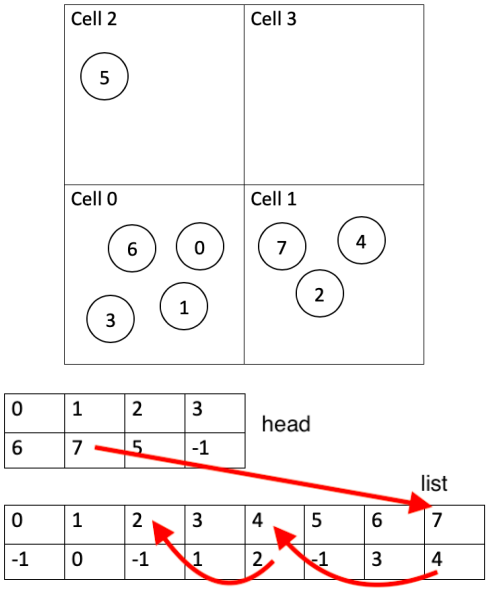


FIG. 2: Schematic diagram of cell linked list data structure. -1 in the "head" means that the cell which its index corresponds to is empty. -1 in the "list" means that the index it corresponds to is the last Boid in the cell. Each element in the "head" stores the integer that points to the Boid with the largest index within each cell. As the arrows have indicated, each element in the "head" also points to the Boid with the next largest index within the cell, and so on forward until the next element is -1.

2.2 CELL LINKED LIST DATA STRUCTURE

The cell linked list data structure used in the simulation program consists of two static 1-D integer arrays (See FIG. 2). Before the cell linked list is constructed, all elements of the two arrays are set to -1. When constructing the cell linked list, following steps are executed for all Boids.

1. Convert the spatial coordinate of the *i*th Boid into cell index c_i that it is located within.
2. The indices of "head" in FIG. 2 corresponds to c_i. So move the c_ith element in the "head" to the *i*th element of "list".
3. Change the c_ith element in the "head" to *i*.

The time complexity of the cell linked list construction is $O(N)$, meaning it is computationally cheap.

To compute the interactions of each Boid with the others in the 9 cells at each time step, for each Boid:

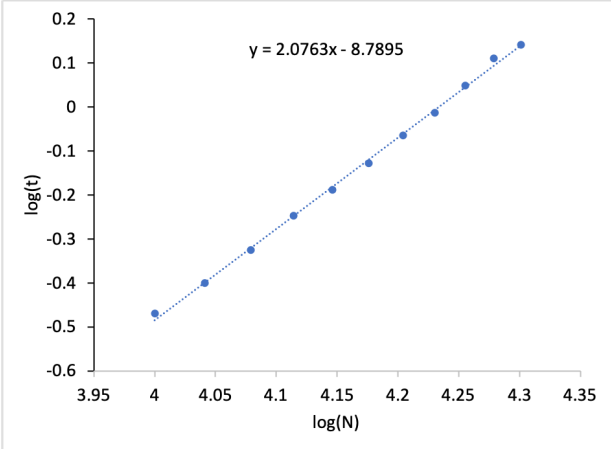
1. Convert the spatial coordinate of *i*th Boid into cell index c_i that it is located within.
2. Search through the data structure to compute the interaction with all Boids within cell c_i (except the *i*th Boid) following the red arrows in FIG. 2.
3. Repeat step 2 for the 8 neighbouring cells.

As FIG. 3 shows, the time complexity of the cell linked list is indeed lower than the brute force method. However, FIG. 4 shows that as the no. of cells per row is lowered, the computational time to update all Boids per unit time step first increases. It then drastically decreases when the no. of cells per row is smaller than 10.

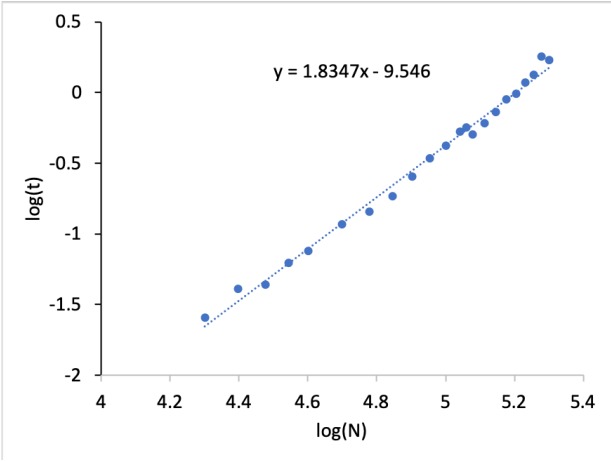
When no. of cells per row is lowered, the area of each cell scales as the inverse square of no. of cells per row. Thus, lowering the no. of cells per row increases the no. of Boids within each cell, which increases the no. of interactions calculations.

However, the preceding argument did not consider the cache misses during the search through the cell linked list. A modern CPU does not access the data stored in the main memory directly. Instead, the data are first moved into cache, a small block of memory. The CPU then fetches the data from cache. This results in the locality behavior of memory access. As FIG. 5 indicates, accessing data stored close to each other (within the same block) is faster, since the cache only needs to fetch one block from the main memory. However, accessing data stored far apart from each other requires the cache to fetch two blocks from the main memory, which slows down the program.

When the no. of cells per row is below 10, despite there are more Boids in each cell, the indices of them are closer together. This reduces the length of the jump indicated by the red arrows in FIG. 2 when searching through the "list", resulting in a speed up despite the increasing no. of calculations.



(a)



(b)

FIG. 3: A comparison of the time complexity of the Boid simulation using brute force method plotted in graph (a) and unsorted cell linked list with a cell width of p_{max} (100 cells per row) in graph (b). In both graphs, N is the total no. of Boids in the simulation and t is the computational time it takes to update the positions of all Boids per unit time step (measured in seconds). The gradient of the linear fitting is the time complexity scaling as N grows.

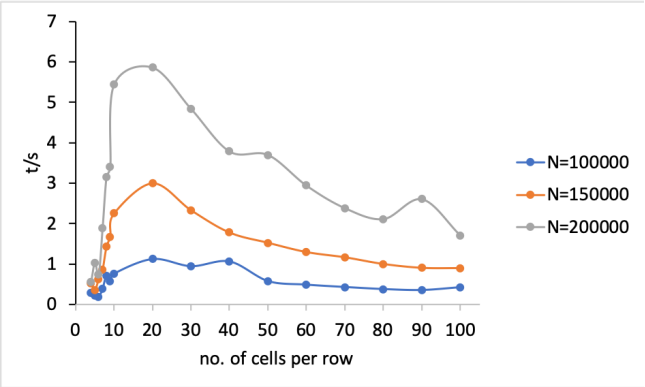


FIG. 4: A plot of the computational time it takes using unsorted cell linked list to update the positions of all Boids per unit time step (measured in seconds) against the no. of cells per row, which is the square root of the total no. of cells that cover the full space. N is the total no. of Boids.

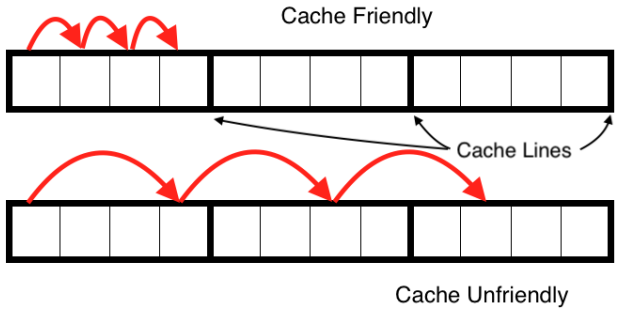


FIG. 5: Schematic diagram of cache-friendly and cache-unfriendly memory access. The size of the cache in this diagram matches a size-4 block (adapted from [2]).

2.3 SORTED CELL LINKED LIST

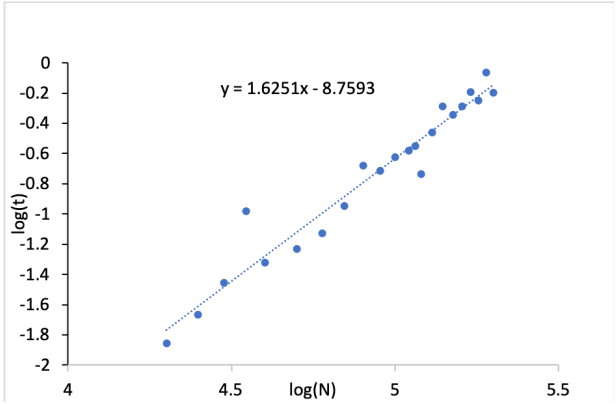
An immediate improvement is to shuffle the elements within the Boid array so that before the cell linked list construction, elements located in the same cells are moved next to each other. After the Boids are updated, the elements must be shuffled back to its original order so that the positions and velocities of all Boids can be written into the hard drive with consistency.

The preceding procedure was achieved using the C++ standard sorting algorithm *std::sort* which is discussed in more detail in FIG. 6. The time complexity of the algorithm is $O(Nlog(N))$ which is lower than $O(N^2)$, but higher than $O(N)$. It is not only computationally expensive, but also has to be executed twice.

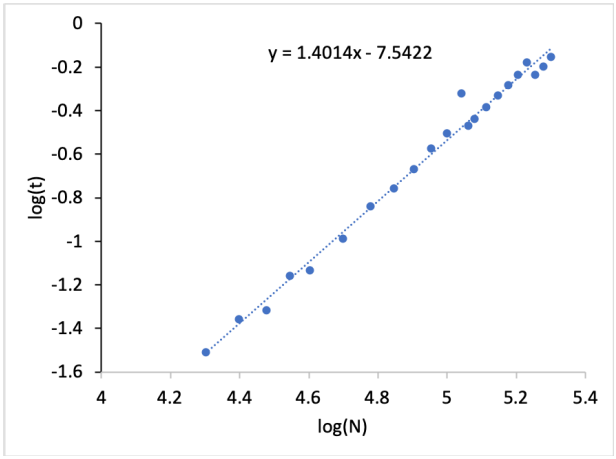
To access *std::sort*, the Boid array also has to be stored as *std::vector*, which is a dynamic container with contiguous memory stored on the heap. This causes additional slow down in the program as the memory access from the heap is slower than that from the stack.

FIG. 7 shows that the time complexity scaling of the sorted cell linked list is lowered to $O(N^{1.4})$, whilst the time complexity of the unsorted cell list is lowered to $O(N^{1.6})$ with 5 cells per row. Despite declaring memories on the heap and using the sorting algorithm twice, the sorted cell list still performed better. Both results in FIG. 7, when compared to FIG. 4, strongly suggests that cache-friendliness is the key to performance.

FIG. 6: A schematic diagram of sorting the Boids. The Array Index is the index to the container. Besides its position and velocity, each Boid also stores two integer Keys. Key 1 stores the original order of the Boid elements within the container. Key 2 stores the cell index that each element belongs. To ensure cache-friendliness of searching through the cell linked list, the container should be sorted by Key 2. To ensure that the positions of all Boids are recorded with consistency, the container should be sorted by Key 1.



(a)



(b)

FIG. 7: A comparison of the time complexity of the Boid simulation using unsorted cell linked list with 5 cells per row in graph (a), and sorted cell linked list with a cell width of p_{max} (100 cells per row) in graph (b). In both graphs N is the total no. of Boids in the simulation and t is the computational time it takes to update the positions of all Boids per unit time step (measured in seconds).

3.1 PARALLELIZATION DEVELOPMENT

The source code of Boid simulation with sorted cell linked list is then elevated into two parallel SPMD programs. One has strict distributed memory, and the other has strict shared memory. The shared memory program uses OpenMP, a C extension for parallelizing the "for" loops with threads. The distributed memory program uses Message Passing Interface (MPI), the standard interface for sending data between processes.

In source code for a single core, the procedure to compute each Boid interacting with the others in the 9 cells is completely independent from one another. Thus this part was chosen for parallelization in both the OpenMP and the MPI program.

One advantage of OpenMP over MPI is that only slight changes must be made to transform a single core program to a parallel one. The cell linked list with OpenMP only contains two additional *#pragma* statements that split up the "for" loop into even chunks. Each chunk is an array of integers with increments of 1. The size of the chunk equals the division of the no. of Boids over the no. of processors available. This ensures that each thread, corresponding to each processor, accesses the elements within the Boid container in a contiguous fashion, preserving the cache-friendiness. The OpenMP program was compiled with g++ 7.1.0, with *-fopenmp* flag since OpenMP is binded with the compiler, and *-O3*, *-funroll-loops*, *-march=native* and *-mtune=native* to ensure performance. The timings of OpenMP program was obtained using omp_get_wtime() function.

Whilst with MPI, a large portion of the program for the single core had to be altered to distribute the data and tasks evenly between processes. Buffers are created to allow data to be transferred between processes. The version of MPI used in the program is MPICH2. It allows the creation of an MPI user-defined data type that matches the memory block of a Boid. This allows arrays of Boids to be sent as buffers, which reduces the complexity of the program. The initial conditions for all Boids are set on process 0, or the master, then broadcasted once to every single process with MPI_Bcast. Then for each time step, each process executes the following procedure:

1. Sorts its own container by Key 2 in FIG. 6
2. Constructs its own cell linked list
3. Only updates the positions of a chunk of Boids
4. Acquires the updates on every other process using MPI_Allgather
5. Sorts its own container by Key 1 in FIG. 6
6. And finally, only the master writes the updates to hard drive.

The collective MPI communications are preferred rather than MPI_Send and MPI_Receive. This is because collective MPI communications are optimized with tree data structure, which reduces the overall cross talks between processes. The MPI program was compiled with mpicxx, with following flags *-O3*, *-funroll-loops*, *-march=native* and *-mtune=native* to ensure performance. The timings of the master process program was obtained using MPI_Wtime() function.

Both programs were run on Bluecrystal Phase 3 linux cluster.

3.2 SPEEDUPS AND EFFICIENCIES OF SHARED AND DISTRIBUTED MEMORY PROGRAMS

All results obtained in this section are calculated by measuring the computational time to update the positions of all Boids for 10 time steps. The speedup for p no. of cores is calculated as $S_p = T_1/T_p$, where T_1 is the computational time of the program executed by only one core and T_p is that by p cores. The efficiency of each core is calculated as $E_p = S_p/p$.

As FIG. 8 and 9 show, the speedup for both MPI (distributed memory) and OpenMP (shared memory) gradually approaches saturation. The efficiency of each process/thread decreases as the no. of processes/threads increases. The two programs have also shown better performance when the total no. of Boids increases. It seems that when there are more processors available or less parallelizable tasks present, the speedup of parallelization is limited by:

1. The communications between processors[3]
2. The work load on each processor that cannot be parallelized[3]

Both programs were then compiled in debug mode with the *-g* flag. They were then executed with Vtune Cluster Studio profiling tools 2015. The profiling tool suggested that for a simulation with 2 million Boids and 10 threads with OpenMP, 33% of the total run time would be spent on dynamically generating and destroying the threads at each time step. Whilst for the same no. of Boids running on 10 processes with MPI, 35% of the total run time would be spent on sending and receiving messages. Within the 35% MPI communication time, roughly 40% were spent on waiting for the other processes.

Two interesting things to notice are the dips in FIG. 8 (a) at 20 processes and FIG. 9 (b) at 2 threads. The dip in FIG. 8 (a) is due to the processes are distributed between nodes that are connected by the interconnection network, which is slower than the interconnection buses within each node. The dip in FIG. 9 (b) could possibly be due to the threads happened to be distributed between sockets within the node, where memories are only shared for processors within each socket. In this scenario, the memories of the two processors are actually localized. Accession of memory is via the interconnection buses. This requires non-uniform memory access (NUMA) between the two threads which results in an increase in memory access time.

As FIG. 8 and 9 show, overall MPI performed better than OpenMP when simulating the same total no. of Boids with same no. of processors available. This could also be due to the NUMA between threads.

4. POTENTIAL FURTHER INVESTIGATIONS

A few directions for further investigations includes the following:

1. Further reducing the no. of calculations at each time step. As FIG. 1 shows, a Boid only interacts with others within the green circle, which is only $\%35$ of the total area which covers the 9 cells.

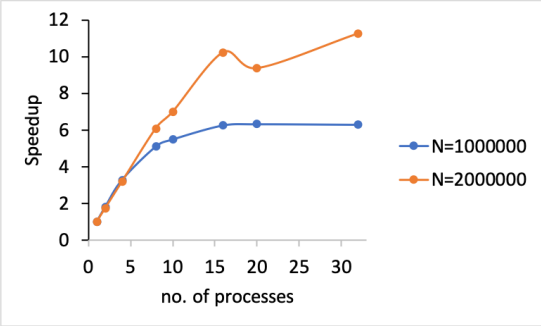
2. Reduce the work that cannot be parallelized on each processor by only partially instead of fully updates the cell linked list at each time step.

3. Investigate the speedup and efficiency of using both threads and processes within the same program.

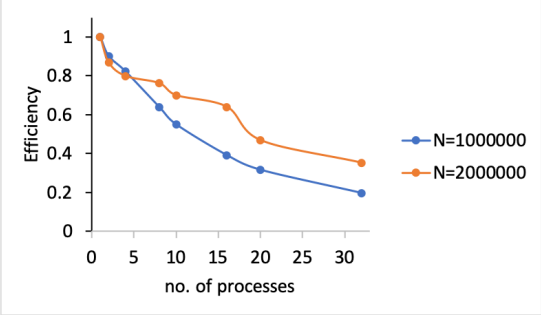
5. CONCLUSION

Overall, the Boid simulation with the unsorted cell linked list data structure performed better than the brute force method. Sorted cell linked list performed better than unsorted one.

The speedup in both the shared and distributed parallel programs are limited by communication and the work that cannot be parallelized. However, due to the specific implementation, the performance of MPI is better than OpenMP.

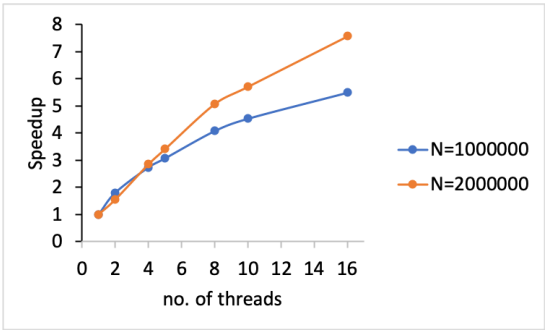


(a)

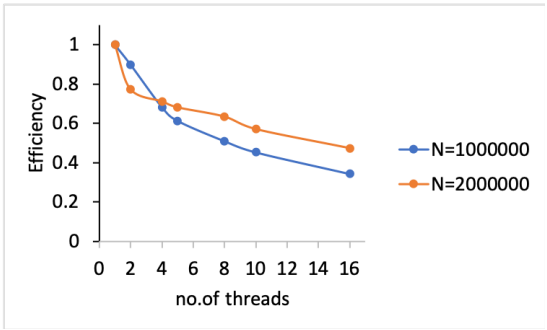


(b)

FIG. 8: Plots of Speedup and Efficiencies for Boid simulation with sorted cell linked list parallelized with MPI. N is the total no. of Boids in the simualtion.



(a)



(b)

FIG. 9: Plots of Speedup and Efficiencies for Boid simulation with sorted cell linked list parallelized with OpenMP. N is the total no. of Boids in the simualtion.

[1] Craig Reynolds. Flocks, herds, and schools: A distributed behavioral model. *ACM SIGGRAPH Computer Graphics*, 21:25–34, 07 1987.

[2] Victor Eijkhout, Robert van de Geijn, and Edmond Chow. *Introduction to High Performance Scientific Computing*. 01 2016.

[3] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery.