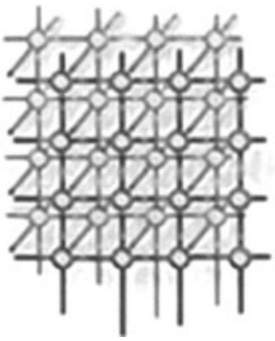
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2009; **21**:725–737
Published online 4 June 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1330



Parallel simulation for a fish schooling model on a general-purpose graphics processing unit

Hong Li[1,*,†], Allison Kolpas[2], Linda Petzold[1,3] and Jeff Moehlis[3]

[1]*Department of Computer Science, University of California, Santa Barbara, CA 93106, U.S.A.*
[2]*Department of Mathematics, University of California, Santa Barbara, CA 93106, U.S.A.*
[3]*Department of Mechanical Engineering, University of California, Santa Barbara, CA 93106, U.S.A.*

SUMMARY

We consider an individual-based model for fish schooling, which incorporates a tendency for each fish to align its position and orientation with an appropriate average of its neighbors' positions and orientations, in addition to a tendency for each fish to avoid collisions. To accurately determine the statistical properties of the collective motion of fish whose dynamics are described by such a model, many realizations are typically required. This carries a very high computational cost. The current generation of graphics processing units is well suited to this task. We describe our implementation and present computational experiments illustrating the power of this technology for this important and challenging class of problems.

Received 26 September 2007; Revised 4 January 2008; Accepted 10 February 2008

KEY WORDS: GPU; fish schooling; parallel computing

*Correspondence to: Hong Li, Department of Computer Science, University of California, Santa Barbara, CA 93106, U.S.A.
†E-mail: hongli@cs.ucsb.edu

Contract/grant sponsor: U.S. Department of Energy; contract/grant number: DE-FG02-04ER25621
Contract/grant sponsor: NIH; contract/grant number: EB007511
Contract/grant sponsor: Institute for Collaborative Biotechnologies; contract/grant number: DAAD19-03-D004
Contract/grant sponsor: National Science Foundation; contract/grant number: NSF-0434328
Contract/grant sponsor: Alfred P. Sloan Research Fellowship in Mathematics

1. INTRODUCTION

Many organisms display ordered collective motion [1], such as geese flying in a Chevron-shaped formation [2], wildebeests herding on the Serengeti plains of Africa [3], locusts swarming in sub-Saharan Africa [4], and fish schooling [5]. The number of individuals involved in such a collective motion can be huge, from several hundred thousand wildebeests to millions of Atlantic cod off the Newfoundland coast. Despite these large numbers, the group can seem to move as a single organism, with all individuals responding very rapidly to their neighbors to maintain the collective motion. Advantages of collective motion include a reduction in predation [6,7], increased harvesting efficiency [5,8], and improved aerodynamic efficiency [9,10].

The mathematical study of biological collective motion has proceeded, broadly speaking, on two fronts. First, one can model each organism individually, with rules specifying its dynamics and interactions with other organisms. Such 'individual-based models' can incorporate experimental observations of the behavior of the organisms, thereby giving biologically realistic models. On the other hand, one can define 'continuum models' to describe the dynamics of the population; for example, one might study a model for the evolution of the density of the organisms. However, available continuum models are often qualitative caricatures that cannot capture all the details that can be included in individual-based models, thereby compromising biological realism.

In this paper, we consider an individual-based model for fish schooling. This model incorporates a tendency for each fish to align its position and orientation with an appropriate average of its neighbors' positions and orientations, in addition to a tendency for each fish to avoid collisions. Furthermore, randomness is included in the model to account for imperfections in the gathering of information and in acting on this information. To accurately determine the statistical properties of the collective motion of fish whose dynamics are described by such an individual-based model, many realizations are typically required; see, e.g. [11]. This can entail a very large amount of computation. After describing the model in more detail, we will describe how a graphics processor unit (GPU) can be used to very efficiently carry out parallel simulations of this model.

With the low cost and high performance processing capabilities of GPUs, computation on general-purpose GPUs (GPGPU) [12] has become an active research field with a wide variety of scientific applications including fluid dynamics, cellular automata, particle systems, neural networks, and computational geometry [12–14]. Previous generation GPUs have required computations to be re-cast into a graphics application programming interface (API) such as OpenGL, which has made programming GPUs for non-graphics applications a significant challenge. NVIDIA's compute unified device architecture (CUDA) [15] is a new technology that directly enables the implementation of parallel programs in the C language using an API designed for general-purpose computation. In this paper, we will show how single instruction multiple data (SIMD) computation as well as parallel processing within a single realization can be implemented on a CUDA-enabled GPU to efficiently perform ensemble simulations of an individual-based fish schooling model.

2. FISH SCHOOLING MODEL

We consider a 2D individual-based model for schooling with local behavioral interactions. This model is similar to that of [16], but without an informed leader, and with different weights of

DOI: 10.1002/cpe

orientation and attraction response. Groups are composed of N individuals with positions $p_i(t)$, unit directions $\hat{v}_i(t)$, constant speed s, and maximum turning rate θ. At every time step of size τ, individuals simultaneously determine a new direction of travel by considering neighbors within two behavioral zones. The first zone, often referred to as the 'zone of repulsion' [11], is represented by a circle of radius r_r about the individual. Individuals repel from neighbors that are within this zone, which typically has a radius of one body length. The second zone, a 'zone of orientation and attraction,' is represented by an annulus of inner radius r_r and outer radius r_p about the individual. This zone also includes a blind area, defined as a circular sector with central angle $(2\pi - \eta)$, for which neighbors within the zone are undetectable. Individuals orient with and are attracted toward neighbors within this zone.

These zones are used to define the behavioral rules of motion. First, if individual i finds agents within its zone of repulsion, then it orients its direction away from the average relative directions of those within its zone of repulsion. Its desired direction of travel in the next time step is given by

$$v_i(t+\tau) = -\sum_{j \neq i} \frac{p_j(t) - p_i(t)}{|p_j(t) - p_i(t)|} \tag{1}$$

This vector is normalized as $\hat{v}_i(t+\tau) = v_i(t+\tau)/|v_i(t+\tau)|$, assuming $v_i(t+\tau) \neq 0$. In the case where $v_i(t+\tau) = 0$, agent i maintains its previous direction of travel as its desired direction of travel, giving $\hat{v}_i(t+\tau) = \hat{v}_i(t)$.

If agents are not found within individual i's zone of repulsion, then it will align with (by averaging the directions of travel of itself and its neighbors) and feel an attraction toward (by orienting itself towards the average relative directions of) agents within the zone of orientation and attraction. The desired direction of agent i is given by the weighted sum of these two terms

$$v_i(t+\tau) = \omega_a \frac{\sum_{j \neq i} \frac{p_j(t) - p_i(t)}{|p_j(t) - p_i(t)|}}{\left| \sum_{j \neq i} \frac{p_j(t) - p_i(t)}{|p_j(t) - p_i(t)|} \right|} + \omega_o \frac{\sum_j \hat{v}_j(t)}{|\sum_j \hat{v}_j(t)|} \tag{2}$$

where ω_a and ω_o are the weights of attraction and orientation, respectively. This vector is then normalized as $\hat{v}_i(t+\tau) = v_i(t+\tau)/|v_i(t+\tau)|$, assuming $v_i(t+\tau) \neq 0$. As before, if $v_i(t+\tau) = 0$, then agent i maintains its previous direction of travel.

We denote $r = \omega_o/\omega_a$ as the ratio of orientation to attraction tendencies. When $r = 0$ ($\omega_o = 0$), individuals have no desire to orient with their neighbors. As r approaches 1, individuals balance their orientation and attraction preferences. For $r > 1$, individuals are more interested in orientation with their neighbors than attraction towards them.

Stochastic effects are incorporated into the model by rotating agent i's desired direction $\hat{v}_i(t+\tau)$ by an angle drawn from a circularly wrapped normal distribution with mean 0 and standard deviation σ. In addition, as individuals can turn only $\theta\tau$ radians in one time step, if the angle between $\hat{v}_i(t)$ and $\hat{v}_i(t+\tau)$ is greater than $\theta\tau$, individuals do not achieve their desired direction, and instead

DOI: 10.1002/cpe

rotate $\theta\tau$ toward it. Finally, each agent's position is updated simultaneously as

$$p_i(t+\tau) = p_i(t) + s\hat{v}_i(t+\tau)\tau \tag{3}$$

3. THE GPU—A NEW DATA-PARALLEL COMPUTING DEVICE

3.1. Modern GPU

GPGPU is becoming a viable option for many parallel programming applications. The GPU has a highly parallel structure with high memory bandwidth and more transistors devoted to data processing than to data caching and flow control, compared with a CPU architecture, as shown in Figure 1 [15]. Problems that can be implemented with stream processing and that use limited memory are well suited to the GPU architecture. SIMD computation, which involves a large number of totally independent records being processed by the same sequence of operations simultaneously, is ideal for GPU application. We will show how the simulation of the fish schooling model can be structured to fit within the constraints of the GPU architecture.

3.2. NVIDIA 8 series GeForce-based GPU architecture

The NVIDIA 8800 GTX chip, released at the end of 2006, has 768 MB RAM and 681 million transistors on a 480 mm^2 surface area. There are 128 stream processors on a GeForce 8800 GTX chip, divided into 16 clusters of multiprocessors with eight streaming processors in each multiprocessor as shown in Figure 2 [15]. The eight processors in each multiprocessor share 16 kB shared memory which brings data closer to the arithmetic logic unit (ALU). The peak computation rate accessible for general-purpose application is (16 multiprocessors $\times$ 8 processors/multiprocessor) $\times$ (2 flops/MAD[‡]) $\times$ (1 MAD/processor-cycle) $\times$ 1.35 GHz = 345.6 GFLOP/s, as the processors are clocked at 1.35 GHz with dual processing of scalar operations supported. The maximum observed bandwidth between system and device memory is about 2 GB/s.

Unfortunately, current GPU chips support only single-precision floating point, whereas current CPUs support 64-bit technology. Additionally, there is only 16K of fast read and write on-chip memory shared between the eight processors on each multiprocessor.

3.3. CUDA

The CUDA software development kit supplies the general-purpose functionality for non-graphics applications on the NVIDIA GPU. The CUDA provides an essential high-level development environment with standard C language, which results in a minimal learning curve for beginners to access the low-level hardware. For development flexibility, the CUDA provides both scatter and gather memory operations. It also supports a fast read and write shared memory [15].

Following the data-parallel model, the structure of CUDA computation allows each of the processors to execute the same instruction sequence on different sets of data in parallel. The data can

[‡]A MAD is a multiply-add.

DOI: 10.1002/cpe

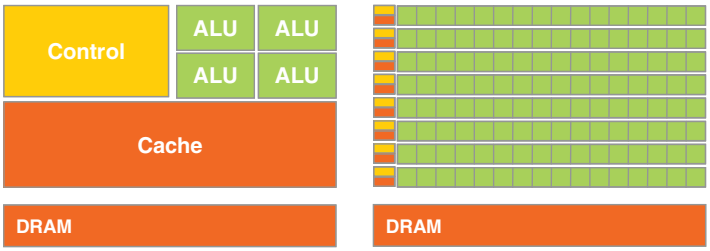


Figure 1. CPU vs GPU architecture. There are more transistors devoted to data processing rather than to data caching and flow control for the GPU, compared with the CPU.

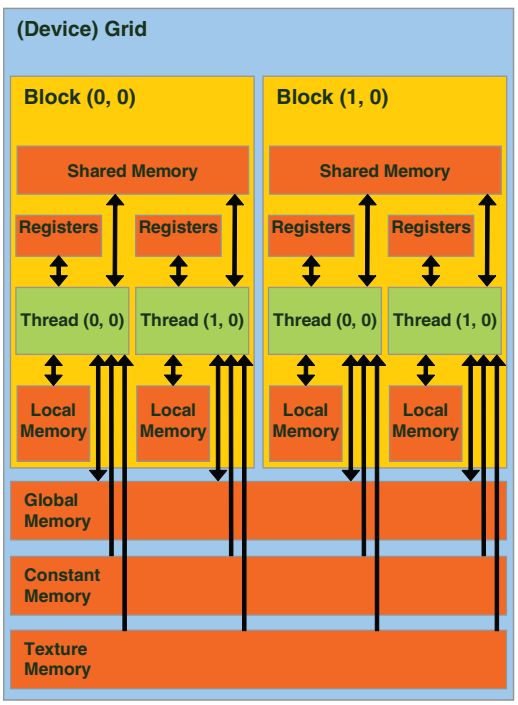


Figure 2. Hardware model: a set of SIMD multiprocessors with on-chip shared memory.

be broken into a 1D or 2D grid of thread blocks. Each block can be specified as a 1D, 2D, or 3D array of threads that collaborate via the shared memory. Up to 512 threads can be active.

Currently, the CPU and GPU cannot run in parallel. In addition, it is not possible to execute multiple kernels at the same time through the CUDA or to download data and run a kernel in parallel. Users can branch in the kernel based on the thread i.d. to achieve multiple tasks on a

DOI: 10.1002/cpe

single kernel, but the simulation will be slowed down with such branches. Individual GPU program launches can run at most 5 s on a GPU with a display attached.

4. PARALLEL SIMULATION

To accurately determine the statistical properties of the collective motion of fish, many realizations of the fish schooling model are typically required for a given set of parameters. This can be very computationally intensive. There are essentially two ways to improve the performance: parallelize the simulation across the realizations, and parallelize the simulation within one realization. We do both.

We also tried simulating our fish model on a cluster. We got a linear performance improvement for parallelization across realizations, but for parallelization within one realization, the message passing overhead is a substantial burden. One might also consider using a shared memory computer to improve the performance of parallelization within one realization, but for multiple realizations where many threads must access the shared resources there are likely to be serious problems with contention. The best part of the GPU is that it is *ours*. It resides on our workstation. We do not have to share it, schedule it, or find machine room space or a system administrator for it.

Migrating a C code from a workstation to the GPU is not a big job if you are familiar with programming multiple threads. Starting from a well-written sequential C code on our workstation, it took one of us about one week to port it to the GPU.

4.1. Random number generation

To generate the initial conditions and to add noise to our calculations at each time step, we need many uniformly distributed random numbers. It is particularly important that these random numbers are independent, so that our statistical results can be trusted. Generating independent sequences of random numbers is one of the most important issues of implementing simulations for ensembles of fish schooling models in parallel.

Originally, we considered pre-generating a sequence of uniform random numbers in the CPU. As the CPU and GPU cannot run in parallel, we can pre-generate a huge number of random numbers and store them in the shared memory, and swap back to the CPU to generate more when they are used up. Alternatively, we could pre-generate a huge number of random numbers and store them in the global memory. These methods spend too much time on data access. Furthermore, the Scalable Parallel Random Number Generators Library [17,18], which might have been an ideal choice because of its excellent statistical properties, cannot be moved to the GPU efficiently due to its complicated data structure. The only solution appears to be to employ a simple random number generator on the GPU. Experts suggest using a mature random number generator instead of inventing a new one, as it requires great care and extensive testing to evaluate a random number generator [19]. Thus, we chose the Mersenne Twister [20] in our application, which has passed many statistical randomness tests including the stringent Diehard tests.

In our implementation, we modified Eric Mills' multithreaded C implementation [20]. As many random numbers are required by this model, we use the shared memory for random number generation to minimize the data launching and accessing time.

DOI: 10.1002/cpe

4.2. Parallelizing across the realizations

Parallelizing the independent realizations is an effective way to improve the performance. Ensembles of fish schooling runs are very well suited for implementation on the GPU through the CUDA.

The whole fish schooling simulation can be put into a single kernel running in parallel on a large set of system state vectors $Px_i^j(t)$, $Py_i^j(t)$, $Vx_i^j(t)$, $Vy_i^j(t)$, respectively, representing the x and y components of the positions of the fish and the x and y components of the velocities of the fish, where i is i.d. of the fish, j is the block i.d., and t represents time. The initial conditions $Px_i^j(0)$, $Py_i^j(0)$, $Vx_i^j(0)$, $Vy_i^j(0)$ are randomly generated by different processors within each multiprocessor. The large set of final state vectors $Px_i^j(t_f)$, $Py_i^j(t_f)$, $Vx_i^j(t_f)$, $Vy_i^j(t_f)$ will contain the desired results. We minimize the transfer between the host and the device by using an intermediate data structure on the device, and batch a few small transfers into a big transfer to reduce the overhead for each transfer.

4.3. Parallelizing within one realization

Before discussing parallelism within one realization, we outline the sequential algorithm. Individual fish are initialized in a bounded region with randomized positions and directions of travel. Then the simulation is executed for $t_f = 3000$ steps to reach a steady state. At each step, we calculate the influence on each fish from all the other fish. To calculate the influence on a fish, we compute the distance from this 'goal fish' to all the other fish and record the influence coming from the different zones in a few variables. Next, we compute the net influences for this 'goal fish,' including the noise term, based on the above data, and save them to an influence array. After computing all the influences for all the fish, we update the position and direction of each fish based on the influence array of this step.

For the GPU, there are some restrictions on the number of threads in each block and the total number of fish stored in the shared memory. The device has very limited shared memory in each multiprocessor but relatively large global memory. The global memory adjacent to the GPU chip has much higher latency and lower bandwidth than the on-chip shared memory: it takes about 400–600 clock cycles to read/write the global memory vs four clock cycles to access the shared memory. To effectively use the GPU, our simulation makes as much use of on-chip shared memory as possible. Although it is better to have at least 128 threads for the best efficiency, we have to take the limited shared memory size into account.

The main goal of parallelizing within one realization is to decompose the problem domain into smaller subdomains. In our implementation, within each block j, the system state vector Px_i^j, Py_i^j, Vx_i^j, Vy_i^j is divided among threads. Assume that we have N fish and we use n threads to run the simulation. Then we need order of $m = N/n$ time to calculate the influences on each of the fish. In the parallel implementation, we initialize all the fish with random positions and directions. At each step we load n fish by loading one fish in each thread. The system state vectors loaded to shared memory at this stage are the 'goal fish.' Each thread holds only one 'goal fish' at a time and calculates the influences on this 'goal fish.' Thus, during one step, each thread processes one m-element subvector. To calculate the distance from the 'goal fish' to all the other fish, we need the position and direction of all the other fish. By using a temporary array of size n in shared memory,

DOI: 10.1002/cpe

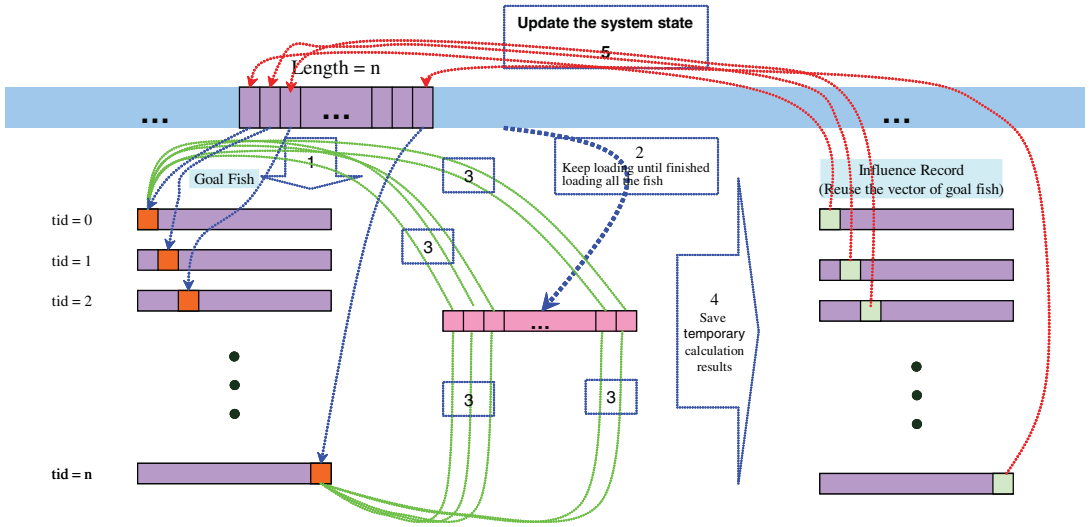


Figure 3. Memory layout of the fish schooling model. There are n threads to simulate one realization with thread i.d. *tid* to identify the thread. In process 1, each thread loads one fish as its 'goal fish' from the device memory to the shared memory; in process 2, each thread loads fish to another array in the shared memory; in process 3, each thread uses the full data from process 2 to compute the influences on its own 'goal fish'; processes 2 and 3 continue until all the influences to the 'goal fish' have been computed; in process 4, each thread saves its influence to the influence record array (this reuses the previous array to save the shared memory); in process 5, each thread updates its 'goal fish' to the system state vector on the device.

we load n fish at a time, with each thread loading one fish. The program will keep loading until all the desired data have been loaded and used for calculation for the goal fish. Then, each thread will compute the influence information for the goal fish of this thread. At the end of the simulation step, each thread writes the results into its own influence subvector. After all the influence calculations are finished, each thread updates its system state subvector by its influence subvector. The memory layout of the whole process is shown in Figure 3. Our implementation does not make use of a concurrent update. During each calculation, each thread reads the full system state vector. The concurrent reads to the same data are supported by the hardware.

Within each realization, most calculations use the shared memory, for which the memory access latency is small. In addition to the parallelization within one realization, we have many blocks doing independent realizations. Whenever a thread is waiting to access the device memory, another thread will be running on the ALUs. Hence, the memory access latency is not a problem.

5. RESULTS

Our simulations were run on the NVIDIA GeForce 8800GTX installed on a host workstation with Intel Pentium 3.00 GHz CPU and 3.50 GB of RAM with physical Address Extension.

To begin a simulation, individuals are placed in a bounded region (so that each agent initially interacts with at least one other agent), with random positions and directions of travel. The parameters

DOI: 10.1002/cpe

were fixed to be $r_r = 1$, $r_p = 7$, $\eta = 350^\circ$ (≈ 6.1 rad), $s = 1$, $\tau = 0.2$, $\sigma = 0.01$, and $\theta = 115^\circ$ (≈ 2 rad). Simulations were run for approximately 3000 steps until they reached a steady state. Groups of size $N = 100$ were explored for a range of ratios r of attraction to orientation weightings.

Two observables were used to measure the structure of the schools: elongation and polarization [16]. Elongation measures the ratio of the length of the axis aligned with group motion to the axis perpendicular to group motion, and polarization

$$P(t) = \frac{1}{N}\left|\sum_{i=1}^{N} \hat{v}_i(t)\right|$$

measures the degree of group alignment. Both quantities take values between 0 and 1. To obtain statistics regarding the group structure for a given size N and ratio r, 1120 realizations (with different initial conditions) were run. The average group elongation and polarization as well as the probability of group fragmentation were recorded.

We were able to nicely resolve the group statistics as a function of the ratio of attraction to alignment. We find that for r close to zero, groups exhibit swarm behavior, with small polarization and elongation near 1. As r is increased, groups become increasingly more aligned, forming elongated dynamically parallel groups and then highly parallel groups as r is further increased; see Figure 4. Thus, by changing the proportion of attraction to alignment tendencies, groups can shift between different collective behaviors. We also see that the probability of fragmentation first increases as the group elongation increases (from $1 < r < 2.5$), then decreases as elongation decreases (from $3 < r < 8$), and then increases again as schools become more highly aligned; see Figure 5.

The simulation performance in generating these results was extraordinary. We simulated 1120 realizations of the code with $N = 100$ individuals. The timing results for different parameters are summarized in Table I. The simulation time of the GPU-based code includes the time of loading data to the GPU and copying the data back to the host. The parallel (GPU) simulation is about 230–240 times faster than the corresponding sequential simulation on the host workstation.

The code on the host workstation was carefully written to reduce memory accesses and combine loops, and was the starting point of the GPU code. Streaming SIMD extensions were not enabled on the workstation operating system, in the way that most practitioners (including us!) would normally use a workstation. The workstation computation was performed in single precision. Even so, one might reasonably ask why is the GPU making so much better use of its CPU resources? The answer is multithreading. In our simulation code, there are many branches and memory access operations, besides the floating point operations. On the GPU, the scheduler can arrange for other threads to run while the current threads are accessing the global memory.

The limitation of single-precision floating point on the GPU seems to have no noticeable effect on the results, for this computation. Schools generated in double precision on the host workstation were found to be statistically indistinguishable from those obtained in single precision.

6. CONCLUSIONS

We considered an individual-based model for fish schooling, which incorporates a tendency for each fish to align its position and orientation with an appropriate average of its neighbors' positions and orientations, in addition to a tendency for each fish to avoid collisions. Randomness is

DOI: 10.1002/cpe

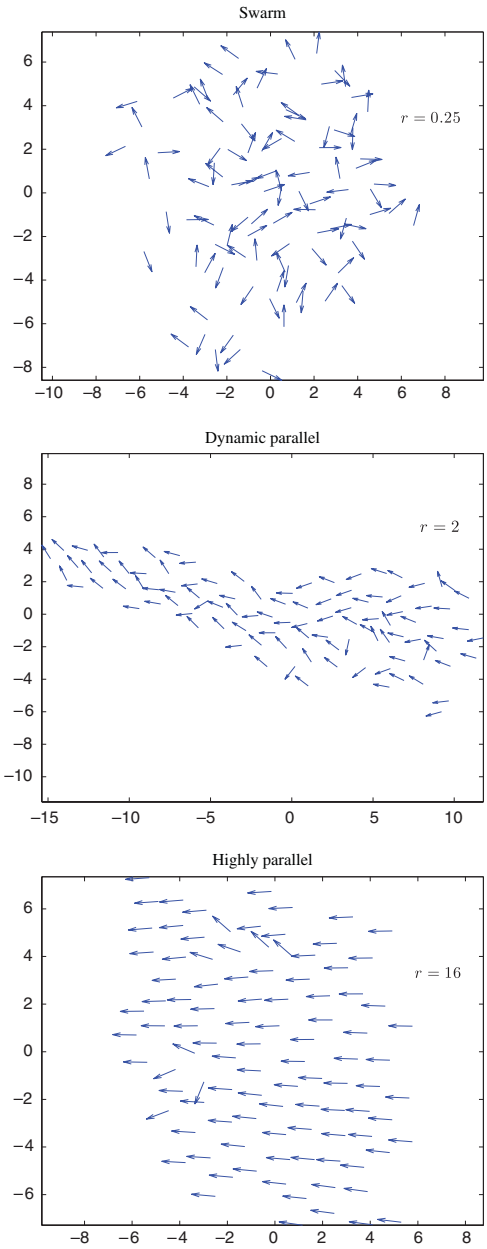


Figure 4. Swarm ($r = 0.25$), dynamic parallel ($r = 2$), and highly parallel ($r = 16$) collective motion for $N = 100$ member schools.

DOI: 10.1002/cpe

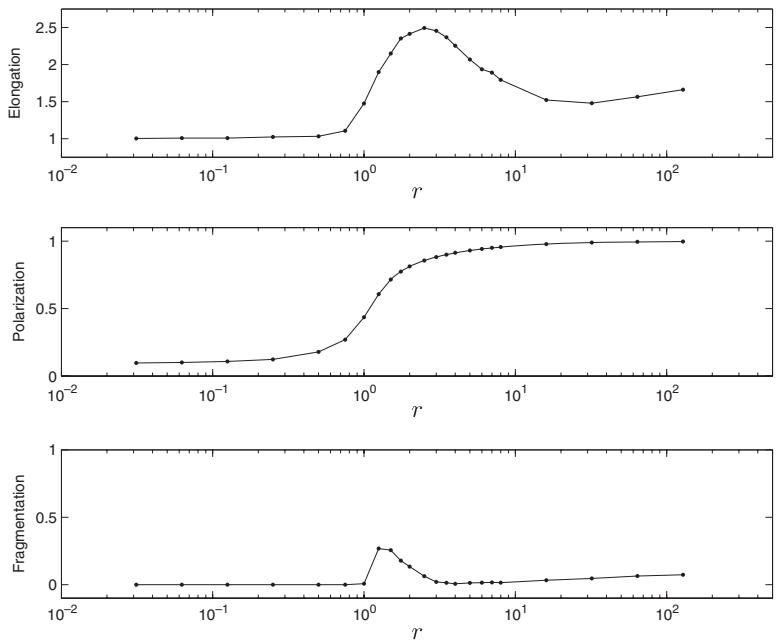


Figure 5. Statistics of group elongation (only including non-fragmented schools), group polarization (only including non-fragmented schools), and probability of group fragmentation, for $N = 100$ member schools. To obtain these statistics, for each value of r, 1120 simulations with different initial conditions were run for 3000 time steps, each using the GPU.

Table I. Performance comparison for different parameters.

Δr_o	Sequential simulation time (ms)	Parallel simulation time (ms)	Speedup
0.03125	7 924 067.2	33 750.6	234.8
0.0625	7 925 786.6	33 767.7	234.7
0.125	7 908 017.5	33 750.6	234.3
0.25	7 938 803.4	33 790.0	235.0
0.5	7 934 238.6	33 774.8	235.0
1	7 902 735.7	33 855.3	233.4
2	7 799 067.5	33 755.0	231.0
4	8 009 521.9	33 859.2	236.6
8	8 091 104.2	33 681.4	240.2
16	8 091 787.8	33 706.8	240.1

included in the model to account for imperfections in the gathering of information and in acting on this information. For the problem of calculating collective motion statistical measures from many realizations, we observed speedups of 230–240 times for our parallelized code running on a GPU over the corresponding sequential simulation on the host workstation. With this impressive

DOI: 10.1002/cpe

performance improvement, in one day we can generate data that would require more than six months of computation with the sequential code.

In the future we hope to simulate larger fish schools using clusters of GPUs. Our computation should scale well for parallelization across the simulations. Effective parallelization within one realization will depend, of course, on the communication speed of the cluster. The usefulness of individual-based models of biological systems, which can incorporate detailed experimental observations of the behavior of the organisms, is limited by the ability to simulate them in a reasonable amount of time. By exploiting the power of GPGPUs as illustrated in this paper, we expect that it will be possible to simulate more detailed models with more agents for longer times and over more realizations. This GPU computational paradigm has the potential to revolutionize our ability to understand how collective behavior at the macroscopic, population level arises from the interaction of individuals with each other and their environment.

ACKNOWLEDGEMENTS

We thank Iain Couzin for helpful discussions related to the model, and Brent Oster for introducing us to the NVIDIA GPU.

This work was supported in part by the U.S. Department of Energy under DOE award No. DE-FG02-04ER25621, by the NIH Grant EB007511, by the Institute for Collaborative Biotechnologies through grant DAAD19-03-D004 from the U.S. Army Research Office, and by the National Science Foundation Grant NSF-0434328. J. M. also was supported by an Alfred P. Sloan Research Fellowship in Mathematics.

REFERENCES

1. Camazine S, Deneubourg JL, Franks NR, Sneyd J, Theraulaz G, Bonabeau E. *Self-organization in Biological Systems*. Princeton University Press: Princeton, NJ, 2003.
2. Lissaman PBS, Shollenberger CA. Formation flight of birds. *Science* 1970; **168**:1003–1005.
3. Sinclair ARE, Norton-Griffiths M. *Serengeti*: *Dynamics of an Ecosystem*. University of Chicago: Chicago, IL, 1979.
4. Uvarov BP. *Grasshoppers and Locusts*. Imperial Bureau of Entomology: London, 1928.
5. Partridge BL. The structure and function of fish schools. *Scientific American* 1982; **245**:90–99.
6. Neill SRSJ, Cullen JM. Experiments on whether schooling by their prey affects the hunting behavior of cephalopods and fish predators. *Journal of Zoology* 1974; **172**:549–569.
7. Driver PM, Humphries DA. *Protean Behavior*: *The Biology of Unpredictability*. Oxford University Press: Oxford, 1988.
8. Martinez DR, Klinghammer E. The behavior of the whale *orcinus orca*: A review of the literature. *Zietschrift fur Tierpsychologie* 1970; **27**:828–839.
9. Wiehs D. Hydrodynamics of fish schooling. *Nature* 1973; **241**:290–291.
10. May RM. Flight formations in geese and other birds. *Nature* 1979; **282**:778–780.
11. Couzin ID, Krause J, James R, Ruxton GD, Franks NR. Collective memory and spatial sorting in animal groups. *Journal of Theoretical Biology* 2002; **218**:1–11.
12. GPGPU-Home. GPGPU Homepage. http://www.gpgpu.org/ [July 2007].
13. Owens JD, Luebke D, Govindaraju N, Harris M, Krger J, Lefohn AE, Purcell TJ. A survey of general-purpose computation on graphics hardware. *Eurographics 2005*, *State of the Art Reports*, Trinity College, Dublin, Ireland, August 2005; 21–51.
14. Li H, Petzold L. Stochastic simulation of biochemical systems on the graphics processing unit. *Technical Report*, Department of Computer Science, University of California, Santa Barbara, 2007; submitted.
15. NVIDIA Corporation. NVIDIA CUDA Compute Unified Device Architecture Programming Guide. http://developer.download.nvidia.com [July 2007].
16. Couzin ID, Krause J, Franks NR, Levin SA. Effective leadership and decision making in animal groups on the move. *Nature* 2005; **433**:513–516.

DOI: 10.1002/cpe

17. Mascagni M. SPRNG: A scalable library for pseudorandom number generation. *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, TX, 1999.
18. Mascagni M, Srinivasan A. SPRNG: A scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software* 2000; **26**:436–461.
19. Brent RP. Uniform random number generators for vector and parallel computers. *Report TR-CS-92-02*, 1992.
20. NVIDIA Forums members. NVIDIA Forums. http://forums.nvidia.com [July 2007].

DOI: 10.1002/cpe