Comparison of Cell and POWER5 Architectures for a Flocking Algorithm: A Performance and Usability Study

CS267 Final Project

Jonathan Ellithorpe  Mark Howison  Daniel Killebrew

ABSTRACT
We have parallelized and optimized an agent-based simulation of emergent flocking behavior for two shared-memory architectures: the POWER5 and the Cell Broadband Engine. The goal of this study was to explore the affordances and trade-offs of both architectures and their available APIs, especially for addressing the load-balancing issues caused by the uneven spatial distribution of flocking agents. We found that for this particular algorithm, the OpenMP API and POWER5 architecture provided higher performance. OpenMP also proved more usable because of its high-level constructs.

Keywords
Cell Broadband Engine, POWER5, OpenMP, shared memory, flocking, agent-based model, particle simulation

1. INTRODUCTION
In our agent-based model of flocking, individual flocking agents adjust their velocity and acceleration according to simple rules involving the properties of nearby agents. As the simulation evolves, these straight-forward interactions lead to an emergent flock formation. That is, flock formation can be viewed as a decentralized phenomenon resulting from individual agents’ behaviors; it isn’t necessary to have a central “leader bird” orchestrating the formation of a flock. Simulations of group behavior can often be cast as agent-based models, for instance in applications in virtual reality, computer games, robotics and artificial life [10]. Agent-based modeling is also an important methodology for the nascent field of complexity studies [2], and has found applications in research in a diversity of fields, from material science to social psychology [7] to evolutionary biology [3] to education [1].

Although there exist agent-based modeling environments (e.g., NetLogo [9]) and toolkits (e.g., Swarm [4]), they are aimed primarily at uniprocessor desktop/workstation architectures. As modeling needs scale, there will likely be an increased demand for optimized parallel implementations of agent-based models, especially with the rising availability of dual- and multi-core processors in workstations. This work is a step in that direction, exploring the obstacles to agent-based modeling on multi-processor architectures.

2. FLOCKING ALGORITHM
Our flocking algorithm consists of the following three phases:

Interaction. An individual agent’s neighbors are defined as the agents lying within a predefined sight radius and angle range (one eight the world size and ±100° in our implementation). An outer loop fixes an agent with position \( x \) and velocity \( v \), while an inner loop runs through that agent’s \( n \) neighbors to calculate the sum of differences vectors,

\[
dx = \sum_{i=0}^{n} dx_i.
\]

This aggregate difference vector is used to calculate four vectors based on following simple rules for agent interaction (see [5], [6], [8], and [10]):

1. Don’t crowd other agents. The crowding vector points away from the neighbors’ average heading, and is given by

\[
v_1 = \frac{-dx}{n_c|dx|^4},
\]

where \( n_c \) is the number of neighbors within the crowding radius (one sixteenth the world size in our implementation).

2. Align your velocity with your neighbors’ average velocity. The alignment vector,

\[
v_2 = \frac{1}{n} dx,
\]

points toward the neighbors’ average velocity.

3. Move toward the center of gravity of your neighbors. The local center of gravity vector is calculated as the average position of the neighbors minus the fixed agent’s position:

\[
v_3 = \left( \frac{1}{n} \sum_{i=0}^{n} x_i \right) - x.
\]
3.1 PERFORMANCE OF GRID IMPLEMENTATION

We parallelized the moving grid serial implementation for an 8-node POWER5 system using the OpenMP API. Our naive implementation uses a grid size specified at compile time, with the grid box size equal to the sight radius (an 8x8 grid). Initially, we parallelized the interaction procedure using OpenMP’s parallel for construct, which assigns an entire row to each thread. This row layout significantly impairs load-balancing, since a flock moving within a single row is assigned entirely to one thread. We addressed this limitation in our next implementation by using a Hilbert curve to assign grid boxes to processors (see Figure 2). This layout improves performance since a flock in any location and headed in any direction is likely to lie across grid boxes assigned to different threads. A look-up table for the Hilbert curve coordinates is calculated at run-time to reduce the overhead of resolving which grid boxes belong to which threads.

Even with the improved layout, an 8x8 grid has individual grid boxes that are large enough to contain nearly an entire flock for smaller flock sizes, given the fixed sight radius in the algorithm. In our final implementation, we added a runtime parameter for the grid size. The implementation was most efficient (see Figure 3) at larger flock sizes (weak scaling) and finer grid subdivisions, which increase the likelihood that each thread will own a similarly sized portion of the flock. However, for each grid box, we are initially allocating enough memory to hold the entire number of fish. Given our 48B agent structure and the additional 48B of swap space for each agent, a simulation with 2048 agent and a 32x32 grid requires a reasonable 48MB of memory, but scaling this implementation to say one million agents would require an unreasonable 24.5GB of memory. Clearly, an implementation capable of handling larger flocks would require a shared-memory reallocation mechanism that our current implementation lacks. The cache hierarchy of the POWER5 plays a role in performance, as well; the dip at 512 agents in Figure 3 coincides with the agent data exceeding the size of the POWER5’s L1 cache (32KB).

As the size of the flock increases, the proportion of time our implementation spends on moving/sorting the fish and on the reduction calculation of the next time step decreases (see Figure 4). The serial implementation, on the other hand, spends almost all of its time in the interaction phase. This profile explains why we were able to achieve 97% of peak performance for 2048 fish with a 32x32 grid; with those parameters, the algorithm’s profile is dominated by the interaction phase, which requires only shared-memory reads that take advantage of locally cached views of shared mem-

Figure 1: Performance of the moving grid vs. the naive serial implementation at n=50.

4. Move stochastically. A unit vector \( v_4 \) with random direction is calculated.

Using these vectors, each agent is assigned in swap space a new position \( x_{\text{new}} = C_1 v_1 + C_2 v_2 + C_3 v_3 + C_4 v_4 \) and a new acceleration \( a_{\text{new}} = M \cdot v_{\text{new}} \). In our implementation, these weights have the values,

\[
C_1 = 10^{-8}, \quad C_2 = 1.0, \quad C_3 = 0.1, \quad C_4 = 0.05, \quad M = 1.0,
\]

which were hand-tuned by trial-and-error, using animations of the resulting flocking behavior as feedback.

Time Step Calculation. A new time step \( dt \) is calculated as the inverse of the largest magnitude among all the agents’ position, velocity, and acceleration vectors.

Move. Each agent is assigned a new position \( x_{\text{new}} = v_{\text{new}} \, dx \) in swap space. The swap space then becomes the data for the next iteration.

Our serial implementation spatially decomposes the world into a moving grid that follows the agents’ center of gravity. In the serial implementation, the spatial decomposition reduces the complexity of the interaction phase of the algorithm. This is especially noticeable during flock formation, when agents are more uniformly distributed. Thus, Figure 1 shows a higher speed-up at shorter times of our moving grid serial implementation over a naive, non-spatially-decomposed implementation.

The moving grid has additional benefits for parallel imple-

mentations. When a stable flock forms, many of the agents are moving roughly in tandem with the center of gravity, which minimizes the frequency of reallocating agents from one thread’s grid box to another thread’s.

Because we are more interested in exploring the load-balancing issues that arise in non-uniform spatial distributions of agents, we created initial states in which stable flocks had already formed, and used these to seed the experiments reported hereafter.
Figure 2: The Hilbert curve layout for 8x8, 16x16, and 32x32 grid sizes.

Figure 3: Speed-ups for OpenMP implementations using different grid sizes and layouts.
Figure 4: The profile of time spent interacting the fish, calculating the next time step (using a reduction in the OpenMP version), and moving/sorting the fish.

Since every phase of the algorithm could be parallelized, peak performance should theoretically be attainable according to Amdahl’s law.

3.2 Performance of Quadtree Implementation

As an alternative implementation to the grid, we developed an algorithm based on quadtrees. The two dimensional space that the birds moved through was recursively broken up into quads dynamically, based on the number of birds in the quad. The intuition behind the quadtree is that it is a form of adaptive grid refinement. Where there are more birds, the grid becomes more detailed (see Figure 5). This follows from the primary rule of quadtrees: break a quad up into four constituent quads when the occupation of a quad reaches a predefined level.

The quadtree implementation took two parameters as inputs: the maximum number of birds in a quad before the quad was decomposed, and the minimum number of birds in a parent quad. Below this threshold, a parent quad would collapse the subquads beneath it, making itself the owner of all the birds. This minimum threshold serves to prevent birds from checking inside empty quads for interactions. It also prevents birds from checking multiple smaller quads for interactions, when checking a single larger quad would suffice. Optimizing these two parameters was difficult in practice, though. This is because the optimum value is really dependent on the spatial density of birds. If birds are very sparse, then a relatively low value for maximum birds would be desirable, because at low densities, the number of birds in a quad that are close enough to interact would also be low. If this maximum was maintained for a much denser flock, then a bird would have to check against many nearby quads for interactions, instead of just a few, when a few larger neighboring quads would have sufficed. The choice of minimum birds in a quad follows similar logic. In the end, we chose to err on the side of a smaller value for maximum birds, and a minimum value of about 70% of the maximum. The intuition is that it’s better to check against more quads than necessary, all of which contain birds within interaction range, than checking against quads with birds that are not within interaction range.

While the quadtree serves to limit the number of excess interactions that are calculated, it also provides a way of distributing birds to threads for computation. At the beginning of the simulation, the quadtree is created, and thread are assigned quads in such a way as to give all threads their fair share of birds. This results in each thread owning a certain area of the simulation space. If quadtree ownership was not redistributed from time to time, load imbalance would occur. Whether or not the simulation started randomly, or after flocking had already occurred, once the birds started moving, they would soon end up on one two processors, because they were so tightly clumped. This severe load imbalance would greatly hurt scaling, as shown in Figure 6. The balanced algorithm redistributes the quadtrees whenever any single thread indicates that too much imbalance is present (if a thread has more than 150% of the birds it started with, for example). This balancing algorithm demonstrates very linear speedup, in contrast to the algorithm which never performs quadtree reassignment. The downside of our re-balancing algorithm was that quadtree reassignment was done serially, limiting the potential maximum speedup of our algorithm. In practice, we did not have enough processors available to see the effect of this serial portion on our speedup.

One feature of our quadtree implementation that we thought particularly clever was the method in which birds that had moved out of their owning thread into another were dealt with. During the move phase, if a bird was detected to
move into another thread’s quad, it was put into a special data structure. Each thread has a multiple lists, one for each of the other threads. For example, if a bird in thread 1’s quadtree moved into one of thread 2’s quads, it was put into thread 1’s ‘list of birds moving to thread 2’. At the end of the move phase, all threads flushed data to memory to ensure consistency. Then each thread moved the birds that now belonged to myThreadID + 1 into the quads of myThreadID + 1. All threads barierred, and then transferred birds to myThreadID + 2, and so on. This resulted in maximal parallelization of birds being transferred between threads, instead of the naive algorithm where a single master thread performs interthread transfer.

3.3 Usability
Our implementation takes advantage of the built-in work-sharing, data-scoping, and synchronization constructs in OpenMP. Two features of our implementation required more customization. First, the C implementation of OpenMP has a limited number of built-in reduction operators, and we had to implement our own reduction for the time step calculation. Second, the default thread layout using the parallel for construct was not suitable for load-balancing, and we implemented the Hilbert curve layout described above. Third, we took advantage of the OpenMP threading model with the quadtree algorithm by giving each thread ownership of a set of birds, making it responsible for moving these birds. The shared memory model made calculating interactions across thread boundaries very easy, and threads were only synchronized when necessary to maximize performance.

4. CELL BROADBAND ENGINE
4.1 Iterative Refinement of Parallelization
4.1.1 Basic Parallelization of the Algorithm - Function-Offload Model
The flocking algorithm, at it’s heart, performs two operations each iteration. One is the calculation of the forces imposed on the agents by each other, and the other is using this force to change the agents’ velocities and positions. Our first attempt at parallelizing the algorithm was to use a Function-Offload model for the part of the algorithm that calculates agent forces (the interact_fish() function). Thus, at each iteration of the algorithm, the SPEs transfer their agents and neighboring agents from main memory into their local stores, calculate all forces on the fish that they own (apart of their assigned bucket) and write their agents in their bucket back to main memory. The PPE, in the meantime, waits until all SPEs have finished, then applies the calculated forces (the move_fish() function), sorts the fish into their new buckets, and repeats.

4.1.2 Extending the Functional Offloading
Our second version of parallelization involved offloading the move operation as well. The difficulty associated with this was that a gather-scatter operation was required on part of the PPE to send the SPEs the appropriate \( dt \), the amount of time to move forward in the simulation. In order to reduce error in the simulation due to using discrete time steps, the time steps themselves become smaller and smaller as the fish move faster and faster. To accomplish this, the algorithm looks at the maximum of all movement vectors in the simulation (velocity and acceleration), and uses them to calculate the current iteration’s \( dt \) (which is inversely related to the maximum of all velocities and accelerations). Thus, it becomes necessary for the SPEs that hold only a partial state of the algorithm to find the maximum of their own agents’ velocities and accelerations, report this to the PPE, and wait for the PPE to calculate and broadcast the iteration’s \( dt \). This gather-scatter operation is performed entirely by using the CBEA’s mailbox system, a way to communicate a single 32-bit message from the SPE’s to the PPE, or from the PPE to the SPEs. Thus, the PPE waits on all SPEs to send their information, and then the SPEs wait for the PPE to send them \( dt \). Given the ring structured interconnection network of the cell cores, a more efficient approach would have been to have the SPEs perform an all-to-all-scatter and calculate \( dt \) themselves, avoiding the overhead and latency of having the PPE send back its calculated \( dt \). Unfortunately doing this is easier said than done because the cell does not have default support for SPE to SPE mailbox communication. To do something like this requires setting up DMA transfers between pre-arranged areas of each other’s local stores.

4.1.3 Using more SPEs
The first versions of the code were limited to using only 4 of the 7 SPEs available on the cell platform on which we were working. Normally a cell processor will have 8 SPEs, however our development platform for this project was a Sony Playstation 3, for which the 8th SPE is disabled. We chose 4 because of its property as a perfect square, significantly simplifying the code’s implementation on the architecture. The reason for the simplicity of using a fixed instead of generic number of processors is due to the intimacy of the code with the hardware (or rather the lack of hardware abstraction). To generalize the code to use a parameterized number of SPEs involved careful hand coding of pointer passing and making sure that DMA transfers were always for a multiple of 16 bytes, a constraint imposed by the architecture. In fact, some grid layouts (using a 2x3 grid versus a 2x4 grid, for example) require manually editing the code to meet the 16 byte alignment constraint.

The development environment does, however, provide a construct for virtualizing the physical SPEs, called SPE “contexts”. If the number of contexts in use is greater than the number of physical SPEs then the cell context scheduler
will swap contexts in and out pre-emptively. This context scheduling inhibited our attempts to run 8 SPEs when using versions of the code that required all SPEs to communicate with the PPE, since a swapped out SPE would stall the program indefinitely. Thus, the maximum number of SPEs we were able to successfully utilize was 6 in a 2x3 grid.

4.1.4 Single-Precision versus Double-Precision Floating Point Arithmetic
The original flocking simulation used double-precision floating point arithmetic, although the cell’s SPEs are optimized for single-precision operations. Double-precision instructions have a 13-cycle latency, the first 6 cycles of which are unpipelined. Additionally, no other instructions can be issued (of any type) during these 6 cycles. Single-precision instructions, however, are fully pipelined and can be issued back to back for an IPC of 1.

4.1.5 Shifting the dt Calculation
Without sacrificing any perceivable amount of simulation accuracy, we circumvented the reduction of \( dt \) by simply calculating \( dt \) before the SPEs grab the fish from memory and calculate accelerations. Our studies on the error caused by doing this revealed that when 1000 steps of the simulation were run, it was at the 326th step where we saw the first difference in \( dt \). Even then the error amounted to only 0.003% (see Figure 7).

In any case we consistently monitored code performance for only 100 steps for each version of the cell code, and so experienced the same \( dt \) values as when we did a proper reduction of the value.

4.1.6 Moving from Function-Offload Model to Streaming Model
For simplicity of synchronization, our attempts at parallelization involved using the SPEs in essentially a remote-procedure call manner. Each iteration created new contexts, loaded the code, ran the code, and destroyed the contexts. In moving to a streaming model, it became necessary to synchronize all the SPEs at every time step. This was done using simple mailbox messages, where the PPE sends either a “GO” or “STOP” message to the SPE. The SPE will then DMA the new agent data into its local store, process the data, DMA it back, and wait for the next “GO” or “STOP” message. In this way we avoid loading and destroying contexts at every iteration.

4.2 Performance
With these 6 revisions of the code, each building upon the last, we experienced the simulation times found in Figure 8 for a 100 step simulation. As a reference to the versions:

1. Function-Offload Model Parallelization of Acceleration Vector Calculation
2. Parallelized the Movement of Agents
3. Moved from 4 SPEs to 6 SPEs
4. Moved from Double Precision to Single Precision
5. Circumvent \( dt \) reduction
6. Implement Streaming Model with Mailbox Synchronization

We actually experienced a decline in performance when moving from version 1 to version 2. Upon profiling the code we found that one problem was that SPEs were waiting a significant portion of execution time for the \( dt \) reduction (see Figure 9).

Thus, in version 5 of the algorithm we found a way to circumvent this reduction without sacrificing noticeable accuracy.
We can also see from the breakdown of time for version 3 of the code, the $dt$ reduction actually becomes more of a bottleneck to performance, as expected since there are now 6 SPEs synchronizing on one data item instead of 4.

Moving from double-precision to single-precision floating point operations had a significantly positive impact on performance, helping more for larger simulations than smaller ones. This of course makes sense since larger simulations are spending more time calculating agent interactions than smaller simulations.

For 100 simulated time steps (see Figure 10) we can see that the error for a single agent’s position, measured as the magnitude of the vector from its position at step $n$ under double precision arithmetic to its position at step $n$ under single precision arithmetic, is increasing with every time step. When running for 1000 steps, we generate Figure 11.

Our next two steps were to circumvent the $dt$ reduction and implement the flocking calculations in a Streaming data model, continually feeding continually running SPEs with data to crunch. Eliminating the communication of $dt$, once we realized that it could be done without incurring too much error, naturally caused the time spent in communication to reduce to practically nothing, and had a positive performance boost. The streaming SPE implementation also had a positive performance boost, as we no longer needed to startup and stop the SPEs at every iteration. By monitoring this versions performance, we found that now our performance bottleneck became waiting on mailboxes (see Figure 12).

Additionally we can see a load balancing problem on the SPEs by the fact that 4 is spending much more of its time performing computation than it is waiting on its mailbox. This is, in turn, causing the other SPEs to have a spike in mailbox wait time, because SPE 4 must finish before starting the next simulation step. What’s surprising here is that
Figure 12: Time Doing Computation, Waiting for DMA Transfers, and Waiting on a Mailbox Message
with the PPE, and would allow asynchronous updates to solve this problem, as it eliminates the need to synchronize likely solve. SPE to SPE communication would also help to work in the area of load balancing using a QuadTree would still face a large communication bottle neck, which further while avoiding large errors in agent positions. In the end we use single-precision arithmetic in place of double-precision, accuracy. Furthermore, for small time steps we were able to extent by changing our algorithm without losing much ac-

We hit some performance bottlenecks straight away, includ-
gram loading. We were able to easily overcome spe pro-
size thus limiting problem state size, the use of double-
precision arithmetic, and overhead associated with spe pro-
gram loading. We were able to easily overcome spe pro-
gram loading overhead by moving to a stream processing

5. CONCLUSIONS
Ultimately, The highest parallel performance we achieved was the 7.75x speed-up for the Hillbert curve OpenMP implementation running on 8 POWER5 nodes with 2048 agents (quadtree achieved a 7x speed-up). This is a good example of weak scaling, since that same implementation performed below 4x speed-up for 64 and 128 agents. However, 64 or 128 agents can already be simulated in real-time using similar NetLogo flocking models on uniprocessor architecture. The reason for implementing this algorithm on multi-
processor architectures is to increase the problem size, so we were pleased that our implementation performed so well at 1024 and 2048 agents.

The Cell provides a plethora of options for boosting per-
formance of code. Some optimizations not explored here inlude branch hint statements for the SPEs, SIMDization of the code, exploiting SPE to SPE communication, and multi-
buffering DMA accesses (which would become important for future simulations involving a larger number of agents). We hit some performance bottlenecks straight away, including waiting time for SPEs to synchronize, large SPE code size thus limiting problem state size, the use of double-
precision arithmetic, and overhead associated with spe pro-
gram loading. We were able to easily overcome spe pro-
gram loading overhead by moving to a stream processing

In terms of what it takes to get these performance boosts, a programmer must be reasonably well acquainted with the hardware. Setting up explicit transfers between known ad-
dresses of memory can be tricky to keep track of, especially if these addresses themselves must be DMA transfered into an SPE’s local store. At the sacrifice of programmer efficiency, however, such control does also mean flexibility and more opportunity for performance tuning. A dedicated instruction cache would have freed up space in the local store and made the programmer’s job easier, though, and we wonder why the Cell designers did not implement such an option.

6. REFERENCES