PARALLEL MODEL OF EVOLUTIONARY GAME DYNAMICS

AMANDA PETERS

MIT 18.337
MAY 13, 2009

I. INTRODUCTION

COOPERATION has been a highly debated topic in evolutionary biology for years. Its existence seems contradictory with Darwin's theories of natural selection. Natural selection dictates that the fittest will survive the competition between reproducing individuals and seems to predict the emergence of selfish strategies over altruistic ones. Evolutionary theory suggests that behaviors will only be favored for selection if they increase the fitness of the individual, which is often in contradiction with the group. This would lead to selection for selfish behaviors benefiting the individual and the disappearance of the other behaviors over time.

So why did cooperation evolve? From the point of view of natural selection, one would think the optimal strategy is to dominate opponents and eliminate opposition. Individual selection seems to dictate selection of strategies that support this notion of strong control. However, this is not what we see in practice in biology. There are many examples of where cooperative behavior emerges from interactions of humans to viruses to bacteria, leaving evolutionary biologists to debate the evolution of these behaviors.

What we see is that conflicts between individuals are often resolved by trials of strength resulting in the weaker animal surrendering goods without suffering serious injury or death. Why does the dominant animal not eliminate or strongly punish its opponent? Natural selection would seem to dictate the use of maximally effective weapons or fighting styles for a "total war". What emerges, however, is a type of cooperative behavior or 'limited war' with inefficient weapons that rarely inflicts serious injury. An example from nature provided by Maynard Smith and Price is the way snake species often settle intraspecific fights through wrestling matches without use of their deadly fangs. Previously, this was explained with the idea of group selection, meaning that the actions evolved for the good of the species. [2] The idea of group selection falls short though. When looking at the individual's point of view, one would select what is optimal for that individual. If there is incentive for the individual to defect from group strategy, it poses a destabilizing force to this theory and raises the question of why we see the emergence of 'limited war' strategies.

Furthermore, natural selection would seem to support punishing of weaker opponents. As discussed in Maynard Smith and Price's paper, the stable 'limited war' strategy that emerge contain elements of retaliation and are therefore similar to Tit-for-Tat strategies. As shown in Axelrod's Prisoner's Dilemma tournaments, TFT strategies lead to the emergence of cooperation. [3] If the threat of reciprocation is a key factor in the evolution of cooperation, what would happen if that was taken one step further? Could costly punishment be a rational strategy to help promote cooperation? If so, is it to the benefit of the individual?

A computational model of the interactions of individuals adopting the various strategies will help facilitate an investigation of these questions. Maynard Smith and Price laid out the foundations for a model that can be expanded to test other strategies and can be expanded to explain further biological questions. This paper will investigate will focus on writing a parallel program to run the simulations that will provide a firm base for future research. In the first section, I will discuss the model in more detail. I then explain the approach I took in developing the code, and then will discuss both methods of parallelism. I will first explain the GPU architecture and the methods to parallelize the code via CUDA on the GPU. I will then describe the IBM Blue Gene architecture and the subsequent methods to port the code to C/MPI. Finally, I will discuss the results from the parallelization and the demonstrated 97% time reduction using the GPU and the 99% time reduction using the Blue Gene implementation. Finally, I will discuss the potential for scaling the problem up and the future research directions this opens up.

II. Overview of the Model

IN traditional evolutionary studies, biologists relied on the idea of group selection to explain the existence of various traits. Under this idea, phenotypes were favored that would benefit the fitness of the overall group. Evolutionary adaptations were seen to come about for the better of the species. It wasn't until the late 1960s and early 1970s that this idea was really questioned. Maynard Smith and Price helped to establish the idea of individual selection being a guiding principle in evolutionary biology. In their 1973 paper "The Logic of Animal Conflict", they put forth the following theorem:

"Limited war" strategies are evolutionarily stable strategies

What is an ESS?

While the above theorem will be the focus of this discussion, the idea of evolutionarily stable strategies described in this paper play a fundamental role. When looking at the field of evolutionary biology, it is first important to understand the field of play. In this case, the payoffs involved with the game are biological fitness and hence the ability to reproduce as well as the ability to fight off invading strategies. In traditional game theory, it is generally considered common knowledge that players are aware of the game, attempting to maximize payoffs, and predicting the other players' moves. These factors contribute to the Nash Equilibrium solution which is a strategy with which the player has no incentive to deviate. In evolutionary games, however, the players do not choose their strategies, they inherit them. The player may not even be aware of his/her strategy or even of the overall game itself. The payoffs to the game are generally tied to the biological fitness of the individuals. This is a key point as natural selection illustrates the connection between fitness and species survival. New strategies are introduced via random mutations and the solution necessary is a strategy that is resistant. [4] This leads to the need for "evolutionarily stable strategies" or ESS described by Maynard Smith and Price. These are strategies that "if most of the members of a population adopt it, there are no 'mutant' strategy that would give higher reproductive fitness." [2] ESS is a refinement of the NE for the context of evolutionary games.

In order to demonstrate a strategy is ESS, one must show that in a population that has adopted primarily that strategy there is no incentive to deviate to another. This will lead to natural selection weeding out the undesired strategies. On the other hand if there is incentive to deviate, instability will arise as the population shifts to express the deviant phenotype.

The Computer Model and Simulation Test

Maynard Smith and Price were interested in investigating animal behavior. As mentioned above, they focused on the use of 'limited war' tactics vs. 'total war' tactics. When engaging in conflict with one another, one might intuitively expect to see 'total war' tactics that have a higher likelihood of ensuring victory for the animal. In practice, however, this is not the case.

Before this paper, the idea of group selection had been used to explain this paradox. They instead posed the following theorem as an explanation:

"Limited war" strategies are evolutionarily stable strategies for both the species and the individual

A computer model and simulation test were used in order to investigate the role individual selection has on 'limited war' behavior. In the model, five strategies were outlined and given to the contestants. The 'Hawk' strategy was the only 'total war' meaning it always plays the destructive weapons (D). Of the 'limited war' strategies 'Mouse' never plays D and retreats immediately when D is played against it, 'Bully' will play D on the first move and in response to conventional weapons (C) but will play C in response to D, and both the 'Retaliator' and 'Prober-Retaliator' escalate in response to an attack while 'Prober-Retaliator' will probe by playing D with a low probability. By looking at the outcomes, one can see whether the 'limited' or 'total' war strategies were favored. The strategy of the contestant and its opponent were the only variables. Set probabilities were used to describe the likelihood of serious injury, initial move, and retaliation. Payoffs were calculated as follows: winning+=60, receiving serious injury += -100, bonus for short game with no injury += 20 and each non-serious injury += -2. [2]

III. Application

The first step was to write a serial version of the application. The goal of the code was to provide framework for future game theory analysis, so I began by understanding the model put forth by Maynard Smith and Price and writing an application to capture this model. My goal is to enable analysis of more behavioral strategies and variables values, but in order to get there I first needed to replicate the results put forth in their paper.

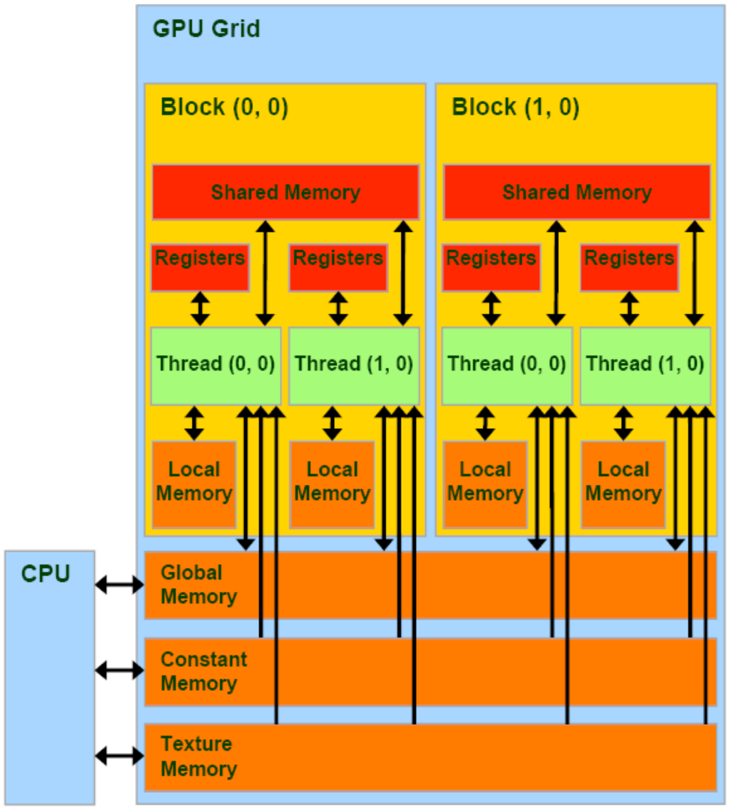
I set up my program with four key concepts: the game, a round, strategy, and player. A game would be defined as the conflict between two players. Each game would have a set number of rounds and the payoffs across these rounds would be averaged to show the results of the interaction between players. Each player would be assigned a behavioral strategy. For this, I used the five examples described in the previous section. The goal of the application is to analyze the conflict between every behavioral strategy with every other behavioral strategy to determine if a dominant strategy arises. To this end, the call of the game play function was iterated over by every pairing of strategies. The game play function then simulated that game through by calling the function to simulate each round. In each round, turn style play was used with a randomized first player. The results were all fed back to the main function.

The final results as will be discussed in a later section were consistent with the findings in the paper [2].

Parallel version 1: CUDA GPU

Architecture

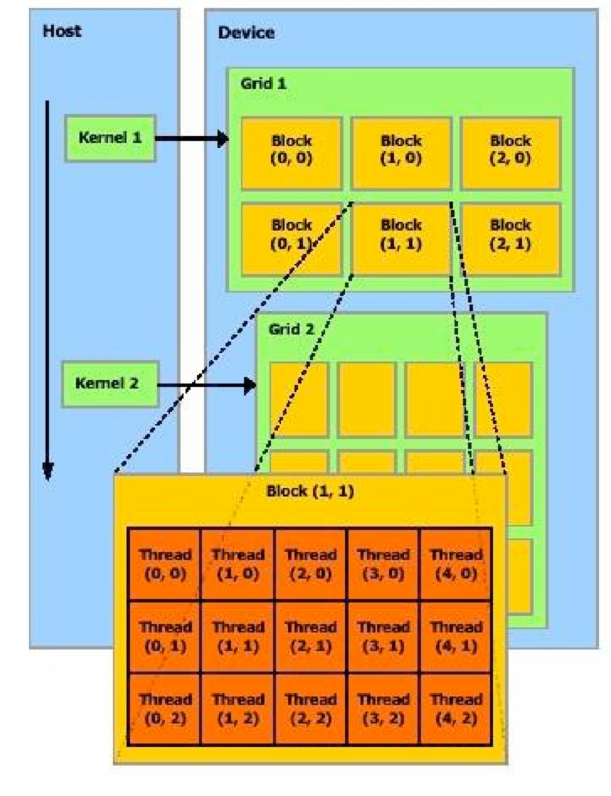
The first step in this project was to complete background research on both GPUs and the CUDA programming language. Although I have had experience programming in parallel environments, this has been primarily with MPI based codes. I was completely unfamiliar with the GPU architecture and the accompanying programming model. By reading the manuals NVIDIA provides, I was able to get a strong understanding of the underlying hardware architecture, memory model, and therefore the challenges of its programming model. In order to begin programming for GPUs, it seems important to understand both the interaction between the CPU and GPU as well as the GPU grid/block setup. I initially needed to understand the memory model. The image below shows the CUDA Memory Model:



NVIDIA CUDA Memory Model [5]

The setup is such that an application begins running on the CPU (i.e. the host) and sets up necessary data. This data can then be copied into memory onto the device and kernels will be executed in blocks of threads. These threads have several levels of memory as shown above. Understanding this setup allows the programmer to move data between the host and device efficiently. It is clear that many of the bottlenecks encountered in parallel codes on these systems will be tied closely to inefficient data movement. Anytime the GPU needs to access global memory, the program will be significantly slowed down. By having a firm grasp of the memory model, the programmer can optimize data access and maximize overlap of the communication and computation.

Aside from understanding how to get data to the GPU, it was important to learn about the CUDA programming model. A strong advantage of the GPU architecture is the ability to leverage a high number of threads for the application. The programming model is such that the host will execute a number of kernel invocations to the device. These kernels will be executed as a batch of threads organized as a grid of thread blocks as shown below.



Thread Batching [5]

In the parallelization of my application, I leveraged this setup to have each thread compute the payoff output for one game at a time. This allowed up to 512 games to be simultaneously simulated in each grid. This will be discussed further in the following section. [5]

Parallelization

To prepare the code to run on the GPU it first needed to be ported from C to CUDA. This was fairly straightforward as CUDA is a C based language. The tricky part was identifying functions for parallelization and setting up the appropriate kernels. I decided that the best method would be to run pre-and post-processing steps on the CPU and offload the simulations of each interaction. As the rounds themselves could be calculated with no shared information between them, the code is considered embarrassingly parallel. This allowed me to set the code up so that grids were spawned that would allow each round to be handled by a separate thread with no dependence on other calculations. When working with CUDA, the biggest potential bottleneck is the data transfer. Anytime the device needs to return to the CPU to access memory, you face a large potential time sink. It is therefore important to optimize the data usage and especially the data movement. Future potential options for data optimization will be discussed in the final section of this paper.

The setup of my CUDA code was to define a global kernel that would be called on the device for each game. This would provide the structure for splitting the round computation across the threads. The code sample below shows this kernel:

```
__global__ void gameGPU(int player1, int player2, float* d_payoff1, float* d_payoff2,float* rand_si, int max_rounds){
   //Thread index
   const int tid=blockDim.x * blockIdx.x + threadIdx.x;

   //Total number of threads in grid
   const int THREAD_N = blockDim.x * gridDim.x;

   int max_moves=500;
   for (int round = tid; round < max_rounds; round += THREAD_N)
   {
     play_round(player1, player2, d_payoff1[round], d_payoff2[round], rand_si[round],max_moves);
   }
}
```

You can see that this allows no more than the maximal number of rounds to be computed but at the same time offloads them evenly to the threads. The function would then call another inlined kernel on the GPU that would play out each round. Therefore each thread would calculate each rounds play but running through a mock game between two players with specified strategies and return the payoff. At the end of all the rounds, the payoff array would hold the payoff values for each player in each round and would then be copied back to the host for post processing. Care was taken in coding to keep most of the data in local memory for each thread. The only major call to global memory was to return the payoff results for that thread. As this is the data that will need to be copied back to the host at the end of the parallel section, this setup seemed ideal. As will be discussed later, this could be further optimized by interlacing the post processing with the next set of games running on the device or else having one thread calculate the average while on the device. This would optimize the amount of data transfer by only copying back to the host the average value instead of the entire payoff array. Future tests would need to be made to determine the optimal setup. It will likely have a strong dependence on the number of rounds played per game and number of games played.

IV. Parallel Version 2: C/MPI on IBM Blue Gene

Architecture

The system used in this portion of the study is the IBM Blue Gene\L supercomputer. [6] The architecture for these systems is based on low cost embedded PowerPC technology. Some of the architectural features are relevant for this study, so I will briefly summarize some of the key components in this section.

BG/L is a massively parallel supercomputer. This system uses a distributed memory,
Message-passing programming model. The basic building block is a custom system-on-a-chip (SoC) that integrates processors, memory, and communications. The chip contains two standard 32-bit embedded PowerPC 440 cores that run at a frequency of 700MHz with the addition of two floating-point units (FPU). Each core can perform four floating point operations per cycle resulting in a theoretical peak performance of 5.6 Gflops/chip. The chip constitutes the compute node. Two compute nodes attached to a processor card alongside the memory constitute the compute card. In the case of BG/L you can have either 1GB of RAM per compute node or 512 MB of RAM per compute node. In addition to the compute nodes, the system contains a number of I/O nodes. These are physically very similar to the compute nodes but have an integrated Ethernet enabled for communication with the external file systems. The I/O cards and compute cards are then plugged into node cards. A rack holds 32 node cards, or 1024 compute nodes. The largest system is currently at Lawrence Livermore and consists of 104 racks for a total of 106,496 compute nodes. [6]

The utilization of SoC technology allows a dense packaging demonstrated below in Figure 1.

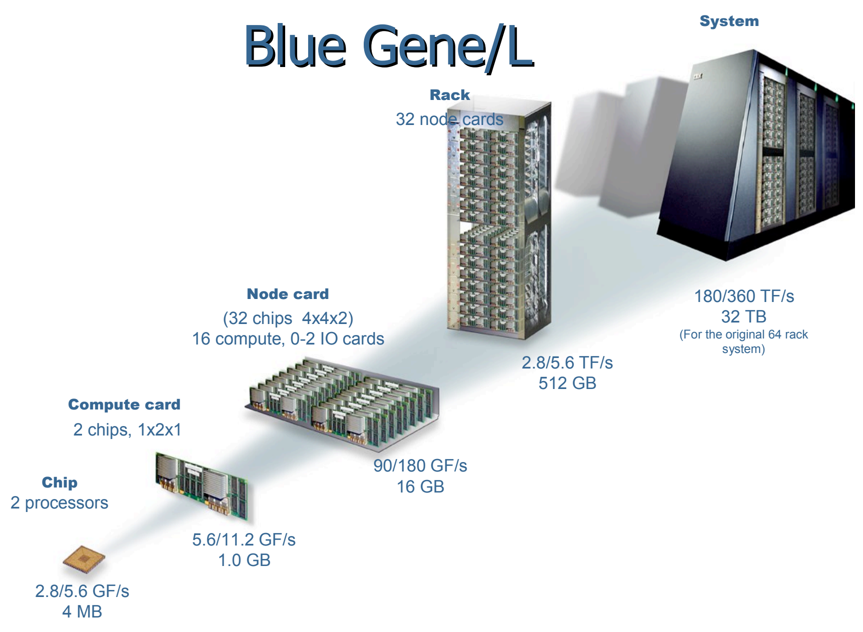


Figure 1. The packaging hierarchy of Blue Gene/L supercomputer

The nodes are connected by four highly optimized networks: a three-dimensional torus, global collective network, control system network, and Gigabit Ethernet networks. The majority of messaging is conducted via the torus network which supports low-latency, high bandwidth point to point messaging. For more information regarding the networks refer to [7]

As scalability is essential to the following parallelization discussion, it is necessary to get a high-level understanding of the system software architecture. This is presented in Figure 2. below.

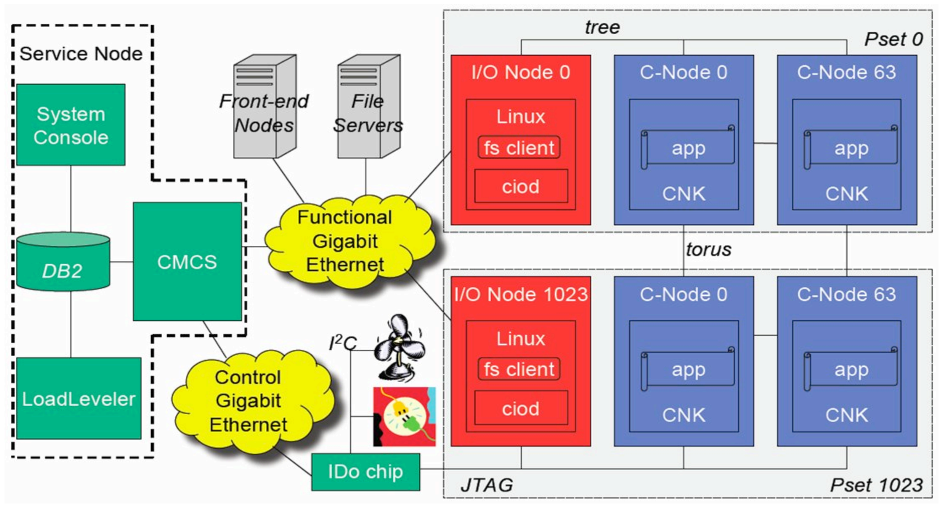


Figure 2. Overview of System Software

The main point is that the computational core is partition into logical processing sets (psets). Each of these contains one I/O node running Linux and 64 compute nodes running the custom BLRTS kernel. Access to the computational core is only achieved via two Ethernet networks for I/O and low level management.

Parallelization

One of the key advantages to this code was that it has no I/O and is strongly computationally bound. The only overhead introduced would be the communication between nodes which gave this application strong parallel potential. As mentioned earlier, the code itself is embarrassingly parallel. The setup of this code was such that small units of computation were conducted over a number of rounds for each game in the sequence. My job was made a bit easier here as the calculation for each base round only depended on information calculated within that round. Therefore if all of the information for that game resided on each node, it could calculate the payoffs for that round independently and in parallel. The only catch was then at the end of each game, all of the data across the nodes would need to be updated to get the payoffs collected for all rounds.

As I had determined that parallelization could occur across the rounds in the game, I then needed to define which scheme I would use. In general, there are two main methods employed to parallelize code: Master/Worker and Static Partitioning/Distributed Algorithm. For embarrassingly parallel codes, the most popular method is Master/Worker. In this case, one node acts as the 'master' and is generally responsible for all I/O functions as well as distributed work units to other nodes and collecting the results. This method can be especially beneficial when there are a changing number of work units or they each vary extremely in their runtime. The Master/Worker scheme allows for finer control over load balancing if the computation is not uniform. Static Partitioning, on the other hand, is useful when there are defined chucks of work of equal size to be distributed to a set number of nodes. In the case of the code in this paper, the work units each take approximately the same amount of time. This means that the problem where some nodes may hang and force the others to wait will be avoided. As the Master/Worker setup requires that the Master distribute the work via message passing, Static Partitioning can be used to reduce messaging overhead.

I maintained a ‘bookkeeper’ node that collected the results from each node, conducted minimal post processing, and handled the small amount of I/O to print the results. To handle the work breakup, I opted for distributed algorithm leveraging static partitioning. Similar to the GPU case, units of work were completed by each processor (in the GPU case it was each thread). The application would dynamically determine the work unit size depending on the partition size allocated and have each processor assigned to compute a small chunk of trials. In this case, the work unit size is determined by dividing the total work (i.e. the length of the game) by the partition size. Loops that initially iterated over the entire length of the game would now be split up to go from set markers in the round sequence. These markers could be calculated on each node itself based on its MPI rank and therefore be set without any message passing required. The code sample below demonstrates how each processor would contribute to its section of the overall array:

```
int chunk = max_rounds/NP;
for (int round = 0; round < chunk; round += 1)
{
  payoff[0]=0;
  payoff[1]=0;
  play_round(player1, player2, max_moves, payoff);
  payoff1[rank*chunk+round] = payoff[0];
  payoff2[rank*chunk+round] = payoff[1];
}
```

To coalesce all of the data back on all of the nodes, MPI_Reduce was used. This command combines values from all nodes and distributes it back to all nodes in the communicator. A nice aspect of the command is that it allows you to specify the method of combination. In this case, I set every value in the vector to zero except for the values that the specific node was calculating. I then had the MPI_Reduce combine the data by summing the vectors. [8] This meant that each node would contribute a zero in each place except for the slots that were computed on this node. Therefore the data could be propagated easily in one step.

```
Foreach species:
  Foreach species:
        gamePlay(var1…);
        MPI_Reduce(var1…);
        If (rank==0) Calculate_averages();
If (rank==0) Print_game_results;
```

This node was then able to complete the post processing step of computing the averages and write the results of that game to the file. In the simulations I ran, each game consisted of 4096 rounds each with a maximum of 500 moves. There were fifteen games played: one for each species combination.

V. Results

Biological

The first level to look at for results would be whether or not they were consistent with the results from Maynard Smith and Price, and whether they answered the intended question: can we identify an evolutionarily stable strategy for conflict? The results from my simulations, be it on the CPU, GPU, or Blue Gene, were consistent within +/- 2% of the results Maynard Smith and Price published. This indicated a successful simulation and supported the findings that 'limited war' strategies in which potential reciprocity played a role were ESS. The results from my serial test are below:

To identify which strategies are ESS against the other four, look at the column of that strategy. A strategy is considered ESS if in a population primarily consisting of players employing that strategy, there is no reason to deviate to another. In the case of 'Mouse', you can tell that it is not ESS as every other strategy does at least as well if not better than it does against itself. This would imply that when in conflict with a player of the 'Mouse' strategy, the individual has incentive to deviate to any strategy other than 'Mouse' for itself. By looking at the other columns, you see that 'Retaliator' is an ESS and 'Prober-Retaliator' is almost an ESS. This simulation strongly demonstrates that individual selection will select for ‘limited war’ strategies over ‘total war’ strategies. It demonstrates that cooperation can evolve and is furthered by the idea of potential reciprocity. This work supports the later research supporting strategies like Tit-for-Tat in IPD tournaments. [9] Choosing strategies that promote cooperation are stable for the individual. The potential work that can build from this will be discussed in a later section.

Parallelization

The evaluation of the parallel version of the code was carried out with 4096 rounds per game, 500 maximal moves per round, and 15 games were run (one for each strategy combination). This was carried out on the CPU, GPU, and Blue Gene architectures. It is worth nothing that while the same problem size was used in each case in this paper, this is not indicative of optimal hardware usage. As the game size and number of strategies increase, the problem size will grow to more thoroughly justify use of either a system like Blue Gene or an MPI/CUDA hybrid to leverage multiple GPUs.

GPU

The hardware used for this data was the GeForce 9400M of the Macbook Pro. There are 16 cores, 8192 registers per block, 512 maximal threads per block, and a maximal grid size of 65535 x 65535 x 1. The corresponding CPU was used to obtain the CPU result as well. Running on the GPU and leveraging 512 threads, I was able to reach a 97% time reduction when compared to the overall time the simulation took on the CPU. The fact that this code is truly embarrassingly parallel with little communication or IO overhead introduced, parallelizing the code was able to significantly reduce the overall runtime. The resulting times are shown in the following graph:

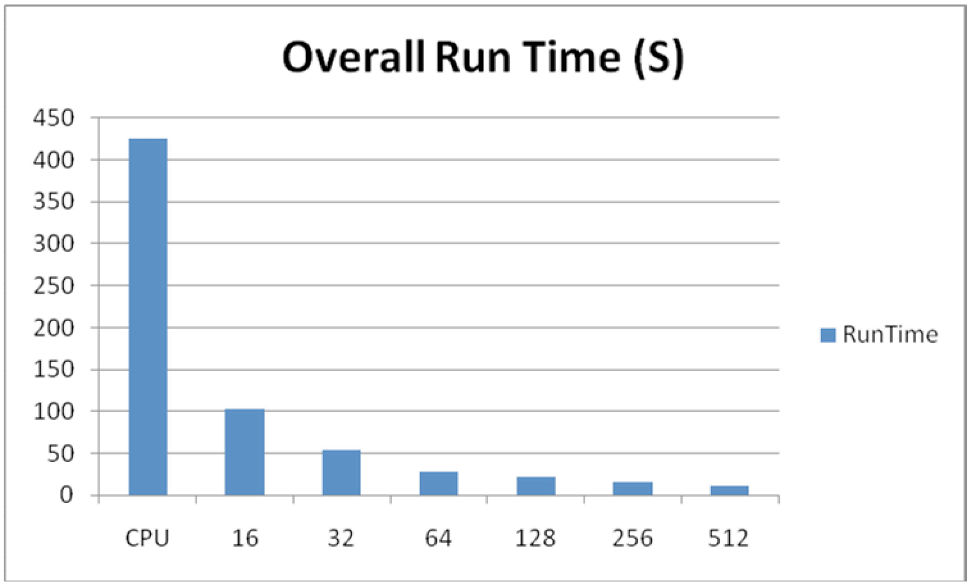


Figure X. This shows the overall runtime when the simulation is run on the CPU or the GPU with thread counts of 16, 32, 64, 128, 256, and 512.

The next important point to look it is the efficiency of the speedup. This will allow you to determine the optimal size for a grid needed by this problem size while demonstrating the efficient use of the hardware. The speedup achieved on the GPU hardware is shown in the graph below:

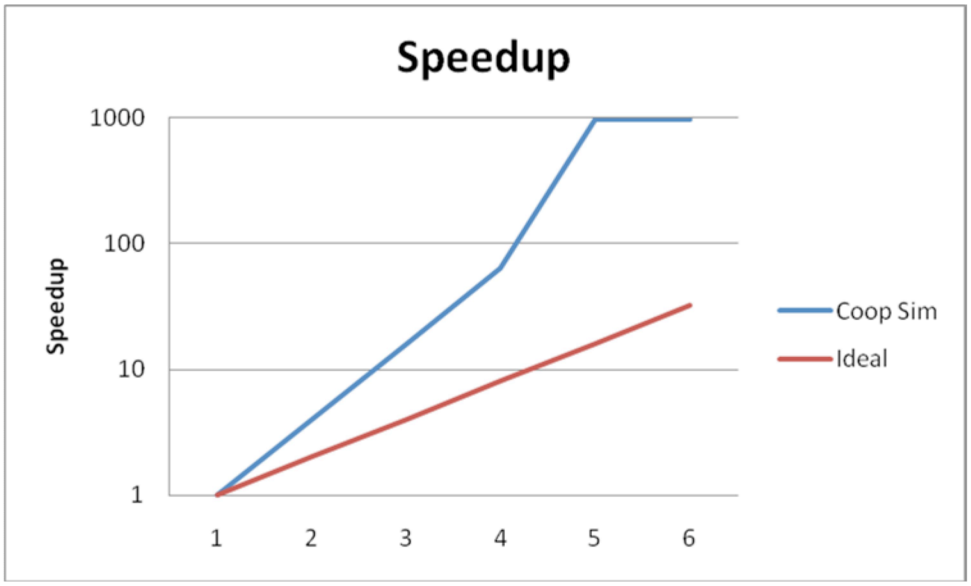


Figure X. This shows the overall speedup achieved by the cooperation simulation code compared to the ideal linear speedup.

The Cooperation Simulation is exhibiting 92% efficiency at 64 threads and 35% efficiency at 512 threads. This is evidence of strongly successful parallelization that will only improve for larger problem sizes. One of the main reasons we start to see a trail off in efficiency here is the the computation time for each thread is greatly reduced as the larger thread counts are approached. The time for the computation starts to be more on the timescale of the memory management. As we move to larger problem sizes, larger grid sizes will be justified and needed as the overall runtime for the program will grow exponentially with the problem size. This will provide a strong opportunity to see higher efficiency as we scale up the system size to match the problem.

Blue Gene/L

The hardware used for the Blue Gene tests is a Blue Gene/L with 512 Mb of memory per node. For this paper, my goal was to demonstrate the parallelism of the code and build a foundation for future evolutionary game dynamics research. To this end, it was most important to simulate the games involving the set 5 behavioral strategies discussed previously. In the results section, I will discuss the path to scaling to large systems but for the initial work I used partition sizes of 1 node to 32 nodes.

Table 1. Runtime for Cooperation Simulation on Blue Gene with various partition sizes.

It is important to note the 99% overall time reduction. Also, note that the time stabilizes at about the 16 node partition size. This is because the data has hit a threshold for parallelization where the time it takes to compute each chunk is comparable to the other overhead such as the time for MPI communication. At this point, there is no need to distribute the work any further. It does, however, lend itself well to scaling the problem up at this point.

The corresponding speedup graph is shown below demonstrates results that raise question and lead to future directions.

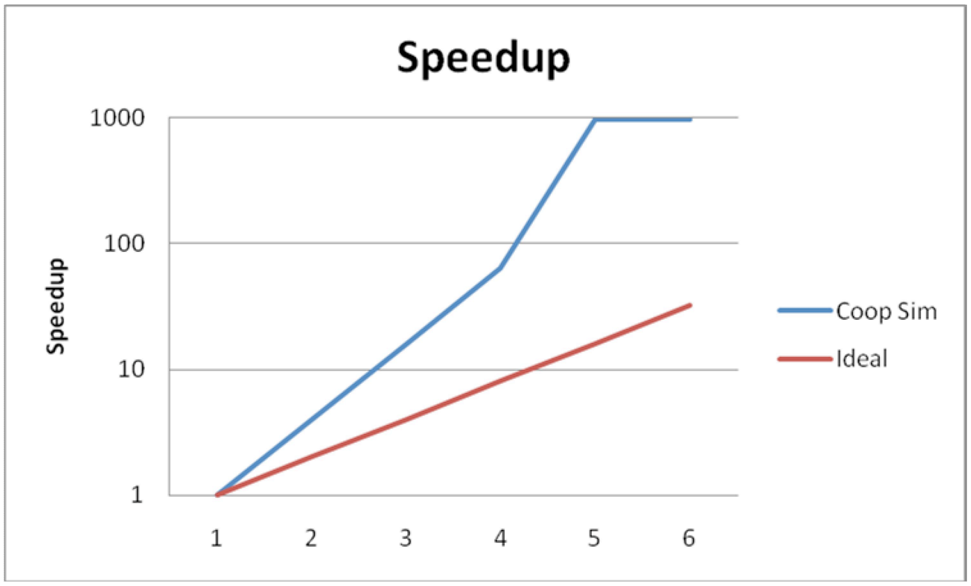


Figure X. Speedup seen for the Cooperation Simulation as compared to the ideal linear speedup.

The speedup exhibited is superlinear to an unexpected degree. The work units being computed by each node are so small that there is a good chance what we are seeing here is the result of cache effects. It is possible that everything is fitting in L1/L2 cache rather than main memory. The code base being used is the same serial code as leveraged by the CUDA example. I have implemented a verbose flag for debugging that has each node print out the work it is completely. This ability to access the processes running on the nodes for debug purposes was a big advantage and proved that the program was running properly and completing all work in the simulation. This leaves the question of the results for future work. Testing of different problem sizes to stress test the cache theory would be the next step.

VI. FUTURE WORK

I wrote this application to provide a test platform for future work in evolutionary game dynamics. To this end there are two potential paths that need to be explored for future studies: the application of the program and the optimization of the parallelization.

Future Applications

The simulation of the game dynamics shown in this application provides a strong base to investigate further properties of biological systems and their behaviors. The model that is encapsulated here can be expanded to not only take into account more behavioral strategies, but to test the affect of the different variables. By varying the thresholds for factors like the likelihood of serious injury, we can start to represent varying levels of fitness of the individuals as well as levels of weapon intensity. Moreover, other expansions to the current behaviors can add to help gain insight to the various proposed methods for the evolution of cooperation. For example, there is debate about the role punishment plays in individual selection. By adding another option to the actions the behaviors are choosing, we can model the impact punishment could have. Instead of having the options simply be cooperate, defect, or retreat, we could add a fourth option to punish. In limited models this has been shown to promote cooperation at the cost of payoffs. This phenomenon could be further studied here.

There are many instances where cooperation has evolved in biology where natural selection seems like it should select against it. The application will allow this behavior to be modeled on a large scale and hopefully help lead to solutions to these questions.

Parallel Optimizations

There are options that can be investigated for both parallel versions of the code to further optimize them. For the CUDA implementation, further optimization regarding data management could be completed. Currently, the entire payoff array for both players in a game is copied back from the device memory to the host for post processing. Work should be done to determine the trade-offs between interlacing the CPU post processing and the next set of work for the GPU or having the device code sync the data in global memory and have the device do the post processing, returning only this result to the host. Along these lines, tests to further refine the data layout would be beneficial. More work could be done to compare various methods of distributing the games over different grid sizes. This will also likely change depending which hardware is being used and the memory sizes. Also, as mentioned many times, the goal with this work is to scale up the problem. In the future, this will likely lead to a hybrid MPI/GPU code to enable use of a cluster of GPUs.

As for the C/MPI code, there are a few key places that are ripe for optimization. Aside from the necessary work to test the caching and extreme superlinearity exhibited in the results, work on single node optimization would help improve hardware efficiency. As this paper was focused on parallel methods, little time was spent finely tuning the main computations. Tests on hardware usage and optimizations here could improve overall runtime. It would also be advantageous to move to the Blue Gene/P architecture which allows for multiple threads on a processor. This could enable further parallelization of the subroutines.

REFERENCES

1. Gadagkar, Raghavendra. "The Logic of Animal Conflict." Resonance 11 2005 5. Web.9 May 2009. <http://ces.iisc.ernet.in/hpg/ragh/publication_list/Gadagkar_Publications/Gadagkar_2005c.pdf>.

2. Smith JM, Price GR (1973) "The logic of animal conflict."Nature 246:15–18.

3. Nowak MA, K Sigmund (1993). Chaos and the evolution of cooperation. P Natl Acad Sci USA 90: 5091-5094.

4. "Evloutionary Stable Strategy." Wikipedia: The Free Encyclopedia. 03 2009. Wikipedia. 7 May 2009 < http://en.wikipedia.org/wiki/Evolutionarily_stable_strategy >.

5. NVIDIA CUDA Programming Guide. http://developer.download.nvidia.com/compute/CUDA/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pd f

6. A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas, ‘‘Overview of the Blue Gene/L System Architecture,’’ IBM J. Res. & Dev. 49, No. 2/3, 195–212. 2005.

7. N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. E. Giampapa, P. Heidelberger, S. Singh, B. D. Steinmacher-Burow, T. Takken, M. Tsao, and P. Vranas, “Blue Gene/L Torus Interconnection Network,” IBM J. Res. & Dev. 49, No. 2/3, 265–276 (2005, this issue).

8. X. Martorell, N. Smeds, R. Walkup, J. R. Brunheroto, G. Almási, J. A. Gunnels, L. DeRose, J. Labarta, F. Escalé, J. Giménez, H. Servat, and J. E. Moreira, “Blue Gene/L Performance Tools,” IBM J. Res. & Dev. 49, No. 2/3, 407–424. 2005.

9. Axelrod, R. & Hamilton, W. D. The evolution of cooperation. Science 211, 1390 (1981).