The Pronto Pup Problem

Christopher Saiko
University of Minnesota
saik0008@umn.edu

1. Introduction

Computer simulations of real world situations are one of the primary areas that we use, as computer scientists, to describe and model the world around us. By creating and running these simulations, we may advance our knowledge and ideas about how the real world reacts to systems that we create. One such large system is the yearly Minnesota State Fair. This yearly human gathering sees millions of people gather in one place over the course of 10 days. Summer of 2018 saw an attendance record of over two million people [1], so it is important to understand how people may move, and how resources may be allocated within such a system.

One part of such a system is one of the myriad of food stands which are frequented by fair attendees, and which might also be modeled in such a way as to simulate it. One example, a Pronto Pup (corn dog) stand, could be implemented using a combination of Producer-Consumer buffers for the fryers and staff, and a Sleeping Barber implementation [2] for the queue of fair goers. Other variables, such as ambient temperature, time of day, acceptable waiting time, and others, could be modeled within the system using probabilistic methods.

Once modeled, various Artificial Intelligence techniques could be applied to this system to find an allocation of resources to serve a maximum number of customers, while minimizing the cost to operate the food stand. After creation of such a model in a sequential algorithm, of the techniques used to solve it, the Hill Climbing algorithm saw the best success at tackling this so called "Pronto Pup Problem". While a sequential model did produce reasonable results, moving to a parallel model could introduce drastic speedup to the problem solution, delivering more accurate results in less time.

2. Design Overview

To handle this problem, a sequential model was first developed which returned reasonable results. That sequential model was then moved to a parallel model, utilizing several parallel programming methods. On program execution, several arguments were passed to the program, for AI search method, time limit, and repeat simulations per set of simulation variables. Dynamic parallel programming was used to call kernels in a parent/child relationship. On selection of an AI method, the program calls a specific kernel for that method, which then begins looking for a set of variables which, on average, gives the best score. For each set of variables, the kernel runs an initial child kernel to set up random number states for every thread, then runs the corn dog stand simulation child kernel. Where the sequential algorithm runs the repeated simulations sequentially, the parallel version has a thread handle one of these repetitions each, to speed up program execution. The algorithm returns an average value to the parent AI method kernel, which makes a decision to keep the value as the best value, and best variables for the simulation, or to discard them and move on to a new set of simulation variables. See Figure 1 below for the flow chart of the final algorithm.

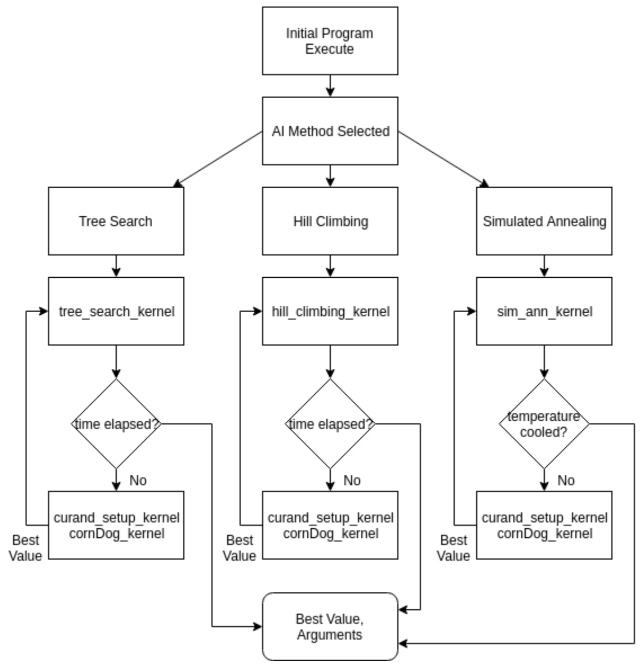

Figure 1: Corn Dog Stand Simulation Program Flow Chart

3. Implementation

Before moving to a parallel model, a sequential model was first developed, to both compare results, and to verify correctness. The sequential algorithms were then moved to a parallel implementation, with each algorithm becoming its own kernel. An additional kernel to set up random number seed states was also used, as each thread needs its own unique random number generator. These kernels then work in tandem to return a best found value for sets of simulation variables.

To determine an efficient allocation of required resources, a number of simulations needed to be run. To start, a function was developed which may be passed multiple variables. Within this function, a day of corn dog stand operation was simulated using the desired parameters and stored variables, and an objective function score returned at the end. This score is determined by four primary metrics.

Fair attendees served, a maximized item, represents a total running count of how many customers have been successfully served food or a beverage. Wasted food items, a minimized item, keeps track of the amount of wasted product at the end of the business day. Third, the total amount of lost customers who exited the line queue before being served are tracked. Finally, total profit, a maximized item, represents the total dollar amount that the stand generated above operating expenses over the course of a business day. An algorithm sequentially ran a series of these corn dog stand simulations to determine the best allocation of resources to the stand. Several Artificial Intelligence techniques were be used in the algorithm when running the simulation to determine a best solution.

3.1 Algorithms

3.1.1 Corn Dog Stand Simulation

The simulation algorithm implemented is sequential in nature, to ease implementation requirements, and to reduce possible varying data collection speeds on multi-core processor systems. Within the simulation are a possibility of five bounded buffers; two fryers, two lines for customers, and a general food buffer to place cooked items. These bounded buffers are be able to be adjusted in size to account for larger or smaller space allowed. Other variables passed to the algorithm include worker wage rate, the hours of operation, corn dog and beverage prices, corn dog and beverage costs, corn dog cook time, daily hourly temperature as an array, number of customer lines, and acceptable customer wait time. Finally, customer behaviors are passed in the form of an array, as a series of probabilities between zero and one. Customer arrival chance per minute is the probability that in any given minute, a customer will arrive, and be placed into an available line queue. Corn dog order chance, and corn dog order chance mealtime offset, respectively, details the probability that a given customer will order a corn dog, as well as an increase in that chance dependent on the proximity to mealtime, coded as 8AM, 12PM, 4PM, and 8PM. Beverage order chance, beverage order chance mealtime offset, and beverage order chance temperature offset, like corn dog order chance, represents the probability that a given customer will order a beverage, with a similar offset for proximity to mealtime and an additional chance that the beverage order will be adjusted based on the current temperature.

Although some choices for the variables may be arbitrary, real world limits are placed on the majority of the variables. Fryer number is limited to one or two fryers, and customer lines are limited to one or two lines. Fryer size capacity is limited to be between three and six corn dogs, chosen arbitrarily to roughly fit real world fryers. Customer lines size are not be limited to any particular size, since customers will naturally grow tired of waiting for service, and leave of their own accord. To ease simulation requirements, the ambient temperatures have been chosen arbitrarily, as mentioned above. In real world usage could be decided beforehand by consulting a weather forecast. Customer service time was fixed at one customer serviced per minute.

Using all these variables, the algorithm simulates the passage of a single day in steps of minutes, cooking corn dogs in the fryer queue(s), transferring cooked corn dogs into a food queue, and placing customers into line queues for service. At the end of the simulated day, the algorithm calculates profit, wasted food, total customers served, and total customers lost. These values are used to calculate a score, with varying weights associated with each value. The simulation is run a total of 1000 times, or whatever value is placed in program execution arguments, with a thread each running one of the simulations. Each thread returns a value to an array. The AI algorithm

3.1.2 Tree Search

To implement a Tree Search approach to the problem, this algorithm tests each possible permutation of the variables. Since all variables must be assigned to have a full state, only the leaves of the resulting tree structure will have possible solutions, so a Breadth First Search or Depth First Search will be functionally the same. It is expected that this will not be useful in finding a quality solution, as the number of variables over which to search is quite large, and time complexity may be too large for a Tree Search to handle. A limit on the number of simulations is required for Tree Search. Since limiting in this way may not allow reaching of certain portions of the tree, this method is to be treated as a lower bound for performance of the simulation. To that end, a time limit was placed on the algorithm to achieve a possible solution, and the best scores found and recorded for each time allotment.

3.1.3 Hill Climbing

Local search techniques were then also attempted, specifically Hill Climbing and Simulated Annealing. [3] These algorithms were compared, and their effectiveness measured, in the context of the score each algorithm achieves. To measure this, each algorithm recorded the time complexity required, in the form of total algorithm wall running time, and the best solution found, in the form of the best score value returned from the objective function. A simple neighbor selection function was be designed, to randomly select a argument of the simulation algorithm and to randomly shift it to a different valid value. These neighbors were used to generate a way for the algorithms to choose a new permutation of variables to simulate.

In the Hill Climbing algorithm, the algorithm selects a neighbor and runs a simulations for that neighbor. If a better score is found, the new set of variables that resulted in the score is saved, and a neighbor to that new set of variables is found to test. If a better score is not found, a new neighbor to the last set of variables to have a better score are found, and tested. This method is repeated, and as neighbors are found in higher scoring areas of the tree, the algorithm heads in that direction, climbing the hill to find maximum values.

3.1.4 Simulated Annealing

The Simulated Annealing algorithm, like the Hill Climbing algorithm, utilizes a neighbor function to get new sets of variables to test. However, unlike Hill Climbing, it may travel further away from an given state, even with lower resulting scores, attempting to get out of any local maximums which would otherwise trap the Hill Climbing algorithm. To that end, the algorithm takes three additional values: alpha, temperature, and end. As the algorithm searches and finds scores among neighbors, it may accept a worse score with lessening probability as the algorithm proceeds. [4] If a worse solution is found, instead of rejecting it outright, the algorithm generates a random number between zero and one, and compares it with exp(-score difference / temperature). If the random number is higher, the algorithm reject the solution, but if not, it will be accepted. Every iteration of the algorithm, the temperature is reduced by multiplying the temperature by alpha. Over time, as the temperature approaches zero, the value of exp(-score difference / temperature) approaches zero, and the likelihood of worse values being accepted will be reduced. When the temperature is reduced enough to be below the end value, the algorithm exits, with the last best value being returned.

3.1.5 Random Numbers

Finally, a kernel needed to be run before the calls to the parallel corn dog stand simulation kernel to set up random number states. Since random numbers on the GPU need an initial state, like a seed, each thread requires its own unique state to generate its own unique random numbers. This kernel simply takes an array and initializes a state for each element in an array, with an each element corresponding to a unique thread. This array is then passed to the simulation kernel, that each thread may access the unique state generated for that thread.

4. Verification

Testing procedure for a parallel implementation in a problem such as this can be difficult to accomplish. Since the algorithm uses probability in calculation of certain areas, such as human behavior, answers between the two implementations will always vary, at least to some extent. However, with several identical runs, some trends could be observed between the two implementations. The code was first run with the sequential algorithm, for certain program arguments. Data was output to the screen, and recorded for comparison later. For the same arguments, the parallel algorithm was then run, with the same data output to the screen, and recorded.

The first algorithm to be compared was Tree Search, with the number of simulation repetitions was set to 1000, 2000, and 4000, each for 60 seconds. The number of iterations and the score were recorded for a set of six identical runs of the sequential and parallel programs. Data taken for Tree Search is shown in Table 1 and Table 2.

Table 1. Tree Search Sequential Algorithm Scores

Run	**1000**	**2000**	**4000**
1	5829	5200	5133
2	5898	5208	5128
3	5894	5205	5129
4	5832	5204	5134
5	5830	5208	5133
6	5841	5214	5128
Mean	5854	5206.5	5130.8
Std Dev	29.97	4.31	2.54

As shown by the data in Table 1, the sequential algorithm had mean scores trending down, as expected. Since more simulation repetitions were run, this meant that the algorithm could encounter fewer nodes of the tree within the allowed time, which could have the potential for better scores. The standard deviation is quite low for the larger simulation repetitions, showing an accurate number.

In Table 2, the parallel algorithm shows similar scores, with the scores somewhat trending down as the number of simulation repetitions increased, again owing to the fact that with less time, fewer of the possible nodes may be visited, and the potential for higher scores is lessened. A higher relative standard deviation to the sequential can be explained by the difference of the number of iterations (nodes visited), shown in Table 3.

Table 2. Tree Search Parallel Algorithm Scores

Run	**1000**	**2000**	**4000**
1	5815	6217	5286
2	6107	6407	5956
3	5496	6292	5945
4	5471	4657	5422
5	5685	5772	5906
6	6535	5883	5963
Mean	5851.5	5871	5746
Std Dev	372.5	587.1	280.8

Table 3. Tree Search Mean Iterations - Nodes Visited

Algorithm	**1000**	**2000**	**4000**
Sequential	192	97	49
Parallel	361791	235572	135649

As shown in Table 3, the parallel algorithm encounters vastly more nodes in its allowed time compared to the sequential version. Whether or not the mean score values are correct is hard to judge, and thousands more trials might be needed to verify within some degree of accuracy. Figure 2 shows that the parallel algorithm did find better mean scores at higher simulation repetitions, showing it to be more efficient at exploring the tree with those higher numbers.

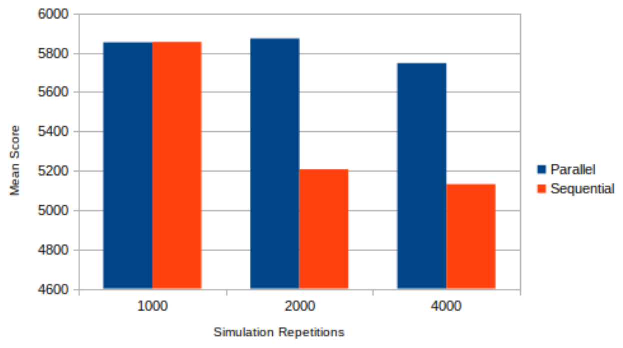

Figure 2: Mean Score vs Simulation Repetitions

The second algorithm to test was Hill Climbing. Since Hill Climbing was expected to perform much better at lower values, instead of simulation repetitions being modified between runs, the allowable run time was changed. The number of allowed running time was set to 10, 20, and 60 seconds, with each running for 1000 simulation repetitions. Again, the number of iterations and the score were recorded for a set of six identical runs of the sequential and parallel programs.

Table 4. Hill Climbing Sequential Algorithm Scores

Run	10 sec	20 sec	60 sec
1	12398	28236	24657
2	14081	14606	22712
3	11393	19902	37261
4	11733	12715	31179
5	10252	20700	29641
6	13614	15577	30728
Mean	12245.2	18622.7	29363
Std Dev	1305.66	5142.61	4725.80

Table 4 shows the scores returned by the sequential algorithm, by allowable running time. Again, as expected, as the algorithm is allowed to run for longer sets of time, the mean score it achieves is higher. Although the standard deviation is high, it is expected that given enough time, higher scores could be found, eventually settling into a local maximum. Table 5 is the scores for the parallel implementation of the Hill Climbing algorithm, and it shows a very different picture.

Table 5. Hill Climbing Parallel Algorithm Scores

Run	10 sec	20 sec	60 sec
1	3077	3089	3089
2	2778	3082	3080
3	3032	2778	3089
4	2778	3089	3089
5	2778	2778	3077
6	3089	3089	3089
Mean	2922	2984.17	3085.5
Std Dev	145.04	145.8	5.02

The data shown in Table 5 is not expected, and likely indicates a problem with the algorithm as it is implemented. Although the standard deviation does trend downward, as expected, the mean scores achieved by the algorithm are not close to expected values. Some of the values are repeated, both across time trials and within the trials themselves. The data would indicate an encounter with a local maximum of some sort, but with so many iterations again occurring, similar to the Tree Search trials, it seems unlikely that the sequential algorithm would avoid this, while the parallel implementation would always fall into it.

Table 6. Hill Climbing Mean Iterations - Nodes Visited

Algorithm	10 sec	20 sec	60 sec
Sequential	34.8	65.3	188.2
Parallel	74401.8	145411	439479.5

Table 6 shows the mean number of nodes encountered while performing both sets of trial runs. The parallel algorithm once again visits a vast quantity more nodes than the sequential, but despite this, it returns markedly worse numbers. Figure 3 shows the drastic difference between the performance of the parallel and sequential implementations of the Hill Climbing algorithm. It would appear that there is more work to be done on the Hill Climbing algorithm.

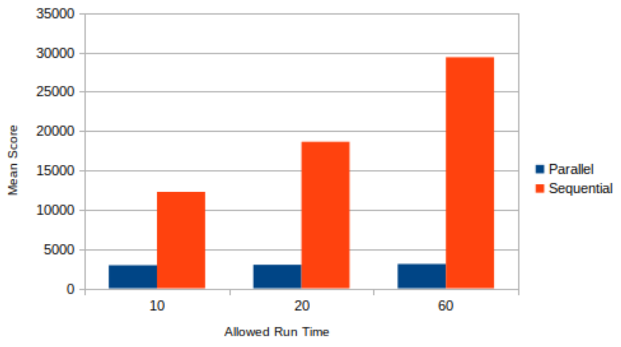

Figure 3: Mean Score vs Allowed Running Time

Simulated Annealing was ultimately not tested, as an unknown error at the time appeared to hang the program. It is not expected that the Simulated Annealing program would be of much performance increase in future, as detailed in the Performance section below.

5. Performance

Performance goals were achieved on Tree Search with performance roughly equal to the sequential implementation, with a slight edge to the parallel algorithm, if the scores are to be trusted. Although the number of iterations of the parallel algorithm were vastly higher than the same argument runs of the sequential, scores were overall the same. One might expect that with so many more iterations, the parallel implementation should be able to encounter better sets of values within the tree and evaluate them, but it is hard to say. Ultimately, hundreds (or thousands) of trial runs might need to take place in order to verify the algorithm correctness.

Performance goals were not achieved on Hill Climbing, as the data showed there to likely be a problem with the implementation of the AI algorithm in parallel. Given the drastic performance increase that the sequential Hill Climbing algorithm enjoys over Tree Search, it is definitely an area of future work for the parallel implementation.

CUDA utilizing dynamic parallelism worked well to get an initial performance boost to near sequential levels. Beforehand, every time the corn dog stand simulation kernel was called, a memory load operation to load the arguments to the GPU was performed, and a memory load from the GPU was performed at the simulation set’s end. With one happening for each and every new permutation of the AI algorithm, this hampered performance. It was only when dynamic parallelism was used, so that the AI algorithms stayed on the GPU itself, did performance finally increase.

CUDA hindered performance on many levels in this project. The modulus operation, used to in functions to generate random numbers, is not implemented in GPU hardware. Given that the probability portions of the simulation algorithm run hundreds of thousands of modulus operations within one simulation alone, the potential impact these would have on run time is significant.

Given enough development time, a faster implementation could be realized. As mentioned above, by developing a random number algorithm that does not use a modulus operation, the performance of the corn dog simulation algorithm itself could be increased. As well, redesigning the AI algorithms to launch several simulation kernels at once, with different unique arguments being sent to each, could enable far more simulations to run at once. The AI method could then select the best of them to continue. This area might not be as much use to the simulated annealing algorithm, as that method does not rely specifically on whether a score is better, or not. Rather it relies on random walks over time to avoid becoming trapped in a local maxima. Running multiple kernels and selecting the best of them would defeat the purpose of the inherent randomness to Simulated Annealing. Finally, a small performance gain might be realized by implementing a simple reduction sum algorithm at the tail end of the simulation algorithm, so that the additional work is not taken on by the single parent thread running the AI algorithm. As the number of simulation repetitions for each run is fairly low, this might not bring much discernible improvement.

This particular problem seems only limited by the processor speeds. The simulation itself does not require much memory, and it is the number of operations per simulation repetition that drives performance. With faster GPU hardware in the future, faster speeds could be possible, and more complex simulations could be run.

6. Conclusion

The ultimate goal of the project was to implement a simulation of a corn dog stand, where an ideal setup for the stand could be determined. Although apparently it can be solved in a sequential implementation, a parallel implementation could in theory be used to get results far faster. In the Tree Search verification tests, it was shown that although the parallel implementation did result in higher scores, standard deviation of those scores was far higher than sequential. As well, in the case of Tree Search, there was a performance increase, which could be increased by the launch of multiple kernels at once within the parent Tree Search kernel.

Simulated Annealing was ultimately discarded as a potential method, regardless of the errors it encountered, as the structure of the algorithm does not lend itself well to a parallel algorithm. Using the launch of many additional kernels and selecting the best of the returned score, assuming one is found, effectively turns the algorithm into Hill Climbing. Sadly, performance gains were not achieved with the aforementioned Hill Climbing algorithm. The verification test data showed that it likely was not implemented correctly, to the same degree as the sequential.

On all of the test verification trials, the primary constant was the apparent increase in the number of iterations through the AI algorithms in a given set of time for the parallel implementation. Being that more iterations results in more nodes visited to test more simulation variable sets, the parallel implementation still shows great promise and it shows that this is an area worthy of continued study.

7. References

[1] https://www.mnstatefair.org/about-the-fair/attendance/. *Online, mnstatefair.org*

[2] Dijkstra, E.W. ``Cooperating sequential processes,'' *Technical Report EWD-123*, Eindhoven University of Technology, The Netherlands, 1965.

[3] Damouth, Daniel E. and Durfee, Edmund H. ``Local search in the coordination of intelligent agents,'' *Proceedings of the National Conference on Artificial Intelligence*, vol. 2, pp. 1437-1437, 1994.

[4] https://www.codeproject.com/Articles/13789/Simulated-Annealing-Example-in-C. *Online, codeproject.com*