A STUDY IN ACCELERATION OF SELECTED ARTIFICIAL INTELLIGENCE COMPUTATIONS USING THREAD-LEVEL PARALLELISM

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Electrical Engineering

in the

College of Graduate Studies

University of Idaho

by

Kisron Niles

July 2014

Major Professor: Gregory W. Donohoe, Ph.D.

Authorization to Submit Thesis

This thesis of Kisron Niles, submitted for the degree of Master of Science with a major in Electrical Engineering and titled "A Study in Acceleration of Selected Artificial Intelligence Computations Using Thread-Level Parallelism," has been reviewed in final form. Permission, as indicated by the signatures and dates given below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor: _____ Date: _____
Dr. Gregory W. Donohoe

Committee Members: _____ Date: _____
Dr. Milos Manic

_____ Date: _____
Dr. James Frenzel

Department Administrator: _____ Date: _____
Dr. Fred Barlow

Discipline's College Dean: _____ Date: _____
Dr. Larry Stauffer

Final Approval and Acceptance

Dean of the College of Graduate Studies _____ Date: _____
Dr. Jie Chen

Abstract

This study demonstrates a practical implementation of selected Artificial Intelligence computations using thread-level parallelism with C++11 on a four-core processor, with a primary goal of reducing execution times. These programs spend a large percentage of the execution time searching and learning, both of which can benefit from the speed advantages offered by thread-level parallelism. As computer hardware architectures have moved from serial execution to concurrent multithreaded execution, new software programming techniques are needed to take advantage of concurrent hardware. C++11 is a new C++ standard with many new features and this study will focus on applying the new multithreading libraries including the new atomic memory model available in C++11 to solve these problems. Serial and multithreaded programs are compared in terms of execution time and programming effort to help determine when thread-level parallel designs should be considered.

Acknowledgements

My sincere thanks go out to the University of Idaho for enabling me to pursue my MSEE while working and living in Colorado through the excellent Engineering Outreach program and to Dr. Donohoe and my thesis committee for all of their inputs and wisdom in assisting me with the development of this thesis. Thank you also to all of my friends and family who have been very patient through this entire process.

Table of Contents

List of Figures

List of Tables

List of Equations

Chapter 1: Background

Despite decades of increasing processor performance, there are still many current problems that require even greater processing capability than a single processor can provide such as drug discovery, climate modeling and big data analysis as stated in [1]. As single processor performance has increased by Moore's Law in terms of execution frequency, the power consumption and associated heat have also increased. As frequency scaling continued, the increase in heat grew to a point where unreliable processor behavior existed, and this heat threshold has become the upper limit for scaling the frequency. Increases in transistor density can still be realized, so processor designers began attaching multiple single processors to the same chip, yielding the current standard of multicore processors, where core is synonymous with a Central Processing Unit (CPU) [1]. Since parallelism, including multicore, constitutes the current path forward toward increasing processor performance, industries requiring increased computational power have shifted to parallel programming techniques including execution of multiple, concurrent process instances at the operating system (OS) level. Multiple concurrent processes are helpful when multiple programs must be executed at once, but they do not speed up the execution of a single program instance. To increase the performance of a single program, parallel programming techniques must be applied, using programming language libraries to facilitate interaction with the multicore hardware [1].

Chapter 2: Types of Parallelism

Data-level parallelism uses a single instruction, multiple data streams (SIMD) architecture, where the same operation is executed on multiple data items in parallel. In SIMD computers, each processor has its own memory, but there is a single instruction memory and a single control processor. Vector machines are the largest class of SIMD architectures, where a traditional Graphics Processing Unit (GPU) is a vector machine that is used to handle dedicated processing for high performance graphics needs [6]. In graphics, the same operations are applied repeatedly to streams of data to create complex graphics objects from basic building blocks such as triangles and lines, tasks that are ideal for the SIMD model.

Thread-level parallelism uses a multiple instruction streams, multiple data streams architecture (MIMD). In this architecture, each processor has its own instructions and operates on its own data. Since multiple threads operate in parallel, this architecture exploits thread-level parallelism [6]. The test computer used in this research is an MIMD computer with four processors (cores), each capable of executing two threads per core. As MIMD suggests, each core has its own instructions and memory allowing for thread-level parallel tasks to be run simultaneously. Since thread-level parallelism is generally more flexible than data-level parallelism, it is generally more applicable to diverse implementations [6]. Since this research addresses parallel programs for differing AI computations, thread-level parallelism is well-suited to tackle the various parallel designs needed for all of the problems.

Hybrid parallel implementations exist, such as General Purpose Graphics Processing Units (GPGPUs) which can be used for general purpose computations by using a

non-graphics application programming interface (API) such as NVIDIA's CUDA which allows the GPU to be programmed using C constructs [7]. GPU Algorithmic Logic Units (ALUs) on a single GPU core use SIMD parallelism, but the cores of current GPUs can execute independent instruction streams, behaving like a MIMD system [1].

Each of these choices – SIMD, MIMD, or hybrid – requires an investment in hardware purchase and software development. This thesis will focus on MIMD parallelism.

We focus on the use of thread-level parallelism to reduce the execution time required by algorithms to maintain high efficiency and increase speedup relative to the serial programs. The speedup value is a direct measure of the performance benefit gained, but it comes with costs associated with software development. Speedup can be justified when programming efforts in real-world cases use code that is frequently applied and reused over time and in programs that require long execution times. A quantitative measurement can be made by calculating a programming efficiency value which is used to determine if a multithread implementation is worth the efforts to develop it. The programming efficiency value $E_{programming}$ was derived by the author and is calculated as:

$$E_{programming} = \frac{\sum \left(execution_{serial} - execution_{multithread} \right)}{developTime_{multithread} - developTime_{serial}}$$

Equation 1 Programming Efficiency Calculation

The sum is over the lifecycle of the code and the efforts are justified when efficiency values exceed 1.0. For real-time programs, it is the author's opinion that qualitative justification may outweigh any quantitative factors since increasing the quality of an answer in a real-time program through multithreading may allow more calculations per calculation cycle, yielding better answers. Determining the added value gained by using a multithread program over a serial program requires estimates beforehand that may not be exact, so expert judgment is a viable measurement for decision-making in the real-world.

Chapter 3: Selected Artificial Intelligence Computations

This section covers the selected problems along with the serial designs and implementations used to study the effects of multithreading in C++ 11 on a subset of AI algorithms. Table 1 below shows a summary of the problems, associated data structures, algorithms used and goal solutions.

Table 1 Summary of Problems, Data Structures, Algorithms and Goals

Problem	Data Structures	Algorithm	Goal
Transmission Tower Placement	2-D graph with transmitter placements	Hill Climbing with Simulated Annealing	Achieve a global or near-global optimal coverage of receivers
	2-D graph with receiver placements (static)		
Pathfinding	2-D map representing the terrain and environment	Breadth First Search, realized with a First In First Out Queue	Find a path to the goal state on the 2-D map from the start state
	Tree to realize search states		
Connect 4	Game board to keep track of the current game state	Recursive Minimax search with Alpha-Beta Pruning	A win for the AI player
	Test board to simulate moves and perform heuristic evaluations on the AI moves		
	Tree to realize all moves to a user-defined depth		
Deciphering Encryption	Population of keys	Genetic Algorithm with mutation, crossover and random key generation	Find a key that decrypts a cipher passage into an English passage
	Tournament to evaluate keys		
Planetary Lander	Artificial Neural Network that takes various inputs from the lander	Hill climbing with random restarts	Train the lander to successfully land in an environment with random wind

3.1 Hill Climbing with Simulated Annealing: Tower Placement

Hill climbing is a search algorithm that continually moves in the direction of increasing fitness, which is uphill in a “problem space” [3]. One of the major challenges to this search occurs when a problem space has numerous peaks varying in fitness values as shown in Figure 1 below. When the top of a peak is found there are no adjacent states that are better, so the other peaks in the problem are not considered, causing the search to potentially get stuck at a non-optimal peak which is commonly referred to as a local optimum. One way to handle this issue is simulated annealing, a search that combines hill climbing with a random walk which can yield both efficiency and completeness [3].

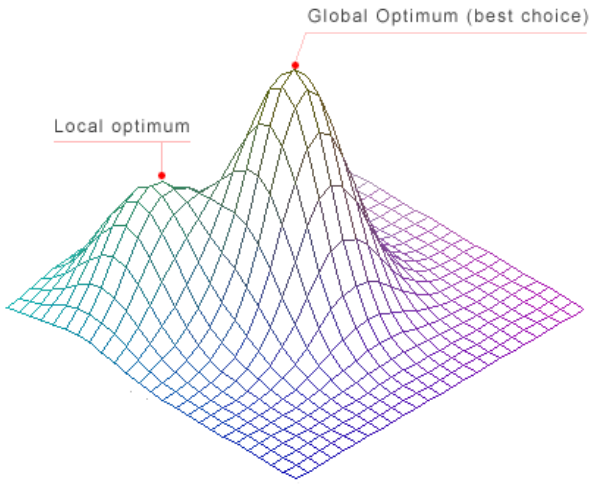

Figure 1 Local and Global Optima

In this problem, 30 transmission towers are used to cover as many receivers as possible within 129 US and Canadian cities using only hill climbing and simulated annealing in 2-D space. The units are not to the scale of the actual US and Canadian city locations but are meant to symbolically represent cities for the sake of

the problem relevance. Each transmission tower covers a radius of 1 unit and one transmission tower must cover multiple cities if possible, since there are many more receivers than towers. The towers are initially placed randomly and then moved around using hill climbing and simulated annealing until a desired number of search iterations is completed. The fitness value is calculated based on how many cities are covered by the towers and by the distance of the closest receiver to a transmitter, and vice versa. This problem has many local optima so it requires simulated annealing to search widely which increases the likelihood of finding the global optimum.

The program entails five key elements: random transmitter placement for initialization, cumulative fitness evaluation of all the transmitter positions relative to the receiver positions, simulated annealing to get out of local optima, movement of the transmitter positions to search by hill climbing and checking for a goal state, which indicates completion of the run. The goal state in this problem is a user-defined count limit, and the run is complete when the number of iterations completed equals the count limit value. After the transmitters are randomly placed at initial locations, one transmitter is selected each search iteration, moved by a random x or y value and the fitness of the new transmitter location is evaluated. If a better fitness is found than the current best fitness, the new transmitter location is stored as the best location for that transmitter and the best fitness value is updated. Otherwise, the transmitter is returned to its previous position. Simulated annealing is achieved using a temperature value with a warming and cooling schedule. A random unit generator

is used to get a random double from 0 to 1 which is compared to the annealing value. The annealing value was chosen by the author and is shown below in Equation 2.

$$annealing = e^{\frac{\delta_{fitness}}{temperature}}$$

Equation 2 Annealing Calculation

The delta value is the difference between the most recent fitness calculated and the best fitness. If the current fitness is better than the best fitness so far, delta will be a positive value and the annealing value will always be greater than or equal to 1, which will make the program accept this new fitness as the best fitness regardless of the temperature. If the current fitness is less than the best fitness, it will be accepted if the random number between 0 and 1 is less than the annealing value. The temperature value is adjusted with a heating / cooling schedule. As the temperature value approaches zero, the acceptance will approach that of hill climbing. When the temperature value is raised again, the acceptance will allow more freedom of exploration to move out of local optima. Figure 2 below illustrates a sample run of the algorithm, showing the city / receiver locations, the initial random placement of the transmitters and the placement of the transmitters after the search has completed. Figure 3 below shows a flow chart of the serial program.

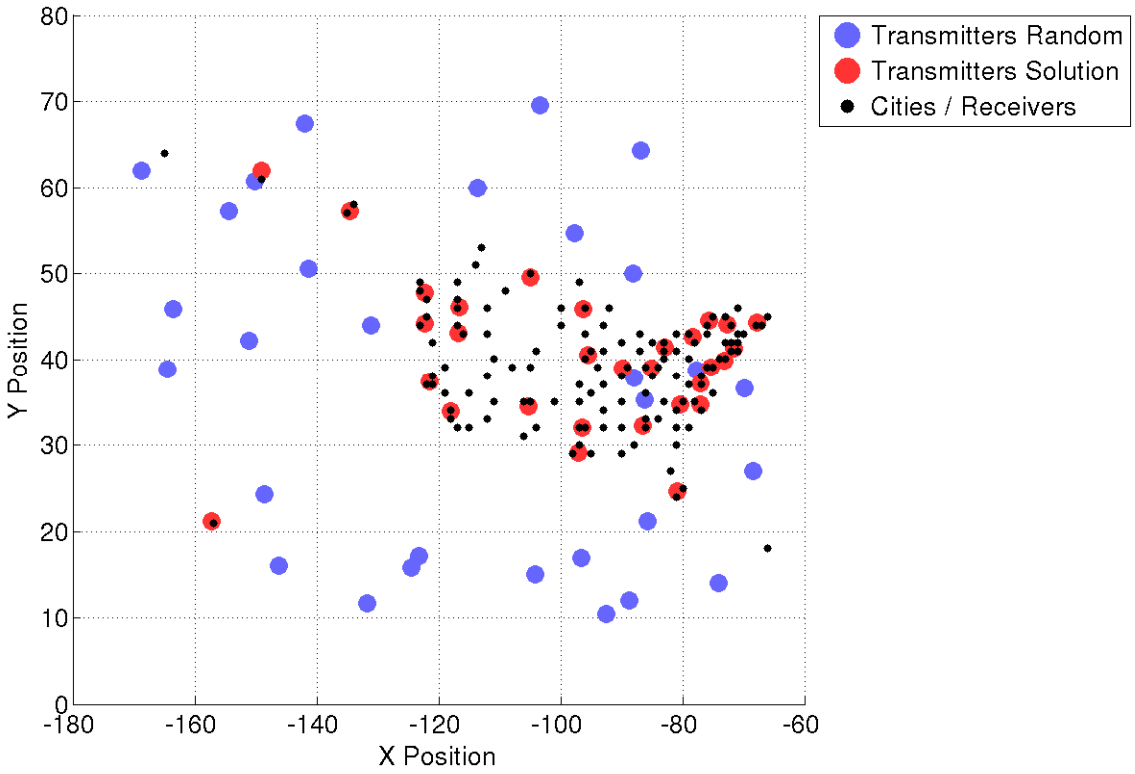

Figure 2 Visualization of Hill Climbing Problem Space

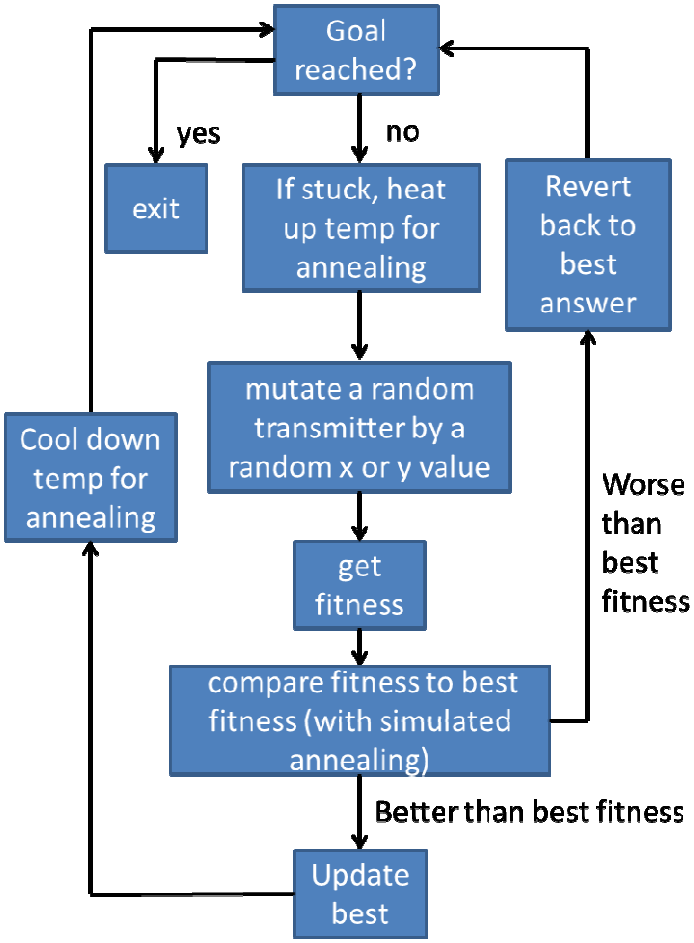

Figure 3 Flow Chart of Transmission Tower Placement Serial Program

3.2 Pathfinding: Breadth First Search

Breadth first search (BFS) creates a search tree that always expands the shallowest node using a first in first out (FIFO) queue. Each node is expanded until the goal state is reached. BFS is an uninformed search that is exhaustive and complete, meaning it is guaranteed to find a goal state when one exists. If every movement between states has the same cost then BFS is optimal and will find the best path. BFS is not optimal if the path costs are not uniform. The downside to BFS is that it has exponentially increasing time and space complexities as the number of nodes in the search tree increases [3].

In this problem, an agent is used to search within a 2-D terrain which has associated costs with each move. The terrain is represented as a map with ASCII characters that indicate the terrain. An example terrain is shown below in Figure 4 and the start and goal positions are given by the file that is read in which also includes the search map. The program run completes when the goal position is found by the search. The path cost and description for each character are shown below in Table 2.

```
M M M h h f f f f f f f f f
M M M M M h h f f f f f f f
f R f f f W W W W W F F F F
f R f f W W W W W W W W F F
f R R f f f W W W W W r W r
f f R R R R f f f f r r f f
f f f f f f R f f f f r f f
h f f f f f R R R R R R R R
M h h f f f f f f f f f f f
M M h h h f f f f f f f f f
```

Figure 4 Sample Map of Terrain for Searching

Table 2 Map Terrain and Cost Legend

Character	Meaning	Movement Cost
R	Road	1
f	Field	2
F	Forest	4
h	Hills	5
r	River	7
M	mountains	10
W	Water	cannot be entered

The basic components of this algorithm are a search tree with node objects that represent each searched state of the path. The nodes also contain an action that the

node performs relative to its parent's location and a cost for that particular move. Each node also contains a pointer back to its parent unless it is the root node. A FIFO queue is used to store unexpanded nodes and a Boolean vector is used to keep track of which nodes have been explored. Shared pointers were used since C++ 11 supports them, and they handle memory management automatically so that pointers are properly freed when they go out of scope. For this BFS, all path costs are calculated but not used in the search since it is based solely on the FIFO queue and not on path costs. A class object is used to organize the variables and functions for the problem, and a structure is used to store and organize the data for each node of the tree. The node structure holds a pointer to the child's parent which creates an upward associative tree. The tree is expanded with the FIFO queue until the goal state is reached. During each search iteration, the next member from the queue is popped and used as the parent to generate four children representing each move, with each child pointing to the parent. A simple Boolean vector that is the same size as the map array is also used to keep track of the explored set and each map location is marked as true when that space has been searched. This strategy ensures that redundant states are not searched. When the goal is reached, the goal state is back-propagated to the start state to trace the search path and calculate the path cost. The console output for a search is shown below in Figure 5.

```
TTTTTTTSTTTTTTT
TTTTTTT*TTTTTTT
TTTTTTT*TTTTTTT
TTTTTTT*TTTTTTT
TTTTTTT*TTTTTTT
TTTTTTT*TTTTTTT
TTTTTTT*TTTTTTT
TTTTTTT*TTTTTTT
TTTTTTT*TTTTTTT
TTTT****TTTTTTT
TTT**WWWWWTTTTT
TTT*WWWWWWWWTTT
TTT*TTWWWWWWWTT
TTT*TTTTTTWWTTT
TTT*TTTTTTTTTTT
TTT*TTTTTTTTTTT
TTT*TTTTTRTTTTT
TTT*TTTTfffTTTT
TTT****EffffTTT
TTTTTTTffffffTf

Travelled = T
Start = S
End = E
Solution Path = *

cells searched: 263 / 300
cells to solution: 27
nodes created: 1009
total cost: 68
```

Figure 5 Pathfinding BFS Console Output After Search

Figure 6 below shows a flow chart of the serial program.

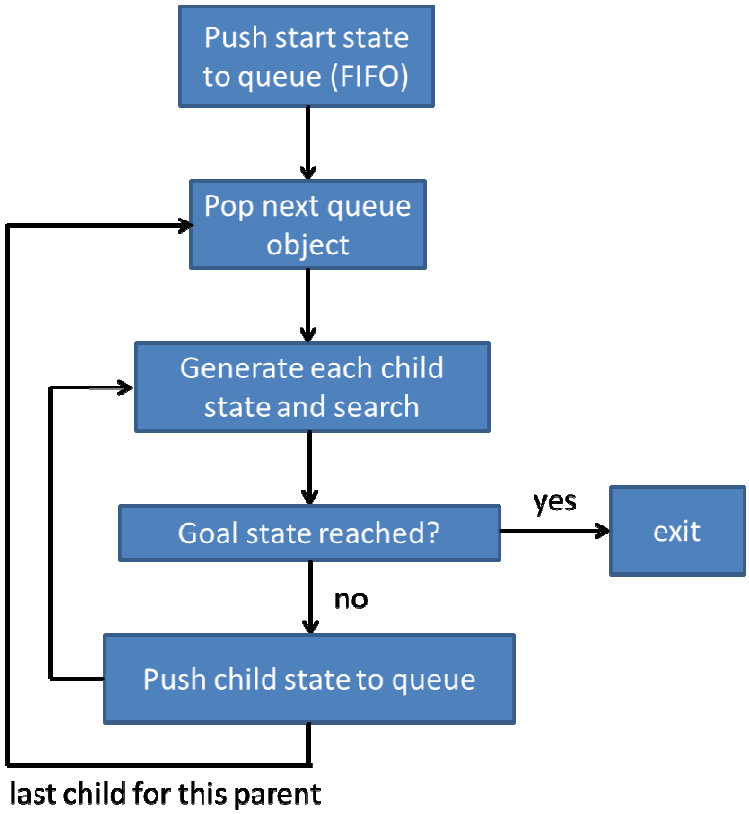

Figure 6 Flow Chart of Pathfinding BFS Serial Program

3.3 Game Playing: Connect 4

Connect 4 is a 2-player, deterministic game that is fully observable where each player drops checkers onto a vertical board in a top-down fashion. One player uses black checkers and the other uses red. The first player to get four of his or her checkers in a row on the board wins. Four in a row counts in vertical, horizontal and diagonal directions. The board is shown below in Figure 7.

Figure 7 Connect 4 Game Visualization

In this problem, the computer is the AI player and the human is the opponent. Connect 4 does not have a high depth compared to many other games since an entire game can only consist of a maximum of 42 moves; however, this is too many moves for the AI player to always see a goal state such as a win for either player or a draw. The inability to see the goal state requires implementation of a heuristic evaluation that will determine what move is best when neither an AI player win nor a block against the human player's win is available. Many search techniques can be used in a 2-player game like Connect 4, but this project focuses solely on the minimax algorithm. The minimax search algorithm used here is a recursive limited depth-first search that creates a tree with each node representing a move in the game. See Figure 8 below to visualize the minimax process. The AI player is the "Max" player and the human player is "Min". If it is Min's turn at a given depth, the function will return the minimum outcome move for that depth, based on the assumption that Min will pick the best move available. If it is Max's turn at a given

depth, the maximum outcome move for that depth will be returned since that is the best move available for the Max player [3].

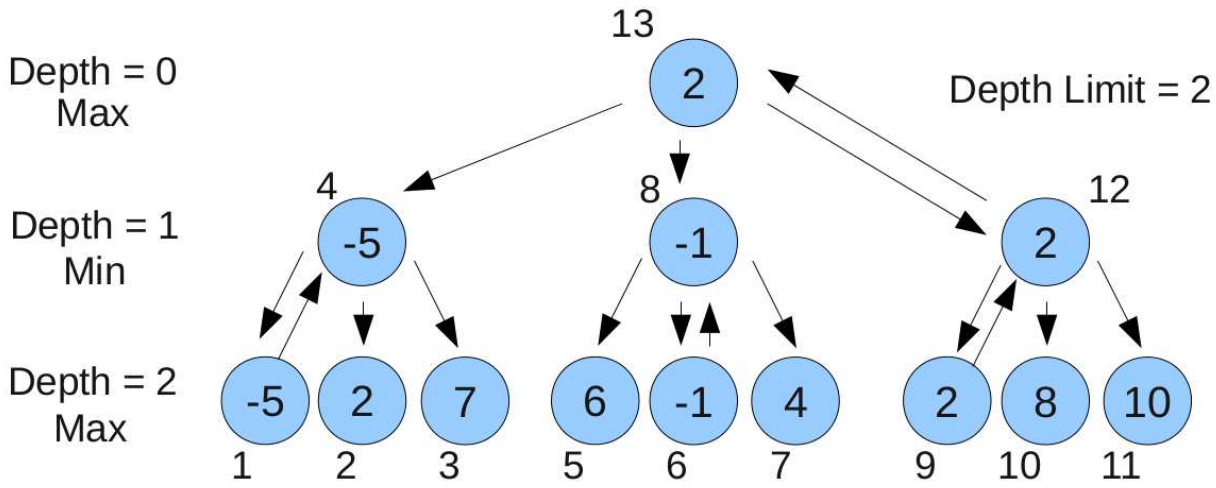

Figure 8 Minimax Algorithm

The heuristic evaluation is a function that evaluates a set of moves to determine their value to the player whose turn it is. The function will determine the value of a move from the perspective of the AI and adjust the return value based on whose turn is being evaluated. A good evaluation function will set up multiple, redundant win scenarios while also blocking optimal scenarios for the human. This evaluation can be effectively implemented for Connect 4 with fast heuristic checking.

An important technique in optimizing the execution times of the minimax algorithm is alpha-beta pruning. If a move is evaluated and found to be a worse choice than the current best choice for the player whose turn is being evaluated, this entire branch all the way down to the depth of the search can be disregarded. Applying this pruning removes several branches and drastically minimizes the search space and time complexity of the limited depth-first search without affecting the outcome of the decision [3]. A sample of the game play is shown below in Figure 9.

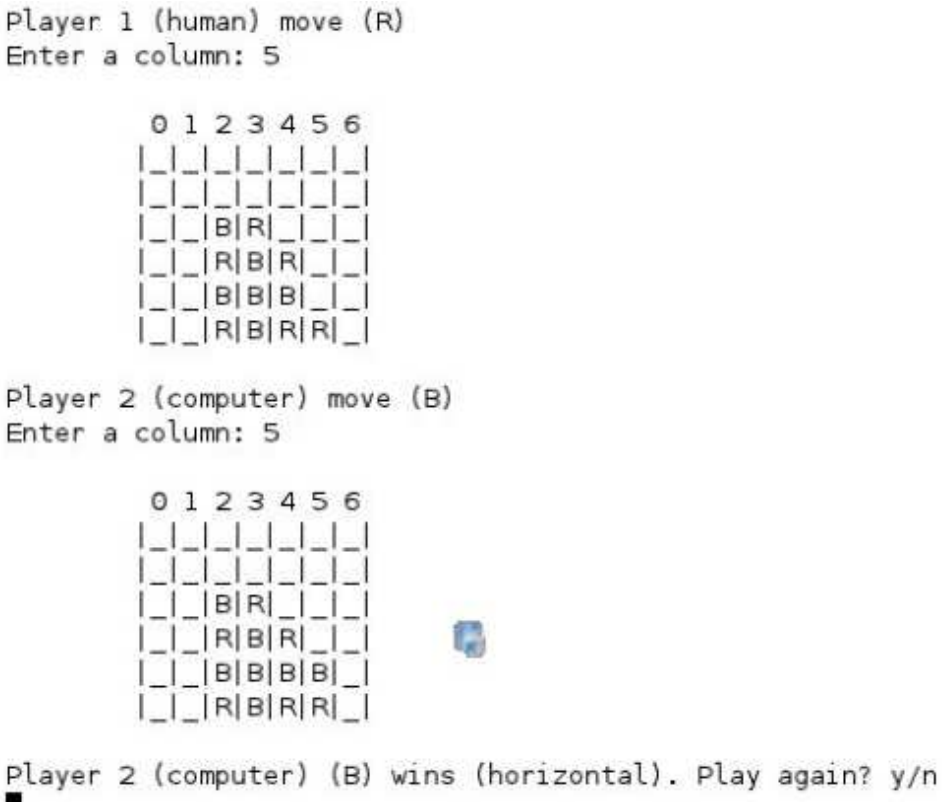

Figure 9 Connect 4 Gameplay in Console

Figure 10 shows a flow chart of the serial program.

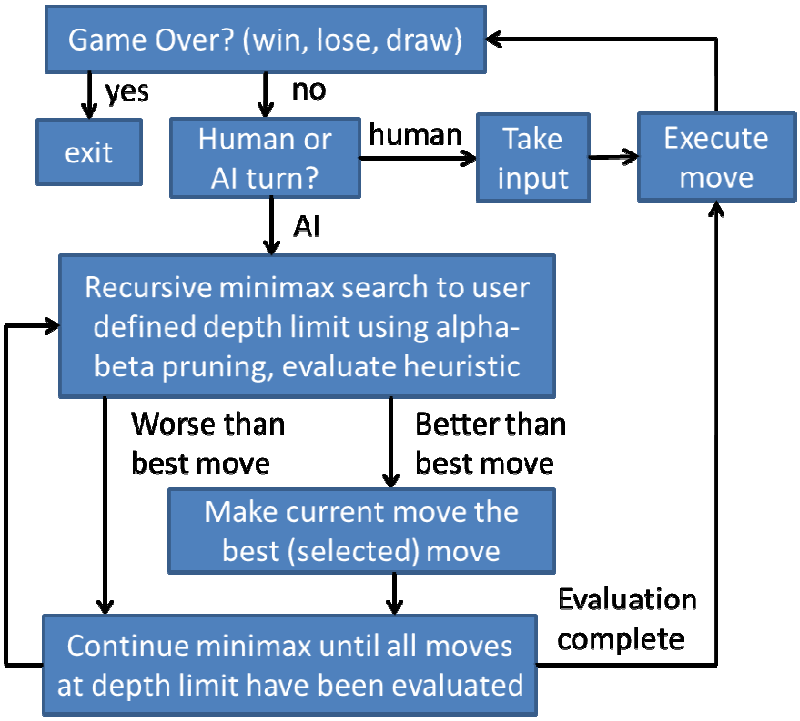

Figure 10 Flow Chart of Connect 4 Serial Program

3.4 Genetic Algorithms: Deciphering encryption

Deciphering encryption or decryption is a common process used to apply a key to sets of bits to convert an encrypted data set to an unencrypted set. This program uses a Genetic Algorithm (GA) that generates a key which is used to decrypt a clear-text passage as best as possible. A GA is an evolutionary algorithm that is largely considered as a function optimization method [2]. In this case, the goal is to decipher an encrypted passage to English, using an English digraph table to determine a fitness value that estimates the solution key's Englishness. The solution key is unknown, and the program runs until a count limit is reached, at which time the goal is reached.

In this problem, the message is encoded by taking the clear text and changing all uppercase letters to lowercase letters and removing all whitespace and punctuation.

The message is then enciphered with a substitution cipher determined by the key and the message is then broken up in to blocks of 5 letters each as shown below in Figure 11.

Clear text:	**Hello World**
Encrypted text:	**hzqqe oefqt**
Unencrypted alphabet:	**abcdefghijklmnopqrstuvwxyz**
Encrypted key:	**cpmtzkrhlsquniebdfaygwovjx**

Figure 11 Sample Encryption Method

An English digraph table, also known as a contact table, is used to correlate groupings or pairs of letters to their statistic occurrence within a large sample of English text. By using this table, the occurrence of the encrypted pairings from a given passage are compared to the unencrypted pairings of the digraph table to determine a fitness value for a given key. It is apparent that this approach yields diminishing results as the used encrypted passage grows smaller since the correlation is based on a probability model which degrades for small samples. A sample of the digraph table is shown below in Table 3. These pairings from the digraph table are normalized by dividing the occurrence of a given pairing by the total summed value of the occurrences from all of the pairings. The pairings from the encrypted passage are also normalized in the same fashion so the results can be compared accurately regardless of sample size.

Table 3 Non-Normalized English Digraph Table Sample

aa 11	**ab 122**	**ac 298**	**ad 210**	**ae 19**	**af 54**	**ag 116**	**ah 23**
ai 254	**aj 6**	**ak 154**	**al 632**	**am 231**	**an 1614**	**ao 11**	**ap 105**
aq 1	**ar 861**	**as 451**	**at 973**	**au 123**	**av 214**	**aw 55**	**ax 13**
ay 225	**az 16**	**ba 108**	**bb 31**	**bc 1**	**bd 1**	**be 597**	**bf 0**

A steady-state GA was used in this problem with a population size of 100. A class is used to organize the variables, parameters and functions for the problem. Structures are used to organize the data for the population individuals and the tournament. The tournament function randomly picks a specified number of members from the population and compares their fitness values. The population member with the best fitness (smallest number in this case) is kept in the population while the losers are replaced with random keys or crossover children. A tournament size of 3 was used in this program. The tournament structure also holds a reference to the population members that are being compared including a reference value to indicate the winner of the tournament.

The population is a vector which serves as a holder for many individuals, each containing a key and an associated fitness. Keys are randomly generated to initialize the population and each population member's fitness is evaluated. While determining the fitness of the members, a best position reference value is updated to point to the best member of the population.

The program runs for a set number of iterations, outlined by the atomic counter "count" which is compared to the count limit "tries". For each iteration, a mutation is made to the key of a randomly selected member of the population. The mutate function swaps a given number of key locations. A 2-op mutate will swap 2 key elements while a 3-op will swap 3 as shown below in Figure 12. The mutated key is then evaluated and assigned a fitness value. If the mutated key's fitness is better than the original key, it will replace it in the population. If the mutated key's fitness is worse, it will be discarded and the original key will remain unchanged. The mutation

method will start as a 2-op mutation. If a better fitness is not reached in 10 mutation attempts, a 3-op mutation will be used. If a better fitness is not reached in 10 mutation attempts, a 2-op method will be used again. Alternating between 2-op and 3-op mutations helps the search move out of local optima.

original	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
2-op	a	b	p	d	e	f	g	h	m	j	k	l	i	n	o	c	q	r	s	t	u	v	w	x	y	z
3-op	s	b	p	d	e	f	g	h	m	j	k	l	i	n	o	c	q	r	a	t	u	v	w	x	y	z

Figure 12 Mutation Example for 2-op and 3-op Operations

A crossover function is applied periodically to replace losers of a tournament run with crossover children of two randomly selected population members regardless of fitness value. The crossover function randomly selects two parents from the population, making sure not to select the members that have just lost in the tournament. These parents are combined using a permutation crossover. The permutation crossover randomly combines elements from each parent to create a child. The child replaces the losing tournament member regardless of fitness value. A permutation parameter of 14 is used so that approximately half of the key is taken from each parent to create the child. Great caution should be used when creating a permutation crossover function since it must include methods to ensure that duplicate key elements do not appear in the child. A sample permutation crossover is shown below in Table 4.

Table 4 Sample Permutation Crossover

* Number of elements to crossover from Parent 1 = 14																									
Parent 1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Parent 2	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Child	0	24	2	3	4	20	19	18	8	16	15	11	12	13	14	10	9	17	7	6	5	21	22	23	1	25

Similarly, a key generation function periodically applies random replacement keys for all losers of a tournament regardless of the fitness value.

The fitness function the author used for this program sums up the square of the difference between the normalized English digraph values and the normalized Cipher digraph values using Equation 3 shown below.

$$fitness = \sum \left(\frac{digraph_{english}(ij)}{\sum digraph_{english}(all)} - \frac{digraph_{cipher}(ij)}{\sum digraph_{cipher}(all)} \right)^2$$

Equation 3 Deciphering Encryption Fitness Calculation

This fitness function will determine if the English digraph and cipher digraph are close matches. All cipher digraph frequencies are compared to the English digraph frequencies, and the total sum of each comparison is the overall fitness for a particular cipher key. The cipher key with the lowest overall fitness is the closest match to the English key.

Figure 13 shows a flow chart of the serial program.

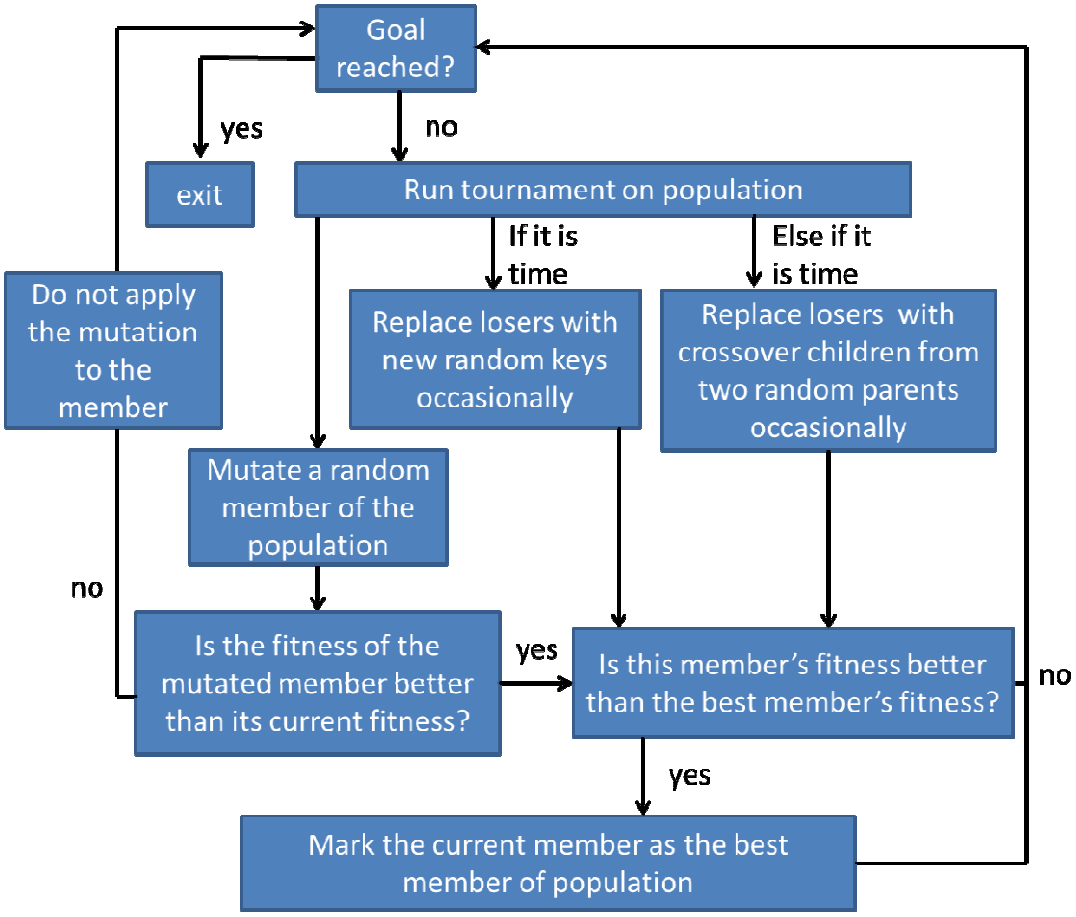

Figure 13 Flow Chart of Deciphering Encryption Serial Program

3.5 Artificial Neural Networks: Planetary Lander

The goal for the ANN Planetary Lander is to train the node weights of a feed-forward multi-layer ANN using hill climbing and random-restarts to control and safely land a planetary lander which is capable of landing within a small area at an acceptable velocity given a constant gravity and random wind values for the environment where the landing occurs. The training for this program optimizes a single landing, but optimization over multiple landings could be trained as well. This program will focus on the training of the ANN. Once the ANN is trained, it can be executed at will. Parallelism could be applied to execute concurrent landings, but the execution phase

is not covered since it is considered to be a standalone implementation at that point, outside of the training scope. The wind is a random number between +-.3 and is set at the start of each landing attempt. The problem space is 2-D and the ANN takes six inputs: height, xPosition, Yvelocity, Xvelocity, wind and fuel. The ANN generates two outputs, burn and thrust, to adjust the horizontal and vertical movement which offset the vertical gravity force and the horizontal wind force as shown below in Figure 14.

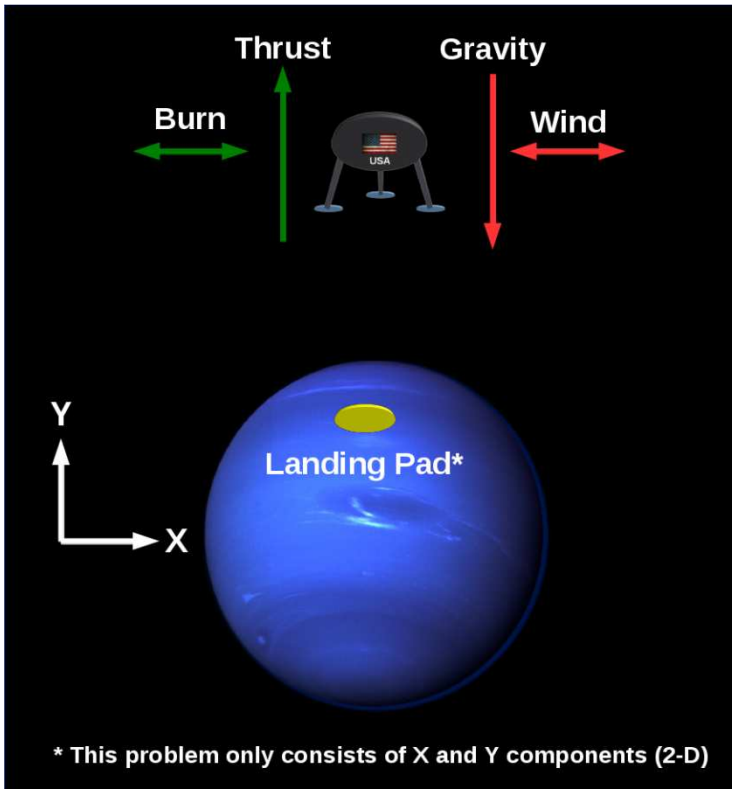

Figure 14 Planetary Lander Diagram

Additionally, there are 2 parameters that denote a successful landing:

1. **Yvelocity <= 4.0**
2. **-.2 =< xPosition <= .2**

This program uses a 3-layer ANN with 6 total nodes: 2 input, 2 hidden and 2 output. The algorithm implements a fully connected feed-forward multilayer network where

the final outputs are derived solely from the 6 inputs described above and the ANN node weights. By using a hidden layer, this network is able to extract higher-order statistics from its inputs [4]. A lander class is used to provide the appropriate parameters, variables and functions for the lander and a node class is used to organize each node's parameters and provide node-level functions such as updating or resetting weights. The node objects are organized into an ANN using a vector object. The ANN design is shown below in Figure 15.

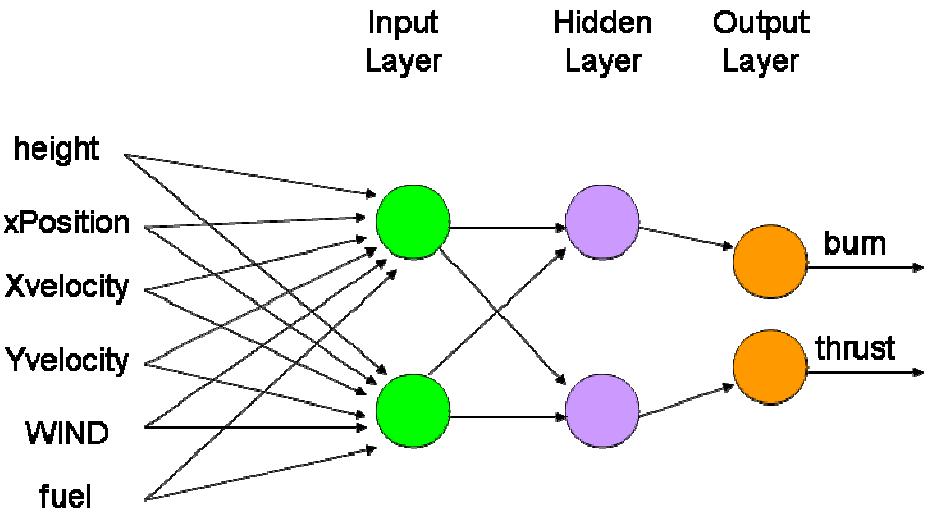

Figure 15 Planetary Lander Feed-Forward Multi-Layer ANN Structure

The search algorithm used is a random-restart hill climbing search algorithm where weights are reset to random values when the search gets stuck. The optimization of the fitness is considered stuck when a counter reaches a restart limit. The restart counter is incremented for each consecutive iteration that the current fitness is not better than the best fitness. If the current fitness is better than the best fitness, the restart counter is reset. For each training iteration, a single random input weight from a random node is chosen and adjusted to a random value between -1 to 1. The

planetary lander is landed n times, where n is an internal user-defined landing limit value. The combined fitness of these landings is then compared to the best fitness to determine if the current weight set yields an equal or better total fitness than the current best total fitness. If this case is true, the current weight will be updated to reflect the new weight that yields the better fitness value. Otherwise, the new weight will be reverted back to the original weight value. One landing was chosen per evaluation in this program since it was found it profiled the wind characteristics well and sped up execution times by limiting the total number of landings per run.

The fitness is calculated immediately following a landing by taking the xPosition and Yvelocity after landing. The author developed Equation 4 below to calculate the fitness for a landing.

$$fitness = |position_x| + |velocity_y - 2|$$

Equation 4 ANN Fitness Calculation

Notice that the current_fitness value is negative and is being maximized, where the absolute best fitness value possible is 0. The Yvelocity value is maximized to Yvelocity – 2 or Yvelocity = 2 since the tested ideal Yvelocity is somewhere relatively close to 2 on average. A Yvelocity value around 2 is ideal since it is centered symmetrically between 0 and 4, which are both the limits of what a successful landing Yvelocity is allowed to be.

When the lander is in the process of landing, an update function is called repeatedly until the landing is completed and the update function applies a lander control function. The lander update function calls the control function to feed the current

input values to the ANN and determine updated burn and thrust values. When being sent to the ANN, the Yvelocity, fuel and height values are all normalized by dividing their current values by their start values to get values between 0 and 1. These values are scaled so they will be closer to other input values which are much smaller. For each node, the inputs are multiplied by their corresponding weights and summed before being sent through the activation function to produce the node's output value. The activation that is used in this problem is a hyperbolic tangent (tanh) function, which yields an activation with outputs ranging from -1 to 1. A lambda value can be used within the tanh function to scale the activation and create a harder or softer activation. A lambda value of one, creating a Heaviside step function, is used for this problem. The values are propagated through the ANN and burn and thrust outputs are generated. A landing test is used for each update call to determine if a landing is not complete, complete and successful or complete but unsuccessful.

When calculating the burn and thrust values in the control function, these two outputs are both scaled to appropriate values to achieve a successful landing. The burn output values are scaled from +-1 to 1-6. Notice the range 1-6 is centered on 3.5, and this value was chosen based on testing and experimentation. Other ranges can be used with similar results such as 1-5 or 0-6, but no obvious improvements were observed using such ranges.

The thrust output values are scaled from +-1 to +-.5. This range of values is a good balance between constraints that are too tight and too loose; however, it should be noted that the random wind value makes it difficult to hone in on an exact best range.

Again, experimentation showed this range to be reasonable along with other ranges that were not chosen.

A configured reset count is established, and if this limit is reached, all of the ANN input weights are re-initialized to a random double between -1 and 1. The reset limit keeps track of how many training cycles have been executed without an equal or better fitness value being found. Simulated annealing could have been used, but it was found that the hill climbing local search with resets yielded adequate results.

Figure 16 shows a flow chart of the serial program.

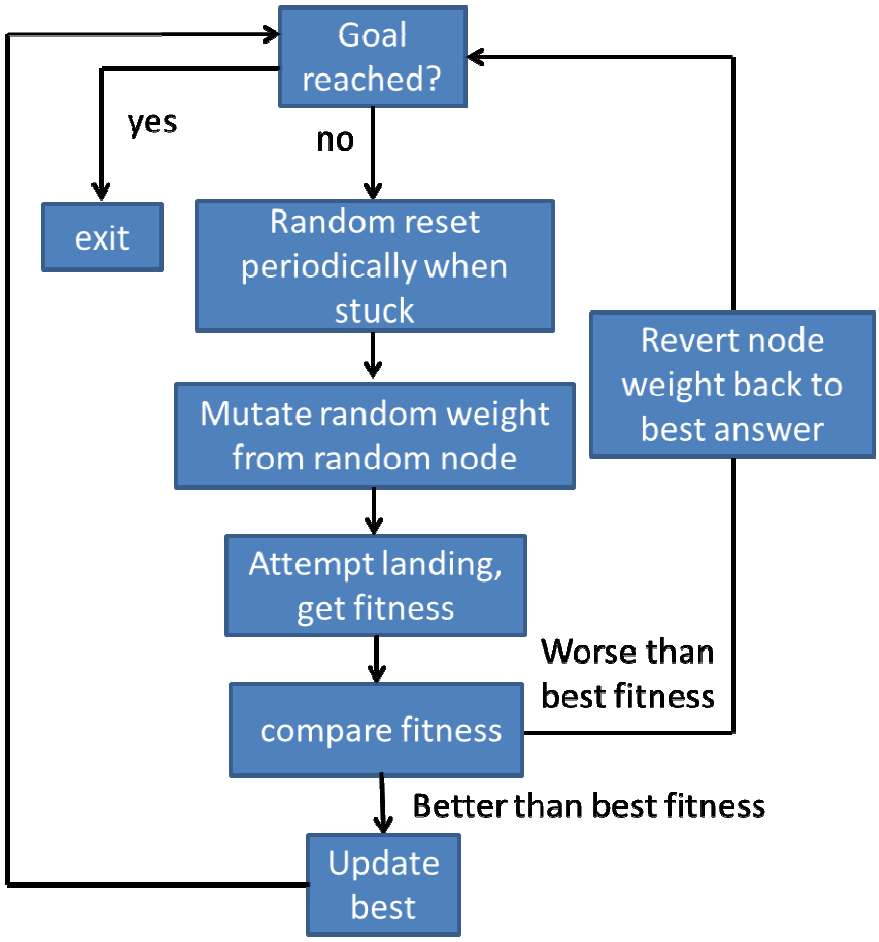

Figure 16 Flow Chart of Training for Planetary Lander Serial Program

Chapter 4: Parallel Implementation of Problems

This section covers the detailed parallel designs used to convert the serial programs listed in Section 3 to parallel programs. All of the algorithms from this section are shown in greater detail in Appendix B – Pseudocode.

4.1 Hill Climbing with Simulated Annealing: Tower Placement

The multithread programming design for this problem is very similar to the serial design with the addition of several synchronization mechanisms. The hill climbing and simulated annealing are treated the same, but multiple threads are used to search the problem space. Each thread has its own set of transmitter location values for x and y and searches independently with its own random number draws. If one thread finds a better fitness or passes the simulated annealing criteria to allow a poorer fitness, it will update the shared best transmitter locations to its current locations. Otherwise, it will synchronize its transmitter locations back to the shared best locations. The collection of the best transmitter locations is shared across all of the threads as a class variable, giving all of the threads read / write access since they are all executed within the class. Each thread is spawned with a function call to hillclimb::solve(), and variables are created with a scope that is limited to the running thread, giving local transmitter locations and other necessary non-synchronized variables. A global atomic “totalCount” variable is also used and incremented by all threads, and the program execution completes when this count limit is reached. This method is used to ensure that the number of search iterations is uniform between threads. Note that searching based on a fitness limit is misleading for this case since the execution time will be based solely on the random initialization of the transmitter

locations including the random numbers generated for the search. By using a count limit, this randomness does not influence the benchmarks for execution time. The simulated annealing temperature value and the heating / cooling schedules are shared between the threads so that the simulated annealing functionality is applied uniformly regardless of the number of threads running the program. Unique locks with mutexes are used to synchronize reads and writes associated with the best transmitter locations. A unique_lock and mutex are used to lock read / write access to the best transmitter collection's x locations while another set are used to lock the read / write access to its y locations. Using independent locks gives finer lock granularity over the parameters than locking both x and y, allowing an x location and y location to be updated simultaneously. An atomic variable is used for the best fitness so that it does not require a lock to be read or updated.

Figure 17 shows a flow chart of the parallel program.

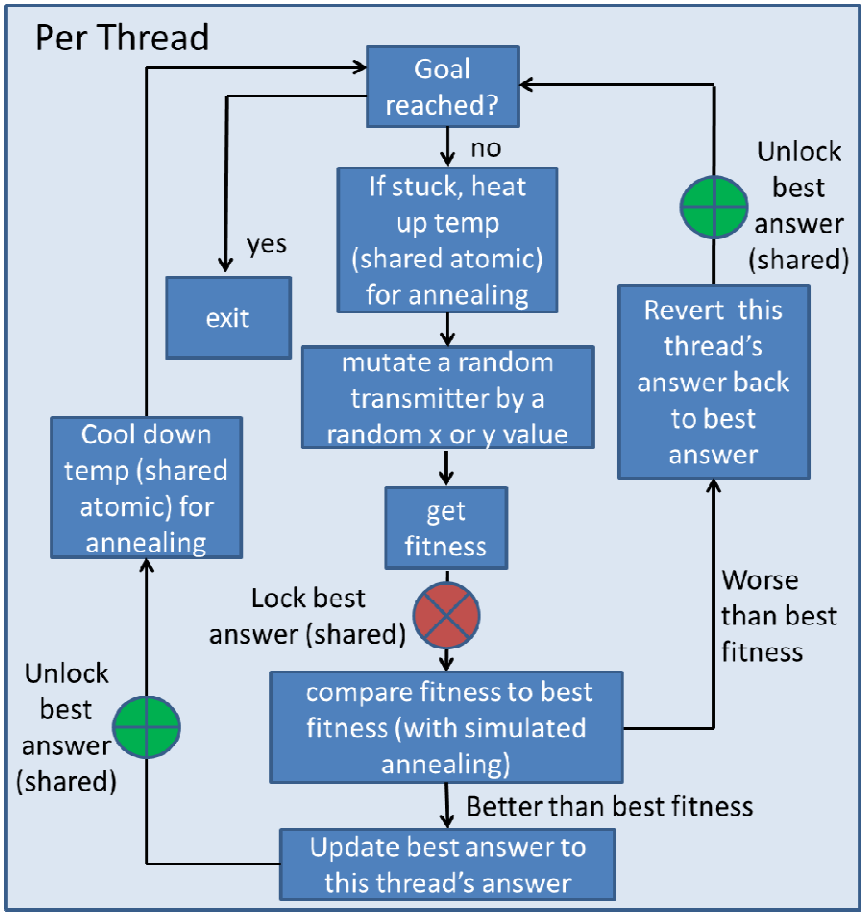

Figure 17 Flow Chart of Transmission Tower Placement Parallel Implementation

4.2 Pathfinding: Breadth First Search

Since the FIFO queue is used in this problem, a producer-consumer was deemed a viable multithread design. One producer thread is used to produce search states which are then pushed to a consumer queue where they are pulled by consumer threads. When a state is pushed to the consumer queue by the producer, a condition variable is used to notify a consumer thread that work is available. The consumer threads sleep and wait for the condition variable notification instead of polling until the work is available. Polling decreases system performance by keeping threads busy, while a condition variable allows threads to sleep until the work is available. A

sleeping thread frees up system resources so that other software threads can then make use of the available hardware thread. This strategy can be used in complicated problems to keep the system resources as busy as possible with useful work. Each consumer thread then searches the next state, updating information. A determineTerrain() function was created to simulate calculations for determining the terrain and associated movement cost. The original problem assumed the state was known, but determining the terrain is more realistic and more appropriate for comparing threads. Without a determineTerrain() function, multithreading was not as fast since the synchronization efforts took longer than the search, but this method was not practical for a real-world case. Once the state is searched, a thread waits until its turn to synchronize since the states must be expanded sequentially to meet the criteria for BFS. The consumer threads take the work from the consumer queue sequentially, but they will commonly get out of order by the end of the state expansion and determineTerrain() function call. When this sequence check is successful, the consumer thread will then push its state to a second queue which is the producer queue. A condition variable is then used to notify the producer queue that a state is ready to be expanded. This process is applied repeatedly, using the condition variables and queues to synchronize the work between the producer thread and the consumer threads. This method also ensures that the threads only access the queues when data is ready for use. Unique locks are used with two mutexes to synchronize the queue accesses between the threads. One unique lock is used for each queue and all queue accesses are locked since the queue is not a thread-safe

object. A run is complete when the goal state is reached. This state is defined by the user.

Figure 18 shows a flow chart of the parallel program.

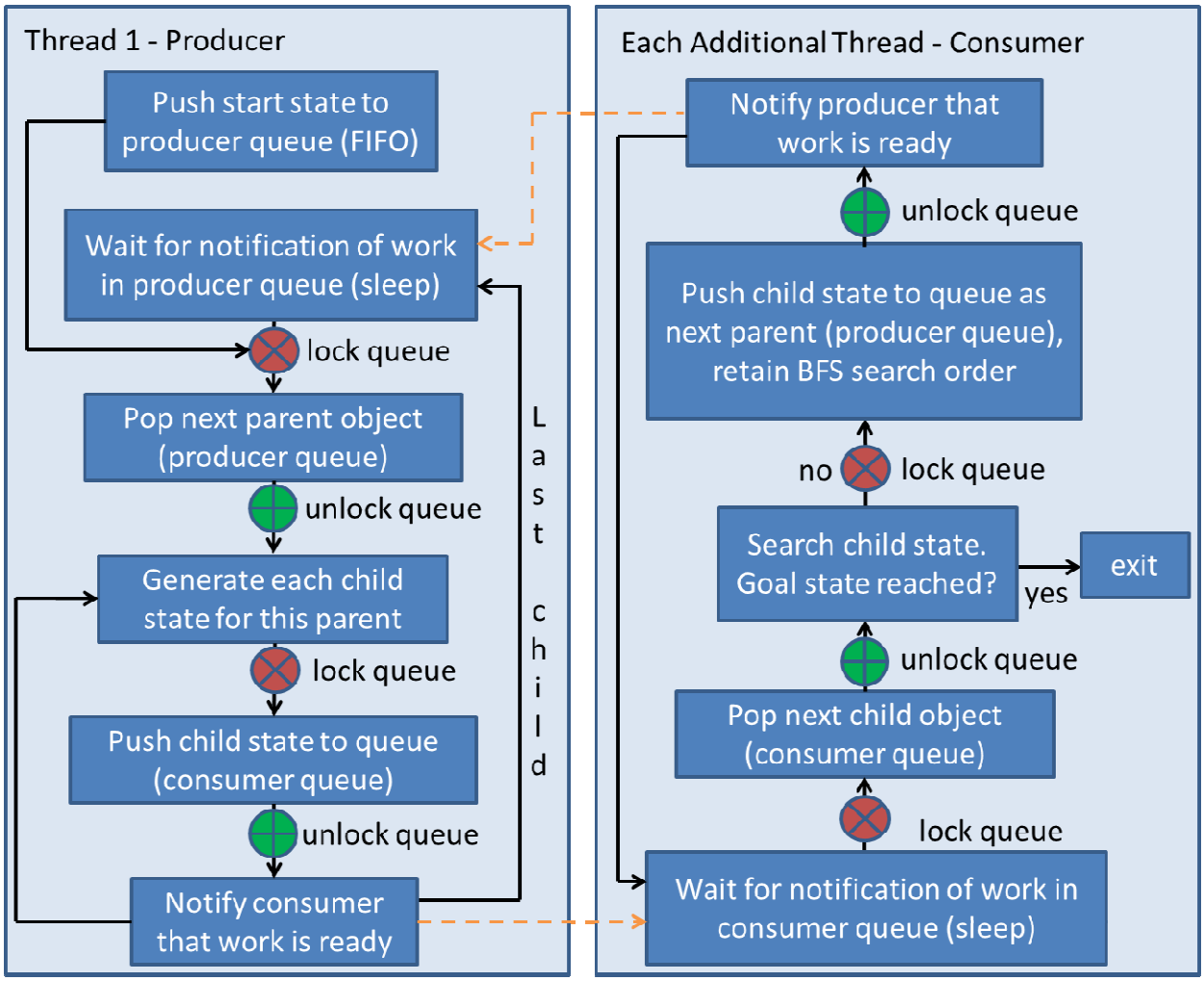

Figure 18 Flow Chart of Pathfinding BFS Parallel Program

4.3 Game Playing: Connect 4

This problem posed some issues when designing for multithreading since it uses a tree structure with recursive function calls. A producer-consumer design, similar to the one used in the BFS Pathfinding program could have been used, but a simpler, more-straightforward implementation was selected. Each column represents a top-

level branch of the tree, so the branches were split between the worker threads. For example, a run with 7 threads assigns the search of each column or tree branch to 1 thread per branch. For a run with 3 threads, 2 branches are assigned to 2 of the threads and 3 branches are assigned to the third thread to assign the work as evenly as possible. This implementation makes the synchronization of picking the best move very simple. The tree expansion and alpha-beta pruning techniques are still applied for each thread's search with the exception that each thread uses its own alpha and beta values for simplicity in implementation. A "threadMember" structure is used to store all of the local variables for each thread. For each AI turn, each thread copies the game state over to its local variables before searching so it can execute independently using up-to-date game information. The game state is locked during this read operation in case a write were to occur at the same time with an updated game state returned by another thread.

With this multithread design, each branch returns a fitness value and a corresponding move after searching, which are synchronized with a unique lock and mutex back to the best fitness variable and best move choice, both of which are shared across the class. The best move from all of the branches is selected as the AI's move. No other locks or synchronization are needed since the rest of the required variables are created within each thread's scope. When one of the players wins or a draw is reached, the run is complete.

Figure 19 shows a flow chart of the parallel program.

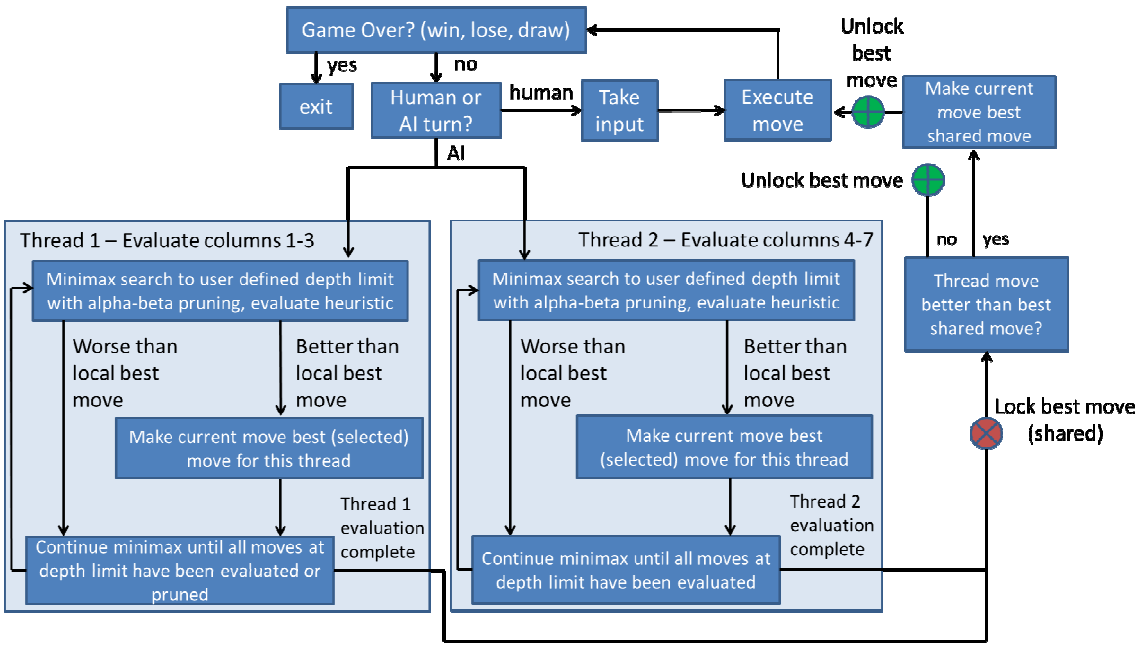

Figure 19 Flow Chart of Connect 4 Parallel Program

4.4 Genetic Algorithms: Deciphering encryption

The serial program is expanded to a multithreaded program by creating a population of 100 members times the number of threads being run. A run with 8 threads has a population size of 800. This allows each thread to work on a sub-group of the population that is 100 members, which is how the serial program operates. The same tournament runs, mutations, crossovers and random key generations are applied to each thread's population. The threads do not interact except to update the best position variable which references the best member in the population. A unique lock and mutex are used to serialize access to the best position. Counters related to mutations, key generation and crossovers are all within the scope of the thread so they can operate independently, and individual random number generators are used as well. A shared atomic counter is used to synchronize the iteration count. Again, a count limit is used to determine when the algorithm run is complete since this

program and its fitness values are highly dependent on random number generation. The count limit is a uniform comparison for runs of varying thread numbers.

Figure 20 shows a flow chart of the parallel program.

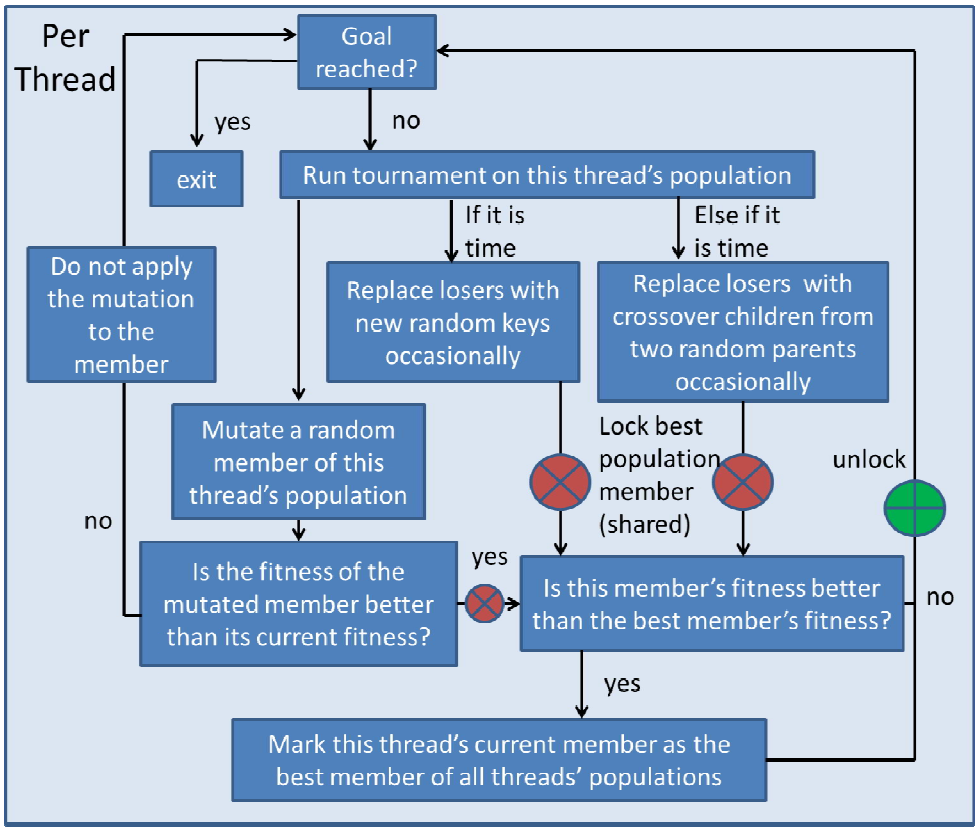

Figure 20 Flow Chart of Deciphering Encryption Parallel Program

4.5 Artificial Neural Networks: Planetary Lander

The multithread program for this problem is similar to the hill climbing program used in the Transmission Tower Placement problem except this program uses random resets instead of simulated annealing. Each thread implements its own ANN and searches independently with its own random number draws. A best ANN is shared across the threads and it contains the best set of weights and the best fitness value. If one thread finds better ANN than the best ANN, it will update the nodes and the fitness of the best ANN to reflect its own. Otherwise, it will synchronize its nodes and

fitness back to those of the best ANN. A unique lock with a mutex is used to provide serial access to the best ANN. An atomic variable, no_change, is used to determine when a reset should occur based on a tryLimit variable threshold. When no_change is greater or equal to the tryLimit, the best ANN weights are reset randomly, and all of the thread ANN's are synced to reflect the changes. This process ensures that the number of resets will be consistent regardless of how many threads are run. Each iteration of the program sets the lander to initial values, runs the update function until a landing is achieved, calculates a fitness value based on the landing and then trains the ANN based on the fitness. The program runs for a defined number of iterations, based on an atomic counter. Using a count limit instead of a fitness limit ensures that the execution time comparisons are more consistent between runs for varying numbers of threads.

Figure 21 shows a flow chart of the parallel program.

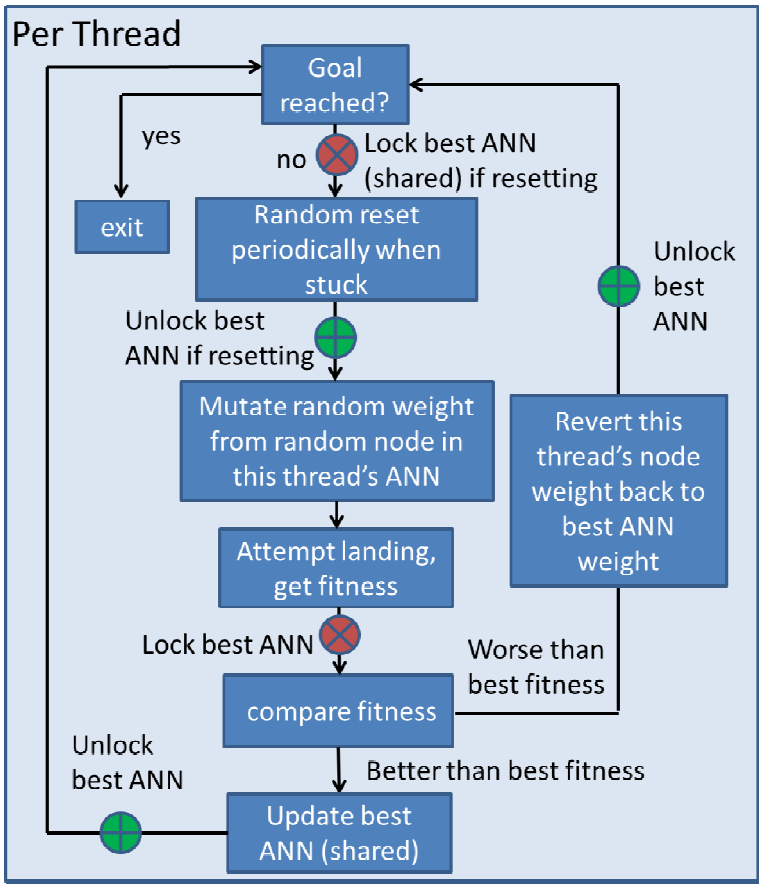

Figure 21 Flow Chart of Planetary Lander Parallel Program

Chapter 5: Experimental Setup

For this study, a computer with a 2nd generation Intel Core i7-2670QM processor was used. This processor has a 6 MB Intel Smart Cache which is shared across its 4 cores. The Intel Smart Cache is a last level cache (L3 in this case) that allows all of the cores to access and share all of the cache space. This approach allows for more flexibility and efficiency since the full cache space can be dedicated from a single core up to all cores, depending on how many are active. Shared data between cores only has to be loaded into the cache once in this design also [9].

Each physical core for this processor can simultaneously support 2 threads due to the Intel Hyper-Threading technology which allows one physical core to present two logical cores to the operating system. It is important to understand that this technology is designed to make full use of each core's performance, but it is not equivalent to running 8 cores. For the processor used in this thesis, there are 8 threads, but only 4 cores, so performance gains similar to an 8 core system should not be expected. This will be shown in Section 6.2 Thread Performance Comparisons. The two logical processors in the Intel Hyper-Threading design share most execution resources, and the desired benefit is to improve the efficiency of the instruction scheduling for that core. Given this capability, there are cases where small gains should be expected, such as bottlenecks from synchronizing shared data or instruction scheduling for applications that are already extremely efficient in their scheduling [8].

This processor's architecture is illustrated in Figure 22 below. The test system has 8GB of RAM and is running openSUSE 13.1 x86_64 Linux as the OS, which is the

latest and most up-to-date OS available from openSUSE. Multithreading and Hyper-Threading are supported in this version of openSUSE and both were enabled in the kernel and BIOS for the execution of these programs.

C++ 11 was used for all programming and the GCC 4.8.1 compiler was used which supports compilation of all C++11 features. The libstdc++-devel 4.8-2.1.2 library package was used which provides the necessary C++11 libraries, and these were the only libraries used in this project. QtCreator 2.8.1 was used as an Integrated Development Environment (IDE). All of the serial algorithms were developed previously and were converted to parallel algorithms for this project.

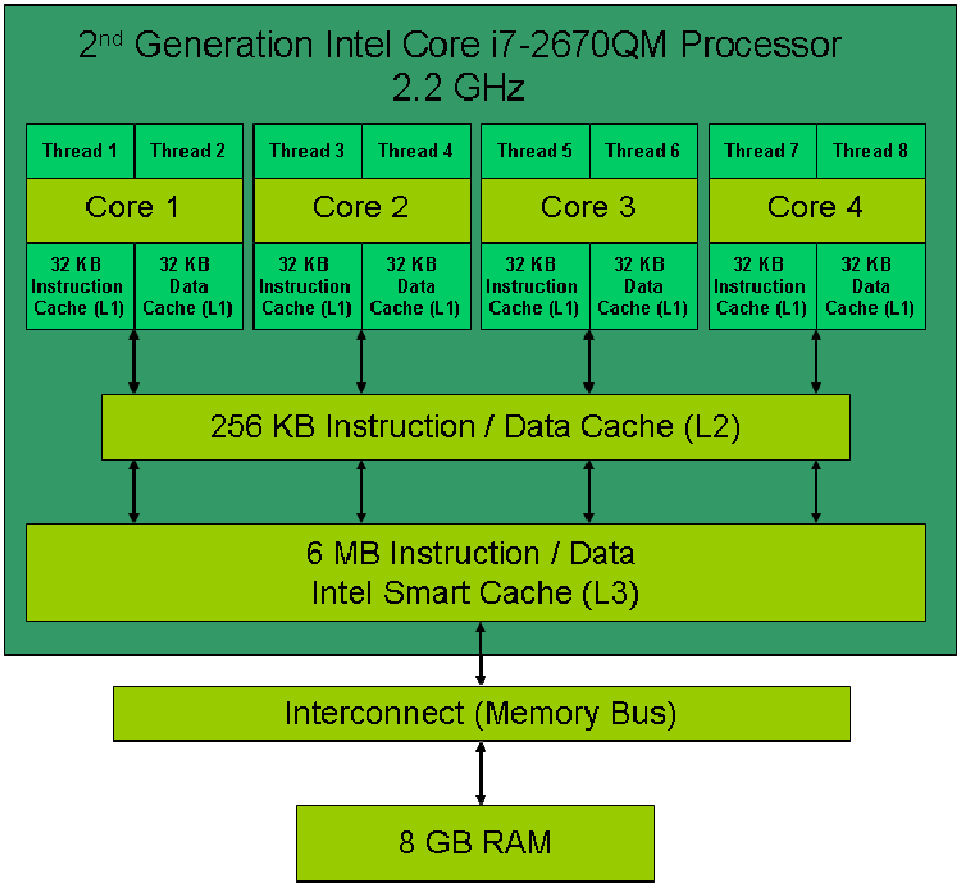

Figure 22 Test Architecture Diagram [10]

Debugging and program profiling were completed using several tools within Valgrind 3.9.0. These were Helgrind for multithread debugging, Memcheck for memory management debugging and profiling and Callgrind for function and execution time profiling.

For all of these programs, GCC compilation flags were used. The –std=c++11 flag was used as the compiler currently supports the old and new standards and will default to old standard without this flag. The –O3 optimization flag was used for timing tests to ensure that the compiler optimized the code by rearranging it as needed. The –g flag was used for debug runs with Valgrind since it provides more detailed output from the debugging tools.

Chapter 6: Experimental Results

6.1 Performance Profiling With Valgrind

Valgrind is a suite of tools that is commonly used by developers to characterize performance within a program and to identify issues associated with several programming errors. One Valgrind tool called Callgrind helps determine how much time a given function takes up in a program's execution or how many times a function is called during program execution [5]. This information can be helpful in identifying where parallel code can be implemented in a program since the areas that take the most time to run are potential candidates for large performance gains with parallel coding. Since all of the programs covered in this study spawn threads that run at high levels, function timing profiling was not necessary, but Callgrind outputs were still observed to confirm the programs behaved as expected. Kcachegrind is a GUI tool in Linux that displays Callgrind outputs in graphical form, making it much easier to visualize the results from the Callgrind runs. A sample Callgrind run from the ANN Planetary Lander problem, run with 4 threads, is shown in kcachegrind in Figure 23 below.

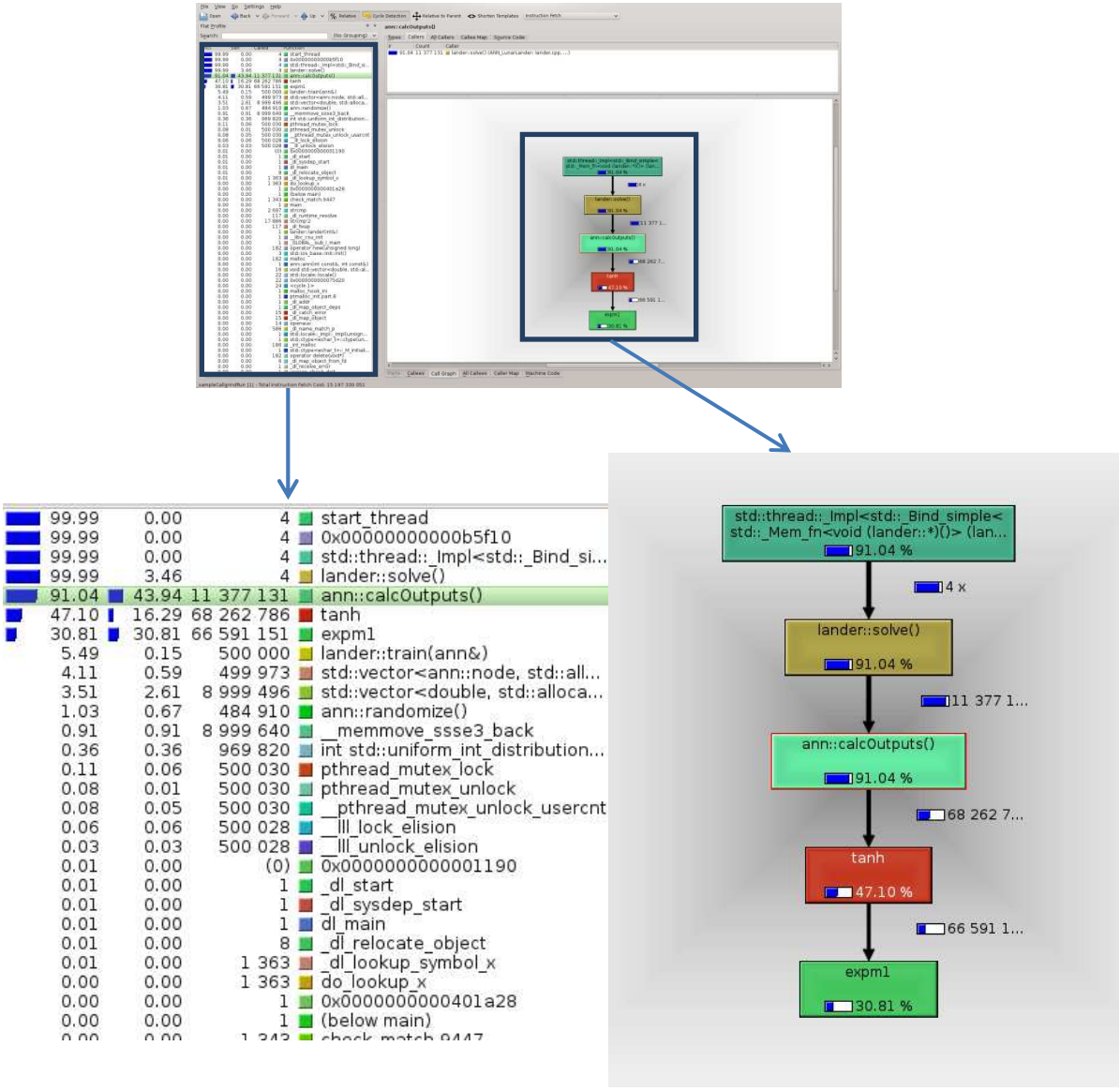

Figure 23 Sample Kcachegrind Run

Both of the zoomed callouts from Figure 24 show the same information displayed in two formats. The callout on the left shows all of the significant function calls. The top function call start_thread and the lander::solve() functions both show that 99.99% of the time running the program is used within thread calls. This is consistent with the workload in this problem since the lander::solve() function is a top-level function used

to spawn the threads and it is also expected that start_thread is the first call. Most of the lander::solve() execution time, 91%, is spent calculating the ANN outputs for burn and thrust with the function call ann::calcOutputs(). 47% of the time in ann::calcOutputs() is used calling the cmath tanh function which is obviously very time consuming. The rest of the time spent in the ann::calcOutputs() function is spent feeding the inputs through the ANN and calculating the resultant outputs. Callgrind and especially Kcachegrind make it much easier for the programmer to understand how time is being used in a program and can help identify areas where inefficiencies exist.

Memcheck is another tool within Valgrind which is used to identify memory issues such as improper initialization and leaks due to improper allocation and freeing of memory. It can also show memory usage statistics such as the total heap usage during a run [5]. Since shared pointers were used in place of dumb pointers when memory allocations were needed, it was expected that memory management issues would not exist, but Memcheck was used to ensure this was the case and to ensure that other non-pointer initializations were handled properly. A sample Memcheck run from the BFS Pathfinding problem, running with 4 threads, is shown below in Figure 24. This problem creates a shared pointer for each node that is created.

```
kniles@robot:~/Desktop/MultithreadThesis/ThesisCode/Pathfinding_BreadthFirstSearch/Pathfinding_BFS> r
==16339== Memcheck, a memory error detector
==16339== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==16339== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright info
==16339== Command: Pathfinding_BreadthFirstSearch
==16339==
Threads to run: 4

Working... Run 1
Threads: 4
cells searched: 7888 / 10000
cells to solution: 109
nodes created: 31552
total cost: 517

Time Elapsed(s): 9.24639
Average Time Elapsed(s): 9.24639
==16339==
==16339== HEAP SUMMARY:
==16339==     in use at exit: 0 bytes in 0 blocks
==16339==   total heap usage: 74,418 allocs, 74,418 frees, 4,209,297 bytes allocated
==16339==
==16339== All heap blocks were freed -- no leaks are possible
==16339==
==16339== For counts of detected and suppressed errors, rerun with: -v
==16339== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
kniles@robot:~/Desktop/MultithreadThesis/ThesisCode/Pathfinding_BreadthFirstSearch/Pathfinding_BFS>
```

Figure 24 Sample Valgrind Memcheck Run

Notice that there are no errors and that all of the allocated memory in the heap was freed. This ensures the programmer that no memory leaks exist. There is also a heap usage summary which shows how many total bytes of memory are allocated in the heap. This statistic does not show real-time heap usage, but another Valgrind tool called Massif is available if real-time sampling of the heap is needed [5]. The total heap usage for all of the cases in this study are small so further information was not gathered.

The last Valgrind tool that was used was Helgrind which is a multithreading tool that was used for debugging the multithreaded code. Helgrind identifies potential errors in pthreads parallel programs, and the C++11 thread libraries make use of pthreads at the core, so this tool was ideal for these cases and was used to identify deadlocks and race conditions. Race conditions are notoriously difficult to track down since they either occur rarely or do not pose obvious differences in the outcome of the program

execution. Deadlocks are usually obvious since they will lock up the program during execution. Several race conditions were found, mostly due to improper locking of read accesses and were easily corrected with Helgrind outputs. Helgrind also has many false positives when dealing with atomic variables since it currently sees them as regular variables. As a result, Helgrind expects locking mechanisms to be used for reads and writes to the atomic variables and produces errors as a result. Atomic variable reads and writes are guaranteed to be thread-safe in C++ 11 [11], so these warnings were disregarded. Helgrind can be configured to suppress specific outputs such as this one to ease troubleshooting for the programmer. A sample Helgrind output from the ANN Planetary Lander problem running with 2 threads is shown below in Figure 25.

A thread announcement is printed for the creation of each new thread. There is also a possible data race error shown which is labeled as a potential conflict by one thread reading a memory location while another thread is writing to it. This error message can indicate a race condition where a mutex was not properly included to restrict the access to that data. After the lines telling whether the conflict is related to a read or write access, the output tells whether or not a lock was held by either access. In this case, both of the accesses do not hold a lock. The next line below that shows a trace to help identify why this error is occurring, starting with the most recent function call and working back to the highest level function call. Notice the right side of the first line of the trace indicates that in the lander::solve() function, an error is observed relating to the atomic_base header file which is a C++ 11 header file for atomic variables. This same file is referenced by both accesses to the data

and indicates that Helgrind thinks that a data race is occurring with an atomic variable.

```
~> helgrind ANN_PlanetaryLander
==15937== Helgrind, a thread error detector
==15937== Copyright (C) 2007-2013, and GNU GPL'd, by OpenWorks LLP et al.
==15937== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright info
==15937== Command: ANN_PlanetaryLander
==15937==
==15937== ---Thread-Announcement------------------------------------------
==15937==
==15937== Thread #3 was created
==15937==    at 0x595B8CE: clone (in /lib64/libc-2.18.so)
==15937==    by 0x565D2C9: do_clone.constprop.3 (in /lib64/libpthread-2.18.so)
==15937==    by 0x565E782: pthread_create@@GLIBC_2.2.5 (in /lib64/libpthread-2.18.so)
==15937==    by 0x4C2C1C7: pthread_create_WRK (hg_intercepts.c:269)
==15937==    by 0x4C2D01A: pthread_create@* (hg_intercepts.c:300)
==15937==    by 0x4EEA17E: std::thread::_M_start_thread(std::shared_ptr<std::thread::_Imp
==15937==    by 0x40160E: main (thread:135)
==15937==
==15937== ---Thread-Announcement------------------------------------------
==15937==
==15937== Thread #2 was created
==15937==    at 0x595B8CE: clone (in /lib64/libc-2.18.so)
==15937==    by 0x565D2C9: do_clone.constprop.3 (in /lib64/libpthread-2.18.so)
==15937==    by 0x565E782: pthread_create@@GLIBC_2.2.5 (in /lib64/libpthread-2.18.so)
==15937==    by 0x4C2C1C7: pthread_create_WRK (hg_intercepts.c:269)
==15937==    by 0x4C2D01A: pthread_create@* (hg_intercepts.c:300)
==15937==    by 0x4EEA17E: std::thread::_M_start_thread(std::shared_ptr<std::thread::_Imp
==15937==    by 0x40160E: main (thread:135)
==15937==
==15937== ----------------------------------------------------------------
==15937==
==15937== Possible data race during write of size 1 at 0xFFEFFF938 by thread #3
==15937== Locks held: none
==15937==    at 0x4038EA: lander::solve() (atomic_base.h:474)
==15937==    by 0x4EE9F2F: ??? (in /usr/lib64/libstdc++.so.6.0.18)
==15937==    by 0x4C2C34D: mythread_wrapper (hg_intercepts.c:233)
==15937==    by 0x565E0DA: start_thread (in /lib64/libpthread-2.18.so)
==15937==    by 0x595B90C: clone (in /lib64/libc-2.18.so)
==15937==
==15937== This conflicts with a previous read of size 1 by thread #2
==15937== Locks held: none
==15937==    at 0x403608: lander::solve() (atomic_base.h:496)
==15937==    by 0x4EE9F2F: ??? (in /usr/lib64/libstdc++.so.6.0.18)
==15937==    by 0x4C2C34D: mythread_wrapper (hg_intercepts.c:233)
==15937==    by 0x565E0DA: start_thread (in /lib64/libpthread-2.18.so)
==15937==    by 0x595B90C: clone (in /lib64/libc-2.18.so)
==15937==
Threads to Run: 1 121.62 Average Time Elapsed(s): 121.62

==15937==
==15937== For counts of detected and suppressed errors, rerun with: -v
==15937== Use --history-level=approx or =none to gain increased speed, at
==15937== the cost of reduced accuracy of conflicting-access information
==15937== ERROR SUMMARY: 2 errors from 1 contexts (suppressed: 71169 from 16)
```

Figure 25 Sample Valgrind Helgrind Run

As stated previously, Helgrind does not currently deal well with atomic variables in C++ 11 since they do not use the mutex locking mechanism that is typically expected to avoid race conditions. The C++11 standards have ensured that the memory model within C++11 handles atomic variables in a way that will not allow race conditions [11]. In the error summary, there are 2 errors shown from 1 context which indicates Helgrind has seen 2 errors related to the one and only atomic context that was printed. Notice also that many errors are suppressed by Helgrind by default. Although Helgrind cannot currently handle all cases related to multithreading code, especially those where locking is used more than necessary, it is still useful in identifying potential issues. One must pay close attention to errors and understand which ones are potential hazards and which ones are related to shortcomings in the error detection.

6.2 Thread Performance Comparisons

Each program was created to run with a user-defined number of threads and was designed for parallel execution. Multiple runs were completed for each problem to benchmark several statistics. Average execution time, speedup, efficiency, total memory usage, average best fitness and memory frees and allocations are all compared between runs. For cases where the fitness is determined by the random number generation, such as the Hill Climbing and Deciphering Encryption problems, set iteration cutoffs were used instead of fitness cutoffs to ensure that runs were fair between varying thread numbers in terms of execution time. For Connect 4, runs were completed for 1 to 7 threads and a constant time was then used for values above 7 since the design for that problem is bound by the number of columns, which

is 7. For all other problems, runs were completed for 1 to 8 threads in 2^n increments. Interpolation was applied in Matlab between these values to create smoother curves for the analysis plots. Figure 26 below shows the average execution time for varying threads on all of the problems. The average execution time was calculated over 10 runs for each thread value. On all of the programs, the execution times drop from 1 to 4 threads and then taper off for thread values above 4. The Connect 4 average execution time has a curve that is not smooth due to the implementation that was used. Because the number of columns could not be evenly distributed for all thread values, some runs have asymmetric execution times that are directly associated with the asymmetric column distributions. As described in Section 3.3 Game Playing: Connect 4, the chosen implementation distributes the evaluation of AI moves by columns as shown below in Table 5. The distribution of work is the number of columns, which is 7 for Connect 4, divided by the number of threads used, and the remainder is then handed to out to as many threads as needed to take care of the remaining columns:

7 / 2 = 3 r.1 7 / 4 = 1 r.3

For a run with 2 threads, the first thread evaluates the minimax algorithm for columns 1-3 and 7. The second thread evaluates the algorithm for columns 4-6. An asymmetry occurs when a remainder must be distributed among the threads, and this additional workload becomes a bottleneck to the execution time. This implementation was chosen since it made the parallel distribution of the workload much simpler to program than other methods.

Table 5 Connect 4 Multithread Program Work Distribution

Total Threads	Thread Number	Columns Per Thread	Remainder	Columns Covered
1	1	7	0	7
2	1	3	2	1 , 2, 3, 7
	2			4, 5, 6
4	1	1	3	1, 5
	2			2, 6
	3			3, 7
	4			4, 7
7	1 - 7	1	0	1

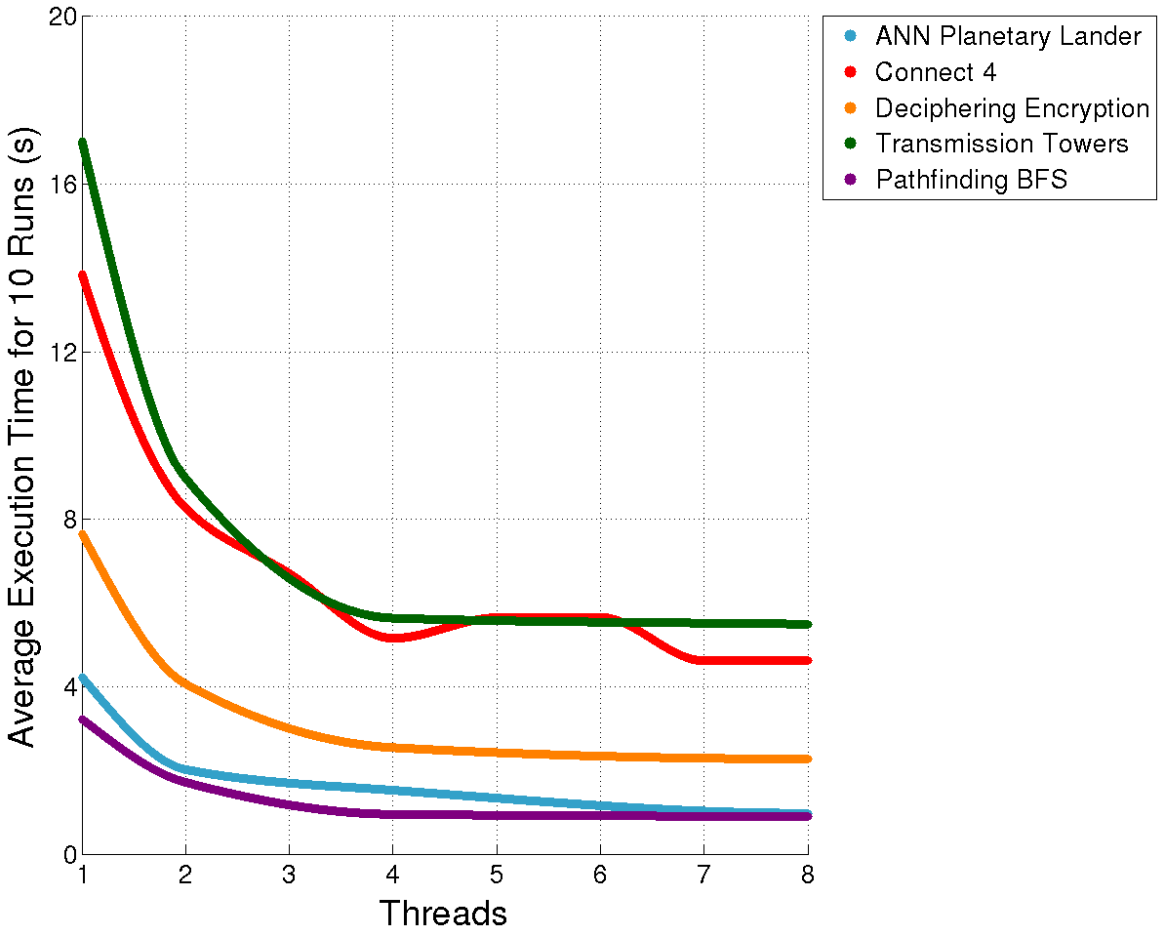

Figure 26 Average Execution Time Using 1 to 8 Threads for All Programs

The average execution time is an important measure when looking at multithreading, but the speedup and efficiency values are much more useful when trying to determine performance gains. Speedup is the serial execution time divided by the parallel execution time [1]. The highest theoretical speedup for a run is equal to the number of threads where a 2 thread run shows a speedup of 2. The speedup values for all of the programs with various thread counts are shown below in Figure 27.

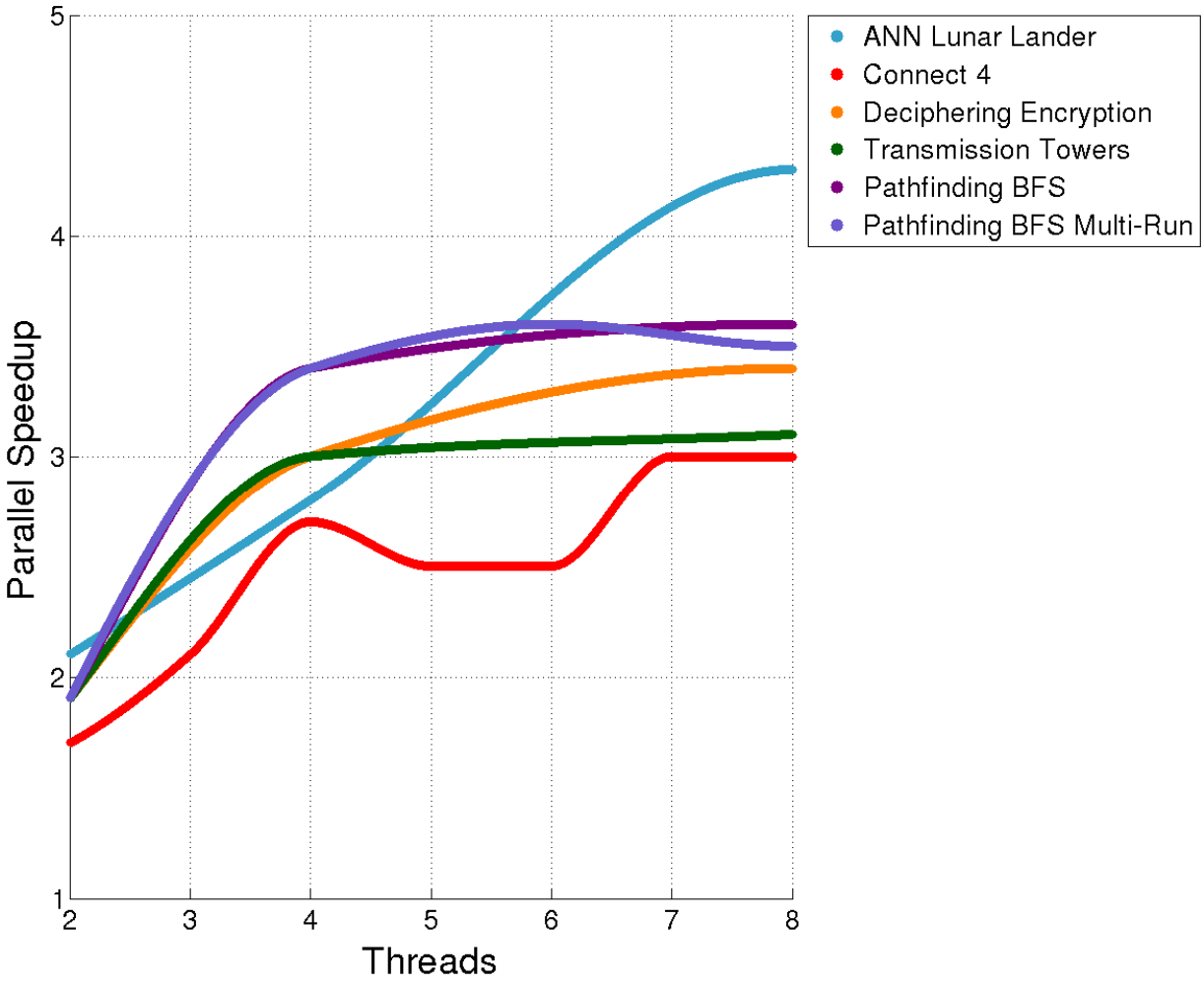

Figure 27 Speedup Using 2 to 8 Threads for All Programs

This speedup plot helps in deriving many useful conclusions. Note that there is an additional set of runs called Pathfinding BFS Multi-Run. These runs were added to

compare multithreaded execution at the OS process level. For varying thread numbers, runs of the Pathfinding BFS program were executed as separate processes. The values from these runs indicate running multiple single-threaded processes of the program at the same time as opposed to running several threads within the program. Using this method, all of the speedup values are based solely on the hardware and OS since internal program locking and synchronization between multiple threads will not exist. By comparing these runs to the multithreaded runs inside the program, one can see how much the speedup values are affected by the hardware and OS as opposed to the internal locking mechanisms associated with mutex and atomic variables.

Note that the external multithread runs and internal runs for the Pathfinding problem are almost identical in speedup values and that the speedup values for runs with 2 and 3 threads yield speedup values of approximately 2 and 3. The speedup value stays close to 3.5 for runs where 4 or more threads are used. This outcome shows that the speedup values are limited by the hardware and OS and almost unaffected by the locking and synchronization for this particular problem. Remember that the tested processor only has 4 cores and these results are consistent with 4 cores once you take into account the system-level processing that is taking place within the OS while the runs were being tested.

In the Pathfinding BFS curve, it appears that mild speedup increases occur from 4 to 8 threads, but these gains are not indicative of the gains one would expect with an 8 core system. These limited gains show that the Hyper-Threading technology does not improve the performance to the level of an 8 core system, but there are still modest

gains when using the logical processors. The apparent speedup increase due to Hyper-Threading is shown below in Table 6. In all cases, the Hyper-Threading improves the results, but the improvements are limited due to the fact that there are only 4 physical cores on the test system. Another explanation for the modest gains is that thread contention increases as the number of threads used increases and they all begin competing for access to the shared variables. This contention grows quickly in all of these programs since frequent synchronization operations must be used. Thread contention would yield diminished results for 8 physical cores as well.

Table 6 Apparent Hyper-Threading Speedup for All Programs

Problem	Threads	Cores	Apparent Hyper-Threading Speedup Increase %
ANN Planetary Lander	8	4	36.06
Connect 4	8	4	10.46
Deciphering Encryption	8	4	10.64
Transmission Towers	8	4	2.58
Pathfinding BFS	8	4	4.80
Pathfinding Multi-Run	8	4	3.21

The ANN Planetary Lander results are misleading because of the random elements used in the program. Using an iteration limit, it was expected that the runs would be consistent in execution time, but closer observation showed that there is no way to mitigate the impact of the randomness on the execution time since they are interconnected. For example, one iteration is equivalent to one landing, but some runs land much faster than others due to the varying of ANN weight values. Some weight values cause the lander to execute little or no burn or thrust values, causing the lander to crash quickly due to acceleration from gravity. Other weight values

cause the lander to run burn or thrust values that are too high which cause the lander to overcome gravity, increase altitude and then crash later after the fuel runs out and the gravity overtakes the vehicle's upward speed. Combining these two cases will cause large fluctuations in the execution times between iterations, especially since they happen with completely random frequency. Using a Gaussian random number generator would probably yield better results for execution times, but it is not ideal for the search which should be uniformly random. Regardless of the randomness, it is easy to see that the ANN Planetary Lander speedup is increased as threads are increased, but the noise involved in the random number generation makes it difficult to pinpoint how reliable these speedup values are.

Since the two Pathfinding curves match closely, they are a good baseline of comparison for the other programs. Notice that all of the other programs, excluding the ANN Planetary Lander, have similar speedup curves to the Pathfinding curves, but they have decreased speedup values as the number of threads increases. These differences can be attributed to inefficiencies related to locking, synchronization or design. For example, the Connect 4 program has much worse speedup values for all thread numbers, but this is expected since the program was designed to for simplicity more so than efficiency. Even though the Connect 4 program is not as efficient as the other programs, it still yields good results by increasing speedup to almost 3 using 7 threads. This speedup is near the maximum speedup value of 3.5 observed by the Pathfinding program.

As a side note, the original implementation of the Pathfinding program put the consumer threads to sleep to simulate work for determining the terrain costs for a

move. This was eventually changed to arbitrary calculations that keep the threads busy since it was found the speedup results were misleading with the use of thread sleeping. When a sleep is used, a thread is not busy and can be used for other work, which is quite desirable in many cases. In this case, it made the speedup results look as though they exceeded limits imposed by the processor architecture. It was found that using sleep calls, all 4 processor cores were not fully used until the program was run with 32 threads. Arbitrary calculations were added to ensure that comparisons between programs were uniform.

The efficiency value divides the speedup by the number of threads used on a run to give a value between 0 and 1 [1]. A value of 1 indicates 100% efficiency and is the highest theoretical efficiency that can be obtained. Efficiency values for all programs are shown below in Figure 28. Since efficiency values are calculated using speedup values, the results are based on the same data as the previous speedup plots but displayed in a different manner.

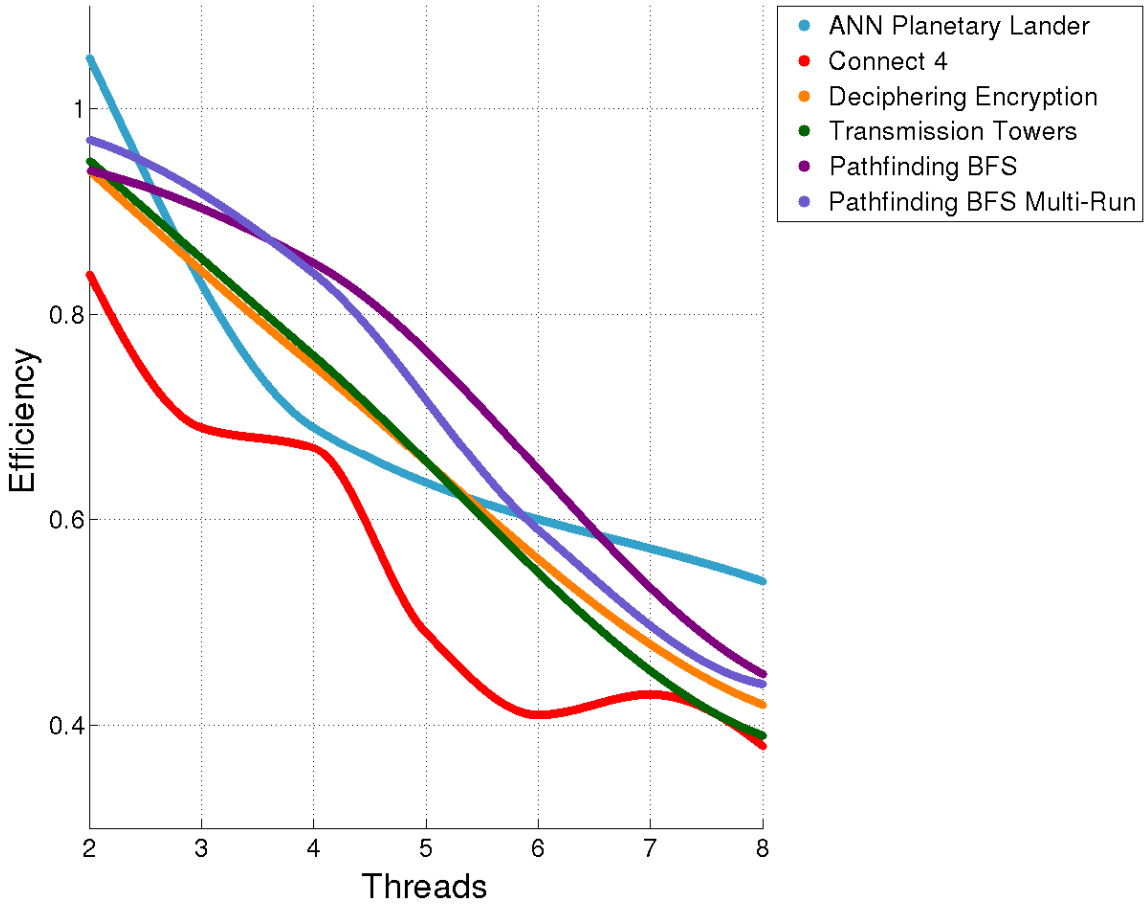

Figure 28 Efficiency Using 2 to 8 Threads for All Programs

The speedup and efficiency values for the two Pathfinding runs are shown below in Table 7 to give a clearer view of how close the different methods of execution were.

Table 7 Speedup and Efficiency Comparison Between Program and OS

Problem	Threads	Parallel Speedup	Parallel Efficiency
Pathfinding BFS	2	1.9	0.94
	4	3.4	0.85
	8	3.6	0.45
Pathfinding Multi-Run	2	1.9	0.97
	4	3.4	0.84
	8	3.5	0.44

Plots of total memory allocated to heap space for thread runs from 1 to 128 are shown below in Figure 29. Notice that the memory increases as the threads increase but by a very small amount. These plots show that there are limited additional memory resources used in multithreading for all of the cases covered in this study. Additional analysis plots and statistics are available in Appendix C – Additional Analysis Plots and Statistics.

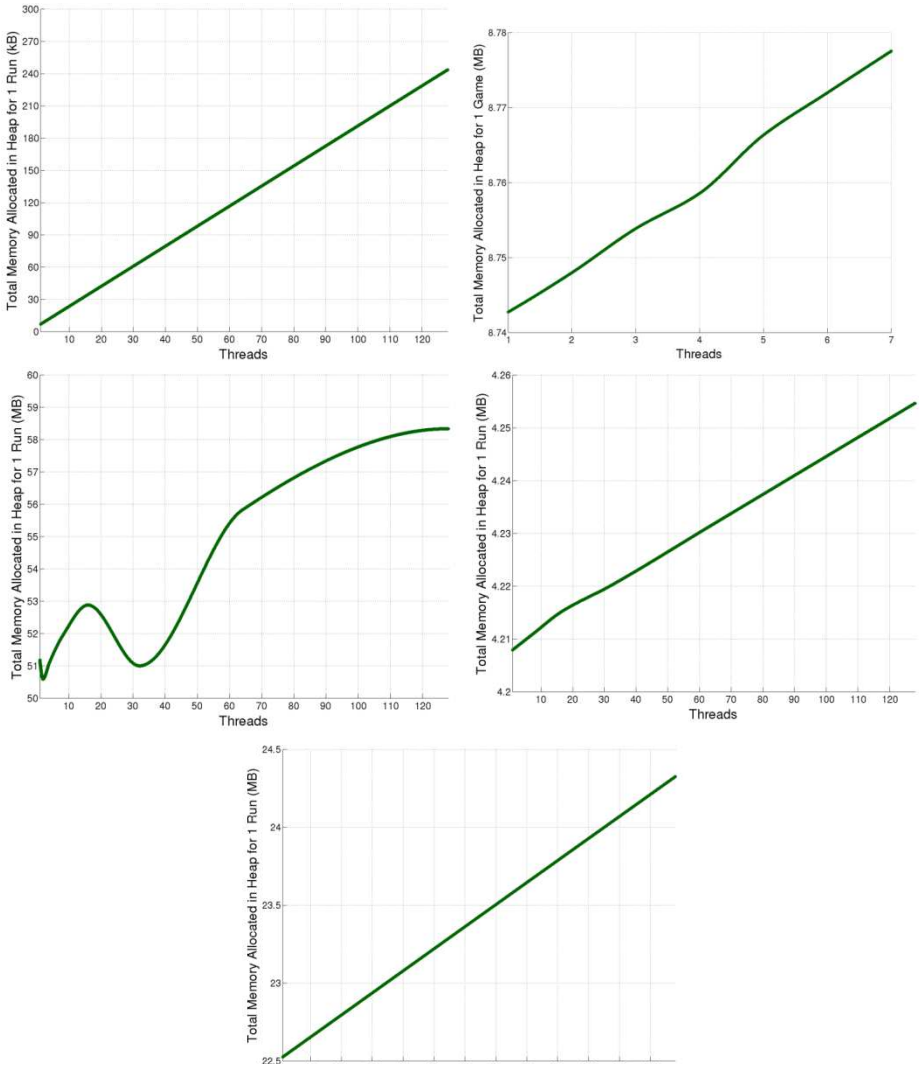

Figure 29 Total Memory Using 1 to 128 Threads for All Programs

Chapter 7: Conclusions

The majority of time investment for this research was devoted to learning how the new C++11 multithread libraries and the atomic memory model work. Several experiments were then run to understand how to apply these concepts and create usable programs. Once a programmer has a firm grasp of how to make use of the tools, many of the concepts covered in this research can be implemented with modest programming efforts. The most important aspect to consider when creating a multithreaded program is the design. A simple design can oftentimes create large performance gains without requiring too much refactoring as was demonstrated in Section 4. The programs covered here have a small scope and there can be potential issues when scaling the same concepts to much larger projects where threads may need to be shared across multiple classes and functions. Some of these issues were seen in these programs when threads were shared across many functions, but altering the scope of variables was easily accomplished to make use of the shared nature of class objects.

Taking the cost and benefits of multithread implementations into account, the benefits gained from implementing parallel code were well worth the effort for all of the cases covered in this research. As shown in Section 6.2, all of the programs provided high speedup values compared to the boundaries imposed by the processor architecture and OS.

I would recommend using thread-level parallelism to decrease the execution time of all of these and similar algorithmic programs since they rely heavily on repetitive iteration, which makes designing parallel programs straightforward based on the

iterative tasks as demonstrated in Section 4. The algorithms can also be restructured in a more generic way to be reused many times, with each use building greater value on the implementation.

References

[1] P. S. Pacheco, *An Introduction to Parallel Programming*, Burlington: Elsevier, 2011.

[2] A. E. Eiben, J. E. Smith, *Introduction to Evolutionary Computing,* Berlin, Germany: Springer-Verlag, 2003.

[3] S. Russell, P. Norvig, *Artificial Intelligence A Modern Approach,* 3rd ed. New Jersey: Pearson, 2010.

[4] S. Haykin, *Neural Networks and Learning Machines*, 3rd ed. New Jersey: Pearson, 2009.

[5] Valgrind Developers (2013, Oct.). Valgrind Documentation. [Online]. Available: http://valgrind.org/docs/manual/manual.html

[6] J. L. Hennessy, D. A. Patterson, *Computer Architecture A Quantitative Approach*, 4th ed. Oxford: Elsevier, 2007.

[7] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips (2008, May). GPU Computing: Graphics Processing Unit – powerful, programmable, and highly parallel – are increasingly targeting general-purpose computing applications [Online]. Available: http://cs.utsa.edu/~qitian/seminar/Spring11/03_04_11/GPU.pdf

[8] A. Valles (2009, Nov.). Performance Insights to Intel Hyper-Threading Technology [Online]. Available: https://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology

[9] T. Tian and C. Shih (2012, Mar.). Software Techniques for Shared-Cache Multi-Core Systems [Online]. Available: https://software.intel.com/en-us/articles/software-techniques-for-shared-cache-multi-core-systems

[10] Intel Corp. (2013, Jun.). 2nd Generation Intel Core Processor Family Desktop, Intel Pentium Processor Family Desktop, and Intel Celeron Processor Family Desktop Datasheet [Online]. Available: http://www.intel.com/content/www/us/en/processors/core/2nd-gen-core-desktop-vol-1-datasheet.html

[11] B. Stroustrup, *The C++ Programming Language*, 4th ed. New Jersey: Pearson, 2014.

Appendix A – Glossary of Terms and Acronyms

Alpha-Beta Pruning – search method used to minimize the number of tree nodes that must be expanded and searched by pruning away unnecessary branches

Application Programming Interface (API) – used in programming to specify how software components should interact with each other

Arithmetic Logic Unit (ALU) – a digital circuit that performs integer arithmetic and logical operations

Artificial Intelligence (AI) – the field of science dedicated to understanding and building intelligent entities

Artificial Neural Network (ANN) – implementation based on the neuroscience hypothesis that mental activity consists primarily of electrochemical activity in networks of brain cells called neurons

Breadth First Search (BFS) – complete search that expands the shallowest nodes first and is optimal for unit step costs, but has exponential space complexity

Concurrent Processing – a program where multiple tasks can be in progress at any instant

Central Processing Unit (CPU) – traditional single processor, also referred to as a core on a multicore system

Crossover – the process of selecting portions of each parent to create a successor in GA's

Efficiency – Speedup divided by the number of threads used to run a task

First In First Out (FIFO) Queue – a queue where the first item pushed is the first to be popped and an item is pushed to the back of the queue and popped from the front of the queue much like a line at a store

Fitness – a measured value of how good a given implementation is, used as reinforcement for learning

Genetic Algorithm (GA) – Search where successor states are generated by combining two parent states rather than by modifying a single state

General Purpose Graphics Processing Unit (GPGPU) – GPU ALUs on a single GPU core use SIMD parallelism, but the cores of current GPUs can execute independent instruction streams, behaving like an MIMD system, exhibiting hybrid characteristics

Heuristic – function that estimates the cost of an implementation or move
Hill climbing – search that continually moves in the direction of increasing value (uphill)

Hyper-Threading – Intel technology that uses processor resources more efficiently, enabling multiple threads to run on each core by allowing one physical core to present two logical cores to the operating system

Integrated Development Environment (IDE) – a software application that provides comprehensive facilities to programmers for software development

Learning – improvement of performance on future tasks after making observations about an environment

Minimax – a recursive algorithm that selects a min or max value dependent on the depth of the tree node it is evaluating, used in gameplay

Multicore – multiple cores on one chip where each core is a relatively simple, complete processor

Multiple instruction streams, multiple data streams (MIMD) – each processor has its own instructions and operates on its own data, exploits thread-level parallelism

Multiprocessor – more than one multicore processor in a system, typically used in servers

Multithread – the use of multiple threads of control within a single process

Mutation – the process of selecting an element within a member and changing its value in GA's

Parallel Processing – a program where multiple tasks cooperate closely to solve a problem

Parallel Programming – programming strategy that makes use of multiple cores

Population – Collection of individuals or solutions used in GA's

Resource Acquisition Is Initialization (RAII) – technique for managing resources with local objects by calling a destructor independently of whether a function is exited normally or because of an exception

Reinforcement Learning – learning where reinforcements such as rewards and punishments are used to drive the learning process

Searching – the process of looking for a sequence of actions that reaches a goal

Simulated Annealing – search that combines hill climbing with a random walk by using a heating / cooling schedules and a temperature value with some probabilistic acceptance criteria

Single instruction stream, multiple data streams (SIMD) – the same instruction is executed by multiple processors using different data streams

Speedup – the execution time of a serial task divided by the execution time of a multithreaded task

Standard Template Library (STL) – a framework of algorithms and containers used in C++ since the 1998 standard

Supervised Learning – learning based on a training set of input-output pairs

Thread – a thread of control, which is a sequence of statements in a program

Thread-Level Parallelism – parallelism through the simultaneous execution of multiple threads, providing coarse-grained control over parallel execution

Appendix B – Pseudocode

Hill Climbing with Simulated Annealing: Transmission Tower Placement

```
Main{
  Create hillclimb class object ;
  Spawn_threads( hillclimb::solve() );
  Join_threads;
}

Hillclimb::solve(){
  While (count < count_limit){
  Count++;
  Local_count++;
  If ( local_count > local_count_limit ) local_temperature = start_temperature; // heating for annealing per
  thread
  Mutate_random_transmitter_x_or_y();
  Fitness = getFit();
  If ( random_unit_draw ) < e^ ( ( fitness – best_fitness ) / temperature ){
    Best_fitness = fitness;  // atomic variable update, no lock needed
    X_transmitter_lock.lock(); // unique locks used
    Best_x_transmitter_positions = mutated_x_transmitter_positions
    X_transmitter_lock.unlock();
    Y_transmitter_lock.lock();
    Best_y_transmitter_positions = mutated_y_transmitter_positions;
    Y_trasmitter_lock.unlock();
  }
  Else{
    X_transmitter_lock.lock();
    Mutated_x_transmitter_positions = best_x_transmitter_positions;
    X_transmitter_lock.unlock();
    Y_transmitter_lock.lock();
    Mutated_y_transmitter_positions = best_y_transmitter_positions;
    Y_trasnmitter_lock.unlock();
  }
  Local_temperature = local_temperature / cooldown; // cooldown schedule determined by cooldown
  variable
}
```

Pathfinding: Breadth First Search

```
Main{
  Create frontier class object ;
  Spawn_producer_thread( frontier::produce() );
  Spawn_consumer_threads( frontier::consume() );
  Join_consumer_threads;
  Join_producer_thread;
}

Frontier::produce(){
  While( done == false){
    Wait_for_produce_condition_variable_notify( producer_lock ); // acquire unique_lock producer_lock
    when
                                                                  // notified
    Parent = producer_queue.front(); // parent is a shared pointer used as a holder
    Prodcuer_queue.pop();
    Producer_lock.unlock();
    Generate_child_states_from_parent;
    Consumer_lock.lock();
```

```
    Consumer_queue.push( child );
    Notify_consumers_with_notify_condition_variable;
    Consumer_lock.unlock();
  }
}

Frontier::consume(){
  Wait_for_consume_condition_variable_notify( consumer_lock);
  Child = consumer_queue.front(); // child is a shared pointer used as a holder
  Consumer_queue.pop();
  Consumer_lock.unlock();
  Apply_child_action_and_apply_move_cost;
  Traveled_lock.lock();
  Update_traveled_location_map; // prevents duplicate moves from being applied, saves time
  Traveled_lock.unlock();
  If ( goal_has_been_reached ) done = true;
  While( child_operation != next_operation) thread_sleep ( 1 nanosecond); //poll for status, this is used to
                                                                          // synchronize the moves as a
                                                                          // serial execution would do
  Producer_lock.lock();
  Producer_queue.push( child );
  Notify_producer_with_notify_condition_variable;
  Producer_lock.unlock();
}
```

Game Playing: Connect 4

```
Main{
  Create connect4 class object ;
  Connect4::solve();
}

Connect4::solve(){
  While( done == false){
    AlternateTurn();
    GetMove();
    DoMove();
    ShowBoard();
    CheckForDraw();
    If ( draw == true ) done = true;
    TestForWin();
    If ( win == true ) done = true;
    If ( done == true ) playAgain(); // ask player if he / she would like to play again, if yes, reset game and
    set
                                     // done back to false
  }
}

Connect4::getMove(){
  If ( it_is_human_move) get_and_execute_human_move;
  Else{
    Spawn_threads( connect4::threadMinimax() ); // get AI move
    Join_threads;
  }
}

Connect4::threadMinimax(){
  Best_fitness = 0;
  Update_best_lock.lock(); // unique_lock
```

```
  Local_game_state = game_state; // set each thread's game states to those of current game state
  Update_best_lock.unlock();
  Assign_search_branches_evenly_to_each_thread;
  Local_fitness = Execute_search_for_my_branches; // gather heuristics, apply minimax, apply alpha-beta
                                                  // pruning, and get a best move and best
                                                  fitness
                                                  // for this thread's search
  If ( local_fitness > best_fitness ){
    Best_fitness = local_fitness; // atomic so no lock needed
    Update_best_lock.lock();
    Game_state_score = local_score; // synchronize calculated best score found by this thread to the
                                    // game state
    Game_state_next_move = local_next_move; // synchronize the best next move found by this thread to
                                            // game state
    Update_best_lock.unlock();
  }
}
```

Genetic Algorithm: Deciphering encryption

```
Main{
  Create geneticAlgorithm class object ;
  Spawn_threads( geneticAlgorithm::solve() );
  Join_threads;
}

geneticAlgorithm::solve(){
  While (count < count_limit){
    Count++;
    runTournament();
    if ( keygen_threshold_reached ){
      replace_loser_keys_with_random_keys();
      get_new_key_fitness();
      fitness_update_lock.lock(); //unique_lock
      if ( new_key_fitness < best_individual_fitness ) best_individual_in_population = new_key_individual;
      fitness_update_lock.unlock();
    }
    Else if ( crossover_threshold_reached ){
      replace_loser_keys_with_crossover_keys(); // crossover performed by combining two random
      parents
                                                // from population
      get_new_key_fitness();
      fitness_update_lock.lock();
      if ( new_key_fitness < best_individual_fitness ) best_individual_in_population = new_key_individual;
      fitness_update_lock.unlock();
    }
    if ( mutate_threshold_reached ){
      mutate_key_of_random_individual();
      get_new_key_fitness();
      if ( mutated_key_fitness < original_key_fitness ){
      fitness_update_lock.lock();
      update_individual_key;
      if ( new_key_fitness < best_individual_fitness ) best_individual_in_population = new_key_individual;
      fitness_update_lock.unlock();
      }
    }
  }
}
```

Artificial Neural Network: Planetary Lander

```
Main{
  Create lander class object ;
  Spawn_threads( lander::solve() );
  Join_threads;
}

lander::solve(){
  While (count < count_limit){
    Count++;
    setANN(); // set initial values for landing attempt
    while ( lander_has_not_landed ){
      updateLander(); // update over and over until landing or crash has been completed
      testLanding(); // test to see if landing or crash has occurred
    }
    ANN_lock.lock(); //unique_lock
    Calculate_my_thread_ANNFitness(); // must be locked since the fitness value can also be written
                                      // elsewhere
    ANN_lock.unlock();
    train();
  }
}

Lander::train(){
  If ( my_thread_ANN_fitness >= best_ANN_fitness ){
    Best_ANN_lock.lock(); // unique_lock
    Best_ANN_nodes = my_thread_ANN_nodes;
    Best_ANN_fitness = my_thread_ANN_fitness;
    Best_ANN_lock.unlock();
  }
    Else if ( reset_criteria_met ){  // we are stuck, random reset everything
    ANN_lock.lock();
    ResetANNWeights();
    all_other_thread_ANN = my_thread_ANN;
    ANN_lock.unlock();
    Best_ANN_lock.lock();
    Best_ANN = my_thread_ANN;
    Best_ANN_fitnes = -100000; // reset best fitness
    Best_ANN_lock.unlock();
  }
  Else{
    ANN_lock.lock();
    My_thread_ANN_nodes = Best_ANN_nodes;
    ANN_lock.unlock();
  }
  My_thread_ANN_randomize(); // update random weight with random value;
  ANN_lock.lock();
  My_thread_ANN_fitness = 0; // reset fitness to setup for new landing attempt
}
```

Appendix C – Additional Analysis Plots and Statistics

Table 8 Speedup and Efficiency Stats for All Programs Except Pathfinding*

Problem	Threads	Parallel Speedup	Parallel Efficiency
ANN Planetary Lander	2	2.1	1.05
	4	2.8	0.69
	8	4.3	0.54
	16	4.1	0.25
	32	4.4	0.14
	64	4.6	0.07
	128	4.4	0.03
Connect 4	2	1.7	0.84
	3	2.1	0.69
	4	2.7	0.67
	5	2.5	0.49
	6	2.5	0.41
	7	3.0	0.43
Deciphering Encryption	2	1.9	0.94
	4	3.0	0.75
	8	3.4	0.42
	16	3.4	0.21
	32	3.4	0.10
	64	3.3	0.05
	128	3.3	0.03
Transmission Towers	2	1.9	0.95
	4	3.0	0.76
	8	3.1	0.39
	16	3.3	0.20
	32	3.3	0.10
	64	3.3	0.05
	128	3.2	0.03

*For Pathfinding Speedup and Efficiency Stats, see Table 7.

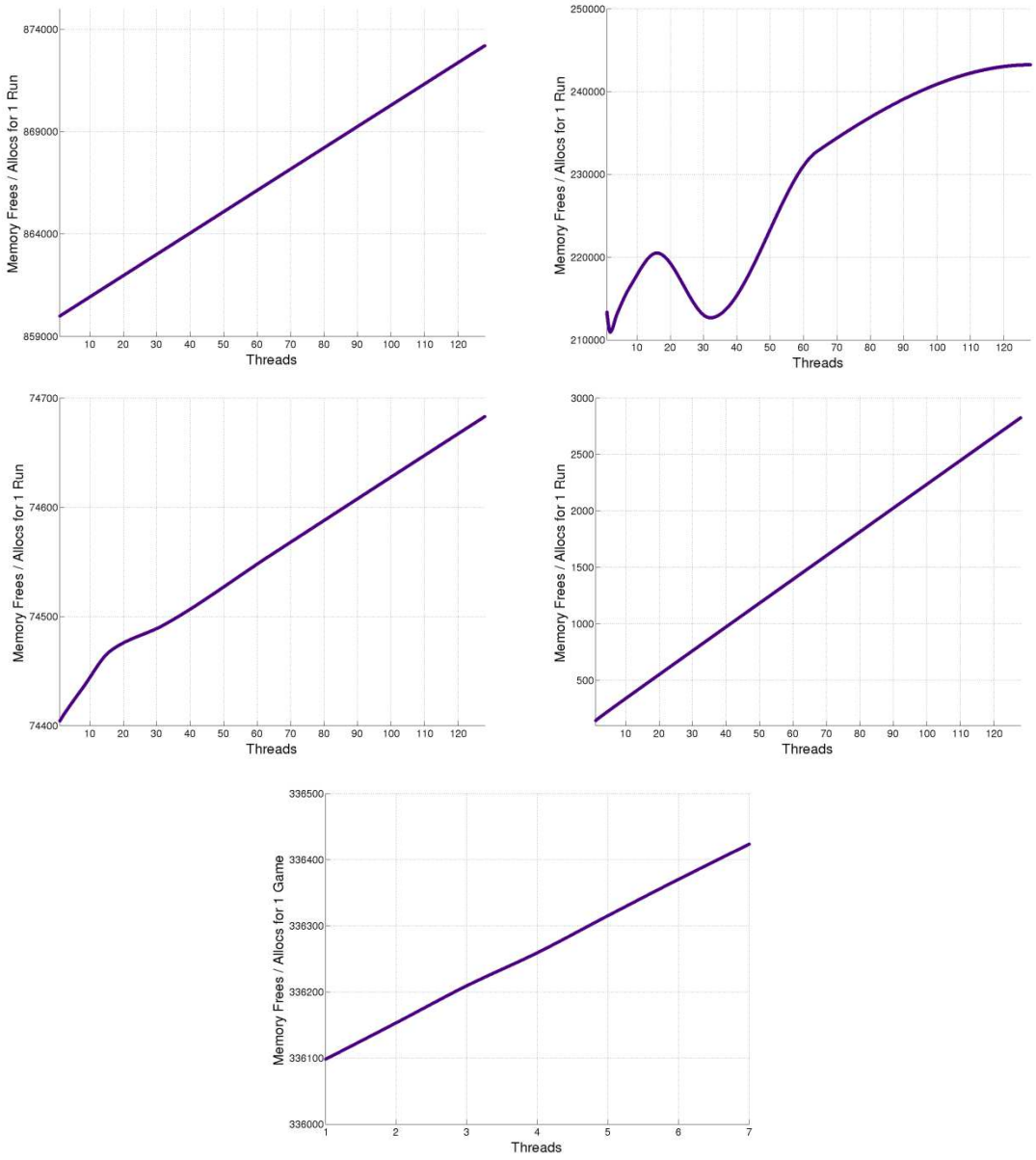

Figure 30 Memory Allocs / Frees Using 1 to 128 Threads for All Programs

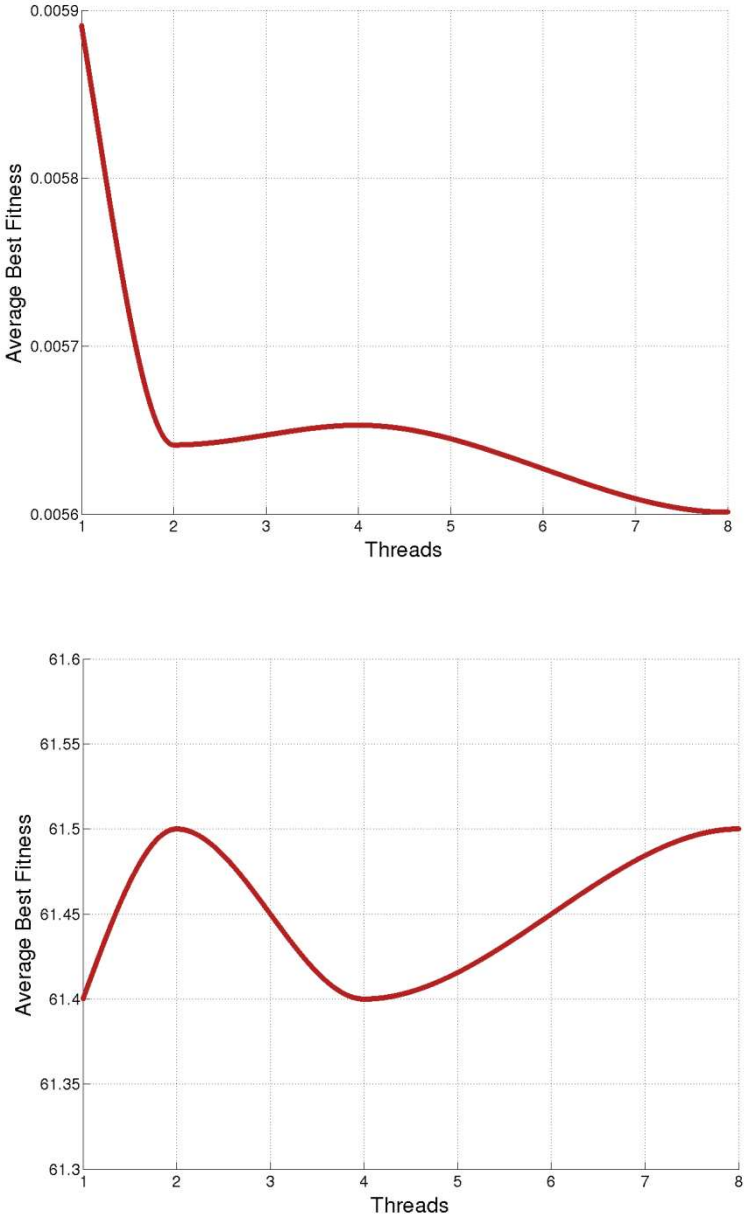

Figure 31 Average Best Fitness for Deciphering Encryption (Top) and Transmission Tower Placement (Bottom)