DISTRIBUTED COMPUTING APPROACHES TO PATHFINDING PROBLEMS

by

Robert Vital Myers

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science

Middle Tennessee State University
August 2016

Thesis Committee:

Dr. Joshua L. Phillips, Chair

Dr. Yi Gu

Dr. Salvador Barbosa

ACKNOWLEDGEMENTS

Thank you to all the people that assisted me with advice and encouragement. Many thanks to my advisor, Dr. Joshua Phillips, who has provided much insightful support to the project. Thanks also to my other committee members, Dr. Yi Gu and Dr. Salvador Barbosa for their input and suggestions throughout the project. It has been a privilege to work closely with this group who has provided their valuable time in giving constructive criticism and input that assisted in the efforts of this project.

ABSTRACT

The problem of determining the existence of a path between vertices in problem domains with large graphs is outpacing the increases in commonly available processor speeds. This presents a growing need for pathfinding algorithms which can capitalize on parallel approaches. These approaches are often based on parallelizing the search on a single machine. However, some problems may be so large that it becomes appropriate to use distributed computing. This research explores the Distributed Fringe Search algorithm as a more conducive approach for pathfinding problems over multiple distributed machines. The work presented here is novel in its extension of DFS by developing the Distributed Computing Fringe Search. Additionally, this research proposes the Hash Distributed Fringe Search that utilizes space abstraction techniques for work distribution and a more uniform memory requirement. Finally, results are presented to show the impact of the approaches in large searches; these results inform suggestions for future work.

TABLE OF CONTENTS

LIST OF TABLES vi

LIST OF FIGURES vii

LIST OF SYMBOLS AND ABBREVIATIONS viii

Chapter

I. INTRODUCTION 1

II. BACKGROUND 4

Pathfinding Algorithms 5

Memory Architectures 14

Parallelization Approaches 18

III. METHODS 21

Problem Generation 22

Testing Environment 25

Design Decisions 26

DCFS Development 28

HDFS Development 37

IV. RESULTS 44

Fringe Search 45

Distributed Computing Fringe Search 48

Hash Distributed Fringe Search 51

V. DISCUSSION 56

DCFS Review ... 58

HDFS Review ... 66

Conclusion ... 71

Future Work ... 72

BIBLIOGRAPHY ... 75

APPENDICES ... 76

APPENDIX A – SUPPLEMENTAL MATERIALS ... 77

LIST OF TABLES

Table 1 – Fringe Search statistics 45

Table 2 – Distributed Computing Fringe Search statistics 48

Table 3 – HDFS statistics with non-nested hashing 52

Table 4 – HDFS statistics with nested hashing.. 54

LIST OF FIGURES

Figure 1 – Distributed memory diagram 16

Figure 2 – Single Program Multiple Data (SPMD) diagram 19

Figure 3 – Gaussian example problems 24

Figure 4 – DCFS communication pseudocode 30

Figure 5 – Fringe distribution issue 34

Figure 6 – Hashing and nested hashing 40

Figure 7 – HDFS nested hashing for 16 processors 41

Figure 8 – FS successful searches 47

Figure 9 – DCFS comparison for 11 Gaussians 50

Figure 10 – HDFS comparison for G11 tests with non-nested hashing 53

Figure 11 – HDFS comparison for G11 tests with nested hashing 55

Figure 12 – All algorithms compared for G11 tests 58

Figure 13 – DCFS cache comparisons 59

Figure 14 – DCFS timing components 61

Figure 15 – HDFS timing components 68

Figure 16 – HDFS level one and two nesting time components 68

Figure 17 – Comparison of HDFS for the G11 tests with and without nesting 69

LIST OF SYMBOLS AND ABBREVIATIONS

AI – Artificial Intelligence

BeFS – Best First Search

BrFS – Breadth First Search

CPU – Central Processing Unit

DAG – Directed Acyclic Graph

DCFS – Distributed Computing Fringe Search

DeFS – Depth First Search

DFS – Distributed Fringe Search

DLP – Data Level Parallelism

HDFS – Hash Distributed Fringe Search

ILP – Instruction Level Parallelism

MPI – Message Passing Interface

NPC – Non Player Character

NUMA – Non-Uniform Memory Access

OS – Operating System

SDD – Structured Duplicate Detection

SMP – Symmetric Multi-Processor

SPMD – Single Program Multiple Data

UMA – Uniform Memory Access

CHAPTER I

INTRODUCTION

Pathfinding is a problem which has an array of applications from the small to the extremely large scale. The problems themselves are useful for representing domains in which there is a known beginning point and a desired end point which must be reached through a series of decisions. These decisions may be movement directions such as in the case of a real or virtual agent navigating some space. Alternatively, these decisions can be used in puzzle style problems, such as the popular sliding tile puzzle in which pieces must be rearranged to form a desired pattern; in this case the decisions yield different states to generate a 'path' in higher special dimensions. Common applications for pathfinding include video games in which one or more non-player character (NPC) must be able to traverse the game environment through an effective and efficient route. The pathfinding problems posed by these and similar scale applications are designed to be solvable by a single machine using tried and tested algorithms which have been optimized for such use cases. However, there are an increasing number of applications which consider a much larger state space that is impractical at best and infeasible at worst to solve on a single machine system, even with a powerful CPU [1-2, 4, 6]. For problems such as these, it is desirable to be able to use distributed computing where multiple processing elements are connected over some network. However, the distributed nature of these systems brings with it considerations that must be accounted for that are naturally not present in single machine systems. Such considerations require more nuanced approaches for implementing most non-trivial algorithms including those related to pathfinding. This is primarily due to the separate memory spaces of such systems and

the unique challenges that this presents. These challenges also exacerbate the difficulty of attempts to parallelize such algorithms to utilize the clustered processing elements. Given these challenges, the research conducted here explores the application of pathfinding algorithms that are more adaptable to distributed computing environments. The adaptability of such algorithms can then be leveraged to provide an approach that is capable of handling larger scale problems than a standard parallel implementation of the original pathfinding algorithm. This implies that the efficiency of the solution is not the entire goal with the approaches developed here. While it is certainly the intent to make effective algorithms for distributed pathfinding applications, it is also focused on the ability to potentially handle such large problems that would not even be feasible on a single system due to memory limitations. Because of this, the cost of efficiency may be considered acceptable since it provides a means of processing much larger problems overall.

The basis for the research conducted here was formed during earlier research into the family of current pathfinding algorithms. That work focused on identifying parallel and distributed pathfinding algorithms with the goal of extracting the various techniques used to achieve the parallelization. This prior exploration resulted in identifying common challenges and solutions related to the parallelization of pathfinding algorithms as well as discovering certain algorithms that used approaches which allowed for the extension from a purely parallel approach to a distributed one. The previous work, although unpublished, was also beneficial in focusing the efforts of this study by informing the creation of a foundation for the topics studied in this work. Of the approximately 20 different algorithms that were surveyed, the majority utilized a centralized planning

algorithm with synchronous operations within a shared memory environment. In this context, centralized planning refers that a single process that is capable of interpreting the problem space and planning a route through it. The synchronous nature refers to how the algorithm expands through the search space and may communicate with multiple 'branches' of a search; some algorithms may make use of multiple searches executing in parallel and then attempt to somewhat stitch them back together for an ultimate, although possibly sub-optimal, path [3-6]. Because most of these algorithms were based on a symmetric multiprocessor (SMP) system, further investigation was needed to determine how some, if any, of these approaches were conducive to being extended to distributed environments. This was a somewhat challenging task, considering the fundamental differences in design decisions that are required in significantly different processing and memory systems. To accomplish this however, each candidate algorithm was reviewed individually in order to gain a better understanding of the nuances of the approaches. Specifically, the algorithms were studied to determine their potential for scalability to a distributed system and any other facets of interest that they may provide. This process concluded that the ideal pathfinding algorithm for parallel processing in a distributed environment is one that minimized the need for dependency between concurrently executing processes throughout the execution of the algorithm.

CHAPTER II

BACKGROUND

The following sections will provide background information as it relates to the area of pathfinding algorithms, optimization approaches, and the specific algorithms that are expanded upon in this research. This section begins with a summary of the general pathfinding algorithms that are widely accepted and utilized. It should be noted here that the algorithms summarized in this section by no means represent an exhaustive list of pathfinding algorithms; instead this should be taken as a selection of those algorithms which have proved adaptable and which have provided the basis for other versions featuring various optimizations for speed and/or memory usage. Following this, information is provided regarding the two architectural differences for memory systems that arise when devising algorithms suited to distributed systems. These differences are significant in understanding the limitations they impose for the environment in which the algorithms will be processed. Finally, this section provides information on the basic approaches that are used in the field of parallel processing. In a distributed computing environment, a primary concern is the minimization of intra-network communications between processing elements. By minimizing these communications, less overhead is created and the processing elements can be dedicated to process the algorithm more efficiently. This is a significant concern that must be considered when designing the parallelization of an algorithm that is suitable for a distributed computing environment. These and other concerns that are less relevant in single machine systems will be discussed in more detail in the following sections as they are extremely relevant and inform the process taken in developing the enhancements that are presented in this study.

Pathfinding Algorithms

Pathfinding algorithms are designed to find the best possible path from a source to a destination within some graph structured environment. This path is usually intended to be the shortest or at least very close to it. In the context of pathfinding algorithms, the shortest path is said to have the 'optimal' cost, in which cost refers typically refers to the distance traveled; although cost may be based on other factors, the problems studied here are grid based searches and distance is appropriate. The most basic pathfinding algorithms are commonly referred to as uninformed searches. There are two basic approaches for these searches, each having their respective strengths and weaknesses. As suggested by the name, a depth first search (DeFS) is conducted by visiting a vertex followed by subsequently deeper children until a leaf or terminal vertex is reached. A benefit of this type of search is that it may find an acceptable solution quickly without needing to expand much of the unused space. However, this is not guaranteed to be the case and it is in fact common that it may need to search a large amount of the space; additionally it is susceptible to sub-optimal solutions where the path identified is not the cheapest. Alternatively, a breadth first search (BrFS) visits all of the immediate children of a vertex before proceeding to a deeper layer. The benefit of this type of search is most notably that it can provide an optimal solution. In terms of processing time, it will find a solution as fast as the worst case of a DeFS applied to the same problem. So while it may not be any faster than its depth-based alternative, it is no slower and is capable of providing better solutions. There have been various attempts at utilizing both of these kinds of algorithms for distributed and parallelized versions. There have also been efforts at improving these algorithms such as Iterative Deepening Depth First Search (IDDFS)

where the depth of the search is increased gradually until the goal is found. IDDFS has a smaller memory footprint than the basic BrFS and allows for optimality guarantees. However, this requires repetitive processing of nodes due to its iterative nature [2, 5-6]. Similar to the BrFS approach, Dijkstra's algorithm is one of the most established for pathfinding and is considered a best first search (BeFS) algorithm [5-10]. A BeFS is classified where the most ideal vertices are searched first. In the case of the Dijkstra's algorithm, it operates by selecting the vertex with the cheapest cost to visit at each new expansion. Uninformed searches frequently require a large amount of vertex visits in order to locate the goal vertex because, by definition, they have no information about what areas are best to search. Conversely, informed searches use some additional information in the search so as to hopefully guide it to the goal and reduce the amount of vertices searched. Arguably the most popular informed BeFS algorithm is the A* algorithm, which is itself based on Dijkistra's algorithm.

Because the A* algorithm and its characteristics are so commonly applied to and borrowed from, they will be examined in more detail here. The A* algorithm improves upon the basic pathfinding algorithm by including the notion of a 'heuristic' component to guide the search of the space. This additional heuristic measurement is used to prioritize the assessment of the nodes within the search space where each vertex has a 'F' cost which is equal to the addition of a 'G' cost representing the incurred cost plus the 'H' cost of the heuristically estimated remaining cost. Under certain conditions, a heuristic may reduce the amount of space searched. These heuristic measurements can be computed in various ways; for grid based problems, the Euclidean and Manhattan are the most common methods and the latter can be effectively extended to other problems such

as the sliding tile puzzle. The former heuristic is computed as the straight line distance between the current point and the goal point whereas the latter is computed as the sum of the lateral distances between the two points. For problem domains that are not based in a geometric space, such as puzzle solving applications, a heuristic can be developed that provides a reasonable approximation of the ‘moves’ remaining until a goal state can be achieved. Care must be taken when designing such heuristics however, because if a heuristic is allowed to overestimate the actual cost it no longer qualifies as ‘admissible’ and therefore can produce sub-optimal solutions. This means that a heuristic has to be chosen very carefully in order to provide a reasonable expectation of the total distance yet to traverse to the destination. It is desirable that the heuristic provide an approximate underestimation of the actual cost remaining to the goal. By doing this, the heuristic qualifies as admissible, that is, it should always estimate less than the actual number of edges necessary to traverse. With the use of heuristics, if the function computes an inadmissible heuristic value, then A* may expand fewer nodes overall but may also produce a suboptimal path. Absent a heuristic function being used or in the case that the heuristic measurements are all zero, A* becomes a Uniform Cost Search in which nodes are expanded based on priority of their G cost.

The initial step of the A* algorithm is to assess the beginning state of the problem space. The beginning state, i.e. the source from which the path is to be found, is queued into the open list for processing. This open list represents a container for vertices that are in a queue to be processed; this queue is sorted based on the F values of the nodes. From this point, there are just a few key actions that are repeated on each element pulled from the open list until a goal state is reached; in the event that the open list is emptied before

the destination is reached, it signals that no 'path' exists between the goal and the start state. For each element removed from the queue, it is first compared to the goal state, if the end has been reached then it retraces the path using the element's reference to its parent. Otherwise, the element is placed in the closed set and its adjacent elements are processed. As each new element is processed, the algorithm expands the vertex by looking at its neighbor states; these states represent the potential states that can be reached from the current state. Each of these 'children' is then assigned their appropriate information, much of which is based on their 'parent' including a reference back to that state. Information about the parent is necessary for reconstructing the path taken to solve the problem. At this point, each child is then examined against the closed set. This is necessary to prevent the algorithm from falling into cycles and helps prevent unnecessary vertex creation. If a child is already in the closed set, this means that it represents a state that has already been expanded, and therefore does not need to be examined again. Because of the priority sorted open list, once a vertex has been closed it can be considered to have found the optimal route to that vertex from the given start; this means that if the same state is seen again and has been closed, there is no benefit in reevaluating it. Any children that were not in the closed set are then added to the open list where they are sorted by priority among the elements already in the list. At this point, the loop can then repeat itself by pulling elements from the open list and examining them until the goal is reached or the queue is empty.

The A* algorithm is capable finding paths relatively quickly due to its informed nature, assuming an admissible heuristic, and with the assistance of the open and closed lists, the former of which is priority sorted to search the most ideal vertex at each

iteration. Because open and closed lists are used to retain information about which nodes have been visited before, an A* search will essentially ignore nodes that have total cost estimates until those costs rise to the front of the priority sorted open list. In this way, the informed search is guided through the space and can perform well with a good heuristic. Although this assists the search by limiting the area searched, it is associated with two significant costs. First is the memory requirement necessary to retain each vertex visited in the open list and all expanded nodes in the closed list. This means that for large problem domains, the memory requirement will grow significantly even for relatively direct paths due to the exponential nature of the search expansion; for extremely large searches this could exhaust the memory resources of a single machine system. The second main cost associated with A* is that of re-sorting the open list after each iteration. As the number of nodes in the open list grows exponentially, the time needed to sort the open list after each list grows commensurately; even with optimized sorting routines, this is a significant cost to the overall search since it must be performed after each iteration. While this sorting can be acceptable for some applications, it is extremely inhibitive to parallelization, especially when attempting to use distributed means.

As a response to these challenges, a modified approach to A* uses the iterative deepening technique which operates with A* similar to how IDDFS operates with DeFS. This deepening approach is combined with A* to conduct the typical search pattern with a depth threshold based on the heuristic estimate of the beginning point. The significant difference between this and classic A* is that no supporting open and closed lists are used. To account for these complications, IDA* uses the iterative deepening technique to gradually expand the search under some threshold cost which is computed by A* in the

typical form as the sum of the incurred and estimated costs. Once all nodes have been expanded along the search up to the threshold, a single iteration has been completed. If the space under that total cost is searched and the end point is found, the search terminates. If not, the search restarts from the beginning with the threshold increased to the minimum of the estimated next steps; during each iteration, the next highest cost of expansion that is above the current threshold is recorded to provide this increase. The iterative approach provides for a memory savings since at each iteration, only the nodes under the current heuristic estimate need to be stored in memory for processing. Because the search is completely restarted with a new threshold for each iteration, the open and closed lists are not required since all nodes will be revisited with the addition of those between the previous threshold and the newly increased one for the next iteration. Additionally, because no open list needs to be maintained, there is no sorting in between each iteration. Although this algorithm can function without an entire open list, some form of cache is typically used to prevent processing nodes in cycles repeatedly; however this can be implemented as a hash table of the states and is still significantly less expensive to maintain than an open list and associated sorting. The tradeoff with not having the costs of open and closed lists is that the algorithm repeats itself with each new iteration, so there is much redundant processing.

Although it is possible to parallelize A* on a single machine environment, it proves non-trivial and frequently requires tuned data structures that are optimized for the application. Due to the required tuning, a naïve parallel approach is very likely to prove less efficient than a comparable serial one [5, 7-10]. This is primarily due to the challenges to parallelization posed by the two aforementioned factors, i.e. maintaining

open and closed lists as well as resorting the former list after each iteration. Intuitively, it may seem that a multiprocessor approach could allow each processor to pull an item from the open list, process its adjacencies, i.e. determine which vertices are connected to it, and update the open and closed lists respectively. However this has two less obvious implications that hinder such an approach. First is the common issue with parallel approaches in that, if each processor were to expand a vertex and update the open and closed lists, access to those data structures must be regulated to prevent data hazards on reading and writing the data. For instance, a processor may check the closed list to see if a vertex has been processed immediately before another processor marks it as so, resulting in redundant processing. A similar and possibly more severe situation can occur on the open list side if a processor were to process the adjacencies of a vertex and update the open list just after another processor has taken the 'best' element from the open list because it did not have the newly processed data. The impact of the later situation is a possible loss of optimality in the search due to the fact that one processor may find a sub-optimal end point while the actual optimal end point has yet to be processed. Applications in which pure optimality is not required may accept a small sacrifice in accuracy or may opt to use a strategy to regain the optimality by requiring a solution to be held until it can be guaranteed optimal. Besides the issues of redundant processing to loss of, or at least delayed, optimal path finding, the more serious issue is related how each processor is allowed to access the lists to actually perform the updates. This relates back to the issue of regulating access to the data, which is typically achieved through memory locks. Although regulating access to the data structures will ensure that only one processor can read from or write to it at a time, there is almost inevitably a significant

amount of time spent idle by each processor waiting on acquiring a lock. This is an issue that is not solvable by simply allocating more processing elements, since each new processor will still need to wait for its turn to access the data as well. Altogether, these issues represent the challenges faced when attempting to adapt the A* algorithm directly for parallelization. Combined with the high memory requirements for A* searches and parallelization perils an interesting opportunity is present for exploring distributed computing applications of attempting to accommodating these challenges; this thesis explores such attempts further.

Because the open and closed lists can place a large burden on memory for large search space, a natural suggestion would be to use a distributed system where the memory of multiple systems can be applied to the problem. Of course, some interface must be used to handle the passing of data back and forth, but it seems reasonable that the additional memory should make the problem more easily solvable. However, it is the sorting requirement of the A* open list that complicates this because, even though it is feasible to distribute the open list in memory between separate machines, the sorting would require a significant level of cross communication. Additionally, even if a mechanism is used to allow each processor access to the open and closed lists, there would need to be a significant amount of regulated access to ensure consistent processing by each processor; this presents a common issue of parallelized algorithms that suffer due to inactivity while a processor waits to acquire and subsequently release a lock. Though this is again technically possible, the sheer amount of communication between systems would be so costly that it can outweigh the benefit of using multiple machines in the first place. As a side note, the author came to this realization a few years ago when attempting

an intuitive implementation of A* on a cluster; it was this discovery that prompted what eventually resulted in the research presented here. Based on this early discovery, a more extensive study was conducted of distributed pathfinding problems.

After exploring specific approaches to parallelizing pathfinding algorithms, two candidate algorithms were identified that appeared viable for conversion into a distributed computing environment. One algorithm identified was Asynchronous Hash Distributed A* which was designed by Burns et al. in 2010 and utilizes space abstraction techniques [6]. This approach uses an abstraction function to create a condensed graph that preserves the overall connectivity while reducing the states into coarser blocks; state space abstraction is a logical approach for representing large spaces and will be discussed later in this work. This algorithm was itself based on a pre-existing distributed implementation designed by Kishimoto et al. in 2009 [4]. Because such a distributed computing approach had already been proposed, the focus turned to another previously identified algorithm titled Distributed Fringe Search (DFS) designed by Brand & Bidarra, in 2011 [5-6, 9-10]. This approach is itself based on Fringe Search (FS) which is described by Bjornsson et al. in 2005 and is itself an enhancement of IDA* as described earlier [3]. FS uses the approach of having unsorted Now and Later lists instead of the Open and Closed lists as A* uses. The Now list contains all vertices at the fringe of the search and then opens any under a given bound where the bound is the heuristic of the beginning vertex which is then increased as the search progresses. The Later list contains vertices that are encountered while expanding the fringe that have a cost estimate greater than the current threshold. This alleviates the issue of a large sorted open list and accompanying closed list used for all vertex expansion operations. In terms of

parallelization efforts, the lack of requiring a sorted list means that it can be distributed without disrupting some order and that additions to any segment of the list does not impact a presorted order; this avoids the need for collecting, resorting, and regulating access issues as described earlier. These adaptations are applied directly in DFS in order to allow for the distribution the Now list data to available cores. In this approach, the FS algorithm expands as it would normally, however the Now list is distributed at the core level, with each core being responsible for a section of the list. Load balancing is accomplished through pointer manipulation of the list with each core taking nodes from the center of the list and growing in either direction. The extension of this approach to a truly distributed memory system would involve translating the operations that DFS uses to allocate portions of the Now list to the cores to instead allocate to separate processors in a distributed system. Although this should be relatively straightforward data parallelism, in order to minimize any additional communications, a distributed approach may benefit from some pre-computation strategies to help inform the potential distribution process. Additionally, while the parallel implementation of DFS showed some good results, it was noted by the authors that the load balancing was less than ideal, as one core typically handled the most nodes. This is due to the nature of the heuristic used in A* which extends the space toward the goal. Any modification that could help better distribute the space initially may allow for improved work distribution among the processing nodes.

Memory Architectures

The implementation of A* or other pathfinding algorithms requires a significant amount of memory coordination. Depending on the scale of the pathfinding problem

being approached, a parallel implementation typically takes on one of two basic memory models. Smaller problems that are capable of being solved by a single machine typically operate in the shared memory architecture. Common applications for this type of pathfinding can be found within games where some AI agent must navigate an environment; it is unrealistic to expect that the player will have the computing resources to allow the agent pathfinding to be distributed to other machines. Execution within a shared memory environment can be convenient because it typically guarantees uniform memory access (UMA) which allows for consistent memory access times; this is commonly realized on a modern SMP machine. For pathfinding algorithms such as A*, a parallel implementation typically involves having separate threads process the next available vertex from the open list. Because of the shared memory, this means that each processing element, often cores or chips on the same board, are able to access all of the memory simultaneously without requiring additional busses or networks. This is desirable during the open list, and to a lesser extent, the closed list operations because it is often easier to regulate access to these data structures. However, care must be taken to assure there is no contention between processing elements, so memory must be locked to prevent concurrency hazards. Although this is a manageable process, excessive locking and unlocking can cause the benefits of parallelization to be lost to the processing elements waiting on lock acquisition [5-10]. Actual implementations of this kind of parallelization are often accomplished through the use of the POSIX threads (Pthreads) functionality standard or similar framework that allows for thread coordination.

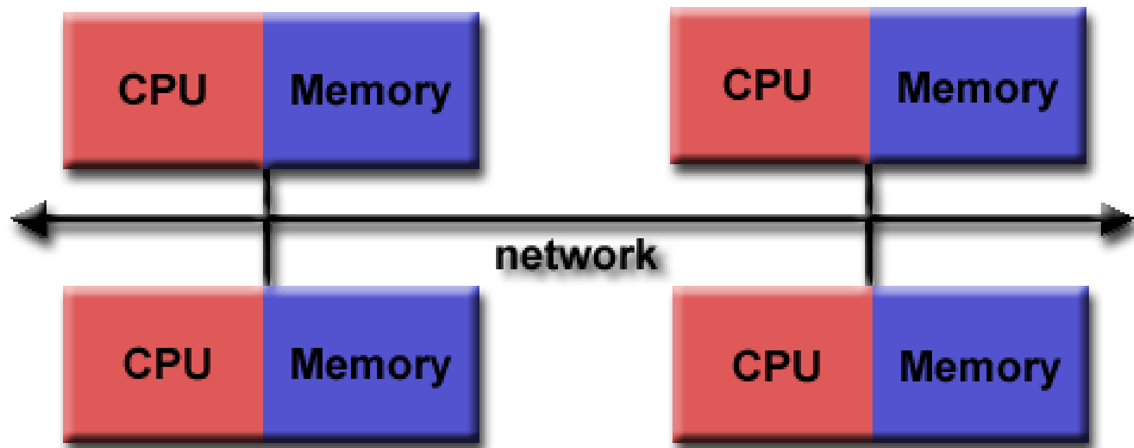

Figure 1. Distributed memory diagram [11]

Implementations of pathfinding algorithms such as A* in distributed computing environments are by nature operating within a distributed memory architecture. This is a basic extension of shared memory in which multiple combined processing elements and memory units are connected over a network. It is important to note that these units connected among the network are likely operating individually on a shared memory architecture which is essentially nested within a higher distributed memory system. This architecture is often referred to as distributed memory due to the distributed nature of the memory a CPU may access either local data or must request data that will need to travel over the network (see Figure 1). The benefit of these combined layers of memory is that each processing element on the network can then subdivide its workload to its own local processing elements for increased efficiency [1, 4, 6, 11]. Systems that utilize distributed memory architecture are designed to use the combined processing power and are ideal for computationally intensive applications. Frequently this includes problems in which it is unrealistic to wait for a single processing element to perform the computations and/or the

size of the graph is so large that it cannot be retained in memory local to a single processing element. In order to resolve these issues however, pathfinding algorithms in a distributed environment require a different approach than the somewhat more intuitive one possible with shared memory. Specifically, this is often accomplished with a Message Passing Interface (MPI) to communicate between the processing elements on the network; although MPI refers to a standardized version for message passing, any available message passing communication method is feasible. Whereas for A* operations in a shared memory model, each processing element is able to access the open list and also determine if it is currently locked by another processing element, this approach, while technically feasible, is ultimately unrealistic. This is because of the sheer amount of overhead that would be required in communicating a single open list between all processing elements on the network; these communications would need to occur for each operation on the open list from each processing element to all other processing elements. As other research has noted, excessive locking and communication can actually reduce the performance of a parallel implementation to below that of its serial counterpart [5-10]. To solve this problem, most methods attempt to break up the need for a consolidated open list by distributing a subset of the space itself to different processing elements. The prime question here is how best to decompose the space in order to avoid data dependencies, maintain data locality to the processing element, and maintain optimality of the solution. There is no clear evidence indicating that any or all of these factors exist in mutual exclusion.

Parallelization Approaches

In the general terms of parallelization, approaches generally entail either data level parallelism (DLP) or instruction level parallelism (ILP). For DLP, the technique involves dividing up a large block of data and distributing the pieces out to separate processing elements; it should be noted here that the 'distributing' does not necessarily imply that this occur within a distributed memory model and instead refers to the distribution of the workload to the processing elements which may exist on the same machine. In the case of ILP, a similar process of distribution occurs except that instead of sharing the data, some set of instructions are executed by different processing elements; this can be commonly implemented with some kind of multithreaded model such as Pthreads. For both of these approaches to parallelization, they may make use of either distributed or shared memory [6]. In either memory architecture, it is desirable and increasingly feasible to utilize multithreaded programming, as even most commodity processors now have multiple cores [5-10]. Given the choice between distributed and shared memory implementation, the most appropriate option depends primarily on the intended application for pathfinding; often implied in this is also the type of systems that the algorithms are even solving the problem in a realistic timeframe. For instance, in the case of game environments with large spaces in which AI agents are expected to traverse, the game world typically exists within a shared memory system [7-10]. This implies that the algorithm will likely be able to access the necessary environment data as needed with minimal memory access times [7-11]. Conversely, distributed memory architectures are more often used in applications in which the search space is too large to be maintained on a single system [1, 2, 4, 6]. Distributed computing systems also frequently use the Single

Program Multiple Data (SPMD) model which allows multiple machines to execute the same base program where each processor can work on different data or even have slightly different tasks (see Figure 2). This brings with it issues related to accessing memory that may not reside on the current processing vertex. Not only can the issue of access latency due to the network be a problem, but the sheer communication overhead can be overwhelming if the memory locality is not managed appropriately.

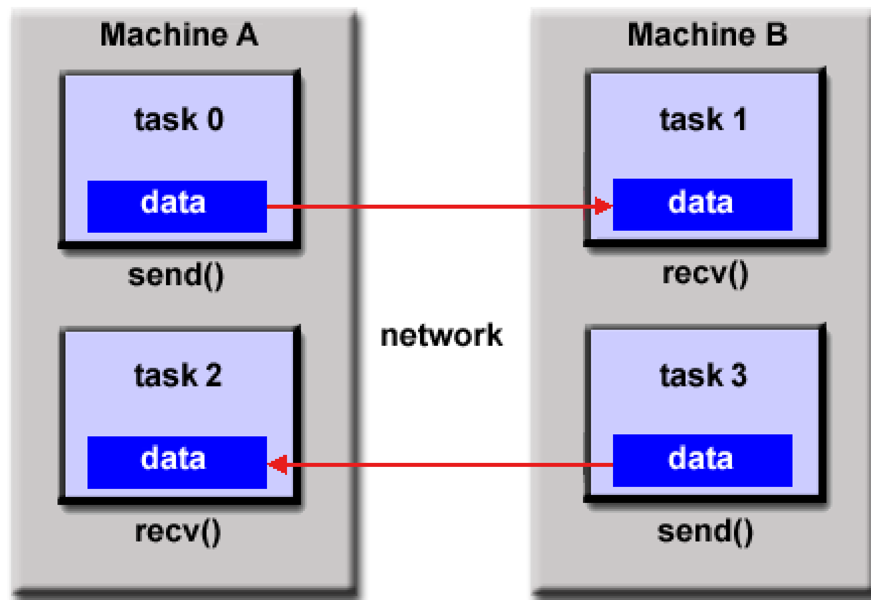

Figure 2. Single Program Multiple Data (SPMD) diagram [11]

Either approach to parallelism of A* frequently involves the management of the open and closed lists and possibly during the adjacency generation phase. A serial implementation of the A* algorithm assumes that the open list will be processed greedily with respect to the F cost of the queued vertices. The subsequent expansion of the ‘best’

vertex will then generate the adjacencies which are expected to, but not necessarily, have improved F cost compared to the other vertices in the open list at the time of their creation. When the open list of items is processed sequentially this way, an optimal path can be guaranteed. As the open list itself is a collection of distinct vertices with no direct interdependencies, this makes it a prime choice for attempted parallelization. However, when this effort is realized, the guarantee of optimality cannot be maintained. Although there are no dependencies between the elements of the queue themselves, the order in which they are processed can influence the result. This is because a parallel execution which allows a processing element to pull from the open list may result in the expansion of a vertex which is along a sub-optimal path to the goal while a different processing element has not yet expanded another vertex which is on the optimal path. Because A* is inherently based on a serial processing of its open list, it will accept the first 'goal' state reached and assume that it has discovered the optimal path assuming that the open list was processed according to the necessary order. Because of this issue with adapting A* and other pathfinding algorithms to operate in parallel, the typical resolution is to hold on to a solution temporarily for a certain number of expansions or until the open list has been exhausted under the temporary solution cost. Although this typically involves more vertexes being processed, it does allow for the benefit of pruning once a possible solution has been found. These considerations are most significant in shared memory systems in which the processing elements are able to access all of the same memory at any given point. However in a distributed computing model, each processing element has its own memory that it can access directly but may also occasionally need to access memory which resides local to a different processing element.

CHAPTER III

METHODS

The following section will provide more details on the development and implementation of the selected algorithms as well as the design of experiments and selection of related metrics for this study. This research began with a desire to propose an approach that is more suitable to a distributed computing environment compared to existing algorithms. These algorithms may have been designed with parallelization in mind, however they were also intended to be executed on a SMP system; so while these previous approaches provided a basis for the current work, the research conducted here is novel in its enhancement of some of those concepts to a distributed computing environment. Initially, this project used information gathered in previous work conducted by the author in reviewing algorithms that were either themselves distributed pathfinding algorithms or ones that provided interesting approaches to solving pathfinding related problems in parallel. Ultimately, that work was condensed into the identification of the Distributed Fringe Search (DFS) developed by Sandy Brand [5]. DFS was selected due to the fact that the algorithm which it is based on, namely Fringe Search, is innately more adaptable to the alterations needed to work within a distributed computing environment. Primarily, this is related to the alleviated need of a sorted queue for processing elements in addition to being slightly less memory intensive than its A* relative; please refer to the Background chapter for a more detailed description on the operational differences between A* and FS. It is these features of FS that allowed Brand to develop DFS for a multicore approach. By extension, the work conducted here can be thought of as similar in that, instead of distributing work to other cores for processing during execution, the

work is instead capable of being distributed to another processor that may be on the same machine or connected across the network of a cluster.

Problem Generation

A need for a consistent set of problems to compare the potential performance of algorithms was identified in the early stages of this research. By identifying a standard and varied selection of problems, all algorithms developed through the course of this project could be tested equitably. Additionally, by providing some variation on the structure of the problems themselves, it is possible to highlight behaviors of the algorithm that may not be evident with less varied testing problems. In this work, the domain is composed of grid based test problems where the vertices are arranged similar to the points in a Cartesian coordinate system. All grids points are the standard one unit apart on each axis and $\sqrt{2}$ apart at the diagonals; diagonal moves are allowed in all algorithms implemented here. Euclidean distance is used as the heuristic for all algorithms tested in this study. For all problems, the beginning point is considered to be at (0, 0) in the conceptual upper left corner of the grid and the end point at the lower right corner. Because the algorithms presented here are intended for applications which must find long paths through large spaces, these constant beginning and end points are reasonably justified; testing with randomized paths could have led to arbitrarily short paths which would not accurately represent the performance of these algorithms and the problems that they are designed to solve. Rather than test each algorithm against a specific time benchmark, statistics were gathered from each permutation of the runs including time and data about the actual data processed. For the distributed algorithms, additional data points were recorded related to the timing the components of interest, especially as it relates to

communication that is not present in the serial algorithm. In generating the actual problems, they were designed to illustrate the behavior of each algorithm being tested. The problems themselves were composed of 300 randomly generated grids of one million points each which were built using custom sum of Gaussians problem generator. This generator created the grid with a variable number of Gaussian centers and adjustable 'C' factor where C is represented in the Gaussian function as:

$$ae^{-\frac{(x-b)^2}{2c^2}}$$

In the application of this function, C controls the relative 'width' of the distribution. By adjusting this value, the centers become smaller or larger while still maintaining the same relative distribution. It is necessary to adjust this factor when tuning the input problems themselves. In order to test on a variety of problems, a failure rate of 5-15% was considered acceptable, i.e. no path exists from the beginning to the end point. The complicating factor that arises when this is combined with a variable number of centers is that the width must be reduced as the total number of centers is increased so as not to completely obscure the path and to stay within the desired failure percentages. It was experimentally obtained that for grids with 2, 11, and 20 center points that the corresponding C values were 100, 50, and 25 with their respective failure rates of 5, 11, and 7% (see Figure 3).

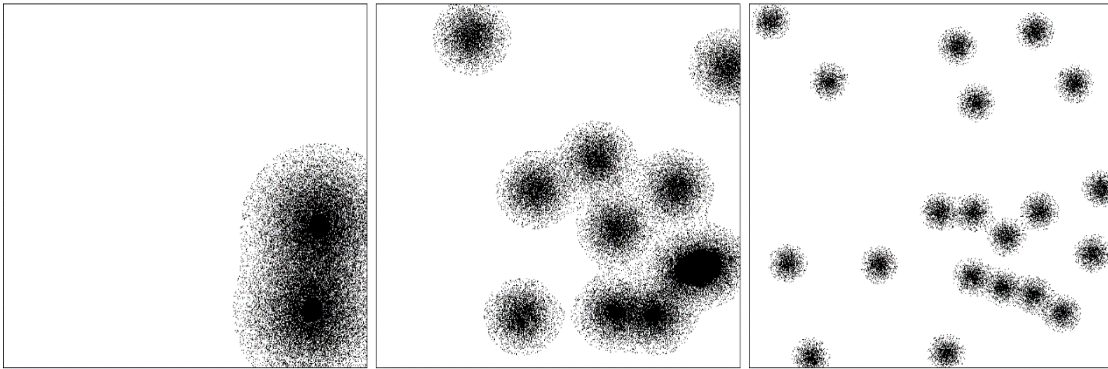

Figure 3. Gaussian example problems

The problems themselves are used in testing against each of the three algorithms, FS, DCFS, and HDFS. This provides similar data for the performance of each algorithm on a standard set of tests. These tests are used to examine the differences in performance of each algorithm and provide information about the related causes and effects. For the DCFS and HDFS algorithms, the additional testing factor of how many processes were allocated to the algorithm becomes relevant. Both algorithms were tested with 4, 8, 16; this equates to running on one, one, and two machines in the cluster respectively which will be explained in more detail in the following section on the testing environment. Selecting the above number of processes allows for the testing of a distributed setting in which the distribution is actually occurring over a single machine or two separate machines; testing with 4 processors was included as a base point to be similar to the original DFS by Brand [5]. In this way, it is intended to elicit discernable behavior that may be more difficult to detect with too few or too many processes; future tests on larger problems could consider a higher process count as appropriate.

Testing Environment

To accomplish the distributed communications, the approach developed in this project utilizes the Message Passing Interface (MPI) framework, more specifically the MPICH 3 implementation [12]. Using MPI, the algorithms developed here are able to communicate the necessary data to each process regardless of which machine in the cluster it may be on; to use MPI terminology, this is often referred to as the 'rank' of the process which equates to a single processor that is responsible for some part of the program. This illustrates the Single Program Multiple Data (SPMD) model that MPI is designed for; that is, a single program is written once and different processors can handle different datasets while all running concurrently. For all experiments conducted here, the environment consists of machines with eight 64 bit processors available on each. For instance, if the program were to be run with eight processes, MPI will coordinate how and where each of those processes should be executed on. To extend the previous example, if a program were run with 16 processes, the first eight would all be allocated to one machine and the others to the next machine; by doing this, there is a slight potential for saving for some processes that may communicate over the same machine rather than over a slower network, but all still using the same MPI standard for communication. Although some machines in this cluster were technically capable of running 16 processes, only eight of those would have been on real cores with the others being virtualized. Because of this, it was decided to use only the number of real processor available on each machine so that each machine had the capability to contribute evenly to the overall computations. In terms optimizations, the only features used external to the code itself was during compilation. All executables were compiled using g++ or mpicxx as

appropriate with the additional O2 flag for some possible assembly level improvements. Because this flag is an available standard through the compiler itself, this is not considered a significant advantage that would distort the testing results in comparison to other potential implementations.

Design Decisions

Three algorithms were implemented in this research and each was tested on the same set of problems, with only small variation being due to additional testing factors possible in the distributed algorithms that were not applicable to a serial algorithm. First, the serial FS algorithm was implemented as described by its authors [3]. Based on the descriptions and pseudocode provided by its authors, the version of FS tested here is comparable in its overall design and operation. FS was completed relatively easily and tested against several small problems that confirmed its ability to backtrack correctly. Additionally, the preliminary FS output showed that it is correctly expanding nodes in the fringe based on the original algorithm. By implementing the serial FS first, it allowed for the development of a baseline of the algorithm for debugging and testing as well as providing a foundation from which to develop DCFS. Additionally, this provided a serial basis for comparisons against the distributed versions. Also, any issues with the base algorithm could be corrected more easily without the fringe distribution potentially confounding the issues. A significant aspect in the design of this algorithm is the use of hash table maps to represent the now, later, and cache lists; these associative containers allow a string representation of the state to be indexed to a ‘Node’ structure that contains all the pertinent data regarding that vertex in the space. By default, these containers would be sorted by their string representation. Because this is only a textual name of the

state, a custom sorting rule was applied to preserve some notion of ordering between the states. According to this rule, the items in the maps are 'sorted' internally based on their relative row and column in the grid, with 'lower' values having lower row then column numbers. For instance with a (row, column) ordering, position (1, 2) is considered 'less' than (2, 1) and similarly (1, 1) is less than (1, 2). Ordering based on this criteria gives rise to a more regular processing pattern than the simple ordering that would be the case with only the string representation of the state used as the key in the hash table. It is important to note here that, while this is a more intelligent ordering than a simple string representation of the state, it comes with a cost. While significant tests were not conducted to determine the extent of the impact, a few preliminary tests were conducted in which the cache map also used the ordering scheme. Even in these few tests, processing times increased approximately 40% on the one grid problem tested. Because of this, only the now and later maps are enforced with the sorting, since it is necessary to maintain for their subsequent collection and distribution. However, it should also be noted that imposing the ordering on the cache likely results in a more significant reduction in performance since the cache map is being accessed more frequently than the Now list and Later list. In terms of the data structure used, this and all other structures were from the C++ Standard Template Library (STL). By using standard structures, this work presents a general approach for these implementations without needing to rely on extensively customized and tuned data structures or external libraries to support the processing. Using this map structure, the FS algorithm is capable of performing all necessary operations on the now and Later lists as well as retrieving vertex data from the cache map. It is worth noting that the implementation developed here differs slightly

from the original FS algorithm description. In this project, a different method is used for how all of the hash tables are checked and updated during processing; this deviates from the pseudocode provided by the original FS authors. Specifically, this relates to the use of the cache lists as it relates to the 'fringe' list that they describe. In their implementation, a single fringe list holds both the now and Later lists that are conceptually separate. Complementary to this, the cache contains all data as it is observed when processing the adjacencies of each vertex. Since only visiting a vertex adds it to the cache, this is significantly different from similar processes such as in A* where it is the full 'expansion' of a vertex that would add it to the closed list. Although the cache is not a closed list per se, it does provide a quick access data structure to check which nodes have already been seen; this is of special importance in grid based problems due to the high interconnectivity of the vertices such as the ones used for testing here. DFS also uses similar techniques for maintaining a set of states that have already been closed.

DCFS Development

The original author of DFS was contacted for any additional implementation details of the algorithm. Mr. Sandy Brand provided some additional documentation on DFS, however no DFS source code is readily available for direct comparison with the work conducted as part of this project. This algorithm is the foundation from which the DCFS is based. The primary difference here is that DCFS communication is modeled after the multicore DFS approach designed by Brand with the necessary alterations to enhance its capabilities to work in a distributed computing environment. In the original multicore approach of DFS, load balancing was achieved by distributing the Later list between four cores using some pointer manipulations allowing each core to have

approximately equivalent work for each new iteration. This would allow the work to be distributed to other processes, whether they were on the same machine or over the network, using MPI constructs; the process here is analogous to the ‘distribution’ of data to available cores in the original algorithm. Because distributed communications have a much higher overhead cost however, design decisions were made to allow a single rank, specifically the master, to begin computations and continue until the Later list is large enough to justify the communication to other nodes. Once the Later list reaches the predetermined threshold, the master will evenly distribute the data in the Later list with any remainder retained by the master in order to minimize communication costs; DCFS as tested here uses a constant 1,000 as its threshold value. After the distributed nodes have been processed by their respective ranks, the master gathers all of the generated Later lists and merges them. In the meantime, the workers will remain idle while waiting for new work. The process can then repeat by scattering the Later list if it is over the threshold and the ranks will process their share of the nodes. Once a rank finds the goal, a special sentinel value is scattered from the master and processing halts; in the likely event that a rank other than the master finds the goal, a sentinel value is sent to the master first and then scattered. In the event that no viable path exists in the space, the master detects that all ranks contributed zero new elements and stops processing. In this implementation, the customary designation of MPI rank 0 as the master is used and all other ranks operating as workers including the master when it is not busy distributing work to the other ranks. DCFS accomplishes this with considerable effort devoted to orchestrating the communication to provide for a clear and consistent set of rules for communication. After the Now list is exhausted in each iteration, communications are performed (see Figure 4).

```
If master
     If work previously distributed
          Gather processed data from workers
          If no new data or end found
               Kill all
     If collected data < distribution threshold
          Transform for self-process
     If collected data > threshold
          Transform nodes for distribution
          Scatter work evenly to workers and self
If worker
     If work processed
          Send data back to master
     Wait for new work
     If found or no path
          Kill self
     Transform data for processing
```

Figure 4. DCFS communication pseudocode

The ‘transformation’ in the above procedure refers to an additional step necessitated by the use of MPI in the algorithm. MPI is capable of handling the difficult intricacies of packaging the data and sending it to the correct destination. In order for MPI to accomplish this however, it requires that the data to be one of the predefined MPI types. Although there is a provision for user defined MPI types, which was used here, the container of those types itself is more restrictive. Moreover, it is at odds with the hash table maps used by the algorithm in processing the vertex data. Because of this, it becomes necessary to have a contiguously allocated container, such as a vector, with the MPI data to be sent; this requires a ‘transformation’ process to extract the Later list data from the hash map to a vector and then the easier process of inserting received data back into the hash map. Despite this seeming like a time intensive process just to allow for the

communications, the results empirically show that it is a trivial amount of time, even relative to the communication time. Additionally, this can be considered as an associated cost of operating in distributed environment and one that may be reduced, but is not completely unavoidable. However, it is still valuable to take efforts to maximize the effectiveness of each communication when it must occur. Another complication brought on by MPI is the additional complexity in the communications required to send potentially variable amounts of data to the different worker ranks. Because the work is split evenly with the exception of the master who retains any remainder, the number of vertices is not constant, nor cannot it be known before the current iteration has actually completed since it is based on the growing size of the Later list. In order to adjust for the unknown amount of data that will need to be sent, two collective communications are actually required. In this context, 'collective' implies that all ranks must participate in the communication before they will all proceed; this also implies that a rank will block and remain idle until all process have reached that point. A priming communication is used to inform all ranks of how much data they will be sending or receiving so that each rank can allocate the correct amount of local memory. Following this, the actual data is sent or received between the ranks. Whereas the regular DFS algorithm is able to distribute work relatively cheaply due to the cores in the SMP system, a distributed system requires more care to be effective and efficient in its communications of data. To balance this communication cost, the threshold value is used for deciding when to actually distribute work. By using this, the master will continue working with the data until the Later list grows to a length were it can be considered worthwhile to take the effort to distribute the work. Setting the threshold value itself requires somewhat of a balancing act. A low

threshold will allow work to be distributed relatively 'sooner' as the search expands, but will also cause more communications overall, which are generally the most expensive part of the process; this is due to the packaging costs as mentioned before, where data must be packed data structures to be sent via MPI, as well as the inevitable latency between processes where one may finish before another and then is idle for some amount of time while waiting for the other process(s) to finish and participate in the communications. Conversely, a high threshold will allow the master rank to delay the expensive communications and work on the problem by itself. However, this provides poor utilization of the distributed approach since the other processes will remain idle until they are given work by the master. Because the master is also a worker, i.e. that it is responsible for all work coordination and also participates in the computations on the data, a threshold greater than the total vertex size would essentially be the same as running the serial algorithm; practically speaking there exists a threshold value less than the total size which will also have the same result, since a search can never have all nodes in the Later list simultaneously. For the implementation here, the threshold was set to 1000 which was low enough to cause distribution when there was adequate work to be done, yet not so large that the threshold may never be reached and the master would be left to do all the work thereby negating the distribution. Ideally, this value should be a function of at least the factors of the size of the space and the number of processes allocated to the problem. Since the Later list is divided evenly among the processes, with the master retaining any remainder in order to minimize communicated data, a constant threshold implies that as the number of processes is increased, each process will receive less work and in turn process it faster and require communication of their results. While

this is good in terms of utilization, it may place a higher burden on the communication costs; ideally the value would allow the master to provide enough work that would keep each process reasonably busy to lighten its workload. At this point it is important to note that this threshold value is not an immediate trigger to cause the distribution. Because the underlying FS algorithm continues placing vertices in the Later list until the Now list is depleted, it is possible for the Later list to grow larger than the threshold within a single iteration; only after the entire iteration is complete, i.e. the Now list is emptied, will the size of the Later list be assessed against the distribution threshold. Despite this seeming somewhat counterintuitive, it is actually quite sensible considering that it allows the search to continue consuming space so long as nodes are found that are under the estimated cost of the current iteration. For instance, consider a perfectly clear space with no obstructions. In such a case, only a single iteration need occur since each subsequent expansion will produce a vertex along the optimal path. While it may be possible to distribute an intermediary Later list, this would likely prove complex and would require more communication than before with potentially little if any real benefit.

A separate issue of the DCFS algorithm has to do with its pattern of distribution of the data. As with regular DFS, there is no intuition on which nodes should be distributed to which cores other than the natural progression of the search itself. Unlike in DFS however, where all cores are able to access the same local memory, albeit through some memory regulation, the DCFS algorithm does not operate with a single globally accessible view of the data being processed. In fact it is because of this that the algorithm is able to scale for larger problems, since each processor only needs to retain the parts of the space that it encounters during processing. However, an issue arises when this is

considered in light of the blind distribution technique. Because the Later list is simply divided among all processes, no way is provided to determine which nodes should be delivered to which processes in order to capitalize on their existing cache contents. In essence, a processor may work on expanding some part of the space at one point during the search, and then a different region later in the search depending on how the frontier expands. Although the custom sorting of the hash table that represents the Later list itself provides some level of relative grouping based on location in the space, it alone cannot guarantee which parts of the Later list will be distributed to which processor during each iteration. Because of this, each processor may end up expanding nodes that it has not explored before, but that were explored by other processors during a prior iteration. For instance, assume vertex A is expanded by rank 0 and generates the child vertices B and C to the Later list. Assume that vertex B is processed by rank 0 and vertex C is sent to rank 1 for processing. When rank 0 expands vertex B, it will have already closed A in its local cache and will not reconsider it. However, rank 1 has no prior information about A, so when it expands C it may reprocess A during the expansion (see Figure 5).

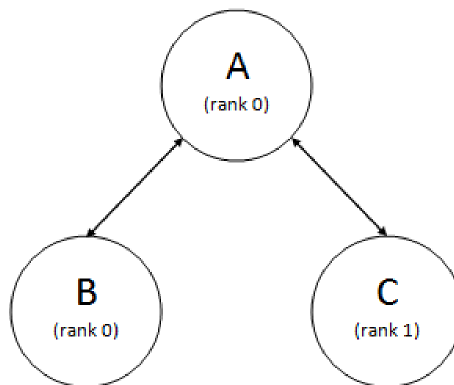

Figure 5. Fringe distribution issue

This issue is compounded by the fact that each successive iteration has no guarantee about which section of nodes that it will receive; that is to say that for a given rank that expands some set of nodes *N* into a Later list set of set *M* that after a subsequent merge with and redistribution by the master rank, it is not necessarily true that for all vertices *V* in the next set *N* are necessarily a subset of *M*. Although this may seem to imply that all ranks will inevitably reprocess all the same elements and result in the same caches, this is not the case due to the incrementally increasing cost threshold which limits which nodes are expanded during an iteration. A scheme is necessary that allows for new nodes to be cleared against the caches of other ranks. Absent this, there will be significant duplication of effort between the ranks. Theoretically, this could be accomplished by each rank passing its 'new' states around a ring of the ranks and have each rank remove any state that it has already checked during previous iterations. While this would essentially eliminate the redundant processing, it would also require an inordinate amount of communication, since each rank would need to communicate its data to all other ranks which would result in a total of $\frac{n\,(n-1)}{2}$ communication per iteration of a rapidly increasing cache size. Clearly this is impractical, as even with a distributed computing environment the benefits would be lost to communication times rather than actual processing; this does not even mention the additional communication complexity it could require to have all nodes communicate to all others in a circular manner. Overall, a cache clearing mechanism is not only useful, but practically necessary for the sake of efficiency. Because of these additional complexities, the approach developed here is to check the incoming vertex data against the existing cache data and only insert it for

processing if it has either never been seen or has a lower G value than what was last recorded for that state. Using this strategy is simple to implement and resembles the same check that is used in the core algorithm for deciding if an adjacency should be placed in the Now list. Although this does not prevent redundant data from necessarily being communicated, it will not be reprocessed by any rank more than once and has the benefit of not requiring any additional communications. Additionally, since the check is performed by the workers and the master when gathering the Later lists, this further reduces the chance for excessive duplicated processing.

In his original work on DFS, Brand presents the concept of 'cost relaxation' as a means to allow the search to consume the space faster with the tradeoff being that there may be more extraneous processing and potentially suboptimal points up to the difference between the optimal and the relaxation amount [5]. Brand sets the cost threshold, which determines if a vertex is to be expanded or deferred to the Later list, to be the minimum between the two values of the current threshold and the local minimum of the elements in the Now list plus some relaxation constant. By doing this, the actual threshold value is adjusted if a new local 'minimum' is found after an iteration. It seems to suggest that the cost relaxation could have a cumulative impact on the threshold value during the course of the search. To avoid this potential issue, this approach decided to only allow cost relaxation to be factored in during the actual comparison to decide if a vertex is expanded or placed in the Later list. In doing so, all threshold values are the same as they would be computed in FS or DFS without the cost relaxation factor. Although this factor was added in during development for potential inclusion, it was not tested during the course of this research. However, a small relaxation value of .01 was used to avoid floating point errors

that were observed during some preliminary testing. Technically this is a relaxation value, but because it is smaller than the minimum possible traversal cost between any two nodes, it does not actually modify how the search will expand through the space.

HDFS Development

The HDFS algorithm was designed as a variation of the DCFS algorithm using similar MPI constructs but with an altered communication structure with the intent of attempting to address the load balancing issues with respect to the cache. This algorithm borrows from the concepts proposed by Zho and Hansen in their work on Structured Duplicate Detection (SDD) as a means of representing the problem space abstractly and then using it to distribute the space [2]. These distributions of the space are based on what SDD refers to as the 'images' of the space which are said to be the 'projection' of an individual state; that is, that many states project to a single image. HDFS differs from DCFS in its communication structure in that instead of collecting, combining, and then redistributing the Later list after each distribution of work, each state is associated with a projection that determines which rank will handle it. In practice, this involves 'hashing' of the coordinates of a vertex within the space; it is worth noting that the use of this type of space distribution is novel in that it has not been combined with any DFS version publically available at the time of this writing. By doing this, each processor becomes responsible for an image which includes all of the vertices which project or 'hash' to it. In the HDFS algorithm, each of these smaller spaces become the responsibility of an individual rank. That is, a rank will only process data related to states in its image of responsibility. By doing this, a single rank only has the memory requirement of the states in its own space and will not have to retain states expanded in other areas of the space.

Although the search may expand in such a way that one rank receives a vertex from another rank multiple times, each rank will necessarily only store one copy of such vertex. Because of this, the maximum cache size for any rank can be expressed as $C = \frac{N}{P}$ where C is the cache size, N is the total number of vertices in the space and P is the number of processors applied to the problem. An additional benefit of this approach is that it does not require the additional cost of custom comparisons to insert vertices in the hash map. In order to communicate this state information to the appropriate ranks, when a rank expands a state it will determine the projection of each adjacency which identifies the corresponding rank to which the state belongs. If the projection of a vertex is a different image than the current rank, it is queued for sending at the end of the current Now list iteration. This allows for a less centralized system of communication as compared to DCFS, since each rank is determining which other ranks to communicate to according how the search expands through the space and which ranks are responsible for those sets of states. By using such abstraction approaches, a distributed system is able to scatter amounts of data in such a way that the various processing elements can search the space within their assigned data region with minimal request for data that resides local to another processing element and without the barrier of more frequent collective communications. Other approaches that actually distribute the space for processing may sacrifice efficiency for a potential loss in optimality due to the separate processes not communicating relevant cost data amongst themselves [1, 4, 6]. However, because HDFS is fundamentally based on FS, the loss of optimality is avoided despite the distributions due to the communication of the cost threshold during each iteration to all workers. Because of this, each rank will only work on processing data that is less than the

minimally increased cost threshold for each iteration. HDFS borrows from the most fundamental part of the space abstraction concept and works by assigning each processing rank one or more images that it then becomes responsible for. In this way, each rank has a constant bound on the amount of nodes that it must process and store in memory.

When a rank begins to process a vertex, if a vertex is expanded and one or more of its adjacent states are not strictly within its image, they are sent to the rank who own that image. Internally this is maintained by simply adding the external vertex to a vector to be sent to its appropriate processor after the current iteration completes; this is drastically simpler than the DCFS methods which requires transformation of nodes from hash maps to lists and then a more complicated communication structure to actually transmit the data. A benefit of the abstracted communication approach is that it ensures that no single rank will ever have more than a worst case number of nodes in its cache. Contrasted with DCFS, in which the master is responsible for much of the work and coordinating the communication, HDFS allows each rank to process its own data and only communicate to others when it needs to send or receive data from them. To determine the image projection of a state, a 'hash' is used based on its position within the grid. For grid based problems, the hashing is conceptually simple to visualize in the most intuitive way of breaking a rectangular space into smaller rectangles of equal size. While the hashing approach would maintains an even distribution of closed list nodes, it would necessarily mean that some ranks would sit idle until the fringe expanded into its sector and for some paths a rank may never actually work. A possible improvement on this

would be to nest the hashing approach resulting in a somewhat checkerboard distribution of the space (see Figure 6).

1	2
3	4

1	2	1	2
3	4	3	4
1	2	1	2
3	4	3	4

Figure 6. Hashing and nested hashing

In the figures above, the number represents which rank would be responsible for each portion of the overall space. The first image would work as already described. The second image allows each rank to have the same total area of coverage by 'nesting' the hash distribution inside the subspace, while spreading it out so that the ranks are more likely to receive work as the fringe expands. The side effect of this is that although the area is the same, the perimeter of each zone has increased which will imply more communication between ranks. Although there may be more communications, this may be offset by the fact that more ranks doing work sooner, resulting in better utilization. Nesting was tested at level one and two for each processor count which is visually represented in Figure 6. A minor adjustment was made for the level two nesting with 16 processors because the test problem dimensions do not evenly divide by 16 twice in order to produce even sets of images. To circumvent this issue, that particular set of tests was altered to perceive the space as a 1024x1024 space which is appropriately divisible by 16 at two levels for even sets of image sizes. However, the problems themselves were still the standard 1000x1000 problems that were tested on the other algorithms and the end

point remained the same. Because of this, the behavior of the search is preserved with the only technical issue being the periphery of some nodes would be able to map vertices that do not exist in the problem to some images. A visual representation of this is shown in Figure 7. In that figure, the light portion represents the complete space with the two darkest portions being where the beginning and end point exist, just as with the other problems and the lightly shaded region on the border represents what is technically part of the image but not actually represented in the space. By using this alteration, even though all ranks do not have the same share of the actual space, each rank will still process only parts of the usable space.

Figure 7. HDFS nested hashing for 16 processors

Initially, the HDFS communication scheme was designed to be asynchronous in an attempt to take better advantage of the decentralized structure. Although such a

scheme is not impossible, it became significantly more complex to implement due to avoiding communication blocking. In that approach, each rank would record a queue of items to be sent to each rank and would use a probe to test for incoming messages from other ranks. The problem with the probing technique is that, as the number of ranks increases, it will be more likely that one rank will probe just before another rank posts the message which will cause the posting rank to block until that message is probed for again and the communication can be completed. While there are non-blocking message passing options, they still require that the sending or receiving buffer be maintained until the communication is completed. Because of this, it would be very problematic to move things into a software level buffer that then later re-initiates the communication. The only potential benefit of asynchronous communication would be a rank could theoretically keep searching without being interrupted to check for messages that may not be waiting and instead wait for a probe to be signaled. However, even when the probe was signaled, it would still require some synchronization between ranks prior to communicating. While such communications may be possible, it would require some level of agreement between the ranks that they were about to communicate. Such a complex communication scheme may be possible, however it is not necessarily central to the HDFS algorithm in itself and could be a potential area of future work. Moreover, the potential benefit of some ranks to continue processing with only occasionally interrupting to probe is arguably outweighed by the complexity required to enable any rank to communicate with any other rank at any time; additionally this still runs the risk that the rank may then lose any former efficiency gains by waiting for the corresponding rank to probe and begin communicating. The revised approach which was developed here retains the decentralized nature for

communication between the ranks but is synchronized to allow ranks to determine when they should send and/or receive from other ranks. To accomplish this, all ranks now participate in a single 'all to all' collective communication in which each rank sends a count of how many vertices each other ranks should expect from it. From there, each rank knows how many items it needs to send to each other rank and how many it should expect from each rank. Using this data, each rank begins a non-blocking send of the ranks it has data for. It is critical that the sends be non-blocking so that the ranks can send all data necessary before posting their blocking receives. If the sends were blocking, it would be possible for instance that rank 0 attempts to send data to rank 1 and rank 1 attempts to send data to rank 0 and both are waiting for each other to post the corresponding receive, causing a communication deadlock. Similarly, it is necessary to perform the sending before receiving so that all data is 'sent' before a rank can receive. Because the non-blocking sends may not be immediately 'sent' it is possible that they are buffered at some level by MPI until the communication channel can complete the transmission to the receiving rank. To account for this, it is necessary for the sending rank to perform a 'wait' on the status variables associated with each send. This allows the rank to ensure that the data has been received and the send buffer can be cleared in preparation for the next iteration of processing. The wait could potentially slow a rank down if there was a particularly large send/receive operations between ranks. However, this is minimized by the fact that all ranks are synchronized directly by participating in the collective all-to-all just before the sending. Also, if a rank did not send or receive any information, it can continue processing and will pause at the next communication.

CHAPTER IV

RESULTS

The results of this research review are significant in its adaptation of the Distributed Fringe Search for a distributed computing environment as well as its application of Structured Duplicate Detection concepts for developing a means to distribute a problem space. This research has built upon prior research, both by the author and external, and has developed approaches for applying distributed computing systems to large pathfinding problems. These approaches prove fairly adaptable to solving such problems within a reasonable time and memory environment that may be used on such large problems. Results in this section will be examined for each of the three algorithms implemented, namely Fringe Search, Distributed Computing Fringe Search, and Hash Distributed Fringe Search. Of these, the latter two are original to this work and the former is used as a serial algorithm for base comparison. Descriptions of the problems, testing environment, and specific design decisions used in the implementation of the tested algorithms can be found in the previous chapter. It is worth noting that some of the results here may have potentially been impacted by other computational processes running outside the control of the research conducted here. Because all tests were conducted on a cluster owned and operated by Middle Tennessee State University, other researches may have had long running programs that have slightly delayed the processing due to system level task switching. Although it is possible that some of the system usage factors delayed some tests, the actually data processed during the run would not be altered and therefore is still valid for reporting here. Linear regression analysis is provided for most figures presented here and include an R^2 value where applicable.

Fringe Search

The FS algorithm was tested on all 300 randomly generated test problems and data was gathered for statistical purposes, especially for comparison to the distributed algorithms. This data was then compiled and analyzed in order to assess any observable trends. These trends could then be used to contrast with the distributed algorithms to provide a more in depth explanation for them. Statistical data for the FS tests are summarized in Table 1.

Table 1. Fringe Search statistics

	Pass			**Fail**		
	Cache	*Iterations*	*Time*	*Cache*	*Iterations*	*Time*
G2C100	104787	2776	9.19	737933	40003	110.20
G11C50	168466	4374	15.74	393944	24796	62.82
G20C25	98088	1701	8.30	667826	35389	102.14
Average	123780	2950	11.08	599901	33396	91.72

Of all the data collected, the cache size with respect to time is of key importance, since it is what requires the most memory to maintain for very large searches. For most statistics reported here, the focus will be on the successful trials since they are of more interest; some statistics on the failed trials will be included for completeness. In terms of simple averages, the FS algorithm as implemented performed well on the tested problems, with an average solve time of 11.08 seconds for successful tests and over eight times longer at 91.72 seconds for unsuccessful ones. In terms of cache size, successful tests retained an average of 123,780 vertices in its cache at the end of the search; this constitutes approximately 12% of the total search space. For unsuccessful tests, the

average number retained was over four times more with 599,901 vertices in cache. The FS algorithm itself had to iterate, that is it expanded until no vertices were available under the current cost threshold, an average of 2,950 times for successful tests and nearly 12 times more for unsuccessful tests at 33,396 iterations. The data represent a clear cost of expanding through a large complicated space, especially for the failed tests which necessarily expanded more nodes. It should be noted that some tests failed relatively quickly due to a blockage near the beginning point while in others the blockage was closer to the end point. In the latter case, this leaves much more of the space available for the search to process while essentially looking for alternate possible routes. Because of this, some of the shorter failed attempts expanded considerably less nodes than most. Additionally, due to the Gaussian distribution of the centers, the periphery of those obstacles is naturally porous like, which complicates the search and results in significantly more iterations; this is likely the cause of the dramatically increased number of iterations for the failed tests. The nature of what causes difficulty can be observed upon inspection of some of the select problem types tested. For the problems with 11 Gaussian centers with a width value of 50 (G11C50), the problems took longer on average than the other two problem types. This is explainable by the fact that those problems just happened to have more ‘complexity’ due to the high count of midsized obstacles. Evidently from this data, the problems with two large centers were actually relatively easier than a space with many small obstacles; this is not unrealistic given such a large problem space. For the G2 tests, the longest runtime was just under 50 seconds and retained just over 400,000 vertices after execution. The G11 tests took the longest overall with a worst case of nearly 60 seconds and retaining almost 500,000 vertices.

Fastest of the three types, the G20 tests took only over 30 seconds and retained less than 300,000 vertices at completion. While there is some slight difference between the individual problem types, the trend between processing time and the total number of vertices retained appears to be sub-linear in growth. This seems to suggest that FS is potentially a good candidate for large scale searches if it can be applied in a distributed fashion. The results for all successful tests are compared by problem type in Figure 8.

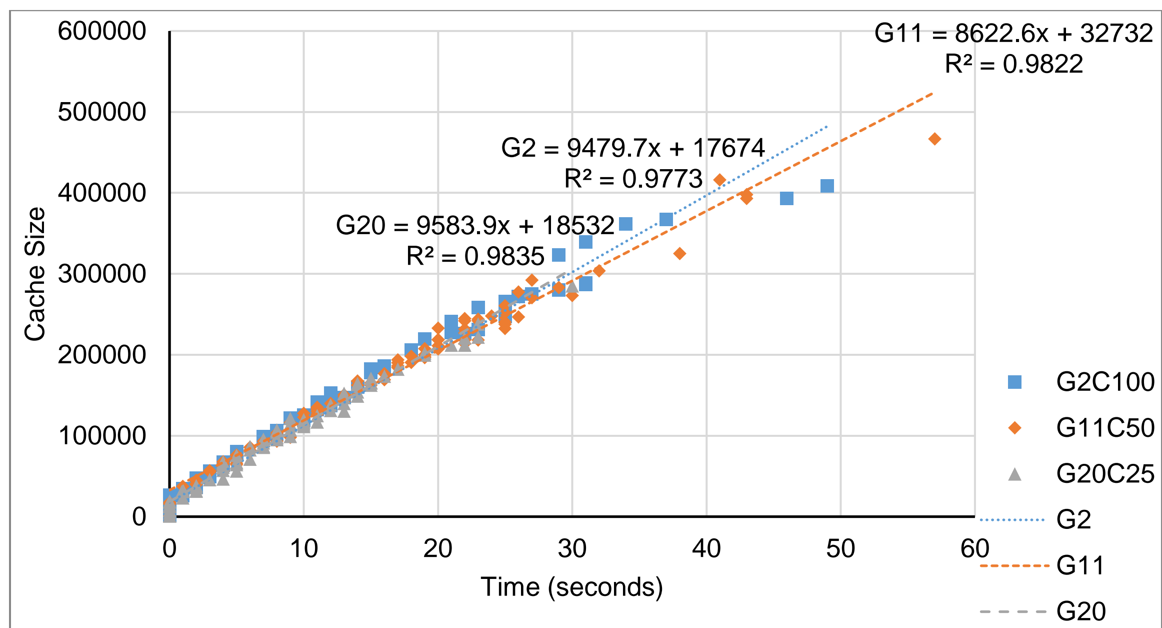

Figure 8. FS successful searches

One interesting point of this graph is that the cache size growth becomes slightly erratic after around the 300,000 vertex mark. A possible explanation is that by that point approximately a third of the space has been searched and the frontier of the searched space could extend through a large portion of the space. It is possible that this achieves some sort of 'critical mass' of the space at which fewer nodes overall are added to the

cache since so much of the space has already been searched. This is at least one potential explanation; further testing on larger problems could inspect if cache sizes began to increase slower after approximately one third of the space had been processed.

Distributed Computing Fringe Search

The DCFS was tested on all problem and specific data was tracked related to the timing of its communication components. This data showed an expected increase in the overall processing time compared to the serial version with the tradeoff being the benefit of less overall vertices retained in the cache for each processor. These increases for both successful and unsuccessful tests can be seen in Table 2.

Table 2. Distributed Computing Fringe Search statistics

	Pass				Fail			
	Cache	*Comm.*	*Idle*	*Time*	*Cache*	*Comm.*	*Idle*	*Time*
D4G2	40651.88	48.65	23.36	190.46	440309.73	589.36	1707.53	4286.50
D4G11	65258.43	87.19	32.75	323.57	245465.50	321.11	1179.30	2631.61
D4G20	37486.45	47.46	17.02	173.99	386809.57	464.94	1748.67	8175.48
D4 Avg.	47798.92	61.10	24.38	229.34	357528.27	458.47	1545.17	5031.19
D8G2	30557.49	30.37	18.09	114.47	401052.90	276.85	2699.76	4123.52
D8G11	47207.75	46.69	24.70	170.42	231905.42	159.38	1554.62	2283.17
D8G20	28074.36	26.00	12.56	87.57	345785.20	275.34	2786.74	3947.20
D8 Avg.	35279.87	34.35	18.45	124.16	326247.84	237.19	2347.04	3451.30
D16G2	27361.90	31.57	24.33	100.27	384760.85	276.30	3835.91	4281.31
D16G11	40639.93	47.83	33.18	156.35	232675.59	140.16	2306.65	2811.45
D16G20	25819.81	27.95	41.27	87.03	378623.27	160.95	4381.15	5447.96
D16 Avg.	31273.88	35.78	32.92	114.55	332019.90	192.47	3507.90	4180.24
Average	38117.56	43.74	25.25	156.02	338598.67	296.04	2466.70	4220.91

By utilizing the distribution of work between processors, the cache size was effectively reduced by almost 70% on average for successful trials. Associated with this better distribution of the cache is the cost of communication. In terms of overall time, the

DCFS took approximately 14 times longer for successful trials on average. Since FS maintains a single cache in memory throughout the search, its size can be considered as an upper bound of the minimum number of vertices that should be encountered in a given search. As discussed earlier however, DCFS suffers from vertex duplication due to the distributed nature of the processing with separate caches for each processor. So while each individual processor retains 3.2 times fewer vertices in cache compared to FS, the average of combined cache sizes is 2.6 times more than its serial counterpart. Absent duplication between processors, it would be expected that the same number of vertices would be processed for a given test with an even distribution of the cache space amongst the processors. That is to say that if the serial FS algorithm could perfectly distribute its cache during processing, each processor would receive an equal share. From this, the amount of increase in total cache size suggests that on average 51% of all the vertices are duplicated across the processors to some degree. In terms of relative problem difficulty, the results are consistent with the result from the FS trials in that the G20 problems were the easiest, with the G2 problems being just slightly harder and the G11 problems being the hardest. This may appear counterintuitive, however it can be explained by the relative density of the Gaussian points; although the number of center points is increased in the different problems, recall that the width of the problems is also decreasing which works to reduce the complexity of each point of clustered obstacles. The average cache sizes with respect to time are represented in Figure 9 for the 11 Gaussian problems. This visualizes the clear reduction in overall time with additional processors while maintaining a relatively balanced increase in the reduced cache size per processor. However, it is noted that there appears to be a diminishing return, since the reduction from using four to

eight is approximately 26% whereas the reduction from eight to 16 is only 12%. While the reduction in time is not significant when using 16 compared to only eight processors, only about 12 more nodes are required in memory per processor per second during the search as represented by their different linear regression slopes. Because this is not influenced by the means of the work distribution, i.e. whether it is being transmitted over a local bus or over a network, it is more likely due to relationship between the distribution threshold and the problems that are being used. For instance, it is possible that as the frontier of the search is broken up into smaller pieces, the issue of cache duplication is exacerbated since more processors have a higher chance of seeing unfamiliar vertices during the search. At its extreme, a hypothetical situation where one processor received one vertex for each iteration would likely need to retain many duplicate vertices since it is only seeing a small portion of the search space at a time.

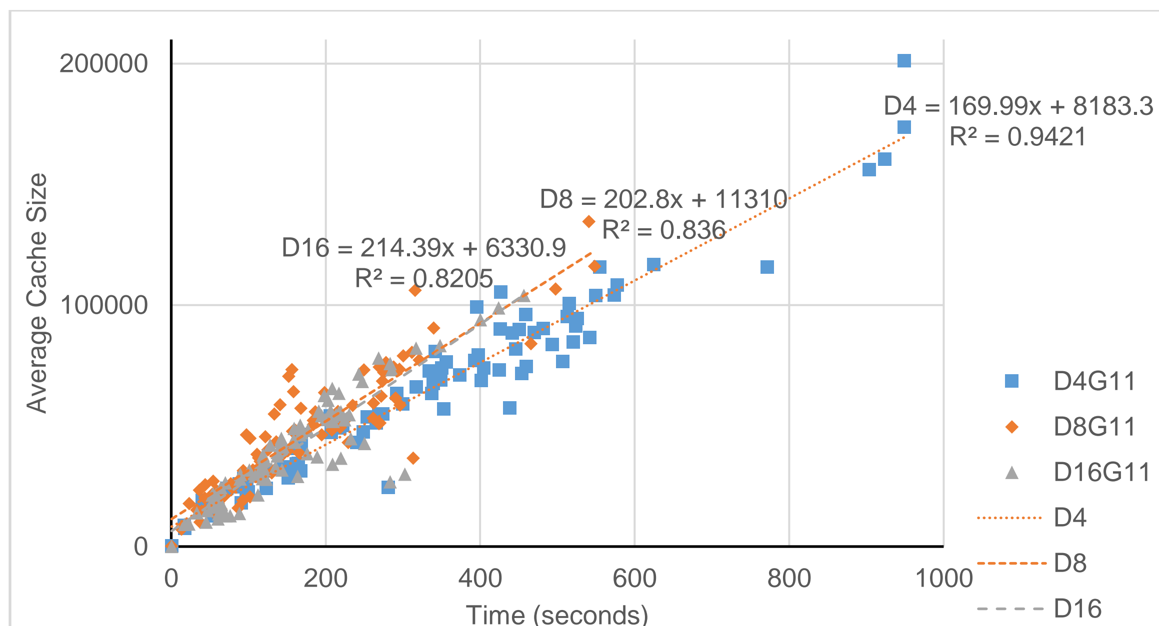

Figure 9. DCFS comparison for 11 Gaussians

The unsuccessful problems, while perhaps not as interesting, did show some unique trends. This includes the fact that DCFS was still able to maintain a reduction in the number of vertices cached by over 42% on average compared to serial FS. These data also showed an extremely high increase in processing times that went from approximately 8 times longer for failed FS test to about 27 times longer for failed DCFS tests. It is thought that this is possibly exacerbated by the 'porous' nature of the Gaussian problems which contain many small alternative paths which may complicate the search. Although the successful problem are of the primary interest, the unsuccessful problems can be used as an indicator to how these algorithms may perform on much larger search spaces. If a feasible search were to conducted through a space large enough to actually require the level of processing as seen for the unsuccessful trials here, it would take approximately just as long and would exhibit similar behavior. In this way, although the problems tested here only consisted of one million searchable nodes, the unsuccessful tests inform what a similar successful trial may perform like in a larger space than was actually tested here. With this generality in mind however, the unsuccessful results will not be examined in detail here.

Hash Distributed Fringe Search

The HDFS algorithm was tested under the same problems as the previous two algorithms. This set of tests is similar to DCFS with runs of 4, 8, and 16 processors used over the cluster. These tests also included an additional factor of 'nesting' level as described in the Methods chapter. Data collected for all problems is in Table 3.

Table 3. HDFS statistics with non-nested hashing

	Pass				Fail			
	Cache	*Comm.*	*ATA*	*Time*	*Cache*	*Comm.*	*ATA*	*Time*
H4G2	26331.27	5.19	5.01	8.17	184483.25	35.22	32.52	69.64
H4G11	42135.67	8.50	8.18	13.49	98486.02	21.50	19.93	40.78
H4G20	24556.17	4.40	4.24	7.15	166956.46	32.45	29.94	65.11
H4 Avg.	31007.70	6.03	5.81	9.61	149975.25	29.73	27.47	58.51
H8G2	13267.05	3.63	3.53	5.02	92241.63	27.73	26.11	42.50
H8G11	21081.53	5.72	5.55	8.03	49243.01	16.25	15.30	24.94
H8G20	12325.50	2.89	2.80	4.17	83478.23	23.68	22.22	37.70
H8 Avg.	15558.02	4.08	3.96	5.74	74987.62	22.55	21.21	35.04
H16G2	6152.98	3.18	3.13	4.13	46120.81	30.16	28.93	37.78
H16G11	10357.55	5.56	5.46	6.83	24621.51	17.97	17.27	22.18
H16G20	6150.12	3.22	3.18	3.86	41739.12	26.55	25.48	33.62
H16 Avg.	7553.55	3.99	3.92	4.94	37493.81	24.89	23.89	31.20
Average	18039.76	4.70	4.57	6.76	87485.56	25.72	24.19	41.58

Utilizing the distribution of vertices to other ranks based on a hash representation allowed for an effective processing of the search space. On average, each processor stored 85% less than the serial FS implementation tested here and 51% less than DCFS. Because each rank processes nodes only as the search extends into its portion of the space, it does not suffer from the multiple ranks having different versions of an expanded vertex in memory at once. In terms of communication costs, HDFS performed considerably better than DCFS, requiring 90% less time. Of the time spent communicating, 97% of it is the All to All (ATA) collective call that acts to synchronize all processors and causes them to intercommunicate as necessary. Remaining communication time is spent performing the send, receive, the additional wait to ensure that the sends have been received successfully. Despite the notion of a ‘wait’ seeming expensive due to the possibility of processor idle times, it accounted for less than 1% of the overall communication times. Overall timing actually showed a 37% decrease in time

over serial FS and a drastic 95% decrease over DCFS on average for all successful trials. The data for HDFS trials on the 11 Gaussian problems is presented in Figure 10 for a visual comparison of the average per processor cache sizes with respect to time.

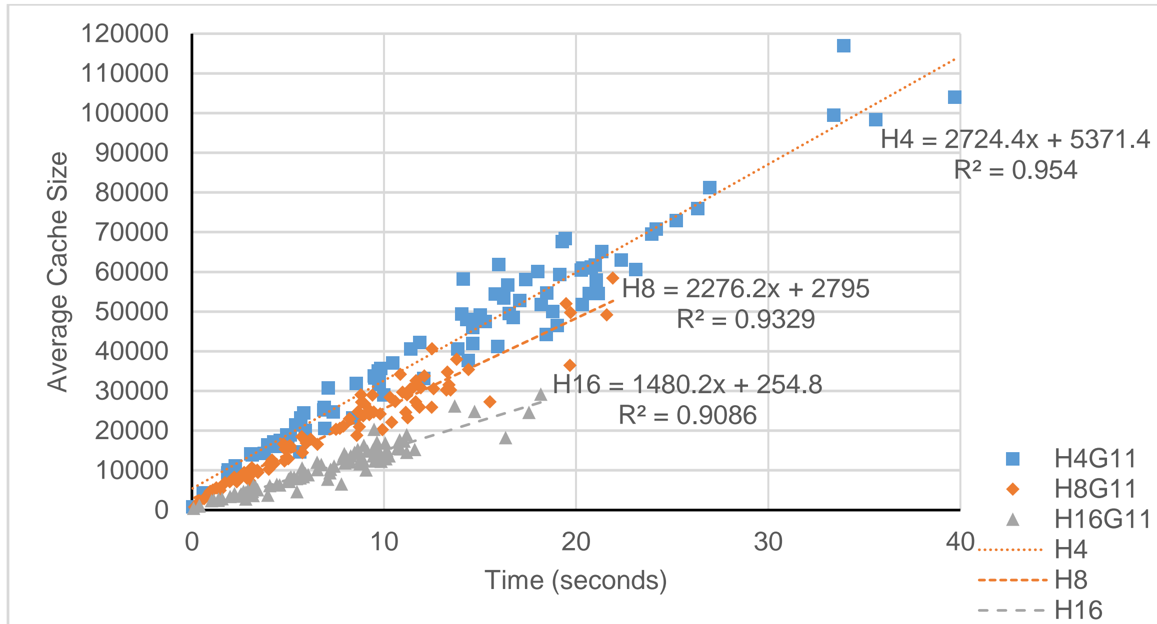

Figure 10. HDFS comparison for G11 tests with non-nested hashing

The previous results are all indicative of the overall performance of HDFS using a single level of ‘hashing’ the problem space. This provided one image for each rank that was arranged in rectangles covering the space. These images were then tested with a second layer of nesting in which N ranks have N images where the total area of the images is equal to what would have been provided in level one nesting. With only one layer of hashing, the image of a rank may be in a section of the space that the search frontier will expand little if any through. By performing this recursive nesting of the images, the intent was to allow each rank to have a portion of what would have been a

larger image. The same set of tests were run as with the non-nested hashing and the other algorithms with the data being listed in Table 4.

Table 4. HDFS statistics with nested hashing

	Pass				Fail			
	Cache	*Comm.*	*ATA*	*Time*	*Cache*	*Comm.*	*ATA*	*Time*
H4G2	26568.63	2.83	2.63	5.82	184483.25	33.49	30.61	68.84
H4G11	50905.05	6.37	5.86	9.91	98486.02	16.04	14.44	35.00
H4G20	24603.56	3.06	2.89	5.82	95402.19	26.76	24.17	58.72
H4 Avg.	34025.75	4.09	3.79	7.18	126123.82	25.43	23.07	54.19
H8G2	13408.88	1.82	1.69	3.18	92241.63	18.21	16.27	33.37
H8G11	21118.72	3.03	2.80	5.33	49243.01	10.07	8.94	18.55
H8G20	12353.58	1.51	1.41	2.77	83478.23	15.91	14.16	29.90
H8 Avg.	15627.06	2.12	1.97	3.76	74987.62	14.73	13.12	27.27
H16G2	6790.73	1.82	1.71	2.52	46120.81	19.80	17.94	27.18
H16G11	10574.84	2.92	2.72	4.09	24621.51	11.19	10.07	15.47
H16G20	6216.65	1.50	1.42	2.14	41739.12	17.52	15.77	24.72
H16 Avg.	7860.74	2.08	1.95	2.92	37493.81	16.17	14.59	22.46
Average	19171.18	2.76	2.57	4.62	79535.09	18.78	16.93	34.64

For these tests, cache sizes increased about 6% on average, representing more usage. Due to the increased total perimeter of images belonging to a rank, the expectation that the times would increase due to more frequent communication. Surprisingly however, times were instead decreased by approximately 41% compared to non-nested hashing. Overall times were reduced by just over 31% thanks mostly to the reduced communication times compared to HDFS without nested hashing. The data for HDFS tests with nested hashing are visually represented for comparison in Figure 11. Contrasted with Figure 10 which represents the same data but with non-nested hashing, some interesting features can be observed. Specifically, in examining the relative linear regression slops for the various plots, non-nested hashing shows a 16% reduction from

four to eight processors while nested hashing only shows a 5% reduction. On the other hand, going from eight to 16 processors with non-nested hashing shows only a 34% reduction while in the nested hashing tests it shows a 66% reduction; future research further explore this behavior.

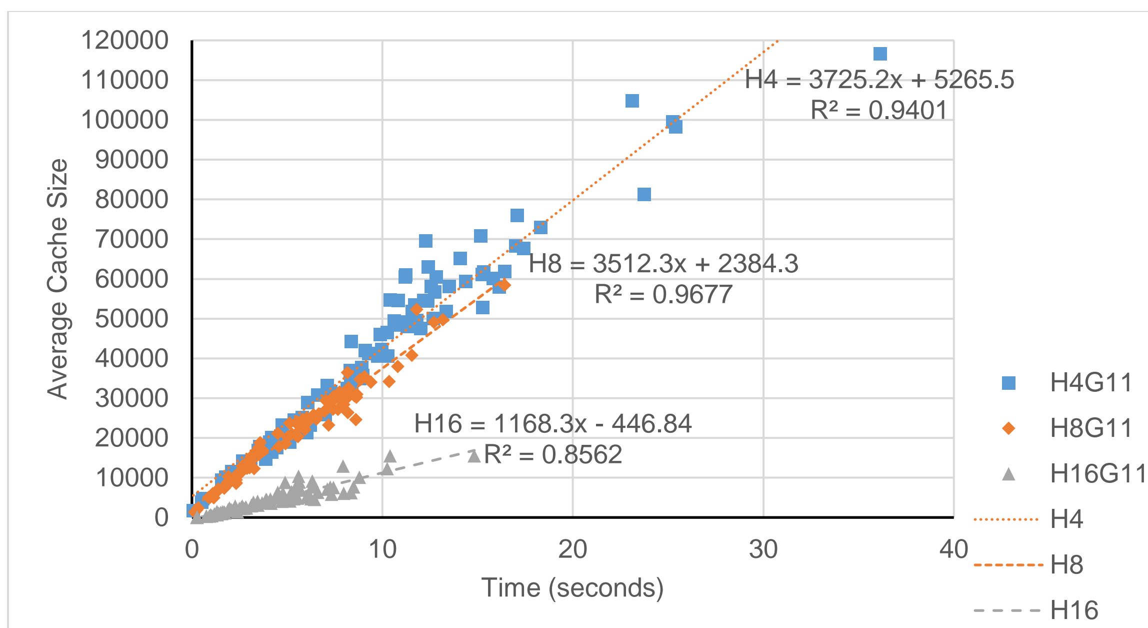

Figure 11. HDFS comparison for G11 tests with nested hashing

CHAPTER V

DISCUSSION

The algorithms developed through the course of this project are designed to provide approaches for distributed computing of pathfinding problems. This work has implemented a serial Fringe Search algorithm as well as proposing two new algorithms, namely the Distributed Computing Fringe Search (DCFS) and Hash Distributed Fringe Search (HDFS). These algorithms have been extensively tested and demonstrated an ability to reduce the burden of elements in the supporting cache list of the FS algorithm. By utilizing the applied approaches for distributing the cache and the processing of the search data, these algorithms provide a more suitable solution for solving large scale pathfinding problems that may be otherwise difficult or practically impossible for a SMP system. A consistent theme observable in the results presented here is the dichotomy between utilization of the allocated processor ranks and the intercommunication between them. Communication has been shown to be the most costly element of applying the FS algorithm to a distributed system. However, in order for the ranks to actually assist in the processing, communications of some kind must occur. If communication must occur, ideally it will only be a few small messages. In order to have good utilization however, it is desirable to send large amounts of data to other ranks so that it may be processed in parallel. DCFS is designed to operate along these lines, where work is sent once a large enough set has been generated to essentially ‘justify’ incurring the communication costs. Alternatively to sending a few large sets of data, many small sets of data could be sent so that each processor may stay relatively busy. In this approach, since the data is being sent more frequently, there might not be enough for each processor to work on; this is also

dependent on how the distribution of work is design. HDFS operates more similar to this approach, where the work is designed to be distributed to other processors based on where the vertex is represented in the space. By doing this, some processors may have little to know work the entire duration of the algorithm, meanwhile others may be working significantly because the fringe of the search expands more prevalently in their region of the space. Overall, the challenge is striking the balance between relatively minimal communication and good utilization of the available processors. A comparison of all three algorithms can be seen in Figure 12. By comparing the average performance on the hardest set of problems, the ones with 11 Gaussians (G11), the performance differences in terms of cache size required and time required become apparent. Contrasts between FS and DCFS clearly show a significant reduction in processor cache size, however at the cost of increased communication time. Contrasting FS with HDFS shows an even more significant reduction in cache sizes and without the communication costs of DCFS. The comparisons between DCFS and HDFS will be discussed in more detail in the following sections. Additionally, the HDFS section will explore the differences between HDFS with and without nested hashing.

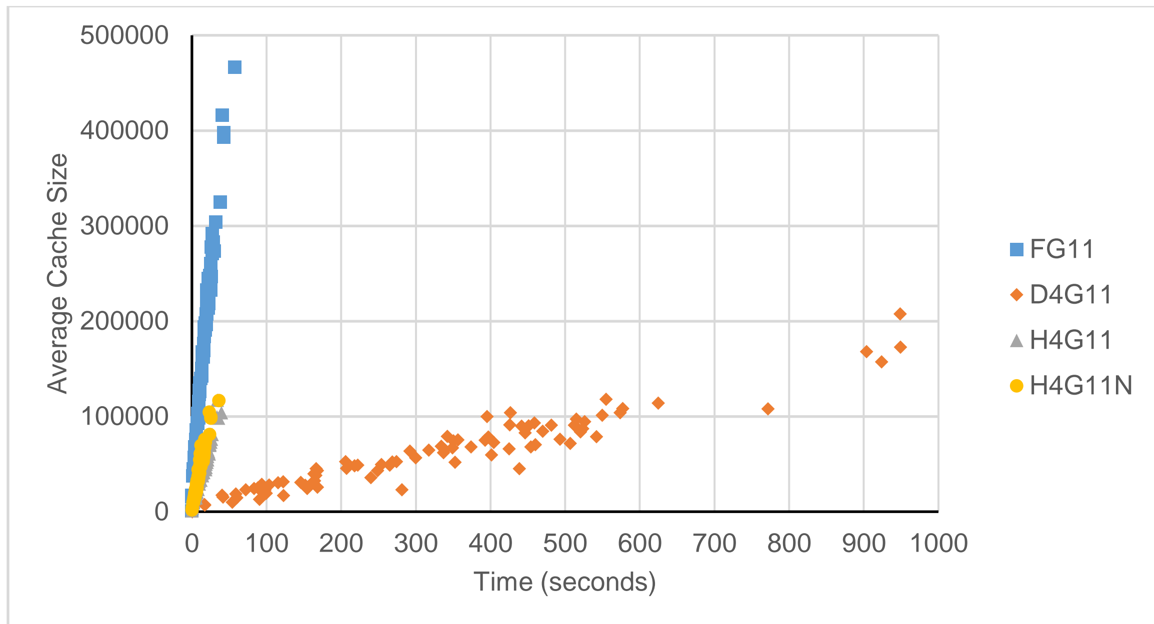

Figure 12. All algorithms compared for G11 tests

DCFS Review

The DCFS test results provide insight into on an extension of the original DFS designed for SMP systems to a distributed system. This implementation is a fairly direct translation, with the necessary adjustments made for the communication over a cluster. These adjustments however appear to be a significant source of the additional cost associated with the distribution. DCFS does succeed in reducing the amount nodes in cache for any single processor, therefore allowing for the potential of handling larger scale problem overall. However, it pays a high price for the distribution due to the complexities of coordinating and packaging the data in order to be sent via MPI. Additionally, due to the fixed threshold value, the master must perform some of work until the distribution threshold is reached. However, this prevents the costly distribution for trivial amounts of work (see Figure 13).

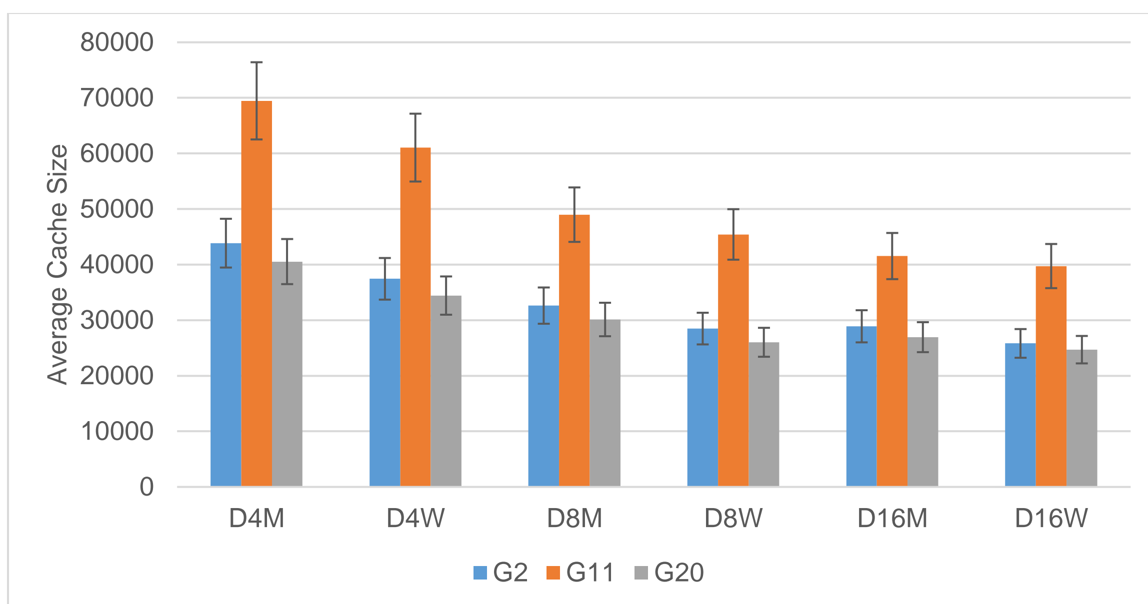

Figure 13. DCFS cache comparisons

Examining the communications in more detail shows where the time is actually being used. For the purposes of the data reported here, communication costs are considered any costs associated outside of the core search processing. Although this includes the 'transformation' time discussed in the Methods chapter, the timing for these components was less than expected, amounting to less than 1% of all communication time costs. Because of this insignificance to the time components, those data are not analyzed further here. Processors overall spend an average of approximately 28% of their time communicating necessary data amongst themselves. Idle time represents approximately 16% of the overall time. Because the master rank is constantly working, it is never truly idle in the sense that it has no work to be processing. It will only necessarily pause to conduct the scatters and gathers, however those components allow for more work to be performed and any momentary 'wait' time is associated with the

scatter or gather time component. Since the master is either processing or communicating, the idle times reported here are indicative of the time that the workers are waiting to receive work from the master. All workers will incur at least some idle time because of the application of a distribution threshold, i.e. the point at which the distributed communications take place. One question that is not addressed by the idle time data gathered here is what portion of that time is a constant overhead while waiting for the initial distribution threshold to be reached and what of it is from idle times between communications and processing. Of the time spent communicating to the workers, the overall averages showed that about 60% of the time was used during the scatter with the remainder being taken up by the gather process. Initially it might seem like the gathering process would be prone to take longer since the master must wait on all of the workers to complete their work and then participate in the communication. Although this can still be occurring, it seems that the fairly even distribution of work is allowing most processors to complete at roughly equal rates and does not leave other processes idle for significant amounts of time or leave the master waiting on the collective communication call. It is likely that the scattering process takes slightly longer because it is slightly more involved due to the requirements of determining how much work needs to be distributed and then packing that data to be sent. The comparison of communication time components can be seen in Figure 14. Note that time values reported are only simple averages and do not indicate the extent of overlap between waiting processes. Because of this, one process may only wait for a relatively short amount of time because it was the last to ‘arrive’ at the point of communication. On the other hand, another process may have reached the communication point much earlier and would incur

idle time while waiting for the other process(s) required for the communication. Additionally, some process may be the first to arrive after one iteration and the last to arrive at after another just depending on how their work is processed. Due to this potential overlapping of timing components, it would be incorrect to simply add the communication and idle times and suggest that the remainder of the time is exclusively being used in conducting the core FS algorithm.

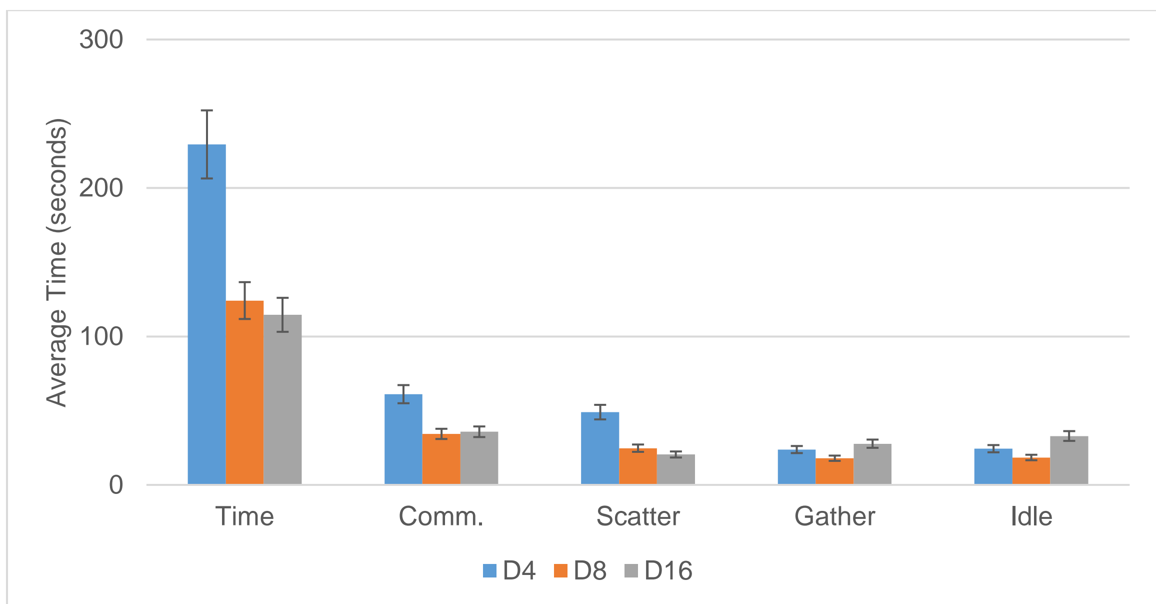

Figure 14. DCFS timing components

Although DCFS was slower than the serial version, a speedup is observed as more processors are applied to DCFS. Increasing the used processes from four to eight resulted in a 45% reduction in time which equates to an approximate 11% speedup. Going from eight to 16 still showed a reduced processing time overall, with a reduction of about 8% in time compared to using eight processors, although this only represents only 1%

speedup. However, this is to be somewhat expectable given the configuration of the testing environment. Because MPI was configured to only allow eight processors per machine, moving from eight to 16 necessarily causes a jump to separate machine for the other processors. Due to this, some of the communication must then happen over a network which will be considerably slower than even using MPI to communicate between processors on the same local machine. A significant part of this is due to the two stage communications that are currently required. Both stages are required since the first informs each rank of how much it is about to send in order for the correct buffer space to be allocated and the second stage actually performs the exchange of data. A bottleneck also exists at this stage since all communications are routed through the master rank. Centralizing the communication scheme around the master is somewhat useful since it allows for the search to begin and continue processing until enough work is encountered to justify performing the more expensive communications. While a decentralized approach may be theoretically possible, it is would likely require even more communication between the processes in order for each processor to stay synchronized during the search and for work to be distributed fairly. A more realistic option that may prove beneficial is the use of an independent communication hub that is responsible for only the packaging and distribution of work after each iteration. Due to the fact that DCFS spends less time on gathers than scatters, this suggests that the work distribution of work is fair enough to where processors are not typically spending much time on 'hold' waiting for a gather to occur. However, the master is still responsible for its own share of work in addition to the packaging of the data to be sent to the other workers. If a separate rank were responsible for the communication of work, it would free the former master

and allow it operate simply as another worker. Doing this would require no additional communication calls, it would simply be directed at a different centralized point. Related to the issue of work distribution is the item of selecting an appropriate threshold. Results show that the threshold allowed for work to be distributed in such a way that all workers stayed relatively busy. However, because the tests conducted here used a fixed threshold, further testing of DCFS distribution threshold would be useful in determining a value that potentially strikes a better balance between communication costs and idle times. Ideally, a relationship could be derived between the size of the space and the number of processors to be used on the problem; information on the approximate density of the problem may also prove useful, however this would likely be difficult to obtain for many practical real life problems.

The communication complications are a significant factor in the time costs of DCFS, however any corrections to those alone will not address the cache clearing issue. This issue is apparent in the data since the combined size of cache from each processor is considerably larger than the total cache size of the serial FS algorithm; although this is somewhat more acceptable due to a distributed environment, improvements in this area could greatly extend the practicality of the algorithm overall. These improvements to the cache clearing strategy may also reduce the communication costs since less duplicated vertices are being communicated during the search. However, while it may be possible to devise other means for cache clearing between the ranks, it is likely to require a non-trivial amount of coordinated communication, at which point the benefit of not simply recollecting the data may be lost. One possible approach to simply keeping the overall cache sizes at each rank reasonable would be to redistribute nodes from the cache to other

ranks with smaller cache sizes; this is somewhat similar to how the fringe is distributed at each iteration. Although this would require some mechanism for polling of the cache size to know how to 'fairly' balance an oversized cache, this could be included in some of the existing communications relatively simply. This strategy also could be slightly enhanced by associating each state with some sort of a 'hit' counter that would be relatively cheap to maintain. In doing this, the rank could have a better idea of which states are least recently used and decide to distribute those out to save space, since the more recently used states are more likely to be revisited and should remain with the original rank. The communication involved with such an approach is less than a 'ring' style approach since it only needs to happen on an as needed basis if the cache of a rank becomes too large. However, it would still require the expensive collective communications once the cache thresholds begin to reach a level to where they need to be distributed throughout the system. Additionally, some coordination would likely be necessary which may place more work on the master as each rank attempts to balance its cache and must have a way to decide which rank it should balance with; the use of an independent coordination point as discussed above may also assist with such a scheme.

While cache clearing may assist in reducing duplicative communication and subsequent processing, a true solution would be to address the very nature of how the work is distributed to the workers. Because the distribution is currently 'blind' which entails that the master has no intuition as to which other processors may or may not have seen each vertex that is about to be distributed. A primitive solution would be to have the master record some level of vertex routing information as it encounters then distributes states to other ranks. Although this would be fairly simple to implement, it would also

require additional processing and storage of this routing style information. A question for further research into such a technique would be if these additional costs were less extensive than the current ones associated without such information available to the master. Instead of recording where data has already been sent, the other option is to predetermine which ranks should process which vertices. At least one form of this is HDFS since it ensures only one rank is responsible for any given vertex in the space; in fact it was partially an early realization that such a solution would prevent duplication which motivated the design of HDFS.

Another possible improvement to DCFS would be to have each of the workers receive its initial load and then begin working with it and only communicate back to the master if it runs out of work. Although this would require less frequent communications of collecting and then redistributing the data to be worked with, it is likely to produce much redundancy since there is no real opportunity for the cache clearing strategy to be in place between the communications. Additionally, the iterations of the FS algorithm would then no longer be synchronized since the minimum cost increase would no longer be communicated between separate processes; the result of this is that every processors were 'behind' in the search would likely only examine nodes that were already examined by other processors which had already increased their local cost thresholds. In order to counter this, it would be necessary for the processor to know not only which vertices had been distributed, but also have some idea of their approximate region in order to avoid processing in that part of the search space.

HDFS Review

The HDFS test results show a fairly successful approach for adapting the FS algorithm to a distributed computing environment. This work has presented data which demonstrates the ability of HDFS to significantly reduce the average space required for each processor to maintain its cache. These processors are each responsible for a discrete portion of the search space, and therefore only require as much space for its portion of the total space. Contrasted with DCFS, HDFS has significantly less communication overhead, thanks in part due to its decentralized nature. Additionally, the design of allowing each rank to perform its non-blocking sends prior to receiving its data proves to be an efficient approach to transferring the data between ranks. Because each rank 'sends' all of its data through non-blocking methods and subsequently begins to receive its any new data for itself form other ranks, most ranks are able to begin receiving data almost immediately after transmitting any data to other ranks. So while the use of non-blocking sends does necessitate a following 'wait' to ensure successful transmission, the timing results show that this was an insignificant amount of the communication time. Because the amount of data being sent after any given iteration will be small, it will require very little transmission time. Also, the wait times are able to remain low since data is only transmitted to other ranks if the search expands through the area belonging to another rank. While this approach appears to scale well for the varying levels of complexity of problems tested here, future research may wish to explore larger problem spaces to confirm such scalability. The timing components are visually represented in Figure 15 and show that the ATA uses the majority of the communication time. This

implies that any possible improvements to this would likely see a reduction in overall processing time due to the reduced communications between the ranks during the ATA.

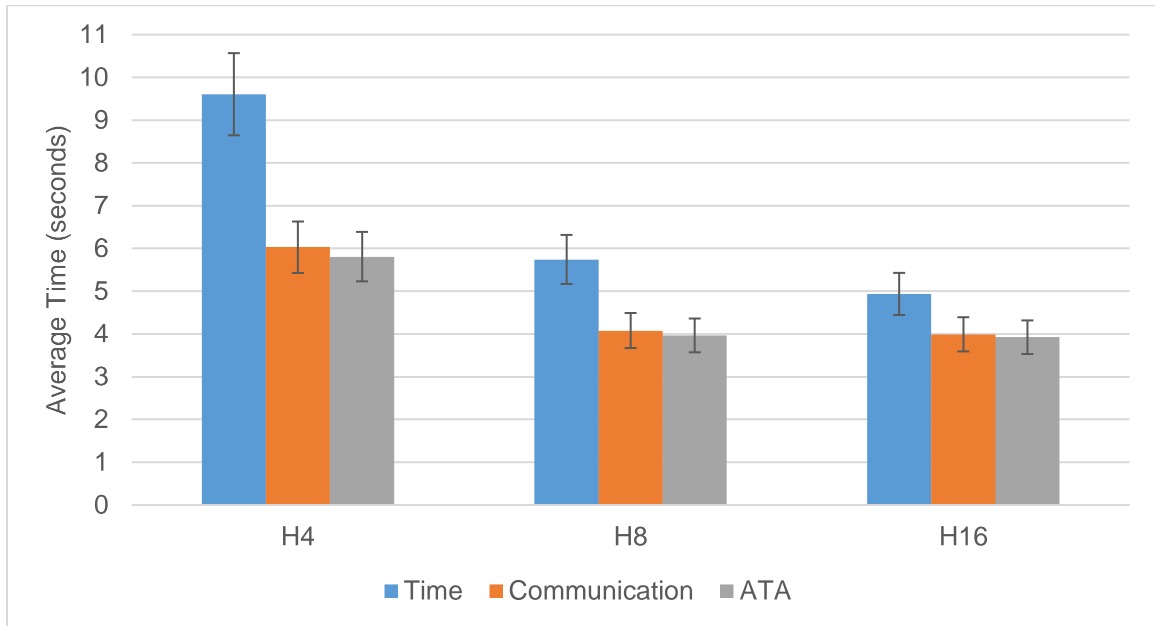

Figure 15. HDFS timing components

At least one possible means of reducing the ATA timing component has been demonstrated by this research through the use of nested hashing. By default, only one ‘level’ of hashing is performed which is over the entire initial search space. Considering each of the original images that are generated as a sub-space to be further dividing, a second ‘level’ of hashing can be computed recursively, thereby adding a nested hashing distribution of the abstract images of the overall search space. Distributing the space in this nested hashing does increase the total perimeter of all the images owned by a rank, however the speedup suggests that this is not causing a corresponding increase in communication (see Figure 16). Explanation for this may be because the vertices on a

given border at a time is decreased, resulting in smaller amount of data communicated between each ranks; further testing is required to investigate this possibility.

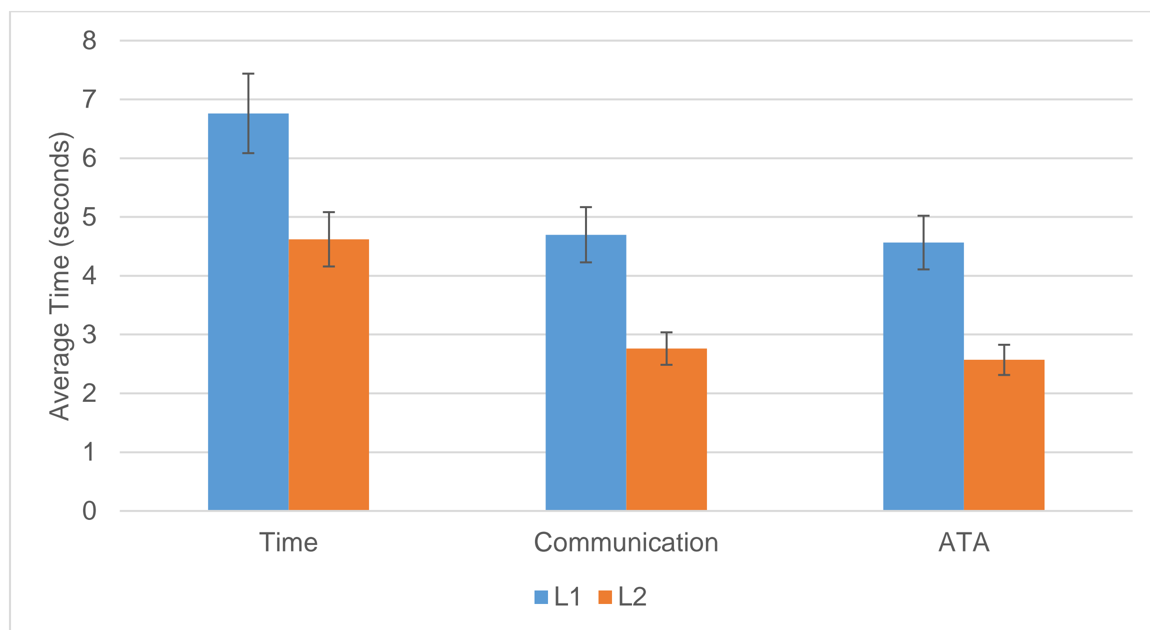

Figure 16. HDFS level one and two nesting time components

The issue of utilization is visually represented in Figure 17 which comparison the HDFS performance on the 11 Gaussian problems with and without nested hashing. This figure, in combination with the result data reported in Tables 3 and 4, show an interesting contrast in the speedups when additional processors are applied to the problems. DCFS showed time reductions of 45% and 8% when moving from four to eight and from eight to 16 processors respectively. HDFS without nested hashing showed reductions for 40% and 13% for these same measurements while HDFS with nested hashing demonstrated reductions of 47% and 22% respectively. These results show that HDFS without nested hashing may have had better times than DCFS but it showed slightly worse reductions

when applying more processors; the slight increase when moving from eight to 16 processors is negligible considering that DCFS also contended with a significant increase in communication costs for 16 processor tests. The decrease in relative decreases in time is explainable by a poor utilization, where HDFS without hashing may create images that are responsible for portion of the space that are never searched which leaves some of the available ranks under-utilized. This is further corroborated by the fact that enabling the nested hashing significantly improved the reductions in time as more processors were applied to the problems; by breaking the images of the space into sub-images assigned to the available ranks, it was more likely that the ranks would cover part of the space that was used during the search. These increased utilization are then nicely complemented by the decentralized and synchronous nature of the HDFS communication design which structures the transfer of data to require significantly less time than the DCFS approach.

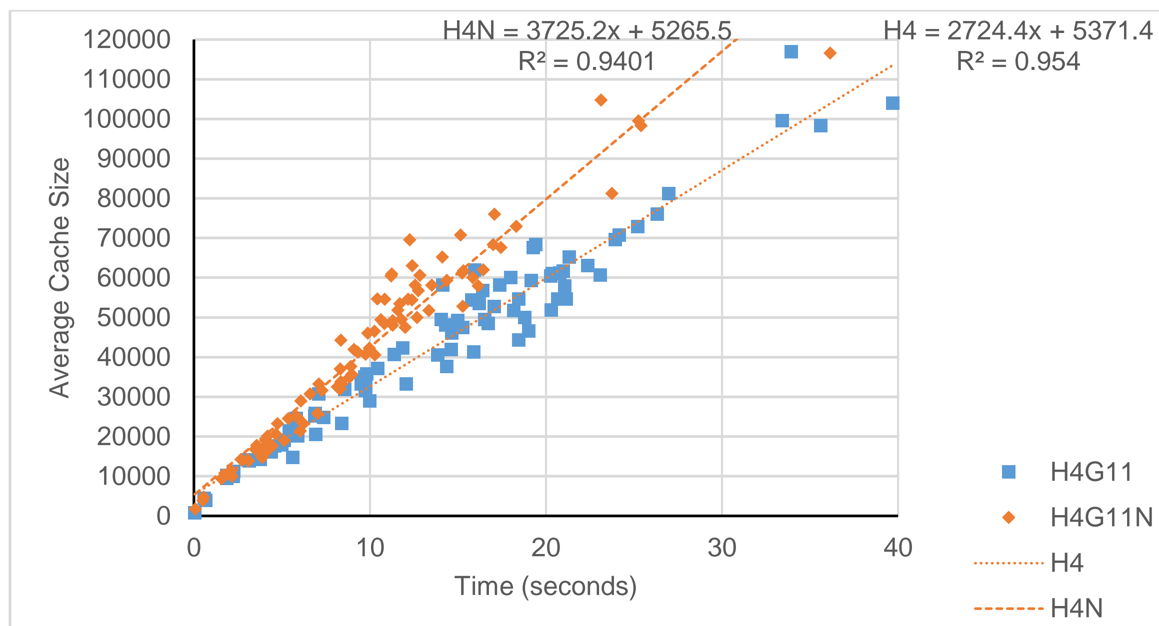

Figure 17. Comparison of HDFS for the G11 tests with and without nesting

Although the HDFS algorithm performs rather admirably on the problems tested here, there is almost always room for improvement. One potential option to improve on the work utilization of HDFS is to somewhat borrow from the more direct work distribution of regular DFS. In this approach, the fringe could be distributed as in the basic DFS, however the cache division would be based on the hashing approach. This would mean that other ranks would help process each vertex in their share of the fringe and then report its results back to the ranks which are responsible for the respective states. Although this would allow for better utilization of the ranks than the regular hashing approach, the benefit may be minimal due to the fairly cheap cost of expanding a vertex versus the costs of packaging everything for communication. In order for this to actually be worth the overhead costs, the expansion of a vertex would likely have to be time intensive and/or the problem would need to generate a large volume of nodes at each iteration to justify the distribution.

One potential pathological problem that may have unusually high communication costs using HDFS is if a search path were to follow along a boundary between two images; this is somewhat of an edge case and not one tested in this study. In this case, each rank could expand vertices that belong to the neighboring image and rank and then queue those elements to be sent. Although the receiving rank in this situation will clear them against its local authoritative cache, it is still redundant communication. A strategy that may reduce this is a temporary set of only the state names that have been recently sent that could then be cleared at some decay rate.

Conclusion

The initial goal of this efforts conducted here were to move towards finding a solution to using distributed computing resources on large scale pathfinding problems. This research has expanded some potential opportunities for further exploration as well as provided insight into why such approaches may or may not be successful. Two main approaches were employed here, namely the use simple fringe distribution over a cluster and using a space abstraction as means of distributing the space over a cluster. Results from the testing are promising in that they show a viable means of reducing the cache size of a FS algorithm. DCFS provides good resources utilization at each step by ensuring that all ranks have data to process. The issue here is that due to the expensive communications and vertex duplication amongst the separate processor caches, not all of that work is productive toward advancing the search. Although HDFS has less communication overhead compared to DCFS, it also has less utilization of the processors since it is not concerned with ensuring all ranks are busy during each iteration. But it is actually because of this that it allows the search to proceed through the space as it would in a serial implementation and then only expand once it reaches the boundary of the a given image; these image boundaries are established at run time based on the number of processors available and the desired nesting level. Ultimately the data show that the approach taken by HDFS was largely more successful, even if it means that the cluster resources are underutilized. While the results are mostly positive, further review is appropriate to assess these algorithms against other metrics that were not considered here. Extensions of this work may also include application to larger pathfinding problems or more nuanced problem including non-geometric domains such as puzzle style problems.

Future Work

The research conducted here has provided a fairly extensive study of Fringe Search as well as demonstrated the strengths and weaknesses of two new pathfinding algorithms developed here, namely Distributed Computing Fringe Search and Hash Distributed Fringe Search. This research can assist further exploration of the subject by serving as a reference in developing pathfinding algorithms with similar behavior that is desirable to be distributed over a cluster environment. These potential extensions of this work may benefit from the methods used by this work to reduce a cache size of a single processor and/or the methods used for communicating such data over a distributed system. Besides future external work, the algorithms developed here provide a basis for many interesting and potentially successful enhancements. Experimentation with the framework of tests here would also allow for equitable testing between the new enhancements and the algorithms as they were developed originally.

An aspect that was not tested here due time and scope limitations was that of cost relaxation as proposed by Brand [5]. His implementation of DFS was tested with constant cost relaxation values. Cost relaxation allows the search to consume the space quicker because more nodes are considered under the threshold for expansion; refer to the Methods chapter for a more detailed discussion of this behavior. Although cost relaxation allows for less optimal paths to be acquired, it may be appropriate for large scale problems that can allow for sub-optimal solutions. Somewhat more interesting but also less clear would be the use of a cost relaxation to a more dynamic basis that would allow the frontier of the search to expand faster in some direction than others. Such an application of cost relaxation could allow for more effective processing of the space in

such a way that one rank could elect to search its area, such as in HDFS, more aggressively rather than searching parts of the space that would induce cross communication. Using a dynamic cost threshold to essentially prioritize the search through the space in order to reduce communication could prove beneficial to these and potentially other similar algorithms.

Another area of future work would be to use more advanced MPI techniques, such as incorporating lightweight POSIX threads with each heavyweight process spawned through MPI. Although this may add some complexity to the overall process, it may also potentially simplify the communication structure and require less synchronization points that cause blocking and idle time. Such techniques could allow a separate communicator thread to probe for an incoming messaging signal and then only stop to process it if one was being sent. This is likely most effective in HDFS since it does not always have new data for all processes at once and could then allow each process to continue expanding in its area until more data was sent to it. By reducing the amount of data transferred during the ATA or even potentially eliminating the need for that synchronization point, the algorithm may see significant enhancement. DCFS may also benefit from such a hybridized thread system by letting some of the communication be handled by simultaneous thread. While theoretically plausible, it is less clear how this could be structured in a significantly beneficial way; therefore this would require more study.

Adapting these algorithms for non-grid based problems could also serve to advance further study into solving large pathfinding like problems in a distributed environment. Problems such as the sliding tile puzzle or other similar problems that can be represented as a series of decisions from a beginning to an end could adapt these

algorithms for larger versions of those problems. Hashing of the tile puzzle problems is slightly more abstract than grid based problem. However, it seems reasonable to hash states out based on the position of the blank, as subsequent states are derived on that information. Additionally, while it would be slightly more conceptually difficult to apply the HDFS nesting to such problems, such a method may prove very effective in such a problem domain. Even without the addition of nesting, by extending these algorithms to other domains, it enhances their generalizability and also provides for more testing on different problem types.

Possibly the most interesting and involved enhancement of the algorithms developed here would be to introduce the use of the Asynchronous Dynamic Load Balancing (ADLB) framework [13]. ADLB provides a means to send work of user defined 'types' to a central processor that then acts a server of that that work. Because ADLB is optimized for efficiency of extremely large scale jobs, it could very well allow pathfinding problems to be scaled well if incorporated into DCFS or HDFS. In particular DCFS may stand to benefit from such an approach where a master would send all work the ADLB 'server' which could then be requested by the other workers on as-needed basis. Further, it may be possible to have all workers send their processed data to the server, thus relieving the master from that additional overhead and allowing it to focus more on processing the search space. Ultimately this has the potential to significantly improve its performance. HDFS may also potentially benefit from utilizing ADLB in its design. However, HDFS already performs fairly well in its current decentralized but synchronized design, so much larger problems would likely be necessary to determine its degree of scalability and opportunities to transfer certain components to an ADLB server.

BIBLIOGRAPHY

[1] Z. Cvetanovic and C. Nofsinger, "Parallel Astar search on message-passing architectures," in *System Sciences, 1990., Proceedings of the Twenty-Third Annual Hawaii International Conference on*, 1990, vol. 1, pp. 82–90.

[2] R. Zhou and E. A. Hansen, "Structured duplicate detection in external-memory graph search," in *AAAI*, 2004, pp. 683–689.

[3] Y. Björnsson, M. Enzenberger, R. C. Holte, and J. Schaeffer, "Fringe Search: Beating A* at Pathfinding on Game Maps.," *CIG*, vol. 5, pp. 125–132, 2005.

[4] A. Kishimoto, A. S. Fukunaga, and A. Botea, "Scalable, Parallel Best-First Search for Optimal Sequential Planning.," in *ICAPS*, 2009.

[5] S. Brand, "Efficient obstacle avoidance using autonomously generated navigation meshes," M.S. thesis, Electr. Eng. App. Math. Comp. Sci., Delft Univ. Tech., Delft, Netherlands, 2009.

[6] E. Burns, S. Lemons, W. Ruml, and R. Zhou, "Best-first heuristic search for multicore machines," *Journal of Artificial Intelligence Research*, pp. 689–743, 2010.

[7] D. Cohen and M. Dallas, "Implementation of parallel path finding in a shared memory architecture," *Department of Computer Science Rensselaer Polytechnic Institute: Troy, NY*, 2010.

[8] J. Nohra and A. J. Champandard, "The secrets of parallel pathfinding on modern computer hardware," *Intel Software Network, Intel Corp*, 2010.

[9] S. Brand and R. Bidarra, "Parallel ripple search–scalable and efficient pathfinding for multi-core architectures," in *Motion in Games*, Springer, 2011, pp. 290–303.

[10] S. Brand and R. Bidarra, "Multi-core scalable and efficient pathfinding with Parallel Ripple Search: Multi-core pathfinding with Parallel Ripple Search," *Computer Animation and Virtual Worlds*, vol. 23, no. 2, pp. 73–85, Mar. 2012.

[11] "Introduction to Parallel Computing." [Online]. Available: https://computing.llnl.gov/tutorials/parallel_comp/. [Accessed: 1-Jun-2016].

[12] "MPICH." [Online]. Available: http://www.mpich.org/. [Accessed: 1-Jun-2016].

[13] "Asynchronous Dynamic Load Balancing." [Online]. Available: https://cs.mtsu.edu/~rbutler/adlb/. [Accessed: 1-Jun-2016]

APPENDICES

APPENDIX A

SUPPLEMENTAL MATERIALS

All source code developed in the course of this research is available upon request.

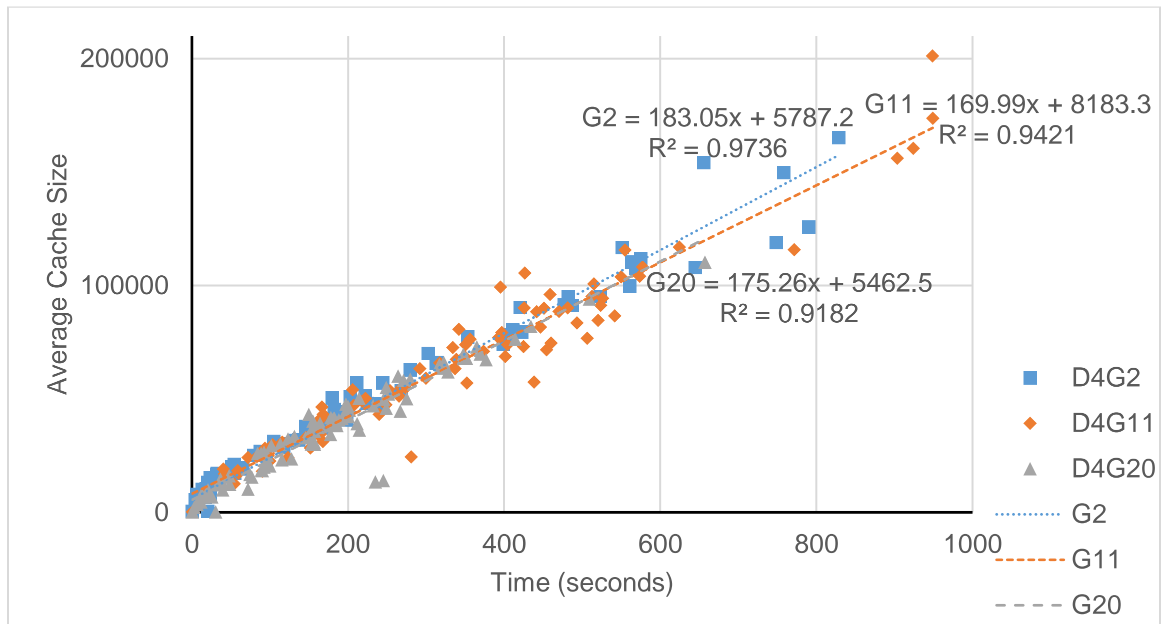

Figure A. DCFS with 4 ranks

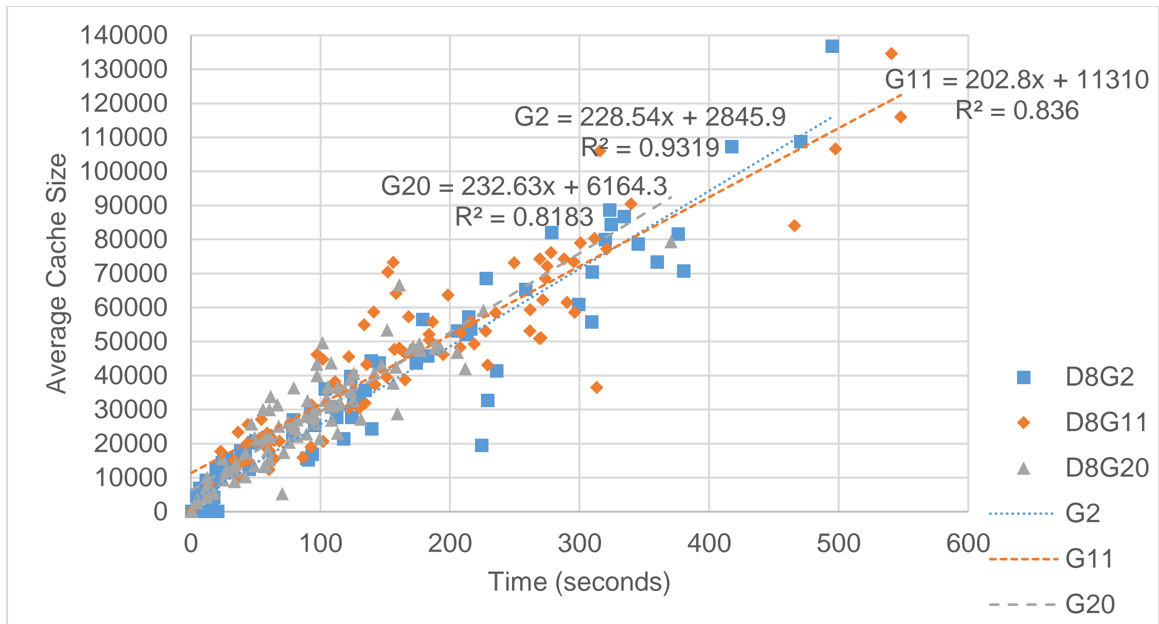

Figure B. DCFS with 8 ranks

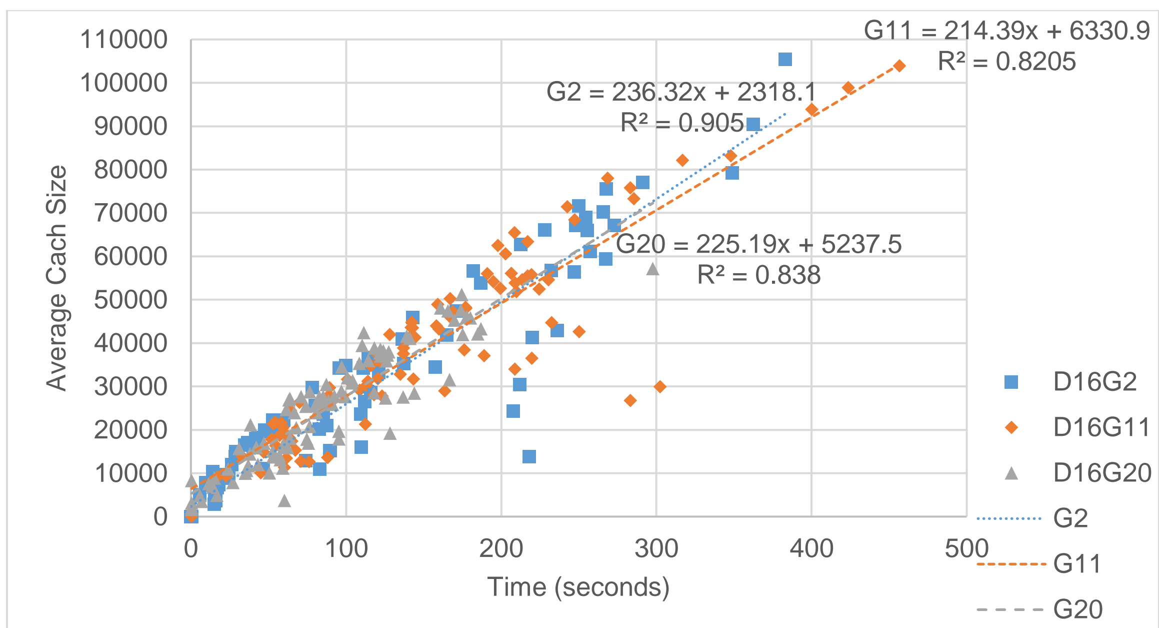

Figure C. DCFS with 16 ranks

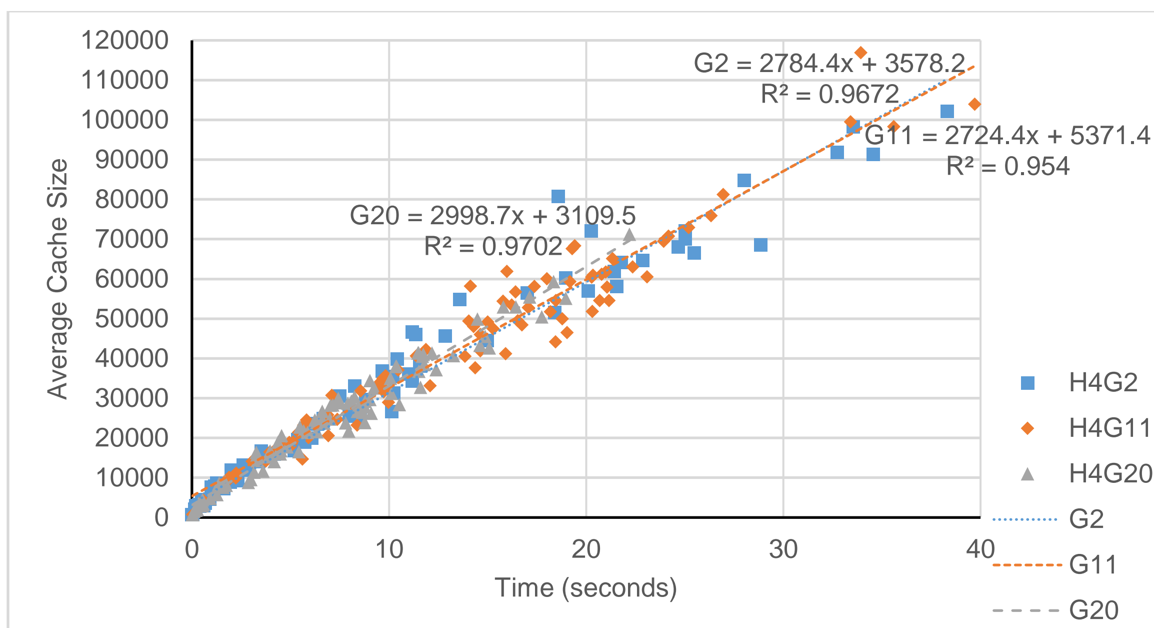

Figure D. HDFS with 4 ranks and non-nested hashing

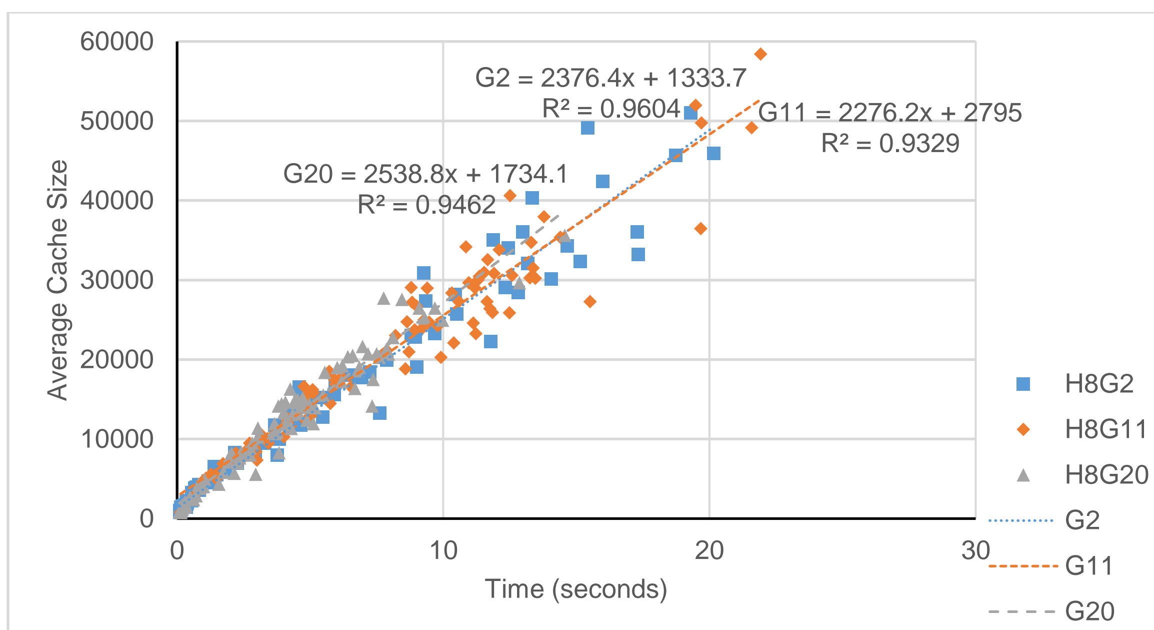

Figure E. HDFS with 8 ranks and non-nested hashing

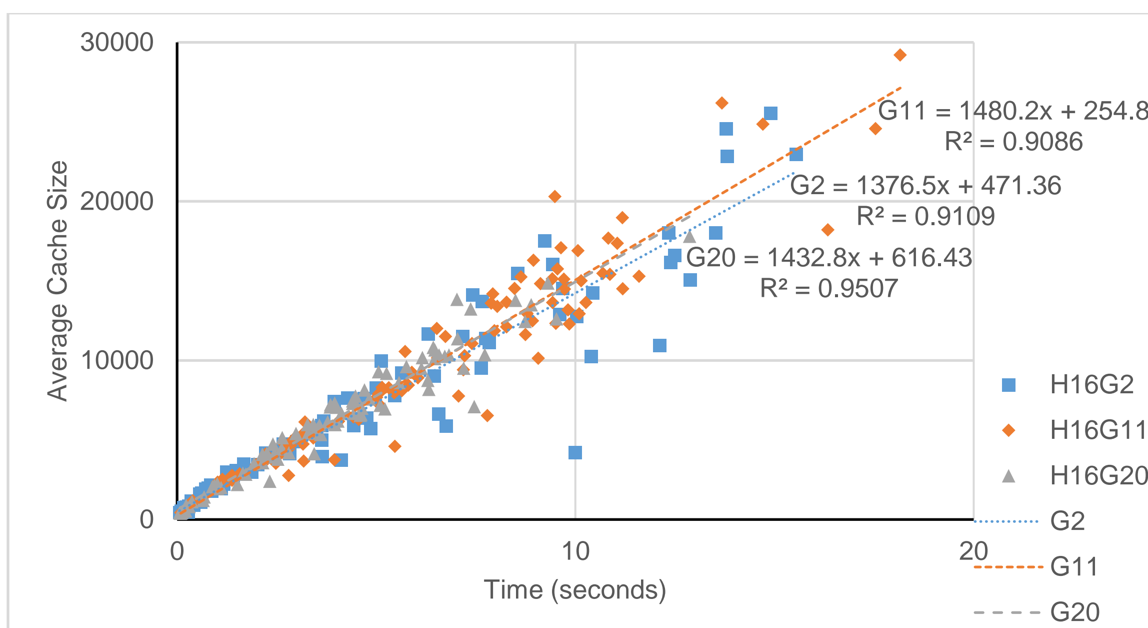

Figure F. HDFS with 16 ranks and non-nested hashing

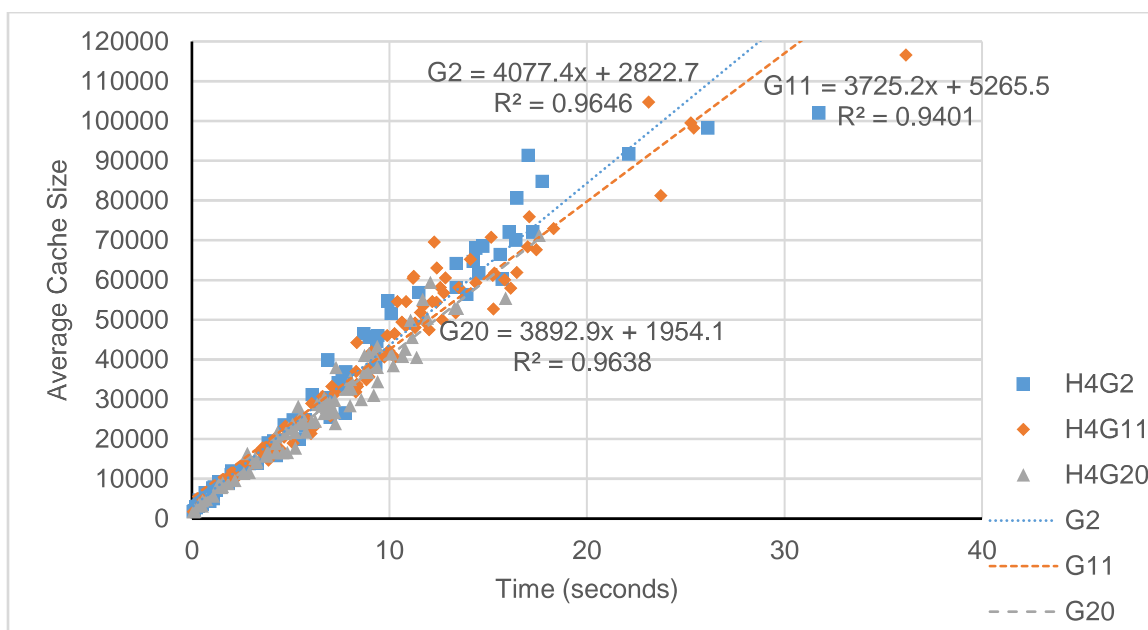

Figure G. HDFS with 4 ranks and nested hashing

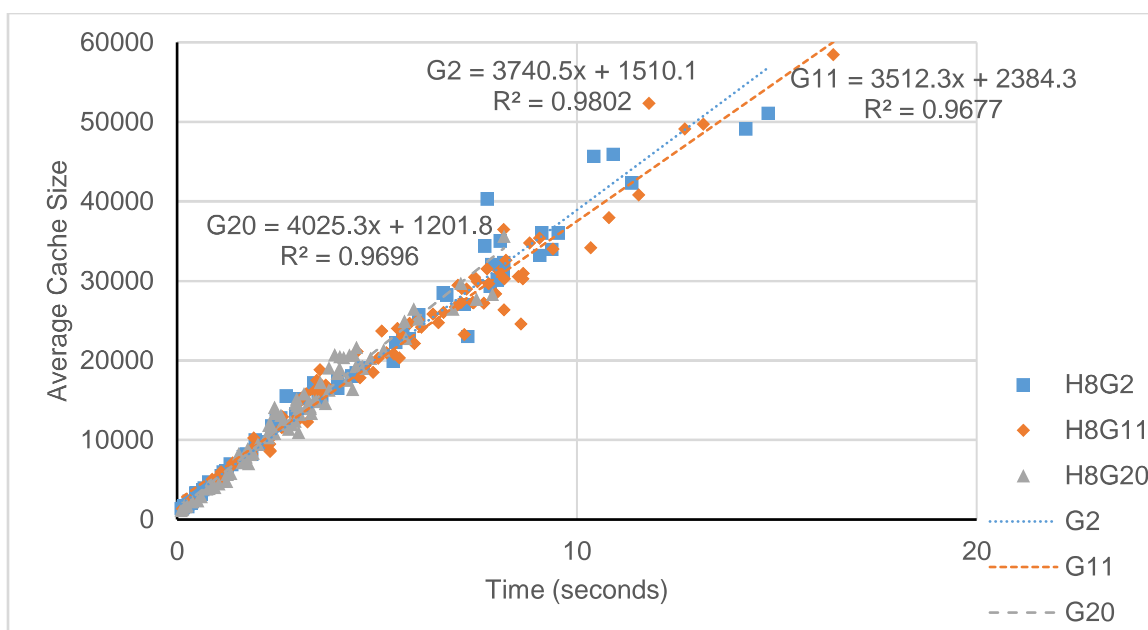

Figure H. HDFS with 8 ranks and nested hashing

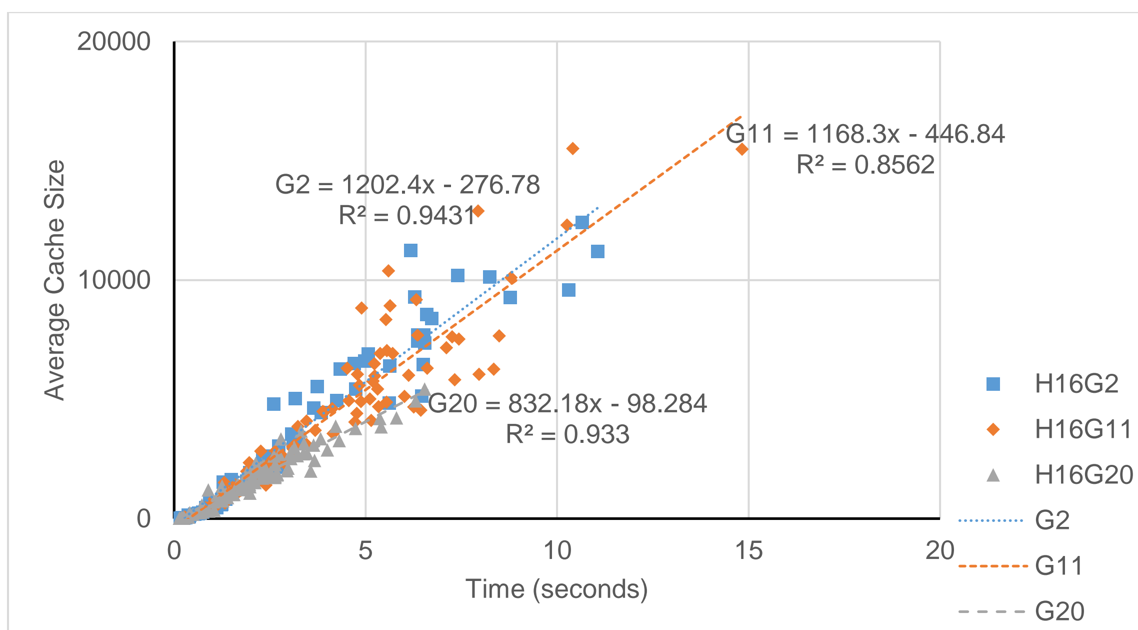

Figure I. HDFS with 16 ranks and nested hashing