Efficiency of Parallel Implementations of the A* Pathfinding Algorithm in a Purely-Functional Programming Language

Prince Bernie B. Colis
Bachelor of Science in Computer Science
John Kenneth S. Lesaba
Bachelor of Science in Computer Science
Jon Ariel N. Maravilla
Bachelor of Science in Computer Science
Karl Frederick R. Roldan
Bachelor of Science in Computer Science

Senior project submitted to the faculty of the
Department of Computer Science
College of Computer Studies, Ateneo de Naga University
in partial fulfillment of the requirements for their respective
Bachelor of Science degrees

Project Advisor: Adrian Leo Pajarillo
First Panel Member
Second Panel Member
Third Panel Member

Month Day, 2020
Naga City, Philippines

Keywords: parallel programming, functional programming, combinatorial optimization

The Senior Project entitled

Efficiency of Parallel Implementations of the A* Pathfinding Algorithm in a Purely-Functional Programming Language

developed by

Prince Bernie B. Colis
Bachelor of Science in Computer Science

John Kenneth S. Lesaba
Bachelor of Science in Computer Science

Jon Ariel N. Maravilla
Bachelor of Science in Computer Science

Karl Frederick R. Roldan
Bachelor of Science in Computer Science

and submitted in partial fulfillment of the requirements of their respective Bachelor of Science degrees has been rigorously examined and recommended for approval and acceptance.

First Panel Member
Panel Member
Date signed: _____

Second Panel Member
Panel Member
Date signed: _____

Third Panel Member
Panel Member
Date signed: _____

Adrian Leo Pajarillo
Project Advisor
Date signed: _____

The Senior Project entitled

Efficiency of Parallel Implementations of the A* Pathfinding Algorithm in a Purely-Functional Programming Language

developed by

Prince Bernie B. Colis
Bachelor of Science in Computer Science

John Kenneth S. Lesaba
Bachelor of Science in Computer Science

Jon Ariel N. Maravilla
Bachelor of Science in Computer Science

Karl Frederick R. Roldan
Bachelor of Science in Computer Science

and submitted in partial fulfillment of the requirements of their respective Bachelor of Science degrees is hereby approved and accepted by the Department of Computer Science, College of Computer Studies, Ateneo de Naga University.

Marianne P. Ang, MS
Chair, Department of Computer Science
Date signed: _____

Joshua C. Martinez, MIT
Dean, College of Computer Studies
Date signed: _____

Declaration of Original Work

We declare that the Senior Project entitled

Efficiency of Parallel Implementations of the A* Pathfinding Algorithm in a Purely-Functional Programming Language

which we submitted to the faculty of the

Department of Computer Science, Ateneo de Naga University

is our own work. To the best of our knowledge, it does not contain materials published or written by another person, except where due citation and acknowledgement is made in our senior project documentation. The contributions of other people whom we worked with to complete this senior project are explicitly cited and acknowledged in our senior project documentation.

We also declare that the intellectual content of this senior project is the product of our own work. We conceptualized, designed, encoded, and debugged the source code of the core programs in our senior project. The source code of third party APIs and library functions used in my program are explicitly cited and acknowledged in our senior project documentation. Also duly acknowledged are the assistance of others in minor details of editing and reproduction of the documentation.

In our honor, we declare that we did not pass off as our own the work done by another person. We are the only persons who encoded the source code of our software. We understand that we may get a failing mark if the source code of our program is in fact the work of another person.

Prince Bernie B. Colis

3 - Bachelor of Science in Computer Science

John Kenneth S. Lesaba

3 - Bachelor of Science in Computer Science

Jon Ariel N. Maravilla

3 - Bachelor of Science in Computer Science

Karl Frederick R. Roldan

3 - Bachelor of Science in Computer Science

This declaration is witnessed by:

Adrian Leo Pajarillo

Project Advisor

Efficiency of Parallel Implementations of the A* Pathfinding Algorithm in a Purely-Functional Programming Language

by

Prince Bernie B. Colis, John Kenneth S. Lesaba, Jon Ariel N. Maravilla, and Karl Frederick R. Roldan

Project Advisor: Adrian Leo Pajarillo

Department of Computer Science

EXECUTIVE SUMMARY

The A* algorithm is one of the most popular branch and bounding algorithms for solving combinatorial problems such as pathfinding. This paper aims to find an efficient implementation of the A* algorithm on a parallel computing environment using a functional programming language. Specifically, the algorithms experimented on are HDA* and PNBA* star, two distinct parallel A* algorithms. The proposed project hopes to combine the efficiency of the A* algorithm and the correctness of functional programming.

I dedicate this research work to all of humanity.

ACKNOWLEDGEMENTS

I thank everyone who helped me finish this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

The A* pathfinding algorithm is a best-first pathfinding algotithm for graphs commonly used for graph traversal applications such as artificial intelligence, video games, flight paths, and more. However, while most games are written in an imperative and object-oriented language such as C#, C++, and JavaScript, it is possible to write video games in a functional language using a reactive functional programming approach.[5] Likewise, the need for other correct critical software led organizations such as NASA to use Haskell[1], a purely-functional programming language, to be used in systems where high-level assurance and provable programs are a must.[28]

This paper assumes concrete differences between *parallel* and *concurrent* where the former is defined to be a hardware feature of having multiple processors or cores to compute a problem whereas the latter is defined to be a software-based approach to decrease the impact of computation bottlenecks by switching between different computations when a computation takes too long.[35] One of the major challenges of parallel programming is controlling the order of execution to prevent *race conditions*, which can often lead to bugs and are hard to maintain. However, since pure functional languages, such as Haskell, have no mutability and computations lead to the same result regardless of the order, they are a perfect candidate for writing parallel programs.[14] This research aims to find a parallel implementation of the existing A* pathfinding algorithms using a purely-functional setting with attention to program performance. In turn, this helps in the advancement of different functional programming approaches for parallel graph computations which could eventually lead to critical systems to use a more provable programming language.

1.1 Project Context

The Therac-25 medical radiation machine caused at overdose accidents in which it harmed at least six patients using the machine. One of the main causes of the problem was due to poor software engineering practices and that testing the software was not enough.[23] Most critical systems such as flight software, navigation, and military software is written in languages that may not guarantee correctness of programs such as C where adding 1 to an `int8` whose value is 65535 may cause the program to show the incorrect sum of -65536. These kinds of problems often lead to software crashes and cause unexpected failures. Also, the video game industry is dominated by imperative programming languages. However, due to the rise of languages such as TypeScript and PureScript, and frameworks such as ReactJS which favors a reactive functional programming approach, it is not fanciful to say that games might start being written in the functional style as well.

However, the functional style of programming has a different set of problems compared to its imperative counterpart such as obscurity for some programmers (though, this is subjective) and increased space complexity for the same algorithm. Multiple algorithms for the imperative approaches may not translate well and may even increase the time and space complexity for the functional approach due to its nature of copying the data structure instead of iterating over.

The advantages of functional programming lies with its *referential transparency* which means that a function definition or a variable will never change its definition throughout the runtime of the program.[20, 14] Hence, mathematically proving functional programs might be easier and can be aided by proof assistants such as Coq or Agda.[4, 37, 8] Likewise, splitting functions into smaller functions and reasoning about those smaller components much like lemmas would mean that functions would be modular and composed of proven subfunctions.[3, 18] Hence, functional programming languages are excellent candidates for parallel programming since the languages do not have mutable states and are therefore, instances of shared variables are abstracted away from the programmer. Similarly, the order of execution of pure functions does not matter as the program will still yield the same results.[20] A parallel and purely-functional approach to the A* algorithm would eventually lead to more applications such as shortest distance in a map, flight paths, web server searching, and more to use a more provable and type-safe language which could lead to less system failures and high availability.

1.2 Purpose and Description

This research aims to utilize the existing parallel A* pathfinding algorithm [10, 38] and find a way to develop a reasonably-efficient purely-functional implementation of the algorithm using parallel data structures such as STMs or MVars[26].

The A* Pathfinding algorithm is used heavily in video games, telephone traffic, and other graph traversal problems[17]. This research aims to aid in the development of video games and in developing safer critical systems with stricter error-checking and type safety.[28]

1.3 Objectives

The main objective of the research is to develop functional implementations of two different parallel A* implementations. The research will be done using Haskell and Rust as a comparative metric for imperative languages. Likewise, concrete comparisons between the number of cores and logical threads will be used to measure the most efficient runtime and space complexity of both algorithms.

The researchers aim to complete the following tasks:

- Model multiple combinatorial problems such as the n-queens problem and Sudoku. These problems may or may not have different problem sizes.
- A solver will be written both in Haskell, a laze purely-functional programming language, and Rust, a relatively modern systems programming language that shares multiple features with Haskell.
- Two algorithms will be written in both languages and their performance will be recorded. The program Threadscope will be utilized for recording the performance of the Haskell solvers, such as thread and core activities while the program is being run.

1.4 Scope and Limitations

The research will only cover Haskell, though it may generalize to other functional languages that support a parallel and concurrent approach. Translation to other functional programming languages is not a priority and thus, the use of abstract machines or lambda calculus notation will not be used. The researches deem that using a pure functional language such as Haskell will enable it to generalize

well even on impure languages such as LISP. Also, only problems where solutions exist will be tested on.

The concrete implementation and analysis is planned to be tested only on four CPUs such as Intel Core i7-9750H and AMD Ryzen 5 3500x. Other CPU architectures are not planned to be tested on.

Chapter 2

Review of Related Systems and Related Literature

Due to the trend of parallelization in the modern computing era, [32, 38, 10, **?**] several researchers developed more efficient and faster implementations of the A* algorithm by parallizing the algorithm in various ways.

2.1 The A* Algorithm

The A* pathfinding algorithm, first described by Hart, Nilsson, and Raphael, is a *best-first search* algorithm for finding a path between two vertices in a directed weighted graph.[17] The A* search algorithm can be seen as a variant of Dijkstra's algorithm with an added heuristic function to guide its search for an optimal solution. That is, the A* search functions the same as Dijkstra's algorithm if the heuristic function $h(n) = 0$ for any n.[12]

Hart, Nilsson, and Raphael had proven that the A* algorithm is *admissible* whenever the heuristic function h is admissibe. That is, A* is guaranteed to return an optimal solution. The optimality of the path generated by the A* algorithm depends on its heuristic function which will be discussed further in Chapter 3 of this paper.

2.2 Parallel A* Algorithm

Zaghloul, Al-Jami, Bakalla, et al. had shown that an implementation of a parallel A* algorithm had a speed up of 3-4 times on problem sizes of 2000x2000 to 4000x4000.[10] The implementation took advantage of reducing the problem into subgraphs where the neighbors of a vertex in a graph will have threads assigned to them and a neighbor n_i will be the new start vertex of the A* search for that given thread with the same goal vertex. By Ahmdal's Law, runtime efficiency will decrease when the number of threads are increased due to the overhead of garbage collection and thread creation.

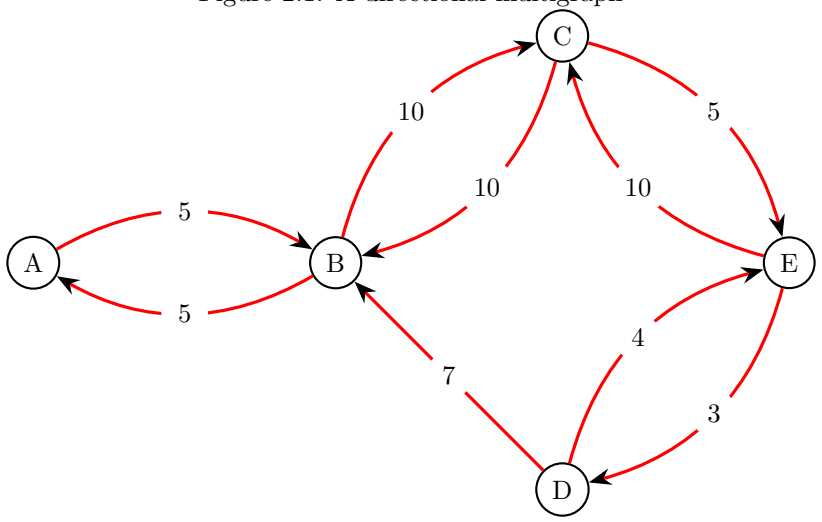
2.2.1 Bidirectional A* Search Algorithms

Bidirectional search is said to be possible when there is exactly one start vertex and exactly one goal vertex. Since the computer cannot differentiate between the two if they are reversible, it is possible to start searching from the goal g to the start s and vice versa.[19] Bidirectional search is useful when implemented on a unidirectional multigraph(2.1) where there can be an efficient path between from one vertex to another in one direction but the same path cannot exist in the other direction.

The BHPA algorithm is a bidirectional search employing two A* algorithms running sequentially with a sequential computer switching directions multiple times. The BHPA algorithm terminates whenever two search paths from both directions meet and returns the minimum f cost of either direction or by adding the g costs of the two directions. However, Kaindl and Kainz had proven that the best case performance of BHPA* is not significantly better than the unidirectional A* algorithm due to the cost of the BHPA's termination condition.[19] Kaindl and Kainz developed a more efficient implementation of a non-tradition bidirectional heuristic search that is effective on machines with limited memory. The algorithm instantiates by assigning memory to a unidirectional approach and uses A* to find a path from the start to the end vertices. If there is such a path given the memory, then the algorithm terminates. Otherwise, a reverse search from the end vertex to an intermediate vertex from the first search.

The New Bidirectional A* Search (NBA*) uses the same technique as the BHPA in the sense that the execution of each search direction alternates.[29] However, NBA* employs the reverse search on a reverse graph. That is, let $V(G)$ and $E(G)$ be the vertex set and edge set of graph G and if $u, v \in V(G)$ and $uv \in E(G)$ but $vu \notin E(G)$, then for the reverse graph rev(G) with $u, v \in V(\text{rev}(G))$

Figure 2.1: A directional multigraph



and $vu \in E(\text{rev}(G))$ but $uv \notin E(\text{rev}(G))$. The edges uv in $E(G)$ and vu in $E(\text{rev}(G))$ have the same weights. Likewise, if s_1 and t_1 are the start and goal vertices of graph G, then s_2 and t_2 are the start and goal vertices of graph $\text{rev}(G)$ with $s_2 = t_1$ and $t_2 = s_1$. As a corollary, the vertex set of $\text{rev}(G)$ is equal to the vertex set of G denoted by V from this point.

In NBA*, all vertices $v \in V$ are initially labeled $g(v) = \infty$ and the set $\mathcal{S} = \varnothing$ is the set of vertices with permanent labels. Likewise, there are shared variables $\mathcal{L} = \infty$ and $\mathcal{R} = \varnothing$ which will hold the the best cost so far and the rejected vertices, respectively. During the runtime of the algorithm, a vertex is either expanded if its heuristics or label satisfies the necessary conditions with respect to $\mathcal{L}$ or rejected and thus added in the set $\mathcal{R}$. The algorithm halts when there is one search side with no more candidates to be expanded.

Parallel New Bidirectional A* (PNBA*)

PNBA* was proposed by Rios and Chaimowicz by parallelizing the NBA* algorithm.[32] PNBA* leverages on a shared memory model and runs both search processes in parallel. Due to this, the PNBA* algorithm is faster than the NBA* and thus has a better runtime performance. The implementation of PNBA* has a variable $\mathcal{S}$, like the NBA*, shared by both processes and contains the vertices, say v, with finite labels $g(v)$. However, since both $\mathcal{L}$ and $\mathcal{S}$ are updated by both processes in parallel, it is possible that there would be a race condition and some vertex in one

thread may be expanded due to the race conditions. However, this does not affect the admissibility of PNBA*. While this may incur some overhead due to expanding additional vertices, parallelism makes up for the additional overhead costs.[32] Further discussion is in Chapter 3 of this paper.

2.2.2 Parallelism by Hashing

The Hash Distributed A* Algorithm (HDA*) builds on the the hash-based work distribution of PRA* and asynchronous communication by TDA.[21, 11, 33] The HDA* search introduces *ownership* where each processor has their own local closed and open lists and the search space is divided among processors. Unlike the implementation by Zaghloul, when a vertex is expanded by a processor P_1 in HDA*, P_1 will not immediately own the generated vertices but instead send those vertices to the message queue to be hashed by available processors which can then be processed. A pseudo-random hash function is important to make sure work is distributed equally among available processors. If the hash function is static, that is it only assigns every generated vertex to the same processor that generated said vertices, then the algorithms performs the same way as the classic A* algorithm. Due to the parallel nature of HDA* and having local open and closed sets per processor, there is no guarantee that an expanded vertex will no longer be expanded.

The simplest approach for the termination condition of HDA* is the *barrier method* wherein a processor waits for other processors when its open set is empty. This incurs some overhead and is not viable. Weinstock and Holladay describes the *sum flags method* where a binary flag is raised by a processor if its open set is empty.[38] Let p_i be processors for all $i \in 1..n$ where n is the number of processors in the computer. The algorithm will terminate if $p_1 \wedge p_2 \wedge \cdots \wedge p_n = True$.

The original paper by Kishimoto, Fukunaga, and Botea showed that HDA* performs significantly better than A*, WSA*[1], and PRA* when using message passing on a number of problems. When the HDA* is run using 4 CPU cores, the speedup averages 3 times and is shown to be very efficient compared to the other parallel implementations and the sequential A* algorithm. Likewise, the speedup is even more significant at an average of 5 times when using 8 cores. Kishimoto, Fukunaga, and Botea expounded further that the HDA* is also efficient in a distributed computing environment by showing a speedup of an average of 30-60 times in a 128-core distributed computer.[21]

[1] Another parallel implementation of the A* algorithm that involves work stealing

2.3 Real-time Performance of Functional Programs

The dominant programming languages used for time-critical software systems such as GPS systems and flight guidance softwares are written in mid-level programming languages that has direct hardware access such as C or C++. Frame and Coffey were concerned that the correctness of C and C++ were refutable and thus benchmarked and compared the real-time performance of the same program written in C++ and in Haskell, the latter being a programming language known to its correctness due to its strict typing rules.[13] Frame and Coffey hypothesized that imperative and functional programming may differ in their implementations but both are powerful enough to take on math-intensive problems.

Scott and Frame also tacked the ease of use of the programming language to the programmers and they concluded that the functional programming language may seem unfamiliar to an imperative language user at first but may be easier to understand and maintain in the long run.[13] This is especially true in static typing languages such as Haskell where programmers will be able to make less typing errors such as adding a `uint_8` to a `bool` which may cause an overflow.[15] The paper by Scott and Frame showed that functional programs are less verbose and are more expressive than their imperative counterparts for the same problem, however it still lacks in performance power. However, despite the slower runtime performance of functional programs, mathematically provable functional programs are still capable in being written as runtime verification systems in ultra-critical systems, as demonstrated by the National Aeronautics and Space Administration (NASA).[28] Due to this, it is important to develop more performant, perhaps parallel, algorithms for functional programs especially in the area of combinatorial optimization, such as the problem posed in this paper.

Harris, Marlow, and Jones evaluated the parallel performance of Haskell and overhead costs when using parallel data structures. Since Haskell is a garbage collected programming language, there would be some performance cost whenever data is being garbage collected. However, due to the lazy-nature of Haskell, unecessary data that will never be used in any computation will not be evaluated. The parallel impleentation of the Glasgow Haskell Compiler (GHC) does not really state explicit parallelism. Instead, a programmer may annotate (called *sparking*) whether a computation can be parallelized or not with different strategies. It is up to the Haskell runtime system whether the annotation will be assigned to a different processor or thread. This abstraction enables the programmer to be worry-free about how parallelism will be implemented and diverts it to other matters such as performance and how many sparks will be created.[16, 27]

Chapter 3

Technical Background

3.1 Graph Theoretic Preliminaries

A weighted, directed graph $G = (V, E)$ with V being the set of vertices of G and E being the set of edges of G is said to be a graph with an associated weight function $w : E \to \mathbb{R}$ mapping the edges of G to real-valued weights. The (u, v)-walk, where $u, v \in V$ is a sequence of vertices $(u, u_1, u_2, \ldots, v)$ where consecutive vertices are adjacent to each other in G. A (u, v)-path is a (u, v)-walk with no repeating vertices, implicitly, no repeating edges.

In this paper, an edge is denoted by the two vertices incident of the edge. Example, for two vertices u, v in a graph, an edge between u and v would be denoted by uv. Hence, a path $(v_0, v_1, \ldots, v_n)$ has edges $\{v_0v_1, v_1v_2, \ldots, v_{n-1}v_n\}$.

3.2 Minimum Cost Paths

The minimum cost paths is a discrete optimization problem wherein the goal is to determine the shortest path from a start vertex in a graph to an end vertex in a graph. This paper only deals with single-source shortest path. The shortest path problem can be defined formally as follows.

The function W of a path $W(p)$ where p is a path is denoted by the sum of the edges in the path.[6] That is, if some path $p = (v_0, v_1, \ldots, v_n)$, the function W is defined as

$$W(p) = \sum_{i=1}^{k} w(v_{i-1}v_i).$$

Hence, the shortest path weight can be defined by the function $\zeta : V \to V$ (a mapping from a vertex in G to another vertex in G) as follows

$$\zeta(u, v) = \begin{cases} \min(S_{u \to v}) & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

where $S_{u \to v}$ is the set of weights of all paths from u to v, that is, $S_{u \to v} = \{W(p) | p = (u, v_1, v_2, \ldots, v)\}$. Hence, the **shortest path** is defined as the path with the shortest weight, that is $W(p) = \zeta(u, v)$.[6]

3.3 A* Algorithm

While the shortest path in a graph problem can be solved by dynamic programming, other algorithms exist such as branch and bounding algorithms like the A* search.[17] The A* search algorithm employs the use of heuristic functions to estimate the cost or distance from the current vertex to the final vertex, while keeping in mind the current best. Hence, the A* search can be thought of as an intelligent algorithm in the classical sense. A heuristic function is **admissible** if it is proven that it will never overestimate the cost of reaching the end vertex. That is, the heuristic function may underestimate or equal the actual cost. Assuming the heuristic function is admissible, then the A* algorithm is considered to be admissible.

Likewise, the heuristic function of the A* algorithm is **consistent** or monotone if the estimate of the cost from the current vertex is always less than or equal to the cost of its neighbors in the open set (vertices that haven't been analyzed) to the end vertex plus the cost of reaching the neighbor from the current vertex.

In this version of the A* algorithm, the f-value, that is the heuristic that the algorithm is minimizing, is computed and added as the sorting factor for the priority queue, which may be implemented as a minimum binary heap. Assume that every mention of priority queue is a minimum priority queue. Likewise, the pseudocode does not enforce strict typing, that is, it is possible to operate on two data different data structures with the same type class, for easier readability. However, note that this is illegal in actual Haskell.

Definition 3.1. If a vertex v is in the closed set, then it is **expanded**. Likewise, **expanding** v means adding v to the closed set.

```
1  astar :: Graph → Vertex → Vertex → HashMap Vertex Vertex
2  astar g u = astar' trail open closed g u
3      where open   = newMinPQueue (u, f u)  -- Add the vertex u to the priority queue
4                                            -- And along with it, its heuristic  f-value
5            closed = emptyMinPQueue         -- The closed sed is initially  empty.
6            trail  = emptyHashMap           -- A hashmap with the vertex as the key and its value will be the vertex
7                                            -- that it comes from (with respect to the start  vertex)
```

Listing 3.1: A* Pseudocode

```
1  astar' ::  HashMap Vertex Vertex -- A HashMap to store the vertex and its parent
2         → PriorityQueue → PriorityQueue -- open and closed sets
3         → Graph                  -- The graph in question
4         → Vertex → Vertex        -- start and end vertices
5         → HashMap Vertex Vertex -- Return the hashmap. Use this for path reconstruction
6  astar' trailMap open closed g u v
7      | a == v || null open = trailMap  -- A* has reached the goal or open set is null
8      | otherwise           = astar' trailMap' newOpen' newClosed g u v
9      where
10         -- Get from the open set, the node with lowest f-value
11         ((a, fa), open') = getMinFromPQueue open
12         -- neighbors of a in graph g
13         currentN         = neighbors a g
14         -- neighbors of a in g not in  the closed  set
15         notExpandedN     = filter (flip . notElem closed) currentN
16         -- neighbors of a in the closed  set
17         closedSetN       = filter (flip . elem closed) currentN
18         -- Compute the f-value of each n in currentN first,
19         -- and then add them to the open priority queue
20         newOpen          = foldl insertPQueue open' (f <$> notExpandedN)
21         -- For all elements n in closedSetN, check if  the f-value is  lower
22         -- than the cost when n is not expanded.
23         -- If f(n) is  lower,  then add n to the open set
24         newOpen'         = foldl insertPQueue newOpen (f <$> closedSetN)
25         -- Add all members of currentN to the hashmap with a as their parent vertex
```

```
26          trailMap'       = foldl (\acc x → insertHashMap acc x a) trailMap currentN
27          -- Get the intersection of the closed set and newOpen'
28          -- and subtract the intersection from closedSetN
29          -- This is simply to get all the vertices that will not be expanded
30          closed'         = (intersection closed newOpen') \\ closedSetN
31          -- Add all members of closed' to the new closed set
32          newClosed       = foldl insertPQueue closed closed'
```

Listing 3.2: A* Helper Function Pseudocode

3.4 Parallel A* Algorithm

Two parallel methods described will be used in this paper. Specifically, HDA* and PNBA*.[21, 32]

3.4.1 PNBA*

The PNBA* is a parallelization of the bidirection A* algorithm. One of the main advantages of computing the shortest path bidirectionally is that it reduces the search space and search effort.[19, 29]. The execution of the PNBA* can be done on a computer with two CPU cores (or two CPUs). One CPU searches forwardly, that is, from the start vertex, it finds a path to the end vertex. The other CPU searches backwardly wherein it starts from the end vertex to the start vertex. The PNBA* heuristic function need not impose that the used heuristic function is admissible. That is, only consistency is needed. As discussed in Chapter 2, the PNBA* is a parallelized version of the NBA*. Since the NBA* employs alternating execution between the two directions, it may be distributed on two different CPU cores as demonstrated by Rios and Chaimowicz. The PNBA* can use a data structure such as a priority queue. Each direction will have their own priority queue to serve as the open set Ω_p for $p \in \{1, 2\}$, for each process p, as stated in the paper by Rios and Chaimowicz.

Definition 3.2. A vertex v is **labeled** if the estimate $g(v)$ (the shortest path found so far, that is, an estimate of $\zeta(v_{start}, v)$) is finite and v hasn't been expanded.

The paper establishes the following notational convention to be used for the rest of the paper with regards to the PNBA* algorithm. Two processes exist, namely process p in $p \in \{1, 2\}$. Whenever p is

used, it is implied that p is the process number, i.e the CPU being referenced. Let $h_p(v) = \zeta_p(v, v_{goal})$ be the heuristic for estimating the cost from the current vertex v to the goal vertex v_{goal}. Likewise, let $g_p(v) = \zeta_p(v_{start}, v)$ be shortest path so far. The shared set $\mathcal{M}$ contains the vertices in the middle of the two searches. Initially, all vertices are in $\mathcal{M}$. The shared variable $\mathcal{L}$, the best solution so far, initially set to ∞ since the algorithm will need to minimize $\mathcal{L}$ as small as possible. The variable F_p is the lowest f_p-value of Ω_p where $f_p(v) = h_p(v) + g_p(v)$. The variables f_p, g_p, and F_p can be written only by process p but can be read by both. Note that it is best to implement the open sets Ω_p.

Though an abuse of notation, if $p \in \{1, 2\}$ then $\neg : \{1, 2\} \to \{1, 2\}$ is described as follows:

$$\neg p = \begin{cases} 1 & \text{if } p = 2 \\ 2 & \text{if } p = 1. \end{cases}$$

The $\neg p$ notation will be used whenever the other processor needs to be referenced while the current processor is running.

Algorithm 1 PNBA* (Parallel New Bidirectional A*) Search Algorithm

Input: G, the graph

Input: u, start vertex and v, the end vertex

Let $\mathcal{M}$ contain all vertices in G and for all $m \in \mathcal{M}$, let $g_p(m) = \infty$.

Let $s_1 = t_2 = u$ and $s_2 = t_1 = v$.

Set $g_p(s_p) = 0$ and add s_p into Ω_p.

From here on, all instructions are run in parallel.

while $\Omega_1 \neq \varnothing$ or $\Omega_2 \neq \varnothing$ **do**

Get the vertex $n \in \Omega_p$ with the minimum $f_p(n)$.

if $n \in \mathcal{M}$ **then**

if $f_p(n) < \mathcal{L}$ and $g_p(n) + F_{\neg p} - h_p(n) < \mathcal{L}$ **then**

for all m is a neighbor of n and m is not expanded **do**

if $m \in \mathcal{M}$ and $g_p(m) > g_p(n) + \zeta_p(nm)$ **then**

Set $g_p(m) = g_p(n) + \zeta_p(nm)$

Set $f_p(m) = g_p(m) + h_p(m)$

if $m \in \Omega_p$ **then**

Remove m from Ω_p

end if

Insert m to Ω_p.

if $g_1(m) + g_2(m) < \mathcal{L}$ **then**

Lock $\mathcal{L}$ to ensure that F_p and L are monotonic.

if $g_1(m) + g_2(m) < \mathcal{L}$ **then**

Set $\mathcal{L} = g_1(m) + g_2(m)$.

end if

Unlock $\mathcal{L}$ after the update.

end if

end if

end for

end if

Remove vertex n from $\mathcal{M}$.

end if

if $\Omega_p \neq \varnothing$ **then**

Set F_p with the lowest f_p-value in Ω_p.

end if

end while

3.4.2 HDA*

Unlike the PNBA*, the Hash-Distributed A* Search can employ the use of as many CPU cores as desired.[21] One of the main advantages of this is that it can easily be implemented on a GPU with hundreds or thousands of cores, thereby, execution would be faster. In the PNBA*, each process has different closed and open sets but in the HDA*, these sets are a parallel data structure shared among each CPU cores. However, there is a hash function $k : V(G) \to P$ where $V(G)$ is the set of vertices in G and P is the set of processors. When a new vertex is found and the hash function is computed for the vertex, the algorithm determines which CPU core should own the vertex based on the hash function.[1] Thereby, while the open and closed sets are implemented as a parallel data structure, each processor owns a space in the open and closed set denoted by Ω_p and Γ_p respectively for processor p, based on the hash key.

Algorithm 2 HDA* (Hash-Distributed A*) Search Algorithm

Input: G, the graph
Input: u, the start vertex
Input: v, the goal vertex
  Let Ω be the open set and Γ be the closed set. Let f be the evaluating function.
  Add u to Ω
  Everything from here on is run in parallel. Each processor p runs the while loop below.
  while global $\Omega \neq \varnothing$ **do**
    if p has a new vertex a in its message queue. **then**
      if $a \notin \Gamma_p$ **then**
        Add a to Ω_p.
      end if
    else
      Get the vertex a with the minimum $f(a)$ from Ω_p.
      Let N be the set of neighbors of a not in Γ_p.
      Compute hash key $k(n)$ for all $n \in N$
      Let p' be the processor that owns the hash key $k(n)$.
      while The message queue of p' is locked by another processor **do**
        Wait
      end while
      Lock p' message queue
      Send n to p'
      Unlock $'$ message queue.
    end if
  end while

[1] Like how a hashmap works

3.5 Haskell Parallel Runtime

Haskell's parallel execution is implicit and can be done in a monadic environment.[26] Since Haskell is a functional program, the task of parallelization is made easy by the runtime by `rpar` and `rseq` and by specifying strategies on how haskell should parallelize the task. However, the Glasgow Haskell Compiler (GHC) runtime may or may not parallelize the function at all, this is because `rpar` sparks the function for parallelization, that is, there is a potential that it may be parallelized.[**?**] A common problem for parallelization in Haskell is that the runtime garbage collection might take more time compared to the actual algorithm execution, therefore decreasing efficiency significantly. This can be mitigated and prevented by analyzing the granularity of the data structures. For example,

```
1  -- We have a list of 10000 elements
2  xs = [0..10000]
3  -- square all elements of xs
4  xs' = (^2) <$> xs
5  -- execute in parallel
6  runEval (evalList xs')
```

This example is inefficient since the GHC runtime will assign each element of `xs` to a different thread, thereby increasing CPU overhead and garbage collection since the inexpensive task of squaring numbers is undermined by the more expensive task of garbage collection. A better implementation would be

```
1  -- We have a list of 10000 elements
2  xs = [0..10000]
3  -- square all elements of xs
4  xs' = (^2) <$> xs
5  -- execute in parallel
6  runEval (parListChunk 5000 xs')
```

In this example, the GHC runtime will divide the list into two and have two different processors evaluate the two sublists. This way, the processor assignment and garbage collection overhead will be minimized.

Hence, in Haskell, problems of parallelization switched from the implementation details such as deadlocks, race conditions, and others commonly encountered in other languages, to that of the dividing the task into chunks that GHC will be performant.

Chapter 4

Methodology

4.1 Experimentation

The researchers would write multiple programs in Haskell to solve combinatorial problems. Likewise, previous papers from HDA* and PNBA* will be reimplemented using Rust due to its Algebraic Data Types, much like Haskell, and its zero-cost abstraction features, which will be utilized as to compare the performance of the algorithm if implemented in either programming disciplines.[21, 32]

4.2 Data Gathering and Documentation

Run times on different systems with different CPU cores will be recorded, such as idle times, thread sparking, and garbage collection, using Threadscope.[2] Results will be recorded in a tabular format and plotted with its performance metrics with respect to the problem size and performance for each number of CPU cores. Comparisons between both CPU run times and CPU core activities will be made to examine each problem for each alorithm for both the Haskell implementation and Rust implementation. Metrics will be plotted in the same figure for ease of readability.

4.3 Combinatorial Problems

To make the results of the paper more reliable, multiple combinatorial problems will be tested upon and recorded. The problems are handpicked from previous researches and some of the problems are

classic combinatorial problems that might provide interesting results when testing the program.

- **Freecell** is a solitaire-like card game employing the standard 52-deck card. An optimal solution is when the program finds a way to stack all the cards in their corresponding stack with the same suit and in a chronological order.

- **Sokoban** is a puzzle video game in which the player's requirement to win is to push the boxes into corresponding storage locations. Since Sokoban has many levels that rage from easy to difficult, these levels of difficulty can be used as a *problem size* when recording the experiment. An optimal solution guarantees the least number of moves to push the crates and cover their corresponding storage locations.

- **n-queens problem** is a classic combinatorial problem where there is a standard chess board and n queens positioned in a way such that no two queens are threatening each other. The n-queens problem is a generalization of the eight queens problem which has a total of 92 solutions.

- **Sudoku** is another classical combinatorial game with some cells containing numbers and the goal is to solve the remaining cells.

- **Knapsack Problem** is one of the classic optimization problems where the reward is maximized while the cost is minimized. The problem stated here will be the 0/1 Knapsack Problem which is itself solvable by a branch and bounding algorithm like the A*.

Appendix A

Code Listing

REFERENCES

[1] *Haskell programming language*. `https://haskell.org`.

[2] *Threadscope*. `https://github.com/haskell/ThreadScope`.

[3] A. Abel, M. Benke, A. Bove, J. Hughes, and U. Norell, *Verifying haskell programs using constructive type theory*, 01 2005, pp. 62–73.

[4] J. Breitner, A. Spector-Zabusky, Y. Li, C. Rizkallah, J. Wiegley, and S. Weirich, *Ready, set, verify! applying hs-to-coq to real-world haskell code*, 2018.

[5] M. H. Cheong, *Functional programming and 3d games*, (2006).

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, The MIT Press, 3rd ed., 2009.

[7] L. Dhulipala, G. E. Blelloch, and J. Shun, *Theoretically efficient parallel graph algorithms can be fast and scalable*, in Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA '18, New York, NY, USA, 2018, Association for Computing Machinery, p. 393–404.

[8] Y. El Bakouny, T. Crolard, and D. Mezher, *A coq-based synthesis of scala programs which are correct-by-construction*, Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs, (2017).

[9] M. Erwig, *Inductive graphs and functional graph algorithms*, Journal of Functional Programming, 11 (2001), pp. 467–492.

[10] Z. et al., *Parallelizing a* path finding algorithm*, International Journal Of Engineering And Computer Science, 6 (2017), pp. 22469–22476.

[11] M. Evett, J. Hendler, A. Mahanti, and D. Nau, *Pra*: Massively parallel heuristic search*, Journal of Parallel and Distributed Computing, 25 (1995), pp. 133–143.

[12] D. Ferguson, M. Likhachev, and A. Stentz, *A guide to heuristic-based path planning*, in Proceedings of the international workshop on planning under uncertainty for autonomous systems, international conference on automated planning and scheduling (ICAPS), 2005, pp. 9–18.

[13] S. Frame and J. W. Coffey, *A comparison of functional and imperative programming techniques for mathematical software development*, Journal of Systemics, Cybernetics and Informatics, 12 (2014), pp. 1–10.

[14] K. Hammond, *Why parallel functional programming matters: Panel statement*, in Reliable Software Technologies - Ada-Europe 2011, A. Romanovsky and T. Vardanega, eds., Berlin, Heidelberg, 2011, Springer Berlin Heidelberg, pp. 201–205.

[15] S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, and A. Stefik, *An empirical study on the impact of static typing on software maintainability*, Empirical Software Engineering, 19 (2014), pp. 1335–1382.

[16] T. Harris, S. Marlow, and S. P. Jones, *Haskell on a shared-memory multiprocessor*, in Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell, 2005, pp. 49–61.

[17] P. E. Hart, N. J. Nillson, and R. Betram, *A formal basis for the heuristic determination of minimum cost paths*, IEEE Transactions of Systems Science and Cybernetics, SSC-4 (1968).

[18] J. Hughes, *Why Functional Programming Matters*, Computer Journal, 32 (1989), pp. 98–107.

[19] H. Kaindl and G. Kainz, *Bidirectional heuristic search reconsidered*, Journal of Artificial Intelligence Research, 7 (1997).

[20] M. Kesseler, *The Implementation of Functional Languages on Parallel Machines with Distributed Memory*, PhD thesis, Radboud University Nijmegen, 1996.

[21] A. Kishimoto, A. Fukunaga, and A. Botea, *Scalable, parallel best-first search for optimal sequential planning*, Association for the Advancement of Artificial Intelligence, (2009).

[22] D. E. Knuth, *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*, Addison Wesley Longman Publishing Co., Inc., USA, 1997.

[23] N. Leveson and C. Turner, *An investigation of the therac-25 accidents*, Computer, 26 (1993), pp. 18–41.

[24] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, S. Priebe, Á. J. Rebón, and P. W. Trinder, *Comparing parallel functional languages: Programming and performance*, Higher-Order and Symbolic Computation, 16 (2003), pp. 203–251.

[25] A. Lumsdaine, D. P. Gregor, B. Hendrickson, and J. W. Berry, *Challenges in parallel graph processing*, Parallel Process. Lett., 17 (2007), pp. 5–20.

[26] S. Marlow, *Parallel and Concurrent Programming in Haskell*, O'Reilly Media, Inc., 2013.

[27] S. Marlow, S. Peyton Jones, and S. Singh, *Runtime support for multicore haskell*, in Proceedings of the 14th ACM SIGPLAN international conference on Functional programming, 2009, pp. 65–78.

[28] I. Perez, F. Dedden, and A. Goodloe, *Copilot 3*, Tech. Rep. 20200003164, National Aeronautics and Space Administration (NASA), May 2020.

[29] W. Pijls and H. Post, *A new bidirectional search algorithm with shortened postprocessing*, European Journal of Operational Research, 198 (2009), pp. 363–369.

[30] R. C. Prim, *Shortest Connection Networks And Some Generalizations*, Bell System Technical Journal, 36 (1957), pp. 1389–1401.

[31] F. Rabhi and G. Lapalme, *Algorithms; A Functional Programming Approach*, Addison-Wesley Longman Publishing Co., Inc., USA, 1st ed., 1999.

[32] L. H. O. Rios and L. Chaimowicz, *Pnba*: A parallel bidirectional heuristic search algorithm*, in ENIA VIII Encontro Nacional de Inteligê ncia Artificial, 2011.

[33] J. Romein, A. Plaat, H. Bal, and J. Schaeffer, *Transposition driven work scheduling in distributed search*, in AAAI National Conference, 1999, pp. 725–731.

[34] V. Sanz, A. De Giusti, and M. Naiouf, *Improving hash distributed a* for shared memory architectures using abstraction*, in International Conference on Algorithms and Architectures for Parallel Processing, Springer, 2016, pp. 431–439.

[35] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, Wiley Publishing, 10th ed., 2018.

[36] S. S. Skiena, *The Algorithm Design Manual*, Springer, London, 2008.

[37] A. Spector-Zabusky, J. Breitner, C. Rizkallah, and S. Weirich, *Total haskell is reasonable coq*, Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, (2018).

[38] A. Weinstock and R. Holladay, *Hash distributed a* distributed termination detection path reconstruction*, 2016.

VITA

`/*TODO*/` are BS Computer Science student of the Department of Computer Science at the Ateneo de Naga University.