BACHELOR'S THESIS / ENGINEERING 2024

Parallel Strategies for Best-First Generalized Planning

Alejandro Fernández Alburquerque

upf. Universitat Pompeu Fabra Barcelona

Parallel Strategies for Best-First Generalized Planning

TREBALL DE FI DE GRAU DE

Alejandro Fernández Alburquerque

Javier Segovia Aguas

Bachelor's degree in Computer Engineering

Year 2023 - 2024

Universitat Pompeu Fabra Barcelona

Escola d'Enginyeria

Acknowledgements

I'd like to give a huge thanks to my tutor, Javier, who has been a great source of knowledge, help, and motivation and has made this work possible. I also want to give many thanks to Verónica Moreno, who has helped me find the discipline required to finish such a daunting task.

Abstract

In recent years, Artificial Intelligence (AI) has become the big trend in computer science, and many related areas are seeing renewed interest. One of these areas is Generalized Planning (GP), which studies the automated synthesis of algorithmic-like solutions capable of solving multiple classical planning instances. Tightly coupled with the success of AI, there has been a steady increase in computational power thanks to multi-core CPUs and GPUs. This has encouraged active research in parallel programming, which is needed to use the full potential of current hardware. In this work, we explore parallelization strategies for tree-search algorithms. Furthermore, we propose one algorithm to parallelize Best-First Generalized Planning (BFGP), a heuristic search approach to GP. We show that our algorithm can scale linearly with the number of processors, but we also discuss some cases in which pathological behavior can cause performance degradation.

Resum

Recentment, la Intel·ligència Artificial (IA) s'ha convertit en la gran moda de l'enginyeria informàtica, la qual cosa ha significat que moltes àrees relacionades estan veient un interès renovat. Una d'aquestes àrees és Generalized Planning (GP), que estudia la síntesi automatitzada de solucions algorítmiques capaces de resoldre múltiples instàncies de problemes de Classical Planning d'un mateix domini. L'èxit de la IA està molt lligat a l'increment en capacitat computacional proporcionat per processadors multinucli i GPUs. Això ha propiciat la recerca sobre programació paral·lela, necessària per extreure el màxim potencial dels ordinadors actuals. En aquest treball, explorem estratègies de paral·lelització d'algoritmes de cerca d'arbres. A més a més, proposem un algoritme per paral·lelitzar Best-First Generalized Planning (BFGP), una estratègia basada en cerca heurística de GP. També demostrem que el nostre algoritme pot escalar linearment amb el número de processadors, i discutim alguns casos patològics que podem causar una degradació del rendiment.

Contents

List of Tables

List of Algorithms

Chapter 1

INTRODUCTION

Automated planning (AP) research is mainly focused on constructing sequences of actions (commonly known as plans) to go from a specific initial state to a goal state across various domains [Ghallab et al., 2016]. A wide range of models can deal with AP problems with varying degrees of complexity, from deterministic systems with full observability to non-deterministic and partially observable models. The former is the simplest model and is referred to as *classical planning*. Typical examples of classical planning include blocksworld problems, transportation domains like grid problems, or solving the Towers of Hanoi puzzle.

With the increasing appeal of developing general-purpose artificial intelligence (AI) systems, renewed interest has emerged around the idea of synthesizing "generalized" plans that can solve multiple instances from the same class of planning problems or domain (instead of synthesizing a specific plan for each instance). The problem of finding these general plans is called *generalized planning* (GP). The research area of generalized planning is especially promising in bridging the gap between AP and program synthesis. In contrast to Large Language Models (LLMs) (the big trend in AI nowadays), solutions generated by GP may have theoretical guarantees like terminating, sound and provably general.

Traditionally, generalized planning systems have suffered several limitations compared to state-of-the-art classical planning solvers. On the one hand, most approaches to GP constrain the instances to have the same amount of world objects. On the other hand, generalized plan representations avoid incorporating loops of actions because of the added complexity involved [Srivastava, 2011]. However, in recent years, active research has reduced the gap between the performance and capabilities of classical planning and GP solvers. In particular, the heuristic-based approach of Best-First Generalized Planning (BFGP) has obtained very positive results.

Tightly coupled with the recent success of AI is the use of parallelization techniques to accelerate the intensive computations involved. In the last two decades,

there has been an upward trend in the development of programs that can take advantage of multi-core processors' parallelization capabilities, as chip makers have found that increasing the clock frequency or the instructions per cycle (IPC) of CPUs is no longer a viable strategy for substantially increasing their performance [Kishimoto et al., 2013; Shimoda and Fukunaga, 2023]. Therefore, there has been a progressive shift towards chip multi-processor systems [Blake et al., 2010], which are best utilized by multithreaded applications.

More recently, the use of GPUs for general computing tasks has further increased the performance of parallel programs, reaching up to several orders of magnitude of improvement in best-case scenarios. However, developing applications to be run on GPUs remains a complex endeavor, and it is only suitable for tasks that involve a particular set of operations like parallel matrix multiplications.

This thesis presents our development of parallelization strategies for BFGP in multi-core processor systems. The document is structured as follows: Chapter 2 introduces the theoretical background and formalisms upon which BFGP is built. Chapter 3 presents the concept of tree-based searching problems and parallelization strategies suitable for such problems. Chapter 4 explains our parallelized BFGP implementation and compares its performance with the original single-threaded execution. Chapter 5 provides a general overview of previous work on generalized planning and parallelization techniques. Finally, Chapter 6 wraps up our work and discusses open issues and future work.

Chapter 2

PRELIMINARIES

In this section, we first cover the project's theoretical basis. First, we will review the fundamental concepts of classical planning, which will help us understand the motivation and goals of Generalized Planning (GP). Once we have established the main ideas of GP, we will analyze the Best-First Generalized Planning (BFGP) algorithm, a heuristic search approach to GP that is at the core of this project.

2.1 Classical planning

Classical planning is defined as the task of finding a sequence of actions to accomplish a goal in a discrete, deterministic, static, fully observable environment [Russell and Norvig, 2021]. In contrast to other Artificial Intelligence (AI) approaches like search algorithms or reinforcement learning, classical planning does not require domain-specific knowledge. Furthermore, classical planning also stands out thanks to its ability to compactly represent problems with huge state spaces using factored representations. Finally, another key trait of classical planning is its explainability. Classical planning solvers generate a sequence of actions, known as plans, and this white-box approach is crucial in environments where human interaction is mandatory, such as health, law, or defense. Below, we introduce the same formalization of the classical planning model as in Segovia-Aguas et al. [2022b].

A classical planning *problem* can be defined by the pair $\langle \mathcal{D}, \mathcal{I} \rangle$, where $\mathcal{D}$ is the problem *domain* and $\mathcal{I}$ is the particular problem instance information. A problem's domain is composed of a set of predicates Ψ, and a set of action schemes A. These action schemes are specified by preconditions and effects, given by a set of ground atoms $p(x_1, ..., x_k)$ or their negations, where $p \in \Psi$ and each x_i represents an argument of the action scheme. A classical planning *instance* is defined by the tuple $\langle \Omega, I, G \rangle$, where Ω is the finite set of objects that exist in this particular instance, I is the initial configuration of the world objects, and G is the set of all

possible goal configurations of the world objects. I and G are specified using a set of ground atoms $p(o_1, ..., o_l)$ or their negation, where each o_i represents a world object.

A classical planning problem P has an associated set of states $S(P)$, which are the possible sets of ground atoms. The initial state is $s_0 = I$, and the goal states are the states $s_g \in S(P)$ such that $G \subseteq s_g$. Given all the states of a classical planning problem, we can define the *state graph*, where each node is one of the possible states in $S(P)$. Edges are defined as follows: for each pair of states $s \in S(P)$ and $s' \in S(P)$, there is a directed edge between s and s' if there exists a ground action a that is applicable in s (i.e., the preconditions of the action are satisfied in s) and the effects of which transform state s into $s' = f(s, a)$.

A *solution* of a planning problem instance is a list of actions $\pi = \langle a_1, ..., a_m \rangle$, called *plan*, that need to be taken to transform the initial state into a goal state. The execution of a plan defines a trajectory $\tau = \langle s_0, a_1, s_1, ..., a_m, s_m \rangle$, where each action a_i is applicable in state s_{i-1} and the successor of applying such action is of $s_i = f(s_{i-1}, a_i)$. A plan π solves problem P if, when applied to $s_0 = I$ (the initial configuration), the last state is a goal state, i.e., $G \subseteq s_m$.

2.2 Generalized planning

Generalized Planning is defined as the task of solving multiple classical planning problems of the same domain $\mathcal{D}$ with a single algorithmic-like solution [Jiménez et al., 2019].

A generalized planning *problem* is a finite non-empty set $\mathcal{P} = \{P_1, ..., P_T\}$ of T classical planning problem instances that belong to the same domain $\mathcal{D}$ (i.e., they are all represented by the same predicates and action schemes). Each instance P_i may differ in the number or identity of the world objects and their initial or goal configurations.

A generalized planning *solution* Π, called *generalized plan*, solves a GP problem $\mathcal{P} = \{P_1, ..., P_T\}$ iff, for every classical planning instance $P_t \in P$, $1 \leq t \leq T$, the execution of Π on P_t produces a classical plan that solves P_t. The execution of a generalized plan Π on an instance P_t is denoted as $exec(\Pi, P_t) = \langle a_1, ..., a_m \rangle$.

2.3 Planning programs

BFGP represents generalized planning problems solutions as *planning programs*. Formally, a planning program is a sequence of n instructions $\Pi = \langle w_0, ..., w_{n-1} \rangle$, where each instruction w_i is associated with a *program line* $0 \leq i < n$ and can be of one of the following three types:

A planning action $w_i \in A$.

A goto instruction $w_i = goto(i', !y)$, where i' is a program line such that $0 \leq i' < n$ and $i' \neq i$, and y is a proposition. Proposition y can be the result of an arbitrary expression on state variables.

A termination instruction $w_i = \mathsf{end}$. The last instruction of a planning program is always a termination instruction.

The execution model for a planning program P is a *program state* (s, i), where s is a planning state, i.e., $s \in S(P)$ and i s.t. $0 \leq i < n$ is a program counter that indicates the current line of the program. Given a program state (s, i) and an instruction w_i, the execution of w_i is defined as follows:

- If $w_i \in A$ and w_i is applicable in s (i.e., the preconditions of w_i hold in s) such that $s' = f(s, a)$, then the resulting program state is $(s', i + 1)$. On the other hand, if w_i is not applicable in s, then the resulting program state is $(s, i + 1)$.
- If $w_i = goto(i', !y)$ and y is true, then the new program state is $(s, i + 1)$[1]. If, on the contrary, y is false, the new program state is (s, i').
- If $w_i = \mathsf{end}$, then the program terminates.

When executing a planning program on a particular classical planning problem instance $P = \langle \mathcal{D}, \mathcal{I} \rangle$, where $\mathcal{I} = \langle \Omega, I, G \rangle$, the initial program state is set to $(I, 0)$ (the initial state of P and the initial line of Π). A planning program solves the problem instance P iff it terminates in a state (s, i) such that the planning state s is a goal configuration, i.e., $G \subseteq s$, and $w_i = \mathsf{end}$. Therefore, there are only two conditions on which a planning program can fail to solve the instance:

Incorrect program if $w_i = \mathsf{end}$ but $G \not\subseteq s$.

Infinite program if the program never terminates. This happens when an end instruction is never reached.

2.4 Classical planning with pointers

The space size of planning programs depends on the propositions used for goto instructions, which are determined by the domain size of state variables and are unbounded in the worst case [Segovia-Aguas et al., 2021] (e.g., integer domain).

[1]Here we use the convention of jumping to line i' if the proposition y is false

Such dependency poses a scalability limitation, making the presented planning programs only viable for classical planning instances of small size. In order to have a tractable solution space for generalized planning regardless of the size of the problem instances, the notion of pointers over world objects is introduced.

A *pointer* $z \in Z$ is a bound variable for indexing world objects of a classical planning instance $\mathcal{I}$. Consequently, pointers have a finite domain $D_z = [0, |\Omega| - 1]$, where $|\Omega|$ represents the amount of world objects of $\mathcal{I}$.

With this new concept, we can redefine planning programs as follows. Planning actions $w_i \in A$ are no longer grounded over the instance objects in Ω. Instead, the action schemes are now instantiated over pointers in Z, forming a new set of instantiated actions that we denote as A_Z. Consequently, the execution model of planning programs is slightly modified as well; planning actions $w_i = \alpha[\vec{z}] \in A_Z$ first assign each pointer to its indexed world object, producing a ground action $a = \alpha[\vec{o}] \in A$, then the previously defined execution applies.

Furthermore, the set of possible instructions of a planning program is extended with the following set of *primitive pointer operations*: $\{\texttt{inc}(z_1), \texttt{dec}(z_1), \texttt{clear}(z_1), \texttt{set}(z_1, z_2) | z_1, z_2 \in Z\}$ over the pointers in Z, and $\{\texttt{test}_p(\vec{z}) | \vec{z} \in Z^{ar(p)}\}$ over the list of pointers in $Z^{ar(p)}$ for each predicate $p \in \Psi$ in a given planning domain $\mathcal{D}$ [2]. Here is a brief description of each of these instructions:

- $\texttt{inc}(z_1)$ increments the pointer by 1 if $z_1 < |\Omega| - 1$. Otherwise, this instruction is not applicable.
- $\texttt{dec}(z_1)$ decrements the pointer by 1 if $z_1 > 0$. Otherwise, this instruction is not applicable.
- $\texttt{clear}(z_1)$ sets the pointer to 0.
- $\texttt{set}(z_1, z_2) | z_1, z_2 \in Z$ sets the value of pointer z_2 to the value of pointer z_1, i.e., make z_2 point to the same object indexed by z_1.
- $\texttt{test}_p(\vec{z}) | \vec{z} \in Z^{ar(p)}$ returns the interpretation of $p(\vec{z})$ at the current state. Equivalently to the execution model mentioned before, this first requires mapping the pointers to their indexed objects, such that we finally get a grounded atom $p(\vec{o})$.

Finally, *goto* instructions of planning programs can no longer be conditioned by an arbitrary proposition y. Instead, they are now restricted to be conditioned by a single boolean y_z dedicated to storing the outcome of the last executed pointer

[2] $ar(p)$ denotes the arity of predicate $p \in \Psi$, where the arity is the number of arguments of the predicate.

operation primitive. This change makes the solution space tractable regardless of instance size. Below is the formal definition of y_z:

$$y_z \equiv \begin{cases} \text{False}, & \text{if applicable } w_i = \texttt{inc}\,(z_1)\,, \\ (z_1 == 1)\,, & \text{if applicable } w_i = \texttt{dec}\,(z_1)\,, \\ \text{True}, & \text{if } w_i = \texttt{clear}\,(z_1)\,, \\ (z_2 = 0)\,, & \text{if } w_i = \texttt{set}\,(z_1, z_2)\,, \\ \neg p(\vec{z}), & \text{if } w_i = \texttt{test}_p(\vec{z}), \\ \text{True}, & \text{if inapplicable } w_i. \end{cases}$$

2.5 The BFGP algorithm

This section presents the fundamentals of the BFGP algorithm for generalized planning, first introduced by Segovia-Aguas et al. [2021], and later covered in more detail by Segovia-Aguas et al. [2024]. This algorithm uses a planning as heuristic search approach (see Appendix A) in the space of planning programs of n lines and $|Z|$ pointers. The combinatorial search is guided by novel native heuristics for GP, so we will first cover those and then delve into the algorithm itself.

2.5.1 Heuristic functions for generalized planning

There are two sources of information for evaluating the fitness of a planning program that should solve a GP problem $\mathcal{P}$. The first one is its *structure*, and the second one is its ability to solve the classical planning instances $P_t, 1 \leq t \leq T$ of $\mathcal{P}$, also known as the *performance* of the program. We will only introduce the ones used in this work.

Program structure Given a *partially specified planning program* Π (i.e., a planning program of n lines where not all the lines are programmed yet), the following evaluation functions are defined:

- $f_{lc}(\Pi)$, the number of *goto* instructions in Π.
- $f_{ilc}(\Pi)$, the negated number of *goto* instructions in Π (i.e., $-f_{lc}$).
- $f_{mri}(\Pi)$, the number of repeated actions in Π.

Program performance These functions need to execute the current, partially specified, planning program Π on all classical instances $P_t, 1 \leq t \leq T$ of $\mathcal{P}$ to be evaluated. They measure the potential of a given program to

solve a GP problem Π. However, they are much more computationally expensive than evaluation functions based on the program structure (which can be computed linearly in the size of the program).

- $f_{ed}(\Pi, \mathcal{P}) = \sum_{P_t \in \mathcal{P}} f_{ed}(\Pi, P_t)$, where

$$f_{ed}(\Pi, P_t) = \sum_{x \in X_t} (v_x - G_t(x))^2$$

  Here, $v_x \in D_x$ is the value for state variable $x \in X_t$, after executing Π on the classical planning instance $P_t \in \mathcal{P}$, and $G_t(x)$ is the goal value of this state variable. Note that grounded predicates are boolean state variables, but nothing prevents to have functions over objects with integer domain [Segovia-Aguas et al., 2021].

It is essential to highlight that all these functions are cost functions, so the lower their value, the better a program is considered.

2.5.2 Best first search for generalized planning

The BFGP algorithm starts with an *empty planning program* of n lines, where the first $n-1$ lines are undefined, and the last line is an end instruction. In order to generate child nodes, only the PC^{MAX} line (i.e., the maximum line reached after executing the current program on the classical planning instances in $\mathcal{P}$) is allowed to be programmed. With this approach, two key properties are achieved: the set of successor nodes of any given node is tractable, and it is guaranteed not to generate duplicate successor nodes.

Consistent with this design, BFGP is a *frontier search* algorithm, meaning that only an open list of generated nodes is kept in memory, while the closed list of expanded nodes is not stored. This decision translates into a high degree of memory efficiency. BFGP sequentially expands the most promising node, which identifies a partially specified planning program. A priority queue, ordered with the evaluation of the heuristic functions defined in subsection 2.5.1, is used to determine which one of the nodes is the most promising. When evaluating the *performance* of a partially specified program Π, if the execution of Π fails on any of the instances P_t of $\mathcal{P}$, it means that the node is a dead-end and it is not added to the open list of generated nodes. On the contrary, if the execution of a program Π solves all the instances of $\mathcal{P}$, then it means that Π is a valid solution of the generalized planning problem $\mathcal{P}$ and the search can be terminated.

Chapter 3

PARALLEL SEARCHING STRATEGIES FOR TREES

In the BFGP algorithm (see Section 2.5), the search initiates with an empty program, and the successor function expands nodes in a way that guarantees that all generated partial programs are new (i.e., they will not be generated again in future expansions). Thus, BFGP is a frontier-search algorithm [Korf et al., 2005] applied to a tree-structured graph where each node is a partial program. These types of algorithms do not require a closed list for detecting visited nodes in the search, so the only source of shared memory is the open list, which incurs high memory and time savings. This property makes the algorithm very suitable for parallelism.

This chapter focuses on tree-based search problems to reproduce the frontier search property of the BFGP algorithm and how different parallel strategies in a multi-core CPU architecture may impact the average performance of several benchmarks.

3.1 Tree-based searching problems

Definition 1 (Rooted tree) *A* rooted tree *is a connected acyclic graph where one vertex has been designated as the* root *and all the edges have been oriented away from the root, thus obtaining a directed graph. In a rooted tree, if* (u, v) *is an edge, we will say that* u *is the* parent *of* v *and that* v *is a* child *of* u*. A* leaf *is a vertex with no children. The* depth *of a node in a rooted tree is the path length from the root to this node. The* branching factor *is the number of children at each node.*

A tree-based searching *problem* can be defined by the tuple $\langle T, \mathcal{G} \rangle$, where $T = (V, E, r)$ is a rooted tree with a set of nodes V, a set of edges E and root $r \in V$, and $\mathcal{G}$ is a goal test function. The tree of a problem can be specified

either explicitly or implicitly. We say that the tree is *explicitly defined* if the set of nodes V and the set of edges E are finite and fully known. On the other hand, we say that a tree is *implicitly defined* if V and E are not known, but we have a successor function $\sigma(u) = \{v \in V | (u, v) \in E\}$ that given a node u, it returns the set of children of the node. The goal test function $\mathcal{G} : V \to \{\texttt{true}, \texttt{false}\}$ is a predicate that takes a node $v \in V$ as input and returns `true` if v satisfies the search criteria (i.e. if v is a goal node) and `false` otherwise. The search criteria can be based on various properties of the nodes, such as:

Value-based criteria Find a node $v \in V$ such that $f(v) = t$, where f is a function assigning values to nodes and t is a target value.

Structural criteria Find a node $v \in V$ that satisfies a certain structural property, such as being a leaf node or having a specific depth.

A *solution* to a tree-based searching problem is a path $P_g = \langle v_0, ..., v_k \rangle$ of length $k \geq 0$ such that $v_0 = r$ is the root of the tree and v_k satisfies the search criteria (i.e., $\mathcal{G}(v_k) = \texttt{true}$). If no node $v_g \in V$ exists such that $\mathcal{G}(v_g) = \texttt{true}$, then a failure should be reported.

In tree-based searching problems, the solution is commonly computed with tree-traversal algorithms. These algorithms sequentially visit all the nodes of a tree, starting from the root. We say that a tree-traversal algorithm is *complete* if it guarantees finding a solution when there is one and correctly reporting failure when not. Furthermore, a tree-traversal algorithm is *sound* if it can only output correct solutions (i.e., if the algorithm outputs a solution, it is guaranteed to be valid).

3.2 Parallel execution model

Parallel searching strategies involve executing multiple search operations simultaneously to reduce computation time [Buluç and Madduri, 2011]. A simultaneous execution can be accomplished with various parallelization paradigms. In this work, we have focused on leveraging *multi-core CPU architectures* and *multi-threading*. However, some of the presented ideas could also be extended to distributed systems or GPU computing. Below, we introduce some basic concepts of this parallel execution model.

A *multi-core processor* (also known as a chip multi-processor) is an integrated circuit to which two or more processing units, called *cores*, have been attached for enhanced performance, reduced power consumption, and more efficient simultaneous processing of multiple tasks [Sirhan, 2020]. Each of the cores is capable of reading and executing program instructions simultaneously.

In computer science, a *thread* is the smallest sequence of ordered programmed instructions that can be managed independently by an operating system scheduler [Lamport, 1979]. A thread typically belongs to a particular process (an instance of a computer program), and all the threads that belong to the same process share a set of resources, such as memory. *Multithreading* is the ability of a CPU (or a single core in a multi-core processor) to concurrently execute instructions of different threads [Ungerer et al., 2002]. In the case of multi-core processors, multithreading allows the execution of multiple threads in parallel [1].

A *task* is the abstraction of a piece of work that could potentially be done concurrently. Threads allow for concurrent execution of tasks, enabling multiple program parts to run in parallel in multi-core processors. In the context of tree search algorithms, threads can be used to explore different branches of the tree simultaneously, significantly speeding up the search process.

3.3 Parallel searching strategies

The basic idea of a parallel search algorithm for (rooted) trees is to explore multiple branches simultaneously (e.g., exploring one different branch in each core of a multi-core processor). The more branches are explored in parallel, the faster a solution will be reached. Note that in a tree-based searching problem, we can use any successor of the root of the original tree as the root of a new *subtree*, which, in conjunction with all of its successors, defines a subproblem of smaller size.

Several details are noteworthy when designing a parallel search algorithm for trees in a multithreaded environment. The first is that it is unnecessary to check for infinite loops since a tree does not have cycles by definition. Consequently, there is no need to keep a shared list of expanded nodes, which saves a lot of potential synchronization complexity between threads. The second, and perhaps most important, detail to keep in mind is that a proper distribution of work is vital in achieving good performance results. A tree search starts with a bottleneck, in that we only have one node, the root node, from where to start the search. Furthermore, different tree branches can wildly differ in depth and branching factor, making the workload of threads potentially unbalanced. The last detail about trees that should be emphasized is the possible existence of multiple goals. In the case

[1] It is necessary to distinguish *concurrency* and *parallelism*. Concurrency refers to executing instructions of different threads in an overlapping time period (i.e. out of order). For example, given two threads with four instructions each, a concurrent execution could be to execute the first two instructions of thread one, then all the instructions of thread two, and finally the remaining two instructions of thread one. In contrast, a sequential execution would imply that *all* the instructions of one of the two threads must be executed before the execution of the other thread. Parallelism refers to the *simultaneous* execution of multiple instructions of different threads, which is only possible in multi-core CPUs.

that more than one node satisfies the goal conditions, special care is required to determine if all goal nodes are equally valid or if, on the contrary, we are only interested in those that we consider to be optimal in a particular regard.

The biggest challenge is to devise an algorithm robust enough to handle all sorts of unbalanced trees without a noticeable performance hit. In order to overcome it, we have worked on two different approaches:

- A static workload distribution, where we first expand multiple successors from the root (sequentially) and then start a parallel search once we have a reasonable number of starting nodes for each thread.

- A dynamic workload distribution based on how busy each thread is.

The first strategy is based on the statistical assumption that the more starting nodes (or branches) each thread has, the more likely its workload will be balanced compared to other threads. For this to be true, the starting nodes of all threads should be at an equal or similar depth. This approach does not require synchronization between threads, which is usually the most computationally costly part of parallel algorithms. However, its performance can be undermined in worst-case scenarios where the search tree is really unbalanced. Furthermore, some empirical tuning is required to find the right balance for determining the correct number of starting nodes, as the more starting nodes are generated, the more time the search is being executed sequentially.

The second strategy is much more complex and requires synchronization by construction, but the added cost could pay off in large or slow-to-explore trees. Therefore, when designing an algorithm with this approach in mind, the goal is to ensure that the additional cost of synchronization does not outweigh the performance benefit of having a better distribution of work. There are two main alternatives for this strategy:

Work stealing pattern All threads generate their own work and can request (or steal) work from other threads when their work queue empties.

Master-worker pattern The master thread is responsible for distributing work among worker nodes, which are in charge of executing the work. Worker threads can not communicate between them.

In the following subsections, we will describe the proposed static and dynamic work distribution strategies in more detail.

Algorithm 1 BFS until frontier size reaches a limit (`BFS_WITH_LIMIT`)

Input: GP Problem P, frontier queue q, desired number of opened nodes n.

Output: Solution to problem P, or `NULL` if no solution is found. Frontier q is updated to have at least n opened nodes.

```
1: while not IS_EMPTY(q) and SIZE(q) < n do
2:     node ← FRONT(q)
3:     POP(q)
4:     if IS_GOAL(node, P) then
5:         return node
6:     end if
7:     children ← EXPAND(node, P)
8:     for all child in children do
9:         PUSH(child, q)
10:    end for
11: end while
12: return NULL
```

3.3.1 Auxiliary algorithms

First, we define two partial modifications to the Best-First Search algorithm that we have found helpful in a multithreading-capable system. Algorithm 1 is a straightforward modification to BFS that stops the search if the frontier queue reaches a specific size. We use this algorithm to generate a set of multiple starting nodes from which to start the search.

Algorithm 2 BFS that can be externally interrupted (`INTERRUPTIBLE_BFS`)

Input: GP Problem P, initial frontier q, shared atomic stop token st.

Output: Solution to problem P, or `NULL` if no solution is found. It uses st to signal that a solution has been found.

```
1: while not IS_EMPTY(q) and not STOP_REQUESTED(st) do
2:     node ← FRONT(q)
3:     POP(q)
4:     if IS_GOAL(node, P) then
5:         REQUEST_STOP(st)
6:         return node
7:     end if
8:     children ← EXPAND(node, P)
9:     for all child in children do
10:        PUSH(child, q)
11:    end for
12: end while
13: return NULL
```

Algorithm 2 is also a minor modification of BFS. INTERRUPTIBLE_BFS allows the external termination of the search by using a shared atomic flag. It is specially useful to stop the search once a thread has already found a solution.

Algorithm 3 Static work distribution of work with one asynchronous task per starting node (ASYNC_NODE_BFS)

Input: GP Problem P, number of starting nodes N.
Output: Solution to problem P, or NULL if no solution is found.

```
1: q ← queue with ROOT(P) as its only element
2: possible_solution ← BFS_WITH_LIMIT(P, q, N)
3: if possible_solution ≠ NULL then
4:     return possible_solution
5: end if
6: st ← create stop token
7: futures ← empty vector of futures
8: while not IS_EMPTY(q) do
9:     node ← FRONT(q)
10:    POP(q)
11:    future ← ASYNC(INTERRUPTIBLE_BFS, P, node, st)
12:    PUSH(future, futures)
13: end while
14: solution ← NULL
15: for all future in futures do
16:    result ← GET_RESULT(future)
17:    if result ≠ NULL then
18:        solution ← result
19:    end if
20: end for
21: return result
```

3.3.2 Static work distribution algorithm

Here, we present the first of our parallel search strategies for rooted trees. As mentioned earlier, the main idea is to split the tree search into two steps:

1. Start a sequential BFS until the frontier has a specific number of opened nodes yet to be expanded. We want this step to be as short as possible while ensuring that the number of starting nodes generated allows for a reasonably balanced distribution of work.

2. Distribute the starting nodes among a pool of threads and start a parallel search.

For step two, we propose two alternatives. The first alternative is to launch a concurrent task for each starting node. With this approach, we can use modern asynchronous programming features and Operating System (OS) scheduling. Instead of creating a separate thread for each starting node, we can let the OS handle the job of choosing how many threads to use to solve all tasks as fast as possible. In Algorithm 3, we showcase this procedure. The second alternative is to specify a desired number of threads and evenly divide all starting nodes into separate subqueues, one for each thread. It is a lower-level approach, but as an advantage, it allows each thread to prioritize the most promising nodes of its starting frontier (in case of having a heuristic search). Algorithm 4 describes this second alternative.

Algorithm 4 Static work distribution with multiple starting nodes per asynchronous task (`ASYNC_QUEUES_BFS`)

Input: GP Problem P, number of threads T, number of start nodes per thread n.
Output: Solution to problem P, or `NULL` if no solution is found.

$q \leftarrow$ queue with `ROOT`(P) as its only element
$possible\_solution \leftarrow$ `BFS_WITH_LIMIT`(P, q, n)
if $possible\_solution \neq$ `NULL` **then**
    return $possible\_solution$
end if
$st \leftarrow$ create stop token
$threads \leftarrow$ create vector of T threads
$futures \leftarrow$ empty vector of futures (that will hold the output of the threads)
$init\_queues \leftarrow$ divide q into T, smaller queues, s.t. each queue has n starting nodes
for $i = 0$ **to** $T - 1$ **do**
    $task \leftarrow$ `CREATE_TASK`(`INTERRUPTIBLE_BFS`, $P, init\_queues[i], st$)
    $future \leftarrow$ `LAUNCH_THREAD`$(threads[i], task)$
    `PUSH`$(future, futures)$
end for
$solution \leftarrow$ `NULL`
for all $future$ **in** $futures$ **do**
    $result \leftarrow$ `GET_RESULT`$(future)$
    if $result \neq$ `NULL` **then**
        $solution \leftarrow result$
    end if
end for
return $solution$

3.3.3 Dynamic work distribution algorithm

Here, we present our second parallel search strategy for rooted trees. We have decided to implement a master-worker architecture since it is easier to design due

to less complex synchronization requirements. A work-stealing architecture, in which all threads can dynamically share work between them, is left as a possible future development.

Algorithm 5 Dynamic work distribution (DYNAMIC_BFS)

Input: GP Problem P, number of threads T, number of start nodes N.
Output: Solution to problem P, or NULL if no solution is found.

```
 1: q ← queue with ROOT(P) as its only element
 2: possible_solution ← BFS_WITH_LIMIT(P, q, N)
 3: if possible_solution ≠ NULL then
 4:    return possible_solution
 5: end if
 6: workers ← start T − 1 workers with one node of q for each
 7: while not SOLUTION_FOUND() and not IS_EMPTY(q) do
 8:    possible_solution ← BFS_WITH_LIMIT(P, q, SIZE(q) + 1)
 9:    if possible_solution ≠ NULL then
10:       return possible_solution
11:    end if
12:    for all worker in workers do
13:       success ← TRY_GIVE_WORK(worker, FRONT(q))
14:       if success then
15:          POP(q)
16:       end if
17:    end for
18: end while
19: SIGNAL_MASTER_FINISHED()
20: wait for all worker threads to finish
21: for all worker in workers do
22:    possible_solution ← GET_RESULT(worker)
23:    if possible_solution ≠ NULL then
24:       return possible_solution
25:    end if
26: end for
27: return NULL
```

Algorithm 5 provides a high-level overview of the work performed by the master thread. It starts by expanding the root node and its successors until it has a main queue of $N > T - 1$ starting nodes. Then, it creates $T - 1$ worker threads, giving each of them one of the previously opened nodes to start the search. Next, while no solution is found, the master thread keeps expanding the remaining nodes of the main queue. At the same time, it checks if any worker thread needs more work, distributing it accordingly. Once the main thread has no more nodes to expand, it signals no more work is available. Then, the worker threads continue

until they finish their respective work or until one of them finds a solution.

Each worker thread has a private queue protected by a *mutex*[2]. While a thread has nodes to expand, its private queue is locked, and any attempt to add additional work is rejected. Once the private queue is empty, the mutex is unlocked, and the thread waits to receive one additional search node from the master thread. Each worker thread continues this process until a solution is found or until both the master thread's queue and its private queue are empty.

[2]A mutex, a "mutual exclusion object," is a synchronization mechanism used to make regions of code accessible by only one thread at a time.

3.4 Results

In this section, we present the results of multiple experiments designed to evaluate the performance of the proposed parallel tree-search algorithms:

- `ASYNC_NODES_BFS` (Algorithm 3)
- `ASYNC_QUEUES_BFS` (Algorithm 4)
- `DYNAMIC_BFS` (Algorithm 5)

All experiments are performed in a Fedora Linux 39 computer, with an Intel® Core™ i7-12700 12-core (8 performance cores with SMT and 4 efficient cores) processor and 32 GB of RAM. For all algorithms, the number of starting nodes generated sequentially is set to four times the number of hardware threads ($N = 4 \times 20$ for Algorithm 3 and Algorithm 5 and $n = 4 \times 20$ for Algorithm 4). [3]

The experiments solve multiple randomly generated tree-search problems with all of our algorithms and then compare the execution time with a sequential BFS. Each toy search problem is defined by an initial state, a randomly generated tree of states, and a random set of goal states. A sequence of unsigned integer numbers represents the search problem states (the tree nodes), and the children of a node are generated by adding a unique random number to the sequence of the parent, which guarantees all nodes are unique. The root of the tree is the empty sequence. Since the goal set is random, not all problems have a solution. Once a problem has been created, it is written to a file in order to ensure that all algorithms solve the same problems. Each experiment evaluates the performance of the algorithms for different types of trees.

Given the simplistic nature of our toy problems, we decided not to use any heuristic function to guide the parallel search. A uniform-cost approach is more straightforward to implement and should keep relative performance results the same. We provide the results of the experiments in the following subsections.

3.4.1 Performance for different branching factors

A rooted tree's *branching factor* is the number of children at each node. If this value is not uniform, we will consider the average number of children at each node. The larger the branching factor is, the more significant the exponential growth of a tree is at each added layer. A tree with a uniform branching factor of b and depth d has b^d nodes.

[3] The source code, evaluation scripts, and used benchmarks can be found at: `https://github.com/Alejandro-FA/parallel_bfs`

In this experiment, the branching factor B of each node is determined by a binomial distribution $B \sim Bin(n, p)$, with $n = 10$ and $p = b/n$, where b is the desired average branching factor for the whole tree. Table 3.1 compares the performance of our parallel algorithms for trees with depth 7 and an average branching factor ranging from 2 to 8. Algorithm 3 (`ASYNC_NODES_BFS`) has the best performance for branching factors larger than 4.0, although Algorithm 5 (`DYNAMIC_BFS`) is not far behind.

Algorithm	$b = 2.0$			$b = 3.0$			$b = 4.0$		
	Avg	Mdn	SD	Avg	Mdn	SD	Avg	Mdn	SD
Sequential	**0.0768**	**0.06**	**0.0687**	0.9228	0.80	0.581	6.14	5.53	2.90
Async nodes	0.767	1.18	0.588	1.23	1.24	0.272	1.59	1.45	**0.335**
Async queues	0.339	0.33	0.0811	**0.427**	**0.395**	**0.119**	**1.16**	**1.03**	0.475
Dynamic	0.279	0.37	0.205	0.502	0.485	0.169	2.15	1.84	1.56
	$b = 5.0$			$b = 6.0$			$b = 7.0$		
Sequential	25.4	25.2	10.3	84.6	85.1	31.5	202	203	45.2
Async nodes	**2.74**	**2.73**	**1.03**	**5.41**	**5.06**	**2.91**	**13.4**	**11.9**	5.61
Async queues	4.22	4.26	1.54	13.3	13.7	4.02	37.2	37.5	9.07
Dynamic	6.11	4.02	1.54	9.86	8.87	8.70	19.9	18.9	**3.98**
	$b = 8.0$								
Sequential	484	503	84.3						
Async nodes	**25.5**	**22.9**	9.99						
Async queues	107	106	30.1						
Dynamic	32.6	32.1	**6.38**						

Table 3.1: We report for each branching factor b, mean CPU execution time (ms), median CPU execution time (ms), and standard deviation (ms) for 50 different randomly generated trees. The best results are in bold.

3.4.2 Performance for different goal set sizes

As mentioned in previous sections, a tree can have multiple goal states. Table 3.2 compares the performance of our algorithms for different sizes of goal sets. There are two noteworthy results to discuss: the first one is that Algorithm 5 (`DYNAMIC_BFS`) performs very poorly for small goal sets, and the second one is that parallel strategies improve their performance with large goal sets more than what a sequential search does. In this experiment, Algorithm 3 (`ASYNC_NODES_BFS`) performs the best across the board.

Algorithm	$\|G\| = 1$			$\|G\| = 10$			$\|G\| = 10^2$		
	Avg	Mdn	SD	Avg	Mdn	SD	Avg	Mdn	SD
Sequential	564	613	178	526	491	202	356	380	97.6
Async nodes	**60.1**	**62.2**	**20.2**	**54.8**	**54.1**	**22.0**	**28.5**	28.5	**14.3**
Async queues	66.7	65.5	22.2	66.0	65.0	23.2	55.9	56.2	14.4
Dynamic	327	361	162	249	232	201	30.3	**22.7**	41.9
	$\|G\| = 10^3$			$\|G\| = 10^4$			$\|G\| = 10^5$		
Sequential	345	355	102	344	366	102	375	383	110
Async nodes	**13.7**	**12.1**	**6.50**	**9.52**	**7.99**	**4.98**	**9.79**	**9.33**	5.48
Async queues	49.6	52.2	14.2	45.4	49.6	14.8	44.6	46.3	14.9
Dynamic	18.9	17.2	6.97	15.2	14.7	6.14	13.7	13.5	**5.15**

Table 3.2: We report for each goal set size $|G|$, mean CPU execution time (ms), median CPU execution time (ms), and standard deviation (ms) for 50 different randomly generated trees. The best results are in bold.

3.4.3 Performance for different goal depths

Our testing framework also allows us to specify at which depth goal states are generated. The deeper the goals are, the more nodes the algorithms need to expand. Table 3.3 compares the performance of our parallel algorithms for different depths. The results are not of particular interest; at low depths, the search can be completed very fast, and at high depths Algorithm 3 (`ASYNC_NODES_BFS`) has the best performance. The higher the depth at which goals are found, the better the performance of parallel algorithms is compared to a sequential Best-First Search.

3.4.4 Performance for different levels of imbalance

As mentioned in previous sections, a tree's balance is a property that can have a massive impact on performance. An unbalanced tree could have very shallow branches and very deep branches, or branches with very few children and branches with many children. To test the performance of our algorithms with moderately unbalanced trees, we have determined the number of children of each node by a uniform distribution $B \sim U(a, b)$ (where a is the lower bound of the uniform distribution and b is the upper bound). Given a fixed average branching factor $\bar{B} = \frac{a+b}{2}$, unbalanced trees could be obtained if the difference between the minimum possible number of children a and the maximum number of children b is very large because the variance of the branching factor $\mathrm{Var}[B] = \frac{(b-a+1)^2-1}{12}$ would increase.

Table 3.4 shows the results obtained for different values of a and b for an

Algorithm	$d = 6$			$d = 8$			$d = 10$		
	Avg	Mdn	SD	Avg	Mdn	SD	Avg	Mdn	SD
Sequential	**0.198**	**0.185**	1.129	1.34	1.28	0.825	7.66	6.82	3.89
Async nodes	1.16	1.17	0.266	1.30	1.23	0.364	1.55	1.42	**0.679**
Async queues	0.375	0.350	**0.149**	**0.459**	0.43	**0.0981**	1.81	1.74	0.812
Dynamic	9.32	0.355	62.5	1.44	**0.395**	7.00	**1.23**	**0.985**	1.20
	$d = 12$			$d = 14$					
Sequential	63.7	62.5	33.7	426	432	166			
Async nodes	**5.09**	**3.72**	**4.07**	**36.4**	**37.5**	**16.9**			
Async queues	10.2	9.32	5.29	52.5	56.6	21.2			
Dynamic	12.2	6.60	18.3	88.8	35.1	106			

Table 3.3: We report for each goal depth d, mean CPU execution time (ms), median CPU execution time (ms), and standard deviation (ms) for 50 different randomly generated trees. The best results are in bold.

average branching factor $\bar{B} = 6$. Following the trend of previous experiments, Algorithm 3 (`ASYNC_NODES_BFS`) is the one that performs best. Furthermore, taking into account that Algorithm 5 (`DYNAMIC_BFS`) theoretically has a finer-grained workload distribution, it performs worse than expected. However, it is important to note that all algorithms have a noticeable performance decay for large differences between a and b.

3.4.5 Performance for computationally costly search problems

One limitation of our testing framework is that trees are quickly explored since no computationally costly operations are involved (we are only comparing integer numbers). To better understand how the algorithms would perform in complex problems, we have also measured the performance when we add an artificial delay to the goal-checking function.

Table 3.5 compares the results for different delays. These results show that the fine-grained workload distribution of Algorithm 5 (`DYNAMIC_BFS`) can compensate for the additional synchronization overhead for computationally costly search problems, and it is the best-performing strategy for delays larger than 20 microseconds. However, Algorithm 3 (`ASYNC_NODES_BFS`) is faster for inexpensive problems. It is important to note, though, that in a best-case scenario, Algorithm 5 only achieves a 35% decrease in average execution time with respect to Algorithm 3.

Algorithm	$a = 5, b = 7$			$a = 4, b = 8$			$a = 3, b = 9$		
	Avg	Mdn	SD	Avg	Mdn	SD	Avg	Mdn	SD
Sequential	413	429	82.3	453	455	140	515	510	188
Async nodes	**20.8**	**20.2**	**8.71**	**27.2**	**26.7**	**12.0**	**43.0**	**38.6**	**18.7**
Async queues	72.7	71.9	20.1	87.3	70.8	38.8	98.1	85.3	48.5
Dynamic	48.0	46.6	10.7	45.5	47.6	13.5	55.6	54.0	22.5
	$a = 2, b = 10$			$a = 1, b = 11$			$a = 0, b = 12$		
Sequential	630	579	322	836	763	467	879	762	530
Async nodes	**58.6**	**57.0**	**32.0**	**81.9**	**82.6**	**43.8**	**92.8**	**87.4**	**52.7**
Async queues	107	82.3	63.7	138	114	87.6	147	136	91.3
Dynamic	140	65.5	158	354	223	364	361	254	319

Table 3.4: We report for each branching factor distribution $B \sim U(a, b)$, mean CPU execution time (ms), median CPU execution time (ms), and standard deviation (ms) for 50 different randomly generated trees. The best results are in bold.

Algorithm	$\Delta t = 0$			$\Delta t = 5$			$\Delta t = 10$		
	Avg	Mdn	SD	Avg	Mdn	SD	Avg	Mdn	SD
Sequential	9.26	9.48	3.01	64.8	64.1	18.9	120	119	34.5
Async nodes	1.55	1.32	**0.506**	**1.79**	**1.65**	**0.350**	2.59	2.49	**0.689**
Async queues	2.03	1.91	0.588	4.11	4.14	0.962	6.13	6.38	1.28
Dynamic	**0.910**	**0.730**	0.549	1.88	1.75	0.513	**2.47**	**2.27**	0.758
	$\Delta t = 20$			$\Delta t = 50$			$\Delta t = 100$		
Sequential	230	227	66.0	560	553	160	1110	1097	317
Async nodes	**3.87**	**3.64**	**1.09**	13.2	12.8	5.12	24.0	22.7	10.1
Async queues	10.4	10.6	2.17	23.1	24.1	4.87	44.3	45.3	9.14
Dynamic	4.09	3.77	1.30	**8.63**	**7.97**	**3.07**	**16.1**	**14.8**	**5.91**
	$\Delta t = 200$			$\Delta t = 500$					
Sequential	2210	2184	631	5512	5447	1573			
Async nodes	49.0	42.9	20.0	118	101	49.5			
Async queues	85.9	88.8	18.4	212	219	45.4			
Dynamic	**31.5**	**29.5**	**11.5**	**76.8**	**72.1**	**28.0**			

Table 3.5: We report for each delay Δt $(\mu\mathrm{s})$, mean CPU execution time (ms), median CPU execution time (ms), and standard deviation (ms) for 50 different randomly generated trees. The best results are in bold.

Chapter 4

PARALLEL BEST-FIRST GENERALIZED PLANNING

This chapter applies the parallel searching strategies discussed in Chapter 3 to BFGP. In particular, we have decided to parallelize the BFGP++ framework, an iteration over BFGP that changes `goto` instructions by `for` and `if` instructions, and the synthesized plans use C++ grammar [Segovia-Aguas et al., 2022b].[1]

4.1 BFGP searching problems

BFGP search trees have some particularities compared to the toy trees we have used in Chapter 3 to evaluate parallel Best-First Search algorithms. First of all, they have a much more significant branching factor (in some cases it can go upwards of 50 children per node). This increase in size is possible because BFGP explores search trees procedurally, so it does not need to fit them in memory all at once. Secondly, BFGP performs a heuristic search instead of a uniform-cost search, which implies that some nodes are given a higher priority than others. Finally, BFGP prunes[2] any node (i.e., partially specified program) that already fails to solve at least one of the instances of the generalized planning problem [Segovia-Aguas et al., 2024]. These differences imply three considerations:

- There is a high likelihood of having unbalanced trees
- To have an even and efficient distribution of work, we need a similar amount of nodes with a similar evaluation of the heuristic function. It is better to

[1] The source code of BFGP++, the C++ implementation of the BFGP algorithm by the same authors can be found at: `https://github.com/jsego/bfgp-pp`. The parallelized version presented in this work can be found at `https://github.com/Alejandro-FA/bfgp-pp`.

[2] Pruning is the action of discarding a whole branch of the search tree without evaluating it.

spread promising nodes into multiple threads than to have all of them in a single thread.

- Given that search trees are larger, a poor distribution of work might impact the performance more negatively.

However, BFGP also has some fundamental similarities with the problems of Chapter 3. Firstly, BFGP is a frontier search algorithm (as explained in subsection 2.5.2), so we do not need to keep a closed list of expanded nodes to check for cycles. Furthermore, BFGP is a greedy algorithm designed to return the first solution found to a generalized planning problem, which is not necessarily the most optimal one. Therefore, our parallel algorithm can stop execution once one of the threads finds a solution.

Algorithm 6 Interruptible BFGP (`INTERRUPTIBLE_BFGP`)

Input: GP Problem $\mathcal{P}$ (which includes pointers Z, program lines n, and a list of evaluation functions $\mathcal{F}$), frontier (priority queue) $Open_{\mathcal{F}}$, frontier size limit l, stop token st.

Output: A generalized plan Π that solves $\mathcal{P}$, or `NULL` if no solution is found. Frontier $Open_{\mathcal{F}}$ is updated with its state at the end of the execution.

```
1:  while not IS_EMPTY(Open_F) and SIZE(Open_F) < l do
2:      Π ← EXTRACT_BEST_PROGRAM(Open_F)
3:      ChildrenPrograms ← EXPAND_PROGRAM(Π)
4:      for all Π' in ChildrenPrograms do
5:          if STOP_REQUESTED(st) then
6:              return NULL
7:          else if IS_DEAD_END(Π', P) then
8:              continue
9:          else if IS_GOAL(Π', P) then
10:             return Π'
11:         end if
12:         Open_F ← INSERT_PROGRAM(Open_F, Π')
13:     end for
14: end while
15: return NULL
```

4.2 Parallel BFGP algorithm

As in Chapter 3, we first modify the single-threaded BFGP so that it can be interrupted either when the frontier priority queue (called $Open_{\mathcal{F}}$) reaches a specific size limit or when another thread has sent a termination signal (because it has already found a solution). This time, we merge both Algorithm 1 and Algorithm 2

into a single algorithm to avoid code duplication. Algorithm 6 shows this minor modification to BFGP. It is important to note that since BFGP uses the early goal test optimization, we need to adjust the location of the termination request check accordingly.

Algorithm 7 Parallel BFGP (`PARALLEL_BFGP`)

Input: GP Problem $\mathcal{P}$ (which includes pointers Z, program lines n, and a list of evaluation functions $\mathcal{F}$), number of threads T, and number of starting nodes per thread N.

Output: A generalized plan Π that solves $\mathcal{P}$, or `NULL` if no solution is found.

1: $Futures \leftarrow$ empty vector of futures
2: $QueueSizeLimit \leftarrow T \times N$
3: $st \leftarrow$ create stop token
4: **for** $i = 0$ **to** $T - 1$ **do**
5: $\quad gpp \leftarrow$ `DUPLICATE`$(\mathcal{P})$
6: $\quad future \leftarrow$ `ASYNC`$(\texttt{WORKER\_TASK}, gpp, T, i, QueueSizeLimit, st)$
7: $\quad$ `PUSH`$(future, futures)$
8: **end for**
9: $solution \leftarrow$ `NULL`
10: **for all** $future$ **in** $futures$ **do**
11: $\quad result \leftarrow$ `GET_RESULT`$(future)$ {Waits for the thread to finish}
12: $\quad$ **if** $result \neq$ `NULL` **then**
13: $\quad\quad solution \leftarrow result$
14: $\quad$ **end if**
15: **end for**
16: **return** $solution$

For the parallel search algorithm, we have implemented a mixture of Algorithm 3 and Algorithm 4. We would have also liked to explore dynamic workload distribution alternatives (like Algorithm 5) but could not due to time constraints. The current C++ implementation of the BFGP framework was designed with single-threaded environments in mind, and making the necessary modifications to implement parallel algorithms capable of dynamically distributing nodes from one thread to another (in a thread-safe manner) would require a major refactoring of the code. Luckily, it is possible to combine Algorithm 3 and Algorithm 4 such that each thread is entirely independent and generated nodes are contained within their threads. Algorithm 7 showcases our parallel BFGP algorithm, where `WORKER_TASK` is defined in Algorithm 8.

The distinctive feature of our parallel algorithm is how it distributes the workload. As suggested earlier in the section, the main limitation of the framework is that we can not move or copy search nodes from one thread to another (it would cause data races). Therefore, each thread needs to start the search from the be-

Algorithm 8 Worker task of parallel BFGP (`WORKER_TASK`)

Input: GP Problem $\mathcal{P}$ (which includes pointers Z, program lines n, and a list of evaluation functions $\mathcal{F}$), number of threads used T, thread index i, frontier size limit l, stop token st.

Output: A generalized plan Π that solves $\mathcal{P}$, or `NULL` if no solution is found.

1: $Open_{\mathcal{F}} \leftarrow \{\Pi_{empty}\}$
2: $PossibleSolution \leftarrow$ `INTERRUPTIBLE_BFGP`$(\mathcal{P}, Open_{\mathcal{F}}, l, st)$
3: **if** $PossibleSolution \neq NULL$ **then**
4: **return** $PossibleSolution$
5: **end if**
6: $QueueSize \leftarrow$ `SIZE`$(Open_{\mathcal{F}})$
7: $NewOpen_{\mathcal{F}} \leftarrow \{\Pi_{empty}\}$
8: **for** $j = 0$ **to** $QueueSize - 1$ **do**
9: $\Pi \leftarrow$ `EXTRACT_BEST_PROGRAM`$(Open_{\mathcal{F}})$
10: **if** $(j \bmod T) == i$ **then**
11: $NewOpen_{\mathcal{F}} \leftarrow$ `INSERT_PROGRAM`$(NewOpen_{\mathcal{F}}, \Pi)$
12: **end if**
13: **end for**
14: $l \leftarrow$ `INF` {Remove queue size limit}
15: **return** `INTERRUPTIBLE_BFGP`$(\mathcal{P}, NewOpen_{\mathcal{F}}, l, st)$

ginning. The process for ensuring that each thread explores a different part of the tree is divided into two steps:

1. Start the search from the root (empty program) until the frontier reaches a size limit. All threads will expand the same nodes in parallel, which is equivalent to performing this step in the main thread.

2. Remove items from the queue so each thread keeps different nodes. To perform this step, we use the modulo operation (see Algorithm 8), which not only ensures that no node will be kept in more than one queue but also ensures that high-priority nodes are spread among different threads.

Furthermore, it is also interesting to note that by combining Algorithm 3 and Algorithm 4, we get the advantage of having explicit control over how many threads are launched. This feature is especially useful from a user perspective since they have the control of specifying how many CPU resources are dedicated to the task.

4.3 Theoretical Properties

Lemma 1 (Termination) *Given a generalized planning problem $\mathcal{P}$, a finite set of pointers Z, and a finite number of program lines n,* `Parallel_BFGP` *(Algorithm 7) always terminates its execution.*

Proof. `Parallel_BFGP` is composed of two stages: (i) the creation of multiple asynchronous tasks and (ii), the aggregation/collection of the results produced by them. Therefore, if each asynchronous task terminates, so does the algorithm. Each task consists of executing Algorithm 8, which is characterized by two calls to `INTERRUPTIBLE_BFGP` (Algorithm 6). Given that the execution of the BFGP algorithm on a generalized planning problem $\mathcal{P}$, with a finite set of pointers Z, and a finite number of program lines n always terminates [Segovia-Aguas et al., 2024], `INTERRUPTIBLE_BFGP` always terminates since it is BFGP with two extra termination conditions (the size of the frontier and the status of the stop token). Since all asynchronous tasks are guaranteed to terminate (and return a value), `Parallel_BFGP` is guaranteed to terminate. □

Lemma 2 (Soundness) *If the execution of* `Parallel_BFGP` *on a GP problem $\mathcal{P}$ outputs a generalized plan Π, this means that Π is a solution for $\mathcal{P}$.*

Proof. `Parallel_BFGP` runs until all futures associated with the parallel execution of Algorithm 8 return a value. If at least one of the futures returns a solution, then `Parallel_BFGP` returns one of the solutions found. The only possible return values of Algorithm 8 are directly produced by `INTERRUPTIBLE_BFGP`. By construction, if `INTERRUPTIBLE_BFGP` returns a solution, it is a valid solution according to BFGP, and since BFGP is sound [Segovia-Aguas et al., 2024], it is also a valid solution of $\mathcal{P}$. Therefore, any solution output by `Parallel_BFGP` is valid. □

Lemma 3 (Completeness) *Given a GP problem $\mathcal{P}$, a maximum number of program lines n, and a maximum number of pointers $|Z|$, if there is a planning program Π within these bounds that solves $\mathcal{P}$, then* `Parallel_BFGP` *can compute it.*

Proof. The BFGP algorithm is complete [Segovia-Aguas et al., 2024]. Consequently, if Algorithm 8 ensures that the workload distribution does not lose any search node, then `Parallel_BFGP` will also be complete. During the generation of initial nodes, Algorithm 8 checks if any expanded nodes are already a goal, so no search nodes are lost in this phase. Furthermore, the modulo operation $j \bmod T$ is guaranteed to return a value x such that $0 \leq x \leq T-1$; therefore, all nodes are guaranteed to be assigned to a thread. Consequently, `Parallel_BFGP` is complete.

4.4 Results

This section presents the experiments we have used to evaluate the performance of PARALLEL_BFGP. First, we set a performance baseline with a single-threaded execution. Then, we rerun the same experiment using two, four, and eight threads to assess the performance scaling achieved with the parallelization.

The first experiment reproduces the results of Table 4 of Segovia-Aguas et al. [2024] but with the current (single-threaded) version of the BFGP++ framework and the C++ grammar. To do so, we evaluate the BFGP++ framework with the same 9 classical planning domains:

- *Corridor*, an agent moves from an arbitrary initial location to a destination location in a corridor.
- *Gripper*, a robot must pick all balls from room A and drop them in room B.
- *Visitall*, starting from the bottom-left corner of a squared grid, an agent must visit all cells.
- *Fibonacci*, compute the n^{th} term of the Fibonacci sequence.
- *Find*, counts the number of occurrences of a specific value in a list.
- *Reverse*, for reversing the content of a list.
- *Select*, find the minimum value of a list.
- *Sorting*, for sorting the values of a vector.
- *Triangular Sum*, compute the n^{th} triangular number.

For each domain, 10 randomly generated instances are used to define a generalized planning problem. In the *corridor* domain, instances go from corridors of length 3 to 12; in *gripper*, instances have from 2 to 11 balls in room A to be dropped in room B; in *visitall* instances are squared grids ranging from 2×2 to 11×11 cells; Fibonacci and triangular sum instances ranging from the 2^{nd} to the 11^{th} number in the sequence; and the remaining domains, *find*, *reverse*, *select* and *sorting* have instances with vectors from size 2 to 11 that are initialized with random content. In addition to using the equal domains, we also use the same combination of evaluation functions: f_{ilc} with f_{ed} to solve ties.

For the *corridor* domain we search for a program solution of 11 lines, for *gripper* 8 lines, for *visitall* 14 lines, for *Fibonacci* 7 lines, for *find* 4 lines, for *reverse* 7 lines, for *select* 7 lines, for *sorting* 9 lines and for *triangular sum* 5 lines. These number of lines are the lowest values for which a solution is found.

In the second experiment, we repeat the same tests but using our PARALLEL_BFGP algorithm with two, four, and eight threads. We also include the single-thread results of the previous section for comparison. Table 4.1 showcases the results for both experiments, so that they can be compared easily.

Domain	$n = 1$				$n = 2$			
	Time	Mem.	Exp.	Eval.	Time	Mem.	Exp.	Eval.
Corridor	3097.64	116	3.20M	3.21M	1726.83	210	3.41M	3.43M
Fibonacci	2.95	8.5	5.1K	13.5K	1.00	**5.9**	3.7K	6.7K
Find	**0.00**	**4.3**	**2**	**10**	**0.00**	4.5	**4**	**20**
Gripper	86.95	109	139K	366K	40.06	70.3	118K	260K
Reverse	0.54	5.4	1.7K	3.8K	0.28	5.3	1.7K	3.3K
Select	0.11	5.1	290	1.1K	0.06	**4.8**	**269**	**1.0K**
Sorting	**0.01**	**4.3**	**39**	182	0.02	4.5	62	219
Triangular Sum	0.01	**4.5**	16	98	**0.00**	4.6	**5**	17
Visitall	**344.28**	**12.3**	**147K**	**183K**	357.44	15.5	235K	276K
	$n = 4$				$n = 8$			
	Time	Mem.	Exp.	Eval.	Time	Mem.	Exp.	Eval.
Corridor	709.27	302	2.66M	2.85M	**27.41**	**107**	**351K**	**412K**
Fibonacci	**0.27**	6.1	**2.1K**	**3.5K**	0.38	9.1	5.1K	9.4K
Find	**0.00**	**4.3**	6	22	**0.00**	4.8	5	26
Gripper	20.99	76.1	113K	268K	**9.41**	**54.4**	**111K**	**213K**
Reverse	0.10	**5.1**	1.3K	2.3K	**0.06**	6.4	**1.1K**	**2.0K**
Select	**0.04**	5.4	385	1.4K	0.06	7.8	1.1K	2.2K
Sorting	**0.01**	4.6	69	171	**0.01**	5.1	161	**127**
Triangular Sum	0.01	5.4	33	48	0.01	7.4	143	**9**
Visitall	361.73	26.8	957K	1.0M	372.94	589	3.7M	4.0M

Table 4.1: We report for each number of threads n used, search time in seconds, peak memory usage (MB), and number of expanded and evaluated nodes. The best results for each domain are in bold.

With these results we can see in best-case scenarios the performance scales linearly with the number of threads, or even better than linearly like in the *corridor* domain. However, some domains (in particular *visitall*), present a pathological behaviour. Adding more threads to the search only contributes to exploring irrelevant parts of the search tree, wasting computer resources. This problem can be mitigated by adjusting the initial number of nodes of each thread (N in Algorithm 7). The value used for obtaining the results of Table 4.1 is $N = 10$, but using $N = 160$ for the $visitall$ domain reduces the search time for eight threads to 86s.

Chapter 5

RELATED WORK

Synthesizing plans capable of handling multiple automated planning problem instances is a longstanding open problem in Artificial Intelligence (AI), generally called Generalized Planning (GP). This goal stems from the realization that classical planning solvers' performance highly depends on the number of objects in the instance. However, most classical planning domains have algorithmic solutions that could be used to solve multiple problem instances [Srivastava et al., 2011a]. The challenge is developing strategies for an automated generation of these algorithmic solutions. In this field of study, it is interesting to highlight the work of Bonet et al. [2010], who have introduced a model-based method for automatically deriving finite-state automatons capable of solving multiple planning problem instances, and the research of Srivastava et al. [2011a], who have identified sequences of actions that can be used to measure progress when using loops.

One critical challenge in developing Generalized Planning solvers is finding good heuristics. Ideally, these heuristic functions should be capable of solving problem instances from multiple domains with comparable performance to domain-specific heuristics. Potential heuristics form a family of compact evaluation functions that have demonstrated promising results in guiding a greedy search to synthesize generalized plans of multiple domains [Francès et al., 2019]. Potential heuristics are defined as a weighted sum over state variables [Pommerening et al., 2017], and they can be used even in symbolic search problems [Fišer et al., 2022].

Regarding the development of parallelization techniques, there has been a lot of research about cost-optimal parallel search strategies. One of the most popular algorithms is HDA*, which uses a hash function to distribute states to unique processors [Kishimoto et al., 2013], thus avoiding work duplication. However, parallel Greedy Best-First Search (GBFS) strategies (where the goal is to quickly find any solution path regardless of its cost, like with `Parallel_BFGP`) are under-

developed [Shimoda and Fukunaga, 2023]. Kuroiwa and Fukunaga [2021] show that similar strategies to HDA* applied to GBFS can perform much worse than single-threaded executions because they can expand orders of magnitude more states. Meanwhile, GPU computing has gained a lot of traction over recent years. However, accelerating graph algorithms on GPUs is very challenging, and current research to accelerate BFS still struggles to compete with the best CPU implementations [Luo et al., 2010a; Wen and Zhang, 2019].

Finally, other forms of optimization, besides parallelization, can be explored to accelerate the resolution of GP problems. For example, Srivastava et al. [2011b] introduce the idea of using a directed approach to Generalized Planning, where problem instances are progressively activated each time a candidate solution fails to solve them. This idea is then further developed by Segovia-Aguas et al. [2022a], who also apply it to BFGP. Another possible source of acceleration is the use of negative examples, which are planning instances that must not be solved by the generalized planner [Segovia-Aguas et al., 2020].

Chapter 6

DISCUSSION AND FUTURE WORK

This work shows that implementing parallel algorithms to generalized planning solvers is not straightforward, and naive solutions can even lead to performance degradation. In contrast to optimal-cost searching, parallel greedy best-first strategies still struggle to scale well for a large number of processors.

Even so, generalized planning is a fascinating research topic with enormous future potential, and BFGP closes the gap to achieving automated programming for problem-solving. Performance improvements are still required for a more comprehensive set of applications to become available, especially now that current Large Language Models have demonstrated excellent capabilities at simple programming tasks. Here we discuss some areas of future work:

Learning for planning. In the Learning Track of the last International Planning Competition (IPC) Taitler et al. [2024], all competitors were at best on pair with non-learning system like the classical planner LAMA Richter et al. [2011]. In the case of BFGP-based approaches, they failed because of two reasons: (i) solution representation; and (ii) small bounds in the number of lines and pointers. We have shown how this work can make BFGP to scale-up on the same benchmarks, so it has potential to address more complex problems that require larger bounds, like the public benchmarks from the IPC.

Limitations of Parallel BFGP. This work is still limited by the proposed representation of BFGP solutions which do not consider tests of getting closer to (sub-)goals. Thus, the proposed parallel algorithm improves the performance but shares the intrinsic limitations of BFGP-based family of algorithms.

Parallel strategies beyond Generalized Planning. The contribution of this work is on applying sound and complete parallel strategies to produce solutions to generalized planning problems as C++ programs. However, the scope of the proposed strategies goes beyond generalized planning, being applicable to any

tree-based searching or planning problem.

From multi-core CPU to GPU search. One of the major successes in Artificial Intelligence has been in the field of Deep Learning and its related applications, being the one of the main reasons the exploitation of GPUs with parallel and distributed algorithms. Bringing all that computational capacity to (generalized) planning (e.g. Breadth-First Search implementations to exploit GPUs Luo et al. [2010b]), could be worth to the planning systems.

Bibliography

Blake, G., Dreslinski, R. G., Mudge, T., and Flautner, K. (2010). Evolution of thread-level parallelism in desktop applications. *SIGARCH Comput. Archit. News*, 38(3):302–313.

Bonet, B. and Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, 129(1):5–33.

Bonet, B., Palacios, H., and Geffner, H. (2010). Automatic derivation of finite-state machines for behavior control. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, pages 1656–1659.

Buluç, A. and Madduri, K. (2011). Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12.

Fišer, D., Torralba, A., and Hoffmann, J. (2022). Operator-Potential heuristics for symbolic search. *Proceedings of the ... AAAI Conference on Artificial Intelligence*, 36(9):9750–9757.

Francès, G., Corrêa, A. B., Geissmann, C., and Pommerening, F. (2019). Generalized potential heuristics for classical planning. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 5554–5561. International Joint Conferences on Artificial Intelligence Organization.

Ghallab, M., Nau, D., and Traverso, P. (2016). *Automated Planning and Acting*. Cambridge University Press.

Jiménez, S., Segovia-Aguas, J., and Jonsson, A. (2019). A review of generalized planning. *The Knowledge Engineering Review*, 34:e5.

Kishimoto, A., Fukunaga, A., and Botea, A. (2013). Evaluation of a simple, scalable, parallel best-first search strategy. *Artificial intelligence*, 195:222–248.

Korf, R. E., Zhang, W., Thayer, I., and Hohwald, H. (2005). Frontier search. *Journal of the ACM (JACM)*, 52(5):715–748.

Kuroiwa, R. and Fukunaga, A. (2021). On the Pathological Search Behavior of Distributed Greedy Best-First Search. *Proceedings of the ... International Conference on Automated Planning and Scheduling/Proceedings of the ... International Conference on Automated Planning and Scheduling*, 29:255–263.

Lamport, N. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *I.E.E.E. transactions on computers/IEEE transactions on computers*, C-28(9):690–691.

Luo, L., Wong, M., and Hwu, W.-m. (2010a). An effective gpu implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, page 52–55, New York, NY, USA. Association for Computing Machinery.

Luo, L., Wong, M., and Hwu, W.-m. (2010b). An effective gpu implementation of breadth-first search. In *Proceedings of the 47th design automation conference*, pages 52–55.

Pommerening, F., Helmert, M., and Bonet, B. (2017). Higher-Dimensional potential heuristics for optimal classical planning. *arXiv (Cornell University)*, pages 3636–3643.

Richter, S., Westphal, M., and Helmert, M. (2011). Lama 2008 and 2011. In *International Planning Competition*, pages 117–124. ICAPS Freiburg, Germany.

Russell, S. and Norvig, P. (2021). *Artificial Intelligence: A Modern Approach, Global Edition*. Pearson Higher Ed.

Segovia-Aguas, J., Celorrio, S. J., Sebastiá, L., and Jonsson, A. (2022a). Scaling-Up Generalized Planning as Heuristic Search with Landmarks. *Proceedings, the ... International Symposium on Combinatorial Search*, 15(1):171–179.

Segovia-Aguas, J., E-Martín, Y., and Jiménez, S. (2022b). Representation and synthesis of c++ programs for generalized planning.

Segovia-Aguas, J., Jiménez, S., and Jonsson, A. (2020). Generalized Planning with Positive and Negative Examples. *Proceedings of the ... AAAI Conference on Artificial Intelligence*, 34(06):9949–9956.

Segovia-Aguas, J., Jiménez, S., and Jonsson, A. (2021). Generalized planning as heuristic search. *Proceedings of the International Conference on Automated Planning and Scheduling*, 31(1):569–577.

Segovia-Aguas, J., Jiménez, S., and Jonsson, A. (2024). Generalized Planning as Heuristic Search: A new planning search-space that leverages pointers over objects. *Artificial intelligence*, page 104097.

Shimoda, T. and Fukunaga, A. (2023). Improved exploration of the bench transition system in parallel Greedy best first search. *Proceedings, the ... International Symposium on Combinatorial Search*, 16(1):74–82.

Sirhan, N. N. (2020). Multi-Core Processors: Concepts and Implementations. *Social Science Research Network*.

Srivastava, S. (2011). Foundations and applications of generalized planning. *AI Communications*, 24(4):349–351.

Srivastava, S., Immerman, N., and Zilberstein, S. (2011a). A new representation and associated algorithms for generalized planning. *Artificial Intelligence*, 175(2):615–647.

Srivastava, S., Immerman, N., Zilberstein, S., and Zhang, T. (2011b). Directed search for generalized plans using classical planners. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 21, pages 226–233.

Taitler, A., Alford, R., Espasa, J., Behnke, G., Fišer, D., Gimelfarb, M., Pommerening, F., Sanner, S., Scala, E., Schreiber, D., et al. (2024). The 2023 international planning competition. *AI Magazine*, pages 453–460.

Ungerer, T., Robič, B., and Šilc, J. (2002). Multithreaded Processors. *The Computer Journal*, 45(3):320–348.

Wen, H. and Zhang, W. (2019). Improving parallelism of breadth first search (bfs) algorithm for accelerated performance on gpus. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7.

Appendix A

PLANNING AS HEURISTIC SEARCH

A wide array of algorithms can be used to solve classical planning problem instances. Every two years, the International Conference on Automated Planning and Scheduling Systems (ICAPS) organizes the International Planning Competition (IPC), where a set of challenges is proposed to encourage research on advanced planning solvers. However, heuristic search has traditionally been among the most successful approaches [Segovia-Aguas et al., 2021].

Planning as heuristic search strategies tackle the goal of finding sequential plans as a combinatorial search in the space of the states reachable from the initial state. This combinatorial search can be addressed in multiple ways, but a forward search led by heuristics extracted from the problem representation is one of the most common implementations [Segovia-Aguas et al., 2024]. In forward search, the algorithm begins at the initial state and explores actions applicable in the current state that aim to reach the goal state. Therefore, viewing this kind of state space explorations as graphs is prevalent.

A.1 Heuristic graph search

A *graph* is defined as a tuple $G = \langle V, E \rangle$, where V is a nonempty set, whose elements are called vertices or nodes, and E is a set of pairs of vertices $\{u, v\}$ such that $u, v \in V$, called edges. A *loop* is an edge that connects a vertex to itself. If the graph is *undirected*, edges are formed by unordered pairs of vertices $\{u, v\}$ with $u, v \in V$, but if the graph is *directed*, the pair of vertices is ordered, meaning that if $(u, v) \in E$, then there is a directed edge from u to v. Two vertices in a graph are said to be adjacent if an edge joins them. A walk in a graph is a nonempty sequence of vertices $v_0, ..., v_k$ such that v_i and v_{i+1} are adjacent for all

$0 \leq i < k$, and the length of the walk is k. A *path* is a walk where all the vertices are different. A *cycle* is a walk $v_0, ..., v_k$ with $k \geq 2$ such that $v_0, ..., v_{k-1}$ is a path and $v_k = v_0$.

We say that two vertices in a graph are *connected* if a path that begins at one and ends at the other exists. A directed graph is said to be strongly connected if, for every pair of vertices u and v, there is a path from u to v. A directed graph is said to be weakly connected if the (undirected) graph obtained from eliminating the directions of the edges is connected. A directed graph is called a directed acyclic graph if it does not contain any cycles, also called loops.

Definition 2 (Tree) *A* tree *is a connected acyclic graph.*

When modeling planning problems as graphs (or trees depending on the type of problem), nodes represent states, edges represent actions that transition between states, and finding a sequential plan that reaches a goal state from the initial state is equivalent to finding a path in the graph that connects the initial state with a goal state. This type of modeling allows well-established graph search algorithms to be used. For example, Best-First Search (BFS) has been proven to perform very well as a heuristic search planner [Bonet and Geffner, 2001].

Best-First Search prioritizes nodes to explore based on a heuristic evaluation function, which estimates the cost to reach the goal from the current node. If this function never overestimates, then we say that it is admissible and can be used to compute optimal solutions. Admissible heuristics are commonly derived from relaxed plans, which are simplifications of the problems and are usually much cheaper to solve. The heuristic function used in Best-First Search is crucial for its performance. A well-designed heuristic can significantly reduce the number of nodes explored, leading to faster solutions.