Parallel Lifted Planning via Semi-Naive Datalog Evaluation

Dominik Drexler, Oliver Joergensen, Jendrik Seipp

Linköping University, Sweden
dominik.drexler@liu.se, oliver.harold.joergensen@liu.se, jendrik.seipp@liu.se

Abstract

Lifted classical planners operate directly on first-order planning tasks to avoid the computationally demanding grounding step. However, lifted planning is typically slower, as planners must repeatedly instantiate ground structures during search. Many core components of lifted classical planning, such as successor generation, axiom evaluation, task grounding, and delete-relaxed heuristics, have previously been studied through the lens of Datalog evaluation. We build upon this line of work and extend it by developing and analyzing an execution model with two levels of parallelism: rule-level parallelism and grounding parallelism. We further specialize this solver for planning-specific workloads with a grounder based on clique enumeration, which we extend to support semi-naive Datalog evaluation. Our experimental evaluation using lazy greedy best-first search with the FF heuristic shows that our implementation already solves more tasks than the baselines on a single core, and the gap widens as additional cores are used. Moreover, on hard-to-ground tasks where 97.6% of the total runtime is spent in Datalog execution on average, our execution model exhibits an average parallel fraction of 92.4% and achieves up to a 6-fold speedup on 8 cores.

1 Introduction

Classical planning systems commonly solve problems using a grounded representation of the planning task. While grounding enables efficient search and heuristic computation, it may become infeasible when the number of ground atoms and actions exceeds available memory. Lifted planners avoid this issue by operating directly on the first-order representation of the planning task. However, lifted reasoning introduces additional overhead, as planners repeatedly generate and evaluate ground structures during search.

A natural way to exploit multiple cores in heuristic search planning is to parallelize the search itself, for example by expanding or evaluating multiple states concurrently (Kishimoto, Fukunaga, and Botea 2009; Vidal, Bordeaux, and Hamadi 2010; Vernhes, Infantes, and Vidal 2013). This is effective when individual successor-generation and heuristic-evaluation calls are relatively cheap. Lifted planning creates a different bottleneck: each state evaluation may require substantial first-order reasoning, including repeated rule evaluation for successor generation and delete-relaxed heuristics. In such settings, a search-level parallel planner would still invoke these lifted computations as expensive serial subroutines. Our goal is therefore to expose parallelism inside these subroutines. This does not replace parallel search; it addresses a lower-level bottleneck that parallel search algorithms can use as a component.

Several components of lifted classical planning correspond to rule evaluation problems. In particular, successor generation (Corrêa et al. 2020), axiom evaluation, task grounding (Helmert 2009), and delete-relaxed planning graph heuristics (Corrêa et al. 2021, 2022) can be expressed as Datalog rule evaluations over a database of facts. Previous work exploits these connections and shows that semi-naive Datalog evaluation (Bancilhon and Ramakrishnan 1986; Ceri, Gottlob, and Tanca 1989) can be used within lifted planners to address difficult-to-ground tasks. However, these approaches evaluate rules sequentially and do not exploit inherent parallelism.

In this paper, we propose a *parallel lifted planner* based on the connection between lifted planning and Datalog. Numerous parallel Datalog architectures have been proposed, with scalability varying considerably across techniques and benchmarks (e.g., Jordan, Scholz, and Subotić 2016; Shkapsky et al. 2016), making it challenging to identify designs that perform well for planning workloads, where general-purpose engines are often not directly applicable. Parallel Datalog systems can broadly be categorized into synchronous and asynchronous architectures: in synchronous evaluation, rule applications proceed in globally coordinated rounds with barriers, whereas asynchronous evaluation allows independent updates to shared relations. While synchronous architectures are less expressive, they are not necessarily less efficient in practice and can even outperform asynchronous ones due to more controlled communication patterns (Das and Zaniolo 2019). We therefore adopt a synchronous semi-naive execution model with two complementary levels of parallelism: rule-level parallelism evaluates rules of a stratum concurrently, while grounding parallelism distributes the enumeration of rule instances across workers.

Our Datalog solver is specialized for classical planning using a grounder based on k-clique enumeration in k-partite substitution consistency graphs (Ståhlberg 2023). Database-based rule evaluation techniques provide exact semantics and can be very efficient for planning rules with favorable join structure (Corrêa et al. 2021). We use clique enumeration as a complementary execution mechanism for cyclic binary

rule structures: it keeps pairwise consistency constraints explicit, avoids materializing high-arity intermediate joins, and does not require rule-splitting transformations that introduce auxiliary predicates and additional synchronization points in a parallel semi-naive implementation. Soundness and completeness hold for rules whose body literals have arity at most two; the remaining cases are handled by an explicit applicability test. We extend this approach to semi-naive evaluation by seeding clique enumeration at newly consistent edges (Wang, Yu, and Long 2024). We evaluate on classical planning benchmarks using lazy greedy best-first search with the FF heuristic (Hoffmann and Nebel 2001), achieving speedups of 2–4x and up to 6x on 8 cores.

The main contributions of this work are:

- We propose Δ-KPKC, a semi-naive Datalog solver for lifted classical planning based on k-clique enumeration in k-partite graphs.
- We develop a synchronous parallel execution architecture for Δ-KPKC with two levels of parallelism: rule-level parallelism and grounding parallelism.
- We provide a detailed empirical analysis of the effectiveness and scalability of Δ-KPKC on classical planning benchmarks, including average empirical speedup limits.

2 Related Work

We build on prior work on the lifted FF heuristic and on parallel Datalog evaluation, and contrast our approach with parallel heuristic search.

2.1 The Lifted FF Heuristic

The FF heuristic (Hoffmann and Nebel 2001) is one of the most widely used heuristics in classical planning. It explores the delete-relaxed planning graph over grounded actions, followed by backchaining from the goal, to retrieve informative heuristic estimates and preferred actions. Corrêa et al. (2022) investigated lifted variants of the FF heuristic to avoid full grounding of the planning task. Our work builds on this connection between lifted planning and Datalog evaluation. While previous approaches use semi-naive evaluation to derive lifted planning algorithms, they evaluate rules sequentially. In contrast, we exploit the inherent parallelism of rule evaluation by developing a parallel semi-naive execution architecture for heuristic search with the FF heuristic.

2.2 Parallel Datalog Evaluation

Efficient evaluation of Datalog programs has been extensively studied in the database and knowledge representation communities. Semi-naive evaluation (Bancilhon and Ramakrishnan 1986) is the standard technique for avoiding redundant rule instantiations during bottom-up evaluation. A number of systems have explored parallel and distributed Datalog execution, including both shared-memory and distributed systems (e.g., Jordan, Scholz, and Subotić 2016; Shkapsky et al. 2016). However, high-performance Datalog evaluation often relies on workload-specific optimizations and evaluation strategies. As a result, general-purpose Datalog engines are rarely directly applicable to classical planning workloads. Instead, planning systems typically implement specialized rule evaluation mechanisms tailored to the structure of planning domains. Our work follows this approach by developing a specialized semi-naive Datalog solver for lifted classical planning that exploits parallelism in both rule evaluation and grounding.

2.3 Parallel Heuristic Search

Parallel heuristic search usually exploits concurrency at the search level: workers expand, generate, or evaluate different states in parallel. Hash-distributed best-first search is a prominent example. HDA* and related algorithms distribute states across workers according to a hash function, enabling parallel exploration while reducing duplicate generation (Kishimoto, Fukunaga, and Botea 2009, 2013). Subsequent work from the same line studied how hash functions affect load balance, locality, and communication overhead in parallel best-first search (Jinnai and Fukunaga 2017).

Parallelization has also been studied for satisficing and greedy search. Adaptive K-parallel best-first search and landmark-based meta best-first search parallelize domain-independent planning at the search level (Vidal, Bordeaux, and Hamadi 2010; Vernhes, Infantes, and Vidal 2013). More recent work on parallel greedy best-first search investigates how to separate generation from heuristic evaluation and how to bound deviations from sequential search behavior (Shimoda and Fukunaga 2025a,b).

These methods are complementary to ours. They assume an implementation of successor generation and heuristic evaluation for each state, whereas our work parallelizes precisely these lifted computations. This distinction matters on hard-to-ground tasks, where a single lifted successor-generation or FF-evaluation call may require substantial Datalog evaluation. A parallel search algorithm could therefore still use our parallel rule evaluator as its per-state reasoning component.

3 Background

We review classical planning, the substitution consistency graph underlying our grounder and Datalog as the rule formalism our parallel solver evaluates.

3.1 Classical Planning

A *classical planning task* is a tuple $\langle \mathcal{P}, \mathcal{A}, \mathcal{O}, s_0, \mathcal{G} \rangle$. The *planning domain* comprises $\langle \mathcal{P}, \mathcal{A} \rangle$, and the *task information* comprises $\langle \mathcal{O}, s_0, \mathcal{G} \rangle$.

Predicates and Literals. $\mathcal{P}$ is a set of predicate symbols. Each *predicate symbol* P in $\mathcal{P}$ has an associated arity $ar(P)$ in $\mathbb{N}$. An *atom* over a predicate P of arity $ar(P) = k$ is an expression $P(x_1, \ldots, x_k)$ where x_i with $i = 1, \ldots, k$ are variables or constants. A *literal* is either an atom p or its negation $\neg p$.

Objects and Substitutions. $\mathcal{O}$ is a set of objects (constants). A *substitution function* ρ over some set of variables $\mathcal{X}(\rho)$ and objects $\mathcal{O}$ is a function $\rho : \mathcal{X}(\rho) \to \mathcal{O}$ mapping each variable in $\mathcal{X}(\rho)$ to an object in $\mathcal{O}$. We write $\mathcal{R}(\mathcal{X}, \mathcal{O})$ for the set of all such ρ. We write $x_1/o_1, \ldots, x_n/o_n$ for a

substitution function ρ over variables $\{x_1, \ldots, x_n\}$ and objects $\mathcal{O}$ such that $\rho(x_i) = o_i$ for all $i = 1, \ldots, n$. We write $\rho \subseteq \rho'$ iff $\mathcal{X}(\rho) \subseteq \mathcal{X}(\rho')$ and $\rho(x) = \rho'(x)$ for all $x \in \mathcal{X}(\rho)$. The application of ρ to a syntactic element y (e.g., an atom, literal, or action), denoted by $y[\rho]$, replaces every occurrence of each variable x_i with the corresponding object $o_i = \rho(x_i)$. The set of free variables occurring in an element y is denoted $\mathcal{X}(y)$, and the arity $ar(y)$ is defined as $|\mathcal{X}(y)|$. We say that y is *ground*, denoted by $\overline{y}$, iff $\mathcal{X}(y) = \emptyset$; a substitution covering all variables of y thus yields a *ground instantiation* $\overline{y}$.

States. s_0 is the *initial state*. A *state* s consists of a set of ground atoms $\overline{P}(s)$ that hold in s. Ground atoms not in $\overline{P}(s)$ are false. A positive ground literal $\overline{p}$ *holds* in s iff $\overline{p} \in \overline{P}(s)$, and a negative ground literal $\neg\overline{p}$ *holds* iff $\overline{p} \notin \overline{P}(s)$.

Goals. $\mathcal{G}$ is the goal, which is a set of ground literals.

Actions and Successors. $\mathcal{A}$ is a set of action schemas. Each *action schema* a in $\mathcal{A}$ consists of two sets $pre(a)$ and $\mathit{eff}(a)$, where $pre(a)$ is a set of precondition literals and $\mathit{eff}(a)$ is a set of effect literals. We write $pre^+(a)$, $pre^-(a)$, $\mathit{eff}^+(a)$, and $\mathit{eff}^-(a)$ for the sets of atoms of positive and negative literals in $pre(a)$ and $\mathit{eff}(a)$. A ground action $\overline{a}$ is *applicable* in a state s iff every ground literal $\overline{\ell}$ in $pre(\overline{a})$ holds in s. Applying a ground action $\overline{a}$ in a state s where it is applicable yields the *successor state* s' with $\overline{P}(s') = (\overline{P}(s) \setminus \mathit{eff}^-(\overline{a})) \cup \mathit{eff}^+(\overline{a})$.

The objective for a given classical planning task is to find a *plan*, which is a sequence of ground actions that, when successively applied starting from the initial state, yields a state in which all ground literals mentioned in the goal hold.

3.2 Substitution Consistency Graph

Ståhlberg (2023) proposed clique enumeration as an alternative to database techniques (Corrêa et al. 2021) for computing applicable ground actions in lifted classical planning. Database-based methods evaluate joins over the rule body and are sound and complete for the full rule, with strong performance when the join structure is favorable, for example for acyclic rules or rules with similarly simple join structure. For cyclic rule bodies, however, conventional join plans may materialize large high-arity intermediate relations before all pairwise constraints have been applied, a standard limitation of join-based evaluation (Ngo et al. 2018). Such cases can be mitigated by more sophisticated join algorithms or by rule-splitting transformations, but these techniques add auxiliary relations and, in a parallel semi-naive implementation, additional materialization and synchronization points. Clique enumeration makes a different tradeoff. Given a state, it computes a polynomial-size factored overapproximation of the variable substitutions satisfying the action preconditions, keeps the original pairwise consistency structure explicit, enumerates complete compatible assignments directly, and filters the resulting candidates by a final applicability check. Its running time is output-sensitive in the size of the explicitly represented overapproximation. On the vast majority of classical planning benchmarks, clique enumeration is sound and complete, i.e., the enumerated candidates coincide with the applicable ground actions; outside this fragment, the final applicability check restores correctness at the cost of enumerating additional candidates. Before stating the necessary conditions, we define the core data structure of this approach, the substitution consistency graph.

The *substitution consistency graph* $\mathcal{G}_{a,s}$ for an action a in a state s is a tuple $\langle \mathcal{V}(\mathcal{G}_{a,s}), \mathcal{E}(\mathcal{G}_{a,s}) \rangle$ where $\mathcal{V}(\mathcal{G}_{a,s}) = \{x/o \mid x \in \mathcal{X}(a), o \in \mathcal{O}\}$ is the set of vertices, each representing a possible variable substitution, and $\mathcal{E}(\mathcal{G}_{a,s}) = \mathcal{V}(\mathcal{G}_{a,s}) \times \mathcal{V}(\mathcal{G}_{a,s}) \setminus (\mathcal{I}^{\neq} \cup \mathcal{I}^{+} \cup \mathcal{I}^{-})$ is the set of edges, each representing all possible pairs of variable substitutions (Ståhlberg 2023). An atom $P(x_1, \ldots, x_n)$ *matches* an atom $P'(x'_1, \ldots, x'_n)$ iff $P = P'$ and for all $i = 1, \ldots, n$, x_i or x'_i are variables, or else $x_i = x'_i$, i.e., both are objects. The sets $\mathcal{I}^{\neq}, \mathcal{I}^{+}, \mathcal{I}^{-}$ are:

- $\mathcal{I}^{\neq} = \{\{x/o_1, x/o_2\} \mid x \in \mathcal{X}(a), o_1, o_2 \in \mathcal{O}, o_1 \neq o_2\}$, i.e., the set of edges whose corresponding substitution function would assign different objects to the same variable,
- $\mathcal{I}^{+} = \{\{v, v'\} \mid v, v' \in \mathcal{V}(\mathcal{G}_{a,s}), \exists p \in pre^+(a), \forall p' \in s \;.\; p[v, v'] \text{ does not match } p'\}$, i.e., the set of edges for which some positive precondition atom, after applying the corresponding substitution, does not match any ground atom in the state.
- $\mathcal{I}^{-} = \{\{v, v'\} \mid v, v' \in \mathcal{V}(\mathcal{G}_{a,s}), \exists p \in pre^-(a) \;.\; p[v, v'] \in s\}$, i.e., the set of edges for which some negative precondition atom, after applying the corresponding substitution, yields a ground atom that is contained in the state.

The following theorem, from Ståhlberg (2023), establishes the conditions for precisely identifying all applicable ground actions via k-clique enumeration.

Theorem 1. *Consider an action a with arity $k \geq 2$ where each literal $\ell \in pre(a)$ has arity at most 2, and a state s. Then, there exists a k-clique $\mathcal{C} = \{v_1, \ldots, v_k\}$ in $\mathcal{G}_{a,s}$ iff all ground literals $\overline{\ell} = \ell[v_1, \ldots, v_k]$ with $\ell \in pre(a)$ hold in s.*

Theorem 1 enables using k-clique algorithms to enumerate candidate applicable ground actions. If all literals in $pre(a)$ have arity at most 2, every enumerated k-clique corresponds to a ground instance $\overline{a}$ of a that is applicable in s. For actions containing literals of arity greater than 2, we explicitly check whether those ground literals hold in s. For actions of arity one, one can define a vertex removal criterion analogous to the edge removal criteria above. Moreover, actions of arity zero are already ground, and we can simply check their applicability, for instance using efficient lookup data structures for ground actions (Helmert 2006).

3.3 Datalog

A *positive Datalog program* (program) $\mathcal{D}$ is a pair $\langle \mathcal{F}, \mathcal{R} \rangle$ where $\mathcal{F}$ is a set of facts (ground atoms) and $\mathcal{R}$ is a set of rules. Each rule r in $\mathcal{R}$ has the form $h \leftarrow b_1, \ldots, b_k$ where $k \geq 0$, h is an atom (the "head"), and $b_1, \ldots, b_k$ are atoms (the "body"). We write $\mathrm{H}(r)$ for the head and $\mathrm{B}(r)$ for the body of a rule r.

We use the standard *semipositive Datalog* extension to represent negative preconditions. In semipositive programs, rule bodies may contain negative literals, but predicates that

occur negatively are extensional: they are fixed by the input facts and do not occur in any rule head. Thus, a semipositive rule still has an atom as its head, while its body is a set of literals. Analogous to classical planning, a ground rule $\overline{r}$ is *applicable* in a fact set J if every literal $\overline{\ell}$ in $\mathrm{B}(\overline{r})$ holds in J.

More general Datalog programs with negation are commonly interpreted via stratification. A program $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ with predicates $\mathcal{P}$ induces a *predicate dependency graph* with a vertex for each predicate, and a directed edge $\langle P, P' \rangle$ if P occurs in the body and P' in the head of a rule $r \in \mathcal{R}$, labeled $\top$ (resp. $\bot$) if P occurs positively (resp. negatively) in r.

A program is *stratifiable* if its predicates can be partitioned into strata $\mathcal{P}_1, \ldots, \mathcal{P}_\ell$ such that for every edge $\langle P, P' \rangle$ in the dependency graph labeled $\top$ (resp. $\bot$), we have $P \in \mathcal{P}_i$ and $P' \in \mathcal{P}_j$ with $i \leq j$ (resp. $i < j$). This stratification induces rule strata $\mathcal{R}_1, \ldots, \mathcal{R}_\ell$, where $r \in \mathcal{R}_i$ if the predicate in $\mathrm{H}(r)$ belongs to $\mathcal{P}_i$. Semipositive programs are a restricted special case of stratifiable programs, with all negated predicates fixed before rule evaluation.

The *model* $\mathcal{M}(\mathcal{D})$ of a stratifiable program $\mathcal{D}$ is defined inductively over the rule strata $\mathcal{R}_1, \ldots, \mathcal{R}_\ell$. Let $\mathcal{F}_0 := \mathcal{F}$. For each $i = 1, \ldots, \ell$, we define the consequence operator

$$\mathcal{T}_i(J) = \mathcal{F}_{i-1} \cup \{ \mathrm{H}(\overline{r}) \mid \overline{r} \in \mathrm{gr}(\mathcal{R}_i, J) \},$$

where $\mathrm{gr}(\mathcal{R}_i, J)$ is the set of all ground instances of rules in $\mathcal{R}_i$ whose body is applicable in J, i.e., $J \vDash \mathrm{B}(\overline{r})$. Then $\mathcal{F}_i$ is the least fixed point of $\mathcal{T}_i$, and $\mathcal{M}(\mathcal{D}) := \mathcal{F}_\ell$.

The least fixed point of $\mathcal{T}_i$ can be computed more efficiently using *semi-naive bottom-up* evaluation (Bancilhon and Ramakrishnan 1986). For each stratum i, let $J_i^0 := \mathcal{F}_{i-1}$ and $\Delta_i^0 := \mathcal{F}_{i-1}$. For $t \geq 0$, define

$$\Delta_i^{t+1} := \{\mathrm{H}(\overline{r}) \mid \overline{r} \in \mathrm{gr}(\mathcal{R}_i, J_i^t), \mathrm{B}(\overline{r}) \cap \Delta_i^t \neq \emptyset\} \setminus J_i^t,$$

and update $J_i^{t+1} := J_i^t \cup \Delta_i^{t+1}$. Thus, in each iteration, only those rule instances are considered whose body contains at least one atom that was derived in the previous iteration. The evaluation terminates when $\Delta_i^t = \emptyset$ for some t, and we set $\mathcal{F}_i := J_i^t$.

4 Datalog Compilations

Many core components of lifted classical planning can be expressed as Datalog programs, including successor generation (Corrêa et al. 2020), axiom evaluation, task grounding (Helmert 2009), and delete-relaxed planning graph (RPG) heuristics (Corrêa et al. 2021, 2022). In this paper, we present semipositive Datalog compilations for applicable action generation and relaxed planning graph construction. Action generation underlies all state-space search methods, while RPG heuristics often provide strong search guidance. In the following, let $\langle \mathcal{P}, \mathcal{A}, \mathcal{O}, s_0, \mathcal{G} \rangle$ be a planning task and s a state.

4.1 Applicable Action Program

The *applicable action program* is $\mathcal{D}_\mathcal{A} = \langle \mathcal{F}, \mathcal{R}_\mathcal{A} \rangle$, where $\mathcal{F}$ is the state s, and for each action schema $a \in \mathcal{A}$ with variables $\mathcal{X}(a)$, positive precondition atoms $pre^+(a) = \{p_1, \ldots, p_m\}$, and negative precondition atoms $pre^-(a) = \{p_{m+1}, \ldots, p_n\}$, there is a rule $r_a \in \mathcal{R}_\mathcal{A}$ of the form

$$a\text{-applicable}(\mathcal{X}(a)) \leftarrow p_1, \ldots, p_m, \neg p_{m+1}, \ldots, \neg p_n.$$

This program is semipositive: the only derived predicates are the a-applicable predicates, and these predicates do not occur in any rule body.

Each ground instance of a-applicable$(\mathcal{X}(a))$ in the model $\mathcal{M}(\mathcal{D}_\mathcal{A})$ with substitution function ρ induces a ground action $\overline{a} = a[\rho]$ that is applicable in s. Taking the union of these ground actions over all $a \in \mathcal{A}$ yields exactly the set of ground actions applicable in s.

4.2 Delete-Relaxed Planning Graph Program

The *RPG program* is $\mathcal{D}_{\text{RPG}} = \langle \mathcal{F}, \mathcal{R}_{\text{RPG}} \rangle$, where $\mathcal{F}$ is the state s, and for each action schema $a \in \mathcal{A}$ with variables $\mathcal{X}(a)$, positive precondition atoms $pre^+(a) = \{p_1, \ldots, p_m\}$, and static negative precondition atoms $pre_s^-(a) = \{p_{m+1}, \ldots, p_n\}$, $\mathcal{R}_{\text{RPG}}$ contains, for each positive effect atom $p_e \in \mathit{eff}^+(a)$, the rule

$$p_e \leftarrow p_1, \ldots, p_m, \neg p_{m+1}, \ldots, \neg p_n.$$

This program is semipositive because negative literals are restricted to static predicates, so no negated predicate can be derived by an RPG rule.

The restriction to static negative preconditions, i.e., atoms over predicates whose truth value never changes, is necessary to preserve monotonicity of the Datalog model. In contrast, this restriction is not needed for the applicable action program, since no rule head occurs in any rule body and semi-naive evaluation therefore terminates after one iteration. The cost of a ground atom in the head is defined by an aggregation function over the costs of the positive ground atoms in the body, e.g., summation for the additive heuristic or maximum for the max heuristic. For the FF heuristic, a relaxed plan is extracted by backward chaining from the goal atoms over supporting ground rule instances. The heuristic value is the number of distinct ground actions inducing these rule instances, and the preferred actions are those ground actions that are applicable in s.

5 Semi-naive Program Evaluation (Δ-KPKC)

We now extend clique-based grounding from action schemas to semi-naive Datalog evaluation, where the main challenge is to enumerate exactly those rule instances whose bodies contain at least one newly derived fact.

The approach of Ståhlberg (2023) for generating ground applicable actions naturally extends to Datalog rules since the rule bodies are syntactically equivalent to action preconditions. However, semi-naive bottom-up evaluation requires enumerating each relevant k-clique exactly once within a monotonically growing fact set for each rule stratum. More precisely, for stratum $\mathcal{R}_i$ and iteration t, we must identify all ground rules $\overline{r} \in \mathrm{gr}(\mathcal{R}_i, J_i^t)$ with $\mathrm{B}(\overline{r}) \cap \Delta_i^t \neq \emptyset$, as required by the definition of Δ_i^{t+1}.

Definition 1 (Δ-edges)**.** Consider a stratum $\mathcal{R}_i$, two consecutive semi-naive iterations $t-1$ and t with fact sets J_i^{t-1} and J_i^t, and a rule $r \in \mathcal{R}_i$. Let $\mathcal{G}(r, J_i^{t-1})$ and $\mathcal{G}(r, J_i^t)$ denote the substitution consistency graphs for rule r under J_i^{t-1} and J_i^t, respectively. The set of Δ-edges of r in J_i^t, is defined as

$$\Delta\mathcal{E}_{i,r}^t = \mathcal{E}(\mathcal{G}(r, J_i^t)) \setminus \mathcal{E}(\mathcal{G}(r, J_i^{t-1})).$$

Δ-edges arise from facts in Δ_i^t that were added in the previous iteration. Hence, we can use Δ-edges to seed the k-clique enumeration. Since a clique may contain multiple Δ-edges, we must ensure that such duplicates are filtered out.

We adopt a branch-and-bound algorithm that combines an edge-oriented strategy (Wang, Yu, and Long 2024) with techniques for enumerating k-cliques in k-partite graphs (Phillips et al. 2019), building on the classical Bron-Kerbosch algorithm for clique enumeration (Bron and Kerbosch 1973). The main idea of edge-oriented strategies is to seed the clique-enumeration algorithm at an edge, thereby partially initializing the candidate clique with its incident vertices. A clique is reported only if its unique owner, defined as a canonical edge among the edges induced by the clique, coincides with the seeding edge (Wang, Yu, and Long 2024). In our setting, we are interested in generating each relevant k-clique exactly once during semi-naive evaluation. Hence, we use the Δ-edges in $\Delta\mathcal{E}_{i,r}^t$ as seeds, since new edges may appear between two vertices that were already consistent. We define clique ownership for Δ-edges as follows.

Definition 2 (Clique Owner)**.** Let $rank$ be a fixed total order on edges. Consider a k-clique $\mathcal{C} = \{v_1, \ldots, v_k\}$ with $k \geq 2$ that contains at least one edge from $\Delta\mathcal{E}_{i,r}^t$. The owner $\omega(\mathcal{C})$ is the edge $\langle v, v' \rangle \in \Delta\mathcal{E}_{i,r}^t$ with $v, v' \in \mathcal{C}$ and minimum rank $rank(\langle v, v' \rangle)$.

Alternative strategies avoid duplicate clique generation by imposing a canonical ordering on vertices (Wang, Yu, and Long 2024) or, in the k-partite case, on partitions. In contrast, our approach retains the flexibility to select pivot partitions dynamically, allowing the search space to be pruned during branch-and-bound by prioritizing partitions with the smallest candidate sets. The following lemma establishes the key property of Δ-edges, which guarantees that every clique corresponding to a ground rule with at least one newly supported body literal contains at least one Δ-edge.

Lemma 1 (Δ-edge witness)**.** *Let $r \in \mathcal{R}_i$ and let $\bar{r} \in gr(\{r\}, J_i^t)$ be a ground rule such that $\mathrm{B}(\bar{r}) \cap \Delta_i^t \neq \emptyset$. Let $\mathcal{C}(\bar{r})$ be the corresponding k-clique in $\mathcal{G}(r, J_i^t)$. Then $\mathcal{C}(\bar{r})$ contains at least one edge from $\Delta\mathcal{E}_{i,r}^t$.*

Proof. Since $\mathrm{B}(\bar{r}) \cap \Delta_i^t \neq \emptyset$, some body literal of $\bar{r}$ corresponds to a fact in Δ_i^t, and is therefore present in J_i^t but not in J_i^{t-1}. The clique $\mathcal{C}(\bar{r})$ encodes the ground substitution induced by the body of $\bar{r}$ in the substitution consistency graph. Hence, the consistency edge corresponding to that newly available body fact is present in $\mathcal{G}(r, J_i^t)$ but absent from $\mathcal{G}(r, J_i^{t-1})$. Consequently, $\mathcal{C}(\bar{r})$ contains an edge in

$$\mathcal{E}(\mathcal{G}(r, J_i^t)) \setminus \mathcal{E}(\mathcal{G}(r, J_i^{t-1})) = \Delta\mathcal{E}_{i,r}^t,$$

i.e., at least one Δ-edge. □

Theorem 2 (Correctness of Δ-edge anchored clique enumeration)**.** *Let $\mathcal{R}_i$ be a rule stratum and let $t \geq 0$. For each rule $r \in \mathcal{R}_i$ with arity at least 2, where each literal in $\mathrm{B}(r)$ has arity at most 2, enumerate all k-cliques in $\mathcal{G}(r, J_i^t)$ that are seeded at some edge in $\Delta\mathcal{E}_{i,r}^t$, and report a clique only if its owner coincides with the seeding edge. Let H_i^t be the set of heads of the corresponding ground rules. Then*

$$H_i^t = \{\mathrm{H}(\bar{r}) \mid \bar{r} \in gr(\mathcal{R}_i, J_i^t), \mathrm{B}(\bar{r}) \cap \Delta_i^t \neq \emptyset\}.$$

Consequently, $\Delta_i^{t+1} = H_i^t \setminus J_i^t$.

Proof. We show soundness, completeness, and uniqueness.

Soundness. Consider a k-clique $\mathcal{C}$ enumerated by the algorithm. By construction, $\mathcal{C}$ is a clique in $\mathcal{G}(r, J_i^t)$, hence it corresponds to a consistent substitution satisfying all literals of $\mathrm{B}(r)$ in J_i^t. Therefore, the corresponding ground rule $\bar{r}$ belongs to $\mathrm{gr}(\{r\}, J_i^t)$. Moreover, the clique enumeration is seeded at an edge in $\Delta\mathcal{E}_{i,r}^t$, so $\mathcal{C}$ contains at least one Δ-edge. This implies that at least one body literal of $\bar{r}$ is supported by a fact in Δ_i^t, i.e.,

$$\mathrm{B}(\bar{r}) \cap \Delta_i^t \neq \emptyset.$$

Thus $\mathrm{H}(\bar{r})$ belongs to the set on the right-hand side of the equality.

Completeness. Let $\bar{r} \in \mathrm{gr}(\{r\}, J_i^t)$ be a ground rule such that

$$\mathrm{B}(\bar{r}) \cap \Delta_i^t \neq \emptyset.$$

By Lemma 1, the corresponding clique $\mathcal{C}(\bar{r})$ contains at least one edge in $\Delta\mathcal{E}_{i,r}^t$. Therefore, the algorithm will consider $\mathcal{C}(\bar{r})$ when seeding the enumeration from that edge. Consequently, the clique corresponding to $\bar{r}$ will be generated.

Uniqueness. A clique may contain multiple Δ-edges and could therefore be discovered multiple times. However, a clique is reported only if its owner edge coincides with the seeding edge. Since $rank$ defines a total order on edges, every clique has a unique owner edge. Hence, each clique is reported exactly once.

Combining soundness, completeness, and uniqueness, H_i^t equals

$$\{\mathrm{H}(\bar{r}) \mid \bar{r} \in \mathrm{gr}(\mathcal{R}_i, J_i^t), \mathrm{B}(\bar{r}) \cap \Delta_i^t \neq \emptyset\}.$$

By the definition of semi-naive evaluation, Δ_i^{t+1} is obtained by removing already known facts, yielding

$$\Delta_i^{t+1} = H_i^t \setminus J_i^t.$$ □

For rules containing body literals of arity greater than 2, these literals must be verified explicitly for each candidate ground instance. If a candidate is inapplicable due to such a higher-arity literal, it must be queued and rechecked in every subsequent iteration, since Δ-edge seeding provides no guarantee of regenerating it.

6 Parallelized Architecture

The Δ-edge anchored enumeration procedure from the previous section provides the rule-level work units needed for semi-naive evaluation; we now show how these work units are scheduled in a synchronous two-level parallel architecture. Figure 1 shows the synchronous execution architecture with two-level parallelization where the coordination of facts happens after each iteration. Level L1 is the primary driver of parallelism, evaluating rules concurrently. Depending on the structure of the input program, the per-rule execution time may vary greatly, or the number of rules may be too small,

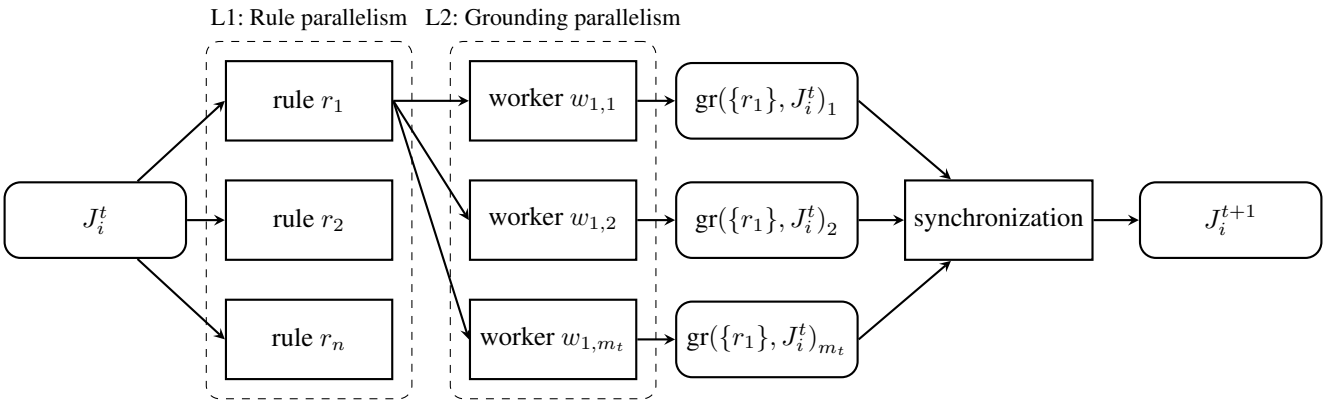

Figure 1: Synchronous execution architecture with two-level parallelization. Level L1 denotes rule-level parallelism, where rules are evaluated concurrently, and Level L2 denotes grounding parallelism, where workers ground instances of a rule in parallel. Let $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ be a Datalog program with rule strata $\mathcal{R}_1, \ldots, \mathcal{R}_\ell$. During semi-naive evaluation, rule stratum $\mathcal{R}_i = \{r_1, \ldots, r_n\}$ is processed with current fact set J_i^t. For each rule $r \in \mathcal{R}_i$, a rule task grounds all newly applicable instances $\mathrm{gr}(\{r\}, J_i^t)$. Each rule task r creates $m_t \geq 1$ worker tasks that ground disjoint subsets, i.e., $\mathrm{gr}(\{r\}, J_i^t) = \mathrm{gr}(\{r\}, J_i^t)_1 \uplus \cdots \uplus \mathrm{gr}(\{r\}, J_i^t)_{m_t}$. Finally, a synchronization phase merges the heads of all derived ground rules into the fact set J_i^t to obtain J_i^{t+1}.

limiting parallel scaling. Therefore, level L2 parallelizes the grounding phase of each rule by distributing Δ-edges among worker tasks, where each worker emits only those cliques that it owns according to the seeding edge. Choosing the number of workers m_t is the responsibility of the load balancer, which aims to minimize the sequential execution time of iteration t.

The two levels of parallelism target different work. L1 is coarse-grained: rules of the same stratum can run concurrently, synchronizing only when derived tuples are merged into the next database. L2 is finer-grained, partitioning generated Δ-edges within one expensive rule. Some domains expose enough independent rule work for L1, whereas skewed rule costs may require L2 to avoid idle workers.

The synchronous structure keeps ownership and merge points explicit. Workers derive tuples locally from a fixed input database and merge local outputs only after all work units finish. This avoids fine-grained synchronization during clique enumeration and makes duplicate avoidance local to the Δ-edge ownership rule, while retaining standard semi-naive fixed-point semantics.

To reduce contention in memory allocation, all data structures used during search are stored in geometrically growing flat byte arrays that serialize dynamically sized objects.

7 Experiments

We implemented our planner, Tyr, in C++ with Python bindings. It uses the Loki PDDL parser and normalizer (Drexler 2026), which implements the PDDL normalization described in Section 4 of Helmert (2009). We will make Tyr available online. We use $N \in \{1, 2, 4, 8\}$ rule-level workers, denote the corresponding configuration by Tyr-N, and use $m_t = 1$ grounding worker by default. For $N = 8$, we also consider the configuration Tyr-8-2, which uses $m_t = 2$ grounding workers whenever more than 1024 Δ-edges are generated, distributing them equally in round-robin fashion. More sophisticated load-balancing techniques are beyond the scope of this paper. As a lifted baseline, we consider the Powerlifted (PL) planner (Corrêa et al. 2020), and as a ground baseline, we consider the Fast Downward (FD) planner (Helmert 2006). All planners use lazy greedy best-first search with the h_{FF} heuristic and alternate between preferred and non-preferred queues (Richter and Helmert 2009). Since lazy search uses deferred heuristic evaluation, we follow the boosted dual-queue setting proposed for lazy search: we increase the preferred-queue weight by 1000 whenever the current best heuristic estimate improves, and decrease it by 1 after each node expansion. We evaluate all planners on the Autoscale Agile (AS) and Hard-To-Ground (HTG) benchmark sets, using a time limit of 10 minutes and a memory limit of 16 GiB.

7.1 Speedup Analysis

Parallelizing Datalog evaluation can improve planner performance only if two conditions are met: the sequential implementation must incur little avoidable overhead, and a sufficiently large fraction of the remaining runtime must lie in parallelizable phases. We therefore first analyze Tyr's runtime structure before comparing its end-to-end performance against the baselines. This analysis isolates the components affected by rule-level and grounding-level parallelism and derives empirical speedup limits from the observed time distribution.

We focus on the HTG benchmark set, whose tasks are challenging for ground planners and exhibit substantial Datalog execution times, making them the primary target for parallelization. In particular, we address the following central questions:

Q1 How much of the total time is effectively spent in the Datalog execution of our planner?

Q2 Is our single-core implementation sufficiently efficient to motivate its parallelization?

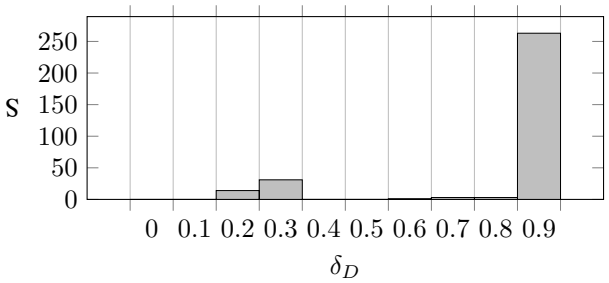

Figure 2: The number of solved tasks (S) grouped by their total wallclock time spent in the Datalog execution (δ_D).

		Action			FF		
δ_D	N	$\delta^{\text{seq}}_{\text{phase}}$	$\delta^{\text{inter}}_{\text{phase}}$	$\delta^{\text{intra}}_{\text{phase}}$	$\delta^{\text{seq}}_{\text{phase}}$	$\delta^{\text{inter}}_{\text{phase}}$	$\delta^{\text{intra}}_{\text{phase}}$
$[0, 0.5)$	45	0.264	0.543	0.999	0.068	0.999	0.999
$[0.5, 1]$	270	0.004	0.779	0.407	0.976	0.947	0.772
$[0, 1]$	315	0.046	0.559	0.944	0.831	0.949	0.775

Table 1: Fraction of total wall-clock time spent in each execution phase, defined as $\delta_{\text{phase}} = \frac{\sum_i T_{\text{phase},i}}{\sum_i T_{\text{total},i}}$, where the sums range over all tasks in the respective δ_D interval.

Q3 What are the limits of the speedup given our empirical data?

Q4 What are the current weaknesses of our approach and how can we address them?

Datalog Fraction. The Datalog fraction, denoted by δ_D, is defined as T_D/T_{tot}, where T_D is the total wall-clock time spent in Datalog execution and T_{tot} is the total wall-clock time of the planner execution.

Figure 2 shows the distribution of solved tasks grouped by δ_D for tasks whose total wall-clock time is at least 6 seconds (1% of the time limit). Among the 315 such tasks, 45 have $\delta_D \in [0, 0.5)$ and 270 have $\delta_D \in [0.5, 1]$. Roughly 250 tasks fall into the interval $\delta_D \in [0.9, 1]$. A large Datalog fraction suggests substantial potential for parallelization. For instance, if $\delta_D = 0.9$, Amdahl's law bounds the maximum achievable speedup by $1/(1 - 0.9) = 10$. Even with perfect parallelization of the Datalog component, the achievable speedup is therefore limited by the remaining sequential fraction. Before analyzing these speedup limits in more detail, we first evaluate the efficiency of the single-core implementation to justify parallelization.

7.2 Search Time Per Expanded Node

To assess whether our single-core implementation is sufficiently efficient, Figure 3 compares the search time per expanded node of PL and Tyr-1. For tasks with $\delta_D \in [0.5, 1]$, the picture is largely balanced: neither configuration dominates, and performance differences are task-dependent. This indicates that the single-core implementation is sufficiently efficient to motivate parallelization.

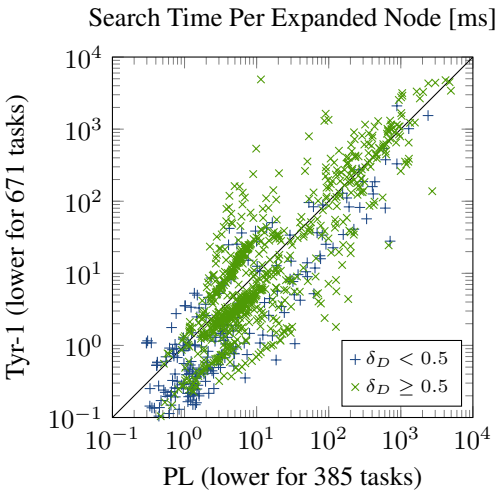

Figure 3: Search time per expanded node in milliseconds of PL against the single-core Tyr-1 configuration. We separate tasks with $\delta_D \in [0, 0.5)$ from tasks with $\delta_D \in [0.5, 1]$.

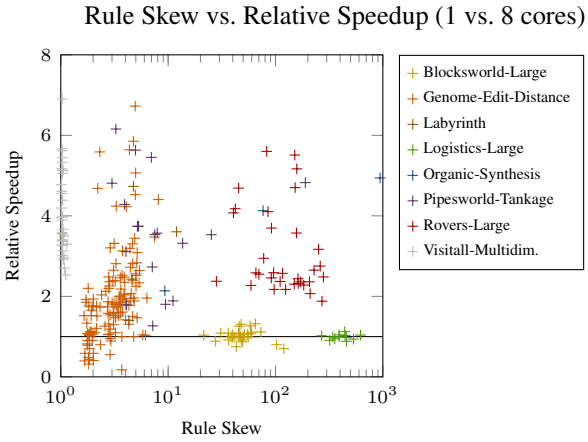

Figure 4: Per domain average rule skew against relative speed from Tyr-1 to Tyr-8. The horizontal line depicts no speedup.

7.3 Empirical Speedup Limits

Using the single-core measurements that separate sequential and parallelizable phases, we derive average empirical speedup limits. Table 1 reports the fraction of wall-clock time spent in the Action (δ_{Action}) and FF (δ_{FF}) Datalog evaluation phases. Across all tasks ($\delta_D \in [0, 1]$), Action accounts for 4.6% of total wall-clock time, while FF accounts for 83.1%. For $\delta_D \in [0.5, 1]$, where parallelization is feasible, FF reaches 97.6%. The parallelizable share within FF, i.e., rule evaluation rather than the sequential merge phase ($\delta^{\text{inter}}_{\text{FF}}$), is 94.7%. Since only Datalog execution is parallelized, the overall rule-parallel fraction is $0.976 \times 0.947 \approx 0.924$. The Amdahl bound is therefore $1/(1 - 0.924) \approx 13$. The intra-phase fraction ($\delta^{\text{intra}}_{\text{FF}}$) of 77.2% further bounds intra-rule speedups through load balancing by $1/(1 - 0.772) \approx 4.39$, which would be required to attain this average limit.

	Domain	Baselines				Ours									
		FD		PL		Tyr-1		Tyr-2		Tyr-4		Tyr-8		Tyr-8-2	
		S	T	S	T	S	T	S	T	S	T	S	T	S	T
	ASA Summary (990)	**622**	0.96	487	4.73	494	2.94	506	2.25	509	2.44	503	2.99	506	2.97
Hard-To-Ground	Blocksworld-Large (40)	12	9.41	7	36.86	39	0.73	**40**	0.75	39	0.82	39	0.80	**40**	0.77
	Childsnack-Large (144)	113	1.38	78	4.82	**124**	1.51	123	1.52	123	1.64	122	1.62	123	1.61
	Genome-Edit-Distance (312)	**312**	1.89	306	5.07	303	3.71	308	2.86	309	2.81	311	3.57	311	3.52
	Labyrinth (40)	**12**	–	0	–	7	–	8	–	8	–	8	–	10	–
	Logistics-Large (40)	36	207.31	**40**	1.89	**40**	3.95	**40**	3.80	**40**	3.87	**40**	3.82	**40**	3.82
	Organic-Synthesis (56)	21	3.78	**48**	0.04	39	0.14	39	0.14	39	0.14	39	0.17	39	0.16
	Pipesworld-Tankage (50)	19	5.66	26	0.90	28	2.07	33	1.58	**36**	1.29	**36**	1.23	35	1.21
	Rovers-Large (40)	14	99.34	**40**	15.89	**40**	12.68	**40**	8.93	**40**	5.96	**40**	5.56	**40**	5.42
	Visitall-Multidim. (180)	66	16.22	134	0.67	131	0.46	134	0.37	139	0.31	142	0.32	**143**	0.29
	HTG (902)	605	10.72	679	2.21	751	1.17	765	0.96	772	0.90	776	0.95	**781**	0.95
	Total Summary (1892)	1227	1.64	1166	3.50	1245	2.56	1271	1.94	1281	1.97	1279	2.30	**1287**	2.29

Table 2: Comparison of planners using lazy greedy best-first search with the FF heuristic: Fast Downward (FD), Powerlifted (PL), and our lifted planner with N cores (Tyr-N). For each domain, we report solved tasks (S) and, for commonly solved tasks, the geometric mean total time in seconds (T). The lifted planners with the highest domain coverage are shown in bold.

7.4 Observed Empirical Speedups

Figure 4 compares, for a subset of tasks, the *rule skew*, defined as the ratio of the maximum to the median total wall-clock rule execution time across rules, and the *speedup*, defined as the total wall-clock execution-time ratio of Tyr-1 to Tyr-8. Most speedups fall in the range of 2 to 4, with fewer between 4 and 7. For the Table 2 domains where Tyr improves speed, namely Visitall-Multidim., Pipesworld-Tankage, Logistics-Large, and Rovers-Large, we observe the following. In Visitall, rule skew is close to 1, with speedups of at least 2 and up to 6. In the other domains, speedups remain significant but lower, motivating better load balancing via grounding parallelization. Runtime also improves in Rovers-Large, where Tyr-1 already solves all tasks. In Blocksworld-Large, where rule skew is consistently high, we observe no coverage improvements.

7.5 Overall Results

Table 2 shows that no single planner dominates uniformly across all domains. The results instead split by benchmark set and planner type. On Autoscale Agile, FD achieves the highest coverage, indicating that grounding is often not the main bottleneck and that mature ground search remains highly competitive. On Hard-To-Ground, the lifted planners clearly outperform FD in coverage, consistent with the benchmark design: avoiding full grounding helps when the grounded representation is difficult to construct or store.

Among lifted planners, Tyr improves most consistently on HTG, where Tyr-8-2 solves 781 tasks compared to 679 for PL and 605 for FD. This advantage is domain-dependent, suggesting that performance reflects the interaction between the lifted representation, generated Datalog program, and search-space structure. PL transforms the Datalog program to improve join-based evaluation, whereas Tyr avoids additional relations to reduce synchronization overhead. We therefore interpret the domain-level differences primarily as complementary strengths rather than uniform dominance: Tyr performs particularly well on Blocksworld-Large, while PL remains stronger on Organic-Synthesis.

The HTG runtimes further indicate that Tyr benefits from rule-level parallelism on domains with expensive Datalog evaluation. Runtime decreases almost consistently as the number of rule-level workers increases, for example on Rovers-Large-Simple. The additional grounding-level parallelism in Tyr-8-2 yields only modest gains, suggesting that the dominant exploitable parallelism in the current implementation and benchmark set is coarse rule-level evaluation rather than splitting individual rules into multiple grounding tasks. This limitation is specific to the simple grounding-level splitting strategy evaluated here; it does not preclude stronger gains from finer-grained Datalog parallelization, such as the parallel relation operations used in systems like Soufflé (Jordan, Scholz, and Subotić 2016).

8 Conclusions

We presented a lifted planner based on semi-naive bottom-up Datalog evaluation with two levels of parallelism: rule-level parallelism and grounding parallelism. Our approach uses a Δ-edge-anchored clique-enumeration scheme to generate the ground rule instances required for semi-naive evaluation without duplicates. The empirical results indicate that this architecture scales well with the number of cores, typically achieving speedups of 2–4x and up to 6x on 8 cores, while our parallel-fraction analysis suggests a speedup limit of about 13x. In addition, the resulting planner substantially improves overall benchmark coverage, establishing a new state of the art for lifted FF-style planning in our evaluation. Future work includes better load-balancing strategies and combining our approach with hash-distributed search to scale the method horizontally and further improve performance.

9 Acknowledgements

This work was supported by the Swedish Foundation for Strategic Research and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The computations were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS), partially funded by the Swedish Research Council through grant agreement no. 2022-06725.

References

Bancilhon, F.; and Ramakrishnan, R. 1986. An Amateur's Introduction to Recursive Query Processing Strategies. In Zaniolo, C., ed., *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, 16–52. ACM Press.

Bron, C.; and Kerbosch, J. 1973. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16: 575–577.

Ceri, S.; Gottlob, G.; and Tanca, L. 1989. What you Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering*, 1: 146–166.

Corrêa, A. B.; Francès, G.; Pommerening, F.; and Helmert, M. 2021. Delete-Relaxation Heuristics for Lifted Classical Planning. In Goldman, R. P.; Biundo, S.; and Katz, M., eds., *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)*, 94–102. AAAI Press.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2020. Lifted Successor Generation using Query Optimization Techniques. In Beck, J. C.; Karpas, E.; and Sohrabi, S., eds., *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)*, 80–89. AAAI Press.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2022. The FF Heuristic for Lifted Classical Planning. In Honavar, V.; and Spaan, M., eds., *Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI 2022)*, 9716–9723. AAAI Press.

Das, A.; and Zaniolo, C. 2019. A Case for Stale Synchronous Distributed Model for Declarative Recursive Computation. *Theory and Practice of Logic Programming*, 19: 1056–1072.

Drexler, D. 2026. Loki: A PDDL Parser and Normalizer. https://doi.org/10.5281/zenodo.20081136.

Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds. 2009. *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*. AAAI Press.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26: 191–246.

Helmert, M. 2009. Concise Finite-Domain Representations for PDDL Planning Tasks. *Artificial Intelligence*, 173: 503–535.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14: 253–302.

Jinnai, Y.; and Fukunaga, A. 2017. On Hash-Based Work Distribution Methods for Parallel Best-First Search. *Journal of Artificial Intelligence Research*, 60: 491–548.

Jordan, H.; Scholz, B.; and Subotić, P. 2016. Soufflé: On Synthesis of Program Analyzers. In Chaudhuri, S.; and Farzan, A., eds., *Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2016), Part II*, volume 9780 of *Lecture Notes in Computer Science*. Springer-Verlag.

Kishimoto, A.; Fukunaga, A.; and Botea, A. 2009. Scalable, Parallel Best-First Search for Optimal Sequential Planning. In (Gerevini et al. 2009), 201–208.

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

Ngo, H. Q.; Porat, E.; Ré, C.; and Rudra, A. 2018. Worst-case Optimal Join Algorithms. *Journal of the ACM*, 65(3): 16:1–16:40.

Phillips, C. A.; Wang, K.; Baker, E. J.; Bubier, J. A.; Chesler, E. J.; and Langston, M. A. 2019. On Finding and Enumerating Maximal and Maximum k-Partite Cliques in k-Partite Graphs. *Algorithms*, 12.

Richter, S.; and Helmert, M. 2009. Preferred Operators and Deferred Evaluation in Satisficing Planning. In (Gerevini et al. 2009), 273–280.

Shimoda, T.; and Fukunaga, A. 2025a. Decoupling Generation and Evaluation for Parallel Greedy Best-First Search. In Likhachev, M.; Rudová, H.; and Scala, E., eds., *Proceedings of the 18th Annual Symposium on Combinatorial Search (SoCS 2025)*, 201–205. AAAI Press.

Shimoda, T.; and Fukunaga, A. 2025b. Parallel Greedy Best-First Search with a Bound on Expansions Relative to Sequential Search. In Shah, J.; and Kolter, Z., eds., *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2025)*, 26668–26677. AAAI Press.

Shkapsky, A.; Yang, M.; Interlandi, M.; Chiu, H.; Condie, T.; and Zaniolo, C. 2016. Big Data Analytics with Datalog Queries on Spark. In Özcan, F.; Koutrika, G.; and Madden, S., eds., *Proceedings of the 2016 International Conference on Management of Data*, 1135–1149. ACM.

Ståhlberg, S. 2023. Lifted Successor Generation by Maximum Clique Enumeration. In Gal, K.; Nowé, A.; Nalepa, G. J.; Fairstein, R.; and Rădulescu, R., eds., *Proceedings of the 26th European Conference on Artificial Intelligence (ECAI 2023)*, 2194–2201. IOS Press.

Vernhes, S.; Infantes, G.; and Vidal, V. 2013. Landmark-based Meta Best-First Search Algorithm: First Parallelization Attempt and Evaluation. In *ICAPS 2013 Workshop on Heuristics and Search for Domain-independent Planning (HSDIP)*, 44–52.

Vidal, V.; Bordeaux, L.; and Hamadi, Y. 2010. Adaptive K-Parallel Best-First Search: A Simple but Efficient Algorithm for Multi-Core Domain-Independent Planning. In Felner, A.; and Sturtevant, N., eds., *Proceedings of the Third Annual Symposium on Combinatorial Search (SoCS 2010)*, 100–107. AAAI Press.

Wang, K.; Yu, K.; and Long, C. 2024. Efficient k-Clique Listing: An Edge-Oriented Branching Strategy. *Proceedings of the ACM on Management of Data*, 2(1): 7:1–7:26.