Distributed Parallel Datalog for Lifted Planning

Oliver Joergensen, Dominik Drexler, Jendrik Seipp

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

Abstract

Lifted classical planners avoid the combinatorial cost of full grounding by searching directly over the first-order task representation. Recent work has parallelized lifted heuristic search using techniques from parallel Datalog evaluation. We study an orthogonal approach: distributing search across multiple workers, which has proved effective in grounded planning and Datalog solvers. We take first steps toward distributed parallel lifted planning by generalizing hash-based distributed search to the lifted setting. Grounded workload-distribution methods often exploit domain transition graphs, whose nodes correspond to mutually exclusive ground-atom groups. Since these graphs are unavailable without grounding, we use invariants as a lifted proxy for such groups and evaluate whether they provide better partitioning keys than predicates alone. Initial experiments with a Python prototype show no clear benefit from incorporating mutex information alone, but suggest that hash-distributed lifted search is a promising direction.

1 Introduction

Classical planning aims to find a sequence of actions from a given initial state to a state satisfying the goal. Solving hard planning problems requires informative heuristics and efficient search. *Lifted* planners avoid the combinatorial cost of full grounding by operating directly on the symbolic, first-order task representation, making them effective on problems with large object sets. This expressiveness is costly: each successor generation and heuristic evaluation reasons over first-order structure and is far more expensive than its grounded counterpart. Exploiting the many cores of modern hardware is therefore an attractive way to speed up lifted search.

Modern lifted planners exploit that central computations, such as successor generation and heuristic evaluation, can be expressed as *database inference rules* in Datalog (Corrêa et al. 2020, 2022). Recent work has parallelized these computations using techniques from parallel Datalog evaluation, achieving speedups of 2–6$\times$ on 8 cores for difficult-to-ground benchmark tasks (Drexler, Joergensen, and Seipp 2026). However, this rule-level parallelism can be limited by *rule skew*, where some rules dominate the inference workload and become bottlenecks.

An orthogonal approach is to *distribute* the search itself: whereas parallel workers share memory on a single machine, distributed workers hold private memories and cooperate purely via message passing. Distribution therefore sidesteps rule skew, since each worker runs its own Datalog evaluation, and adds an axis along which compute can scale beyond a single machine. Hash-distributed search realizes this idea and has been highly effective in grounded planning (Kishimoto, Fukunaga, and Botea 2009; Jinnai and Fukunaga 2017): it assigns each state to an owner worker via a hash function, which determines where generated states are inserted and deduplicated. The hash function faces two competing objectives: balancing the workload evenly across workers and minimizing communication, i.e., the fraction of states whose owner differs from the worker that generates them. A uniform hash optimizes balance but maximizes communication; structure-aware hash functions reduce communication by co-locating related states on the same worker while retaining enough granularity for balance. Our work targets this trade-off: we seek a structure-aware hash function that is computable in the lifted setting. In grounded planning, effective hash functions can be derived from domain transition graphs (DTGs) (Helmert 2006), which capture how finite-domain state variables change under actions. Lifting this idea is non-trivial because these graphs rely on grounded finite-domain variables and grounded action transitions.

Invariants (e.g., Fox and Long 1998; Rintanen 2000; Helmert 2009; Fišer 2020) provide lifted access to part of this structure. Their ground instances induce mutually exclusive ground-atom groups, which underlie the finite-domain variables used in domain transition graphs. DTG-based hashing exploits two sources of information: mutex groups define meaningful state components, while the transition graph identifies components whose values change slowly and therefore reduce communication (Jinnai and Fukunaga 2017). We focus on the first source and evaluate whether invariant-induced mutex groups alone provide better partitioning features than predicate-based hashing, without constructing grounded transition graphs.

We take the first steps toward distributed parallel lifted planning by generalizing hash-based distributed search to the lifted setting. We replace DTG-based state-space partitioning with invariant-based partitioning derived from the lifted task and define an invariant-based Zobrist hash that supports efficient incremental updates. We implement our framework in a Python prototype and evaluate it on 1058 instances from hard-to-ground benchmark domains as an initial study of the

design space. Each worker maintains a local search space, including its own set of encountered ground atoms, and uses a hash function to route generated states to their owners. Our results show that no state-space partitioning strategy dominates across domains. In particular, invariant-induced mutex information, while structurally motivated, does not consistently improve workload balance relative to predicate-based hashing. At the same time, the experiments counter the intuition that maintaining a local frontier per worker would cause memory usage to grow substantially with the number of workers. Overall, the experiments provide initial evidence for the feasibility of distributed lifted search, while also showing that realizing robust speedups will require further engineering of the prototype implementation.

Our contributions are:

- A distributed greedy best-first search framework combining Message Passing Interface (MPI) ranks with local thread parallelism for lifted planning.
- An invariant-based work-distribution function computable from the lifted task, requiring no full grounding and supporting incremental hash updates.
- An empirical evaluation on 1058 benchmark instances analyzing compute allocation and invariant-based workload distribution in distributed lifted search.

2 Related Work

Our approach draws on three lines of research: hash-distributed best-first search, parallel greedy best-first search, and parallel lifted planning via Datalog evaluation.

Hash-Distributed Best-First Search. Parallel best-first search algorithms based on hash-distributed state ownership have been widely studied for classical planning. Hash Distributed A* (Kishimoto, Fukunaga, and Botea 2009) assigns each generated state to a unique worker process via a global hash function, enabling asynchronous parallel search while ensuring each state is expanded at most once. The performance of such algorithms depends critically on the choice of hash function, which determines the balance between communication overhead, the fraction of generated states sent to other workers, and search overhead, the additional states expanded relative to sequential search. Jinnai and Fukunaga (2017) formalize this trade-off and propose Abstract Zobrist Hashing (AZH), which applies Zobrist hashing over abstract feature projections derived from the planning domain. Their most general method, GRAZHDA*, automatically generates these projections by partitioning the DTGs of the planning task to minimize the communication-to-search-overhead ratio. DTGs are compact representations of individual variable transitions in the planning domain, but their construction requires a fully grounded representation of the planning task. Consequently, this method is inherently problematic on problems that are difficult to ground.

Parallel GBFS with Bounded Expansions. When multiple threads share open and closed lists, parallel greedy best-first search (GBFS) can expand significantly more states than sequential GBFS, because threads may simultaneously pursue regions of the search space that sequential search would never visit concurrently. Shimoda and Fukunaga (2025) show that prior parallel GBFS algorithms provide no worst-case bound on this excess work: for any constant T, there exist search topologies on which they expand more than T times as many states as sequential GBFS. They propose OBAT (One Bench At a Time), which restricts how threads interleave exploration of states at different heuristic value levels, and prove that the number of states it expands is within a constant factor of the number expanded by sequential GBFS under some tie-breaking policy. Their practical variant, OBAT_S, decouples successor generation from heuristic evaluation to increase state throughput while retaining the guarantee. The penalty incurred by redundant expansions depends directly on the cost of expanding a single state: in grounded planning, where state expansion and heuristic evaluation are comparatively cheap, excess expansions are less damaging, whereas in lifted planning each expansion is substantially more expensive.

Parallel Lifted Planning via Datalog Evaluation. Several core components of lifted classical planning can be expressed as Datalog rule evaluations over a database of ground facts, including successor generation (Corrêa et al. 2020), task grounding (Helmert 2009), and relaxed planning graph heuristics (Corrêa et al. 2022). Building on this connection, a recent parallel lifted planner based on synchronous semi-naive Datalog evaluation has been proposed (Drexler, Joergensen, and Seipp 2026). Its execution model exploits two complementary levels of parallelism: rule-level parallelism evaluates the rules of a stratum concurrently, while grounding parallelism distributes the enumeration of rule instances across worker tasks. On hard-to-ground benchmarks where the majority of runtime is spent in Datalog execution, this approach achieves an average parallel fraction of 90%, with speedups of 2–4$\times$ and up to 6$\times$ on eight cores. Practical gains are primarily limited by rule skew and the inherently sequential fraction of heuristic computation, with some problems gaining little benefit from parallel evaluation.

3 Background

We cover classical planning, Datalog and semi-naive evaluation (Bancilhon and Ramakrishnan 1986), and hash-based distributed search, which together form the formal basis of our approach.

3.1 Classical Planning

A *classical planning task* is a tuple $\langle P, A, O, S_0, G\rangle$, where $\langle P, A\rangle$ is the *planning domain* and $\langle O, S_0, G\rangle$ is the *task information*. The domain fixes the predicate symbols and actions, while the task information instantiates them with concrete objects.

P is a set of predicate symbols, each with an arity in $\mathbb{N}$. An *atom* of arity k over P has the form $q(x_1, \ldots, x_k)$ for some $q \in P$, where each x_i is a variable or constant, and a *literal* is either an atom p or its negation $\neg p$. The constants are drawn from O, a set of *objects*. A *substitution function* $\rho : X(\rho) \to O$ maps a set of variables $X(\rho)$ to objects; we write $\mathcal{R}(X, O)$ for all such functions over X and O, and $x_1/o_1, \ldots, x_n/o_n$ for the one with $\rho(x_i) = o_i$. We write $\rho \subseteq \rho'$ iff $X(\rho) \subseteq X(\rho')$ and ρ, ρ' agree on $X(\rho)$. Applying

ρ to a syntactic element y, written $y[\rho]$, replaces each variable with its image under ρ. The free variables of y are $X(y)$, its arity is $ar(y) = |X(y)|$, and y is *ground*, written $\bar{y}$, iff $X(y) = \emptyset$.

We identify a *state* S with its set of true ground atoms $\overline{P}(S)$; atoms outside $\overline{P}(S)$ are false. A literal $\bar{p}$ holds in S iff $\bar{p} \in \overline{P}(S)$, and $\neg\bar{p}$ holds iff $\bar{p} \notin \overline{P}(S)$. The task information further comprises the *initial state* S_0 and the *goal* G, a set of ground literals.

Each *action* $a \in A$ has precondition literals $pre(a)$ and effect literals $\mathit{eff}(a)$, and we write $pre^+(a)$, $pre^-(a)$, $\mathit{eff}^+(a)$, $\mathit{eff}^-(a)$ for the positive and negative atoms in each. A ground action $\bar{a}$ is *applicable* in S iff every $\bar{l} \in pre(\bar{a})$ holds in S, yielding the *successor state* S' with $\overline{P}(S') = (\overline{P}(S) \setminus \mathit{eff}^-(\bar{a})) \cup \mathit{eff}^+(\bar{a})$. A *plan* is a sequence of ground actions that, applied from S_0, reaches a state satisfying G.

3.2 Datalog

A *Datalog program* $D = \langle F, R \rangle$ consists of a set of *facts* F (ground atoms) and a set of *rules* R. Each rule $r \in R$ has the form $h \leftarrow b_1, \ldots, b_k$ ($k \geq 0$), where the head $H(r) = h$ is an atom and the body $B(r) = \{b_1, \ldots, b_k\}$ is a set of literals. A ground rule $\bar{r}$ is *applicable* in F if every $\bar{l} \in B(\bar{r})$ holds in F.

A program D over predicates P induces a *predicate dependency graph* with an edge (q, q') whenever q appears in the body and q' in the head of some $r \in R$, labeled $\top$ or $\bot$ for a positive or negative occurrence. D is *stratifiable* if its predicates partition into strata $P_1, \ldots, P_\ell$ such that $\top$-edges satisfy $i \leq j$ and $\bot$-edges satisfy $i < j$, inducing rule strata $R_1, \ldots, R_\ell$.

The *model* of D is the set of all facts derivable from F by exhaustively firing the rules in R, processing one stratum at a time so that negation only references already-completed strata. In practice the model is computed by *semi-naive bottom-up evaluation* (Bancilhon and Ramakrishnan 1986): each stratum is evaluated to a fixed point, but at every iteration only rules whose body matches at least one newly derived fact are reconsidered, avoiding the redundant work of recomputing already-derived consequences.

3.3 Distributed Planning

In hash-based distributed best-first search, N workers cooperatively explore a state space, each maintaining a local open and closed list. A *work distribution function* $\omega : \mathcal{S} \to \{0, \ldots, N-1\}$ maps each state S to a unique *owner* worker $\omega(S)$, partitioning the state space into N disjoint regions (Jinnai and Fukunaga 2017). In hash-based approaches, ω is defined via a hash function $h : \mathcal{S} \to \mathbb{N}$ as

$$\omega(S) = h(S) \bmod N.$$

When a worker generates a successor state S', it forwards S' to worker $\omega(S')$, which alone is responsible for detecting duplicates and inserting S' into its open list. This ownership invariant ensures that each state is stored and expanded by exactly one worker, eliminating the need for global synchronization.

3.4 Zobrist Hashing

Zobrist hashing (Zobrist 1970) is the standard uniform hash for state distribution. Let R map each ground atom p to a uniformly random fixed-width bit string $R[p]$, sampled once before search. The *Zobrist hash* of a state S is

$$Z(S) := \bigoplus_{p \in \overline{P}(S)} R[p],$$

where $\bigoplus$ denotes bitwise XOR. Because XOR is commutative and associative, $Z(S)$ depends only on $\overline{P}(S)$ as a set, and setting $h = Z$ yields a work distribution function that assigns states uniformly at random across workers. Given an applicable ground action $\bar{a}$, the hash of the successor S' can be computed incrementally as

$$Z(S') = Z(S) \oplus \bigoplus_{p \in \mathit{eff}^-(\bar{a})} R[p] \oplus \bigoplus_{p \in \mathit{eff}^+(\bar{a})} R[p]$$

by XORing out deleted atoms and XORing in added atoms, requiring only $|\mathit{eff}(\bar{a})|$ table lookups rather than a full recomputation.

4 Theory and Implementation

The Tyr planner's local thread-level parallelism accelerates Datalog evaluation on a single node, but practical speedups are bounded by rule skew: when one rule in a stratum dominates execution time, additional threads yield diminishing returns as they sit idle waiting for the bottleneck (Drexler, Joergensen, and Seipp 2026). Separately, existing hash-distributed search algorithms use work distribution functions to reduce communication, but the best-performing approaches rely on DTGs, whose construction requires a fully grounded task (Jinnai and Fukunaga 2017), ruling them out in the lifted setting.

We address both limitations together. Distributing search across N MPI ranks lets us trade underutilized local cores for search-space coverage: when rule skew limits the benefit of k local threads within a single Datalog evaluation, the same compute budget instead drives independent workers exploring different regions of the search space. Replacing DTG-based abstractions with invariant-derived ones yields a domain-aware work distribution function computable directly from the lifted task.

4.1 Distributed Greedy Best-First Search

An MPI *rank* is an independent process with private memory; ranks communicate exclusively by message passing and may therefore reside on different compute nodes. Each of the N ranks maintains a local open list, a local closed set, and a local fragment of the search graph. A work distribution function $\omega : \mathcal{S} \to \{0, \ldots, N-1\}$ assigns each state a unique *owner* rank. When a rank generates a successor S' that it does not own, it forwards S' to the owner $\omega(S')$; the owner alone decides whether S' is a duplicate, inserting it into its open list only if not. This ownership invariant eliminates global synchronization on duplicate detection.

Lazy Heuristic Evaluation. The Datalog-based heuristics we build on require a full Datalog program execution per state; while not every lifted heuristic is computed this way, this cost profile is shared by the relaxation heuristics that dominate lifted planning practice (Corrêa et al. 2022). We adopt a *lazy* strategy: a generated successor S' is enqueued with the heuristic value $h(S)$ of its parent as a provisional priority. When S' is dequeued for expansion, its true value $h(S')$ is computed; if $h(S') = \infty$ the state is discarded as a dead end. States that arrive at their owner rank but are pruned as duplicates are never evaluated, so heuristic cost is paid only for states that actually reach expansion.

Termination and Goal Detection. Global termination is confirmed by a distributed token-ring protocol that tracks in-flight message counts, guaranteeing no state can still be in transit when termination is declared. Goal detection is decentralized: any rank that generates a goal state broadcasts a signal, after which all ranks cooperate to reconstruct the solution from their local search-graph fragments.

4.2 Abstraction-Based State Distribution

The choice of ω directly determines the communication-to-search-overhead trade-off identified by Jinnai and Fukunaga (2017): a function that co-locates structurally related states reduces the fraction of successors sent to remote ranks. A uniform random hash minimizes search overhead but ignores problem structure, while a domain-aware hash can substantially reduce communication at the cost of some design effort. The abstract hashing scheme we review next is due to Jinnai and Fukunaga (2017); the invariant-based abstraction, hash and selection strategy that follow are novel.

Abstract Hash Functions. A natural approach is to *project* each state to an abstract feature and hash that feature, co-locating states that share the same abstraction:

$$\omega(S) = h(\phi(S)) \bmod N,$$

where $\phi : \mathcal{S} \to \mathcal{A}$ maps states to abstract features and h hashes those features. When many successors of a state project to the same abstract feature as their parent, they are routed to the same rank, saving communication. This is the principle underlying AZH (Jinnai and Fukunaga 2017), which derives ϕ automatically from DTGs. Since DTG construction requires a grounded task, AZH is not directly applicable in our setting.

As a concrete illustration, consider the $k \times k$ sliding puzzle, with PDDL fluent atoms $\mathsf{at}(t, x, y)$ recording each tile's position. Projecting each tile to the board half it occupies (upper, $y < k/2$, or lower, $y \geq k/2$) and ignoring exact column position yields a simple abstract feature ϕ. Moves within a half do not change $\phi(S)$ and therefore incur no communication, while cross-half moves do. This hand-crafted strategy illustrates the principle and provides a concrete baseline (Figure 1), but cannot be generalized automatically to other domains.

Planning Invariants. Planning invariants provide a domain-general source of abstract features synthesizable directly from the lifted task, without requiring full grounding. A *planning invariant* ι is a set of k_ι lifted atom templates $\{A_1, \ldots, A_{k_\iota}\}$ such that in every reachable state exactly one ground instance of some A_j holds. Template parameters are partitioned into two kinds: *rigid variables* $r_0, \ldots, r_{n_r(\iota)-1}$ are shared across all templates and identify which specific mutex group is active, while *counted variables* $X_c(\iota) = \{c_0, \ldots, c_{n_c(\iota)-1}\}$ are local to individual templates and range over the members within a group.

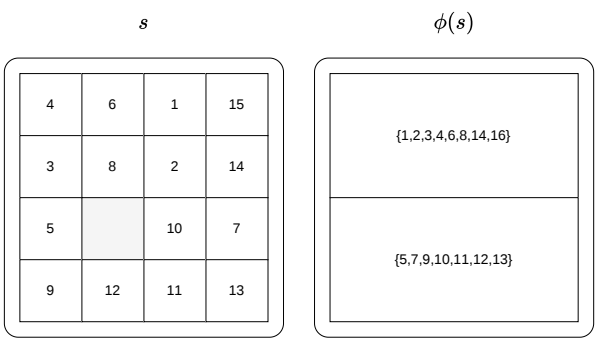

Figure 1: Abstract feature $\phi(S)$ for a 4×4 sliding-puzzle state S. The abstraction records only which numbered tiles occupy the upper two rows (rows 0–1) versus the lower two rows (rows 2–3), discarding exact column positions within each half and ignoring the blank tile (gray). Moves that keep every tile in the same half leave $\phi(S)$ unchanged and therefore incur no inter-rank communication.

A *rigid substitution* ρ_1 assigns an object from O to each rigid variable. It is *valid* for ι if it is type-compatible: for each template A_j and each argument position where a rigid variable appears, the bound object must lie in the predicate-domain of A_j's predicate at that position. We write $Rho(\iota)$ for the set of all valid rigid substitutions. Under a fixed $\rho_1 \in Rho(\iota)$, we define the *mutex group* as

$$\mathcal{M}(\iota, \rho_1) = \big\{\, \bar{A}_j[\rho_1][\rho_2] \;\big|\; j \in \{1, \ldots, k_\iota\}, \\ \rho_2 \in \mathcal{R}(X_c(\iota), O) \,\big\}$$

where $\mathcal{R}(X_c(\iota), O)$ is the set of all total substitutions from the counted variables of ι to objects O. It contains all ground atoms licensed by ι under ρ_1; exactly one member of $\mathcal{M}(\iota, \rho_1)$ holds in every reachable state. Its cardinality $|\mathcal{M}(\iota, \rho_1)|$ provides a reasonable, tractable upper bound on the *counted-variable space* of (ι, ρ_1).

Example 1. In the logistics PDDL domain, the invariant $\iota = \{\mathsf{at}(x, l),\ \mathsf{in}(x, t)\}$ tracks each package x (the *rigid* variable, whose binding selects the mutex group) across locations l and trucks t, which are the *counted* variables ranging over the members within the group.

Invariant-Based Abstractions. Let $\mathcal{I} = \{\iota_0, \ldots, \iota_{m-1}\}$ be the selected invariants. The *invariant abstraction* of a state S is the set of invariant-indexed atom pairs

$$\phi_{\mathcal{I}}(S) := \big\{\, (i,\, p) \;\big|\; p \in \overline{P}(S),\ i \in \{0, \ldots, m{-}1\}, \\ \exists\, \rho_1 \in Rho(\iota_i) : p \in \mathcal{M}(\iota_i,\, \rho_1) \,\big\}.$$

Concretely, $(i, p) \in \phi_{\mathcal{I}}(S)$ whenever fluent atom p holds in S and lies within some mutex group induced by invariant ι_i.

Two states share an abstract feature when they agree on $\phi_{\mathcal{I}}$. An atom may belong to mutex groups of several invariants; all such pairs (i, p) are retained, so a single atom can contribute via multiple invariants.

Example 2. Continuing Example 1, suppose selection picks only the package invariant, $\mathcal{I} = \{\iota_0\}$ with $\iota_0 = \{\mathsf{at}(x, l), \mathsf{in}(x, t)\}$, and consider a state S with $\overline{P}(S) = \{\mathsf{at}(p_1, l_1), \mathsf{in}(p_2, t_1), \mathsf{at}(t_1, l_2)\}$. The package atoms lie in the mutex groups $\mathcal{M}(\iota_0, x/p_1)$ and $\mathcal{M}(\iota_0, x/p_2)$, so $\phi_{\mathcal{I}}(S) = \{(0, \mathsf{at}(p_1, l_1)), (0, \mathsf{in}(p_2, t_1))\}$, while the truck position is not tracked by any selected invariant. Driving the truck to another location changes only $\mathsf{at}(t_1, l_2)$ and leaves $\phi_{\mathcal{I}}(S)$, and hence the owner rank, unchanged, whereas loading p_1 into t_1 changes the abstraction and may route the successor to a different rank.

Zobrist Invariant Hashing. Let $v(i, p)$ be a deterministic hash value for the pair (i, p) whose outputs are well-spread over N workers, so that XOR combinations remain approximately uniform. The *invariant hash* is a Zobrist hash over $\phi_{\mathcal{I}}(S)$, with rank assignment given by

$$H_{\mathcal{I}}(S) := \bigoplus_{(i,p) \in \phi_{\mathcal{I}}(S)} v(i, p),$$
$$\omega_{\mathcal{I}}(S) = H_{\mathcal{I}}(S) \bmod N.$$

Here $v(i, p)$ plays the role of $R[p]$, replacing a shared random table with a deterministic, locally computed hash.

Concretely, we compute $v(i, p)$ as the 64-bit FNV-1a hash of the pair (i, p); including the invariant index i ensures that the same atom produces independent contributions under different invariants. FNV-1a spreads its outputs approximately uniformly over the 64-bit range, so XOR combinations and their residues modulo N remain well-spread. Because v is derived purely from the pair itself, all ranks compute identical hash values without sampling, storing or exchanging a random table.

For efficient computation, define the *aggregate contribution* of atom p as the XOR of $v(i, p)$ over all invariants that pair with p in the abstraction:

$$C_{\mathcal{I}}(p) := \bigoplus_{i\,:\,(i,p) \in \phi_{\mathcal{I}}(S)} v(i, p),$$

for any state S with $p \in \overline{P}(S)$; the set of matching invariants depends only on p, so $C_{\mathcal{I}}(p) = 0$ when no invariant covers p. Since $\bigoplus$ is commutative and associative, $H_{\mathcal{I}}(S) = \bigoplus_{p \in \overline{P}(S)} C_{\mathcal{I}}(p)$, so each atom's contribution can be precomputed once at task-initialization time. Because $H_{\mathcal{I}}$ is a set function over $\overline{P}(S)$, hashing a successor reduces to a small incremental update: for a ground action $\bar{a}$,

$$H_{\mathcal{I}}(S') = H_{\mathcal{I}}(S) \oplus \bigoplus_{p \in \mathit{eff}^-(\bar{a})} C_{\mathcal{I}}(p) \oplus \bigoplus_{p \in \mathit{eff}^+(\bar{a})} C_{\mathcal{I}}(p),$$

requiring at most $|\mathit{eff}(\bar{a})|$ hash computations per transition.

Invariant Selection. An invariant with larger $|\mathcal{M}(\iota, \rho_1)|$ induces more distinct hash values and separates states more finely. Since the exact value requires full reachability information, we estimate it as $\prod_{c_j \in X_c(\iota)} |D_j(\iota)|$, where $D_j(\iota)$ is the union of objects appearing at the argument position of c_j across all templates of ι. We rank invariants by this estimate in ascending order; preferring smaller invariants first gives more controllable granularity and potentially cleaner partitions of the state space. We then traverse this ranking greedily, enumerating the valid rigid substitutions of each invariant one by one and selecting the induced mutex groups, until the product of the selected groups' estimated cardinalities reaches a target of at least βN for a constant oversampling factor $\beta > 1$, providing a buffer against the hash co-locating different abstract states on the same rank. The lazy enumeration avoids materializing all rigid substitutions of large invariants upfront. The predicate-based baseline in our experiments uses the same scheme with predicates in place of mutex groups: it ranks fluent predicates by their estimated ground-atom space, selects the smallest predicates until the product of their sizes reaches βN, and hashes the atoms of selected predicates with the same FNV-1a construction.

5 Experimental Setup

System. We run all experiments on a compute cluster with Intel Xeon Gold 6130 processors. Each node has 32 physical cores and 96 GiB of RAM, and nodes are interconnected via 100 Gbps Intel Omni-Path. We confine each experiment to a single node and limit each run to 30 minutes and 32 GiB of memory.

Configurations. We implemented our distributed lifted planner based on the Tyr planner (Drexler, Joergensen, and Seipp 2026). Each of the N ranks allocates k worker threads to a local Tyr instance, which compiles the task into three stratified Datalog programs: a successor generator, an axiom evaluator, and an FF-RPG heuristic (Corrêa et al. 2022). All configurations use greedy best-first search with the FF-RPG heuristic, lazy heuristic evaluation, and no preferred operator queues. We additionally provide results for Powerlifted (Corrêa et al. 2022) and a C++ implementation of Tyr to contextualize our implementation. The Python bindings used in our prototype are likely to have a substantial impact on performance. Reporting both baselines gives an indication of the gains that porting our Python prototype to C++ could yield, which in turn helps to assess the potential of our approach.

We compare the following configurations:

- POWERLIFTED: single-threaded Powerlifted.
- TYR-1-CPP: single-threaded C++ implementation of Tyr.
- TYR-k: multi-threaded Tyr with k local threads on a single node, serving as our single-node baseline.
- DIST-1-N-ω: our distributed planner with N MPI ranks, each running a single Tyr thread ($k = 1$).
- DIST-k-N-ω: our distributed planner with N ranks and k local threads per rank.

The tag ω denotes the work distribution strategy used to assign states to workers:

- RAND: assigns states to workers uniformly at random.
- PRED: hashes states using a selected subset of predicates.

- INV: uses the invariant-based Zobrist router $\omega_{\mathcal{I}}$ described in Section 4.

All configurations use at most $N \cdot k = 16$ cores on a single node, so every configuration receives the same compute budget. Configurations that only the distributed setting can realize, such as DIST-16-16, would span multiple nodes and allocate substantially more hardware than the single-node baselines, precluding a fair comparison; we therefore exclude them here. This ability to scale beyond a single node is, however, precisely the additional scaling axis that distribution offers, and we consider a multi-node study important future work.

Performance Metrics. For each planner configuration we aggregate results across all benchmark tasks with four metrics:

- *#C* is *coverage*: the number of solved tasks.
- *#M* is the number of tasks that ran out of memory.
- *#T* is the number of tasks that ran out of time.
- *MS* is the sum of *memory scores*.

The memory score for a single run is 0 if the planner exceeds the time limit or the 32 GiB memory limit, and 1 if it uses at most 2 MiB. Between these extremes the score is interpolated logarithmically:

$$MS(m) = 1 - \frac{\ln m}{\ln U},$$

where m is the memory used by the state set and $U = 32\,\text{GiB}$ is the memory limit. This provides more granular insight into memory efficiency than *#M* alone.

Benchmarks. We evaluate on the *Hard-To-Ground* (HTG) benchmark suite,[1] comprising 1058 tasks generated from IPC planning domains. We choose this suite because its domains are particularly well-suited to the lifted setting: grounding these tasks is often infeasible.

6 Experiment Results

We evaluate our distributed parallel planner, measuring coverage, runtime, and parallel speedup under varying numbers of workers N and local threads k.

Coverage and Runtime. Table 1 summarises results over 1058 problem instances. TYR-16 achieves the highest overall coverage (584), while DIST-1-16-PRED attains the best memory score (327), suggesting that distributing search across ranks can yield more memory-efficient solutions even when coverage is comparable. Among the distributed configurations, the invariant-based router consistently produces the fewest out-of-memory failures: DIST-4-4-INV records only 35 memory exhaustions, compared to 40 for DIST-4-4-PRED and 54 for DIST-4-4-RAND. The random router performs worst across the board, with notably lower coverage and higher out-of-memory counts, confirming that unstructured state assignment is an ineffective distribution strategy. The differences between the remaining distributed configurations are small relative to the benchmark size, so we refrain from strong claims about their ranking and leave a statistical per-domain analysis to future work.

[1] https://github.com/abcorrea/htg-domains

	#C	#M	#T	MS
Powerlifted	623	257	178	439
Tyr-1-CPP	625	65	368	399
Tyr-1	554	64	440	306
Tyr-16	**584**	72	402	303
Dist-1-16-INV	550	56	452	312
Dist-1-16-PRED	582	113	363	**327**
Dist-1-16-RAND	521	114	423	295
Dist-4-4-INV	555	35	468	307
Dist-4-4-PRED	540	40	478	299
Dist-4-4-RAND	532	54	472	295

Table 1: Results for all planner configurations on the 1058 *Hard-To-Ground* tasks, reporting coverage (#C), the number of tasks that ran out of memory (#M) or time (#T) and the total memory score (MS). The best value per metric is shown in bold.

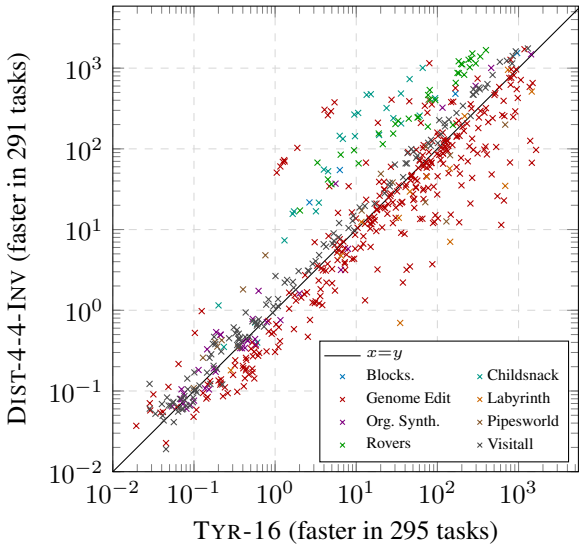

Figure 2: Total runtime for TYR-16 vs. DIST-4-4-INV across all instances.

Baseline Implementation. The single-threaded POWERLIFTED baseline solves 623 tasks, exceeding even the multi-threaded TYR-16 (584). Since TYR-1-CPP performs comparably to POWERLIFTED, we attribute this gap to the overhead of the Python bindings used in our prototype. The prototype implements the search loop, the open and closed lists, state routing and MPI communication in Python, while Datalog evaluation for successor generation and heuristic computation runs natively in Tyr's C++ core. Every generated state therefore crosses the language boundary, where it is converted into Python objects for routing and open-list bookkeeping; this per-state conversion and interpreter overhead dominates the runtime gap. Porting our distributed planner from Python to C++ would therefore likely yield sizeable coverage gains; Section 6.1 reports a preliminary native implementation that

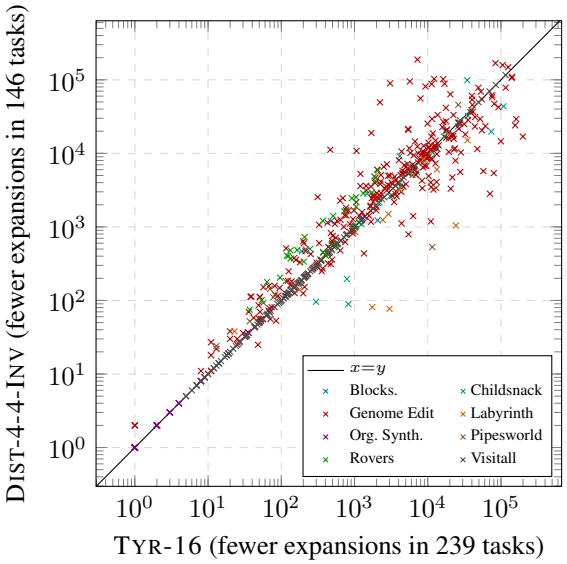

Figure 3: Total expansions for TYR-16 vs. DIST-4-4-INV across all instances.

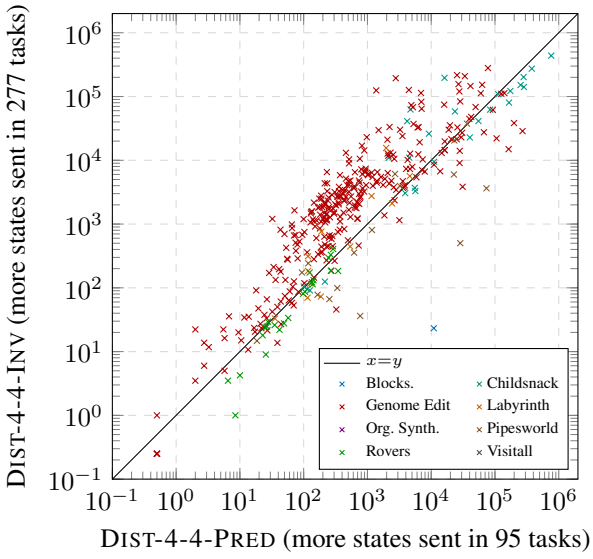

Figure 4: Average number of states sent per worker for DIST-4-4-PRED vs. DIST-4-4-INV.

confirms this.

Parallel Speedup. Figure 2 compares per-instance total runtime between TYR-16 and DIST-4-4-INV. Performance is clearly domain-dependent: Genome and Labyrinth instances are consistently faster to solve in the distributed configuration, whereas Rovers and Organic-Synthesis instances are better served by local parallelism alone. This split indicates that the relative merit of distributing work over allocating additional local threads is strongly domain-dependent.

Figure 3 shows the corresponding total expansions. The conventional expectation is that adding workers increases the total number of expansions: TYR-16 parallelises the operations within a single search and explores no additional states, whereas the four distributed workers each maintain their own open list and pursue distinct frontiers, and although the ownership invariant guarantees that no state is expanded by two workers, the breadth of simultaneously active frontiers should inflate the total expansion count relative to a single-frontier search. The data only partially follow this expectation: the distributed configuration expands slightly more states on average, but a substantial fraction of instances expand *fewer* states under DIST-4-4-INV than under TYR-16. This matches earlier observations that diversifying search effort, e.g., by searching with many agents simultaneously (Knight 1993) or by dovetailing over multiple parameter settings (Valenzano et al. 2010), can reduce the work needed to find a solution as the workers' distinct frontiers can discover a goal state earlier than a single greedy frontier would. We observe no clear domain in which one configuration consistently expands more states than the other.

Communication Overhead. Figure 4 compares the average number of states sent per worker between the predicate-based and invariant-based work distribution functions on DIST-4-4. The invariant-based variant sends more states than the predicate-based one on the majority of instances (277 vs. 95 tasks). A plausible explanation lies in the dynamics of the tracked atoms: invariants may include atoms which are more likely to change often, meaning they are more likely to get routed. The predicate-based selection ranks predicates only by their estimated ground-atom space and can therefore also land on predicates whose atoms change rarely, in which case most successors remain with their generating worker. A second factor is granularity, i.e., the size of the induced distribution space, which neither router controls beyond the selection target βN and which we do not explore systematically: a selected predicate of low arity or with few objects occupies few distinct values, co-locating many states irrespective of their structure, so part of the communication gap may reflect differences in effective granularity rather than in the dynamics of the tracked atoms. Indeed, a non-trivial fraction of instances under the invariant variant complete or fail before any meaningful distribution occurs, indicating that the resulting abstraction granularity is coarser than ideal. Adjusting granularity, either by tuning the oversampling factor β or by adopting a finer-grained distribution strategy, would likely mitigate these effects, but a systematic sweep was beyond our resource budget.

6.1 A Native Implementation

The coverage gap between TYR-1-CPP and our Python prototype in Table 1 motivated a native C++ implementation of our distributed planner, DIST-k-N-CPP, which reuses Tyr's C++ core unchanged and only reimplements the search harness. As a preliminary study, we compare it to the native baseline (TYR-1-CPP, TYR-16-CPP) on the same 1058 instances under a 5-minute budget and 32 GiB limit, so the numbers are not comparable to Table 1. Because the planners terminate differently and report peak memory on incompatible bases, we compare only coverage and, on commonly solved instances, runtime.

In coverage the native planner matches the baseline (513

vs. 505 and 557 vs. 545); as the two share the same engine, this small edge reflects the termination asymmetry rather than faster search, and the real gain is over the Python prototype. Distributing across four ranks raises coverage to 597 and, on the 539 instances both solve, DIST-4-4-CPP (PRED.) is faster than TYR-16-CPP on 421.

Predicate-based routing again beats invariant-based routing (597 vs. 583), consistent with the main study; this may be specific to our single-node setup, where communication is cheap, and invariant-based routing could become favorable on a multi-node deployment, the regime for which hash-distributed search was developed (Kishimoto, Fukunaga, and Botea 2009; Jinnai and Fukunaga 2017).

7 Conclusions

We presented a distributed parallel planner that addresses two complementary limitations of existing approaches. First, rule skew bounds the practical speedup of intra-node thread parallelism: when one rule in a stratum dominates, additional threads sit idle waiting for the bottleneck. Second, the best-performing domain-aware work distribution functions for hash-distributed search rely on DTGs, whose construction requires a fully grounded task and is therefore unavailable in the lifted setting. Our approach combines MPI-based distributed search with an invariant-based work distribution function computable directly from the lifted task; this function requires no grounding and supports efficient incremental hash updates per state transition.

Empirically, no single allocation of a fixed compute budget dominates across domains: distributing across N ranks outperforms using k local threads on Genome and Labyrinth, while local parallelism retains an advantage on Rovers and Organic-Synthesis. The invariant-based distribution reduces out-of-memory failures but sends more states than the predicate-based one and does not improve coverage in the same setting, likely because the selected mutex groups cover frequently changing atoms, while the abstraction is simultaneously too coarse on some instances to distribute states at all. A comparison against Powerlifted indicates that the Python prototype leaves substantial coverage on the table, which a preliminary native C++ implementation confirms: reusing Tyr's C++ core, it removes the prototype's language-boundary overhead and performs on par with the TYR-1-CPP baseline, while distribution adds coverage on an orthogonal scaling axis. Future work includes a parameter sweep over the oversampling factor β, an exploration of alternative distribution strategies and a statistical analysis of the per-domain differences. Further directions are a multi-node study that exploits the additional scaling axis distribution provides, and an extension of the approach to cost-optimal search, where the interaction between distribution and node-ordering guarantees raises additional correctness concerns.

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 *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, 16–52. New York, NY, USA: Association for Computing Machinery.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2020. Lifted Successor Generation using Query Optimization Techniques. In *Proc. ICAPS 2020*, 80–89.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2022. The FF Heuristic for Lifted Classical Planning. In *Proc. AAAI 2022*, 9716–9723.

Drexler, D.; Joergensen, O.; and Seipp, J. 2026. Parallel Lifted Planning via Semi-Naive Datalog Evaluation. arXiv:2605.07584.

Fišer, D. 2020. Lifted Fact-Alternating Mutex Groups and Pruned Grounding of Classical Planning Problems. In *Proc. AAAI 2020*, 9835–9842.

Fox, M.; and Long, D. 1998. The Automatic Inference of State Invariants in TIM. *Journal of Artificial Intelligence Research*, 9: 367–421.

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.

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.

Kishimoto, A.; Fukunaga, A.; and Botea, A. 2009. Scalable, Parallel Best-First Search for Optimal Sequential Planning. In *Proc. ICAPS 2009*, 201–208.

Knight, K. 1993. Are Many Reactive Agents Better Than a Few Deliberative Ones? In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI 1993)*, 432–437. Morgan Kaufmann.

Rintanen, J. 2000. An Iterative Algorithm for Synthesizing Invariants. In *Proc. AAAI 2000*, 806–811.

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

Valenzano, R.; Sturtevant, N. R.; Schaeffer, J.; Buro, K.; and Kishimoto, A. 2010. Simultaneously Searching with Multiple Settings: An Alternative to Parameter Tuning for Suboptimal Single-Agent Search Algorithms. In *Proc. ICAPS 2010*, 177–184.

Zobrist, A. L. 1970. A New Hashing Method with Application for Game Playing. Technical Report 88, Computer

Science Department, The University of Wisconsin, Madison,
WI, USA.