# Abstract Zobrist Hashing: An Efficient Work Distribution Method for Parallel Best-First Search

**Yuu Jinnai** and **Alex Fukunaga**
Department of General Systems Studies
Graduate School of Arts and Sciences
The University of Tokyo

**Abstract**

Hash Distributed A* (HDA*) is an efficient parallel best first algorithm that asynchronously distributes work among the processes using a global hash function. Although Zobrist hashing, the standard hash function used by HDA*, achieves good load balance for many domains, it incurs significant communication overhead since it requires many node transfers among threads. We propose Abstract Zobrist hashing, a new work distribution method for parallel search which reduces node transfers and mitigates communication overhead by using feature projection functions. We evaluate Abstract Zobrist hashing for multicore HDA*, and show that it significantly outperforms previous work distribution methods.

## 1 Introduction

The A* algorithm (Hart, Nilsson, and Raphael 1968) is used in many areas of AI, including planning, scheduling, pathfinding, and sequence alignment. In order to solve large scale problems, both the CPU and memory requirements can be performance bottlenecks. Parallelization is one way to cope with these bottlenecks.

Hash Distributed A* (HDA*) is a parallel best-first search algorithm in which each processor executes A* using local OPEN/CLOSED lists, and generated nodes are assigned (sent) to processors according to a global hash function (Kishimoto, Fukunaga, and Botea 2013). HDA* can be used in distributed memory systems as well as multi-core, shared memory machines, and has been shown to scale up to hundreds of cores with little search overhead.

The performance of HDA* depends on the hash function used for assigning nodes to processors. Kishimoto et al. (2009; 2013) showed that HDA* using the Zobrist hash function (1970) achieves good load balance and low search overhead. Burns et al (2010) noted that Zobrist hashing incurs a heavy communication overhead because many nodes are assigned to processes that are different from their parents, and proposed AHDA*, which used an abstraction-based hash function originally designed for use with Parallel Best-*N*block First (PBNF) (Burns et al. 2010). Abstraction-based work distribution achieves low communication overhead, but at the cost of high search overhead. Thus, there is a tradeoff between communication overhead and search overhead, and previous methods have optimized one at the expense of the other.

In this paper, we investigate node distribution methods for HDA*. We propose *Abstract Zobrist hashing*, a new distribution method which is a hybrid of Zobrist hashing and abstraction. Abstract Zobrist hashing explicitly seeks to balance the uniform hashing property (low search overhead) of Zobrist hashing with the locality property (low communication overhead) provided by abstraction.

The rest of this paper is structured as follows. First, we review HDA* and previous node distribution methods that have been used with HDA*. We then describe Abstract Zobrist hashing, which combines Zobrist hashing and abstraction. We then experimentally evaluate HDA* node distribution methods on the 15-puzzle, 24-puzzle, multiple sequence alignment, and domain-independent planning, focusing on the efficiency, search overhead, and communication overhead achieved by each method. We show that Abstract Zobrist hashing successfully balances communication and search overheads, resulting in higher overall efficiency than either Zobrist hashing or abstraction.

## 2 Background

Hash Distributed A* (HDA*) (Kishimoto, Fukunaga, and Botea 2013) is a parallel A* algorithm which incorporates the idea of hash based distribution of Parallel Retraction A* (PRA*) (Evett et al. 1995) and asynchronous communication of Transposition Table Driven Work Scheduling (TDS) (Romein et al. 1999). In HDA*, each processor has its own OPEN and CLOSED. There is a global hash function which assigns a unique owner thread to every search node. Each thread $T$ executes the following:

1. For all new nodes $n$ in $T$'s message queue, if it is not in CLOSED (not a duplicate), put $n$ in OPEN.
2. Expand node $n$ with highest priority in OPEN. For every generated node $c$, compute hash value $H(c)$, and send $c$ to the thread that owns $H(c)$.

HDA* has two distinguishing features compared to other parallel A* variants. First, the work distribution mechanism is simple, requiring only a hash function. Unlike work-stealing approaches, nodes are not reassigned among

Copyright © 2016, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

threads. Second, there is little coordination overhead because HDA* communicates asynchronously, and locks for an access to shared OPEN/CLOSED are not required because each thread has its own local OPEN/CLOSED. However, contention for the memory bus, which can be considered a form of coordination overhead on multicore machines, can have a significant impact on performance (Kishimoto, Fukunaga, and Botea 2013).

### 2.1 Parallel Overheads in HDA*

Although an ideal parallel best-first search algorithm would achieve a N-fold speedup on N threads, there are several parallel overheads which can prevent HDA* and other parallel search algorithms from achieving perfect linear speedup.

**Communication Overhead (CO)**: We define communication overhead as the ratio of nodes transferred to other threads: CO $:= \frac{\text{\# nodes sent to other threads}}{\text{\# nodes generated}}$. CO is detrimental to performance because of delays due to message transfers, as well as access to data structures such as message queues. HDA* incurs communication overhead when transferring a node from the thread where it is generated to its owner according to the hash function. In general, CO increases with the number of threads. If nodes are assigned randomly to the threads, CO will be proportional to $1 - \frac{1}{\#thread}$.

**Search Overhead (SO):** Parallel search usually expands more nodes than sequential A*. In this paper we define search overhead as SO $:= \frac{\text{\# nodes expanded in parallel}}{\text{\#nodes expanded in sequential search}} - 1$.

In parallel search, SO can arise due to inefficient load balance. We define load balance as $LB := \frac{\text{Max \# nodes assigned to a thread}}{\text{Avg. \# nodes assigned to a thread}}$.

If load balance is poor, a thread which is assigned more nodes than others will become a bottleneck – other threads spend their time expanding less promising nodes, resulting in search overhead. In HDA*, the load balance is determined by the quality of the hash function – with a good hash function, HDA* has been shown to achieve very good load balance (Kishimoto, Fukunaga, and Botea 2013).

There is a fundamental trade-off between CO and SO. Increasing the amount of communication can reduce search overhead at the cost of communication overhead, and vice versa. The optimal tradeoff depends on the characteristic of the problem domain/instance, as well as the hardware/system on which HDA* is executed.

### 2.2 Zobrist Hashing and Abstraction

Since the work distribution in HDA* is solely dependent on a global hash function, the choice of the hash function is crucial to its performance. Kishimoto et al. (2013) used Zobrist hashing (1970), which is widely used in 2-player games such as chess. Figure 1a illustrates the calculation of Zobrist hashing on the 8 Puzzle. The Zobrist hash value of a state $s$, $Z(s)$, is calculated as follows. For simplicity, assume that $s$ is represented as an array of $n$ propositions, $s = (x_0, x_1, ..., x_n)$. Let $R$ be a table containing preinitialized random bit strings.

$$Z(s) := R[x_0] \; xor \; R[x_1] \; xor \; \cdots \; xor \; R[x_n]$$

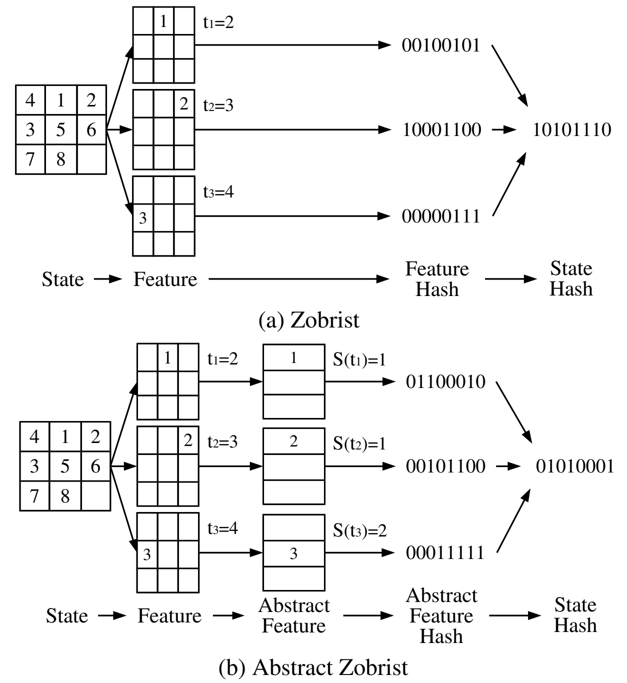



Figure 1: Comparison of calculation of Zobrist hash $Z(s)$ and Abstract Zobrist hash (AZH) value $AZ(s)$ for the 8-puzzle: State $s$ is represented as $s = (t_1, t_2, ..., t_8)$, where $t_i = 1, 2, ..., 9$. The hash value of $s$ is the result of xor'ing a preinitialized random bit vector $R[t_i]$ for each feature (tile) $t_i$. AZH incorporates an additional step which projects features to abstract features (for each feature $t_i$, look up $R[A(t_i)]$ instead of $R[t_i]$).

The main strength of Zobrist hashing is that on domains where the time to process each node (i.e., heuristic computation, node generation) is constant, then, with a hash function which distributes nodes uniformly among threads, it is possible to achieve almost perfect load balance (however, as we shall see later, load balance can be extremely poor if a bad hash function is used). Another advantage to Zobrist hashing is that it can be incrementally computed very efficiently: the hash value of a child node is computed by simply xor'ing the parent node's hash value with the feature hash of the difference between the parent and child states.

Zobrist hashing seeks to distribute nodes uniformly among all threads, without any consideration of the neighborhood structure of the search space graph. As a consequence, communication overhead is high. Assume an ideal implementation that assigns nodes uniformly among threads. Every generated node is sent to another thread with probability $1 - \frac{1}{\#threads}$. Therefore, with 16 threads, $> 90\%$ of the nodes are sent to other threads, so communication costs are incurred for the vast majority of node generations.

Figure 5e shows the CO of HDA* with Zobrist hashing (ZHDA*) on the 24-puzzle. With 4 threads, 72% of the nodes are transferred to another thread. The CO increases to 92% when running on 16 threads. As a consequence, efficiency of ZHDA* clearly decreases as the # of threads

increases (Figure 5b). Since Zobrist hashing achieves good load balance, SO does not increase as the # of threads is increased (Figure 5e). Thus, in this case, CO is the most likely cause of efficiency degradation.

In order to minimize communication overhead in HDA*, Burns et al (2010) proposed AHDA*, which uses *abstraction* based node assignment. AHDA* applies the state space partitioning technique used in PBNF (Burns et al. 2010), which in turn is based on Parallel Structured Duplicate Detection (PSDD) (Zhou and Hansen 2007). Abstraction projects nodes in the state space into *abstract states*, and abstract states are assigned to processors using a modulus operator. Thus, nodes that are projected to the same abstract state are assigned to the same thread. If the abstraction function is defined so that children of node $n$ are usually in the same abstract state as $n$, then communication overhead is minimized. The drawback of this method is that it focuses solely on minimizing communication overhead, and there is no mechanism for equalizing load balance, which can lead to high search overhead. In fact, Figure 5e shows that abstraction has roughly 2-4 times higher search overhead compared to Zobrist hash on the 24-puzzle.

## 3 Abstract Zobrist Hashing

*Abstract Zobrist hashing* (AZH) is a hybrid hashing strategy which incorporating the strengths of both Zobrist hashing and abstraction. AZH augments the Zobrist hashing framework with the idea of projection from abstraction. The AZH value of a state, $AZ(s)$ is:

$$AZ(s) := R[A(x_0)] \; xor \; R[A(x_1)] \; xor \; \cdots \; xor \; R[A(x_n)]$$

where $A$ is a *feature projection function*, which is a many-to-one mapping from each raw feature to an *abstract feature*, and $R$ is a precomputed table defined for each abstract feature. Thus, AZH is a 2-level, hierarchical hash function, where raw features are first projected to abstract features, and Zobrist hashing is applied to the these abstract features. Figure 1 illustrates the computation of AZH for the 8-puzzle, and Figure 2 illustrates the distributions of nodes according to AZH for a set of 8-puzzle states.

AZH seeks to combine the advantages of both abstraction and Zobrist hashing. Communication overhead is minimized by building abstract features that share the same hash value (abstract features are analogous to how abstraction projects states to abstract states), and load balance is achieved by applying Zobrist hashing to the abstract features of each state.

Compared to Zobrist hashing, AZH incurs less communication overhead due to abstract feature-based hashing. While Zobrist hashing assigns a hash value for each node independently, AZH assigns the same hash value for all nodes which shares the same abstract features for all features, reducing the number of node transfers.

In contrast to abstraction-based node assignment, which minimizes communications but does not optimize load balance and search overhead, AZH also seeks good load balance, because the node assignment takes into account all

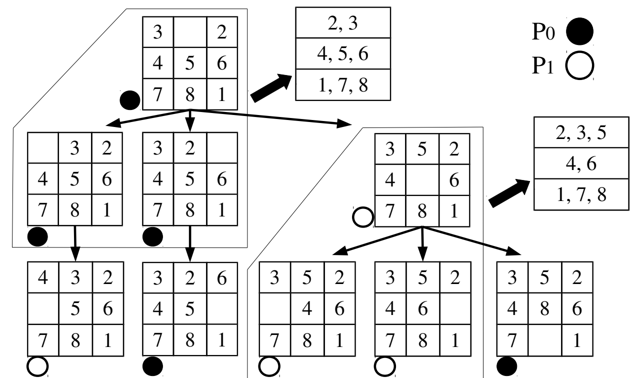



Figure 2: Distribution of nodes according to Abstract Zobrist hashing on 2 threads, using a projection from a cell to a row of the puzzle. States inside the lines are assigned to the same processor because they share the same abstract features.

features in the state, rather than a subset of features.[1]

By adjusting the size of the abstract feature, AZH allows explicit tuning of the trade-off between communication and search overheads (CO vs SO). In general, larger abstract features result in smaller CO and large SO, while smaller abstract features result in larger CO and smaller SO – standard Zobrist hashing is a special case of AZH with abstract feature size 1.

AZH is simple to implement, requiring only an additional projection per feature compared to Zobrist hashing, and we can precompute this projection at initialization. Thus, there is no additional runtime overhead per node during search.

## 4 Experiments

We evaluated the performance of the following HDA* variants on several standard benchmark domains with different characteristics.

- AZHDA*: HDA* using Abstract Zobrist hashing
- ZHDA*: HDA* using Zobrist hashing
- AHDA*: HDA* using Abstraction based work distribution (Burns et al. 2010)

The experiments were run on an Intel Xeon E5-2650 v2 2.60 GHz CPU with 128 GB RAM, using up to 16 cores.

The 15-puzzle experiments in Section 4.1 are based on the code by Burns et al (2010), which includes their implementations of HDA* (called "PHDA*" below), AHDA*, and SafePBNF (we implemented 15-puzzle ZHDA* and AZHDA* as an extension of their code).

[1]To see why ignoring features can lead to poor load balance: consider a blocksworld problem where the goal is on(a,b), on(b,c), on(c,floor), and on(c,floor)=T in the initial state. With a reasonable heuristic, almost all states generated will have on(c,floor)=T. However, if the work distribution function distributes states based only on the feature on(c,floor), then almost all states will be assigned to the same processor, resulting in disastrous load balance.

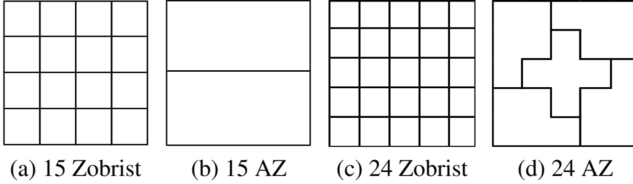

(a) 15 Zobrist (b) 15 AZ (c) 24 Zobrist (d) 24 AZ

Figure 3: The abstract features used by Abstract Zobrist hashing for 15 and 24 puzzle.

For the 24-puzzle, multiple sequence alignment (MSA), and PDDL planning, we used our own implementation of HDA*, using the Pthreads library, try_lock for asynchronous communication, and the Jemalloc memory allocator (Evans 2006). We implemented OPEN as a 2-level bucket for the 24-puzzle and planning (Burns et al. 2012), and a binary heap for MSA (binary heap was faster for MSA).

### 4.1 15 Puzzle

We solved 100 randomly generated instances with solvers using the Manhattan distance heuristic. The average runtime of sequential A* solving these instances was 52.3 seconds. In addition to AZHDA*, ZHDA*, and AHDA*, we also evaluated SafePBNF (Burns et al. 2010) and the configuration of HDA* which was evaluated in (Burns et al. 2010), which we call PHDA* in this paper.

PHDA* uses a perfect hashing scheme by (Korf and Schultze 2005), which maps permutations (tile positions) to lexicographic indices (thread IDs). A perfect hashing scheme computes a unique mapping from permutations (abstract state encoding) to lexicographic indices (thread ID). While this encoding is effective for its original purpose of efficient representation of states for external-memory search, it was *not* designed for the purpose of work distribution.

In the original experiments in (Burns et al. 2010), all of the 15-puzzle parallel A* variants used a binary heap implementation for OPEN. However, it has been shown that for the 15-Puzzle, 2-level bucket OPEN is more than twice as fast as with a binary heap (Burns et al. 2012). Thus, we evaluated with both configurations, binary heap and 2-level bucket. We show results using 2-level bucket OPEN, as it was faster for every algorithm and using binary heap OPEN did not change the results qualitatively.

The features used by Zobrist hashing in ZHDA* are the position of each tile $i$. The projections we used for Abstract Zobrist hashing in AZHDA* are shown in Figure 3. The abstraction used by AHDA* ignores the positions of all tiles except tiles 1,2, and 3 (we tried (1) ignoring all tiles except tiles 1,2, and 3, (2) ignoring all tiles except tiles 1,2,3, and 4, (3) mapping cells to rows, and (5) mapping cells to the blocks which AZHDA* use for a feature projection, and chose (1) because it performed the best).

First, as discussed in Section 2, high search overhead is correlated with load balance. Figure 4, which shows the relationship between load balance and search overhead, indicates a very strong correlation between high load imbalance and search overhead.

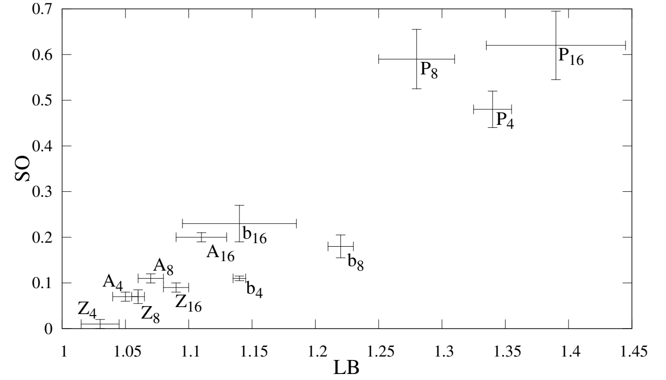



Figure 4: Load balance (LB) and search overhead (SO) on 100 instances of the 15-Puzzle for 4/8/16 threads. A: AZHDA*, Z: ZHDA*, b: AHDA*, P: PHDA*, e.g., $Z_8$ is the LB and SO for Zobrist hashing on 8 threads. 2-D error bars show standard error of the mean for both SO and LB.

Figure 5a shows the efficiency of each method. PHDA* performed extremely poorly compared to all other HDA* variants and SafePBNF. The reason is clear from Figure 5d, which shows the communication and search overheads. PHDA* has both extremely high search overhead and communication overhead compared to all other methods. This shows that the hash function used by PHDA* is not well-suited as a work distribution function.

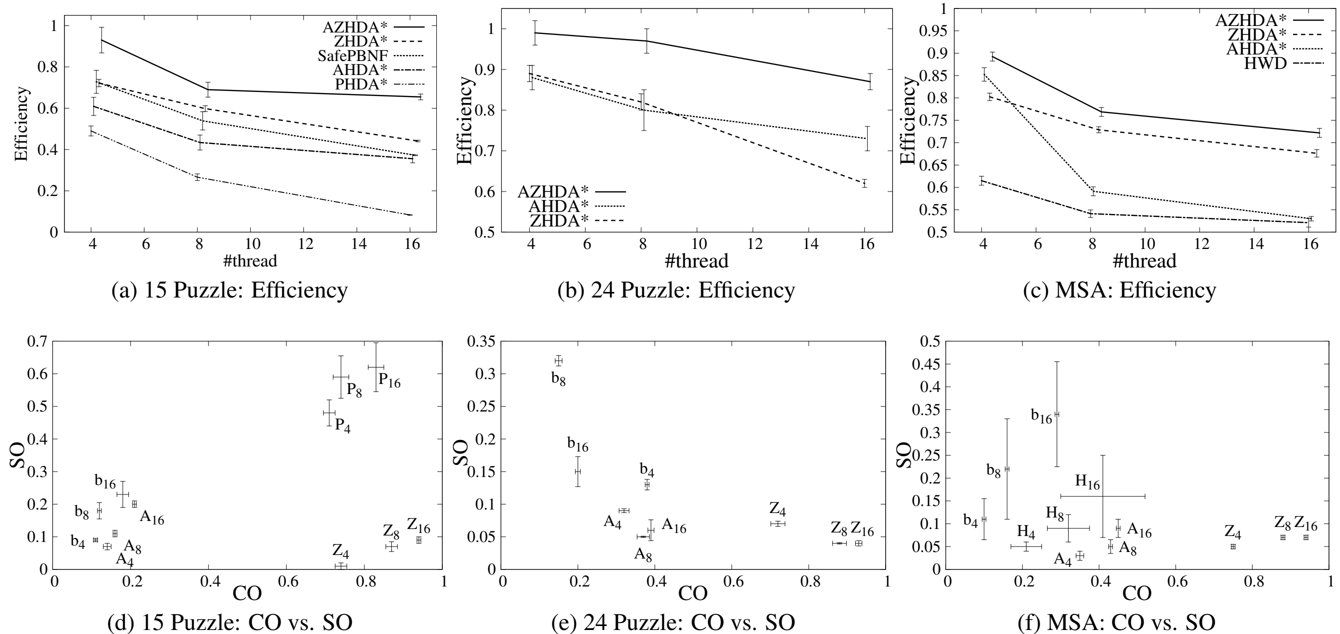
AHDA* had the lowest CO among HDA* variants (Figure 5d), and significantly outperformed PHDA*. However, AHDA* has worse LB than ZHDA* (Figure 4), resulting in higher SO. For the 15-puzzle, this tradeoff is not favorable for AHDA*, and Figures 5a-4 show that ZHDA*, which has significantly better LB and SO, outperforms AHDA*.

According to Figure 5a, SafePBNF outperforms AHDA*, and is comparable to ZHDA* on the 15-puzzle. Although our definition of communication overhead does not apply to SafePBNF, SO for SafePBNF was comparable to AHDA*, 0.11/0.17/0.24 on 4/8/16 threads.

AZHDA* significantly outperformed ZHDA*, AHDA*, and SafePBNF. As shown in Figure 5d, although AZHDA* had higher SO than ZHDA* and higher CO than AHDA*, it achieved a balance between these overheads which resulted in high overall efficiency. The tradeoff between CO and SO depends on each domain and instance. By tuning the size of the abstract feature, we can choose a suitable tradeoff.

### 4.2 24 Puzzle

We generated a set of 100 random instances that could be solved by A* within 1000 seconds. We chose the hardest instances solvable given the memory limitation (128GB). The average runtime of sequential A* on these instances was 219.0 seconds. The average solution lengths of our 24-puzzle instances was 92.9 (the average solution length in (Korf and Felner 2002) was 100.8). We used a disjoint pattern database heuristic (Korf and Felner 2002). Figure 3 shows the feature projections we used for 24 Puzzle. The abstraction used by AHDA* ignores the numbers on all of




(a) 15 Puzzle: Efficiency

(b) 24 Puzzle: Efficiency

(c) MSA: Efficiency

(d) 15 Puzzle: CO vs. SO

(e) 24 Puzzle: CO vs. SO

(f) MSA: CO vs. SO

Figure 5: Efficiency, Communication Overhead (CO), and Search Overhead (SO) for 15-puzzle (100 instances), 24-puzzle (100 instances), and MSA (60 instances) on 4/8/16 threads. OPEN is implemented using a 2-level bucket for sliding-tiles. OPEN for MSA is implemented using a binary heap. In the CO vs SO plot, A: AZHDA*, Z: ZHDA*, b: AHDA*, P: PHDA*, H: HWD, e.g., $Z_8$ is the CO and SO for Zobrist hashing on 8 threads. Error bars show standard error of the mean.

the tiles except tiles 1,2,3,4, and 5 (we tried (1) ignoring all tiles except tiles 1-5, (2) ignoring all tiles except tiles 1-6, (3) ignoring all tiles except tiles 1-4, (4) mapping cells to rows, and (5) mapping cells to the blocks which AZHDA* use for a feature projection, and chose (1), the best performer).

Figure 5b shows the efficiency of each method. AZHDA* outperformed ZHDA* and AHDA*, and Figure 5e suggests that AZHDA* had smaller overall overheads. As on the 15-puzzle, ZHDA* and AHDA* succeed in mitigating only one of the overheads (SO or CO). AZHDA* outperformed both methods as it had comparable SO to ZHDA* while its CO was roughly equal to that of AHDA*.

### 4.3 Multiple Sequence Alignment (MSA)

Multiple Sequence Alignment is the problem to finding a minimum-cost alignment of a set of DNA or amino acid sequences by inserting gaps in each sequence. MSA can be solved by finding the min-cost path between corners in a $n$-dimensional grid. We used 60 benchmark instances, consisting of 10 actual amino acids from BAliBASE 3.0 (Thompson et al. 2005), and 50 randomly generated amino acids. Edge cost is based on the PAM250 matrix score with gap penalty 8. Since there was no significant difference between the behavior of HDA* among actual and random instances, we report the average of all 60 instances. We used pairwise sequence alignment heuristic (Korf et al. 2005).

We set the positions of each sequence to be the feature for Zobrist hashing and Abstract Zobrist hashing. Thus, with $n$ sequences, nodes in the $n$-dimensional hypercube with edge length $l$ share a same hash value. We grouped up $l$ positions in row into a abstract feature for Abstract Zobrist hashing. The abstraction used by AHDA* only considers the position of the longest sequence and ignores the others. We chose this abstraction as it performed the best compared to (1) considering only the longest sequence, (2) the two longest sequences, and (3) the three longest sequences. We also evaluated the performance of Hyperplane Work Distribution (HWD) (Kobayashi, Kishimoto, and Watanabe 2011). ZHDA* suffers from node duplication in non-unit cost domains such as MSA. HWD seeks to reduce node duplication by mapping the $n$-dimension grid to hyperplanes. We determined the plane thickness $d$ using the tuning method shown in (Kobayashi, Kishimoto, and Watanabe 2011) where $\lambda = 0.003$, which yielded the best performance among 0.0003, 0.003, 0.03, and 0.3.

Figure 5c compares the efficiency of each method, and Figure 5f shows the CO and SO. AZHDA* outperformed the other methods. With 4 or 8 threads, AZHDA* had smaller SO than ZHDA*. This is because like HWD, AZHDA* reduced the amount of duplicated nodes in some domains compared to ZHDA*. Our MSA solver expands 300,000 nodes/second, which is relatively slow compared to, e.g., our 24-puzzle solver, which expands 1,400,000 nodes/sec. When node expansions are slow, the relative importance of CO decreases, and SO has a more significant impact on performance in MSA than in the 15/24-Puzzles. Thus, AHDA*, which incurs higher SO, did not perform well compared to ZHDA*. HWD did not perform well, but it was designed for

large-scale, distributed search, and we observed HWD to be more efficient on difficult instances than on easier instances – it is included in this evaluation only to provide another point of reference for evaluating ZHDA* and AZHDA*.

### 4.4 Cost-optimal PDDL Planning with Automatically Generated Abstract Features

We implemented a domain independent planner which handles a subset of PDDL (STRIPS + action costs + types), and uses A* with the pattern database (PDB) heuristic used in (Edelkamp 2001). In the blocksworld domain, our planner running single thread HDA* expanded 211,000 nodes/second, whereas Fast Downward (Helmert 2006) using the PDB heuristic expanded 232,000 nodes/sec. Thus, although the heuristic function is not competitive with the state-of-the-art, we believe this implementation is sufficient as a platform for evaluating parallel efficiency and overheads in domain-independent planning. We evaluated the HDA* planner on 13 instances from the IPC benchmark. We chose 1 nontrivial instance from each domain which was solvable by sequential A* with the PDB heuristic.

Unlike the hand-designed abstract features and abstractions used for the sliding tile puzzle and MSA, our AZHDA* planner builds abstract features completely automatically – we first analyze the domain to find atom groups, which are often used for constructing PDBs (Edelkamp 2001). An atom group is a set of mutually exclusive propositions which exactly one will be true for each reachable state. We build atom groups as in (Edelkamp and Helmert 1999), trying to merge predicates in the order they are written in PDDL file. Each atom group is partitioned into 2 abstract features $S_1$ and $S_2$, based on the atom group's undirected transition graph (nodes are propositions, edges are transitions), as follows: (1) assign the minimal degree node to $S_1$; (2) add to $S_1$ the unassigned node which shares the most edges with nodes in $S_1$; (3) while $|S_1| < n/2$ repeat step (2); (4) assign all unassigned nodes to $S_2$. For AHDA*, we built abstractions using the greedy abstraction algorithm described in (Zhou and Hansen 2006). The greedy abstraction algorithm adds one atom group to the abstract graph at a time, choosing the atom group which minimizes the maximum out-degree of the abstract graph, until the graph size (# of nodes) reaches the threshold given by a parameter $N_{max}$. We set $N_{max} = 10^3$ (we tried $10^2, 10^3, 10^4$ and chose $10^3$ because it performed best).

Table 1 shows the results for 8 threads. Although the efficiency differences are not as pronounced as in the 15/24-puzzle and MSA, AZHDA* outperforms both ZHDA* and AHDA* (higher average efficiency and lower total wall-clock runtime for all problems). Figure 6 shows the CO and SO. The overall trends are similar to the results for our domain-specific solvers for the sliding tiles puzzles and the MSA. AZHDA* has lower search overhead than AHDA*, and lower communication overhead than ZHDA*.

Th greedy algorithm for generating abstract features is simple, and more sophisticated methods are a direction for future work. However, our results show that even this simple method is sufficient to generate abstract features that can reduce CO compared to ZHDA*, resulting in modest improvement on efficiency and the lowest total runtime.

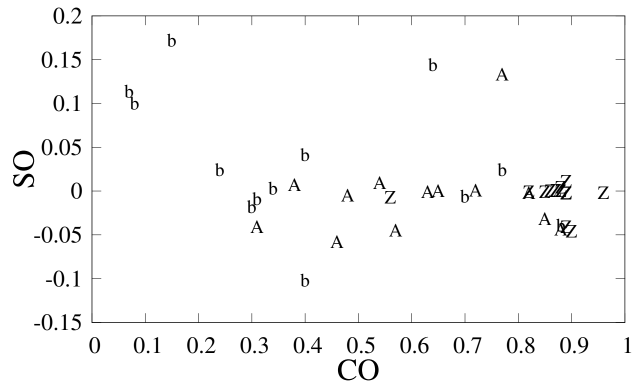



Figure 6: Planning: Communication Overhead (CO) vs. Search Overhead (SO) on 8 threads. A: AZHDA*, Z: ZHDA*, b: AHDA*.

## 5 Conclusions and Future Work

We investigated node distribution methods for HDA*, and showed that previous methods suffered from high communication overhead (ZHDA*), high search overhead (AHDA*), or both (PHDA*), which limited their efficiency. We proposed AZHDA*, which uses Abstract Zobrist hashing, a new distribution method which combines the strengths of both Zobrist hashing and abstraction. Our experimental results showed that AZHDA*, achieves a successful trade-off between communication and search overheads, resulting in better performance than previous work distribution methods.

While the abstract features for our domain-specific sliding-tile and MSA solvers were designed by hand, our preliminary results using automatically generated abstract features for domain-independent planning showed that AZHDA* can achieve a favorable tradeoff between communication and search overheads using completely automatically generated abstract features.

Although our experiments were performed on a single multicore processor because it is easier to perform controlled experiments on a single machine, HDA* has been shown to scale on large-scale distributed memory systems (Kishimoto, Fukunaga, and Botea 2013), and extending this work to large-scale clusters is a direction for future work. Since communication overhead becomes an increasingly more serious problem as (a) the number of cores increases and (b) the connection between cores becomes relatively slower, reducing communication overhead will be increasingly important on large-scale systems.

The comparison with SafePBNF on the 15-Puzzle (Sec 4.1) indicates that ZHDA* and AZHDA* are competitive with SafePBNF. However, evaluation of HDA* as a state-of-the-art multicore search algorithm is not the main goal of this paper, and a full comparison of HDA* and SafePBNF is an avenue for future work.

AZHDA* successfully combines the idea of feature hashing from HDA* and the idea of abstraction from PSDD

| #thread=8 | A* | | AZHDA* | | | ZHDA* | | | AHDA* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Time (sec) | Expd | Eff | CO | SO | Eff | CO | SO | Eff | CO | SO |
| Airport9 | 156.59 | 4902509 | 0.799 | 0.480 | -0.005 | 0.808 | 0.560 | -0.007 | 0.720 | 0.240 | 0.024 |
| Barman1-1 | 30.32 | 4719078 | 0.641 | 0.770 | 0.133 | 0.707 | 0.960 | -0.002 | 0.613 | 0.640 | 0.144 |
| Blocks10-2 | 234.27 | 39226226 | 0.988 | 0.540 | 0.009 | 0.932 | 0.890 | 0.011 | 1.013 | 0.400 | 0.041 |
| Elevators11-26 | 142.66 | 5329127 | 0.654 | 0.380 | 0.007 | 0.601 | 0.850 | -0.001 | 0.648 | 0.070 | 0.114 |
| Floortile1-1 | 254.74 | 30523550 | 0.768 | 0.850 | -0.032 | 0.806 | 0.900 | -0.046 | 0.905 | 0.300 | -0.018 |
| Freecell5 | 141.74 | 9142007 | 0.774 | 0.650 | 0.000 | 0.751 | 0.890 | -0.002 | 0.375 | 0.080 | 0.100 |
| Gripper7 | 77.23 | 10101217 | 0.766 | 0.720 | 0.001 | 0.743 | 0.880 | 0.004 | 0.849 | 0.340 | 0.003 |
| Logistics98-32 | 25.44 | 1837543 | 0.758 | 0.460 | -0.058 | 0.628 | 0.880 | 0.000 | 0.808 | 0.310 | -0.009 |
| PipesNoTk14 | 231.50 | 34862961 | 0.835 | 0.820 | -0.002 | 0.883 | 0.860 | -0.000 | 1.016 | 0.400 | -0.102 |
| Sokoban11-13 | 129.66 | 10048931 | 0.893 | 0.630 | -0.001 | 0.874 | 0.820 | -0.001 | 0.288 | 0.150 | 0.172 |
| Transport08-14 | 85.89 | 9830158 | 0.831 | 0.310 | -0.041 | 0.694 | 0.870 | 0.001 | 0.695 | 0.770 | 0.024 |
| Visitall11-7Half | 10.13 | 1952096 | 0.873 | 0.880 | -0.044 | 0.867 | 0.890 | -0.041 | 0.804 | 0.880 | -0.039 |
| Zenotravel8 | 26.52 | 1330916 | 0.784 | 0.570 | -0.045 | 0.661 | 0.890 | -0.003 | 0.749 | 0.700 | -0.006 |
| Average | 118.97 | 12600486 | **0.797** | 0.620 | -0.006 | 0.766 | 0.857 | -0.007 | 0.729 | 0.406 | 0.034 |
| Total | 1646.74 | 163806319 | Time (sec) | | **282.29** | Time (sec) | | 298.92 | Time (sec) | | 341.73 |

Table 1: Performance of work distribution methods on PDDL planning (8 threads). Time: Walltime on sequential A*, Expd: Node expansion on sequential A*, Eff: Efficiency relative to speedup, CO: Communication overhead, SO: Search overhead.

(Zhou and Hansen 2007) and PBNF. The success of AZHDA* suggests that further combinations of ideas from both lines of work may be fruitful. For example, structure-based parallel search (PSDD, PBNF) partitions the search space into many blocks, and appear to be well-suited for domains with many distinct f-values, which necessitates a binary heap implementation of OPEN, so incorporating fine-grained OPEN lists into HDA* is an avenue for future work.

Finally, Abstract Zobrist hashing is a general work distribution method which is not limited to HDA*. Applications to other algorithms such as TDS (Romein et al. 1999) is an interesting avenue for future work.

## References


Burns, E. A.; Lemons, S.; Ruml, W.; and Zhou, R. 2010. Best-first heuristic search for multicore machines. *Journal of Artificial Intelligence Research* 39:689–743.

Burns, E. A.; Hatem, M.; Leighton, M. J.; and Ruml, W. 2012. Implementing fast heuristic search code. In *Proc. 5th Annual Symposium on Combinatorial Search*, 25–32.

Edelkamp, S., and Helmert, M. 1999. Exhibiting knowledge in planning problems to minimize state encoding length. In *European Conference on Planning (ECP)*, 135–147.

Edelkamp, S. 2001. Planning with pattern databases. In *European Conference on Planning (ECP)*, 13–24.

Evans, J. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. BSDCan Conference*.

Evett, M.; Hendler, J.; Mahanti, A.; and Nau, D. 1995. PRA*: Massively parallel heuristic search. *Journal of Parallel and Distributed Computing* 25(2):133–143.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107.

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

Kishimoto, A.; Fukunaga, A. S.; and Botea, A. 2009. Scalable, parallel best-first search for optimal sequential planning. In *Proc. 19th International Conference on Automated Planning and Scheduling (ICAPS)*, 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.

Kobayashi, Y.; Kishimoto, A.; and Watanabe, O. 2011. Evaluations of Hash Distributed A* in optimal sequence alignment. In *Proc. 22nd International Joint Conference on Artificial Intelligence (IJCAI)*, 584–590.

Korf, R. E., and Felner, A. 2002. Disjoint pattern database heuristics. *Artificial Intelligence* 134(1):9–22.

Korf, R. E., and Schultze, P. 2005. Large-scale parallel breadth-first search. In *Proc. 20th National Conference on Artificial Intelligence (AAAI)*, 1380–1385.

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

Romein, J. W.; Plaat, A.; Bal, H. E.; and Schaeffer, J. 1999. Transposition table driven work scheduling in distributed search. In *Proc. 16th National Conference on Artificial Intelligence (AAAI)*, 725–731.

Thompson, J. D.; Koehl, P.; Ripp, R.; and Poch, O. 2005. BAliBASE 3.0: Latest developments of the multiple sequence alignment benchmark. *Proteins: Structure, Function and Genetics (PROTEINS)* 61(1):127–136.

Zhou, R., and Hansen, E. A. 2006. Domain-independent structured duplicate detection. In *Proc. 21st National Conference on Artificial Intelligence (AAAI)*, 1082–1087.

Zhou, R., and Hansen, E. A. 2007. Parallel structured duplicate detection. In *Proc. 22nd AAAI Conference on Artificial Intelligence (AAAI)*, 1217–1223.

Zobrist, A. L. 1970. A new hashing method with application for game playing. *reprinted in International Computer Chess Association Journal (ICCA)* 13(2):69–73.