Proceedings of the Twenty-Sixth International Conference on
Automated Planning and Scheduling (ICAPS 2016)

Automated Creation of Efficient Work Distribution Functions for Parallel Best-First Search

Yuu Jinnai and Alex Fukunaga
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. We investigate domain-independent methods for automatically creating effective work distribution functions for HDA*. First, we propose a new method for generating abstract features for the recently proposed abstract Zobrist hashing method. Second, we propose a method for modifying Zobrist hashing such that selected operators are guaranteed to generate children that are mapped to the same process as the parent, ensuring that no communications overhead is incurred for such operators. Finally, we present an improvement to state-abstraction based HDA* which dynamically selects the abstraction graph size based on problem features. We evaluate these new work distribution methods for a domain-independent planner on a cluster with 48 cores and show that these methods result in significantly higher speedups than previous 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. Parallelization is one way to speed up domain-independent planning, and parallel planners are becoming increasingly more common, as evidenced by, e.g., the recent IPC-14 Multicore track. In addition, parallelization is an effective way to overcome memory limitations – while it may not be possible to optimally solve a problem due to limited memory on a single machine, the aggregate memory available in a cluster or cloud environment can allow problems that can not be optimally solved using 1 machine to be solved – it has been argued that this is perhaps more important than obtaining speedup (Fukunaga, Kishimoto, and Botea 2012). Thus, designing scalable, parallel search algorithms that make efficient use of resources poses an important challenge.

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 using the Zobrist hash function (1970), HDA* could achieve 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 PSDD (Zhou and Hansen 2007) and PBNF (Burns et al. 2010). Abstraction-based work distribution achieves low communication overhead, but at the cost of high search overhead. Abstract Zobrist hashing (AZH) (Jinnai and Fukunaga 2016) achieves both low search overhead and communication overhead by incorporating the strengths of both Zobrist hashing and abstraction. While the Zobrist hash value of a state is computed by applying an incremental hash function to the set of features of a state, AZH first applies a feature projection functions mapping features to abstract features, and the Zobrist hash value of the abstract features (as opposed to the raw features) is computed. This results in reduced communication overhead and effective load balance. On the 24-puzzle, 15-puzzle, and multiple sequence alignment problem, AZH with hand-crafted, domain-specific feature projection function was shown to significantly outperform previous methods on a multicore machine with up to 16 cores.

In addition, (Jinnai and Fukunaga 2016) proposed a method for automatically generating abstract feature projection functions for STRIPS planning problems, as a proof of concept that abstract features could be generated automatically (this paper refers to this method, described below in Section 2.6 as GreedyAFG). While AZHDA* using GreedyAFG slightly outperforms standard Zobrist hashing, there is much room for improvement, since there are many domains where the abstract features found by GreedyAFG completely fail to reduce communications overhead, and AZHDA* using GreedyAFG ends up behaving much like standard ZHDA*.

In this paper, we propose and evaluate three new, domain-independent methods for automatically generating work distribution functions. First, we propose fluency-dependent abstract feature generation (FluencyAFG), a new abstract fea-

ture generation method for AZHDA*. FluencyAFG seeks to filter out poor candidates for feature abstraction according to the feature's "fluency", which indicates how often its value changes in state space. Second, we propose operator-based Zobrist hashing, a method for setting the bitstrings used to compute Zobrist hash values for ensuring that the successors of some selected state s are assigned the same Zobrist hash value as s. Third, we propose a small improvement to AHDA* (Burns et al. 2010) which dynamically adjusts the abstract graph size based on problem characteristics. We evaluate these approaches on domain-independent planning on a cluster of 6 machines with 48 total cores, as well as a shared-memory multicore machine.

The rest of this paper is structured as follows. First, Section 2 reviews HDA*, abstraction, and abstract Zobrist hashing. We then propose fluency-dependent AFG (Section 3), operator-based Zobrist hashing (Section 4), and dynamic abstraction-based HDA* (Section 5). Section 6 presents an experimental evaluation of these methods. Finally, Section 7 concludes with a discussion and directions for future work.

2 Background

2.1 Hash Distributed A*

Hash Distributed A* (HDA*) (Kishimoto, Fukunaga, and Botea 2013) is a parallel A* algorithm which incorporates the idea of hash based distribution of PRA* (Evett et al. 1995) and asynchronous communication of 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 nodes c, compute hash value $H(c)$, and send c to the thread that owns $H(c)$.

2.2 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* 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 (e.g., network communications), as well as access to data structure 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.

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

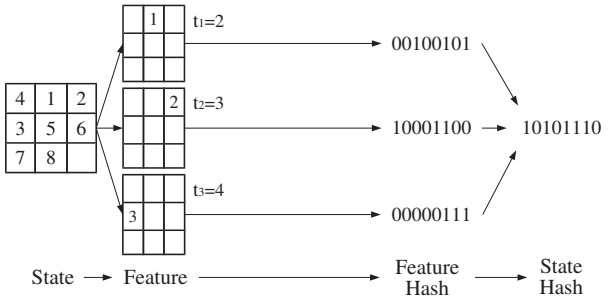
In parallel search, SO can arise due to inefficient load balance. We define load balance as $LB := \frac{\text{Max \# nodes assigned to a thread}}{\text{Avg. \# nodes assigned to a thread}}$. If load balance is poor, a thread which is assigned more nodes than others will become a bottleneck – other threads spend their time expanding less promising nodes, resulting in search overhead.

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.

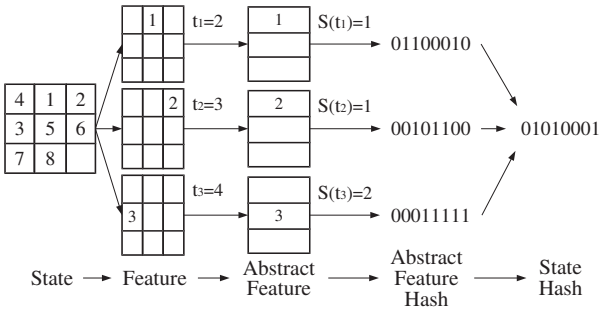
2.3 Zobrist Hashing and ZHDA*

Since the work distribution in HDA* is solely dependent on a global hash, 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 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.

$$Z(s) := R[x_0] \; xor \; R[x_1] \; xor \; \cdots \; xor \; R[x_n] \qquad (1)$$



(a) Zobrist



(b) Abstract Zobrist Hashing

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 $R[t_i]$).

Zobrist hashing seeks to distribute nodes uniformly among all threads, without any consideration of the neigh-

borhood 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 threads 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.

It was shown that CO of HDA* with Zobrist hashing (ZHDA*) increases to 92% when running on 16 threads on the 24-puzzle (Jinnai and Fukunaga 2016). Thus, efficiency of ZHDA* decreases as the # of threads increases.

2.4 Abstraction and AHDA*

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. Abstraction is generally constructed by ignoring subset of features. It has been shown that abstraction has roughly 2-4 times higher search overhead compared to Zobrist hashing on the 24-puzzle (Jinnai and Fukunaga 2016).

2.5 Abstract Zobrist Hashing and AZHDA*

Abstract Zobrist hashing (AZH) (Jinnai and Fukunaga 2016) is a hybrid hashing strategy which incorporates 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, 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.

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 features in the state, rather than a subset of features.

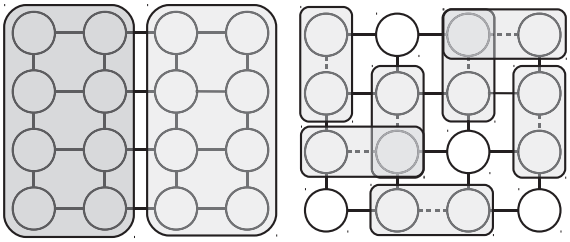
2.6 Automatic Generation of Feature Projection Functions

The feature projection function plays a critical role in determining the performance of AZH, because AZH relies on the feature projection in order to reduce communications overhead. The previous work on AZH focused on hand-crafted feature projection functions. In this section we investigate domain-independent methods for completely automatically constructing a feature projection function. We first review GreedyAFG, a method which was proposed in (Jinnai and Fukunaga 2016), and then propose a new method that address the weakness of GreedyAFG.

A state in STRIPS planning is a set of propositions, and the obvious way to implement Zobrist hashing is to use STRIPS propositional variables directly as features. In the case of a planner which internally uses a SAS+ representation, these features can be trivially recovered from the SAS+ variable assignment. For example, if some variable A has a value of v, the propositional variable $is(A, v)$ is true, and $is(A, v)$ is false if $A \neq v$. This implementation of Zobrist hashing was used in (Kishimoto, Fukunaga, and Botea 2013), as well as our implementation of ZHDA*. Thus, below, the raw features used for STRIPS planning are these propositional features, and the abstract features correspond to groups of propositional features.

Greedy Abstract Feature Generation and GAZHDA* *Greedy abstract feature generation* (GreedyAFG) is a simple, domain-independent abstract feature generation method, which partitions each feature into two abstract features (Jinnai and Fukunaga 2016). We first 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, e.g., the values of a SAS+ multi-valued variable (Bäckström and Nebel 1995). We used SAS+ variable values as atom groups. Each atom group G 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) greedily add to S_1 the unassigned node which shares the most edges with nodes in S_1; (3) while $|S_1| < |G|/2$ repeat step (2); (4) assign all unassigned nodes to S_2.

Abstract Zobrist hashing using abstract features generated by GreedyAFG has been shown to perform slightly better than standard Zobrist hashing (Jinnai and Fukunaga 2016). Figure 2a shows an example of the abstract features built by GreedyAFG for a grid problem.



(a) Abstract Zobrist hashing (b) Operator-based Zobrist hashing

Figure 2: Comparison of abstract Zobrist hashing vs. operator-based Zobrist hashing in a grid domain. Each node in the graph corresponds to a variable representing "a location of the robot". GreedyAFG builds abstract feature by dividing the graph into two abstract features. In contrast, operator-based Zobrist hashing, tries to assign hash values so that key actions does not incur CO.

3 Fluency-Dependent Abstract Feature Generation (FAZHDA*)

AZHDA* using the abstract features found by GreedyAFG fails to reduce CO when there are variables whose values are frequently changed by actions. Consider the standard blocks domain. Figure 3 shows the transition graph for this domain. The SAS+ variable v_0 represents the state of the robot hand, so the possible values are handempty and not-handempty. The atom group (mutex group) handempty and not-handempty will be partitioned into two singleton sets by GreedyAFG, resulting in two abstract features, one representing handempty and another representing not-handempty, and these two abstract features are assigned different hash values. However, in blocksworld, *all* actions (pick-up, put-down, stack, unstack) change the value of v_0 from handempty to not-handempty, or vice versa (equivalently, in a propositional representation, the ishandempty proposition changes is flipped by every single action). This means that for every state s in the search space, if s has the abstract feature for handempty, all the successor states of s will not have the handempty abstract feature and have the not-handempty abstract feature instead, and if s has the not-handempty abstract feature, than all successors of s will have the handempty abstract feature. This makes it highly likely that for every child c, $AZH(c) \neq AZH(s)$, which in turn means that the children will be assigned to different nodes than s, resulting in high communications overhead, i.e, in domains with such variables, the behavior of AZHDA* using GreedyAFG will resemble HDA* using standard Zobrist hashing. For example, Table 1 shows that GreedyAFG does not reduce CO compared to Zobrist hashing does not reduce CO at all on blocks.

To overcome this issue, we present a new feature projection function, *fluency-dependent abstract feature generation (FluencyAFG)*, which is based on the notion of a variable's *fluency*. We define the *fluency* of a variable to be the number of ground actions which change the value of the variable divided by the total number of ground actions in the problem. Therefore $fluency(v) \in [0, 1]$, and the larger $fluency(v)$ is, the more frequently the value of v changes. For example, $fluency(v_0)$ in the blocksworld example is 1.0. Variables with high fluency are common in a wide range of domains. For example, in domains modelling an agent which moves around in an environment (e.g. robot domains, logistics related domains), where the multivalued variable that represent the agent's location. In such domains, variables which represent the state (including location) of the agent frequently changes whereas the variables of environment seldom changes. In this case, ignoring agent-related (high-fluency) variables and only taking environment (low-fluency) variables into account is an effective way to builds up efficient abstract features for AZH.

FluencyAFG implements this policy, by applying a filter based on fluency. We experimented with various specific fluency-based filtering criterion. The current implementation of FluencyAFG first computes $fluency(v)$ for all values, and ignores variables whose fluency is in the top 30% of the variables. Then, GreedyAFG is applied to the remaining variables. As shown in Section 6, FluencyAFG using this filtering criterion is quite successful in reducing CO and increasing speedup compared to GreedyAFG.

While FluencyAFG is similar to the construction of Structured Duplicated Detection (SDD) (Zhou and Hansen 2006; 2007), in the sense that both methods generate atom groups for a factored representation of abstract states, there are two significant differences between FluencyAFG and SDD. First, FluencyAFG (or AZH in general) uses abstract features to generate abstract states while SDD uses raw features. Second, FluencyAFG applies fluency-based filtering criteria to choose atom groups to include in the abstract state, while SDD tries to minimize bounded outdegree of the abstract state space graph.

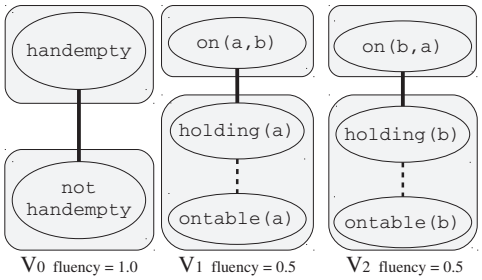


Figure 3: Greedy abstract feature generation (GreedyAFG) applied to blocksworld domain. The hash value for a state s is given by $H(v_0)\ xor\ H(v_1)\ xor\ H(v_2)$. Grey squares are abstract features generated by GreedyAFG, so all propositions in the same square have same hash value (e.g. H(holding(a)) = H(ontable(a))). $fluency(v_0) = 1.0$ since all actions in blocks world domain change its value. In this case, any abstract features based on the other variables are rendered useless, as all actions change v_0 and thus change hash value for the state. In this example, Fluency-dependent AFG will filter v_0 before calling GreedyAFG to compute abstract features based on the remaining variables.

4 Operator-Based Zobrist Hashing (OZHDA*)

Although FluencyAFG significantly improves upon GreedyAFG by avoiding creating abstract features for some variables that high fluency as explained above, the basic idea behind FluencyAFG is to *avoid* creating harmful abstract features that induce large communications overheads. While successful, avoiding harmful abstract features does not address the problem of *creating* useful abstract features that minimize overhead. In AZHDA*, communications overhead is minimized during search state generation when a generated state s is hashed to the same thread as its parent p, which requires that all features of s belong to the same abstract features as the corresponding features of p. If even one variable (feature) in n is projected to a different abstract feature than the corresponding feature in p, the hash value of p will almost certainly be different than that of n, and states with different hash values are likely to be assigned to different threads, resulting in search overhead.[1] A fundamental challenge with Abstract Zobrist hashing is the difficulty of evaluating the utility of an abstract feature. FluencyAFG sidesteps this issue by analyzing the raw feature and eliminating features that could lead bad abstract features.

In contrast, *Operator-based Zobrist hashing* (OZH) explicitly seeks to construct hash functions that minimize communications overhead. This is possible by focusing on actions rather than features, and directly generating Zobrist hash bitstrings in such a way that make it more likely that children of state s have the same hash value as s. Assume that all add effects and delete effects of action a are applied to state s in order to generate state s'. Then,

$$H(s') = H(s)\ xor\ H(a) \tag{2}$$

where $H(s)$, $H(s')$ are Zobrist hash values of state s, s', and $H(a)$ is a Zobrist hash value of action a, which is computed by xor'ing all hash values of propositions in its add and delete effects. For example, action put-down(b) in blocksworld has add effects {clear(b), handempty, ontable(b)} and delete effect {holding(b)}. Therefore $H(a) = H(\text{clear(b)})\ xor\ H(\text{handempty})\ xor\ H(\text{ontable(b)})\ xor\ H(\text{holding(b)})$. From equation 2, if $H(a) = 0$ then $H(s') = H(s)$. By assigning hash values to propositions so that $H(a) = 0$, the successors of s will have the same hash value as s whenever action a is applied, so communications overhead is *never* incurred by applying action a.

Algorithm 1 shows OZH. Recall that Zobrist hashing uses an array of bitstrings, R, where $R[p]$ is a random bit string for each proposition p (Equation 1). First, OZH initializes the array of bitstrings R with random values (if we stopped at this point and simply used R, this would be standard Zobrist hashing). For each action a, OZH tries to build a abstract feature by rewriting the values in the R so that $H(a)$ becomes 0. Values are only revised if $flag[p] = false$, indicating that it has not yet been processed by OZH. If it succeeds in setting $H(a) = 0$, then set $flag[p]$ for all its effect propositions so that the abstract feature would not be disrupted by later iterations.

[1] In HDA* the owner of a state is computed as *thread(s) = hash-value(s)* mod *numthreads* , so it is possible that states with different hash values are assigned to the same thread. Also, while extremely unlikely, it is theoretically possible that s and p may have the same hash value even if they have different abstract features due to the randomized nature of Zobrist hashing.

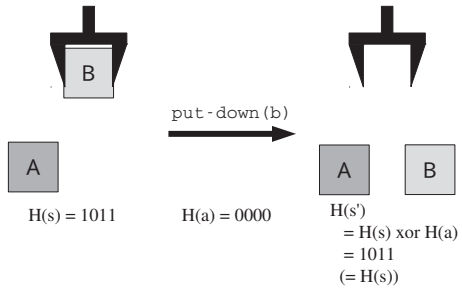


Figure 4: Operator-based Zobrist hashing (OZH): The Zobrist hash value $H(s')$ can be calculated by incrementally xor'ing the hash value of its parent state $H(s)$ and the hash value of the action a. If $H(a) = 0$, then $H(s) = H(s')$, thus no communication overhead is incurred when executing action a. OZH seeks to set the bitstrings $R[p]$ used in Equation 1 such that $H(a) = 0$.

The order in which Algorithm 1 iterates through the actions can influence the performance of OZH. Based on preliminary experiments to tune the action ordering, we attempt to process actions with fewer preconditions first, based on the assumption that edges representing actions with fewer preconditions appears more often in the search graph. Improved action orderings is a direction for future work.

Algorithm 1 Operator-based Zobrist hashing (OZH)

```
let P be the set of all propositions (features)
    R[p]is initialized as a random bitstring∀p ∈ P
let flags[p] ← false, ∀p ∈ P
for a in Actions do
    effects ← a.adds ∪ a.deletes
    for p in effects do
        if flags[p] = false then
            h ← 0
            for p' in effects \ p do
                h ← h xor R[p']
            end for
            R[p] ← h
            for p in effects do
                flags[p] ← true
            end for
            break
        end if
    end for
end for
```

5 Dynamic AHDA* (DAHDA*)

This section presents an improvement to AHDA* (Burns et al. 2010), which is based on the abstraction strategy originally used in PSDD (Zhou and Hansen 2007). In our exper-

iments, we used AHDA* as one of the baselines for evaluating our new AZHDA* strategies. The baseline implementation of AHDA* is based on the greedy abstraction algorithm described in (Zhou and Hansen 2006), and selects a subset of atom groups. 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}. PSDD requires a N_{max} to be derived from the size of the available RAM. We found that AHDA* with a static N_{max} threshold as in PSDD performed poorly for a benchmark set with varying difficulty because a fixed size abstract graph results in very poor load balance. While poor load balance can lead to low efficiency and poor performance, a bad choice for N_{max} can be catastrophic when the system has a relatively small amount of RAM per core, as poor load balance causes concentrated memory usage in the overloaded processors, resulting in early memory exhaustion (i.e., AHDA* crashes because a thread/process which is allocated a large number of states exhausts its local heap). The AHDA* results in Table 1 are for a 48-core cluster, 2GB/core, and uses $N_{max} = 10000$ nodes (we tried $10^2, 10^3, 10^4, 10^5, 10^6$) and chose 10000 because it performed best). Note that AHDA* fails on Blocks10-2 and Gripper8 due to memory exhaustion issue caused by extremely poor load balance. In our preliminary experiment, all other values of N_{max} also result in failure on at least 1 problem). Although the total amount of RAM in current systems is growing, the amount of RAM per core has remained relatively small because the number of cores has also been increasing (and is expected to continue increasing). Thus, this is a significant issue with the straightforward implementation of AHDA* which uses a static N_{max}.

To avoid this problem, N_{max} must be set dynamically according to the size of the state space for each instance. Thus, we implemented Dynamic AHDA* (DAHDA*), which dynamically set the size of the abstract graph according to the number of atom groups (the state space size is exponential in the # of atom groups). We set the threshold of the total number of features in the atom groups to be 30% of the total number of features in the problem instance (we tested 10%, 30%, 50%, and 70% and found that 30% performed best). Note that the threshold is relative to the number of features, not the state space size as in AHDA*, which is exponential in the # features. Therefore, DAHDA* tries to take into account of certain amount of features, whereas AHDA*/base sometimes use only a fraction of features.

6 Experiments

We evaluated the performance of the following HDA* variants on domain-independent planning.

- FAZHDA* : AZHDA* using Fluency-dependent abstract feature generation (Sec. 3)
- OZHDA*: HDA* with Operator-based Zobrist hashing (Sec. 4)
- DAHDA*: HDA* using abstraction based work distribution with dynamic threshold (Sec. 5)
- AHDA*/base (Burns et al. 2010): HDA* using abstraction based work distribution – baseline implementation with fixed N_{max} threshold of 10000 nodes (chosen because it was best among $10^2, 10^3, 10^4, 10^5, 10^6$ nodes)
- GAZHDA* (Jinnai and Fukunaga 2016): AZHDA* using greedy abstract feature generation (see Sec. 2.6)
- ZHDA*: HDA* using Zobrist hashing (Kishimoto, Fukunaga, and Botea 2013)[2]

We implemented these HDA* variants on Fast Downward (Helmert 2006) (version of February, 2014) using merge&shrink (LFPA) heuristics (Helmert, Haslum, and Hoffmann 2007) with the abstraction size set to 1000. We used the merge&shrink heuristic because of its fast node generation rate, which makes efficient parallelization challenging. As benchmark problems, we use classical planning instances from past IPC benchmarks. We selected the hardest instances which were solvable by A* with RAM on a single processor. We implemented inter-process communication using asynchronous buffering communication (MPI_Bsend and MPI_Iprobe) on MPICH 3 (Gropp et al. 1996). Our code uses Jemalloc memory allocator (Evans 2006). We ran experiments on a cluster with total 48 cores.

We ran our experiments on a cluster of 6 nodes, where each node has 8 core Intel Xeon E5410 2.33 GHz with 6144 KBshared L2 cache and 16 GB memory. Nodes are interconnected with 1000 Mbps Ethernet. For a cluster, we packed 100 states to reduce the number of messages (Romein et al. 1999). We ran 10 trials for each configuration.

Table 1 shows the speedups (time for 1 processes / time for 48 processes), communications overhead (CO), and search overhead (SO). We included the time for initializing abstract features (FluencyAFG, GreedyAFG), operator-based bitstring tables (OZHDA*), and state abstraction (DAHDA*, AHDA*), but none of these initializations took more than 1 second on any of the runs.

We excluded the time for initializing the abstraction table for the merge&shrink heuristic. Overall, we observed:

- FAZHDA* achieved the highest overall speedup.
- OZHDA* outperformed FAZHDA* in some instances where FluencyAFG fails to find feature abstractions that achieve low communications overhead.
- Both of the new ZHDA* variants (FAZHDA* and OZHDA*) significantly outperformed the baseline strategies.

[2]Although Kishimoto et al (2013) include results for an MPI-based implementation HDA* using the merge-and-shrink heuristic on a "commodity cluster" with a very similar CPU, the results are not directly comparable due to several factors. First their "commodity cluster" used 2x1Gbit bonded Ethernet whereas our cluster uses 1x1Gbit Ethernet, which means that communication costs are significantly higher on our cluster. Second, their code was based on a substantially older version of Fast Downward (from 2009) Third, they used an older version MPICH. Finally, our implementations are completely independent so there are likely other implementation differences. For example, our ZHDA* implementation solves the `Trucks5` instance in 51 seconds on 8 cores, while Kishimoto et al's implementation of ZHDA* on 8 processes on the Xeon L5410 solved `Trucks5` in 91 seconds (p. 228, Table 2).

- DAHDA* had the lowest CO among HDA* variants, but it had significantly higher search overhead than the other HDA* variants. DAHDA* significantly outperformed AHDA*/base (AHDA*/base failed on Blocks-10-2 and Gripper8 because poor load balance results in an excessive concentration of states being sent to some process, which resulted in memory exhaustion).

The effect of the number of cores on speedup Figure 5 shows the speedup of the algorithms as the number of cores increased from 8 to 48. FAZHDA* outperformed consistently outperformed the other methods. The performance gap between the better methods (FAZHDA*, OZHDA*, DAHDA*) and the baseline ZHDA* increases with the number of the cores. This is because as the number of cores increases, communications overheads increases with the number of cores, and our new work distribution methods successfully mitigates communications overhead.

The relationship between CO, SO, and speedup Figure 6a shows the relationship between communications overhead and speedup, and Figure 6b shows the relationship between search overhead and speedup. The results indicate negative correlation between communications overhead and speedup, and a strongly negative correlation between search overhead and speedup. There was no clear correlation between communications and search overheads.

Figure 7a and 7b shows the relationship between CO, SO, and speedup on a single multicore machine (Intel Xeon E5-2650 2.60 GHz) using 8 cores. Note that although there is no network communication within a single multicore machine, transferring a search state to another MPI process incurs overhead (e.g. manipulating message buffers), and Figure 7a shows the negative correlation between CO and speedup.

7 Conclusions and Future Work

We investigated new methods for automatically generating work distribution functions for parallel best-first search, and evaluated them on domain-independent classical planning. Our main contributions are: (1) Feature-dependent abstract feature generation, a new abstract feature generation method for abstract Zobrist hashing that avoids harmful feature projection functions by filtering candidates based on the notion of feature fluency. This significantly improves upon the greedy abstract feature generation method proposed in (Jinnai and Fukunaga 2016) as well as all other baseline methods. (2) Operator-based Zobrist hashing, a new method for generating Zobrist hash bitstrings that ensure that successors generated using selected actions are hashed to the same processor as their parent. (3) DAHDA*, an improvement to AHDA* (Burns et al. 2010) which uses a new, dynamic criterion for determining the abstract graph size according to the number of atom groups in the problem. (4) this is the first evaluation of AZHDA* on a cluster.

While fluency-dependent abstract feature generation resulted in the best performance overall, Operator-based Zobrist hashing, as well as DAHDA* perform well on some domains, so there is no clear, dominant work distribution strategy. Automatic selection among these methods based on a structural analysis of the problem is an interesting avenue for future work. AZHDA* and OZHDA* are orthogonal approaches, since AZHDA* seeks to cause state that share abstract features to be assigned the same hash value, while OZHDA* seeks to force successor states of some actions to be assigned to the same processor as the parent. Combining these approaches is a promising direction for future work.

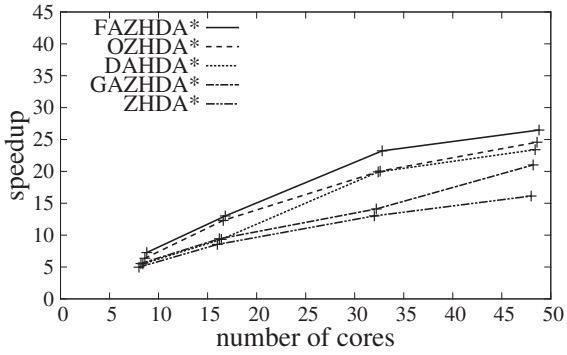


Figure 5: Speedup of HDA* variants (average over all instances in Table 1. Results are for 1 node (8 cores), 2 nodes (16 cores), 4 nodes (32 cores) and 6 nodes (48 cores).

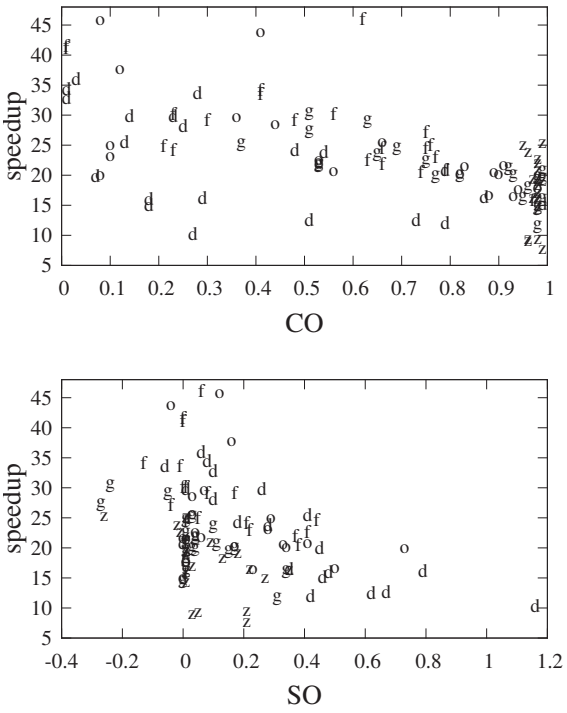


Figure 6: Relationship between Communications Overhead (CO), Search Overhead (SO), and Speedup (on 48 cores). Each letter represents the results of a single method on a problem instance. F: Fluency-dependent abstract feature generation (FAZHDA*), O: Operator-based Zobrist hashing (OZHDA*), G: Greedy abstract feature generation (GAZHDA*), D: Dynamic AHDA* (DAHDA*), Z: ZHDA*

Table 1: Performance of HDA* with different work distribution strategies on a 48-core (6-node) cluster. Values shown are mean of 10 runs (with standard error in parentheses). speedup: Speedup of wall time compared to A*. CO: communication overhead. SO: search overhead. The average speedups are weighted by the A* runtimes (because speedups on harder problems problems are more important than speedups on easier problems). The "expanded" column for A* is the number of nodes expanded by A*. The "applied" column for OZH is the fraction of actions for which OZH modified the Zobrist bitstring so that the successors generated using the action are assigned to the same process as the parent. The G_{abst} column for DAHDA* is the abstraction size limit determined by DAHDA* based on the number of atom groups in the problem.

Instance	A*		FAZHDA*			OZHDA*			
	time	expanded	speedup	CO	SO	speedup	CO	SO	applied
Blocks10-0	519.42	51781104	25.00(0.61)	0.76(0.00)	0.05(0.03)	20.81(0.00)	0.89(0.02)	0.33(0.00)	0.49
Blocks10-1	445.62	43176318	**27.19**(0.71)	0.75(0.00)	-0.04(0.03)	20.35(0.00)	0.90(0.01)	0.34(0.00)	0.49
Blocks10-2	228.16	21609943	**20.56**(1.43)	0.74(0.00)	0.38(0.12)	16.88(0.00)	0.88(0.02)	0.50(0.00)	0.49
Elevators08-5	182.28	9654685	21.96(0.28)	0.66(0.00)	0.37(0.02)	20.21(0.49)	0.08(0.00)	0.73(0.02)	0.16
Elevators08-6	507.98	18632725	**46.10**(2.00)	0.62(0.00)	0.06(0.04)	44.04(0.68)	0.41(0.00)	-0.04(0.01)	0.12
Gripper8	483.87	50068804	**29.17**(0.10)	0.48(0.00)	0.17(0.00)	20.48(0.10)	0.82(0.00)	0.17(0.00)	0.50
Logistics00-7-0	136.06	10881373	22.64(0.78)	0.63(0.00)	0.41(0.02)	**37.91**(0.97)	0.12(0.00)	0.16(0.01)	0.39
Logistics00-8-0	196.78	14716135	24.63(0.72)	0.66(0.00)	0.44(0.03)	**45.97**(1.79)	0.08(0.00)	0.12(0.01)	0.41
Miconic-10-0	269.79	17488571	**41.58**(0.30)	0.01(0.00)	-0.00(0.01)	21.99(0.18)	0.53(0.00)	0.06(0.00)	0.10
Miconic-10-1	276.13	17882561	**41.16**(0.31)	0.01(0.00)	0.00(0.00)	22.80(0.30)	0.53(0.00)	0.04(0.01)	0.10
Nomprime5	290.17	3982810	20.83(0.15)	0.79(0.00)	0.01(0.00)	17.85(0.21)	0.94(0.00)	0.01(0.00)	0.06
Openstacks11-11	181.11	11995225	**34.17**(0.35)	0.41(0.00)	-0.13(0.00)	23.33(0.35)	0.10(0.00)	0.28(0.00)	0.09
Openstacks11-13	134.78	8193065	**33.59**(0.44)	0.41(0.00)	-0.01(0.07)	25.20(0.35)	0.10(0.00)	0.29(0.00)	0.09
Parcprinter11-7	184.27	8639813	24.94(0.22)	0.21(0.00)	0.02(0.00)	28.75(0.62)	0.44(0.00)	0.03(0.00)	0.54
PipesNoTank10	147.79	3026313	15.89(0.13)	0.97(0.00)	0.01(0.00)	**18.03**(0.43)	0.98(0.00)	0.01(0.00)	0.10
PsrSmall49	374.90	32622577	**30.21**(0.10)	0.23(0.00)	0.01(0.00)	20.97(0.67)	0.56(0.00)	0.41(0.03)	0.27
Scanalyzer08-6	173.31	9328498	29.23(0.75)	0.30(0.00)	0.08(0.00)	25.79(0.92)	0.66(0.00)	0.03(0.01)	0.05
Sokoban08-15	176.69	21598353	**23.05**(0.71)	0.77(0.00)	0.22(0.00)	16.71(0.71)	0.93(0.00)	0.23(0.00)	0.39
Trucks5	208.22	10158856	24.24(0.08)	0.23(0.00)	0.21(0.01)	**29.84**(1.00)	0.36(0.00)	0.07(0.02)	0.03
Woodwork11-3	136.33	3110344	**30.17**(0.22)	0.56(0.00)	0.00(0.00)	21.82(0.16)	0.91(0.00)	0.00(0.00)	0.11
Woodwork11-4	355.18	11999093	24.53(0.35)	0.75(0.00)	0.01(0.01)	21.69(0.12)	0.83(0.00)	0.01(0.00)	0.12
Average	279.29	18999384	**28.14**(0.45)	0.52(0.00)	0.11(0.02)	24.83(0.41)	0.57(0.00)	0.18(0.01)	0.68
Total wall time [sec]	5907.90	381758487	Time [sec]		**204.89**(5.59)	Time [sec]		245.55(5.84)	

Instance	DAHDA*				AHDA*/base			GAZHDA*			ZHDA*		
	speedup	CO	SO	$\|G_{abst}\|$	speedup	CO	SO	speedup	CO	SO	speedup	CO	SO
Blocks10-0	**33.69**(7.67)	0.28(0.00)	-0.06(0.16)	14641	18.28(0.25)	0.39(0.00)	0.06(0.01)	20.00(0.68)	0.99(0.00)	0.15(0.03)	21.10(1.16)	0.98(0.00)	0.09(0.06)
Blocks10-1	16.18(0.31)	0.29(0.00)	0.79(0.00)	14641	10.52(0.11)	0.41(0.00)	0.75(0.02)	21.22(0.79)	0.99(0.00)	0.11(0.04)	22.92(0.65)	0.98(0.00)	-0.01(0.03)
Blocks10-2	10.28(0.27)	0.27(0.00)	1.16(0.04)	14641	memory exhausted			16.64(0.23)	0.99(0.00)	0.34(0.02)	16.46(0.18)	0.97(0.00)	0.35(0.02)
Elevators08-5	12.02(0.67)	0.79(0.00)	0.42(0.05)	1500	10.67(0.30)	0.88(0.00)	0.34(0.02)	**24.13**(1.96)	0.65(0.00)	0.10(0.09)	22.17(0.25)	0.98(0.00)	0.01(0.01)
Elevators08-6	16.47(0.50)	0.87(0.00)	0.35(0.03)	73125	10.66(0.16)	0.87(0.00)	0.48(0.01)	29.61(0.54)	0.63(0.00)	-0.05(0.01)	24.05(0.77)	0.96(0.00)	-0.02(0.02)
Gripper8	24.26(0.95)	0.48(0.00)	0.18(0.00)	39366	memory exhausted			20.53(0.07)	0.82(0.00)	0.17(0.00)	19.28(0.09)	0.98(0.00)	0.18(0.00)
Logistics00-7-0	23.87(0.75)	0.54(0.00)	0.28(0.01)	2400	10.14(1.02)	0.62(0.00)	0.35(0.05)	21.70(0.30)	0.92(0.00)	0.01(0.01)	18.53(1.66)	0.98(0.00)	0.13(0.07)
Logistics00-8-0	12.65(0.82)	0.51(0.00)	0.67(0.07)	2400	13.59(0.28)	0.58(0.00)	0.34(0.01)	20.56(0.22)	0.93(0.00)	0.03(0.01)	16.61(2.06)	0.98(0.00)	0.22(0.11)
Miconic-10-0	32.83(0.11)	0.01(0.00)	0.10(0.00)	1024	23.33(0.10)	0.01(0.00)	0.13(0.00)	22.35(0.48)	0.53(0.00)	0.04(0.01)	9.25(0.66)	0.96(0.00)	0.03(0.01)
Miconic-10-1	34.43(0.10)	0.01(0.00)	0.08(0.00)	1024	22.64(0.24)	0.01(0.00)	0.14(0.01)	22.15(0.38)	0.53(0.00)	0.04(0.01)	9.58(0.63)	0.96(0.00)	0.05(0.01)
Nomprime5	**20.91**(0.29)	0.79(0.00)	0.00(0.00)	4194304	20.12(0.25)	0.43(0.00)	0.28(0.00)	16.70(0.12)	0.95(0.00)	0.01(0.00)	16.17(0.21)	0.98(0.00)	0.01(0.00)
Openstacks11-11	16.01(0.00)	0.18(0.00)	0.48(0.00)	262144	12.34(0.00)	0.24(0.00)	0.59(0.00)	27.84(0.42)	0.51(0.00)	-0.27(0.00)	25.55(1.09)	0.99(0.00)	-0.26(0.00)
Openstacks11-13	15.11(0.16)	0.18(0.00)	0.46(0.01)	1048576	15.74(0.25)	0.10(0.00)	0.24(0.01)	30.95(0.87)	0.51(0.00)	-0.24(0.01)	15.26(3.50)	0.98(0.00)	0.27(0.21)
Parcprinter11-7	**29.92**(0.37)	0.14(0.00)	0.01(0.00)	32768	4.62(0.01)	0.04(0.00)	0.15(0.00)	22.94(0.68)	0.75(0.00)	0.01(0.00)	17.20(0.31)	0.98(0.00)	0.03(0.00)
PipesNoTank10	15.24(0.06)	0.99(0.00)	0.00(0.00)	32768	18.51(0.21)	0.43(0.00)	0.15(0.00)	15.01(0.16)	0.98(0.00)	0.00(0.00)	14.55(0.10)	0.98(0.00)	0.01(0.00)
PsrSmall49	25.54(0.18)	0.13(0.00)	0.41(0.01)	65536	19.65(0.11)	0.12(0.00)	0.34(0.00)	25.13(0.30)	0.69(0.00)	0.01(0.00)	25.28(0.21)	0.99(0.00)	0.01(0.00)
Scanalyzer08-6	**36.04**(0.17)	0.03(0.00)	0.06(0.00)	16384	15.83(0.69)	0.26(0.00)	0.16(0.01)	20.36(0.94)	0.77(0.00)	0.04(0.01)	19.70(0.18)	0.98(0.00)	0.01(0.00)
Sokoban08-15	12.51(0.35)	0.73(0.00)	0.62(0.00)	2097152	1.03(0.01)	0.36(0.00)	0.55(0.00)	12.16(0.71)	0.98(0.00)	0.31(0.00)	9.64(0.35)	0.98(0.00)	0.21(0.00)
Trucks5	19.97(0.38)	0.07(0.00)	0.45(0.01)	3328	14.55(0.04)	0.14(0.00)	0.38(0.00)	25.78(0.14)	0.37(0.00)	0.03(0.00)	7.86(1.10)	0.99(0.00)	0.21(0.05)
Woodwork11-3	29.84(0.28)	0.23(0.00)	0.26(0.00)	1327104	25.10(0.13)	0.16(0.00)	0.42(0.00)	20.19(0.12)	0.98(0.00)	0.01(0.00)	19.65(0.21)	0.97(0.00)	0.01(0.00)
Woodwork11-4	**28.15**(0.57)	0.25(0.00)	0.10(0.01)	294912	18.42(0.45)	0.19(0.00)	0.30(0.00)	18.53(0.12)	0.96(0.00)	0.01(0.00)	19.22(0.14)	0.99(0.00)	0.01(0.00)
Average	22.19(0.65)	0.37(0.00)	0.33(0.02)		N/A	N/A	N/A	21.64(0.48)	0.78(0.00)	0.04(0.01)	17.62(0.93)	0.98(0.00)	0.07(0.03)
Total time [sec]	Time [sec]		279.79(10.54)		Time [sec]		N/A	Time [sec]		267.41(7.79)	Time [sec]		338.14(18.66)

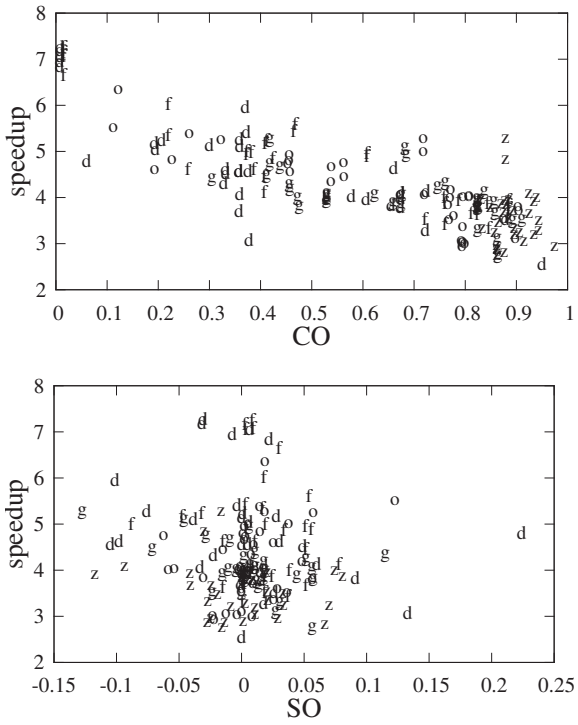


Figure 7: Relationship between Communications Overhead (CO), Search Overhead (SO), and Speedup (on 8 cores). Each letter represents the results of a single method on a problem instance. F: Fluency-dependent abstract feature generation (FAZHDA*), O: Operator-based Zobrist hashing (OZHDA*), G: Greedy abstract feature generation (GAZHDA*), D: Dynamic AHDA* (DAHDA*), Z: ZHDA*

References

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS+ planning. *Computational Intelligence* 11(4):625–655.

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

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. of the 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.

Fukunaga, A.; Kishimoto, A.; and Botea, A. 2012. Iterative resource allocation for memory intensive parallel search algorithms on clouds, grids, and shared clusters. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence (AAAI)*.

Gropp, W.; Lusk, E.; Doss, N.; and Skjellum, A. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing* 22(6):789–828.

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

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS)*, 176–183.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research (JAIR)* 26:191–246.

Jinnai, Y., and Fukunaga, A. 2016. Abstract Zobrist hashing: An efficient work distribution method for parallel best-first search. In *Proceedings of the Thritieth AAAI Conference on Artificial Intelligence (AAAI)*.

Kishimoto, A.; Fukunaga, A. S.; and Botea, A. 2009. Scalable, parallel best-first search for optimal sequential planning. In *Proceedings of the Nineteenth 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.

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

Zhou, R., and Hansen, E. A. 2006. Domain-Independent Structured Duplicate Detection. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI)*, 1082–1087.

Zhou, R., and Hansen, E. A. 2007. Parallel structured duplicate detection. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI)*, 1217–1223.

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