Evaluations of Hash Distributed A* in Optimal Sequence Alignment

Yoshikazu Kobayashi and **Akihiro Kishimoto** and **Osamu Watanabe**
Department of Mathematical and Computing Sciences
Graduate School of Information Science and Engineering
Tokyo Institute of Technology, Japan
{kobayas6, kishimoto, watanabe}@is.titech.ac.jp

Abstract

Hash Distributed A* (HDA*) is a parallel A* algorithm that is proven to be effective in optimal sequential planning with unit edge costs. HDA* leverages the Zobrist function to almost uniformly distribute and schedule work among processors.

This paper evaluates the performance of HDA* in optimal sequence alignment. We observe that with a large number of CPU cores HDA* suffers from an increase of search overhead caused by reexpansions of states in the closed list due to nonuniform edge costs in this domain. We therefore present a new work distribution strategy limiting processors to distribute work, thus increasing the possibility of detecting such duplicate search effort. We evaluate the performance of this approach on a cluster of multi-core machines and show that the approach scales well up to 384 CPU cores.

1 Introduction

The A* algorithm [Hart *et al.*, 1968] has been incorporated into many applications such as path-finding, planning, and sequence alignment in biology. However, when A* is required to solve large-scale problems, the main causes of performance bottlenecks are the CPU and memory requirements.

Incorporating parallelism into A* in distributed-memory environments is one way to improve both speed and solving ability by utilizing larger CPU and memory resources. This approach has become especially important to obtain significant speedups from the hardware because of limited speed improvement for each individual CPU core and ubiquitous availability of many machines connected in a network.

Hash Distributed A* (HDA*) is a simple but efficient parallel A* algorithm that has achieved a notable success in optimal unit-edge cost sequential planning [Kishimoto *et al.*, 2009]. HDA* can effectively use a vast amount of memory distributed over the network. HDA* scaled well up to 512 CPU cores and improved its solving ability in [Kishimoto *et al.*, 2010]. However, HDA* often suffered from performance deterioration with 1024 cores. Although Kishimoto *et al.* observed a significant search overhead, they did not analyze the reason why it occurred.

Parallelizing A* to search in a graph rather than a tree has been a difficult issue especially in distributed-memory environments, since the closed list used to detect duplicate detections cannot be shared without incurring parallel overheads. While HDA* is considered to have an advantage of efficiently detecting duplicate nodes by using the Zobrist function [Zobrist, 1970] (see the definition in Subsection 3.2), Kishimoto *et al.*'s performance evaluation was limited to search spaces with a uniform edge cost.

This paper contains an in-depth performance evaluation of HDA* in a graph with nonuniform edge costs to further elucidate its behavior. Multiple sequence alignment is an ideal testbed, since there are many paths to the same state with a variety of edge costs. Additionally, ideas invented in this domain are often transferred to other domains that have similar graph representations. The contributions of this paper are:

- A new insight that HDA* frequently fails in duplicate detections in multiple sequence alignment, resulting in drastically increasing search overhead.
- A new *hyperplane work distribution* (HWD) strategy with an application to multiple sequence alignment.
- Experimental results clearly demonstrating the superiority of HWD to the Zobrist function over 300 CPU cores.

2 Multiple Sequence Alignment

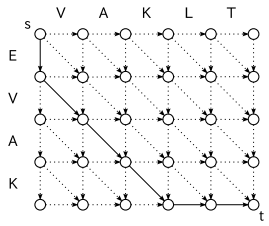

Figure 1: Grid representation of sequence alignment

Given several DNA or amino acid sequences as sets of strings, the purpose of the multiple sequence alignment (MSA) problem is to align these sequences with the best score by inserting gaps in each sequence.

MSA can be solved by an algorithm finding a lowest-cost path from corner to corner in an n-dimensional grid. Figure

1 shows an example with two sequences VAKLT and EVAK. One sequence is placed vertically from top to bottom and the other horizontally from left to right. Inserting a gap between two characters in a sequence corresponds to moving down or right from the current node and the diagonal move indicates matching characters in the sequences (see circles and dotted lines and representing nodes and edges, respectively). The bold line is the best path from s to t, which is turned into the best alignment (i.e., EVAK-- and -VAKLT).

Each edge cost is assigned by a scoring function based on the similarity of two characters and a penalty for a gap. Such a function is usually constructed by biologists. Note that the edge cost is nonuniform in this paper.

Since MSA is a path-planning problem, A* can be used to solve it. Although a more memory-efficient approach is available without maintaining the closed list [Korf *et al.*, 2005] (see Section 6 for detailed discussions), we focus on standard A* as a sequential baseline because HDA* must maintain the closed list for detecting duplicate nodes. Our main purpose is to get a more complete picture of the behavior of HDA*.

3 Background

3.1 Parallel Overheads

Achieving good parallel performance in search algorithms is notoriously difficult mainly due to the following *overheads*:

1. *Search overhead* is a useless part of a search tree built only by parallel search as compared to sequential search. It is approximated by the following value:

$$\text{SO} := \frac{\text{Total number of nodes expanded in parallel}}{\text{Number of nodes expanded by sequential search}} - 1.$$

2. *Synchronization overhead* is incurred when some processors must sit idle to wait for the others' computation. The existence of synchronization overhead is estimated by load balancing that refers to how evenly work is distributed among processors and that is defined as:

$$\text{LB} := \frac{\text{Maximal number of nodes expanded by a processor}}{\text{Average number of nodes expanded by each processor}}.$$

   The ideal case of $\text{LB} = 1$ hardly occurs in practice due to various work granularity assigned to processors.

3. *Communication overhead* is the communication delay caused by exchanging messages over the network to distribute work and share information among processors.

None of these overheads is independent. Since it is usually a difficult task to theoretically find the best case of minimizing the overheads, trade-offs are tuned empirically in practice.

One crucial factor affecting parallel performance is that there are many paths to the same node in many applications including sequential planning and MSA. Sequential A* can often omit searching duplicate nodes with the help of the closed list or some other duplication detection techniques (e.g. [Korf *et al.*, 2005; Zhou and Hansen, 2006]). However, detecting duplicate nodes efficiently becomes a difficult issue in parallel A* in distributed-memory environments, because the closed and open lists cannot be shared without incurring several overheads. In practice, balancing the workload without sharing these structures (e.g., [Karp and Zhang, 1988]) results in excessive search overhead incurred by duplicate search effort [Kishimoto *et al.*, 2010].

3.2 Hash Distributed A*

HDA* is based on ideas behind the hash-based work distribution strategy in PRA* [Evett *et al.*, 1995] and asynchronous communication in transposition-table driven work scheduling [Romein *et al.*, 2002]. Pioneering work that is very similar to HDA* appears in [Mahapatra and Dutt, 1997]. However, Mahapatra and Dutt incorporate not only a different hash function for work distribution but also a few complicated procedures such as the quality equalizing strategy [Dutt and Mahapatra, 1994] that possibly reassigns nodes to other processors and a procedure of partially expanding nodes.

In HDA* the closed/open list is partitioned over processors disjointly. This structure can be regarded as one large closed/open list where each processor owns a partition of the search space. The unique *owner* of each node is determined by a hash function. See [Kishimoto *et al.*, 2009] for detailed descriptions of HDA*.

In HDA* each processor P works similarly to A* except for an inclusion of the work distribution mechanism. P dequeues a node N with the lowest f-value from its own open list to generate N's children and saves N in P's closed list. Then, instead of enqueueing these generated children to P's local open list, P sends out them to their owners. P also periodically checks if nodes sent by others arrive at P. If so, P checks its local closed list to check whether they are duplicates or should be saved in its local open list. Let $g(N)$ be a g-value of node N. A node N is considered as a duplicate, if the node M in the closed list is identical to N and $g(M) \leq g(N)$ holds. HDA* continues these steps until an optimal solution is found[1].

HDA* has several merits. First, because HDA* sends and receives nodes asynchronously, it can overlap communications and computations. In other words, once each processor sends the work, it can immediately work on the next node saved in its local open list. Second, duplicate nodes are detected efficiently by their owners. The owners detect such duplicates by merely checking their local closed lists. Third, effective load balancing is achieved with an almost uniformly distributed hash function. Kishimoto *et al.* uses the Zobrist function [Zobrist, 1970], defined below in case of MSA.

In MSA, a node can be represented by a location in the n-dimensional grid. Let l_i be the length of the ith sequence and a node $\boldsymbol{x}$ be $(x_1, x_2, \cdots, x_n)$ where x_i is an integer ($0 \leq x_i \leq l_i$). Let R_i be a precomputed random table with $l_i + 1$ values. The Zobrist function $\text{Z}(\boldsymbol{x})$ is then defined as:

$$\text{Z}(\boldsymbol{x}) := R_1[x_1] \text{ xor } R_2[x_2] \text{ xor } \cdots \text{ xor } R_n[x_n].$$

One important enhancement to HDA* is to pack more than one node into one message when these nodes are sent to the same owner. The best pack size is determined empirically. This approach reduces communication overhead. On

[1] Sequential A* terminates immediately when selecting a goal node for expansion. However, a (first) solution is not always optimal in HDA*. HDA* must continue searching until *proving* that no better solution exists.

the other hand, it may delay sending out important nodes and increase search overhead (see Subsection 4.1).

4 Improving HDA* Based on Hyperplane Work Distribution

4.1 A Drawback of HDA*

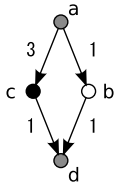

Figure 2: An example showing that HDA* may expand nodes in a non-optimal order and resulting in duplicate search effort

Once sequential A* with consistent heuristics saves nodes in the closed list, it never places these nodes back in the open list. On the other hand, in HDA* each processor selects the best node in its local open list and the selected node may not be globally best. Hence, while HDA* allocates node N via different paths to the same owner of N, it does not always detect duplicate search effort even with consistent heuristics. It may have to reopen N, because N may be received in an arbitrary order. This problem becomes more crucial in MSA where the edge cost is nonuniform, and can be explained with the help of Figure 2. Assume that P_1 owns nodes a and d. Let the owners of b and c be P_2 and P_3, respectively. P_1 must reopen d if nodes are expanded as follows:

1. P_1 expands a and generates b and c. While b is sent to P_2, c is sent to P_3.
2. P_3 expands c and sends d with $g(d) = 3 + 1 = 4$ to P_1.
3. P_1 expands d and d is saved in P's local closed list with $g(d) = 4$.
4. P_2 expands b and generates d. d is then sent to P_1.
5. P_1 receives d with $g(d) = 1 + 1 = 2$. Now P_1 must reopen d because $g(d) = 4$ in the closed list.

This is a less serious issue in unit-edge cost sequential planning, because of the uniform edge cost. Because the path length of $a \to c \to d$ is equal to that of $a \to b \to d$, d via $a \to b \to d$ is detected as a duplicate in this scenario.

4.2 Hyperplane Work Distribution

We present the hyperplane work distribution (HWD) strategy overcoming an issue of Zobrist-based work distribution. Our new function determining the owner of node $\boldsymbol{x}$ is defined as:

$$\mathrm{Plane}(\boldsymbol{x}, d) := \begin{cases} \lfloor \frac{1}{d} \sum x_i \rfloor & \left(d \in \{1, 2, 3, ...\}\right) \\ \frac{1}{d} \sum x_i + \left(\mathrm{Z}(\boldsymbol{x}) \bmod \frac{1}{d}\right) & \left(d \in \{\frac{1}{2}, \frac{1}{3}, ..., \frac{1}{p}\}\right) \end{cases}$$

where p is the number of processors and d is an empirically determined parameter indicating the *thickness* of the hyperplane. Let $P(\boldsymbol{x}) := \mathrm{Plane}(\boldsymbol{x}, d) \bmod p$. Processor P_i $(0 \le i < p)$ then owns $\boldsymbol{x}$ where $i = P(\boldsymbol{x})$. Once $\boldsymbol{x}$ is moved to P_i, P_i can check its local closed list to determine if $\boldsymbol{x}$ is a duplicate.

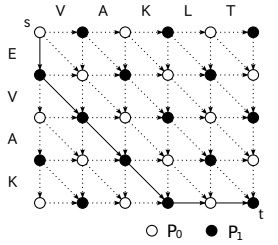

Figure 3: Example of hyperplane work distribution ($d = 1$)

Figure 3 illustrates an example of HWD in case of $d = 1$ and $p = 2$. White and black circles are assigned to processors P_0 and P_1, respectively. The child generated from node $\boldsymbol{x}$ by moving diagonally is allocated to the processor owning $\boldsymbol{x}$ and the remaining children must be sent to the other processor.

Let $\mathrm{Children}(\boldsymbol{x})$ be a set of children generated from node $\boldsymbol{x}$ owned by processor P_i and n be the number of sequences. $\mathrm{Plane}(x, d)$ satisfies the following remark for P_i:

Remark 1

$$\#\left(\bigcup_{x\,:\,P(x)=i} \{P(\boldsymbol{x'}) \mid \boldsymbol{x'} \in \mathrm{Children}(\boldsymbol{x})\}\right) \le \left\lfloor \frac{n}{d} + \max(1, \frac{1}{d}) \right\rfloor$$

Proof Let $\boldsymbol{x} = (x_1, x_2, \cdots, x_n)$ and $\boldsymbol{y} = (y_1, y_2, \cdots, y_n) \in \mathrm{Children}(\boldsymbol{x})$. Because $y_i \le x_i + 1$ $(1 \le i \le n)$ holds in MSA, the following calculation is derived:

$$0 \le \sum_{i=1}^{n} \frac{y_i}{d} - \sum_{i=1}^{n} \frac{x_i}{d} \le \sum_{i=1}^{n} \frac{x_i + 1}{d} - \sum_{i=1}^{n} \frac{x_i}{d} = \frac{n}{d}.$$

Now we consider the following cases:

Case $d \ge 1$**:** By counting the number of non-negative integers between 0 and n/d, we can prove that the number of processors owing $\boldsymbol{y}$ is limited to $\left\lfloor \frac{n}{d} + 1 \right\rfloor$.

Case $d < 1$**:** Let $\mathrm{ZZ}(\boldsymbol{y}) := \mathrm{Z}(\boldsymbol{y}) \bmod (1/d)$ where $1/d$ is a positive integer. It holds that $0 \le \mathrm{ZZ}(\boldsymbol{y}) \le 1/d - 1$. The following calculation is therefore easily derived:

$$0 \le \sum_{i=1}^{n} \frac{y_i}{d} + \mathrm{ZZ}(\boldsymbol{y}) - \sum_{i=1}^{n} \frac{x_i}{d} \le \frac{n}{d} + \mathrm{ZZ}(\boldsymbol{y}) \le \frac{n+1}{d} - 1.$$

Hence, the number of processors owing $\boldsymbol{y}$ is at most $\left\lfloor \frac{n+1}{d} \right\rfloor$.

■

Remark 1 indicates that HWD can bound the number q of processors to which each processor sends generated children at each step by $\lfloor n/d + \max(1, 1/d) \rfloor$, which is much smaller than $\min(p, 2^n - 1)$ if p and n are large. The bound of HWD is controlled by d. This property increases the chance of allocating generated children to the same processor that orders these children more reasonably, thus reducing unnecessary

node reexpansions. For example, assume that nodes b and c are assigned to P_2, and that b and c have the same heuristic value in Figure 2. P_2 can then expand b to generate d before expanding c because $g(b) < g(c)$. In this case, since d via $a \rightarrow b \rightarrow d$ reaches P_1 first, P_1 detects d via $a \rightarrow c \rightarrow d$ as a duplicate. Additionally, HWD helps to reduce the delay time for sending nodes to other processors. In a typical enhancement, each processor P packs k nodes as one message that is sent to processor Q to reduce the communication overhead; that is, until P generates k nodes that should be owned by Q, P does not send the message. If we assume that nodes are distributed uniformly among P's destination processors, the probability that each node generated by P is owned by Q is $1/q$, which is much larger in HWD than Zobrist-based work distribution. Hence HWD can accumulate k nodes much more quickly. One drawback is that HWD might achieve less effective load balancing than the Zobrist function. However, this can be controlled by d, although there is a trade-off between effective load balancing and scalability.

4.3 Other Implementation Details

In order to reduce redundant search effort, our sequential and parallel implementations incorporate the technique in [Ikeda and Imai, 1999]. They first run weighted A* (WA*) search to prove an upper-bound ub on the optimal solution and then performs sequential/parallel A* with ub. If $f(N) > ub$ holds for node N, A* immediately discards N. We set the weight of WA* to 1.02. The computational overhead incurred by WA* is much smaller than A* search (see the next section).

The number of packed nodes is set to 256, which returned the best results in our initial experiments in both HDA* + HWD and HDA* using the Zobrist function.

We use a typical consistent heuristic function based on pairwise sequence alignments (e.g., [Korf *et al.*, 2005]). Each edge cost is based on the well-known PAM250 matrix score[2] with a gap penalty of 8.

While selecting d for $\mathrm{Plane}(\boldsymbol{x}, d)$ impacts performance, it depends on many factors such as the number of processors p and the total lengths of sequences l. By performing a few initial experiments[3], we empirically define d as:

$$d := \begin{cases} \mathrm{round}(\frac{\lambda l}{\log p}) & (\frac{\lambda l}{\log p} \geq 1) \\ (\mathrm{round}(\frac{\log p}{\lambda l}))^{-1} & (\frac{\lambda l}{\log p} < 1) \end{cases}$$

where $\mathrm{round}(x)$ rounds off x and λ is a constant set to 0.003, which achieved reasonable speedups in our preliminary experiments.

5 Experiments

We selected test problems from BAliBASE3.0 [Thompson *et al.*, 2005], which is a popular database for protein alignments. Because easy problems are not very informative in an evaluation of parallel scaleup, we chose seven difficult problems that were solved by our sequential implementation between five minutes and one hour (see Table 1 for the runtime and node expansions), and four more difficult problems that required at least 12 cores to solve within one hour.

Table 1: Performance of sequential A* (n: number of sequences). Init. time indicates the execution time for the WA* search. Runtime shown in seconds excludes init. time.

name	n	node expansion	runtime	init. time (ratio)
BB12022	5	166,237,332	1900.66	0.27 (1.410×10^{-4})
BBS12023	5	251,495,502	3026.12	19.01 (6.243×10^{-3})
BBS11037	5	324,063,825	3399.74	47.72 (1.384×10^{-2})
BBS11026	7	13,097,629	312.05	6.28 (1.973×10^{-2})
BB12036	7	64,729,569	1849.95	1.85 (1.001×10^{-3})
BBS12010	7	65,595,366	2967.83	0.37 (1.232×10^{-4})
BBS12032	9	4,020,764	446.41	2.55 (5.681×10^{-3})

All experiments were performed on a cluster of multi-core machines where each compute node has two hexa core Xeon X5670 processors at 2.93GHz (12 CPU cores) with 54 GB memory interconnected by QDR Infiniband $\times 2$ (80 Gbps). We used at most 32 compute nodes, resulting in a total of 384 CPU cores. As in [Kishimoto *et al.*, 2009], when N CPU cores were used, N MPI processes were invoked and communicated by MPI operations even if memory can be shared on the same compute node.

The following programs were prepared and parallelized using MVAPICH2[4], an implementation of the MPI library:

- ZOBRIST: HDA* using the Zobrist function for work distribution as in [Kishimoto *et al.*, 2009].
- PLANE: HDA* using our hyperplane work distribution.

Table 2 shows the runtime and speedups of ZOBRIST and PLANE on 12-384 CPU cores. When the speedups were measured, the initialization time spent by WA* was not included. The thickness d of the hyperplane was automatically determined as explained in Subsection 4.3.

Both ZOBRIST and PLANE scale well up to 96 cores. For example, compared to one core, Zobrist achieved speedups in the range between 63 and 80 on 96 cores, while the speedups of PLANE ranged between 67 and 93[5]. However, two distinct tendencies were observed in ZOBRIST with 192 cores. For the four problems (BB12036, BBS12010, BBS12032, and BB12003), observed speedups started to deteriorate, while the runtime was still improved for the others. On 384 cores, compared to 192 cores, performance degradation was observed for most problems. Additionally, four problems that ZOBRIST solved with a small number of cores were unsolved with 384 cores within 600 seconds. On the other hand,

[2] `http://prowl.rockefeller.edu/aainfo/pam250.htm`

[3] More specifically, we tried functions such as ones linear to p, $\sqrt{p}$ and $\log p$. The former two did not perform well with large p since they returned large values.

[4] `http://mvapich.cse.ohio-state.edu/`

[5] These numbers were occasionally larger than theoretical speedups calculated from parallel overheads. We used the fixed size of the closed list constructed as a hash table per compute node for sequential and parallel versions to maximize the amount of available memory for both versions. By using a performance profiler, we observed that hash table access became slower if the table size was increased. We hypothesize that sequential search had a higher cache miss rate in accessing the table than parallel search, because it used only one CPU and resulted in a smaller amount of cache available for the hash table per compute node compared to parallel search.

Table 2: Speedup comparison (n: number of sequences, l: total length of n sequences, and p: number of CPU cores)

				runtime (speedup)					
name	n	l	hash func.	$p = 12$	$p = 24$	$p = 48$	$p = 96$	$p = 192$	$p = 384$
BB12022	5	1280	ZOBRIST	218.19 (8.71)	**106.07 (17.92)**	54.31 (34.99)	30.25 (62.84)	14.99 (126.82)	> 600.0
			PLANE	**207.04 (9.18)**	108.71 (17.48)	**51.32 (37.04)**	**23.77 (79.95)**	**12.17 (156.19)**	**6.75 (281.75)**
BBS12023	5	3481	ZOBRIST	300.44 (10.07)	146.74 (20.62)	75.06 (40.32)	39.21 (77.18)	20.40 (148.33)	> 600.0
			PLANE	**280.52 (10.79)**	**144.55 (20.94)**	**70.35 (43.01)**	**32.32 (93.63)**	**16.52 (183.22)**	**10.50 (288.34)**
BBS11037	5	1870	ZOBRIST	**311.67 (10.91)**	164.31 (20.69)	85.11 (39.94)	42.71 (79.59)	21.84 (155.67)	> 600.0
			PLANE	325.26 (10.45)	**157.65 (21.56)**	**80.09 (42.45)**	**38.04 (89.37)**	**19.12 (177.79)**	**10.54 (322.67)**
BBS11026	7	604	ZOBRIST	**29.57 (10.55)**	**14.69 (21.25)**	**7.74 (40.31)**	4.98 (62.72)	3.43 (91.05)	34.98 (8.92)
			PLANE	30.12 (10.36)	15.58 (20.03)	7.74 (40.29)	**4.28 (72.88)**	**2.45 (127.15)**	**1.86 (168.12)**
BB12036	7	1499	ZOBRIST	175.08 (10.57)	**86.26 (21.45)**	44.80 (41.29)	**23.46 (78.87)**	48.27 (38.33)	64.87 (28.52)
			PLANE	**169.91 (10.89)**	87.97 (21.03)	**44.09 (41.96)**	25.92 (71.36)	**15.88 (116.49)**	**15.03 (123.11)**
BBS12010	7	2221	ZOBRIST	288.49 (10.29)	140.09 (21.19)	69.30 (42.82)	41.02 (72.36)	87.26 (34.01)	79.76 (37.21)
			PLANE	**261.03 (11.37)**	**130.86 (22.68)**	**65.23 (45.49)**	**32.36 (91.72)**	**16.67 (178.06)**	**10.14 (292.56)**
BBS12032	9	586	ZOBRIST	**39.33 (11.35)**	**20.19 (22.11)**	**10.18 (43.83)**	**5.70 (78.35)**	15.76 (28.32)	9.53 (46.87)
			PLANE	40.99 (10.89)	21.03 (21.23)	10.59 (42.16)	6.70 (66.58)	**3.64 (122.58)**	**2.68 (166.75)**
BB12023	5	3593	ZOBRIST	694.53 (–)	297.81 (–)	140.07 (–)	73.73 (–)	39.78 (–)	> 600.0
			PLANE	**567.77 (–)**	**286.19 (–)**	**133.43 (–)**	**62.91 (–)**	**32.96 (–)**	**21.78 (–)**
BB12003	8	586	ZOBRIST	970.38 (–)	**478.98 (–)**	**237.87 (–)**	**123.00 (–)**	139.30 (–)	135.69 (–)
			PLANE	**956.58 (–)**	492.05 (–)	244.59 (–)	175.67 (–)	**66.00 (–)**	**42.31 (–)**
BBS12005	9	1815	ZOBRIST	631.41 (–)	311.99 (–)	157.94 (–)	82.36 (–)	51.70 (–)	99.00 (–)
			PLANE	**610.53 (–)**	**310.29 (–)**	**153.60 (–)**	**79.72 (–)**	**42.22 (–)**	**33.78 (–)**
BB12032	9	613	ZOBRIST	1709.26 (–)	**844.25 (–)**	**428.21 (–)**	264.09 (–)	157.44 (–)	162.67 (–)
			PLANE	**1688.52 (–)**	880.70 (–)	442.14 (–)	**221.28 (–)**	**117.66 (–)**	**74.97 (–)**

PLANE constantly improved the runtime even over 100 cores and resulted in achieving speedups that ranged between 123 and 323 on 384 cores.

Table 3 shows the total number of nodes reopened from the closed list by all the cores and search overhead on 96-384 cores. While sequential A* never reopens nodes in the closed list due to the consistency of our heuristic, reopening nodes occurred in parallel A*. There was a strong correlation between the number of nodes reopened by parallel search and search overhead. While there were both cases where ZOBRIST reopened fewer nodes than PLANE and vice versa on 96 cores, the superiority of PLANE to ZOBRIST was clearly observed in the cases of 192 and 384 cores. On 384 cores PLANE always reopened much fewer nodes than ZOBRIST (i.e., by a factor of one or two orders of magnitude). For example, in solving BBS11026 with 384 cores, ZOBRIST placed back 52,432,014 nodes from the closed list to the open list, while this number was 823,296 for PLANE. This resulted in a drastic difference in speedup values of the two methods (8.92-fold versus 168.12-fold in Table 2). In particular, for the problems unsolved by ZOBRIST, we observed that most nodes were uselessly reopened, which was the major culprit making ZOBRIST fail in solving the problems.

While results in [Kishimoto *et al.*, 2010] showed that HDA* scaled well up to 512 cores in unit-edge cost sequential planning, the scaling behavior of ZOBRIST was more modest. Assume that there are only two paths p_1 and p_2 that lead to the same node N and the lengths of p_1 and p_2 are identical. In unit-edge cost sequential planning, the local closed list of HDA* can always detect that N is a duplicate in any search order – i.e., N via p_1 is first saved in the closed list and then N via p_2 is received, or the other way around. However, in MSA, assume that the path cost of p_1 is larger than that of p_2 and N via p_1 is first saved in the closed list of N's owner. If N via p_2 is received, N must be reopened. In fact, about 90% of reopened nodes were caused by the case of which the lengths of two paths were the same but their path costs were different. This would be one reason why it was difficult to achieve similar scaling behaviors to [Kishimoto *et al.*, 2010] in MSA.

Table 4 shows speedups and load balancing when d for $\mathrm{Plane}(\boldsymbol{x}, d)$ was varied in solving BB12036. There was clearly a trade-off between the choice of d and the number of processors. Thicker planes ($1 \leq d \leq 2$) could be used to solve the problems with a relatively small number of cores. However, load balancing worsened drastically with large d with 384 cores. Note that our function for automatically setting d is not best in this case. For example, on 384 cores, while the best d was $\frac{1}{4}$, the function set d to 1. Improving the technique to automatically tune d remains as future work.

6 Other Related Work

This section refers to related work addressing issues on sequential and parallel A* variants when they are applied to domains that have many paths to the identical nodes.

Previous work on sequential algorithms tries to overcome the memory limitation. While Iterative Deepening A* [Korf, 1985] is a popular approach, its primary drawback is the time complexity in search spaces with multiple paths to the same state [Korf *et al.*, 2005]. At the price of a very small re-search overhead incurred by a divide-and-conquer approach, frontier search [Korf *et al.*, 2005] reduces the memory requirement of A* by storing only the open list. However, one drawback of frontier search is that it requires to mark used operators to avoid duplicate search in MSA. Since the number of possible operators is $2^n - 1$ with n sequences, the structure required to manage a set of used operators suffers from a large blowup. In [Korf *et al.*, 2005] n was set to at most five, while we varied n between five and nine.

Table 5 shows the ratio of the maximal amount of memory allocated as the open list of the A* version of frontier search to that for A*'s open and closed lists in solving test problems

Table 3: Number of nodes reopened and search overhead

		$p = 96$				$p = 192$				$p = 384$			
		ZOBRIST		PLANE		ZOBRIST		PLANE		ZOBRIST		PLANE	
name	n	reopen	SO	reopen	SO	reopen	SO	reopen	SO	reopen	SO	reopen	SO
BB12022	5	**26,931,867**	**0.23**	31,406,029	0.25	45,206,738	0.32	**34,649,396**	**0.28**	unsolved	–	**38,539,535**	**0.36**
BBS12023	5	37,192,810	**0.19**	**29,155,948**	0.21	55,657,404	0.29	**32,080,412**	**0.20**	unsolved	–	**36,633,254**	**0.27**
BBS11037	5	**37,750,926**	**0.12**	61,706,272	0.20	**45,495,779**	**0.14**	63,877,579	0.15	unsolved	–	**57,623,110**	**0.18**
BBS11026	7	1,389,909	0.14	**139,358**	**0.05**	3,010,496	0.26	**296,010**	**0.12**	52,432,014	4.11	**823,296**	**0.23**
BB12036	7	**791,046**	**0.02**	1,016,997	0.06	34,342,573	0.60	**1,544,408**	**0.07**	75,757,905	1.23	**3,479,174**	**0.11**
BBS12010	7	12,153,632	0.27	**1,265,006**	**0.10**	85,180,615	1.87	**1,507,513**	**0.18**	107,473,282	2.11	**2,237,737**	**0.46**
BBS12032	9	**57,683**	**0.02**	100,910	0.03	1,603,316	0.41	**115,733**	**0.04**	6,203,252	1.55	**243,561**	**0.07**
BB12023	5	**43,100,914**	–	75,639,061	–	**68,259,973**	–	73,777,509	–	unsolved	–	**66,966,386**	–
BB12003	8	**292,734**	–	8,779,841	–	20,701,650	–	**443,640**	–	40,359,186	–	**2,027,297**	–
BBS12005	9	240,489	–	**164,489**	–	884,404	–	**718,479**	–	24,526,060	–	**1,067,810**	–
BB12032	9	4,446,211	–	**288,224**	–	5,857,432	–	**725,350**	–	14,637,606	–	**3,063,362**	–

Table 4: Parallel performance with various d for PLANE (BB12036, p: number of cores)

	$p = 12$			$p = 24$			$p = 48$			$p = 96$			$p = 192$			$p = 384$		
d	runtime (speedup)	SO	LB	runtime (speedup)	SO	LB	runtime (speedup)	SO	LB	runtime (speedup)	SO	LB	runtime (speedup)	SO	LB	runtime (speedup)	SO	LB
Plane($\boldsymbol{x}, 1/4$)	182.22 (10.15)	0.04	1.04	91.66 (20.18)	0.01	1.02	45.47 (40.69)	0.04	1.04	**22.55 (82.05)**	0.03	1.05	**12.25 (151.02)**	0.06	1.06	**7.29 (253.69)**	0.11	1.14
Plane($\boldsymbol{x}, 1/2$)	177.03 (10.45)	0.04	1.05	89.95 (20.57)	0.04	1.05	44.53 (41.54)	0.01	1.04	22.86 (80.92)	0.06	1.06	13.12 (140.98)	0.06	1.16	8.64 (214.00)	0.09	1.37
Plane($\boldsymbol{x}, 1$)	**169.00 (10.95)**	0.01	1.02	**87.97 (21.03)**	0.03	1.06	**44.09 (41.96)**	0.03	1.04	25.92 (71.36)	0.06	1.16	15.88 (116.49)	0.07	1.37	15.03 (123.11)	0.11	2.39
Plane($\boldsymbol{x}, 2$)	169.91 (10.89)	0.02	1.05	88.74 (20.85)	0.03	1.04	50.35 (36.74)	0.05	1.17	31.39 (58.93)	0.07	1.37	29.17 (63.43)	0.11	2.35	29.02 (63.74)	0.15	4.43
Plane($\boldsymbol{x}, 4$)	179.69 (10.30)	0.03	1.03	102.54 (18.04)	0.05	1.15	62.67 (29.52)	0.07	1.36	55.34 (33.43)	0.09	2.34	55.13 (33.55)	0.13	4.52	48.79 (37.92)	0.07	8.26

Table 5: Comparison on memory efficiency between sequential A* and frontier A* (n: number of sequences)

name	n	Frontier A*/A*
BB12022	5	0.977
BBS12023	5	0.599
BBS11037	5	0.525
BBS11026	7	4.381
BB12036	7	3.637
BBS12010	7	4.148
BBS12032	9	21.176

shown in Table 1. Frontier A* achieved memory savings with $n = 5$ as shown in [Korf *et al.*, 2005]. However, due to an additional data structure for marking used operators, frontier A* suffered from much more memory consumption than standard A* if $n \geq 7$. This implies that another approach such as [Zhou and Hansen, 2003] would be necessary to overcome this issue, which is a possible extension to explore as future work.

In order to parallelize frontier search, one obstacle is how to detect duplicate nodes efficiently and correctly without the closed list. Correctness of used operators is based on the node ordering of sequential A* with consistent heuristic functions. HDA* expands a node that is locally best even with consistent heuristic functions. Therefore, the used operators cannot correctly detect redundant paths to the identical node if they are combined with HDA*.

The first attempt to parallelizing frontier search (PFA*-DDD) was presented in [Niewiadomski *et al.*, 2006] with delayed duplication detection (DDD) [Korf, 2003]. It dynamically splits the search space into *intervals* computed by sampling to try to evenly distribute the workload. However, two drawbacks of PFA*-DDD discussed in [Burns *et al.*, 2010] are an expensive workload distribution scheme and a large synchronization step incurred by sorting nodes and duplicate detection methods for their data structures. Additionally, PFA*-DDD must manage used operators, which may be an issue with a larger number of sequences aligned in MSA. Niewiadomski *et al.* used test problems with five sequences. On the other hand, in HDA* + HWD, each processor expands a node with the locally best f-value without exchanging any information on the globally best f-value. Additionally, it requires no synchronization step to achieve load balancing, because the workload is determined by $\mathrm{Plane}(\boldsymbol{x}, d)$.

Abstraction is used to partition the search space and efficiently detect duplicate nodes in [Burns *et al.*, 2010]. Several parallel algorithms including AHDA* and PBNF leverage abstract information to reduce the overhead caused by thread contentions in a shared-memory machine. Their abstraction has similarity to HWD in the sense that both methods exploit locality of work. However, their approach is currently experimented in shared-memory settings.

Mahapatra and Dutt present not only the idea behind HDA* called Global Hashing of Nodes but also Local Hashing of Nodes (LOHA) that partitions the search space to disjoint processor groups on the Traveling Salesperson Problem in [Mahapatra and Dutt, 1997]. HWD and LOHA are similar in the sense that both provide a way to limit the number of destination processors to which each processor sends work by using a hash function in a levelized graph. However, LOHA was designed to minimize the global communication costs on a Hypercube machine where these communication costs vary among pairs of processors. It first allocates *coarse-grained work* to its corresponding subcube that contains a set of processors, because of slow communications among subcubes. It then splits such work finely to the processors inside the subcube, where communications are faster. On the other hand, HWD directly splits *fine-grained work* to limited processors

and aims at reducing search overhead incurred by failures in duplicate detections on a state-of-the-art hardware architecture that is a very different and more established platform than Hypercube. Communication delays that were essential for LOHA to avoid are not an obstacle in our environment and our in-depth analysis demonstrates that successful duplicate detections play an important role in scaling well.

7 Conclusions and Future Work

By using MSA as a test domain, we evaluated the scaling behavior of HDA* that was originally adapted to optimal unit-edge cost sequential planning. Our experimental results demonstrated that HDA* suffered from the increased search overhead caused by failures in duplicate detections due to the nonuniform edge cost. We therefore developed the hyperplane work distribution strategy selecting a subset of processors to distribute work. This approach scaled well up to 384 cores, while HDA*'s scalability was limited to 96 cores.

There are a few possible extensions yet to explore. One direction is to combine the idea behind HDA* with frontier search. An obstacle is that HDA* relies on the closed list for duplicate detections, while frontier search does not manage it. Additionally, it is necessary to reduce the size of used operators to solve MSA problems with a large number of sequences. The other possibility is to apply the idea behind hyperplane work distribution to other domains, because the idea itself is so natural that it is not limited to MSA. Cost-optimal planning is a strong candidate in our current research, since a similar issue caused by multiple paths to the same state and nonuniform edge costs would arise in this domain.

Acknowledgments

We would like to thank Adi Botea and Alex Fukunaga for their beneficial feedback on this paper. This research is supported by the JSPS Global COE program "Computationism as a Foundation for the Sciences".

References

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

[Dutt and Mahapatra, 1994] S. Dutt and N. Mahapatra. Scalable load balancing strategies for parallel A* algorithms. *Journal of Parallel and Distributed Computing*, 22:488–505, 1994.

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

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

[Ikeda and Imai, 1999] T. Ikeda and H. Imai. Enhanced A* algorithms for multiple alignments: Optimal alignments for several sequences and k-opt approximate alignments for large cases. *Theoretical Computer Science*, 210(2):341–374, 1999.

[Karp and Zhang, 1988] R. Karp and Y. Zhang. A randomized parallel branch-and-bound procedure. In *Proceedings of the 20th ACM Symposium on Theory of Computing (STOC)*, pages 290–300, 1988.

[Kishimoto *et al.*, 2009] A. Kishimoto, A. Fukunaga, and A. Botea. Scalable, parallel best-first search for optimal sequential planning. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS-2009)*, pages 201–208, 2009.

[Kishimoto *et al.*, 2010] A. Kishimoto, A. Fukunaga, and A. Botea. On the scaling behavior of HDA*. In *Proceedings of the 3rd Symposium on Combinatorial Search (SoCS'2010)*, pages 61–62. AAAI Press, 2010.

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

[Korf, 1985] R. Korf. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[Korf, 2003] R. E. Korf. Delayed duplicate detection: extended abstract. In *Proceedings of IJCAI-03*, pages 1539–1541, 2003.

[Mahapatra and Dutt, 1997] N. Mahapatra and S. Dutt. Scalable global and local hashing strategies for duplicate pruning in parallel A* graph search. *IEEE Transactions on Parallel and Distributed Systems*, 8(7):738–756, 1997.

[Niewiadomski *et al.*, 2006] R. Niewiadomski, J. N. Amaral, and R. C. Holte. Sequential and parallel algorithms for frontier A* with delayed duplicate detection. In *Proceedings of AAAI-06*, pages 1039–1044, 2006.

[Romein *et al.*, 2002] J. W. Romein, H. E. Bal, J. Schaeffer, and A. Plaat. A performance analysis of transposition-table-driven work scheduling in distributed search. *IEEE Transactions on Parallel and Distributed Systems*, 13(5):447–459, 2002.

[Thompson *et al.*, 2005] J. D. Thompson, P. Koehl, R. Ripp, and O. Poch. BAliBASE 3.0: Latest developments of the multiple sequence alignment benchmark. *Proteins: Structure, Function, and Bioinformatics*, 61:127–136, 2005.

[Zhou and Hansen, 2003] R. Zhou and E. Hansen. Sparse-memory graph search. In *Proceedings of IJCAI'2003*, pages 1259–1266, 2003.

[Zhou and Hansen, 2006] R. Zhou and E. Hansen. Domain-independent structured duplicate detection. In *Proceedings of AAAI-06*, pages 1082–1087, 2006.

[Zobrist, 1970] A.L. Zobrist. A new hashing method with application for game playing. Technical Report 88, University of Wisconsin, 1970.