On the Scaling Behavior of HDA*

Akihiro Kishimoto
Tokyo Institute of Technology
and JST PRESTO

Alex Fukunaga
University of Tokyo

Adi Botea[*]
NICTA and
The Australian National University

1 Introduction

Parallel search on parallel clusters has the potential to provide the memory and the CPU resources needed to solve challenging search problems, including planning instances.

The Hash Distributed A* (HDA*) algorithm (Kishimoto, Fukunaga, and Botea 2009) is a simple, scalable parallelization of A*. HDA* runs A* on every processor, each with its own open and closed lists. HDA* uses the work distribution mechanism of PRA* (Evett et al. 1995), in which a hash function assigns each state to a unique "owner" processor, and newly generated states are sent to their owner. This allows effective load balancing and duplicate pruning.

Some recent approaches parallel best-first search (Korf and Schultze 2005; Zhou and Hansen 2007; Burns et al. 2009) are multi-threaded, and limited to shared-memory environments, which typically have few processors (currently, 8 or fewer). HDA*, on the other hand, does not rely on shared memory. It is implemented using the standard MPI message passing library, which allows to run it on a wide array of parallel environments.

Using AI planning instances, we recently showed that HDA* scaled up well on both shared memory multicores with up to 8 cores, and distributed memory clusters with up to 128 cores (Kishimoto, Fukunaga, and Botea 2009). In this paper, using both planning and 24-puzzle instances, we show that the algorithm scales up further on larger clusters, with up to 1024 cores and two terabytes of RAM.

2 Experiments

The experiments were performed on a Sun Fire X4600 cluster. Each node has 32 GB RAM and 8 AMD dual core Opteron processors (total 16 cores per node), with a clock speed of 2.4GHz. We used 1-64 nodes (i.e., 16-1024 cores).

We evaluate HDA* on two high-performance problem solvers. The first solver is the Fast Downward domain-independent planner, using the optimal version with the merge-and-shrink abstraction heuristic reported by Helmert, Haslum, and Hoffmann (2007). The second solver is the application-specific 24-puzzle solver by Korf and Felner (2002). The latter processes states 2-5x faster, which emphasizes parallel-specific costs such as the communication and the synchronization overhead. Hence, integrating HDA* into both types of solvers helps us get a more complete picture of its efficiency across a range of problems.

	1 core RAM: 128GB	16 cores 1 node 32GB	64 cores 4 nodes 128GB	128 cores 8 nodes 256GB	512 cores 32 nodes 1TB	1024 cores 64 nodes 2TB	Abst time	Opt len
Driverlog13	n/a	n/a	n/a	n/a	41.01	27.43	0.63	26
Freecell6	n/a	n/a	n/a	137.08	50.36	23.39	8.45	34
Freecell7	2864.64	n/a	66.85	34.00	11.59	18.92	10.52	41
		n/a	42.85	84.25	247.16	151.41		
Rover12	1149.74	109.62	17.68	14.50	4.06	3.47	0.17	19
		10.49	65.03	79.29	283.19	331.34		
Satellite7	n/a	n/a	502.51	209.73	55.31	37.64	0.34	21
ZenoTrav11	546.67	61.26	16.58	8.55	2.76	9.05	0.35	14
		8.92	32.97	63.94	198.07	60.41		
ZenoTrav12	n/a	n/a	n/a	n/a	180.80	90.14	0.48	21
PipNoTk24	1396.29	139.25	40.34	20.47	6.11	11.60	7.24	24
		10.03	34.61	68.21	228.52	120.37		
Pegsol28	1010.65	83.22	21.77	10.41	3.23	2.93	1.04	35
		12.14	46.42	97.08	312.90	344.93		
Pegsol29	5143.24	n/a	n/a	52.28	13.86	7.89	16.76	37
		n/a	n/a	98.38	371.09	651.87		
Pegsol30	n/a	n/a	n/a	n/a	34.14	18.19	3.30	48
Sokoban24	2635.37	n/a	57.29	33.52	15.97	47.91	1.55	205
		n/a	46.00	78.62	165.02	55.01		
Sokoban26	n/a	n/a	155.20	77.60	25.29	26.50	2.34	135
Sokoban27	n/a	n/a	n/a	n/a	36.53	26.57	3.68	87

Table 1: Search time (excluding abstraction initialization) and speedup relative to 1 core (bottom line in cells, where applicable). "n/a" = exhausted-memory failure. No speedup can be computed in cases with n/a in the "1 core" cell.

We first describe the planning results, obtained with abstraction size 1000. As shown in Table 1, the speedup of HDA* vs. A* was between 9-12 with 16 cores, 33-65 with 64 cores, 64-98 with 128 cores, 165-371 with 512 cores, and 55-651 with 1024 cores. The parallel efficiency of HDA* relative to 1 core ranges between 0.56-0.76 for 16 cores, 0.52-1.02 for 64 cores, 0.50-0.77 for 128 cores, 0.32-0.72 with 512 cores, and 0.05-0.64 with 1024 cores.

Figure 1 highlights more clearly the speedup. We show the number of processors on the x-axis and relative runtime on the y-axis, where the *relative runtime* of HDA* on an in-

[*]NICTA is funded by the Australian Government.

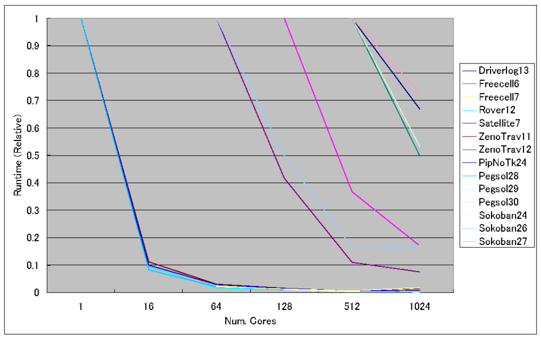

Figure 1: HDA* relative runtime in planning.

stance for n cores is defined as t_n/t_{min}, where t_n is the runtime with n cores, and t_{min} is the runtime for the minimal number of cores $c_{min} \in \{1, 16, 64, 128, 256, 512, 1024\}$ which was required to solve the problem. For the easier problems which could be solved with only a single core, HDA* scales well for up to 64 cores, but from 128 cores the speedups begin to deteriorate. For the hardest problems, HDA* displays good scaling even for 1024 cores – for the 4 problem instances that required 512 cores to solve, the relative runtime with 1024 cores ranges from 0.5-0.7. In other words, while the scalability of HDA* eventually degrades for easier problems, the algorithm scales well, relative to the minimal number of processors needed to solve the problem, on the hardest problems.

We briefly summarize other results from the planning experiments. Kumar et al (1988), as well as Karp and Zhang (1988) proposed a *random work distribution strategy* for best-first search where generated nodes are sent to a random processor. This achieves good load balancing but, unlike hash-based distribution, has no *global* duplicate detection. When both are implemented in HDA*, hash-based distribution is more than one order of magnitude better than random distribution in terms of execution time and expanded nodes.

As the number of machines increases, the larger aggregate RAM allows to solve more hard IPC instances. For example, with sequential Fast Downward on a machine with 32GB, we could only solve 22 out of the 30 IPC-6 Sokoban instances. Using 512 cores and aggregate 1024GB RAM, 28 instances were solved. Furthermore, it was possible to solve all 30 Pegsol instances with 512 cores. As shown in Table 1, Driverlog13, Zenotrav12, Pegsol30, and Sokoban-27 required more than 128 cores and 256GB aggregate memory.

One counterintuitive result is that memory contention within each node is more expensive than the communication between nodes. Thus, on clusters with fast interconnections (ours uses Infiniband), if the number of cores is kept constant, it can be better to distribute them among more nodes.

Now we turn to the 24-puzzle experiments, which used the 50-instance set reported in (Korf and Felner 2002), Table 2. First, we investigated the scalability of HDA* by solving two instances (p1 and p29) on 1-1024 cores, using the maximal number of cores per node. For a given number of cores, the speedups reported in Table 2 are lower than most speedups observed in planning. States are processed much faster in the application specific 24-puzzle solver, and therefore parallel-specific overheads have a greater weight in the total running time. Note the speedup degraded as the number of cores grows from 512 to 1024. Similarly to planning we observe a significant search overhead (Table 2).

Execution time and speedup					
	1 core	64 cores 4 nodes	128 cores 8 nodes	512 cores 32 nodes	1024 cores 64 nodes
p1	828.07	30.13 (27.48)	16.93 (48.91)	8.94 (92.63)	15.95 (51.92)
p29	1144.36	39.17 (29.22)	20.55 (55.69)	8.00 (143.05)	13.92 (82.20)
Node generation (millions)					
p1	382.09	559.21	603.20	1161.53	3704.54
p29	513.80	706.86	728.58	999.06	3122.84
Load Balance (Node Generation)					
p1		1.20	1.35	1.34	1.36
p29		1.21	1.33	1.34	1.32

Table 2: 24 puzzle scaling.

Next, we ran the 24-puzzle solver on a set of 64 nodes, which have 1024 cores in total, and varied the number of cores used per node between 1-16, so that 64-1024 cores were used. HDA* solved 31 problems using 512 cores (4GB/core), 34 problems on 256 cores (8GB/core), 36 problems on 128 cores (16GB/core), and 39 problems on 64 cores (32GB/core). That is, *reducing the number of processes down to one process per node increases the number of solved instances*. Using more cores on one node increases the search overhead and partitions the local RAM into more equal-size parts, one for each core. These two effects combined make it more likely that the RAM partition of one core will be exhausted before finding a solution.

References

Burns, E.; Lemons, S.; Zhou, R.; and Ruml, W. 2009. Best-first heuristic search for multi-core machines. In *Proc. IJCAI*.

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.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *Proceedings of ICAPS-07*, 176–183.

Karp, R., and Zhang, Y. 1988. A randomized parallel branch-and-bound procedure. In *Symp. on Theory of Computing*, 290–300.

Kishimoto, A.; Fukunaga, A.; and Botea, A. 2009. Scalable, parallel best-first search for optimal sequential planning. In *ICAPS-09*, 201–208.

Korf, R. E., and Felner, A. 2002. Disjoint pattern database heuristics. *Artificial Intelligence* 134(1-2):9–22.

Korf, R., and Schultze, P. 2005. Large-scale parallel breadth-first search. In *Proc. AAAI*, 1380–1386.

Kumar, V.; Ramesh, K.; and Rao, V. N. 1988. Parallel best-first search of state-space graphs: A summary of results. In *Proceedings of AAAI-88*, 122–127.

Zhou, R., and Hansen, E. 2007. Parallel structured duplicate detection. In *Proc. AAAI*, 1217–1223.