

Parallel Best-First Search for Optimal Sequential Planning (Research Statement)

Akihiro Kishimoto

Tokyo Institute of Technology
and JST PRESTO

Alex Fukunaga

Tokyo Institute of Technology

Adi Botea

NICTA and
The Australian National University

In classical planning, many problem instances remain hard for state-of-the-art planners, and both the memory and the CPU requirements are performance bottlenecks. The problem is especially pressing in sequential optimal planning. Despite significant progress in recent years in developing domain-independent admissible heuristics search strategies, and internal problem representations, scaling up optimal planning remains a challenge. Recent results suggest that improving heuristics may provide diminishing marginal returns, suggesting that research in orthogonal methods for speeding up search is necessary (Helmert & Roger 2008).

Multi-processor, *parallel planning* has the potential to provide both the memory and the CPU resources required to solve challenging problem instances. Furthermore, since parallelism is an inevitable trend, there is a strong need to develop techniques to enable the domain-independent planning technology to scale further. Previous work in parallel planning (Zhou & Hansen 2007; Burns *et al.* 2009) has taken a multi-threaded approach on a single, multicore machine. Thread-based approaches are specific to shared-memory environments which have less memory and CPU cores than a distributed-memory environment. Zhou & Hansen (2007) address the memory bottleneck by resorting to the external memory, which introduces an additional time overhead caused by the expensive I/O operations.

Our goal is to push the scalability further by using the large memory and CPU resources available in distributed memory clusters. We have developed Hash Distributed A* (HDA*), an algorithm that extends A* to a parallel environment. HDA* combines successful ideas from PRA* (Evet *et al.* 1995), which is based on A*, and TDS (Romein *et al.* 1999), a parallel version of IDA*.

HDA* is a distributed, asynchronous, version of A*. As introduced in PRA*, a hashing function assigns each state to a unique process. A newly generated state is sent to its destination process, instead of being queued for expansion locally in the process that generated it. The main advantage of this idea is that state duplicate detection can be performed locally, with no communication overhead. Despite this, PRA* incurs a considerable synchronization overhead caused by using synchronous communication. TDS introduced hash-based work distribution for IDA* which uses asynchronous, non-blocking communication. HDA* adopts hash-based work distribution and asynchronous, non-

blocking communication strategies from PRA* and TDS and applies it to A*.

In a typical, straightforward implementation of a hash-based work distribution scheme on a shared memory machine, each processing thread owns a local open/closed list which is implemented in shared memory, and when a state is assigned to some thread, the writer thread obtains a lock on the target shared memory, writes the state, then releases the lock. Note that whenever a processor P “sends” a state s to a destination $dest(s)$, P must wait until the lock for the shared open list (or message queue) for $dest(s)$ is available and not locked by any other processor. This results in significant synchronization overhead. In contrast, the open/closed lists in HDA* are not explicitly shared among the processors. Thus, even in a multi-core environment where it is possible to share memory, all communications are done between separate MPI processes using non-blocking send/receive operations. This enables HDA* to utilize highly optimized message buffers implemented in MPI.

In addition to reduced overhead and high performance, another key feature of HDA* is simplicity, which is especially important in parallel algorithms, as debugging a program on a multi-machine environment is very challenging.

We implemented HDA* on top of the optimal version of the Fast Downward planner (Helmert, Haslum, & Hoffmann 2007). Rather than using threads, our implementation is a distributed, message passing implementation using MPI, which allows parallelization in distributed memory environments as well as shared memory and mixed environments (cluster of multi-core machines), and supports mechanisms for both synchronous and asynchronous communication.

First, we experimentally compared HDA* with sequential A*, as well as optimized, thread-based, shared-memory implementations of PRA* and WSA* (work-stealing A*, an implementation of a standard approach to parallel search where each process maintains a local work queue, and steals work from other queues when its own queue is empty) on a dual quad-core 2.66GHz Xeon E5430 with 6MB L2 cache (total of 8 cores) and 16 gigabytes of RAM. Table 1 shows results of HDA*, PRA*, and WSA* for 4 cores and 8 cores. In addition to runtimes for all algorithms, the speedup and the parallel *efficiency* are shown for HDA*. The time to initialize the abstraction tables (Helmert, Haslum, & Hoffmann 2007) are not included in the runtime. The efficiency

# of cores	A*	WSA*	PRA*	HDA*			WSA*	PRA*	HDA*			Abstraction Initialization
	1	4	4	4			8	8	8			
	time	time	time	time	speedup	eff.	time	time	time	speedup	eff.	time
Depots10	173.16	122.72	128.59	66.16	2.62	0.65	97.01	92.36	47.8	3.62	0.45	2.03
DriverLog8	95.82	80.19	74.29	33.54	2.86	0.71	68.44	54.85	24.01	3.99	0.5	0.12
Freecell5	113.09	52.68	52.66	33.65	3.36	0.84	38.06	35.38	20.22	5.59	0.7	5.26
Rover12	375.03	295.38	269.28	122.84	3.05	0.76	234.46	196.27	80.66	4.65	0.58	0.11
Zenotravel9	135.88	119.06	106.65	47.4	2.87	0.72	103.98	79.76	35.69	3.81	0.48	0.16
PipesNoTk14	165.37	100.82	94.28	51.59	3.21	0.8	81.01	65.36	32.89	5.03	0.63	1.16
Pegsol28	661.14	392.82	363.61	190.57	3.47	0.87	318.79	249.95	115.43	5.73	0.72	0.49
Sokoban30	290.39	143.19	145.53	79.05	3.67	0.92	105.97	98.46	45.92	6.32	0.79	1.05

Table 1: Comparison of sequential A*, WSA* (work-stealing), PRA*, and HDA* on 1, 4, and 8 cores on a 2.66GHz, 16GB 8-core machine. Runtimes (in seconds), speedup, efficiency, and abstraction heuristic initialization times (not included in runtimes) are shown.

	1 core	16 cores			64 cores			128 cores			Abstraction	Opt.
	time	time	speedup	eff	time	speedup	eff	time	speedup	eff	Initialize Time	Plan Len.
Depot13	325.74	38.23	8.52	0.53	11.86	27.46	0.43	10.28	31.70	0.25	5.59	25
Rover12	521.13	58.09	8.97	0.56	16.09	32.38	0.51	10.01	52.04	0.41	0.20	19
ZenoTrav11	2688.93	n/a	n/a	n/a	82.41	32.63	0.51	44.90	59.88	0.47	0.40	14
PipesNoTk24	1269.16	165.59	7.66	0.48	42.27	30.03	0.47	34.20	37.11	0.29	7.26	24
Pegsol29	4509.22	n/a	n/a	n/a	109.65	41.13	0.64	75.35	59.84	0.47	15.15	37
Sokoban30	378.30	31.49	12.01	0.75	13.05	29.00	0.45	11.91	31.78	0.25	2.20	290
Sokoban25	n/a	n/a	n/a	n/a	n/a	n/a	n/a	129.05	n/a	n/a	3.85	134
Driverlog13	n/a	n/a	n/a	n/a	n/a	n/a	n/a	179.28	n/a	n/a	0.66	26

Table 2: Execution time (for search), speedup, and efficiency of HDA* on a large-scale cluster with using up to 128 2.4GHz cores, 2GB RAM per core (Abstraction size = 1000). The 1-core results use a machine with 128GB RAM. “n/a” = failure due to exhausted memory.

of a parallel computation is defined as S/P , where S is the speedup and P is the number of cores. As shown in Table 1, HDA* performs significantly better than WSA* and PRA*. With 4 cores, the speedup of HDA* ranges between 2.62 and 3.67, and efficiency ranges between 0.65 and 0.92. With 8 cores, the speedup of HDA* ranges between 3.62 and 6.40, and efficiency ranges between 0.45 and 0.80.

We also investigate the scaling behavior of HDA* for on a large Sun Fire X4600 cluster, where each node has 8 AMD dual core Opteron processors (total 16 cores/node) and 32 GB RAM per node, with a clock speed of 2.4GHz. We used 1-8 nodes in our experiments (i.e., 16-128 cores). Table 2 shows the runtimes and speedup, and efficiency of HDA* on 16, 64, and 128 cores, relative to sequential A*. There was 2GB RAM per process, i.e., 32GB, 128GB, and 256GB aggregate RAM for 16, 64, 128 cores, respectively. Since there are 16 cores and 32GB memory per processing node in our cluster, 2GB is the maximum amount usable per node. Using more RAM per node is not efficient, e.g., if we allocate the full 32GB RAM on a node to a single core, 15 cores would be left idle. The sequential A* was run on a special machine with a very large amount of RAM (128GB) with a 2.6GHz Opteron, compared to the 2.4GHz clock speed of the parallel processors. Thus, the runtimes for the sequential A* are scaled by a factor of 2.6/2.4. Overall, HDA* achieved a search speedup of 8-12 with 16 cores, 27-41 with 64 cores, and 31-60 with 128 cores, demonstrating reasonably good scalability for a large number of processors. The parallel efficiency of HDA* ranges between 0.48-0.75 for 16 cores, 0.43-0.64 for 64 cores, and 0.25-0.47 for 128 cores. Using 256GB of distributed memory allowed HDA* to solve

Sokoban25 and Driverlog13, which could not be solved with the memory available in our largest single machine.

Thus, our results so far are quite promising. The asynchronous, hash-based work distribution mechanism implemented in HDA* is a simple, scalable approach to parallelizing A* which allows to efficiently exploit parallel resources in both multicore, shared memory and multi-machine, distributed memory environments.

Acknowledgements

This research is supported by the JSPS Compview GCOE, the Japan MEXT program, “Promotion of Env. Improvement for Independence of Young Researchers”, and the JST PRESTO. NICTA is funded by the Australian government’s *Backing Australia’s Ability* initiative.

References

- Burns, E.; Lemons, S.; Zhou, R.; and Ruml, W. 2009. Best-First Heuristic Search for Multi-Core Machines. In *Proc. IJCAI-09*.
- Evetts, 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., and Roger, G. 2008. How Good Is Almost Perfect? In *Proc. AAAI-08*, 944–949.
- Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible Abstraction Heuristics for Optimal Sequential Planning. In *Proc. ICAPS-07*, 176–183.
- Romein, J. W.; Plaat, A.; Bal, H. E.; and Schaeffer, J. 1999. Transposition Table Driven Work Scheduling in Distributed Search. In *Proc. AAAI-99*, 725–731.
- Zhou, R., and Hansen, E. 2007. Parallel Structured Duplicate Detection. In *Proc. AAAI-07*, 1217–1223.