Scalable, Parallel Best-First Search for Optimal Sequential Planning

Akihiro Kishimoto
Tokyo Institute of Technology
and JST PRESTO
kishimoto@is.titech.ac.jp

Alex Fukunaga
Tokyo Institute of Technology
fukunaga@is.titech.ac.jp

Adi Botea
NICTA and
The Australian National University
adi.botea@nicta.com.au

Abstract

Large-scale, parallel clusters composed of commodity processors are increasingly available, enabling the use of vast processing capabilities and distributed RAM to solve hard search problems. We investigate parallel algorithms for optimal sequential planning, with an emphasis on exploiting distributed memory computing clusters. In particular, we focus on an approach which distributes and schedules work among processors based on a hash function of the search state. We use this approach to parallelize the A* algorithm in the optimal sequential version of the Fast Downward planner. The scaling behavior of the algorithm is evaluated experimentally on clusters using up to 128 processors, a significant increase compared to previous work in parallelizing planners. We show that this approach scales well, allowing us to effectively utilize the large amount of distributed memory to optimally solve problems which require hundreds of gigabytes of RAM to solve. We also show that this approach scales well for a single, shared-memory multicore machine.

Introduction

In classical planning, many problem instances remain hard for state-of-the-art planning systems. Both the memory and the CPU requirements are main causes of performance bottlenecks. The problem is especially pressing in sequential optimal planning. Despite significant progress in recent years in developing domain-independent admissible heuristics (Haslum and Geffner 2000; Edelkamp 2001; Helmert, Haslum, and Hoffmann 2007), scaling up optimal planning remains a challenge. Recent results suggest that improving heuristics may provide diminishing marginal returns (Helmert and Roger 2008), suggesting that research in orthogonal methods for speeding up search is necessary.

Multi-processor, *parallel planning*[1] has the potential to provide both the memory and the CPU resources required to solve challenging problem instances. Parallel planning has received little attention in the past, two notable exceptions being the work of Zhou and Hansen (2007) and Burns et al. (2009). While multiprocessors were previously expensive and rare, multicore machines are now ubiquitous. Future generations of hardware are likely to continue to have an increasing number of processors, where the speed of each individual CPU core does not increase as rapidly as in past decades. Thus, exploiting parallelism will be the only way to extract significant speedups from the hardware. Since parallelism is a ubiquitous trend, there is a need to develop techniques to enable the domain-independent planning technology to scale further.

Previous work in parallel planning (Zhou and Hansen 2007; Burns et al. 2009) has taken a multi-threaded approach on a single, multicore machine. The number of processors is limited to relatively small values (typically up to 8). Thread-based approaches are specific to shared-memory environments (Lin and Snyder 2009), which have less memory and CPU cores than a distributed-memory environment. Zhou and 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 introduce Hash Distributed A* (HDA*), an algorithm that extends A* (Hart, Nilsson, and Raphael 1968) to a parallel environment. HDA* combines successful ideas from previous parallel algorithms, such as PRA* (Evett et al. 1995), which is based on A*, and TDS (Romein et al. 1999), a parallel version of IDA* (Korf 1985). As introduced in PRA*, a hash 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 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. As in TDS, HDA* performs asynchronous, non-blocking communication. Unlike TDS, HDA* is a parallelization of A*. Besides performance, another key feature of HDA* is simplicity. Simplicity is especially important in parallel algorithms, as debugging a program on a multi-machine environment is very challenging.

We implement HDA* on top of the optimal version of the Fast Downward planner, which is described by Helmert,

[1] In this paper, parallel planning refers to multi-processor planning, as opposed to computing parallel plans with a serial algorithm.

Haslum, and Hoffmann (2007). Rather than using threads, our implementation is a distributed, message passing implementation using MPI (Snir and Gropp 1998), which allows parallelization in distributed memory environments as well as shared memory and mixed environments (cluster of multi-core machines), and support mechanisms for both synchronous and asynchronous communication.

The scaling behavior of the algorithm is evaluated experimentally on clusters using up to 128 processors, a significant increase compared to previous work in parallelizing planners. We show that this approach scales well, allowing us to effectively utilize the large amount of distributed memory to optimally solve problems which require hundreds of gigabytes of RAM. We also show that HDA* works well on a single, multi-core machine, outperforming algorithms such as PRA* and a parallel implementation of A* based on *work stealing*, a standard strategy in parallel search.

The rest of the paper is organized as follows. The next section presents background information, followed by related work. Our planning algorithm is described in the fourth section. We then present an empirical analysis, followed by concluding remarks and future work ideas.

Background

Efficient implementation of parallel search algorithms is challenging largely due to several types of overhead. *Search overhead* occurs when a parallel implementation of a search algorithm generates more states than a serial implementation. The main cause of search overhead is partitioning of the search space among processors, which has the side effect that the access to non-local information is restricted. For example, a sequential A* algorithm terminates immediately after a solution is found, as it is guaranteed to be optimal. In contrast, when a parallel A* algorithm finds a (first) solution, that is not necessarily an optimal solution. Better solutions might exist in non-local portions of the search space.

The search overhead can be negative, which means that parallel search expands fewer states than sequential search. A negative search overhead possibly results in achieving super-linear speedup and often indicates an inefficiency in the serial implementation or algorithm.

The *synchronization overhead* is the idle time wasted at synchronization points, where some processors have to wait for the others to reach the synchronization point. For example, in a shared-memory environment, the idle time can be caused by mutual exclusion (mutex) locks on shared data that cannot be accessed by more than one processor at a time. The *communication overhead* refers to the extra cost of inter-process information exchange, and mainly occurs in a distributed-memory environment.

The key to achieving a good speedup in parallel search is to minimize such overheads. This is often a difficult task, in part because the overheads depend on one another. For example, reducing the search overhead usually increases the synchronization and the communication overheads. It is hard to theoretically characterize the best trade-off in minimizing the overheads. In practice, the trade-offs are evaluated and tuned experimentally.

Work stealing is a standard approach for parallel search, and is used in many applications in shared-memory environments. It aims at achieving good load balancing (i.e., keep all processors busy at all times). In work-stealing, each processor maintains a local work queue. When a processor P generates new work (i.e., new states to be expanded) w, it places w in P's own local queue. When P has no work in its queue, it steals work from the queue of a busy processor. Several strategies have been studied to select a processor offloading the work (e.g. (Feldmann 1993; Frigo, Leiserson, and Randall 1998; Rao and Kumar 1987)). A typical implementation of the local work queue for parallel A^* is to simply use the local open list of a processor.

In many search applications, including planning benchmarks, the search space is a graph rather than a tree. More than one path can lead to the same state. Sequential best-first search can detect and handle this by using a closed list (i.e., hash table) or duplicate detection techniques (e.g. (Korf and Zhang 2000; Zhou and Hansen 2006)). Efficient duplicate detection is critical for performance, both in serial and parallel search algorithms, and can potentially eliminate vast amounts of redundant work.

In parallel search, performing duplicate state detection in parallel incurs several overheads. The cause of an overhead depends on the choices of algorithms and machine environments. In a shared-memory environment, many approaches, including work-stealing, need mutex operations for the open and closed lists, to guarantee that these structures are correctly managed. For example, an early work on parallel A^* which shares one open list among all processors (Kumar, Ramesh, and Rao 1988) has a severe bottleneck caused by the contention for the open list.

A discussion on how such challenges are addressed in actual algorithms can be found in the next section.

Related Work

Parallel Retracting A^* (PRA*) (Evett et al. 1995) simultaneously addresses the problem of work distribution and duplicate state detection. In PRA*, each processor maintains its own open and closed lists. A hash function maps each state to exactly one processor. When generating a state, PRA* distributes it to the corresponding processor. If the hash keys are distributed uniformly across the processors, load balancing is achieved. After receiving states, PRA* has the advantage that duplicate detection can be performed efficiently. All the checks are done locally at the destination process.

On the other hand, PRA* incurs a significant synchronization overhead, as it uses synchronous communication to distribute states. When a processor P generates a new state s and sends it to the destination processor Q, P blocks and waits for Q to confirm that s has successfully been received and stored. This is needed because PRA* was implemented on a Connection Machine, where each processor had a limited amount of local memory. When a processor's memory became full, a *retraction* mechanism was used to remove nodes in order to free memory.

Transposition-table driven work scheduling (TDS) (Romein et al. 1999) is a distributed memory, parallel IDA* algorithm. Similarly to PRA*, TDS distributes work using a state hash function. As opposed to PRA*, it has no synchronization overhead. Extensive performance analysis on TDS was later performed (Romein et al. 2002). The transposition table is partitioned over processors to be used for detecting and pruning duplicate states that arrive at the processor. In this way, TDS exploits the large amounts of local memory that are available on modern machines. As the number of processing nodes increases, the amount of RAM in the system increases, allowing more effective duplicate state detection and pruning. This allows TDS to exhibit a very low (even negative) search overhead, compared to a sequential IDA* that runs on a single computational node, with a limited amount of memory.

A very important contribution of TDS is to make *all* communication asynchronous. After processor P sends a state to its destination Q, P expands the next state from its local open queue without waiting for Q to reply. Instead, each processor periodically checks if a new state arrives. A possible concern with TDS is large communication overhead, but Romein et al. showed that this was not a significant concern because several states that a processor sends to the same destination can be packed into one message to reduce the communication overhead. TDS achieved impressive speedups in applications such as the 15-puzzle, the double-blank puzzle, and the Rubik's cube, on a distributed-memory machine. The ideas behind TDS have also been successfully integrated in adversarial two-player search (Kishimoto and Schaeffer 2002; Romein and Bal 2003).

External memory search, a related but different technique, has been used to address memory bottlenecks (e.g., (Edelkamp and Jabbar 2006)). An issue with using external memory (such as disk) is the overhead of expensive I/O operations. In contrast, parallel search can potentially handle both memory and time bottlenecks. Interestingly, some solutions to reducing the I/O overhead in external memory search could in principle be adapted to handle the interprocess communication overhead in parallel search. For example, Zhou and Hansen (2007) and, more recently, Burns et al. (2009) adapt the idea of structured duplicate detection, which was originally introduced for external memory search (Zhou and Hansen 2004), to parallel search.

Zhou and Hansen introduce a parallel, breadth-first search algorithm. Parallel structured duplicate detection seeks to reduce synchronization overhead. The original state space is partitioned into collections of states called blocks. The duplicate detection scope of a state contains the blocks that correspond to the successors of that state. States whose duplicate detection scopes are disjoint can be expanded with no need for synchronization. Burns et al. (2009) have investigated best-first search algorithms that include enhancements such as structured duplicate detection and speculative search. These techniques were effective in a shared memory machine with up to 8 cores.

Hash Distributed A*

We now describe HDA*, a parallelization of A*. HDA* is a simple algorithm which combines the hash-based work distribution strategy of PRA* and the asynchronous communications of TDS. Unlike PRA*, HDA* does not incorporate any mechanism for node retraction. This combination results in a simple algorithm which achieves scalability for both speed and memory usage.

In HDA* the closed and open lists are implemented as a distributed data structure, where each processor "owns" a partition of the entire search space. The partitioning is done via a hash function on the state, and is described later.

The overall HDA* algorithm begins by the expansion of the start state at the head processor.

Each processor P executes the following loop until an optimal solution is found:

1. First, P checks if a new state has been received in its message queue. If so, P checks for this new state s in P's closed list, in order to determine whether s is a duplicate, or whether it should be inserted in P's local open list[2].
2. If the message queue is empty, then P selects a highest priority state from its local open list and expands it, resulting in newly generated states. For each of the newly generated states s_i, a hash key $K(s_i)$ is computed based on the state representation, and the generated state is then sent to the processor which owns $K(s_i)$. This send is asynchronous and non-blocking. P continues its computation without waiting for a reply from the destination.

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 $dest(s)$, then P must wait until the lock for shared open list (or message queue) for $dest(s)$ is available and not locked by any other processor. This results in significant synchronization overhead – for example, it was observed in (Burns et al. 2009) that a straightforward implementation of PRA* exhibited extremely poor performance on the Grid search problem, where multi-core performance for up to 8 cores was consistently *slower* than sequential A*. While it is possible to speed up locking operations by using, for example, highly optimized lock operations implementations in inline assembly language such as those which are commonly used in the two-player game community[3], the performance degradation due to the increase in synchronization points caused by locks remains a considerable problem (see discussion in the next section).

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. Our implementation

[2] Even if the heuristic function (Helmert, Haslum, and Hoffmann 2007) is consistent, parallel A* search may sometimes have to re-open the state saved in the closed list. For example, P may receive many identical states s with various priorities from different processors and these s may reach P in any order.

[3] `http://gps.tanaka.ecc.u-tokyo.ac.jp/osl/osl/doc/html/lightMutex\_8h-source.html`

achieves it by using MPI_Bsend and MPI_Iprobe. This enables HDA* to utilize highly optimized message buffers implemented in MPI. Additionally, more than one state can be packed to reduce the number of MPI communications.

In parallel A*, even if a process discovers a goal state, it is not guaranteed to be optimal (Kumar, Ramesh, and Rao 1988). When a processor discovers a goal state G, the processor broadcasts the cost of G to all processors. The search cannot terminate until all processors have proved that there is no solution with a cost better than that of G. In order to correctly terminate parallel A*, it is not sufficient to check the local open list at every processor. We must also prove that there is no work (states) currently on the way to arrive at a processor. Various algorithms to handle termination exist. In our implementation of HDA*, we used the time algorithm of Mattern (1987), which was also used in TDS.

In a hash based work distribution scheme, the choice of the hash function is essential for achieving uniform distribution of the keys, which results in effective load balancing. Our implementation of HDA* uses the Zobrist function to map a SAS+ state representation (Bäckström and Nebel 1995) to a hash key. The Zobrist function (Zobrist 1970) is commonly used in the game tree search community. It is a very fast hash function based on incrementally XOR'ing the components of a state. The Zobrist function was previously used in a sequential planner by Botea et al. (2005).

Results

We experimentally evaluated HDA* by running Fast Downward + HDA* on classical planning instances from the IPC-3, IPC-4, and IPC-6 planning competitions. Our experimental code is based on a sequential optimal version of Fast Downward, enhanced with an explicit state abstraction heuristic (Helmert, Haslum, and Hoffmann 2007). HDA* is implemented in C++ and compiled with g++, parallelized using the MPI message passing library. While HDA* and other parallel search algorithms have nondeterministic behavior and there will be some differences between identical invocations of the algorithms, on the runs where we collected multiple data points, we did not observe significant differences between runs of HDA*. Therefore, due to the enormous resource requirements of a large-scale experimental study[4], the results shown are for single runs.

Experiments on a Single, Multi-Core Machine

First, we investigate the scaling of HDA* on a single machine. We compare HDA* with sequential A* and shared-memory implementations of PRA* and WSA* (work-stealing A*). All of our algorithms use TCMalloc (`http://code.google.com/p/google-perftools/`), a fast and thread-safe memory management library.[5] In addition to locks available in the Boost C++ library, we also incorporated spin locks used in GPSshogi[6] in order to speed up WSA* and PRA*. The spin lock implementation is based on the "xchgl" assembly operation.

In WSA*, there is a trade-off between searching the most promising states in parallel and working on the states in a local open list for avoiding synchronization overhead. The best work strategy is selected, after comparing several work-stealing implementations such as techniques in (Feldmann 1993; Kumar, Ramesh, and Rao 1988). Our WSA* manages the current best score of the states in the open lists. If a thread expands a less promising state in its local open list, it tries to steal work from a thread having the most promising states in the next state selection phase.

These experiments were run on a dual quad-core 2.66GHz Xeon E5430 with 6MB L2 cache (total of 8 cores) and 16 gigabytes of RAM. Each algorithm used the full 16 gigabytes of RAM. That is, n-core HDA* spawns n processes, each using 16/n gigabytes of RAM, sequential A* used the full 16GB available, and the multithreaded PRA* and WSA* algorithms shared 16GB of RAM among all of the threads.

Table 1 shows the speedup 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 efficiency 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 from 2.62 to 3.67, and the efficiency ranges from 0.65 to 0.92. With 8 cores, the speedup of HDA* ranges from 3.62 to 6.40, and the efficiency ranges from 0.45 to 0.80.

Although it is not possible to directly compare these results with the results in (Burns et al. 2009) due to many factors (different underlying search code which uses different heuristics, different # of cores, different problem instances[7], etc.), it is possible to make some observations about comparative speedups. Compared to sequential A*, the algorithms proposed in (Burns et al. 2009) achieve excellent, even super-linear speedups, and this is because of techniques such as speculative expansion – their algorithms do not directly parallelize A*; rather, they implement a different, node expansion strategy in order to overcome parallelization overheads, and this resulted in a strategy which outperformed A*. On the other hand, if we look at the scaling behavior of each algorithm implementation as the number of cores is increased (i.e., comparison of the exact same code running on

[4] We are using a shared cluster for our experiments, and large-scale experiments have issues of resource contention, because we are competing for these resources with hundreds of other users. In addition, some of the resources such as the 128GB RAM machine incur a usage cost per CPU hour.

[5] Using TCMalloc also resulted in a slight speedup of our baseline sequential A* compared to the version obtained from Helmert.

[6] `http://gps.tanaka.ecc.u-tokyo.ac.jp/gpsshogi/pukiwiki.php?GPSshogi`

[7] Different instances are used because key aspects of the algorithms, such as the heuristic, are different. Instances which are time-consuming for the algorithms in (Burns et al. 2009) are not necessarily difficult for HDA*, and vice versa – for example, the depots-7 and freecell-3 instances are solved in 9.9 seconds and 4.2 seconds by sequential Fast Downward (abstraction size parameter 1000). Instances solved quickly by sequential runs do not yield much useful information, because they are dominated by startup and termination overheads when parallelized. Thus, we chose our instances based on preliminary experiments which indicated the difficult problems for sequential Fast Downward+LFPA.

	A*	WSA*	PRA*	HDA*			WSA*	PRA*	HDA*			Abstraction
# of cores	1	4	4	4			8	8	8			Initialization
	time	time	time	time	speedup	eff.	time	time	time	speedup	eff.	time
Depots4	74.87	47.98	48.31	24.38	3.07	0.77	37.92	33.67	15.22	4.92	0.61	4.21
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
PipesTank15	60.25	31.53	32.13	18.18	3.31	0.83	24.26	22.03	11.36	5.31	0.66	4.92
Pegsol26	44.88	27.45	26.22	12.83	3.5	0.87	22.6	18.57	8.83	5.08	0.64	6.64
Pegsol27	152.79	92.13	81.39	44.01	3.47	0.87	74.26	57.96	26.57	5.75	0.72	0.82
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
Sokoban4	38.45	22.66	21.65	10.71	3.59	0.9	18.42	15.2	6.27	6.13	0.77	2.5
Sokoban9	43.69	26.24	24.29	12.34	3.54	0.89	21.62	17.02	6.82	6.4	0.8	1.4
Sokoban13	41.59	27.22	25.84	12.85	3.24	0.81	22.06	18.05	7.44	5.59	0.7	0.77
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 Xeon E5430. Runtimes (in seconds), speedup, efficiency, and abstraction heuristic initialization times (not included in runtimes) are shown.

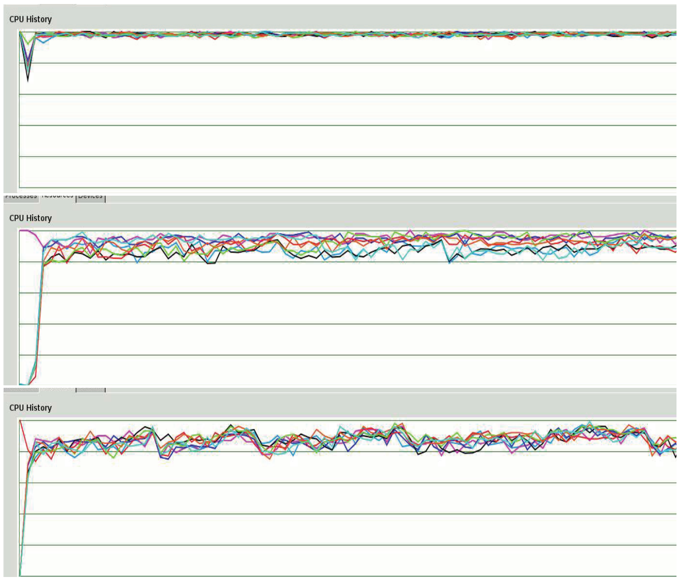

Figure 1: CPU Utilization of HDA* (top), PRA* (middle), and WSA* (bottom) on 8 core multi-core (Depots-10).

1 core vs. n cores), the scaling of speedups and efficiency in HDA* is competitive with the scaling of speedups and efficiency in the algorithms in (Burns et al. 2009).

Figure 1 shows snapshots of CPU usage for 8-core runs of HDA*, PRA*, and WSA*, respectively, on the Depots-10 problem instance. The horizontal and vertical lines represent the time and the load factor respectively. Because of its asynchronous communications, HDA* keeps all 8 cores at close to 100% usage. On the other hand, the CPU usage for PRA* and WSA* fluctuates significantly, due to the synchronization overhead. This explains the superior performance of HDA* compared to PRA* and WSA*.

Experiments on a Cluster

Next, we investigate the scaling behavior of HDA* for clusters of machines. These parallel experiments were performed on a Sun Fire X4600 cluster, where each node has 8 AMD dual core Opteron processors (total 16 cores per 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). Although the CPU for this machine is a 2.6GHz Opteron, we scaled the results for sequential A* by a factor of 2.6/2.4 in Tables 2–3, so that the sequential results could be compared with the parallel results which were run on 2.4GHz Opteron CPUs. We verified the correctness of this scaling factor by using sequential search runtimes on easier problems with both this 2.6GHz machine and a single core on the 2.4GHz parallel machine. Although this 128GB machine has 16 cores, we use just one core, and use all of the 128GB of RAM for a sequential A* process.[8]

The times shown in Table 2 include the time for the search algorithm execution, and *exclude* the time required to compute the abstraction table for the LFPA heuristic, since this phase of Fast Downward+LFPA has not been parallelized yet[9] and therefore requires the same amount of time to run

[8] Due to the runtime scaling, as well as the architectural differences between Opteron and Xeon processors, the 1-core Opteron results in Tables 2-3 are not directly comparable with the 1-core Xeon results in Table 1.

[9] We are currently completing a straightforward parallelization

	1 core	16 cores			64 cores			128 cores			Abstraction Initialize	Opt. Plan
	time	time	speedup	eff	time	speedup	eff	time	speedup	eff	Time	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
Pegsol28	886.01	75.69	11.71	0.73	21.75	40.73	0.64	17.54	50.51	0.29	0.99	35
Pegsol29	4509.22	n/a	n/a	n/a	109.65	41.13	0.64	75.35	59.84	0.47	15.15	37
Sokoban12	466.90	37.47	12.46	0.78	14.24	32.80	0.51	12.37	37.75	0.29	2.39	172
Sokoban14	2201.76	n/a	n/a	n/a	55.75	39.50	0.62	34.48	63.86	0.50	1.34	205
Sokoban15	2639.30	n/a	n/a	n/a	76.55	34.48	0.54	45.15	58.46	0.46	2.77	155
Sokoban21	1529.45	145.76	10.49	0.66	45.74	33.44	0.52	29.25	52.28	0.41	3.77	162
Sokoban23	589.89	45.89	12.85	0.80	16.26	36.28	0.57	13.23	44.60	0.35	2.06	177
Sokoban24	950.55	76.82	12.37	0.77	26.11	36.41	0.57	18.06	52.62	0.41	2.37	125
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, excluding abstraction initialization), speedup, and efficiency on a large-scale cluster with using up to 128 2.4GHz Opteron cores, 2GB RAM per core (Abstraction size = 1000). The 1-core results use a Opteron-based machine with 128GB RAM. "n/a" = failure due to exhausted memory.

regardless of the number of cores. The abstraction heuristic initialization times are shown separately in Table 2. For example, the IPC6 Pegsol-28 instance, which requires 4509 seconds with 1 core, was solved in 75 seconds with 128 cores, plus 15.15 seconds for the abstraction table generation. The "n/a" in the Sokoban-14/15 entries for 16 cores indicates that 32GB was insufficient to solve these instances (additional memory would allow 16-cores to solve these instances). The Sokoban-25 and Driverlog-13 instances were only solved using 128 cores, because only the 128-core run had sufficient memory (256GB).

Overall, HDA* achieved a search speedup of 8-13 with 16 cores, 27-41 with 64 cores, and 31-64 with 128 cores, demonstrating reasonably good scalability for a large number of processors. The parallel efficiency of HDA* ranges between 0.48-0.77 for 16 cores, 0.43-0.64 for 64 cores, and 0.25-0.50 for 128 cores.

The *search overhead*, which indicates the extra states explored by parallel search, is defined as:

$$SO = 100 \times (\frac{\text{number of states searched by parallel search}}{\text{number of states searched by sequential search}} - 1).$$

Figure 2 shows the search overhead, plotted against the length of the optimal plan for the 16, 64, and 128 core data in Table 2. Although most of the data points are at the lower left corner (low search overhead), there are some data points with very high search overhead. The figure shows that search overhead in HDA* is clearly correlated with solution length. The data points in the right side, with long solutions and high overhead, are IPC6 Sokoban instances.

A common metric for measuring how evenly the work is distributed among the cores is the *load balance*, defined as the ratio of the maximal number of states searched by a core and the average number of states searched by each core. For 128 cores, load balance ranges from 1.03 to 1.13 on the instances shown. One possible reason for the imbalance in some domains may be the "hotspots" – frequently generated duplicate nodes which are sent to a small number of cores by the hash function.

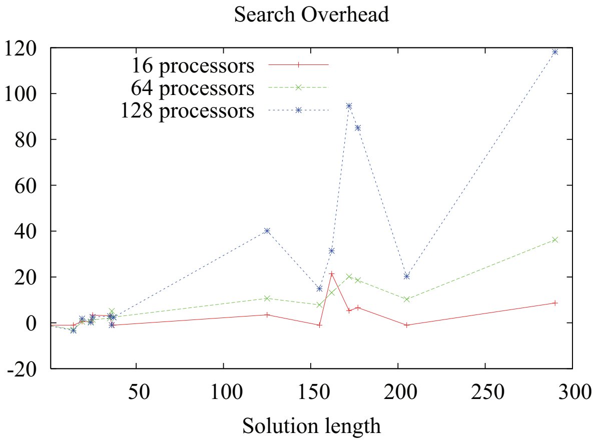

Figure 2: Search overhead as a function of solution length.

The size of the heuristic abstraction table is a control parameter for Fast Downward. While almost all of the data presented in this paper is based on a value of 1000 for the abstraction size parameter, preliminary experiments have shown that search speedups were not dependent on the abstraction size parameter. As an example, Table 3 compares search times for 1 and 16 cores using an abstraction size of 5000. The speedups and parallel efficiencies are comparable to the 16-core results in Table 2 for the same instances using abstraction size 1000. Of course, as the abstraction size parameter is increased, the amount of time spent for

which should require $O(log(n))$ time, instead of the current $O(n)$, for n SAS+ variables in a problem instance.

	1 core	16 cores			Abstraction
	time	time	speedup	effic.	init time
Rover12	404.51	48.04	8.44	0.53	1.12
ZenoTravel11	12.13	2.08	5.82	0.36	1.29
PipesNoTk24	162.52	25.33	6.42	0.40	21.81
Pegsol28	974.63	83.33	11.70	0.73	16.35
Sokoban24	979.21	102.05	9.6	0.60	6.52

Table 3: Search execution time, speedup, and efficiency for abstraction size = 5000. Times are in seconds (same machines as in Table 2).

initializing the abstraction table, a serial bottleneck in our current implementation, increases, so this increases the total runtime. Furthermore, there is a tradeoff between the size of the abstraction table and the amount of RAM left available for the local open/closed lists. Increasing the abstraction table size generally results in more efficient search, but if this also results in a large reduction in available memory, it can lead to search failures due to RAM exhaustion.

Finally, we note that as the number of machines increases, we increase not only the number of CPU cores available, but also the aggregate amount of RAM available. This is very important for algorithms such as A*, where the amount of RAM is the limiting factor. We have observed that as we increase the aggregate RAM from 16GB to 256GB, we are able to solve an increasing number of hard IPC instances. For example, with sequential Fast Downward on a machine with 16GB, we could only solve 11 out of the 30 IPC6 Sokoban instances (with an abstraction size of 1000), but using a cluster with aggregate 256GB RAM, 21 instances were solved. Fast Downward exhausts 128GB RAM within 3-6 hours, depending on the problem. The Sokoban-25 and Driverlog-13 instances could only be solved using 128 cores and 256GB aggregate cluster memory.

Discussion and Conclusion

In order to scale up the capabilities of sequential optimal planning algorithms, this paper investigated the parallelization of the search algorithm. We developed Hash Distributed A*, a parallelization of A* for a distributed memory cluster. HDA* relies on two key ideas: (1) distribution of work using a hash value for generated states, which is from PRA* (Evett et al. 1995), and (2) completely asynchronous operation, which was shown to be effective in TDS, a parallel IDA* algorithm (Romein et al. 1999). We implemented HDA* as a replacement for the sequential A* search engine for the state-of-the-art, optimal sequential planner, Fast Downward+LFPA (Explicit State Abstraction Heuristic) (Helmert, Haslum, and Hoffmann 2007).

Our experimental evaluation shows that HDA* scales well, achieving 30–60x speedup on 128 processing cores. HDA* exploits the large amount of distributed memory available on a modern cluster, enabling larger problems to be solved than previously possible on a single machine. We are in the process of scaling the experiments to larger number of cores (up to 1024).

One particularly attractive feature of HDA* is its simplicity. Work distribution is done by a simple hash function, and there is no complex load balancing mechanism. All communications are asynchronous, so complex synchronization protocols are not necessary. Despite its simplicity, HDA* achieves significant speedup over the state-of-the-art, Fast Downward+LFPA planner. Simplicity for parallel algorithms is very important, particularly for an algorithm that runs on multiple machines, as debugging a multi-machine, multi-core algorithm is extremely challenging. For comparison, we have also started to implement a distributed memory, work-stealing algorithm, and have found that it is significantly more difficult to implement correctly and efficiently compared to HDA*.

While we developed HDA* for distributed memory parallel search on a distributed memory cluster of machines, we have also shown that HDA* achieves reasonable speedup on a single, shared memory machine with up to 8 cores, with results that are superior to two, previous approaches: thread-based work-stealing (WSA*) and PRA*. HDA* yields speedups of 3.6-6.3 on 8 cores. We have also shown that, on an 8-core machine, HDA* keeps all processors almost completely busy, while PRA* and WSA*, allow processors to be idle due to synchronization overhead.

Thus, the main contributions of this paper are: (1) the proposal of HDA*, a simple, parallel best-first algorithm combining the hash-based work distribution strategy of PRA* and the asynchronous communication strategy of TDS; (2) an experimental demonstration that HDA* can significantly speed up the state-of-the-art Fast-Downward+LFPA planner; (3) an experimental demonstration that HDA* scales up reasonably well to 128 cores; and (4) a demonstration that HDA* performs reasonably well on a single machine, outperforming standard thread-based techniques (WSA* and PRA*). This work has shown that HDA* is a promising approach to parallelizing best-first search for sequential optimal planning.

Currently, HDA* uses a single process per core. Although the machine it runs on can be a shared memory machine (most modern machines are multicore, shared memory machines), HDA* executes as a set of independent processes without sharing any memory resources among cores that are on the same machine. This means that the memory used for the LFPA abstraction heuristic (Helmert, Haslum, and Hoffmann 2007) is unnecessarily replicated n times on an n-core machine, which can be a significant source of inefficiency in memory usage. We are currently investigating a hybrid, distributed/shared memory implementation of HDA* which eliminates this inefficiency. One possible direction for such a hybrid implementation is to distribute work among machines using hash-based distribution, but within a single machine incorporate techniques such as speculative expansion that have been shown to scale well on a shared memory environment with up to 8 cores (Burns et al. 2009).

The speedups we obtain are more modest than the results obtained by Romein et al. (1999) for TDS in puzzle domains, who report linear speedups compared to sequential IDA*. One reason why such impressive speedups are possible for parallel IDA* might be that increasing the aggregate RAM results in a larger, distributed transposition table for

IDA*, which leads to more pruning, and therefore actually improves the search efficiency relative to sequential IDA* on a single machine with less aggregate RAM. In search spaces with duplicate states, the search overhead incurred by sequential IDA* by exploring duplicate states is enormous, and therefore, a massive, distributed transposition table results in search efficiency improvements which make up for any overheads incurred by parallelization. In contrast, for parallel A* algorithms, increasing the amount of aggregate RAM affects whether problems can be solved or not (i.e., whether memory is exhausted before search completes), but by itself, increased memory does not improve the number of nodes explored by the search algorithm, since sequential A* does not reopen duplicate states. On the other hand, it is also possible to use massive amounts of aggregate RAM in different ways to improve performance (e.g., increasing the size of an abstraction-based heuristic table). This remains an area for future work.

Another area of future work is an in-depth investigation of the scalability as the number of nodes increases. Our parallel experiments have used clusters of multicore nodes (16 cores per node), so even with 128 cores, this involved 8 nodes. Since inter-node communications overhead is more significant than intra-node communications within a single node, further investigation is needed to understand the impact of inter-node communications on the scalability of HDA*.

Finally, we note that the serial computation of the abstraction heuristic table (Helmert, Haslum, and Hoffmann 2007) results in a serial bottleneck, as illustrated in our results. We are currently parallelizing an abstraction algorithm.

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

Bäckström, C., and Nebel, B. 1995. Complexity Results for SAS$^+$ Planning. *Computational Intelligence* 11(4):625–655.

Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators. *Journal of Artificial Intelligence Research* 24:581–621.

Burns, E.; Lemons, S.; Zhou, R.; and Ruml, W. 2009. Best-First Heuristic Search for Multi-Core Machines. In *Proceedings of IJCAI*.

Edelkamp, S., and Jabbar, S. 2006. Cost-optimal external planning. In *Proceedings of AAAI*, 821–826.

Edelkamp, S. 2001. Planning with Pattern Databases. In *Proceedings of European Conference on Planning*, 13–34.

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.

Feldmann, R. 1993. *Spielbaumsuche auf Massiv Parallelen Systemen*. Ph.D. Dissertation, University of Paderborn. English translation titled *Game Tree Search on Massively Parallel Systems* is available.

Frigo, M.; Leiserson, C. E.; and Randall, K. H. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN Conferences on Programming Language Design and Implementation (PLDI'98)*, 212–223.

Hart, P.; Nilsson, N.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. on Systems Science and Cybernetics* 4(2):100–107.

Haslum, P., and Geffner, H. 2000. Admissible Heuristics for Optimal Planning. In *Proc. Fifth International Conference on AI Planning and Scheduling*, 140–149.

Helmert, M., and Roger, G. 2008. How Good Is Almost Perfect? In *Proceedings of AAAI-08*, 944–949.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible Abstraction Heuristics for Optimal Sequential Planning. In *Proceedings of ICAPS-07*, 176–183.

Kishimoto, A., and Schaeffer, J. 2002. Distributed Game-Tree Search Using Transposition Table Driven Work Scheduling. In *Proceedings of the 31st International Conference on Parallel Processing ICPP-02*, 323–330.

Korf, R. E., and Zhang, W. 2000. Divide-and-conquer frontier search applied to optimal sequence alignment. In *Proceedings of AAAI-2000*, 910–916.

Korf, R. 1985. Depth-first Iterative Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence* 97:97–109.

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.

Lin, C., and Snyder, L. 2009. *Principles of Parallel Programming*. Addison–Wesley.

Mattern, F. 1987. Algorithms for distributed termination detection. *Distributed Computing* 2(3):161–175.

Rao, V. N., and Kumar, V. 1987. Parallel depth-first search on multiprocessors part I: Implementation. *International Journal of Parallel Programming* 16(6):479–499.

Romein, J. W., and Bal, H. E. 2003. Solving awari with parallel retrograde analysis. *IEEE Computer* 36(10):26–33.

Romein, J. W.; Plaat, A.; Bal, H. E.; and Schaeffer, J. 1999. Transposition Table Driven Work Scheduling in Distributed Search. In *Proceedings of AAAI-99*, 725–731.

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

Snir, M., and Gropp, W. 1998. *MPI: The Complete Reference*. MIT Press.

Zhou, R., and Hansen, E. 2004. Structured Duplicate Detection in External-Memory Graph Search. In *Proceedings of AAAI-04*, 683–689.

Zhou, R., and Hansen, E. 2006. Domain-independent structured duplicate detection. In *Proc. AAAI-06*, 683–688.

Zhou, R., and Hansen, E. 2007. Parallel Structured Duplicate Detection. In *Proceedings of AAAI-07*, 1217–1223.

Zobrist, A. L. 1970. A new hashing method with applications for game playing. Technical report, Dept of CS, Univ. of Wisconsin, Madison. Reprinted in *International Computer Chess Association Journal*, 13(2):169-173, 1990.