22nd National Conference on Artificial Intelligence (AAAI-07) • Vancouver, Canada • July 2007

Parallel Structured Duplicate Detection

Rong Zhou
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304
rzhou@parc.com

Eric A. Hansen
Dept. of Computer Science and Engineering
Mississippi State University
Mississippi State, MS 39762
hansen@cse.msstate.edu

Abstract

We describe a novel approach to parallelizing graph search using structured duplicate detection. Structured duplicate detection was originally developed as an approach to external-memory graph search that reduces the number of expensive disk I/O operations needed to check stored nodes for duplicates, by using an abstraction of the search graph to localize memory references. In this paper, we show that this approach can also be used to reduce the number of slow synchronization operations needed in parallel graph search. In addition, we describe several techniques for integrating parallel and external-memory graph search in an efficient way. We demonstrate the effectiveness of these techniques in a graph-search algorithm for domain-independent STRIPS planning.

Introduction

Graph search is a central problem-solving technique in many areas of AI, including planning, scheduling, and combinatorial optimization. Because graph-search algorithms are both computation-intensive and memory-intensive, developing techniques for improving the efficiency and scalability of graph search continues to be an active and important topic of research. An important category of research questions relates to how to best exploit available hardware resources in graph search. The possibilities include using external memory, such as disk, to increase the number of visited nodes that can be stored in order to check for duplicates, as well as using parallel processors, or multiple cores of the same processor, in order to improve search speed.

In this paper, we describe a novel approach to efficient parallelization of graph search. Grama and Kumar (1999), in a survey of parallel search algorithms, point out that "Decreasing the communication coupling between distributed [OPEN] lists increases search overhead, and conversely, reducing search overhead using increased communication has the effect of increasing communication overhead." This dilemma is faced by virtually all previous approaches to parallel graph search (Kumar, Ramesh, & Rao 1988; Dutt & Mahapatra 1994). Although the assumption is often made, for the purpose of parallelization, that a large search problem can be decomposed into a set of smaller ones that are independent from each other, most graph-search problems have sub-problems that interact in complex ways via paths that connect them in a graph. For graphs with many duplicate paths, achieving efficient parallel search remains a challenging and open research problem.

Many researchers have recognized that external-memory algorithms and parallel algorithms often exploit similar problem structure to achieve efficiency. This has inspired some recent work on parallelizing graph search using techniques that have proved effective in external-memory graph search. Delayed duplicate detection is an approach to external-memory graph search in which newly-generated nodes are not immediately checked against stored nodes for duplicates; instead, they are written to a file that is processed later, in an I/O-efficient way, to remove duplicates. Based on this idea, some recent approaches to reducing communication overhead in parallel graph search delay duplicate-detection-induced communication operations so that they can be combined later into fewer operations, and performed more efficiently (Korf & Schultze 2005; Niewiadomski, Amaral, & Holte 2006; Jabbar & Edelkamp 2006; Korf & Felner 2007). But delaying communication between multiple processing units can increase search overhead by creating a large number of duplicates that require temporary storage and eventual processing.

Structured duplicate detection is an alternative approach to external-memory graph search that exploits the structure of a search graph in order to localize memory references (Zhou & Hansen 2004; 2005; 2006b; 2007). It can outperform delayed duplicate detection because it removes duplicates *as soon as* they are generated, instead of storing them temporarily for later processing, and thus has lower overhead and reduced complexity. In this paper, we describe a generalization of structured duplicate detection, called *parallel structured duplicate detection*, that reduces communication overhead in parallel graph search using techniques that do not subsequently increase search overhead. We show that this results in efficient parallelization.

Because graph search is typically memory bound, parallelizing it, by itself, will not usually improve scalability. Therefore we show how to integrate our approach to parallel graph search with the approach to external-memory graph search based on structured duplicate detection. We demonstrate the effectiveness of these techniques in a graph-search algorithm for domain-independent STRIPS planning.

Structured duplicate detection

We begin with an overview of previous work on structured duplicate detection. Structured duplicate detection (SDD) is an approach to external-memory graph search that leverages local structure in a graph to partition stored nodes between internal memory and disk in such a way that duplicate detection can be performed immediately, during node expansion, instead of being delayed.

The local structure that is leveraged by this approach is revealed by a state-space projection function that is a many-to-one mapping from the original state space to an abstract state space. If a state x is mapped to an abstract state y, then y is called the *image* of x. One way to create a state-space projection function is by ignoring the value of some state variables. For example, if we ignore the positions of all tiles in the Fifteen Puzzle and consider only the position of the "blank," we get an abstract state space that has only sixteen abstract states, one for each possible position of the blank.

Given a state-space graph and projection function, an *abstract state-space graph* is constructed as follows. The set of nodes in the abstract graph, called *abstract nodes*, corresponds to the set of abstract states. An abstract node y' is a successor of an abstract node y iff there exist two states x' and x in the original state space, such that (1) x' is a successor of x, and (2) x' and x map to y' and y, respectively. Figure 1(b) shows the abstract state-space graph created by the simple state-space projection function that maps a state into an abstract state based on the position of the blank. Each abstract node B_i in Figure 1(b) corresponds to the set of states with the blank located at position i in Figure 1(a).

In SDD, stored nodes in the original search graph are divided into "$n$blocks," where an $n$block corresponds to a set of nodes that map to the same abstract node. Given this partition of stored nodes, SDD uses the concept of *duplicate-detection scope* to localize memory references. The *duplicate-detection scope* of a node x in the original search graph is defined as all stored nodes (or equivalently, all $n$blocks) that map to successors of the abstract node y that is the image of node x under the projection function. In the Fifteen Puzzle example, the duplicate-detection scope of nodes that map to abstract node B_0 consists of nodes that map to abstract node B_1 or B_4.

The concept of duplicate-detection scope allows a search algorithm to check duplicates against a fraction of stored nodes, and still guarantee that all duplicates are found. An external-memory graph search algorithm can use RAM to store $n$blocks within the current duplicate-detection scope, and use disk to store other $n$blocks when RAM is full. SDD is designed to be used with a search algorithm that expands a set of nodes at a time, such as breadth-first search, where the order in which nodes in the set are expanded can be adjusted to minimize disk I/O. SDD's strategy for minimizing disk I/O is to order node expansions such that changes of duplicate-detection scope occur as infrequently as possible, and involve change of as few $n$blocks as possible. When RAM is full, $n$blocks outside the current duplicate-detection scope are flushed to disk. When expanding nodes in a different $n$block, any $n$blocks in its duplicate-detection scope that are stored on disk are swapped into RAM.

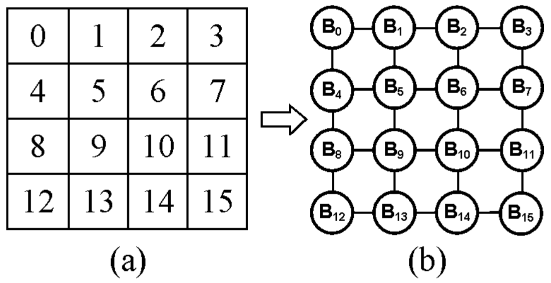

Figure 1: Panel (a) shows all possible positions of the blank for the Fifteen Puzzle. Panel (b) shows an abstract state-space graph that is created by the state-space projection function that considers the position of the "blank" only.

SDD has been shown to be an effective approach to external-memory graph search in solving problems as diverse as the Fifteen Puzzle, the Four-Peg Towers of Hanoi, multiple sequence alignment, and domain-independent STRIPS planning. For domain-independent STRIPS planning, the state-space projection function that is used by SDD is created automatically, and adapted to the search graph of each planning domain (Zhou & Hansen 2006b). SDD has also been used to create external-memory pattern database heuristics (Zhou & Hansen 2005).

Parallel structured duplicate detection

In the rest of the paper, we show that the kind of local structure exploited by SDD to create an efficient external-memory graph-search algorithm, can also be exploited to create an efficient parallel graph-search algorithm. Our approach, which we call *parallel structured duplicate detection* (PSDD), can be used in both shared-memory and distributed-memory parallelization. In this paper, we give the details of a shared-memory parallel graph-search algorithm. Thus, we will speak of using this approach to reduce synchronization overhead rather than communication overhead. At the end of the paper, we briefly discuss how to use this approach in distributed-memory parallel graph search, but we leave the details of this extension for future work.

We start by introducing a concept that plays a central role in PSDD. Let abstract node $y = p(x)$ be the image of node x under a state-space projection function $p(\cdot)$ and let $successors(y)$ be the set of successor abstract nodes of y in the abstract state-space graph.

Definition 1 *The duplicate-detection scopes of nodes x_1 and x_2 are* disjoint *under a state-space projection function $p(\cdot)$, iff the set of successors of x_1's image is disjoint from the set of successors of x_2's image in the abstract graph, i.e., $successors(p(x_1)) \cap successors(p(x_2)) = \emptyset$.*

Theorem 1 *Two nodes cannot share a common successor node if their duplicate-detection scopes are disjoint.*

Although this theorem is obvious, it provides an important guarantee that we can leverage to reduce the number of synchronization operations needed in parallel graph search. The key idea is this: by using PSDD to localize memory references for each processor, we reduce the number of synchronization operations that must be performed by processors competing for the same data, and this can dramatically simplify coordination of concurrent processors.

To enforce data locality, PSDD partitions the set of generated and stored nodes into $n$blocks, one for each abstract node, as in SDD. Because nodes in the same $n$block share the same duplicate-detection scope, both Definition 1 and Theorem 1 generalize to $n$blocks, in addition to holding for individual nodes. We use the concept of disjoint duplicate-detection scopes to parallelize graph search by assigning $n$blocks with disjoint duplicate-detection scopes to different processors. This allows processors to expand nodes in parallel *without* having to synchronize with each other, because it is impossible for one processor to generate a successor node that could also be generated by another processor.

Note that when an $n$block is assigned to a processor for node expansions, the same processor is also given exclusive access to all of the $n$blocks in the duplicate-detection scope of that $n$block. So, we say that the duplicate-detection scope of the assigned $n$block is *occupied* by the processor. This means the processor does not need to worry about other processors competing for the $n$blocks it needs to access while generating successors for the assigned $n$block.

For the example of the Fifteen Puzzle, Figure 2(a) shows an abstract state-space graph with four duplicate-detection scopes (enclosed by dashed lines) that are disjoint. Because these are the duplicate-detection scopes of nodes that map to the abstract nodes B_0, B_3, B_{12}, and B_{15} (drawn in double circles), the corresponding four $n$blocks can be assigned to four processors $P_0 \sim P_3$, and processors can expand nodes in parallel without requiring any synchronization.

PSDD also reduces the complexity of managing concurrent access to critical data structures. As we will show, it only needs a single mutually exclusive (mutex) lock to guard the abstract state-space graph, *and it needs no other locks to synchronize access to shared data, in particular, the Open and Closed lists*. This both simplifies the implementation of the parallel search algorithm and avoids the space overhead for storing multiple mutex locks.[1]

Because SDD assumes the underlying search algorithm expands nodes on a layer-by-layer basis, PSDD perform layer-based synchronization to ensure that all processors work on the same layer, as is also done in (Korf & Schultze 2005; Zhang & Hansen 2006). To determine when all $n$blocks in a layer are expanded, PSDD uses a counter that keeps track of the number of (non-empty) unexpanded $n$blocks in the current layer; each time an $n$block is expanded, the counter is decremented by one. The search proceeds to the next layer when this counter reaches zero. In our implementation, the counter is protected by the same lock that guards the abstract state-space graph, and the processor that decrements it to zero is responsible for moving the search forward to the next layer, including initialization of the counter for the next layer.

Finding disjoint duplicate-detection scopes

To allow a synchronization-free period of node expansions for an $n$block, the search algorithm needs to find duplicate-detection scopes that are disjoint from the ones currently in use (i.e., occupied) by some other processor. Given the one-to-one correspondence between $n$blocks and abstract nodes, this task is reformulated as a problem of counting how many successors of an abstract node are currently in use by other processors, and choosing the ones that have a count of zero. An abstract node is being used by a processor if its corresponding $n$block is either (1) assigned to the processor for node expansions or (2) part of the duplicate-detection scope occupied by the processor. In our implementation, each abstract node stores a counter, denoted σ, that keeps track of the number of successor abstract nodes that are currently in use by other processors. The system also keeps a list of abstract nodes with a σ-value of zero for each layer. Initially, all abstract nodes have their σ-values set to zero, since no processors are using any abstract nodes at the beginning.

Let y be the abstract node that corresponds to an $n$block that has just been selected for parallel node expansions. Let $\sigma(y)$ be the σ-value of y, and let $predecessors(y)$ be the set of predecessor abstract nodes of y in the abstract graph. As soon as y is selected for expansions, it is removed from the list of abstract nodes with a σ-value of zero for the current layer. In addition, the following steps are performed to update the σ-values of the abstract nodes that are affected.

1. $\forall y' \in \textit{predecessors}(y) \wedge y' \neq y$, $\sigma(y') \leftarrow \sigma(y') + 1$
2. $\forall y' \in \textit{successors}(y)$, $\forall y'' \in \textit{predecessors}(y') \wedge y'' \neq y$, $\sigma(y'') \leftarrow \sigma(y'') + 1$

The first step updates the σ-values of all abstract nodes that include y in their duplicate-detection scopes, since y is assigned to a processor for node expansions. The second step updates the σ-values of all abstract nodes that include any of y's successors in their duplicate-detection scopes, since all of y's successors are occupied by the processor.

Once a processor is done expanding nodes in y, it releases the duplicate-detection scope it occupies by performing the same steps, except in the reverse direction (i.e., decreasing instead of increasing the σ-values by one). The reason the σ-value of y is not updated in both steps is to avoid unnecessary re-computation, since the σ-value is always the same (zero) just before and after it is expanded. After the release of its duplicate-detection scope, y is added to the list of abstract nodes with a σ-value of zero for the next layer, which prevents it from being selected again for the current layer. Note that in order to perform these updates, the processor needs to obtain a mutex lock on the abstract graph to avoid data corruption. But since the computation involved in these two steps is inexpensive, each processor only needs to lock the abstract graph for a very short time.

[1] Korf and Schultze (2005) describe an approach to parallel graph search based on delayed duplicate detection that also uses only a single lock, which is for synchronizing access to work queues. But because delayed duplicate detection generates and stores duplicates before later removing them, it does not need to enforce mutual exclusion in hash table lookups in the conventional way, as does PSDD. Korf and Felner (2007) have since shown that an approach to parallel graph search based on delayed duplicate detection can be implemented without any locks. Although PSDD only needs a single lock, it needs at least one lock.

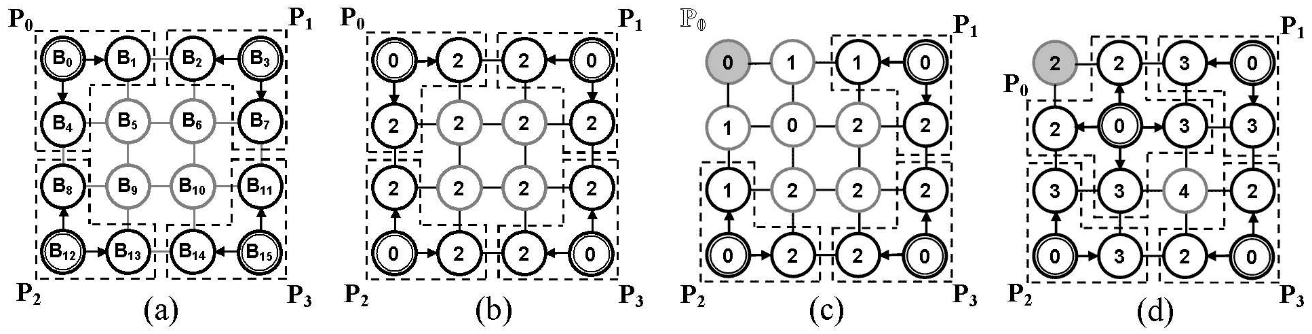
Figure 2: Panel (a) shows that nodes that map to abstract nodes B_0, B_3, B_{12}, and B_{15} have disjoint duplicate-detection scopes, each of which can be assigned to one of the four processors $P_0 \sim P_3$ for parallel node expansions. Panel (b) shows the σ-value of each abstract node for the parallel configuration in (a). Panel (c) shows the σ-value of each abstract node after the release of the duplicate-detection scope occupied by processor P_0 in (b). Panel (d) shows a new duplicate-detection scope occupied by P_0 and the new σ-values. Abstract nodes filled with gray are those that have already been expanded.

Example An example illustrates how the σ-values of abstract nodes allow quick identification of disjoint duplicate-detection scopes. Figure 2(b) shows the σ-value of each abstract node for the disjoint duplicate-detection scopes shown in Figure 2(a). In Figure 2(b), the four abstract nodes with σ-values of zero correspond to the $n$blocks that are currently being expanded in parallel. Those with non-zero σ-values have at least one successor that is in use by some processor, and the count in each abstract node shows how many of its successors are being used. Now suppose processor P_0 has finished expanding all nodes that map to abstract node B_0 and subsequently releases the duplicate-detection scope it had access to. After the release of abstract nodes B_0, B_1, and B_4 by processor P_0, the σ-values of affected abstract nodes are updated as shown in Figure 2(c), and abstract node B_5, identified in Figure 2(a), now has a σ-value of zero. Figure 2(d) shows the updated σ-values after abstract node B_5 is assigned to processor P_0 for node expansions. In order to avoid expanding the same abstract node twice in the same layer of a search graph, PSDD keeps track of which abstract node it has expanded in a layer, as illustrated by the gray-shaded node in Figures 2(c) and 2(d).

Because the number of disjoint duplicate-detection scopes increases with the granularity of the projection function, PSDD allows many processors to work in parallel. If the projection function for the Fifteen Puzzle considers the positions of any two tiles in addition to the position of the blank, for example, the size of the abstract graph increases to $16 \times 15 \times 14 = 3360$ nodes. This is large enough for hundreds of disjoint duplicate-detection scopes, since an abstract node has at most four neighbors in any abstract graph for the Fifteen Puzzle. Of course, the presence of hundreds of disjoint duplicate-detection scopes in the abstract graph does not guarantee that PSDD can use as many processors to expand nodes in parallel. This is possible only if there are states in the original search space that map to all (or most of) these abstract nodes. This may not be the case for easy problems. But as problem size grows, it becomes more likely, and so the number of processors PSDD can use in parallel tends to increase with the hardness of a search problem.

Hierarchical hash table

A conventional graph-search algorithm uses a single hash table to store generated nodes. By contrast, PSDD uses a set of hash tables, one for each non-empty $n$block. For efficiency, PSDD keeps a pool of blank hash tables. A processor can request a hash table for any $n$block that does not yet have one assigned to it. When an $n$block becomes empty, the hash table assigned to it can be returned to the pool.

In PSDD, finding a hash slot for a search node is a two-step process. The first step determines which $n$block the node belongs to. The second step computes the hash slot of the node inside the hash table assigned to the $n$block identified in the first step. This hashing scheme can be viewed as a two-level hierarchical hash table in which the list of $n$blocks is the top-level hash table; the top-level hash function is the state-space projection function of PSDD. The hash table assigned to an $n$block is a second-level hash table indexed by a regular hash function.

This hierarchical organization of the hash table reflects local structure that is exploited to achieve efficient duplicate detection. Because only disjoint duplicate-detection scopes can be assigned to multiple processors, the set of hash tables used by one processor is guaranteed to be disjoint from the set of hash tables used by another processor. As a result, operations on hash tables such as query, insertion, and deletion can be performed simultaneously by multiple processors without any synchronization.

Managing shared memory

In a shared-memory environment, available memory is shared by all processors and this can create a tension between conserving memory and reducing synchronization overhead. In order to save memory, each processor should be allocated just enough memory to store the nodes it generates. But since it is very difficult to accurately predict memory use in graph search, most practical implementations allocate memory on an as-needed basis. To appropriately allocate RAM among all processors, our implementation maintains a central memory pool from which each processor can request memory to store newly-generated nodes. In a multi-threading environment, however, the central memory pool can be a source of contention, since concurrent node-allocation requests can occur frequently.

Thus, PSDD uses a memory-allocation strategy in which each processor (or thread) has a private memory pool. When the private memory pool is exhausted, it taps into the central memory pool for refilling, and there is a *minimum refill size*, m, for each refill request. Let n be the number of processors. The amount of memory (measured in terms of nodes) wasted by this strategy is bounded by $O(m \cdot n)$, which is often a tiny fraction of the total number of nodes stored by a graph-search algorithm, for reasonable values of m and n.

External-memory PSDD

We next consider some implementation issues that must be addressed when integrating our approach to parallel graph search with external-memory graph search, using structured duplicate detection as a common framework.

I/O-efficient order of *n*block expansions

The order in which $n$blocks are expanded can have a big impact on the number of I/O operations needed by external-memory PSDD. A simple and effective heuristic is to expand $n$blocks in order of a breadth-first traversal of the abstract state-space graph. However, there are two issues with applying this heuristic to external-memory PSDD. First, it is not designed for reducing the scope changes from one set of duplicate-detection scopes to another set, as needed by external-memory PSDD. Second, the order of the breadth-first traversal is static, which does not adapt to the nondeterministic search behaviors that are caused by PSDD.

To overcome these issues, we developed a new strategy for selecting the order of $n$block expansions that uses a more direct approach to reducing the number of I/O operations. The idea is to store with each $n$block a *disk-successor counter* that keeps track of the number of successor $n$blocks that are currently stored on disk, since the disk-successor counter of an $n$block corresponds to how many $n$blocks in its duplicate-detection scope need to be read from disk if it is selected for expansion. PSDD also maintains a list of non-empty $n$blocks that are ordered by their disk-successor counters. To select the next $n$block to expand, the algorithm simply picks the $n$block with the minimum disk-successor counter. Since disk-successor counters are integers within a small range (from 0 to the maximum out-degree of the abstract graph), the list can be implemented as an array with constant-time operations.

I/O-efficient strategy for *n*block replacement

Recall that when RAM is full, $n$blocks that do not belong to the duplicate-detection scopes of nodes being expanded can be flushed to disk. Since there are usually multiple "flushable" $n$blocks stored in RAM, PSDD needs a strategy for deciding which subset of these $n$blocks to flush. We call this an *nblock-replacement strategy* because of its similarity to a page-replacement strategy for virtual memory. While SDD can either use an optimal strategy (Zhou & Hansen 2004), or adapt the *least-recently used* (LRU) strategy (Belady 1966) for this purpose, neither strategy is directly applicable to PSDD, for the following reasons.

Problem	SDD	PSDD 1 thread	PSDD 2 threads	PSDD 3 threads
logistics-6	1.5	1.8	1.2	0.9
blocks-14	7.4	7.0	4.7	4.3
satellite-6	125.2	72.3	38.7	26.3
freecell-3	161.9	149.0	75.4	59.5
depots-7	204.4	183.1	100.2	76.5
driverlog-11	234.3	187.0	97.0	68.0
elevator-12	247.0	212.0	103.7	74.9
gripper-8	475.0	456.3	253.3	184.8

Table 1: Comparison of running times (in wall clock seconds) for internal-memory versions of structured duplicate detection (SDD) and parallel structured duplicate detection (PSDD) with 1, 2, and 3 threads.

First, in order to use the optimal strategy, the algorithm needs to know the order in which $n$blocks will be expanded in the future. But this is nondeterministic in PSDD because it may depend on the (relative) speed of each processor. Second, it is difficult (if not impossible) to efficiently adapt the LRU strategy for PSDD because the least-recently used $n$block may not be flushable, if it is assigned to a slower processor. Moreover, the LRU strategy is based on information about the past, and, ideally, a decision about which $n$blocks to remove from RAM should be based on information about the future.

This motivates the development of a new I/O-efficient replacement strategy for PSDD. It decides whether or not to replace an $n$block based on the number of its unvisited (and non-empty) predecessor $n$blocks in a layer, since this reflects the likelihood of needing the $n$block in RAM during expansion of the remaining nodes in the layer. This strategy is fairly robust with respect to the uncertainty of the processor speed, and thus works well for PSDD. Note that the new strategy also works for SDD, which can be viewed as a special case of PSDD for a single processor.

Computational results

We implemented external-memory PSDD with POSIX threads (Nichols, Buttlar, & Farrell 1996) in a domain-independent STRIPS planner that uses as its underlying graph-search algorithm *breadth-first heuristic search* (Zhou & Hansen 2006a). The search algorithm performs regression planning to find optimal sequential plans, guided by the max-pair admissible heuristic (Haslum & Geffner 2000). We tested the planner on eight domains from the biennial planning competition. Experiments were performed on a machine with dual Intel Xeon 2.66 GHz processors, each having 2 cores. The machine has 8 GB of RAM and 4 MB of L2 cache.

Using breadth-first heuristic search, the size of the layer containing the goal is typically very small. For our experiments, this is an advantage because it means that the nondeterministic tie-breaking behavior of PSDD has little or no effect on the total number of expanded nodes, and the number

			SDD			PSDD Secs			
problem	Len	RAM	Disk	Exp	Secs	1 thread	2 threads	3 threads	4 threads
logistics-6	25	10,000	157,146	339,112	33.2	18.8	17.9	17.5	17.2
gripper-7	47	500,000	2,528,691	16,031,844	173.4	123.3	77.4	57.1	46.9
blocks-16	52	5,000,000	2,488,778	18,075,779	214.0	198.0	129.3	114.3	105.9
depots-7	21	5,000,000	6,942,229	16,801,408	228.4	209.5	128.9	95.9	89.0
satellite-6	20	400,000	1,966,875	3,484,031	246.3	227.6	156.5	121.7	116.7
driverlog-11	19	3,000,000	12,745,885	18,606,444	443.9	413.7	316.8	287.3	278.1
freecell-4	26	36,000,000	89,513,363	208,743,830	8,475.0	7,584.6	3,952.5	3,016.6	2,700.9
elevator-15	46	6,500,000	121,238,231	430,804,933	14,378.9	14,300.0	7,129.4	4,911.2	3,891.1

Table 2: Comparison of the external-memory versions of structured duplicate detection (SDD) and parallel structured duplicate detection (PSDD) with 1, 2, 3, and 4 threads. Columns show solution length (Len), peak number of nodes stored in RAM for both SDD and PSDD (RAM), peak number of nodes stored on disk (Disk), number of node expansions (Exp), and running time in wall clock seconds (Secs).

of node expansions is virtually the same for both SDD and PSDD. Therefore the timing results shown in the tables can be straightforwardly used to compare the node-generation speed of SDD and PSDD. Because we are primarily interested in the *relative* speedup of PSDD over SDD, and not absolute running times, we used optimal solution costs as upper bounds in our experiments, and the results shown in the tables are for the last iteration of breadth-first iterative-deepening A*.

In our experiments, we first tested the speed of the parallel algorithm on problems that can fit in RAM. To ensure the accuracy of our timing results, we did not use all four cores, since at least one core needs to be reserved for the OS and other programs running on the same machine. Results are presented in Table 1. For comparison, they include timing results for a sequential algorithm that shares the same code base but uses SDD instead of PSDD. It is interesting that the parallel version based on PSDD is faster even when it uses a single thread (i.e, no parallel search), despite the added overhead for managing threads. Our explanation is that the hierarchical organization of a set of (relatively) small-sized hash tables allows the CPU to be more selective in caching the most relevant part of an otherwise monolithic hash table, and this leads to improved cache performance.

The results in Table 1 show that speedup from parallelization is different for different problems. For small problems such as *logistics-6* and *blocks-14*, the speedup is less than ideal for two reasons. First, the overhead of creating and stopping threads is less cost-effective for small problems. Second, and more importantly, disjoint duplicate-detection scopes are more difficult to find for small problems, preventing multiple processors from working on different parts of the search graph simultaneously. But since this can be easily detected by PSDD, which immediately releases any CPU resources it does not need, PSDD can quickly adapt to the difficulty level of a problem instance, which is useful on systems where the algorithm using PSDD is not the only program running. For the larger problems in Table 1, the speedups are much closer to linear. Note that the search graphs for planning problems have many duplicate paths, and are especially challenging for duplicate detection.

Table 2 compares running times (in wall-clock seconds) for SDD and PSDD with up to 4 threads on problems that do not fit in the amount of RAM given to the algorithms (specified in the column labeled “RAM”). Since the timing results for external-memory PSDD using 4 threads are less negatively affected by the overhead of the OS and other programs running on the same machine (due to I/O parallelism), we include these results to give a better sense of scalability. One similarity between the results comparing external-memory versions of SDD and PSDD and the results comparing internal-memory versions is that our approach appears less effective for small problems, in both cases, but achieves better scalability for large problems. Another similarity is that PSDD using a single thread is more efficient than SDD. Again, improved cache performance partly explains this. But more importantly, it appears that a single thread of PSDD is more efficient than SDD because of the improved techniques for determining the order in which to visit $n$blocks, and the order in which to replace $n$blocks, which are presented in the section on External-memory PSDD.

Because we only used a single disk in our experiments, the external-memory search algorithm quickly becomes more I/O-bound than CPU-bound as the number of threads increases. Using multiple disks, we expect that our results could be substantially improved.

Conclusion and future work

We have introduced a novel approach to parallelizing graph search called parallel structured duplicate detection. The approach leverages the concept of disjoint duplicate-detection scopes to exploit the local structure of a search graph in a way that significantly reduces the overhead for synchronizing access to stored nodes in duplicate detection. A hierarchically-organized hash table supports this approach and requires only a single mutex lock to efficiently resolve all contention. Finally, we described I/O-efficient techniques for integrating parallel graph search with external-memory graph search.

The degree of parallelism allowed by this approach can be increased by increasing the number of disjoint duplicate-detection scopes in the abstract graph. One way to do so is by increasing the granularity of the projection function, as described in this paper. An alternative, or complementary, approach would be to use a strategy of incremental node expansion, called *edge partitioning* (Zhou & Hansen 2007).

Although we focused in this paper on shared-memory parallelization of graph search, we expect a similar approach to be effective for distributed-memory parallelization, and we plan to develop this approach in future work. The key idea is to exploit the local structure of a search graph to partition stored nodes in a way that allows different processors to expand nodes independently, without requiring communication to remove duplicates, or requiring only limited communication. Because communication overhead usually causes more delay than synchronization overhead, parallel structured duplicate detection is likely to result in even greater improvement in a distributed-memory environment.

Acknowledgements We thank the anonymous reviewers for helpful suggestions about how to improve this paper.

References

Belady, L. 1966. A study of replacement algorithms for virtual storage. *IBM Systems Journal* 5:78–101.

Dutt, S., and Mahapatra, N. 1994. Scalable load balancing strategies for parallel A* algorithms. *Journal of Parallel and Distributed Computing* 22(3):488–505.

Grama, A., and Kumar, V. 1999. State of the art in parallel search techniques for discrete optimization problems. *IEEE Transactions on Knowledge and Data Engineering* 11(1):28–35.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Proc. of the 5th International Conference on AI Planning and Scheduling*, 140–149.

Jabbar, S., and Edelkamp, S. 2006. Parallel external directed model checking with linear I/O. In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2006)*, 237–251.

Korf, R., and Felner, A. 2007. Recent progress in heuristic search: A case study of the four-peg towers of Hanoi problem. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, 2334–2329.

Korf, R., and Schultze, P. 2005. Large-scale parallel breadth-first search. In *Proc. of the 20th National Conference on Artificial Intelligence (AAAI-05)*, 1380–1385.

Kumar, V.; Ramesh, K.; and Rao, V. 1988. Parallel best-first search of state-space graphs: A summary of results. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI-88)*, 122–127.

Nichols, B.; Buttlar, D.; and Farrell, J. P. 1996. *PThreads Programming*. O'Reilly.

Niewiadomski, R.; Amaral, J.; and Holte, R. 2006. Sequential and parallel algorithms for frontier A* with delayed duplicate detection. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI-06)*, 1039–1044.

Zhang, Y., and Hansen, E. 2006. Parallel breadth-first heuristic search on a shared-memory architecture. In *Heuristic Search, Memory-Based Heuristics and Their Applications: Papers from the AAAI Workshop*, 33–38. AAAI Press. Technical Report WS-06-08.

Zhou, R., and Hansen, E. 2004. Structured duplicate detection in external-memory graph search. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04)*, 683–688.

Zhou, R., and Hansen, E. 2005. External-memory pattern databases using structured duplicate detection. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-05)*, 1398–1405.

Zhou, R., and Hansen, E. 2006a. Breadth-first heuristic search. *Artificial Intelligence* 170(4-5):385–408.

Zhou, R., and Hansen, E. 2006b. Domain-independent structured duplicate detection. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI-06)*, 1082–1087.

Zhou, R., and Hansen, E. 2007. Edge partitioning in external-memory graph search. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, 2410–2416.