Parallel External Directed Model Checking With Linear I/O

Shahid Jabbar and Stefan Edelkamp

Computer Science Department
University of Dortmund, Dortmund, Germany
{shahid.jabbar, stefan.edelkamp}@cs.uni-dortmund.de

Abstract. In this paper we present Parallel External A*, a parallel variant of external memory directed model checking. As a model scales up, its successors generation becomes complex and, in turn, starts to impact the running time of the model checker. Probings of our external memory model checker IO-HSF-SPIN revealed that in some of the cases about 70% of the whole running time was consumed in the internal processing. Employing a multiprocessor machine or a cluster of workstations, we can distribute the internal working load of the algorithm on multiple processors.
Moreover, assuming a sufficient number of processors and number of open file pointers per process, the I/O complexity is reduced to linear by exploiting a hash-function based state space partition scheme.

1 Introduction

In explicit-state model checking software [3], state descriptors are often so large, so that main memory is often not sufficient for a lossless storage of the set of reachable states during the exploration even if all available reduction techniques, like *symmetry* or *partial-order reduction* [20, 24] have been applied. Besides advanced implicit storage structures for the set of states [19] three different options have been proposed to overcome the internal space limitations for this so-called *state explosion* problem, namely, *directed*, *external* and *parallel* search.

Directed or heuristic search [23] guides the search process into the direction of the goal states, which in model checking safety properties is the set of software errors. The main observation is that using this guidance, the number of explored states needed to establish an error is smaller than with blind search. Moreover, *directed model checking* [29, 7] often reduces the length of the counter-example, which in turn eases the interpretation of the bug.

External search algorithms [26] store and explore the state space via hard disk access. States are flushed to and retrieved from disk. As virtual memory already can exceed main memory capacity, it can result in a slow-down of speed due to excessive page-faults if the algorithm lacks locality. Hence, the major challenge in a good design for an external algorithm is to control the locality of the file access, where block-transfers are in favor to random accesses. Since hashing has a bad reputation for preserving locality, in *external model checking* [27, 11] duplicate

elimination is delayed by applying a subsequent external sorting and scanning phase of the state set to be refined. During the algorithm only a part of the graph can be processed at a time; the remainder is stored on a disk. However, hard disk operations are about a $10^5 - 10^6$ times slower than main memory accesses. More severely, this latency gap rises dramatically. According to recent estimates, technological progress yields about annual rates of 40-60 percent increase in processor speeds, while disk transfers only improve by 7 to 10%. Moreover, the costs for large amount of disk space has considerably decreased. At the time of writing, 500 gigabytes can be obtained at the cost of 300-400 US dollars.

Parallel or distributed search algorithms[1] are designed to solve algorithmic problems by using many processors / computers. An efficient solution can only be obtained, if the organization between the different tasks can be optimized and distributed in a way that the working power is effectively used. *Distributed model checking* [28] tackles with the state explosion problem by profiting from the amount of resources provided by parallel environments. A speedup is expected if the load is distributed uniformly with a low inter-processes communication cost.

In *large-scale parallel breadth-first search* [14], the state space is fully enumerated for increasing depth. Using this approach a complete exploration for the *Fifteen-Puzzle* with 16!/2 states has been executed on six disks using a maximum of 1.4 terabytes of disk storage. In model checking such an algorithm is important to verify safety properties in large state spaces. In [11], the authors presented an external memory directed model checker that utilizes hard disk to store the explored states. It utilizes heuristics estimates to guide the search towards the error state.

In LTL model checking, as models scale up, the density of edges in the combined state space also increases. This in turn, effects the complexity of successors generation. Probing our external directed model checker IO-HSF-SPIN [11] revealed some bottlenecks in its running time. Surprisingly, in a disk-based model checker internal processing such as successor generations and state comparisons were sometimes dominating even the disk access times.

In this paper we present a parallel variant of external memory directed model checking algorithm that improves on our earlier algorithm in two ways. Firstly, the internal workload is divided among different processors that can either be residing on the same machine or on different machines. Secondly, we suggest an improved parallel duplicate detection scheme based on multiple processors and multiple hard disks. We show that under some realistic assumptions, we achieve a number of I/Os that is linear to the explored size of the model.

The paper is structured as follows. First we review *large-scale parallel breadth-first search*, the combined approach of external and parallel breadth-first search.

[1] As it refers to related work, even for this text terminology is not consistent. In AI literature, the term *parallel search* is preferred, while in model checking research, the term *distributed search* is commonly chosen. In theory, parallel algorithms commonly refers to a synchronous scenario (mostly according a fixed architecture), while *distributed algorithms* are preferably used in an asynchronous setting. In this sense, the paper considers the less restricted distributed scenario.

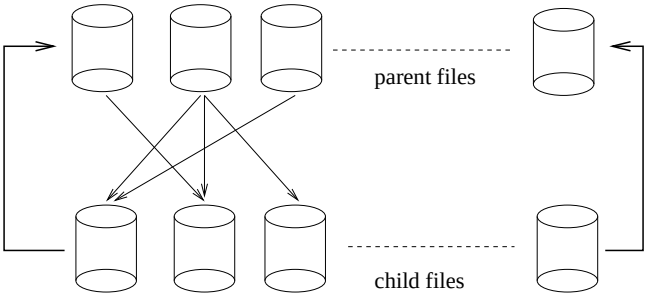

Fig. 1. Externally stored state space with parent and child files.

Then, we turn to directed model checking and the heuristics used in model checking. Next, we recall *External A**, the external version of the A* algorithm that serves as a basis to include heuristics to the search. Afterwards, we propose *Parallel External A** and provide algorithmic details for large-scale parallel A* search. Subsequently, we discuss some of the complexity aspects of the algorithm. Its application to distributed and external model checking domains is considered in the experimental part, where we have successfully extended the state-of-the-art model checker SPIN to include directed, external and parallel search. The paper closes with further links to related work.

2 Large-Scale Parallel Breadth-First Search

In *large-scale parallel breadth-first search* [14], the entire search space is partitioned into different files. The hash address is used to distribute and to locate states in those files. As the considered state spaces like the Fifteen-Puzzle are regular permutation games, each state can be perfectly hashed to a unique index. Since all state spaces are undirected, in order to avoid regenerating explored states, *frontier search* [15] stores, with each node, its used operators in form of a bit-vector in the size of the operator labels available. This allows to distinguish neighboring states that have already been explored from those that have not, and, in turn to omit the list of already explored states.

Hash-based delayed duplicate detection uses two orthogonal hash functions. When a state is explored, its children are written to a particular file based on the first hash value. In cases like the sliding-tile puzzle, the filename correspond to parts of state vector. For space efficiency it is favorable to perfectly hash the rest of the state vector to obtain a compressed representation. The representation as a permutation index can be computed in linear time w.r.t. the length of the vector.

Figure 1 depicts the layered exploration on the external partition of the state space. Even on a single processor, multi-threading is important to maximize the performance of disk-based algorithms. The reason is that a single-threaded

implementation will run until it has to read from or write to disk. At that point it will block until the I/O operation has completed. Moreover hash-based delayed duplicate detection is well-suited to be distributed. Within an iteration, most file expansions and merges can be performed independently.

To realize parallel processing a work queue is maintained, which contains parent files waiting to be expanded, and child files waiting to be merged. At the start of each iteration, the queue is initialized to contain all parent files. Once all the neighbors of a child file are finalized, it is inserted in the queue for merging. Each thread works as follows. It first locks the work queue. Two parent files conflict if they can generate states that hash to the same child file. The algorithm checks whether the first parent file conflicts with any other file currently being expanded. If so, it scans the queue for a parent file with no conflicts. It swaps the position of that file with the one at the head of the queue, grabs the non-conflicting file, unlocks the queue, and expands the file. For each child file it generates, it checks to see if all of its parents have been expanded. If so, it puts the child file to the queue for merging, and then returns to the queue for more work. If there is no more work in the queue, any idle threads wait for the current iteration to complete. At the end of each iteration the work queue is initialized to contain all parent files for the next iteration.

3 Directed Model Checking

Directed model checking [7] incorporates heuristic search algorithms like A* [23] to enhance the bug-finding capability of model checkers, by accelerating the search for errors and finding (near to) minimal counterexamples. In that manner we can mitigate the state explosion problem and the long counterexamples provided by some algorithms like depth-first search, which is often applied in explicit state model checking.

One can distinguish different classes of evaluation functions based on the information they try to exploit. *Property specific* heuristics [7] analyze the error description as the negation of the correctness specification. In some cases the underlying methods are only applicable to special kinds of errors. A heuristic that prioritizes transitions that block a higher number of processes focuses on deadlock detection. In other cases the approaches are applicable to a wider range of errors. For instance, there are heuristics for invariant checking that extract information from the invariant specification and heuristics that base on already given errors states. The second class has been denoted as being *structural* [9], in the sense that source code metrics govern the search. This class includes coverage metrics (such as *branch count*) as well as concurrency measures (such as *thread preference* and *thread interleaving*). Next there is the class of *user heuristics* that inherit guidance from the system designer in form of source annotations, yielding preference and pruning rules for the model checker.

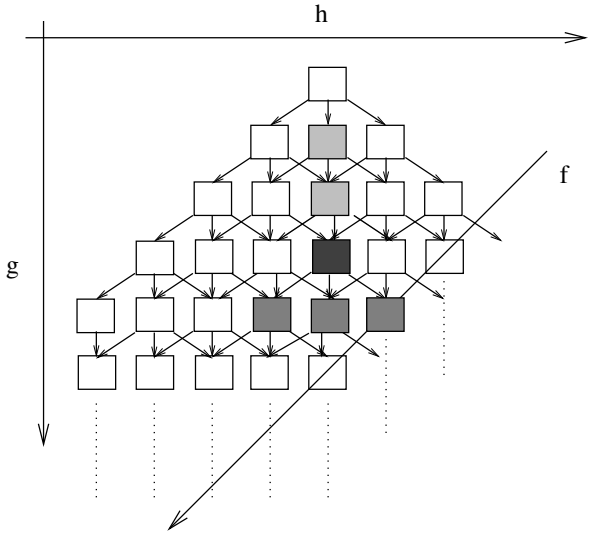

Fig. 2. Exploration in External A*.

4 External A*

*External A** [6] maintains the search horizon on disk. The priority queue data structure is represented as a list of buckets. In the course of the algorithm (cf. Figure 2), each bucket (i, j) will contain all states u with path length $g(u) = i$ and heuristic estimate $h(u) = j$. As similar states have same heuristic estimates, it is easy to restrict duplicate detection to buckets of the same h-value. By an assumed undirected state space problem graph structure, we can restrict aspirants for duplicate detection furthermore. If all duplicates of a state with g-value i are removed with respect to the levels i, $i-1$ and $i-2$, then no duplicate state will remain for the entire search process. For breadth-first-search in explicit graphs, this is in fact the algorithm of [22]. We consider each bucket as a different file that has an individual internal buffer. A bucket is *active* if some of its states are currently expanded or generated. If a buffer becomes full, then it is flushed to disk. The algorithm maintains the two values $g_{\min}$ and $f_{\min}$ to address the correct buckets. The buckets of $f_{\min}$ are traversed for increasing $g_{\min}$-value unless the $g_{\min}$ exceeds $f_{\min}$. Due to the increase of the $g_{\min}$-value in the $f_{\min}$ bucket, a bucket is finalized when all its successors have been generated. Given $f_{\min}$ and $g_{\min}$, the corresponding h-value is determined by $h_{\max} = f_{\min} - g_{\min}$. According to their different h-values, successors are arranged into different horizon lists. Duplicate elimination is delayed.

Since External A* simulates A* and changes only the order of elements to be expanded that have the same f-value, completeness and optimality are inherited from the properties of A*. The I/O complexity for External A* in an implicit unweighted and undirected graph with monotone heuristic is bounded

by $O(sort(|E|) + scan(|V|))$, where $|V|$ and $|E|$ are the number of nodes and edges in the explored subgraph of the state space problem graph, and $scan(n)$ ($sort(n)$) are the number of I/Os needed to externally scan (sort) n elements.

For challenging exploration problems, external algorithms operate in terms of days and weeks. For improving fault-tolerance, we have added a *stop-and-resume* option on top of the algorithm, which allows the continuation of the exploration in case of a user interrupt. An interrupt causes all open buffers to be flushed and the exploration can continue with the active buffer at the additionally stored file pointer position. In this case, we only have to redo at most one state expansion. As duplicates are eliminated, generating a successor twice does not harm the correctness and optimality of the algorithm. As we flush all open buffers when a bucket is finished, in case of severe failures e.g. to the power supply of the computer we have to re-run the exploration for at most one bucket.

I/O Efficient Directed Model Checking [11] applies variants of External A* to the validation of communication protocols. The tool IO-HSF-SPIN accepts large fractions of the Promela input language of the SPIN model checker. The paper extends External A* to weighted, directed graphs, with non-monotone cost functions as apparent in explicit state model checking and studies the scope for delayed duplicate within protocol verification domains.

5 Parallel External A*

The distributed version of External A* *Parallel External A** is based on the observation that the internal work in each individual bucket can be parallelized among different processors. Due to the dynamic allocation of new objects in software model checking, our approach is also compatible with state vectors of varying length. We first discuss our method of disk-based queues to distribute the work load among different processes. Our approach is applicable to both a client-server based environment or a single machine with multiple processors.

5.1 Disk-based Queues

To organize the communication between the processors a working queue is maintained on disk. The working queue contains the requests for exploring parts of a (g, h) bucket together with the part of the file that has to be considered[2]. For improving the efficiency, we assume a distributed environment with one master and several slave processes[3]. Our approach applies to both the cases when each

[2] As processors may have different computational power and processes can dynamically join and leave the exploration, the number of state space parts under consideration do not necessarily have to match the number of processors. By utilizing a queue, one also may expect a processor to access a bucket multiple times. However, for the ease of a first understanding, it is simpler to assume that the jobs are distributed uniformly among the processors.

[3] In our current implementation the *master* is in fact an ordinary process defined as the one that finalized the work for a bucket.

slave has its own hard disk or if they work together on one hard disk residing on the master. Message passing between the master and slave processes is purely done on files, so that all processes can run independently. For our algorithm, master and slave work fully autonomously. We do not use spawning of child processes. Even if slave processes are killed, their work can be re-done by any other idle process that is available.

One file that we call the *expand-queue*, contains all current requests for exploring a state set that is contained in a file. The filename consists of the current g and h value. In case of larger files, file-pointers for processing parts of a file are provided, to allow for better load balancing. There are different strategies to split a file into equi-distance parts or into chunks depending on the number and performance of logged-on slaves. As we want to keep the exploration process distributed, we select the file pointer windows into equidistant parts of a fixed number of C bytes for the states to be expanded. For improved I/O, number C is supposed to divide the system's block size B. As concurrent read operations are allowed for most operating systems, multiple processes reading the same file impose no concurrency conflicts.

The expand-queue is generated by the master process and is initialized with the first block to be expanded. Additionally we maintain a total count on the number of requests, i.e., the size of the queue, and the current count of satisfied requests. Any logged-on slave reads a requests and increases the count once it finishes. During the expansion process, in a subdirectory indexed by the slave's name it generates different files that are indexed by the g and h value of the successor states.

The other queue is the *refine-queue* also generated by the master process once all processes are done. It is organized in a similar fashion as the expand queue and allows slaves to request work. The refine-queue contains filenames that have been generated above, namely the slave-name (that does not have to match with the one of the current process), the block number, and the g and h value. For a suitable processing the master process will move the files from subdirectories indexed by the slave's name to ones that are indexed by the block number. As this is a sequential operation executed by the master thread, we require that changing the file locations is fast in practice, an assumption that is fulfilled in all modern operating systems.

In order to avoid redundant work, each processor eliminates the requests from the queue. Moreover, after finishing the job, it will write an acknowledge to an associated file, so that each process can access the current status of the exploration, and determine if a bucket has been completely explored or sorted.

Since all communication between different processes is done through shared files, proper mechanism for mutual exclusion is necessary. We utilized a rather simple but efficient method to avoid concurrent writes accesses to the files. When ever a process has to write on a shared file, e.g., to the *expand-queue* to deque the request, it issues an operating system move (`mv`) command to rename the file into *<process ID>.expand-queue*, where *process ID* is a unique number that is automatically assigned to every process that enters the pool. If the command

fails, it implies that the file is currently being used by another process. Since the granularity of a kernel-level command is much finer than any other program implemented on top of it, the above technique performed remarkably well.

5.2 Sorting and Merging

For each bucket that is under consideration, we establish four stages in the algorithm. These phases are visualized in Figure 3 (top to bottom). The zig-zag curves visualize the sorting order of the states sequentially stored in the files. The sorting criteria is defined by the state's hash key, which dominates low-level state comparison based on the compressed state descriptor.

In the *exploration stage*, each processor p flushes the successors with a particular g and h value to its own file (g, h, p). Each process has its own hash table and eliminates some duplicates already in main memory. The hash table is based on chaining, with chains sorted along the state comparison function. However, if the output buffer exceeds memory capacity it writes the entire hash table to disk. By the use of the sorting criteria as given above, this can be done using a mere scan of the hash table.

In the *first sorting stage*, each processor sorts its own file. In a serial setting, such sorting has a limited effect on I/O when using external merge-sort afterwards. In a distributed setting, however, we exploit the advantage that the files can be sorted in parallel. Moreover, the number of file pointers needed is restricted by the number of flushed buffers, which is illustrated by the number of peaks in the figure. Based on this restriction, we only need to perform a merge of different sorted buffers - an operation in linear I/O.

In the *distribution stage*, a single processor distributes all states in the pre-sorted files into different files according to the hash value's range. This is a parallel scan with a number of file pointers that is equivalent to the number of files that have been generated. As all input files are pre-sorted this is a mere scan. No all-including file is generated, keeping the individual file sizes small. This is of course a bottleneck to the parallel execution, as all processes have to wait until the distribution stage is completed. However, if we expect the files to be on different hard drives, traffic for file copying is needed anyway.

In the *second sorting stage*, processors sort the files with buffers pre-sorted w.r.t the hash value's range, to find further duplicates. The number of peaks in each individual file is limited by the number of input files (= number of processors), and the number of output files is determined by the selected partitioning of the hash index range. The output of this phase are sorted and partitioned buffers. Using the hash index as the sorting key we establish that the concatenation of files is in fact totally sorted.

6 Complexity

The complexity of external memory algorithm is usually measured in terms of I/Os, which assumes an unlimited amount of disk space. Using the complexity measurement [6] shows that in general it is not possible to exceed the

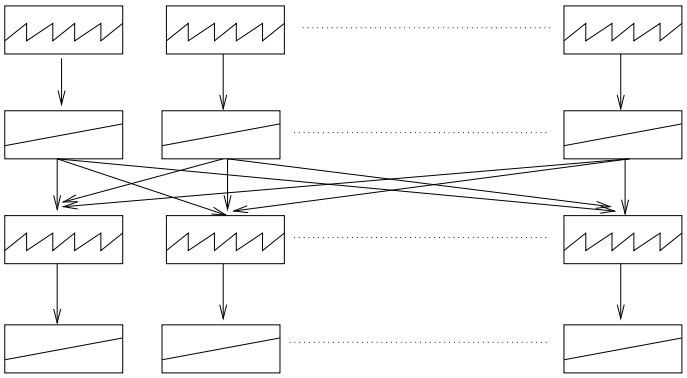

Fig. 3. Stages of bucket expansions in *Parallel External A*.*

external sorting barrier i.e., delayed duplicate detection. The lower bound for the I/O complexity for delayed duplicate bucket elimination in an implicit unweighted and undirected graph A* search with consistent estimates is at least $\Omega(sort(|E|))$, where E is the set of explored edges in the search graph. Fewer I/Os can only be expected if structural state properties can be exploited. We will see, that by assuming a sufficient number of processors and file pointers, the I/O complexity is reduced to linear (i.e. $\Omega(scan(|E|)) = |E|/|B|$ I/Os) by exploiting a hash-function based state space partition scheme. We assume that the hash function provides a uniform partitioning of the state space.

Recall that, the complexity in the external memory model assumes no restriction to the capacity of the external device. In practice, however, we have certain restrictions, e.g on the number of open file pointers per process.

We may, however, assume that the number of processes is smaller than the number of file pointers. Moreover, by the given main memory capacity, we can also assume that the number of flushed buffers (the number of peaks w.r.t. the sorted order of the file) is also smaller than the number of file pointers.

Using this we can achieve in fact a linear number of I/O for delayed duplicate elimination. The proof is rather simple. The number of peaks k in each individual file is bounded either by the number of flushed buffers or by the number of processes, so that a simple scan with k-file pointers suffices to finalize the sorting.

An important observation is that the more processors we invest, the finer the partitioning of the state space, and the smaller the individual file sizes in a partitioned representation. Therefore, a side effect on having more processors at hand is an improvement in I/O performance based on existing hardware resource bounds.

7 Experiments

We have implemented a slightly reduced version of *Parallel External A** in our extension to the model checker SPIN. Our tool, entitled IO-HSF-SPIN and first introduced in [11], is in fact an extension of the model checker HSF-SPIN developed by [18]. Instead of implementing all four stages for the bucket exploration, we restricted to three stages and use one server processor to merge the sorted outcome of the others. A drawback is that all processors have to wait for a finalized merge phase. Moreover, the size of the resulting file is not partitioned and, therefore, larger. Given that the time complexity is in fact hidden during the exploration of states, so far these aspects are not a severe limitation. As all input files are pre-sorted the stated I/O complexity of the algorithm is still linear.

We chose two characteristics protocols for our experiments, the CORBA-GIOP protocol as introduced by [12], and the Optical Telegraph protocol that comes with SPIN distribution. The CORBA-GIOP can be scaled according to two different parameters, the number of servers and the number of clients. We selected three settings: *a*) 3 clients and 2 servers, *b*) 4 clients and 1 server, and *c*) 4 clients and 2 servers. For the Optical Telegraph, we chose one instance with 9 stations. CORBA-GIOP protocol has a longer state vector that puts much load on the I/O. On the other hand, the Optical Telegraph has a smaller state vector but takes a longer time to be computed which puts more load on the internal processing. Moreover, the number of duplicates generated in Optical Telegraph is much more than in CORBA-GIOP.

To evaluate the potential of the algorithm, we tested it on two different (rather small) infrastructures. In the first setting we used two 1 GHz Sun Solaris Workstations equipped with 512 megabyte RAM and a NFS mounted hard disk space. For the second setting we chose a Sun Enterprise System with four 750 MHz processors working with 8 gigabyte RAM and 30 gigabyte shared hard disk space. In both cases, we worked with a single hard disk, so that no form of disk parallelism was exploited. Throughout our experiments the sizes of individual processes remained less than 5% of the total space requirement.

Moreover, we used the system *time* command to calculate the CPU running times. The compiler used is GCC v2.95.3 with default optimizations and `-g` and `-pg` options turned on for debugging and profiling information. In all of the test cases, we searched for the deadlock using number of active processes as the heuristic estimate. We depict all three parameters provided by the system: *real* (the total elapsed time), *user* (total number of CPU-seconds that the process spent in user mode) and *system* (total number of CPU-seconds that the process spent in kernel mode). The speedup in the columns is calculated by dividing the time taken by a serial execution by the time taken by the parallel execution.

In the first set of experiments, the multi-processor machine is used. Table 1 and 2 depict the times[4] for three different runs consisting of single process, 2 processes, and 3 processes. The space requirements by a run of our algorithm

[4] The smallest given CPU time always corresponds to the process that established the error in the protocol first.

is approx. 2.1 GB, 5.2 GB, 21 GB, and 4.3 GB for GIOP 3-2, 4-1, 4-2, and Optical-9, respectively. For GIOP 4-1, we see a gain by a factor of 1.75 for two processors and 2.12 for three processors in the total elapsed time. For Optical Telegraph, this gain went up to 2.41, which was expected due to its complex internal processing.

In actual CPU-time (*user*), we see an almost linear speedup that depicts the uniform distribution of internal workload and, hence, highlighting the potential of the presented approach.

GIOP 3-2	1 process	2 processes		Speedup	3 processes			Speedup
real	25m 59s	17m 30s	17m 29s	1.48	15m 55s	16m 6s	15m 58s	1.64
user	18m 20s	9m 49s	9m 44s	1.89	7m 32s	7m 28s	7m 22s	2.44
system	4m 22s	4m 19s	4m 24s	0.98	4m 45s	4m 37s	4m 55s	0.92
GIOP 4-1	1 process	2 processes		Speedup	3 processes			Speedup
real	73m 10s	41m 42s	41m 38s	1.75	37m 24s	34m 27s	37m 20s	2.12
user	52m 50s	25m 56s	25m 49s	2.04	18m 8s	18m 11s	18m 20s	2.91
system	10m 20s	9m 6s	9m 15s	1.12	9m 22s	9m 8s	9m 0s	1.13
GIOP 4-2	1 process	2 processes		Speedup	3 processes			Speedup
real	269m 9s	165m 25s	165m 25s	1.62	151m 6s	151m 3s	151m 5s	1.78
user	186m 12s	91m 10s	90m 32s	2.04	63m 12s	63m 35s	63m 59s	2.93
system	37m 21s	29m 44s	30m 30s	1.25	30m 19s	30m 14s	29m 50s	1.24

Table 1. CPU time for Parallel External A* in GIOP on a multiprocessor machine.

Optical-9	1 process	2 processes		Speedup	3 processes			Speedup
real	55m 53s	31m 43s	31m 36s	1.76	23m 32s	23m 17s	23m 10s	2.41
user	43m 26s	22m 46s	22m 58s	1.89	15m 20s	14m 24s	14m 25s	3.01
system	5m 47s	4m 43s	4m 18s	1.34	3m 46s	4m 45s	4m 40s	1.22

Table 2. CPU time for Parallel External A* in Optical Telegraph on a multiprocessor machine.

Tables 3 and 4 show our results in the scenario of two machines connected together via NFS. In GIOP 3-2, we observe a small speed-up of a factor of 1.08. In GIOP 4-1, this gain increased to about a factor of 1.3. When tracing this limited gain, we found that the CPUs were not used at full speed. The bottleneck turned out to be the underlying NFS layer that was limiting the disk accesses to only about 5 Megabytes/sec. This bottleneck can be removed by utilizing local hard disk space for successors generation and then sending the files to the file server using secure copy (scp) that allows a transfer rate of 50 Megabytes/sec.

In the Optical Telegraph, we see a bigger reduction of about a factor of 1.41 because of complex but small state vector and more dependency on internal

computation. As in the former setting, the total CPU-seconds consumed by a process (*user*) in 1-process mode is reduced to almost half in 2-processes mode.

GIOP 3-2	1 process	2 processes		Speedup
real	35m 39s	32m 52s	33m 0s	1.08
user	11m 38s	6m 35s	6m 34s	1.76
system	3m 56s	4m 16s	4m 23s	0.91
GIOP 4-1	1 process	2 processes		Speedup
real	100m 27s	76m 38s	76m 39s	1.3
user	31m 6s	15m 52s	15m 31s	1.96
system	8m 59s	8m 30s	8m 36s	1.05

Table 3. CPU time for Parallel External A* in GIOP on two computers and NFS.

Optical-9	1 process	2 processes		Speedup
real	76m 33s	54m 20s	54m 6s	1.41
user	26m 37s	14m 11s	14m 12s	1.87
system	4m 33s	3m 56s	3m 38s	1.26

Table 4. CPU time for Parallel External A* in Optical Telegraph on two computers and NFS.

8 Related Work

There is much work on external search in explicit graphs that are fully specified with its adjacency list on disk. In model checking software the graph are implicit. There is no major difference in the exposition of the algorithm of Munagala and Ranade [22] for explicit and implicit graphs. However, the precomputation and access efforts are by far larger for the explicit graph representation. The breadth-first search algorithm has been improved by [21].

Even for implicit search, the body of literature is rising at a large pace. Edelkamp and Schrödl [8] consider external route-planning graphs that are naturally embedded into the plane. This yields a spatial partitioning that is exploited to trade state exploration count for improved local access. Zhou and Hansen [30] impose a projection function to have buckets to control the expansion process in best-first search. The projection preserves the duplicate scope or locality of the state space graph, so that states that are outside the locality scope do not need to be stored. Korf [13] highlights different options to combine A*, frontier and external search. His proposal is limited as only any two options were compatible. Edelkamp [5] extends the External A* with BDDs to perform a external

symbolic BFS in abstract space, followed by an external symbolic A* search in original space that take the former result as a lower bound to guide the search. Zhou and Hansen [31] propose structure preserving state space projections to have a reduced state space to be controlled on disk. They also propose external construction of pattern databases. The drawback of their approach is that it applies only to the state spaces that have a very regular structure - something that is not available in model checking.

In Stern and Dill's initial paper on external model checking in the Murϕ Verifier variants of external breadth-first search are considered. In Bao and Jones [1], we see another faster variant of Murϕ Verifier with magnetic disk. They propose two techniques: one is based on partitioned hash tables, and the other on chained hash table. They targeted to reduce the delayed duplicate detection time by partitioning the state space that, in turn, diminishes the size of the set to be checked. They claim their technique to be inherently serial having less room for a distributed variant. In the approach of Kristensen and Mailund [16] repeated scans over the search space in a geometric scan-line approach with states that are arranged in the plane wrt. some *progress measure* based on a given partial order. The scan over the entire state space is memory efficient, as it only needs to store the states that participate in the transitions that cross the current scan position. These states are marked visited and the scan over the entire state space is repeated to cope with states that are not reachable with respect to earlier scans. Their dependeny on a good *progress measure* hinders its applicability to model checking in general. They have applied it mainly to Petri nets based model checking where the notion of *time* is used as a *progress measure*.

While some approaches to parallel and distributed model checking are limited to the verification of safety properties [2, 10, 17], other work propose methods for checking liveness properties expressed in linear temporal logic (LTL) [4, 18]. Recall that LTL model checking mainly entails finding accepting cycles in a state space, which is performed with the nested depth-first search algorithm. The correctness of this algorithm depends on the depth-first traversal of the state space. Since depth-first search is inherently sequential [25], additional data structures and synchronization mechanisms have to be added to the algorithm. These requirements can waste the resources offered by the distributed environment. Moreover, formally proving the correctness of the resulting algorithms is not easy. It is possible, however, to avoid these problems by using partition functions that localize cycles within equivalence classes. The above described methods for defining partitions can used for this purpose, leading to a distributed algorithm that performs the second search in parallel. The main limitation factor is that scalability and load distribution depend on the structure of the model and the specification.

Brim et. al. [4] discusses one such approach where the SPIN model checker has been extended to perform nested depth-first search in a distributed manner. They proposed to maintain a dependency structure for all the accepting states visited. The nested parts for these accepting states are then started as separate procedures based on the order dictated by the dependeny structure. Lluch-

Laufente [18] improves on the idea of nested depth-first search. The proposed idea is to divide the state space in strongly connected components by exploiting the structure of the *never claim* automaton of the specification property. The nested search is then restricted only to the corresponding component. If during exploration, a node that belongs to some other component is encountered, it is inserted in the *visited* list of its corresponding component. Unfortunately, there is little or almost no room externalize the above two approaches. Depth-first search lacks *locality* and hence not suited to be externalize. Stern and Dill [28] propose a parallel version of the Murϕ model checker. They also use a scheme based on run-time partitioning of the state space and assigning different partitions to different processors. The partitioning is done by using a universal hash function that uniformly distributes the newly generated states.

9 Conclusion

Enhancing directed model checking is essential to improve error detection in software. The paper contributes the first study of combining external, directed and parallel search to mitigate the state-explosion problem in model checking. We have shown a successful approach to extend the external A* exploration in a distributed environment, as apparent in multi-processor machines and workstation clusters. Exploration and delayed duplicate detection are parallelized without concurrent write access, which is often not available.

Error trails provided by depth-first search exploration engines are often exceedingly lengthy. Employed with a lower-bound heuristic, the proposed algorithm yields counter-examples of optimal length, and is, therefore, an important step to ease error comprehension for the programmer / software designer. Under reasonable assumptions on the number of file pointers per process, the number of I/Os is linear in the size of the model, by means the external work of exploring the model matches the complexity of scanning it.

The approach is implemented on top of the model checker IO-HSF-SPIN and the savings for the single disk model are very encouraging. We see an almost linear speedup in the CPU-time and significant gain in the total elapsed time. Compared to the potential of external search, the models that we have looked at are considerably small. In near future, we expect to implement the multiple-disk version of the algorithm as mentioned in this text. To conduct empirical observations for external exploration algorithm is a time-consuming task. In very large state spaces algorithms can run for weeks. For example the complete exploration of the Fifteen Puzzle consumed more than three weeks. Given more time, we expect larger models to be analyzed. We also expect further fine-tuning to increase the speed-up that we have obtained. Moreover, the approach presented is particular to model checking only, and can be applied to other areas where searching in a large state space is required.

Acknowledgments: The authors wish to thank Mathias Weiss for helpful discussions and technical support that made running of the presented experiments possible.

References

1. T. Bao and M. Jones. Time-efficient model checking with magnetic disks. In *Tools and Algorithms for the Construction and Analysis of Systems(TACAS)*, pages 526–540, 2005.
2. S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In *Formal methods in Computer-Aided Design (FMCAD)*, pages 390–404, 2000.
3. B. Bérard, A. F. M. Bidoit, F. Laroussine, A. Petit, L. Petrucci, P. Schoenebelen, and P. McKenzie. *Systems and Software Verification.* Springer, 2001.
4. L. Brim, J. Barnat, and J.Stribnra. Distributed LTL model-checking in SPIN. In *Workshop on Software Model Checking (SPIN)*, pages 200–216, 2001.
5. S. Edelkamp. External symbolic pattern databases. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 51–60, 2005.
6. S. Edelkamp, S. Jabbar, and S. Schrödl. External A*. In *German Conference on Artificial Intelligence(KI)*, volume 3238, pages 226–240, 2004.
7. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology*, 5(2–3):247–267, 2004.
8. S. Edelkamp and S. Schrödl. Localizing A*. In *National Conference on Artificial Intelligence (AAAI)*, pages 885–890, 2000.
9. A. Groce and W. Visser. Heuristic model checking for java programs. *International Journal on Software Tools for Technology Transfer*, 6(4), 2004.
10. T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *International Conference on Computer-Aided Verification (CAV)*, pages 20–35, 2000.
11. S. Jabbar and S. Edelkamp. I/O efficient directed model checking. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 3385, pages 313–329, 2005.
12. M. Kamel and S. Leue. Formalization and validation of the General Inter-ORB Protocol (GIOP) using PROMELA and SPIN. *International Journal on Software Tools for Technology Transfer*, 2(4):394–409, 2000.
13. R. Korf. Best-first frontier search with delayed duplicate detection. In *National Conference on Artificial Intelligence (AAAI)*, pages 650–657, 2004.
14. R. E. Korf and P. Schultze. Large-scale parallel breadth-first search. In *National Conference on Artificial Intelligence (AAAI)*, pages 1380–1385, 2005.
15. R. E. Korf and W. Zhang. Divide-and-conquer frontier search applied to optimal sequence allignment. In *National Conference on Artificial Intelligence (AAAI)*, pages 910–916, 2000.
16. L. M. Kristensen and T. Mailund. Path finding with the sweep-line method using external storage. In *International Conference on Formal Engineering Methods (ICFEM)*, pages 319–337, 2003.
17. F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Workshop on Software Model Checking (SPIN)*, 1999.
18. A. Lluch-Lafuente. *Directed Search for the Verification of Communication Protocols.* PhD thesis, University of Freiburg, 2003.
19. K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Press, 1993.
20. K. L. McMillan. Symmetry and model checking. In M. K. Inan and R. P. Kurshan, editors, *Verification of Digital and Hybrid Systems*, pages 117–137. Springer-Verlag, 1998.

21. K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *European Symposium on Algorithms (ESA)*, pages 723–735, 2002.
22. K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Symposium on Discrete Algorithms (SODA)*, pages 87–88, 2001.
23. J. Pearl. *Heuristics*. Addison-Wesley, 1985.
24. D. A. Peled. Ten years of partial order reduction. In *Computer-Aided Verification (CAV)*, volume 1427, pages 17–28, 1998.
25. J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20:229–234, 1985.
26. P. Sanders, U. Meyer, and J. F. Sibeyn. *Algorithms for Memory Hierarchies*. Springer, 2002.
27. U. Stern and D. Dill. Using magnetic disk instead of main memory in the murphi verifier. In *International Conference on Computer Aided Verification (CAV)*, pages 172–183, 1998.
28. U. Stern and D. L. Dill. Parallelizing the Murphi verifier. In *International Conference on Computer-Aided Verification (CAV)*, pages 256–278, 1997.
29. C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *Conference on Design Automation (DAC)*, pages 599–604, 1998.
30. R. Zhou and E. Hansen. Structured duplicate detection in external-memory graph search. In *National Conference on Artificial Intelligence (AAAI)*, 2004. 683–689.
31. R. Zhou and E. Hansen. External-memory pattern databases using delayed duplicate detection. In *National Conference on Artificial Intelligence (AAAI)*, pages 1398–1405, 2005.