Efficient Large-Scale Model Checking

Kees Verstoep
versto@cs.vu.nl
VU University, Amsterdam, The Netherlands

Joint work with:
Henri Bal bal@cs.vu.nl VU, Amsterdam, NL
Jiří Barnat, Luboš Brim {barnat,brim}@fi.muni.cz
Masaryk University, Brno, Czech Republic

IPDPS 2009
Rome, Italy
Outline

- Context:
  - Collaboration of VU University (High Performance Distributed Computing) and Masaryk U., Brno (DiVinE model checker)
  - DAS-3/StarPlane: grid for Computer Science research

- Large-scale model checking with DiVinE
  - Optimizations applied, to scale well up to 256 CPU cores
  - Performance of large-scale models on 1 DAS-3 cluster
  - Performance on 4 clusters of wide-area DAS-3

- Lessons learned
Some history

- VU Computer Systems has long history in high-performance distributed computing
  - DAS “computer science grids” at VU, UvA, Delft, Leiden
  - DAS-3 uses 10G optical networks: StarPlane
- Can **efficiently** recompute complete search space of board game Awari on wide-area DAS-3 (CCGrid’08)
  - Provided communication is properly **optimized**
  - Needs 10G StarPlane due to network requirements
- Hunch: communication pattern is much like one for distributed model checking (PDMC’08, Dagstuhl’08)
DAS-3

272 nodes (AMD Opterons)
792 cores
1TB memory
LAN:
  - Myrinet 10G
  - Gigabit Ethernet
WAN:
  - 20-40 Gb/s OPN
Heterogeneous:
  - 2.2-2.6 GHz
  - Single/dual-core
Delft no Myrinet
Model Checking basics

- First construct an abstract **model** of a system or protocol to be analyzed, and instantiate it into a concrete instance
  - components are partially-**nondeterministic** state machines, specifying the transitions that can occur in the system
- In addition, specify **properties** that should hold in a state or sets of states
  - can be simple invariants or assertions, but can also use a temporal logic: “if A holds in a state, **always eventually** B should hold”
- Model checking tool constructs the entire **state space** consisting of **all possible interleavings** of transitions in the state machines, checking the system properties while space is being built (**on-the-fly**) or afterwards
Distributed Model Checking

- **State Explosion** problem: even for tiny models the state space can be **huge**, forcing the user to restrict the concrete model to artificially small dimensions

- Attractive solution: use **distributed memory** on a cluster or grid, also (much) improving response time
  - Distributed algorithms introduce overheads, so not trivial

- We used the open source “DiVinE” model checker
  - Base strategy is breath-first search, being well parallelizable
  - To efficiently analyze temporal logic properties, need algorithm to search for **accepting cycle** in state graph
  - Several distributed algorithms: we look at OWCTY and MAP
  - Thus far only evaluated on a small (20 node) cluster
Algorithm 1: OWCTY (Topological Sort)

Idea:

- Directed graph can be topologically-sorted iff it is acyclic
- Remove states that cannot lie on an accepting cycle
- States on accepting cycle must be reachable from some accepting state and have at least one immediate predecessor

Realization:

- Parallel removal procedures: REACHABILITY & ELIMINATE
- Repeated application of removal procedures until no state can be removed
- Non-empty graph indicates presence of accepting cycle
Algorithm 2: MAP  
(Max. Accepting Predecessors)

Idea:

- If a reachable accepting state is its own predecessor: reachable accepting cycle
- Computation of all accepting predecessors too expensive: compute only maximal one
- If an accepting state is its own maximal accepting predecessor, it lies on an accepting cycle

Realization:

- Propagate max. accepting predecessors (MAPs)
- If a state is propagated to itself: accepting cycle found
- Remove MAPs that are outside a cycle, and repeat until there are accepting states
- MAPs propagation can be done in parallel
Distributed graph traversal

while (!synchronized()) {
    if (((state = waiting.dequeue()) != NULL) {
        state.work();
        for (tr = state.succs(); tr != NULL; tr = tr.next()) {
            tr.work();
            newstate = tr.target();
            dest = newstate.hash();
            if (dest == this_cpu) waiting.queue(newstate);
            else send_work(dest, newstate);
        }
    } else idle();
    process_messages(&waiting);
}

- Induced traffic pattern: irregular all-to-all, but typically evenly spread due to hashing
- Sends are all asynchronous
- Need to frequently poll for pending messages
DiVinE on DAS-3

- Examined large benchmarks and realistic models needing over 100 GB memory:
  - Five DVE models from BEEM model checking database
  - Two realistic Promela/SPIN models (using NIPS plugin)
- Compare MAP and OWCTY checking LTL properties
- Experiments:
  - 1 cluster, 10 Gb/s Myrinet
  - 4 clusters, Myri-10G + 10 Gb/s light paths
- Up to 256 cores (64*4-core hosts) in total, 4GB/host
Optimizations applied

- Improve user-level timer management (TIMER)
  - gettimeofday() system call is fast in Linux, but not free
- Auto-tune receive rate (RATE)
  - Avoid unnecessary polls, dynamically tuning polling rate
  - Arrival rate is algorithm-, phase-, and platform-dependent!
- Prioritize I/O tasks (PRIO)
  - Only do time-critical things in the critical path
- Optimize message flushing (FLUSH)
  - Flush when running out of work and during syncs, but gently
- Pre-establish network connections (PRESYNC)
  - Some of the required N^2 TCP connections may be delayed by ongoing traffic, causing huge amount of buffering
Impact of optimizations

- Graph is for Anderson.8/OWCTY with 256 cores
- Simple TIMER optimization was vital for scalability
- FLUSH & RATE optimizations also show large impact
- Note: not all optimizations are independent
  - PRIIO itself has less effect if RATE is already applied
- PRESYNC not shown: big impact, but only for grid
Scalability improvements

- Medium-size problem: can compare machine scaling
- Performance improvements up to 50%
- After optimizations, also efficient for multi-cores
  - Independent of core/node configuration (up to 4 core/node)
# Efficiency of MAP & OWCTY

Indication of parallel efficiency using medium-size model (Anderson.6; sequential run on host with 16GB)

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Total cores</th>
<th>Time MAP</th>
<th>Time OWCTY</th>
<th>Eff MAP</th>
<th>Eff. OWCTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>956.8</td>
<td>628.8</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>73.9</td>
<td>42.5</td>
<td>81%</td>
<td>92%</td>
</tr>
<tr>
<td>16</td>
<td>32</td>
<td>39.4</td>
<td>22.5</td>
<td>76%</td>
<td>87%</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>20.6</td>
<td>11.4</td>
<td>73%</td>
<td>86%</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>19.5</td>
<td>10.9</td>
<td>77%</td>
<td>90%</td>
</tr>
<tr>
<td>64</td>
<td>128</td>
<td>10.8</td>
<td>6.0</td>
<td>69%</td>
<td>82%</td>
</tr>
<tr>
<td>64</td>
<td>256</td>
<td>7.4</td>
<td>4.3</td>
<td>51%</td>
<td>57%</td>
</tr>
</tbody>
</table>
Impact on communication

- Data rates are MByte/s sent (and received) per core
  - Cumulative throughput: $128 \times 29$ MByte/s = 30 Gbit/s (!)
- MAP/OWCTY iterations easily identified: during first (largest) bump, the entire state graph is constructed
- Optimized running times $\Leftrightarrow$ higher data rates
  - For MAP, data rate is consistent over runtime
  - for OWCTY, first phase is more data intensive than the rest
# Large-scale models used

<table>
<thead>
<tr>
<th>Model</th>
<th>Description</th>
<th>Space (GB)</th>
<th>States *10^6</th>
<th>Trans. *10^6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anderson</td>
<td>Mutual exclusion</td>
<td>144.7</td>
<td>864</td>
<td>6210</td>
</tr>
<tr>
<td>Elevator</td>
<td>Elevator Controller, instance 11 and 13</td>
<td>123.8</td>
<td>576</td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>370.1</td>
<td>1638</td>
<td>5732</td>
</tr>
<tr>
<td>Publish</td>
<td>Groupware protocol</td>
<td>209.7</td>
<td>1242</td>
<td>5714</td>
</tr>
<tr>
<td>AT</td>
<td>Mutual exclusion</td>
<td>245.0</td>
<td>1519</td>
<td>7033</td>
</tr>
<tr>
<td>Le Lann</td>
<td>Leader election</td>
<td>&gt;320</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>GIOP</td>
<td>CORBA protocol</td>
<td>203.8</td>
<td>277</td>
<td>2767</td>
</tr>
<tr>
<td>Lunar</td>
<td>Ad-hoc routing</td>
<td>186.6</td>
<td>249</td>
<td>1267</td>
</tr>
</tbody>
</table>
Scalability of consistent models (1)

- Similar MAP/OWCTY performance
  - Due to small number of MAP/OWCTY iterations
- Both show good scalability
Scalability of consistent models (2)

- OWCTY here clearly outperforms MAP
  - Due to larger number of MAP iterations
  - Same happens for Lunar-2 (same basic model as Lunar-1, only with different LTL property to check)
- But again: both show good scalability
Scalability of inconsistent models

- Same pattern for other inconsistent models
- OWCTY needs to generate entire state space first
  - Is scalable, but can still take significant time
- MAP works **on-the-fly**, and can often find counter example in a matter of seconds
DiVinE on DAS-3/StarPlane grid

- Grid configuration allows analysis of larger problems due to larger amount of (distributed) memory
- We compare a 10G cluster with a lightpath-based 10G grid
  - See paper by Jason Maassen et al. at HPGC09@IPDPS09
  - 1G WAN is insufficient, given the cumulative data volumes
  - DAS-3 clusters used are relatively homogeneous: only up to ~15% differences in clock speed
- Used 2 cores per node to maintain balance (some clusters only have 2-core compute nodes, not 4-core)
Cluster/Grid performance

- Increasingly large instances of Elevator (DVE) model
- Instance 13 no longer fits on single DAS-3/VU cluster
- For all problems, grid/cluster performance quite close!
  - due to consistent use of asynchronous communication
  - and plenty (multi-10G) wide-area network bandwidth
- Latency insensitivity recently confirmed in practice
  - 0.7->7.0 ms latency via Hamburg: similar performance!
**Impact of TCP overhead**

- TCP impact depends on model-specific induced message rate
- When scaling up, TCP version is hurt by increasing polling costs due to larger number of endpoints
- For many realistic models, overhead is acceptable
Many parallels between DiVinE and Awari

- Random state distribution for good load balancing, at the cost of network bandwidth
- Asynchronous communication patterns
- Similar data rates (10-30 MByte/s per core, almost non-stop)
- Similarity in optimizations applied, but now generalized (e.g., ad-hoc polling optimization vs. self-tuning to traffic rate)

Some differences:

- States in Awari *much* more compressed (2 bits/state!)
- Much simpler to find alternative (even realistic and useful) model checking problems than suitable other games.
Lessons learned

- **Efficient Large-Scale Model Checking** indeed possible with DiVinE, both on clusters and grids, given fast network
  - Need suitable distributed algorithms that may not be theoretically optimal, but quite scalable
    - both MAP and OWCTY fit this requirement
  - Using latency-tolerant, *asynchronous* communication is key
  - When scaling up, expect to spend time on optimizations
    - As shown, can be essential to obtain good efficiency
    - Optimizing peak throughput is not always most important
    - Especially look at *host processing overhead* for communication, in both MPI and the run time system
Future work

- Tunable *state compression*
  - Handle still larger, industry scale problems (e.g., UniPro)
  - Also allows reducing network load when needed
- Deal with *heterogeneous* machines and networks
  - Need application-level flow control
- Look into *many-core* platforms
  - Current single-threaded/MPI approach is fine for 4-core
- Use *on-demand* 10G links via StarPlane
  - “allocate network” same as compute nodes
- Look into a *Java/Ibis*-based distributed model checker (VU Univ. grid programming environment)
Acknowledgments

- People:
  - Brno group: DiVinE creators
  - Michael Weber (UTwente): NIPS; SPIN models
  - Cees de Laat (UvA): StarPlane

- Funding:
  - DAS-3: NWO/NCF, Virtual Laboratory for e-Science (VL-e), ASCI, MultiMediaN
  - StarPlane: NWO, SURFnet (lightpaths and equipment)

Download

http://divine.fi.muni.cz
Extra
Solving Awari

- Solved by John Romein [IEEE Computer, Oct. 2003]
  - Computed on a Myrinet cluster (DAS-2/VU)
- Recently used wide-area DAS-3 [CCGrid, May 2008]
- Determined score for 889,063,398,406 positions
- Game is a draw

Andy Tanenbaum: 
``You just ruined a perfectly fine 3500 year old game”
Parallel retrograde analysis

- Work backwards: simplest boards first
- Partition state space over compute nodes
  - Random distribution (hashing), good load balance
  - Special iterative algorithm to fit every game state in 2 bits (!)
- Repeatedly send jobs/results to siblings/parents
  - Asynchronously, combined into bulk transfers
- Extremely communication intensive:
  - Irregular all-to-all communication pattern
  - On DAS-2/VU, 1 Petabit in 51 hours
Impact of Awari grid optimizations

- Scalable synchronization algorithms
  - Tree algorithm for barrier and termination detection (30%)
  - Better flushing strategy in termination phases (45%)
- Assure asynchronous communication
  - Improve MPI_Isend descriptor recycling (15%)
- Reduce host overhead
  - Tune polling rate to message arrival rate (5%)
- Optimize grain size per network (LAN/WAN)
  - Use larger messages, trade-off with load-imbalance (5%)
- Note: optimization order influences relative impacts
Optimized Awari grid performance

- Optimizations improved grid performance by **50%**
- Largest gains **not** in peak-throughput phases!
- Grid version now only 15% slower than Cluster/TCP
  - Despite huge amount of communication (14.8 billion messages for 48-stone database)
  - Remaining difference partly due to heterogeneity