Parallel and Distributed Methods in Verification

Rhishikesh Limaye

EE 219C Spring 2007 Class Presentation

Outline

- Model checking
  - Eddy
  - Grumberg05
- SAT
  - Feldman05

Basic problem / Parallel approach

- State space exploration
- Divide the state space into chunks. Multiple processes search them concurrently.
- Issues
  - Load balancing – how to divide the state space evenly
    - "bounding" prunes search tree at various depths.
  - Computation-communication balance

About parallel platforms

- Parallel = shared memory
  - E.g. – multicore processor with a memory
  - Programmed using Pthreads, OpenMP
- Distributed = message passing
  - Multiple nodes connected through network (Ethernet or some specialized networks)
  - Programmed using MPI – Message Passing Interface.

Parallel and Distributed Model Checking

- We will see two implementations – Eddy, Grumberg05
- Both do reachability analysis using BFS
  - Eddy – explicit state
  - Grumberg05 – symbolic (BDDs)
- Basic algorithm:
  - A state is assigned its *home node*, which owns it.
  - Each node iterates as:
    1. Perform one step of BFS
    2. Exchange newly discovered states according to their ownership.

Synchronous implementation

- Synchronous iteration:
  1. Perform BFS step
  2. Wait for others
  3. Exchange states
- Problems:
  - Fast processes idle
  - Communication overhead
    - Exchanging small messages is bad.
  - Synchronization overhead
    - Higher for larger number of processes.
    - Thus, an obstacle in spawning as many workers as possible.
  - Thus, less adaptive to problem and underlying parallel platform.
  - Problem for large, heterogeneous clusters.

Asynchronous Implementation

- Two levels of concurrency:
  - Within each node, computation-communication overlap
    - BFS continues as the communication happens in background
  - No synchronization between nodes at the end of every iteration.
- *Potentially* better use of available parallelism
  - Have to tune the ratio of computation / communication
- Both, Eddy and Grumberg05, are asynchronous.

Eddy

- Eddy_Murphi:
  - For Murphi modeling language
  - Safety property checker
  - Explicit-state breadth-first search
  - Parallel and distributed (Pthreads + MPI)
- Unique points:
  - Simple design – making it modular across platforms / modeling languages
  - Efficient – at least the reported results are very good, but several potential issues unaddressed.

Eddy – Design

- Partition function – each state is mapped to a node.
  - The paper doesn't give details of the function, and claims that their work is orthogonal to the choice of function.
- Each node has two threads:
  - Worker
  - Communicator
  - This design:
    - overlaps computation and communication
    - makes it easy to adapt to different modeling language (e.g. Promela)

Eddy – worker thread

- Much like sequential BFS with
  - BFS_Queue for to-be-processed states
  - Hash table for visited states
- Differences:
  - Hash table only stores states owned by this node. Non-owned states are given to the communicator to send to their owner nodes.
  - BFS_Queue is filled in two ways:
    1. Owned states discovered during BFS
    2. States received by communicator from other nodes.
  - Worker sleeps (instead of terminating) when BFS_Queue is empty.

Optimizing communication

- Simple model for distributed platforms:

  Cost of message = latency + (size of message) * (throughput).
- Desirable to send large messages
  - Communicator waits for LineSize states to collect before sending them.
  - But very large message – takes time to fill up, causing the receiver to idle
- Multiple (NumLines) buffers of size LineSize:
  - Communicator and worker can work in parallel
- Have to tune NumLines, LineSize to achieve performance.

Termination

- When property fails:
  - Node that discovers error state broadcasts termination signal
  - All nodes listen for termination signal
- When property passes:
  - All nodes idle – no BFS, no pending communication.
  - This is detected by attempting to form a token ring:
    - Token = {#sends, #receives}
    - If a node is idle, it adds it's send/receive counts to the token received, and forwards it.
    - If token returns back with #sends = #receives, then all nodes are idle and no messages are in flight.

Results

- Linear speed-up for several protocol examples:
  - Murphi_time / Eddy_Murphi_time = number of nodes.
- Possible to verify very large protocol (FLASH) having 3 X 10^9 states, which was impossible in Murphi

Grumberg05

- Distributed (MPI)
- For hardware verification
  - Implemented on top of Forecast by Intel.
- Asynchronous
- BDD-based
  - Partitioning of state space is done using window functions:
    - $w_1, w_2, \ldots, w_k$ – complete and disjoint partition of the Boolean space.

Types of processes

- Workers
  - perform BFS
- Coordinators:
  - exch_coord:
    - stores list of active and free workers
    - stores window functions of all workers
    - is notified of every split and merge
    - handles distributed termination
  - small_coord:
    - merges underutilized workers
  - pool_mgr:
    - maintains the pool of free workers

Communication

- Complicated because of asynchrony + dynamic workload management:
  - Send some states to their owner, and that owner might have split and given that part of state space to another worker.
- Uses distributed forwarding mechanism:
  - A worker forwards the non-owned BDDs in the received message to their correct owner.
- Central coordinator keeps track of splits and ownership of states.

Communication

- Send operation: from P_i to P_j:
  1. P_i queries exch_coord the window, w_j, of P_j
  2. BDD message is (T, w): T = BDD for $N \cap w_j$, $w = w_j$. (where N = newly discovered states)
- Receive operation: at P_j:

  If received message is (T, w), and w_j is window of P_j,
  1. Keep $T \cap w \cap w_j$
  2. For every k, If $w \cap w_k \neq \Phi$, forward $(T, w \cap w_k)$ to P_k.

Dynamic workload management

- Workload splitting is done when:
  - Memory limit exceeded: this can happen during two operations:
    - image computation
    - receiving states
  - Adaptive early splitting: occurs based on:
    - Progress of computation of a worker -- split if it hasn't split for a long time.
    - And availability of free workers -- split if too many workers idle

    Good because:
    - Exploits parallelism.
    - BDD sizes remain small.
    - Scalable with respect to number of nodes.
- Also, merging of work of underutilized workers.

Termination detection

- Two-phase algorithm:
  1. Phase 1:
     1. A worker sends *want_term* signal to the coordinator.
     2. The coordinator waits for *want_term* from all workers that were ever active.
  2. Phase 2:
     1. The coordinator sends regret query to all the workers that sent *want_term*.
     2. Workers either reply *want_term* or *regret_term* meaning they don't want to be terminated.
  3. If coordinator receives all *want_term*s and no *regret_term*s, then terminate.
- Worker sends *want_term* when:
  - No local work pending
    - Local fixed point
    - No received messages pending.
  - And, all send operations are complete
    - This is detected by making the receiver acknowledge every received message.

Results

- Comparison between:
  - Forecast – sequential
  - Forecast-D – distributed, synchronous
  - Forecast-AD – distributed, asynchronous
- Forecast-AD is 1-10X faster
  - But speed-up is not linear with number of workers.
- Forecast-D performs slower than Forecast for a few test cases.
- Distributed versions can solve larger test cases.

Parallel SAT – Feldman05

- Complete backtrack-search parallel algorithm
  - Has most of the innovations done in sequential SAT: watched literals, conflict analysis, non-chronological backtracking
- Platform: shared memory system (multithreaded multicore CPU).
- Actually degrades performance!

Feldman05 – Algorithm

- Based on DPLL:
  - Guiding path – stack of partial assignments to variables
  - Open variable: a variable along the guiding path, whose alternative value is not yet explored.
  - Closed variable: a variable along the guiding path, whose both possible values have been explored.
- How threads pick new tasks:
  - One thread explores one guiding path
  - Another free thread picks up an open variable on first thread's guiding path, copies the part of guiding path preceding that variable, and starts exploration along the alternate value of the open variable.

Feldman05 – Algorithm

- Which open variable to pick?
  - Heuristic of choosing the topmost variable
    - Why? – will lead to greater chunk of search space and hence size of the new task is not too small.
- Global, shared list of all open variables:
  - Typically, number of open variables is large compared to number of available threads. Hence, each thread only adds the topmost open variable to the global list.
- This dynamic search space partitioning keeps all threads busy.

Feldman05 – Algorithm

- Conflict clauses:
  - Global, shared list of conflict clauses.
  - When a thread hits a conflict, it adds conflict clause to the list.
  - Threads keep checking the list for new clauses and initialize them in their own context.
- In all, two shared data structures:
  - List of open variables
  - List of conflict clauses

Performance results

- Platforms:
  - 1, 2, 4 CPU systems, two having Hyperthreading.
- Performance varies a lot between different runs of the same problem on same system:
  - Unpredictability of multithreaded environment + imbalance of search space
- Performance is actually worse than sequential SAT.
  - And goes consistently worse with:
    - Increasing number of threads
    - Increasing number of CPUs – worse degradation on 2-CPU, HT-enabled system.

Performance analysis

- Tracked counts of processor-monitoring events
- Ratios degrade significantly just from single thread to 2 threads:
  1. L2 M-state lines allocated / DMRs (10X degradation)
  2. L2 cache request misses / DMRs (6X degradation)
  3. L2 M-state lines evicted / DMRs
  4. External bus cycles / clock ticks
  5. Instructions decoded / clock ticks
- 3, 4, 5 seem to be consequences of 1, 2.

Reasons for poor cache performance

- Reasons that they conjecture for poor cache performance:
  - Why more number of cache lines allocated?
    - Auxiliary data structures of DPLL are duplicated in threads. Hence, greater memory requirement.
  - Why more cache misses?
    - More cache lines allocated
    - No correlation of memory accesses of two different threads.
- Sequential SAT implementations are optimized for cache performance.