PARALLEL BINARY DECISION DIAGRAMS

Evan Bergeron, Kevin Zheng

May 9, 2016

Carnegie Mellon University

We implemented a parallel binary decision diagram library, focusing on spatial locality and cache coherence.

We used BFS tree traversal to maximize spatial locality and reduce communication overhead.

Binary decision diagrams (BDDs) are directed graphs that represent boolean functions. Once constructed, these graphs provide constant time equivalence checking. Unfortunately, constructing these graphs can be costly.

MAIN DATA STRUCTURES

A collection of lockfree hash tables that double as memory allocators and directed graphs.

A fine-grained-locking hash table with versioning, yielding a constant time delete_all operation.

APPROACH

DFS

Initial serial implementation DFS'd on the graph using the if-then-else normal form operation as described in [2].

We wanted to focus on getting BFS to work for high degrees of data parallelism.

A lossy memoization cache is shared between workers to avoid duplicate work.

Additionally, the DAG and unique table are merged as described in 2 to reduce memory footprint.

THE QUEST FOR SPATIAL LOCALITY

Our initial unique table used separate chaining with linked lists in each bucket. After decided that we wanted to focus on memory locality, we switched a linear-probing, open-addressing scheme.

In a parallel context, this hash table must be able to be read and written to concurrently. Our final implementation is lockfree, heavily based on 11.

NODE MANAGERS

To improve spatial locality, we implemented node managers, as described in `3, 4`.

Separate arrays are used for each variable id.

A NEW STRUCT DEFINITION

Similar to 3, we inline the variable in the “pointer” we pass around by value. This avoids unnecessary memory dereferences.

A NEW STRUCT DEFINITION

```
struct bdd_ptr_packed {
  uint16_t varid;
  uint32_t idx;
} __attribute__((packed));

struct bdd {
  bdd_ptr_packed lo;    // 6 bytes
  bdd_ptr_packed hi;    // 6 bytes
  uint16_t varid;       // 2 bytes
  uint16_t refcount;    // 2 bytes
};
```

CAVEAT

In our current setup, this struct definition prevents us from having a dynamically-sized node manager. An additional layer of indirection would need to be added to allow this, something we did not get around to implementing.

IT'S PRETTY SMALL, THOUGH

16bytes, which lets us we a compare and swap.

BFS

We have an expand and reduce phase. We use an ITE formulation of BFS, similar to 8.

Lets us reuse code from DFS and if need be, switch between the two.

WE REUSE THE EXPAND AND REDUCE QUEUE

We only saw this approach in a couple of papers. A lot of the literature has thread-local expand queues and a global reduce queue.

Because of this reuse, the expand queue is shared between workers.

THE REQUEST TABLE

Two parts: a dynamically resizing, lockfree array and a fine-grained-locking, versioned hash table.

We use atomic_test_and_set primitives to implement spin locks - this avoids being descheduled by the OS.

Maps ite triples to a result node. Using the ite formulation forces the keys to be pretty large, forcing us to use fine-grained-locking in leu of compare and swaps.

REUSE ACROSS CALLS

This is what saved us. Each BFS call needs a fresh request table. It's far too expensive to reallocate the table each time.

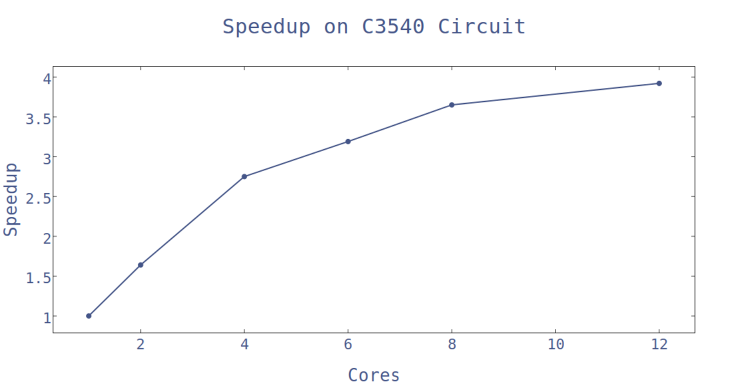
We introduce a version counter. To delete_all, simply increment the version by one. Table entries with obsolete versions are treated as empty.

RESULTS

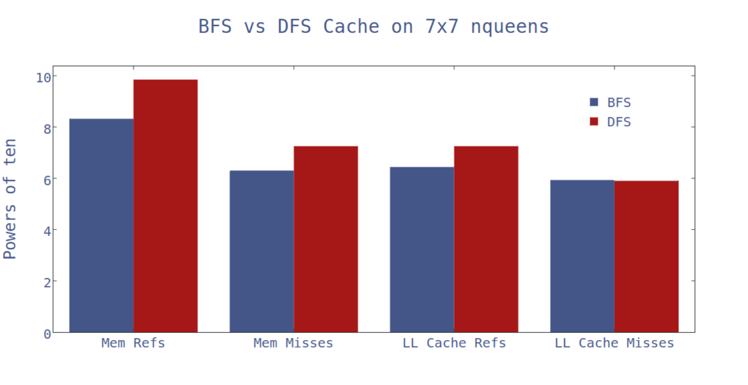
Overall, pretty promising. Our raw wall time is a good bit slower than most state-of-the-art implementations.

We’re within an order of magnitude of BuDDy on small examples, though.

Our speedup is competitive with published results by O'Hallaron.



GREATLY REDUCED NUMBER OF MEMORY ACCESSES



TAKEAWAYS

- First order of business is allow node manager resizing.
- Handful of sequential optimizations - variable reordering, standard triples, complement edges.
- Hybrid DFS/BFS