Low-Connectivity State Space Exploration using Swarm Model Checking on the GPU

Leonard Techel

August 12, 2021

Table of Contents

Motivation

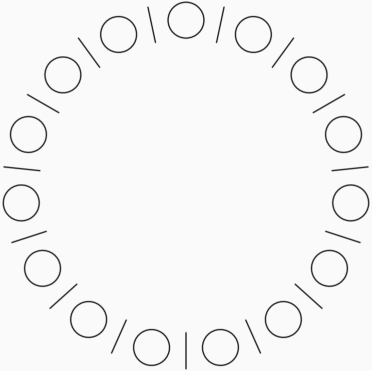

Figure 1: Dining Philosophers, $N = 15$

Motivation 2/4: Dining Philosophers

- 4 States
- Can only start eating when both forks are picked up
- **Goal:** Verify that there is never the case where all philosophers have picked up only the left fork, all waiting for each other
- How to do that algorithmically?

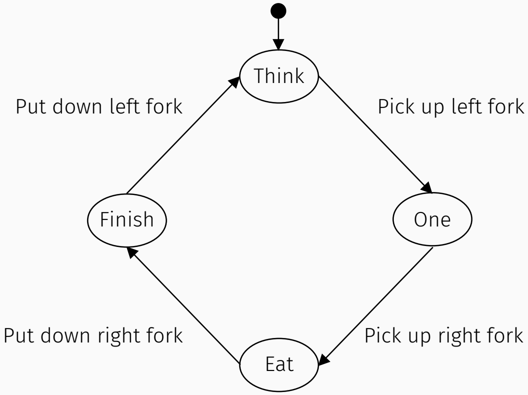

Figure 2: A dining philosopher's state machine

Motivation 3/4: Verification

```
while there are unvisited states do
    mark state as visited
    if state violates spec then
        report path to state
```

Figure 3: State Space Exploration Loop

Motivation 4/4: Observations

- We've invented a *model checker*
- We have $3^{15} - 1 = 14.348.906$ states to check
- Each state of the system needs at least $15 \cdot 2 + 15 = 45$ bit
- **Problem 1:** State Space Explosion
- **Problem 2:** State Space Exploration Loop gets slow quickly
- What can we do?

Model Checking, Swarm Verification

Model Checking 1/3

- Formal verification method
- The model checker finds out whether a state machine *models* a specification
- Two branches: Explicit-State Model Checking and Symbolic Model Checking
- How does it work?

Model Checking 2/3: State Space Exploration Loop

```
while there are unvisited states do
    mark state as visited
    if state violates spec then
        report path to state
```

- Violating states are reported
- Typical algorithms: BFS / DFS
- Typical implementation: Single-Threaded
- How to make it faster?

Model Checking 3/3: Parallelized Exploration Loop

- Breadth-First Search can be parallelized
- **Problem:** Approaches use shared memory
- Scaling beyond a few thread: Communication overhead
- How can we scale onto massively parallel architectures, e.g. GPUs?

Swarm Verification 1/2

From the paper *Swarm Verification* by G. J. Holzmann et al. [2]

- New approach on parallelized model checking
- **Idea:** Split state space exploration into small, independent tasks
- Tasks are called Verification Tests (VTs)
- Each VT only covers a subset of the state space
- **Result:** VTs can be massively parallelized on heterogeneous architectures
- How is the state space exploration split up?

Swarm Verification 2/2: Diversification Techniques

Diversification techniques:

- **Statistically independent hash functions**
- Reversing search order
- Randomizing order of nondeterministic choice
- ...

Does it perform good?

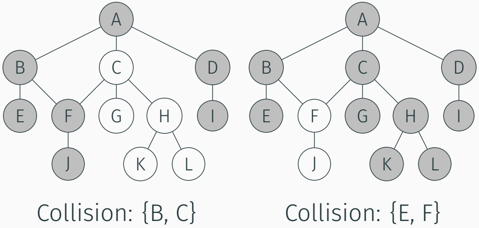

Figure 4: State pruning using hash collisions

Low-Connectivity Models

Low-Connectivity Models 1/3

- Performance measure: # of VTs to achieve nearly 100% state space coverage
- When does it perform good?
  - Waypoints model
    - 8 processes, each in control of 4 bits of a 32 bit int
    - On each transition, a process toggles one random bit
    - 2^{32} = over 4 billion states
    - 100 randomly distributed violations (waypoints)
  - Models with $< 100\%$ hash table utilization
- When does it perform less?
  - Low-Connectivity models

A model has *low connectivity* if at least one of the following properties is satisfied:

- **Generally Linear**: The average # of edges per state is close to 2: One inbound, one outbound
- **Bottleneck Structures**: A single state or group of states other than the initial state that needs to be passed to reach most of the state space

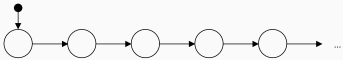

Figure 5: Example of a generally linear graph

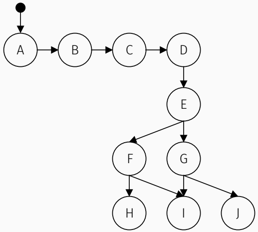

Figure 6: Example of a bottleneck structure

Low-Connectivity Models 3/3: What to do?

Goal: Maximize the state space coverage while minimizing the amount of VTs to do so.

1. How can we (automatically) classify a model as low-connectivity?
2. How can we improve the state space exploration of such models?

How can Swarm Verification be implemented?

Implementation: CUDA, Grapple Model Checker

Grapple Model Checker

From the paper *Swarm model checking on the GPU* by R. DeFrancisco et al. [1]

- A framework for parallel Swarm Verification model checking on the GPU using CUDA
- Why GPUs? — GPUs are massively parallel, widely available and cheap
- What is CUDA?

CUDA 1/3

- CUDA: NVIDIA's proprietary GPU programming framework
- Code is written in C/C++
- Automatic massive scalability
- How does the programming model work?

CUDA 2/3: Programming model

- Write a *kernel*
- Kernel execution: Define the amount of *threads* and *blocks*
- Each thread in a block executes the kernel in SIMT
- Each thread should work on a separate piece of memory
- GPU maps the blocks onto its *streaming multiprocessors*
- Code example?

```
__global__ void VecAdd(float *A, float *B, float *C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main(){
    ...
    VecAdd<<<1, N>>>(A, B, C);
}
```

Figure 7: Add two arrays A, B of length N in CUDA

How to do Swarm Verification on this architecture?

- Each VT's state space exploration runs in a parallel BFS
- Verification Tests: CUDA Blocks
- Worker in a VT: CUDA Thread
- Can it actually be implemented?

My implementation 1/2

- *It seems to work!*
- Currently only implements the *Waypoints* benchmark model
- Hash table diversification seems not to be too good
- Philosophers model is on its way

50.000 runs, each with 250 VTs. Memory leak killed it after $\sim$ 16.000 runs

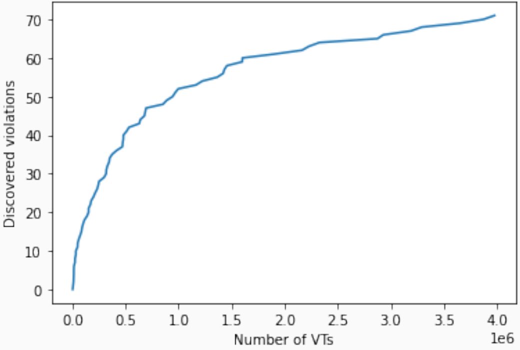

Figure 8: Number of discovered violations in relation to number of executed VTs

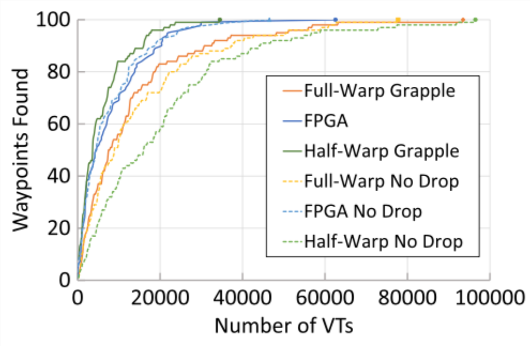

Figure 9: Discovered waypoints in relation to executed VTs in the paper

What’s next?

What’s next?

- Find out how the hash table collisions can be controlled better
- Study low-connectivity models more in-depth: Try out the different approaches from the paper (PDS, process-PDS, scatter-PDS, alternating between search strategies, …?)

References

- Richard DeFrancisco et al. “Swarm model checking on the GPU”. In: *International Journal on Software Tools for Technology Transfer* 22.5 (Oct. 2020), pp. 583–599. ISSN: 1433-2787. DOI: 10.1007/s10009-020-00576-x.
- Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. “Swarm Verification”. In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. 2008, pp. 1–6. DOI: 10.1109/ASE.2008.9.