Thread-Parallel Vampire

the *really* dark side of theorem proving

Michael Rawson, Giles Reger

Vampire?

Vampire

- A fully-automatic theorem prover for classical first-order logic with equality.
  - Plus “theories”, induction, higher-order logic
- Not really just one theorem prover, but a *collection* of techniques
  - Parameterised by options
- Glued together by *strategies* and *schedules*
- Mature, large, and successful
- Mostly C++
- Integrates Z3, MiniSAT
- BSD 3-Clause Licence
- https://vprover.github.io/

Core Vampire Technology

- Preprocessing for efficient clausal form
  - May introduce additional symbols as names for subformulas
- Saturation loop
  - Compute closure of initial clauses under some inference system using best-first search
- Superposition/Resolution - Term Orderings
  - Restrict inferences to producing `smaller' clauses (without losing completeness)
- Redundancy Elimination
  - Remove clauses that aren't needed (without losing completeness)
- Clause splitting using SAT/SMT solvers (AVATAR)
  - Control search space by selecting consistent sub-problems

Opportunities and Challenges for Parallelism!

Strategies in Vampire

- **Proof often found quickly (seconds), or not at all**
- Therefore, stop after a while and try a different *strategy*
- Precompute *schedule* of strategies
- If you have multiple cores, use them
- Unreasonably effective for first-order logic
- Different schedules available for different settings

Parallelism in Vampire

- Vampire is already parallel!
- A problem is loaded and left (mostly) unmodified
- For each strategy:
  - Fork entire process, duplicating (CoW) system state
  - In new process:
    - Preprocess problem according to strategy
    - Try and prove problem according to strategy
    - Exit strategy, leaving original state clean

Thread-Parallel Vampire

Motivation

- Want to communicate, learn from other strategies (past or present)
- Can send or share e.g. clauses, simplifications, ground information
- Need to exchange information somehow
- Not especially convenient/efficient with disjoint-memory *processes*...

Threading Model

- Share code, data between *threads* of execution
- Allows convenient communicating by *sharing memory*
  - Rather than sharing memory by communicating?
- Portable (ish) as of C++11
- So...just use threads?

No

- Vampire is a large, complex codebase
- Components interact (unsynchronised) in non-obvious ways
- Implicit invariants
- Global, mutable state
- Performance critical for success
- Bugs hard to find even normally
  - May subtly affect soundness, completeness
- Bugs almost certainly present even beforehand!
- Processes (!) receive signals to e.g. enforce time limits
- Widespread use of static temporaries to avoid allocations
- ...

Actually yes

- Tooling can help find bugs
- Language features help where they can
- Core data structures in Vampire are frequently immutable
- Not everything has to be shared
- Coarse locks frequently acceptable
- Hubris

Approach

- Could try and squint/static-analyse/whiteboard our way out of this
  - Not at all feasible: too big, too hard, too scary
  - Static analysis nonetheless useful
- Could write a new theorem prover!
  - Not unreasonable for e.g. SAT
  - First-order/saturation more involved, sadly
  - Safer languages? Reasoning-as-a-toolkit? Maybe for the next edition of Vampire…
- Reality: **just add threads, see what breaks**
- A lot of things break, often very quietly and far from the issue
- Tooling: *ThreadSanitiser*

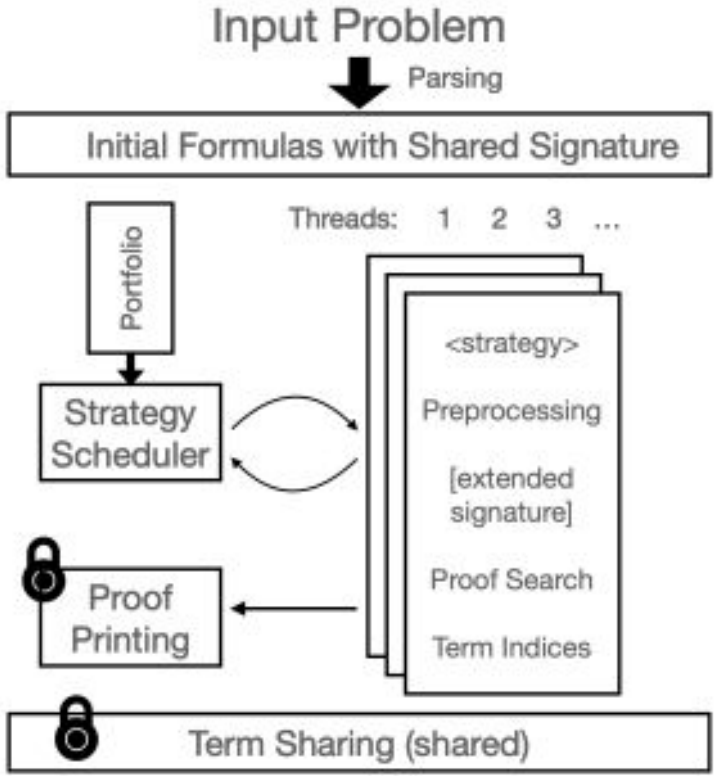
```
x is 0
==================
WARNING: ThreadSanitizer: data race (pid=16500)
  Write of size 4 at 0x000001551f4c by thread T2:
    #0 set_x(int) test/basic-data-race.cpp:11:5 (basic-data-race+0x4bc654)
    #1 thread2() test/basic-data-race.cpp:19:3 (basic-data-race+0x4bc6c6)
    #2 void std::__invoke_impl<void, void (*)()>(...) include/c++/8/bits/invoke.h:60:14 (basic-data-race+0x4bd39d)
    [...]

  Previous read of size 4 at 0x000001551f4c by thread T1:
    #0 print_x() test/basic-data-race.cpp:7:27 (basic-data-race+0x4bc5c8)
    #1 thread1() test/basic-data-race.cpp:15:3 (basic-data-race+0x4bc685)
    #2 void std::__invoke_impl<void, void (*)()>(...) include/c++/8/bits/invoke.h:60:14 (basic-data-race+0x4bd39d)
    [...]

  Location is global 'x' of size 4 at 0x000001551f4c (basic-data-race+0x000001551f4c)

  Thread T2 (tid=16503, running) created by main thread at:
    #0 pthread_create <null> (basic-data-race+0x427e96)
    #1 std::thread::_M_start_thread(std::unique_ptr<...> >, void (*)()) <null> (libstdc++.so.6+0xbd994)
    #2 main test/basic-data-race.cpp:24:15 (basic-data-race+0x4bc70c)

  Thread T1 (tid=16502, finished) created by main thread at:
    #0 pthread_create <null> (basic-data-race+0x427e96)
    #1 std::thread::_M_start_thread(std::unique_ptr<...> >, void (*)()) <null> (libstdc++.so.6+0xbd994)
    #2 main test/basic-data-race.cpp:23:15 (basic-data-race+0x4bc6f7)

SUMMARY: ThreadSanitizer: data race test/basic-data-race.cpp:11:5 in set_x(int)
==================
ThreadSanitizer: reported 1 warnings
```

Fixing It

- Diagnose root cause
- Usually, something is shared that shouldn’t be
- In order of preference:
  - Use an atomic (but only if it’s e.g. a counter, freshener)
  - Make it thread_local (thank you, C++11!)
  - Coarse lock (if you don’t break *external* invariants)
  - Finer locks (if you also don’t break *internal* invariants)
  - Actually fix it properly (ugh)

Architecture

- Share:
  - **Problem symbols**, which occur in
  - **Terms** - useful for
  - **Shared Information** - our motivator!
- Also lock proof output
- **Don’t share** anything else!



Retrospective: multithreaded Vampire

- Multithreaded Vampire (mostly?) works
- Non-trivial effort!
- Difficult to balance performance, correctness, clarity
- Most important: decide how to segregate data **and enforce it**

Communicating Strategies

Persistent Grounding

- Idea: different strategies derive different facts
- Ground everything we derive and stick it in a SAT solver
- Keep it there as strategies come and go
- Occasionally solve, UNSAT means we’re done. :-)
- Not especially interesting, but provides supporting evidence for...

CAPS: the Collaborative Architecture for Proof Search

- AVATAR: the Advanced Vampire Architecture for Theories and Resolution
  - Idiosyncratic combination of saturation, clause splitting and SAT solving
  - Central incremental SAT solver “organises proof search”
  - Allows refuting certain *parts* of the search space
  - Very effective!
  - Usually per-strategy
- Idea: share the SAT solver between threads?
- One strategy might refute a certain part faster than others
- Pick up where another left off
- Future Work - currently working, if buggy

Wrap Up

- Converting an existing system to be multithreaded hard, but not impossible
- Allows interesting new proof techniques, heuristics
- Plenty of room for engineering questions:
  - How do you make a shared term index fast?
  - What’s the fastest way to maintain shared perfect term sharing?
  - Can we reuse introduced symbols (Skolems, definitions…) from other attempts?

Questions