PReach: A real-world, parallel program in Erlang

Brad Bingham
binghamb@cs.ubc.ca

University of British Columbia, Canada

October 24, 2013

CPSC 418

Outline

Terminology *(Stolen from wikipedia)*

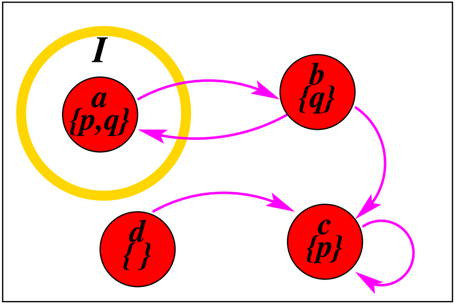
Kripke Structure ("system"): A 4-tuple (S, I, R, L) where

- S is a (finite) set of states,
- $I \subseteq S$ are the initial states,
- $R \subseteq S \times S$ is the transition relation,
- $L : S \to 2^{AP}$ is the labelling function – where AP is a set of atomic propositions (boolean variables)

Reachable states: The set of all states $s \in S$ for which there is a path from some $s_{init} \in I$ through R to s

Example: *(also stolen from wikipedia)*

- $AP = \{p, q\}$
- $S = \{a, b, c, d\}$
- $I = \{a\}$
- $R = \{(a, b), (b, a), (b, c), (c, c), (d, c)\}$
- $L = \{(a, \{p, q\}), (b, \{q\}), (c, \{p\}), (d, \{\})\}$
- $Reachable = \{a, b, c\}$
- $Bad \equiv \neg p \wedge \neg q$



More Terminology

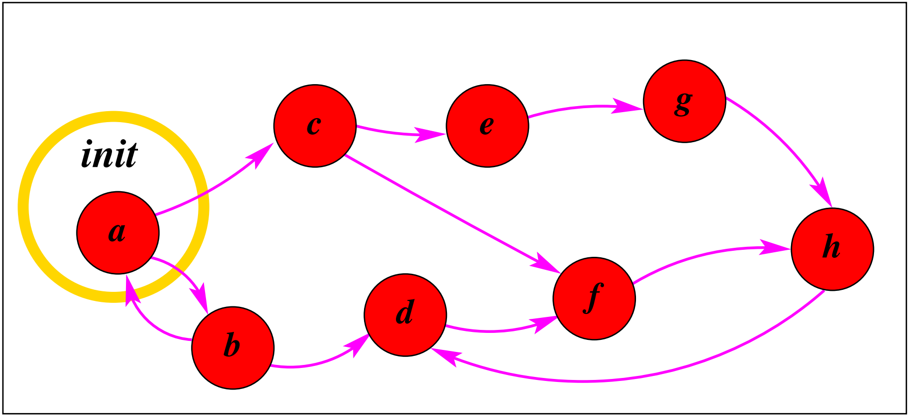
- Model Checking (MC): An automatic technique for checking if a system adheres to a specification, given by a formula expressed in some logic (e.g. CTL, LTL, CTL*, etc)
  - The simplest specification is safety, i.e. "is there a reachable *Bad* state?"
  - *Bad* is a predicate over *AP*
- Explicit-State Model Checking: A model checking algorithm that represents each reachable state distinctly in memory
  - A brute-force approach to MC
  - Alternative to explicit-state MC is symbolic MC, where sets of states are represented by a formula over *AP*, (i.e., BDDs, Interpolants, IC3).

Murφ

- A language for describing hardware systems and an associated explicit-state model checker (for safety properties)
- Murφ system has 4 parts:
  1. variables (think booleans or enumerated types, describing AP),
  2. initial states (a predicate over the variables describing I),
  3. guarded commands (of the form $g \Rightarrow a$, where g is a "guard" and a is an update action, describing R).
  4. invariants (a predicate for *Bad* states).
- Model checking a Murφ system has 3 possible outcomes: pass, fail with counter example, or run out of memory

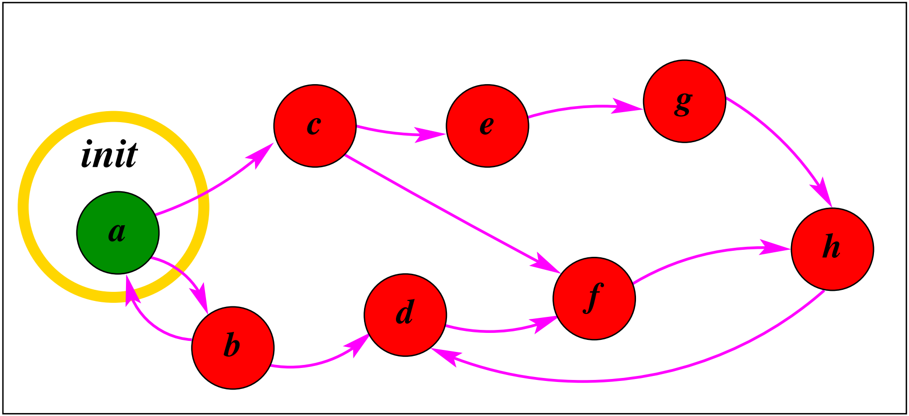
Explicit-state MC by BFS

- Set of visited states **V** $= \emptyset$
- Queue of expanded states **WQ** $= []$



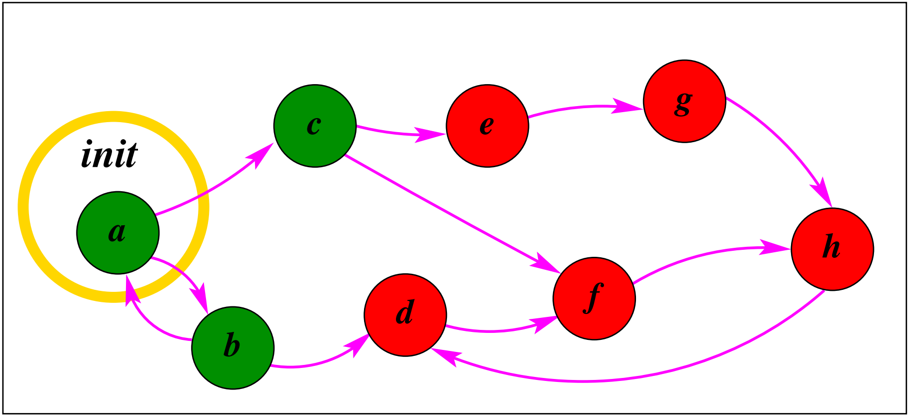
Explicit-state MC by BFS

- Set of visited states **V** $= \{a\}$
- Queue of expanded states **WQ** $= [a]$



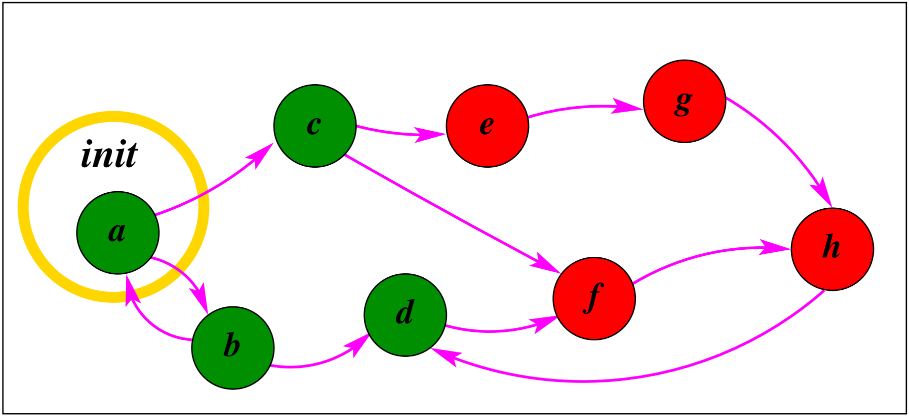
Explicit-state MC by BFS

- Set of visited states **V** $= \{a, b, c\}$
- Queue of expanded states **WQ** $= [b, c]$



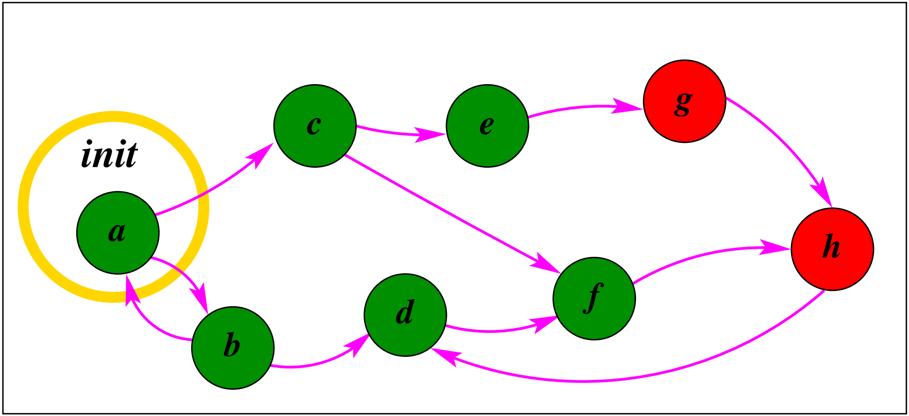
Explicit-state MC by BFS

- Set of visited states **V** $= \{a, b, c, d\}$
- Queue of expanded states **WQ** $= [c, d]$



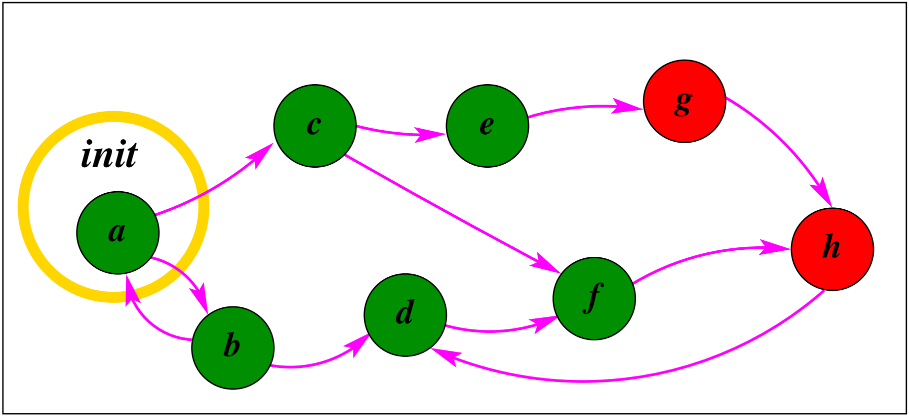
Explicit-state MC by BFS

- Set of visited states **V** $= \{a, b, c, d, e, f\}$
- Queue of expanded states **WQ** $= [d, e, f]$



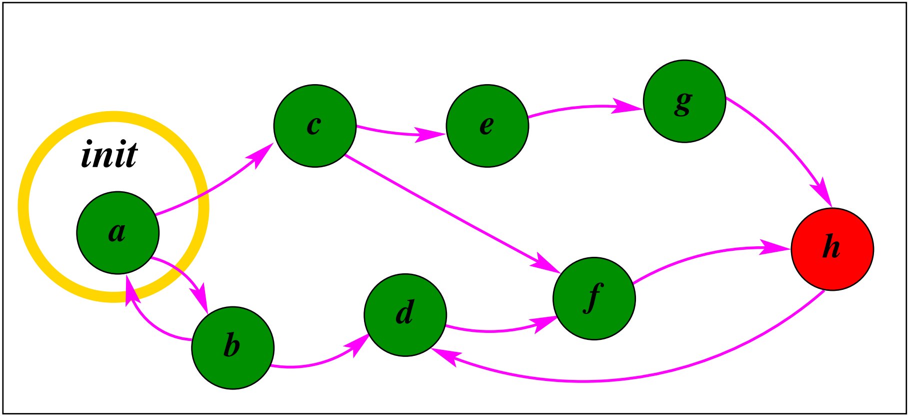
Explicit-state MC by BFS

- Set of visited states **V** $= \{a, b, c, d, e, f\}$
- Queue of expanded states **WQ** $= [e, f]$



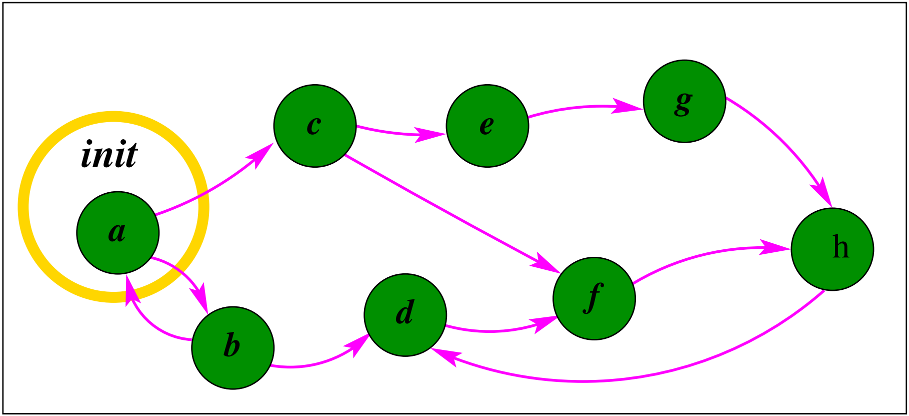
Explicit-state MC by BFS

- Set of visited states **V** $= \{a, b, c, d, e, f, g\}$
- Queue of expanded states **WQ** $= [f, g]$



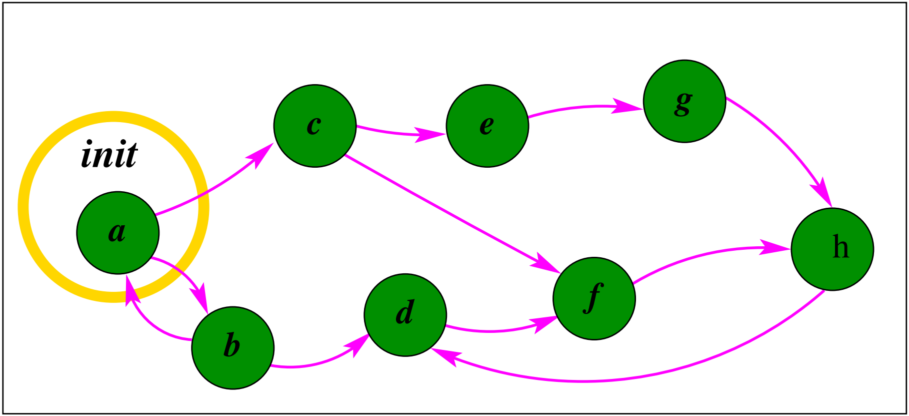
Explicit-state MC by BFS

- Set of visited states **V** $= \{a, b, c, d, e, f, g, h\}$
- Queue of expanded states **WQ** $= [g, h]$



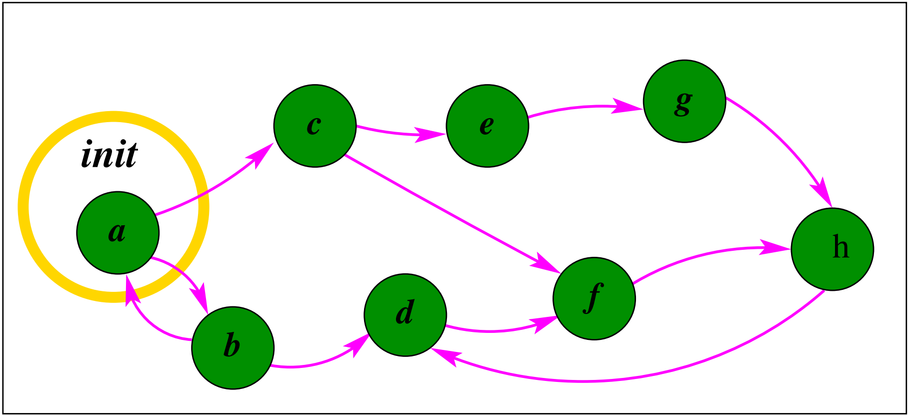
Explicit-state MC by BFS

- Set of visited states **V** $= \{a, b, c, d, e, f, g, h\}$
- Queue of expanded states **WQ** $= [h]$



Explicit-state MC by BFS

- Set of visited states **V** $= \{a, b, c, d, e, f, g, h\}$
- Queue of expanded states **WQ** $= []$



State-space explosion

- Bad news: *The number of reachable states tends to blow up exponentially in the number of variables, i.e* $|Reachable| \propto 2^{|AP|}$.
- In other words, adding just one more boolean variable to the system can cause the number of states to DOUBLE! This means double the memory and double the runtime for explicit-state MC of safety.
- ALL methods of MC suffer from this problem.
- Methods to curb: abstraction, symmetry reduction, partial order reduction
- Another method: distribute the MC computation among a network of machines!

Why Distributed Explicit State Model Checking?

- State-space explosion assures us that we can always use more memory (and cycles)
- Easily takes advantage of the aggregate memory of commodity machines and multiple cores

Question: Who cares about increasing our MC capabilities by a factor of $[100, 1000]$ when we face an exponential explosion?

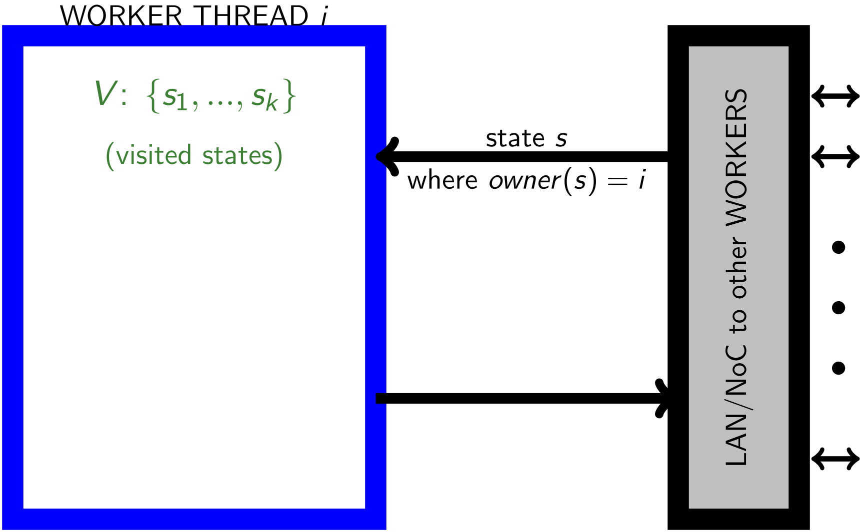
1. This factor can make the difference between verifying a very high level model and one that includes critical details
2. Techniques of abstraction/decomposition require human effort – terminate the human task sooner and hand it off to a large cluster

Outline

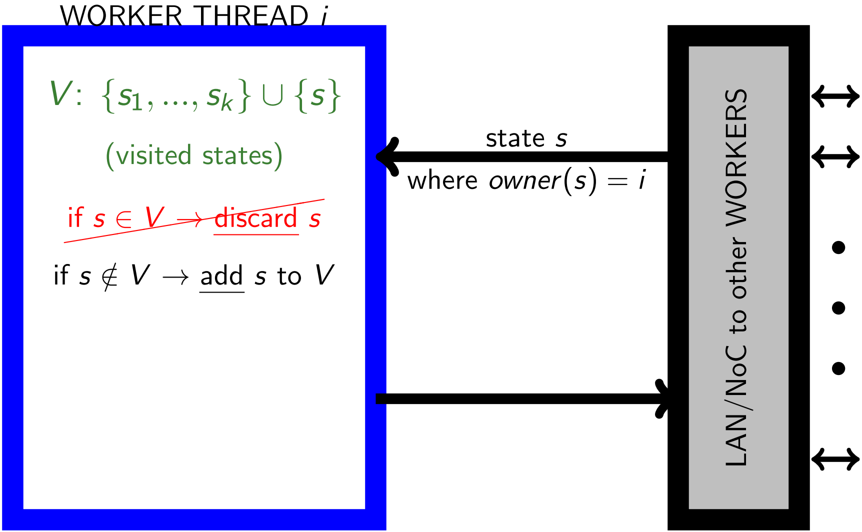
Stern-Dill Algorithm[SD97] Overview

- Simple and fundamental approach to distributing explicit-state model checking (for safety)
  - Assumes a uniform random hash function $owner : States \rightarrow PIDs$
  - Thread PID i only stores states s such that $owner(s) = i$.
- Each PID maintains two data structures:
  - **V**: Set of (owned) states visited so far
  - **WQ**: List of states waiting to be expanded
- Start: compute initial states and send to their owners
- Iterate: state sucessors are sent to their respective owners
- Termination: when each **WQ** is empty and no messages are in flight

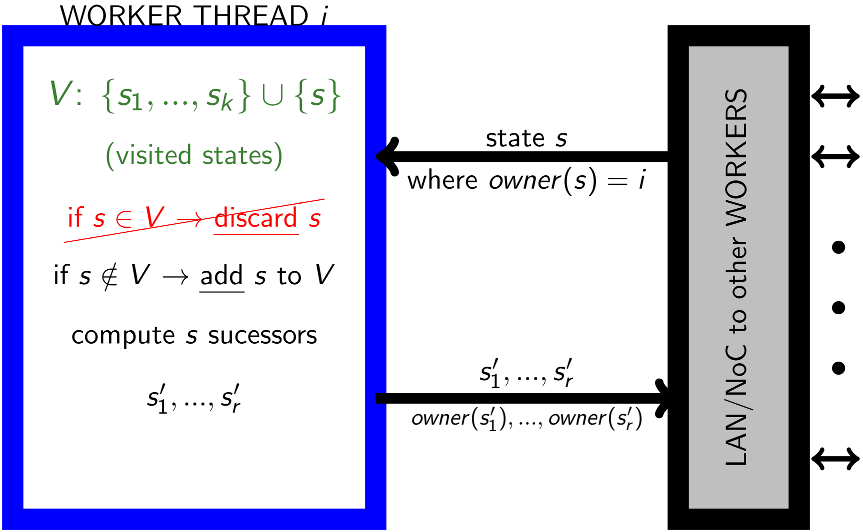
Stern-Dill Pseudocode



Stern-Dill Pseudocode



Stern-Dill Pseudocode



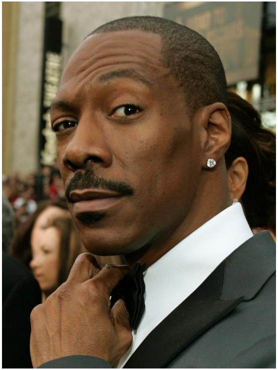
Some Optimizations

- "State Batching": from [SD97], delay sending states to another thread until enough states accumulate
  - Mitigates the overhead of network message passing
  - Important: appropriate proviso to avoid deadlock, i.e. always send states eventually
  - Every DEMC tool does this in some form
- "Less-Uniform Partitioning": from parallel Spin [LS99]
  - States are composed of sets of variables $(s_1, s_2, ..., s_k)$ for k Promella processes
  - Instead of *owner* depending on each s_i, let *owner* depend on only one s_i
  - If most transitions don't change s_i's variables, state sucessors stay local to the thread (no communication necessary)
  - However: may be less balanced than *owner* depending on all variables

Eddy (Murφ)[MPS+09]

Parallel implementation of the Murφ model checker, from Univ. of Utah

- Use MPI for distributed communication, split one Stern-Dill thread into two p-threads
- For each peer: maintain a communication queue of 8 batches of 1024 states
- Computation thread: expands states and writes to comm. queues
- Communication thread: wakes up when a message arrives or a batch fills up
- Overlaps message handling with state expansion; good from a software engineering perspective



Outline

What is PReach[BBdP+10]?

- PReach (*P*arallel *REACH*ability) is a distributed explicit-state model checker (UBC and Intel)
- Input: the Murφ modeling language; checks state invariants
  - New: Murφ syntax extended by PReach to support deadlock freedom and transition invariants
- Runs on a network of heterogenous machines including multicore
- Communication is handled by *Erlang*, a distributed functional language, while C++ libraries handle compute-intensive model checking tasks
- Emphasis on scalability – billions of states
  - Robustness
  - Simplicity
  - Performance? (a secondary concern)

Stern-Dill in PReach

WQ: list of states; stored on disk
V: set of states; Murφ hash table in memory

```
while ¬TERMINATED() {
    if ¬EMPTY(WQ) {
        s := DEQUEUE(WQ);
        foreach r in SUCCESSORS(s) {
            OWNER(r) ! r;   }}     # send successor state r
    if RECEIVE(s) {
        if ¬IS_MEMBER(s, V) {
            ADD_ELEMENT(s, V);
            CHECK_INVARIANTS(s);
            ENQUEUE(s, WQ);
}}}
```

Distributed Termination

How can we be sure that each **WQ** is empty and no messages are in flight?

Distributed Termination

How can we be sure that each **WQ** is empty and no messages are in flight?

Ideas?

Distributed Termination

How can we be sure that each **WQ** is empty and no messages are in flight?

Ideas?

1. Each thread keeps two counters, *NumSent* and *NumRecd*
   - *NumSent*++ when sending a state; *NumRecd*++ when a state is received

Distributed Termination

How can we be sure that each **WQ** is empty and no messages are in flight?

Ideas?

1. Each thread keeps two counters, *NumSent* and *NumRecd*
   - *NumSent*++ when sending a state; *NumRecd*++ when a state is received
2. When my **WQ** has been empty for some threshold amount of time, send message im_idle to root

Distributed Termination

How can we be sure that each **WQ** is empty and no messages are in flight?

Ideas?

1. Each thread keeps two counters, *NumSent* and *NumRecd*
   - *NumSent*++ when sending a state; *NumRecd*++ when a state is received
2. When my **WQ** has been empty for some threshold amount of time, send message im_idle to root
3. When root receives im_idle, broadcast request_stats to all workers

Distributed Termination

How can we be sure that each **WQ** is empty and no messages are in flight?
Ideas?

1. Each thread keeps two counters, *NumSent* and *NumRecd*
   - *NumSent*++ when sending a state; *NumRecd*++ when a state is received
2. When my **WQ** has been empty for some threshold amount of time, send message im_idle to root
3. When root receives im_idle, broadcast request_stats to all workers
4. When a worker receives request_stats, HALT computation and report:
   - If my **WQ** is nonempty: send im_not_done to root
   - If my **WQ** is empty: send {my_stats, *NumSent*, *NumRecd*} to the root

Distributed Termination

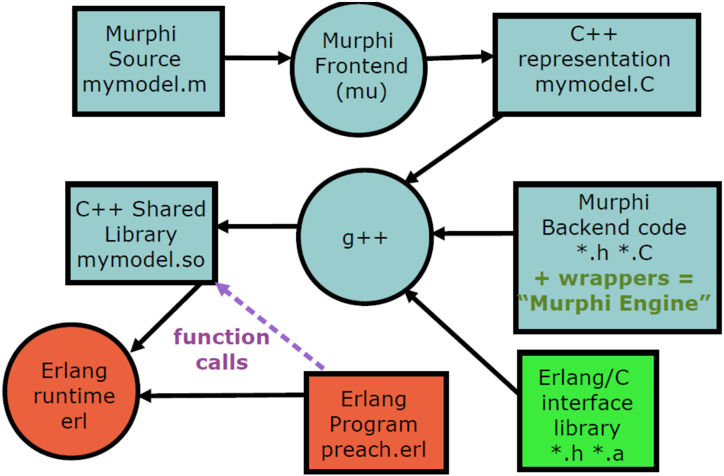
How can we be sure that each **WQ** is empty and no messages are in flight?

Ideas?

1. Each thread keeps two counters, *NumSent* and *NumRecd*
   - *NumSent*++ when sending a state; *NumRecd*++ when a state is received
2. When my **WQ** has been empty for some threshold amount of time, send message im_idle to root
3. When root receives im_idle, broadcast request_stats to all workers
4. When a worker receives request_stats, HALT computation and report:
   - If my **WQ** is nonempty: send im_not_done to root
   - If my **WQ** is empty: send {my_stats, *NumSent*, *NumRecd*} to the root
5. Root decides if we're really done:
   - If root receives my_stats messages from all workers AND $\sum NumSent_i = \sum NumRecd_i$, broadcast terminated
   - Otherwise: broadcast resume message

Architecture

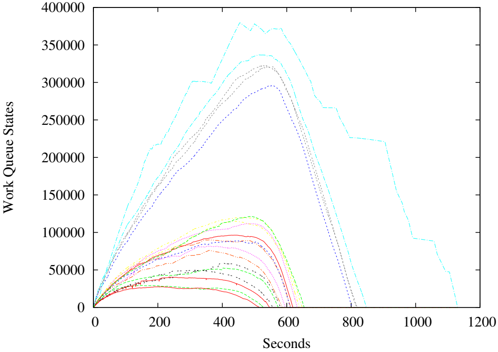
Murφ + tweaks + Erlang = PReach



Load Balancing

Bad News: While state space is partitioned evenly, dynamic load (**WQ** length) can vary a lot

- Some threads will finish early and idle
- Heterogenous computing environment exacerbates the problem



Good News: We can effectively balance load **without** altering the static state space partition

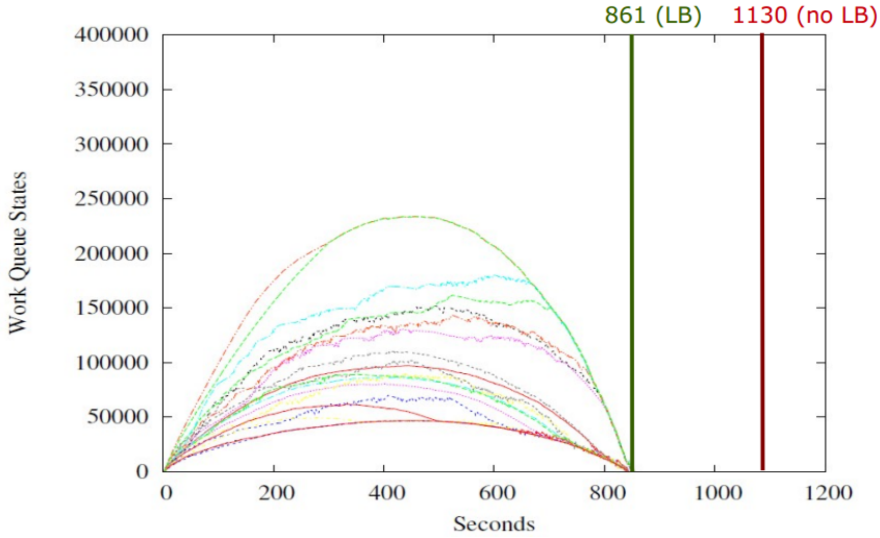
Stern-Dill in PREACH

WQ: list of states;
V: set of states;

```
while ¬TERMINATED() {
    if ¬EMPTY(WQ) {
        s := DEQUEUE(WQ);
        foreach r in SUCCESSORS(s) {
            OWNER(r) ! r;    }}
    if RECEIVE(s) {
        if ¬IS_MEMBER(s, V) {
            ADD_ELEMENT(s, V);
            CHECK_INVARIANTS(s);
            ENQUEUE(s, WQ);
}}}
```

Insight: After *s* is added to **V**, it doesn't matter which thread computes the sucessors of *s*!

Load Balancing Enabled



Results and Status

The "BIG Model": Intel industrial cache coherence protocol.

- $\approx$ 95 billion states! ☺
- $\approx$ 10 days runtime on 120 cores
- $\approx 110,000$ states/second; $\approx$ 900 states/second/core

Status:

- Currently in use at Intel
- Used by computer architects at Duke University, and a handful of other people at various institutions
- Available for download [BEBdP11]

Outline

0. Background: Explicit-State Model Checking
1. Stern-Dill Algorithm: Distributed BFS
   - + Other Tools
2. The PReach Model Checker
3. Remarks
4. Erlang Tips + Tricks

Erlang Pros + Cons

Was Erlang a good choice to build an industrial, explicit-state model checker?

Short answer is YES:

- Erlang is easier to program than C/C++ with MPI – original PReach prototype written in one weekend
- A good choice for this project where parallel speedup is not paramount, rather stability and scalability
- Small codebase, $\approx$ 1000 lines.
- I agree with these statements from `erlang.org/faq/how_do_i.html`:
  - Lines of code: "A reasonably complex problem involving distribution and fault tolerence will be roughly five times shorter in Erlang than in C"
  - Performance: Number crunching is about 10 times slower in Erlang than C; communication heavy programs are about the same speed.

Erlang Pros + Cons

Was Erlang a good choice to build an industrial, explicit-state model checker?

Short answer is YES, **however...**

- Documentation for Erlang isn't great, and some of the more obscure features aren't explained well
- The method of interfacing with C code (.so files) is miserable, and the API seems to change with new Erlang versions ☹
  - We learned how to do this from some random blog
  - PReach uses $\approx$ 30 interface functions that call into Murφ C code, took some trial and error to learn how to pass various data types

A Few Directions

1. Bottlenecks: in PReach (+ other tools), the bottleneck is state-expansion – especially bad in industrial models with ≈ 5000 guarded commands!
   - Several studies have considered GPU-accelerated model checking;
   - Recent work [BBBC10] is the first (that I've seen) to use more than one GPU – although they only use 2, achieving a factor of 5 speedup
2. Crash Recovery: an important consideration when running hundreds of machines for days
   1. Snapshot **V** and **WQ** periodically: can recover from model checking thread crashes
   2. Duplicate state ownership: can recover from a machine going down
3. PReach has a modest number of parameters for load balancing, batching, flow control
   - Use machine learning techniques [HHLBS09] to tune parameters according to a new hardware configuration!

Outline

Profiling

- There's 4 profiling tools in Erlang: fprof, eprof, cover, cprof
- My preference is eprof
- Easy to use:
  - At the start of your program, insert the lines
    ```
    eprof:start(),
    eprof:start_profiling([self()]),
    ```
  - At the end of your program, inser the lines
    ```
    eprof:stop_profiling(),
    io:format("Here's the eprof output:~n"),
    eprof:analyze(),
    ```
- Gives the number of times each function was called as well as the total time spent in each function
- Program slowdown is modest

The Process Dictionary

For when you REALLY want global variables...

- Each process has it's own "dictionary" that can be used to store global variables
- Set and get with `put(Key,Value)` and `get(Key)`; delete with `delete(Key)`
- Really useful for debugging or gathering program statistics

Case vs. If

I slowly learned to always use case, and never use if

- Suppose we want to implement a set with a list (i.e. only insert elements that are new).
- With case:
```
insert(X,Set) ->
case lists:member(X,Set) of true -> Set;
false -> [X | Set] end.
```
- With if:
```
insert(X,Set) ->
IsInSet = lists:member(X,Set),
if IsInSet -> Set;
true -> [X | Set] end.
```

Beware Big Mailboxes

Warning: The time it takes to receive a message is proportional to the number of messages waiting in the inbox!

- Ignoring this issue in PReach causes crashes that arise from some workers slowing down to a halt
- As soon as one worker falls a little behind, it will never catch up because it takes longer to receive states than the others
- Solved with a crediting mechanism
- Lesson: make sure your inboxes don't blow up (say with stale messages)
- Inbox size can be checked with

```
{_, InboxSize} = process_info(self(),message_queue_len)
```

Erlang Pointers

- erlang.org has OK documentation, but I prefer
- Tutorial Blog "Learn you some Erlang for great good!" by Frederic Trottier-Hebert
  learnyousomeerlang.com/content
- Joe Armstrong's Book, "Programming Erlang"
- PReach source
  https://bitbucket.org/jderick/preach

Contact

Brad Bingham
binghamb@cs.ubc.ca

Brad Bingham
binghamb@cs.ubc.ca

Thank-you!

References I

J. Barnat, P. Bauch, L. Brim, and M. Ceska, Employing multiple cuda devices to accelerate ltl model checking, Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on, 2010, pp. 259–266.

B. Bingham, J. Bingham, F. M. de Paula, J. Erickson, G. Singh, and M. Reitblatt, Industrial strength distributed explicit state model checking, Parallel and Distributed Model Checking, 2010.

Jesse Bingham, John Erickson, Brad Bingham, and Flavio M. de Paula, Open-source PReach, http://bitbucket.org/jderick/preach, 2011.

Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle, ParamILS: an automatic algorithm configuration framework, Journal of Artificial Intelligence Research **36** (2009), 267–306.

References II

- F. Lerda and R. Sisto, Distributed-memory model checking with SPIN, Proc. of SPIN 1999, volume 1680 of LNCS., Springer-Verlag, 1999, pp. 22–39.
- I. Melatti, R. Palmer, G. Sawaya, Y. Yang, R. M. Kirby, and G. Gopalakrishnan, Parallel and distributed model checking in eddy, Int. J. Softw. Tools Technol. Transf. **11** (2009), no. 1, 13–25.
- U. Stern and D. L. Dill, Parallelizing the murphi verifier, International Conference on Computer Aided Verification, 1997, pp. 256–278.