Industrial Strength Distributed Explicit State Model Checking (with PReach)

Brad Bingham*, Jesse Bingham†, Flavio M. de Paula*, John Erickson,† Gaurav Singh†, Mark Reitblatt‡

*University of British Columbia, Canada

†Intel Corporation, USA

‡Cornell University, USA

Outline

- **The PReach Tool**
  - Motivation
  - Algorithm
  - Software Architecture
- Features
  - Crediting Mechanism
  - Lightweight Load Balancing
- Status & Future Work

Punch Line: The PReach tool *increases* the size of the largest models that can be *tractably* verified with explicit state model checking.

What is PReach?

- PReach (Parallel REACHability) is a distributed explicit state model checker (UBC, Intel)
- Input: the Murphi modeling language
  - Checks state-invariants
- Communication is handled by *Erlang*, a distributed functional language, while C++ libraries handles more compute-intensive work
- Clean and simple implementation
- Scalable to very large models (approx 30 B)
- Released under BSD license

Why PReach?

- At Intel, they want to check very large models
  - Usually, capacity is the issue
  - They have a cluster of a few hundred machines
- Available tools were evaluated to have robustness/correctness issues
- Let's build our own tool emphasizing
  - Robustness
  - Simplicity
  - **Scalability**
  - Performance? (a secondary concern)

Stern-Dill Algorithm [1996]

- Simple approach to distributing explicit-state model checking computation
  - Assumes a uniform hash function
    `owner : State -> PID`
  - PID **`p`** only stores states **`s`** such that
    `owner(s) = p`
- Start by sending initial states to respective owners
- When successors are computed they are sent to their respective owners
- Each PID maintains two data structures:
  - V: Set of states visited so far
  - WQ: List of states waiting to be visited
- Terminates when all **WQ**s are empty and there are no messages in flight

Stern-Dill Algorithm

```
WQ : list of States;
V : Set of States;

if i_am_master {
   foreach s in initial_states() {
      owner(s) ! s; // send
   } }
while !terminated() {
   if !empty(WQ) {
      s := dequeue(WQ);
      foreach r in Successors(s) {
         owner(r) ! r; // send
      } }
   if receive(s) {
      if !member(s,V) {
         add_element(s,V);
         check_invariants(s);
         enqueue(s,WQ);
      } } }
```

Stern-Dill Algorithm

```
WQ : list of States;
V : Set of States;

if i_am_master {
   foreach s in initial_states() {
      owner(s) ! s; // send
   } }
while !terminated() {
   if !empty(WQ) {
      s := dequeue(WQ);
      foreach r in Successors(s) {
         owner(r) ! r; // send
      } }
   if receive(s) {
      if !member(s,V) {
         add_element(s,V);
         check_invariants(s);
         enqueue(s,WQ);
      } } }
```

In PReach's implementation:

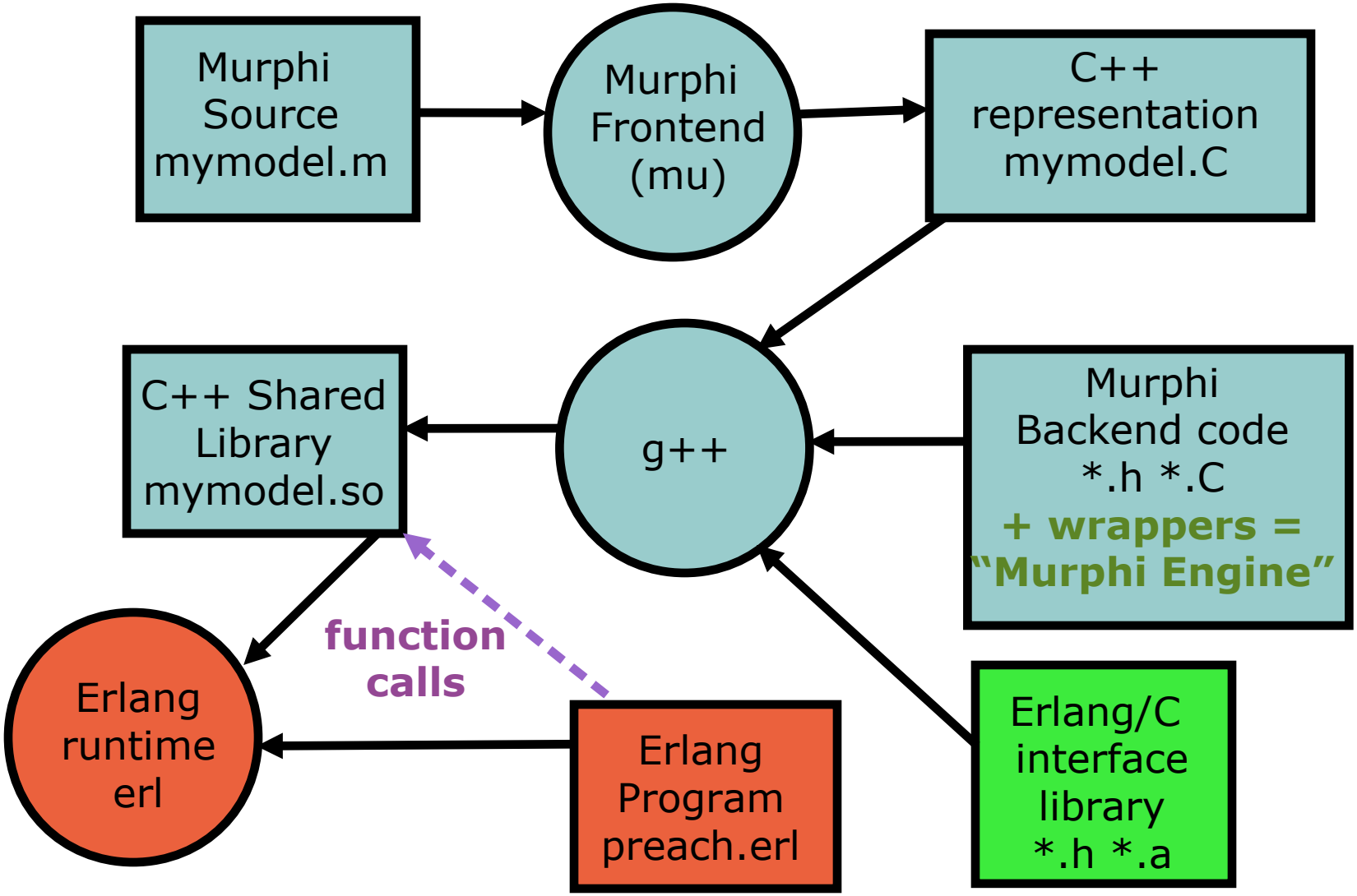
- V is stored as a Murphi hash table in memory

- WQ is stored on disk; more space for hash table in memory (i.e., larger model capacity) at a small performance penalty

PReach Architecture

- Murphi already has an input language, symmetry reduction, hash compaction, state successor generation and is highly optimized.
  - Let's reuse it!
- Wrote interface between PReach's Erlang code and Murphi's C++ code
- Good trade-off between performance and complexity

Murphi + tweaks + Erlang = PReach



MurphiEngine/Erlang Interface

Here **X** and **Y** are states

- `startstates()` // state list
- `nextstates(Y)` // state list
- `checkInvariants(X)` // boolean
- `init_hash(Size)` // void
- `checkHashTable(X)` // boolean
- `probNoOmission()` // float
- `canonicalize(X)` // state
- `equivalentStates(X,Y)` // boolean
- `numberOfHashCollisions()` // integer
- `normalize(X)` // state

For counter example generation only

- `stateToString(X)` // string
- `print_diff(X,Y)` // string
- `fireRule(X,Rule)` // state
- `rulenumToName(RuleNum)` // string
- `startstateToName(SSNum)` // string
- `whatRuleFired(X,Y)` // integer

Outline

- The PReach Tool
  - Motivation
  - Algorithm
  - Software Architecture
- **Features**
  - Crediting Mechanism
  - Lightweight Load Balancing
- Status & Future Work

Punch Line: The PReach tool *increases* the size of the largest models that can be *tractably* verified with explicit state model checking.

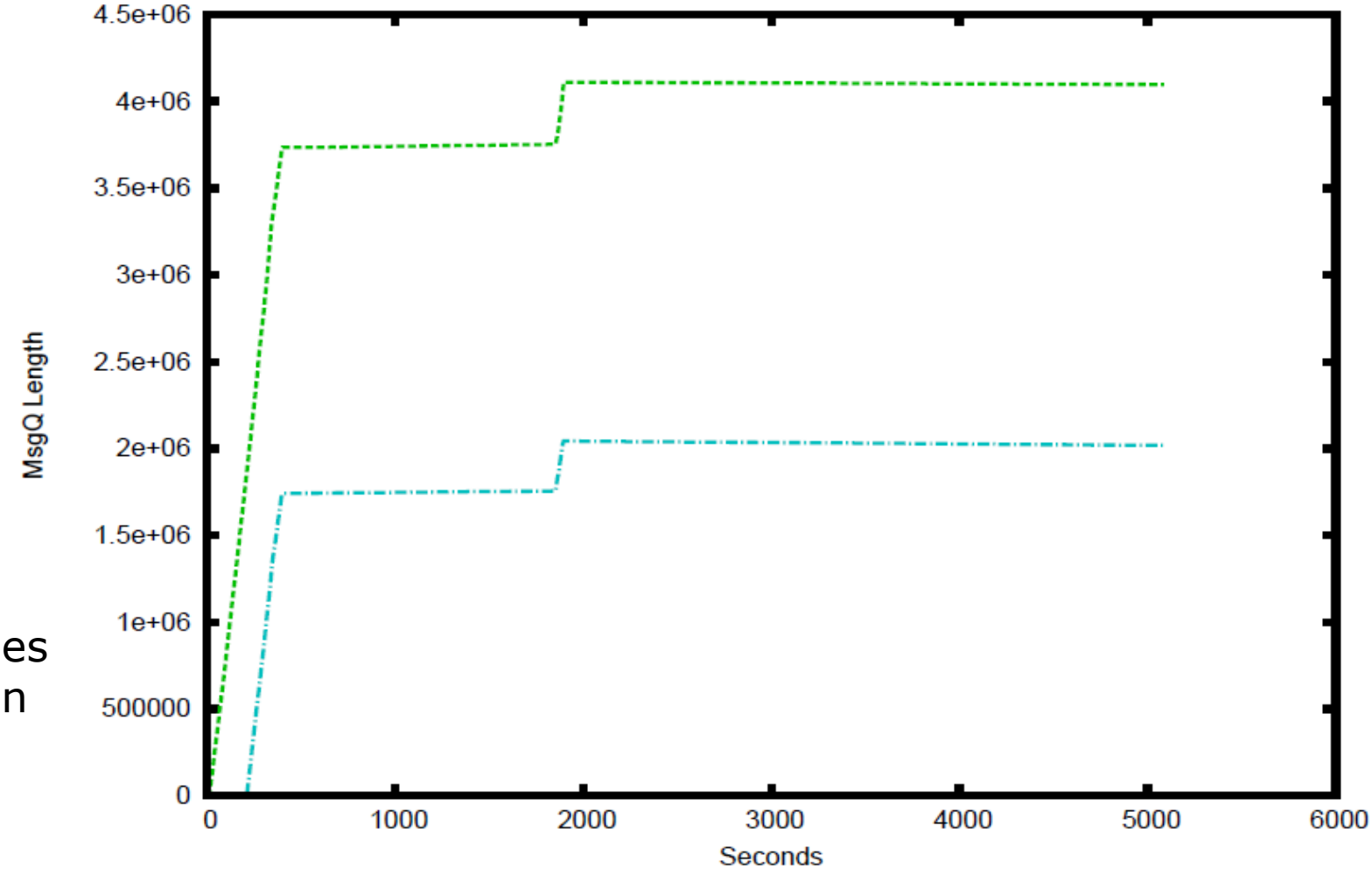
Flow Control Problems

- Problem: Often a few nodes would massively slowdown and even crash; memory usage also blew up
- Diagnosis: Slower nodes could be overwhelmed with messages piling up in the Erlang Runtime message queue.
  - We call this a "bogged down" node.
- Solution: Implement a crediting protocol

PReach without Crediting

LDASH protocol with 8 nodes

2 of them explode, and 6 nodes aren't even visible!



Crediting

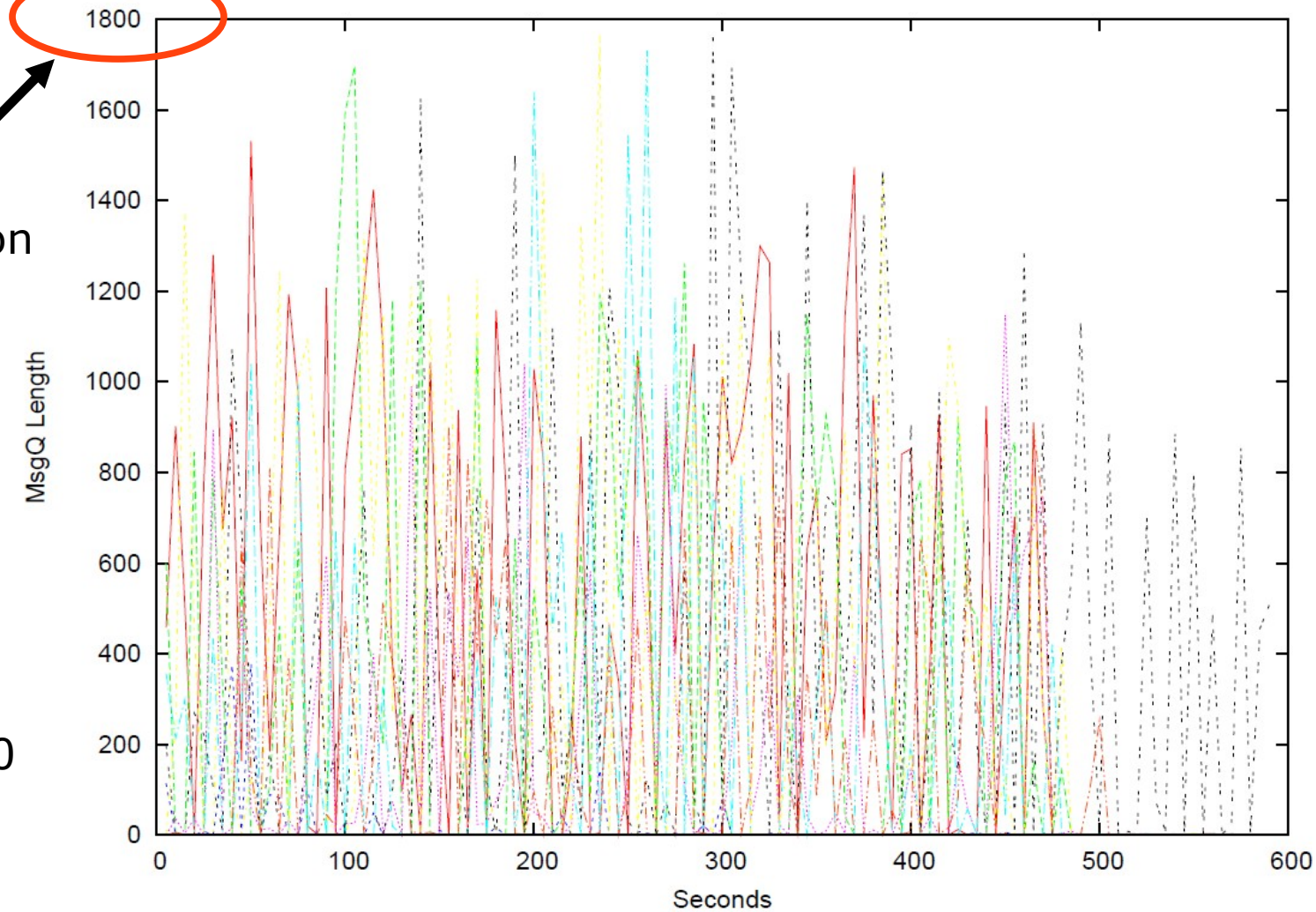
- Each worker given C credits for each other worker;

  1 credit = 1 unacknowledged message
- Credits returned via ack messages
- Provides hard bound N*C on size of message queue
- If no credits available, states stored in outgoing message queue, and written to disk if large enough (workers don’t block)

PReach with Crediting

1800
<<
1.5 Million

LDASH protocol with 8 nodes, C = 1000



Load Balancing

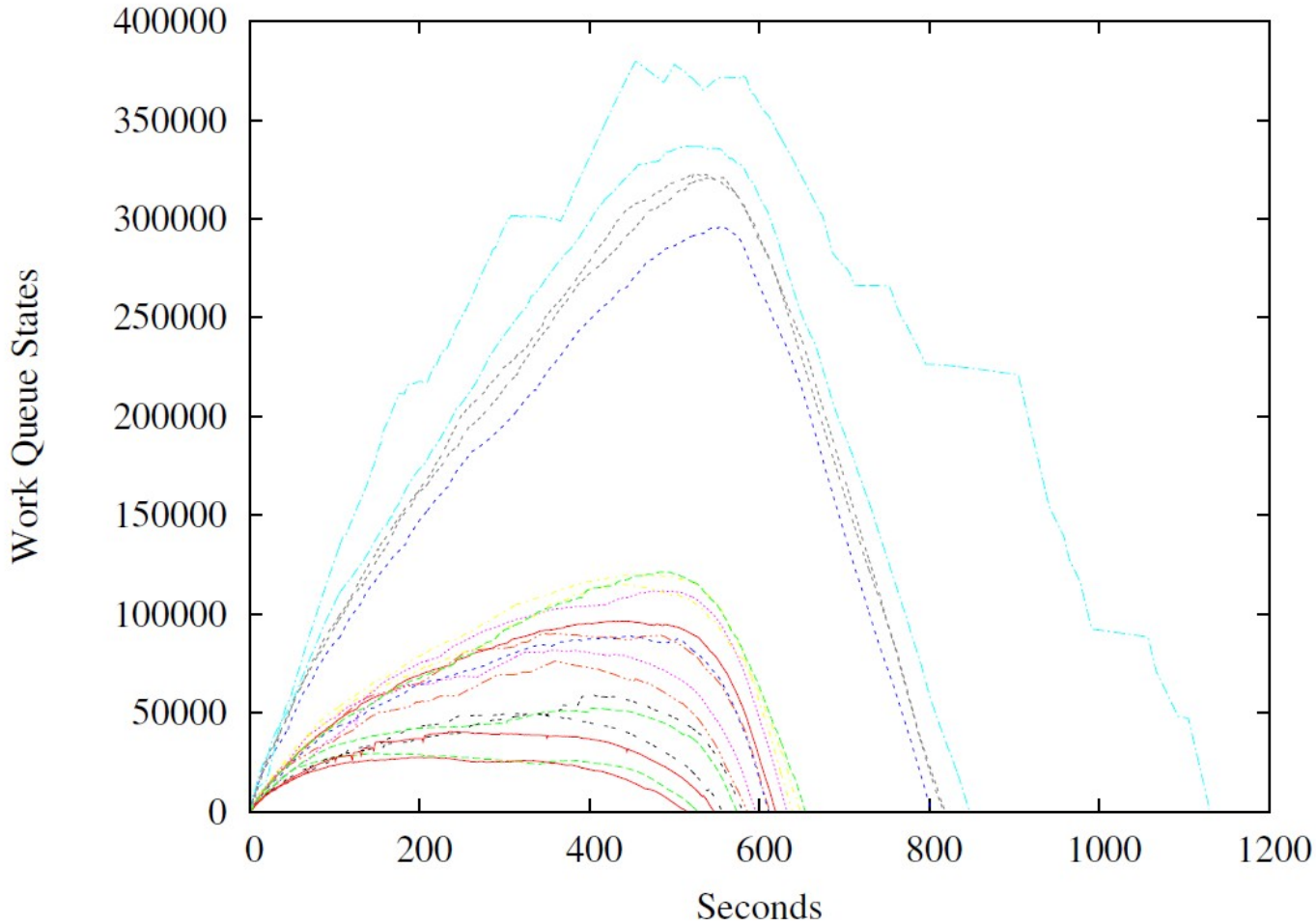
- Bad News: State space is partitioned evenly, but dynamic load (WQ length) can vary a great deal
  - Some nodes will finish early and idle while others are still working
  - Heterogeneous computing environment exacerbates this problem
- Good News: We can load balance ***without*** altering the static state space partition

Stern-Dill Algorithm

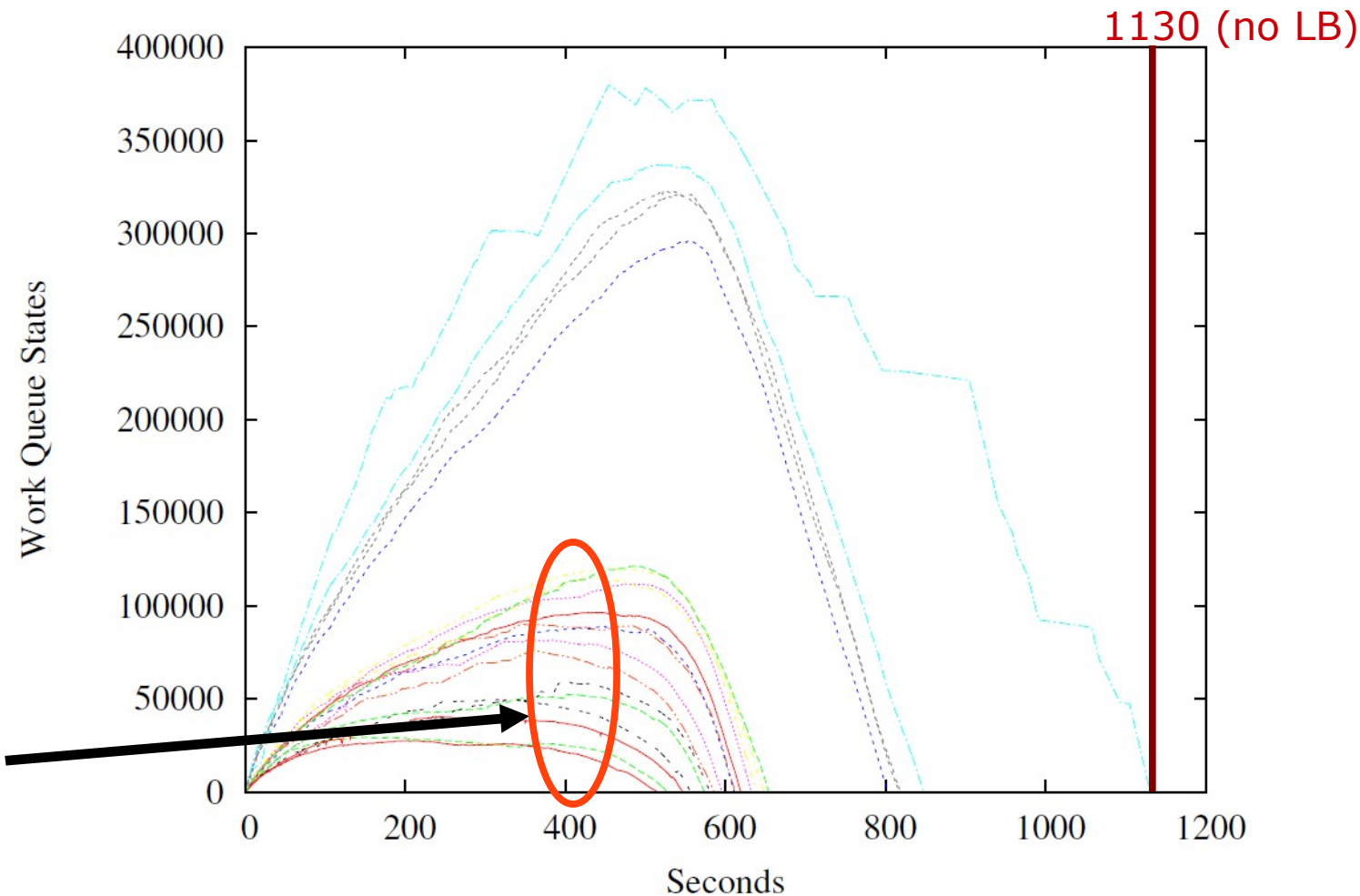
```
WQ : list of States;
V : Set of States;

if i_am_master {
   foreach s in initial_states() {
      owner(s) ! s; // send
   } }
while !terminated() {
   if !empty(WQ) {
      s := dequeue(WQ);
      foreach r in Successors(s) {
         owner(r) ! r; // send
      } }
   if receive(s) {
      if !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 successors of s!

PReach without Load Balancing



PReach without Load Balancing

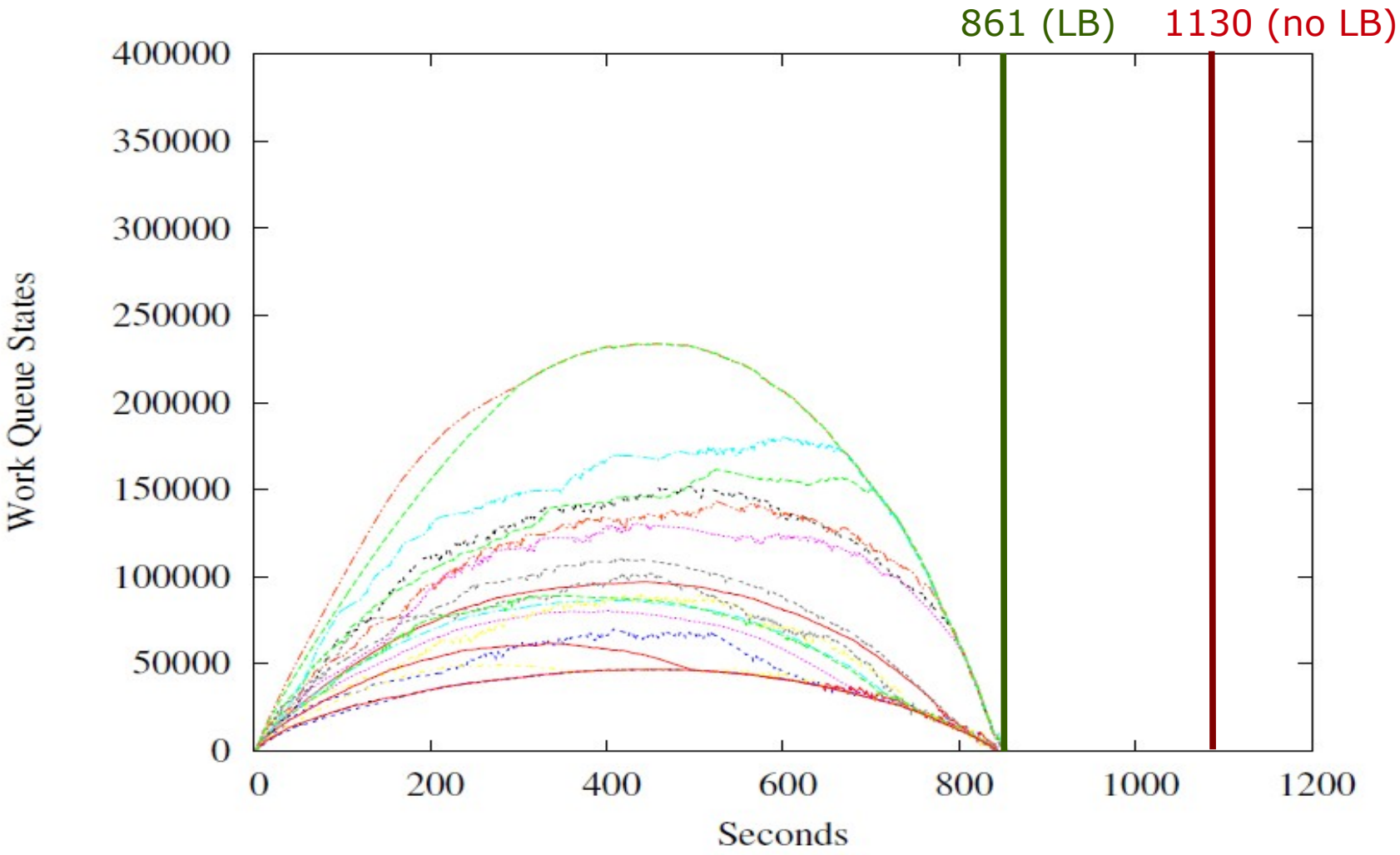


These nodes are near-idle for last half of run!

Lightweight Load Balancing

- Prev work [Kumar & Mercer 2004]: keep size of WQ equal across all workers
- Insight: too much LB adds unnecessary work, no need to minimize max WQ size.
  - Instead, avoid idle workers
- Technique: when one worker notices another has X times fewer states in work queue, it sends some from it's own work queue (X=~5)
- Results: reduction in runtime of up to 40% (vs non-LB)

PReach with Lightweight LB



Future Work

- Performance…
  - Currently, each worker is a factor of X the speed of Murphi for X in (0.2, 0.5). However, X is near 0.5 for industrial models (i.e. 50%)
  - Profiling Erlang programs has proven to be challenging
- Support limited class of CTL properties for deadlock detection
- Fault tolerance ("PReachDB")
- Partial Order Reduction?
- Support other modeling languages?

Current Status

- PReach is stable, tested with up to 256 nodes
- Tested with real models up to 30 Billion states (up to 100B on toy model)
- Currently in use to detect deadlocks and other protocol properties using a uArch model of critical protocol components
- Public Release under BSD license
  - *http://www.bitbucket.org/jderick/preach*

Current Status

- PReach is stable, tested with up to 256 nodes
- Tested with real models up to 30 Billion states (up to 100B on toy model)
- Currently in use to detect deadlocks and other protocol properties using a uArch model of critical protocol components
- Public Release under BSD license
  - *http://www.bitbucket.org/jderick/preach*
- **Questions?**

Explicit State Model Checking Memory Limitations

- The set of visited states has to eventually store all reachable states
- Use compression on states in
  - Hash compaction (Murphi)
  - Bloom filter (Spin)
- Symmetry reduction
  - Only store a single representative for each equivalence class of states in a symmetric system

Stern-Dill Algorithm [1996]

- Simple approach to distributing explicit-state model checking computation
  - Assumes a uniform hash function
    `owner : State -> PID`
  - PID **`p`** only stores states **`s`** such that
    `owner(s) = p`
- Start by sending initial states to respective owners
- When successors are computed they are sent to their respective owners
- Upon Receiving state **`s`**
  - Is s in my hash table? Yes → discard **`s`**
  - No → Add **`s`** to both hash table and **`WQ`**
- Terminates when all **`WQ`**s are empty and there are no messages in flight

Stern-Dill Algorithm

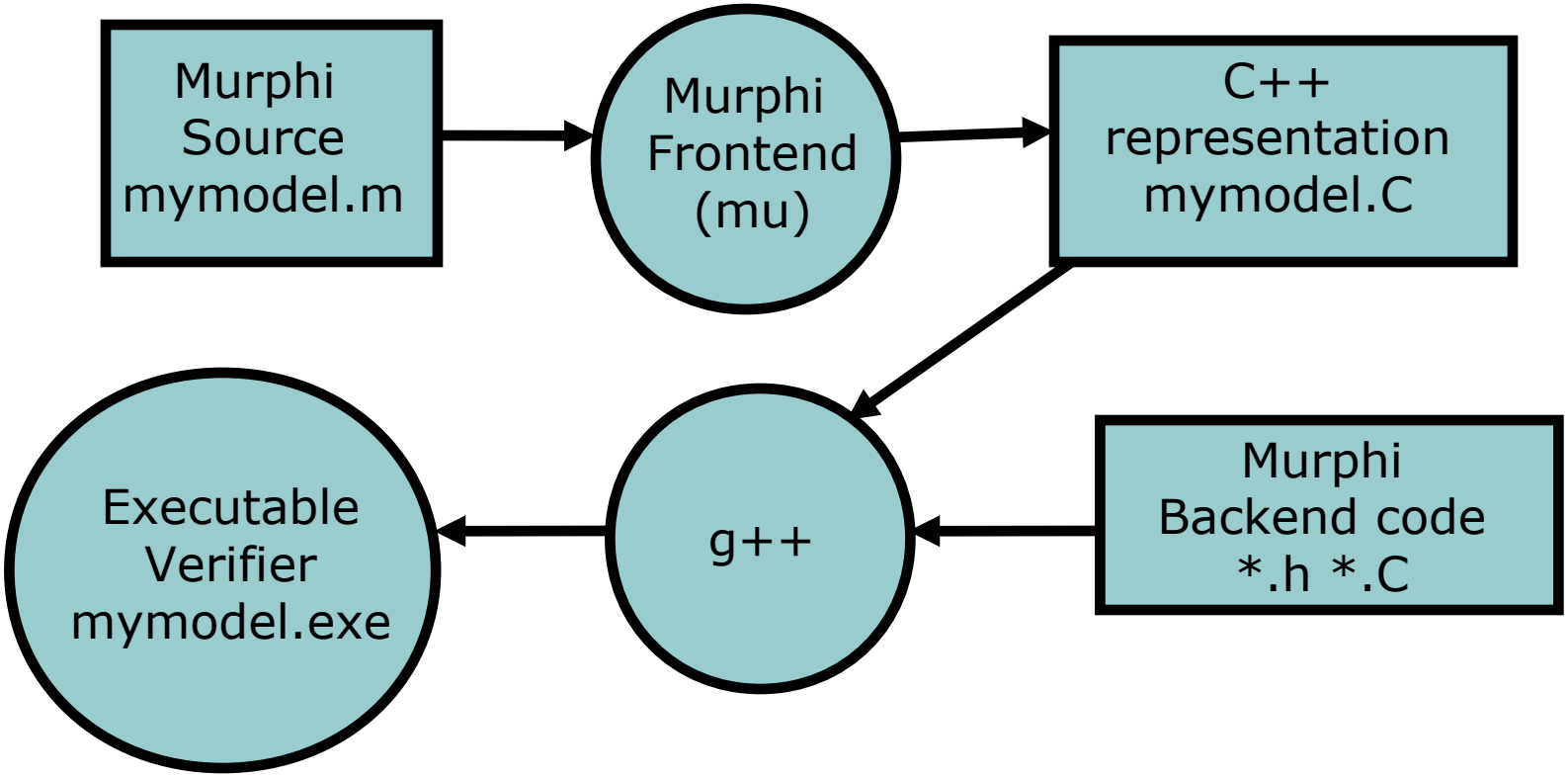
```
WQ : list of States;
V : Set of States;

if i_am_master {
   foreach s in initial_states() {
      owner(s) ! s; // send
   }
}
while !terminated() {
   if !empty(WQ) {
      s := dequeue(WQ);
      foreach r in Successors(s) {
         owner(r) ! r; // send
      }
   }
   if receive(s) {
      if !member(s,V) {
         check_invariants(s);
         enqueue(s,WQ);
         add_element(s,V);
      }
   }
}
```

In PReach's implementation:

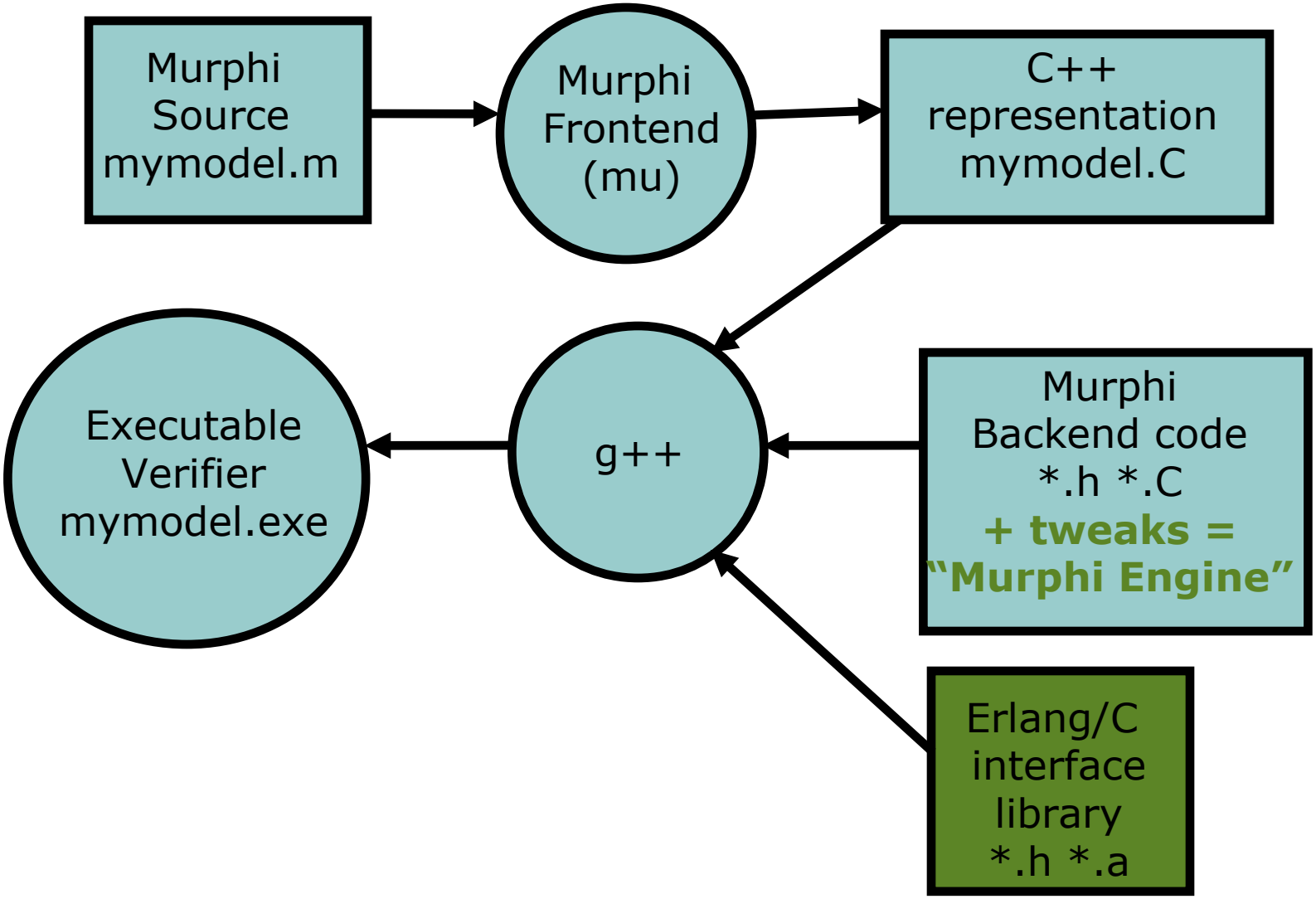
- V is stored as a Murphi hash table in memory

- WQ is stored on disk; more space for hash table in memory (i.e., larger model capacity) at a small performance penalty

Murphi

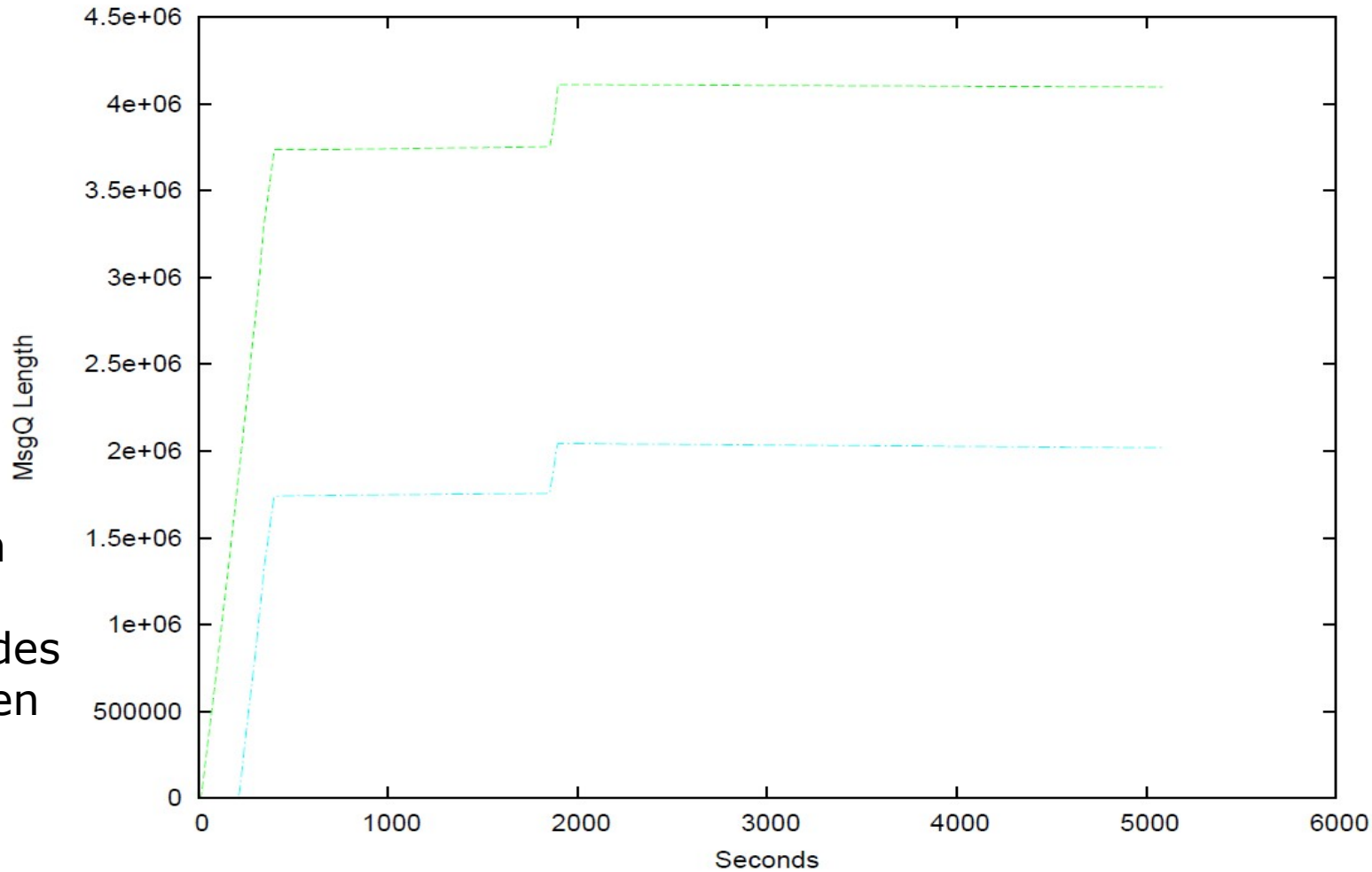

Murphi
Source
mymodel.m
Murphi
Frontend
(mu)
C++
representation
mymodel.C
Executable
Verifier
mymodel.exe
g++
Murphi
Backend code
*.h *.C

Murphi + tweaks



PReach without Crediting

LDASH protocol with 8 nodes



2 of them explode, and 6 nodes aren't even visible!