Industrial Strength, Parallel Model Checking

Mark Greenstreet

CpSc 418 – Mar. 31, 2017

- The Big Five
- A Parallel Algorithm for Model Checking
- Implementing a Parallel Model Checker
- Summary, Preview, & Review

Mark's Five Questions: 1–3

1. What problem does the paper address?
   - Model checking of real problems in industry requires the combined memory of many machines. Existing implementations of parallel model checkers failed to successfully run on real examples. A robust, distributed, parallel model checker is needed.
2. What is the key idea in the paper?
   - Combine murφ and Erlang.
   - Use the existing murφ code to implement the computationally intensive part of the code.
   - Use Erlang to handle the communication and coordination between worker processes.
3. How do the authors validate their idea?
   - They implemented a model checker.
   - They performed experiments to identify robustness issues and performance bottlenecks. They implemented solutions to these problems.
   - They used the PReach model checker on several benchmark examples and some real designs from Intel.

Mark's Five Questions: 4–5

1. Is the paper convincing?
   - **Yes:** they showed that they could check properties of real designs. They set new records for model-size for explicit-state model checking.
   - **No:** the run times for large examples can be several days.
   - **20/20 hindsight:** Definitely. With continued work, the performance of PReach has been improved. Brad Bingham went on to show that this explicit state approach could solve liveness verification problems that other methods cannot.
2. Any other comments?
   - PReach lacks a crash-recovery mechanism.
   - PReach doesn't take advantage of having multiple cores on a single CPU: This would require revising the base murφ code to create shared state tables, etc.

Model Checking: the algorithm

```
initialState = ...;
workList = queue(); // initially empty
knownStates = set(); // initially empty
workList.insert(initialState);
while len(workList) > 0:
    s = workList.removeNext();
    check s' for mutual exclusion;
    for s' in next_states(s):
        if s' not in knownStates:
            add s' to workList and knownStates;
```

How can we make this algorithm parallel?

- Where does the code spend most of the time?

Model Checking: the algorithm

```
initialState = ...;
workList = queue(); // initially empty
knownStates = set(); // initially empty
workList.insert(initialState);
while len(workList) > 0:
    s = workList.removeNext();
    check s' for mutual exclusion;
    for s' in next_states(s):
        if s' not in knownStates:
            add s' to workList and knownStates;
```

How can we make this algorithm parallel?

- Where does the code spend most of the time? `next_states(s)`

Model Checking: the algorithm

```
initialState = ...;
workList = queue(); // initially empty
knownStates = set(); // initially empty
workList.insert(initialState);
while len(workList) > 0:
    s = workList.removeNext();
    check s' for mutual exclusion;
    for s' in next_states(s):
        if s' not in knownStates:
            add s' to workList and knownStates;
```

How can we make this algorithm parallel?

- Where does the code spend most of the time? `next_states(s)`
- What are the **dependencies**?

Model Checking: the algorithm

```
initialState = ...;
workList = queue(); // initially empty
knownStates = set(); // initially empty
workList.insert(initialState);
while len(workList) > 0:
    s = workList.removeNext();
    check s' for mutual exclusion;
    for s' in next_states(s):
        if s' not in knownStates:
            add s' to workList and knownStates;
```

How can we make this algorithm parallel?

- Where does the code spend most of the time? `next_states(s)`
- What are the **dependencies**?
  - As written, the code is very sequential.
  - BUT, the correcteness of the algorithm does not depend on the order in which states are removed from `workList`.

Model Checking: the algorithm

```
initialState = ...;
workList = queue(); // initially empty
knownStates = set(); // initially empty
workList.insert(initialState);
while len(workList) > 0:
    s = workList.removeNext();
    check s' for mutual exclusion;
    for s' in next_states(s):
        if s' not in knownStates:
            add s' to workList and knownStates;
```

How can we make this algorithm parallel?

- Where does the code spend most of the time? `next_states(s)`
- What are the **dependencies**?
  - As written, the code is very sequential.
  - BUT, the correcteness of the algorithm does not depend on the order in which states are removed from `workList`.
- What uses most of the memory?

Model Checking: the algorithm

```
initialState = ...;
workList = queue(); // initially empty
knownStates = set(); // initially empty
workList.insert(initialState);
while len(workList) > 0:
    s = workList.removeNext();
    check s' for mutual exclusion;
    for s' in next_states(s):
        if s' not in knownStates:
            add s' to workList and knownStates;
```

How can we make this algorithm parallel?

- Where does the code spend most of the time? `next_states(s)`
- What are the **dependencies**?
  - As written, the code is very sequential.
  - BUT, the correcteness of the algorithm does not depend on the order in which states are removed from `workList`.
- What uses most of the memory? `knownStates` and `workList`

Making it Parallel

```
initialState = ...;
workList = queue(); // initially empty
knownStates = set(); // initially empty
workList.insert(initialState);
while len(workList) > 0:
    s = workList.removeNext();
    check s' for mutual exclusion;
    for s' in next_states(s):
        if s' not in knownStates:
            add s' to workList and knownStates;
```

- Divide `knownStates` and `workList` across the worker processes? How?
  - Sending each new state, `s'` to every worker is a bad idea. Why?
  - We'll use hashing instead: send state `s'` to worker process `hash(s') mod P`.
- Each worker maintains `knownStates` for the states that hash to the worker process.
- Each worker adds each **new** state it receives to `knownStates` and `workList`.
- Each worker processes the states on its own worklist.

Parallel Model Checking: the Pseudo-Code

```
modelCheck(KnownStates, WorkList) ->
   receive
      S when member(S, KnownStates) -> % already seen
         modelCheck(KnownStates, WorkList);
      S -> modelCheck(add(S, KnownStates),
                      add(S, WorkList))
      after 0 ->
         case WorkList of
            [] -> is_everyone_else_done();
            [S | Tl] ->
               [owner(S') ! S' || S' <- next_states(S)],
               modelCheck(KnownStates, Tl)
         end
   end.
```

- See Parallelizing the murφ model checker, U. Stern and D.L. Dill.

Parallel Model Checking: Performance Analysis

- A sequential implementation of the model checking algorithm requires $\mathcal{O}(SR)$ time, where S is the number of reachable states, and R is the number of rules.
- A parallel implementation requires $\mathcal{O}(SR/P)$ compute time, and sends worst-case $\mathcal{O}(SR)$ messages.
  - In practice, the average number of successors of each state (i.e. the degree of the state-graph) is relatively small. If we assume this is a small constant, then we get $\mathcal{O}(S)$ messages.
- Consider the case where each worker process generates σ new successor states, s', per second.
  - These are send to the other processes uniformly at random (if we have a good hash function).
  - Half of these messages cross the bisection of any network.
  - That means we need a bisection bandwidth of $\sigma P/2$.
- For real-life networks, bisection bandwidth grows much more slowly than P.
  - If we scale this algorithm to a large enough number of processors, network bandwidth will be the limiting constraint.
  - This is a common performance pattern in parallel computing.

From Algorithm to Industrial Adaptation

What is needed for real-world verification?

- Lots of memory:
  - Memory and time are both concerns for model checkers, but memory tends to be the more critical concern.
  - A parallel implementation offers the combined memory of a large number of machines.
- Robustness:
  - Simple architecture and re-use stable, well-exercised code.
  - Prevent “overwhelm and crash”.
  - Load balancing.
- Flexibility:
  - Solve problems that other tools cannot
  - In particular, liveness properties such as “response”.

Erlang for high-performance computing (really)

- Use existing C++ code for murφ.
  - It has been carefully optimized – it's fast.
  - It has been widely used over the past 25 years – it's robust.
- Use Erlang to make it parallel
  - Erlang handles the communication between processes.
  - The code is simple: it works and it's flexible.
  - Erlang can call the C++ functions:
    - ★ The compute intensive part is done in C++.
    - ★ The Erlang code is not a serious bottleneck.

Memory

- The worklist is the dominant use of memory (in practice)
  - Why?
  - The worklist needs complete state descriptions.
  - The known-state set can use much smaller hashes.
- Solution: store the worklist on disk.
  - Disks are slow – is this crazy?
  - It works just fine because we can access the worklist in **any** order.
  - Keep a large piece of the worklist in main memory.
  - If the in-memory work list grows too large, then copy a large chunk to disk.
  - If the in-memory work list becomes too small, then read a large chunk from disk.
  - The disk reads and writes can be performed asynchronously.
  - See Using magnetic disks . . . in the murφ model checker, U. Stern and D.L. Dill.
- Storing the known-state set on disk is much less practical because it's a random-access structure.

Batching Messages

- If we send each new state, s', one at a time to its owner process, communication overhead dominates the run time.
- **Key lesson**: pay attention to λ.
- The Erlang code maintains a separate buffer for each worker process
  - Add states that should be sent to that process to the buffer until we have enough.
  - Then send them as a batch.
  - A process that is running low on work sends requests to the other workers to ask them to flush their buffers.
  - These flush requests are bundled with state batches to avoid extra messages.
- Erlang makes the communication architecture simple and easy to extend.

Overwhelm and Crash

The dangers of using Erlang for high-performance computing

- The Erlang in-box is a list.
  - Newly received messages are prepended to the list.
  - A receive gets the oldest message that matches a pattern of the receive.
  - This means that the time for receive is linear in the number of pending message.
- This leads to a performance catastrophe
  - If a process gets slightly behind, its inbox will fill a little more than the other processes.
  - This means that a process that falls behind will slow down, and its inbox will fill even more.
  - Eventually, the process crashes.

Credits

Preventing “overwhelm and crash”

- Drain the inbox into another buffer whenever possible.
- Maintain a credit system
  - When a process X sends a message to process Y, X decrements its credit-count for Y.
  - If the credit-count is 0, X waits to send its message.
  - When Y moves a message from X out of its inbox, it sends a credit back to X.
  - Of course, these messages are piggy-backed on the new-state messages.

Load Balancing

- Not all processes have the same amount of work, and they don't all run at the same speed.
- This can lead to **idle processors**.
- Solution:
  - Processes include the length of their worklist in their messages to other workers.
  - If a worker has a short worklist, it asks for half of the worklist of the worker with the longest worklists.
- This is a very coarse-grained approach
  - PReach makes no effort to keep worklist lengths equal.
  - The coarse-grained approach requires very few messages: **avoid** λ.
  - The performance is very good.

Flexibility

- The Erlang code for PReach is simple.
  - The version described in the paper is about 1000 lines of code.
- This makes PReach a flexible platform for experiments:
  - Checking response properties: e.g. every request is eventually granted.
  - Exploiting symmetry: there are times we can verify a protocol for two or three nodes and conclude with certainty that it is correct for any number of nodes.
  - And others.
- Applications:
  - Used by Intel architects when exploring protocols for on-chip networks.
  - Used in other companies and universities.
  - It's been run on hundreds of machines with models of hundreds of billions of states.
  - Symbolic methods are faster than PReach for safety properties (showing that the model never reaches a bad state)
    - ★ PReach is faster for handling liveness properties: showing that some condition will eventually be satisfied.

Termination

- How do we know when we're done?
- Well, times up for this lecture.
- More seriously, in PReach we need to know when
  - When every worker process has an empty worklist,
  - **And** there are no messages in flight.
- Both conditions must hold at the same time.
  - This is the topic for Monday's lecture.

Summary

- PReach shows how the ideas from this class can be used to build real-world, high-performance, large-scale, parallel systems.
- Lessons learned:
  - Erlang is a great environment for building large-scale, parallel/distributed code.
  - Use the C/C++ call interface to use native C/C++ code for the compute intensive parts of the code.
    - ★ Erlang provides three such interfaces!
  - Watch out for overwhelm and crash
    - ★ If you're going to send a lot of messages, you need some kind of flow control mechanism.

Preview

April 3: Distributed Termination Detection

April 5: Party: 50^{th} Anniversary of Amdahl's Law

Review: for today's lecture

- To make a parallel implementation of a computation, we often need to identify a set (or many set(s)) of operations that can be performed in any order.
  - For model checking, what set of operations did we find that can be performed in any order? Does this exactly replicate the sequential version, or does it perform an equivalent computation?
  - Same questions as above, but for reduce.
  - Scan? Sorting? Matrix multiplication?
- Why did slow processes in PReach tend to become catastrophically slower? What was the solution?
- What is load balancing? Compare the load balancing mechanisms of PReach and Google's map-reduce.
- Is Erlang suitable for large-scale, high-performance, parallel computing? Why or why not?

Review: for March 29 lecture

- What is model checking?
- What is mutual exclusion?
- How does the model checker presented in the March 29 slides show that Dekker's algorithm guarantees mutual exclusion.
- Describe the role of the `knownStates` and `workList` data structures in the model checking algorithm.
- What is a guarded command?
- Write a murφ rule for another statement from Dekker's algorithm.