Our Model-Checker with MPI

Problems and Solutions

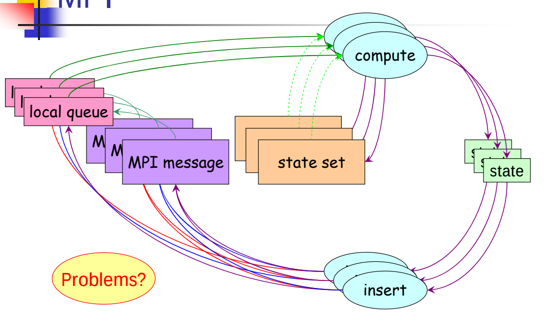
Alexandre David
1.2.05

Problems

- Design with message passing only.
  - No shared memory!
- Detecting termination.
  - How do you know you've finished?
  - No shared memory!
- Getting back the result.
  - How to get the trace?
  - Again, no shared memory!

MPI

Parallel computations.
Local queues.
Local state-sets.



Design

- Every state belongs to a process.
- The (global aggregate) state-set is partitioned and distributed on the processes.
- Partitioning done with the hash value. Uniform hash distribution → load balance.

Principles of Message-Passing Programming

Minimize interactions. Local accesses.

- 2 key attributes:
  - partitioned address space &
  - only explicit parallelization.

Expensive but costs are explicit.

- Logical view: *p* processes, each with its own exclusive address space.
  - Each piece of data must belong to a partition, i.e., explicit partitioned & placed.
  - All interactions require cooperation of two processes. Point to point communication.

Refined Design

- Non-blocking communication is desirable.
  - Can't afford to block for every state sent.
  - Use a local buffer to store states and try to send them when possible.
- ? Termination is a problem in itself, we need a protocol for that.
- The same holds for getting the result in the end (and showing progress).

Issues

- States may be generated many times by different processes but only one knows if they are visited or not!
  - Work-around: Cache.
- Termination: Normally a simple token protocol would work but not here!
  - When a process goes idle, it can receive more work later.
  - First try: Dijkstra's token algorithm (11.4.4).

Termination Detection: The Model

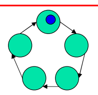
- A process is either active or inactive.
- An inactive process may not send messages.
- An active process may turn inactive.
- An inactive process stays inactive unless it receives a message.
- Find out when we can terminate.

What is the Problem?

- A message can turn an inactive process active.
  - You don't know if an inactive process will be turned active later...
- Find out whether all processes are inactive **and** whether there are no more messages in the system.
  - And avoid races, like message sent not yet received...

Simple Token Algorithm

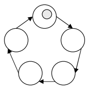
Processes arranged in a ring.



Process 1 inserts a token that will travel around back to 1.
The token leaves a process only if it's inactive.
Process 1 determines when to terminate.

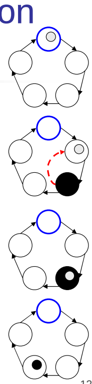
That does not work here:
- A process may become active after having sent the token.
- Who sent that message?
- Fix this: Dijkstra.

Dijkstra's Token Termination Detection Algorithm - Idea



- All processes are initially colored white.
- A process i sending a message to process j with j < i is a suspect for reactivating a process ⇒ It turns black.
- If a black process receives a token, it colors it black.

Dijkstra's Token Termination Detection Algorithm



1) When P_1 turns inactive, it turns white and sends a white token to P_2.
2) If P_i sends a message to P_j and $j < i$ then P_i turns black.
3) If P_i has the token and is idle, it passes the token. The token becomes black if P_i is black.
4) After passing tokens, processes become white.
5) The algorithm terminates when P_1 receives a white token and it is idle.

Cost

- The token consumes *O(P)* in time.
  - P_1 may become active again before getting back the token.
  - For a small number of processes, algorithm is acceptable.
  - For large numbers of processes, this becomes a significant overhead.
  - So far so good?

What Can Go Wrong Will Go Wrong

- What happens if P_i sends a message to P_j, j > i?
  - P_i may be white when it receives a white token later and forwards a white token. *Token faster than the message - race*.
  - Messages must be delivered in order for the protocol to work!
- MPI guarantees that messages are non-overtaking: M_1 sent before M_2 from **the same** process will arrive before M_2.
  - But no in-order guarantee!
  - Not good enough!

Dijkstra-Scholten Algorithm

1) Every process keeps a message count.
   1) Increment the count for received messages.
   2) Decrement the count for sent messages.
2) P_1 is the initiator and sends a white token with a count=0.
3) If P_i sends or receive messages, it turns black.
4) If P_i receives the token,
   1) it keeps it while it is active,
   2) if it is black, the token becomes black,
   3) when it is inactive, it forwards the token with its message count added and turns white.
5) If P_1 is white, it receives a white token, and the message count+its count == 0, then P_1 has detected termination.

Getting Back the Results

- When P_1 has detected termination, it can act as a master and
  - send a terminate message to everyone,
  - collect the results and print them,
    - Collecting the results could be done in parallel too!
  - send a shutdown message to everyone,
  - stop.