A Debugger for Flow Graph Based Parallel Applications

A. Al-Shabibi, S. Gerlach, R.D. Hersch, B. Schaeli
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland
http://dps.epfl.ch
Outline

• Flow graph based parallel applications
  – Dynamic Parallel Schedules framework
• Debugger architecture
• Features
  – Application monitoring capabilities
  – Influencing application execution
• Debugging example
• Scalability issues and future work
Dynamic Parallel Schedules

- Parallelization expressed as a directed acyclic graph of serial operations
  - Messages define edges between operations
- Operations contain arbitrary user code
Flow graph

- Four operation types
  - **Split**: 1 input, $n$ outputs
  - **Leaf**: 1 input, 1 output
  - **Merge**: $n$ inputs, 1 output
  - **Stream**: $m$ inputs, $n$ outputs

- The actual number of operations executed is only known at runtime
  - Unfolded flow graph
Threads

- Operations execute onto **threads**, grouped into **thread collections**
  - A state can be attached to each thread
Debugger architecture

• Separate Java program (the *debugger*) provides a GUI to interact with application execution

• Hooks within the framework code send notifications to the debugger
  – Thread collections and flow graph (upon creation)
  – Beginning / end of operation execution
    \[ \text{opStart} / \text{opStop} \]
  – Posting / reception of messages
    \[ \text{send} / \text{recv} \]
Debugger architecture

Network traffic within application

Network traffic with debugger
Notifications

- \texttt{opStart}, \texttt{opStop} and \texttt{send} notifications are blocking
  - Threads wait until acknowledgement is received from debugger before resuming execution
- Enables debugger to suspend execution on particular events
- Provides ordering guarantees
  - Causally dependent events are received in the correct order by the debugger
Notifications

- Threads share a single TCP connection to debugger
  - `opStart` received after `recv`
- Blocking notifications provide ordering guarantees at debugger
  - `recv` received after `send`
Notifications

• **send** notification includes message being sent
• **opStart** includes identifier of message being processed and operation identifier
• **opStop** includes identifier of operation
• **recv** includes identifier of message received

• The debugger links messages to operations and operations to messages
  – Draws unfolded flow graph as it is deployed
Global Step-by-Step

• Steps through flow graph level events
  – Allows user to quickly see how the flow graph unfolds

• Debugger holds all acknowledgements
  – Pressing the “Continue” button releases one acknowledgement per thread
  – Each thread is suspended again after sending the next blocking notification
Operation breakpoints

- Debugger generates an operation / thread matrix
  - Enables setting breakpoints on particular operation / thread combination
Message tracing

• Successor messages of breakpointed operation can be traced
  – Debugger automatically breakpoints all operations that process traced messages
  – Outputs of all subsequent operations are also traced

• Enables following a particular branch of the unfolded flow graph
  – All other operations keep running
Thread display

- Threads may be hidden to reduce amount of information displayed
  - Corresponding operation breakpoints are not shown
  - Operations running on hidden threads are not shown in flow graph view, and notifications from these threads are ignored
Operation level debugging

Attach serial debugger by double-clicking an operation in the flow graph view.
Message queues

- The state of queues is displayed within the debugger
  - Messages awaiting processing for the selected operation (*Op Queue*)
  - Messages awaiting processing on the thread (*Thread Queue*)
  - Lists of messages processed / sent by selected operation

```
Thread Queue    Op Queue   Processed   Sent
VectorDataObject (src: SplitVector on master[0], dest: Sort on sorters[0])
```
Message queues

- Messages can be reordered in thread and operation queues

- Enables testing for message races
  - i.e. changing the ordering of messages should not change the outcome of the program
Messages

- DPS provides automatic serialization
  - DPS messages are serializable C++ objects

- Binary serialization
  - Message transfers between application instances

- Textual serialization
  - Transfers between the application and the debugger
  - Includes member names and types
Message breakpoints

• We can set breakpoints based on message content
  – Data object type
  – Member name and value

• Detailed data object content displayed within the debugger
Serializable data objects

class VectorDataObject : public dps::AutoSerial
{
    CLASSDEF(VectorDataObject)
    MEMBERS
        ITEM(int, target)
        ITEM(std::vector<int>, v)
    CLASSEND;
};

VectorDataObject
element : std::vector<int, std::allocator<int>> : [183, 86, 1]
size : size_t : 100
target : int : 3
Message alteration

• When the debugger receives a *send* or *opStart* notification, message about to be sent or processed can be modified within the debugger

• The modified message is piggybacked onto the acknowledgment

• Thread deserializes the modified message and discards the original message
Thread states

- Thread states are serializable data objects
- If no operation is running on a thread, its state can be displayed and modified within the debugger
Neighborhood-dependent computation

Expected execution (1 iteration)
Neighborhood-dependent computation
Neighborhood-dependent computation
Neighborhood-dependent computation

Thread queue for Store[1]
- Split
- Send
- Send

Reordered messages for Store[1]
- Send
- Split
- Send
- Split

Thread queue for Store[1]
- Send
- Split
Scalability issues: performance

• Debugger receives all notifications
  + Has perfect knowledge about execution state
  + Provides fine grain control over execution

• Serializes reception and processing of notifications
  – Large slowdown when many short-lived operations or when large messages are sent
  – Slowdown grows with number of nodes
Scalability issues: performance

• Distribute breakpoint evaluation
  – Within debugger servers close to application instances
  – Directly within threads

• A thread starts sending notifications only when breakpoint is hit
  – Other threads keep executing

⇒ How to return to “full notification mode”? 
Scalability issues: information display

• Size of thread / operation matrix grows with number of threads and operations
  – Filter / collapse / search operation breakpoints

• Display of unfolded flow graph does not scale well

• Hidden threads are currently ignored by the debugger
  – Thread visibility should be independent of notification processing
Future work

• Leverage thread / application checkpointing
  – Restart application from known state rather than from beginning
    • Deeper look within a certain time frame
  – Compare checkpoints and highlight differences

• Knowledge about events causality enables consistent operation undo
  – Undoes caudally dependent message transfers and operations by reverting threads to previous checkpoints
Conclusion

• Message content modification and conditional breakpoints
• Message race testing through message reordering
• Flow graph representation provides instantaneous feedback about current execution state
  – Tracing of causally dependent messages for following a single flow graph branch
  – Operation breakpoints
  – Macro step-by-step
Thank you!

http://dps.epfl.ch