Application-level Checkpointing of Parallel Programs

Greg Bronevetsky, et al.
Cornell University

Presented by: Bryan Rabeler

Outline

- Motivation & Background
- Shared memory programs
- Distributed memory programs
- References
Motivation

- Trends in parallel computer systems:
  - Number of processors is increasing
  - Shift towards low-cost clusters
  - Running time of many applications is longer than the MTBF of hardware
- Must tolerate hardware faults in parallel systems…

Common Solutions

- Message Logging
  - All messages sent between processes are logged
  - On recovery, surviving processes replay messages to the failed process
  - **Advantage:** Restarts computation on failed process only, other processes continue
  - **Disadvantage:** Overhead is overwhelming, parallel programs communicate more than distributed programs
- Checkpointing
  - Periodically save state to stable storage
  - On recovery, all processes rolled back to the last checkpoint
  - **Advantage:** Time between checkpoints can be varied depending on reliability requirements
  - **Disadvantage:** All processes must roll back, state can be very large in massively parallel systems
Checkpointing Techniques

- System-level or Application-level
  - **System-level**: Entire state of the system is saved (impractical for massively parallel systems)
  - **Application-level**: Necessary state of the application is saved (complicates coding of application)

- Uncoordinated or Coordinated
  - Uncoordinated
    - No coordination among processes
    - Possible exponential rollback on restart
  - Coordinated
    - **Blocking**: All processes brought to a halt before taking the checkpoint
    - **Non-blocking**: All processes participate in taking each checkpoint while computation continues, requires coordination protocol

Fault Model

- Two common classes
  - Stopping (fail-stop)
    - Faulty process stops and fails to respond, does not send/receive messages
  - Byzantine
    - Process makes computational errors at random and continues to function
Shared vs. Distributed Memory

- **Shared Memory**
  - Machine 1: P0, P1, P2, Cache → RAM

- **Distributed Memory**
  - Machine 1: P0, P1, P2, Cache → RAM
  - Machine 2: P0, P1, P2, Cache → RAM
  - Machine 3: P0, P1, P2, Cache → RAM
  - Network

---

Distributed Memory Systems

- **Machine 1**
  - Program → TreadMarks → Message Passing → Network
  - Software DSM

- **Machine 2**
  - Program → TreadMarks → Message Passing → Network

- **Machine 1**
  - Program → Message Passing (MPI) → Network

- **Machine 2**
  - Program → Message Passing (MPI) → Network

- **Message Passing System**
Outline

- Motivation & Background
- **Shared memory programs**
- Distributed memory programs
- Conclusion

OpenMP in a Nutshell

- Fork/join model

- All variables are either shared or private
  - **Shared**: All threads read from one address
  - **Private**: Each thread has a local copy
C³ for OpenMP: System Overview

- Programmer annotates possible checkpoint locations
  - Call to `potentialCheckpoint()`
- C³ pre-compiler transforms source code to include checkpointing code
- Compiled with native compiler, linked with coordination layer library
  - Layer sits between app and OpenMP
  - No modification to OpenMP

### Blocking Protocol

#### Checkpointing:
- Each thread calls a barrier
- Each thread saves its private state, thread 0 saves the system’s shared state
- Each thread calls a second barrier

#### Recovery:
- All threads restore private variables to their checkpointed values, thread 0 restores all the shared addresses to their checkpointed values
- Every thread calls a barrier
- Every thread continues execution
Saving Application State

- **Heap**
  - Custom heap library tracks memory that is allocated and freed

- **Call stack**
  - **Location Stack (LS):** Tracks sequence of function calls which lead to place where checkpoint was taken
  - **Variable Description Stack (VDS):** Records local variables in these function invocations that must be saved
  - On recovery:
    - LS is used to re-execute sequence of function calls and re-create stack frames
    - VDS is used to restore variables into stack

- **Global Variables**
  - Similar approach to VDS

---

**Example #1**

```c
main() {
    int a;
    VDS.push(&a, sizeof(int));
    if(restart)
        load LS;
        copy LS to LS_Old;
        jump dequeue(LS_Old);
    ...
    func1();
    ...
    label_0:
    func2();
    LS.pop();
    ...
    omp_set_num_threads(read_original_num_threads());
    #pragma omp parallel
    { parallel code }
    ...
    VDS.pop();
}
```

```c
func1() {
    ...
}
```

```c
func2() {
    int b;
    VDS.push(&b, sizeof(int));
    if(restart)
        jump dequeue(LS.old);
    ...
    LS.push(label_1);
    potentialCheckpoint();
    label_1:
    if(restart)
        load VDS;
        restore variables;
        LS.pop();
    ...
    VDS.pop();
}
```
Example #2

```c
main() {
    int a;
    VDS[0].push(&a, sizeof(int));
    if(restart)
        load LS[0];
    copy LS[0] to LS_Old[0];
    jump dequeue(LS_Old[0]);
    ...
    LS[0].push(label_0);
    label_0:
    omp_set_num_threads(read_original_num_threads());
    #pragma omp parallel
    {
        int b;
        VDS[thread_num].push(&b, sizeof(int));
        if(restart)
            jump dequeue(LS_Old[thread_num]);
        ...
        LS[thread_num].push(label_2);
        potentialCheckpoint();
        label_2:
        if(restart)
            load VDS[thread_num];
            restore variables;
            LS[thread_num].pop();
            ...
        VDS[thread_num].pop();
    }
    LS[0].pop();
    ...
    VDS[0].pop();
}
```

Synchronization: Barriers

- OpenMP barriers will match calls in threads even if not at the same source code location
- In example, threads 1 & 2 take a checkpoint while thread 0 continues computing
- On recovery, OpenMP barrier semantics violated
Solution for Barriers

- Ensure no checkpointing region ever crosses an application barrier
  - Associate a `potentialCheckpoint()` call with every call to an application barrier
- Problem: By the time a thread decides to take a checkpoint, thread 0 may already be blocked on its application barrier
  - Introduce a global `checkpointFlag` variable

```c
void ccc_barrier(){
    #pragma omp barrier
    while(checkpointFlag){
        // only do this if checkpoint started while waiting on application barrier
        save application state
        checkpointFlag=FALSE
        #pragma omp barrier

        // trying to wait on application barrier again
        #pragma omp barrier
    }
}
```

```c
void potentialCheckpoint(){
    // update checkpointFlag
    #pragma omp flush(checkpointFlag)
    // if time to checkpoint or others checkpointed
    if (checkpointFlag or initiateCheckpoint()){  
        checkpointFlag = true;
        #pragma omp barrier
        save application state
        checkpointFlag = FALSE
        #pragma omp barrier
    }
}
```
Synchronization: Locks

- Problem is similar to that of barriers
  - Additional complexity: threads holding locks at checkpoint time must hold the same locks on recovery

Solution for Locks

- Associate a `lockCheckpointFlag` with every lock
- Before first barrier of checkpoint, thread will:
  - Set each lock’s `lockCheckpointFlag` to TRUE
  - Remember which locks it is holding and release them
- Upon lock acquisition, thread will check value of the lock’s `lockCheckpointFlag`
  - If FALSE, lock acquired normally
  - If TRUE, must take a checkpoint
- On recovery:
  - All `lockCheckpointFlag` set to FALSE
  - Each thread reacquires the locks it had before the checkpoint
Solution for Locks (Cont.)

```c
ccc_set_lock(lock){
  omp_set_lock(lock)
  while(lock.lockCheckpointFlag){
    // only do this if checkpoint started while
    // waiting to acquire lock
    #pragma omp barrier
    for all held locks
      lock.lockCheckpointFlag=TRUE
    record which locks are being held
    release all locks
    save application state
    save lock state
    for all locks that were held
      reacquire lock
      lock.lockCheckpointFlag=FALSE
    #pragma omp barrier

    // try to acquire the lock again
    omp_set_lock(lock)
  }
}
```

```c
potentialCheckpoint(){
  #pragma omp barrier
  for all held locks
    lock.lockCheckpointFlag=TRUE
  remember which locks are held
  release all locks
  save application state
  save lock state
  for all locks that were held
  reacquire lock
  lock.lockCheckpointFlag=FALSE
  #pragma omp barrier

  Threads
  0
  Lock/Unlock
  1
  Lock/Unlock
```

Execution Time Overhead

![SPLASH-2 Linux Experiments](image)
Checkpoint Sizes

![Checkpoint Sizes Graph](image)

Checkpointing & Recovery Time

![Checkpointing & Recovery Time Graph](image)
Outline

- Motivation & Background
- Shared memory programs
- Distributed memory programs
- Conclusion

MPI in Nutshell

- All processors execute the same program
  - Only communication is via message passing
C³ for MPI: System Overview

- Programmer annotates possible checkpoint locations
  - Call to `potentialCheckpoint()`
- C³ pre-compiler transforms source code to include checkpointing code
- Compiled with native compiler, linked with coordination layer library
  - Layer sits between app and MPI
  - No modification to MPI

Assumptions

- Fail-stop fault model
- Reliable communication channels
- Communication channels are not FIFO at the application level
  - MPI processes can use tag matching
Epochs & Message Classification

Delayed State Saving

- System-level checkpoints may be taken at any time
  - Use scheduling to avoid early messages
- Application-level checkpoint can only be taken at `potentialCheckpoint()` calls
  - Checkpoint delayed until call to `potentialCheckpoint()` reached
  - Must handle both early and late messages
Late and Early Messages

- P will not resend late message to Q
  - Identify late messages and save them with the checkpoint
  - Replay late messages to receiving process during recovery
- Q will resend early message to R
  - Identify early messages
  - Ensure early messages are not resend during recovery
  - Problem with non-deterministic events

Non-blocking Protocol

- **Phase 1**: Initiator process sends control message *pleaseCheckpoint* to all processes
- **Phase 2**: At some point each process takes a checkpoint
  - Saves local state & early messages
  - Starts logging all late messages received and all non-deterministic decisions it makes
  - Once all late messages received, sends control message *readyToStopLogging* back to initator, but keeps logging non-deterministic decisions
Non-blocking Protocol (Cont.)

- **Phase 3:** When initiator process receives control message `readyToStopLogging` from all processes, it sends control message `stopLogging` to all processes.
- **Phase 4:** All processes stops logging
  - Occurs when either:
    - A process receives control message `stopLogging` from the initiator
    - A process receives a message from another process that it has stopped logging
  - All processes send control message `stoppedLogging` to initiator
  - Once initiator receives `stoppedLogging` message from all processes, it terminates the protocol

Piggybacked Information on Messages

- Values piggybacked on all messages:
  - `epoch` (integer): The current epoch that the process is in
  - `amLogging` (boolean): True when the process is logging, false otherwise
  - `nextMessageID` (integer)
    - Initialized to 0 at beginning of each epoch
    - Incremented for each message sent
    - Uniquely identifies messages sent by a given process in a given epoch
How do we know when all late messages are received?

- In each epoch
  - Process P maintains how many messages sent to every other process Q: \( \text{sendCount}(P \rightarrow Q) \)
  - Process Q maintains how many messages it received from every other process P: \( \text{receiveCount}(Q \leftarrow P) \)
- P sends a \textit{mySendCount} message to other process upon taking a checkpoint
  - Contains number of message sent to them in previous epoch
  - Q compares with value of \( \text{receiveCount}(Q \leftarrow P) \)

How do we suppress early messages on recovery?

- A process determines a message is early by comparing epoch numbers
  - Logs the pair \(<\text{sender},\text{messageID}>\) at each checkpoint
  - Retrieved from storage on recovery by the receivers
  - Senders are informed of the messageIDs so that resending can be suppressed
Saving State

- Technique to take local checkpoints is independent of the coordination protocol
- Almost identical to technique for shared memory (OpenMP)
  - No shared variables
  - MPI library state
    - Use level of indirection & pseudo-handles
    - Different techniques for transient and persistent objects

Execution Time Overhead

- Graphs showing running time overhead for Dense Conjugate Gradient, Laplace Solver, and Neurons.

The number above each set of bars is the size of the application state for that problem size.
Reducing Checkpoint Size

- Avoid saving dead and read-only variables
- Detect distributed redundant data
- Re-compute instead of saving

References