EuroMPI'15, Bordeaux, France, September 21-23, 2015

Scalable and Fault Tolerant Failure Detection and Consensus

Amogh Katti, Giuseppe Di Fatta,
University of Reading, UK

Thomas Naughton, Christian Engelmann
Oak Ridge National Laboratory, USA

**Funded By:** Felix Scholarship and US Department of Energy, Advanced Scientific Computing Research
Outline

• Motivation
• Overview of related work
• Proposed approach
• Experimental Results
• Conclusion
• Future work
Motivation

- The need for resilience in High Performance Computing (HPC)
- Algorithm Based Fault Tolerance (ABFT) can help
- ABFT needs a fault tolerant MPI
- User Level Failure Mitigation (ULFM) is being proposed
- MPI_Comm_shrink and MPI_Comm_agree need a failure detection and consensus protocol
The need for resilience in HPC

• Resilience is a critical challenge
  • Increasing component count, decreasing component reliability and increasing software complexity
  • Parallel application correctness and efficiency are essential for success of extreme-scale systems

• Cost effective, hardware and software cooperative resilience approach necessary

• Global checkpoint-restart, the dominant resilience strategy, will be less efficient at scale
ABFT can help

• Application-specific techniques, like Algorithm Based Fault Tolerance (ABFT), can be more effective
• Loss of application state can be dealt with through reconfiguration and adaptation
• ABFT applications incorporate the needed fault tolerance logic
• Some ABFT techniques that can be used:
  • Error correction using data redundancy or encoding
  • Re-execution using local checkpoints
ABFT needs a fault tolerant MPI

• Failure detection and notification
• Reconfiguration without global restart based on consensus on detected failures

User Level Failure Mitigation

- MPI’s Fault Tolerance Working Group (FTWG) has proposed User Level Failure Mitigation (ULFM)
- Specifies semantics/interfaces for an MPI implementation’s behaviour in the presence of process failures
  - Fail-stop: Failed processes stop communicating
- In a conformant MPI implementation:
  - No operation hangs in the presence of failures but completes by returning an error
  - Global knowledge of failures can be achieved whenever necessary
- Local failure detection left to the implementations

ULFM requires a failure detection and consensus protocol

- **MPI_Comm_shrink**
  - Creates a new communicator by excluding the failed processes in the old communicator
  - An agreement is reached on the failed processes

- **MPI_Comm_agree**
  - Agrees on a value among the non-failed processes

- Both operations need to be supported by a fault-tolerant failure detection and consensus algorithm

Related work in failure detection and consensus protocols

• Coordinator based protocols
  • Assume failures to be pre-detected at each process
  • Use consensus algorithm to achieve consistent failure detection
  • Typically good log-based scaling
  • Not completely fault tolerant (failures occurring during the failure detection are not detected)

• Completely distributed Gossip-based protocols
  • Consistent failure detection in phases:
    • Failure suspicion, failure detection and consensus
  • Very poor scalability
  • Completely fault tolerant
Approach

• Gossip-based failure detection and consensus
• Assumptions
• Algorithm 1: Consensus using global knowledge
• Algorithm 2: Efficient heuristic consensus
Gossip-based failure detection and consensus

• Gossiping is a randomized communication scheme
• Gossip-based protocols are intrinsically fault tolerant and extremely scalable
• Two Gossip-based failure detection and consensus algorithms are proposed
  • Maintaining global knowledge
  • Efficient heuristic consensus
• Based on a combined method for detecting failures locally and quickly disseminating detections to achieve consensus using Gossip
Assumptions

- Detects fail-stop failures
- Reliable communication medium
- Failures are permanent
- Synchronous system with bounded message delay
- Failures during the algorithm will stop at some point to allow the algorithm to complete with successful consensus detection
- A process once detected as failed is detected to have failed by all the processes eventually
Algorithm 1: Consensus using global knowledge

• At each process $p$ $F_p[n, n]$, where $n$ is system size, is maintained
  • $F_p[r, c]$ is the view at process $p$ of the status of process $c$ as detected by process $r$. 0 if alive; 1 otherwise

• Algorithm executed at each process
  • Initialization – assume all processes are alive
  • At each Gossip cycle
    • Direct failure detection using stochastic pinging
      • Send PING gossip message with $F_p$ to a random process and post a timeout event for receiving REPLY gossip message
      • Timeout event and no reply received - direct failure detection of the PINGed process
      • Update $F_p$ to reflect the failure detection
    • Gossip reception event – Merge Fault Matrices (Indirect local failure detection and propagation)
    • Consensus detection – When all fault-free processes detect the failed process
Algorithm 2: Efficient heuristic consensus

- At each process p Fault list \( L_p = \{< r, ccnt >,...\} \) is maintained.
  - An entry in this list is a 2-tuple \(< r, ccnt >\), where \( r \) is the rank of the failed process and \( ccnt \) is the consensus count associated with it.

- Algorithm executed at each process
  - Initialization – assume all processes are alive
  - At each Gossip cycle
    - Direct failure detection using stochastic pinging
      - Send PING gossip message with \( L_p \) to a random process and post a timeout event for receiving REPLY gossip message
      - Timeout event and no REPLY received - direct failure detection of the PINGed process
      - Add the pinged process to \( L_p \) with \( ccnt \) set to 0
    - Gossip reception event – Merge Fault Lists (Indirect failure detection and propagation)
    - Consensus detection – Wait for \( \log(n) \) number of cycles
      - When \( ccnt \) for an entry \(< r, ccnt >\) reaches MIN_CCNT, which indicates with a high probability that the failed process \( r \) is recognized by all the processes, consensus on failure of \( r \) is reached.
Results

• Overview of implementations
• Overview of the use of xSim
• Overview of the hardware environment
• Overview of the simulative environment
• Results for Algorithm 1
• Results for Algorithm 2
• Comparison (Algorithm 1 and Algorithm 2)
Overview of implementations

• Algorithms have been implemented as MPI applications using point-to-point operations
• Fault Matrix in algorithm 1 implemented as integer matrix
• Failed process id in algorithm 2 implemented as an integer
• Failures were simulated by restraining a process from participating in communications
Overview of the use of xSim

• To evaluate the algorithms at significantly larger scale than the available physical system

• Extreme-scale Simulator (xSim) is an application performance and resilience investigation toolkit

![Diagram](image)

Fig. 1. xSim's implementation architecture and design.

Overview of the hardware environment

• Experiments on the Linux cluster computer:
  • One head node and 16 compute nodes
  • Head node has two AMD Opteron 4386 3.1 GHz processors with eight cores/processor and 64 GB RAM
  • Compute nodes have one Intel Xeon E3-1220 3.1GHz processor with four cores/processor and 16 GB RAM
  • Nodes are connected by Gigabit Ethernet
  • System is running the Ubuntu 12.04 LTS operating system and Open MPI 1.6.5
Overview of the simulation environment

• Simulations using the Extreme-scale Simulator (xSim) atop the Linux cluster
  • One simulator MPI process per physical processor core
  • Multiple simulated MPI processes per simulator MPI process (oversubscription)
  • Processor model is set with a 1-to-1 performance match to the physical AMD processor core
  • Network interconnect model with a basic star topology, 1 us link latency, and infinite bandwidth
  • Processor and network models are set to evaluate the algorithms, and not the system the algorithm runs on
Results for Algorithm 1: Consensus using global knowledge

Consensus on single failure with $2^4$-$2^{11}$ MPI ranks

Consensus on four failures with $2^4$-$2^{11}$ MPI ranks

Figure 4: Number of cycles to achieve global consensus after a single failure injection (algorithm 1)

Figure 7: Number of cycles to achieve global consensus after multiple (4) failures, which were injected before algorithm execution (algorithm 1)
Results for Algorithm 1: Consensus using global knowledge

- Exponential consensus propagation
- Asynchronous consensus detection

Figure 5: Local consensus progress at a process after a single failure injection for system size of 2048 (algorithm 1)

Figure 6: Consensus detection spread for a system size of 2048 (algorithm 1)
Results for Algorithm 1: Consensus using global knowledge

Fault tolerant consensus propagation and detection

Figure 8: Number of cycles to achieve global consensus with multiple (4) failures, which were injected during algorithm execution (algorithm 1)
Results for Algorithm 2: Efficient heuristic consensus

Consensus on single failure with $2^4$-$2^{20}$ MPI ranks

![Graph](image)

Figure 9: Number of cycles to achieve global consensus after a single failure injection (algorithm 2)
Comparison of Algorithm 1 vs. Algorithm 2

Consensus on single failure with $2^2-2^{20}$ MPI ranks

Figure 10: Total bandwidth utilization of the consensus algorithms with a single failure injection
Conclusion

• Failure detection and consensus for a fault-tolerant MPI enable HPC applications to adopt ABFT
• Two novel Gossip-based failure detection and consensus algorithms were presented
  1. Global knowledge at each process
  2. Efficient heuristic consensus
• Results confirm their scalability and fault tolerance
• The second algorithm uses significantly lower memory and bandwidth and achieves a perfect consensus synchronization
Future work

• Better method to efficiently detect consensus
• Mechanisms to avoid false positives
• Further experimental analysis and comparison with other methods
Questions?