Journal Title
XX(X):1–13
©The Author(s) 2015
Reprints and permission:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Epidemic Failure Detection and Consensus for Extreme Parallelism

Amogh Katti[1], Giuseppe Di Fatta[1], Thomas Naughton[2] and Christian Engelmann[2]

Abstract

Future extreme-scale high-performance computing systems will be required to work under frequent component failures. The MPI Forum's User Level Failure Mitigation proposal has introduced an operation, *MPI_Comm_shrink*, to synchronize the alive processes on the list of failed processes, so that applications can continue to execute even in the presence of failures by adopting algorithm-based fault tolerance techniques. This *MPI_Comm_shrink* operation requires a failure detection and consensus algorithm. This paper presents three novel failure detection and consensus algorithms using Gossiping. Stochastic pinging is used to quickly detect failures during the execution of the algorithm, failures are then disseminated to all the fault-free processes in the system and consensus on the failures is detected using the three consensus techniques. The proposed algorithms were implemented and tested using the Extreme-scale Simulator. The results show that the stochastic pinging detects all the failures in the system. In all the algorithms, the number of Gossip cycles to achieve global consensus scales logarithmically with system size. The second algorithm also shows better scalability in terms of memory and network bandwidth usage and a perfect synchronization in achieving global consensus. The third approach is a three-phase distributed failure detection and consensus algorithm and provides consistency guarantees even in very large and extreme-scale systems while at the same time being memory and bandwidth efficient.

Keywords

Fault-tolerant MPI, User-level failure mitigation, Failure detection, Consensus, Gossip protocols

1 Introduction

Resilience[1] is a critical challenge as high-performance computing (HPC) systems continue to increase component counts, individual component reliability decreases (due to shrinking process technology[2] and near-threshold voltage operation[3]), and software complexity increases[4]. In spite of frequent faults, errors and failures, parallel application correctness and execution efficiency are essential to ensure the success of extreme-scale HPC systems.

As cost constraints limit resilience mitigation in hardware, a cooperative approach between hardware and software is needed to efficiently mitigate faults, errors, and failures at the appropriate layer. However, application-level checkpoint/restart has been the dominant HPC fault tolerance method for decades. Every detectable uncorrectable error results in a complete application abort and restart from previously saved checkpoint state, even if the error could have been more efficiently handled by the application, e.g., using forward error correction with erasure codes[5].

Algorithm-based fault tolerance (ABFT)[5–7] may be able to deal with loss of application state, such as caused by a failing compute process or node, more efficiently through reconfiguration and adaptation without the need for a more drastic recovery measure, such as a global rollback. The employed fault tolerance techniques by the application may include error correction using data redundancy or encoding, and re-execution using local checkpoints.

The Message Passing Interface (MPI) is the dominant parallel programming interface for facilitating communication between compute processes in HPC. Despite its popularity, MPI is not fault tolerant. Recent efforts in MPI fault tolerance focused on user-level failure mitigation (ULFM)[8]. The proposed ULFM extensions to the MPI standard enable applications to be notified of MPI process failures, to create a new MPI communicator object that excludes known failed MPI processes from further communication using the *MPI_Comm_shrink()* operation, and to achieve a uniform agreement on a value among the non-failed MPI processes using the *MPI_Comm_agree()* operation.

This paper particularly focuses on the implementation of the proposed *MPI_Comm_shrink()* operation and its need to perform an agreement on the group of failed

[1]Department of Computer Science, University of Reading, Reading, RG6 6AY, UK. {a.p.katti, G.DiFatta}@reading.ac.uk
[2]Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831-6173, USA. {naughtont, engelmannc}@ornl.gov

This work was sponsored by the U.S. Department of Energy's Office of Advanced Scientific Computing Research. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

Corresponding author:
Amogh Katti, Department of Computer Science, University of Reading, Reading, RG6 6AY, UK.
Email: a.p.katti@reading.ac.uk

MPI processes among the non-failed MPI processes, even while MPI process failures occur. More precisely, the new MPI communicator object created by the collective *MPI_Comm_shrink()* operation contains a consistent group of MPI processes at every participating MPI process that excludes at least every failed MPI process that has previously raised a failure notification to the application. MPI processes that fail during the *MPI_Comm_shrink()* operation may be excluded as well, but consistently at every participating MPI process.

According to the ULFM proposal, only fail-stop process failures are considered; when a process fails, it stops communicating with the rest of the processes. The method of failure detection is not defined. However, the ULFM proposal specifies that an operation involving a failed MPI process must always complete in a finite amount of time. If an operation does not involve a failed MPI process, it must not raise a MPI process failure exception. This provides implementers with different options for failure detection. For example, a correct MPI implementation may provide failure detection only for MPI processes involved in an ongoing operation and may postpone detection of other failures until necessary. The knowledge about detected failed MPI processes is local and only constructed as globally consistent state in the form of a new MPI communicator object using the *MPI_Comm_shrink()* operation, thus requiring a fault-tolerant consensus algorithm to uniformly agree on the group of failed MPI processes.

This work investigates the use of Gossip-based protocols to detect failures and disseminate this information in a consistent manner to support the *MPI_Comm_shrink()* operation. The proposed consensus algorithms can also be used to support the *MPI_Comm_agree()* operation.

Epidemic (or Gossip-based) protocols are a robust and scalable communication paradigm to disseminate information in a large-scale distributed environment using randomized communication. They have the advantage of inherent robustness and scalability with respect to global communication schemes based on deterministic communication patterns. Applications and services based on Gossip-based protocols for large- and extreme-scale systems have been proposed in many fields of distributed computing. Recently, Gossip-based protocols have also been applied in the context of HPC[9–11].

In the paper[12], two Gossip based failure detection and consensus algorithms using randomized pinging were developed and tested by means of simulations. Stochastic pinging is used to quickly detect failures and disseminate them to fault-free processes during Gossip. The first algorithm stores the system view in a matrix at each process to facilitate consensus detection. It detects failures before and during the execution of the algorithm and is hence completely fault tolerant. The second algorithm detects consensus on the failed processes using a heuristic method based on a list of failed processes, thus improving the memory scalability with respect to the first algorithm. It also transfers less Gossip data as only the list of failed processes is sent and hence consumes negligible network bandwidth. This paper complements and extends the work in [12] by adding a third algorithm. The third algorithm presented in this paper is analogous to the three-phase commit[13] protocol but it is completely distributed. It accurately detects consensus on the failures using a Gossip-based aggregation protocol while still maintaining only the list of failed processes. Thus it also has the high memory scalability and consumes less network bandwidth than the first algorithm. For all algorithms, the number of Gossip cycles to detect consensus scales logarithmically with the system size.

The paper is structured as follows. Section 2 details the proposed Gossip-based failure detection and consensus algorithms, where Section 2.1 focuses on failure detection using stochastic pinging, Section 2.2 on achieving consensus using global knowledge and a deterministic consensus detection method, and Section 2.3 on achieving consensus using a heuristic consensus detection method. Section 3 introduces the three-phase consensus detection method on the occurred failures. Section 4 discusses the algorithms' performance and presents experimental results. Section 5 discusses related work. Section 6 concludes the paper with a summary and a discussion of future work.

2 Gossip-based Failure Detection and Consensus

The *MPI_Comm_shrink()* operation must implement a consensus algorithm that achieves agreement on the set of failed MPI processes that have previously raised a failure notification, i.e., on the group of known failed MPI processes at each participating MPI process at the start of the operation. MPI process failures that occur during the operation will eventually be detected and corresponding failure notifications will be raised during successive MPI communication operations. To avoid coarse-grain iterative agreement on the group of failed MPI processes with successive calls to *MPI_Comm_shrink()* by an application, the consensus algorithm may additionally include an agreement on the group of MPI processes that fail during the *MPI_Comm_shrink()* operation, i.e., during the consensus algorithm.

In both cases, the consensus algorithm needs to be fault tolerant, i.e., deal with already known or newly detected MPI process failures. Also, in both cases, a complete failure detector is implemented that detects fail-stop MPI process failures by combining failure detection and consensus. Every MPI process independently detects MPI process failures. Consensus on the failed MPI processes is then achieved by aggregating these MPI process failure detections with the help of a consensus algorithm.

A straightforward way to detect MPI process failures during the *MPI_Comm_shrink()* operation is pinging, wherein a process asks another whether it is alive. A reply indicates a positive response, while a failure notification from the underlying MPI runtime marks a negative response. The failure detector in the MPI runtime can be based on a simple communication timeout. In an alternative method, a process periodically sends a heartbeat message to let another process know that it is alive. The receiving process monitors the incoming heartbeat messages and marks a process as failed upon a failure notification from the underlying MPI runtime. The failure detector can be based on a simple communication timeout for the periodic heartbeat. The work presented in this paper is based on pinging for failure detection during the *MPI_Comm_shrink()* operation.

The consensus during the *MPI_Comm_shrink()* operation involves all fault-free processes agreeing on the group of failed processes. In general, a solution to the consensus problem exists only in certain environments[14]. For instance, consensus is not possible in completely asynchronous environments[15]. However, in an asynchronous environment, failures can still be detected with completeness and accuracy, leading to a uniform view of the system at each process[16], although, group membership may not be agreed upon[17]. In the context of MPI, the proposed ULFM extensions are based on realistic assumptions, such as fail-stop, no recovery, a synchronous model, and only short periods with exceptionally high MPI process failure rates.

This paper presents three scalable failure detection and consensus algorithms for *MPI_Comm_shrink()* based on Uniform Gossiping. The Gossip messages are implicitly used to implement stochastic failure detection. Consensus is achieved and detected by maintaining the system state in a matrix or by maintaining a failed process list at each process. Gossip messages carry failure information with them and disseminate known failures at exponential speed. When a process p sends a Gossip message to process q, this process q comes to know about the failures that were directly detected by p, thus detecting failures indirectly. This Gossip message from p not only contains the failures directly detected by p, but also indirectly detected failures through received Gossip messages. These indirect detections of process p are propagated as well to process q, resulting in exponential information dissemination. When the failure information of a process is disseminated to all fault-free processes, consensus on its failure is reached in a logarithmic number of Gossip cycles (as shown in the experiments in Section 4).

The proposed failure detection and consensus algorithms work under the following assumptions.

1. Processes are assumed to be connected by a reliable communication medium.
2. A synchronous system model is assumed, i.e., a non-failed MPI process responds to a message within a known, finite amount of time.
3. The fail-stop model is assumed, i.e., a failed MPI process stops communicating.
4. Faults are assumed to be permanent, i.e., a failed MPI process does not recover.
5. A process once detected as failed is detected to have failed by all the processes eventually.
6. Periods of system stability are assumed, i.e., MPI process failures during the consensus algorithm will, at some point, stop for a long enough period to reach consensus.

2.1 Failure Detection using Stochastic Pinging

In this Section, we discuss the MPI process failure detection feature of a *MPI_Comm_shrink()* operation using stochastic pinging as part of a Gossip-based protocol. Every process independently detects failures by pinging a random process periodically. During a Gossip cycle of length T_{Gossip} units, a process i randomly selects a process j to ping according to a uniform probability distribution function. If process j replies by the end of the current Gossip cycle, then process i finds it to be alive; failed otherwise. Figure 1 shows the pseudocode for the algorithm.

```
At process $i$
At each cycle (every $T_{Gossip}$ time units):
1    $j = getRandomProcess()$
2    send a ping message to $j$
At event: received a ping message from $j$:
3    send a reply message to $j$
At event: timeout without receiving a reply from $j$:
4    mark $j$ to have failed
```

Figure 1. Failure detection using pinging

During a cycle of Uniform Gossiping, the probability of a process being selected as destination of zero, one or more ping messages follows a binomial distribution and hence places minimum burden on each process in turn increasing the scalability of failure detection. Ultimately any failed process is quickly detected by one or more of the non-failed processes, thus initiating the epidemic exponential propagation of information to achieve consensus. The adopted Gossip-based approach can tolerate moderate-to-low message loss rates and delays as it is intrinsically fault tolerant.

2.2 Consensus using Global Knowledge

In this Section, we discuss achieving consensus on the set of failed MPI processes during the *MPI_Comm_shrink()* operation by maintaining global knowledge at each MPI process. As shown in the algorithm of Figure 2 each process p detects failures by pinging random processes and maintains a fault matrix F_p to store the status of all processes as believed by it and also by all the other processes. An entry $F_p[i, j]$ in a process's fault matrix indicates the status of process j as detected by process i (1 if detected to have failed; 0 otherwise). The algorithm can be divided into four logical tasks according to the functions performed: (1) initialization, (2) failure detection, (3) fault matrix update, and (4) check for consensus.

Every process initializes with the assumption that every other process in the system is alive and no other process has yet detected any failures (lines 1-5 of Figure 2).

To detect failures, every T_{Gossip} time units a process p randomly selects a process q and sends a ping message to it piggybacking the fault matrix F_p (lines 6-7). Sending the entire matrix facilitates the propagation of not only process p's detections but also other processes' detections known to process p, thus exponentially propagating information. A timeout event is then set to receive a reply from q during the current Gossip cycle (line 8). An asynchronous reply follows upon reception of this ping message (lines 20-22). At the expiry of the current Gossip cycle if a reply message from q has not been received by p, q is detected (directly) to have failed (line 32).

Upon reception of a Gossip (ping or reply) message at p from r, the local fault matrix F_p is updated by performing an OR operation of the corresponding elements in F_r except the row p (lines 23-31 except line 28). The row p in F_p is updated to include the detections of process r (row r in F_r) and thus performing indirect failure detections at p (line 28).

```
At process $p$
Require: $T_{Gossip}$ Length of a Gossip cycle,
         $T_{out}$ Timeout period
Initialisation:
        //Fault Matrix $F_p[j,k]$ where $0 \leq j,k < n$
        //$F_p[j,k]$ - status of process $k$ as detected by process $j$
1       for $(j = 0, j < n, j++)$
2           for $(k = 0, k < n, k++)$
3               $F_p[j,k] = 0$
4           endfor
5       endfor
At each cycle (every $T_{Gossip}$ time units):
        //failure detection using pinging
6       $q = getRandomProcess()$
7       send a message of type ping to $q$ piggybacking $F_p$
8       create timeout event $E_q =< curr\_cycle\_no + T_{out}, q >$
         for receiving reply message from $q$
9       for $(k = 0, k < n, k++)$ //check for consensus on $k$
10          $temp = 0$
11          for $(j = 0, j < n, j++)$
12              if $(Fp[j,k] || Fp[p,j])$
13                  $temp = temp + 1$
14              endif
15          endfor
            //consensus is reached when all fault-free
            //processes have detected the failed process
16          if $(temp == n)$
17              $consensus\_reached(k)$
18          endif
19      endfor
At event: received a message from $r$ piggybacked
          with $F_r$:
20      if$(message\_type == ping)$
21          send a message of type reply to $r$ piggybacking $F_p$
22      endif
        //merge the fault matrices
23      for $(k = 0, k < n, k++)$
24          for $(j = 0, j < n, j++)$
                //propagation of remote failure detections
25              if$(j \neq p)$
26                  $F_p[j,k] = F_p[j,k] || F_r[j,k]$
27              else
                    //indirect local failure detection
28                  $F_p[p,k] = F_p[p,k] || F_r[r,k]$
29              endif
30          endfor
31      endfor
At event: timeout $E_q$ and no reply message received
          from $q$ within $T_{out}$:
        //mark $q$ to have failed (direct failure detection)
32      $F_p[p,q] = 1$
```

Figure 2. Failure consensus by global knowledge (algorithm 1)

Finally, to check if consensus has been reached on the failure of a process k at p, a logical OR operation is performed between the corresponding elements of the k^{th} column of the fault matrix and its p^{th} row. Consensus is reached when all fault-free processes have detected the faulty one (lines 9-19).

In this algorithm processes have to maintain local knowledge of the entire system state, with $O(n^2)$ memory requirement, in order to check for consensus. Network bandwidth consumption is also high due to the transfer of this local system state as part of the Gossip. Designing scalable consensus algorithms, which achieve consensus without maintaining global knowledge at each process and with moderate network utilization, is a challenging task. Previous work[18] has shown how to detect convergence in epidemic aggregation by means of heuristic methods, such as comparing local state with a random sample of remote states or running multiple protocol instances locally. In the next section a similar heuristic method for consensus detection with a more efficient memory and bandwidth utilization is presented.

2.3 Efficient Heuristic Consensus

Maintaining a matrix of size n^2 elements, where n is the number of processes in the system, and sending it as part of the Gossiping consumes a lot of memory and network bandwidth. Storing only the failed processes in a list and sending the same while Gossiping can avoid this. The algorithm in Figure 3 uses a list instead of a matrix and detects consensus using an heuristic method.

```
At process $p$
Require: $T_{Gossip}$ Length of a Gossip cycle,
         $T_{out}$ Timeout period
Initialisation:
        //Fault list $L_p = \{< r, ccnt >, ...\}$
1       $L_p \leftarrow \{\}$
At each cycle (every $T_{Gossip}$ time units):
        //failure detection using pinging
2       $q = getRandomProcess()$
3       $extractProcessIds(Lid_p, L_p)$
4       send a message of type ping to $q$ piggybacking $Lid_p$
5       create timeout event $E_q =< curr\_cycle\_no + T_{out}, q >$
         for receiving reply message from $q$
        //check for consensus
6       foreach entry $< r, ccnt >$ in $L_p$
7           if$(curr\_cycle > log(n))$ && $(ccnt \geq MIN\_CCNT)$
8               $consensus\_reached(r)$
9           endif
10      end foreach
At event: received a message from $r$ piggybacked
          with $L_r$:
11      if$(message\_type == ping)$
12          $extractProcessIds(Lid_p, L_p)$
13          send a message of type reply to $r$ piggybacking $Lid_p$
14      endif
        //merge the contents of the two lists and update ccnt of
        //each entry (indirect failure detection and propagation)
15      $L_p \leftarrow merge(L_p, L_r)$
At event: timeout $E_q$ and no reply message received
          from $q$ within $T_{out}$:
        //mark $q$ to have failed and add it to the list (direct
        //failure detection)
16      $L_p \leftarrow L_p \cup \{< q, 0 >\}$
```

Figure 3. Efficient failure consensus (algorithm 2)

Each process maintains a fault list L_p to store the failed processes known to it. 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. $ccnt$ is the length of an unbroken sequence of pinged processes that have r in their failed process list. The algorithm can be divided into four logical tasks according to the functions performed: (1) initialization, (2) failure detection, (3) updating the fault list and (4) checking for consensus.

Every process starts with the assumption that every other process in the system is alive and hence has its fault list empty (line 1 of Figure 3).

To detect failures, every T_{Gossip} time units a process p randomly selects a process q and sends a ping message to it, piggybacking the list containing only the failed process ids Lid_p (lines 2-4). A timeout event is then set to receive a reply from q during the current Gossip cycle (line 5). An asynchronous reply follows upon reception of this ping message (lines 11-14). At the expiry of the current Gossip cycle, if a reply message from q has not been received by p, q is detected to have failed and is added to the fault list with $ccnt$ initialized to 0 (line 16).

Upon reception of a Gossip (ping or reply) message at p from r, the remote fault list L_r is merged with the local fault list L_p (line 15). This merge operation includes incrementing the $ccnt$ of an element $< r, ccnt >$ in the local list if r is also present in the received fault list. The $ccnt$ of the element is reset if the received fault list does not contain the element $< r, ccnt >$. If the received fault list contains an element $< r, ccnt >$ not present in the local fault list, it is added to the local fault list with $ccnt$ set to 0.

The failure detection and this failure information propagation continues for the initial $log(n)$ cycles (propagation phase). Finally, to check if consensus has been reached on the failure of process r, the value of the associated $ccnt$ is checked after the propagation phase. When a minimum value of the consensus counter (MIN_CCNT) is reached, consensus on the failure of process r is detected (lines 6-10).

In algorithm 1, when a process p has detected consensus on k it is certain that all processes have detected the failure of k, although they may not have detected consensus yet. Whereas in algorithm 2 when a process p has detected consensus on k, some processes may have not detected the failure of k yet. The initial propagation phase of $log(n)$ cycles before checking consensus makes sure with a very high probability that the information on the failure of k has reached all the processes. After the propagation phase, the probability of a process receiving successive Gossip messages not containing the information on the failure of k follows a binomial distribution. Hence, a small value of the threshold MIN_CCNT can be adopted.

3 Three-phase Consensus

Gossip-based aggregation protocols are used to compute global aggregation functions like sum, count, average, etc. in a distributed system. The number of Gossip cycles required to compute an aggregation function is shown to be logarithmic in the system size[19]. Checking for consensus on a failed process involves counting the number of fault-free processes that have detected the failed process. To this aim, an epidemic aggregation protocol for the count function can be employed. In the approach proposed in this section, processes detect failures that have occurred in the system in the same way as in the previous approaches; they store failed processes in a local list like algorithm 2 and adopt an ad-hoc epidemic aggregation protocol to detect the consensus. The Gossip-based aggregation protocol proposed in[20] is used here.

```
At process $p$
Require: $T_{Gossip}$ Length of a Gossip cycle,
         $T_{out}$ Timeout period
Initialisation:
     //Failed process list $F_p$ = $\{e =< r, d, i, v_d, v_c, w >, ...\}$
1    $F_p \leftarrow \{\}$
Every $T_{Gossip}$ time units:
     //update entries in $F_p$ before Gossip
2    foreach entry $e =< r, d, i, v_d, v_c, w >$ in $F_p$
3       $e \leftarrow < r, d, i, v_d/2, v_c/2, w/2 >$
4    end foreach
5    $q = getRandomProcess()$
6    create consensus protocol message $m$, where
7       $m.type \leftarrow ping, m.list \leftarrow F_p$
8    send $m$ to $q$
9    create timeout event $E_q =< curr\_cycle\_no + T_{out}, q >$
      for receiving reply message from $q$
10   foreach entry $e \leftarrow < r, d, i, v_d, v_c, w >$ in $F_p$
11      $n_d = v_d/w, n_c = v_c/w$
        //check for consensus on $r$
12      if($|(n_d/(N - |F_p|)) - 1| < T$)
13         $e =< r, d, i, v_d, v_c + 1, w >$
14         $consensus(r)$
15      endif
        //check for commit on $r$
16      if($|(n_c/(N - |F_p|)) - 1| < T$)
17         $commit(r)$
18      endif
19   end foreach
At event: received message $m_j$ from $j$
20   if($m_j.type == ping$)
21      foreach entry $e =< r, d, i, v_d, v_c, w >$ in $F_p$
22         $e \leftarrow < r, d, i, v_d/2, v_c/2, w/2 >$
23      end foreach
24      create consensus protocol message $m$, where
25         $m.type \leftarrow reply, m.list \leftarrow F_p$
26      send $m$ to $q$
27   endif
     //merge entries
28   foreach entry $e_2 =< r_2, d_2, i_2, v_{d2}, v_{c2}, w_2 >$ in $m_j.list$
29      if($F_p$ contains $e_1 =< r_1, d_1, i_1, v_{d1}, v_{c1}, w_1 >$
     ↪        such that $r1 == r2$)
           //Handle multiple direct detections of same failure
30         if($(i_2 < i_1)||((i_2 == i_1)\&\&(d_2 < d_1))$)
              //Keep earliest detection or process with lowest id
31            $e1 \leftarrow < r_2, d_2, i_2, v_{d2} + 1, v_{c2}, w_2 >$
32         else if( $(i_2 == i_1)\&\&(d_2 == d_1)$)
              //Info. diffusion with exponential propagation
33            $e1 \leftarrow < r_2, d_2, i_2, v_{d1} + v_{d2}, v_{c1} + v_{c2}, w_1 + w_2 >$
34         endif
35      else //Indirect detection
36         $F_p \leftarrow F_p \cup < r_2, d_2, i_2, v_{d2} + 1, v_{c2}, w_2 >$
37      endif
38   end foreach
At event: $E_q$ timeout and no reply message received
          from $q$ within $T_{out}$:
     //Direct detection
39   $F_p \leftarrow F_p \cup < q, p, curr\_cycle\_no, 1, 0, 1 >$
     //Restore lost mass
40   foreach entry $e_1 =< r_1, d_1, i_1, v_{d1}, v_{c1}, w_1 >$ in $E_q.m.list$
41      foreach entry $e_2 =< r_2, d_2, i_2, v_{d2}, v_{c2}, w_2 >$ in $F_p$
42         if($(r_1 == r_2)\&\&(i_1 == i_2)\&\&(d_1 == d_2)$)
43            $e_2 \leftarrow < r_2, d_2, i_2, v_{d1} + v_{d2}, v_{c1} + v_{c2}, w_1 + w_2 >$
44         endif
45      end foreach
46   end foreach
```

Figure 4. Three-phase consensus (algorithm 3)

The epidemic consensus protocol operates in three phases: (1) failure detection and propagation, (2) consensus and (3) commit. In phase 1, failures are detected and propagated to all the processes using stochastic pings and Gossiping. During this phase, an epidemic aggregation is also used to build local estimates of the number of fault-free processes which have detected a failure. The local estimate of this detection count at a process is indicated as n_d. When this count reaches the expected number of fault-free processes in the system, consensus on the failure is detected locally (local detection of consensus) and the process makes the transition to phase 2. In phase 2, a process needs to determine if all other fault-free processes have also detected consensus on the failure. This is achieved by means of a second epidemic aggregation to estimate a consensus count, n_c. When this count reaches the expected number of fault-free processes in the system, global convergence of the consensus is detected (local detection of global convergence) and the process makes the transition to phase 3. In phase 3, the process can safely commit the failure as it has determined that all processes have detected consensus.

The algorithm shown in Figure 4 is executed at every process p. The processes use the same length of the Gossip cycle, T_{Gossip}, and the timeout period, T_{out}, for Gossip messages. Each process maintains a fault list F_p to store information about detected failed processes. An entry in this list is a 6-tuple $< r, d, i, v_d, v_c, w >$ where r is the rank of the failed process, d is the rank of the first process to have detected the failure (direct failure detection), i is the Gossip iteration number (cycle) when failure of r was detected by d, and v_d, v_c and w are the two values and the weight for the epidemic aggregations in phase 1 and 2. The triplet (r, d, i) in the tuple describes the event: at cycle i the process d has inferred by a timeout event that process r has failed. The pair (v_d, w) in the tuple is used to estimate the number of processes which are aware of this event. Similarly, the pair (v_c, w) is used to estimate the number of processes which have detected consensus on the failure of r. Initialisation of the tuple (creation of the tuple) is done by $d :< r, d, curr\_cycle\_no, 1, 0, 1 >$ when it detects the failure of r (direct detection). When a process p receives and inserts a new tuple in its local list, it increments v_d. When the process makes the transition to phase 2, it increments the value v_c.

After initialization a process p performs three tasks during each Gossip cycle: failure detection, failure list update and check for consensus. Every process starts with the assumption that every other process in the system is alive: at initialisation the local fault list is empty (line 1).

To detect failures, every T_{Gossip} time units a process p randomly selects another process q and sends a ping message to it with the fault list F_p. F_p contains all the elements in the list with values (v_d and v_c) and weights (w) halved (lines 2-8). A timeout event is then set to receive a reply from q (line 9). An asynchronous reply follows upon reception of this ping message. F_p sent with the message, again, contains all the elements in the list with values and weight halved (lines 20-27). At timeout, if a reply message from q has not been received by p, q is detected (directly) to have failed and an entry $< q, p, curr\_cycle\_no, 1, 0, 1 >$ is added to the fault list (line 39). To restore the mass ($\sum_i v_i$ and $\sum_i w_i$) in the system, the halved values and weight are added (lines 40-46).

Upon receipt of a Gossip (ping/reply) message at p from j, the remote fault list, F_j, is merged with the local fault list, F_p. If an entry in the remote list is not present in the local list, it is added to the local list and thus, a failure is indirectly detected by process p; v_d is incremented by 1 to contribute towards the failure detection count whereas v_c from the remote list entry is propagated (lines 35-37). If an entry in the remote list is already present in the local list, it means that the failure of the rank corresponding to the entry has already been detected by rank p. In this situation, earliest detection of the failure or detection by the lowest ranked process is retained (if both the failures were detected during the same Gossip cycle) and v_c from the remote list entry is propagated (lines 29-31). If an entry is present in both the remote and local lists with the same detection (detected during the same Gossip cycle and the same detector process), they are merged to enable exponential failure propagation (lines 32-34).

Finally, to check if consensus has been reached on the failure of a process r in the list, n_d for r, is checked. When this value becomes equal to the expected fault-free system size (with an allowed tolerance), consensus is reached on the failure of process r. This implies that all the fault-free processes have recognized the failed process r. A process, thus, detecting consensus locally contributes to the global consensus count by incrementing v_c by 1. When n_c becomes equal to the expected fault-free system size (with an allowed tolerance) commit is reached on the failure of process r. This indicates that all the fault-free processes have reached consensus on the failure (lines 10-19).

4 Performance Evaluation

4.1 Algorithm Analysis

In all the three algorithms, the failures are detected using stochastic failure detection which adopts a ping-reply communication during each Gossip cycle. The number of ping messages a process receives during a Gossip cycle follows binomial distribution. Hence the probability of a process receiving multiple ping messages is very low and so is the probability of a failed process not receiving a ping message. Therefore, any failed process in the system is detected within the first few Gossip cycles.

Once a failure is detected, it is disseminated to the fault-free processes in the system using the ping-reply Gossip messages. [21;22] show that the number of Gossip cycles required for a piece of information to be disseminated from the originator to all the processes in the system is logarithmic in system size. Algorithm 1 and 2 hence achieve consensus on the detected failure in logarithmic number of Gossip cycles. The two messages - ping and reply - in a Gossip cycle double the information dissemination speed. Moreover, processes independently detecting failures directly and merging these multiple failure detection information while dissemination further increases the propagation speed and reduces the consensus time.

Algorithm 3 disseminates the failures detected and counts the number of fault-free processes that this information has reached using the aggregation protocol. [19] shows that the number of Gossip cycles required for computing the aggregation function is $O(log(n) + log(\epsilon^{-1}) + log(\delta^{-1}))$, where n is system size, ϵ is the maximum approximation error and δ is the maximum probability of the error being greater than ϵ. Hence the consensus and commit times in algorithm 3 have logarithmic complexity.

The number of Gossip messages in each cycle is 2 and hence the number of Gossip messages needed by the algorithms to detect consensus at each process is twice the number of Gossip cycles taken.

The first algorithm requires n^2 units of memory at each process to store the fault matrix whereas the second and third algorithms have a space complexity O(m), where m is the number of failed processes ($m << n$).

4.2 Experimental Results

All the three algorithms were implemented in the form of MPI applications, using basic MPI point-to-point communication primitives. The fault matrix of algorithm 1 is implemented as an integer matrix, whereas failed process details in algorithm 2 and algorithm 3 are either integers or floating point numbers as appropriate. Fault injection was simulated by excluding a process from further communication.

The experiments were performed on two Linux cluster computers, one at the University of Reading (UREAD) and one at the Oak Ridge National Laboratory (HAL9000). Algorithm 1 and 2 were tested on the UREAD cluster and Algorithm 3 was tested on the HAL9000 cluster.

The UREAD cluster has one head node and 16 compute nodes. The head node has two AMD Opteron 4386 3.1 GHz processors with eight cores per processor and 64 GB RAM. The compute nodes have one Intel Xeon E3-1220 3.1GHz processor with four cores per processor and 16 GB RAM. The entire system has a total of 80 compute cores. The nodes are connected by Gigabit Ethernet. The system is running the Ubuntu 12.04 LTS operating system and Open MPI 1.6.5.

The HAL9000 cluster has one head node and 16 compute nodes. The head node has two AMD Opteron 2378 2.4 GHz processors with four cores per processor and 8 GB RAM. The compute nodes have two AMD Opteron 2378 2.4 GHz processors with four cores per processor and 8 GB RAM per node. There are 128-cores for compute and 8-cores on the login/head node. The nodes are connected by Gigabit Ethernet (1 Gbps). The system is running Ubuntu 14.04 LTS, Open MPI 1.10.1 and GCC 4.8.

Experiments were executed using the Extreme-scale Simulator (xSim 0.5 on UREAD and xSim 0.8 on HAL9000)[23–25] atop the Linux cluster(s) to evaluate the algorithms at significantly larger scale than the available physical system. xSim is a performance investigation toolkit that permits running MPI applications in a controlled environment with a large number of concurrent execution threads, while observing application performance and resilience in a simulated extreme-scale system. Using a lightweight parallel discrete event simulation, xSim executes an MPI application on a much smaller system in a highly oversubscribed fashion with a virtual wall clock time, such that performance data can be extracted based on a processor and a network model. xSim is designed like a traditional MPI performance tool, as an interposition library that sits between the MPI application and the MPI library, using the MPI profiling interface. In previous experiments, it has been run up to 134,217,728 communicating MPI ranks using a 960-core Linux cluster.

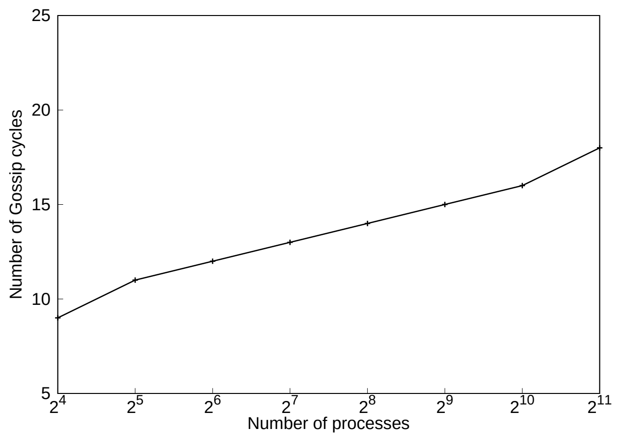

Figure 5. Number of cycles to achieve global consensus after a single failure injection (algorithm 1)

The simulator was deployed on the Linux cluster computer by associating one simulator MPI process per physical processor core. Within each simulator MPI process, a number of concurrent execution threads are executed, each representing an individual MPI process that is located on a processor core within a simulated HPC system. The execution timing of these simulated MPI processes is based on a processor model with a 1-to-1 performance match to the physical AMD processor core the simulator is running on and a network interconnect model with a basic star topology, 1 μs link latency, and infinite bandwidth. The model parameters were set to nominal values as the experiments investigate the number of Gossip cycles required to reach consensus.

The maximum number of Gossip cycles in the experiments was set to $5log(n)$, where n is the number of simulated MPI processes. This is large enough to allow all the alive processes to reach consensus on the injected failures as the information dissemination speed of Gossip-based protocols is exponential. The length of each cycle was set to 10 ms, unless otherwise specified, considering the difference in processors' start up time and network round trip time. It can be tailored to suit the HPC platform. Failures were injected into randomly chosen simulated MPI processes. The processes detect/reach consensus/commit on the injected failure(s) at different cycles. Hence, the cycle number of the last process doing so is recorded.

4.2.1 *Consensus using Global Knowledge:*

The Gossip cycle length for a given system size was set to allow the matrix merge operations to complete within the cycle. The cycle length was 10, 100 and 1000 ms respectively for system sizes {16,32,64,128},{256,512} and {1024,2048}. This is necessary as matrix merge operations consume substantial amount of the cycle time. Experiments were carried out to test the algorithm's scalability and fault tolerance.

Failures were injected right before the failure detection and consensus algorithm is run. Figure 5 shows the relation between the number of cycles taken to reach consensus

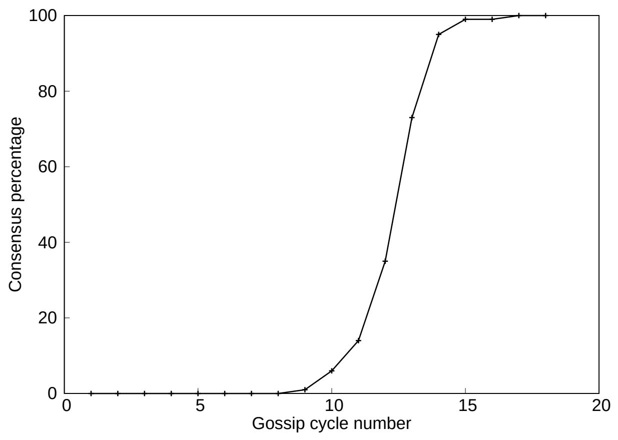

Figure 6. Local consensus progress at a process after a single failure injection for system size of 2048 (algorithm 1)

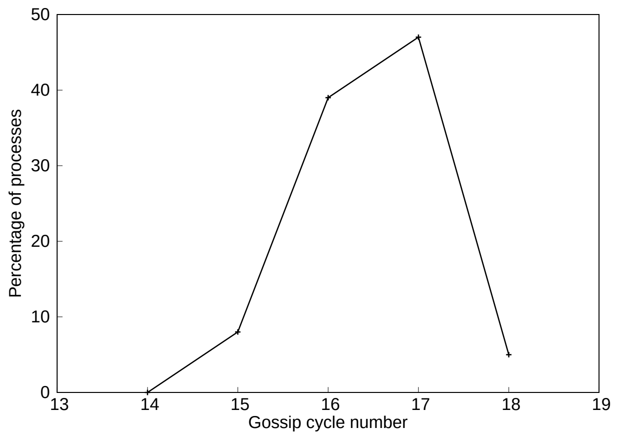

Figure 7. Consensus detection spread for a system size of 2048 (algorithm 1)

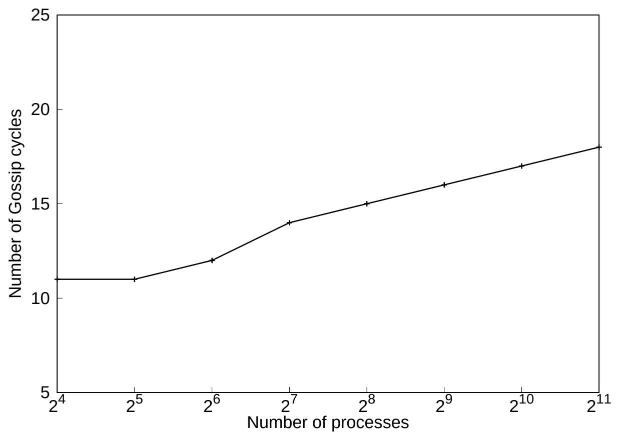

Figure 8. Number of cycles to achieve global consensus after multiple (4) failures, which were injected before algorithm execution (algorithm 1)

and the system size for a single failure injected before the algorithm. It is evident that the number of cycles to

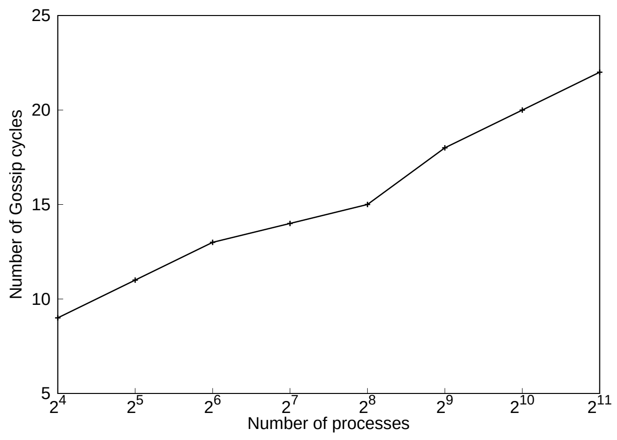

Figure 9. Number of cycles to achieve global consensus with multiple (4) failures, which were injected during algorithm execution (algorithm 1)

reach consensus varies logarithmically with the system size. Figure 6 shows the exponential spreading of failure detection information at a particular process for the injected failure. Both figures demonstrate the logarithmic complexity of the algorithm. Figure 7 shows the distribution of the cycle number at which different processes reach consensus. In Figure 8, multiple (four) failures were injected before the algorithm and their effect on consensus time was observed. It took only 1 or 2 cycles more than in the single failure case (Figure 5).

Failures were injected during its execution to test the fault tolerance property of the algorithm. In Figure 9, multiple (four) failures were injected into randomly chosen processes at random cycles. The number of cycles needed to reach consensus increased slightly. The algorithm is completely fault tolerant.

4.2.2 Efficient Heuristic Consensus: The MIN_CCNT was set to 3. Smaller values for MIN_CCNT may detect consensus falsely and larger values may unnecessarily delay consensus detection.

Figure 10 shows the relation between the number of cycles taken to reach consensus on a single injected failure and the system size. It is evident that the number of cycles to reach consensus varies logarithmically with the system size as expected. Moreover, in this case no variance in the cycle number at which different processes reach consensus was observed, thus achieving perfect synchronization. The algorithm will detect consensus on any number of failures injected before it is run.

The algorithm is scalable in terms of memory as it needs to store only the list of failed processes at each process. The two algorithms were compared for their bandwidth utilization. The amount of data exchanged between simulated MPI processes is reported by xSim. Figure 11 shows the bandwidth consumed per process at increasing system sizes. It can be observed that the heuristic-based algorithm transfers a negligible amount of data and is significantly more efficient than the algorithm using global knowledge.

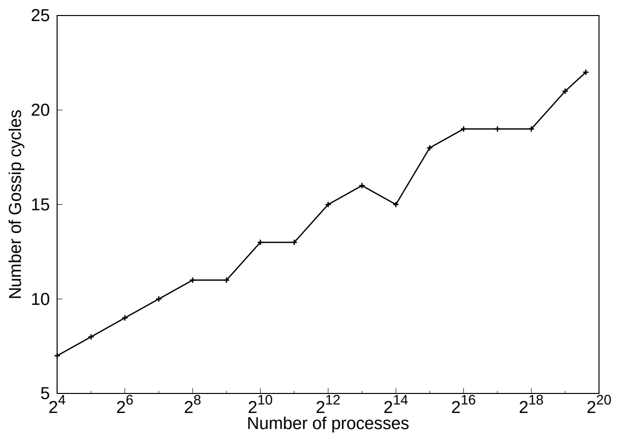

Figure 10. Number of cycles to achieve global consensus after a single failure injection (algorithm 2)

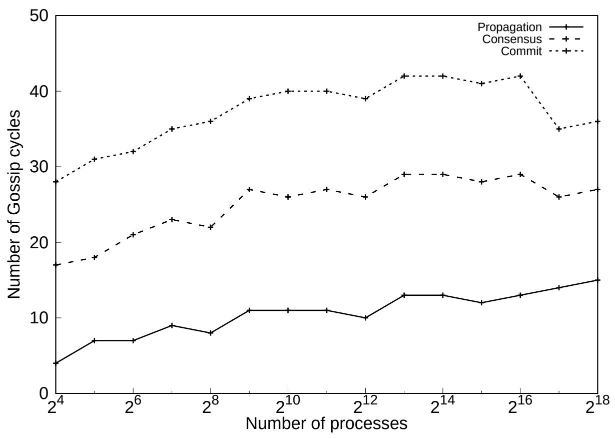

Figure 12. Number of cycles taken for propagation, consensus and commit (algorithm 3)

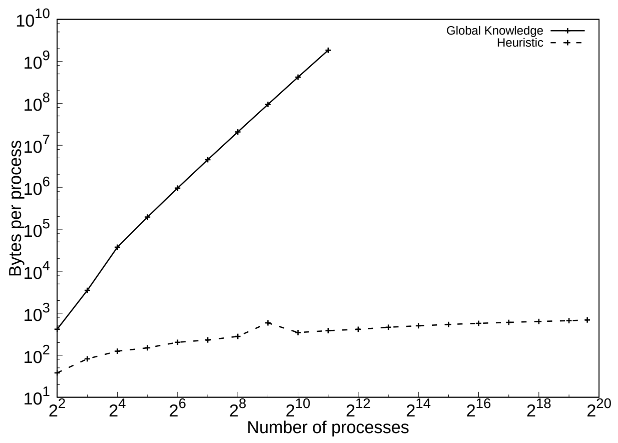

Figure 11. Total bandwidth utilization of the consensus algorithms with a single failure injection (algorithms 1 and 2)

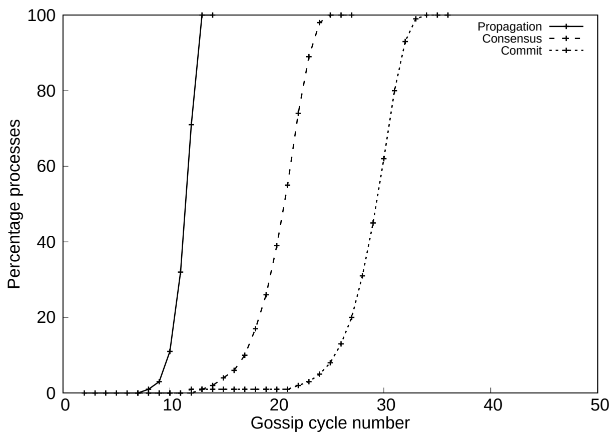

Figure 13. Propagation, consensus and commit progress (algorithm 3)

4.2.3 Three-phase Consensus: Epidemic three-phase failure consensus algorithm was tested by injecting a single failure. It can detect any number of failures injected before it is run. The single injected failure was detected, propagated to all the processes (phase 1), the processes detected consensus (phase 2) and committed on the injected failure (phase 3). Measurements of these three phases have been collected for different system sizes.

Figure 12 shows the relation between the system size and the number of cycles taken to propagate, reach consensus and also to commit on the injected failure. The tolerance between the estimated counts (for both propagation and consensus) and the actual number of fault-free processes in the system was set to 0.1%. The propagation, consensus and commit times all vary logarithmically with the system size.

Figure 13 shows the exponential progress of the propagation, consensus and commit phases for the system size of 262,144 at 0.1% tolerance. The injected failure is detected and propagated at exponential speed; consensus and commit are, then, detected on it and they progress at exponential speed in the system.

The estimated count values for propagation and consensus at rank 0 for the system size 262,144 at 0.1% tolerance is shown in the Figure 14. They quickly converge to the number of fault-free processes in the system as the Gossip progresses. It is an indication that the failure information has propagated to all the fault-free processes in the system and consensus is detected by each fault-free process.

The algorithms performance for different tolerance values is shown in the Figure 15 for the system size of 65,536. Consensus and commit detection are delayed as the tolerance is reduced for increased accuracy.

5 Related Work

This section discusses failure detection and consensus algorithms that have been considered for HPC and compares them with the proposed algorithms.

5.1 Failure Detection Algorithms

5.1.1 Heartbeat-based Approaches: A failure detection algorithm, for fail-stop type failures, using heartbeat messages was proposed in[26]. Every process maintains a log,

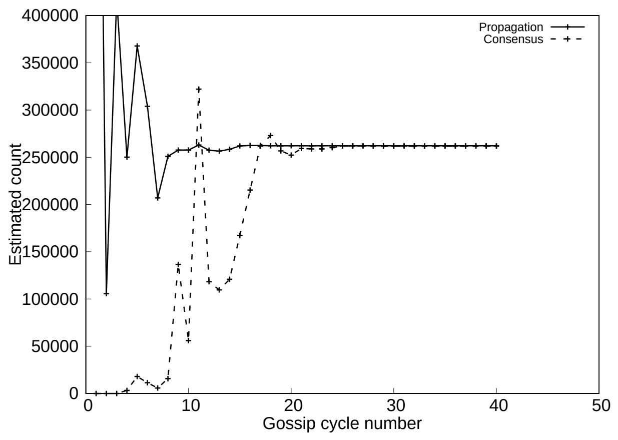

Figure 14. Convergence of estimated count values over the Gossip cycles (algorithm 3)

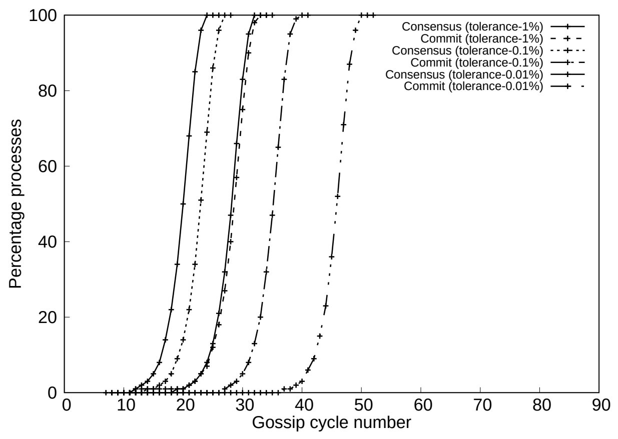

Figure 15. Consensus and commit comparison for different tolerances (algorithm 3)

called Gossip list, that contains a number (called heartbeat value) for each member process to represent its aliveness. Every T_{Gossip} time units the process increments its own heartbeat value in the list and Gossips the list to a randomly chosen process. When a process receives a remote Gossip list, it updates its Gossip list by setting the heartbeat value for each process to the maximum value in the two lists. The Gossip list is monitored continuously and if the heartbeat value for a process has not risen for $T_{cleanup}$ time units, it is suspected to have failed. There is, however, chance of an alive process to be falsely suspected as the algorithm is based on randomized Gossiping.

In the proposed algorithms, there is direct failure detection without passing through the suspicion phase of the above Gossip-based failure detection. Also, since suspicions become detections after a majority vote, the assumption that no more than one third of processes do not fail during a single Gossip iteration is relaxed in the proposed approach.

Another heartbeat based low overhead failure detection algorithm was proposed in [27]. Processes are organized in the form of a ring and each process monitors only its predecessor using heartbeats and timeouts. A process periodically sends heartbeat messages to its monitor and the monitor detects the process's failure if it does not receive heartbeat message for a predefined period. Once a process detects a failure, it broadcasts this information to all the fault-free processes using an all-to-all reduction topology. It assumes that no more than k failures occur, where $2^k \leq n < 2^{k+1}$ and n being the number of processes, to enable the broadcast to complete.

In contrast, the proposed algorithms detect failures and consistently disseminates them to all the fault-free processes, simultaneously, thus requiring no failure free operation overhead. Also, the epidemic algorithms are intrinsically fault tolerant and can tolerate any number of failures without any constraint on the failure strike frequency.

5.1.2 Ping-based Approaches:

Algorithm for failure detection given in[28] is based on randomized pinging. A process p randomly pings another process q. q is found to be alive if a reply is received in time. Otherwise the p asks k randomly chosen peers to ping q as well. If no peer receives a reply, p detects q to have failed. This approach to failure detection takes network link failures into account along with process failures.

5.2 Consensus Algorithms

Both centralized, i.e., using a coordinator, and completely distributed consensus algorithms are available. Fault tolerant versions of the two-phase and three-phase consensus algorithms are discussed in the following. Distributed consensus algorithms based on Gossiping, which is inherently fault tolerant, are also discussed.

5.2.1 Coordinator-based Approaches:

Over a static tree structure: A two-phase consensus algorithm to aggregate locally known failures is given in[29]. Failures are presumed to be detected prior to invoking the algorithm. It is a fault tolerant log-scaling two-phase algorithm running over a tree topology. It is based on reliable gather and broadcast operations, both built on a multi-level, tree structure communication topology. The coordinator is at the root of the tree and makes the final decision based on the votes by the participants organized in the form of parent and children below it. Votes from the participants at the leaves and intermediate levels of the tree are gathered by the coordinator at the root going through the intermediate parent nodes and the decision from the coordinator is broadcast in the reverse direction. In the presence of faults, the parent of a failed participant recursively adopts its children. A child, upon detection of the failure of its parent, queries its grandparent to know how it should continue in the algorithm. If it has not voted yet, it will participate in the voting phase. If it has voted but the parent failed before propagating the vote up, it will cast the vote again; if the parent propagated the vote, it will participate in the decision phase. A termination detection algorithm is invoked upon failure of the coordinator. The parent directly below the coordinator decides the termination status: success with failed process list if it received decision from the coordinator, or abort if it has not voted yet. The participant, thus making the decision, propagates it below the tree. However, the termination detection will not be of any help if the coordinator failed without propagating its decision after the voting phase.[29] handles this by enabling a fault-free process taking the role of the coordinator and restarting the algorithm. The tree is rebalanced with the new alive processes between invocations of the algorithm and thus the approach involves maintenance of the communication structure, which is not needed by the algorithms proposed in this paper.

Failures that happen during the execution of the two-phase consensus are not included in the final list and a coordinator failure aborts the algorithm. Hence, it is not a completely fault tolerant consensus algorithm. In contrast, the first two algorithms presented in this paper are completely fault tolerant, as they tolerate failures during the execution.

Over a dynamic tree structure: The approach in[30] assumes that a process knows of a few failed processes (a failure detector is again assumed to be readily available) to contribute towards the global list of failed processes. It is built around a reliable broadcast algorithm that dynamically constructs the broadcast tree and a three-phase consensus algorithm.

In the *BALLOTING* phase, the root generates a ballot (which is a sequence number to differentiate between iterations) and broadcasts it, including the known list of failed processes. The child, upon receipt of the broadcast message, checks whether it has any new failed processes known to it but not in the ballot it received. It sends a *REJECT* message piggybacked with the *ACK* message including the new failed processes if any; it accepts the ballot otherwise and sends an *ACCEPT* message piggybacked with the *ACK* message. The root starts the next phase if it receives an *ACCEPT* message piggybacked with the *ACK* message from all its children and if any child rejects the ballot it updates its set of failed processes and tries again. In the second phase, the root broadcasts the *AGREE* message with the ballot. Now the participants know that the ballot has been agreed upon by everyone and they agree to the ballot for the second phase. Then the root starts the third phase by broadcasting the *COMMIT* message. The participants upon receipt of the *COMMIT* message commit to the ballot.

A failure of the root is checked by every process and when a process detects that all processes with ranks lower than itself have failed it appoints itself as the new root. The new root restarts the algorithm from whatever state it is in. Note that if the root fails when it is in the *BALLOTING* state, the new root has to start all over again, wasting all the iterations performed so far. Failures of participants are handled by repeating phase one of the algorithm with a new sequence number included in the ballot. When a broadcast message belonging to an old iteration arrives at a process, which is not in the *BALLOTING* state, a *NAK* with *AGREE_FORCED* is forwarded to the root to clean up the old circulating broadcast messages. This message is also used by the root to start Phase 2 of the algorithm (with the assumption that process failures will subside and cease).

It was tested by injecting failures into randomly chosen processes before and during the execution. The processes that fail during the operation of the algorithm may or may not be included into the final list of failed processes. Moreover, every failure that happens while the algorithm is running requires the algorithm to start all over again by rebuilding the communication structure. The algorithms proposed in this paper do not require any communication structure.

5.2.2 Gossip-based Approaches: In Gossip-based approaches, failure detection is performed as explained in Section 2.1 and consensus is (in combination with failure detection) also implemented using Gossiping[26], and hence completely fault tolerant. Each process maintains a suspicion matrix S to store the status of processes as detected by all the processes. An entry $S_{i,j}$ in a process's suspicion matrix indicates the status of process j as detected by process i (1 if suspected to have failed 0 otherwise). It also maintains a fault vector F to store the processes' status as decided by all processes. An entry F_i in this vector is 1 if the majority of the processes suspect process i to have failed. The suspicion matrix is sent to other processes as part of Gossiping. When a process receives this suspicion matrix, it merges it with its own suspicion matrix. A process updates its fault vector by examining the suspicions of all processes. If the majority of the processes have suspected a process to have failed, it decides that the process has indeed failed and updates its fault vector to reflect this. Consensus is reached when all processes have detected a process to have failed. Upon detection of consensus a process broadcasts a consensus message to all the live processes.

Because every process needs to maintain a suspicion matrix of $O(n^2)$, where n is the system size, this algorithm is memory intensive and does not scale well. For increasing process numbers (beyond 48) the consensus time was found to increase exponentially. The first algorithm proposed in this paper also uses a matrix at each process, but the second and the third algorithms use only a list at each process. The algorithms given in this paper scale logarithmically with system size.

The proposed approach in this paper bypasses the failure suspicion phase based on distributed diagnosis. Since suspicions become detections after a majority vote, the assumption that no more than one third of processes do not fail during a single Gossip iteration is relaxed in this approach.

Experiments in the state of the art HPC failure detection and consensus literature, have featured not more than a few thousand processes, whereas the proposed algorithms (algorithm 2 and algorithm 3) scaled to hundreds of thousand processes on a small cluster computer.

6 Conclusion and Future Work

Failure detection and consensus for a fault-tolerant MPI enable HPC applications to adopt algorithm-based fault tolerance techniques to cope with MPI process failures more efficiently. Centralized methods for failure detection and consensus are based on a coordinator and do not scale well to very large and extreme-scale systems. Completely distributed algorithms based on Gossiping that were previously proposed in the literature consume an inordinate amount of time, memory and network bandwidth.

In this work three novel failure detection and consensus algorithms that use randomized pinging were presented. The first approach is based on global knowledge: each process maintains a local view of the entire system state to achieve consensus on failed processes. A Gossip protocol is used to detect failures and to exponentially propagate them in the system until the local views converge. The second algorithm does not rely on global knowledge and adopts a heuristic method to achieve consensus on failures. The third algorithm maintains only a list of failures as in the second algorithm and detects consensus using three phases incorporating a Gossip-based aggregation protocol. The same Gossip messages used for failure information dissemination are also used for detecting the failures stochastically thus integrating both failure detection and consensus.

All the algorithms were implemented as MPI applications and tested using the Extreme-scale Simulator. The results confirm their expected scalability and fault tolerance properties. In the algorithms, the number of Gossip cycles to achieve consensus on failures scales logarithmically with the system size. The second algorithm has significantly lower memory and bandwidth utilization and has shown to be able to achieve a perfect consensus synchronization as well. The third algorithm retains the scalability of the second algorithm and detects consensus accurately using an aggregation protocol for the count aggregation function.

The first algorithm can be implemented with boolean matrices at each process to increase scalability. The second and third algorithms' memory scalability can be further improved by maintaining the status of processes in a bit vector, if the number of failures in the system is high. It would also be interesting to investigate an efficient algorithm with a different heuristic approach for detecting consensus asynchronously and without the guaranteed initial propagation phase: this would allow detecting consensus on failures that happen both before and during the execution of the algorithm. Investigating processes entering alive state from faulty state is also interesting. This would avoid false positives and also allow process recovery.

Further future work in this area focuses on implementing *MPI_Comm_shrink()* and *MPI_Comm_agree()* with different approaches (static and dynamic tree, as well as, the different Gossip-based variants) and compare them using the Extreme-scale Simulator with architectural models of future-generation HPC systems.

7 Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

8 Funding

The author Amogh Katti was supported by the Felix Scholarship for his PhD project.

This work was sponsored by the U.S. Department of Energy's Office of Advanced Scientific Computing Research. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy.

References

1. Geist A. Supercomputing's monster in the closet. *IEEE Spectrum* 2016; 53(3): 30–35.
2. Baumann RC. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability (TDMR)* 2005; 5(3): 305–316.
3. Kaul H, Anders M, Hsu S et al. Near-threshold voltage (NTV) design: Opportunities and challenges. In *Proceedings of the 49th Annual Design Automation Conference*. DAC '12, New York, NY, USA: ACM, pp. 1153–1158.
4. Snir M, Wisniewski RW, Abraham JA et al. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications (IJHPCA)* 2014; 28(2): 127–171.
5. Kaplan L, Briggs P, Ohlrich M et al. Resilience to various failures for read-mostly in-memory data structures. In 26^{th} *IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2012: Workshops*. pp. 1572–1580.
6. Huang KH and Abraham JA. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers (TC)* 1984; C-33(6): 518–528.
7. Ltaief H, Gabriel E and Garbey M. Fault tolerant algorithms for heat transfer problems. *Journal of Parallel and Distributed Computing (JPDC)* 2008; 68(5): 663–677.
8. Bland W, Bosilca G, Bouteiller A et al. A proposal for user-level failure mitigation in the mpi-3 standard. *Department of Electrical Engineering and Computer Science, University of Tennessee* 2012; .
9. Soltero P, Bridges P, Arnold D et al. A gossip-based approach to exascale system services. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, p. 3.
10. Straková H, Niederbrucker G and Gansterer WN. Fault tolerance properties of gossip-based distributed orthogonal iteration methods. *Procedia Computer Science* 2013; 18: 189–198.
11. Barak A, Drezner Z, Levy E et al. Resilient gossip algorithms for collecting online management information in exascale clusters. *Concurrency and Computation: Practice and Experience* 2015; 27(17): 4797–4818.
12. Katti A, Di Fatta G, Naughton T et al. Scalable and fault tolerant failure detection and consensus. In *Proceedings of the 22nd European MPI Users' Group Meeting*. ACM, p. 13.
13. Skeen D. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*. SIGMOD '81, New York, NY, USA: ACM, pp. 133–142.
14. Turek J and Shasha D. The many faces of consensus in distributed systems. *Computer* 1992; 25(6): 8–17.
15. Fischer MJ, Lynch NA and Paterson MS. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* 1985; 32(2): 374–382.
16. Chandra TD and Toueg S. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)* 1996; 43(2): 225–267.
17. Chandra TD, Hadzilacos V, Toueg S et al. On the impossibility of group membership. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. ACM, pp. 322–330.
18. Poonpakdee P, Orhon NG and Di Fatta G. Convergence detection in epidemic aggregation. In *Euro-Par 2013: Parallel Processing Workshops*. Springer, pp. 292–300.
19. Kempe D, Dobra A and Gehrke J. Gossip-based computation of aggregate information. In *Foundations of Computer Science,*

2003. Proceedings. 44th Annual IEEE Symposium on. IEEE, pp. 482–491.

20. Blasa F, Cafiero S, Fortino G et al. Symmetric push-sum protocol for decentralised aggregation. *Proc of the Int l Conf on Advances in P2P Systems* 2011; : 27–32.
21. Pittel B. On spreading a rumor. *SIAM Journal on Applied Mathematics* 1987; 47(1): 213–223.
22. Shah D. *Gossip algorithms*. Now Publishers Inc, 2009.
23. Engelmann C. Scaling to a million cores and beyond: Using light-weight simulation to understand the challenges ahead on the road to exascale. *Future Generation Computer Systems (FGCS)* 2014; 30(0): 59–65.
24. Naughton T, Engelmann C, Vallée G et al. Supporting the development of resilient message passing applications using simulation. In *Proceedings of the* 22^{nd} *Euromicro International Conference on Parallel, Distributed, and network-based Processing (PDP)*. Turin, Italy: IEEE Computer Society, Los Alamitos, CA, USA, pp. 271–278.
25. Böhm S and Engelmann C. xSim: The extreme-scale simulator. In *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS) 2011*. Istanbul, Turkey: IEEE Computer Society, Los Alamitos, CA, USA, pp. 280–286.
26. Ranganathan S, George AD, Todd RW et al. Gossip-style failure detection and distributed consensus for scalable heterogeneous clusters. *Cluster Computing* 2001; 4(3): 197–209.
27. Bosilca G, Bouteiller A, Guermouche A et al. Failure detection and propagation in hpc systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '16, Piscataway, NJ, USA: IEEE Press. ISBN 978-1-4673-8815-3, pp. 27:1–27:11.
28. Gupta I, Chandra TD and Goldszmidt GS. On scalable and efficient distributed failure detectors. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*. ACM, pp. 170–179.
29. Hursey J, Naughton T, Vallee G et al. A log-scaling fault tolerant agreement algorithm for a fault tolerant mpi. In *Recent Advances in the Message Passing Interface*. Springer, 2011. pp. 255–263.
30. Buntinas D. Scalable distributed consensus to support mpi fault tolerance. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, pp. 1240–1249.

9 Author Biography

Dr Amogh Katti is a Post-Doctoral Research Associate at the Department of Electrical and Computer Engineering, University of Minnesota, USA. He received his PhD in Computer Science at the University of Reading, UK in 2016. He did his Master of Technology (M Tech) and Bachelor of Engineering (B E) from Visweswariah Technological University, India in 2010 and 2006 respectively. He was an Assistant Professor at Walchand Institute of Technology, Solapur, India, from 2010 to 2012, and a Software Engineer at Mphasis, Bangalore, India in 2007. His research interests are fault tolerance and energy efficiency in parallel computing, and software engineering.

Dr. Giuseppe Di Fatta is an Associate Professor of Computer Science and currently the Head of the Department of Computer Science at the University of Reading, UK. In 1999, he was a research fellow at the International Computer Science Institute (ICSI), Berkeley, CA, USA. From 2000 to 2004, he was with the High-Performance Computing and Networking Institute of the National Research Council, Italy. From 2004 to 2006, he was with the University of Konstanz, Germany. His research interests include data mining algorithms, distributed and parallel computing, and multidisciplinary applications. He has published over 90 articles in peer-reviewed conferences and journals. He serves in the editorial board of the Elsevier Journal of Network and Computer Applications. He is the co-founder of the IEEE ICDM Workshop on Data Mining in Networks and has chaired several international events, such as the 2015 International Conference on Internet and Distributed Computing Systems.

Dr. Thomas Naughton is an R&D Staff Scientist at Oak Ridge National Laboratory. His research interests include resilience and fault-injection tools for high-performance computing (HPC). He also works on resource management/runtime systems for HPC, which includes research involving system-level virtualization and isolation. Dr. Naughton received a B.A. in Philosophy and B.S. in Computer Science from the University of Tennessee-Martin in 1998, a M.S. in Computer Science from Middle Tennessee State University in 2000, and a Ph.D. in Computer Science from the University of Reading in 2014. He is a member of the ACM.

Dr. Christian Engelmann is an R&D Staff Scientist at Oak Ridge National Laboratory. His primary expertise is in extreme-scale high-performance computing (HPC) resilience. He received the 2015 US Department of Energy Early Career Award for research in resilience design patterns for extreme scale HPC. His secondary expertise is in lightweight simulation of future-generation extreme-scale supercomputers with millions of processors. Dr. Engelmann earned a M.Sc. in Computer Systems Engineering from the University of Applied Sciences Berlin, Germany, in 2001, a M.Sc. in Computer Science from the University of Reading, UK, in 2001, and a Ph.D. in Computer Science from the University of Reading in 2008. He is a member of the ACM, IEEE, and USENIX.