Failure Detection within MPI Jobs: Periodic Outperforms Sporadic

Kishor Kharbas[1], Donghoon Kim[1], Kamal KC[1], Torsten Hoefler[2], Frank Mueller[1]

[1]Dept. of Computer Science, North Carolina State University, Raleigh, NC 27695-7534
[2]NCSA, University of Illinois at Urbana-Champaign, Urbana, IL 61801

Abstract. Reliability is one of the challenges faced by exascale computing. Components are poised to fail during large-scale executions given current mean time between failure (MTBF) projections. To cope with failures, resilience methods have been proposed as explicit or transparent techniques. For the latter techniques, this paper studies the challenge of fault detection.
This work contributes generic fault detection capabilities at the MPI level and beyond. A first approach utilizes a periodic liveness check while a second method promotes sporadic checks upon communication activities. The contributions of this paper are two-fold: (a) We provide generic interposing of MPI applications for fault detection. (b) We experimentally compare periodic and sporadic methods for liveness checking. We show that the sporadic approach, even though it imposes lower bandwidth requirements and utilizes lower frequency checking, results in *equal or worse* application performance than a periodic liveness test for larger number of nodes. We further show that performing liveness checks in separation from MPI applications results in lower overhead than interpositioning. Hence, we promote separate periodic fault detection as the superior approach for fault detection.

1 Introduction

The current road map to exascale computing faces a number of challenges, one of which is reliability. Given the number of computing cores, projected to be as large as a million, with ten of thousands of multi-socket nodes, components are poised to fail during the execution of large jobs due to a decreasing mean time between failures (MTBF) [14,19]. When faults become the norm rather than the exception, the underlying system needs to provide a resilience layer to tolerate faults. Proposed methods for resilience range from transparent techniques, such as checkpointing and computational redundancy, to explicit handling, such as in probabilistic or fault-aware computing. The latter approach requires significant algorithmic changes and is thus best suited for encapsulation into numerical libraries [6]. This paper focuses on the former techniques. It builds on recently developed techniques such as checkpointing (with restarts or rollbacks) or redundant computing in high-performance computing [1, 3, 5, 12, 22] or API extensions for checkpointing [6, 13]. A common challenge of transparent resiliency lies in the detection of faults, which is the focus of this paper.

Previous work suggested guidelines on the theoretical methodology for designing fault detection services. However, a practical implementation still poses a challenge in terms of completeness and accuracy because of the diversity of parallel system environments in terms of both hardware and software, which exposes a number of complications due to potentially unreliable failure detectors [4].

The fault model of this work is that components are subject to fail-stop behavior. In other words, components either work correctly or do not work at all. Transient or byzantine failures are not considered. A component is an entire compute node or a network connection / link between any two nodes. In such a framework, we base fault detection on timeout monitoring between endpoints. The focus of our work is to study the impact of timeout monitoring on application behavior such as to perturb application performance as little as possible.

Contributions: In this paper, we implement a fault detector (FD) to detect failures of an MPI application. An FD can be included at various layers of the software stack. First, we choose the MPI communication layer to implement the FD. We detect MPI communication failures and, at the same time, also utilize the MPI layer as a means to implement detection. This approach has the advantage that it does not require any failure detection support from the underlying software/hardware platform. Second, we implement a separate FD as stand-alone processes across nodes.

In this framework, we observe the effect of a failure, such as lack of response for communication between any two nodes due to node or network failures. We do not perform root cause analysis, which is orthogonal to our work. We assume that the system model provides temporal guarantees on communication bounds (sometimes also combined with computation bounds) called "partially synchronous" [18]. The FD utilizes a time-out based detection scheme, namely, a ping-pong based implementation with the following properties:

- Transparency: The FD can be embedded in MPI applications without any additional modification or side-by-side to MPI applications. For the former, the FD runs independently with a unique communicator different from an application program. When MPI applications call MPI_Init, the FD module is activated for each MPI task (on each node) as an independent thread through the MPI profiling interposing layer.
- Portability: MPI applications can be compiled without the FD. Applications only need to be re-linked with the profiling layer of MPI and the FT module. It is not necessary for MPI applications to change in their environment, design or source code. The FD works for arbitrary MPI implementations and has been tested with MPICH, Open MPI, and the LAM/MPI-family.
- Scalability: The FD operates in two modes. It can be configured to send a check message sporadically whenever the application has invoked a communication routine. An alternative setting performs periodic checks at configurable intervals.

The rationale behind sporadic and periodic liveness probing is that the former can be designed as low-cost background control messages that are only triggered when an application is causally dependent on other nodes. The latter, in contrast, can be designed independent of any communication pattern but requires constant out-of-band checking but is agnostic of application communication behavior.

The experimental results show that the FD satisfies the above three properties. The results further indicate that the sporadic approach imposes lower bandwidth requirements of the network for control messages and results in a lower frequency of control messages per se. Nonetheless, the periodic FD configuration is shown to result in *equal or better* application performance overall compared to a sporadic liveness test for larger number of nodes, which is a non-obvious result and one of our contributions. We also

observe that separation of the FD results in lower overheads as opposed to integration into the MPI applications. Our resulting implementation can easily be combined with reactive checkpoint/restart frameworks to trigger restarts after components have failed [3, 5, 8–11, 15, 17, 20–24].

2 Design

In principle, an FD can be designed using a variety of communication overlays to monitor liveness. A traditional heartbeat algorithm imposes high communication overhead in an all-to-all communication pattern with a message complexity $\Theta(n^2)$ and a time complexity of $\Omega(n)$. This overhead can be high, and a single node does not need to inquire about liveness of all other nodes in an MPI application.

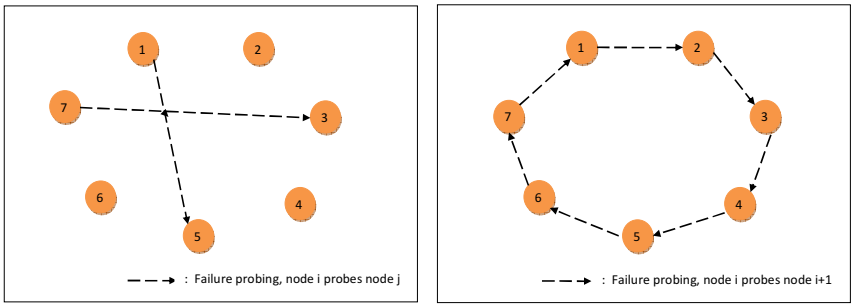
A tree-based liveness check results in $\Theta(n)$ messages with a $\Omega(log(n))$ time complexity where the root node has a collective view of liveness properties. However, mid-level failures of the tree easily result in graph partitioning so that entire subtrees may be considered dysfunctional due to the topological mapping of the overlay tree onto a physical network structure (e.g., a multi-dimensional torus).

We have designed two principle types of failure detection mechanisms. First, we support a sporadic (or on-demand) FD. Second, we have devised a periodic, ring-based FD. The periodic FD can be integrated into MPI applications or may function as a stand-alone liveness test separate from MPI applications. These approaches differ in their liveness probing periods and their network overlay structure.

2.1 Failure Detector Types

Periodic Ring-Based Failure Detection In this approach, starting from initialization of the MPI environment, we form a ring-based network overlay structure wherein the i-th node probes the $(i+1)$-th node in the ring (see Figure 1(b)). Thus, each node probes its neighbor in the ring irrespective of whether there is any active application communication between the two nodes or not. The probing is performed until the application terminates.

This structure results in $\Theta(n)$ messages for liveness checking and imposes $\mathcal{O}(1)$ time (assuming synchronous checking) or up to $\mathcal{O}(n)$ time (for asynchronous checking that has to propagate around the ring), yet liveness properties are only known about immediate neighbors. For MPI applications, we argue that local knowledge is sufficient to trigger reactive fault tolerance at a higher level.



(a) Sporadic Fault Detection. (b) Periodic Fault Detection.

Fig. 1. Fault Detection Techniques.

Sporadic/On-demand Failure Detection In this approach, a node p probes a node q only if p and q are engaged in an application-level point-to-point message exchange. If p needs to wait beyond a timeout interval for q to resume its work, a control message from p to q is issued (see Figure 1(a)). This happens when node p makes a blocking MPI call, such as MPI_Recv() or MPI_Wait(). Similarly, if the application is engaged in collective communication, such as MPI_Bcast(), and the MPI call does not return within a timeout interval, a ring-based liveness check is triggered. If the liveness test is successful but the application-level MPI call has not been completed, the liveness check is periodically repeated.

This method of liveness check imposes $\mathcal{O}(1)$ message and time overhead, and lifeless properties are only known to immediate neighbors. The control message overhead of this approach may be zero when responses to application messages are received prior to timeout expiration. In such a setting, the overhead is localized to a node and amounts to request queuing and request cancellation (in the best case).

3 Implementation

Our implementation assumes that there are reliable upper bounds on processing speeds and message transmission times. If a node fails to respond within a time-out interval, the node is assumed to have failed under this model (fail-stop model). The implementation builds on this assumption when a node starts probing another node. Node pairs are determined by a causal dependency implied from the application communication pattern (for sporadic point-to-point communication) or through network overlays (for sporadic collectives and all periodic liveness checks). Probing is implemented via ping-pong messages monitoring round trip time (RTT) timeouts. Probing for failure detection can be parametrized as follows: (a) INTER-PROBE: This interval determines the frequency of probing, i.e., the time between successive probes by a node. Longer values may cause late detection of failure while shorter intervals allow for early detection but increase the overhead imposed by control messages. (b) TIME-OUT: This interval determines the granularity of failure detection but also impacts the correctness of the approach. If the interval is chosen too small, a slow response may lead to false positives (detection of failure even though the node is functional). Conversely, a large interval may delay detection of failures. Determination of a "good" timeout interval is non-trivial, even if we assume an upper bound on network and processing delay (see above).

We have used the MPI profiling layer to implement one version of the FD module. Wrappers have been written for MPI functions. These wrappers take appropriate FD actions before and after calling the PMPI versions of the corresponding communication function. When the application calls MPI_Init(), a duplicate communicator, DUP_MPI_COMM_WORLD, is created, which is subsequently used to send control messages for failure detection. The software stack of the FD in conjunction with an application is depicted in Figure 2. Application-level MPI calls (depicted as Fortran calls) trigger a language-neutral wrapper before calling the interposed MPI function. In the PMPI wrapper, the native MPI function is called (prefixed with PMPI_). The fault detector governs back-channel exchanges of control message over the duplicated communicator. Another version of the FD implements periodic checks as stand-alone processes separate from an MPI application.

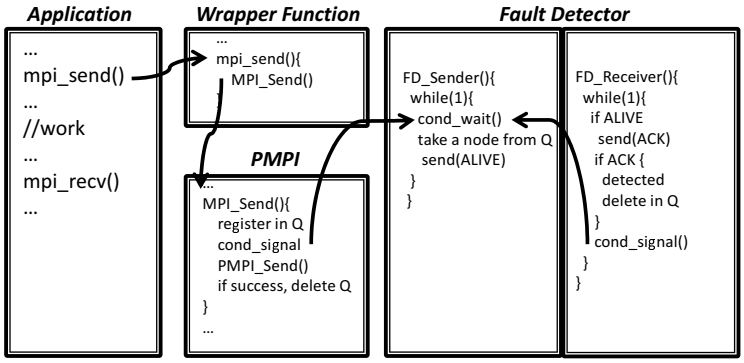


Fig. 2. Interaction of Application and Fault Detection Software Stacks

The fault detector is implemented as a pair of threads, namely sender and receiver threads. We require MPI to support multi-threading, i.e., MPI_Init_thread() should support MPI_THREAD_MULTIPLE to ensure thread support. The sender thread triggers an ALIVE message (ping) or waits for an acknowledgment (ACK) message (pong) up to a given timeout. The receiver thread receives ALIVE queries over the new communicator from the sender thread and responds with an ACK message. The failure detection module maintains a queue of events in sorted order of event times. An event could be "sending out a new probe to some node i" or "end of timeout interval for a probe sent to some node i". Upon such an event, the sender thread is activated and performs the respective action.

4 Performance Evaluation

We measured the overhead incurred by the FD module for the set of NAS parallel benchmarks (NPB V3.3) with input classes C and D [2]. Using gettimeofday(), wall-clock times of applications were measured between MPI_Init() and MPI_Finalize() calls with and without failure detector interpositioning or backgrounding. Tests were performed by running each configuration five times and computing the average overhead for different inter-probe intervals and number of processes.

4.1 Experimental Platform

Experiments were conducted on a 128 node cluster with Infiniband QDR. Each node is a 2-way shared-memory multiprocessor with two octo-core AMD Opteron 6128 processors (16 cores per nodes). Each node runs CentOS 5.5 Linux x86_64. We used Open MPI 1.5.1 for evaluating the performance of the FD.

4.2 Benchmark Results

Figures 3 and 4 depict the relative overheads of fault detection for 128 processes (over 64 nodes) with periodic and sporadic fault detection, respectively. (Due to space limitations, results for fewer nodes are omitted in Figures but are still discussed.) Overheads of the FD approach for fault tolerance with inter-probe frequencies of 1-10 second ("FD 1sec" to "FD 10 sec") are plotted relative to application execution without fault tolerance ("No FD"), i.e., in the absence of the FD module (normalized to 100%).

We first observe that both the sporadic and periodic FD have overheads ranging from less than 1% to 21% averaging around 10%. We further observe that periodic either matches or outperforms by 2-6% the sporadic approach. This trend is also visible

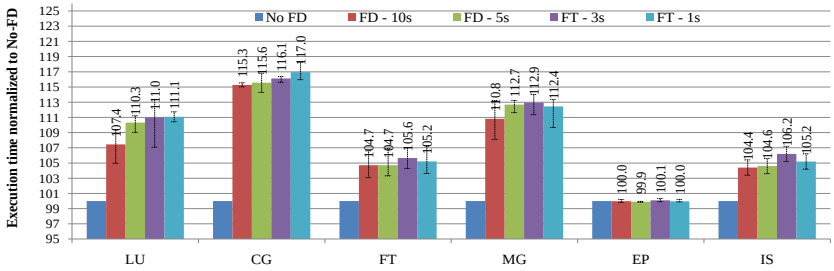


Fig. 3. Overhead of Periodic Fault Detection for 128 Processes

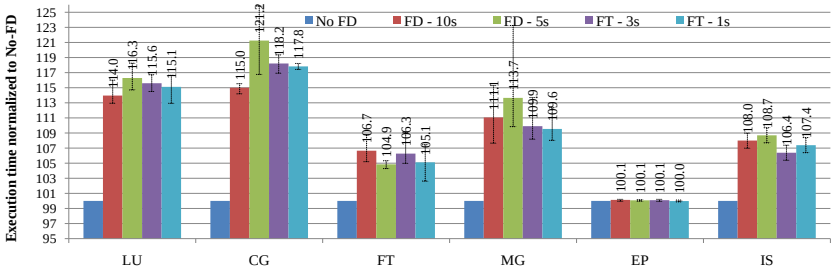


Fig. 4. Overhead of Sporadic Fault Detection for 128 Processes

for smaller number of tasks (although less pronounced) and can be explained as follows: As overall communication is increasing, timeouts in the sporadic mode happen more frequently, in particular for collectives where communication results in contention (e.g., for all-to-all collectives). Sporadic control messages only add to this application-induced contention. In contrast, the periodic approach has the advantage that control messages are evenly likely to occur across the entire application duration. This probabilistic spread frequently results in control messages being issued during computation, i.e., when the network interconnect is not utilized at all. This trend increases with strong scaling (larger number of nodes).

We further conducted experiments with periodic liveness checking as a background activity in separate processes across nodes that an MPI application is running on. The results (omitted due to space) show absolutely no overhead for NPB codes over 128 processes except for IS with an overhead of 4.5%. We also varied the number of MPI tasks per node and found these results to remain consistent up to 15 tasks per node. Only at 16 tasks per node did overheads spike to up to 28-60% depending on the NPB code. This shows that as long as a spare core is available for background activity, the impact of out-of-band communication on application performance is minimal. In HPC, applications tend to utilize only a subset of cores for high-end multi-core nodes as in our case, which ensures that communication does not become a bottleneck [16].

Besides these Infiniband experiments, we investigated the impact of our FD approaches under Gigabit Ethernet (result omitted). We found that the performance of NPB codes is significantly higher for Ethernet as execution becomes constrained by network contention given the lower bandwidth available. Hence, the overhead of FD was overshadowed by contention of application messages and did not result in a noticeable overall impact. However, we consider such a high contention scenario not realistic for well-balanced, tuned HPC codes.

Overall, the results show that periodic failure detection performs better than sporadic for communication intensive codes and that separation of the FD from MPI applications reduces their perturbation.

5 Related Work

Chandra and Toueg classify eight classes of failure detectors by specifying completeness and accuracy properties [4]. They further show how to reduce the eight failure detectors to four and discuss how to solve the consensus problem for each class. This paper has influenced other contemporary work as it raises the problem of false positives for asynchronous systems. In our work, we focus on single-point failure detection. Consensus is an orthogonal problem, and we simply assume that a stabilization after multi-component failures eventually allows reactive fault tolerance, such as restarts from a checkpoint, to occur in a synchronous manner. Sastry et al. discuss the impact of celerating environments due to heterogeneous systems where absolute speeds (execution progress) could increase or decrease [18]. Bichronal timers with the composition of action clocks and real-time clocks are able to cope with celerating environments. Our work, in contrast, only relies on the clock of a local node. Genaud and Rottanapoka implemented a fault detector in a P2P-MPI environment utilizing a heartbeat approach [7]. They address failure information sharing, reason about a consensus phase and acknowledge overheads of fault detection due to their periodic heartbeat approach. Our work, in contrast, results in much lower message and time complexity. Consensus is orthogonal to our work, as discussed before.

6 Conclusion

In summary, our work contributes generic capabilities for fault detection / liveness monitoring of nodes and network links both at the MPI level and stand alone. We designed and implemented two approaches to this end. The first approach utilizes a periodic liveness test and utilizes a ring-based network overlay for control messages. The second method promotes sporadic checks upon communication activities and relies on point-to-point control messages along the same communication paths utilized by the application, yet falls back to the ring-based overlay for collectives. We provide a generic interposing of MPI applications to realize fault detection for both cases plus a stand-alone version for the periodic case. Experimental results indicate that while the sporadic fault detector saves on network bandwidth by generating probes only when an MPI call is made, its messages are increasingly contending with application messages as the number of nodes increases. In contrast, periodic fault detection statistically avoids network contention as the number of processors increases. Overall, the results show that periodic failure detection performs better than sporadic for communication intensive codes and that separation of the FD from MPI applications reduces their perturbation.

References

1. A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations. In *Proc. of the 8th IEEE Intl. Symp. on High Perf. Distr. Comp.*, 1999.
2. D. H. Bailey et al. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.
3. G. Bosilca, A. Boutellier, and F. Cappello. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Supercomputing*, Nov. 2002.

4. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43:225–267, March 1996.
5. J. Duell. The design and implementation of berkeley lab's linux checkpoint/restart. Tr, Lawrence Berkeley National Laboratory, 2000.
6. G. E. Fagg and J. J. Dongarra. FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world. In *Euro PVM/MPI Meeting*, volume 1908, pages 346–353, 2000.
7. S. Genaud and C. Rattanapoka. Evaluation of replication and fault detection in p2p-mpi. In *Intl. Par. and Distrib. Proc. Symp.*, 2009.
8. R. Gioiosa, J. C. Sancho, S. Jiang, and F. Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Supercomputing*, 2005.
9. J. Heo, S. Yi, Y. Cho, J. Hong, and S. Y. Shin. Space-efficient page-level incremental checkpointing. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1558–1562, New York, NY, USA, 2005. ACM.
10. S.-T. Hsu and R.-C. Chang. Continuous checkpointing: joining the checkpointing with virtual memory paging. *Softw. Pract. Exper.*, 27(9):1103–1120, 1997.
11. J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In *12th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems*, 03 2007.
12. H. Jitsumoto, T. Endo, and S. Matsuoka. Abaris: An adaptable fault detection/recovery component framework for mpis. In *Intl. Par. and Distrib. Proc. Symp.*, 2007.
13. A. Moody, G. Bronevetsky, K. Mohror, and B. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Supercomputing*, Nov. 2010.
14. I. Philp. Software failures and the road to a petaflop machine. In *HPCRI: 1st Workshop on High Performance Computing Reliability Issues*. IEEE Computer Society, 2005.
15. J. Ruscio, M. Heffner, and S. Varadarajan. Dejavu: Transparent user-level checkpointing, migration, and recovery for distributed systems. In *Intl. Par. and Distrib. Proc. Symp.*, 2007.
16. J. Sancho, D. Kerbyson, and M. Lang. Characterizing the impact of using spare-cores on application performance. In *Euro-Par Conference*, pages 74–85, Sept. 2010.
17. S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proceedings, LACSI Symposium*, Oct. 2003.
18. S. Sastry, S. M. Pike, and J. L. Welch. Crash fault detection in celerating environments. In *Intl. Par. and Distrib. Proc. Symp.*, 2009.
19. B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the 2006 International Conference on Dependable Systems and Networks (DSN-2006)*, Philadelphia, PA, June 2006.
20. G. Stellner. CoCheck: checkpointing and process migration for MPI. In IEEE, editor, *Proceedings of IPPS '96. The 10th International Parallel Processing Symposium: Honolulu, HI, USA, 15–19 April 1996*, pages 526–531, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.
21. J. Varma, C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Scalable, fault-tolerant membership for MPI tasks on hpc systems. In *International Conference on Supercomputing*, pages 219–228, June 2006.
22. C. Wang, F. Mueller, C. Engelmann, and S. Scott. A job pause service under LAM/MPI+BLCR for transparent fault tolerance. In *Intl. Par. and Distrib. Proc. Symp.*, Apr. 2007.
23. C. Wang, F. Mueller, C. Engelmann, and S. Scott. Proactive process-level live migration in hpc environments. In *Supercomputing*, 2008.

24. S. Yi, J. Heo, Y. Cho, and J. Hong. Adaptive page-level incremental checkpointing based on expected recovery time. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1472–1476, New York, NY, USA, 2006. ACM.

7 Appendix

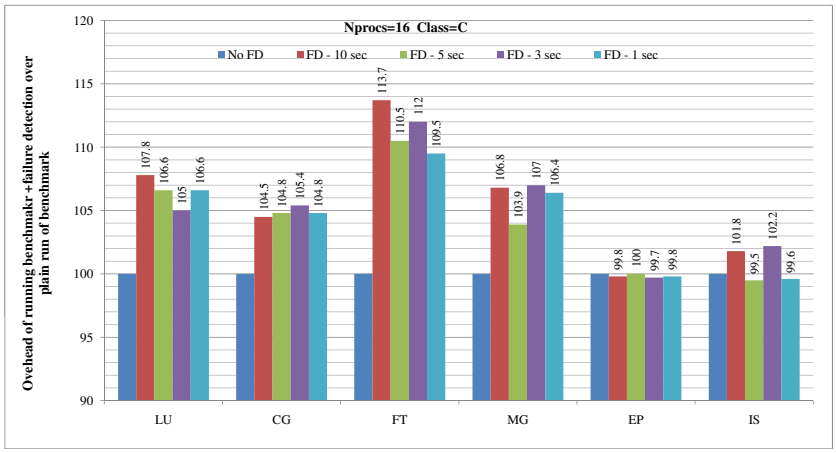


Fig. 5. Overhead of Periodic Fault Detection for 16 Processes

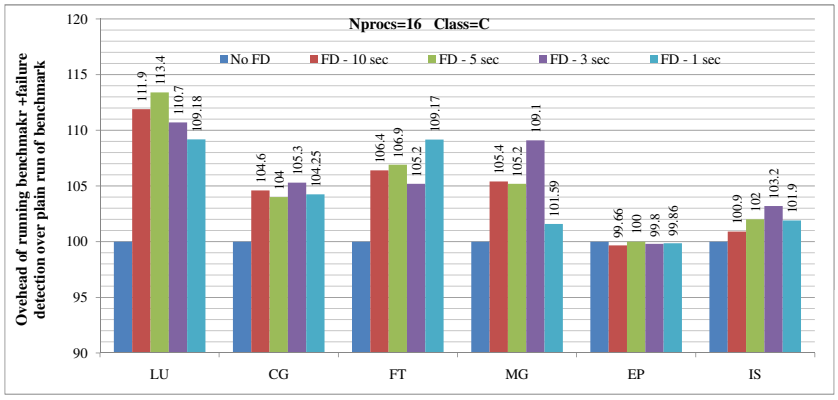


Fig. 6. Overhead of Sporadic Fault Detection for 16 Processes

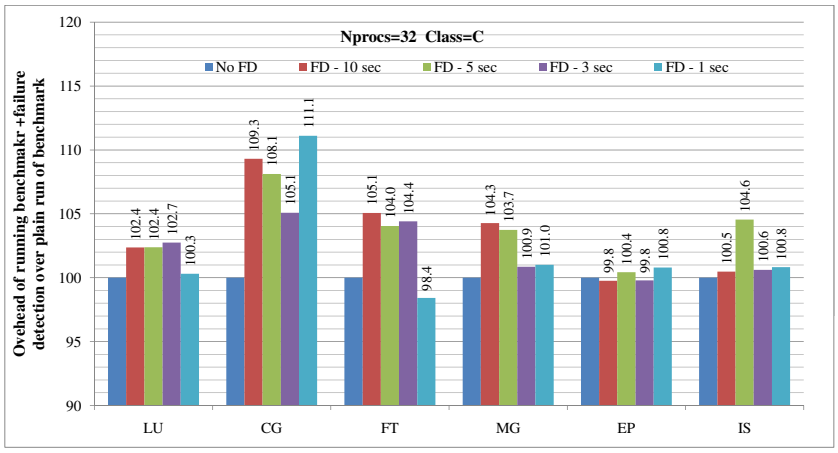


Fig. 7. Overhead of Periodic Fault Detection for 32 Processes

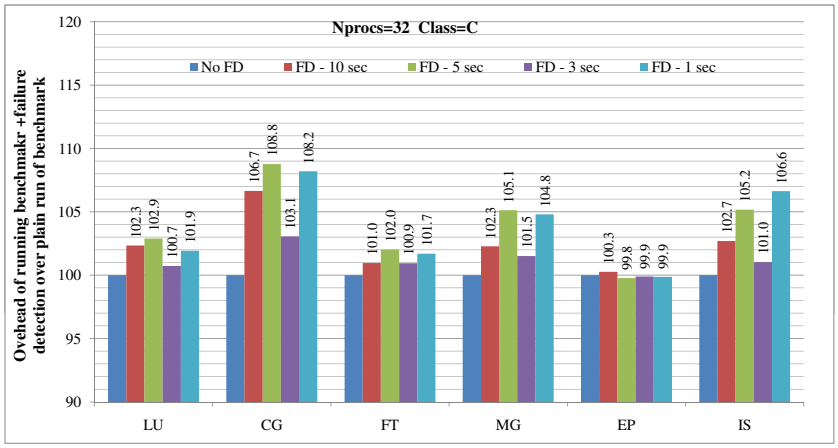


Fig. 8. Overhead of Sporadic Fault Detection for 32 Processes

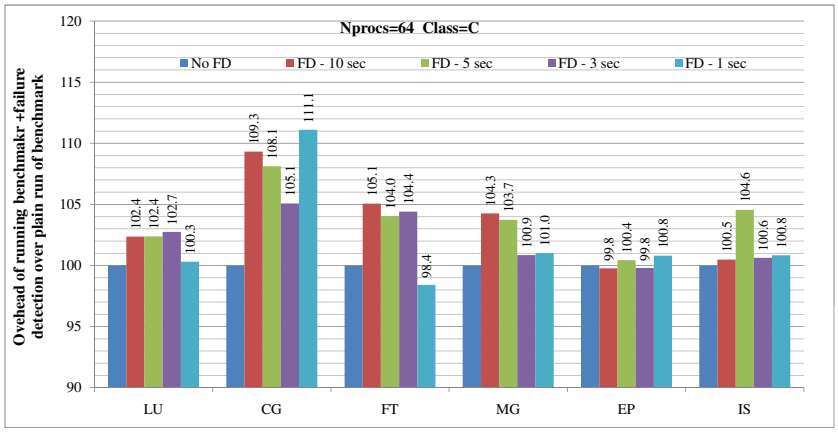


Fig. 9. Overhead of Periodic Fault Detection for 64 Processes

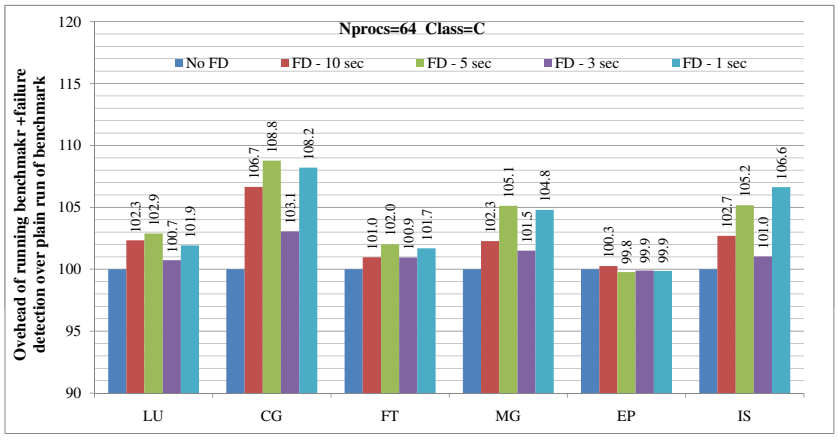


Fig. 10. Overhead of Sporadic Fault Detection for 64 Processes

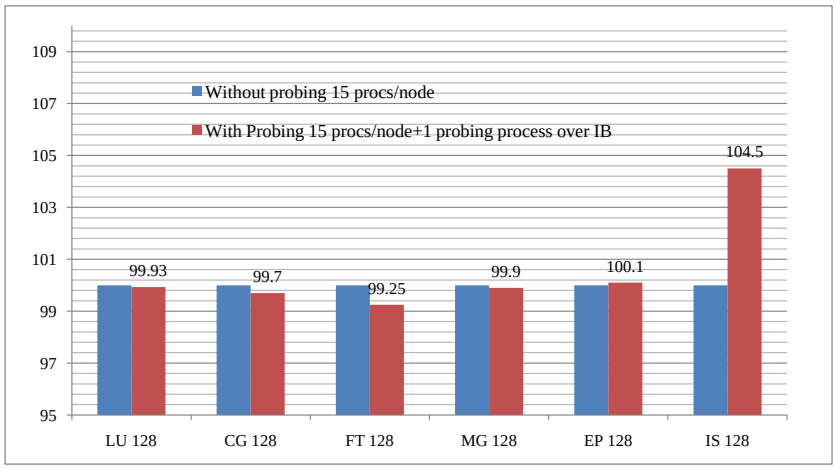


Fig. 11. Overhead of Failure detection as a seperate process using IB (directly over IB, without TCP)- 15 MPI process/node

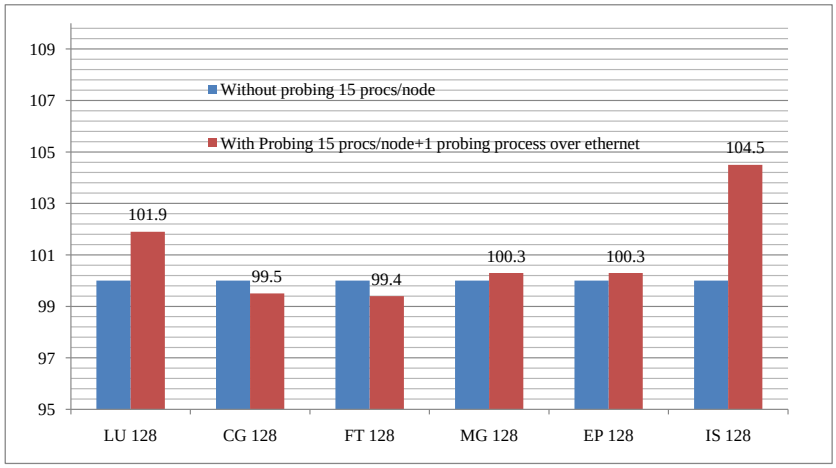


Fig. 12. Overhead of Failure detection as a seperate process using ETH (TCP over ETH) - 15 MPI process/node

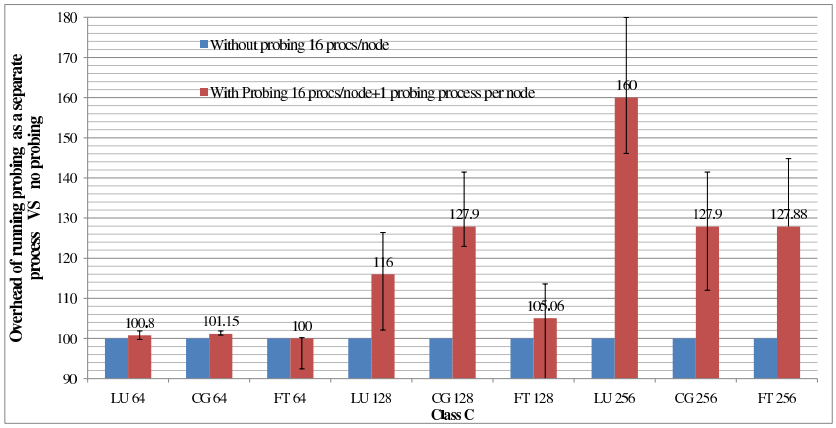


Fig. 13. Overhead of Failure detection as a seperate process using ETH (directly over IB, without TCP)- 16 MPI process/node

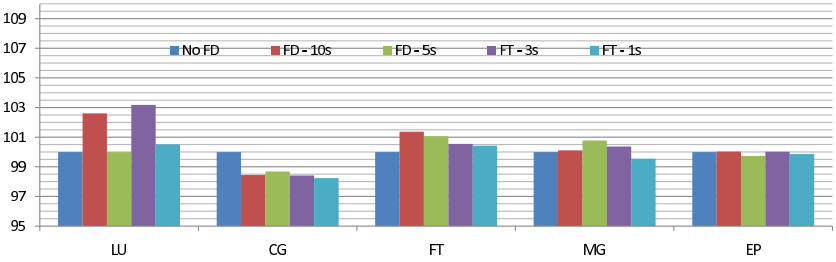


Fig. 14. Overhead of integrated periodic Failure detection for 128 processes (4 * 32 nodes) using ETH

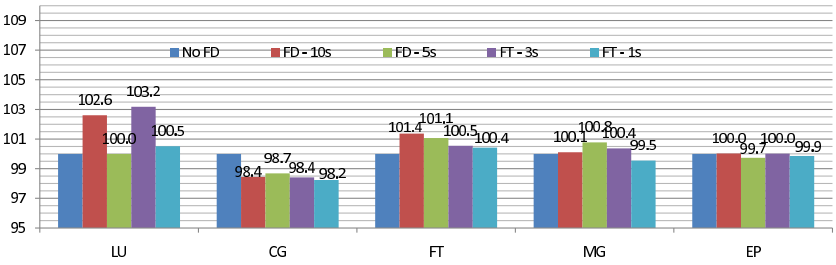


Fig. 15. Overhead of integrated sporadic Failure detection for 128 processes (4 * 32 nodes) using ETH