Camargo and Duarte Jr. *Journal of the Brazilian Computer Society* (2018) 24:5
https://doi.org/10.1186/s13173-018-0069-z

Journal of the
Brazilian Computer Society

RESEARCH Open Access

Running resilient MPI applications on a Dynamic Group of Recommended Processes

Edson Tavares de Camargo[1,2*] and Elias P. Duarte Jr.[1]

Abstract

High-performance computing systems run applications that can take several hours to execute and have to deal with the occurrence of a potentially large number of faults. Most of the existing fault-tolerant strategies for these systems assume crash faults that are permanent events are easily detected. This is not the case in several real systems, in particular in shared clusters, in which even the load variation may cause performance problems that are virtually equivalent to faults. In this work, we present a new model to deal with this problem in which processes execute tests among themselves in order to determine whether the processors (or cores) on which they are running are *recommended* or *non-recommended*. Processes classified as recommended form a Dynamic Group of Recommended Processes (DGRP) that runs the application. The DGRP is formed only by processes that have not been tested as non-recommended by all DGRP processes. A process not in the DGRP that is continuously tested as recommended can rejoin the DGRP after a round of consensus executed by DGRP processes. Experimental results are presented obtained from a MPI-based implementation in which the HyperQuickSort parallel sorting algorithm reconfigures itself at runtime to tolerate up to $N - 1$ faults (in a system with N processes) while sorting up to 1 billion integers.

Keywords: Dynamic Group of Recommended Processes (DGRP), Resilience, Fault tolerance, MPI applications, HPC systems

Introduction

High-performance computing (HPC) systems are used to execute complex industrial and scientific simulations, as well as other computing-intensive applications. These systems, in particular petascale and future exascale systems, are required to cope with an increasingly smaller mean time between failures (MTBF) [1]. For example, the Blue Waters petascale system has an average MTBF of 4.2 h [2]. Future exascale systems should present an even lower MTBF and experience various kinds of faults [3, 4]. In [5], the term "performance fault" is introduced to describe performance anomalies that can harm the operation of HPC applications as much as failures. For example, a processor may reduce the core operation frequency when the temperature rises above a safe threshold or to maintain the system within the power budget target.

The Message-Passing Interface (MPI), which is a de facto standard to program parallel and distributed applications defined and maintained by the MPI Forum [6, 7], assumes a reliable underlying infrastructure [1, 8]. MPI does not prescribe how implementations must deal with failures [9, 10]. As a consequence, the most widely used MPI implementations, such as OpenMPI and MPICH, abort the entire application if a single process fails. The application must then restart from the beginning. In order to circumvent that issue, the MPI Forum has recently proposed the User Level Failure Mitigation (ULFM) proposal [9]. The ULFM proposal has a set of primitives to enable developers of a MPI application to deal with process faults. ULFM assumes the fail-stop model, in which every process that fails is identified and removed from the system.

In this work, we describe a strategy for identifying and maintaining a Dynamic Group of Recommended Processes (DGRP) in a MPI-based HPC system. The DGRP was inspired by group systems such as Isis [11]. A DGRP can be seen as a wormhole [12] consisting of a dynamic set of processes that continuously execute the application. Processes execute *tests* among themselves in order to determine whether the processors (or cores) on which they are running are *recommended* or *non-recommended*.

*Correspondence: edson@utfpr.edu.br
[1]Department of Informatics, Federal University of Paraná (UFPR), Curitiba, Brazil
[2]Federal Technology University of Paraná (UTFPR), Toledo, Brazil

In other words, a test procedure is defined to "measure" whether the behavior of a given process is "good enough" for the application at hand. The test procedure is modular and can be defined for each particular system, depending on the characteristics of the environment. For instance, the tester may send a program to be executed by the tested process and, depending on how long it takes to receive the corresponding reply, classifies the tested node as good enough to join the group or not. The test procedure must be carefully chosen in order not to interfere on the performance of the system itself.

A process that does not pass a test is considered to be *non-recommended* by the tester. On the other hand, a *recommended* process is one that presented the correct, expected behavior. Note that a process can be non-recommended for a short period of time due for instance to a load surge. Such a process may not be able to run the application for that time interval but can later revert the status and rejoin the system. On the other hand, some other process may remain non-recommended most of the time. These processes should be removed from the systems as soon as possible. Figure 1 shows the performance of a single processor of a shared cluster in which a parallel MPI program for approximating Π (pi) is executed using 16 cores. This is a representative set of results showing 100 consecutive executions. Note that the performance presents a significant variation along the time. The first executions took long to complete. Results such as these can be used to assess whether this processor is or is not good enough to run an application for a given time frame. A process that is in the DGRP has not been tested as *non-recommended* by any other DGRP process. Processes that are tested as *non-recommended* are removed from the DGRP. A process not in the DGRP that is tested as recommended for ζ consecutive tests by others can rejoin the DGRP after a round of consensus executed by DGRP process. DGRP is particularly suitable for moldable applications that can be reconfigured at runtime [13].

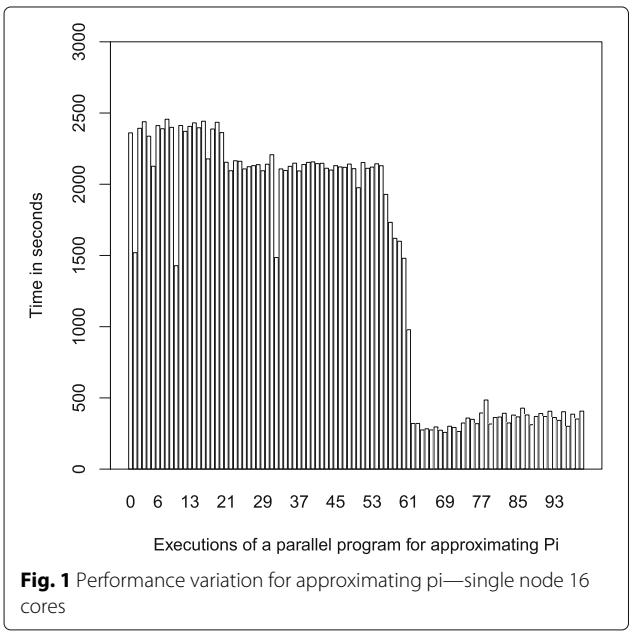

Fig. 1 Performance variation for approximating pi—single node 16 cores

The system model for deciding on the recommendations is based on system-level diagnosis theory [14]. The objective of system-level diagnosis is to employ tests to identify which units are working according to the specification and which are not. Diagnosis is thus based on processing the results of the set of tests performed between the units of a system. Most diagnosis models assume that a fault-free unit executes tests and reports test results reliably, i.e., a fault-free tester can always correctly determine whether the tested unit is faulty or fault-free [15, 16]. The model we propose in this work circumvents this assumption in the sense that tests are not meant to perfectly determine whether the tested entity is faulty or fault-free but only that the test criterion was not met. Furthermore, two different testers may reach different conclusions about the state of a given tested process.

The DGRP abstraction was implemented using MPI on top of ULFM. We report results for running the parallel algorithm HyperQuickSort [17] for sorting up to 1 billion integers. This algorithm organizes the processes as a virtual hypercube. In our work, HyperQuickSort was adapted to reconfigure itself at runtime in order to proceed even if up to $N - 1$ processes become non-recommended (N is the total number of processes). The overhead of DGRP is obtained by comparing the execution of HyperQuickSort over DGRP and with ULFM. We also show the performance of DGRP for monitoring a system as nodes are detected as non-recommended and removed from the DGRP and also as nodes previously classified as non-recommended rejoin the DGRP.

The rest of the paper is organized as follows: Related work is described in the "Related work" section. The proposed strategy is presented in "The Dynamic Group of Recommended Processes" section. The implementation is described in the "DGRP implementation and case study" section, and experimental results are presented in the "Results" section. The conclusions follow in the "Conclusion" section.

Related work

As DGRP is implemented on top of ULFM, it is also capable of dealing with process failures. Related works in this section describe different approaches to make MPI systems fault-tolerant and also strategies for MPI monitoring, including the detection of performance anomalies.

MPI allows the development of parallel and distributed applications based on the message-passing paradigm [7, 8]. The MPI standard does not currently have a specification for fault tolerance [8]; there is no definition of the precise behavior that MPI implementations must take to deal with faults [9, 10]. Basically, a fault is treated as internal to an application, e.g., the violation of memory space. Thus, process and network faults are treated as application problems, and the responsibility to deal with them is left to the application programmer. Note that the standard does define error handlers that are associated with the MPI communicator to handle application program errors.

Most approaches for the design of fault-tolerant MPI applications (actually fault-tolerant HPC in general) are based on checkpoint-restart [1]. By using checkpoint-restart, an application can deal with a process failure without completely losing previously computed results. It has been noted though that checkpoint-restart may not be effective against a short MTBF [1, 4, 18]. However, recently, several checkpoint-restart protocols have been proposed for dealing with a short MTBF in large HPC systems, e.g., [4, 19, 20].

Several efforts have been made to add interfaces and semantics to allow MPI to deal with faults. Fault-tolerant MPI (FT-MPI) [21] is the first strategy proposed to offer an alternative to traditional checkpoint-restart, defining MPI primitives that enable the application to survive faults. Both the MPI communicator and processes are assigned states. Processes can be either *ok*, *unavailable*, *joining*, or *failed*. After an error indicates the presence of a fault, the system acts immediately taking into account the state of the communicator and the recovery mode adopted by the application. For example, in one recovery mode, information about processes that have failed is received by correct processes, and processes that failed are removed from the system. The FT-MPI specification does not include details about the strategy used for detecting faults [22]. Despite representing by itself a key contribution, the FT-MPI standard was deprecated. Other works, including [10, 22, 23] followed the same approach defining MPI primitives that allow the application to survive faults.

Another strategy based on FT-MPI has been proposed: Non-Stop and Fault-Resilient MPI (NR-MPI) [23]. NR-MPI implements the semantics of FT-MPI on MPICH, one of the most widely used MPI implementations [24]. NR-MPI assumes the fail-stop model. It is designed to allow MPI itself to recover from faults internally and automatically. If recovery is not possible, the state of the world communicator (`MPI_COMM_WORLD`) becomes invalid, meaning that unrecovered faults have occurred. Faulty processes need to be replaced, either by spawning new processes or by using spare processes. One of the main contributions of NR-MPI is that it includes mechanisms for detecting faults. There are two modules called Failure Arbiter (FA) and Failure Detector (FD). These modules are integrated to a Resource Management System (RMS); this is a separate system that provides monitoring information. The RMS consists of a Resource Manager and a Process Manager. The FA is located at the Resource Manager and the FD is located at the Process Manager. FDs detect process faults (crash) by monitoring system calls. FA uses a periodic heartbeat to detect FD faults. A drawback of NR-MPI is its strong reliance on the external RMS.

The MPI Forum created in 2009 is a working group with the responsibility of optimizing the MPI standard to allow the development of portable, scalable, and fault-tolerant HPC applications [25]. Efforts of the working group resulted in two draft proposals: Run-Through Stabilization (RTS) [25] and User Level Failure Mitigation (ULFM) [9]. ULFM is currently a proposed standard. ULFM also aims at allowing the application to survive despite the occurrence of faults. But, unlike FT-MPI, recovery is not automatic. ULFM defines a set of primitives that allow the application developer to implement any suitable recovery strategy.

ULFM assumes the fail-stop model. Error handlers, defined in the MPI standard, are used to inform the application about faults. Fault detection is local, in the sense that a fault is detected only by processes that directly communicate with the faulty process. Essentially, the fault is detected whenever a correct process cannot communicate with another process—which is then assumed to be faulty. ULFM assumes that transient network and process faults do not occur, but at the implementation level, it is possible to deal with these types of faults. If a correct process identifies another process as unresponsive (even if that process is not responding for a short period of time), the correct process classifies this process as faulty and from that point on ignores and discards any message received from the faulty process.

A consensus protocol to build fault-tolerant HPC applications which proposes an agreement algorithm implemented within the ULFM API is proposed in [26]. The algorithm assumes the fail-stop model. The communication channels are reliable. In the algorithm, all processes propose a unique value. The decided value is the result of a combination of all values proposed. Previously, in [27, 28], agreement algorithms were proposed to be used within RTS.

Other approaches for programming fault-tolerant MPI applications are based on Algorithm-Based Fault Tolerance (ABFT) [29]. Chen and Dongarra use ABFT to matrix computations, but they modified the original ABFT strategy to allow its application to more general HPC systems by supporting process faults [30]; they assume the fail-stop model. Using ABFT, the application can recover

it errors and process faults without using checkpoints or message logs. A drawback of this technique is that it is restricted to the application domain in which it is proposed.

Fenix [31, 32] is a framework that allows transparent runtime recovery of MPI applications. The framework makes use of the ULFM specification to survive failures and employs the diskless checkpoint technique: application data is saved in the memory of neighboring nodes [33]. Primitives are available for the developer to define checkpoints on essential data. Fenix adopts implicit checkpoints calls which are saved in a non-coordinated way; however, depending on where the checkpoints are inserted into the code, there is a guarantee that consistent global states are always generated by the application. The evaluation of Fenix was performed using an MPI application running thousands of processes. Unfortunately, the framework is not publicly available.

Ferreira et al. [34] employ state-machine replication [11] in the context of HPC systems. Despite the potentially high cost of state-machine replication, high availability is guaranteed. In case a MPI process fails, redundant processes allow the application to continue its execution transparently, without the need for rollback recovery.

Genaud et al. [35] also use replication as the means to achieve reliability applied to a grid middleware called P2P-MPI. The system has a module for fault detection based on execution monitoring. Other works that apply fault tolerance to MPI settings have been proposed, including the detection and correction of silent errors before they lead to system restart [36]. Despite these efforts, it is well-known that running fault-tolerant large-scale MPI applications is still an open problem.

Adaptive MPI (AMPI) [37] is an adaptive implementation of MPI built on Charm++ [38]. Charm++ is an object-based, message-driven parallel programming framework that embodies the concept of processor virtualization. AMPI inherits most of the advantages of Charm++, as adaptive overlapping of communication and computation, automatic load balance, and fault tolerance. One of the drawbacks of AMPI is that to be used, it requires the MPI code to be modified.

In [5], the authors introduce the term "performance fault" to describe anomalous behavior in HPC systems. They propose the design and implementation of a monitoring system that continuously inspects the evolution of running applications, and report performance anomalies. They use an approach which had been originally proposed in [39] through which performance problems are detected as discrepancies between (1) the actual execution of an application and (2) a performance model. Sensors continuously inspect the evolution of running applications and collect information about the application and system "health". The execution is considered to be correct if it meets the performance levels dictated by the model. In this way, the performance model can be used in runtime to disambiguate false positives from real hardware/software problems. A drawback of that approach is that the user has to provide the model in advance. But an advantage is that this approach provides a setting for adapting multiple performance models, in the sense that any specific model can be plugged to the system without modifying to the kernel-level monitoring system or the system monitor interface. The main differences of that work to ours is that we employ tests to evaluate and identify performance anomalies at runtime; furthermore, we use the obtained information to build the group of recommended processes.

In [40], the authors state that there are few tools for monitoring the performance of HPC systems at runtime. According to the authors, performance data collected during runtime can be useful to distribute and balance the workload in different ways and for different purposes, for example, to reduce power consumption. They build an extension of the Integrated Performance Monitoring (IPM) tool[1] that provides online runtime access to application execution performance data through a set of primitives. Thus, it is possible to take decisions on how to guide computations according to the observed performance state. The performance data collected by the tool could be used to build a performance model to detect anomalies.

The works in [41, 42] built on top of Tuning and Analysis Utilities (TAU) [43] are tools for online monitoring. TAU is a profiling and tracing toolkit for performance analysis of parallel programs. It is capable of gathering performance information through instrumentation of functions, methods, basic blocks, and statements as well as event-based sampling. TAUg [41] considers the problem of runtime support for application level access to global parallel performance data. It uses the MPI library in order to share performance data among processes. TAUg allows users to access one metric at a time and only provides raw performance data. TAUoverSupermon [42] also uses TAU as a monitoring system, and Supermon [44] is used to collect the distributed measurements. This system provides information on performance from different contexts and delivers the data to monitoring consumers.

Table 1 summarizes all proposals and the main strategies described in this section. The table also presents DGRP. Our group system allows not only to deal with faults, once it is built on top of ULFM, but also with performance issues—the purpose is to keep a group of processes that have a high probability of presenting good performance. Checkpoint-restart, ABFT, and state-machine replication strategies can all be applied on top of the group of recommended processes, that is, our

Table 1 Proposals and main strategies

Related work	Main strategies adopted
FT-MPI (fault-tolerant MPI) [21], Non-Stop and Fault-Resilient MPI (NR-MPI) [23], Run-Through Stabilization (RTS) [25], User Level Failure Mitigation (ULFM) [9], Consensus Protocol [26–28], Adaptive MPI (AMPI) [37]	Primitives for dealing with fault tolerance at the application level
Fenix [31, 32]	Checkpoint-restart at the application level
Dealing with process faults using ABFT [30]	Algorithm-Based Fault Tolerance (ABFT)
Ferreira et al. [34], P2P-MPI [35], Fiala, et al. [36], Silent error [36]	State-machine replication
Gioiosa et al. [5], Aguilar et al. [40], TAUoverSupermon [42]	Monitoring system for performance
DGRP	Monitoring system that recommends a group of processes to run an application

contribution is orthogonal and can be used by these strategies. The next section presents DGRP in detail.

The Dynamic Group of Recommended Processes

In this work, we describe a strategy for identifying and maintaining a Dynamic Group of Recommended Processes (DGRP) in MPI-based HPC systems. DGRP was inspired by group systems such as Isis [11]. The similarity is in the sense that a DGRP is a process group with self-managed membership. However, Isis and other related systems present so many expensive features that are not required by a DGRP (virtual synchrony, atomic/causal broadcast primitives, different levels of consistency, etc.); thus, we believe they fall into a completely different category. DGRP is simply a self-managed group of processes that allocates the tasks of the next computing round based on test results. A DGRP can be also seen as a wormhole [12] consisting of a dynamic set of processes that are good enough to execute tasks of the parallel MPI application. The *wormholes* distributed system model was proposed by Veríssimo [12]. According to this model, current network environments often present a spectrum of synchrony that varies from components that present perfectly predictable behavior to those that have a completely uncertain behavior. These properties can be defined in time, i.e., during the timeline of their execution, systems become faster or slower, presenting lower or higher bounds to execute. These properties can also be defined in space: some components are more predictable and/or faster than others and actions performed in or among these nodes present better defined and/or smaller bounds. In the wormholes model, different loci of the system have different properties which correspond to different sets of assumptions.

The DGRP system model is defined based on system-level diagnosis theory; a brief overview of the key concepts of diagnosis is presented next. This is followed by a description of the proposed DGRP and the proposed model.

System-level diagnosis: a very brief overview

System-level diagnosis is an approach for system monitoring based on the execution of tests among the system units [14, 45]. The set of test results is called the syndrome. By processing the syndrome, it is possible to determine which nodes are faulty or fault-free (fault-free means behaving as expected, according to the specification). The first system-level diagnosis model, the PMC model [46], was proposed 50 years ago. Since then, a large number of models, approaches, and theoretical results have been presented and applied to an enormously broad spectrum of contexts, from the diagnosis of chip defects at the wafer-scale integration level to diagnosis of multiprocessor computers based on several topologies to monitoring computer networks, among many others.

The PMC model defines a diagnosis model for a system S that consists of a set of N independent units that execute tests on each other. A test involves the controlled application of stimuli and the observation of the corresponding responses from the tested unit. In fact, a test is defined as a "diagnostic program" tailored for each system. The PMC model assumes that a fault-free unit is able to execute tests and report test results reliably. No assumptions are made about tests executed by faulty units, that is, they may produce incorrect test outcomes. The definition of which units test which other units is called the *connection assignment* and is represented by a direct graph. The syndrome is processed by a reliable external entity which diagnoses the system, that is, determines the state of all system units. A system is said to be t-diagnosable if up to t faulty nodes can be correctly diagnosed.

In adaptive diagnosis [47], instead of always executing a fixed set of tests, a node determines which tests they will execute based on the results of previously executed

tests. In distributed diagnosis [48], fault-free nodes process the syndrome in order to diagnose the system (the original PMC model employs a central observer responsible for that task). Nodes execute tests and exchange test results with each other. Later, adaptive and distributed diagnosis models and algorithms appeared that allowed the theory to be implemented to monitor computer networks [49].

A node running an adaptive and distributed diagnosis algorithm executes tests at a periodic testing interval. A testing round is the interval in which all fault-free nodes have executed their assigned tests. If an event corresponds to a fault-free node becoming faulty or vice versa, the diagnosis latency is the number of testing rounds it takes for all fault-free nodes to diagnose an event. Most adaptive and distributed diagnosis algorithms present the diagnosability equal to $N - 1$, i.e., even if all but one node are faulty, diagnosis still completes correctly.

DGRP: description

DGRP is based on a distributed and adaptive system-level diagnosis model. Processes execute tests among themselves in order to determine whether they are *recommended* or *non-recommended*. In other words, a test procedure is defined to measure whether the behavior of a given process is good enough for the application at hand, if it is then the tested process is recommended. A process that is in the DGRP has not been tested as non-recommended by any other DGRP process. Processes that are tested as non-recommended are thus removed from the DGRP. Figure 2 shows a DGRP formed by nodes 0, 2, 5, and 7.

A process not in the DGRP that is tested as non-recommended for ζ consecutive tests by others can rejoin the DGRP after a round of consensus executed by DGRP processes. Figure 3 shows a DGRP formed by nodes 2, 5, 6, and 7. Comparing DGRP of Fig. 3 with that of Fig. 2 process 0 was tested as non-recommended and was removed from DGRP. On the other hand, process 6 was reintegrated to the DGRP.

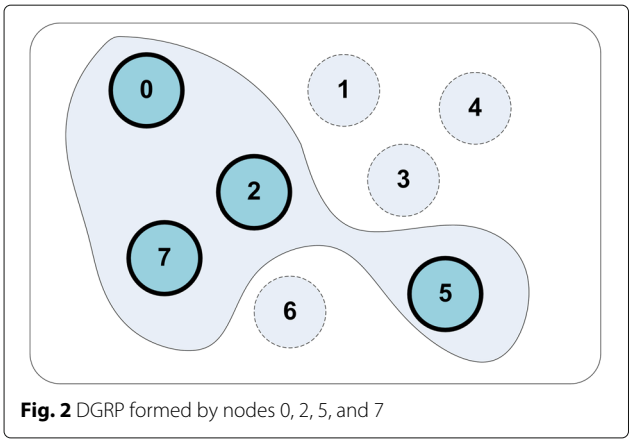

Fig. 2 DGRP formed by nodes 0, 2, 5, and 7

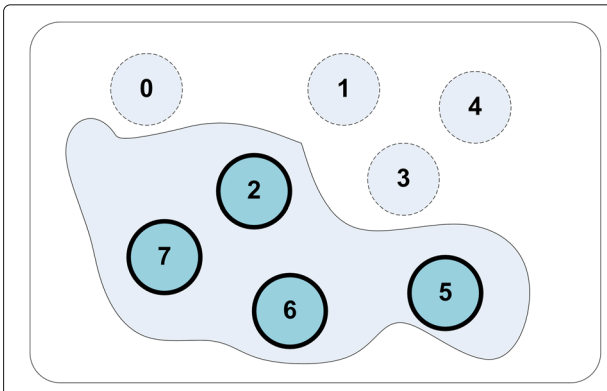

Fig. 3 DGRP formed by nodes 2, 5, 6, and 7

Processes in the proposed model execute in *computing rounds*, in which tasks of the parallel application are executed, along with *diagnosis*, as shown in Fig. 4. In this figure, we show the application process executing (in black), diagnosis is being concurrently executed (shown in blue), and the barriers at which processes synchronize (vertical bars). In a *computing round*, the parallel application is executed up to a *barrier*, which is an abstraction that allows processes to synchronize, i.e., a process only goes through a barrier after all processes have arrived at the barrier. Diagnosis runs concurrently, as an underlying monitoring system. As processes arrive at a barrier, they check the DGRP composition in order to reassign tasks among recommended processes. The barrier can be configured to determine how much time is spent on a computing round, for example, taking into consideration the expected MTBF.

DGRP: system model

The system is represented as a complete undirected graph $G = (V, E)$; the set of vertices V corresponds to the set of processes and an edge $i, j \in E \mid i, j \in V$ represents the ability of processes i and j to communicate directly, without intermediates. The system is not synchronous, i.e., there are no known bounds on message transmission delays and the relative speeds of processes. Communication channels are unreliable. Process i can be in one of two possible states: *recommended* or *non-recommended*. A process executes a test procedure on another process in order to determine its state. A tested process that passes a test is classified as recommended; otherwise, it is considered to be non-recommended. An event corresponds to a process state toggling from recommended to non-recommended or vice versa.

Tests are executed at periodic *testing intervals*, e.g., 10 ms or 3 s. At each testing interval, a recommended process in the system executes tests on other processes,

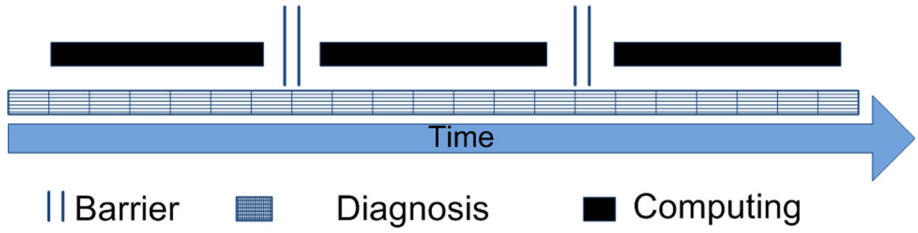

Fig. 4 A computing round ends on a barrier

according to the connection assignment or testing graph. The testing graph $T = (V, A)$ is a directed graph in which the set of vertices V corresponds to the set of processes. A directed arc $(i, j) \in A$ corresponds to a test node i has executed on node j. Each process employs its local clock to determine the testing interval. The test procedure is assumed to be complete enough for the tester to assess the state of the tested process (from its point of view). Thus, the specification of a test often depends on the system technology. A *testing strategy* defines which tests are assigned to which testers. A *testing round* is defined as the period of time by which every recommended process in the system has executed its assigned tests. The *diagnosis latency* is defined as the number of testing rounds required for all recommended nodes in the system to complete the diagnosis of an event.

Each process stores information about the states of all other processes locally. Actually, each node stores a counter of events [50]; initially, every process is assumed to be recommended and the corresponding counter is set to 0; after an event is detected, the counter is incremented to 1 (the new state is non-recommended) and so on. An even counter corresponds to a recommended process and an odd counter to a non-recommended process. A recommended process that receives diagnostic information from another recommended process checks whether the state counter of any process is greater than that currently maintained locally. In this case, the state counter is updated with the new information. Counters help the identification of how often a process has been toggling states.

The *Dynamic Group of Recommended Processes* - DGRP is defined as follows: If process $i \in$ DGRP then $\forall j \in$ DGRP $\mid j$ tested i, i passed the test, i.e., i is tested as recommended by all recommended processes. After a recommended node is tested as non-recommended by a process $\in$ DGRP, the tester disseminates this event information to all other processes in the DGRP. Our implementation employs reliable broadcast to disseminate newly detected events. A tester waits until all its tests are executed to report all events by reliably broadcasting a single message with all information. Using the received information, recommended processes update the DGRP membership. Note that information received from processes $\notin$ DGRP is ignored.

Note that this diagnosis model allows two different recommended (fault-free) processes (for example i and node j) to test a given process (for example, k) in the same round and obtain different results. In this case, as diagnosis information reaches both nodes, the tested node (k) is removed from the DGRP before the next computing round.

A non-recommended process can rejoin the DGRP if it is tested as recommended for a sequence of ζ testing rounds, and after these rounds, the DGRP processes execute consensus to agree on the recommendation of the process. In our implementation, we used Paxos [51] as the consensus algorithm.

DGRP implementation and case study

In this section, we describe a DGRP implementation. As a case study, we implemented the parallel sorting algorithm HyperQuickSort on top of DGRP.

DGRP implementation

DGRP was implemented using MPI. Each MPI process i belongs to a single MPI communicator and has a unique identifier, called the *rank*. The communicator is the data structure that defines the communication context and the set of processes that belongs to this context. Processes exchange messages using the point-to-point communication primitives `primitives MPI_Send() and MPI_Recv()`, which in turn use the network transport (e.g., TCP).

Each process i maintains in the local array $syndrome_i[\,]$ information about the test outcomes of all processes. An entry $syndrome_i[j]$ returns the counter of events for process j. Initially, all counters are set to 0 assuming the processes are recommended; after a test is executed, if the tester detects that an event has occurred on the tested node, the corresponding counter is incremented. Note that if $syndrome_i[j]$ is even, then j is considered to be recommended by i; otherwise, it is considered to be non-recommended. Process i may obtain the syndrome from other processes. If $syndrome_j[k] > syndrome_i[k]$ then process i updates its local entry.

A tester uses its local clock to measure the total time it takes to receive the corresponding reply. A *recommendation threshold* is adaptively computed based on how long the tested node takes to reply. A node is considered to be non-recommended if the test reply takes more than the threshold to arrive. A timeout is also employed and is also computed for each tested node adaptively. Timeouts are important so that a tester does not wait indefinitely for a reply. Note that MPI by itself does not recognize network faults, for instance if the link between two processes is broken and one process is waiting for a message from the other process it will keep waiting indefinitely.

In our implementation, both the recommendation threshold and the timeout are computed using a similar version of TCP algorithm [52–54], with an added factor to increase the delay tolerance, as described next.

In Algorithm 1, *start_time* and *arrival_time* are the local clock times at which a test is started and the corresponding reply arrives at the tester, respectively. Variable *test_time* corresponds to the time a tester takes to receive a reply. Variable *mean* maintains the weighted mean of the time to receive replies. The dispersion of the time to receive a reply is kept in *variance* and is also computed as a weighted mean. The higher the *variance*, the larger the *timeout*. Constants α and β were assigned 0.9 and 4, respectively, which are frequently used as approximations of the original values proposed by Jacobson [52]. Variable *threshold* is computed by adding the *timeout* to a factor defined by the user. This factor is frequently added to avoid false positives, i.e., incorrect suspicions.

Algorithm 1 Timeout and threshold (for each process p testing q)

1: Initialization
2: $test\_time \leftarrow 0$ *{time to complete a test}*
3: $mean \leftarrow 0$ *{weighted mean}*
4: $variance \leftarrow 0$
5: $timeout \leftarrow 0$
6: $threshold \leftarrow 0$
7: Begin
8: $test\_time \leftarrow arrival\_time - start\_time$
9: $mean \leftarrow (\alpha * mean) + (1 - \alpha) * test\_time$
10: $variance \leftarrow \alpha * variance + (1 - \alpha) * |mean - test\_time|$
11: $timeout \leftarrow mean + \beta * variance$
12: $threshold \leftarrow timeout +$ getFactor()
End

The implementation was based on a fully connected test assignment, i.e., each node tests all others. To be considered recommended, the tested process must reply correctly and within the time interval defined by the recommendation threshold. At the end of a testing interval, if a tester has detected events in which at least one process is considered to be non-recommended, it reliably broadcasts the information to the other processes.

If a process not in the DGRP (i.e., non-recommended) is tested as recommended for ζ consecutive testing intervals, the DGRP processes run a consensus round to possibly reincorporate the process to the DGRP. Note that consensus is executed *within* the DGRP. We claim that the DGRP corresponds to a wormhole that preserves the timing properties required to run consensus effectively. The leader is the DGRP process with the smallest *rank*. Initially, the leader receives a request from a DGRP process that triggers the execution of consensus, sending the proposal to the other processes in the DGRP. If extra requests are received by the same process, they are ignored. A participant agrees with the proposal if it has tested the process as recommended for more than $\zeta/2$ consecutive testing rounds. The leader waits for replies, and if the majority of the processes currently in the DGRP agrees, the decision is sent.

HyperQuickSort algorithm

HyperQuickSort is a parallel sorting algorithm that employs a hypercube as a logical topology representing the communication among the processes [17]. In our implementation, HyperQuickSort adapts itself and continues its execution even if up to $N - 1$ processes become non-recommended/crash; N is the total number of processes running the algorithm. Figure 5 shows the DGRP architecture with HyperQuickSort as the application.

HyperQuickSort executes in sorting rounds which correspond to the computing rounds in the DGRP execution model. We use the `MPI_Barrier` to implement the barrier described in our model which separates a computing round from a diagnosis round.

The sorting problem on the hypercube consists in using a set P of processes, $P = \{b_0, b_1, \ldots, b_{p-1}\}$,

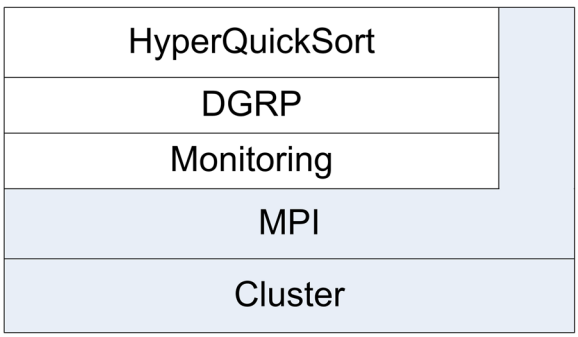

Fig. 5 Proposed system architecture

where $|P|$ is a power of 2 to sort a list K of numbers $K = \{a_0, a_1, \ldots, a_{k-1}\}$. Initially, the $|K|$ numbers are divided equally between the $|P|$ processes. Each process is responsible for sorting a list of $\frac{|K|}{|P|}$ numbers. Sorting is executed in rounds, in which each process i exchanges with process j part of its assigned list using a *pivot number*. At the end of $\log_2 P$ sorting rounds, the lists are sorted so that in each process b_i, the largest number is less than or equal to the lowest number on process $b_i + 1, 0 \leq i \leq |P| - 2$. Algorithm 2 shows HyperQuickSort's pseudocode.

Algorithm 2 ***HyperQuickSort*** (for each process p running in parallel)

```
1:  Initialization
2:    dim ← log2 |P|                          {Hypercube dimension}
3:    rank ← process_id                       {Each process has a unique id between
      0..2^dim − 1}
4:    list ← K                                {Original list number at each process}
5:    n ← |K|                                 {List size at each process}
6:  Begin
7:    quicksort(list, n)
8:    while dim > 0 do
9:      cluster_i ← processes(rank, dim)
10:     root_process ← root(rank, dim)
11:     if rank == root_process then
12:       pivot ← median(list)
13:     broadcast(root_process, pivot, cluster_i)
14:     create_lists(higher_list, lower_list, list, pivot)
15:     partner ← rank ⊕ 2^(dim−1)
16:     if rank > partner then
17:       send(lower_list, partner)
18:       receive(new_higher_list, partner)
19:       list ← union(higher_list, new_higher_list)
20:     else if rank < parter then
21:       send(higher_list, partner)
22:       receive(new_lower_list, partner)
23:       list ← union(lower_list, new_lower_list)
24:     dim ← dim − 1
25:     quicksort(list, n)
    End
```

Initially, each process locally orders its list (line 7). The processes are organized in virtual clusters of sizes that decrease by a power of 2 at each sorting round (line 9). Figure 6 shows the cluster sizes for a 3-dimensional hypercube. In the first round, there are eight processes in a single cluster. In the second round, there are two clusters with four processes each. In the last round, there are four clusters that group two processes each. For each *dim* round, the algorithm runs the following steps.

Clusters are formed in the respective sorting round (line 9). In a sorting round, the process with the lowest rank in each cluster is defined as the *root process* (line 10). The root process distributes its pivot (number) to the other processes in its cluster (lines 11–13). The pivot is the medium element of the list (line 12). After receiving the pivot, each process divides its list into two lists: a list of numbers less than the pivot and another list of numbers greater than the pivot (line 14). Then, each process finds a partner within the cluster using an exclusive or operation on its rank (line 15). The lowest rank process sends the list with numbers greater than the pivot to the partner with a higher rank and receives the list of numbers less than the pivot from the partner (16–23). After this exchange of lists, each process merges the received list with its list number that was not exchanged (line 19 and 23). Then, each process sorts its new list (line 25).

Figure 6 shows an example execution of HyperQuickSort for 8 processes. It takes 3 rounds to complete sorting these numbers. In the first round, there is 1 cluster with 8 processes, and process 0 is the root process. The following pairs of processes are established and exchange their lists according to the pivot received from process 0: (0, 4), (1, 5), (2, 6), and (3, 7). All processes rearrange their subsets of numbers to sort. In the second sorting round, there are two clusters each with 4 processes. Processes 0 and 4 are the pivots of their respective clusters. Each root process sends its pivot number to the other processes of its cluster. So, the lists are exchanged between the process pairs in sorting round 2. Finally, in the third round, the whole process is repeated considering clusters and process clusters in accordance with sorting round 3. The execution of HyperQuickSort using a DGRP is described next.

HyperQuickSort using DGRP

At the beginning of each sorting round, each process has a list containing the non-recommended processes. Given the DGRP composition (some processes are recommended, others non-recommended, and only recommended processes run the algorithm), a mapping function was implemented to define the partners p_i and p_j in a sorting round. In this mapping, each recommended process becomes responsible for sorting lists of numbers that were supposed to be sorted by non-recommended processes. A process p_i can handle up to $n - 1$ non-recommended processes (in case only one process remains recommended). Process p_i, besides performing its tasks according the algorithm, also performs the task of the non-recommended process. A process p_j also needs to know process p_i, which assumes the tasks of a non-recommended process. To implement these tasks, we employed the $C_{i,s}$ function defined in [49]. The $C_{i,s}$

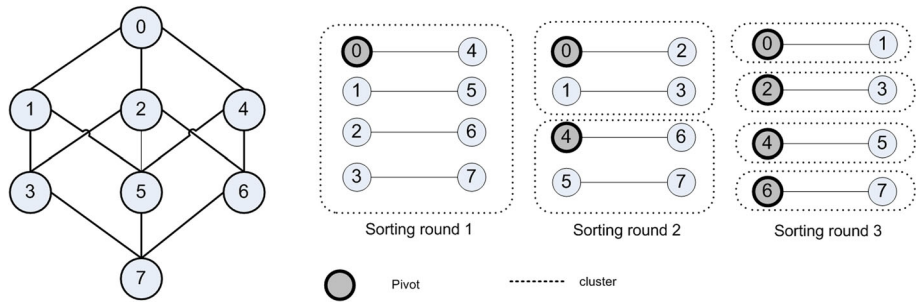

Fig. 6 Parallel sorting with HyperQuickSort

function helps determining the clusters to which process i belongs in sorting round s considering only recommended processes. $C_{i,s}$ is a function executed by node i that returns the sequence of nodes in its s-*th* cluster cluster. For instance, this function executed by node 0 returns node 1 (the node to be tested by node 0) when s is equal to 1.

$$c_{i,s} = \left(i \oplus 2^{s-1}, c_{i\oplus 2^{s-1},1}, \ldots, c_{i\oplus 2^{s-1},s-1}\right)$$

The $c_{i,s}$ function considers that the processes are logically organized in a hypercube: i represents a process p_i and s is related to a particular sorting round. Initially, $s = dim$. The symbol $\oplus$ represents the binary operation XOR. Table 2 shows an example of $c_{i,s}$ applied to a 3-dimensional hypercube, i.e., a hypercube with 8 processes. For example, the process p_0 for $s = 3$ has the following result: 4, 5, 6, and 7. That means that process p_0 and p_4 are partners in sorting round 3 whether both are recommended.

The partners in each sorting round are established according to Algorithm 3. As the algorithm shows, the partner of a process p_i in sorting round s is the first recommended process in $c_{i,s}$. Therefore, for the first sorting round, process p_0 must exchange its list number with the first recommended process in $c_{0,3}$. If p_4 is recommended, then p_0 and p_4 are partners and exchange its list number according to lines 16–23 in Algorithm 2 (assuming p_0 also is recommended). However, if p_4 is non-recommended, then p_0 must exchange its number list with process p_5. But, if both p_5 and p_6 are non-recommended, then p_0 does not exchange its list with any process in the corresponding sorting round. Each recommended process performs a local checkpoint on a shared file upon finishing its task in a sorting round.

Table 2 $C_{i,s}$ for a system with 8 processes

s	$C_{0,s}$	$C_{1,s}$	$C_{2,s}$	$C_{3,s}$	$C_{4,s}$	$C_{5,s}$	$C_{6,s}$	$C_{7,s}$
1	1	0	3	2	5	4	7	6
2	2 3	3 2	0 1	1 0	6 7	7 6	4 5	5 4
3	4 5 6 7	5 4 7 6	6 7 4 5	7 6 5 4	0 1 2 3	1 0 3 2	2 3 0 1	3 2 1 0

Algorithm 3 Function to find a partner in a cluster *dim*

```
1: Function partner(rank, round)
2: nodes ← c_{rank,round}
3: j ← 0
4: while j ≤ size(nodes) do
5:     if nodes[j] ∉ non-recommended then
6:         return nodes[j]
7:     j ← j + 1
8: return ⊥
```

Algorithm 4 Function to replace a partner in a cluster *dim*

```
1: function replace(rankNon-recommended, dim)
2: s ← 1
3: while s ≤ dim do
4:     j ← 0
5:     nodes ← c_{rankNon-recommended,s}
6:     while j ≤ size(nodes) do
7:         if nodes[j] ∉ non-recommended then
8:             return nodes[j]
9:         j ← j + 1
10:    s ← s + 1
11: return ⊥
```

The process which replaces a non-recommended process in a corresponding sorting round is chosen according to Algorithm 4. A non-recommended process p_i is replaced by the first recommended process in $c_{i,s}$, starting from $s = 1$ and i is the identifier of a non-recommended process. If there is no recommended process in cluster s, then s is incremented until a recommended process is found. For example, considering the first sorting round and processes p_0 and p_4, if process p_4 is non-recommended, then p_5 assumes p_4 tasks in the corresponding sorting round ($c_{4,1} = 5$, see Table 2). However,

if p_5 is also non-recommended, then p_4 is replaced by p_6 because p_6 is the first recommended process in $c_{4,2}$.

Another example is given next. Suppose process 2 is non-recommended in the first sorting round. This round corresponds to the largest cluster ($s = 3$), as shown in Fig. 6. In this sorting round, processes p_2 and p_6 are supposed to form a pair whether both are recommended. However, the first recommended process in $c_{6,3}$ is process p_3, i.e, $c_{6,3} = 3$. Process p_3 is responsible for process p_2 because it is the first recommended process in $c_{2,1}$. Then, process p_3 is then assigned the task of process p_2. Process p_3 then reads the list number of process p_2 and communicates with process p_6 replacing process 2. Remember that process p_3 also performs its task with process p_7 because both p_3 and p_7 are recommended in that sorting round. At the end of each sorting round, each process saves its sorted list on a shared file system. After completing their tasks in a sorting round, each process reaches a barrier.

HyperQuickSort using ULFM

Next, we briefly give an overview of how we implemented HyperQuickSort using ULFM. A function similar to `MPI_Barrier` is used to detect process faults. Whenever this function returns an error code (`MPI_ERR_PROC_FAILED`), the MPI communicator is revoked (`MPI_Comm_revoke()`). This function makes the MPI communicator invalid. After this, all processes call an agreement function (`MPI_Agree()`). `MPI_Comm_agree` executes a collective operation between the correct processes of the communicator. This function notifies processes that the communicator is invalid. The routines `MPI_Comm_failure_ack()` and `MPI_Comm_failure_get_acked()` are used to identify which processes within a communicator are faulty. After that, the `MPI_Comm_shrink()` routine allows the application to create a new communicator, eliminating all the failed processes. This primitive is collective and executes consensus to ensure that all processes have the same vision of the new communicator. A consensus algorithm implemented in ULFM is the one proposed by Herault et al. [26].

Results

In this section, we present experimental results obtained from running the DGRP implementation described in the previous section. The experiments were executed on the LCPAD-UFPR cluster. This is a multi-user shared cluster that runs applications of several users simultaneously. The cluster consists of 18 machines each of which with 32 Intel(R) Xeon(R) CPU E5-2670 cores at 2.60 GHz with 128 GB of RAM and 20480 KB of cache, interconnected by a Gigabit Ethernet network. The code was written in the MPI/C language. We employed the Open MPI library version 1.7 extended with ULFM[2].

Two sets of experiments are presented. The first set consists of results for DGRP itself, including the performance of monitoring. The second set consists of results obtained for the execution of the MPI implementation of HyperQuickSort using DGRP and using ULFM.

DGRP: monitoring

In this subsection, we present results for the DGRP implementation with a focus on monitoring. Each machine executes a single process; hosts/processes are assigned identifiers from 0 to 17. The testing interval was set to 1 s. The test procedure consists of the computation of the prime numbers between 1 and 1000. As mentioned above, the recommendation threshold is computed using the TCP timeout algorithm multiplied by a constant; initially, it was set to 8 s. As soon as the tests are executed and the actual delays to receive test replies are measured, the threshold gets closer to the mean time testers take to receive a reply. This takes a couple of testing rounds. A process is classified as recommended if it correctly sends the test reply within the threshold interval. Otherwise, it is classified as non-recommended. For all experiments, ζ (the number of times a process must be tested as recommended before it can rejoin the DGRP) was set to 5 ($\zeta = 5$).

Figure 7 shows the results for process 1 testing process 0 for 150 consecutive testing intervals. In this experiment, we employed a recommendation threshold that is very close to the time to execute a test. The continuous black curve shows the time to a complete test. The dashed red curve shows the corresponding recommendation threshold. It is possible to conclude from the several peaks in

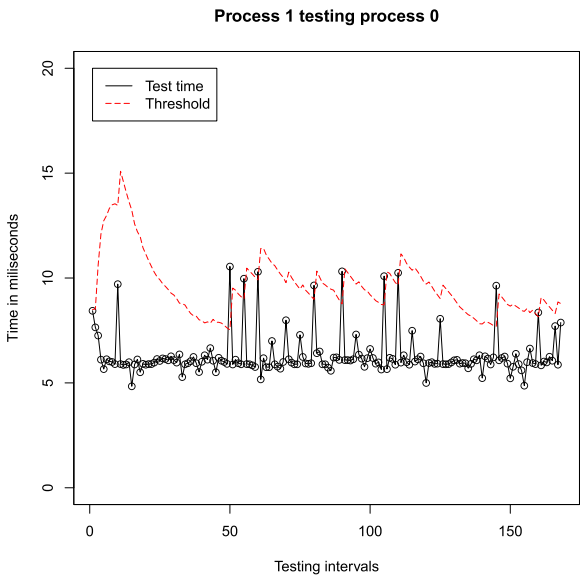

Fig. 7 Process 1 tests process 0

time to test curve that the process 0 very frequently takes longer to reply than expected. The curves also show that the recommendation threshold is updated according to the variation of the time to test. Despite of this adaptation, the time to test does exceed the threshold on some points, and when this occurs, process 0 ends up being classified as non-recommended by process 1.

Next, we present results obtained from monitoring the 18-node cluster for 30,000 consecutive testing intervals. For this and the next experiments, the threshold was set to two and a half times the value computed using TCP's timeout algorithm. In this experiment, process 11 remained recommended during all testing intervals. Process 2 in the end was classified as recommended, but it was the process that most often became non-recommended, alternating its status 11 times. Figure 8 shows the point of view of process 11 (tester) about process 2 (tested). Figure 9 shows the point of view of process 6 (tester) about the process 2 (tested). Process 6 tested process 2 as non-recommended seven times. On the other hand, process 11 did not test process 2 as non-recommended even once.

Figure 10 shows the point of view of process 0 (tester) about process 11 (tested). Comparing results in Figs. 8 and 9 with those in Fig. 10, it is possible to see that in the first two figures, process 2 presents a significant variation of the time to send test replies. In Fig. 10, process 11 presents little variation, a more stable behavior.

From the experiments, we learned that in the cluster, the hosts with lower identifiers had almost always most of their cores busy running jobs. Table 3 below correlates the cluster load with the state of each process at the end of the experiment. Each host has 32 cores. It is possible to see that the host in which process 2 runs ($host_2$) has all its cores running jobs. This may explain why process 2 presented the behavior described above. We could conclude that processes that became non-recommended fewer times were running on hosts with fewer jobs.

Figure 11 is a zoom on Fig. 9 and shows process 6 testing process 2 from testing interval 13,200 to 13,700. It is possible to see that process 6 detects twice the instability of process 2 (blue circles in Fig. 11). In testing round 13,226, the test delay was 62.39 ms and the recommendation

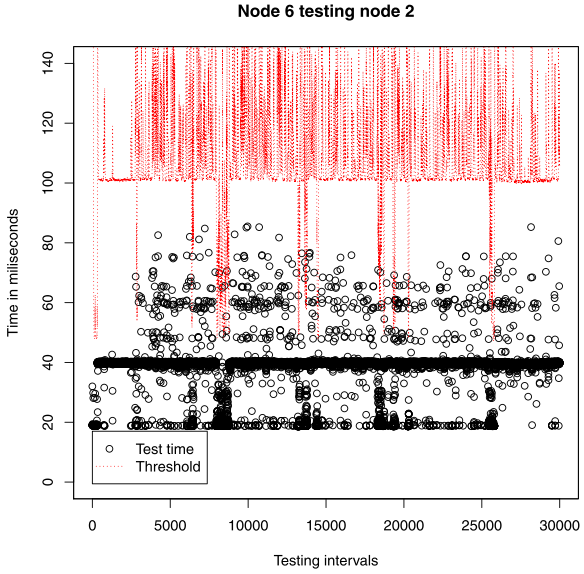

Fig. 9 Process 6 tests process 2

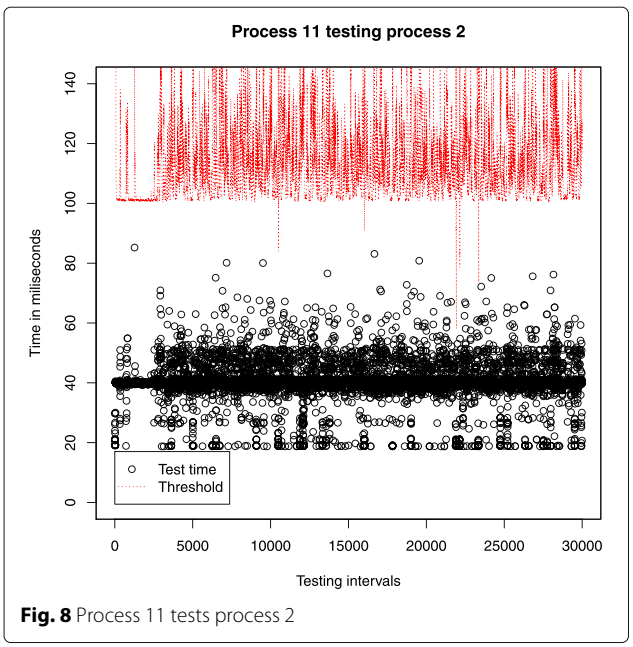

Fig. 8 Process 11 tests process 2

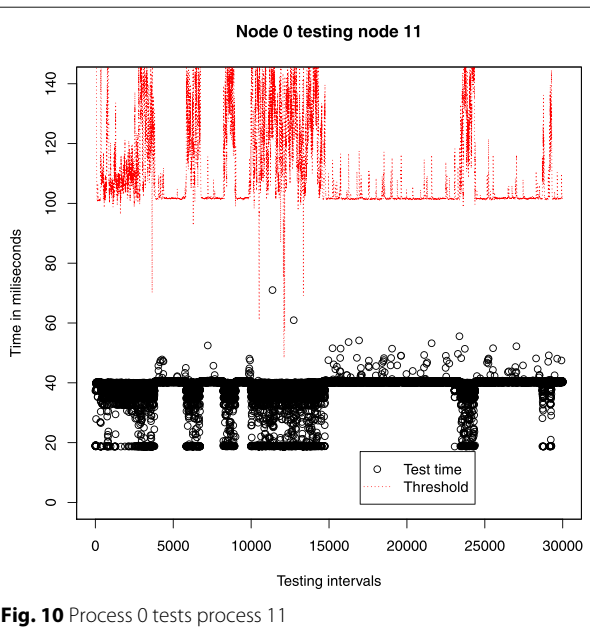

Fig. 10 Process 0 tests process 11

Table 3 Cluster load and test results

host	No. of jobs running	*state*
$host_0$	9	6
$host_1$	33	10
$host_2$	33	22
$host_3$	32	14
$host_4$	32	12
$host_5$	32	12
$host_6$	3	2
$host_7$	2	4
$host_8$	3	6
$host_9$	4	6
$host_{10}$	8	6
$host_{11}$	5	0
$host_{12}$	32	6
$host_{13}$	2	2
$host_{14}$	1	2
$host_{15}$	31	12
$host_{16}$	31	8
$host_{17}$	31	14

threshold was equal 48.42 ms. In testing round 13,641, the test delay was 68.16 ms and recommendation threshold was equal to 62.1 ms.

Every time, non-recommended process 2 is tested as recommended process 6 incremented the corresponding counter. Figure 12 shows the variation of this counter. When the counter reaches the target constant $\zeta = 5$, a consensus execution is triggered to possibly allow process 2 to rejoin the DGRP. Process 6 invoked Paxos nine times in order to change the state of process 2 back to recommended. Twice, process 6 voted favorably to reintegrate process 2 to the DGRP. As previously mentioned, if a participant has the counter greater than $\zeta/2$ then its vote is favorable. From testing interval 5000 to 8500, process 2 changed its state five times.

Figure 13 shows how the DGRP composition varied along the 30,000 testing intervals. A zoom showing the DGRP composition between testing intervals 1200 and 1300 is also shown. At the beginning, all 18 processes are in the DGRP. Several times, one process was removed from DGRP. A greater instability can be perceived in

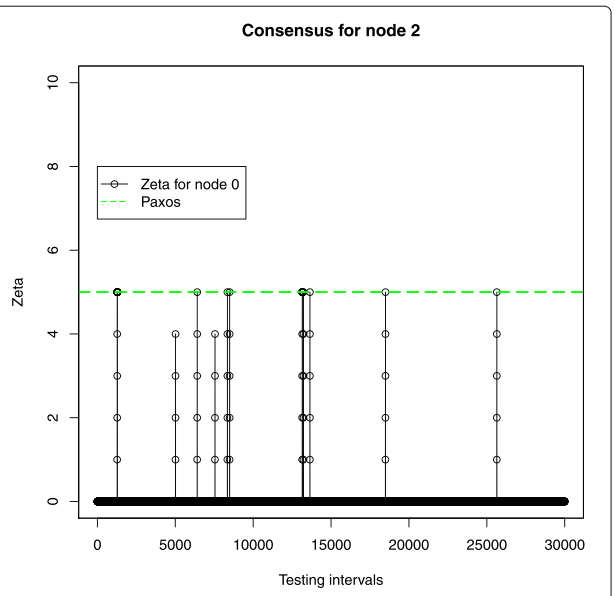

Fig. 12 Zeta for node 2

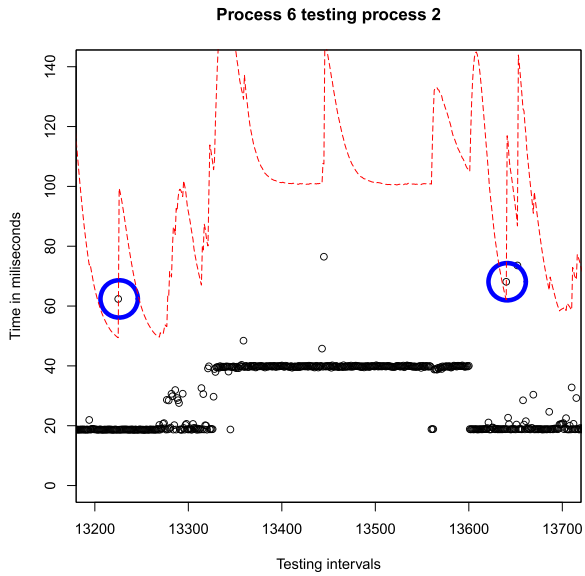

Fig. 11 Process 6 tests process 2

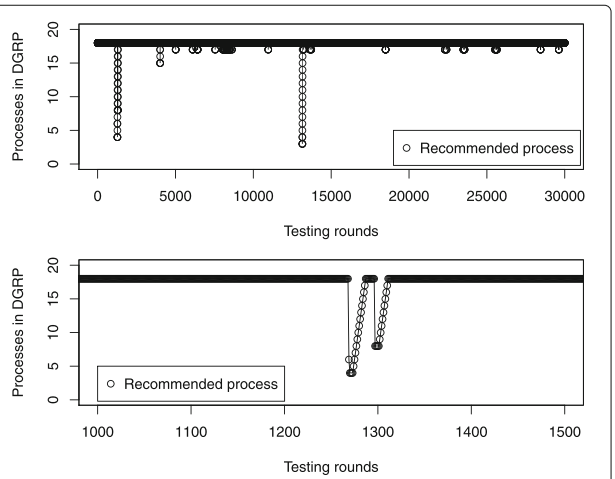

Fig. 13 DGRP composition variation

testing round 1271, when DGRP consists of only four processes. Soon after that, the recommended group recovers. In testing round 1.298, the DGRP is formed by eight processes. A similar situation happens again in testing round 13,149, when DGRP consists of only three processes.

Performance of HyperQuickSort over DGRP and ULFM

The performance of HyperQuickSort on top of DGRP was evaluated both considering that non-recommended processes can later rejoin the DGRP and have tasks assigned and also considering that the non-recommended processes are removed forever from the DGRP. For measuring the overhead of DGRP, we also evaluated the performance of HyperQuickSort using ULFM. In this implementation, ULFM excludes faulty processes allowing the computation to proceed but they are never used again, i.e., in this case, new processes are not launched to replace faulty ones. HyperQuickSort was executed to sort 1 billion integers.

Figure 14 shows the performance of HyperQuickSort running on 16 processes. The 95% confidence interval is shown for results. The following scenarios were executed: (1) only recommended processes, (2) only one process becomes non-recommended, (3) half of the processes become non-recommended, and (4) only one process remains recommended. Instability was injected randomly during the algorithm execution; it was implemented by killing our own application processes or making them take longer to reply. As shown in Fig. 14, the darker bar presents the performance of ULFM. The bar in the middle shows results for the implementation on DGRP that eliminates non-recommended processes forever, i.e., with no recovery. The lighter bar shows the performance of DGRP with non-recommended processes later rejoining the DGRP.

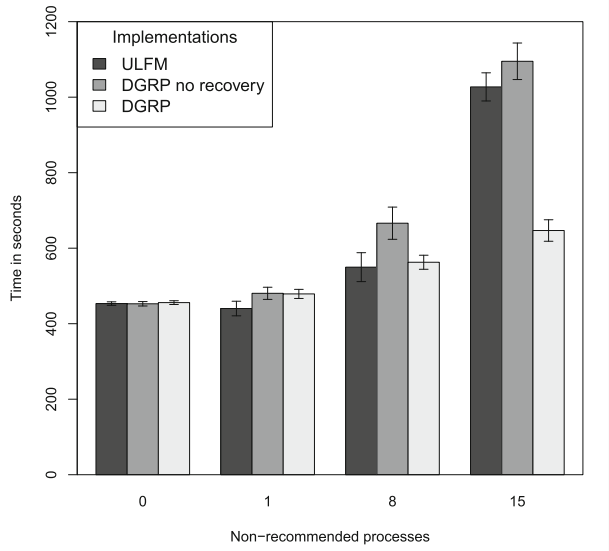

Fig. 14 Results for 16 processes sorting 1.024×10^6 integer numbers

The overhead of DGRP is around 9% when we compare HyperQuickSort using ULFM with DGRP with no recovery. On the other hand, in the last two scenarios, the execution time of HyperQuickSort when processes can later rejoin the DGRP is lower than the performance of DGRP with no recovery. This is because even if a process does not participate in a sorting a round, it can be reintegrated in a subsequently sorting round.

Conclusion

In this work, we introduced a new approach for recommending a group of processors to run applications in MPI-based HPC systems. The recommendation is based on tests executed on processes that eliminates both those that have crashed and those that are presenting slow responsiveness. A DGRP is defined as a self-managed dynamic group of recommended processes that employs monitoring and reconfiguration at runtime, allowing the application to keep running even as processes become non-recommended and are removed from the DGRP. A non-recommended process can later rejoin the DGRP if it passes a sequence of tests and after a round of consensus executed by DGRP processes. The application execution is organized in computing rounds that use barriers to allow processes to synchronize, obtaining a fresh view of the DGRP so that the application jobs can be properly assigned to DGRP processes. Actually, the barrier can be easily removed, but the removal would make it more complex to describe the proposal. As they are used, they make it very easy for processes to obtain the same DGRP view. In order to remove the barriers, we need to create a callback system, i.e., as soon as an application process gets a different view than expected, it needs to tell the others about the reconfiguration. As a case study, we implemented HyperQuickSort and report results from executions on a shared HPC cluster. DGRP is implemented on top of ULFM, a recent specification for dealing with faults in MPI systems. Results show that the proposed model is efficient and effective.

Several different aspects of running resilient MPI applications on a DGRP can be explored as future work. First of all, we believe the model itself can be extended so that the DGRP can use more information from the diagnosis system. For example, the fact that some processes eventually present much higher state counters can help classify a process as stable or not. On another front, we believe that the definition of MPI primitives to allow tasks to be mapped to DGRP processes can make it straightforward to allow arbitrary parallel applications to use the DGRP, in particular we believe it is possible to extend ULFM to include the DGRP functionality. Running other parallel algorithms, using other diagnosis strategies besides those

presented in the paper and exploring other strategies to maintain DGRP should also be done in the future. As mentioned above, the barrier can also be removed from the DGRP specification and replaced by a callback system, i.e., as soon as an application process gets a different view than expected, it needs to tell the others about the reconfiguration. Future work also includes investigating the applicability of DGRP to Spark [55] and Hadoop [56] which have become increasingly important frameworks for parallel and distributed computing.

Endnotes

[1] http://ipm-hpc.sourceforge.net

[2] http://fault-tolerance.org/ulfm/downloads/

Acknowledgments
We would like to thank the funding agencies and universities involved for the support provided. We also thank the many contributions from the reviewers.

Funding
This work was partially supported by grant 311451/2016-0 from the Brazilian Research Agency (CNPq) and by Conselho Nacional de Desenvolvimento Científico e Tecnológico, award number: 311451/2016-0, recipient: Elias Procópio Duarte Jr., Ph.D.

Availability of data and materials
Not applicable.

Authors' contributions
Both authors contributed equally to this work. Both authors read and approved the final manuscript.

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Author details

Received: 27 May 2017 Accepted: 31 January 2018
Published online: 12 March 2018

References

1. Egwutuoha IP, Levy D, Selic B, Chen S (2013) A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. J Supercomput 65(3):1302–1326
2. Martino CD, Kalbarczyk Z, Iyer RK, Baccanico F, Fullop J, Kramer W (2014) Lessons learned from the analysis of system failures at petascale: The case of blue waters. In: 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. pp 610–621. https://doi.org/10.1109/DSN.2014.62
3. Snir M, Wisniewski RW, Abraham JA, Adve SV, Bagchi S, Balaji P, Belak J, Bose P, Cappello F, Carlson B, Chien AA, Coteus P, Debardeleben NA, Diniz PC, Engelmann C, Erez M, Fazzari S, Geist A, Gupta R, Johnson F, Krishnamoorthy S, Leyffer S, Liberty D, Mitra S, Munson T, Schreiber R, Stearley J, Hensbergen EV (2014) Addressing failures in exascale computing. Int J High Perform Comput Appl 28(2):129–173. https://doi.org/10.1177/1094342014522573
4. Tiwari D, Gupta S, Vazhkudai SS (2014) Lazy checkpointing: exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems. In: 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. pp 25–36. https://doi.org/10.1109/DSN.2014.101
5. Gioiosa R, Kestor G, Kerbyson DJ (2014) Online monitoring system for performance fault detection. In: International Parallel Distributed Processing Symposium Workshops. pp 1475–1484. https://doi.org/10.1109/IPDPSW.2014.165
6. Nielsen F (2016) Introduction to HPC with MPI for data science. 1st edn. Springer, Switzerland
7. Fagg GE, Dongarra J (2000) Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In: Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer, London. pp 346–353. http://dl.acm.org/citation.cfm?id=648137.746632
8. (2015) MPI Forum: document for a standard message-passing interface 3.1. Technical report, University of Tennessee
9. Bland W, Bouteiller A, Hérault T, Bosilca G, Dongarra J (2013) Post-failure recovery of MPI communication capability: design and rationale. IJHPCA 27(3):244–254
10. Gropp W, Lusk E (2004) Fault tolerance in message passing interface programs. Int J High Perform Comput Appl 18(3):363–372. https://doi.org/10.1177/1094342004046045
11. Birman K (2010) Replication. In: A history of the virtual synchrony replication model. Springer, Berlin. pp 91–120. http://dl.acm.org/citation.cfm?id=2172338.2172344
12. Veríssimo PE (2006) Travelling through wormholes: a new look at distributed systems models. SIGACT News 37(1):66–81. https://doi.org/10.1145/1122480.1122497
13. Huang KC, Huang TC, Tsai MJ, Chang HY (2014) Moldable job scheduling for HPC as a service. In: Park JJJH, Stojmenovic I, Choi M, Xhafa F (eds). Future information technology: FutureTech 2013. Springer, Berlin, Heidelberg. pp 43–48. https://doi.org/10.1007/978-3-642-40861-8_7
14. Masson GM, Blough DM, Sullivan GF (1996) Fault-tolerant computer system design. In: System diagnosis. Prentice-Hall, Inc, Upper Saddle River. pp 478–536
15. Ye TL, Hsieh SY (2013) A scalable comparison-based diagnosis algorithm for hypercube-like networks. IEEE Trans Reliab 62(4):789–799. https://doi.org/10.1109/TR.2013.2284743
16. Weber A, Kutzke AR, Chessa S (2012) Energy-aware test connection assignment for the self-diagnosis of a wireless sensor network. J Braz Comput Soc 18(1):19–27. https://doi.org/10.1007/s13173-012-0057-7
17. Wagar B (1987) Hyperquicksort: A fast sorting algorithm for hypercubes. Hypercube Multiprocessors 1987:292–299
18. Cappello F, Geist A, Gropp W, Kale S, Kramer B, Snir M (2014) Toward Exascale Resilience: 2014 update. Supercomputing Frontiers and Innovations 1(1). http://superfri.org/superfri/article/view/14
19. Ropars T, Martsinkevich TV, Guermouche A, Schiper A, Cappello F (2013) Spbc: Leveraging the characteristics of mpi hpc applications for scalable checkpointing. In: International Conference for High Performance Computing, Networking, Storage and Analysis. pp 1–12. https://doi.org/10.1145/2503210.2503271
20. Bouteiller A, Herault T, Bosilca G, Dongarra JJ (2013) Correlated set coordination in fault tolerant message logging protocols for many-core clusters. Concurr Comput Pract Exp 25(4):572–585. https://doi.org/10.1002/cpe.2859
21. Fagg GE, Dongarra JJ (2004) Building and using a fault-tolerant mpi implementation. Int J High Perform Comput Appl 18(3):353–361. https://doi.org/10.1177/1094342004046052
22. Batchu R, Dandass YS, Skjellum A, Beddhu M (2004) Mpi/ft: A model-based approach to low-overhead fault tolerant message-passing middleware. Clust Comput 7(4):303–315. https://doi.org/10.1023/B:CLUS.0000039491.64560.8a
23. Suo G, Lu Y, Liao X, Xie M, Cao H (2013) Nr-mpi: A non-stop and fault resilient mpi. In: International Conference on Parallel and Distributed Systems. pp 190–199. https://doi.org/10.1109/ICPADS.2013.37
24. Gropp W, Lusk E, Doss N, Skjellum A (1996) A high-performance, portable implementation of the mpi message passing interface standard. Parallel Comput 22(6):789–828. https://doi.org/10.1016/0167-8191(96)00024-5

25. Hursey J, Graham RL, Bronevetsky G, Buntinas D, Pritchard H, Solt DG (2011) Run-through stabilization: an MPI proposal for process fault tolerance. In: Cotronis Y, Danalis A, Nikolopoulos DS, Dongarra J (eds). Recent advances in the message passing interface: 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings. Springer, Berlin, Heidelberg. pp 329–332. https://doi.org/10.1007/978-3-642-24449-0_40
26. Herault T, Bouteiller A, Bosilca G, Gamell M, Teranishi K, Parashar M, Dongarra J (2015) Practical scalable consensus for pseudo-synchronous distributed systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC. ACM, New York. pp 31–13112. http://doi.acm.org/10.1145/2807591.2807665. https://doi.org/10.1145/2807591.2807665
27. Buntinas D (2012) Scalable distributed consensus to support mpi fault tolerance. In: 26th International Parallel and Distributed Processing Symposium. pp 1240–1249. https://doi.org/10.1109/IPDPS.2012.113
28. Hursey J, Naughton T, Vallee G, Graham RL (2011) A log-scaling fault tolerant agreement algorithm for a fault tolerant MPI. In: Cotronis Y, Danalis A, Nikolopoulos DS, Dongarra J (eds). Recent Advances in the Message Passing Interface: 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings. Springer, Berlin, Heidelberg. pp 255–263. https://doi.org/10.1007/978-3-642-24449-0_29
29. Huang KH, Abraham JA (1984) Algorithm-based fault tolerance for matrix operations. IEEE Trans Comput C-33(6):518–528. https://doi.org/10.1109/TC.1984.1676475
30. Chen Z, Dongarra J (2008) Algorithm-based fault tolerance for fail-stop failures. IEEE Trans Parallel Distrib Syst 19(12):1628–1641. https://doi.org/10.1109/TPDS.2008.58
31. Gamell M, Katz DS, Kolla H, Chen J, Klasky S, Parashar M (2014) Exploring automatic, online failure recovery for scientific applications at extreme scales. In: SC14: International Conference for High Performance Computing, Networking, Storage and Analysis. pp 895–906. https://doi.org/10.1109/SC.2014.78
32. Gamell M, Teranishi K, Heroux MA, Mayo J, Kolla H, Chen J, Parashar M (2015) Local recovery and failure masking for stencil-based applications at extreme scales. In: SC15: International Conference for High Performance Computing, Networking, Storage and Analysis. pp 1–12. https://doi.org/10.1145/2807591.2807672
33. Zheng G, Ni X, Kalé LV (2012) A scalable double in-memory checkpoint and restart scheme towards exascale. In: International Conference on Dependable Systems and Networks Workshops (DSN). pp 1–6. https://doi.org/10.1109/DSNW.2012.6264677
34. Ferreira K, Stearley J, Laros III JH, Oldfield R, Pedretti K, Brightwell R, Riesen R, Bridges PG, Arnold D (2011) Evaluating the viability of process replication reliability for exascale systems. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, SC. ACM, New York. pp 44–14412. http://doi.acm.org/10.1145/2063384.2063443. https://doi.org/10.1145/2063384.2063443
35. Genaud S, Jeannot E, Rattanapoka C (2009) Fault-management in p2p-mpi. Int J Parallel Prog 37(5):433–461. https://doi.org/10.1007/s10766-009-0115-8
36. Fiala D, Mueller F, Engelmann C, Riesen R, Ferreira K, Brightwell R (2012) Detection and correction of silent data corruption for large-scale high-performance computing. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC. IEEE Computer Society Press, Los Alamitos. pp 78–17812. http://dl.acm.org/citation.cfm?id=2388996.2389102
37. Huang C, Zheng G, Kal'e L, Kumar S (2006) Performance Evaluation of Adaptive MPI. In: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, New York. pp 12–21. http://doi.acm.org/10.1145/1122971.1122976. http://doi.org/10.1145/1122971.1122976
38. Kale LV, Krishnan S (1993) CHARM++: A Portable Concurrent Object Oriented System Based on C++. In: Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications. ACM, New York. pp 91–108. http://doi.acm.org/10.1145/165854.165874. http://doi.org/10.1145/165854.165874
39. Petrini F, Kerbyson DJ, Pakin S (2003) The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing. ACM, New York. pp 55–. http://doi.acm.org/10.1145/1048935.1050204. http://doi.org/10.1145/1048935.1050204
40. Aguilar X, Laure E, Fürlinger K (2013) Online performance data introspection with ipm. In: 10th International Conference on High Performance Computing. pp 728–734. https://doi.org/10.1109/HPCC.and.EUC.2013.107
41. Huck KA, Malony AD, Shende S, Morris A (2006) TAUg: runtime global performance data access using MPI. In: Mohr B, Träff JL, Worringen J, Dongarra J (eds). Recent advances in parallel virtual machine and message passing interface: 13th European PVM/MPI User's Group Meeting Bonn, Germany, September 17-20, 2006 Proceedings. Springer, Berlin, Heidelberg. pp 313–321. https://doi.org/10.1007/11846802_44
42. Nataraj A, Sottile M, Morris A, Malony AD, Shende S (2007) TAUoverSupermon: low-overhead online parallel performance monitoring. In: Kermarrec A-M, Bougé L, Priol T (eds). 13th International Euro-Par Conference. Springer, Berlin, Heidelberg. pp 85–96
43. Shende SS, Malony AD (2006) The tau parallel performance system. Int J High Perform Comput Appl 20(2):287–311. https://doi.org/10.1177/1094342006064482
44. Sottile MJ, Minnich RG (2002) Supermon: a high-speed cluster monitoring system. In: International Conference on Cluster Computing. pp 39–46. https://doi.org/10.1109/CLUSTR.2002.1137727
45. Duarte Jr. EP, Ziwich RP, Albini LCP (2011) A survey of comparison-based system-level diagnosis. ACM Comput Surv 43(3):22–12256. https://doi.org/10.1145/1922649.1922659
46. Preparata FP, Metze G, Chien RT (1967) On the connection assignment problem of diagnosable systems. IEEE Trans Electron Comput EC-16(6):848–854. https://doi.org/10.1109/PGEC.1967.264748
47. Hakimi SL, Nakajima K (1984) On adaptive system diagnosis. IEEE Trans Comput 33(3):234–240
48. Hosseini SH, Kuhl JG, Reddy SM (1984) A diagnosis algorithm for distributed computing systems with dynamic failure and repair. IEEE Trans Comput C-33(3):223–233. https://doi.org/10.1109/TC.1984.1676419
49. Duarte EP, Nanya T (1998) A hierarchical adaptive distributed system-level diagnosis algorithm. IEEE Trans Comput 47(1):34–45. https://doi.org/10.1109/12.656078
50. Rangarajan S, Dahbura AT, Ziegler EA (1995) A distributed system-level diagnosis algorithm for arbitrary network topologies. IEEE Trans Comput 44(2):312–334. https://doi.org/10.1109/12.364542
51. Lamport L (2001) Paxos made simple. ACM SIGACT News (Distrib Comput Column) 32, 4 (Whole Number 121, December 2001). pp. 51–58
52. Jacobson V (1988) Congestion avoidance and control. In: Symposium Proceedings on Communications Architectures and Protocols, SIGCOMM. ACM, New York. pp 314–329. http://doi.acm.org/10.1145/52324.52356. https://doi.org/10.1145/52324.52356
53. Paxson V, Allman M, Chu HKJ, Sargent M (2011) Computing TCP's retransmission timer. http://www.rfc-editor.org/rfc/rfc6298.txt
54. Moraes DM, Jr EPD (2011) A failure detection service for internet-based multi-as distributed systems. In: 17th International Conference on Parallel and Distributed Systems. pp 260–267. https://doi.org/10.1109/ICPADS.2011.5
55. Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, Meng X, Rosen J, Venkataraman S, Franklin MJ, Ghodsi A, Gonzalez J, Shenker S, Stoica I (2016) Apache spark: a unified engine for big data processing. Commun ACM 59(11):56–65. https://doi.org/10.1145/2934664
56. Manikandan SG, Ravi S (2014) Big data analysis using apache hadoop. In: International Conference on IT Convergence and Security (ICITCS). pp 1–4. https://doi.org/10.1109/ICITCS.2014.7021746