Building and utilizing fault tolerance support tools for the GASPI applications

Journal Title
XX(X):1–12

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Faisal Shahzad[1], Moritz Kreutzer[1], Thomas Zeiser[1], Rui Machado[2], Andreas Pieper[1,3], Georg Hager[1] and Gerhard Wellein[1]

Abstract

Today's High Performance Computing (HPC) systems are made possible by multiple increase in hardware parallelity. This results in the decrease of mean time to failures (MTTF) of the systems with every newer generation, which is an alarming trend. Therefore, it is not surprising that a lot of research is going on in the area of fault tolerance and fault mitigation. Applications should survive a failure and/or be able to recover with minimal cost.

We have used GASPI, which is a relatively new communication library based on the PGAS model. It fulfills the basic requirement of a fault tolerant communication library , i.e., failure of a process do not cause the remaining processes to fail. This work is focused to extend the fault tolerance features of GASPI in form of a supporting health-check (HC) library that applications can benefit from. These features include failure detection, its information propagation, recovery management, communication recovery, etc. To reinforce its utility, we have also developed a fault tolerant neighbor node-level checkpoint/restart (CR) library.

Instead of introducing algorithm-based fault tolerance in its true sense, we demonstrate how (using these supplementary fault tolerance functions) one can build applications to allow integrate a low cost fault detection/recovery mechanism and, if necessary, recover the application on the fly. We showcase the usage of these tools by implementing them in three different applications. Two of the applications fall in the category of linear sparse solvers, whereas the third application is based on a fluid flow solver. We also analyze the overheads involved in failure-free cases as well as various failure cases. Our fault detection mechanism causes no overhead in failure-free cases, whereas in case of failure(s), the failure detection and recovery cost is of reasonably acceptable order and shows good scalability.

Keywords

GASPI, GPI2, fault tolerance, fault detection, health-check library, node-level checkpoint-restart, fault-tolerant iterative solvers

1 Introduction

The advances in computational capacity of HPC clusters have enabled many fields in research and industry to progress far beyond imagination. Still the demand of more computational capacity is never ending. In the recent past, the consistent exponential growth is achieved with the help of extreme levels of hardware parallelism. This causes a severe reduction in mean time to failure (MTTF) of the overall systems and is visible with every new generation of large clusters. For example, the 'Intrepid' BlueGene/P system (debuted as # 4 on the top500* list of June 2008 installed at the Argonne National Laboratory) is reported to have the MTTF of 7.5 days (M. Snir et al. (2014)). In contrast, a more recent BlueGene/Q cluster 'Sequoia' (debuted as # 3 according to Nov. 2013 list, installed at Lawrence Livermore National Laboratory) has a node failure rate of 1.25 per day (Dongarra (2013)). On the way to exascale machines, the MTTF is expected to reduce to the order of hours or minutes (J. Daly et al. (2012); Schroeder and Gibson (2007)). This indicates an alarming behavior which, if not taken care of, will question the usability of clusters at exascale.

In HPC systems, programs can face many kinds of failures during runtime, e.g., hardware and software faults, silent errors, Byzantine failures, etc. According to El-Sayed and Schroeder (2013), 60% of all failures are attributed to either memory or CPU failures. Such failures, in addition to others, eventually lead to process or node-level failures, which are the focus of this work. The majority of the literature regarding fault tolerance towards fail-stop failures falls into either one or a combination of the following four categories: algorithm based fault tolerance (ABFT), checkpoint/restart (C/R), message logging, and redundancy (Hursey (2010)).

So far most algorithms (and underlying communication models) are built under a comprehensible assumption that the communication partners of every process stay alive and functional during the entire run of the program. Consequently, the failure of even a single process leads

[1]Erlangen Regional Computing Center, University of Erlangen-Nuremberg Erlangen, Germany
[2]Fraunhofer Institute for Industrial Mathematics (ITWM), Fraunhofer Platz 1, Kaiserslautern, Germany
[3]Institute of Physics, University of Greifswald, Greifswald, Germany

Corresponding author:
Faisal Shahzad, Erlangen Regional Computing Center, University of Erlangen-Nuremberg Erlangen, Germany
Email: faisal.shahzad@fau.de

*Top500 Website: http://top500.org

to the failure of the whole application. For large scale applications, it is beneficial to drop this assumption. This would mean that the program should continue to run even after the failure of a certain amount of processes, i.e., node failures. This opens a new dimension of research in the field of fault tolerance. In this context, the first step is to build/utilize a communication library that can provide the necessary supporting functionalities for this purpose, i.e., health information of processes, failure detection mechanisms, propagation of failure information to all relevant processes, etc. During the development of MPI 3.0, efforts were underway to introduce process-level failure tolerance (W. Bland et al. (2012)), but were not eventually successful. The fault tolerant working group in the MPI forum is currently working on the User-Level Failure Mitigation (ULFM) standard proposal and its prototype implementation (based on Open MPI) for its potential inclusion in the MPI 4.0 standard (W. Bland et al. (2013)). Despite the apparent attractiveness of this new approach, it comes with a new set of challenges, i.e., correct and consistent failure acknowledgment, rebuilding communication infrastructure, recovering lost data from failed process(es), etc. These building blocks require extra effort and caution during application development stages.

In this work, we use the GPI-2 [†] implementation of the GASPI specification [‡] to design fault tolerance supporting tools that can be easily used in applications to recover from hard-failures. We then use these tools in applications that are capable of recovering dynamically from process failures.

The main contributions of our work are as follows:

1. Design and implementation of a health-check (HC) library that provides fault detection, communication recovery and other useful functions in order to recover from fail-stop failures.
2. Implementation of a fault-aware neighbor-level C/R library that provides supporting functions to asynchronously transfer the node-local checkpoints to the neighbor nodes and fetch them back in case of a data-recovery request.
3. The usage of HC-library is showcased by its utilization in three different applications. Two application consist of iterative sparse matrix vector multiplication (spMVM) based solvers, whereas the third application is a fluid flow solver based on the lattice Boltzmann method (LBM). The underlying spMVM library and LBM communication routines are also made fault tolerant by utilizing HC-library.
4. A benchmark study has been performed to analyze the overheads involved in this fault tolerance approach. This includes a range of application runtime scenarios, i.e., overhead in failure-free case and overhead with 1,2,3 failure recoveries, etc. Moreover, the scalability of the fault detection method is tested with up to 256 nodes.

With this practice, we have developed fault tolerant applications which can heal themselves dynamically after the loss of one or more processes. Thus the time for restarting the job manually and wait-time spent in the queueing system for a new job request are avoided. The concept can be applied to other applications with different communication libraries as well when they support fault tolerance. We think that this is a good starting point to gain experience and become aware of the challenges involved and their potential solutions for utilization of such fault tolerant communication libraries.

The paper follows the following structure. Some preliminary definitions and concepts are described in Sect. 2. In Sect. 3, the GASPI communication interface is briefly introduced along with its fault tolerance features. The design and interface of our fault tolerance supporting libraries (i.e., HC-library and CR-library) are described in detail in Sect. 4. Section 5 provides the benchmark applications detail and the test-bed environment. A detailed example of the integration of HC and CR-libraries in the application is covered. The results and overhead analysis of our FT implementations are presented in Sect. 6. A brief summary of the related work is described in Sect. 7. We conclude the paper in Sect. 8 with a short discussion about the challenges of such a fault tolerance approach and possible improvements.

The current versions of these GPI support libraries (i.e. health-check library and checkpoint/restart library) are available for download at GPI2-HC-lib (2015) and respectively GPI2-CR-lib (2015).

2 Preliminaries

In this paper, we use the term ‘application-driven’ fault tolerance for an approach where an application can heal itself dynamically despite one or more process failures. A process in this context is a GPI process which is akin to MPI process with similar properties. Multiple GPI-processes can run simultaneously on one physical node of a cluster.

There are three central components in application-driven fault tolerance. The first and foremost component is to have a consistent, accurate, and reliable failure detection mechanism. Design and implementation of a fault detector for asynchronous systems is a complex task. A distributed system is termed as ‘asynchronous’ if there is no upper bound on message transmission delays and process execution time, and all real HPC systems fall into this category. There are two basic properties of a fault detector (Chandra and Toueg (1996)): 1) Completeness: The crashed processes are suspected by some healthy processes, after a certain amount of time. A strong completeness implies that every failed process is eventually detected by every healthy process. 2) Accuracy: Every detected failure corresponds to an actual crashed process (i.e. no false-positives). The complete satisfaction of both these properties is theoretically impossible (Chandra and Toueg (1996)). Thus, most fault detection implementations are willing to tolerate some level of inaccuracy but require strong completeness.

Our failure model encompasses fail-stop failures of all kinds i.e. any failure which results in termination of one or more processes in a parallel application. Such fail-stop failure can either be a consequence of any hardware component failure or any soft-error that result in process failure(s). This model is similar to the one followed by K. Kharbas et al. (2012). For failure detection, we follow the approach of having a dedicated health check (HC) process

[†]GPI2 website: http://www.gpi-site.com/gpi2/
[‡]GASPI website: http://www.gaspi.de/

that measures and keeps a global health view of all processes at all times. Due to the PGAS nature of the communication in GASPI, such a global failure detection has certain advantages in terms of simplicity over developing consensus between processes after failure(s) and induces no overhead in the failure-free case (discussed in detail in Sect. 4). This method is in contrast to the technique implemented in W. Bland et al. (2012) for MPI applications, where failure(s) are detected locally between the regular communication peer processes.

After a failure detection, the second step involves the selection of a communication recovery and domain redistribution strategy. Two recovery models can be followed for this purpose (I. Laguna et al. (2014)). 1) Shrinking recovery: In this method, the application proceeds with the rest of available processes after failure(s) and requires redistribution of domain. 2) Non-shrinking recovery: This method involves using new process(es) to replace the failed one(s), where the work distribution of the application is not changed after failures. Depending on the application and the communication library, either of these techniques can be beneficial. In our work, we have used a non-shrinking recovery method because of its ability to easily leverage node-level C/R approach. For non-shrinking recovery, instead of using the same physical node(s) as of failed process(es), we use pre-allocated extra nodes for recovery processes. This strategy ensures that failure prone nodes are no longer used for further work.

The third stage involves the data recovery of the lost process(es) and continuing further computation. Data recovery can either be done by reading a checkpoint or by using an algorithm based fault tolerance (ABFT) approach. An ABFT approach is by nature highly algorithm-specific and cannot be generalized; therefore, we have opted for application specific node-level C/R approach in our application.

3 Introduction to GASPI and GPI-2

GASPI is a communication library for C/C++ and Fortran. It is based on a PGAS style communication model where each process owns a partition of a globally accessible memory space. GPI-2, the GASPI implementation, takes full advantage of the hardware capability to perform remote direct memory access (RDMA). More importantly, there is a focus on providing truly asynchronous communication to overlap computation and communication, and on thread-safe communication which allows multi-threaded applications with a fine-grained communication and asynchronous execution capability.

GASPI defines a very compact API consisting of one-sided communication routines, passive communication, global atomics and collective operations. It also defines groups which are similar to MPI communicators and are used in collective operations. Furthermore, there is the concept of segments. Segments are contiguous blocks of memory and can be made accessible (to read and write) to all threads on all ranks of a GASPI program. Data to be communicated is thus placed in such segments.

Given the aforementioned increasing need for fault tolerant applications, GASPI was designed with that in mind. It supports application-driven fault tolerance on the process level. This means that the failure of a single process does not cause the entire program to fail. The program can in fact react to the failure and try to recover by "healing itself". This is achieved with two simple concepts: timeouts and an error state vector.

GASPI provides a timeout mechanism to all potentially blocking procedures. The user provides a timeout (in milliseconds) and the procedure returns after that time if it could not successfully complete. Furthermore, since a timeout does not necessarily imply an error or failure from the remote part, GASPI provides an error state vector that holds the state of processes. The state vector is set after every erroneous, non-local operation and can be used to detect failures on remote processes. Currently, each rank can either have a state of GASPI_STATE_HEALTHY or GASPI_STATE_CORRUPT. This error state vector can be queried by the application using *gaspi_state_vec_get* to determine the state of a remote partner in case of timeout or error.

Along with the previous mechanisms, for the context of our work, we have extended GPI-2 to provide an extra mechanism (not included in the GASPI specification) for fault-tolerant applications. The procedure *gaspi_proc_ping* tests the availability of a particular GPI-2 rank. As the name indicates, a *ping* message is sent to a particular process. In case a problem is detected, a GASPI_ERROR is returned to which the application can react.

In the next sections, we describe our technique in more detail.

4 Design and Implementation

This section describes our implementation of the basic building building blocks of application-driven fault tolerance namely fault-detection and propagation, communication reconstruction, and data recovery.

Our fault tolerance application implementation relies on the following principles. At the start of the program, some processes are designated as spare processes. This is due to the fact that current GASPI Standard does not provide the possibility of process spawning. Thus, in order to enable non-shrinking application recovery, spare processes are utilized. The processes are divided into 'worker' and 'idle' process categories. The worker-processes form the 'worker-group' and continue with the computations. One of the pre-determined idle process serves as a health-check (HC) process. The rest of the idle processes stay idle until notified by the HC-process to join the worker-group. After a failure is reported, the required number of idle processes join the worker communication and carry on computations after data recovery.

The data recovery is enabled through node-level C/R method. Despite the criticism it faces, the recent neighbor node-level C/R optimizations (A. Moody et al. (2010); K. Sato et al. (2012); L. Bautista-Gomez et al. (2011)) have enabled it to be a good candidate on future exascale systems (J. Daly et al. (2012)). We have implemented a node-level C/R library for GPI-2 applications. Our strict assumption of hard-failure enforces us that the failed-nodes are not accessed once a failure is reported from any particular node.

Thus we have used neighbor-level C/R library, so each checkpoint is stored locally as well as on the neighbor-level nodes. After failure, the neighboring nodes are bound to change, thus the C/R library is also made fault aware.

The functions of both the health check and the checkpoint/restart libraries are presented along with their detail are presented in the following.

4.1 Health Check library:

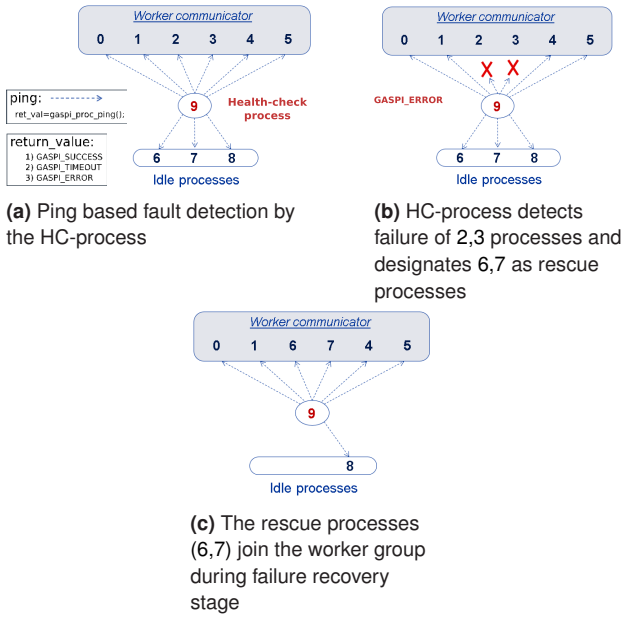
All processes initialize a HC-library object (by calling HC::init()) at the start of the simulation, that sets up the GASPI global memory for health check management. This global memory is used by HC-process to report the list of failed process(es), rescue process(es), etc. to the worker-processes by one-sided GASPI-communications. The most important member functions of HC-library objects are as follows.

DISTRIBUTE_WORKERS_OR_IDLE(int numprocs_idle): By taking input argument of numprocs_idle, this function sets up the program setting by defining the status of each process in HC-class, which can be either as WORKING or IDLE at the start of the program.

GET_STATUS(): This function can be used to get the status of any process. This is helpful to assign different task in the main program based on their status. The currently defined statuses are WORKING, IDLE, and WORKFINISHED.

DEFINE_SUB_COMM(int KEY, gaspi_group_t COMM, gaspi_rank_t * comm_ranks): This call can be used to make a GASPI group that include the processes of matching KEY argument. The function returns the gaspi-group COMM and a list of its processes (comm_ranks).

STAY_IDLE_AND_CHECK_HEALTH(int myrank_active): The most central member function in HC-class, this function is called by all idle processes including the HC-process and performs health check operations and failure management operations. The implementation of this function is shown in the Listing 1. The HC-process periodically checks the health (i.e. aliveness) of all other processes via ping operation using *gaspi_proc_ping()*. If failed process(es) are detected, the HC-process manages to inform all other processes about the list of failed process(es) and rescue process(es). Each rescue process also receives the ID of the process it replaces (myrank_active), which is useful in building new communication infrastructure. Figure 1a shows the schematic diagram of such one-sided ping based fault-detection. A reasonable ping frequency can be set by the user depending on the number of processes. After completion of a single cycle around all healthy processes, the HC-process has an updated global health view. We rely on the fact that each fail-stop failure eventually results in breakage of the communication channel between the HC and the failed process. This consequently leads to the return of ping operation with GASPI_ERROR (shown in Fig. 1b). After detection of failed process(es), the HC-process informs all healthy processes about the failed processes as well as their corresponding rescue



(a) Ping based fault detection by the HC-process

(b) HC-process detects failure of 2,3 processes and designates 6,7 as rescue processes

(c) The rescue processes (6,7) join the worker group during failure recovery stage

Figure 1. The working of the fault detector process. The return value of ping is GASPI_SUCCESS for all healthy processes (a), whereas it returns with GASPI_ERROR for failed processes (b). After failure detection, HC-process informs other processes about failed processes as well as their replacement processes. A new worker group is then built with the help of rescue processes (c).

processes. This is done via one-sided write in the global memory of all healthy processes. Meanwhile, the worker processes communicating directly with the failed processes keep on returning with GASPI_TIMEOUT unless a failure acknowledgment is received. The remaining healthy working processes continue with their work until they also receive a failure acknowledgment signal from the HC-process. After failure acknowledgments, no further regular application communication is performed and processes enter the recovery stage of the algorithm.

RECOVER_COMM(gaspi_group_t COMM_REPAIRED, gaspi_rank_t * comm_ranks): After the occurrence of failure(s) , this call is used to create the healthy group again. Both, the surviving processes in the broken group and the rescue processes must call this function. At the end of successful completion, the call returns a health group and list of its ranks in which failed processes are replaced with the ranks of replaced ones. For example, in case of failure of rank-ID 2 and 3, the original comm_ranks list of 0,1,2,3,4,5 will become 0,1,6,7,4,5 after this call (as shown in 1c). The reordering of ranks in the new group can be beneficial in determining the new communication infrastructure of the program.

IS_FAILURE_REPORTED(): This function can be used by worker-processes to determine if any failures have been reported by HC-process. Depending on whether a failure is reported or not, this function returns with ‘true’ or ‘false’ status correspondingly.

SIGNAL_ALL_PROCESSES(int SIGNAL): This function can be used to pass a signal to all other processes. For

```
int HC::stay_idle_and_check_health(gaspi_rank_t &
  myrank_active){
   while(1){
     if(get_status(myrank)==HC_proceses){
        glo_health_chk();
     }
  if(am_i_rescue_process()==true){
    update_myrank_active(myrank_active);
    break;
     }
     if(is_work_finishied==ture){
      break;
     }
  }
}

int HC::glo_health_chk(){
  int comm_state = WORKING;
  while(comm_state != BROKEN){
    gaspi_return_t retval;
    for(int rem_id=0; rem_id<numprocs(); ++rem_id){
      if(avoid_list[rem_id] != 1){  // avoids testing
        failed processes
        retval = gaspi_proc_ping(rem_id, GASPI_BLOCK);
      }
      if(retval == GASPI_ERROR){
        avoid_list[rem_id]=1;
        comm_state = BROKEN;
      }
    }
    if(comm_state_ == BROKEN){ break; }
  }
  // informing all healthy processes about failed
    processes and their rescue processes.
  if(comm_state == BROKEN){
    make_failed_and_rescue_proc_list(failed_proc_list,
      rescue_proc_list);
    report_healthy_processes_about_failure(failed_proc_list
      ,rescue_proc_list);
  }
  return comm_state;
}
```

Listing 1: The global health check routine. The fault detector process periodically checks the health of all healthy processes.

example, this can be used to pass the WORK_FINISHED signal to idle-processes so that they join the main program again.

In addition to these calls, HC-library also provides fault-tolerant variants of GASPI communication waiting calls. These calls prevent the deadlock in case a blocking communication fails because of the failure of the corresponding communication partner. These calls include 1) FT_gaspi_wait(), 2) FT_gaspi_notify_waitsome(), and 3) FT_gaspi_allreduce().

Fault Detector Properties: The HC-process gets an updated global view of all processes' health at the end of each ping cycle around all healthy processes. Thus, a failed process is bound to be detected by the HC-process in a finite amount of time. The information about failed processes is then given to all remaining healthy processes. Thus the HC-process satisfies the property of completeness. As a rare case, there can be instances that (due to a network related problem) a healthy process is not reachable by the HC-process but is accessible by some or all other processes. This will result in a false-positive signal about a processes failure. Thus the property of accuracy is only loosely satisfied. A false-positive signal will lead the application to undergo the failure recovery phase but it does not affect the correctness of the program results.

Alternative investigated failure detection methods We also investigated the following failure detection mechanisms:

1. Ping-based all-to-all: In this method each process performs periodic all-to-all pings to detect the failures.
2. Ping-based neighbor level: Each process 'i' periodically pings only its neighboring process 'i+1'. In case a failure is detected at the neighbor level, a ping based all-to-all operation is triggered on all processes to get a global health view.

In both of these approaches, a potential deadlock can arise in the cases where multiple processes detect different sets of failed processes. Reaching a consensus about the identified failures adds further complexity in the algorithm. Moreover an all-to-all ping based approach is not a scalable failure detection method and in failure free cases a certain amount of overhead in both cases is introduced. In contrast, a dedicated HC-process with one-sided ping eliminates the potential deadlock situation and causes negligible overhead in failure-free cases.

4.2 Data Recovery: fault-aware node-level Checkpoint/Restart library (CR-library)

As described earlier in Sect. 2, we rely on the node-level C/R method for the data recovery of the failed processes. Each process is responsible itself for creating checkpoints on the local node file system and restarting from it. The neighbor-level checkpoint functionality is realized by implementing a CR-library and facilitates to transfer the local-node checkpoints to neighboring-nodes in an asynchronous manner using a dedicated CR-thread per node. In case of a failure, the CR-library is also responsible for bringing the checkpoint of the failed process from failed-process's neighbor-node. In a fault-tolerant environment, the neighbor-nodes of the processes are bound to change after recovery. Thus, the CR-library has to be fault-aware as well. In order to recover from catastrophic failures, in which more than one consecutive failures occur at the same time, the less occasional checkpoints on the parallel file system can also be made. The user can determine the number of checkpoints to be kept in the file system. The primary functions of CR-library class are detailed below:

CRLIB_INIT(): This is the first call of CRLIB object and is called by all processes. The function takes machine file, processes-status array (either WORKING or IDLE) and checkpoint destination paths as the input argument. It first determines the neighboring-nodes of each corresponding node. A p-thread is then created which is then responsible for transferring the local node-level checkpoints to the neighboring nodes, once the checkpoint-transfer signal is received.

START_CP_TRANSFER(): After creating a checkpoint in the local-node file system, this call is used to signal the CR-library that this checkpoint is ready to be transfered to the neighbor-level nodes.

WAIT_FOR_CP_TRANSFER(): This call can be used at any point in the program to make sure that the last local-node checkpoint has been completely synced with the neighbor-level checkpoints.

FETCH_CP(): This routine is responsible to make sure that the last consistent checkpoint is available on the local-node file system to restart with. In the case of recovery process, it fetches the checkpoint from the failed-process's neighbor-node.

REFRESH_NEIGHBOR_LIST(): This call takes the list of failed-processes and rescue processes as input and prepares the new CR-library threads for their respective refreshed neighbor-nodes.

The practical usage of these function is shown via examples in Sect. 5. The current scheme has following restrictions which we plan to address in future work.

1. The number of failures a program can sustain is equal to the number of idle processes specified at the start of the program.
2. The fault tolerance capability of a program ends if the HC-process becomes a worker.
3. Currently, the failure of HC-process itself ends the fault tolerance capability of the program but does not effect its progress.
4. Only those network failures can be detected that can be uniformly seen by the effected processes as well as by the HC-process.

4.3 Fault Tolerance Overhead

Each of the fault tolerance techniques carries a certain amount of overhead in terms of time and/or resources. Our approach requires the allocation of some extra nodes for fall back scenarios, which is a resource overhead. The calculation of the optimal number of extra nodes for a particular case depends on several factors including job size, job duration, the MTTF of the system, etc. and is out of scope for this paper. In the following, we term overhead as the increase in application run time due to its fault tolerance capabilities.

In principle, the fault detection with global ping messages is an expensive operation. In our method, a HC-process performs one-sided pings to get the global health view. On the other hand, the worker processes check for a failure acknowledgment signal from the HC-process before each communication call. Thus, from the working processes' standpoint, the failure detection mechanism adds very little cost to the overall run-time of the algorithm. As we shall see in Sect. 6, this cost is negligible.

The first major overhead is introduced by checkpointing. There can be two kinds of checkpoints: a global PFS-level checkpoint, and a neighbor level checkpoint. In case of a restart, the data is initialized from a consistent checkpoint. The processes must redo the work, up to the point where the actual failure happened. This is the source of the second overhead and depends on the instant where the failure occurred between two checkpoints.

In case of a failure, all processes go through the failure detection and recovery stage of the program. This constitutes the third form of overhead and can be decomposed into three categories:

- Failure detection overhead/communication with failed processes (OH_{F1}): This is the time it takes for the HC-process to successfully detect and acknowledge the failures to other processes. Meanwhile, the healthy processes trying to communicate with the failed processes end up in timeout-based returns unless a failure acknowledgment is received.
- Rebuilding of work group (OH_{F2}): This step involves the creation of a new worker group, replacing failed processes. Due to the blocking nature of the *gaspi_group_commit()* procedure involved during communication rebuilding, this overhead is non-negligible.
- Reinitialization of data (OH_{F3}): After the new work group is formed, the data structure gets initialized from the last consistent checkpoint. If the checkpoint is not available on the local node, it is fetched from the neighbor node of the failed process by the checkpoint library.

In the next sections, we perform benchmarks and get a quantitative notion of the above mentioned overheads in a practical scenario.

5 Experimental Framework

This section explains the algorithms that are used to showcase the usage of previously described HC and CR libraries in various applications. In order to avoid the communication deadlocks in a fault tolerant environment, no blocking communication operations are used, rather we use the helping communication features implemented in HC-library namely as 1) FT_gaspi_wait(), 2) FT_gaspi_notify_waitsome(), and 3) FT_gaspi_allreduce(). These functions break the waiting-loop for the respective operation after reaching a specified timeout, provided a failure is detected in the group by the HC-processes and transmitted to the worker-processes. After this acknowledgement of the failed processes, no further communication is performed by the worker-process and they can enter the recovery part of the algorithm.

5.1 The lattice Boltzmann method(LBM):

The lattice Boltzmann Method (Succi (2001)) is a fluid flow solver that has gained significant popularity in science and research communities in the recent past due to the potential of its computational efficiency on the modern hardware. The LBM can be seen as a Jacobi-like stencil algorithm, but with two major differences: (1) each cell has not only one, but (as in our case) 19 values and (2) no values read are reused during the same iteration over the lattice. Our benchmark implementation uses the D3Q19 model (Qian et al. (1992)) with the BGK collision operator (Bhatnagar et al. (1954)). In each iteration, the data is read from the 4-D source lattice (three spatial dimensions plus one for the 19 cell values) and modified values are written to the 4-D destination grid. The update of one cell is performed by reading one value of each of the cell's 19 surrounding neighbors. Out of these values new ones are computed, which are used to update the cell's own values in the destination lattice. Thereby the values are

```
int main(){
...
CR * myCR = new CR[1];
myCR->init(machine_file, my_cp_param, status_processes);
HC * myHC = new HC[1];
myHCinit();
myHC->distribute_worker_or_idle(numprocs_idle);
myHC->define_sub_comm(WORKING, COMM_WORKER, comm_ranks);

if(myHC->get_status(myrank) == WORKING){
  // simulation geometry setup according
  // to COMM_WORKER and comm_ranks
}

if(get_status(myrank) == IDLE){
     myHC->stay_idle_and_check_health(myrank_active);
}

for(int iter=0; iter!= total_iters; ++iter){
     if(myHC->get_status(myrank)==WORKING){
        // do-computation-communication

       if(iter%cp_step==0){
         // check for previous successful CP transfer
         myCR->wait_for_CP_transfer();
         write_local_CP();
         myCR->start_cp_transfer();
         }
     }
     if(myHC->is_failure_reported()==TRUE){
       myHC->recover_comm(COMM_WORKER, comm_ranks);
         // setup geometry again according
         // to COMM_WORKER and comm_ranks
       myCR->fetch_cp(myrank_active,
                      myHC->rescue_proc_list);
       myCR->refresh_neighbor_list(myHC->failed_proc_list,
                                   myHC->rescue_proc_list);
       restart(...); // re-init data from local-node-CP
     }
}
myHC->signal_all_processes(WORKFINISHED);
...
delete[] myHC;
delete[] myCR;
...
}
```

Listing 2: The fault tolerace functionality integration of HC-library and CR-library in the LBM-algorithm.

arranged in a structure-of-array data layout (for details see Wellein et al. (2006)).

The variations required to make the program fault-tolerant using the HC-library and CR-library are shown in the Listing 2. All process first define and initialize the HC and CR-library objects. The processes are then designated as either worker or idle-processes. A working group is then formed by the worker processes and the necessary simulation data and communication initializations are performed. The idle-processes stay idle and one pre-designated idle process acts as the HC-process. The worker processes continue with iterative computation-communication routines and make occasional node-level checkpoints which are transfered to the neighbor-level nodes by the CR-library threads in an asynchronous way. In case a failure is reported by HC-process to the worker-processes, communication recovery is made to recover the group, where idle-process(es) replace the failed ones. Before restarting from checkpoints, the CR-library is used to fetch the checkpoint of the failed process to the local-node.

5.2 Sparse linear solvers:

We have implemented our fault tolerance technique on two different sparse linear iterative solvers. These include Lanczos algorithm that is used to find the target eigenvalues of sparse matrices and a Chebyshev recursion scheme named as kernel polynomial method (KPM) to calculate the Hamiltonians of sparse matrices.

As both algorithms are based on sparse linear algebra, we have developed a GASPI based library for basic parallel sparse matrix-vector(spMV) related operations. For a parallel sparse matrix-vector-multiplication (spMVM) operation, the pre-processing stage includes the setup of communication. In this stage, the spMVM operation is divided into two parts, a local part for which the process has right-hand vector values (RHS) locally available, and a remote part for which the process needs to fetch RHS values from other processes. In the pre-processing stage, each process determines the indices of the RHS that it needs from other processes. These indices are communicated to the respective processes, who then write (via one-sided GASPI communication) the RHS values of those indices before every spMVM iteration.

In the fault tolerant versions of these algorithms, each process writes a matrix-checkpoint after the matrix-preprocessing stage. This checkpoint stores information relevant for communication with other processes. After failure recovery, the rescue process reads the checkpoint of the failed process. In this way, the rescue process is informed about the communicating partners and the respective RHS indices to communicate to other processes. Every non-failing process also refreshes its list of communication partners by replacing the rank of the failed process with the new rescue process. Using this method, the program can resume the computations after failure recovery without having to perform the matrix-preprocessing step again (which would add the overall cost of recovery).

The matrix for our benchmark cases arises from the quantum-mechanical description of electron transport properties in graphene. Graphene is a blueprint for quasi two-dimensional materials with many interesting properties and prospective applications in several fields of nanotechnology and nanoelectronics. A matrix generation library tool is used to construct the matrix on the fly. Depending upon the specified geometry size, each process allocates its own chunk of the matrix. This way, the expensive step of reading the matrix from PFS is avoided.

Lanczos: The Lanczos algorithm (Lanczos (1950)) is an iterative scheme to find eigenvalues of a sparse matrix. We use it to find the low-lying eigenvalues of a test matrix. Algorithm 1 shows the pseudo-code of the basic Lanczos algorithm. Each iteration calculates the new Lanczos vectors, α, and β. After obtaining the α and β values of each iteration, the approximated minimum eigenvalues are determined using the QL method and are checked against a convergence criterion. The checkpointing data consists of two consecutive Lanczos vectors, α, and β.

Kernel Polynomial Method: The Kernel Polynomial Method (KPM) is a common method in computational solid state physics (Weiße et al. (2006)) to compute the density of state (eigenvalues) and other spectral properties

Algorithm 1 The Lanczos algorithm for finding eigenvalues of a matrix A

```
for j:=1,2, ..., ConvergenceCriterion do
    function LANCZOS-STEP
        ω_j ← Aν_j
        α_j ← ω_j.ν_j
        ω_j ← ω_j − α_j ν_j − β_j ν_{j−1}
        β_{j+1} ← ||ω_j||
        ν_{j+1} ← ω_j/β_{j+1}
    end function
     CalcMinimumEigenVal()
end for
```

of materials such as graphene and topological insulators. In this algorithm, the most computationally expensive routine is the calculation of the moments (η_i vector) by the scalar products of matrix polynomials (see Alg. 2) In comparison with Lanczos algorithm, it has a high optimization potential to reduce its computational balance (e.g. by avoiding global synchronizations inside the loop (Kreutzer et al. (2015))), thus making it an attractive candidate for fault-tolerance case study. Unlike the other two algorithms, its computationally intensive part consist of two nested loops. This property makes it particularly challenging as both the checkpoints and restart have to be carried out in two different levels of the application.

Algorithm 2 The Kernel Polynomial Method in its Naïve form.

```
for r:=1,2, ..., R do
    ν ← rand()
    Initialization steps and computation of η_0, η_1
    for m:=1,2, ..., M/2 do
        swap(ω, ν)
        ω ← 2a(A − b)ν − ω
        η_{2m} ← ν.ν
        η_{2m+1} ← ω.ν
    end for
     ...
end for
```

The fault tolerant version of both of these algorithms (Lanczos and KPM) have conceptually similar changes in the algorithm as described with the example of LBM in Listing 2.

Testbed: All tests were done on the LiMa cluster[§] at RRZE, whose nodes are equipped with two Intel Xeon 5650 "Westmere" chips with a base frequency of 2.66 GHz. Each node has 24 GB of RAM (12 GB per NUMA domain). The system has Mellanox QDR InfiniBand (IB) and GBit Ethernet interconnects.

6 Results

In this section, we test our benchmark applications with various runtime scenarios and compare their relative overheads. We have verified the recovery mechanism by killing the application processes in following three ways: i) Exiting the processes using 'exit(-1)' within the program ii) Using 'kill -9 <process-id>' iii) Physically introducing a network failure. Turning off complete compute nodes was not done due to limitations of the batch queuing system which would delete the complete job owing to the dead node. In order to accurately record the failure injection time and have a deterministic redo-work time, we have used the first described method i.e. killing the application from within the program.

The GASPI communication timeout value is set to $1000ms$, after which the processes retry communication unless a failure acknowledgement is received from the HC-process. Furthermore, the ping scan frequency of the HC-process is set to $100ms$.

6.1 Lanczos:

In the Lanczos algorithm, the stopping criterion depends on the convergence of the required eigenvalues up to a certain accuracy. For benchmarking purposes, we use a fixed number of iterations (2500) as a stopping criterion. The neighbor-node level checkpoints are taken at every 500th iteration. For the purpose of having a deterministic redo-work time, the failures are injected 100 iterations after making a checkpoint. As discussed in the previous section, the matrix checkpoint is stored once after its pre-processing stage at the start of the algorithm and after every failure-recovery. The used Graphene matrix consists of $1.4 \cdot 10^8$ rows and columns, and $1.8 \cdot 10^9$ non-zeros. The periodic checkpoints consist only of Lanczos vectors and the calculated eigenvalues up to the corresponding iteration, resulting in a combined size of all node-level checkpoints of ≈ 2.3 GBytes.

Figure 2 shows the runtime of the Lanczos benchmark on 256 nodes (256 processes, 12 threads/process) in several cases. The application is started with four idle processes (nodes) reserved for failure recovery. The first case represents the application runtime where no health check is performed and no checkpoints are taken ('w/o HC, w/o CP' bar). To have a fair comparison, the case without health-check is run on 252 nodes. This runtime marks the baseline case of our overhead study. The next case shows the application runtime with health check enabled but without taking any checkpoints ('with HC, w/o CP'). The health check feature does not add any overhead to the application. This is due to the fact that health checks are done by a dedicated HC-process and are performed via one-sided ping operation.

The neighbor-node level checkpoints add very little overhead of around 0.9% ('with HC, with CP' bar') in this case. The significant overhead only appears in the case of failure recovery scenarios. Each failure adds approximately 35 seconds to the total runtime of the application. The average failure detection and acknowledgement by all processes take approximately 2 seconds, after which group reconstruction takes additional ≈ 1 second. The major part of the overhead comes in the form of matrix re-initialization with reconstructed group and then checkpointing the matrix-checkpoint which now contain the communication pattern of reconstructed group. The next step includes reading

[§]LiMa cluster at the Erlangen Regional Computing Center (RRZE): http://www.hpc.rrze.fau.de/systeme/lima-cluster.shtml

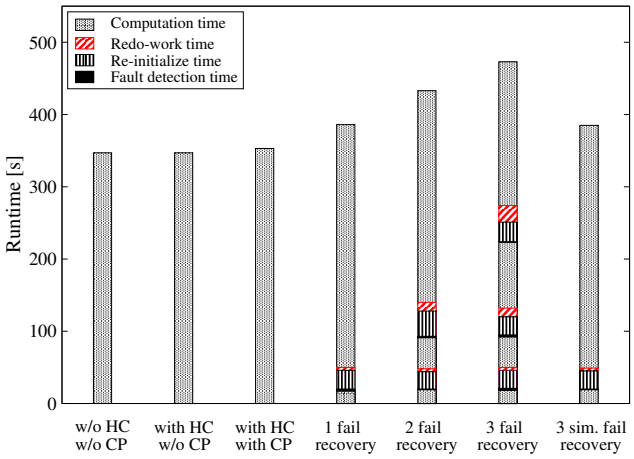


Figure 2. Various runtime scenarios of Lanczos application on 256 nodes. The health-check feature does not add any overhead and the neighbor-level checkpoints also add very marginal overhead. Each failure recovery costs ≈ 35 seconds, in which matrix-rebuilding contributes major part. The case without health-check is performed on 252 nodes.

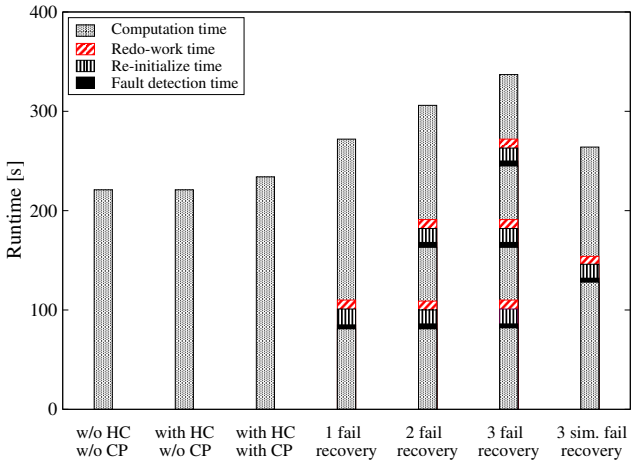


Figure 3. Runtime scenarios of the KPM algorithm on 256 nodes. Each failure-recovery costs an overhead of ≈ 18 seconds. The case without health-check is performed on 252 nodes.

the node-level checkpoints to initialize the vectors and eigenvalues data. The whole recovery time cost around $\approx$ 25 seconds in this case. The redo-work time contributes also a significant part of the total overhead. In practice, the average time for redo-work is the time between two successive checkpoints. Owing to a good checkpoint strategy with very low overhead, the checkpoint frequency can be increased which will lead to the reduction of redo-work time. Recovery from more failures adds approximately proportional overhead as shown in Fig. 2 with two and three recovery case runtimes. In real-time applications, a likely scenario is to have multiple failures simultaneously (e.g., failure of a node with multiple processes). Thus, we have used a threaded HC-process (with 8 threads) to check the state of more than one process by monitoring one-sided pings in parallel. This is highlighted in '3 sim. fail recovery' case, where three simultaneous failures are detected with the overhead of one failure detection.

6.2 KPM:

As the KPM algorithm has the same characteristic nature as Lanczos algorithm, in which sparse linear algebra operations are supported by same spMV-library, the overheads in the benchmarks are also similar in nature. In this case, we are using the matrix of $1.4 \cdot 10^8$ rows and columns and $1.8 \cdot 10^9$ number of non-zeros. The checkpoint in the outer loop consist of a relatively smaller vector 'mu', whereas the inner checkpoint stores the iterative vectors' ('y') information. Besides, some meta-data information is stored with every checkpoint. The combined size of all node-level checkpoint is ≈ 2.3 GBytes. The benchmark is run for 5 outer loop iterations and 100 inner loop iterations. The results of the outer loop are checkpointed after every outer iteration, whereas inner checkpoints are made after every 50 iterations.

Figure 2 shows the KPM benchmarks results on 256 nodes with various runtime scenarios. The runtime with and without health-check feature do not vary at all. The cost of neighbor-level checkpoints add $\approx 5\%$ to the overall runtime. Each failure adds approximately 33 seconds to the overall runtime, in which ≈ 18 seconds are accounted for communication and data recovery, whereas the rest comes in form of redo work.

Num. of Nodes	8	16	32	64	128	256
Avg. ping scan time[s]	6.2×10^{-5}	8.8×10^{-5}	1.7×10^{-4}	3.9×10^{-4}	7.7×10^{-4}	1.6×10^{-3}

Table 1. The average ping scan time of the HC-process with respect to the number of nodes.

6.3 LBM:

The LBM benchmark is performed on a toy-problem of 3D-lid driven cavity. The size of the grid is chosen to be $200 \times 200 \times 50400$, thus each of the 252 worker-nodes have a grid of size 200^3. A total of 2000 iterations are performed with a checkpoint taken at every 500th iteration and the failures are injected 100 iterations after making a checkpoint. As LBM is a grid based application, one complete grid is required for the restart operation. The size of each local-node checkpoint is 1.2 GBytes, making a global-checkpoint of 302 GBytes.

Figure 4 shows the LBM benchmark runtimes with different scenarios on 256 nodes (with 4 idle nodes). As with the previous two applications, the health-check feature does not add any overhead. Despite the large amount of overhead, each checkpoint adds only ≈ 12 seconds to the overall runtime. The failure recovery time takes ≈ 20 seconds, which include the failure detection, group recovery and restart by fetching node-level checkpoints. The rest of the overhead is caused by redo-work.

Health-Check time: It is important to observe and analyze the scaling behavior of a fault detection mechanism. Table 1 shows the ping scan time of the HC-process without any failure. The HC-process takes approximately $5 - 6\mu s$ to perform a ping with each healthy process.

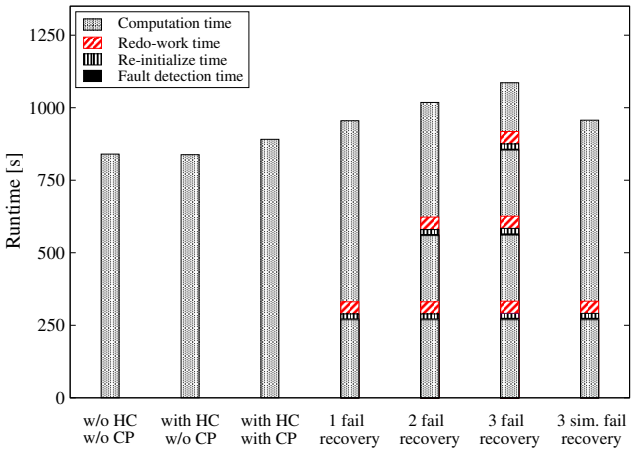


Figure 4. Various runtime scenarios of LBM application on 256 nodes. Each failure recovery takes $\approx$ 20 seconds to recover. The case without health-check is performed on 252 nodes.

7 Related Work

MPICH-V is one of the earliest research effort in terms fault-tolerant for MPI programs (G. Bosilca et al. (2002)). It featured complete checkpointing in combination with message logging to recover the data of the lost process. A more recent prototype implementation of ULFM using Open MPI has produced similar work to ours in terms of failure recovery strategies and/or performance evaluation.

Bland et al. have implemented a fault tolerant Monte Carlo Communication Kernel proxy-app using ULFM in W. Bland et al. (2014). They have demonstrated two recovery modes of the application, a classic C/R approach and an alternative solution in which the critical data is stored on neighboring nodes after each iteration. The recovery is done by spawning an additional process which reads the data from the failed process's neighboring node. In S. Pauli et al. (2013), a different approach of data recovery is implemented to develop a fault tolerant Multi-Level Monte Carlo based application using ULFM. The data from failed processes are altogether discarded and only survivors' data is used to complete the application further. Depending on the number of failures, the estimation error deteriorates the quality of results. A similar work is pursued by Ali et al. to develop a 2D PDE solver in M. M. Ali et al. (2014). They compare three alternative data recovery approaches based on C/R and approximation techniques. All mentioned studies have found ULFM to have acceptable overhead for failure recovery. In I. Laguna et al. (2014), Laguna et al. have conducted a broad and critical evaluation of ULFM. They have first discussed preferable recovery modes based on the nature of the applications and then performed a case study on a ULFM-based implementation of a fault-tolerant molecular dynamics application (ddcMD). They have found the time of revoking and shrinking the communicator to be increasing linearly with increasing number of nodes. In the end, they have suggested improvements to ULFM to make it an attractive option for application developers.

The ULFM studies are based on the detection of faults by communication failure between processes. No explicit failure detection methods are used. In some respects, the work closest to ours is by K. Kharbas et al. in K. Kharbas et al. (2012) where they have evaluated two kinds of fault detector mechanisms for MPI applications based on periodic and sporadic probing. The probing is based on ping-pong style messaging. They have only reported the overhead for failure-free cases which ranges from $1-21\%$ for NAS-parallel benchmarks. The fault detector overhead is negligible when separate background processes are used to check failures on the same set of nodes and the main application. In this case the background processes use Gigabit Ethernet instead of the Infiniband network used by the application, but this is not a suitable design as a large category of faults are network related. They conclude that a separate periodic fault detection mechanism is a superior method as compared to an sporadic approach.

There have been many efforts to introduce fault tolerance protocol in PGAS-model languages (Cunningham et al. (2014), Besta and Hoefler (2014), Ali et al. (2011)) but, to our knowledge, there is no such contribution for GASPI so far. GASPI is arguably the first PGAS based communication library, in which fault tolerance basics are embedded right from its development phase. The challenge is to use those basic features to build a fault tolerant application. Our presented tools provide a way for GASPI developers to develop FT applications, without modifying the algorithm, in an easier and novel fashion.

8 Summary

We have implemented fault tolerant supplementary library on top of GASPI (GPI-2) communication library. It provides useful functions for failure detection, propagation, communication recovery etc. We have used the idea of spare processes, out of which one acts as a health check (HC) process and monitors the health of all processes via periodic one-sided ping operations. Once a failure is detected, it propagates the failure and recovery information to all relevant processes, which then go through communication and data recovery routines. We have also implemented a supporting neighbor node-level checkpoint/restart library which is also fault tolerance aware.

We have used these tools in three different applications to make them fault tolerant. Two of these applications fall in the category of sparse linear algebra (linear solvers) namely Lanczos method and the Kernel Polynomial Method (KPM), whereas the third application is a fluid flow solver based on the lattice Boltzmann method.

The benchmarks performed on 256 nodes show that having an explicit HC-process causes no overhead to the working processes. The total overhead of failure recoveries is highly application dependent. In our tested applications, the failure detection and communication recovery costs $\approx$ 3 seconds. The data reinitialization of data is application dependent and ranges from $\approx 18-25$ seconds. Each additional failure adds similar cost. On the other hand, if multiple failures happen simultaneously (likely scenario for node failures), they have the potential to be detected at the cost of a single failure. Our failure detection mechanism shows good scaling behavior on the scaling test performed up to 256 nodes.

Discussion and Future work: The fault tolerance research is in its infancy stages regarding application-driven fault tolerance. Only having FT communication is not nearly enough. Many other components are needed to build an FT application in its true sense, e.g., failure information propagation to healthy processes, communication reconstruction after failure, data recovery method, etc. These add extra burden in terms of application development. With this work, we have created tools that can provide significant ease to the developers of fault tolerant GASPI application.

Our future work is targeted to remove the current limitations of our approach and making the technique more general and user friendly. The redundancy approach can be implemented to make the HC-process fault tolerant itself. We also plan to compare this fault tolerance approach with the Open MPI's ULFM functionality. In terms of the benchmarked applications, the matrix reinitialization step can be significantly improved to reduce the cost of failure recoveries. The release version of the health-check and checkpoint/restart tools created in this work will soon be made available.

Acknowledgements

This work was partly supported by the German Research Foundation (DFG) through the Priority Programme 1648 "Software for Exascale Computing" (SPPEXA) and partly by Federal Ministry of Education and Research (BMBF) under project "A Fault Tolerant Environment for Peta-scale MPI-solvers" (FETOL) (grant No. 01IH11011C). A preliminary version of this paper was presented in Shahzad et al. (2015).

References

G Bosilca et al (2002) MPICH-V: Toward a scalable fault tolerant mpi for volatile nodes. In: *Supercomputing, ACM/IEEE 2002 Conference*. pp. 29–29.

W Bland et al (2013) Post-failure recovery of MPI communication capability: Design and rationale 27(3): 244–254.

A Moody et al (2010) Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In: *Proceedings of the 2010 ACM/IEEE International Conference for HPC, Networking, Storage and Analysis*. Washington,DC, USA. ISBN 978-1-4244-7559-9, pp. 1–11.

Ali N, Krishnamoorthy S, Govind N, Kowalski K and Sadayappan P (2011) Application-specific fault tolerance via data access characterization. In: *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, pp. 340–352.

Besta M and Hoefler T (2014) Fault tolerance for remote memory access programming models. In: *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14. New York, NY, USA: ACM, pp. 37–48. DOI:10.1145/2600212.2600224.

Bhatnagar PL, Gross EP and Krook M (1954) A model for collision processes in gases: Small amplitude processes in charged and neutral one-component systems. *Phys. Rev.* 94: 511–525.

Chandra TD and Toueg S (1996) Unreliable failure detectors for reliable distributed systems. *J. ACM* 43(2): 225–267. DOI: 10.1145/226643.226647.

Cunningham D, Grove D, Herta B, Iyengar A, Kawachiya K, Murata H, Saraswat V, Takeuchi M and Tardieu O (2014) Resilient x10: Efficient failure-aware programming. *SIGPLAN Not.* 49(8): 67–80. DOI:10.1145/2692916.2555248.

Dongarra J (2013) Emerging Heterogeneous Technologies for High Performance Computing. http://www.netlib.org/utk/people/JackDongarra/SLIDES/hcw-0513.pdf.

El-Sayed N and Schroeder B (2013) Reading between the lines of failure logs: Understanding how HPC systems fail. In: *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

GPI2-CR-lib (2015) Node-level checkpoint/restart toolkit for gpi2. URL https://bitbucket.org/fshahzad/gpi_crlib.

GPI2-HC-lib (2015) Health check toolkit for GPI2. URL https://bitbucket.org/fshahzad/gpi_hclib.

Hursey J (2010) *Coordinated Checkpoint/Restart Process Fault Tolerance for MPI Applications on HPC Systems*. PhD Thesis, Indiana University, Bloomington, IN, USA.

I Laguna et al (2014) Evaluating user-level fault tolerance for mpi applications. In: *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14. ISBN 978-1-4503-2875-3, pp. 57:57–57:62.

J Daly et al (2012) Inter-Agency Workshop on HPC Resilience at Extreme Scale. Technical report.

K Kharbas et al (2012) Assessing HPC failure detectors for MPI jobs. In: *Proceedings of the 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, Munich, Germany*. pp. 81–88.

K Sato et al (2012) Design and modeling of a non-blocking checkpointing system. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Los Alamitos, CA, USA: IEEE Computer Society Press. ISBN 978-1-4673-0804-5, pp. 19:1–19:10.

Kreutzer M, Pieper A, Hager G, Wellein G, Alvermann A and Fehske H (2015) Performance engineering of the kernel polynomal method on large-scale cpu-gpu systems. In: *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. pp. 417–426. DOI:10.1109/IPDPS.2015.76.

L Bautista-Gomez et al (2011) FTI: High performance fault tolerance interface for hybrid systems. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA: ACM. ISBN 978-1-4503-0771-0, pp. 32:1–32:32.

Lanczos C (1950) An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *Journal of Research of the National Bureau of Standards* 45: 255–282.

M M Ali et al (2014) Application Level Fault Recovery: Using Fault-Tolerant Open MPI in a PDE Solver. In: *Parallel Distributed Processing Symposium Workshops (IPDPSW)*. pp. 1169–1178. DOI:10.1109/IPDPSW.2014.132.

M Snir et al (2014) Addressing failures in exascale computing. *International Journal of High Performance Computing Applications* 28(2): 127 171. DOI:10.1177/1094342014522573.

Qian YH, D'Humires D and Lallemand P (1992) Lattice BGK models for Navier-Stokes equation. *EPL (Europhysics Letters)* 17(6): 479–484. URL http://stacks.iop.

org/0295-5075/17/i=6/a=001.

S Pauli et al (2013) A fault tolerant implementation of Multi-Level Monte Carlo methods. In: *Parallel Computing: Accelerating Computational Science and Engineering (CSE), Proceedings of the International Conference on Parallel Computing*. pp. 471–480.

Schroeder B and Gibson GA (2007) Understanding failures in petascale computers. *Journal of Physics: Conference Series* .

Shahzad F, Kreutzer M, Zeiser T, Machado R, Pieper A, Hager G and Wellein G (2015) Building a fault tolerant application using the GASPI communication layer. In: *Proceedings of the 1st International Workshop on Fault Tolerant Systems(FTS 2015), in conjunction with IEEE Cluster 2015*. pp. 580–587.

Succi S (2001) *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Oxford University Press.

W Bland et al (2012) A proposal for User-Level Failure Mitigation in the MPI-3 Standard. http://www.mcs.anl.gov/~wbland/pdf/Bland_2012_A_proposal_for_User-Level_Failure_Mitigation_in_the_MPI-3_standard.pdf.

W Bland et al (2014) Simplifying the recovery model of user-level failure mitigation. In: *Proceedings of the 2014 Workshop on Exascale MPI*.

Weiße A, Wellein G, Alvermann A and Fehske H (2006) The kernel polynomial method. *Rev. Mod. Phys.* 78: 275–306. DOI: 10.1103/RevModPhys.78.275. URL http://link.aps.org/doi/10.1103/RevModPhys.78.275.

Wellein G, Zeiser T, Hager G and Donath S (2006) On the single processor performance of simple lattice Boltzmann kernels. *Computers & Fluids* 35(8–9): 910–919.

Author Biographies

Faisal Shahzad is a PhD student in the HPC group at Erlangen Regional Computing Center (RRZE). He received his B.Sc. degree in Mechanical Engineering in 2006 from GIK Institute of Science and Technology, Topi, Pakistan and M.Sc. degree in Computational Engineering in 2011 from University of Erlangen-Nuremberg, Germany. His research area focuses to research various fault tolerance techniques for variety of algorithms.

Moritz Kreutzer completed his B.Sc. in Computation Engineering in 2009 and M.Sc. in Computation Engineering in 2012 from University of Erlangen-Nuremberg, Germany. Currently, he is working as PhD student in HPC group at Erlangen Regional Computing Center (RRZE).

Thomas Zeiser Thomas Zeiser holds a PhD in Computational Fluid Mechanics from the University of Erlangen-Nürnberg. He is now a senior research scientist in the HPC group of RRZE and among many other things still interessted in lattice Boltzmann methods.

Rui Machado holds a PhD in Computer Science from the University of Evora, Portugal. He is a research fellow at the Competence Center for High-Performance Computing of the Fraunhofer Institute for Industrial Mathematics in Kaiserslautern, Germany.

Andreas Pieper is a physist and is working as a PhD student at the Ernst-Moritz-Arndt-Universität Greifswald.

Georg Hager holds a PhD in Computational Physics from the University of Greifswald. He has been working with high performance systems since 1995, and is now a senior research scientist in the HPC group at Erlangen Regional Computing Center (RRZE). Recent research includes architecture-specific optimization for current microprocessors, performance modeling on processor and system levels, and the efficient use of hybrid parallel systems. His daily work encompasses all aspects of user support in HPC such as lectures, tutorials, training, code parallelization, profiling and optimization, and the assessment of novel computer architectures and tools

Gerhard Wellein holds a PhD in Solid State Physics from the University of Bayreuth and is a regular Professor at the Department for Computer Science at University of Erlangen. He heads the HPC group at Erlangen Regional Computing Center (RRZE) and has more than 10 years of experience in teaching HPC techniques to students and scientists from Computational Science and Engineering. His research interests include solving large sparse eigenvalue problems, novel parallelization approaches, performance modeling, and architecture-specific optimization.