D2.5.2 – Fault-Tolerance and Operating Systems, Programming Models and Integration

WP2: Underpinning and cross-cutting technologies

Project Acronym	CRESTA
Project Title	Collaborative Research Into Exascale Systemware, Tools and Applications
Project Number	287703
Instrument	Collaborative project
Thematic Priority	ICT-2011.9.13 Exascale computing, software and simulation

Due date:	M30
Submission date:	31/03/2014
Project start date:	01/10/2011
Project duration:	39 months
Deliverable lead organization	CRAY UK
Version:	1.0
Status	Final
Author(s):	Stephen Sachs (CRAY UK), Pekka Manninen (CRAY UK), Lorna Smith (UEDIN), Daniel Holmes (UEDIN)
Reviewer(s)	Fang Chen (DLR), Frederic Margoules (CRSA)

Dissemination level	
PU	*PU - Public*

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	29/01/2014	First version of the deliverable	Stephen Sachs (CRAY UK), Pekka Manninen (CRAY UK)
0.2	11/02/2014	Added section 1 and 5.1	Stephen Sachs (CRAY UK)
0.3	17/02/2014	Added section 5.2	Stephen Sachs (CRAY UK)
0.4	17/02/2014	Clean-up work	Stephen Sachs (CRAY UK)
0.5	27/02/2014	Added section 4	Lorna Smith (UEDIN), Daniel Holmes (UEDIN)
0.6	28/02/2014	Added section 3.2 and 5.3, Applied comments from proofreading	Stephen Sachs (CRAY UK)
0.7	24/03/2014	Addressed reviewer comments, added 2.2 and section 6	Stephen Sachs (CRAY UK) Pekka Manninen (CRAY UK)
0.8	26/03/2014	Rearranged sections 5 and 6	Stephen Sachs (CRAY UK), Pekka Manninen (CRAY UK)
1.0	27/03/2014	Final version of the deliverable	Stephen Sachs (CRAY UK), Pekka Manninen (CRAY UK)

Table of Contents

1 EXECUTIVE SUMMARY 1

2 INTRODUCTION 2

2.1 PURPOSE 2
2.2 FOCUS OF THIS WORK AND RELATION TO OTHER CRESTA DELIVERABLES 2
2.3 RELEVANCE TO CRESTA 2
2.4 GLOSSARY OF ACRONYMS 3

3 FAULT-TOLERANT PROGRAMMING MODELS 4

3.1 STRATEGIES FOR MASKING HARDWARE FAULTS 4
3.2 FAULT-TOLERANT VERSION OF MESSAGE-PASSING INTERFACE 6
3.3 FAULT-TOLERANT FEATURES IN FORTRAN COARRAYS 6

4 OPERATING SYSTEM REQUIREMENTS 8

5 APPLICATION CASE STUDIES 10

5.1 ASYNCHRONOUS SCHWARZ METHODS 10
5.2 PRE-PROCESSING STEERING INTERFACE 11
5.3 HIMENO 11

6 IMPLEMENTING FAULT-TOLERANCE: A BRIEF HOW-TO 12

6.1 ERROR HANDLER 12
6.2 APPLICATION PROGRAMMING INTERFACE 12
6.3 IMPLEMENTATION DETAILS 14

7 REFERENCES 16

Index of Figures

Figure 1: Core failure in asyncronous mode 10

Figure 2: Core failure in synchronous mode 10

1 Executive Summary

By raising computational performance through increased parallelism, single core failures in modern supercomputers have become a more important and more expensive issue. This is due to the fact that with core count the number of overall components within a system rises. Thus, as the mean time between failures of every single component does not grow as fast as the number of components in a supercomputer, the overall mean time between failures of the system shrinks.

Failure prevention is used in many parts of the machine; the standard Message Passing Interface (MPI) mechanism of dealing with faults is to abort the entire computation if any of its ranks encounters a failure. The traditional handling of these failures is using checkpoint/restart techniques. However, as the overhead of these implementations grows with core count, their application diverts to inefficiency.

Fault tolerant parallel distributed memory models enable the code to recover from failures and continue execution although some parts of the system have been lost indefinitely. Although not yet part of the MPI standard, there is an active working group around Fault-Tolerant (FT) MPI. The work presented here assesses different developments of fault tolerant parallel execution models and shows the obstacles that have to be resolved in order to be applicable for user codes.

Furthermore current trends in different distributed memory approaches are presented and a classification of error treatment in different system levels is given. The responsibilities of operating systems, communication runtime environments, communication libraries and application environments are specified. The presented work focusses on the operating systems and communication libraries with special focus on the applicability to the CRESTA co-design applications.

2 Introduction

2.1 Purpose

The future roadmap of any High Performance Computing (HPC) hardware supplier shows that the high amount of future computing power will only be realisable by using vast numbers of compute units to work in parallel. The production quality of these units, as well as their respective auxiliary parts, will increase as production processes improve. However, the number of parts in a supercomputer will rise faster than the Mean Time Between Failures (MTBF) of every mounted part, resulting in a decline of the overall MTBF of the entire machine. A study [1] of 22 systems at Los Alamos National Laboratory (LANL) over a time span of 9 years has shown, that the average MTBF of some current systems already is less than 8 hours compared to application run times of often days or weeks. This of course is an extreme comparison and only perfectly valid if the whole machine is used by one application. Nevertheless, the decline of MTBF will be a problem for future large scale machines. Contrary to this trend are the efforts by hardware, firmware and software vendors to detect and act upon faults as soon as they appear. This behaviour has found wide acceptance in various parts of the computer industry, but there has never been a widely accepted standard for fault-aware parallel computing.

This document assesses the current developments in fault-tolerant (FT) distributed computing with respect to the CRESTA co-design vehicles and with regard to realisation on Exascale machines. Furthermore it contains a comparison of the applicability of fault-tolerant MPI and its alternative parallel programming models as well as implementation examples and a best practice guide for fault-tolerant parallel code development.

2.2 Focus of this work and relation to other CRESTA deliverables

This report discusses strategies for masking hardware failures from the point-of-view of programming models as well as of operating systems. Furthermore, a special attention is being given to the needs of applications and applicability to real world problems, and hereby a quite pragmatic approach is being taken. As such, this work does not give a complete overview on fault tolerance in parallel computing but can be regarded as a guideline for users to integrate fault-tolerant features into HPC applications.

Some themes related to fault tolerance (also studied within CRESTA) are being omitted and addressed elsewhere: power management is discussed in CRESTA deliverable (D) 2.6.3 [2]. Proactive fault tolerance will be covered in a later CRESTA D 2.5.3. Fault-tolerant capabilities on hardware and operating system level of future operating systems and micro kernels are further discussed in D2.3.1 [3]. An earlier deliverable [4] dealt with performance faults, e.g. non-fatal faults, resulting in a deteriorating application runtime and in the worst case alteration of results.

2.3 Relevance to CRESTA

A crucial feature of the CRESTA project is its integration of co-design. The co-design vehicles cover a wide range of typical HPC applications as well as prototype some of the obstacles one has to overcome to run on truly large scale. These features paired with the direct contact to the code owners and developers make them the ideal test bed for new parallel development.

As with most large scale industrial and scientific codes today the majority of the co-design applications use checkpoint to disk/restart for error handling. This has been an effective and easy to use tool in the past, but faces one problem today, which will increase on future machines. With the system size and therefore problem size growing, writing full checkpoints is becoming a substantial time effort for large-scale applications. The write and load phase of Checkpoint/Restart (C/R) is becoming too expensive as the scheduler slots (of typically 12 or 24 hours) are not growing with the gap of increasing memory volume against storage speed. Several co-design vehicles

would benefit from the proposed FT algorithms at current scale in addition to being a necessity for the targeted scale.

Additionally to the conventional use of MPI in which the overall result relies on the existence of every single result (e.g. domain decomposition) there are quite a few simpler application cases. The weather community runs their simulations in ensembles. Various ensembles are then coupled via MPI to receive statistically averaged results. This usage of MPI neither requires every single ensemble to produce a result, nor to exit without an error. However, if one of the ensembles crashes because of numerical instabilities or a system fault, the entire computation stops. This again is an application field for software resiliency, which can be solved via FT-MPI.

The rest of the document is organized as follows. We give an overview of approaches and categorisation of strategies on FT in section 3. Special attention is on the late developments in different distributed parallel programming concepts. Section 4 describes the operating system specifics for FT with focus on supporting the models described in section 3. Section 5 describes CRESTA applications which directly benefit from FT and the strategy of introducing FT with minimal code change. Section 6 shows strategies for FT insertion and example implementations of failure recovery and failure handler tools and brief implementation details are given.

2.4 Glossary of Acronyms

API	Application Programming Interface
C/R	Checkpoint/Restart
D	Deliverable
FT	Fault-Tolerant
HPC	High-Performance Computing
LANL	Los Alamos National Laboratory
MPI	Message Passing Interface
MTBF	Mean Time Between Failures
PE	Processing Element
PGAS	Partitioned Global Address-Space
uGNI	User-Level Generic Network Interface
ULFM	User Level Failure Mitigation
UPC	Unified Parallel C
RAID	Redundant Array of Independent Disks

3 Fault-tolerant programming models

Different strategies for fault-tolerant, or fault-resilient, communication are currently pursued throughout various distributed parallel programming models. Parallel programming models, or Partitioned Global Address-Space (PGAS) languages, bring along such features as shown for Coarray Fortran in section 3.3 and recently for OpenSHMEM in [2]. However, the bulk of the attention and development on FT concepts, models and realisation is focused on MPI. Since the mid 1990's MPI has been the most widely distributed state of the art distributed memory parallelisation technique. Through its popularity and portability there has been substantial interest in further developing the performance of MPI. As a matured standard it marks the interface of specially tuned high performance computing and viable industry application.

The current MPI standard (3.0, released Sep 2012) on the topic of error handling (this includes any kind of error that can arise within MPI usage) says:

"An MPI implementation cannot or may choose not to handle some errors that occur during MPI calls."

Within the standard there are two predefined options for handling faults. The default error handler is `MPI_ERRORS_ARE_FATAL`, which tells MPI to end the entire application on an occurring fault with an error message. The other choice is `MPI_ERRORS_RETURN`, which gives control back to the user, though even this *"does not necessarily allow the user to continue to use MPI after an error is detected"*. Alternatively users may implement some error handling routines, which are then invoked by MPI without guaranteeing the completion of further communication. To fill this gap a universal fault-tolerant standard for distributed computing has to be found.

3.1 Strategies for masking hardware faults

There have been multiple approaches to provide a fault-tolerant message passing system, each of which has its own characteristics. The main development can be summarised into three different layers, in which each implementation acts complementary to or independent of the other layers depending on the respective author's focus.

The first layer consists of runtime environments, which act on faults below the MPI layer. These subsystems include, for example, Harness as described in [3, 4] implementing dynamic process management and providing a distributed setup with no single point of failure. Another type of subsystems are self healing networks as described in [4] using redundant storage of contact information the network, message delivery over several nodes and automatic fault recovery. These environments, although very powerful, are subject to individual efforts which in turn will only cover a restricted user base.

The second layer consists of MPI-like implementations of fault-tolerant message passing giving the application programmer the possibility to act upon errors using specific Application Programming Interface (API) calls. Popular implementations include: FT-MPI [5], which introduces API-calls for failure detection, notification and rank recovery; LAM/MPI [6], which includes system level checkpointing and automatic roll back of applications; LA-MPI providing network FT through checksums and retransmission; MPI/FT and MPI-FT.

The third layer consists of application environments on top of MPI. These include e.g. the most widely used Checkpoint/Restart (C/R) mechanism. Diskless checkpointing as in [7] eliminates the performance bottleneck of traditional checkpointing by using local or (nonlocal) memory to store the data emulating a Redundant Array of Independent Disks (RAID) and similar techniques (on memory) for additional resiliency. Application dependent solutions as the Algorithm Based Fault Tolerance described in [8] or Redundant Communication [9] are also part of this layer. Keeping in mind that

individual applications demand individual FT strategies, the straightforward approach is to implement a third layer approach for every individual code. However, these either need some kind of support by the first or second layer subsystem or add to the required compute resources.

Within this subsystem there is only one vendor and application independent interface, namely MPI. Thus the natural and most promising approach is to integrate a minimal set of necessary features for the individual application environments into MPI. This approach does not substitute for powerful runtime environments or for individual strategies for recovery on top of MPI. It enables developers to produce independent solutions, which can benefit from other development.

For wide acceptance among users, the individual requirements of every distributed computing use case must be covered. The CRESTA co-design code owners and their feedback are employed as a broad subset of the HPC user community. A survey among the code owners was conducted. Its conclusions reveal that the core features requested by developers are basically identical. The most valuable and requested tools are the reconstruction of data that has been lost and the re-initialisation of ranks. While the re-initialisation of ranks will be discussed in section **Error! Reference source not found.**, the reconstruction of data is an application and underlying problem dependent feature. For some problems the data can be interpolated from neighbouring data in time or space, others have to roll back all ranks to the last checkpoint in order to preserve consistent data. In addition to the above, the restoration of initial number of ranks was repeatedly requested. This will especially be a challenge for job schedulers in the future, as they are requested to provide replacement hardware on very short notice. In the meantime it is perfectly valid to request a small number of spare ranks for long running large-scale ranks. The compute resources wasted by allocating unused ranks become negligible compared to re-running the entire job as the scale increases. Another feature every code requires is standardised error treatment. There has to be one generic standard error handler applicable to every occurring fault. This handler must be able to provide detailed information over the health of the application and, most important, identify the erroneous ranks. Every code owner has his individual notion of the ideal implementation for recovery. Some applications need the reconstruction of all ranks with the original topology others do not even need global knowledge of the failed ranks. Thus, an ideal implementation can only provide basic tools for the user to create the recovery to fit his needs. Some of the common requests are considered in this work, requirements from typical use cases include:

- Acknowledge the existence and extent of an error and keep on working with the remaining ranks;
- Enable the program to realise that some fault occurred in order to save the remaining data and finalise;
- Restart the entire application run when the failure is recognised while also re-running initialization;
- Isolate the running ranks in the application from the failure and replace the failed ranks by backup compute units running from the beginning (including initialisation);
- Minimize impact of FT on the application run time and programmability.

For each of these feature requests, as well as for individual demands requested by only a subset of code owners, a solution must be provided. Thus a framework has to be constructed to allow the community or the individual code owners to implement individual solutions. For example the first use case can be solved by simply adding an error handler to the user code. This can be a generic handler acting as a report tool for failures, which can be written for multiple different applications using FT in a similar fashion.

The other cases include deeper interaction with user code, i.e. having all ranks act on different states and invoke recovery procedures, or writing extensions to the applications view of the API. Frameworks situated in between the user code and MPI

can make use of more advanced usage of FT. The FT features can be seamlessly integrated into asynchronous frameworks such as [10]. If the supported framework runs truly asynchronously, even with asynchronous initialisation, the API can support this behaviour without any user code change.

3.2 Fault-tolerant version of message-passing interface

The latest and most promising approach of a standard in FT-MPI is currently being prepared for presentation by the "User Level Failure Mitigation" (ULFM) group to the MPI Forum. This approach has evolved from previous attempts in close collaboration with the Forum. In earlier forms the absence of support for e.g. fault tolerant one sided communication or I/O was justification for the rejection from the standard. In this mature form all the obstacles have been included.

This approach enables MPI programs to handle failures by mitigation the effects of broken communicators, identifying the unresponsive part of the communication topology and even restoring communication capabilities destroyed by failures. It consists of a few simple API calls which enable the programmer to create all kinds of requested FT features. This is in line with the rest of the MPI standard, which does not want to specialize on any kind of request. There are no calls in the library to solve any specific problem, but a variety of calls that provide a framework to create solutions to various specific problems. Thus the standard stays independent of vendors, application types or other interest groups.

The working group around ULFM proposed the addition of a set of MPI calls, return codes and attributes to the standard. In the following the central additions and their respective properties in the FT context is outlined.

There are three additional exceptions: `MPI_ERR_PROC_FAILED_PENDING` indicates an outstanding receive to `MPI_ANY_SOURCE` may be receiving from a failed rank, i.e. there is a failed rank within the underlying communicator. `MPI_ERR_PROC_FAILED` informs the caller that the request cannot be completed due to a failed rank. This can appear, for example, during an outstanding receive from a failed rank. If the user code has taken failure notification propagating action on the current or anther rank by marking the communicator as revoked, all non-local calls will return `MPI_ERR_REVOKED`. To identify a broken communicator there is the function `MPI_Comm_revoke()`. Calling this interface will notify all ranks on the communicator that this communicator is faulty and terminate any non-local MPI calls at all ranks. All subsequent calls to this communicator will return the error `MPI_ERR_REVOKED`. `MPI_Comm_shrink()` creates a new communicator including all surviving ranks of comm. The functionality can be described as using `MPI_Comm_split()` with all surviving ranks sing the same colour and their respective rank numbers as key. The function `MPI_Comm_failure_ack()` acknowledges errors on the communicator in contrast to marking the communicator faulty as in revoke. This includes freeing unmatched receives to `MPI_ANY_SOURCE` on the communicator in question. `MPI_Comm_failure_get_acked()` returns the group of locally acknowledged failed processes and `MPI_Comm_agree()` is a collective "and" operator among all surviving ranks of a faulty communicator. This collective as well as `MPI_Comm_shrink()` are the only functions which will never raise `MPI_ERR_REVOKED` on a revoked communicator.

A complete set including the non-blocking (e.g. `MPI_Comm_iagree()`), one sided (e.g. `MPI_Win_revoke()`) and I/O (e.g. `MPI_File_revoke()`) related additions and further information is given in [11].

3.3 Fault-tolerant features in Fortran coarrays

The PGAS programming model provides a global view of the memory across supercomputer nodes and supports a one-sided access to shared data. Examples of programming languages and paradigms employing the PGAS approach include the coarrays concept of the Fortran 2008 standard [11], the Unified Parallel C (UPC)

extension to the C language [12], a coarray class for the C++ language [13], and the Chapel [14] and X10 [15] languages.

Of PGAS approaches, Fortran coarrays and UPC are most often used in real-world HPC applications (but being still a small fraction of all parallel applications), and have most extensive support in the programming environments of today's HPC systems. Here we consider what kinds of features for more fault-tolerant programming style are being offered by Fortran coarrays. In general, the coarrays approach should support in the next Fortran standard similar fault-tolerant functionalities as the FT-MPI discussed earlier.

There is a defined and straightforward mechanism of Fortran coarrays that allows for isolating a failed Processing Element (PE, referred to as an image).The following statements: `change team`, `end team`, `form team`, `sync all`, `sync images`, `sync memory`, `sync team`, `lock`, `unlock`, `event post`, `event wait`, `allocate`, or `deallocate` are able to return a named constant `stat_failed_image`. This return value will occur if there is a failed image in the current team. . This is provided by the intrinsic module ISO_Fortran_env. In the case of `sync all`, `sync images`, or `sync team`, the statement will have successfully synchronized all the images of the specified set that have not failed [16]. With using the team construct, for example as

```
if (num_images(failed=.true.) > 0 ) then
   form team(1, recover)
   change team (recover)
     ! Keep on working with the images/PEs still functioning
     ! ...
   end team
end if
```

would allow for the non-failed images to continue while the failed PEs – due to a node failure, for instance - are being excluded. This could be checked periodically. Having a benefit from this requires an algorithm that is insensitive and/or able to recover from the data loss of the failed image(s).

One option for a program able to deal with node failures, again based on teams, is to run a fully or partially redundant calculation. The program checks whether any images have been failed. If there are failed images, the program checks whether its replica image is still active. If it is, a new team will be formed by excluding the failed image and including its still running replica. If also the replica has failed (or there were none in case of partially redundant calculation), only then the program is terminated.

In case of a node failure, all data of an image, including its share of coarray data, will be lost. However, if an image is able to do a clean exit using the STOP statement prior to failure, the coarray data can still be retrieved from the image. The stopped images can be detected similarly with failed images with the stat specifier value of `stat_stopped_image`. This can enable pre-emptive measures, that is, probing for the health status of the platform, and in case of any suspicious signals stopping an image and retrieving its data before the node failure occurs. This could be applied also to a case when an image is behind a slow link that harms the overall performance of the application.

We note that neither the teams nor the `stat_failed_image` specifier values are part of the current Fortran 2008 standard but are a part of a Technical Specification draft that prepares the parallel processing enhancements for the next Fortran standard.

4 Operating system requirements

The distributed parallel programming model fault tolerant strategies described above will not exist in isolation, to work these will require interaction with the operating system. The most likely faults that will need to be tolerated at the exascale are, generally, *hardware* faults: for example compute-node failures or communication-link failures. Handling the fault, mitigating its effect, determining if recovery is possible, performing recovery and continuing execution are likely to be the responsibility of the application / programming model layer, rather than of the operating system. The application has more knowledge about the effect of each fault, and the cumulative effect of many faults, than the operating system and so is better placed to determine whether those effects can be mitigated and whether recovery is possible. However the operating system provides the interface between computer hardware and software applications and therefore must have a role in the initial detection of the fault.

This may be seen as a departure from established methods of fault-tolerance, such as error-correction in ECC memory chips (which is a hardware function) or reliable-delivery in TCP socket communications (which is a system-software function that may be offloaded to hardware). In these situations, sufficient information to correct the error is available to the hardware or system-software that is attempting to perform the task. For ECC memory, extra information is calculated and stored during each write operation and is used to detect, and correct, errors during subsequent read operations. For TCP communications, the data for packets to be sent kept until the sequence number of that packet is acknowledged by the receiver. If a gap in sequence numbers is detected then the sender re-sends the missing packet(s).

However, both of these established fault-tolerance methods are developments of earlier approaches that required software applications to explicitly handle, mitigate and recover from errors. These non-fault-tolerant methods still exist and are used when error-correction is not a high priority.

The current proposal for fault-tolerance being considered by the MPI Forum divides the problem into four areas: detection, notification, propagation and consensus. All but the first of these are deemed to be the responsibility of the MPI library. Detection of faults is too system-specific to be standardised and recovery requires application-specific decision logic that cannot be incorporated into a standardised communication library. Notification is the process by which the local MPI rank in the application is informed about a fault that has been detected by the system-specific fault detection mechanism. Propagation is the process of disseminating knowledge of a fault from the local MPI rank to some, or all, of the other MPI ranks that are still executing normally. Consensus is the process of deciding on a common value amongst many MPI ranks in a way that tolerates faults occurring during the decision itself. These functions within MPI, when combined with a suitable fault-detection mechanism that interacts with MPI, are sufficient for an application to construct a wide variety of recovery algorithms.

Operating systems already detect faults. When the hardware that the operating system controls does not perform a function as expected, the operating system usually detects the abnormal behaviour and may attempt recovery (e.g. by re-trying the function) or may raise an error to the application that requested the function. Some types of fault are not currently detectable by the operating system. Some types of fault are theoretically detectable by the operating system but are not currently commonly detected. For example, a core becoming unresponsive may be silently handled by the scheduling algorithm in the operating system, because each core executes a scheduler that simply chooses one of the runnable threads available for that core, allows it to execute for a short amount of time, interrupts that thread and chooses another runnable thread for that core. There is likely to be no mechanism in the operating system for the cores to be monitored or tested during execution and so, whilst theoretically an unresponsive or stopped core could be detected, this type of fault is unlikely to be discovered.

Fault detection could be performed by dedicated hardware that monitors the health of particular parts of the system. Several vendors produce systems that contain this type of hardware. Typically they provide system administrators with a management interface that displays information about the current state of the system and affords some control independently of the computational hardware. For example, this may allow a failed node to be re-started remotely even if it is otherwise unresponsive. For this to be used for fault detection without manual intervention, an interface must be provided in the form of system-software or an extension to the current functionality of operating systems.

It is becoming common-practice to implement software libraries, in particular MPI libraries, using OS-bypass techniques. Bypassing the operating system permits user code to control hardware directly, without any assistance from the operating system. This is generally done in an attempt to increase the performance of the software by making use of the hardware in a very specific (and restricted) manner rather than exploiting the full breadth of its capability. However, an unintended consequence of bypassing the operating system is that any fault detection it would normally provide is also bypassed. In this situation, the user code must assume the responsibility for fault detection.

In summary, the role of operating systems in fault tolerance is primarily that of fault detection. Some faults are already detected by current operating systems. Extensions to existing operating systems could enable more types of fault to be detected. Bypassing the operating system to achieve performance may have a negative impact on fault detection and, therefore, on fault-tolerance.

5 Application case studies

Multiple small test cases have been developed to test the functionality of the ULFM implementation described in [11]. During the course of this work a few bugs were found in the implementation and, in close collaboration with the developers, valuable feedback was passed in both directions and lessons in FT implementation were learned. This section will give a short overview of the applications and FT strategies applied to them. The codes listed below constitute an incomplete list of feasible co-design vehicles at the time of writing. There are further applications within CRESTA with potential benefit of FT whose applicability has not been tracked yet.

5.1 Asynchronous Schwarz Methods

D2.5.1 [19] describes an asynchronous algorithm for sparse linear algebra which is robust enough to cope with slow components. The asynchronous approach is an ideal candidate to be expanded to include handling complete failure of components. If any part fails, the API implementation of FT will introduce replacement ranks for the lost workers which will re-initialize individually and detached and finally commence communication with the surviving ranks.

At application start-up a few additional cores are started and put to sleep in `MPI_Init()`. If an active core detects a fault in the communication, the recovery mechanism is started. This can be initiated from within any MPI call. The recovery mechanism detects the erroneous ranks and replaces them with sleeping ones. Then the revoked communicators are rebuild and the replacement ranks exit `MPI_Init()` and run through the ordinary initialisation phase. There is no need for the replacement ranks to read from checkpoint files, as the information of the prior iterations is stored in the boundaries of its neighbours. Due to the nature of the asynchronous algorithm, the replacement ranks will catch up with their neighbours after a few iterations.

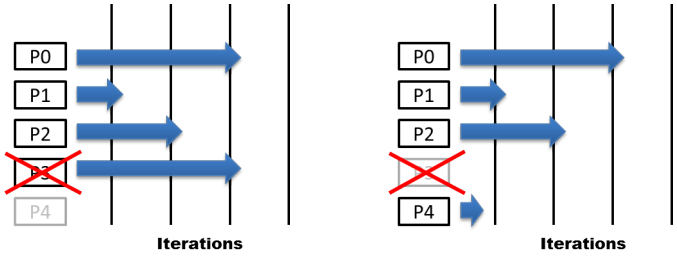

Figure 1: Core failure in asyncronous mode

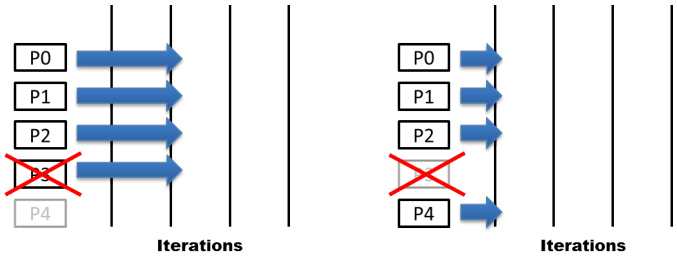

Figure 2: Core failure in synchronous mode

Figure 1 shows the working principle of asynchronous FT distributed computation. After a failure of processor P3, P4 will take over P3s responsibilities and start its iterations from the beginning. All other processes are unaffected. Figure 2 shows the synchronous principle including classical C/R. If P3 fails in this case all ranks rewind to the latest checkpoint, which is at least one iteration behind.

As there are no global synchronization points, the expansion to FT can be applied without alteration of the source code. The entire FT functionality can be hidden in the

MPI calls. This separation of the algorithm and the FT layer is particularly useful for an evolving new standard which is subject to modifications.

5.2 Pre-processing Steering Interface

Another application investigated within CRESTA is the application steering interface PPSTee, which dynamically distributes application load based on reported run times. This iterative process, which has multiple domain decomposition tools available, yields an optimal load distribution. The application is introduces in D5.1.3 [19].

For this application the simplest mode of FT can be applied – acknowledge failures, including actions to prevent further usage of the failed resources, and move on. In case of a failure the interface has to acknowledge the missing ranks, redistribute the work among the remaining ranks and restart the interrupted iteration. The communicator can be shrunken to the remaining ranks. Within PPSTee the failed rank will be assigned a proportional of zero, thus no domain will be assigned to it.

This example illustrates the simplicity of including FT into this application governor. Generally, effort should be spend to include FT at the most abstract level possible.

5.3 Himeno

Himeno [20] is not an official CRESTA application but a simplified stencil solver. It acts as a surrogate for NEK5000, OpenFOAM and similar numerical partial differential equations solvers capturing their basic functionality. The benchmark consists of a small code base and the source code of the C and Fortran version is available online. Multiple modes of FT have been tested with this application and both a C and a Fortran interface were used.

Working on this minimal solver has exposed the importance of fixing the FT strategy prior to starting the implementation. A simple acknowledgement of a failure and shrink up to a full recovery of all working ranks can be implemented, however the different strategies do not necessarily build up on each other. Further details on including FT are given in the next section.

6 Implementing Fault-Tolerance: A brief How-to

We cannot assume that we will receive a report of a hardware or software fault affecting a distributed computation from the failing hardware or software components. In the case of a power outage or a network failure there may be no sign of degradation until the communication stops. Thus the errors in FT-MPI are reported by the ranks trying to communicate with the failed rank. As the scope of communication in MPI is defined by the respective communicator used, it is a natural choice to associate errors with communicators. An FT-MPI implementation based on Open MPI and associating errors with communicators is the User Level Failure Mitigation implementation (fault-tolerance.org) [17]. In the context of CRESTA this Open MPI implementation has been adapted to support the Application Level Placement Scheduler (ALPS) and native User-Level Generic Network Interface (uGNI) on a Cray XE6 as well as generic clusters. In the following a set of exemplary use cases for this implementation and minimal code fragments for the realisation are given.

6.1 Error handler

The simplest case is an application that does not rely on single ranks. In this case every rank can fail, as long as there is at least one rank finishing the run. This can be achieved by activating the error handler `MPI_ERRORS_RETURN`. In contrast to the current MPI standard the state of FT-MPI should be well defined after an error is returned. This however restricts the available communication after faults. To add a little flexibility, an undisturbed communication should be re-established after a failure occurs. Acknowledging an error and keeping on working with the remaining ranks can be implemented by writing an MPI error handler. Error handlers are opaque objects associated with a user defined function. As soon as MPI encounters an error on a communicator, the currently active error handler is invoked. A minimal example of an error handler for FT-MPI is the following:

```
      integer comm, error_code, rc

      while (MPI_ERR_PROC_FAILED.eq.error_code
     &       .or. MPI_ERR_COMM_REVOKE.eq.error_code) do
          call MPI_Comm_failure_ack(comm, rc)
          call MPI_Comm_revoke(comm, rc)
          call MPI_Comm_shrink(comm, new_comm, rc)
          call MPI_Comm_agree(new_comm, rc, error_code)
          call MPI_Comm_free(comm, rc)
          comm = new_comm
      enddo
```

`MPI_Comm_failure_ack()` allows unmatched receives from `MPI_ANY_SOURCE` to continue, `MPI_Comm_revoke()` is invoked to complete all global communications on the current communicator and `MPI_Comm_shrink()` is called to build a new communicator consisting of all surviving ranks in the current communicator. The check for `MPI_ERR_PROC_FAILED` and `MPI_ERR_COMM_REVOKE` insure there is neither a failed processor in the communicator nor have any of the participating ranks spotted an error. `MPI_Comm_shrink()` as well as `MPI_Comm_agree()` are collective calls that return a defined state in case of a defect communicator. Activating an error handler with this function on all communicators used in this application will result in uninterrupted communication between surviving ranks even after failures have occurred.

6.2 Application Programming Interface

A convenient scheme to add functionality to MPI calls is to utilise the profiling interface. For every `MPI_...()` call there is also a `PMPI_...()` call, which enables a developer to easily implement functions extending the mode of operation of any MPI call. For

example the Fortran call to MPI_Init() can be overwritten by a simple reference to PMPI_Init():

```
subroutine MPI_Init(ier)
      implicit none
      include “mpif.h”
      integer ier
      call PMPI_Init(ier)
end
```

Any functionality can be added within the subroutine. Using this feature, a developer can extend the FT capabilities of his personal MPI interface by what is needed is for a specific application. As job scheduler configurations on current production machines usually do not support dynamically adding cores to a running job, a few spare cores will be allocated and included into the job. This is realised by extending MPI_Init(). One possibility is to have these spare cores enter MPI_Init() and then be send to a sleep state until needed. The MPI_COMM_WORLD is then also replaced in MPI_Init() by what the application would expect, i.e. a communicator including all ranks active in the application without the spare ranks in sleep state.

Following this approach the MPI calls can be extended to include most of the fault mitigation and recovery actions, yielding minimal alteration in user code to make use of FT-MPI. For example the error handler can be set to MPI_ERRORS_RETURN and a global colour vector containing the state and availability of every rank can be distributed among the ranks with in MPI_Init(). This colour vector holds an updated list of all ranks indicating if the rank is alive, asleep or dead. Every MPI call in the application can be intercepted as shown above in order to check for, and in some cases replace, the communicator or add additional assertions to insure proper execution. Furthermore, the return value can be checked for its error code and if one of the FT error codes is encountered the appropriate actions are taken. These actions include:

- The current communicator is revoked to complete all global communication;
- The failure is acknowledged in order for receives from MPI_ANY_SOURCE to complete;
- All communicators are shrunk including the actual world communicator that holds both active and sleeping ranks, and the communicator send to the application to mimic MPI_COMM_WORLD.
- The colour vector is updated by elimination of the missing ranks and actualisation of the surviving ranks numbers.

These actions will ensure the stability and integrity of the surviving part of the application. If the same procedure is executed in all MPI calls, there is no need for the application to use a global synchronisation to check for faults. The information of a fault is propagated by revoke and acknowledge and synchronisation of all ranks is ensured at the call to the global function MPI_Comm_shrink().

Thereupon either the application can continue with a reduced number of ranks or the reconstruction of original number of ranks takes place. A strategy for reconstruction within the API calls is outlined in the following.

- The group of active and alive ranks determines a leader for the reconstruction process. This can be for example a global minimum over the rank numbers. Determining a leader each time a fault occurs ensures that there is no single point of failure.
- The leader wakes the number of sleeping ranks required for rebuilding the original communicator.
- The shrunken communicators are merged with MPI_COMM_SELF from the woken ranks and the ranks in the resulting communicator are reordered according to the original communicator.

A basic implementation of this rebuilding mechanism can be implemented by sending all participating ranks back to `MPI_Init()` (this includes some interference in user code). As the sleeping ranks are waiting within `MPI_Init()` the surviving ranks can "pick them up" as they go through `MPI_Init()` once more. A little more sophisticated approaches include a separate initialisation for the replacement ranks without having the entire application roll back to the beginning. This can be implemented with minimal user code interaction if the user code does not use global communication during setup phase.

6.3 Implementation Details

These guidelines are based on the current scope of the FT-MPI extensions described in [11]. As these extensions have not been passed by the MPI Forum at the time of writing, there may be a slight variation of the preliminary interface used here and the one being used in future MPI implementations.

Generally a user should check for a minimal set recovery actions required for successful completion while omitting everything dispensable. It is also highly advisable to hide as much functionality as possible inside the MPI API calls or another separate layer. When adding FT capabilities to a framework special care has to be taken to not affect ease of implementation and keep the necessary overhead as low as possible.

Making FT available for an existing MPI application can be split up into a few separate partly dependent fractions, which will be briefly described. These hints must be taken with caution and critical analysis should be the basis of adopting these recommendations.

- *Initialisation:* During `MPI_Init()` the active and replacement ranks are separated. After that the communicator which will be known to the application as `MPI_COMM_WORLD` is constructed containing only the active ranks. The separation can be controlled by environment variables or command line options as these are passed into `MPI_Init()`.
- *Revoke, acknowledge and shrink:* The failures are propagated to the other ranks in the communicator by calling `MPI_Comm_revoke()`; then `MPI_Comm_failure_ack()` marks all the failed processes currently known to the local rank. Finally `MPI_Comm_failure_get_acked()` returns the group of failed ranks.
- *Finding and translating group of failed ranks:* The group of failed ranks from `MPI_Comm_failure_get_acked()` is translated to the recently revoked communicator. This is executed by creating a group of ranks from the revoked communicator and calling `MPI_Group_translate_ranks()` with the group of failed ranks, a vector containing the numbers 0 through the size of the group of failed ranks and the group of ranks in the revoked communicator:

  ```
  MPI_Group_size(failed_group, &failed_group_size);
  for (i = 0; i < failed_group_size; ++i)
        failed_group_rank[i] = i;
  for ( ; i < ft_comm_size; ++i)
        failed_group_rank[i] = MPI_PROC_NULL;
  MPI_Group_translate_ranks(failed_group,ft_comm_size,
        failed_group_rank, revoked_group, revoked_group_rank);
  ```

  These are all local operations, collective operations would return an error in any case as `MPI_Comm_revoke()` has been called.
- *Determining a leader:* To ensure no single point of failure every rank has to be able to lead the recreation process. This rank is called the leader. In the reference implementation the leader is determined by calling `MPI_Allreduce()` to determine the global minimum of rank numbers on what the application sees as `MPI_COMM_WORLD`. This ensures that a healthy and currently active rank is used to introduce new ranks to the computation.

- *Wake up sleeping rank:* The leader sends the rank number it is supposed to replace as well as the number of replacement ranks to come to the sleeping rank. This enables the sleeping rank to call `MPI_Intercomm_create()` and `MPI_Intercomm_merge()` to join the surviving ranks in one intracommunicator and subsequent `MPI_Comm_split()` to reorder itself into the position of the rank it is replacing.
- *Finalization:* During `MPI_Finalize()` a leader is once more assigned to wake all remaining sleeping ranks in order to exit in an orderly fashion. The leader sends them -1 as the rank to replace, which sends them to `MPI_Finalize()` as well.

When adding functionality to MPI calls care has to be taken to only use the respective PMPI calls within the APIs – this has been omitted where not explicitly needed for readability within this deliverable. A typical example of overwriting an MPI API function in an implementation including the measures described above would for example be:

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)
{
  int ierr;

  if (MPI_COMM_WORLD == comm) {
    ierr = PMPI_Recv(buf,count,datatype,source,tag,app_world,status);
  } else {
    ierr = PMPI_Recv(buf,count,datatype,source,tag,comm,status);
  }

  if (MPI_ERR_IN_STATUS == ierr) {
    ierr = (*status).MPI_ERROR;
  }

  if (MPI_ERR_PROC_FAILED == ierr || MPI_ERR_REVOKED == ierr) {
    revoke_ack_shrink(app_comm_world);
    revoke_ack_shrink(actual_comm_world);
    recreate_comm();
    failure_detected = 1;
  } else if (MPI_SUCCESS != ierr) {
    printf("MPI_Error in recv: %d\n", ierr);
  }

  return ierr;
}
```

Although the implementation is not mature enough to be used in production, the lessons learned in building FT support are valuable. Owing to the wide range of applications and usage modes of FT, the provided interface from MPI vendors has to be very generic. This in turn leaves the implementer a lot of space to find the optimal solution for the considered application. The lessons learned generally apply to new developments which are written with FT support from scratch as well, although a lot of effort in “hiding FT from the application” can be omitted.

7 References

[1] B. Schroeder and G. A. Gibson, "A large scale study of failures in high-performance-computing systems," in *International Symposium on Dependable Systems and Networks* , 2006.

[2] A. Hart, M. Weiland, D. Khabi and J. Doleschal, "D2.6.3 - Power measurement across algorithms," 2014.

[3] D. Holmes, "D2.3.1 - Operating systems at the extreme scale," 2013.

[4] M. Bull and J. Nowell, "D2.5.1 - Fault agnostic and asynchronous algorithms at exascale," 2013.

[5] S. Poole, P. Shamis, A. Welch, S. Pophale, M. G. Venkata, O. Hernandez, G. Koenig, T. Curtis and C.-H. Hsu, "OpenSHMEM Extensions and a Vision for its Future Direction," in *OpenSHMEM* , 2014.

[6] G. E. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, J. Pjesivac-Grbovic and J. J. Dongarra, "Process Fault-Tolerance: Semantics, Design and Applications for High Performance Computing," *International Journal for High Performance Applications and Supercomputing,* 2004.

[7] M. e. a. Beck, "HARNESS: A next generation distributed virtual machine," *Future Generation Computer Systems,* vol. 15, no. 5, pp. 571-582, 1999.

[8] T. Angskun, G. Fagg, G. Bosilca, J. Pjesivac-Grbovic and J. Dongarra, "Self-healing network for scalable fault-tolerant runtime environments," *Future Generation Computer Systems,* vol. 26, pp. 479-485, 2009.

[9] S. Sriram, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargroveand and E. Roman, "The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing," in *LACSI*, Santa Fe, NM, 2003.

[10] J. S. Plank, K. Li and M. A. Puening, "Diskless checkpointing," *Journal IEEE Transactions on Parallel and Distribted Systems,* vol. 9, no. 10, pp. 972-986, 1998.

[11] P. Du, A. Bouteiller, G. Bosilca, T. Herault and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations," in *17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012.

[12] N. Ali, S. Krishnamoorthy, N. Govind and B. Palmer, "A Redundant Communication Approach to Scalable Fault Tolerance in PGAS Programming Models," in *19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2011.

[13] F. Magoulès, "Asynchronous Optimized Schwarz Methods," in *Computational Methods for Engineering Science*, 2012, pp. 425-443.

[14] U. L. F. M. Group. [Online]. Available: http://fault-tolerance.org/ulfm/ulfm-specification/.

[15] Fortran Standard: ISO/IEC 1539-1:2010, 2010.

[16] UPC Language Specifications, v.1.2, LBNL-59208, 2005.

[17] T. Johnson, "Coarray C++," in *7th International Conference on PGAS Programming Models*, 2013.

[18] B. L. Chamberlain, D. Callahan and H. P. Zima, "Parallel Programmability and the Chapel Language," *International Journal of High Performance Computing*

Applications, vol. 21, pp. 291-312, 2007.

[19] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu and D. Grove, "X10 Language Specification Version 2.2," 2012.

[20] J. Reid, "Additional coarray features in Fortran," in *7th International Conference on PGAS Programming Models*, 2013.

[21] W. Bland, A. Bouteiller, T. Herault, G. Bosilca and J. Dongarra, "Post-failure recovery of MPI communication capability: Design and rationale," *International Journal of High Performance Computing Applications,* vol. 27, pp. 244-254, 2013.

[22] G. Matura, "D5.1.3 - Pre-processing: first prototype tools for exascale mesh partitioning and mesh analysis," 2013.

[23] R. Himeno, "Himeno Benchmark," RIKEN, [Online]. Available: http://accc.riken.jp/2444.htm. [Accessed 2014].

[24] B. L. Chamberlain, D. Callahan and H. Zima, "Parallel Programmability and the Chapel Language," *International Journal of High Performance Computing Applications,* vol. 21, pp. 291-312, 2007.