Fenix, A Portable, Flexible Fault Tolerance Programming Framework for MPI Applications

Rob F. Van Der Wijngaart*, Marc Gamell†, Keita Teranishi‡, Eric Valenzuela∗, Michael A. Heroux†, Manish Parashar†

*Intel Corporation, Santa Clara, CA, USA
rob.f.van.der.wijngaart@intel.com
†Rutgers Discovery Informatics Institute, Rutgers University, Piscataway, NJ, USA
{mgamell, parashar}@cac.rutgers.edu
‡Sandia National Laboratories, Livermore, CA, Albuquerque, NM, USA,
{knteran, maherou}@sandia.gov
∗California State University, Channel Islands, CA, USA,
{evalen}@sandia.gov

Categories and Subject Descriptors
1 [System Software]: Support for fault-tolerance and resilience; 2 [Performance]: Resilience

Keywords
MPI, Fault Tolerance, Library, ULFM

1. INTRODUCTION

One of the pillars of extreme-scale high performance computing is fault tolerance: the ability of applications to recover from failures efficiently. Traditional checkpoint/restart-based techniques face scaling hurdles due to decreasing mean time between failures, increasing cost of checkpointing, and increasing application relaunch time. Fenix is a framework enabling MPI applications to recover nearly transparently from losses of data and/or compute resources manifested as observable errors. It is based on the premise that the MPI standard provides facilities for trapping and isolating such errors, allowing the application to retain control of remaining unaffected resources, obviating full program restart.

Building on the success of Fenix’ prototype implementation [3], we describe a significant extension, captured in a formal specification [4], to serve many different application needs and styles. The current implementation of Fenix leverages User Level Fault Mitigation (ULFM), but the specification does not require it. The ULFM MPI fault tolerance proposal [1] is among the most promising facilities considered for inclusion in the MPI standard. It provides a small set of functions to support MPI fault tolerance, but is deemed too cumbersome to be used by application scientists directly [6].

This paper is structured as follows. In Section 2 we summarize the Fenix specification, highlighting its salient features. We also describe various important high-performance computing patterns and fault recovery scenarios supported by Fenix. In Section 3 we describe alternative and complementary approaches to providing MPI fault tolerance.

2. FENIX SPECIFICATION

Fenix has two distinct interfaces: process recovery, and data recovery. The first allows an MPI application to recover from a permanent loss of MPI processes (ranks) that cause MPI calls to fail. This is the most important and novel part of Fenix. Fenix’ data recovery API could be replaced by, or used with, other mechanisms to restore application data.

2.1 Process recovery

Fenix’ basic assumption is that nominally fatal errors in MPI programs are detected by the runtime and reported via error codes. While default error response is application shutdown (MPI_ERRORS_ARE_FATAL), Fenix overrides this, allowing remaining resources (MPI processes) to be informed of the failure, and to call MPI functions to remove the lost resources from their respective communication contexts [3].

To ease adoption of MPI fault tolerance, Fenix automatically captures errors resulting from MPI library calls that experience a failure. Implementations of the Fenix specification can achieve this behavior leveraging error handlers or the MPI profiling interface. Hence, Fenix users need not replace MPI calls with calls to Fenix (for example, Fenix_Send instead of MPI_Send). Subsequently, Fenix repairs communicators transparently and returns execution control to the application. Its detailed behavior is determined by function Fenix_Init, which also initializes the library.

```c
void Fenix_Init(
    MPI_Comm comm,
    MPI_Comm *newcomm,
    int *role,
    int *argc, char ***argv,
    int spare_ranks,
    int spawn,
    MPI_Info info,
    int *error);
```
Communicator `comm` (includes any slack) is used to create resilient communicator `newcomm`. Communicators derived from `newcomm` are automatically resilient. Errors returned by MPI calls involving resilient communicators are intercepted by Fenix, triggering repair of all resilient communicators, such that the application can resume execution. Control is returned to the application at the logical exit of `Fenix_Init`.

Parameter `role` contains the calling rank's most recent history. Possible values are `FENIX_ROLE_X_RANK`, with `X` being `INITIAL`, `SURVIVOR`, or `RECOVERED`, corresponding to “no errors yet”, “not affected by latest failure”, and “rank recovered, but has no useful application data”, respectively.

Parameter `spare_ranks` specifies how many ranks in `comm` are sequestered in `Fenix_Init` to replace failed ranks, and `spawn` determines whether Fenix may create new ranks (`MPI_Comm_spawn`) to restore resilient communicators to their original size. `spare_ranks` and `spawn` together define Fenix' overall communicator repair policy. If both are zero, resilient communicators are shrunk to exclude failed ranks after an error. If only `spawn` is zero, Fenix draws on the spare ranks pool to restore resilient communicators to their original size. Once the pool is depleted, subsequent errors lead to communicator shrinkage. Parameter `info` conveys details about expected Fenix behavior that differs from the default.

### 2.2 Data recovery

Once Fenix process recovery has returned an application to a consistent state, the user needs to consider lost data. Sometimes no action is required, for example because the application is embarrassingly parallel Monte Carlo. However, in most HPC applications, Fenix' prime target, ranks synchronize often, and intermediate state of ranks lost due to error must be restored. We outline several plausible approaches, all supported by Fenix' process and data recovery facilities. However, the user need not use Fenix for data recovery, and may, for example, use Global View Resilience \(2\), Scalable Checkpoint Restart \(7\), a combination of these and Fenix, etc. Fenix aims mostly at providing fast, in-memory synchronization of logically “nearby” data. This approach, also good for bulk-synchronous codes, may apply to relaxation methods.

3. **Shrinking recovery with full data retrieval.** If the user demands online recovery and resources are insufficient to replace defunct ranks, Fenix shrinks damaged communicators. Now there is no simple, unique way to assign recovered data to the reduced number of remaining ranks. More general, flexible Fenix data recovery functions are provided for alternate ways of retrieving and re-assigning such data.

4. **Shrinking recovery with local data retrieval only.** This is a combination of methods \(2\) and \(3\).

To organize redundant storage for data recovery after a fault, Fenix offers *data groups*, containers for sets of data objects (members) that are manipulated as a unit. Data groups also refer to the collection of ranks that cooperate in handling recovery data. This collection need not include all active ranks. Fenix adopts the convenient MPI vehicle of *communicators* to indicate the subset of ranks involved.

A data group is instantiated with the following function:

```c
int Fenix_Data_group_create(
    int group_id,
    MPI_Comm comm,
    int start_time_stamp,
    int depth);
```

It is called by all ranks in `comm` with identical parameter values—defining the group for all of them. The label `group_id` can be used to restore the group after a failure. Parameter `start_time_stamp` initializes a counter to identify versions of the recovery data associated with the data group, and `depth` is the number of successive versions of such data sets maintained by Fenix in addition to the latest.

Once a data group has been created, the user can define its members, which describe the actual application data:

```c
int Fenix_Data_member_create(
    int group_id,
    int member_id,
    void *source_buffer,
    int count,
    MPI_Datatype datatype);
```

The `member_id` distinguishes between different group members, `source_buffer` is the address of the contiguous application data in memory, and `count` (may be different for different ranks) and `datatype` together fix the data’s extent.

Once all group members have been defined, the user can invoke Fenix functions to store application data:

```c
int Fenix_Data_member_store(
    int group_id,
    int member_id,
    Fenix_Data_subset subset_specifier);
```

Set `member_id` to `FENIX_DATA_MEMBER_ALL` to store all group members. Parameter `subset_specifier` controls selective storage. Currently Fenix stores two copies of each member; one in the memory of the calling rank, the other in a peer rank’s in `comm`. Upon failure `SURVIVOR` restore their application data using the local copy, and `RECOVERED` ranks fetch it from their peer. This technique, inspired by Charm++ \(8\), provides resilience through redundancy.

1. **Non-shrinking recovery with full data retrieval.** This is the most common case in bulk-synchronous HPC codes. The programmer defines data/work decomposition that corresponds to a certain number of ranks. After an error, SURVIVORS roll back their state to a prior time. Missing ranks are replaced with `RECOVERED` ranks who instantiate their state using non-local data from their peer. This technique, inspired by Charm++ \(8\), also good for bulk-synchronous codes, may apply to relaxation methods.

2. **Non-shrinking recovery with local data retrieval only.** SURVIVORS roll back their state, but `RECOVERED` ranks approximate their requisite data, for example by interpolation of logically “nearby” data. This approach,
To mark a version of the group’s data as recoverable, it needs to be committed; Fenix labels the stored data with a time stamp (a snapshot). Subsequent store operations of the same group do not contribute to the same snapshot, and receive an incremented time stamp upon the next commit.

\[
\text{int Fenix\_Data\_commit(}
\text{  int group\_id,}
\text{  int *time\_stamp);}\]

When an error occurs and process recovery is non-shrinking, application data for a particular group member can be restored simply using the following collective function.

\[
\text{int Fenix\_Data\_member\_restore(}
\text{  int group\_id,}
\text{  int member\_id,}
\text{  void *target\_buffer,}
\text{  int max\_count,}
\text{  int time\_stamp);}\]

Upon return each rank in the repaired communicator can access the extracted snapshot at address \text{target\_buffer}, if it is within \text{max\_count} units of the member’s MPI data type.

When an error occurs and process recovery is shrinking, there will be fewer ranks after the failure than before. This case is supported by a more general data recovery function.

\[
\text{int Fenix\_Data\_member\_restore\_from\_rank(}
\text{  int group\_id,}
\text{  int member\_id,}
\text{  void *target\_buffer,}
\text{  int max\_count,}
\text{  int time\_stamp,}
\text{  int source\_rank);}\]

This collective function names the source rank (pre-error) explicitly. Let the size of the affected communicator before and after the failure be \(C_a\) and \(C_b\), respectively. The calling rank is \(R\). Calling \text{restore} twice, with \text{source\_rank} equal to \(R\) and \(C_a + R\), respectively, returns all application data, provided ranks \(C_b - C_a\) through \(C_a - 1\) set \text{max\_count} to zero in the second round to avoid searching for non-existent data. Next, data needs to be redistributed among the ranks to avoid load imbalance. This is beyond the scope of Fenix.

In addition to the above, Fenix provides query, synchronization, and implicit and explicit garbage collection functions, as well as non-blocking storage functions to improve performance, and functions to manipulate subsets.

3. RELATED WORK

Reinit [1] functionality is similar to Fenix', with these exceptions: it assumes non-shrinking recovery, whereas Fenix supports shrinking and non-shrinking recovery within the same framework; it offers no facilities for data recovery; it is built directly into the MPI runtime, whereas Fenix is a library built on top of MPI—the current implementation uses ULFM, which our specification does not expose; it changes program structure, replacing the original \text{main} with calls to cleanup handling, whereas Fenix-enabled codes retain their original structure—they can skip all fault tolerance constructs via selective compilation or runtime tests; it supports direct use of \text{MPI\_COMM\_WORLD}, but to accomplish the same with Fenix requires the PMPI profiling interface. This prohibits use of other PMPI tools together with Fenix. Efforts within the MPI Forum target PMPI alternatives supporting multiple tools simultaneously.


Adaptive MPI, leveraging Charm++’s runtime, supports shrinking and non-shrinking recovery. It differs from Fenix in that it does not use a production MPI runtime; it always maintains the same number of MPI ranks (implemented as user-level threads), but redistributes those among remaining resources in a shrinking recovery; it assumes and integrates data recovery through rollback, whereas Fenix decouples process and data recovery.

Without further reference we point to LFLR (Local Failure, Local Recovery) and RTS (Run-Through Stabilization) as precursors to Fenix and ULFM, respectively.

4. REFERENCES


