Adding Fault Tolerance to NPB Benchmarks Using ULFM

Zachary W. Parchman
Tennessee Technological University
zwparchman@gmail.com

Geoffroy R. Vallée
Oak Ridge National Laboratory
valleegr@ornl.gov

Thomas Naughton
Oak Ridge National Laboratory
naughtont@ornl.gov

Christian Engelmann
Oak Ridge National Laboratory
engelmannc@ornl.gov

David Bernholdt
Oak Ridge National Laboratory
bernholdtde@ornl.gov

Stephen L. Scott
Tennessee Technological University
sscott@tntech.edu

ABSTRACT

In the world of high-performance computing, fault tolerance and application resilience are becoming some of the primary concerns because of increasing hardware failures and memory corruptions. While the research community has been investigating various options, from system-level solutions to application-level solutions, standards such as the Message Passing Interface (MPI) are also starting to include such capabilities. The current proposal for MPI fault tolerant is centered around the User-Level Failure Mitigation (ULFM) concept, which provides means for fault detection and recovery of the MPI layer. This approach does not address application-level recovery, which is currently left to application developers. In this work, we present a modification of some of the benchmarks of the NAS parallel benchmark (NPB) to include support of the ULFM capabilities as well as application-level strategies and mechanisms for application-level failure recovery. As such, we present: (i) an application-level library to "checkpoint" and restore data, (ii) extensions of NPB benchmarks for fault tolerance based on different strategies, (iii) a fault injection tool, and (iv) some preliminary results that show the impact of such fault tolerant strategies on the application execution.

CCS Concepts

•**Hardware** → **Failure recovery, maintenance and self-repair;** •**Computer systems organization** → *Parallel architectures;*

Keywords

message passing interface, fault tolerance, benchmark

1. INTRODUCTION

The perpetually increasing scale of high-performance computing (HPC) platforms leads to new challenges, from higher levels of concurrency to resilience concerns. Furthermore, the HPC community can only admit the dominating role of the message passing execution model, which is not expected to disappear anytime soon since the research community seems to agree that Message Passing Interface (MPI) [6] will still have a role to play for exascale computing. In this context, extensions to the MPI standard have been proposed and an effort is currently ongoing to extend the standard with primitives enabling User-Level Failure Mitigation (ULFM) [4], with the goal of addressing the process failure model, which only considers failures of MPI ranks. The ULFM proposal is based on the following principals: (i) failures are reported through returns code of MPI primitives, so *detection can only happen when the application is trying to communicate, not during computation phases*; (ii) new MPI primitives are available to have a global agreement about which MPI ranks failed; and (iii) new MPI primitives are available to get valid MPI objects that can be used for communications despite failures. We want to emphasize that ULFM does not focus on failure recovery; it provides primitives to detect failures and ensure that the MPI state is valid enough to support further communications. Recovery strategies are assumed to be implemented at the application level, eventually using recovery-specific libraries.

The document presents modifications of the MPI implementation of the NASA Advanced Supercomputing Division (NAS) Parallel Benchmark (NPB) [3, 2] using ULFM. Our contributions can be summarized as follow:

- an implementation of fault tolerance strategies for the NPB benchmarks Embarrassingly Parallel (EP) and Data Transfer (DT);
- a library that can be used by application developers to save data that also enables data recovery after an MPI ranks fails;

FTXS'16, May 31 2016, Kyoto, Japan

ACM ISBN 978-1-4503-4349-7/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2909428.2909434

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

- a fault injection tool that enables portable and reproducible results by injecting failures at pre-defined positions and conditions;
- a series of experiments, using the ULFM implementation from the University of Tennessee, Knoxville (UTK) that shows that the proposed modifications behave and perform as expected.

Finally, note that the goal of this study is not to evaluate the actual performance of any ULFM implementation, but rather to investigate what fault tolerance strategies can be implemented using ULFM, what are their tradeoffs in terms on memory and communication overheads, and if the observed behavior and performance comply with the overall strategy (the literature already has some studies of the performance of ULFM implementations [5]). As such, we hope that our work will also be used as a benchmark to validate the functionalities of ULFM implementations, since our modified version of NPB exercises ULFM implementation fairly more extensively than current ULFM tests (which are more micro-tests).

The remaining of the paper is organized as follow. Section 2 presents background information about ULFM. Section 3 details how various NPB benchmarks are extended to support ULFM, as well as an application-level library for saving data. Section 4 describes a fault injection tool that we developed to validate the ULFM support. In Section 5, we evaluate the modified benchmarks to ensure that the witnessed behavior and performance are matching expectations (note that the performance evaluation of the actual ULFM implementation is out-of-scope of this paper). Finally, Section 6 concludes.

2. ULFM OVERVIEW

ULFM is the draft proposal from the MPI Fault Tolerance Working Group (MPI-FTWG) to enhance the MPI specification for the handling of rank failures. The goal of ULFM is to provide failure detection and notification, as well as all the required mechanisms to guarantee correctness during communications after recovery of MPI objects (ULFM does *not* support application-level recovery which is assumed to be the responsibility of the application developer(s)). Failure detection is performed by adding a new error type, `MPI_ERR_PROC_FAILED`, which is issued when a communication to a failed rank is attempted.

Below is a list of the ULFM functions that are available to application developers to help deal with failures of MPI ranks (the 'X' in MPIX is to signify that these functions are not yet part of the MPI standard).

- `MPIX_Comm_revoke`: Revoke a communicator and force all further communications to return `MPIX_ERR_REVOKED` instead of completing successfully. It also applies to point-to-point communications between non-failed MPI ranks, and this is a local operation.
- `MPIX_Comm_shrink`: Shrink a communicator to exclude all known failed MPI ranks; collective operation.
- `MPIX_Comm_failure_ack`: Acknowledge rank failures; local operation.
- `MPIX_Comm_failure_get_acked`: Return a group representing the locally acknowledged failed MPI ranks. This is a local operation and therefore, not necessarily contain the failed MPI ranks that have been acknowledged by other MPI ranks.
- `MPIX_Comm_agree`: Perform a collective *bitwise AND* on its argument: failed MPI ranks do not contribute to the result and therefore this operation performs a global agreement among surviving MPI ranks to identify failed ranks. Note that if one or more MPI rank has failed but was not acknowledged locally with `MPIX_Comm_failure_ack`, then an exception of the class `MPIX_ERR_PROC_FAILED` is raised and the agreement fails (however, it is then possible to acknowledge these failed ranks and call the agreement function again). This is a collective operation.
- `MPIX_Comm_iagree`: This is a non-blocking version of the agreement function (`MPIX_Comm_agree`).

3. ULFM SUPPORT IN NPB

In this section, we first present a proof-of-concept in-memory data checkpointing mechanism. Secondly, we present the ULFM-related modifications to the NPB benchmarks.

3.1 Application-level Data Saver

In the context of this work, we have two main goals: (i) minimize the number of external dependencies to simplify the release of the resulting software as a self-contained benchmark, and (ii) avoid using a checkpoint/restart solution that relies on the file system since our testbeds used for this study do not support any high-performance file I/O. For these reasons, we did not use checkpoint/restart solution such as SCR [8, 7] (which does not mean that such solutions would not be of benefit when extending this work to more generic cases and/or scientific applications), but instead developed a in-memory proof-of-concept solution. This solution is implemented via an application-level library that provides a set of basic interfaces and policies to implement various data fault tolerance and recovery strategies. We opted for generic interfaces that hide the complexity of the data checkpoint and recovery policies. In other words, the application developers do not have to worry about the implementation of the data checkpoint and recovery strategy but instead, simply specify which strategy from the library should be used.

Data checkpointing and recovery is always a trade-off between checkpoint overhead, recovery overhead and the level of fault tolerance provided. For example, it is possible for each rank to save data into a neighbor's memory, which limits the checkpoint and recovery cost (checkpointing into a single neighbor's memory requires only limited additional memory and communications, and therefore limits the overall checkpoint's cost), but if both the rank and its neighbor fail, it is not possible to recover from the two failures. On the other hand, by saving the data onto all other MPI ranks' memory, the data is always available even if a single rank survives but it creates a large overhead since using a lot of additional memory and communications.

At the moment, we implemented two different strategies: *single redundancy* (also noted `SINGLE_REDUNDANCY`) using a single neighbor, and *full redundancy* (also noted `FULL_REDUNDANCY`) using all MPI ranks.

From a technical point of view, the interfaces currently assume that: (i) the data is in a `void*` C array and (ii) the data is checkpointed and restored within a static set of ranks (i.e., we cannot spawn a new rank upon failure and have it restore the data that was previously saved by another rank). Also, byte order and data type size portability are the re-

sponsibility of the calling application. Finally, the data saving strategy is selected at the time the library is initialized by passing a unique identifier (i.e., `SINGLE_REDUNDANCY` or `FULL_REDUNDANCY`).

3.2 NPB Benchmark Extensions for ULFM Support

3.2.1 *Embarrassingly Parallel – EP*

EP generates sets of random numbers that go through a statistical test and, if they pass, are modified to fall into a normal distribution as Gaussian pairs. Verification is a simple value check based on the summation of the Gaussian pairs across all of the ranks.

Original Algorithm.

The original EP algorithm starts on the `GenerateInput` step (Algorithm 1 line 4), creating an array of random numbers using a seed derived from its rank. These random numbers have the `CalculateValues` function applied to them (Algorithm 1 line 5) which adds the value to the local solution when certain conditions are met. When all of the ranks have finished creating their local solution, they gather them in `CollectResults`; rank 0 then outputs the final results. The only phase where communications occur is during the `CollectResults` phase. Figure 1 illustrates the execution flow of the EP benchmark.

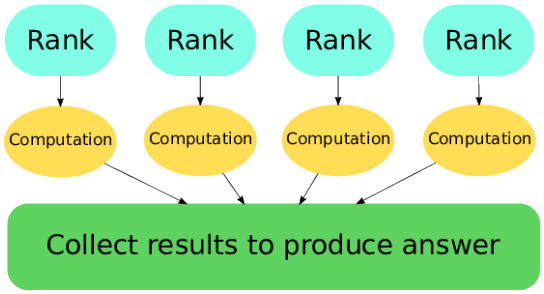

Figure 1: Graphical overview of the EP algorithm.

Modifications for Fault Tolerance.

To make the EP benchmark fault tolerant, it now starts by creating a work array in `GenerateWorkArray` (Algorithm 1 line 2). This work array starts as a simple array from 0 to $n-1$, where n is the number of MPI ranks, so each rank has its own slot in the array and each rank maintains its own array. Then, in `GenerateInputForMyRanksWorkArraySlot` (Algorithm 1 line 4), the ranks each select the i^{th} slot of the array, where i is the current rank. The rank then executes the unmodified `CalculateValues` function using the select data (Algorithm 1 line 5). It is important to note that the only occurring communications happen in `MPI_Init` and a `MPI_Barrier` (to synchronize timer initialization), which are executed before `CollectResults` (Algorithm 1 line 6). The collection of local results (`CollectResults`) has been modified to tolerate failures: both the original and the fault tolerant version perform three `MPI_Allreduce` reductions on some internal data structures but the error codes for these reductions are checked in the context of the fault tolerant version and if a failure occurs, a valid communicator is restored by using `MPIX_Comm_shrink`. The computation is then restarted in order to redistribute the work amongst surviving ranks, until the remaining ranks can successfully complete the operation. In `UpdateWorkArray`, the living ranks remove all other living ranks workload from their own copy of the work array. The array is then compacted so that only work that was not collected, due to failed ranks, is left. The rank then repeats from the work selection step (Algorithm 1 line 4) with any ranks that have no work, skipping ahead to the collection step. It is important to note that the rank in a new communicator resulting from `MPIX_Comm_shrink` may be different from the original one, since communicators cannot be sparse and one or more failed ranks are now evicted from the communicator.

Fault Tolerance Scope.

In the EP benchmark, fault tolerance begins after the initial setup and header output, not shown in Algorithm 1, until just before the `DisplayResults`. Once the fault tolerance section of EP terminates, the critical computation is completed, but some timers have not yet been collected and the final answer has not been validated yet.

3.2.2 *Data Transfer – DT*

DT is an irregular communication benchmark: a directed graph is generated with a payload that needs to be operated on. Communication flows from *source nodes*, i.e., nodes with no incoming edges, to the *sink nodes*, i.e., nodes with no outgoing edges, and *'other' nodes* that have both incoming and outgoing edges.

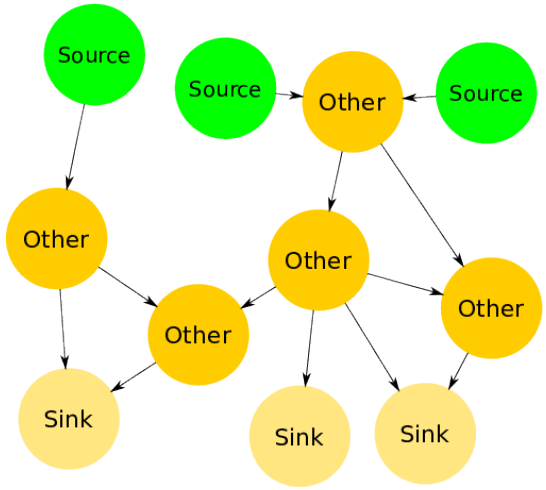

Figure 2: Graphical overview of the DT algorithm. Communications related to fault tolerance are not included

Original Algorithm.

The graph is topologically sorted so that sources come before all other nodes and sinks are last. The topology of the graph is determined by the problem size, set at compile time, as well as by the type of graph that is requested during the invocation of the benchmark. Each rank is in charge of a single graph node meaning that there must be at least one rank for each graph node. Potential extra ranks are not used. In `GenerateGraph` (Algorithm 2 line 2), every rank creates the same graph.When the processing in

Algorithm 1 Pseudocode for EP with fault tolerance (right) and without (left)

```
1: function MAIN
2:
3:
4:     input ← GenerateInput()
5:     a ← CalculateValues(input)
6:     b ← CollectResults(a)
7:
8:
9:
10:
11:    DisplayResults(b)
12: end function
```

```
1: function MAIN
2:     workArray ← GenerateWorkArray()
3:     label WorkTop
4:     input ← GenerateInputForMyRanksWorkArraySlot()
5:     a ← CalculateValues(input)
6:     b, error ← CollectResults(a)
7:     workArray ← UpdateWorkArray(error)
8:     if error ≠ Success then
9:         goto WorkTop
10:    end if
11:    DisplayResults(b)
12: end function
```

`PerformOperations` (Algorithm 2 line 12) begins, each rank performs a computation based on the type of the graph's node (i.e., source, sink, or other) that is assigned to them. The 3 types of computation are the following. (i) Source nodes create a payload to be operated on, by invoking `CreateSourceData` (Algorithm 2 line 5), where the payload is an array of pseudo random double precision floating point values with a variable size. This payload is then sent to the nodes pointed to by the outgoing edges of the graph for this node. (ii) 'Other' nodes receive the data from their incoming edges and combine the received arrays. The result of this combination is now the node's payload, which is then sent to outgoing edges. (iii) Sink nodes receive data from the incoming edges and reduces it into checksums that are sent to $rank_0$ for the validation. After all of the nodes have been processed, $rank_0$ combines all the checksums, in `FinalDataGather` (Algorithm 2 line 13). This is then verified against expected values for the particular graph type and problem size.

Modifications for Fault Tolerance.

Most of the work to implement a fault tolerant DT is into serializing and de-serializing the graph when sending data to other ranks for redundancy. Since every rank starts with the same local representation of the graph, locally serializing and storing the initial graph provides a common checkpoint that all ranks can roll back to without any extra communication. If there is no failure when saving the results of `CreateSourceData` (Algorithm 2 line 5) to remote ranks' memory, that is used as a checkpoint. In the context of a failure, the communicator is shrunk, the original graph is de-serialized and the rank attempts to re-execute the computation. This means that after a failure, a rank may be responsible for a new graph node, but since all of the graphs are the same as before the save has been completed, this is not an issue. If a failure occurs after the save at Algorithm 2 line 3, then the application will rollback to this point (Algorithm 2 line 3). This method requires at least $n + f$ ranks to complete successfully where n is the number of nodes in the graph and f is the number of ranks that will fail. The previous save from Algorithm 2 line 6 is used as a checkpoint and any failure will cause the entire application to roll back to Algorithm 2 line 11, otherwise the application continues as it does in the original algorithm.

Fault Tolerance Scope.

Fault tolerance in DT begins just after `GenerateGraph` (Algorithm 2 line 2) and stops just after performing `FinalDataGather` (Algorithm 2 line 13). Because of the nature of this benchmark, it is difficult to trace the communications to find points where a global checkpoint can be done. However, instead of trying to save more work between communication phases, the strategy provides a fast restart in the event of failures.

4. FAULT INJECTION TOOL

Faults are injected using a library that we developed, named `FaultInjection`. The library is used by placing `FaultInjection_fault` calls in the application code. Note that since ULFM can detect and handle failures only in the context of a MPI function, we limit the injection points around the MPI calls. It would be possible to include more injection points but it would not impact when the failure is detected and how the failure is handled. These pre-defined fault injection points take 4 arguments: where in the code this fault resides, and three arbitrary application defined integers. This interface is a compromise between flexibility and simplicity. Based on our experience, three generic arguments (integers) are enough for expressing all requirements related to fault injection. When all of the values meet a set of pre-defined conditions, described below, the fault is activated, i.e., the error is injected. The actual fault itself is one of the following: (i) a call to the C standard library's `exit` function with a value of 0; (ii) a call to the `_exit` function with an value of 0; or (iii) a call to Linux's `kill` with a signal of 9. The method is set at compile time and they all lead to the same result: the failure of an MPI rank.

Faults are configured through a file that is specified when launching an MPI job. This configuration file is a Lua script (Lua is chosen for its permissive license and ease of use [1]), and enables the description of complex experiments. By using such a configuration file, it is possible to tailor an application with a fault injection configuration, which greatly helps with experimental reproducibility. This configuration script calls a C function in the `FaultInjection` framework to register failures based on their location in the source code, the rank that should fail, and the three application defined integers. When a failure is injected, it is logged to the standard error (`stderr`) stream. An example is shown below.

```
                    Fault requested at
FaultID:    AFTER_KEY_SORT    Rank:2 tag1:3 tag2:0 tag3:0
```

The FaultID is the name of the registered define, in this case "After_Key_Sort", that represents the location in the application's code. The FaultID is intended to be unique, but this is not enforced by the library. Rank is the MPI rank with respect to `MPI_COMM_WORLD`. Tag1, Tag2, and Tag3 are the application defined integers.

Internally the information about when to inject a fault is stored in an array of linked list pointers, each fault location

Algorithm 2 Pseudocode for DT with fault tolerance (right) and without (left)

```
1: function MAIN
2:     input ← GenerateGraph()
3:
4:
5:     a ← CreateSourceData(input)
6:
7:
8:
9:
10:
11:
12:    b ← PerformOperations(a)
13:    c ← FinalDataGather(b)
14:
15:
16:
17:
18:    DisplayResults(c)
19: end function
```

```
1: function MAIN
2:     input ← GenerateGraph()
3:     save1 ← SaveGraph(input)
4:      label Save1_label
5:     a ← CreateSourceData(input)
6:     save2 ← SaveGraph(a)
7:     if ErrorHasOccured then
8:         input ← LoadSave(save1)
9:          goto Save1_label
10:    end if
11:     label Save2_label
12:    b ← PerformOperations(a)
13:    c ← FinalDataGather(b)
14:    if ErrorHasOccured then
15:        a ← LoadSave(save2)
16:         goto Save2_label
17:    end if
18:    DisplayResults(e)
19: end function
```

having its own linked list. If a fault is requested for a location that does not have a fault registered, the lookup is just a check to see if a pointer is NULL, minimizing runtime overheads. If there is at least one fault registered for that location, the linked list is traversed checking each set of parameters. If one of the failures in the list matches the fault request, then the process is killed; if no matches are found, the entire list is traversed and the process continues to run. In general, the cost of traversing this linked list is expected to be quite low and will likely be outweighed by the surrounding code, especially if the traversal is near MPI calls. The `FaultInjection` library does no communication or coordination after the initialization phase where the configuration file is read by $rank_0$ and sent via MPI to all ranks.

4.1 Fault Injection in EP

In EP, faults can be injected into two locations (see Algorithm 3): (i) before `CollectResults` (called `FaultReduce`) and (ii) after `Shrink` (called `FaultRecover`). Injecting a fault before `CollectResults` simulates a failure at any point during the main computation. If a failure occurs here then another failure point can be triggered in the recovery path. At the FaultRecover injection point, after the `FaultReduce` failure has occurred and is detected at `Reduction`, another fault can occur after the shrink. This will cause the reduction to fail again without exiting the `CollectResults` function.

4.2 Fault Injection in DT

Algorithm 4 illustrates where the fault injection points are added to DT. Failure injection points are inserted both before and after `PerformOperations` named `PRE_WORK` and `POST_WORK`, respectively. The first causes little work to be lost when a failure occurs there. If a fault occurs at `POST_WORK`, then almost the entire amount of work done by the application is lost.

Another failure point is inserted before the `SaveGraph` function. This provides an opportunity to inject a failure before the potentially expensive save operation. The cost associated to the save depends on the requested strategy as described in Section 3.1. The previously mentioned `PRE_WORK` can be used to fail after the save operation, if that is desired.

There is another opportunity to insert a failure after `PerformOperations`, at `FAULT_P2_POST_RECV_RESULTS`, which is basically after the last set of receives where $rank_0$ has received the results from the sink nodes and is ready to display the answer. This is the most expensive failure point and will restore the problem to the state before the main communication intensive kernel begins.

Algorithm 3 EP with fault injection points

```
1: function CollectResults(a)
2:     error ← Success
3:      label top
4:     b, partialError ← Reductions(a)
5:     error ← error + partialError
6:
7:     if partialError ≠ Success then
8:         Shrink( )
9:         FaultInjection_fault(FaultRecover)
10:         goto top
11:    end if
12:    return b,error
13: end function
14:
15: function MAIN
16:    workArray ← GenerateWorkArray()
17:     label WorkTop
18:    input ← GenerateInputForMyRanksWorkArraySlot()
19:    a ← CalculateValues(input)
20:    FaultInjection_fault(FaultReduce)
21:    b, error ← CollectResults(a)
22:    workArray ← UpdateWorkArray(error)
23:    if error ≠ Success then
24:        goto WorkTop
25:    end if
26:    DisplayResults(b)
27: end function
```

Other failure points are available in the source code if finer grained control of failure injection points is desired, but are not detailed here because of space constraints.

4.3 Experiment Description

Experiments are driven by configuration scripts used for fault injection, which precisely describe which fault injection points are activated and in the context of which MPI ranks. These configuration files are specified via an environment variable. Because of space limitation, we do not further describe the syntax of these configuration files.

5. EVALUATION

We present in this section the preliminary evaluation of the two fault tolerance strategies that have been implemented by extending the NPB EP and DT benchmarks. As said earlier, our goal is not to evaluate a specific ULFM imple-

Algorithm 4 DT with fault injection points

```
1: function MAIN
2:     input ← GenerateGraph()
3:     save1 ← SaveGraph(input)
4:      label Save1_label
5:     a ← CreateSourceData(input)
6:     save2 ← SaveGraph(a)
7:     if ErrorHasOccured then
8:         input ← LoadSave(save1)
9:          goto Save1_label
10:    end if
11:     label Save2_label
12:    FaultInjection(PRE_WORK)
13:    b ← PerformOperations(a)
14:    FaultInjection(POST_WORK)
15:    c ← FinalDataGather(b)
16:    FaultInjection(FAULT_P2_POST_RECV_RESULTS)
17:    if ErrorHasOccured then
18:        a ← LoadSave(save2)
19:         goto Save2_label
20:    end if
21:    DisplayResults(e)
22: end function
```

mentation but rather evaluate the cost of each fault tolerance strategies, identify problems and challenges with these strategies, as well as ensuring that the witnessed behaviors match the expected ones.

5.1 Experimentation Protocol

The experimental platform is a Linux cluster that is composed of 39 compute nodes, each having two 12-core 1.7 GHz AMD Opteron 6164 HE processors and 64 GB RAM. The platform also has a bonded dual non-blocking 1 Gbps Ethernet interconnect that delivers a 1.7 Gbps for point-to-point communications using TCP. The system is using Ubuntu 12.04 LTS, GCC 4.6 and NPB-3.3. We used ULFM v1.1 from UTK.

Since our goal is to see the impact of the two fault tolerant strategies, we decided to run a single MPI rank per node, i.e., run tests up to 32 ranks (to keep a power of two number of ranks). This choice has two benefits: (i) clearly highlight the communication overhead since all inter-rank communications are going through the network, (ii) the cost associated with the checkpointing and restoring of data between two ranks is constant, since 2 ranks cannot be running on the same node.

We ran each experiment 10 times; all the results are the mean of the 10 runs and have a low standard deviation, except for the class S, which is expected since S has a very short run time and does not aim at being used as representative of any real use-case. We do not present the standard deviation because of space limitations and because it reduces the readability of the graphs. For all experiments, the *native* case represents the unmodified benchmark executed with the MPI implementation without any fault tolerance capability. The *0 failure* case represents the execution of the modified version of the benchmark for fault tolerance, which basically means that data is checkpointed. This case therefore shows the overhead created by the data checkpointing. The *1 failure* case represents the execution of the benchmark and the injection of a single fault during the execution. The *2 failures* case refers to the execution of the modified version of the benchmark and the injection of 2 faults, at the same injection point, for 2 different ranks. In other terms, the 2 failures are expected to be injected at about the same time in the context of 2 different MPI ranks and therefore are expected to trigger a single recovery that will cover the 2 failures.

5.2 Results for the EP Benchmark

The EP benchmark has 5 classes (S, W, A, B, C). For this evaluation, we use a `SINGLE_REDUNDANCY` strategy, i.e., each rank saves data into a neighbor's memory by explicitly sending the data to the remote via a point-to-point MPI blocking send. This benchmark being composed of communications with a main communication phase after the computation is done to exchange results, the communication overhead is relatively low compared to the time spent during the computation phase. We inject failures during this communication phase. As a result, it is expected that an additional computing iteration is required to perform the work lost because of the failure. In other terms, the performance of the benchmark is not expected to significantly drop for the fault tolerant version of the benchmark (the communication required to save the data is cheap compared to the computation time) and the performance is expected to be twice as slow when injecting 1 or 2 failures (the two failures are injected at the same injection point and are expected to trigger a single communicator revoke/shrink and work redistribution). In the context of a single failure, rank 0 is aborted, while rank 0 and 1 are aborted in the context of 2 failures.

Figure 3 shows the results, where each bar represents the number of operations reported by the benchmark (the higher, the better). The observed performance matches the expected behavior. We can even observe the slight cost of checkpointing the data for the fault tolerant version of the benchmark (difference between the *native* performance and the *0 failure* performance).

5.3 Results for the DT Benchmark

The DT benchmark has very specific requirements, especially in terms of the number of MPI ranks for a specific test configuration: DT supports multiple classes of tests (S, W, A, B, C) and for each class, 3 graphs are available (BH, WH, SH), each class/graph having a different number of graph nodes and therefore a different requirement regarding the number of MPI ranks (since a single graph node is assigned to a single MPI rank, extra ranks not being used – but can be used as spares for fault tolerance purposes). Table 1 summarizes the requirement in terms of MPI ranks for each class/graph combination. Note that since we decided to have a single MPI rank per node, it implies that all tests requiring more than 32 ranks are not executed.

For this evaluation, we use a `SINGLE_REDUNDANCY` strategy. When injecting a fault, we always inject it after the first data checkpoint (line 3 in Algorithm 4), by aborting rank 2 in the context of a single failure, and rank 2 and 3 in the context of 2 failures. Note that the experiments could not successfully terminate with 1 failure in the context of the W class and SH graph, and we suspect a bug in the ULFM implementation since the problems occur while shrinking the communicator.

Figure 4 shows the results (the higher the better). The results shows that the cost of checkpointing the data significantly decreases the overall performance of the DT benchmark for all combination of class and graph. While we do not have a detailed break down of the overhead, we believe this is due to the extra communications and synchronizations that are necessary for all ranks to save data into a neighbor's memory via explicit MPI communications. Furthermore, as

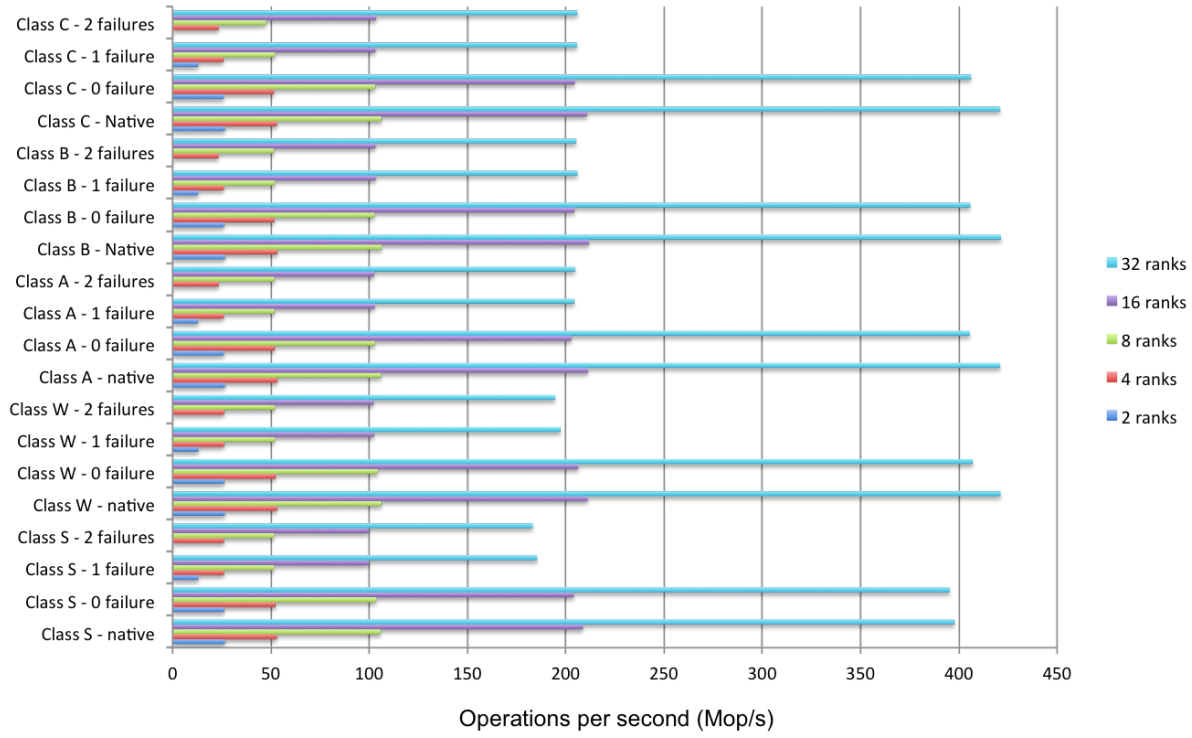

Figure 3: Performance of the EP Benchmark

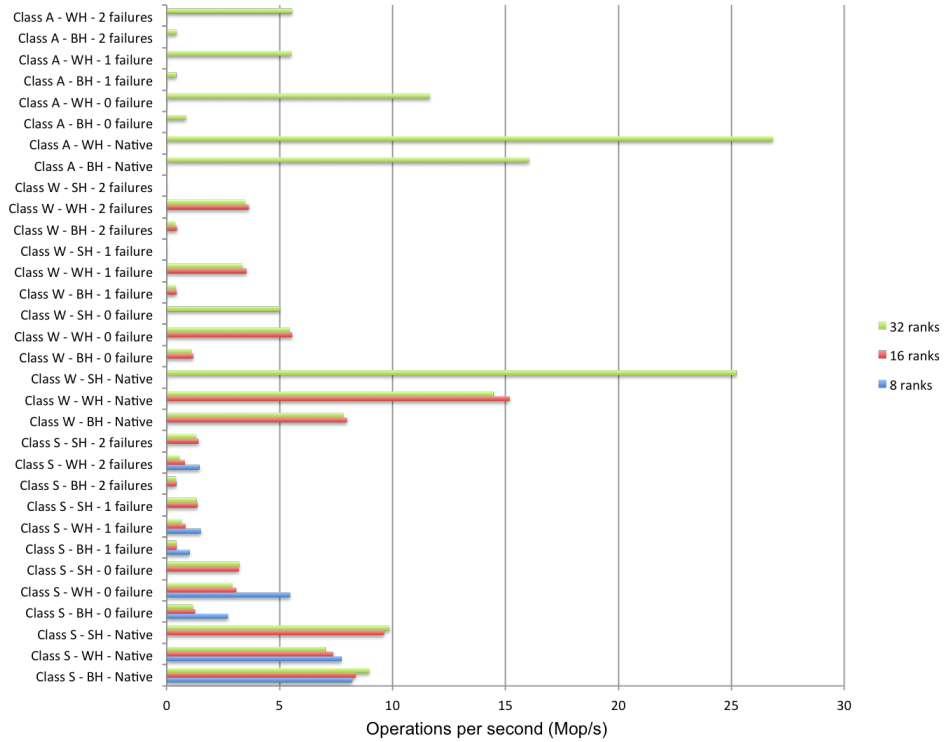

Figure 4: Performance for the Different Combination of Class and Graph for the DT Benchmark

presented in Algorithm 4, the implemented strategy requires multiple checkpointing of the data. As a result, the modified version of the benchmark is at least twice as slow than the unmodified version of the benchmark, even without facing

Class	Graph	Minimum number of ranks
S	BH	5
S	WH	5
S	SH	12
W	BH	11
W	WH	11
W	SH	32
A	BH	21
A	WH	21
A	SH	80
B	BH	43
B	WH	43
B	SH	448
C	BH	85
C	WH	85
C	SH	448

Table 1: Requirements in terms of MPI ranks for each DT class/graph combination.

any failure, down to an order of magnitude slower for the Class A with the graph BH. When injecting failures, performance is dropping even more because of the restoration of the data from a checkpoint and the work redistribution, which implies both extra computation and extra communications. However, the performance difference between the case with a single failure and 2 failure is not significant, which is expected since both failures are injected at the same failure point, i.e., the recovery phase happens only once (the time between the failures is not important enough to require 2 communicator revoke/shrink operations, data restorations and redistributions of the work).

6. CONCLUSION

In this study, we presented: (i) the implementation of fault tolerance strategies for the NPB Embarrassingly Parallel (EP) and Data Transfer (DT) benchmarks; (ii) a library that can be used by application developers to save data that also enables data recovery; (iii) a fault injection tool that allows us to have portable and reproducible results by injecting failures at pre-defined positions and conditions; and (iv) a series of preliminary experiments.

More specifically, we showed that ULFM is a suitable solution for the implementation of various strategies. It also shows that the difficulty of implementing such strategies depends on the nature of the application (i.e., embarrassingly parallel applications are easy to make fault tolerant, while applications that are more irregular and more heavily reliant on communications are more difficult to handle). We also showed that having a fault tolerant version of a complex application that does not suffer of an important performance degradation because of data checkpointing is a non-trivial task. Fortunately, most complex scientific simulations nowadays rely on application-level checkpoint/restart, which should ease the transition to fault tolerant applications based on ULFM (the integration of such application-level checkpoint/restart solutions with ULFM is out-of-the-scope of this study).

As future work, we plan on investigating the limitations of the various data checkpointing strategies based on the overall number of failures, their distribution, as well as the scale of the application's run. For instance, it would be interesting to see whether the `SINGLE_REDUNDANCY` approach is suitable based on the number of failures and the failure distribution of current HPC systems. It would also be interesting to extend addition NPB benchmarks since EP could be considered as the best use-case and DT as the worst use-case (because of its irregular communications and the difficulties for redistributing the work upon failures), and study the efficiency of various strategies. In addition, we are planning on more precisely analyzing the various overheads of the fault tolerant version of the DT benchmark and investigate other strategies that would generate a lower overhead. It would also be interesting to compare and integrate our checkpointing solution with existing solutions such as SCR. Finally, we are expecting that this work will lead to the release of the NPB-ULFM suite that would be available to the community to test and validate current and future implementation of ULFM.

7. REFERENCES

[1] Lua programming language website. http://http://www.lua.org. Accessed: 2015-07-23.

[2] Nas parallel benchmarks. https://www.nas.nasa.gov/publications/npb.html. Accessed: 2015-07-22.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[4] W. Bland, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra. A proposal for User-Level Failure Mitigation in the MPI-3 Standard. Technical report, Tech. rep., Department of Electrical Engineering and Computer Science, University of Tennessee, 2012.

[5] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. Dongarra. An evaluation of User-Level Failure Mitigation support in MPI. *Computing*, 95(12):1171–1184, 2013.

[6] Message Passing Interface Fourm. *MPI : A Message-Passing Interface Standard Version 3.0*, September 2012.

[7] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[8] Scalable Checkpoint/Restart Library. http://sourceforge.net/projects/scalablecr/.